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; March 3, 2020

UNIVERSITY OF
WATERLOO

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". Perhaps confusingly, 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)*
>
> *Learning* is a directed change in synaptic weights **W** while the network is active.

Conversely, we refer to anything that fits into the category of "past-affecting-the-future" phenomena as *adaptation*. Adaptation thus encompasses learning according to our definition, 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 **W** are defined as **W** = **ED** in a Neural Engineering Framework network. While the decoders **D** we are computing are optimal, this does not mean that the entire weight matrix **W** = **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 **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 **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 have a glance at the types of "learning" usually employed by computer scientists (who, unfortunately, chose to not follow the definition of "learning" we made up above). 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 uncreatively named 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 **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 shows 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 mere bunch 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.
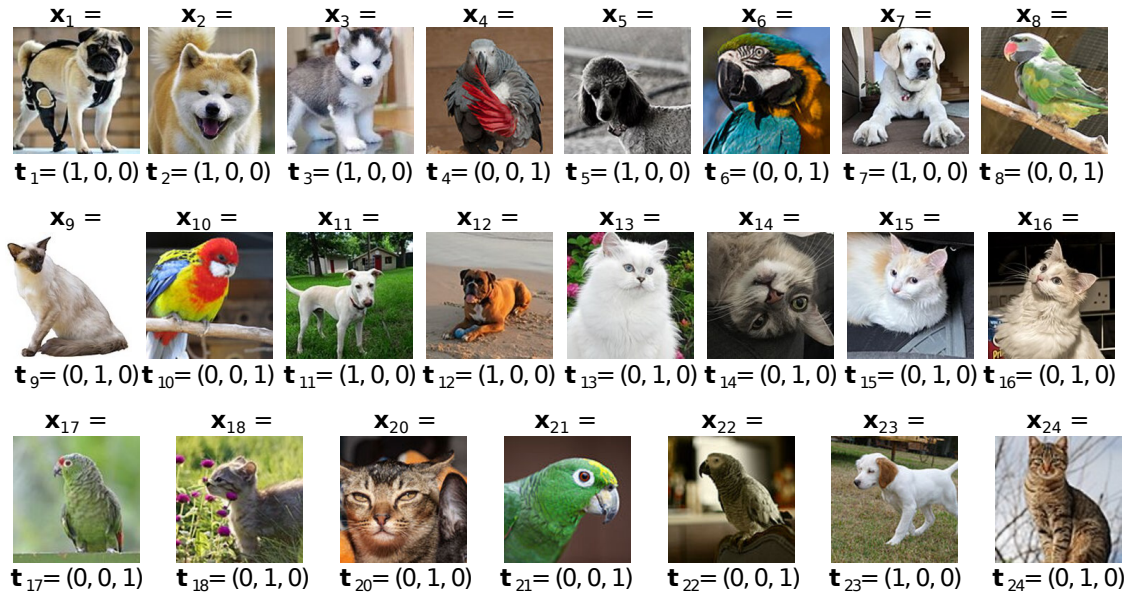
Figure 1: A supervised learning task. Given a set of N examples mapping an image $\mathbf{x}_k$ onto a one-hot encoded category vector $\mathbf{t}_k$, we would like the computer to learn a mapping from unseen images $\mathbf{x}$ onto categories $\mathbf{t}$.

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 $f_{\mathrm{GT}}(\mathbf{x}) = \mathbf{t}$ 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. Often, 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, which is the type of learning we defined above) 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

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

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: $f(\mathbf{x}; \mathbf{w}) = \varphi(\langle \mathbf{w}, \mathbf{x} \rangle)$.
- 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 $f_{\mathrm{GT}}(\mathbf{x}; \mathbf{w})$ 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

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

where $\mathbf{w}$ are our decoders, and $\mathbf{a}(\mathbf{x})$ 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.
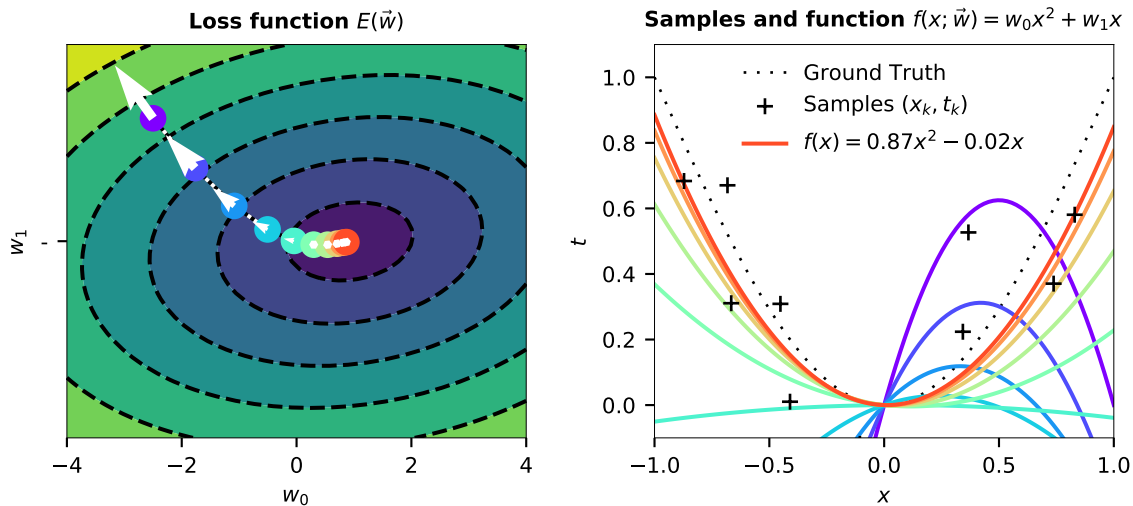
Figure 2: Gradient descent in action. We are trying to find parameters for the polynomial model function $f(x; \mathbf{w}) = w_0 x^2 + w_1 x$ given a set of samples (x_k, t_k). *Left:* Loss function over the parameters $\mathbf{w}$, as well as the trajectory of the paramters $\mathbf{w}$ as part of the 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 **D**. However, so far we have been using what is known as an *offline* method – we compute **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 **D**, followed by the "inference" phase during which we use **D** within our neural network.

