SYDE 556/750
Simulating Neurobiological Systems
Lecture 8: Learning

Andreas Stöckel

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

February 25 & 27, 2020

Accompanying Readings: Chapter 9 of Neural Engineering

Contents

1 Introduction

Note: As mentioned several times in previous lectures, we so far assumed that the connection weights between individual neuron populations are constant. This is of course not the case in biology, where animals change their behaviour in response to experiences sometimes years in the past.

When talking about *learning*, we should first define this term a little more rigorously. A very broad definition of learning from a neuroscientist's perspective (psychologists will have different definitions) could be "any process within a neural system that allows past stimuli to influence future behaviours". Unfortunately, this definition would also include dynamical systems that we have already talked about, such as integrators or the Delay Network.

In this course we will stick to a much more simple definition of learning:

> *Learning* is a directed change in synaptic weights **W** while the network is active.

We refer to anything that fits into the category of "past-affecting-the-future" phenomena as *adaptation*. Adaptation thus encompasses learning in the sense defined above, as well as dynamical systems, and short-term neural firing-rate adaptation.

Of course, now that we have defined what "learning" is, we should briefly discuss why this is a useful concept to talk about, especially within the Neural Engineering Framework. After all, we can already compute the "optimal" decoders, so why would we want to change those after the fact by changing the synaptic weights?

1. **We might not know the function we want to compute at the beginning of a task.**
   Consider a simulated critter that explores its environment in search for food. Some food (the "green food") is nutritious and should be eaten, whereas other food (the "red food") is slightly poisonous and should be avoided.

   A priori, as modellers of this system, we might not know what awaits our critter in its environment. Hence, we have no chance to pre-compute the "correct" transformation that maps colour onto "edibility" of the foot. In fact, we might not even know which sensory modalities are important for this kind of decision (smell, colour, shape, taste, . . .).

   If instead we were able to build a system that was able to *learn* a mapping from sensory stimuli onto "edibility", that might make our lives as modellers much easier.

2. **The desired function might change over time.**
   Consider a dynamical system that controls joint muscle tensions for a given target position. While we may be able to build a system that performs this control efficiently when the system is first deployed, there may be several factors that change the required controller over time.

   For example, injury and ageing might cause wear in joints, growth (in a biological system) will change the lengths of the limbs, or the system may be required to handle loads that are much lighter or heavier than what was originally considered when designing the controller.

3. **The "optimal weights" we are solving for are not optimal.**
   Remember that the synaptic weights $\mathbf{W}$ are defined as $\mathbf{W} = \mathbf{ED}$ in a Neural Engineering Framework network. While the decoders $\mathbf{D}$ we are computing are optimal, this does not mean that the entire weight matrix $\mathbf{W} = \mathbf{ED}$ is optimal! If we optimize the full weight matrix directly, this allows us to fit individual functions in a better way.

   Furthermore, at least when computing $\mathbf{D}$ using the rate-approximation $G[J]$, we are not taking the dynamics of the neurons into account. Remember that we derived $G[J]$ under the assumption of a constant, or static, input current J. Hence, the decoders $\mathbf{D}$ that are optimal in the "static" case are not necessarily optimal for the dynamics encountered in the neural network.

4. **Answering scientific questions about learning in nervous systems.**
   As discussed in the lecture about transformations, our goal so far has been to build models of "expert systems". We are trying to answer the question whether a system that has already learned to accomplish a certain task can be modelled by implementing certain mathematical transformations under optimal circumstances.

   Incorporating learning into our models allows us to answer a slightly different question, namely whether nervous systems with a certain overall connectivity can learn transformations from the stimuli that are available to it during its lifetime or growth (ontogenesis).

These points should make it clear that incorporating learning into NEF networks is desirable. However, in order to do so, we will first take a short excursion to the field of Machine Learning, and the types of "learning" usually employed by computer scientists. We then apply some of these concepts, including supervised and unsupervised learning to biologically plausible neural networks.

2 An Excursion to Machine Learning

Inspired by the learning abilities of biological systems, computer scientists have been working on "learning machines" since the early 1960s, giving rise to the field of machine learning. The main goal of this field is to apply computers to problems that have a less well-defined relationship between input and output.

