Herbert Jaeger

Neural Networks (AI) (KIB.NNKI03)

Lecture Notes

V 0.8, May 25, 2020 (added Section 8)

BSc program in Artificial Intelligence
Rijksuniversiteit Groningen, Bernoulli Institute

Contents

Fasten your seatbelts – we enter the world of neural networks

Let's pick up some obvious facts from the surface and take a look at the bottomless voids that open up underneath them.

What makes you you is your brain. Your brain is a neural network. A neural network is a network made of neurons which connect to each other by synaptic links. Thus, one of the big question of science and life is how YOU are a NETWORK of interconnected NEURONS.

The answer seemed clear enough 77 years ago for the pioneers of what we now call *computational neuroscience*. In the kick-start work on neural networks (McCulloch and Pitts, 1943), a neuron x was cast as binary switch that could have two states — call them 0 and 1, or false and true — and this neuron x becomes switched depending on the 0-1 states of the neurons $y_1, \ldots, y_k$ which have synaptic links to x. A brain thus was seen as a Boolean circuit. The final sentence in that paper is "Thus in psychology, introspective, behavioristic or physiological, the fundamental relations are those of two-valued logic." In other words, brains (and you) are digital computers.

But.

What followed is 77 years of scientific and philosophical dispute, sometimes fierce, and with no winners to the present day. The more we have learnt about neurons and brains and humans and robots and computers, the more confusing the picture becomes. As of now (the year 2020), neither of what is a NEURON, a NETWORK, or a YOU is clear:

NEURONS. Biological neurons are *extremely* complicated physico-chemical objects, with hundreds of fast and slow chemical and genetic processes interacting with each other in a tree-like 3D volume with thousands of branches and roots. Even a supercomputing cluster cannot simulate in real-time all the detail of the nonlinear dynamical processes happening in a single neuron. It is a very much unanswered question whether this extreme complexity of a biological neuron is necessary for the functioning of brains, or whether it is just one of the whims of evolution and one could build equally good brains with much much simpler neurons.

NETWORKS. It is generally believed that the power of brains arises from the synaptic interconnectivity architecture. A brain is highly organized society (Minsky, 1987) of neurons, with many kinds of communication channels, local communication languages and dialects, kings and workers and slaves. However, the global, total blueprint of the human brain is not known. It is very difficult to experimentally trace neuron-to-neuron connections in biological brains. Furthermore, some neuron A can connect to another neuron B in many ways, with different numbers and kinds of synapses, attaching to

different sites on the target neuron. Most of this connectivity detail is almost impossible to observe experimentally. The Human Connectome Project (`https://en.wikipedia.org/wiki/Human_Connectome_Project`), one of the largest national U.S. research programs in the last years, invested a gigantic concerted effort to find out more and found that progress is slow.

YOU. An eternal, and unresolved, philosophical and scientific debate is about whether YOU can be *reduced* to the electrochemical mechanics of your brain. Even when one assumes that a complete, detailed physico-chemical model of your brain were available (it isn't), such that one could run a realistic, detailed simulation of your brain on a supercomputer (one can't), would this simulation explain YOU - entirely and in your essence? the riddles here arise from phenomena like consciousness, the subjective experience of *qualia* (that you experience "redness" when you see a strawberry), free will and other such philosophical bummers. Opinions are split between researchers / philosophers who, on the one side, claim that everything about you can be explained by a reduction to physico-chemical-anatomical detail, and on the other side claim that this is absolutely impossible. Disconcertingly, both have very good arguments. When you start reading this literature you spiral into vertigo.

And, on top of that, let me add, it is also becoming unclear again in these days what a COMPUTER is (I'll say more about that in the last lecture of this course).

Given that the scientific study of neural networks is so closely tied up with fundamental questions about ourselves, it is no wonder that enormous intellectual energies have been spent in this field. Over the decades, neural network research has produced a dazzling zoo of mathematical and computational models. They range from detailed accounts of the functioning of small neural circuits, comprising a few handfuls of neurons (celebrated: the modeling of a 30-neuron circuit in crustaceans (Marder and Calabrese, 1996)), to global brain models of grammatical and semantic language processing in humans (Hinaut et al., 2014); from less or more detailed models of single neurons (1963 Nobel prize for Alan Hodgkin and Andrew Huxley for a electrical engineering style formula describing the dynamics of a neuron in good approximation (Hodgkin and Huxley, 1952)) on the less detailed side, supercomplex geometrical-physiological *compartment models* on the high-detail side (Gouwens and Wilson, 2009)); from statistical physics oriented models that can "only" explain how a piece of brain tissue maintains an average degree of activity (van Vreeswijk and Hansel, 2001) to AI inspired models that attempt to explain every millisecond in speech understanding (Shastri, 1999); from models that aim to capture biological brains but thereby become so complex that they give satisfactory simulation results only if they are super delicately fine-tuned (Freeman, 1987) to the super general and flexible and robust neural learning architectures that have made deep learning the most powerful tool of modern machine

learning applications (Goodfellow et al., 2016); or from models springing from a few most elegant mathematical principles like the *Boltzmann machine* (Ackley et al., 1985) to complex neural architectures that are intuitively put together by their inventors and which function well but leave the researchers without a glimmer of hope for mathematical analysis (for instance the "neural Turing machine" of Graves et al. (2014)).

In this course I want to unfold for you this world of wonder. My goal is to make you aware of the richness of neural network research, and of the manifold perspectives on cognitive processes afforded by neural network models. A student of AI should, I am convinced, be able to look at "cognition" from many sides — from application-driven machine learning to neuroscience to philosophical debates — and at many levels of abstraction. All neural network models spring from the same core idea, namely intelligent information processing emerges from the collective dynamics in networks of simple atomic processing units. This gives the field a certain coherence. At the same time, it is amazing into how many different directions one can step forward from this basic idea. My plan is to present a quite diverse choice of neural network models, all classics of the field and "must-know's" for any serious AI/cognitive science/machine learning disciple. You will see that the world of neural networks has so much more to offer than just "deep learning" networks, which in these days outshine all other kinds in the popular, public perception.

This said, I will nonetheless start the course with an introduction to that currently most visible kind of neural networks, *feedforward neural networks*, from the classical, simple *Perceptrons* via *multilayer perceptrons* to a number of *deep learning* models. I position this material at the beginning of the course for two reasons. First, because I am aware that many if not most of you just (and justly) expect this topic to be treated, thus your thirst for knowledge will be quickly satisfied, and you can follow the rest of the course in a state of appreciative relaxation. Second, because these models most directly lend themselves to practical programming projects, such that you can swiftly start working on the practical project that accompanies the theory lectures.

But, please, be aware that scientific tides come and go, and the day will come when deep learning methods will recede in favor and some other magic will move into the foreground. Then the other themes and methods that you have learnt about in this course will help you to stay up to date with whatever else comes next. In this vein, it may interest you to learn that the now-dominant paradigm of deep learning directly emerged from quite another brand of neural networks, the *Boltzmann machine* (which you will get to know in a few weeks). The celebrated paper which today is widely seen as the starter for deep learning (Hinton and Salakuthdinov, 2006) in fact was written from the perspective of the statistical physics which rules Boltzmann machines, and the option to use them as an initialization submechanism for what today are called deep networks was only mentioned in passing. A few years later, the Boltzmann machine theme receded

into the background and deep deep deep began to rule. Such sudden shifts in focus will happen again! — as I will dare to forecast in the last lecture of this course.

1 A very fast rehearsal of machine learning basics

Because I want to start telling the neural network story with examples that currently dominate machine learning (ML), let us make sure we are on the same page concerning the basics of ML. The presentation in this section will be a condensed summary with minimal explanation because you should already have learnt this material in first-year courses. If you want to read up on details that you have forgotten, I recommend the ML lecture notes from my Master level course, which you can find on the Nestor course page under "Lecture Notes". For basic mathematical notation of sets and formal product constructions, consult the Appendix.

From the three main kinds of machine learning tasks — supervised learning, unsupervised learning, and reinforcement learning — here I only consider the first one. Furthermore, I will limit the presentation to data that are given in real-valued vector format, because that is the format used for most neural networks.

A very good, comprehensive yet concise introduction to basics of machine learning can be found in Section 5, "Machine Learning Basics", of the deep learning "bible" Goodfellow et al. (2016). A free online version is available. Here I cover only a small part of the material that you can find there. What I present here should however suffice to give you enough starter knowledge to use neural networks in supervised learning tasks in your semester project.

1.1 Training data.

A supervised learning tasks starts from *labelled training data*, that is a *sample* $S = (\mathbf{u}_i, \mathbf{y}_i)_{i=1,\dots,N}$ of input-output pairs $(\mathbf{u}_i, \mathbf{y}_i)$. For simplicity we assume that the inputs and output are real-valued vectors, that is $\mathbf{u}_i \in \mathbb{R}^K, \mathbf{y}_i \in \mathbb{R}^M$, although other formats are possible.

Brief note on notation: In most ML textbooks the inputs are denoted as $\mathbf{x}$, not $\mathbf{u}$ (and I used the $\mathbf{x}$ notation also in my machine learning lecture notes). However, we will later often deal with the internal processing states of neural networks. Following traditions in signal processing and dynamical systems maths, I will use the symbol $\mathbf{x}$ for these internal state vectors of a signal processing device, and use $\mathbf{u}$ for external inputs.

Two examples:

- Image classification tasks: the input *patterns* $\mathbf{u}_i$ are vectors whose entries are the red, green, blue intensity values of the pixels of photographic images. For instance, if the images are sized 600×800 pixels, the input pattern vectors are of dimension $n = 600 \cdot 800 \cdot 3 = 1,440,000$. The output vectors might

then be binary "one-hot encodings" of the picture classes to be recognized. For example, if the task is to recognize images of the ten sorts of handwritten digits 0 – 9, the output vector $\mathbf{y}_i$ for a "0" input image would be the 10-dimensional vector that has a 1 in the first component and is zero everywhere else.

- Function approximation: Assume that there is a function $f : \mathbb{R}^K \to \mathbb{R}^M$, and the $(\mathbf{u}_i, \mathbf{y}_i)$ are randomly drawn argument-value examples with some added noise ν, that is $\mathbf{y}_i = f(\mathbf{u}_i) + \nu$. For example (an example that I was tasked with in my past), the $\mathbf{u}_i$ could be vectors of measurement values taken from an array of sensors in a nuclear fission reactor, and the $m = 1$ dimensional output values $\mathbf{y}_i$ would indicate the time in milliseconds until the plasma in the reactor becomes unstable and explosively damages the containment. Function approximation tasks are also often called *regression* tasks.

1.2 Training objectives.

Given a training sample $S = (\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$, the task of supervised learning is to train a *model* of the input-output function f that gave rise to the observed training data. Such an ML model is an algorithm $\hat{f}$ which accepts inputs from $\mathbb{R}^K$ and computes outputs in $\mathbb{R}^M$. (A note on terminology: In statistical modeling, $\hat{f}$ is an *estimate* of the true function f. The $\hat{\cdot}$ notation is often used to denote estimates of something. Also, in statistics, functions that map input values to output values — here we named them f or $\hat{f}$ — are often called *decision functions*, a naming that I adopted in the Machine Learning lecture notes.)

A machine learning algorithm is a compuatational procedure which gets a training sample S as input and "learns" (that is, *computes* — and a stastician would say, *estimates*) a model $\hat{f} : \mathbb{R}^K \to \mathbb{R}^M$. In machine learning, the model $\hat{f}$ will be an executable algorithm (for instance, a neural network). Thus, a machine learning algorithm is an algorithm that transforms data in algorithms!

In order to decide whether an estimated model $\hat{f} : \mathbb{R}^K \to \mathbb{R}^M$ is "good" or "bad", one needs a well-defined measure to quantify the goodness of a model. This is achieved by a *loss function*

$$L : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}^{\geq 0}. \tag{1}$$

The idea is that a loss function measures the "cost" of a mismatch between the correct value $\mathbf{y}$ and the model output $\hat{f}(\mathbf{u})$, given a correct input-output pair $(\mathbf{u}, \mathbf{y})$ from the training data or from testing data. Higher cost means lower quality of $\hat{f}$. By convention, loss functions are non-negative and higher values means worse fit between true and estimated output. Many loss functions are in use. An experienced machine learning spends much care on tailoring the loss function to the learning problem that s/he is dealing with. Two basic examples:

- A loss that counts misclassifications in pattern classification tasks: assume that one is dealing with a pattern classification set-up where the target outputs $\mathbf{y}$ in the training/testing data and the outputs returned by a model $\hat{f}$ are m-dimensional binary "one-hot" class encoding vectors. Let $h : \mathbb{R}^K \to \{0,1\}^M$ be any candidate function for a model. Then the loss

$$L(h(\mathbf{u}), \mathbf{y}) = \begin{cases} 0, & \text{if } h(\mathbf{u}) = \mathbf{y} \\ 1, & \text{if } h(\mathbf{u}) \neq \mathbf{y} \end{cases} \tag{2}$$

  counts misclassifiation errors.

- A loss that penalizes quadratic errors of vector-valued targets:

$$L(h(\mathbf{u}), \mathbf{y}) = \|h(\mathbf{u}) - \mathbf{y}\|^2. \tag{3}$$

  This loss is often just called "quadratic loss", and it is one of the most frequently used ones in regression tasks.

In order to understand the nature of supervised learning tasks, one has to frame it in the context of probability theory. You find a summary of the necessary probability concepts and notation in the Appendix. We assume that the training and testing input-output data $\mathbf{u}, \mathbf{y}$ are obtained from *random variables* U, Y.

Learning algorithms should minimize the *expected* loss, that is, a good learning algorithm should yield a model $\hat{f}$ whose *risk*

$$R(\hat{f}) = E[L(\hat{f}(U), Y)] \tag{4}$$

is small. The expectation here is taken with respect to the true joint distribution $P_{U,Y}$ of the data-generating random variables U and Y. For example, in a case where U and Y are numerical RVs and their joint distribution is described by a pdf p, the risk of a candidate model h would be given by

$$R(h) = \int_{\mathbb{R}^K \times \mathbb{R}^M} L(h(\mathbf{u}), \mathbf{y})\, p(\mathbf{u}, \mathbf{y})\, d(\mathbf{u}, \mathbf{y}).$$

However, the true distribution $P_{U,Y}$ is unknown. The mission to find a model $\hat{f}$ which minimizes (4) is, in fact, hopeless. The only access to $P_{U,Y}$ that the learning algorithm affords is the scattered reflection of $P_{U,Y}$ in the training sample $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$.

A natural escape from this impasse is to tune a learning algorithm such that instead of attempting to minimize the risk (4) it tries to minimize the *empirical risk*

$$R^{\text{emp}}(h) = 1/N \sum_{i=1}^{N} L(h(\mathbf{u}_i), \mathbf{y}_i), \tag{5}$$

which is just the mean loss averaged over the training examples. Minimizing this empirical risk is an achievable goal, and a host of optimization algorithms for

all kinds of supervised learning tasks exist which do exactly this. That is, such learning algorithms find

$$\hat{f} = h_{\text{opt}} = \operatorname*{argmin}_{h \in \mathcal{H}} \; 1/N \; \sum_{i=1}^{N} L(h(\mathbf{u}_i), \mathbf{y}_i). \tag{6}$$

The set $\mathcal{H}$ is the *hypothesis space* – the search space within which a learning may look for an optimal h.

It is important to realize that every learning algorithm comes with a specific hypothesis space. For instance, in decision tree learning $\mathcal{H}$ is the set of all decision trees that use a given set of properties and attributes. Or, in linear regression, $\mathcal{H}$ is the set of all linear functions from $\mathbb{R}^K$ to $\mathbb{R}^M$ (I assume you have learnt about decision trees and linear regression in your first year courses). Or, if one sets up a neural network learning algorithm, $\mathcal{H}$ is typically the set of all neural networks that have a specific connection structure (number of neuron layers, number of neuros per layer); the networks in $\mathcal{H}$ then differ from each other only by the weights associated with the synaptic connections.

The empirical risk is often – especially in numerical function approximation tasks – also referred to as *training error*.

Here is an interim take-home summary:

- The ultimate goal for supervised learning algorithms is to estimate a model $\hat{f}$ which has a low risk (4), that is, which on average gives low-loss ("good") outputs on "testing" data drawn from the distribution $P_{U,Y}$.
- The only source of information that the learning algorithm has is the training sample $S = (\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$.
- Thus, it appears that the best one can do is to design a learning algorithm $\mathcal{A}$ which minimizes the empirical risk ("training error"), that is, upon input S the learning algorithm should return the solution of the minimization problem (6):

$$\mathcal{A}(S) = \operatorname*{argmin}_{h \in \mathcal{H}} \; 1/N \; \sum_{i=1}^{N} L(h(\mathbf{u}_i), \mathbf{y}_i). \tag{7}$$

In the next subsections we will see that the situation is more involved. First, in the kind of complex real-world learning tasks that neural networks usually are used for, algorithms that find exact minimal-training-error solutions do not exist. One only can design learning algorithms that find approximate solutions. Second, if one reduces the learning problem to finding minimal-training-error solutions, one will almost always run into the problem of *overfitting*. This second complication is by far more painful and important than the first one, and I will address it in the following subsection.

1.3 The overfitting problem.

While minimizing the empirical loss is a natural way of coping with the impossibility of minimizing the risk, it may lead to models which combine a low empirical risk with a high risk. This is the ugly face of overfitting.

The overfitting problem is connected to certain properties of learning algorithms which I will collectively refer to as the *flexibility* of a learning algorithm. The flexibility of a learning algorithm can be defined in several ways, and there are several methods to steer the flexibility of a learning algorithm. Flexibility is not a single, rigorously well-defined concept; it is an entire bundle of aspects which have been addressed in machine learning in many ways. But the general idea is always the same and can be stated in intuitive terms as *"a learning algorithm $\mathcal{A}$ is more flexible than another learning algorithm $\mathcal{B}$ if $\mathcal{A}$ can fit its computed models $\hat{f}$ more closely to the training data than $\mathcal{B}$ can do"*. More flexible learning algorithms thus can compute models with lower training error. Maximizing flexibility (by designing learning algorithms) can lead to doing too much of a good thing: a super flexible learning algorithm may even give zero training error, while performing very poorly on new testing data, rendering the found model $\hat{f}$ absolutely useless.

Because overfitting is such a fundamental challenge in supervised machine learning, I illustrate its manifestations with four examples. They are copied with slight adaptation of notation from the machine learning lecture notes.

1.3.1 Example 1: polynomial curve-fitting

This example is *the* standard textbook example for demonstrating overfitting. Let us consider a one-dimensional input, one-dimensional output regression task of the kind where the training data are of form $(u_i, y_i) \in \mathbb{R} \times \mathbb{R}$. Assume that there is some systematic relationship $y = f(u)$ that we want to recover from the training data. We consider a simple artificial case where the u_i range in $[0, 1]$ and the to-be-discovered true functional relationship f is $y = f(u) = \sin(2\,\pi\,u)$. The training data, however, contain a noise component, that is, $y_i = \sin(2\,\pi\,u_i) + \nu_i$, where ν_i is drawn from a normal distribution with zero mean and standard deviation σ. Figure 1 shows a training sample $(u_i, y_i)_{i=1,\ldots,11}$, where $N = 11$ training points u_i are chosen equidistantly.

We now want to solve the task of learning a good approximation for f from the training data (u_i, y_i) by applying polynomial curve fitting, an elementary technique you might be surprised to meet here as a case of machine learning. Consider an k-th order polynomial

$$p(x) = w_0 + w_1 u + \cdots + w_k u^k. \tag{8}$$

We want to approximate the function given to us via the training sample by a polynomial, that is, we want to find ("learn") a polynomial $\hat{f} = p(u)$ such that

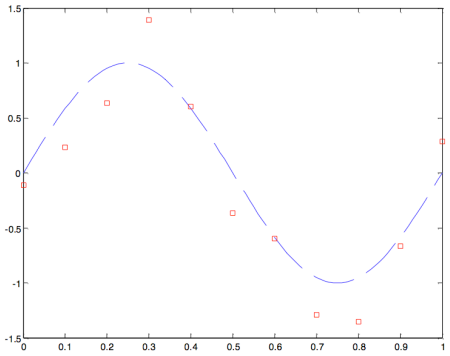

Figure 1: An example of training data (red squares) obtained from a noisy observation of an underlying "correct" function $sin(2\,\pi\,u)$ (dashed blue line).

$p(u_i) \approx y_i$. More precisely, we want to minimize the mean square error on the training data

$$\mathrm{MSE}^{\mathrm{train}} = \frac{1}{N} \sum_{i=1}^{N} (p(u_i) - y_i)^2.$$

At this moment we don't bother how this task is solved computationally but simply rely on the Matlab function *polyfit* which does exactly this job for us: given data points (u_i, y_i) and polynomial order k, find the coefficients w_j which minimize this MSE. Figure 2 shows the polynomials found in this way for $k = 1, 3, 10$.

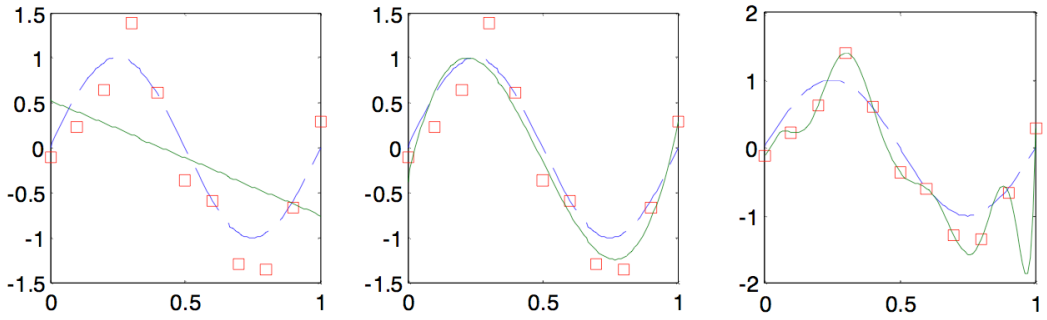

Figure 2: Fitting polynomials (green lines) for polynomial orders 1, 3, 10 (from left to right).

If we compute the MSE's for the three orders $k = 1, 3, 10$, we get $\mathrm{MSE}^{\mathrm{train}} = 0.4852, 0.0703, 0.0000$ respectively. Some observations:

- If we increase the order k, we get increasingly lower $\mathrm{MSE}^{\mathrm{train}}$.

- For $k = 1$, we get a linear polynomial, which apparently does not represent our original sine function well (*underfitting*).
- For $k = 3$, we get a polynomial that hits our target sine apparently quite well.
- For $k = 10$, we get a polynomial that perfectly matches the training data, but apparently misses the target sine function (*overfitting*).

The modelling flexibility is here defined through the polynomial order k. If it is too small, the models are too inflexible and *underfit*; if it is too large, we earn overfitting.

However, please switch now into your most critical thinking mode and reconsider what I have just said. *Why*, indeed, should we judge the linear fit "underfitting", the order-3 fit "seems ok", and the order-10 fit "overfitting"? There is no other ground for justifying these judgements than our visual intuition! In fact, the order-10 fit might be the right one if the data contain no noise! the order-1 fit might be the best one if the data contain a lot of noise! We don't know!

1.3.2 Example 2: pdf estimation

This example comes from the domain of unsupervised learning. In unsupervised learning, the training data are just a collection of points $S = (\mathbf{u}_i)_{i=1,\ldots,N}$ in $\mathbb{R}^K$, taken from a distribution P_U generated by a random variable U, and the objective is to estimate a model of the distribution P_U. One typical way to specify such models is by charactering them through a probability density function (pdf) $p_U : \mathbb{R}^K \to \mathbb{R}^{\geq 0}$. The overfitting problem arises here just as dramatically as in supervised training. I include this example because it is visually striking.

Let us consider the task of estimating a 2-dimensional pdf over the unit square from 6 given training data points $\{\mathbf{u}_i\}_{i=1,\ldots,6}$, where each $\mathbf{u}_i$ is in $[0, 1] \times [0, 1]$. This is an elementary unsupervised learning task, the likes of which frequently occur as a subtask in more involved learning tasks, but which is of interest in its own right. Figure 3 shows three pdfs which were obtained from three different learning runs with models of increasing flexibility (I don't explain the learning algorithms here — for the ones who know about it: simple Gaussian Parzen-window models where the degree of admitted flexibility was tuned by kernel width). Again we witness the fingerprints of under/overfitting: the low-flexibility model seems too "unbending" to resolve any structure in the training point cloud (underfitting), the high-flexibility model is so volatile that it can accomodate each individual training point (presumably overfitting).

But again, we don't really know...

1.3.3 Example 3: learning a decision boundary

Figure 4 shows a schematic of a classification learning task where the training patterns are points in $\mathbb{R}^2$ and come in two classes. When the trained model

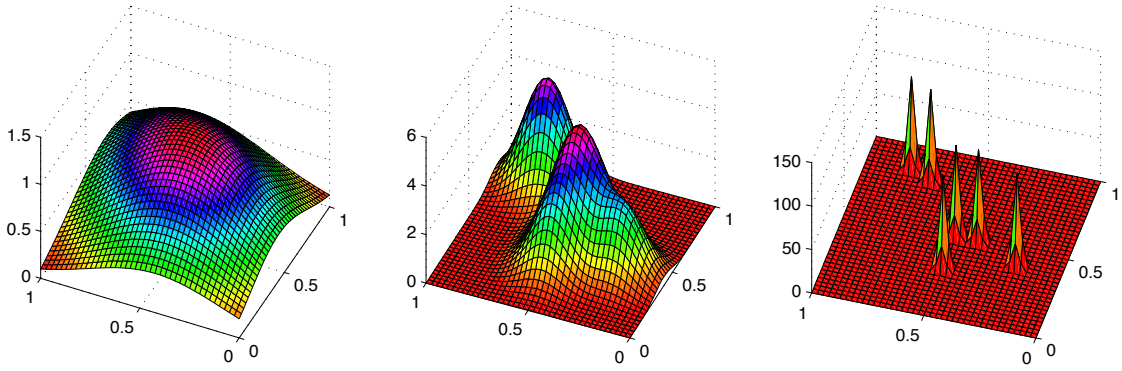

Figure 3: Estimating a pdf from 6 data points. Model flexibility grows from left to right. Note the different scalings of the z-axis: the integral of the pdf is 1 in each of the three cases.

is too inflexible (left panel), the decision boundary is confined to a straight line, presumably underfitting. When the flexibility is too large, each individual training point can be "lasso-ed" by a sling of the decision boundary, presumably overfitting.

Do I need to repeat that while these graphics *seem* to indicate under- or over-fitting, we do not actually *know*?

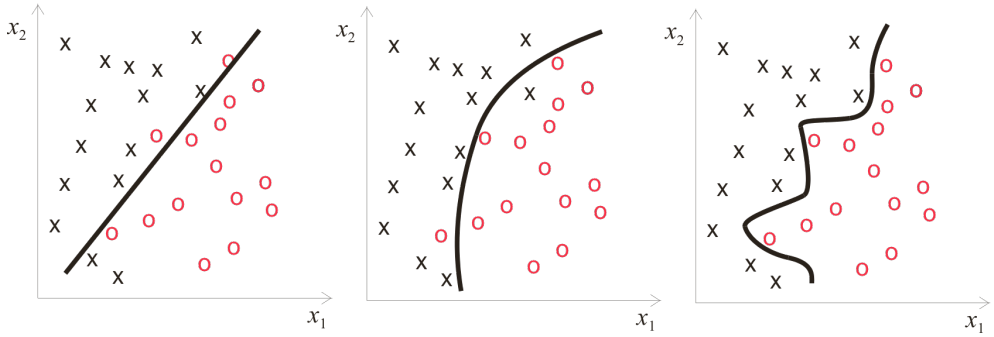

Figure 4: Learning a decision boundary for a 2-class classification task of 2-dimensional patterns (marked by black "x" and red "o").