In order to *learn* synaptic weights **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 **d**, such that $f(x(t)) \approx y^{\mathrm{d}}(t)$. This means that we have the following loss function

$$E(\mathbf{d}) = \frac{1}{2}\left(\left(\sum_{i=1}^{n} d_i a_i\big(x(t)\big)\right) - y^{\mathrm{d}}(t)\right)^2$$

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

$$= \varepsilon(t) a_i\big(x(t)\big),$$

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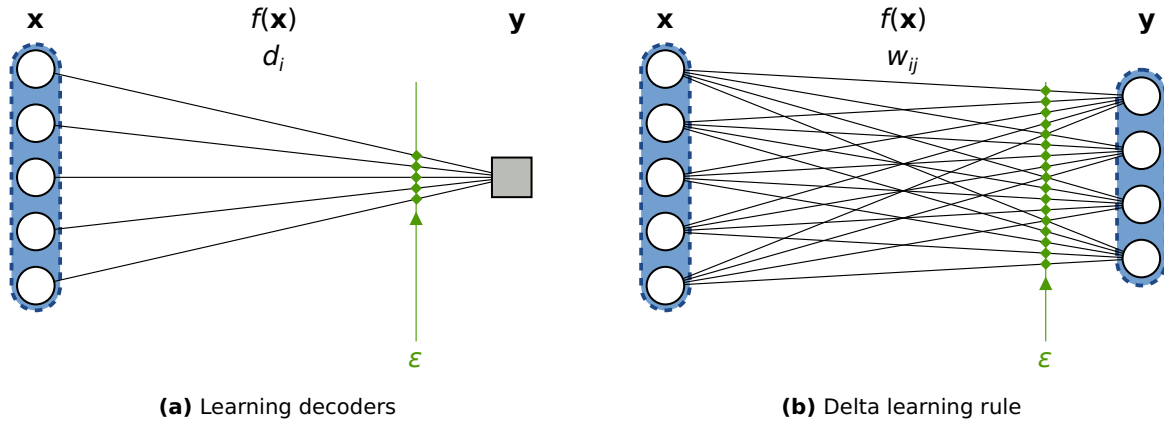
Figure 3: Learning decoders d_i **(a)** and synaptic weights **(b)** in an NEF network. We would like to learn an unknown function $\mathbf{y} = f(\mathbf{x})$ encoded in either the decoding weights d_i or in the weights between two neural populations. Weights are updated according to an error signal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

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

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

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

$$\Delta w_{ij} = \alpha_i e_i \Delta d_j = -\eta \big(\alpha_i \varepsilon(t) e_i \big) a_j(t).$$

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

(Prescribed Error Sensitivity (PES) Learning Rule)

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

Notice 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.

(a) Network used in the experiments depicted below

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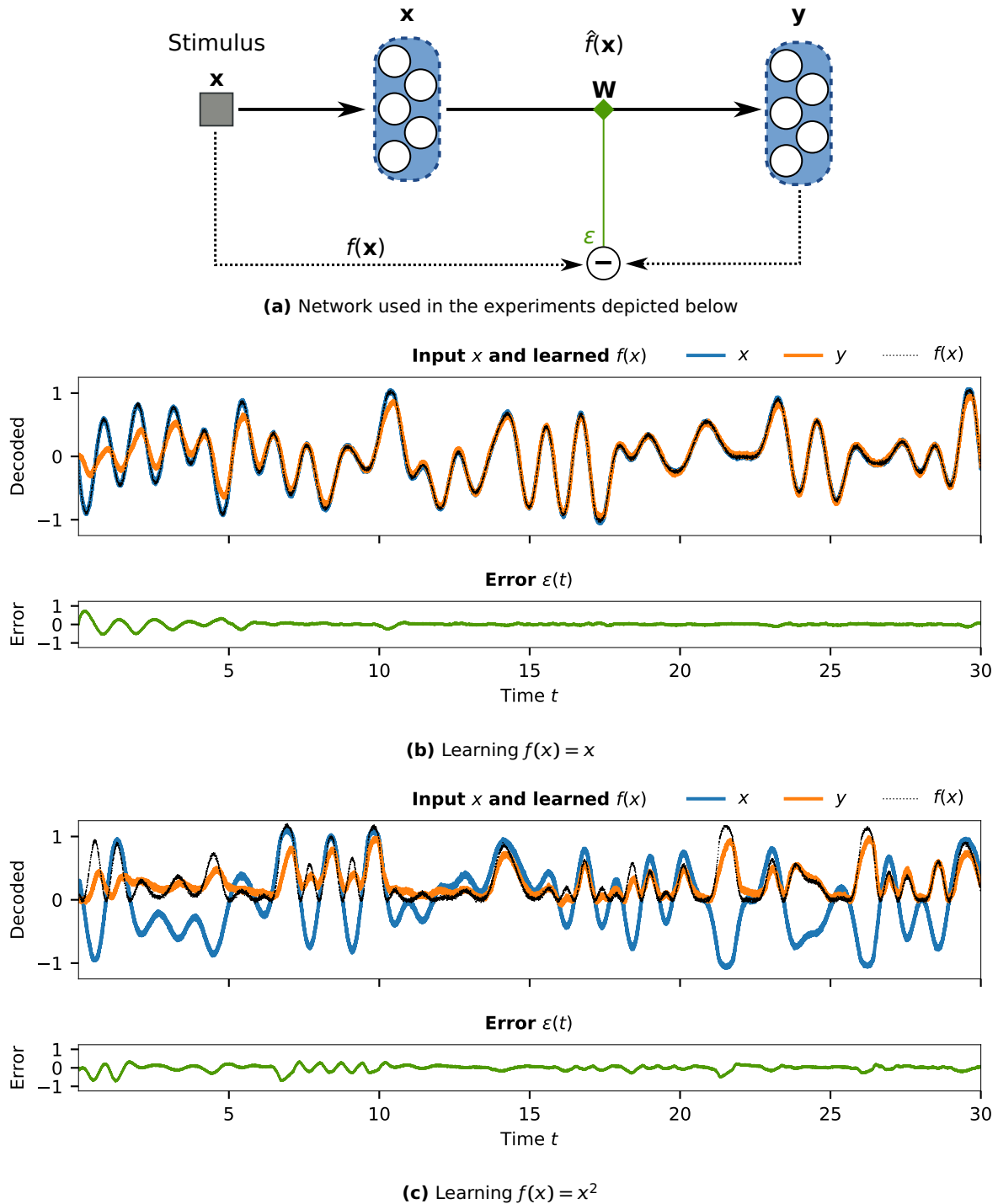
(c) Learning $f(x) = x^2$

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*

8

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 $\varepsilon(t)$, where "artificially" refers to the signal being generated outside of the system. 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.