Example: Consider the following "image recognition" problem. A system receives an image encoded as a high-dimensional vector $\mathbf{x}$ (where each vector component corresponds to the intensity of one colour channel of a pixel in the image), and has to decide whether the image contains a dog, a cat, or a parrot (cf. fig. 1). It would be quite hard to write such a program "by hand". After all, what are the rules by which we would categorize a set of pixels into these categories? The goal of machine learning is to be able to "program by example". Instead of specifying an algorithm, we just specify examples, and the computer "figures out" (learns) on its own how to solve the problem in general.

Figure 1: A supervised learning task. Given a set of N examples mapping an image $\mathbf{x}_k$ onto a one-hot encoded category vector $\mathbf{t}_k$, we would like the computer to learn a mapping from unseen images $\mathbf{x}$ onto categories $\mathbf{t}$.

Within this field, computer scientists typically distinguish between three types of learning:

- **Supervised Learning**
  Given N training samples $(\mathbf{x}_k, \mathbf{t}_k)$, and a model function $f(\mathbf{x}; \mathbf{w})$, find weights/parameters $\mathbf{w}$ such that $f(\mathbf{x}_k) \approx \mathbf{t}_k$. We hope that after training, $f(\mathbf{x}; \mathbf{w})$ will also work for unseen $\mathbf{x}$. In other words, we assume that there is an unknown ground truth function $f_{\text{GT}}(\mathbf{x}) = \mathbf{t}$ that we would like to approximate by tuning the parameters of our model function $f(\mathbf{x}; \mathbf{w})$.
  Examples: Polynomial fitting, Perceptrons, generalised linear models (GLMs), Support Vector Machines (SVMs), Gaussian Processes, Multi-Layer Perceptrons (MLPs).

- **Unsupervised Learning**
  We are just given a set of N samples $\mathbf{x}_k$ and would like to discover some inherent order within this dataset. Sometimes, we would like to map each $\mathbf{x}_k$ onto a lower-dimensional "latent" space λ, where conceptually similar $\mathbf{x}$ are being assigned to similar λ.
  Examples: Clustering, autoencoders, dimensionality-reduction methods (PCA, tSNE, . . .).

- **Semi-supervised (or Reinforcement) Learning**
  A system receives state information $\mathbf{x}(t)$ and a low-dimensional reward signal $r(t)$. Goal is to learn a policy $\pi(\mathbf{x})$ that produces actions $\mathbf{t}(t)$ maximizing the time-cumulative reward.

Furthermore, there are additional constraints on the learning process place that are orthogonal to these concepts, such as "offline" (training happens while the system is not being used), "online" (the system learns on a stream of data), or "life-long" (the system is constantly learning whenever it is active) learning.

In the following sections, we have a closer look at these concepts. Maybe we can borrow some insights from Machine Learning and apply them to our spiking neural network models!

3 Supervised Learning

As summarized above, the goal of supervised learning is to find parameters $\mathbf{w}$ for a model function $f(\mathbf{x}; \mathbf{w})$, such that the function closely maps a set of N training data points $\mathbf{x}_k$ onto a set of desired target points $\mathbf{t}_k$. The quality of the match is defined according to some loss function $E(\mathbf{w})$. For example, if we assume a quadratic loss function, a supervised machine learning problem could be phrased as

$$\mathbf{w} = \operatorname{arg\,min}_{\mathbf{w}} E(\mathbf{w}) = \operatorname{arg\,min}_{\mathbf{w}} \frac{1}{N} \sum_{k=1}^{N} \left\| f(\mathbf{x}_k; \mathbf{w}) - \mathbf{t}_k \right\|^2. \tag{1}$$

While equation eq. (1) looks quite innocuous, this problem is generally difficult because of the function f. There is no constraint whatsoever on what f should look like, except for the very general notion of f being a function mapping from a d-dimensional input space $\mathbb{X} \subseteq \mathbb{R}^d$ onto some d'-dimensional target space $\mathbb{R}^{d'}$. Commonly used examples of model functions f include

- Polynomial of degree n: $f(\mathbf{x}; \mathbf{w}) = \sum_{i=0}^{n} w_i x^i$.
- Linear models: $f(\mathbf{x}; \mathbf{w}) = \sum_{i=0}^{n} w_i \varphi_i(x)$.
- Perceptron: $f(\mathbf{x}; \mathbf{w}) = \varphi\big(\langle \mathbf{w}, \mathbf{x} \rangle\big)$.
- Multi-layer perceptrons/"deep" neural networks.