1.3.4 Example 4: furniture design

Overfitting can also hit you in your sleep, see Figure 5.

Again, while this looks like drastic overfitting, it would be just right if all humans sleep in the same folded way as the person whose sleep shape was used for training the matrass model.

Figure 5: A nightmare case of overfitting. Picture spotted by Yasin Cibuk (2018 ML course participant) on `http://dominicwilcox.com/portfolio/bed/`, designed and crafted by artist Dominic Wilcox, 1999. Quote from the artist's description of this object: "I used my own body as a template for the mattress". From a ML point of view, this means a size $N = 1$ training data set.

1.3.5 THE nonidentical twin curves

There are several ways how to tune the flexibility of a learning algorithm. I will outline some of the most important ones in a moment. But before we consider technicalities, I would like to present THE curve (Figure 6). The x-axis gives a learning algorithm flexibility range, meaning that the further to the right, a more flexible learning algorithm is used. On the y-axis two curves are plotted. For each flexibility adjustment of a learning algorithm (x-axis), these two curves give the risk (testing error) and the empirical risk (training error) of the model found by the respective learning algorithm.

Three points are worth pointing about in this diagram:

- On the left end (very inflexible learning algorithms), both training and testing error are high – due to the low flexibility of the algorithm the obtained models are just too simple to capture relevant structure in the data distribution.

- On the right end (super flexible learning algorithms), the training error is small — often, indeed, it can be pushed down to zero — because the high flexibility of the learning algorithm allowed it to fit itself around each individual training point. In human psychology terms: the model has learnt

the data by heart. But, the testing error is high: memorizing teaching material by heart is stupid; a good learner extracts the underlying regularities and laws from the data and can transfer that valuable, extracted, condensed knowledge to apply in new (testing) situations.

- Somewhere in the mid-range of flexibility the testing error has a minimum. This corresponds to learning with an algorithm whose flexibility has been optimially tuned. It will return models that will perform best when applied to new input data.

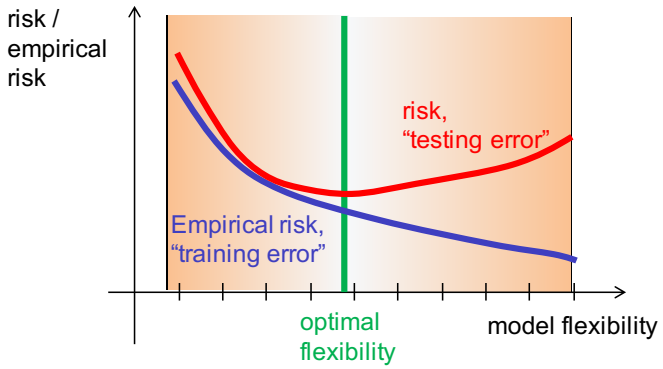

Figure 6: The generic, universal, core challenge of machine learning: finding the right model flexibility which gives the minimal risk.

In order to find the sweet spot of optimial flexibility, two technical conditions must be satisfied:

1. One must have an effective method to change the flexibility of learning algorithms.
2. One must have an effective method to *estimate* the risk (red curve in the figure).

If one has these two mechanisms available, one can find the sweet spot (green line in our figure) by a systematic sweep through flexibilities, learning models for each flexibility, estimate the risk, and settle for the flexibility that has minimal estimated risk. In the next two subsections I outline how these two mechanisms can be instantiated.

1.4 How to tune model flexibility

There are a number of approaches to tune the flexibility of a learning algorithm. In this subsection I list some of the most commonly applied ones (copied with simplifications and cuts from the machine learning lecture notes).

1.4.1 Tuning learning flexibility through model class size

In the polynomial curve fitting example from Section 1.3.1, the model parameters were the monomial coefficients $w_0, \ldots, w_k$ (compare Equation 8). After fixing the polynomial order k, the polynomial $p(u)$ with minimal training error was selected from the set $\mathcal{H}_k = \{p : \mathbb{R} \to \mathbb{R} \,|\, p(x) = \sum_{j=0}^{k} w_j \, u^j\}$, that is, the learning algorithm solved the minimization problem

$$\hat{f} = \operatorname*{argmin}_{p \in \mathcal{H}_k} \; 1/N \; \sum_{i=1}^{N} (p(u_i) - y_i)^2. \tag{9}$$

It is clear that $H_1 \subset H_2 \subset \ldots$. It is also clear that the training error can only shrink when k grows, because the set of candidate solutions H_k grows with k, thus there are more candidate solutions to pick from for the optimization algorithm.

Generalizing from this example, we can see one way to obtain a sequence of learning algorithms of increasing flexibility: A *model class inclusion sequence* is a sequence $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \ldots$ of sets of candidate models, and there is a known way to design learning algorithms that can solve the minimization problem (9) for each $\mathcal{H}_j$.

There are many ways how one can set up a sequence of learning algorithms which pick their respective optimal models from such a model class inclusion sequence. In most cases this will just mean to admit *larger* models with more tuneable parameters for "higher" classes. In polynomial curve fitting this meant to admit polynomials of growing order. In neural network training this means to run the learning algorithm on neural networks of increasing size.

1.4.2 Using regularization for tuning modeling flexibility

The flexibility tuning mechanism explained in this subsection is simple, practical, and in widespread use. It is called *model regularization*. Note that the word "regularization" is also often used in a more general way to denote *any* method for tuning flexibility of a learning algorithm.

When one uses regularization to vary the modeling flexibility, one does not vary the model candidate class $\mathcal{H}$ at all. Instead, one varies the optimization objective (6) for minimizing the training error.

The basic geometric intuition behind modeling flexibility is that low-flexibility models should be "smooth", "more linear", "flatter", admitting only "soft curvatures" in fitting data; whereas high-flexibility models can yield "peaky", "rugged", "sharply twisted" curves (see again Figures 2, 3, 4).

When one uses model regularization, one fixes a single model structure and size with a fixed number of trainable parameters, that is, one fixes $\mathcal{H}$. Structure and size of the considered model should be rich and large enough *to be able to overfit* (!) the available training data. Thus one can be sure that the "right" model is contained in the search space $\mathcal{H}$.

The models in $\mathcal{H}$ are typically characterized by a set of trainable parameters. In polynomial curve fitting these parameters are the monomial coefficients, and for a fixed neural network structure, it would be the set of all synaptic weights. Following the traditional notation in the machine learning literature we denote this collection of trainable parameters by θ. This is a vector that has as many components as there are trainable parameters in the chosen kind of model. We assume that we have M tuneable parameters, that is $\theta \in \mathbb{R}^M$.

Such a high-flexibility model type would inevitably lead to overfitting when an "optimal" model would be learnt using the basic learning equation (6) which I repeat here for convenience:

$$\hat{f} = h_{\text{opt}} = \underset{h \in \mathcal{H}}{\text{argmin}} \; 1/N \sum_{i=1}^{N} L(h(\mathbf{u}_i), \mathbf{y}_i).$$

In order to dampen the exaggerated flexibility of this baseline learning algorithm, one adds a *regularization term* (also known as *penalty term*, or simply *regularizer*) to the loss function. A regularization term is a cost function $\text{reg} : \mathbb{R}^M \to \mathbb{R}^{\geq 0}$ which penalizes model parameters θ that code models with a high degree of geometrical "wiggliness".

The learning algorithm then is constructed such that it solves, instead of (6), the *regularized* optimization problem

$$h_{\text{opt}} = \underset{h \in \mathcal{H}}{\text{argmin}} \; 1/N \sum_{i=1}^{N} L(h(\mathbf{u}_i), \mathbf{y}_i) + \alpha^2 \, \text{reg}(\theta_h). \tag{10}$$

where θ_h is the collection of parameter values in the candidate model h.

The design of a useful penality term is up to your ingenuity. A good penalty term should, of course, assign high penalty values to parameter vectors θ which represent "wiggly" models; but furthermore it should be easy to compute and blend well with the algorithm used for empirical risk minimization.

Two examples of such regularizers:

1. In the polynomial fit task from Section 1.3.1 one might consider for $\mathcal{H}$ all 10th order polynomials, but penalize the "oscillations" seen in the right panel of Figure 2, that is, penalize such 10th order polynomials that exhibit strong oscillations. The degree of "oscillativity" can be measured, for instance, by the integral over the (square of the) second derivative of the polynomial p,

$$\text{reg}(\theta) = \text{reg}((w_0, \ldots, w_{10})) = \int_0^1 \left(\frac{d^2 \, p(u)}{du^2} \right)^2 du.$$

   Investing a little calculus (good exercise! not too difficult), it can be seen that this integral resolves to a quadratic form $\text{reg}(\theta) = \theta' \, C \, \theta$ where C is an 11×11 sized positive semi-definite matrix. That format is more convenient to use than the original integral version.

2. A popular regularizer that often works well is just the squared sum of all model parameters,

$$\text{reg}(\theta) = \sum_{w \in \theta} w^2.$$

This regularizer favors models with small absolute parameters, which often amounts to "geometrically soft" models. This regularizer is popular among other reasons because it supports simple algorithmic solutions for minimizing risk functions that contain it. It is called the *L_2-norm regularizer* (or simply the *L_2 regularizer*) because it measures the (squared) L_2-norm of the parameter vector θ.

Computing a solution to the minimization task (10) means to find a set of parameters which simultaneously minimizes the original risk and the penalty term. The factor α^2 in (10) controls how strongly one wishes the regularization to "soften" the solution. Increasing α means downregulating the model flexibility. For $\alpha^2 = 0$ one returns to the original un-regularized empirical risk (which would likely mean overfitting). For $\alpha^2 \to \infty$ the regularization term entirely dominates the model optimization and one gets a model which does not care anymore about the training data but instead only is tuned to have minimal regularization penalty. In case of the L_2 norm regularizer this means that all model parameters are zero – the ultimate wiggle-free model; one should indeed say the model is dead.

When regularization is used to steer the degree of model flexibility, the x-axis in Figure 6 would be labelled by α^2 (highest α^2 on the left, lowest at the right end of the x-axis).

Using regularizers to vary model flexibility is often computationally more convenient than using different model sizes, because one does not have to tamper with differently structured models. One selects a model type with a very large (unregularized) flexibility, which typically means to select a big model with many parameters (maybe a neural network with hundreds of thousands of synaptic connections).

1.4.3 Tuning model flexibility through adding noise

Another way to tune model flexibility is to add noise. Noise can be added at different places. Here I mention two scenarios.

Adding noise to the training input data. If we have a supervised training dataset $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$ with vector patterns $\mathbf{u}_i$, one can enlarge the training dataset by adding more patterns obtained from the original patterns $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$ by adding some noise vectors to each input pattern $\mathbf{u}_i$: for each $\mathbf{u}_i$, add l variants $\mathbf{u}_i + \nu_{i1}, \ldots, \mathbf{u}_i + \nu_{il}$ of this pattern to the training data, where the ν_{ij} are i.i.d. random vectors (for instance, uniform or Gaussian noise). This increases the number of training patterns from N to $(l+1)N$. The more such noisy variants are added and the stronger the noise,

the more difficult will it be for the learning algorithm to fit all data points, and the smoother the optimal solution becomes – that is, the more one steers to the left (underfitting) side of Figure 6. Adding noise to training patterns is a very common strategy.

Adding noise while the optimization algorithm runs. If the optimization algorithm used for minimizing the empirical risk is iterative, one can "noisify" it by jittering the intermediate results with additive noise. The "dropout" regularization trick which is widely used in deep learning is of this kind. Again, the effect is that the stronger the algorithm is noisified, the stronger the regularization, that is the further one steers to the left end of Figure 6.

1.5 How to estimate the risk of a model

As I mentioned at the end of Section 1.3.5, besides being able to navigate effectively on the flexibility axis of THE twin curve (Figure 6), one also must have an effective means to estimate the risk (testing error) of estimated models $\hat{f}$, if one wants to locate the best degree of flexibility.

There exist a number of analytical formulas which allow one to estimate the risk of some model $\hat{f}$ in a mathematical analytical way (for instance, the *Akaike information criterion*, `https://en.wikipedia.org/wiki/Akaike_information_criterion`, is relatively popular). However, such analytical estimates of the risk are reliable only if additional conditions concerning the nature of the sampling distribution and the model are satisfied, and these conditions may be difficult to verify. Therefore, in daily practice one uses more often another approach, which is robust and generally applicable and easy to implement and needs no insight into analytical properties of the data distribution nor the model. The price to pay is a potentially high compuational cost. This is the method of *cross-validation*.

Here is the basic idea of cross-validation.

In order to determine whether a given model is under- or overfitting, one would need to run it on test data that are "new" and not contained in the training data. This would allow one to get a hold on the red curve in Figure 6.

However, at training time only the training data are available.

The idea of cross-validation is to artificially split the training sample $S = (\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$ into two subsets $T = (\mathbf{u}_i, \mathbf{y}_i)_{i\in I}$ and $V = (\mathbf{u}'_i, \mathbf{y}'_i)_{i\in I'}$. These two subsets are then pretended to be a "training" and a "testing" dataset. In the context of cross-validation, the second set V is called a *validation* set. Unfortunately there is no special word for the subset T. It is also called "training set", which is confusing because it is a subset of the originally given complete training set.

A bit more formally, let the flexibility axis in Figure 6 be parametrized by some appropriate flexibility measure r, where small r means "strong regularization", that is, "go left" on the flexibility axis. Let the range of r be $r = 1, \ldots, L$.

For each setting of flexibility r, the data in T is used to train an optimal model $h_{\text{opt}\,r}$. The test generalization performance on "new" data is then tested

on the validation set. It is determined which model $h_{\text{opt } r^*}$ performs best on the validation data. Its flexibility r^* is then taken to be the flexibility that marks the "sweet spot" indicated by the green bar in Figure 6. After this screening of degrees fo flexibility for the best test performance, a model within the found optimal regularization strength r^* is then finally trained on the original complete training data set $S = T \cup V$.

This whole procedure is called *cross-validation*. Notice that nothing has been said so far about how to split S into T and V.

A clever way to answer this question is to split S into many subsets S_j of roughly equal size ($j = 1, ..., K$). Then, for each flexibility value r carry out K complete screening runs via cross validation, where in the j-th run the subset S_j is withheld as a validation set, and the remaining $K-1$ sets joined together make for a training set. After these K runs, average the validation errors obtained in the K runs. This average is then taken as an estimate for the risk of learning a model with flexibility r. This is called *K-fold cross-validation*. Here is the procedure in detail:

Given: A set $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$ of training data, and a loss function L.

Also given: Some method which allows one to steer the model learning through different degrees r of flexibility. The weakest regularization should be weak enough to allow overfitting.

Step 1. Split the training data into K disjoint subsets $S_j = (\mathbf{u}_i, \mathbf{y}_i)_{i \in I_j}$ of roughly equal size $N' = N/K$.

Step 2. Repeat for $r = 1, \ldots, L$:

Step 2.1 Repeat for $j = 1, \ldots, K$:

Step 2.2.1 Designate S_j as validation set V_j and the union of the other $S_{j'}$ as training set T_j.

Step 2.2.1 Compute the model with minimal training error on data input T_j

$$h_{\mathrm{opt}\,m\,j} = \operatorname*{argmin}_{h \in \mathcal{H}_r} 1/|T_j| \sum_{(\mathbf{u}_i, \mathbf{y}_i) \in T_j} L(h(\mathbf{u}_i), \mathbf{y}_i),$$

where $\mathcal{H}_r$ is the model search space for the regularization strength r.

Step 2.2.2 Test $h_{\mathrm{opt}\,r\,j}$ on the current validation set V_j by computing the validation risk

$$R^{\mathrm{val}}_{r\,j} = 1/|V_j| \sum_{(\mathbf{u}_i, \mathbf{y}_i) \in V_j} L(h_{\mathrm{opt}\,m\,j}(\mathbf{u}_i), \mathbf{y}_i).$$

Step 2.2 Average the K validation risks $R^{\mathrm{val}}_{r\,j}$ obtained from the "folds" carried out for this r, obtaining

$$R^{\mathrm{val}}_r = 1/K \sum_{j=1,\ldots,K} R^{\mathrm{val}}_{r\,j}.$$

Step 3. Find the optimal class by looking for that r which minimizes the averaged validation risk:

$$r_{\mathrm{opt}} = \operatorname*{argmin}_r R^{\mathrm{val}}_r.$$

Step 4. Compute $h_{r_{\mathrm{opt}}}$ using the complete original training data set:

$$h_{r_{\mathrm{opt}}} = \operatorname*{argmin}_{h \in \mathcal{H}_{r_{\mathrm{opt}}}} 1/N \sum_{i=1,\ldots,N} L(h(\mathbf{u}_i), \mathbf{y}_i).$$

This procedure contains two nested loops and looks expensive. For economy, one starts with the low-end r and increases it stepwise, assessing the generaliza-

tion quality through cross-validation for each regularization strength r, until the validation risk starts to rise. The strength r_{opt} reached at that point is likely to be about the right one.

The best assessment of the optimal class is achieved when the original training data set is split into singleton subsets — that is, each S_j contains just a single training example. This is called *leave-one-out cross-validation*. It looks like a horribly expensive procedure, but yet it may be advisable when one has only a small training data set, which incurs a particularly large danger of ending up with poorly generalizing models when a wrong model flexibility was used.

K-fold cross validation is widely used. It is a factual standard procedure in supervised learning tasks when the computational cost of learning a model is affordable.

I should also mention that cross-validation is not only used for finding the right degree of regularization, but can similarly be used for tuning *hyperparameters* of a learning procedure $\mathcal{A}$. The term "hyperparameter" is generally used for all kinds of "knobs to play around with" when tuning a complex learning algorithm. Modern deep learning algorithms have frightfully many tuning options.

2 Feedforward networks in machine learning

After having lubricated our machine learning gearboxes, we are ready for taking off into the lands of neural networks (NNs). In this section I will present those NNs that are most commonly used in practical machine learning applications: feedforward neural networks trained by the backpropagation algorithm. I will present them in historical order, starting with the famous ancestor of them all, the *Perceptron*.

2.1 The Perceptron

The Perceptron was developed by Frank Rosenblatt, an American psychologist / neuro-biologist / electrical engineer, in the years around 1960 (for instance, Rosenblatt (1958)). It was a computational neural architecture inspired by what was known about the human visual processing system at that time, and its workings were demonstrated on visual character classification tasks. Figure 7 gives a diagram of the Perceptron's architecture.

In plain English, the Perceptron processes its input as follows.

- Typical inputs are clean black-and-white images of letters, sensed by an array of photocells (The "retina" in Figure 7 is this array of photocells. Figure 8 A shows one of Rosenblatt's analog hardware realizations of the Perceptron, the *Mark 1*). This gives analog measurement values per "pixel" (= photocell) — the perceptron is non-digital.

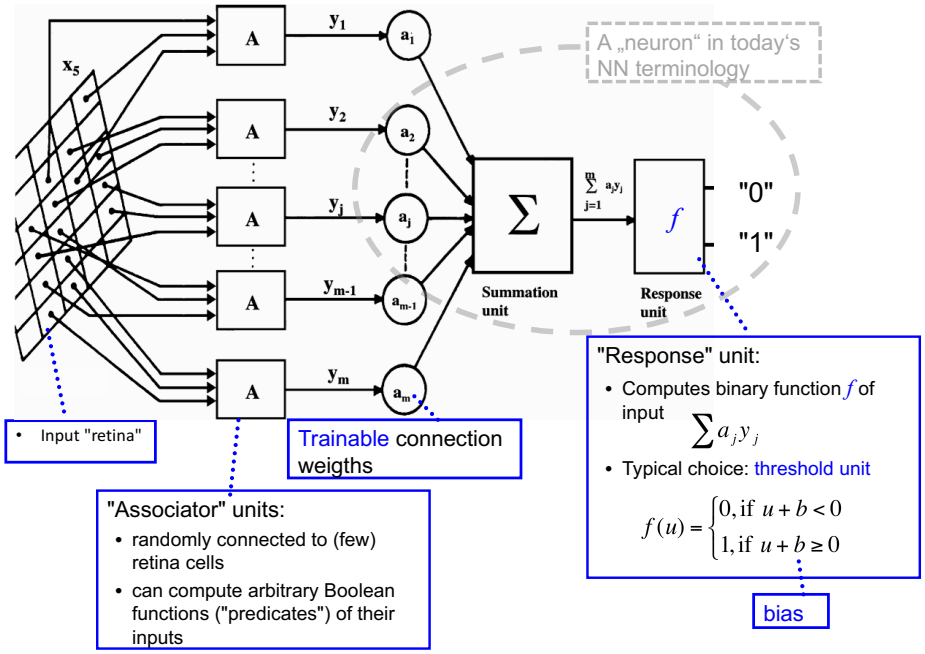

Figure 7: Architecture of the Perceptron. Note that Rosenblatt designed and built many variants of the Perceptron; this is just one of them. The "summation unit" and the "response unit" (and possibly the incoming synaptic connections too) taken together are a "neuron" in today's NN terminology. Graphics taken from Kanal (2001).

- In a next layer of processing, a number of "associator" neurons each pick up a random selection of the retina values and perform some (arbitrary) Boolean function on them. In modern machine learning language, these associator unites compute local random features from the retinal image.

- The activation values (denoted y_j in Figure 7) of the associator units are propagated forward into a single "summation unit", where they are summed up — but not before they have been scaled by "synaptic" weights w_j. Thus, the summation unit together with these weights yields a linear combination of the associator activation values.

- Finally, the output value of the summation unit is passed through a "response unit", which delivers the ultimate output of the Perceptron by applying a thresholding operation which results in a 0 or 1 value.

- The entire processing pipeline of the Perceptron thus transforms retinal sensation patterns into a binary output — which is the format of an image

classification system. The task on which the Perceptron was probed was a binary classification task, for instance letters "A" versus letters "C".

- The Perceptron was revolutionary in that it was a *learning* system. In order to to solve the requested image classification task, the synaptic weights must be the right ones. It is not difficult to find a mathematical formula which computes a set of well-working synaptic weights if such a set exists at all. Rozenblatt's perceptron however "learnt" the right synaptic weights in an incremental, iterative training process. In this process, the network was presented with a series of what today we would call "training patterns". When the network's output was correct, the synaptic weights remained unchanged; if it was wrong, they were adapted by an incremental weight adaptation rule, the *Perceptron learning rule*. Rosenblatt could prove that this learning rule would converge to a perfectly performing network if a perfectly working weight vector exists at all in the first place.

The Perceptron (in its Mark 1 realization) was a remarkable mix of analog signal processing hardware, electrical engineering, biological inspiration, and elementary mathematics. It featured many innovations that today are commonplace for neural networks in machine learning (random nonlinear features, computing outputs by passing the results of a linear combination through a nonlinearity, using image classification as a benchmark demonstration).

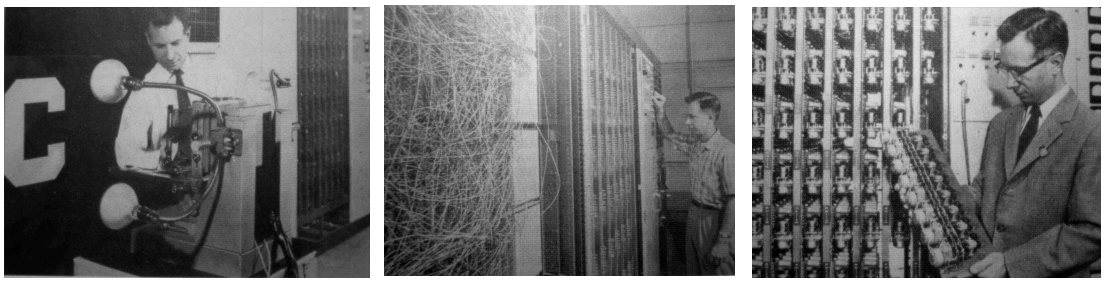

Figure 8: The Mark 1 perceptron — an analog hardware neural network. A: Pattern input: brightly lit B/W images of printed characters sensed by an array of 20 x 20 photocells. B: Wiring — before the invention of printed circuit boards. C: Adaptive weights realized by motor-driven potentiometers (all images from Bishop (2006), Chapter 4.1)

In plain maths, if one strips off all biologism and does not include the random Boolean operations of the associator units in the learning story (because they remain fixed and are not trainable), the Perceptron simply learns a function

$$\begin{aligned} \hat{f}: \quad & \mathbb{R}^K \to \{0,1\} \\ & (x_1,\ldots,x_K)' \mapsto \sigma(\textstyle\sum_{j=1}^{n} w_j\, x_j), \end{aligned} \tag{11}$$

where the argument are vectors $\mathbf{u} = (x_1,\ldots,x_K)'$ containing the n activations of the associator units, and $\sigma : \mathbb{R} \to \mathbb{R}, \sigma(x) = 1$ if $x \geq 0$ else $\sigma(x) = 0$ is a binary thresholding function with threshold at zero. The variables x_i are fixed (not trainable) Boolean functions of the raw (image) inputs $\mathbf{u}$.

Note that different from the schema shown in Figure 7 there is no "bias" b inside the thresholding function. It is common practice in today's machine learning to omit this bias and instead fix one of the inputs $x_1,\ldots,x_K$ to always have a value of 1 in all input patterns. This is mathematically equivalent to the bias b inside the threshold function and leads to a more compact notation.

The Perceptron learning rule goes like this:

Given: A training pattern set $(\mathbf{u}_i, y_i)_{i=1,\ldots,N}$ with $\mathbf{u}_i \in \mathbb{R}^K, y_i \in \{0,1\}$.

Model initialization: set the weight vector $\mathbf{w} = (w_1,\ldots,w_K)$ to some arbitrary initial value, for instance to some random vector or the all-zero vector.

Cycle through training data and adapt weights: Present the training patterns to the Perceptron one after the other, starting again from the beginning when the last pattern has been used. At each presentation of a pattern $\mathbf{u}_i$,

- compute the current output $\hat{y}_i = \sigma(\mathbf{w}'\,\mathbf{u})$ of the Perceptron,
- if $\hat{y}_i = y_i$, do not change the weight vector.
- if $\hat{y}_i \neq y_i$, change the weight vector by $\mathbf{w} \leftarrow \mathbf{w} + (y_i - \hat{y}_i)\,\mathbf{u}_i$.

Stop when in one full cycle through all training examples no weight changes were triggered (that is, all training examples are correctly classified).

Today, even a beginner in machine learning would immediately criticize that this method only minimizes the training error, inviting overfitting; and that this seems a very weak and restricted learning algorithm, because it is only applicable to 2-class classification problems, and the computation (11) does not look very powerful.