> **Note:** Given an appropriate learning rate η and a long enough training period, learning a function in this way will often result in a smaller error than computing the decoders offline, the reason being that the PES rule has access to the actual spike data instead of "just" the rate approximation.
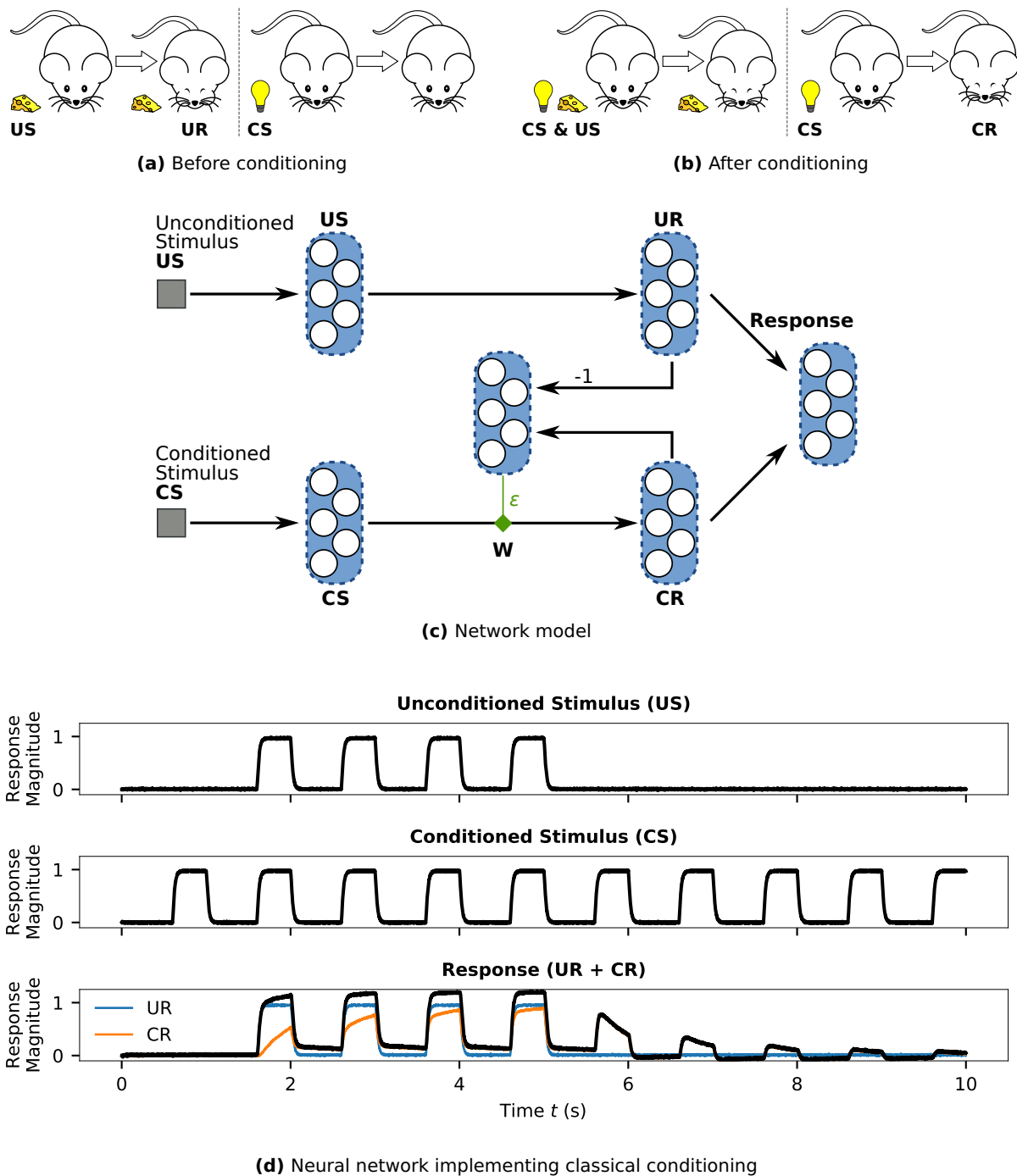
3.5   Example: Classical Conditioning

As mentioned, the above example relies on an "outside" error signal $\varepsilon(t)$. That is, the error signal is produced by the "environment" that had knowledge about the function f that we wanted to learn. That is, of course, unrealistic. What we would like to have is a system that internally generates the error signal $\varepsilon(t)$. One example where we can clearly see how such an error signal could be computed within a neural system itself is 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. In Pavlov's original experiment the unconditional stimulus is the smell of meat presented to a dog (US), that then starts to produce saliva (UR). Another 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). See also fig. 5a.

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 (fig. 5b).

One possible implementation of classical conditioning using the PES learning rule is depicted in fig. 5c. As visible, the system solely receives two modalities as sensory input.

(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 uncondi-tioned stimulus (US, smell of cheese) causes an unconditioned response (UR, mouse is salivating). 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 popula-tions.

4 Unsupervised Learning

Another way to eliminate the need for an external error signal is to not define an explicit target signal $\mathbf{y}^{\mathrm{d}}(t)$ at all, but to instead try to learn structure inherent to samples of data $\mathbf{x}_k$ without any external guidance. Often, what we would like to accomplish is some sort of *dimensionality reduction*. That is, we would like to map each high dimensional datum $\mathbf{x}_k$ onto a low dimensional space λ (the so called "latent space"), where each dimension in the low dimensional space has some dimension that intrinsically describes the data.