The overall goal of this endeavour is *generalisation*. While our training tuples $(\mathbf{x}_k, \mathbf{t}_k)$ may only cover a small portion of the space $\mathbb{X}$ we are interested in, the general hope is that once we find the right parameters $\mathbf{w}$, our function $f(\mathbf{x}; \mathbf{w})$ will also work for tuples $(\mathbf{x}_k, \mathbf{t}_k)$ we have seen before. Underpinning this hope is the assumption that there exists a systematic mapping between $f_{\text{GT}}(\mathbf{x}; \mathbf{w})$ that is unknown to the engineer building the system, and that f will approximate this f_{GT} after training.

Unfortunately, in general, there are not guarantees that this will actually work. To the contrary, we can guarantee that an algorithm solving for $\mathbf{w}$ that generalises well within one particular application domain will not be able to generalise will in another application domain. This is known as the so-called "No Free Lunch Theorem".

Note: *Solving for decoders is a supervised learning problem.* The problem we solve when computing identity or function decoders fits exactly into this framework. In this case, our function $f(\mathbf{x}; \mathbf{w})$ is just a linear model given as

$$f(\mathbf{x}; \mathbf{w}) = \langle \mathbf{w}, \boldsymbol{\alpha}(\mathbf{x}) \rangle,$$

where $\mathbf{w}$ are our decoders, and $\boldsymbol{\alpha}(\mathbf{x})$ is the function mapping from an input $\mathbf{x}$ onto the neural activities. As we saw, we can just solve for the optimal $\mathbf{w}$ using the L_2-regularised Moore-Penrose pseudo inverse.

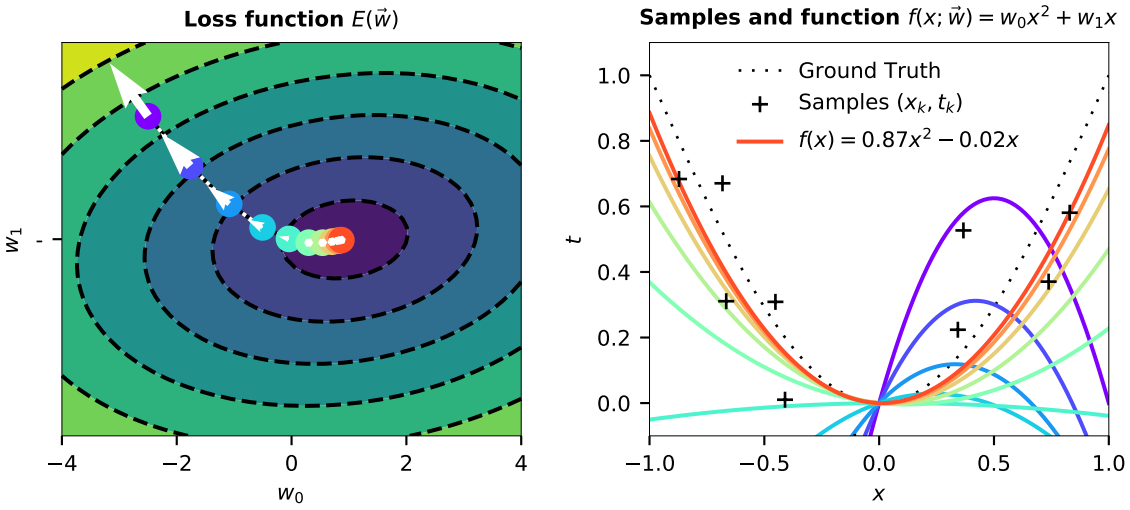


Figure 2: Gradient descent in action. We are trying to find parameters for the polynomial model function $f(x;\mathbf{w}) = w_0x^2 + w_1x$ given a set of samples (x_k, t_k). *Left:* Loss function over the parameters $\mathbf{w}$, as well as the trajectory of the paramters $\mathbf{w}$ as part of the gradient descent algorithm. Arrows indicate the gradient $\Delta\mathbf{w}$. Brighter colours correspond to larger errors. *Right:* Training samples (x_k, t_k), as well as the parametrised model functions corresponding to the parameters highlighted in the left picture. ⌨ *Code*

3.1 Using Gradient Descent to Solve for w

In general, it is impossible to solve for $\mathbf{w}$ in closed form. Instead, computer scientists often fall back onto a heuristic known as "gradient descent".