But, at the time when Rosenblatt presented the Perceptron, machine learning as a field did not exist at all (in retrospect, the Perceptron can be seen as one of the starters), it bundled an entire collection of 100% new and ingenious ideas, and it looked like a little brain. Here is a sniplet from the New York Times ("New Navy Device Learns by Doing", NYT July 8, 1958), after a press conference held by Rosenblatt at the US Office of Naval Research on July 7, 1958 (cited after Olazaran (1996)):

"The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. Later perceptrons will be able to recognize people and call out their names and instantly translate speech [...], it was predicted."

Before you feel immensely superior to the 1958 authors and readership of the New York Times, think about what you, today, think about the powers of deep learning. Brain-like, eh? ... let's wait a little ...

The hype about the Perceptron did not last long. In a book titled "Perceptrons", famous AI pioneers Minsky and Papert (1969) pointed out the obvious: with an architecture that boils down to a simple, thresholded linear combination of input values, one can only learn to classify patterns that are *linearly separable*. That is, in the input space $\mathbb{R}^K$ where patterns come from, there must exist an $n-1$ dimensional linear hyperplane, dividing $\mathbb{R}^K$ into two half-spaces, such that all patterns of class 0 lie on one side of the hyperplane and all class-1 patterns on the other side. Specifically, Minsky and Papert shone a flashlight on the embarassing fact that Perceptrons cannot learn the XOR function, because it is not linearly separable (see Figure 9). As a consequence of this simple insight ("perceptrons can't even learn the simplest Boolean functions"), neural networks become disreputable as an approach to realize machine intelligence, and research on neural networks came to a dead halt and remained frozen — until things happened in the early 1980's that will be related in the next section.

u_1	u_2	XOR(u_1, u_2)
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

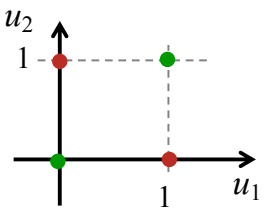

Figure 9: The XOR stopper for early neural network research. The training dataset consists of merely 4 examples $(\mathbf{u}_1, y_1) = ((0,0),0), (\mathbf{u}_2, y_2) = ((0,1),1), (\mathbf{u}_3, y_3) = ((1,0),1), (\mathbf{u}_4, y_4) = ((1,1),0)$, but classifying the $(0,0),(1,1)$ patterns in one class and the $(0,1),(1,0)$ patterns into another is impossible for a Perceptron, because there is no way to draw a line in the pattern space $\mathbb{R}^2$ which separates the two classes (right panel).

2.2 Multi-layer perceptrons

After Minsky's and Papert's shattering blow, neural networks became disreputable in AI (the field "machine learning" did not exist in the year 1969). But research

on neural networks did not come to a complete halt. A little outside AI, at the fringes between neuroscience, signal processing and theoretical physics, researchers secretely dared to continue thinking about neural network models... And in the early 1980's there was a veritable public eruption of new approaches, today often known by the names of their inventors, for instance

Hopfield networks, introduced by theoretical physicist John Hopfield (Hopfield, 1982), a model of neural networks that can explain how memory traces (of images, for instance) can be stored, retrieved, and noise-repaired in a suitable designed neural network. Hopfield networks are a standard basic model of neural long-term memory to the day. I will devote one session of this course to them.

Kohonen networks, also known as *self-organizing (feature) maps (SOMs)*, introduced by engineer Teuvo Kohonen (Kohonen, 1982), are a neural architecture for learning low-dimensional representations of high-dimensional input patterns. The principle of self-organizing maps (or something very similar) is apparently active in the way how the mammalian visual cortex learns to represent geometric features in retinal input patterns. I will not treat Kohonen maps in this course (sadly not enough time), but I devoted Section 4.7 in my machine learning lecture notes to them.

The Boltzmann machine, introduced by cognitive neuroscientists Geoffrey Hinton and Terrence Sejnowski (Hinton and Sejnowski, 1983) (subsequently much more clearly explained in Ackley et al. (1985)), is a universal, unsupervised neural learning system which can, in principle, learn every kind of distribution from training data. It is mathematically very transparent but computationally extraordinarily expensive and thus of almost no practical use. Besides producing many other groundbraking innovations to neural network science, Hinton continued to think about the Boltzmann machine and finally found a way to cut down the computational cost in the *Restricted Boltzmann Machine (RBM)*, presented to a wider scientific community in an article in Science (Hinton and Salakuthdinov, 2006). In passing, in this article it is mentioned how RBMs can be used to initialize the training of feedforward neural networks for pattern recognition, — in retrospect, these inconspicuous remarks turned out to be the starting shot for deep learning. I will devote one session to Boltzmann machines in this course.

Besides these three neural network models, many others were conceived in the 1970 / early 1980 years, often inspired by physics, neuroscience, or psychology (not inspired by AI, since that field was too successful in its own non-neural ways in those days and too proud to reconsider the disreputed Perceptron). These innovative neural network models helped to understand principles of biological and cognitive neural information processing.

But, none of them was really practically useful.

The tides turned in the year 1986, when a two-volume collection of neural network articles was published, "Parallel Distributed Processing" (Rumelhart and McClelland, 1986), of which the first volume soon became known (and still is) as *the PDP book*. This volume contained a number of well-written, detailed introductions to the innovative NN approaches that had been modestly but steadily sprouting in the AI shadows. Suddenly, neural networks were back on stage with.

The immense and lasting impact of the PDP book is largely due to only one of its chapters. In this chapter, Rumelhart et al. (1986) give a transparent introduction how neural networks could be designed and trained for pattern classification tasks in ways that are much more powerful than what could be achieved with the Perceptron. The dreaded XOR learning task suddenly was just a piece of cake. The added powers of the *multi-layer Perceptrons (MLPs)*, as the neural networks described in this chapter became generally named, arose from two innovations. First, while the classical Perceptron only has a single "summation-response" unit, in MLPs many of such units are coupled together in parallel and serial connectivity patterns. Second, the original Perceptron learning rule (which can't be applied to MLPs) was replaced by the *error backpropagation* (or simply "backprop") algorithm, which made it possible to iteratively adapt the synaptic weights in an MLP in order to minimize a given loss function — for instance, in order to classify images. It must be mentioned that the backpropagation algorithm was not freshly invented by the authors of the PDP book chapter, but had been described by Paul Werbos much earlier in his 1974 PhD thesis — which did not attract attention and remained ignored; and in other fields of engineering, similar algorithms had been discovered even earlier. Schmidhuber (2015), Section 5.5, provides a historical overview.

At any rate, after 1986 neural networks took off again. And again, there was a hype that soon withered (I will relate this in the lecture). But the decline of that new hype wasn't as dramatic as what happened to Perceptrons. In the two decades from 1986 to 2006, MLPs remained a respected citizen of the world of AI and machine learning (this field now had come into being). But not more than that — other methods, like support vector machines or Bayesian networks, were more prominent in machine learning. MLPs worked well as far as it went, but that wasn't too far after all: roughly speaking, they worked fairly well in not too nonlinear signal processing and pattern recognition tasks. Their economical impact was not striking.

After this short historical review (I find history fascinating), I will describe MLPs in sober detail. Much of the material in this section is taken from my machine learning lecture notes.

MLPs are used for the supervised learning of vectorial input-output tasks, based on training samples $S = (\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$, where $\mathbf{u} \in \mathbb{R}^K, \mathbf{y} \in \mathbb{R}^M$ are drawn from a joint distribution $P_{U,Y}$.

The MLP is trained to produce outputs $\mathbf{y} \in \mathbb{R}^M$ upon inputs $\mathbf{u} \in \mathbb{R}^K$ in a way that this input-output mapping is similar to the relationships $\mathbf{u}_i \mapsto \mathbf{y}_i$ found in

the training data. Similarity is measured by a suitable loss function.

An MLP is a neural network structured equipped with n *input units* and m *output units*. An n-dimensional input pattern $\mathbf{u}$ can be sent to the input units, then the MLP does some interesting internal processing, at the end of which the m-dimensional result vector of the computation can be read from the m output units. An MLP $\mathcal{N}$ with n input units and m output units thus instantiates a function $\mathcal{N} : \mathbb{R}^K \to \mathbb{R}^M$.

Just like how it worked with the Perceptron, an MLP is defined by its *structure* (often called *architecture*) and by the values of its "synaptic connection" *weights*. The architecture is invented and fixed by the human experimenter / data engineer in a way that should be matched to the given learning task. The architecture is (mostly) not changed during the subsequent training. MLPs have a more complex architecture than the Perceptron. A given neuron in an MLP is typically receiving input from, and sending its output to, many other neurons along synaptic connections. Like in the Perceptron, each such connection is characterized by a weight. These weights are iteratively and incrementally adapted in the training process until the network function $\mathcal{N} : \mathbb{R}^K \to \mathbb{R}^M$ comes close to what one desired.

It is customary in machine learning to lump all trainable parameters of a machine learning model together in one *parameter vector*, which is standardly denoted by θ. For a neural network, θ is thus the vector of all the trainable synaptic connection weights. Since the network function is determined by θ, one also writes $\mathcal{N}_\theta : \mathbb{R}^K \to \mathbb{R}^M$ if one wishes to emphasize the dependance of $\mathcal{N}$'s functionality on its weights.

The learning task is defined by a loss function $L : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}^{\geq 0}$. A convenient and sometimes adequate choice for L is the quadratic loss $L(\mathcal{N}(\mathbf{u}), \mathbf{y}) = \|\mathcal{N}(\mathbf{u}) - \mathbf{y}\|^2$, but other loss functions are also widely used. Chapter 6.2 in the deep learning bible Goodfellow et al. (2016) gives an introduction to the theory of which loss functions should be used in which task settings.

Given the loss function, the goal of training an MLP is to find a weight vector θ_{opt} which minimizes the loss, that is

$$\theta_{\text{opt}} = \underset{\theta \in \Theta}{\operatorname{argmin}} \, E[L(\mathcal{N}_\theta(U), Y)], \tag{12}$$

where U, Y are the input / output data generating random variables. However, this loss cannot be directly minimized because the distribution of U, Y is not known. Therefore, as we saw in Section 1, instead one tries to find a practical algorithm for minimizing the empirical loss (training error), that is

$$\theta_{\text{opt}} = \underset{\theta \in \Theta}{\operatorname{argmin}} \, \frac{1}{N} \sum_{i=1}^{N} L(\mathcal{N}_\theta(\mathbf{u}_i), \mathbf{y}_i), \tag{13}$$

where Θ is a set of candidate weight vectors. This minimization will become a subtask embedded in procedures which take care of preventing over- or underfitting, like cross-validation. For MLPs, however, finding practically working algorithms

for "merely" minimizing the training error is challenging — as we will learn to appreciate.

MLPs are function approximators. "Function approximation" sounds dry and technical, but many kinds of learning problems can be framed as function approximation learning. Here are some examples:

Pattern recognition: inputs $\mathbf{u}$ are vectorized representations of any kind of "patterns", for example images, soundfiles, stock market time series. Outputs $\mathbf{y}$ are hypothesis vectors of the classes that are to be recognized.

Time series prediction: inputs are vector encodings of a past history of a temporal process, outputs are vector encodings of future observations of the process.

Denoising, restoration and pattern completion: inputs are patterns that are corrupted by noise or other distortions, outputs are cleaned-up or repaired or completed versions of the same patterns.

Data compression: Inputs are high-dimensional patterns, outputs are low-dimensional encodings which can be restored to the original patterns using a decoding MLP. The encoding and decoding MLPs are trained together.

Process control: In control tasks the objective is to send *control inputs* to a technological system (called "plant" in control engineering) such that the system performs in a desired way. The algorithm which computes the control inputs is called a "controller". Control tasks range in difficulty from almost trivial (like controlling a heater valve such that the room temperature is steered to a desired value) to almost impossible (like operating hundreds of valves and heaters and coolers and whatnots in a chemical factory such that the chemical production process is regulated to optimal quality and yield). The MLP instantiates the controller. Its inputs are settings for the desired plant behavior, plus optionally observation data from the current plant performance. The outputs are the control inputs which are sent to the plant.

This list should convince you that "function approximation" is a worthwhile topic indeed, and spending effort on learning how to properly handle MLPs is a good personal investment for any engineer or data analyst.

2.2.1 MLP structure

Figure 10 gives a schematic of the architecture of an MLP. It consists of several *layers* of units. Layers are numbered $0, \ldots, k$, where layer 0 is comprised of the input units and layer k of the output units. The number of units in layer κ is L^κ. The units of two successive layers are connected in an all-to-all fashion by synaptic links (arrows in Figure 10). The link from unit j in layer $\kappa - 1$ to unit i in layer κ has a *weight* $w_{ij}^\kappa \in \mathbb{R}$. The layer 0 is the *input layer* and the layer k is

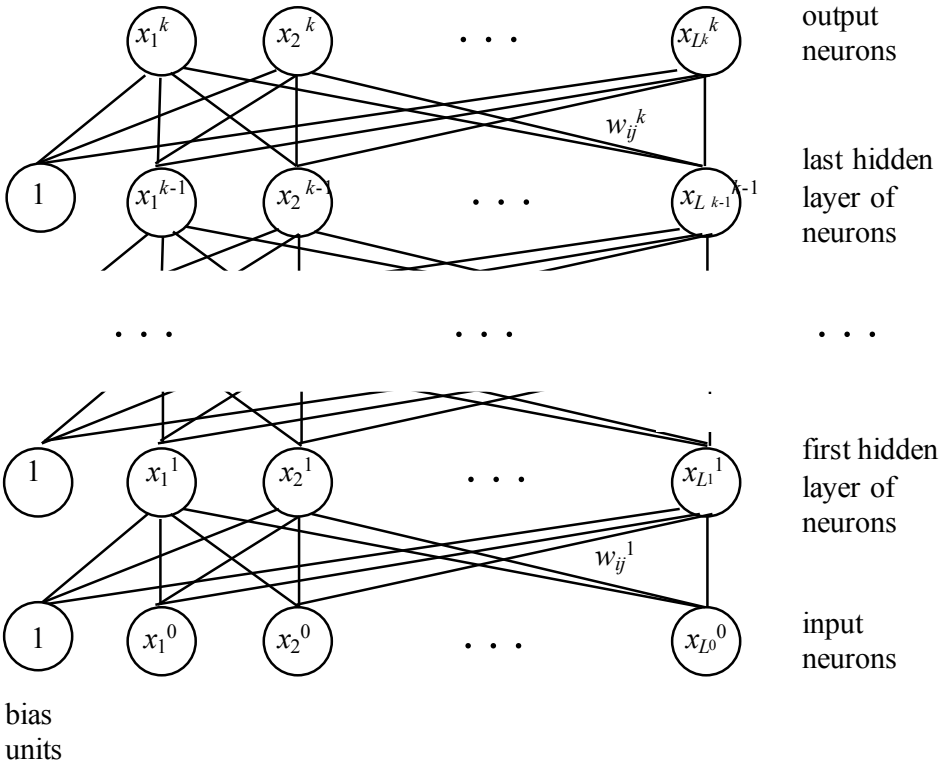

Figure 10: Schema of an MLP with $k-1$ hidden layers of neurons.

the *output layer*. The intermediate layers are called *hidden* layers. When an MLP is used for a computation, the i-th unit in layer κ will have an *activation* $x_i^\kappa \in \mathbb{R}$.

From a mathematical perspective, an MLP $\mathcal{N}$ implements a function $\mathcal{N} : \mathbb{R}^{L^0} \to \mathbb{R}^{L^k}$. Using the MLP and its layered structure, this function $\mathcal{N}(\mathbf{u})$ of an argument $\mathbf{u} \in \mathbb{R}^{L_0}$ is computed by a sequence of transformations as follows:

1. The activations x_j^0 of the input layer are set to the component values of the L^0-dimensional input vector $\mathbf{u}$.

2. For $\kappa < k$, assume that the activations $x_j^{\kappa-1}$ of units in layer $\kappa - 1$ have already been computed (or have been externally set to the input values, in the case of $\kappa - 1 = 0$). Then the activation x_i^κ is computed from the formula

$$x_i^\kappa = \sigma\left(\sum_{j=1}^{L^{\kappa-1}} w_{ij}^\kappa \, x_j^{\kappa-1} + w_{i0}^\kappa\right). \tag{14}$$

That is, x_i^κ is obtained from linearly combining the activations of the lower layer with combination weights w_{ij}^κ, then adding the *bias* $w_{i0}^\kappa \in \mathbb{R}$, then wrapping the obtained sum with the *activation function* σ. The activation

function is a nonlinear, "S-shaped" function which I explain in more detail below. It is customary to interpret the bias w_{i0}^{κ} as the weight of a synaptic link from a special *bias unit* in layer $\kappa - 1$ which always has a constant activation of 1 (as shown in Figure 10).

Equation 14 can be more conveniently written in matrix form. Let $\mathbf{x}^{\kappa} = (x_1^{\kappa}, \ldots, x_{L^k}^{\kappa})'$ be the activation vector in layer κ, let $\mathbf{b}^{\kappa} = (w_{10}^{\kappa}, \ldots, w_{L^{\kappa}0}^{\kappa})'$ be the vector of bias weights, and let $\mathbf{W}^{\kappa} = (w_{ij}^{\kappa})_{i=1,\ldots,L^{\kappa};\, j=1,\ldots,L^{\kappa-1}}$ be the connection weight matrix for links between layers $\kappa - 1$ and κ. Then (14) becomes

$$\mathbf{x}^{\kappa} = \sigma\left(\mathbf{W}^{\kappa}\,\mathbf{x}^{\kappa-1} + \mathbf{b}^{\kappa}\right), \tag{15}$$

where the activation function σ is applied component-wise to the activation vector.

3. The L^k-dimensional activation vector $\mathbf{y}$ of the output layer is computed from the activations of the pre-output layer $\kappa = k - 1$ in various ways, depending on the task setting (compare Chapter 6.2 in Goodfellow et al. (2016)). For simplicity we will consider the case of *linear* output units,

$$\mathbf{y} = \mathbf{x}^{k} = \mathbf{W}^{k}\,\mathbf{x}^{k-1} + \mathbf{b}^{k}, \tag{16}$$

that is, in the same way as it was done in the other layers except that no activation function is applied. The output activation vector $\mathbf{y}$ is the result $\mathbf{y} = \mathcal{N}(\mathbf{u})$.

The activation function σ is traditionally either the hyperbolic tangent (tanh) function or the *logistic sigmoid* given by $\sigma(a) = 1/(1 + \exp(-a))$. Figure 11 gives plots of these two S-shaped functions. Functions of such shape are often called *sigmoids*. There are two grand reasons for applying sigmoids:

- Historically, neural networks were conceived as abstractions of biological neural systems. The electrical activation of a biological neuron is bounded. Applying the tanh bounds the activations of MLP "neurons" to the interval $[-1, 1]$ and the logistic sigmoid to $[0, 1]$. This can be regarded as an abstract model of a biological property.

- Sigmoids introduce nonlinearity into the function $\mathcal{N}_{\theta}$. Without these sigmoids, $\mathcal{N}_{\theta}$ would boil down to a cascade of affine linear transformations, hence in total would be merely an affine linear function. No nonlinear function could be learnt by such a linear MLP.

In the area of deep learning a drastically simplified "sigmoid" is often used, the *rectifier* function defined by $r(a) = 0$ for $a < 0$ and $r(a) = a$ for $a \geq 0$. The rectifier has somewhat less pleasing mathematical properties compared to the classical sigmoids but can be computed much more cheaply. This is of great value in deep learning scenarios where the neural networks and the training samples both are

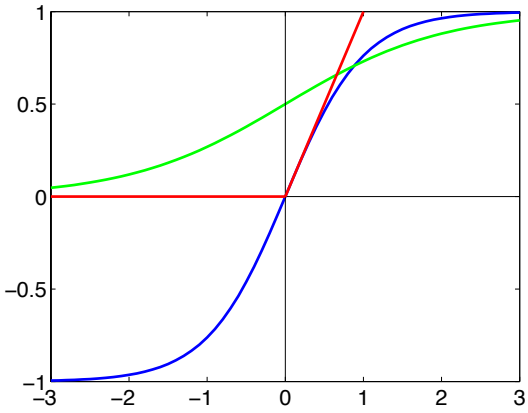

Figure 11: The tanh (blue), the logistic sigmoid (green), and the rectifier function (red).

often very large and the training process requires very many evaluations of the sigmoid.

In intuitive terms, the operation of an MLP can be summarized as follows. After an input vector $\mathbf{u}$ is written into the input units, a "wave of activation" sweeps forward through the layers of the network. The activation vector $\mathbf{x}^\kappa$ in each layer κ is directly triggered by the activations $\mathbf{x}^{\kappa-1}$ according to (15). The data transformation from $\mathbf{x}^{\kappa-1}$ to $\mathbf{x}^\kappa$ is a relatively "mild" one: just an affine linear map $\mathbf{W}^\kappa \mathbf{x}^{\kappa-1} + \mathbf{b}^\kappa$ followed by a wrapping with the gentle sigmoid σ. But when several such mild transformations are applied in sequence, very complex "foldings" of the input vector $\mathbf{u}$ can be effected. Figure 12 gives a visual impression of what a sequence of mild transformations can do.

MLPs are a member of a larger class of neural networks, the *feedforward neural networks* (FFNs). FFNs all have an input layer, an output layer, and in between a directed neural connection network which lets the input activation vector spread forward through the network, undergoing all sorts of transformations, until it reaches the output layer. Importantly, there is never a synaptic connection inside the network that would feed some intermediate activation back into earlier layers. Mathematically speaking, the connection graph of an FNN is *acyclical*. All FFNs are therefore representing a *function* from input vectors to output vectors. Within the class of FNNs, the MLPs are the most simple ones, with all layers having the same basic structure. For a given task this generic, simple MLP architecture may be suboptimal (it very likely is!). Much more sophisticated feedforward architectures have been developed for specific tasks. At the end of this section I will highlight one of these, the convolutional neural networks whose architecture is optimized for image recognition.

It is also possible to design neural networks which host cyclical synaptic connec-

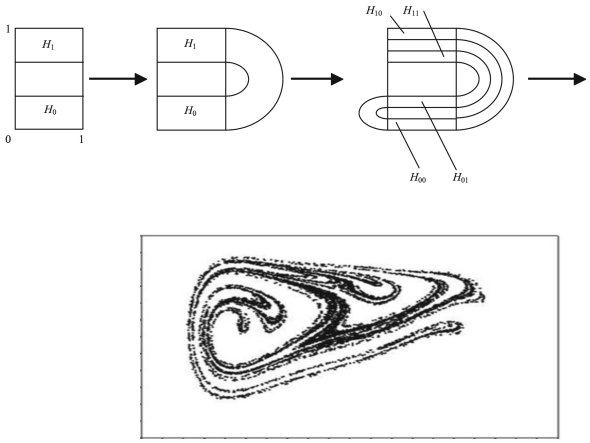

Figure 12: Illustrating the power of iterating a simple transformation. The *baker transformation* (also known as *horseshoe transformation*) takes a 2-dimensional rectangle, stretches it and folds it back onto itself. The bottom right diagram visualizes a set that is obtained after numerous baker transformations (plus some mild nonlinear distortion). — Diagrams on the right taken from Savi (2016).

tion pathways. In such *recurrent neural networks* (RNNs), a neuron's activation can ultimately feed back on the same neuron. Mathematically speaking, RNNs are not representing functions but *dynamical systems*. This is an upper league of mathematics, far more complex, fascinating, and powerful than just functions. Biological brains are always recurrent, and intelligent reasoning has memory — which means it feeds back on itself recurrently. Most of this course will be about RNNs! Only the two initial Sections 2.1 and 2.2 will be about FNNs.

2.2.2 Universal approximation and the powers of being deep

One reason for the popularity of MLPs is that they can approximate arbitrary functions $f : \mathbb{R}^K \to \mathbb{R}^M$. Numerous results on the approximation qualities of MLPs have been published in the early 1990-ies. Such theorems have the following general format:

Theorem (schematic). *Let $\mathcal{F}$ be a certain class of functions $f : \mathbb{R}^K \to \mathbb{R}^M$. Then for any $f \in \mathcal{F}$ and any $\varepsilon > 0$ there exists an multilayer perceptron $\mathcal{N}$ with one hidden layer such that $\|f - \mathcal{N}\| < \varepsilon$.*

Such theorems differ with respect to the classes $\mathcal{F}$ of functions that are approximated and with respect to the norms $\|\cdot\|$ that measure the mismatch between two functions. All practically relevant functions belong to classes that are covered by such approximation theorems. In a summary fashion it is claimed that MLPs are *universal function approximators*. Again, don't let yourself be misled by the dryness of the word "function approximator". Concretely the universal function

approximation property of MLPs would spell out, for example, to the (proven) statement that any task of classifying pictures can be solved to any degree of perfection by a suitable MLP.

The proofs for such theorems are typically constructive: for some target function f and tolerance ε they explicitly construct an MLP $\mathcal{N}$ such that $\|f - \mathcal{N}\| < \varepsilon$. However, these constructions have little practical value because the constructed MLPs $\mathcal{N}$ are far too large for any practical implementation. You can find more details concerning such approximation theorems and related results in my legacy ML lecture notes `http://minds.jacobs-university.de/uploads/teaching/lectureNotes/LN_ML_Fall11.pdf`, Section 8.1.

Even when the function f that one wants to train into an MLP is very complex (highly nonlinear and with many "folds"), it can be in principle approximated with 1-hidden-layer MLPs. However, when one employs MLPs that have *many* hidden layers, the required overall size of the MLP (quantified by total number of weights) is dramatically reduced (Bengio and LeCun, 2007). Even for supercomplex target functions f (like photographic image caption generation), MLPs of feasible size exist when enough layers are used. It is not uncommon for professionally designed and trained deep networks to feature in the order of 20 layers. This is the basic insight and motivation to consider *deep* networks, which is just another word for "many hidden layers". Unfortunately it is not at all easy to train deep networks. Traditional learning algorithms had made non-deep ("shallow") MLPs popular since the 1980-ies. But these shallow MLPs could only cope with relatively well-behaved and simple learning tasks. Attempts to scale up to larger numbers of hidden layers and more complex data sets largely failed, due to numerical instabilities, too slow convergence, or poor model quality. Since about 2006 an accumulation of clever "tricks of the trade" plus the availability of powerful (GPU-based) yet affordable computing hardware has overcome these hurdles.

2.2.3 Training an MLP with the backpropagation algorithm

In this section I give an overview of the main steps in training a non-deep MLP for a mildly nonlinear task — tasks that can be solved with one or two hidden layers. When it comes to unleash deep networks on gigantic training data sets for hypercomplex tasks, the basic recipes given in this section will not suffice. You would need to train yourself first in the art of deep learning. However, what you can learn in this section is necessary background for surviving in the wilderness of deep learning.

General outline of an MLP training project Starting from a task specification and access to training data (given to you by your boss or a paying customer, with the task requirements likely spelled out in rather informal terms, like "please predict the load of our internet servers for the next 3 minutes"), a basic training procedure for a elementary MLP $\mathcal{N}$ goes like follows.

1. **Get a clear idea of the formal nature of your learning task.** Do you want a model output that is a probability vector? or a binary decision? or a maximally precise transformation of the input? how should "precision" best be measured? and so forth. Only proceed with using MLPs if they are really looking like a suitable model class for your problem.