As with supervised learning, the assumption is that there exists a ground-truth function

$$f_{\mathrm{GT}}^{-1}(\lambda) : \mathbb{R}^{d'} \longrightarrow \mathbb{R}^{d}, \text{ with } d' \ll d \text{ and } f_{\mathrm{GT}}^{-1}(\lambda_k) = \mathbf{x}_k + \text{ noise}.$$

That is, we assume that the high-dimensional samples $\mathbf{x}_k$ we observe are actually the result of a process that only depends on a few variables λ.

Example: *Faces.* For a moment, think about a set of greyscale photographs of faces. Naturally, we could describe each of these photographs as a vector of pixel intensities $\mathbf{x}_k \in \mathbb{R}^d$, where d is the product of the width and height of the photographs in number of pixels. Typically, d would be on the order of magnitude of a few thousand (i.e., for a 32×32 pixel image we have $d = 1024$) to a few million ($d = 20 \times 10^6$ for a "20 megapixel" image).

However, we could try to convince ourselves that the most prominent properties of a face can be reduced onto a few dozen dimensions (a good analogy being the character editors present in popular role-playing computer games):

- Ratio between width and height

- Relative eye, mouth, nose distance

- Nose, eye, mouth shape

- Emotion (happy, angry, surprise, ...)

- Skin tone

- Hair color and length

Figure 6. Excerpt from the face database used in the following experiments. Source

We could devise a way to quantify each of these features in terms of a scalar value λ_i. Then, we can write a function f that takes a parameter vector λ and outputs a face.

The goal of unsupervised learning is to automatically determining both the parameter space λ, as well as an encoder f (encoding a set of parameters λ into the high dimensional space), and the decoder f^{-1} (estimating λ given a high-dimensional input).

4.1 Autoencoders

An extremely powerful dimensionality reduction method are so called autoencoders. The idea is to build a neural network $f(\mathbf{x}, \mathbf{w})$ such that after training $f(\mathbf{x}; \mathbf{w}) = \mathbf{x}$, i.e., the output of the network is equal to its input. Crucially however, the network posesses a so called "information bottleneck", a neural layer with just a few dimensions $d' \ll d$. The activities of the neurons in this layer correspond to the latent space λ. See fig. 8 for a sketch of an autoencoder network.

To train an autoencoders we optimize the loss $\|f(\mathbf{x}; \mathbf{w}) - \mathbf{x}\|$. Since the desired output is just the input, this can be interpreted as a supervised learning problem. Correspondingly, we can do stochastic gradient descent using back-propagation (see above). This method can yield quite impressive results (see this ⬚ Video for varying λ in a trained autoencoder).

Unfortunately, back-propagation in a multi-layer network is biologically implausible (cf. [1]). Correspondingly this method does not really qualify for our purpose of building a brain model.

✩ *Aside: Regularisation of λ – Variational Autoencoders.* One caveat when training autoencoders is overfitting (i.e., a small error on the training dataset X, but bad generalisation). While this problem is of course not exclusive to autoencoders, they tend to exhibit this behaviour in a particularly interesting way. Even with a bottleneck dimension of $d' = 1$, the training error goes to zero – the dataset is somehow being compressed into a single scalar λ_1. Of course, this does not mean that the autoencoder actually learned something useful, but merely that the encoding function $f(\lambda; \mathbf{w})$ "remembers" the entire dataset in its connection weights $\mathbf{w}$ and that λ_1 is effectively an index into our datapoints.

Crucially, as impressively shown by Piantadosi, 2018 [2] (see below) this may happen when learning a function with just a single parameter w, since a single real number can encode an infinite amount of information (or up to 64 bit of information when using a double precision floating point number on a computer).

To prevent this from happening, we need to apply some regularisation to the bottleneck of the autoencoder, either by discretising λ to much less than 64 bit, adding noise, or treating λ as parameters of a probability distribution from which we sample the input to the encoding layer. The latter method is known as a *Variational Autoencoder*.
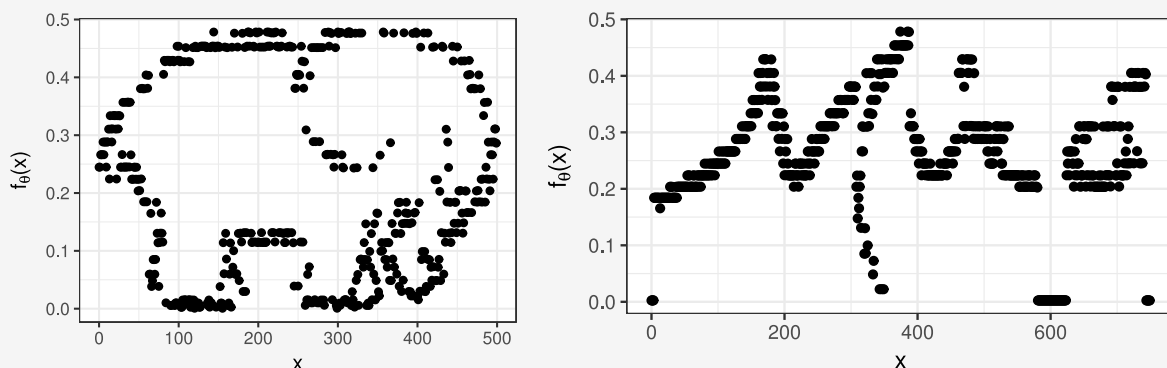
Figure 7. Compressing a large training set $\mathbf{X}$ into a single parameter θ. The depicted $f(x, \theta)$ "stores" two different datasets X in a single parameter $\theta \in [0, 1]$ each. Figure copied from [2].
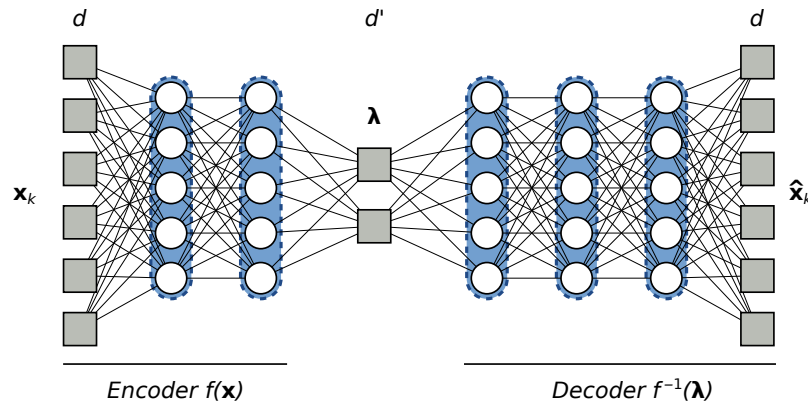
Figure 8: Network diagram of a potential autoencoder architecture. A neural network is trained to reproduce a d dimensional input $\mathbf{x}_k$ while feeding the data through a d' dimensional bottleneck with $d' \ll d$. Note that d would usually be much larger than what is shown here.