In a nutshell, we analyse how infinitesimally small changes to one of the weights w_i would affect our loss function $E(\mathbf{w})$. We use this information in order to apply a small change $-\eta\Delta\mathbf{w}$ to our weights, such that the error is being reduced. In particular, this change is given as

$$\Delta w_i = \big(\nabla E(\mathbf{w})\big)_i = \frac{\partial E(\mathbf{w})}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{N}\sum_{k=1}^{N}\big\|f(\mathbf{x}_k;\mathbf{w}) - \mathbf{t}_k\big\|^2, \quad \text{weight update step } \mathbf{w} \leftarrow \mathbf{w} - \eta\Delta\mathbf{w},$$

where η is the so called *learning rate*. The weight update steps are then repeated until $\mathbf{w}$ converges (i.e., does not change considerably). See fig. 2 for a simple example, yet keep in mind that depending on $f(\mathbf{x};\mathbf{w})$ – the loss function will not have a clear global minimum and the number of parameters can go into the thousands (and not just two as in the example).

We can either compute the partial differential $\frac{\partial}{\partial w_i}$ numerically (i.e., by evaluating E for slightly different $\mathbf{w}$ and computing the difference quotient). For some f, the gradient ∇E can be evaluated efficiently using the chain-rule, leading to an algorithm called *error back-propagation*. There are libraries such as Autograd or Tensorflow that conveniently compute these gradients.

In practice, the loss function E is often not evaluated for all N training samples at once, but for a random subset of $N' \ll N$ samples, where N' is the so called "batch size". Not only does this increase the efficiency of the gradient descent algorithm, it also prevents the algorithm from "getting stuck" in local minima. This variant of gradient descent is called "stochastic gradient descent" (stochastic because of the random selection of samples).

3.2 The Delta Learning Rule

As noted above, we are solving a supervised "learning" problem (in the machine learning sense) when computing decoders $\mathbf{D}$. However, so far we have been using what is known as an *offline* method – we compute $\mathbf{D}$ in one step for all N using the L_2-regularised Moore-Penrose pseudo inverse. That is, we have a training phase during which we compute $\mathbf{D}$, followed by the "inference" phase during which we use $\mathbf{D}$ within our neural network.

In order to *learn* synaptic weights $\mathbf{W}$ in the sense defined in the introduction, we need a so called *online* learning rule. That is, we would like to find a learning rule that allows us to update the weights for a stream of input data while the network is active.

Luckily, we can easily derive such an algorithm in two stages. First, we will implement a decoder learning rule by performing gradient descent for an input-target pair $(\mathbf{x}(t), \mathbf{y}^{\mathrm{d}}(t))$. This is somewhat comparable to stochastic gradient descent with a batch size of $N' = 1$, although we do not randomly sample from a pool of N available samples, but always use the information that is currently (i.e., at time t) available in our network.

Second, we will extend this *decoder* learning rule to a biologically plausible *synaptic weight* learning rule that updates individual entries of the weight matrix w_{ij}.

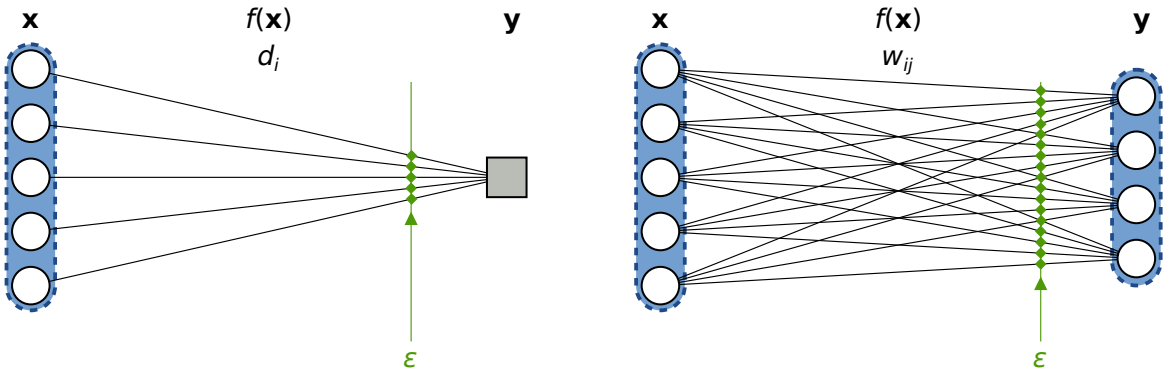
For simplicity, let's assume that $d = 1$, i.e., we are representing a scalar value. Now, our setup is as follows. We have a neuron population representing the input value x and would like to decode an unknown function $f(x)$. In other words, we would like to continuously update our decoders $\mathbf{d}$, such that $f(x(t)) \approx y^{\mathrm{d}}(t)$. This means that we have the following loss function