2. **Decide on a loss function.** Go for a simple quadratic loss if you want a quick baseline solution but be prepared to invest in other loss functions if you have enough time and knowledge (read Chapter 6.2 in Goodfellow et al. (2016)).

3. **Decide on a regularization method.** For elementary MLP training, suitable candidates are adding an L_2 norm regularizer to your loss function; extending the size of the training data set by adding noisy / distorted exemplars; varying the network size; or early stopping (= stop gradient descent training a overfitting-enabled MLP when the validation error starts increasing — requires continual validation during gradient descent). Demyanov (2015) is a solid guide for MLP regularization.

4. **Think of a suitable vector encoding of the available training data,** including dimension reduction if that seems advisable (it *is* advisable in all situations where the ratio *raw data dimension / size of training data set* is high).

5. **Fix an MLP architecture.** Decide how many hidden layers the MLP shall have, how many units each layer shall have, what kind of sigmoid is used and what kind of output function and loss function. The structure should be rich enough that data overfitting becomes possible and your regularization method can kick in.

6. **Set up a cross-validation scheme** in order to optimize the generalization performance of your MLPs. Recommended: implement a semi-automated k-fold cross-validation scheme which is built around two subroutines, (1) "train an MLP of certain structure for minimal training error on given training data", (2) "test MLP on validation data". Note: for large MLPs and/or large training data sets, something like K-fold cross-validation will likely be too expensive. In deep learning, where a single learning run may take two weeks, one uses the cheapest possible cross-validation scheme, called "early stopping": while the iterations of the model optimization are running, assess the generalization qualities of the model in its current learning state on a validation data set. If the validation error starts to rise again after having first dropped (see THE blue curve in Figure 6), stop the model adaptation.

7. **Implement the training and testing subroutines.** The training will be done with a suitable version of the *error backpropagation* algorithm, which will require you to meddle with some global control parameters (like learning rate, stopping criterion, initialization scheme, and more).

8. Do the job. Enjoy the powers, and marvel at the wickedness, of MLPs.

You see that "neural network training" is a multi-faceted thing and requires you to consider all the issues that *always* jump at you in supervised machine learning. It will *not* miraculously give good results just because it's "neural networks inside". The actual "learning" part, namely solving the optimization task (12), is only a subtask, albeit a conspicuous one because it is done with an algorithm that has risen to fame.

Iterative model optimization: general principle This famous algorithm is, of course, the backpropagation algorithm. It is an algorithm for performing a gradient descent minimization. I will first rehearse the general principles and terminology of model optimization through gradient descent. Here is the set-up:

Given: Training data $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\dots,N}$ for a supervised learning task, as usual, where $\mathbf{u}_i \in \mathbb{R}^K, \mathbf{y}_i \in \mathbb{R}^M$.

Given: A fixed model architecture parametrized by trainable parameters collected in a vector θ. In our situation, the architecture would be an MLP with fixed structure, and θ would be containing all the synaptic connection weights in the MLP $\mathcal{N}_\theta$.

The set of all possible models (with the same architecture, distinguished from each other only by different weights) can be identified with a set Θ of all possible weight vectors θ. Θ gives us the *search space* for a good model. If the chosen architecture has M trainable weights, this search space would be $\Theta = \mathbb{R}^M$.

Given: A loss function $L : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}^{\geq 0}$.

Wanted: An optimal model

$$\theta_{\mathsf{opt}} = \operatorname*{argmin}_{\theta\in\Theta} R^{\mathsf{emp}}(\theta) = \operatorname*{argmin}_{\theta\in\Theta} \frac{1}{N} \sum_{i=1,\dots,N} L(\mathcal{N}_\theta(\mathbf{u}_i), \mathbf{y}_i). \qquad (17)$$

The minimization problem (17) cannot be solved analytically. Instead, one designs an iterative algorithm which produces a sequence of models (= parameter vectors) $\theta^{(0)}, \theta^{(1)}, \theta^{(2)}, \dots$ with decreasing empirical risk $R^{\mathsf{emp}}(\theta^{(0)}) > R^{\mathsf{emp}}(\theta^{(1)}) > \dots$. The model $\theta^{(n+1)}$ is typically computed by an incremental modification of the previous model $\theta^{(n)}$. The first model $\theta^{(0)}$ is a "guess" provided by the experimenter.

The hope is that this series converges to a model $\theta^{(\infty)} = \lim_{n\to\infty} \theta^{(n)}$ whose empirical risk is close to the minimal possible empirical risk.

Machine learning research (and mathematics and optimization theory in general) has found a number of quite different principles to design iterative model adaptation procedures which lead to a decreasing risk sequence. Examples are the

family of *Expectation-Maximization* algorithms (explained in my machine learning lecture notes if you are interested), or iteration schemes that exploit stability conditions for fixed points of a map; or various sorts of general-purpose stochastic search algorithms like *genetic algorithms* or *simulated annealing* (look them up on Wikipedia). Or, finally — *gradient descent* algorithms. That is the kind of iterative model optimization algorithm that is typically used with neural networks (although, as we will see later in this course, by no means the only one, and also most likely not the one used by Mother Nature to make our brains learn). Gradient descent algorithms come in many versions and degrees of sophistication. Here I discuss the plain vanilla kind.

Performance surfaces. *First a quick recap: the graph of a function.* Recall that the graph of a function $f : A \to B$ is the set $\{(a, f(a)) \in A \times B \mid a \in A\}$ of all argument-value pairs of the function. For instance, the graph of the square function $f : \mathbb{R} \to \mathbb{R},\ x \mapsto x^2$ is the familiar parabola curve in $\mathbb{R}^2$.

A performance surface is the graph of a risk function. Depending on the specific situation, the risk function may be the empirical risk, the risk, or some other "cost". I will use the symbol $\mathcal{R}$ to denote a generic risk function of whatever kind.

Performance surfaces are typically discussed with parametric model families where a candidate set of models can be identified with a candidate set Θ of parameter vectors θ. The performance surface then becomes the graph of the function $\mathcal{R} : \Theta \to \mathbb{R}^{\geq 0}$.

Specifically, if we have D-dimensional parameter vectors (that is, $\Theta \subseteq \mathbb{R}^D$), the performance surface is a M-dimensional hypersurface in $\mathbb{R}^{D+1}$.

Other terms are variously used for performance surfaces, for instance *performance landscape* or *error landscape* or *error surface* or *cost landscape* or similar wordings.

Performance surfaces can be objects of stunning complexity. In neural network training (stay tuned ... we *will* come to that! I promise!), they are *tremendously* complex. Figure 13 shows a neural network error landscape randomly picked from the web.

Gradient descent on a performance surface Often a risk function $\mathcal{R} : \Theta \to \mathbb{R}^{\geq 0}$ is differentiable. Then, for every $\theta \in \Theta$ the *gradient* of $\mathcal{R}$ with respect to $\theta = (\theta_1, \ldots, \theta_D)'$ is defined:

$$\nabla\mathcal{R}(\theta) = \left(\frac{\partial\mathcal{R}}{\partial\theta_1}(\theta), \cdots, \frac{\partial\mathcal{R}}{\partial\theta_D}(\theta)\right)'. \tag{18}$$

The gradient $\nabla\mathcal{R}(\theta)$ is the vector which points from θ in the direction of the steepest ascent ("uphill") of the performance surface. The negative gradient $-\nabla\mathcal{R}(\theta)$ is the direction of the steepest descent (Figure 14).

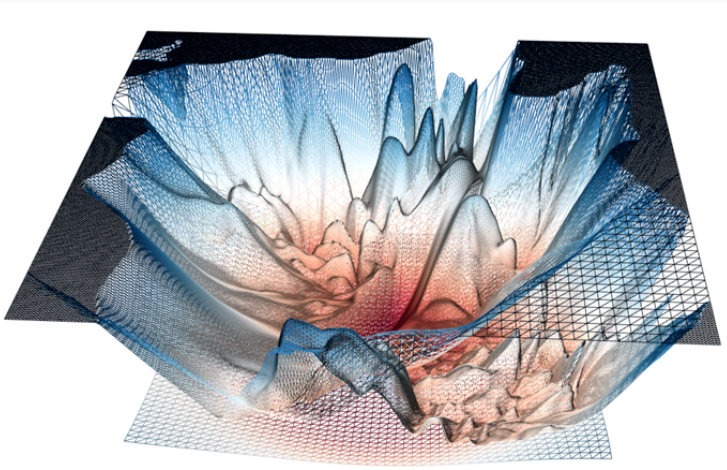

Figure 13: A (2-dimensional cross-section of) a performance surface for a neural network. The performance landscape shows the variation of the loss when (merely) 2 weights in the network are varied. Source: `http://www.telesens.co/2019/01/16/neural-network-loss-visualization/`

The idea of model optimization by gradient descent is to iteratively move toward a minimal-risk solution $\theta^{(\infty)} = \lim_{n\to\infty} \theta^{(n)}$ by always "sliding downhill" in the direction of steepest descent, starting from an initial model $\theta^{(0)}$. This idea is as natural and compelling as can be. Figure 15 shows one such itinerary.

The general recipe for iterative gradient descent learning goes like this:

Given: A differentiable risk function $\mathcal{R} : \Theta \to \mathbb{R}^{\geq 0}$.

Wanted: A minimal-risk model θ_{opt}.

Start: Guess an initial model $\theta^{(0)}$.

Iterate until convergence: Compute models

$$\theta^{(n)} = \theta^{(n-1)} - \mu \nabla \mathcal{R}(\theta^{(n-1)}). \tag{19}$$

The *adaptation rate* (or *learning rate*) μ is set to a small positive value.

An obvious weakness of this elegant and natural approach is that the final model $\theta^{(\infty)}$ depends on the choice of the initial model $\theta^{(0)}$. In complex risk landscapes (as the one shown in Figure 13) there is no hope of guessing an initial model which guarantees to end in the global minimum. This circumstance is generally perceived and accepted. There is a substantial mathematical literature that amounts to "if the initial model is chosen with a good heuristic, the local minimum that will be reached will be a rather good one with high probability".

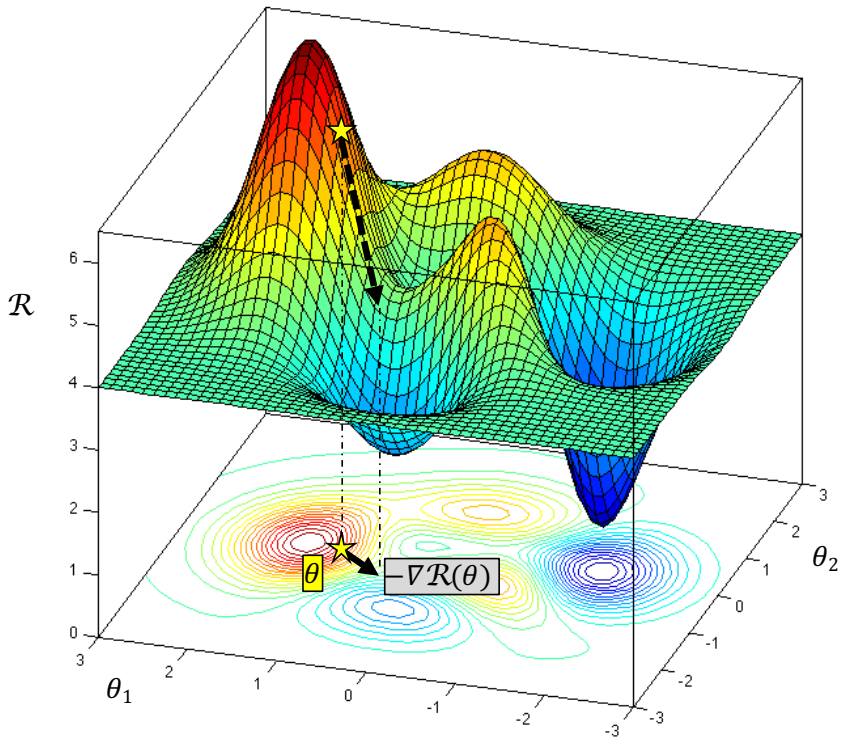

Figure 14: A performance surface for a 2-dimensional model family with parameters θ_1, θ_2, with its contour plot at the bottom. For a model θ (yellow star in contour plot) the negative gradient is shown as black solid arrow. It marks the direction of steepest descent (broken black arrow) on the performance surface.

Gradient descent: stability XOR speed of convergence. Neural networks — whether old-fashioned Tin Lizzie (look it up on Wikipedia) MLPs or the deep Ferraris of our days — are trained by gradient descent on a performance landscape. While gradient descent seems simple and fool-proof, things actually can get wildly out of hand quickly. The situation shown in Figure 15 looks deceptively simple; don't believe that gradient descent works as smoothly as that in real-life applications. In fact, the difficulties are so severe that during the years 1986-2006 between the re-birth of neural networks and the beginning of the deep learning era, they were essentially unsurmountable and forced the neural network community to stay confined to "shallow" MLPs with 1 or 2 hidden layers.

I should however mention that not everybody in the field was entirely helpless. In particular, Yann LeCun ploughed steadily forward and created a series of better and better and deeper and deeper neural networks for image classification, devel-

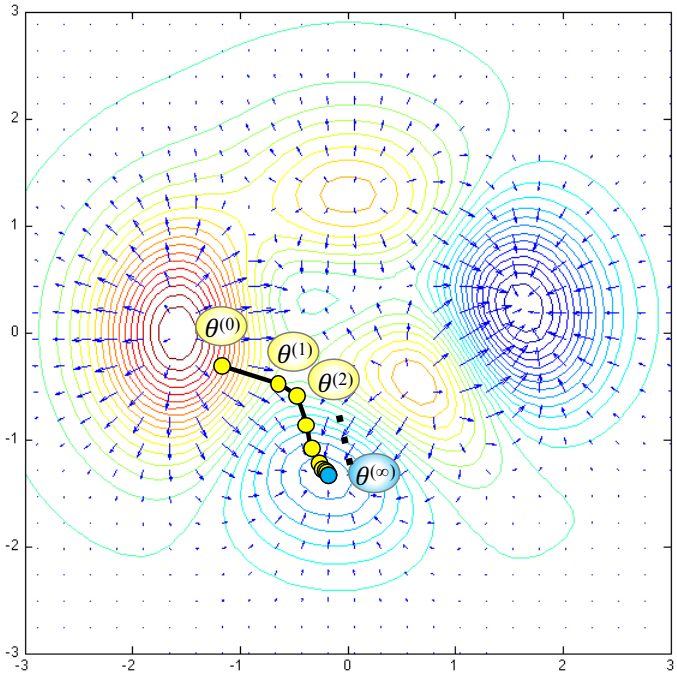

Figure 15: A gradient descent itinerary, re-using the contour map from Figure 14 and starting from the initial point shown in that figure. Notice the variable jump length and the sad fact that from this initial model $\theta^{(0)}$ the global minimum is missed. Instead, the itinerary slides toward a local minimum at $\theta^{(\infty)}$. The blue arrows show the negative gradient at raster points. They are perpendicular to the contour lines and their size is inversely proportional to the spacing between the contour lines.

oping the architecture of convolutional neural networks on his way (LeCun et al., 1998), and being more able than anybody else to train these beasts by gradient descent. However, the techniques that he applied were so subtle and not 100% documented that other researchers were not generally able to reliably reproduce his achievements.

I will now briefly outline one of the most painful show stoppers for simple-minded gradient descent.

First I note the obvious: differentiable performance surfaces are ususally not linear but curved. This simple fact opens the doors for trouble.

In order to understand why curvature makes gradient descent difficult, I consider a special situation where the main argument comes to the surface most clearly. Assume that your gradient descent optimization already has brought you into the close neighborhood of a local minimum, like the point $\theta^{(\mathrm{inf})}$ indicated in Figure 15. In the neighborhood of such a local minimum, the curved shape of the performance surface $\mathcal{R}(\theta)$ can be approximated by the first and second order terms of the Taylor expansion of $\mathcal{R}(\theta)$ around $\theta^{(\mathrm{inf})}$. Allow me to skip some maths here (you find it detailed in the ML lecture notes, Section 11.3.4) which amounts to a sequence of coordinate transformations. These coordinate transformations simplify the picture to the scenario shown in Figure 16.

This graphics shows a contour plot of a performance surface over a two-dimensional parameter space $\Theta = \mathbb{R}^2$ after coordinate shifts that move the local minimum $\theta^{(\infty)}$ to the origin and rotate the entire coordinate system such that the main axes of the "trough" of the surface are aligned with the drawing axes.

Assume that during a gradient descent run you have arrived at some parameter vector $\theta^{(n)}$. In order to determine the next parameter vector $\theta^{(n+1)} = \theta^{(n)} - \mu\,\nabla\mathcal{R}(\theta^{(n)})$, you have to decide on a learning rate μ. If you choose μ too large (red arrow in Figure 16), you will jump forward along the direction of the negative gradient so far that you end up "on the other side of the valley" higher than you started — that is, the risk $\mathcal{R}(\theta^{(n+1)})$ at the next point is larger, not smaller, than at $\theta^{(n)}$. If you would continue to use this too large learning rate even through the next iterations, you would propel yourself upwards out of the "trough". The gradient descent algorithm has become *instable*.

In order to prevent this instability and divergence, the learning rate must be set to a sufficiently small value. Then (indicated by the green arrow in the figure) the gradient descent will continue moving downwards, stably reducing the risk at every step.

The price to pay for stability is slow convergence. If the safe small learning rate is kept for future iterations, the speed of approach to $\theta^{(\mathrm{inf})}$ becomes very slow as soon as the iterations reach the bottom of the "valley" (shaded dots in the figure). But you cannot switch to a larger learning rate: the algorithm would immediately become instable.

What does it mean, concretely, for a learning rate to be "too large" such that gradient descent becomes instable? The maths here are actually not very involved

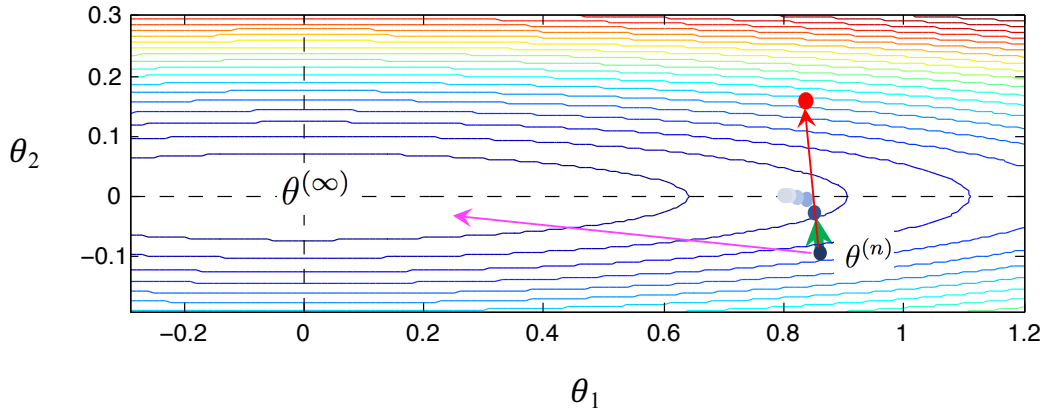

Figure 16: Gradient descent in the neighborhood of a local minimium $\theta^{(n)}$ (centered at the origin of a shifted coordinate system). Red arrow: gradient descent overshoots with too large learning rate. Green arrow: learning rate small enough for stability. Magenta arrow: weight change direction when a second-order gradient descent method would be used. For more explanation see text.

(detailed in the machine learning lecture notes). Here I give a summary of the main insights:

- After the coordinate transformations that were used to produce the situation shown in Figure 16, the geometric shape of the performance landscape (in its Taylor approximation up to order 2) is given by the formula

$$\mathcal{R}(\theta) = \sum_{i=1}^{D} \lambda_i \, \theta_i^2,$$

  where $\theta = (\theta_1, \ldots, \theta_D)'$ and the quadratic terms $\lambda_i \, \theta_i^2$ give the shape of the parabolic cross-sections through the performance surface along the axes θ_i. All λ_i are non-negative.

- In order to prevent instability, the learning rate must be set to a value smaller than $1/\lambda_{\max}$, where $\lambda_{\max}$ is the largest among the coefficients λ_i.

- The optimal speed of convergence is attained if the learning rate is set to

$$\mu = \frac{1}{\lambda_{\min} + \lambda_{\max}},$$

  where $\lambda_{\max}$ is the smallest among the coefficients λ_i.

- Given a stability-ensuring setting of the learning rate, once the iterations $\theta^{(n)}$ have reached the "bottom" of the "valley", the final approach toward the minimum value $\theta^{(\infty)}$ progresses geometrically, that is, the distance from $\theta^{(n+1)}$ to $\theta^{(\infty)}$ is smaller than the distance from $\theta^{(n)}$ to $\theta^{(\infty)}$ by a constant factor $0 < \beta < 1$. This approach factor is equal to

$$\beta = \frac{\lambda_{\max}/\lambda_{\min} - 1}{\lambda_{\max}/\lambda_{\min} + 1}.$$

  The ratio $\lambda_{\max}/\lambda_{\min}$ dictates the boundary conditions for successful gradient descent optimization (close to a local minimum, and up to the order-two Taylor approximation). If $\lambda_{\max} = \lambda_{\min}$ one gets $\beta = 0$ and with an optimally selected learning rate one achieves convergence in a single step. This is possible if the performance surface around the minimum has a perfectly circular shape. If conversely the ratio $\lambda_{\max}/\lambda_{\min}$ is very large (very long and sharp valley shape of the performance surface), β approaches 1 and gradient descent becomes so slow that it amounts to a stillstand.

Bad news: the more layers a network has, the more common it is that the ratio $\lambda_{\max}/\lambda_{\min}$ is very large. In a study of convergence properties of gradient descent minimization (for hyperparameter optimization of a recurrent neural network (Jaeger et al., 2007)) I once numerically measured values for this ratio in the order of 1e+14.

Besides this special case of convergence close to a minimum, there are other geometric scenarios at other places on a performance surface where a gradient descent itinerary will pass by (e.g., saddle points of the performance landscape) which lead to similar inherent conflicts between stability and speed of convergence.

One escape from the instability – slowness dilemma is to go for *second-order* gradient descent methods. These methods determine the direction vector for the next weigth adaption $\theta^{(n)} \rightarrow \theta^{(n+1)}$ based not only on the gradient, but also on the curvature of the performance surface at the current model $\theta^{(n)}$. This curvature is given by the $M \times M$ sized *Hessian* matrix which contains the second-order partial derivatives $\partial^2 \mathcal{R} / \partial\theta_i \, \partial\theta_j$. In ideal scenarios (when the second-order Taylor approximation is precise), second-order gradient descent points into exactly the direction to the target minimum (magenta arrow in Figure 16). However, computing the Hessian is expensive and may be subject to numerical problems, and the preconditions for making second-order methods work well (good approximation of surface by second-order Taylor expansion; being close to local minimum) may be badly violated. Numerous variants of second-order methods have been developed, aiming at reduced costs or more robust coping with local geometry. There is unfortunately no general rule of when or whether it is beneficial to use what type of second-order method.

These difficulties are severe. For two decades they had made it impossible to effectively train MLPs on advanced "cognitive" tasks in image interpretation

and language processing. Finding heuristic mechanisms to modify plain gradient descent in ways that keeps the iterations stable, while maintaining a sufficient speed of convergence, has been one of the enabling factors for deep learning.

The classical error backpropagation algorithm I will now describe the classical algorithm for training MLPs by plain gradient-descent — the error backpropagation algorithm.

Given: Labelled training data in vector/vector format (obtained possibly after preprocessing raw data) $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$, where $\mathbf{u} \in \mathbb{R}^K, \mathbf{y} \in \mathbb{R}^M$. Also given: a network architecture for models $\mathcal{N}_\theta$ that can be specified by M-dimensional weight vectors $\theta \in \Theta$. Also given: a loss function $L : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}^{\geq 0}$.

Initialization: Choose an initial model $\theta^{(0)}$. This needs an educated guess. A widely used strategy is to set all weight parameters $w_i^{(0)} \in \theta^{(0)}$ to small random values. *Remark:* For training deep neural networks this is not good enough — the deep learning field actually got kickstarted by a clever method for finding a good model initialization (Hinton and Salakuthdinov, 2006).

Iterate: Compute a series $(\theta^{(n)})_{n=0,1,\ldots}$ of models of decreasing empirical loss (aka training error) by gradient descent. Concretely, with

$$R^{\mathsf{emp}}(\mathcal{N}_{\theta^{(n)}}) = \frac{1}{N} \sum_{i=1,\ldots,N} L(\mathcal{N}_{\theta^{(n)}}(\mathbf{u}_i), \mathbf{y}_i) \tag{20}$$

denoting the empirical risk of the model $\theta^{(n)}$, and with

$$\nabla R^{\mathsf{emp}}(\mathcal{N}_{\theta^{(n)}}) = \left(\frac{\partial R^{\mathsf{emp}}}{\partial w_i}(\theta^{(n)}), \ldots, \frac{\partial R^{\mathsf{emp}}}{\partial w_D}(\theta^{(n)}) \right)' \tag{21}$$

being the gradient of the empirical risk with respect to the parameter vector $\theta = (w_1, \ldots, w_D)'$ at point $\theta^{(n)}$, update $\theta^{(n)}$ by

$$\theta^{(n+1)} = \theta^{(n)} - \mu \, \nabla \, R^{\mathsf{emp}}(\mathcal{N}_{\theta^{(n)}}).$$

In these iterations, make sure that the adaptation rate μ is small enough to guarantee stability.

Stop when a stopping criterion chosen by you is met. This can be reaching a maximum number of iterations, or the empirical risk decrease falling under a predermined threshold, or some early stopping scheme.

Computing a gradient is a differentiation task, and differentiation is easy. With high-school maths you would be able to compute (21). However, the gradient formula that you get from textbook recipes would be too expensive to compute. The *error backpropagation* algorithm (or simply "backprop" for the initiated, or

even just BP if you want to have a feeling of belonging to this select community) is a specific algorithmic scheme to compute this gradient in a computationally efficient way.

Every student of machine learning *must* have understood it in detail at least once in his/her life, even if later it's just downloaded from a toolbox in some more sophisticated fashioning. Thus, brace yourself and follow along!

Let us take a closer look at the empirical risk (20). Its gradient can be written as a sum of gradients

$$\nabla R^{\mathrm{emp}}(\mathcal{N}_\theta) = \nabla \left(\frac{1}{N} \sum_{i=1,\dots,N} L(\mathcal{N}_\theta(\mathbf{u}_i), \mathbf{y}_i) \right) = \frac{1}{N} \sum_{i=1,\dots,N} \nabla L(\mathcal{N}_\theta(\mathbf{u}_i), \mathbf{y}_i),$$

and this is also how it is actually computed: the gradient $\nabla L(\mathcal{N}_\theta(\mathbf{u}_i), \mathbf{y}_i)$ is evaluated for each training example $(\mathbf{u}_i, \mathbf{y}_i)$ and the obtained N gradients are averaged.

This means that at every gradient descent iteration $\theta^{(n)} \to \theta^{(n+1)}$, all training data points have to be visited individually. In MLP parlance, such a sweep through all data points is called an *epoch*. In the neural network literature one finds statements like "the training was done for 120 epochs", which means that 120 average gradients were computed, and for each of these computations, N gradients for individual training example points $(\mathbf{u}_i, \mathbf{y}_i)$ were computed.