4.2 Principal Component Analysis

An interesting concept that – as we will see – is directly related to unsupervised learning in biologically plausible neural networks, is the so called Principal Component Analysis, often referred to as PCA. In short, the PCA computes a linear basis transformation $\mathbf{T}$ that transforms the input samples $\mathbf{x}_k$ in such a way that the most information (in terms of variance) is explained by the first dimensions of the resulting λ_k.

Basis Transformation In order to understand this a little better, let's first define what a *basis transformation* is. Let $\mathbf{T} \in \mathbb{R}^{d \times d}$ be an orthogonal matrix, that is $\mathbf{T}\mathbf{T}^\mathsf{T} = \mathbf{T}^\mathsf{T}\mathbf{T} = \mathbf{I}$, then $\mathbf{T}$ can be seen as a so called "basis transformation". The mathematical term for an orthogonal transformation matrix is *rotation*. Rotations are quite intuitive in two and three dimensions (they do exactly what their name suggests, i.e., they change the orientation of points in space without changing their distance), where we can easily visualise rotations, but we can no longer to so for $d > 3$. In particular, it holds

$$\lambda = \mathbf{T}\mathbf{x}, \quad \text{and} \quad \mathbf{x} = \mathbf{T}^\mathsf{T}\lambda, \qquad\qquad \text{where } \mathbf{T} \in \mathbb{R}^{d \times d} \text{ and } \mathbf{x}, \lambda \in \mathbb{R}^d.$$

Note the equalities: the transformation from $\mathbf{x}$ to λ is *lossless*. We can go from one space to the other without losing any information.

> **Note:** We have already seen some basis transformations in this course. For example, the Discrete Fourier Transformation (DFT), the Discrete Cosine Transformation (DCT), and the Laplace transformation can all be defined in terms of such a matrix $\mathbf{T}$.

Computing Principal Components The idea behind the PCA is the following. What if we could find the transformation $\mathbf{T}$ that packs the most "information" into the first dimensions of our rotated vectors λ? Then, we might be able to do the following.

1. **Compression**
   If we only look at the first d' dimensions of λ, i.e., discard (set to zero) all dimensions $i > d'$, we might still be able to get a good (though no longer lossless) reconstruction of **x**.

2. **Interpretation**
   If the first d' dimensions contain the "most information", these dimensions could have interesting semantics.

One way to define "the most information", is by the variance of our dataset along a certain axis. Hence, for the first PCA dimension, we want to project our dataset X onto a vector $\mathbf{t}_1$ such that the variance is maximal

$$\mathbf{t}_1 = \arg\max_{\mathbf{t}_1} \text{var}(\mathbf{t}_1\mathbf{X}), \text{ where } \mathbf{t}_1 \in \mathbb{R}^{1\times d} \text{ and } \mathbf{X} \in \mathbb{R}^{d\times N}.$$

We can find "the second best" transformation vector $\mathbf{t}_2$ by subtracting the data that is aligned with $\mathbf{t}_1$ from our dataset and then repeating the above process

$$\mathbf{t}_2 = \arg\max_{\mathbf{t}_2} \text{var}(\mathbf{t}_2(\mathbf{X} - \mathbf{t}_1^\mathsf{T}\mathbf{t}_1\mathbf{X})), \text{ where } \mathbf{t}_1 \in \mathbb{R}^{1\times d} \text{ and } \mathbf{X} \in \mathbb{R}^{d\times N}.$$

This process can be repeated d times, giving us vectors $\mathbf{t}_1, \ldots, \mathbf{t}_d$.

Eigendecomposition   Solving the above optimization problem as we did multiple times before, one can show, that the above process is equivalent to computing the Eigenvectors **V** of the covariance matrix **C** of **X**, sorted by the magnitude of the Eigenvalues λ_i. That is, the Eigenvector with the largest Eigenvalue is equal to $\mathbf{t}_1$, the Eigenvector with the second-largest Eigenvalue is equal to $\mathbf{t}_2$, and so on.

The covariance matrix of **X** is given as

$$\mathbf{C} = \frac{(\mathbf{X} - \bar{\mathbf{x}})^\mathsf{T}(\mathbf{X} - \bar{\mathbf{x}})}{N - 1},$$

the Eigenvalues and Eigenvectors are part of the so called Eigendecomposition. For any real symmetric square matrix **C** with linearly independent eigenvectors

$$\mathbf{C} = \mathbf{V}\Lambda\mathbf{V}^\mathsf{T},$$

where **V** is an orthogonal matrix, and Λ is a diagonal matrix of Eigenvalues.

In Python using numpy, the PCA can be computed as follows

```python
def PCA(X): # X: N x d matrix
    N, d = X.shape
    X_cen = X - np.mean(X, axis=0)
    C = (X_cen.T @ X_cen) / (N - 1)
    L, V = np.linalg.eigh(C) # "eigh" faster than "eig" for symmetric matrices
    return V.T[::-1, :] # d x d matrix
```