$$
\begin{aligned}
E(\mathbf{d}) &= \frac{1}{2}\left(\left(\sum_{i=1}^{n} d_i a_i\big(x(t)\big)\right) - y^{\mathrm{d}}(t)\right)^2 \\
\Rightarrow \Delta d_i = \frac{\partial E(\mathbf{d})}{\partial d_i} &= \underbrace{\left(\left(\sum_{i=1}^{n} d_i a_i\big(x(t)\big)\right) - y^{\mathrm{d}}(t)\right)}_{\varepsilon(t)} a_i\big(x(t)\big) \\
&= \varepsilon(t) a_i\big(x(t)\big),
\end{aligned}
$$

where $\varepsilon(t)$ is the error we are making at the current point in time. This is a version of the so called "Delta Learning Rule" for a linear model. In the next subsection we will extend this rule to individual synaptic weights w_{ij}.

Note: Since we are not stochastically sampling from a large pool of N samples, this method has a problem often referred to a *catastrophic forgetting*. By not looking at older samples, we are temporarily over-fitting our system to the current region of the input-target space.

Aside: Stochastically sampling $(\mathbf{x}(t'), \mathbf{y}^{\mathrm{d}}(t'))$ pairs over a longer history $t' \in [t - \theta, t]$ using the Delay Network with the goal to increase the convergence rate and to somewhat mitigate the problem of catastrophic forgetting would be an interesting course project!



(a) Learning decoders **(b)** Delta learning rule

Figure 3: Learning decoders d_i **(a)** and synaptic weights **(b)** in an NEF network. We would like to learn an unknown function $\mathbf{y} = f(\mathbf{x})$ encoded in either the decoding weights d_i or in the weights between two neural populations. Weights are updated according to an error signal $\varepsilon(t) = \mathbf{y}^{\mathrm{d}}(t) - y(t)$.

3.3 The Prescribed Error Sensitivity (PES) Learning Rule

We now have a simple learning rule that allows us to learn decoders online – essentially, we just multiply the neural activity $a_i(x(t))$ with the error $\varepsilon(t)$ and then update individual decoder values according to the gradient descent update equation

$$d_i \leftarrow -\eta \Delta d_i = -\eta \varepsilon(t) a_i(t)\,.$$

Essentially, if a neuron is active while there is some positive error $\varepsilon(t)$, we decrease its influence on the decoding. If there is a negative error $\varepsilon(t)$, we increase its influence.

The problem with this equation is that it is not biologically plausible. There are no decoders in the brain, so we need to find a way to update synaptic weights w_{ij} instead of decoders d_i.

However, this problem can be easily fixed. Remember that the synaptic weights are given as

$$w_{ij} = \big((\alpha \circ \mathbf{E})\mathbf{D}\big)_{ij} = \alpha_j \langle \mathbf{e}_j, \mathbf{d}_i \rangle\,.$$

Substituting the decoder update Δd_i into this equation (for the scalar case) gives us

$$\Delta w_{ij} = \alpha_j e_j \Delta d_i = -\eta \big(\alpha_j \varepsilon(t) e_j\big) a_i(t)\,.$$

For vectorial quantities $\mathbf{x}$ (i.e., $d > 1$) this just turns out to be

(Prescribed Error Sensitivity (PES) Learning Rule)

$$\Delta w_{ij} = \alpha_j e_j \Delta d_i = -\eta \big(\alpha_j \langle \varepsilon(t), \mathbf{e}_j \rangle\big) a_i(t)\,. \tag{2}$$

Note that the term $\big(\alpha_j \langle \varepsilon(t), \mathbf{e}_j \rangle\big)$ looks very much like the linear part of the current-translation function in the NEF encoding equation. In particular, we are treating the error $\varepsilon(t)$ as if it was an input $\mathbf{x}(t)$ – and in a sense it is exactly that – a so called "modulatory" input.

As it turns out, such modulatory inputs exist in the brain in the form of dopaminergic synapses, which enable synaptic plasticity in the post-synapse.