When training samples are large — as they should be — one epoch can clearly be too expensive. Therefore one often takes resort to *minibatch* training, where for each gradient descent iteration only a subset of the total sample S is used.

The backpropagation algorithm is a subroutine in the gradient descent game. It is a particular algorithmic scheme for calculating the gradient $\nabla L(\mathcal{N}_\theta(\mathbf{u}_i), \mathbf{y}_i)$ for a single data point $(\mathbf{u}_i, \mathbf{y}_i)$. Naive highschool calculations of this quantity incur a cost of $O(D^2)$ (where D is the number of network weights). When D is not extremely small (it will almost never be extremely small — a few hundreds of weights will be needed for simple tasks, and easily a million for deep networks applied to serious real-life modeling problems), this cost $O(D^2)$ is too high for practical exploits (and it has to be paid N times in a single gradient descent step!). The backprop algorithm is a clever scheme for computing and storing certain auxiliary quantities which cuts down the cost from $O(D^2)$ to $O(D)$.

Here is how backprop works in order to compute the loss gradient $\nabla\, L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})$ for a training example $(\mathbf{u}, \mathbf{y})$.

1. BP works in two stages. In the first stage, called the *forward pass*, the current network $\mathcal{N}_\theta$ is presented with the input $\mathbf{u}$ and the output $\hat{\mathbf{y}} = \mathcal{N}_\theta(\mathbf{u})$ is computed using the "forward" formulas (15) and (16). During this forward pass, for each unit x_i^κ which is not a bias unit and not an input unit the quantity

$$a_i^\kappa = \sum_{j=0,\dots,L^{\kappa-1}} w_{ij}^\kappa \, x_j^{\kappa-1} \tag{22}$$

is computed – this is sometimes referred to as the *potential* of unit x_i^κ, that is its internal state before it is passed through the sigmoid.

2. *A little math in between.* Applying the chain rule of calculus we have

$$\frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial w_{ij}^\kappa} = \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial a_i^\kappa} \frac{\partial a_i^\kappa}{\partial w_{ij}^\kappa}. \tag{23}$$

Define

$$\delta_i^\kappa = \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial a_i^\kappa}. \tag{24}$$

Using (22) we find

$$\frac{\partial a_i^\kappa}{\partial w_{ij}^\kappa} = x_j^{\kappa-1}. \tag{25}$$

Combining (24) with (25) we get

$$\frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial w_{ij}^\kappa} = \delta_i^\kappa \, x_j^{\kappa-1}. \tag{26}$$

Thus, in order to calculate the desired derivatives (23), we only need to compute the values of δ_i^κ for each hidden and output unit.

3. *Computing the δ's for output units.* Output units x_i^k are typically set up differently from hidden units, and their corresponding δ values must be computed in ways that depend on the special architecture. For concreteness here I stick with the simple linear units introduced in (16). The potentials a_i^k are thus identical to the output values $\hat{y}_i$ and we obtain

$$\delta_i^k = \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial \hat{y}_i}. \tag{27}$$

This quantity is thus just the partial derivative of the loss with respect to the i-th output, which is usually simple to compute. For the quadratic loss $L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y}) = \|\mathcal{N}_\theta(\mathbf{u}) - y\|^2$, for instance, we get

$$\delta_i^k = \frac{\partial \|\mathcal{N}_\theta(\mathbf{u}) - \mathbf{y}\|^2}{\partial \hat{y}_i} = \frac{\partial \|\hat{\mathbf{y}} - \mathbf{y}\|^2}{\partial \hat{y}_i} = \frac{\partial (\hat{y}_i - y_i)^2}{\partial \hat{y}_i} = 2\,(\hat{y}_i - y_i). \tag{28}$$

4. *Computing the δ's for hidden units.* In order to compute δ_i^κ for $1 \leq \kappa < k$ we again make use of the chain rule. We find

$$\delta_i^\kappa = \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial a_i^\kappa} = \sum_{l=1,\ldots,L^{\kappa+1}} \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial a_l^{\kappa+1}} \frac{\partial a_l^{\kappa+1}}{\partial a_i^\kappa}, \tag{29}$$

which is justified by the fact that the only path by which a_i^κ can affect $L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})$ is through the potentials $a_l^{\kappa+1}$ of the next higher layer. If we substitute (24) into (29) and observe (22) we get

$$\begin{aligned}
\delta_i^\kappa &= \sum_{l=1,\ldots,L^{\kappa+1}} \delta_l^{\kappa+1} \, \frac{\partial a_l^{\kappa+1}}{\partial a_i^\kappa} \\
&= \sum_{l=1,\ldots,L^{\kappa+1}} \delta_l^{\kappa+1} \, \frac{\partial \, \sum_{j=0,\ldots,L^\kappa} w_{lj}^{\kappa+1} \, \sigma(a_j^\kappa)}{\partial a_i^\kappa} \\
&= \sum_{l=1,\ldots,L^{\kappa+1}} \delta_l^{\kappa+1} \, \frac{\partial \, w_{li}^{\kappa+1} \, \sigma(a_i^\kappa)}{\partial a_i^\kappa} \\
&= \sigma'(a_i^\kappa) \sum_{l=1,\ldots,L^{\kappa+1}} \delta_l^{\kappa+1} \, w_{li}^{\kappa+1}. \qquad (30)
\end{aligned}$$

This formula describes how the δ_i^κ in a hidden layer can be computed by "back-propagating" the $\delta_l^{\kappa+1}$ from the next higher layer. The formula can be used to compute all δ's, starting from the output layer (where (27) is used — in the special case of a quadratic loss, Equation 28), and then working backwards through the network in the *backward pass* of the algorithm.

When the logistic sigmoid $\sigma(a) = 1/(1+\exp(-a))$ is used, the computation of the derivative $\sigma'(a_i^\kappa)$ takes a particularly simple form, observing that for this sigmoid it holds that $\sigma'(a) = \sigma(a)\,(1-\sigma(a))$, which leads to

$$\sigma'(a_i^\kappa) = x_i^\kappa \, (1 - x_i^\kappa).$$

Although simple in principle, and readily implemented, using the backprop algorithm appropriately is something of an art, even in basic shallow MLP training. Here is only place to hint at some difficulties:

- The stepsize μ must be chosen sufficiently small in order to avoid instabilities. But it also should be set as large as possible to speed up the convergence. We have inspected the inevitable conflict between these two requirements above. Generally one uses adaptation schemes that modulate the learning rate as the gradient descent proceeds. Clever methods for online adjustment of stepsize have been one of the enabling factors for deep learning. For shallow MLPs typical stepsizes that can be fixed without much thinking are in the order from 0.001 to 0.01.

- Gradient descent on nonlinear performance landscapes may sometimes be crippling slow in "plateau" areas still far away from the next local minimum where the gradient is small in all directions, for other reasons than what we have argued above with the help of Taylor expansions.

- Gradient-descent techniques on performance landscapes can only find a local minimum of the risk function. This problem can be addressed by various measures, all of which are computationally expensive. For deep networks of large size the local minimum problem seems not particularly problematic. These networks afford of such a brute overfitting potential that on the downhill slide along the negative gradient, the point of overfitting and thus "early stopping" is reached earlier than the local minimum one is heading to. Or, in other words: most local minima represent overfitting models, thus one does not want to reach them.

A truly beautiful visualization of MLP training has been pointed out to me by Rubin Dellialisi: `playground.tensorflow.org/`.

Simple gradient descent with the BP algorithm, as described above, is cheaply computed but may take long to converge and/or run into stability issues. A large variety of more sophisticated iterative loss minimization methods have been developed in the deep learning field, which often succeed in combining an acceptable speed of convergence with stability. Some of them refine gradient descent, others ("second-order" methods) use information from the local curvature of the performance surface. The main alternatives are nicely described in `https://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network` (retrieved May 2017, local copy at `http://minds.jacobs-university.de/uploads/teaching/share/NNalgs.zip`).

2.3 A glimpse at deep learning

Today, neural networks in machine learning means "deep learning" (DL). In my opinion deep learning is a technological revolution like the invention of the steam engine or the transistor. Due to its glamour and enormous industrial funding, deep learning research has spell-bound almost all of the best young ML scientists in the last five to ten years or so. The field is extremely productive and has enormously diversified from its early days when it could be defined as "successfully handling MLPs with more than 3 hidden layers". This course cannot give a serious introduction to DL. The standard textbook is Goodfellow et al. (2016).

2.3.1 Convolutional neural networks

But I will allow ourselves at least a glimpse at DL, by taking a closer look at *convolutional neural networks* (aka *convnets*, or CNNs). This is a deep architecture, pioneered in the 1990's by Yann LeCun (`yann.lecun.com` — this personal academic website of his seems not to have been updated since about six years; this is understandable given that LeCun's leading roles in DL have propelled him to become VP and Chief AI Scientist of Facebook). However, as almost always in science, LeCun did not invent CNNs out of thin air. Some 15 years before LeCun's

models, the main structural elements of CNN architectures had already been assembled in the *Neocognitron* NN of Fukushima (1980), a historical fact that was properly acknowledged by LeCun (LeCun et al., 1998) but today is rarely mentioned. The Neocognitron was not trained in a supervised way (the backpropo algorithm was no public knowledge then) but with a biologically inspired unsupervised learning algorithm. This condition, combined with the limited compute powers of those days and the fact that Fukushima's scientific goals were in biological modeling (not machine learning), effected that the Neocognitron has remained under-appreciated in the machine learning community. In turn, Fukushima drew inspiration, and adopted some terminology, from Hubel and Wiesel's Nobel prize winning research on the early visual processing in the mammalian brain in the early 1960's.

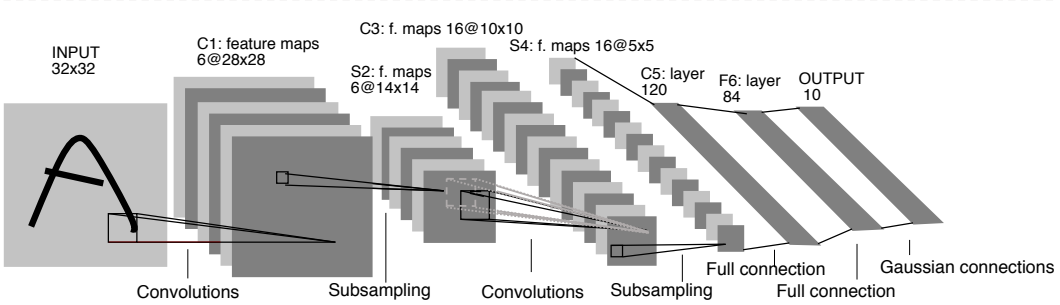

Figure 17: Architecture of LeCun's *LeNet 5* convolutional neural network for handwritten character recognition. For explanation see text. Image adapted from LeCun et al. (1998)

The architecture of CNNs is specialized for image processing tasks. Figure 17 sketches the architecture of *LeNet 5*, the acclaimed common ancestor of all modern deep learning CNNs (see the Wikipedia article `https://en.wikipedia.org/wiki/LeNet` for detail). The defining feature of CNNs is that they implement a repeated sequence of *convolutional* and *subsampling* (aka *pooling*) layers. The input layer is a 2-dimensional "retina" on which pixel images can be projected as external inputs. Following the schema in Figure 17, I give a brief explanation of the next processing layers (very readable detailed description in the original paper LeCun et al. (1998)):

- The first processing layer C1 after the retina is made of 6 *feature maps*. Each feature map is a 2-dimensional array of feature detector neurons. Each such feature detector neuron receives its input from a 5×5 sized *receptive field* in the retina. Thus, each such neuron has 25 (plus 1 bias) weights that need to be trained. Within each of the 6 feature maps, every feature detector neuron however has the same weights as all other neurons in that feature map. This *weight sharing* (aka *weight tying*) is imposed during the backpropagation

training. Mathematically, the transformation from the raw input image to a feature map is a *convolution* of the image array with the 5×5 weight *filter*, giving this kind of networks its name.

- The second processing layer S2 reduces the 28×28 size of the C1 feature maps to a size of 14×14, by creating for each of the six C1 feature maps a subsampled feature map. The neurons in these subsampled maps have 2×2 receptive fields in the C1 feature maps. In LeNet5, the four activations in each of the receptive fields are first averaged, then passed to the receiving subsampling neuron with a trainable weight and bias (also with weight sharing imposed within each of the S2 maps).

- The next layer C3 is again a convolutional layer. Each of its feature maps (of which there are 16 in LeNet5) takes its input from several 5×5 sized receptive fields, located in several of the 6 subsampling layers. In Figure 17, I indicated one neuron in a C3 feature map receiving input from two 5×5 sized receptive fields.

- Then, like what was done from C1 to S2, these 16 feature maps are subsampled into 16 subsampled S4 maps.

After these convolution-subsampling layers, LeNet5 has two further MLP-like processing layers and a final output layer which uses a neuron model that is different from the standard summation-sigmoid model (14).

The background ideas that motivated LeCun (and before him, Fukushima) to think out this CNN design:

- The local feature extraction is biologically inspired;

- the weight sharing reduces the number of trainable parameters, which leads to a regularization effect in the sense of machine learning;

- the weight sharing also favors translational invariance for recognition performance (for instance, a local feature which detects a sharp edge like at the top of the letter A would become activated insensitive to shifts of the position on the input letter A on the retina);

- the subsampling further increases the robustness of the classification performance against input pattern shifts and scalings.

Convnets that were later developed in the DL era had more convolutional layers and subsequent processing stages which are more complex than in their ancestor LeNet5. Das (2017) is an online overview of later CNN architectures.

CNNs can also be designed for 1-dimensional "images". This makes sense for input data vectors $\mathbf{u} = (u_1, \ldots, u_K)'$ where the ordering of vector elements has a geometrical meaning. This would be expected, for instance, when the inputs $\mathbf{u}$ are

energy profiles over a frequency spectrum, or flow velocity measurements along a flow sensor line as in a recent *cum laude* PhD thesis written in the AI department Wolf (2020). In contrast, it would not make any sense whatsoever, for instance, to throw a 1-dim CNN at inputs that are bag-of-words count vectors!

2.3.2 The recklessness of DL: end-to-end training.

A common denominator of today's deep learning architectures is that they are trained by some version of gradient descent in a supervised learning set-up (well, that is a simplification... there are also unsupervised learning set-ups like auto-encoder learning (Vincent et al., 2008), semi-supervised learning set-ups (Kingma et al., 2014), reinforcement learning additions to DL set-ups (Mnih, et al, 2015), or the generative adversarial training principle (tutorial for *generative adversarial networks*, GANs: Goodfellow (2017))... but ok., well, even today, gradient descent optimization in supervised settings is a core component of DL).

These learning architectures are in most cases, but not always, construed as neural networks of some sort (a notable exception: the Turing-machine inspired, "hybrid computing" architecture of Graves et al (2016) which combines a "controller" neural network with a non-neural memory subsystem which is reminiscent of the tape of a Turing machine). The field is called "deep learning" because the learning architectures typically pull their input through a sequence of many processing layers, sometimes through different submodules which operate in parallel. At any rate, the input patterns become transformed many times before they reach the output interface. All of these transformations are differentiable, making the entire, deep input-to-output transformation differentiable. The successes of DL rest on three enabling conditions, which became available only in the last decade or so:

- access to a lot of GPU power — affordable hardware and convenient programming libraries,
- huge datasets for training,
- and an ever-growing collection of computational "tricks of the trade" to put the gradient-descent monster in shackles.

A catchphrase that one often hears in connection with DL is *end-to-end training*. This is not a precisely defined concept. In a narrow technical sense, it describes the fact that modern backprop makes it possible to propagate the errors obtained at the output "end" all the way back through the many processing layers until the input "end" is reached. But often the phrase "end to end" is meant to carry a much stronger, bold idea: namely, that DL architectures can learn their tasks on the basis of being fed with the un-preprocessed "raw" input data from real-world data sources.

From the perspective of conventional machine learning wisdom, this is a very reckless attitude. In my machine learning lecture notes I spend a great effort on explaining how important and helpful it is to pre-process "raw" data for dimension reduction, extracting features which are specifically appropriate for the task at hand. Using raw, high-dimensional data is the way to failure, for a number of good reasons that I unfold in those lecture notes.

So, — whom should you believe and follow?

Answer: both views are right — but in different sitations, not at the same time. The critical issue is overfitting.

Flooding a deep learning system with high-dimensional raw data and training it "end to end" will work if (and only if) one has large quantities of data. Together with the clever modern regularization methods for deep networks (plus, possibly, GAN methods), large data volumes minimize the dangers of overfitting. Such convenient conditions (very large data sets, plus high-performance computing facilities and a lot of professional expertise) is what Google, Facebook and their likes can enjoy.

However, the ordinary mortal machine learning professional most often will only dispose of scarce training data, while nonetheless wanting to solve a complex task. A typical non-Google setting is medical image processing. It is a notorious problem in this field that there are never enough training examples. Yet, the targeted input-output tasks (like cancer diagnostics from liver CT scans) are eminently complex. Training a deep CNN "end to end" with a few hundreds of unpreprocessed CT images will not work. Complex data processing pipelines, only some of them being gradient-descent trained neural networks and others informed by human insight for appropriate feature definition, will be necessary to come up with good solutions (for example, Chlebus et al. (2018)).

3 A high-speed guided tour through dynamical systems

If one wants to design artificial cognitive systems, or if one wants to understand natural cognitive systems, there is no way to not consider *time* as a fundamental dimension of cognition. Seeing, hearing, reasoning, dreaming, speaking, hallucinating, feeling... can you imagine any of these happening not in time?

Feedforward networks are, mathematically speaking, functions. They map an input pattern to an output pattern – a static argument-value relationship. No time involved.

Biological brains are not feedfoward networks but recurrent networks (RNNs). A recurrent network is a neural network that has at least one connection feedback cycle. I will now demonstrate that neural network science isn't easy. Consider a simple RNN made from only two neurons that are recurrently connected with each other and each of them also with itself ("self-feedback"). Not giving this system

any external input, nor a bias, its state update equation is

$$\mathbf{x}(n+1) = \tanh(W\,\mathbf{x}(n)), \tag{31}$$

where $\mathbf{x}(n)$ is the 2-dimensional neural activation vector at time $n = 0, 1, 2, \ldots$ and W is the 2×2 weight matrix of the synaptic connection weights which feeds the activations $\mathbf{x}(n)$ of the two neurons back to determine the next state $\mathbf{x}(n+1)$. We set

$$W_1 = \begin{pmatrix} 4 & -4 \\ 1 & -1 \end{pmatrix}$$

and also consider $W_2 = 3.3 \cdot W_1, W_3 = 4.3 \cdot W_1$, that is, W_2, W_3 are just scaled versions of W_1. We carry out three simulation runs where we iterate (31) for 1000 timesteps. In the first run we use W_1, in the second run we use W_2, in the third run we use W_3, plugging them into (31). Each run is startet at time $n = 0$ from some arbitrary random initial state $\mathbf{x}(0)$. After a short initial transient where the arbitrary initial state is "forgotten" (or "washed out"), the temporal sequence $\mathbf{x}(n)$ stabilizes into a pattern which characteristically depends on W_1, W_2 or W_3, respectively. Figure 18 shows what we get:

- In the run using W_1, all activity "dies out" and the network gets locked in a stable, unchanging state $\mathbf{x}(n) = \mathbf{x}(n+1) = \ldots$.
- With W_2, the network starts to oscillate, here with a period 4, that is $\mathbf{x}(n) = \mathbf{x}(n+4) = \mathbf{x}(n+8) = \ldots$. Other periods (powers of 2) could also be obtained by using other scalings of W_1, not shown.
- When the weight matrix is scaled above some critical value (which here is about 3.55), all regularity breaks down and the behavior becomes *chaotic*. There are no periodicities; patterns will never repeat exactly.

Biological neural networks exhibit the same kinds of characteristics. Stable states occur, for instance, in your motor control circuits of your neck muscles when you keep your head steadily upright. Periodic activation sequences occur extremely often — for instance in the neural ganglia, embedded on the surface of your heart, which generate heartbeat signals; or when you memorize a number by mentally repeating it; or in an epileptic seizure. Numerous occasions have been hypothetically proposed for chaotic activation patterns, for instance as being the carrier of complex sensory representations or as the background activation in an awake, alert, resting state. I should however point out that it is experimentally and mathematically very difficult to distinguish "noise" from "chaos" in high-dimensional neural recordings, thus the final verdicts on the role of chaos in cognitive neurodynamics remain to be spoken.

The dizzy dancing of just two coupled neurons already is hard to follow even with a mathematician's eye. But that's only 2 (two) neurons. You own about 85 billion neurons (or those neurons own you). The door to understanding their

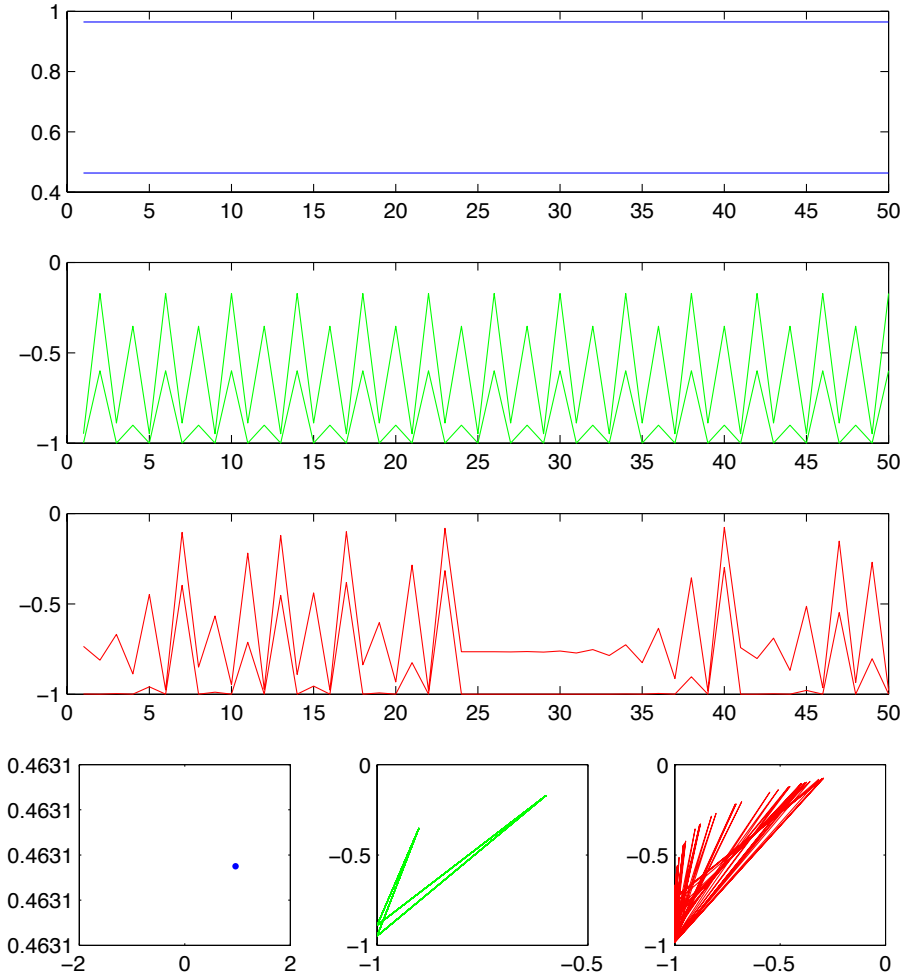

Figure 18: Running the 2-neuron network with weight matrices W_1 (blue, top plot and first plot in bottom row), W_2 (green, second plot and second panel in bottom row), and W_3 (red, third plot and last panel in bottom row). The last 50 time steps from the 1000 step simulation are shown. First three plots show the development of the activations of the two neurons plotted against time. The panels in the bottom row give another visualization of the same traces: here the two axes of the panel correspond to the two components x_1, x_2 of the state vectors $\mathbf{x}(n)$, giving a 2-dimensional representation of the temporal evolution.

dance is the study of recurrent neural networks. And the key to open that door is the theory of dynamical systems (DS). In this session and the next I will treat you to a crash course in DS essentials.

Since I have a nice set of Powerpoint slides for such a DS primer, and since slides are a good medium for online teaching, I will not prepare a written-out chapter for these lecture notes (at least, not in this Corona year). You can find the slides in the "Lecture notes" section on Nestor. There are two versions: DS_primer_4_lecture.pdf gives the slides that I present in the online lectures, and the material in this slideset is mandatory reading and will be asked in exams. This slideset is a subset of a my full DS course slideset, which you can find on Nestor as DS_primer_complete.pdf if you are interested.

4 Recurrent neural networks in deep learning

The deep learning (DL) revolution began with feedforward neural networks, especially CNNs. Wait... that is not quite true; history isn't that simple. The pioneering paper of Hinton and Salakuthdinov (2006), which (not only) I consider as a kickoff work for DL, was about a special sort of RNNs, the *restricted Boltzmann machine* (RBM). We will treat RBMs later in this course. In that pioneering work, the powers of RBMs were demonstrated on information compression tasks. Only in passing it was also mentioned that RBMs can be used to initialize the backprop learning for MLPs — the rest is history. After creating excitation and much respect (because it isn't easy at all to work with RBMs) in the early years of DL, RBMs vanished from the focus of attention again, though they are still being used and explored.

When it comes to recurrent neural networks, the DL stage is nowadays reserved for *Long Short-Term Memory* (LSTM) networks and their derivatives. Like convolutional neural networks, LSTM networks have a history that began before the DL revolution. Essential ideas were introduced already in the diploma thesis of Hochreiter (1991) and became fully worked out in Hochreiter and Schmidhuber (1997). However, LSTM networks only rose to the domineering role that they have today after reliable backpropagation algorithms became available through DL research. In our present time, virtually all advanced machine learning tasks which need to cope with timeseries data are handled with LSTM networks or close relatives of them. In particular, all speech recognition and text translation engines are based on such RNNs.

In order to understand LSTM networks, one has to make friends with two separate algorithmic and architectural techniques: (i) a generalization of the backprop algorithm from feedfoward networks to recurrent networks, called *backpropagation through time* (BPTT), and (ii) a special kind of complex neuron model, the *LSTM unit*. I will treat both topics in turn, but before I do that, I will explain what it means to carry out a supervised learning task with temporal data.

4.1 Supervised training of RNNs in temporal tasks

I restrict this presentation to discrete-time dynamics, where time is represented by integers $\ldots, n-1, n, n+1, \ldots$. Deep learning RNNs always use discrete time. In contrast, biological neural systems, and some of the recent developments in neuromorphic computing which I will hint out in the last session of this course, are based on continuous time.