> ✎ **Note:** *Computing the PCA using Singular Value Decomposition (SVD).* An alternative to computing the PCA using Eigen decomposition is to use the so called "Singular Value Decomposition" (SVD) instead. The SVD is a generalised form of the Eigen decomposition for non-square matrices. This saves having to compute the covariance matrix **X**, making the above code a wee bit shorter:
>
> ```python
> def PCA_SVD(X): # X: N x d matrix
>     return np.linalg.svd(X - np.mean(X, axis=0))[2]
> ```
>
> To see why this works, consider that the SVD decomposes a matrix **X** into a product of three matrices **U**, **S**, **V** with $\mathbf{X} = \mathbf{USV}^\mathsf{T}$, where **S** is a diagonal matrix of the so called *singular values*, and both **U** and **V** are orthogonal matrices.
>
> Assume that we have the SVD of the centered data matrix **X** with
>
> $$(\mathbf{X} - \bar{\mathbf{x}}) = \mathbf{USV}^\mathsf{T}.$$
>
> Comparing this to the Eigen decomposition of the covariance matrix **C** we get
>
> $$\mathbf{C} = \frac{(\mathbf{X} - \bar{\mathbf{x}})^\mathsf{T}(\mathbf{X} - \bar{\mathbf{x}})}{N - 1} = \frac{\mathbf{VSU}^\mathsf{T}\mathbf{USV}^\mathsf{T}}{N - 1} = \mathbf{V}\frac{\mathbf{S}^2}{N - 1}\mathbf{V}^\mathsf{T} = \mathbf{VLV}^\mathsf{T}.$$
>
> Hence the matrix **V** of the SVD of the centered data matrix $\mathbf{X} - \bar{\mathbf{x}}$ is equivalent to the matrix **V** of the Eigen decomposition of the covariance matrix of **X**.

4.3   Example: Applying the PCA to Faces

An example applying the PCA to a set of faces is shown in fig. 9. The face database (available here), contains 84 grayscale images of 12 women with 7 different facial expressions. The images are normalised such that the eyes are located at approximately the same pixel location. Each image is 45 × 60 pixels large (resolution quartered compared to the original dataset), corresponding to a total dimensionality of $d = 2700$.

Since the individual principal components $\mathbf{t}_1, \ldots, \mathbf{t}_d$ are 2700 dimensional vectors, we can draw them as images in the same way we can draw the original $\mathbf{x}_k$ as images (fig. 9b). In this particular example, these images are known as "Eigenfaces". The first principal component vectors correspond to "average" faces in the dataset. That is, each component of λ_k affects the entire image – compare this to vector components in the original image $\mathbf{x}_k$ which only affect a single pixel (cf. fig. 9a). In particular, the first two basis vectors and the corresponding latent space dimensions λ_1 and λ_2 encode information about the skin tone and the background/hair color. Subsequent principal components weight information from individual face areas and can be interpreted as encoding the facial expressions of the subjects (fig. 9c).

Hence, the PCA seems to act as an unsupervised learner. We are able to automatically extract high-level semantic features from a set of data points without specifying what we were looking for – the individual principal components correspond to common patterns in the dataset **X**.

(a) Normal basis $\mathbf{I}$

(b) PCA basis $\mathbf{T}$
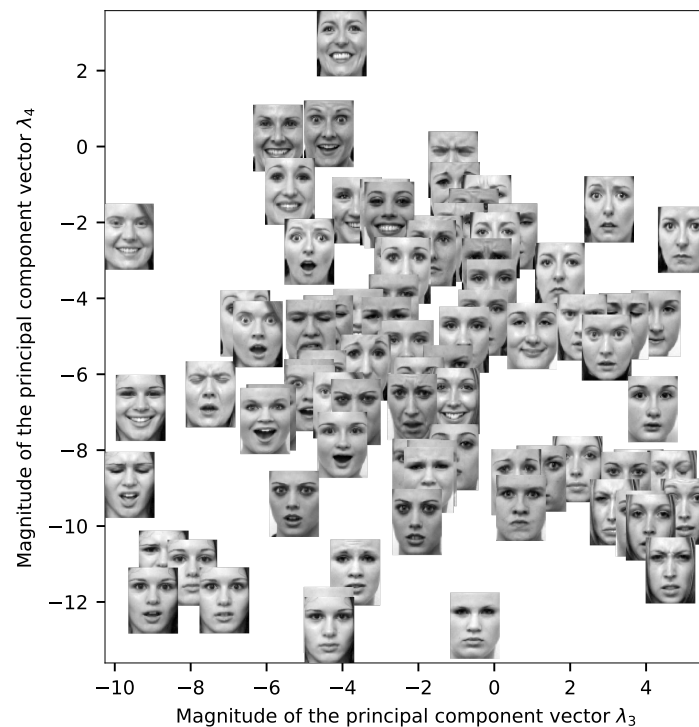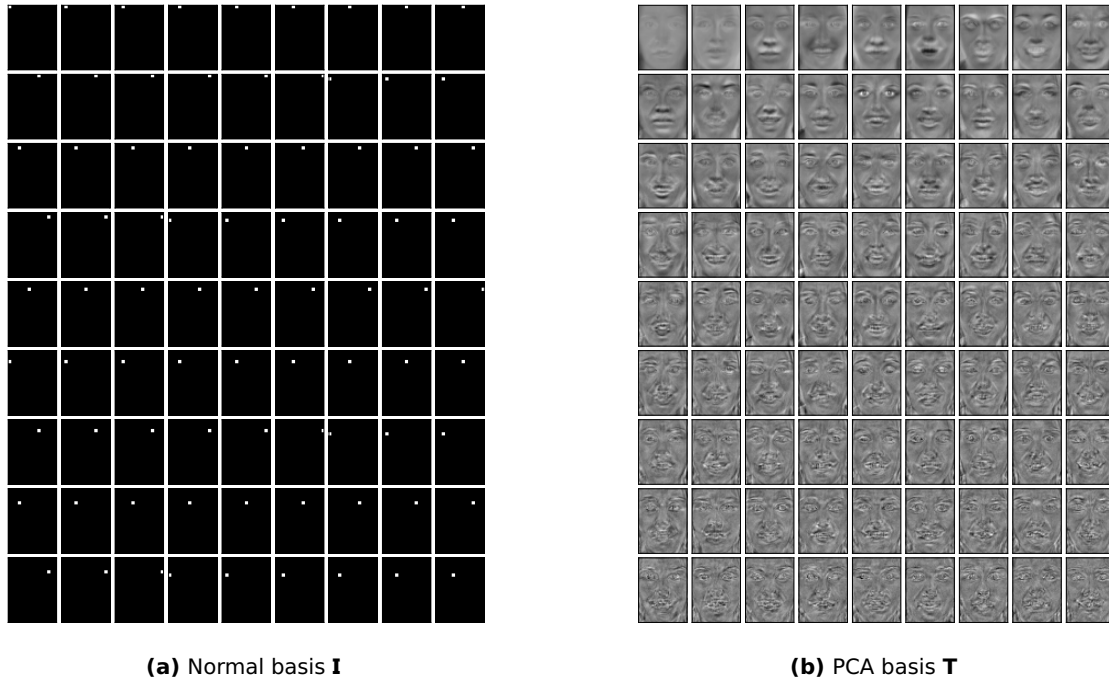
(c) Images plotted along the magnitude of the third and fourth principal components.