3.4 Example: Supervised Learning of Functions f

Figure 4 shows an example of learning a function $f(x) : \mathbb{R} \longrightarrow \mathbb{R}$, with an "artificially" created error signal $\varepsilon(t)$. As clearly visible in fig. 4b, we are able to quickly learn a communication channel, i.e., $f(x) = x$, between two neuron populations.

Learning a more complex function such as $f(x) = x^2$ works as well, as depicted in fig. 4c. However, notice that learning progresses much slower in the second case. Correspondingly, we should note that not only the learning rate η plays a role in how fast a function is being learned, but also the "complexity" of the function that is being learned.

Note: We will have a closer look at why some functions may be harder to learn than others in the next lecture, when we analyse representations.

3.5 Example: Classical Conditioning

Classical conditioning is a phenomenon observed in human- and non-human animals, colloquially known through research performed by Ivan Pavlov on dogs ("Pavlov's dog").

The general experimental setup is the following. An animal "unconditonally" (i.e., without initial training) responds to a stimulus. A well-studied example is the eye-blink reflex: here, an animal blinks whenever a "puff" of air is directed at their cornea. This stimulus-response pair is called "unconditioned stimulus" (US) and "unconditioned response" (UR).

If the unconditioned stimulus is paired with a so called conditioned stimulus (CS) (for example a tone, or a flash of light), the animal gradually learns to associate the conditioned stimulus with the unconditioned stimulus and will produce a conditioned response (CR) mimicking the unconditioned response even in the absence of the unconditioned stimulus.

One possible implementation of classical conditioning using the PES learning rule is given