4.1.1 A basic format of an RNN

One of the most basic kinds of RNNs (simpler than LSTM networks which I will explain later) is given by the following update equations:

$$\mathbf{x}(n+1) = \sigma(\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{\mathsf{in}}\,\mathbf{u}(n+1) + \mathbf{b}) \quad (32)$$

$$\mathbf{y}(n) = f(\mathbf{W}^{\mathsf{out}}\,\mathbf{x}(n)) \quad (33)$$

where

- $n = 0, 1, 2, \ldots, n_{\mathsf{max}}$ or $n = 0, 1, 2, \ldots$ are the time steps, which may run until an end time n_{max} or forever,
- Equation 32 specifies how the network activation state is updated from one timestep to the next, and Equation 33 specifies how the output vector is computed at time n,
- $\mathbf{x}(n)$ is the vector of activations of the neurons inside the RNN — I will generally denote the number of neurons in an RNN by L, so $\mathbf{x}(n) \in \mathbb{R}^L$,
- $\mathbf{W}$ is the $L \times L$ matrix containing the synaptic connection weights $w_{ij} \in \mathbb{R}$ giving the strength of the connection from neuron j to neuron i,
- $\mathbf{W}^{\mathsf{in}}$ is an $L \times K$ dimensional real-valued matrix containing the weights from K input neurons into the RNN — I will generally denote the dimension of the input signal $\mathbf{u}(n)$ by K,
- $\mathbf{b} \in \mathbb{R}^L$ is the vector of biases,
- σ is a sigmoid function (like in MLPs, typically the tanh, the logistic sigmoid, or the rectifier function), which is applied element-wise on the vector $\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{\mathsf{in}}\,\mathbf{u}(n) + \mathbf{b}$,
- $\mathbf{y}(n)$ is the output signal generated at time n — I will generally denote the dimension of the output signal by M,
- $\mathbf{W}^{\mathsf{out}}$ is an $M \times L$ sized output weight matrix containing the weights of synaptic connections from RNN neurons to output neurons,

- f is a wrapper function applied to the linear "readout" $\mathbf{W}^{\text{out}}\,\mathbf{x}(n)$; often again a sigmoid, but often also just the identity function.

Sometimes one wishes to feed back the generated output signal $\mathbf{y}(n)$ into the RNN. Equation 32 is then extended to the format

$$\mathbf{x}(n+1) = \sigma(\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{\text{in}}\,\mathbf{u}(n+1) + \mathbf{W}^{\text{fb}}\,\mathbf{y}(n) + \mathbf{b}),$$

where $\mathbf{W}^{\text{fb}}$ is an $L \times M$ sized matrix for feeding back the generated output signal $\mathbf{y}(n)$ into the RNN. Such *output feedback* is required when the task is not an input-to-output transformation where the output is determined by the input signal, but where instead the output is actively generated by the RNN, even in the absence of input. Such tasks occur specifically in robot motion generation.

In order to get the dynamics (32), (33) started, at time $n = 1$ the state $\mathbf{x}(0)$ in the right-hand side of (32) is replaced by a fixed *initial state* $\mathbf{x}_0$ of the network.

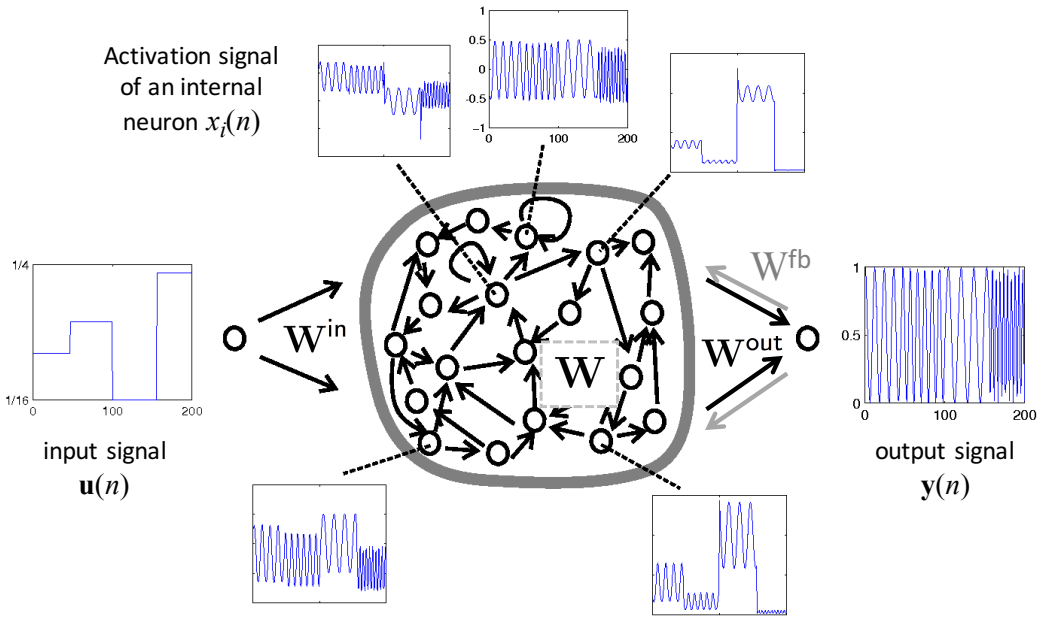

Figure 19: Schema of a basic RNN. Here an example is shown that has one-dimensional input and output signals, and hence only one input neuron and one output neuron. This particular RNN has been trained to generate a sinewave output whose frequency is controlled by the input: high input value gives high frequency output. Output feedback is needed here. Activation signals from a few internal neurons are shown. Image taken from `http://www.scholarpedia.org/article/Echo_state_network`.

Figure 19 illustrates the wiring schema of such a basic RNN. Many variations of the system equations (32), (33) are in use. Also, more complex architectures

than the one shown in the figure are used. For instance, in *hierarchical* RNNs, several recurrent sub-modules are stacked on top of each other. Or the signals from inside the RNN are propagated through a MLP in order to obtain a more complex "readout function" than what you can get with (33).

4.1.2 Some basic types of temporal tasks for RNNs

Temporal supervised learning tasks come in a number of quite different kinds. The common denominator is that the training data contain timeseries.

There are two basic types of tasks, depending whether the used data come from *stationary* or *non-stationary* dynamical systems:

Stationary dynamical systems are not changing their kind of signals as time goes on. Whether you observe a stationary system at early times, say from time $n = 0$ to time $n = 100$, will give you the same kind of signals as if you would observe it from time $n = 10000$ to time $n = 10100$. Examples are oscillations, monitoring a wind turbine, human or robot walking, or a video showing the surf on a beach.

The training data typically consists in a single, long sequence $S = (\mathbf{u}(n), \mathbf{y}(n))_{n=1,\ldots,n_{\mathsf{max}}}$ of paired input-output signals. The learning goal is to train an RNN which, when it is fed with a testing input signal $\mathbf{u}^{\mathsf{test}}(n)$, creates an output signal

$$\hat{\mathbf{y}}(n) = f(\mathbf{W}^{\mathsf{out}}\,\mathbf{x}(n)) = f(\mathbf{W}^{\mathsf{out}}\,\sigma(\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{\mathsf{in}}\,\mathbf{u}^{\mathsf{test}}(n) + \mathbf{b}))$$

which is close (in some appropriate loss function metrics) to the testing output, that is $\hat{\mathbf{y}}(n) \approx \mathbf{y}^{\mathsf{test}}(n)$.

Non-stationary dynamical systems use and generate signals whose characteristics changes with time. Observing such a system at different times will yield different signals. For instance, the microphone signal recorded when a human utters the word "zero" will have a spectral composition which in the beginning reflects the phonological nature of the letter "z", and at the end the phonological nature of the vocal "o". Or a stock index will just continue to grow (lucky investor!) throughout the observation period.

The training data for non-stationary tasks typically consist of many individual recordings of a limited-time observation, for instance in many sound recordings of people uttering the word "zero". Formally, such training data are double-indexed: $S = (\mathbf{u}^{(i)}(n), \mathbf{y}^{(i)}(n))_{n=1,\ldots,n^{(i)}_{\mathsf{max}}}$, where i is the index of the training example.

Stationary and non-stationary signal-generating systems can be deterministic or stochastic. Most real-life systems are stochastic.

The stationary vs. non-stationary distinction is not clear-cut. Whether a system is considered stationary or non-stationary depends on how long it is observed.

For instance, when a pair of dancers performing a waltz is monitored for an extended time, the different waltz figures repeat and the signal can be considered stationary. However, the entire waltzing performance can be *segmented* into short figures (which are individually trained in dancing lessons), where each figure consists in a specific non-stationary sequence of motions. The same holds for speech (words are non-stationary, but a long monologue can be considered stationary), weather (during a day there will be a non-stationary evolution of temperature and wind, but over the years everything repeats — ignoring climate change), and many other interesting signals. In order to learn RNN models for such mixed signals (short-term non-stationary, long-term stationary) one may pursue two strategies: either try to learn one large, complex RNN which is capable of changing its "mode" in shorter time intervals; or try to learn several smaller, simpler RNNs, one for each typical non-stationar subpattern, and then combine them into a compound multi-RNN system where the individual RNNs are activated / deactivated in turn.

I will now present a few distinct scenarios to illustrate the richness of temporal learning set-ups. They all come in both stationary and non-stationary versions.

Dynamical pattern generation tasks. Sometimes one would like to train a RNN such that it generates a temporal pattern in its output units, without needing any input. For example, in speech generation, for every word that one would like to artificially pronounce in a voice generator engine, one might want to train a separate RNN that sends a series of "vocalisation" commands to the voice engine. Or in humanoid robots, one wishes to have so-called *central pattern generator* (CPG) networks which send activation signals to the motors of the robot, such that a walking pattern is created (or a hand waving pattern, or staircase climbing, or any other motor pattern that is needed). Neurobiologists have reason to believe that such CPGs are implemented by RNN circuits in the spinal chord. If reading these lecture notes leaves you with the energy to watch a youtube video, click on `https://www.youtube.com/watch?v=DkS_Yw1ldD4` to see a simulated human with 61 degrees of freedom ("muscles") performing a sequence of motions that are generated by a pattern-generating RNN. Or in automated music composition, one wishes to have an RNN whose output is an improvised piece of music.

The training data for pattern generation tasks consists just of the desired output sequence, $S = (\mathbf{y}(n))_{n=1,\ldots,n_{\mathsf{max}}}$. The objective is to train a RNN such that its output signal $\hat{\mathbf{y}}(n)$ is "similar" to the training signal.

It is not obvious however how "similarity" is measured when there is no input in the testing phase. When the training signal $S = (\mathbf{y}(n))_{n=1,\ldots,n_{\mathsf{max}}}$ is chaotic or stochastic, the signal generated by the trained RNN $\hat{\mathbf{y}}(n)$ can not directly be compared with a "known correct" output. Figure 20 illustrates this difficulty. The teacher pattern is a chaotic signal generated by solving the Mackey-Glass equation (an equation describing the temporal

change of the amount of white blood cells in leukemic patients (Mackey and Glass, 1977); this has become a popular benchmark signal for RNN training). There is not general recipe for quantifying the accuracy of pattern generation learning. The only way is to define measurable features of the signal (for instance mean amplitude or probability to increase the signal by .5 within one timestep) and compare the values of these features measured in the original training signal vs. the generated ones.

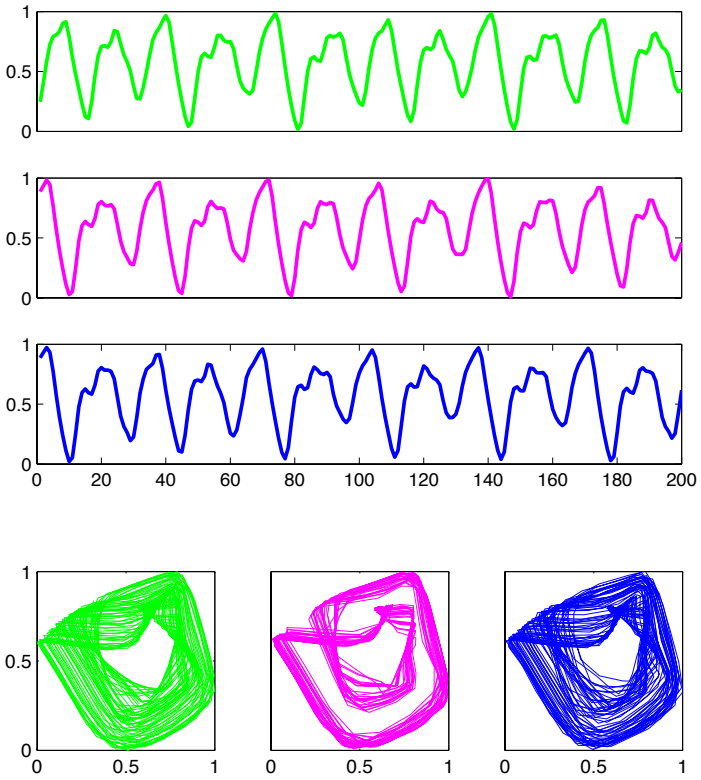

Figure 20: Illustrating the difficulty of assessing the accuracy of a learnt pattern generator. Green: Original (training) pattern. Magenta, blue: patterns generated by two trained RNNs. Top three panels: pattern signal against time. Bottom: time delay embeddings. The blue pattern generator matches the teacher better than the magenta one. This comes out visually clearly in the delay embedding plots, while it would be hard to judge the matching accuracy from the plots against time.

Pattern detection tasks. Sometimes one wants to identify certain patterns when they occur within an ongoing observed process. For example, in cardiological monitoring, one wants to detect moments when a certain pathological abnormality appears in a ECG signal; or in speech-based car control, the car's driver voice recognition system must identify moments when the driver says

"brake"; or the control system of a nuclear fission reactor must trigger an alarm action when the plasma shows signs of instability (a very tricky task where machine learning methods are considered a key for making nuclear fusion based energy possible — query "nuclear fusion plasma disruption neural network" on Google Scholar!).

The training data here often is prepared in the format $S = (\mathbf{x}(n), \mathbf{y}(n))_{n=1,\ldots,n_{\max}}$, where $\mathbf{x}(n)$ is a signal of measurements from the system in which one wants to identify a certain pattern, and $\mathbf{y}(n) \in \{0, 1\}$ is a binary indicator signal which is 0 when the pattern is not present and which jumps to 1 when the pattern occurs. This basic scheme comes in a number of variants. For instance, the indicator flag jumps to 1 only at the end of the pattern, because sometimes a pattern can be reliably identified only after it has come to its end — for instance, identifying the word "stop" in a speech signal can only be done when the "p" has been uttered, in order to not confuse this word with "stomach", "stolid", "stochastic", etc.). Or the indicator jumps to 1 a certain time *before* the target pattern occurs; this is the functionality needed for giving early warning of an impending system malfunction. A point in case is the early warning of an approaching epileptic seizure from EEG recordings (Figure 21) — query Google Scholar with "seizure prediction neural network" to see how much research has been spent on this theme, which cannot be considered solved even today; see Litt and Echauz (2002) for a very instructive survey.

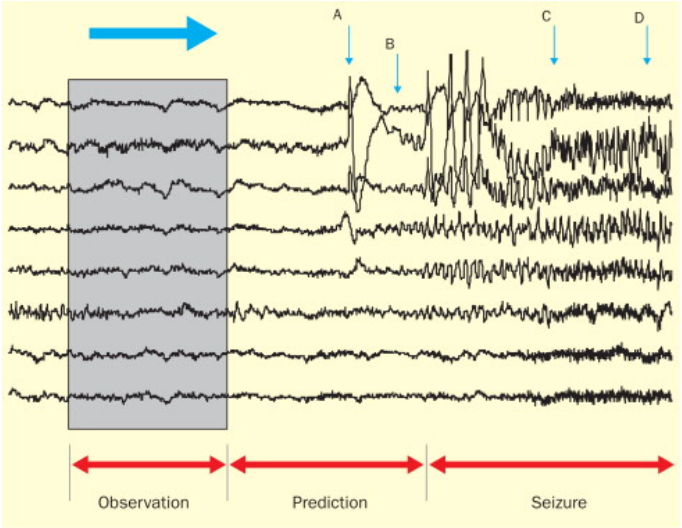

Figure 21: The seizure prediction scenario. The input is a multi-channel EEG recording. Image taken from Litt and Echauz (2002).

Timeseries prediction tasks. Sometimes one wants to look into the future. This task is particularly popular among people who want to get rich by

forecasting financial timeseries, and also among students searching for theses topics which will help them to become the kind of people who get rich. Other applications exist, too, for instance weather forecasting, windspeed prediction for wind energy farming (a theme with an extensive machine learning literature, check out Google Scholar on "wind speed prediction neural network"), or in epidemology where one wants to understand the spreading dynamics of a pandemic.

The concrete kind of training data and best suited RNN architectures vary greatly across application scenarios. Here I illustrate a scenario that involves highly nonstationary and stochastic timeseries: financial forecasting. Figure 22 gives an impression of the apparent difficulty of the task. In fact it is almost impossible to predict financial timeseries; the best one can hope for is to be a tiny little bit better than just repeating the last observed value and call it a prediction.

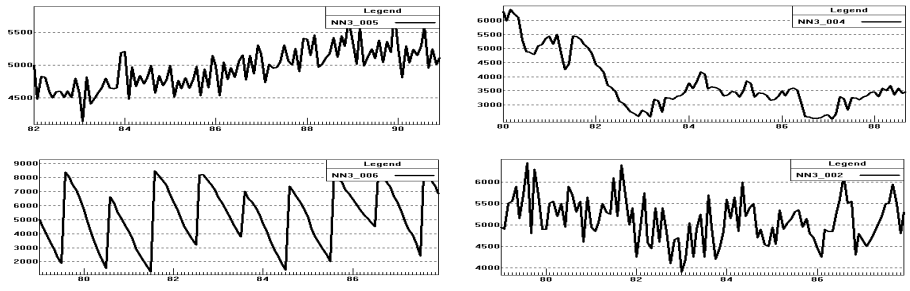

Figure 22: Four samples out of a financial forecasting competition dataset that comprised 111 such timeseries (`http://www.neural-forecasting-competition.com/NN3/`). The competition task was to train a neural network which could predict all of the 111 series for another 18 steps into the future.

A typical approach for timeseries prediction is to train a model that can predict just one step into the future, on the basis of past observations, and then feed back this prediction and iterate. Concretely this works as follows. The training data consists in a number of example timeseries

$$S = (\mathbf{u}^{(i)}(n), \mathbf{y}^{(i)}(n))_{n=1,\ldots,n^{(i)}_{\mathsf{max}}} = (d^{(i)}(n-1), d^{(i)}(n))_{n=2,\ldots,n^{(i)}_{\mathsf{max}}},$$

where i is the index of the training example and $d^{(i)}(n)$ is the n-th value in the i-th training timeseries (there were 111 such timeseries in the example task illustrated in Figure 22). The output timeseries desired from the network is thus equal to the input timeseries shifted one step ahead. An RNN is set up with a single input and a single output node. After training (when the network has learnt to predict all the training series as well as it can), the network is used to forecast the future by iterated one-step prediction in the following way:

Given: a new testing timeseries $d(1), \ldots, d(k)$.

Wanted: a prediction $d(k+1), \ldots, d(k+h)$ for a prediction horizon h.

Phase 1: initialization: The RNN is primed by running it with $d(1), \ldots, d(k)$ as input:

$$\mathbf{x}(n+1) = \sigma(\mathbf{W}\mathbf{x}(n) + \mathbf{W}^{\mathsf{in}}\, d(n) + \mathbf{b}) \quad (n = 1, \ldots, k).$$

The last obtained output $\hat{y}(k) = f(\mathbf{W}^{\mathsf{out}}\, \mathbf{x}(k))$ is saved. This should be a prediction of the next timeseries value, $\hat{y}(k) \approx d(k+1)$.

Phase 2: iterated prediction: Feed network outputs back as input, that is, compute for $n = k+1, \ldots, k+h$

$$\begin{aligned} \mathbf{x}(n) &= \sigma(\mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\mathsf{in}}\, \hat{y}(n-1) + \mathbf{b}), \\ \hat{y}(n) &= f(\mathbf{W}^{\mathsf{out}}\, \mathbf{x}(n)). \end{aligned}$$

For a serious effort to obtain good predictions, this basic scheme must be considerably refined (check out Ilies et al. (2007) for a case study). But that is a common machine learning wisdom: on any nontrivial learning problem, better and better results are achieved by investing more and more work, insight, and computing power.

System modeling tasks. Sometimes — in fact, often — one wants to simulate how a physical system responds to external perturbations or human control input. However, simulation models from first physical principles can be too expensive to be practical. For instance, while the Navier-Stokes equations can be used to exactly model the airflow around an aircraft, which would give a perfect model of how an aircraft responds to pilot commands, the numerical integration of these equations with the necessary precision is far too expensive to be useful for extensive explorations of an aircraft's behavior. In such situations one desires a computationally cheap *model* of the aircraft that faithfully replicates the flight responses of the aircraft to pilot steering input. When the target system behavior is very nonlinear and has memory effects, RNNs are a good candidate to yield simulation models. Figure 23 shows an example of an aircraft model realized by an RNN.

Other intensely investigated examples of modeling the responses of complex systems to input are oceanic flows, local weather dynamics (and more generally, all sorts of turbulent fluid dynamics), all sorts of industrial manufacturing pipelines, robot motion, power grids, etc., etc. — basically, every complex system that one wants to control, understand, or predict through simulations is a candidate for RNN modeling.

The training data here are of the generic format $S = (\mathbf{u}(n), \mathbf{y}(n))_{n=1,\ldots,n_{\mathsf{max}}}$ or $S = (\mathbf{u}^{(i)}(n), \mathbf{y}^{(i)}(n))_{n=1,\ldots,n^{(i)}_{\mathsf{max}}}$, depending of whether one deals with stationary or nonstationary system behavior. These data are obtained either

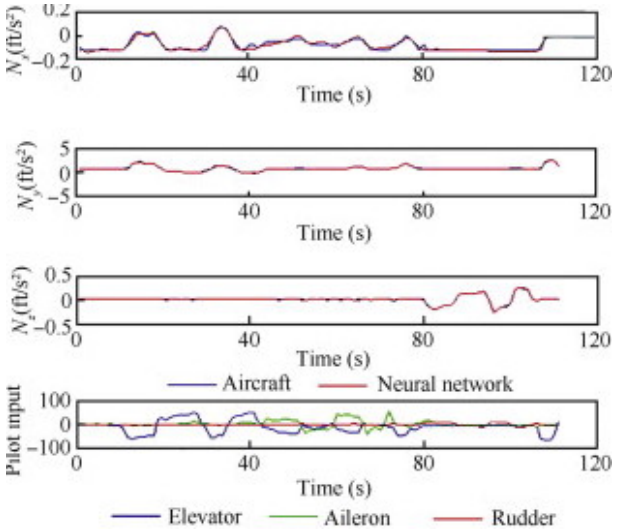

Figure 23: Modeling the response of an aircraft (top three panels showing the aircraft accelerations in the three spatial dimensions) to the pilot's action on elevator, aileron and rudder (bottom). The neural network predictions (red lines in upper three panels) show a good agreement with actual aircraft flight dynamics (blue lines, measured from actual flight experiments). Figure taken from Roudbari and Saghafi (2014).

from physical measurements, or from (expensive) first-principle simulations. In the aircraft modeling example, $\mathbf{u}(n)$ would be the three pilot command signalsand $\mathbf{y}(n)$ the measured accelerations of the aircraft.

This collection of RNN application scenarios is only indicative. There are many more, for instance tasks of data compression, denoising, channel equalization (cancelling echos in wireless antenna signals), nonlinear control, game playing — and soooo many more. Since the real world evolves in time, virtually all real-world data sources are temporal; no wonder that our brain, the best world modeling engine that we know, is recurrent.

4.2 Backpropagation through time

Training an RNN (e.g. of the basic format given in Equations 32 and 33) means to find weight matrices $\mathbf{W}, \mathbf{W}^{\mathsf{in}}, \mathbf{W}^{\mathsf{out}}$ and a bias vector $\mathbf{b}$ such that on test data $(\mathbf{u}^{\mathsf{test}}(n), \mathbf{y}^{\mathsf{test}}(n))_{n=1,\ldots,n_{\mathsf{max}}}$ the network output $\hat{\mathbf{y}}(n)$ is close to $\mathbf{y}^{\mathsf{test}}(n)$ in some suitably chosen loss metric.

How the loss function is defined depends on the specific scenario.

In stationary tasks, it is often sufficient to measure the mismatch between outputs at individual time steps. The loss function then is of the same form as in MLP training:

$$L : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}^{\geq 0}.$$

The risk is then defined as the expected loss $E[L(\hat{\mathbf{y}}(n), \mathbf{y}(n))]$ where $\hat{\mathbf{y}}(n)$ is the RNN output, $\mathbf{y}(n)$ is the correct output, and the expectation is taken over time. An example would be the quadratic loss $L(\hat{\mathbf{y}}(n), \mathbf{y}(n)) = \|\hat{\mathbf{y}}(n) - \mathbf{y}(n)\|^2$.

In nonstationary tasks we find a greater variation. For simplicity I only consider the case where all training and testing sequences have the same length k. Often one is interested in the network output only at the end of processing an input sequence. This happens, for instance, in single-word recognition tasks. In other cases one wants to collect some evidence throughout the entire sequence. Generally speaking, one often wants to weigh mismatches between network outputs $\hat{\mathbf{y}}(n)$ and targets $\mathbf{y}(n)$ differently at different times. A loss function then would look like

$$L(((\hat{\mathbf{y}}(n), \mathbf{y}(n))_{n=0,\ldots,k}) = \sum_{n=0,\ldots,k} a_n \, L_0(\hat{\mathbf{y}}(n), \mathbf{y}(n)),$$

where L_0 is a single-timepoint loss function and a_n are time-dependent weighting factors.

After a loss function has been decided, one carries out an RNN training project in the same way as for MLPs, by

1. installing an optimization algorithm which minimizes the empirical risk ("training error"), that is, an algorithm which solves the problem

$$\theta_{\mathsf{opt}} = \operatorname*{argmin}_{\theta} L(\hat{\mathbf{Y}}_{\theta}^{\mathsf{train}}, \mathbf{Y}^{\mathsf{train}}), \tag{34}$$

   where θ is the vector of all trainable parameters in the weight matrices $\mathbf{W}, \mathbf{W}^{\mathsf{in}}, \mathbf{W}^{\mathsf{out}}$ and the bias vector $\mathbf{b}$; $\hat{\mathbf{Y}}_{\theta}^{\mathsf{train}}$ is the output of the RNN parametrized by θ on training input; and $\mathbf{Y}^{\mathsf{train}}$ is the training output;

2. while at the same time attempting to ensure a good generalization by embedding the training error minimization runs in some outer optimization loop for finding a good level of regularization through cross-validation.

For MLPs, step 1. is done with the backpropagation algorithm. This algorithm crucially depends on the fact that the network topology is "feedforward", that is, there are no connection cycles.

For RNNs, finding weights which minimize the training error is more difficult. A number of algorithms are known which are based on different mathematical principles. The "best" of these algorithms does a true gradient descent on the performance surface and is suitable for online learning (adapting the RNN continuously while a never-ending stream of training data is arriving). This *real-time recurrent learning* (RTRL) algorithm has been introduced 30 years ago by Williams and Zipser (1989). Sadly, RTRL is too expensive for most practical exploits, having a cost of $O((K + L + M)^4)$ per update step. Yet, interest in RTRL has recently been rekindled in a subfield of deep learning called *continual learning*, where the objective is to find learning schemes that enable "life-long" training of

RNNs. If you are interested, a tutorial introduction to RTRL and other RNN training algorithms is Jaeger (2002).