Figure 9: Principal component analysis of a set of faces. **(a)** Orthonormal basis used to represent pixel data. Each vector component in $\mathbf{x}_k$ corresponds to exactly one pixel. **(b)** Orthonormal basis vectors computed by the PCA. Each vector component in λ affects all pixels. Any image in the dataset $\mathbf{X}$ can be reconstructed by forming a weighted sum of these "basis images". **(c)** Plotting a set of images according to the magnitude of their third and fourth principal component. Images appear to be approximately spatially arranged by emotion. ⌨ *Code*

16

✏️ **Note:** *Machine learning and human data.* Be careful when using (unsupervised) machine learning on any data related to humans! In this example, the PCA algorithm determined on its own that skin color (principal component coefficient λ_1) contains the "most information" about the photos. This makes sense from a purely visual perspective (after all, most pixels in the photo depict skin). Luckily, the result of the PCA can be easily analysed as shown in fig. 9. Hence, we can decide not to use this dimension for subsequent processing. However, performing such analyses is much more difficult in more complex machine learning algorithms (i.e., "deep neural networks"). The machine might implicitly discriminate against certain groups of people without the engineers building the system noticing.

✏️ **Note:** *Limitations of the PCA.* While the PCA seems to be quite powerful in the above example, it is generally inferior to an autoencoder. We are restricting ourselves to linear transformations **T**. In many cases, nonlinear transformations are required to extract useful information from datasets.

4.4 Hebbian Learning

We have now discussed Principal Component Analysis, but it is still unclear how this relates to biology. One of the first attempts at characterising learning in biological systems was made by Donald Hebb in his 1949 book "The organization of behavior" [3]:

> When an axon of cell *A* is near enough to excite a cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that *A*'s efficiency, as one of the cells firing *B*, is increased.

An often made qualitative summary of the above statement is "what fires together wires together", that is, the synaptic strength w_{ij} between a post-neuron i and a pre-neuron j increases. We can easily turn this into a mathematical equation:

(Hebbian Learning)

$$\Delta w_{ij} = \eta a_i a_j \tag{3}$$

Evidence for this rule has (among others) been gathered by Eric Kandel in the synaptic connections responsible for the "siphon withdrawal reflex" in the sea slug *Aplysia Californica*.

There are two problems with this equation. First, it is *unstable* – the weights w_{ij} tend to grow to infinity. Correspondingly, this rule is never used directly, and instead, there are several learning rules based on eq. (3) that solve this problem. We will discuss some of these learning rules below. Depending on how the stabilisation of the equation is performed, these learning rules compute the principal components online.

Second, the equation does not take *causation* into account (i.e., which neuron caused the firing of the post neuron). This is solved by a "Spike-Time Dependent Plasticity", which we will also discuss below.

$t = 0$ $t = 1428$ $t = 2856$ $t = 4285$ $t = 5713$ $t = 7142$ $t = 8570$ $t = 9999$
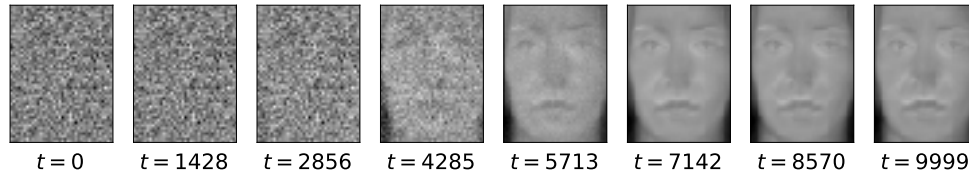
Figure 10: Normalised Hebbian Learning, or, equivalently, the Oja rule. Visualisation of the weight vector $\mathbf{w}_i$ connecting from a pre-population of 2700 neurons representing individual pixel intensities onto a single post-neuron. The initially random weight vector approaches the first principal component over time (left to right, t corresponds to iterations; compare to fig. 9). ▦ *Code*

4.5 Stable Hebbian Learning

Normalised Hebbian Learning   One way to prevent the synaptic weights from diverging is to force normalisation of the post-synaptic weights in the post-neuron i, that is

$$\Delta w_{ij} = \eta a_i a_j, \qquad \text{where } \mathbf{w}_i \leftarrow \frac{\mathbf{w}_i + \Delta \mathbf{w}_i}{\|\mathbf{w}_i + \Delta \mathbf{w}_i\|} \text{ in every time step}.$$

This way, the overall magnitude of the synaptic weight vector $\mathbf{w}_i$ describing all connections to the post-neuron i is limited. This could be interpreted as a "homeostasis" mechanism in the post-neuron. As mentioned above, one can show that — under the assumption of a linear post-neuron model – the weights $\mathbf{w}_i$ converge to the first principal component of the pre-population activity (see also fig. 10). Several methods exist that ensure learning of more principal components, however, this is an active field of research.

From a biological perspective, the above equation is a little implausible in that it assumes that the post neuron is able to compute a vector norm. Correspondingly, several alternative rules exist that implicitly incorporate the normalisation into the weight-update.

Oja rule   The Oja rule (proposed by Erkki Oja, 1982) can be derived from the above "normalised" equation and is equivalent to it for a post-neuron with a linear activation function

$$\Delta w_{ij} = \eta \left(a_i a_j - a_i^2 w_{ij} \right).$$

Instead of requiring repeated computation of the vector norm $\|\mathbf{w}_i\|$, this rule ensures stability by subtracting the weighted squared post-neuron activity. This rule can be used as a learning rule in Nengo connection objects.