3.6 Example: Adaptive Control

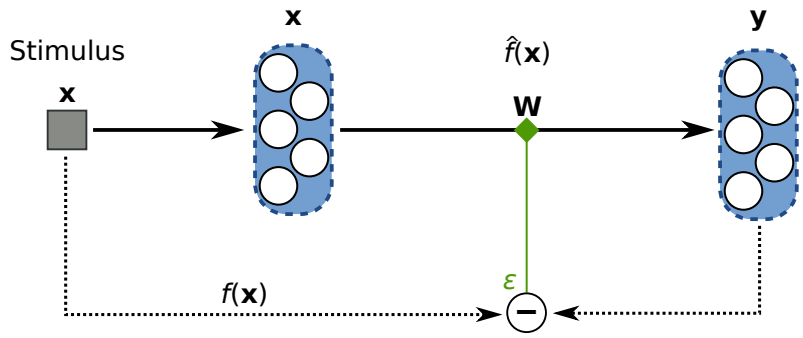
🚧 UNDER CONSTRUCTION 🚧

4 Unsupervised Learning

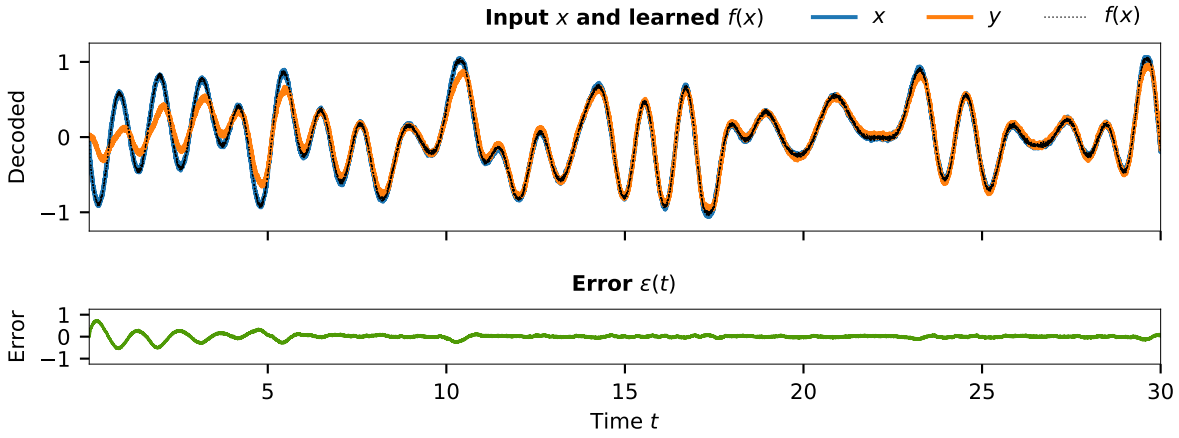
🚧 UNDER CONSTRUCTION 🚧

4.1 Principal Component Analysis

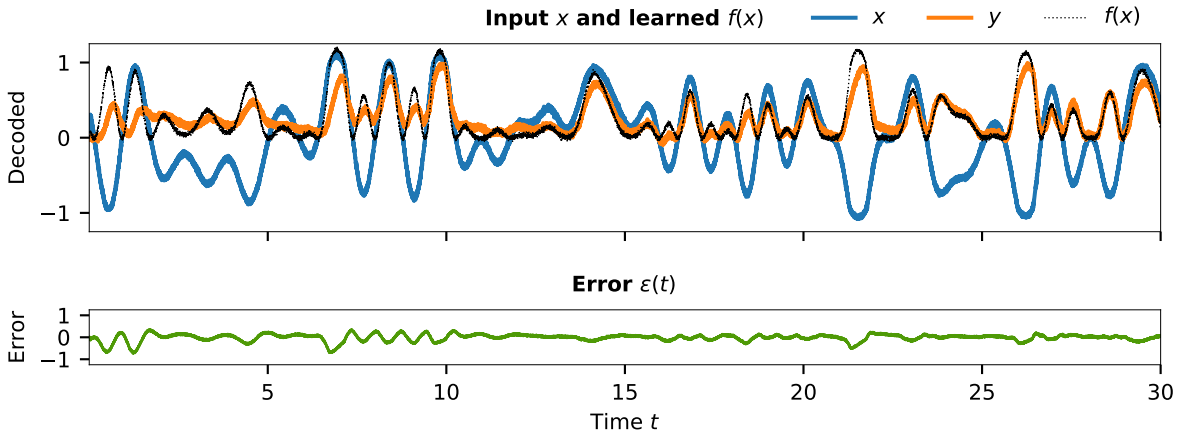
- **Basis Transformation** An orthonormal matrix $\mathbf{T} \in \mathbb{R}^{d \times d'}$ (*orthonormal:* $\mathbf{T}\mathbf{T}^\mathsf{T} = \mathbf{I}$) is a so called "basis transformation":



(a) Network used for the learning experiments depicted below



(b) Learning $f(x) = x$



(c) Learning $f(x) = x^2$

Figure 4: (a) Network used for the learning experiment below. Blue boxes correspond to spiking neuron populations of 100 neurons each. The learning rate is $\eta = 5 \times 10^{-5}$. **(b, c)** Learning different functions f using the PES learning rule. Blue lines correspond to the decoding from the first neuron population representing x, orange lines correspond to the decoding from the second neuron population representing $y = \hat{f}(x)$. The black line corresponds to the optimal target $f(x)$. ▩ *Code*

  - $\lambda = \mathbf{T}\mathbf{x}$
  - $\mathbf{x} = \mathbf{T}^{\mathsf{T}}\lambda$

  λ and $\mathbf{x}$ contain exactly the same information, but from a different perspective.

- **Idea:** Find the $\mathbf{T}$ that packs the "most information" into the first dimensions.
- **Then:**
  - **Can compress data**
    Can "throw away" (set to zero) coefficients λ_i with $i > d'$
  - **Can interpret the first dimensions**
- This is exactly what Principal Component Analysis (PCA) does!
- PCA: Eigenvectors of the data covariance matrix sorted by the magnitude of the eigenvalue
- Algorithm:

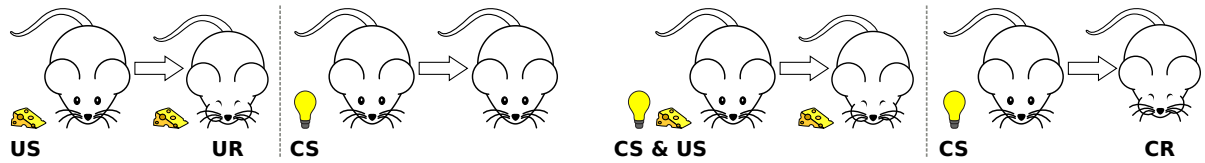
```
# Input X: (N x d) array
# Output T: (d x d) array
X_cen = X - np.mean(X, axis=0)
X_cov = (X_cen.T @ X_cen) / (N - 1)
E, T = np.linalg.eigh(X_cov)
# T sorted by magnitude of E
T = T.T[::-1, :]
```

4.2 Unsupervised Hebbian Encoder Learning

- Hebbian Learning: "What fires together wires together"
- $\Delta w_{ij} = \eta a_i a_j$
- Problem: Unstable weights go to infinity
- BCM rule (Bienenstock, Cooper, & Munro, 1982):