Today, the minimization task (34) is almost always solved by the *backpropagation through time* (BPTT) algorithm. The idea is to "unfold in time" a recurrent neural network into a feedforward neural network, by assigning an identical copy of the RNN to each timestep and re-wiring the internal connections (in the weight matrix $\mathbf{W}$ such that they feed forward into the next copy in time. Figure 24 illustrates this idea.

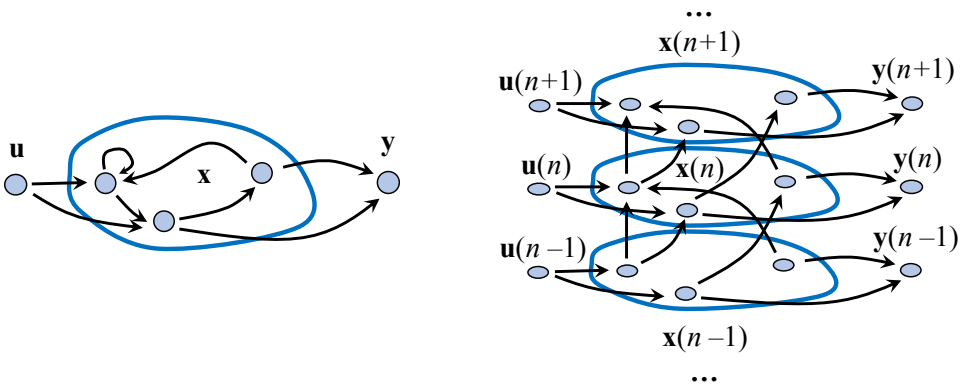

Figure 24: Basic idea of the BPTT algorithm: a recurrent net (left) is identically replicated for every time step and network-internal connections are rewired forward to the next copy (right).

The unfolded network is free of connection cycles, which makes it possible to train it with a special version of the backpropagation algorithm. The difference to the MLP version of backprop is that corresponding weights in different "time layers" must be identical. This is easy to accomodate and I do not present the adapted error backpropagation formalism here.

The BPTT scheme looks straightforward, but it comes with its own new problems.

The first problem is that the stack of temporal copies (right side in Figure24) must have a finite depth to enable the backpropagation algorithm. Call this depth h (for "horizon"). The unfolded RNN then will be a finite, cycle-free network which yields an input-to-output mapping from input sequences $\mathbf{u}(n), \ldots, \mathbf{u}(n+h-1)$ to output sequences $\hat{\mathbf{y}}(n), \ldots, \hat{\mathbf{y}}(n+h-1)$. If the training data has a length n_{max} of more than h steps, the training algorithm involves an averaging over gradient increments collected from, and averaged over, shifting the depth-h network forward through $n_{\mathsf{max}} - h + 2$ length-h segments the training data $((\mathbf{u}(n), \mathbf{y}(n))_{n=i,\ldots,i+h-1})_{i=0,\ldots,n_{\mathsf{max}}-h+1}$. This implies that any effects from input $\mathbf{u}(n)$ on the output that are delayed by more than h timesteps cannot be learnt. (*Note.* This is not entirely and always true and the story to be told here is involved and certainly not fully understood. If there are statistical dependencies between

the input signals at different times which span further than the training horizon h, some information about input $\mathbf{u}(n-h-g)$ from times earlier than $n-h+1$ is encoded in the inputs $\mathbf{u}(n-h+1), \ldots, \mathbf{u}(n)$ available to the learning algorithm, and effects from these encoded earlier on the current output can be learnt by BPTT. In the pure memory learning task that I will use below for demonstration purposes however there are no statistical dependencies between inputs at different times, and the memory limit of h is strict.) This is a significant problem in applications where there are input-output effects with arbitrary long delays — dynamics of that sort cannot be perfectly learnt by BPTT. Many real-world dynamical systems have this property. Examples are all turbulent fluid flows (where "eddies" of increasingly large sizes impose effects on increasingly large timescales), or human language (where what is said now in a text may have an influence on the text meaning at arbitrary distant points in the future).

For all we know, the human brain solves this multi-timescale problem by exploiting a host of different physiological and architectural mechanisms which yield a large compendium of different memory mechanisms, supported by different neuronal circuits and physiological effects, ranging from ultra-short term memories in the millisecond range, over a spectrum of short-term and working memory, to long-term memory mechanisms that operate in the range of the human lifetime. Neuroscience, cognitive science, AI and machine learning so far has only given us a very partial understanding of such memory subsystem cascades. This is an active, interdisciplinary research area in which I am personally involved (check out the EU project MeM-Scales, "Memory technologies with multi-scale time constants for neuromorphic architectures", `https://memscales.eu/`).

The second problem is known as the *vanishing gradient* problem. It also occurs in a mirror version as *exploding gradient* problem. This problem comes to the surface when error gradients are back-propagated through many layers. The deeper a network, the more serious the problem. Unfolding RNNs in time tends to end up with particularly many time-slice layers (up to several hundreds), making them much deeper than commonly used deep MLPs. Thus the vanishing gradients are particularly disruptive in RNN training.

I will demonstrate the mathematical nature of vanishing gradients with a super-simple RNN. It has a single input unit with activations $u(n)$, a single output unit with activations $y(n)$, and the main RNN itself consists of a single linear unit with activation $x(n)$, and no bias. This leads to the following embryonic RNN equations:

$$\begin{aligned} x(n+1) &= w\, x(n) + w^{\mathsf{in}}\, u(n+1), \\ y(n) &= w^{\mathsf{out}}\, x(n). \end{aligned} \tag{35}$$

The task is a pure memory task. The training data is given by $(u(n), y(n))_{n=0,\ldots,n_{\mathsf{max}}} = (u(n), u(n-h))_{n=0,\ldots,n_{\mathsf{max}}}$, that is, the desired output $y(n)$ is the input from h timesteps earlier. The input signal $u(n)$ is an i.i.d. signal sampled from the uniform distribution on $[-1, 1]$, that is, at every time n the signal value $u(n)$ is freshly

randomly chosen from the interval $[-1, 1]$.

We now want to find values for $w, w^{\text{in}}, w^{\text{out}}$ which minimize the quadratic loss, by running a BPTT gradient descent with an unfolded version of our embryonic RNN. We unfold the RNN to depth h, that is we create $h + 1$ time-slice copies, which for this task is the minimal needed depth.

For this simple RNN all gradients can be computed easily without invoking the backpropagation trick. During the gradient descent, the unfolded network is aligned with length-h windows of the input/output teacher signal, from times n to $n + h$. The only task-relevant signal propagation pathway goes from the teacher input $u(n)$ at time n to the teacher output $y(n + h) = u(n)$ at time $n + h$. All other input-unit to output-unit pathways in the unfolded network lead from input signals that are uncorrelated with the output signal at the respective time, and their contributions to the gradient will average to zero as the unfolded network is moved forward through successive time windows $[n, n + h]$ of the training data. The only gradient component that may not average to zero is the one on the pathway from the earliest input node to the latest output node in the unfolded network (see Figure 25).

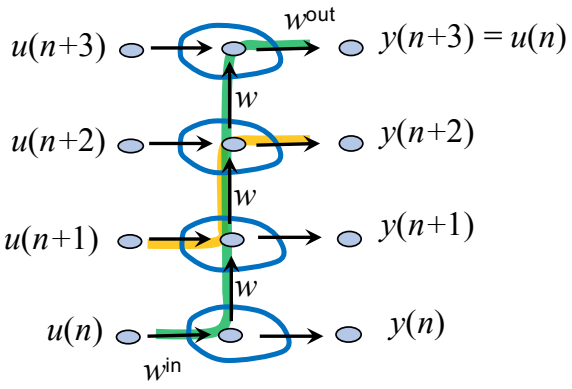

Figure 25: A single-unit embryo RNN for demonstrating the vanishing/exploding gradient. Training this network of depth $h = 3$ on the pure memory task with memory depth h, the only input-node to output-node pathway that leads to a gradient contribution which does not average out to zero is along the green line. The orange pathway, for instance, connects an input signal $u(n+1)$ to a teacher output signal $y(n+2)$. Since the two are uncorrelated and the entire network is linear, this pathway gives a gradient component which averages out to zero over time.

For the quadratic loss, this gradient component of the loss with respect to the

weigth w is equal to

$$\begin{aligned}\frac{\partial}{\partial w}(y(n+h)-\hat{y}(n+h))^2 &= \mathsf{OGCs}+\frac{\partial}{\partial w}(y(n+h)-u(n)\,w^{\mathsf{in}}\,w^h\,w^{\mathsf{out}})^2\\ &= 2\,(y(n+h)-\hat{y}(n+h))\,u(n)\,w^{\mathsf{in}}\,w^{\mathsf{out}}\,h\,w^{h-1}\end{aligned}$$

where $y(n+h)$ is the teacher output, $\hat{y}(n+h)$ the network output, and OGCs are all the Other Gradient Components stemming from other input-output pathways and which give zero contributions in temporal averaging.

The critical term in this gradient is w^{h-1}. I think you can smell the danger: if $w < 1$ and h is large, this term will shrink toward zero — the gradient vanishes; and when $w > 1$, the gradient will explode with the depth h.

In fully grown-up RNNs the analysis of vanishing/exploding gradients is not so simple, but the basic mechanism is the same: since all time-slices of the unfolded RNN are identical copies, gradient components arising from pathways that span large temporal horizons are repeatedly either quenched or expanded at every timestep. If the learning task includes the exploitation of long-term delayed input-to-output effects (long memory), the vanishing / exploding gradient problem will make it practically impossible to let the network find and encode these long-term effects during learning.

You can find a more detailed presentation of the vanishing gradient problem in *the* deep learning textbook Goodfellow et al. (2016), Section 10.7. The conclusion drawn by the authors at the end of that section is *"... as we increase the span of the dependencies that need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful training of a traditional RNN via SGD [= stochastic gradient descent] rapidly reaching 0 for sequences of only length 10 or 20. [...] the problem of learning long-term dependencies remains one of the main challenges in deep learning."*

4.3 LSTM networks

Long short-term memory (LSTM) networks are today the best answer to both the problem of memory over several timescales, and the problem of vanishing gradients.

First a note on the name. "Long short-term..." seems like a contradiction in terms. This terminology has the following background. In neuroscience, different neural mechanisms have been identified (or hypothesized) to take care of memorizing on different timescales. There is a fine-grained terminology which we will not further unfold here. But there is also a very coarse summary distinction between "long-term memory" (LTM) and "short-term memory" (STM). LTM is physiologically realized (according to current neuroscience dogma) by changes in synaptic connection strenghts. The name of your father and mother, for instance, is encoded in your LTM system by very stable synaptic connection patterns which have been burnt into your brain in childhood — according to textbook dogma. (In

parentheses I point out that dogmas in any science may be swept away by revolutionary ideas and discoveries; in fact it becomes increasingly clear that the idea of long-term persistent synaptic strengths is a gross simplification (Zenke et al., 2014; Castello-Waldow et al., 2019).) In contrast, "short-term memory" is a lump naming for any memory mechanism where information is retained for limited periods of time by neuronal activity patterns, without inducing synaptic weight changes. Thus, speaking of "long short-term memory" means memory mechanisms where information is preserved over limited, but possibly rather long times, on the basis of neuronal activation dynamics in RNNs, without synaptic weight adaptation.

In machine learning, "LSTM networks" refers to a specific RNN architecture introduced in stages in the pre-deep learning era by Hochreiter, Schmidhuber and Gers (Hochreiter, 1991; Hochreiter and Schmidhuber, 1997; Gers et al., 2000). The original motivation was to attack the vanishing gradient problem. It soon turned out that the proposed solution, now called LSTM networks, at the same time helped to train RNNs to cope with multiple-timescale tasks. Today, LSTM networks and their descendants, RNNs with *gated recurrent units* (GRU networks, Cho et al. (2014)) are the dominating sort of RNNs used in deep learning.

The basic ideas behind LSTM networks are intuitive, but I am not aware of a transparent mathematical analysis of why and how, exactly, LSTM networks do function so well as they do. The explanations given in Section 10 in the deep learning "bible" Goodfellow et al. (2016) are kind of handwaving. In the following parts of this section I will try to explain the LSTM mechanism through a simplistic worked-out baby LSTM which extends the embryo RNN example from Figure 25. The extended version is shown in Figure 26.

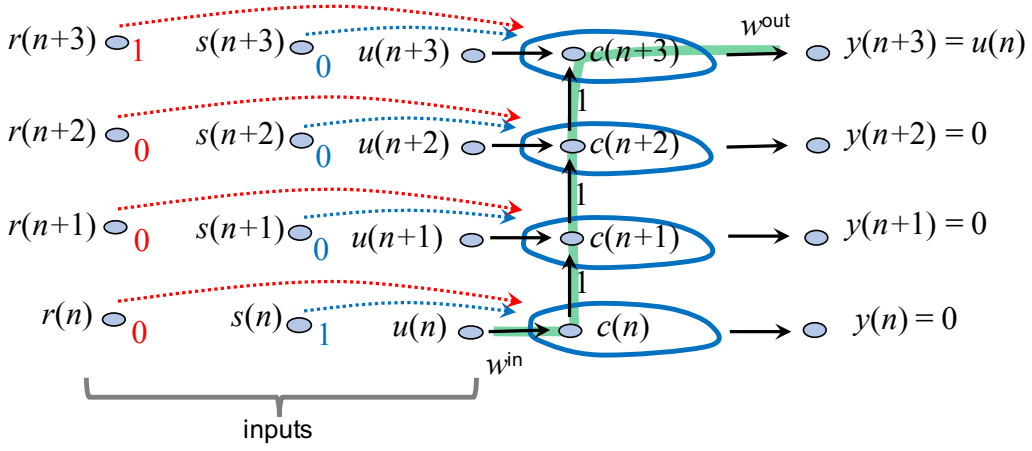

Figure 26: A baby LSTM. For explanation see text.

The baby LSTM set-up involves three input signals $u(n) \in \mathbb{R}, s(n) \in \{0,1\}, r(n) \in \{0,1\}$. The input $u(n)$ plays the same role as in the embryo example before: the task for the network is to identically recall $u(n)$ at a later time $n+h$ in the output, that is, $y(n+h)$ should be equal to $u(n)$. The binary inputs $s(n), r(n)$ (for "store"

and "read") are auxiliary inputs to control this memorizing task. Both are equal to 0 most of the time. At rare times the store signal jumps to $s(n) = 1$. Then, the network should "store" the current value of $u(n)$. At the future time $n + h$, the "read" input jumps to $r(n + h) = 1$, and the output should read the stored value $u(n)$. Figure 26 shows a case where $h = 3$. At all other times $r(n)$ is zero, and the output should be 0 too.

LSTM networks have "neurons" that are quite unlike the neurons that we have seen so far. In the LSTM literature these neurons are called *(memory) cells*, and such a memory cell is embedded in an intricate control circuitry that has little in common with biological systems. The memory cell together with its surrounding control circuitry is called a *memory block* in the core historical paper Gers and Schmidhuber (2001). In later literature by other authors, often the entire circuitry is often called an LSTM "cell". I will stick to the terminology of Gers and Schmidhuber (2001) and use the term "memory cell" only for the central, memory-preserving neuron, and use the "memory block" for the entire circuit.

I will give full equations for LSTM memory blocks later and for the time being only present a simplified set of equations for our baby LSTM network, which is made from a single (simplified) memory block.

The memory block has a memory cell with activation state $c(n) \in \mathbb{R}$ at time n. The update equation for $c(n)$ and the output equation are

$$\begin{aligned} c(n+1) &= 1 \cdot c(n) + s(n) \cdot (w^{\mathsf{in}}\, u(n) - c(n)), && (36) \\ y(n) &= r(n) \cdot w^{\mathsf{out}}\, c(n). && (37) \end{aligned}$$

The "store" signal $s(n)$ acts as a *multiplicative gate* which only lets the input $u(n)$ affect the memory cell at the rare times when $s(n) = 1$. At such times, the memory cell state is set to $w^{\mathsf{in}}\, u(n)$ at the next time step.

Similarly, the "read" signal $r(n)$ only at times when $r(n) = 1$ allows the output unit to read out from the memory cell an output value of $w^{\mathsf{out}}\, c(n)$; at other times the output reading is zero.

Furthermore, as long as there is no "store" input signal, that is, as long as $s(n) = 0$, the memory cell retains it previous value $c(n + 1) = 1 \cdot c(n)$.

The only two trainable weights in this system are w^{in} and w^{out}. If one uses again the quadratic loss, one will find that the squared error at times when $r(n) = 0$ is zero (because the network output and the teacher output are both zero, hence no error), and at times where $r(n) = 1$ is

$$\begin{aligned} \varepsilon^2(n) &= (y(n) - r(n)\, w^{\mathsf{out}}\, c(n))^2 \\ &= (u(n-h) - w^{\mathsf{out}}\, w^{\mathsf{in}}\, u(n-h))^2 \\ &= u(n-h)^2\, (1 - w^{\mathsf{out}}\, w^{\mathsf{in}})^2. \end{aligned}$$

At these times, the gradient of the error with respect to the two trainable weights

is

$$\begin{aligned}\frac{\partial}{\partial w^{\mathsf{in}}}\,\varepsilon^2(n) &= -2\,u(n-h)^2\,(1-w^{\mathsf{out}}\,w^{\mathsf{in}})\,w^{\mathsf{out}}\\ \frac{\partial}{\partial w^{\mathsf{out}}}\,\varepsilon^2(n) &= -2\,u(n-h)^2\,(1-w^{\mathsf{out}}\,w^{\mathsf{in}})\,w^{\mathsf{in}}\end{aligned}$$

Obviously, gradient descent along this gradient will lead to a point where $w^{\mathsf{out}}\,w^{\mathsf{in}} = 1$. The important insight here is that this gradient does not vanish or explode when h gets larger. In fact, the gradient is independent of h.

The key to this independence of the error gradient from the memory depth h is that, as long as $s(n) = 0$, the activation state $c(n)$ is identically copied to the next time step by virtue of the update rule $c(n+1) = 1 \cdot c(n) + 0 \cdot \ldots$ in (36). The weight w that we had in the embryo RNN (35), which made gradients vanish or explode if not equal to 1, is fixed to be equal to 1 in (36).

LSTMs are, by and large, a sophisticated extension on this this trick to carry memory information forward in time through a linear memory cell update $c(n+1) = w\,c(n)$, where w is equal or close to 1.

An LSTM *network* is an RNN which consists of several memory blocks and possibly other, "normal" RNN units. The memory blocks and normal units may receive external inputs and/or recurrent input from other blocks/units within the network.

LSTM memory blocks are made from 5 specialized neurons which have different update equations. The central neuron is the memory cell with state $c(n)$. The remaining four neurons are

an input neuron with state $u(n)$. This corresponds to the input neuron $u(n)$ shown in our baby LSTM. The input neuron may receive external input (as in the baby LSTM) and/or input from other blocks or normal units within the network. The input neuron is a "normal" unit with an update function

$$u(n+1) = f(\mathbf{W}^u\,\mathbf{x}^u(n) + b^u),$$

where f is any kind of sigmoid (tanh, logistic sigmoid or rectifier function), $\mathbf{x}^u(n)$ is a vector of signals composed of external inputs and/or outputs from other blocks or normal neurons in the network, $\mathbf{W}^b$ is a vector of input weights, and b^u is a bias vector;

an input gate neuron with state $g^{\mathsf{input}}(n)$. In our baby LSTM, the role of the input gate neuron was played by the external "store" input $s(n)$. The input gating neuron has the update equation

$$g^{\mathsf{input}}(n+1) = \sigma(\mathbf{W}^{\mathsf{g}^{\mathsf{input}}}\,\mathbf{x}^{\mathsf{g}^{\mathsf{input}}}(n) + b^{\mathsf{g}^{\mathsf{input}}}),$$

where σ is always the logistic sigmoid and $\mathbf{x}^{\mathsf{g}^{\mathsf{input}}}(n)$ is is a vector of signals composed of external inputs and/or outputs from other blocks or normal neurons in the network;

an output gate neuron with state $g^{\mathsf{output}}(n)$. In our baby LSTM, the role of the output gate neuron was played by the external "read" input $r(n)$. The output gate neuron is similar to the input gate neuron and has the update equation

$$g^{\mathsf{output}}(n+1) = \sigma(\mathbf{W}^{\mathsf{g}^{\mathsf{output}}} \mathbf{x}^{\mathsf{g}^{\mathsf{output}}}(n) + b^{\mathsf{g}^{\mathsf{output}}}),$$

where again σ is always the logistic sigmoid and $\mathbf{x}^{\mathsf{g}^{\mathsf{output}}}(n)$ is is a vector of signals composed of external inputs and/or outputs from other blocks or normal neurons in the network; and

a forget gate neuron with state $g^{\mathsf{forget}}(n)$. In our baby LSTM, the role of the forget gate neuron was played by the constant 1 factor in (36). The forget gate neuron is again similar to the input gate neuron and has the update equation

$$g^{\mathsf{forget}}(n+1) = \sigma(\mathbf{W}^{\mathsf{g}^{\mathsf{forget}}} \mathbf{x}^{\mathsf{g}^{\mathsf{forget}}}(n) + b^{\mathsf{g}^{\mathsf{forget}}}),$$

where again σ is always the logistic sigmoid and $\mathbf{x}^{\mathsf{g}^{\mathsf{forget}}}(n)$ is is a vector of signals composed of external inputs and/or outputs from other blocks or normal neurons in the network.

The central element in a memory block is the memory cell $c(n)$. Its update equation, which corresponds to (36), is

$$c(n+1) = g^{\mathsf{forget}}(n+1) \cdot c(n) + g^{\mathsf{input}}(n+1) \cdot u(n+1). \tag{38}$$

The main extensions compared to the baby LSTM are that the memory cell does not necessarily preserve its previous value with a factor of 1, but may "leak" some of it by the multiplication $c(n+1) = g^{\mathsf{forget}}(n+1) \cdot c(n) \ldots$ with the forget neuron value.

The output $y(n)$ of a memory block is (analog to (37)) is given by

$$y(n) = g^{\mathsf{output}}(n) \cdot c(n). \tag{39}$$

This output may be fed as input to other memory blocks and "normal" units in the LSTM network.

Notice that the three gate units must have the logistic sigmoid as their squashing function. This ensures that the gating values $g^{\mathsf{input}}(n), g^{\mathsf{output}}(n), g^{\mathsf{forget}}(n)$ all range between 0 and 1. With a value of 1, the "gate" is fully opened, with a value of 0 it is completely shut.

Figure 27 illustrates the LSTM block that I described.

LSTM networks are trained with BPTT. All the weights $\mathbf{W}^u, \mathbf{W}^{\mathsf{g}^{\mathsf{input}}}, \mathbf{W}^{\mathsf{g}^{\mathsf{output}}}, \mathbf{W}^{\mathsf{g}^{\mathsf{forget}}}$ and biases in all memory blocks of an LSTM network are trainable.

Numerous variations of this memory block architecture are in use. For instance, Gers and Schmidhuber (2001) describe memory blocks that may contain several memory cells, and they also admit the signals $c(n)$ as inputs to the gating neurons of the same block. Significantly simplified versions of LSTM blocks, called "gated recurrent units", are also explored and used. A discussion of these variations is given in Goodfellow et al. (2016), Section 10.10. There appear to exist no clear universally best winners among the large assortment of possible variations.

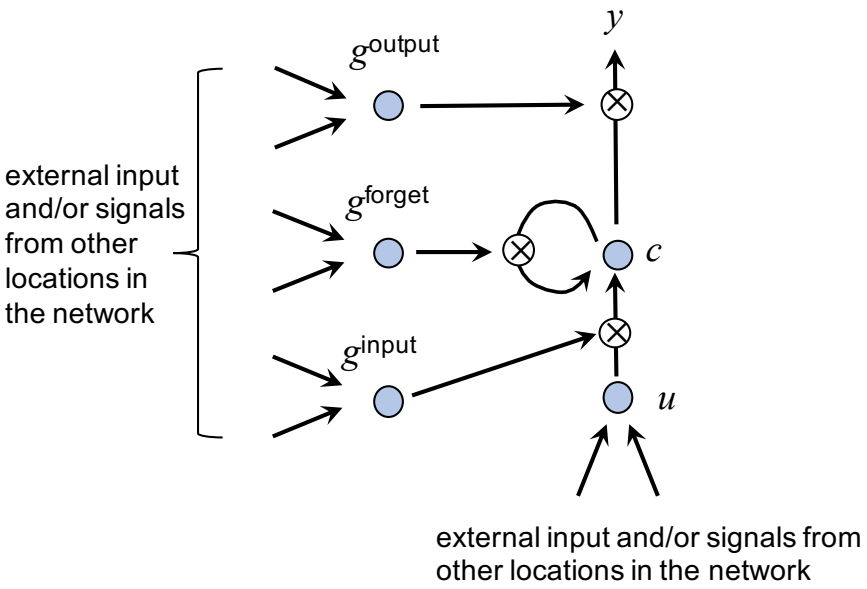

Figure 27: A standard LSTM block layout. For explanation see text.

5 Hopfield networks

An obvious core functionality of biological brains is long-term memory (LTM). You can remember your own name as well as details from your parent's house as well as impressions from the beach of your last summer vacation — and more. In fact you carry with you a rich repository of memories from your previous life's experiences — you carry with you all the things that make *you*. The more you start thinking seriously about all the things that you "know" (= have in your memory), the more you will find that you know *very very much* indeed. How can your little brain store all of that?

The question of human long-term memory has kept psychologists, cognitive scientists, neuroscientists, psychiatrists, AI researchers and even physicists busy since the beginnings of each of the respective branch of science. It is one of the most heavily thought-about scientific riddles, and it is far from being solved.

One thing is clear: memorizing is different from storing items in a shelf. The car of your parents, which you can very well remember in much detail, is not placed as a little model car somewhere in your brain. A neurosurgeon would not be able to find it after opening your skull. Here are some of the research riddles (non satisfactorily cleared) that encompass long-term memory:

- How is LTM distinguished from other forms of memory, like short-term memory, long-short-term memory, working memory?
- Is there a unique dedicated mechanism in the human brain for LTM, or is it a complex system with many functional modules? Connected with this

question, what are commonalities / differences between LTMs in humans vs. dogs vs. frogs vs. honeybees vs. little worms? (They all have their peculiar LTM capacities.)

- To the extent that long-term memory "items" are "stable", there must be *something* physical in brains that is stably persisting through the tides of time. For decades, the dogma in neuroscience (and machine learning, for that matter) had it that LTM memory traces are physically realized through synaptic weights. Memorizing something for good means that some synaptic weights are set and not changed thereafter. Well, this simple dogma is dissolving in these days. It can't be that simple:
  - Human memories change over time. A classical, super fascinating experimental psychology study (Bartlett, 1932) (21K citations) reveals that over the decades of a human life, what one thinks one had clearly "memorized" for good — *changes*, even dramatically. My wife and I are both aware of this and sometimes marvel that, when she is superconvinced that the skirt her sister wore hat at our wedding was blue and long, and I am dead sure that it was red and short, we look at photos and find it was yellow and of medium length.
  - On the microanatomical level, it is becoming clear that neural synapses are incessantly changing, even becoming deleted or re-growing, at amazingly high rates (Castello-Waldow et al., 2019).
  - A long-standing conundrum in artificial NN / machine learning research is that if an MLP is first successfully trained on some task A, leading to a specific formation of synaptic weights, and then subsequently trained again on another task B, it will learn task B but in the process modify synaptic weights such that task A is no longer mastered. This is called the problem of *catastrophic forgetting*. Attempts to overcome it have only very recently become halfway successful, forming the active field of *continual learning* within deep learning (review: Parisi et al. (2018)). Again, what is being found out in this field is incompatible with the dogma of basing LTM on fixing stable synaptic weights.
- If your memory of a car isn't like putting a little toy car on your brain shelves, the question is how are memory items neurally *encoded*? If you think of your grandmother, does this mean that a specific *grandmother neuron* is activated? Like always in neuroscience, there is solid evidence in support of this hypothesis, and equally solid evidence against it (`https://en.wikipedia.org/wiki/Grandmother_cell`). An alternative view is that memories are encoded in a *distributed* way: thinking of your grandmother is the effect of a complex neural activation pattern that involves large portions of your brain.