BCM Rule   The BCM (Bienenstock, Cooper & Munro, 1981) rule balances the weight update by adding a target firing rate θ – if the post neuron is below that firing rate, the weight is increased, otherwise the weight is being decreased:

$$\Delta w_{ij} = \eta a_j a_i (\theta - a_i).$$

Often, θ is modelled as an averaged function of the past neural activity – this way, the post-neuron attempts to maintain a more or less steady firing rate. This rule can also be used as a learning rule in Nengo connection objects.

18

✐ **Note:** *Unsupervised learning from the perspective of individual neurons.* Another, perhaps overly metaphorical way to look at unsupervised learning is from the perspective of an individual neuron. Individual neurons have very little information about global information processing in the brain—all they "know" about the world are the firing patterns of their respective pre-neurons.

Still, the "goal" of an individual neuron could be seen as being "useful" to the organism as a whole. While a precise definition of "useful" is hard, certainly, a neuron not firing at all or just firing all the time is not very useful. Also, a neuron just repeating what their neighbouring neurons are doing might not be very helpful to the system as a whole—this is why we often find so called "lateral inhibition" between neighbouring neurons, acting as a sparsification of the population response.

One possible definition of "usefulness" is to extract as much information as possible from pre-synaptic events, accounting for what neighbouring neurons population are already sensitive to. In other words, a neuron should seek to extract patterns in the pre-neuron activities that other neurons in the population are not sensitive to, for example by each neuron being sensitive to a different principal component of the pre-population.

There are many different ways of characterising the amount of information in a post-neuron, relative to the pre-neuron activities. In case you are interested in this, have a look at Chapter 4 of Dayan and Abbott, *Theoretical Neuroscience*, 2001 [4].

4.6 Spike-Time Dependent Plasticity (STDP)

Note that Hebb's original theory quoted above is a little more precise than what is expressed in eq. (3). The term "takes part in firing it" suggests that there must be a *causal* relationship between the pre-neuron activity a_j^{pre} and the post-neuron activity a_i^{post}. Here, "causal" means that the pre-neuron must contribute to the firing of the post neuron; in other words, two neurons just "randomly" firing independently of each other will on average not cause an increase in synaptic strength.

One way to incorporate such a causal relationship into the learning rule is to consider individual spike times. If post-neuron B spikes *before* pre-neuron A spikes, then neuron A cannot have *caused* the spike. Conversely, if neuron B spikes shortly after neuron A, then there it is more likely that A caused the spike in B.

This is commonly modelled as "Spike-Time Dependent Plasticity" (STDP) [5]. Let $t_1^{\mathrm{pre},j}, \ldots, t_N^{\mathrm{pre},j}$ denote the spike times of the pre-neuron A, and $t_1^{\mathrm{post},i}, \ldots, t_{N'}^{\mathrm{post},i}$ denote the spike times of the post-neuron B. Then, the weight update is given as

$$\Delta w_{ij} = \eta \sum_{k=1}^{N} \sum_{\ell=1}^{N'} W\left(t_k^{\mathrm{post},i} - t_\ell^{\mathrm{pre},j}\right).$$
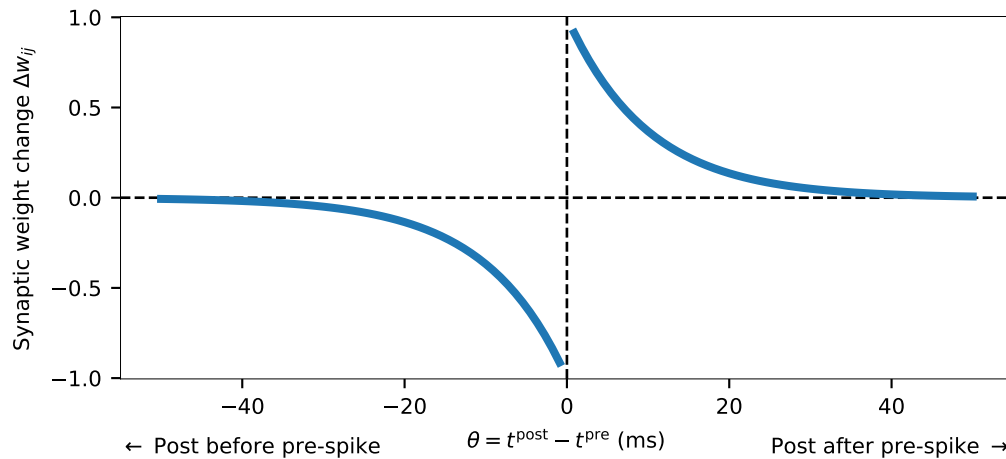
Figure 11: The STDP function $W(\theta)$. If the pre-spike occurs after the post-spike ($\theta < 0$), the synaptic weight decreases. Otherwise, if the pre-spike occurs before the post-spike ($\theta > 0$) the synaptic weight increases.

A popular choice for $W(\theta)$ (see fig. 11) is

$$W(\theta) = \begin{cases} \alpha^+ e^{-\theta/\tau^+} & \text{if } \theta > 0, \\ -\alpha^- e^{-\theta/\tau^-} & \text{if } \theta \leq 0, \end{cases}$$

where the time constants τ^+ and τ^- are typically in the range of $10\,\text{ms}$. There is strong experimental support for this type of learning – see [5] for more details.

References

[1]   Eric Hunsberger. "Spiking Deep Neural Networks: Engineered and Biological Approaches to Object Recognition". PhD thesis. University of Waterloo, 2018. URL: http://hdl.handle.net/10012/12819.

[2]   Steven T. Piantadosi. "One Parameter Is Always Enough". In: *AIP Advances* 8.9 (2018), p. 095118. DOI: 10.1063/1.5031956. eprint: https://doi.org/10.1063/1.5031956. URL: https://doi.org/10.1063/1.5031956.

[3]   Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. A Wiley Book in Clinical Psychology. Wiley, 1949.

[4]   Laurence F. Abbott and Peter Dayan. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Computational Neuroscience. MIT Press, Oct. 2001. 480 pp. ISBN: 978-0-262-04199-7. URL: https://mitpress.mit.edu/books/theoretical-neuroscience.

[5]   J. Sjöström and W. Gerstner. "Spike-Timing Dependent Plasticity". In: *Scholarpedia* 5.2 (2010), p. 1362. DOI: 10.4249/scholarpedia.1362.