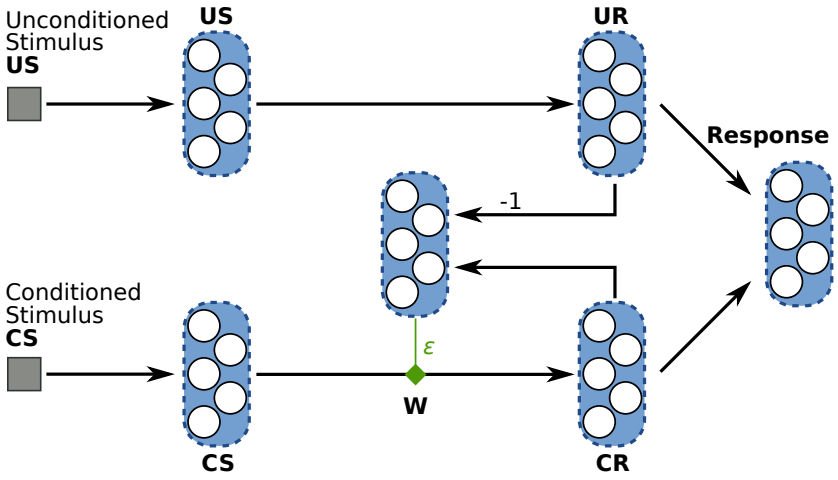
$$\Delta_{ij} = \eta a_i a_j (a_j - \theta)$$

- θ: Activity Threshold (recent average activity of the pre-neuron)
- One can show: weights trained by the Hebbian Learning Rules converge to the first principal component
- "Encoder Learning"
- Active field of research; how to learn multiple PCA-like encoders in a biologically plausible manner?

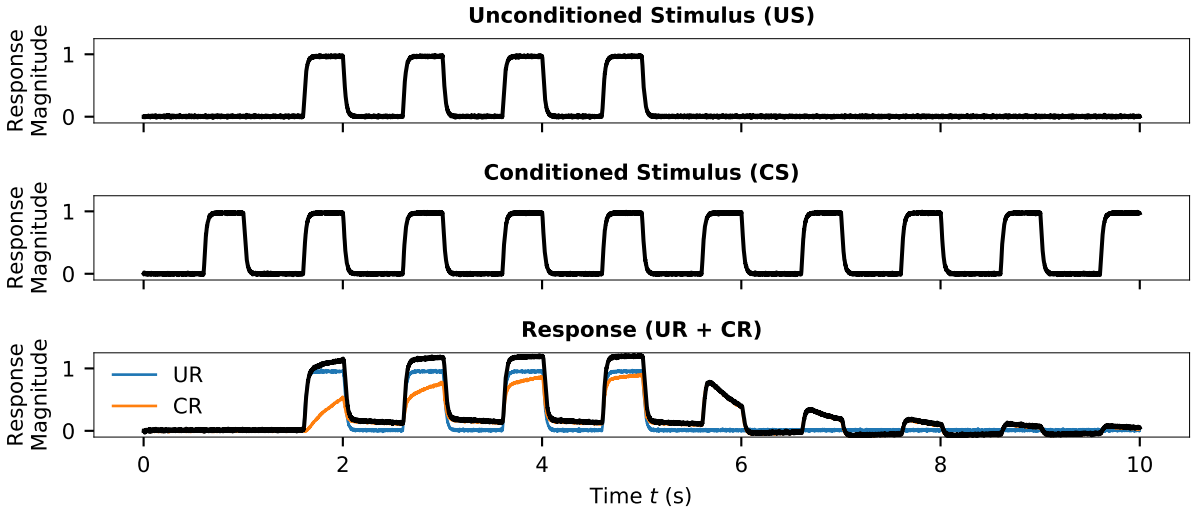


(a) Before conditioning

(b) After conditioning



(c) Network model



(d) Neural network implementing classical conditioning

Figure 5: Classical conditioning experiment. **(a, b)** Illustration of classical conditioning. An unconditioned stimulus (US, smell of cheese) causes an unconditioned response (UR, mouse is salviating). Paring a neutral conditioned stimulus (CS, light) with the unconditioned stimulus causes the animal to produce a conditioned response (CR). Adapted from Wikimedia. **(c)** Network model using the PES learning rule to implement classical conditioning. **(d)** Graphs showing the values represented by the individual populations.