- Given some long-term encoding scheme, by which physiological/anatomical processes are memory traces actually *formed*? In engineering terms: what is the "writing" mechanism? In machine learning, one does the "writing" by the backprop algorithm which nobody believes is biologically feasible (plus, it suffers from catastrophic forgetting). The human brain might need ... sleep! in order to transform memory items, which are provisionally recorded during the day using non-persistent mechanisms (presumably in the hippocampus), into persistent traces in LTM.

- Ok., assuming it is understood how memory items are encoded, and how they are written, how are they *retrieved* when you "recall" them? Recalling leads to two subproblems: *addressing* and *decoding*:

  - How to you mentally "point to" memory items? how do you "know" what to retrieve? In a digital computer, addressing is done by pointers to physical memory registers. But equivalents of C pointers are unlikely to be implemented in biological brains. Instead, it seems more plausible that brains use *content addressing*. This means that in order to access what you have stored about your grandmother, you need to start with some fragments of your grandmother memories — for instance, her name, or you think of her house, or of a family gathering. This way of thinking about mental addressing is known as *associative memory* models. Building theories about how associative memories function has been a mainstream activity in neural networks research for decades. It has led to a mountain of models documented in a wide-spanning literature.
  - The decoding problem is the twin of the encoding problem.

- Finally, it is unclear what a "memory item" is. When you remember your grandmother, you will be recalling different aspects of her in different recall contexts. Apparently it is quite a simplification to think of well-circumscribed memory "items". This immediately leads to the highly disputed problem of the nature of what *concepts* are and how they are neurally represented. Again, a vast literature on this topic exists in psychology, AI and philosophy, and no consensus is in sight. A most instructive, influential, and readable book is Lakoff (1987) (30K Google Scholar cites), and it has a nice title too: "Women, fire and dangerous things" The title outlines the semantic contents of a single concept in an Australian Aboriginal culture and language.

This bundle of riddles is obviously of great interest for cognitive neuroscience. But it is also of central importance for today's research in computer science and AI:

- In deep learning, the problem of continual learning, i.e. the problem of organizing the incremental growth of the representational repertoire of an artificial neural network, is not solved. The magnificent achievements of deep

learning come in the form of networks specialized each on a specific task. A face recognition network cannot be further trained to also recognize cars, let alone to control a robot arm. Partial solutions are emerging in these days, and continual learning is an active topic in the APS group here in the AI institute (`https://scholar.google.com/citations?user=VFr_XuYAAAAJ`, He et al. (2019)).

- In an emerging field called (among many other namings that are floating around) *neuromorphic computing*, one tries to design novel types of computing microchips which are inspired by neural networks. I will give an introduction to this field in the last session of this course. One of the main goals for this line of research is to enable *in-memory computing*. The idea is that in biological brains and many artificial neural networks, there is no distinction between a "processor" (CPU) and a "memory" (RAM). All computing should be done directly at the locations and encoding level of the memory. This would obviate the infamous von-Neumann bottleneck, that is the read/write channel between the CPU and the RAM. This bottleneck eats up most of the time and energy in conventional digital computing technologies and enforces a serial execution of computational operations. The promise of in-memory computing is to enable a thoroughly parallel way of computing directly on the hardware level of memory traces, thereby saving orders of magnitude of energy and time. So far, a generally useful way for in-memory computing has not been found.

Among the many and diverse models of neural long-term memories, there is one which stands out: Hopfield networks (Hopfield (1982) – 21K Google Scholar cites. John J. Hopfield was born in 1933 and is still scientifically active - his homepage is `https://pni.princeton.edu/john-hopfield`). It is simple, mathematically transparent, reasonably powerful, deeply analyzed and almost completely understood. In machine learning it spun off an entire family of *energy-based* neural network models, among which the *Boltzmann machine*, which in turn was instrumental in getting deep learning off the ground. In the cognitive neurosciences it served and still serves as a foundational reference model for associative neural memories.

5.1 An energy-based associative memory

Before I start explaining the formalism and learning algorithm for Hopfield networks (HNs), I will outline what it is meant to achieve.

In a nutshell, a HN can be trained to store a finite number of *patterns* and let them become retrieved by content-adressing through an *auto-association* process. In the context of HNs, a pattern is a binary vector $\xi \in \{-1, 1\}^L$. It is also possible to use 0-1 valued binary vectors, but notation is a little simpler with values $\{-1, 1\}$. In textbook introductions to HNs, one mostly uses patterns ξ whose entries are

arranged in a 2-dimensional "pixel" array, which allows one to display a pattern graphically as a black-and-white pixel image. Given a finite and not too large number of training patterns $\xi_1, \ldots, \xi_N$, they can be encoded ("stored", "learnt") in a Hopfield network, which is a recurrent neural network of a very special design. In the context of HNs we also call these training patterns *fundamental memories* (after they have been stored).

In order to *recall* one of the stored ξ_i, the HN is presented with a cue pattern $\mathbf{u} \in \{-1, 1\}^L$. The cue pattern must have the same dimension as the stored patterns. The cue is typically a corrupted version of one of the stored patterns. "Corrupted" means that the cue pattern agrees with the corresponding stored pattern in some pixels and is different on other pixels. The differences between the cue and the targetted stored pattern can be very substantial. Figure 28 shows two examples.

Figure 28: Cueing a HN by corrupted patterns leads to a retrieval of the uncorrupted, stored pattern. Left: corruption by noise (leading to *pattern restauration* upon retrieval). Right: corruption by omission (leading to *pattern completion* functionality). Images taken from Hertz et al. (1991).

This functionality is quite suggestive of some aspects of human LTM. We are also able to recall memory items from corrupted cues, where "corruption" can mean many things, in particular that only some fragments of the memorized items are needed to recall its full content. Furthermore, according to some theories (very much disputed though – but all cognitive psychology theories are very much disputed) in cognitive psychology, humans represent and encode conceptual items in the form of *prototypes*, that is, "clean", "characteristic" examples. According to protype theories of conceptual representations (introductions: Part III in Evans et al. (2007)), if you see a house or read the word "house", you would mentally access a representation of an "ideal" house. As always in the cognitive and neurosciences there is substantial evidence in favor of prototype theories (if asked repeatedly to draw a house, your drawings will all look similar, and similar to the house drawings of other members of your social community), and much evidence against it (you can also be asked to draw a large house, an old house, Bilbo Baggin's house, or a termite's palatial nest — and all these drawings will look different). These cognitive prototypes would correspond to the fundamental

memories in HNs.

Here are the main design principles for the associative pattern recall functionality in Hopfield nets:

1. Every state $\mathbf{x}$ of the HN corresponds to one possible pattern $\xi \in \{-1, 1\}^L$ and vice versa: there is a bijection between the possible states and all possible patterns. In fact, a HN made to process L-dimensional patterns has L neurons, and each neuron can have an activation from the binary set $\{-1, 1\}$. We can therefore identify HN states with patterns. We will use this convention in the remainder of this section and often write ξ for patterns as well as for HN activation state vectors, using the notations ξ and $\mathbf{x}$ interchangeably. The first notation is more suggestive when we discuss the interpretation of states as patterns, the second is more intuitive when we discuss the computational mechanics inside a HN, because $\mathbf{x}$ is the natural notation for the state of a dynamical system.

2. Each state $\mathbf{x}$ of HN has a well-defined *energy* $E(\mathbf{x}) \in \mathbb{R}$. Note that negative energies are possible, unlike in physics. We will nonetheless see later in this course that the connections between HNs and the concept of energy in physics can be made precise (Hopfield is a theoretical physicist).

3. During the storage learning process, every pattern from the learning set becomes associated with a locally minimal energy. This gives (after learning) and "energy landscape" over the space $\{-1, 1\}^L$ of all binary patterns, where the fundamental memories are placed at local minima. See Figure 29.

4. Recall is started by presenting a (corrupted) input pattern $\mathbf{u}$, which is set to be the initial state of a state trajectory which evolves according to the recurrent dynamics of the HN. The trajectory leads through a state sequence which at every step reduces the energy and thereby necessarily ends in a local minimum — the fundamental memory is retrieved.

Given a HN in which a set $\xi_1, \ldots, \xi_N$ of patterns has been stored, and given an input cue pattern $\mathbf{u}$, the discrete-time state update dynamics of the HN will lead to a sequence of patterns/states $\mathbf{u} = \mathbf{x}(0), \mathbf{x}(1), \ldots, \mathbf{x}(m) = \mathbf{x}(m+1) = \mathbf{x}(m+2) = \ldots = \xi_i$ which at every update yields a state with smaller or same but never larger energy, until at some time m (which depends on the initial cue) the sequence becomes stationary — no further energy reduction is possible, one has landed at a local energy minimum, that is, at a fundamental memory ξ_i. Since the pattern/state space $\{-1, 1\}^L$ is finite, this must happen after some finite time with probability 1. Figure 30 (left) shows such a pattern sequence.

Under the HN state update dynamics, the fundamental memories act as point attractors. The set of all cues $\mathbf{u}$ that are ultimately attracted by a fundamental memory ξ_i is the *basin of attraction* of ξ_i. One may say (and AI theoreticians and cognitive scientists indeed say) that all the patterns in the basin of attraction of ξ_i

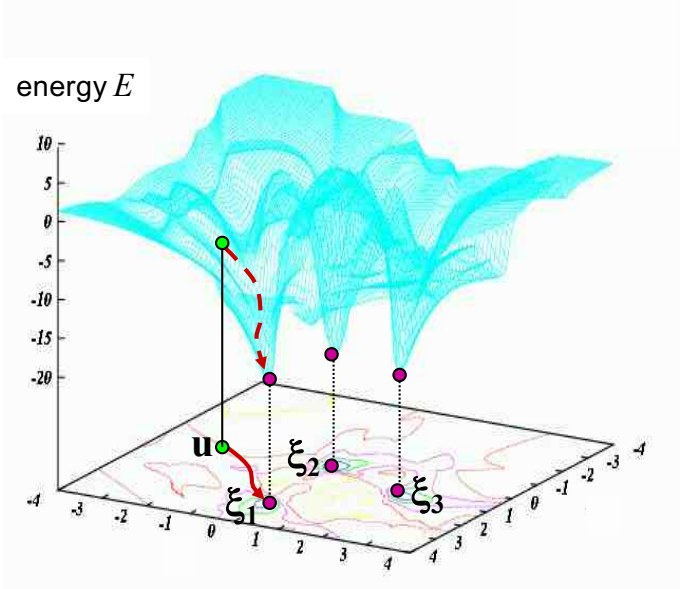

Figure 29: Schematic of energy landscape over the pattern/state space of a HN. The pattern space is here artificially rendered as a 2-dimensional space; this is only for visualization as the pattern space $\{-1, 1\}^L$ does not have such a two-dimensional topology. Fundamental memories ξ_1, ξ_2, ξ_3 mark local minima of the landscape. Upon input of a (corrupted) pattern $\mathbf{u}$, the recurrent dynamics of the HN lead through a state sequence which at every step reduces the energy, until the nearest local minimum is reached. Image retrieved from www.ift.uib.no/∼antonych/protein.html (no longer accessible).

are *instances* of the *category* (or *concept* or *class*) represented by ξ_i. Hopfield networks thus give a specific formal model of cognitive conceptual spaces. According to the HN model, concepts are characterized by "prototypes" ξ_i, and instances $\mathbf{u}$ of a concept are more or less similar to the prototype according to the number of steps it takes to converge from the instance $\mathbf{u}$ to the prototype.

5.2 HN: formal model

A HN is a recurrent neural network without input and output units. If the task is to store and recall L-dimensional patterns, the HN will itself have exactly L neurons which serve equally as input, "internal", and output units.

The formal definition of an L-dimensional HN is simple: it is fully specified by an $L \times L$ sized, real-valued, symmetric weight matrix $\mathbf{W}$ which has zeros on the

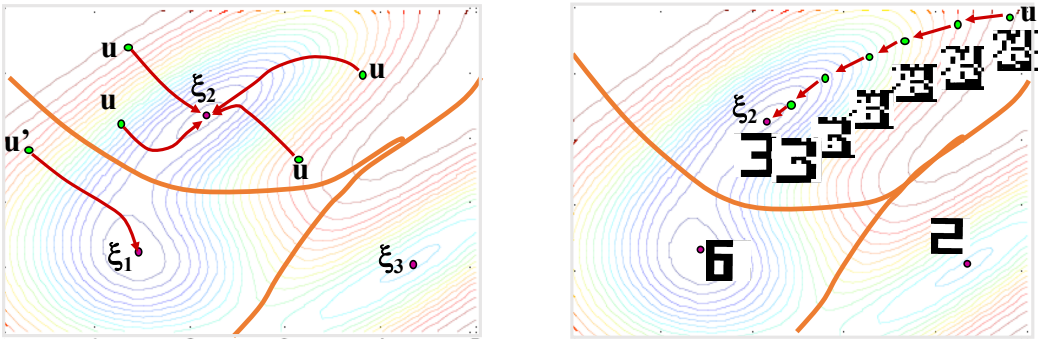

Figure 30: Schematic of recall trajectories in a HN. The energy landscape is rendered as a contour plot. Three fundamental patterns here correspond to three pixel images of digits 6, 3, 2. Since there are three local minima, that is, three point attractors, there are three basins of attraction. They are marked by orange boundary lines. Left: cues $\mathbf{u}$ lying in the basin of attraction of ξ_2 are attracted toward ξ_2. Right: a concrete 7-step trajectory leading from cue $\mathbf{u}$ to ξ_2. Note that the clean gradient-descent (red) lines would look much more jittery in the actual HN dynamics because it is stochastic. The Hopfield dynamics is not a gradient descent dynamics – there is no gradient in the discrete state space of HNs. Pixel images taken from Haykin (1999).

diagonal. Here is a little example with $L = 4$:

$$\mathbf{W} = \begin{pmatrix} 0 & -3 & 0.5 & 0.2 \\ -3 & 0 & 1 & 0.1 \\ 0.5 & 1 & 0 & -2 \\ 0.2 & 0.1 & -2 & 0 \end{pmatrix} \tag{40}$$

The most noteworthy thing here is that the weight matrix is symmetric. This means that the connection weight of the synaptic link between neurons i and j is the same in both directions, $w_{ij} = w_{ji}$. Connections are undirected in HNs — which is biologically unrealistic but opens the doors to "energy"-based computing.

A *state* of a HN is an L-dimensional binary vector with entries from $\{-1, 1\}$.

A key idea about HNs and some other powerful neural network models that we will meet in the next section is that each state is assigned to a real-valued quantity that is called the *energy* of this state. In HNs, given a state $\mathbf{x} = (x_1, \ldots, x_L)' \in \{-1, 1\}^L$, the *energy* of this state is defined by

$$E(\mathbf{x}) = -\sum_{\substack{i,j=1,\ldots,L \\ i<j}} w_{ij}\, x_i\, x_j = -\frac{1}{2}\mathbf{x}'\,\mathbf{W}\,\mathbf{x}. \tag{41}$$

Neural network models in which an energy of states is defined lead to a tight and mathematically rigorous connection between neural network dynamics and

statistical thermodynamics. Such models and the way of thinking behind all of them leads to a subfield of neural network models called *energy-based models*. I will not dig deeper into the links between HNs and theoretical physics at this point but leave that theme for later sections, when we will meet other, more powerful energy-based models.

A side fact: Hopfield is a theoretical physicist. While his work certainly has had a strong impact in theoretical neuroscience and machine learning, it also has triggered a whole school of research within theoretical physics. This line of investigation is hardly perceived outside physics, especially not in neural networks research — one of these strange facts about the disconnectedness of social subcommunities in the sciences.

The rule for the state update dynamics is a *stochastic* rule in HNs. If at time n the state is $\mathbf{x}(n) = (x_1(n), \ldots, x_L(n))'$, the next state $\mathbf{x}(n+1)$ is obtained as follows:

1. Randomly select one neuron x_i.

2. Compute its activation value at time $n+1$ by

$$x_i(n+1) = \text{sign}(\sum_{j \neq i} w_{ij}\, x_j(n)). \tag{42}$$

   where "sign" is the signum function $\text{sign} : \mathbb{R} \to \{-1, 1\}$, $\text{sign}(z) = 1$ if $z > 0$ else $\text{sign}(z) = -1$. In the rare case that $\sum_{j \neq i} w_{ij}\, x_j(n) = 0$, set $x_i(n+1) = x_i(n)$.

3. Set $\mathbf{x}(n+1) = (x_1(n), \ldots, x_{i-1}(n), x_i(n+1), x_{i+1}(n), \ldots, x_L(n))'$, that is, update only the activation of the neuron x_i.

Note that $\mathbf{x}(n+1) = \mathbf{x}(n)$ is a possible outcome of this update operation.

It can be shown (very simple exercise, do it) that such a state update always leads to a reducation of state energy $E(\mathbf{x}(n+1)) < E(\mathbf{x}(n))$ provided that $\mathbf{x}(n+1) \neq \mathbf{x}(n)$, that is, if the update flips the state of the selected neuron. With probability 1 such a random update sequence will end in a local energy minimum state after a finite number of steps.

5.3 Training a HN

The learning problem for a HN is this:

Given n L-dimensional training patterns $\xi_1, \ldots, \xi_N$, find a weight matrix $\mathbf{W}$ which creates an energy landscape that has the training patterns located at the local minima, and every local minimum corresponds to one of the training patterns.

Achieving this goal is not always possible as we will see, but the conditions when it is (not) possible are well understood.

There are two methods of finding a weight matrix $\mathbf{W}$ satisfying the learning objective (if it exists). The first method is very fast and simple: there is an

analytical formula that directly computes $\mathbf{W}$ from the training patterns $\xi_1, \ldots, \xi_N$. The second method is iterative-incremental and may appear unnecessarily time-consuming, but it is biologically plausible and could be used by real brains (whose neurons cannot compute the analytical formula of the first method). I will present the formalism of both and after that explain in intuive terms why they work.

5.3.1 Analytical solution of the learning problem

If a weight matrix $\mathbf{W}$ exists which solves the learning problem, it can be written (and computed) as

$$\mathbf{W} = \frac{1}{L} \left(\sum_{i=1,\ldots,N} \xi_i \, \xi_i' - n\mathbf{I} \right), \tag{43}$$

where $\mathbf{I}$ is the L-dimensional identity matrix.

Proving that this formula places the fundamental patterns at local minima (if possible) requires concepts from dynamical systems that we did not cover (namely Lyapunov functions, you can find a full treatment in Chapter 14 of the textbook Haykin (1999)).

Two technical but not very important details in (43). The prefactor $1/L$ is not mathematically necessary: any nonnegative factor multiplied into $\mathbf{W}$ will only linearly scale the energy landscape but not change the locations of the minima. The factor $1/L$ is included as a reverence to biological plausibility: without it, the total sum of inputs $w_{ij} \, x_j$ (see (42)) hitting a neuron i would grow with the size L of the network, leading to unrealistic large impacts in large networks. With the prefactor $1/L$, the energy levels expressed in (41) are normalized, i.e. they do not grow with network size.

Second, the term $-n\mathbf{I}$ simply sets all self-connections w_{ii} to zero, as demanded by our HN model (notice that each of the matrices $xi_i \, \xi_i'$ has all 1's on the diagonal).

5.3.2 Iterative solution of the learning problem

An iterative method to obtain a weight matrix $\mathbf{W}$ which solves the learning problem goes like this:

Given: a training dataset $\xi_1, \ldots, \xi_N$ of L-dimensional patterns.

Initialization: create a random initial weight matrix $\mathbf{W}(0)$ (symmetric with zeros on diagonal).

Loop:

- At update step k, present one of the training patterns to the network (picked at random or in cyclic order), say $\xi = (\xi^{(1)}, \ldots, \xi^{(L)})'$ is presented. The weight matrix before entering step k is $\mathbf{W}(k-1)$.
- Update all weights $w_{ij}(k-1)$, where $i \neq j$, by

$$w_{ij}(k) = w_{ij}(k-1) + \lambda \, \xi^{(i)} \, \xi^{(j)}, \tag{44}$$

where $\lambda > 0$ is some small *learning rate*, obtaining $\mathbf{W}(k)$.

Stop when you reach a condition of a previously defined stopping criterion. For instance, you can stop when the largest weight (in absolute value) hits a predefined ceiling; or when a test set of corrupted input patterns is recalled correctly; or when the energies of the training patterns are smaller than the energies of all patterns that are similar to training patterns except at one flipped vector entry; or (best) when the pairwise ratios of weights in the sequence $\mathbf{W}(k)$ appear to converge up to a predefined, small residual change.

Notice that the incremental update (44) can be written in matrix form as

$$\mathbf{W}(k) = \mathbf{W}(k-1) + \lambda \left(\xi\, \xi' - \mathbf{I}\right). \tag{45}$$

From this finding is easy to conclude that this incremental learning rule will converge to a weight matrix that is the same as the one obtained from the analytical solution (43), up to a scaling factor which grows larger and and a residual error in matrix entry ratios which decreases the more the longer you run the iterative computation. The local formulation given in (44) is meant to underline biological plausibility: in order to effect the change of a weight w_{ij}, only information that is *locally* available at this synaptic link is needed. Generally speaking, in any neural network architecture any learning rule which, in order to determine a change of a weight w_{ij} needs non-local information which is drawn from neurons other than neurons i and j, is deemed biologically impossible. Biological synapses can be adapted only on the basis of information that is locally available at that very synapse. Specifically, the backpropagation algorithm is non-local.

5.3.3 Why it works. The idea of Hebbian learning

It is not difficult to get an intuition why the rule (45) does its job. This is a straightforward mathematical argument: the weight adaptation (45) lowers the energy of the training pattern ξ which is used in that step. Please check this claim by yourself from the definition (41) of energy – I leave it as an easy exercise. The intuition is thus that we incrementally and repeatedly change $\mathbf{W}$ by making it to yield lower energies for the training patterns. It is however nontrivial to show that this does not at the same time lower energies for patterns outside the training set more than for training patterns.

The learning rule (44) also instantiates a general learning principle that is believed to be ubiquitously effective in biological brains. This is called *Hebbian learning*. Because Hebbian learning is mentioned in the NN literature in many places and in many contexts, I will expand a little on this topic.

Donald O. Hebb (1904-1985) was a psychologist / neuroscientist (he started out as a teacher of English) who gave us one of the main guiding principles for understanding how neural circuits can represent conceptual information. In his

book *The Organization of Behavior* (Hebb, 1949), he developed a theory of biological neural learning mechanisms that could explain how the human brain can memorize, recall and re-generate perceptual and conceptual patterns to which it is repeatedly exposed at learning time. To explain this capacity, he developed a theory of *cell assemblies*. A cell assembly can be thought of as a group of neurons that are mutually exciting each other through positive ("excitatory") synaptic connections. These mutually exciting connections arise in a learning process. If some perception (for instance, a child seeing its mother's face) is repeatedly made, and at each presentation a certain subset of neurons in, say, a visual processing area of the brain is simultaneously activated by this perception, then these repeatedly co-activated neurons will form mutually excitatory links. In Hebb's wording, a *memory trace* is formed: the perceptual experience becomes encoded in the cell assembly. This assembly can then function very much like a trained HN: if some of its neurons are excited by sensory input or input from other brain areas, the entire assembly tends to self-excite. Hebb stated this learning principle in a paragraph that has become one of the most often cited sentences in neuroscience:

"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."

In the folklore of neuroscience, this principle has been shortened to the catchphrase

"Cells that fire together, wire together."

Hebb was, in the first place, a psychologist, and did not use mathematical formalism to make this learning mechanism more precise. He wanted to model the biological brain and tried to relate his principle to what was known at his time about neural microanatomy. He also tried to relate his principle to a large spectrum of findings in cognitive and developmental psychology that were known at his time. He certainly would never have thought of simplifying a neuron to a mathematical variable that can only take the values -1 and $+1$, and also the idea to build artificial neural systems was alien to him — that idea would start its rise but 10 years later, in the form of the Perceptron.

The Hopfield network is nonetheless an obvious instantiation of "Hebbian learning". I conclude this excursion by emphasizing that today there is a large spectrum of rigorously formalized neural learning mechanisms which are Hebbian in their core. They give a spectrum of different answers to problems which arise when one starts working out the consequences of the cell assembly idea. I mention just two:

- The mutual excitatory connections of cell assemblies must in some way be complemented and compensated by inhibitory connections — otherwise the entire brain would burst into flames by global spread of self-excitation.

- Neurons can only excite one another by sending spikes. A spike lasts only about one millisecond. But sensory impressions or the activation of con-

ceptual items in one's mind may last seconds — three orders of magnitude longer. This mismatch in timescales makes it necessary to develop ingenious, nontrivial schemes to account for fine-grained temporal relationships between the spike events in neurons within an assembly.

The first problem is solved in the HN model by giving equal citizenship rights to positive and negative neural activations, and positive and negative synaptic weights. The second problem does not arise in HNs because there are no spikes in the simplisitic neuron model used.

5.4 Limitations

Above I inserted the cautionary clause "... if possible ..." at various places. In fact, it is not always possible to store N training patterns in an L-neuron HN, such that after learning there are exactly N local minima in the energy landscape which correspond to the precise training patterns. Several things can go wrong:

- The fundamental memories are only approximately equal to the original training patterns, that is, they will differ from the training patterns in a few bits.
- Not all training patterns can be stored — the storage capacity of a HN is limited. Specifically, consider the ratio N/L of the number of training patterns over the network size. This fraction is called the *load* of training a HN. The following facts are known for HNs:
  1. For a load $N/L > 0.138$ (I give only first three significant digits, precise number is known), storing training patterns using (43) breaks down entirely: none of the training patterns will be stored, not even approximately.
  2. A HN works really well only for loads $0.03 < n/L < 0.05$, in the sense that the training patterns end up in local minima that correspond to perfect copies or only slightly altered versions of the original patterns.
- Besides at the locations of the stored training patterns, other local minima are created in the performance landscape which do not correspond to training patterns. Such "false memories" are called *spurious states* in the HN literature.

There is one thing that *always* "goes wrong": if ξ is a fundamental memory, then also the sign-inverted pattern $-\xi$ is a fundamental pattern. HNs cannot distinguish between patterns and their sign-inverted versions (why? find out for yourself, it is an easy one).

This is not so good news, inasmuch as these limitations more or less render HNs useless for practical applications.

On the other hand, the reasons why these limitations occur, and which of them strike how badly at what load levels, are almost completely known. The corresponding mathematical analyses have been carried out, mostly by theoretical physicists, in the 1980's and now form a classical body of rigorous insight into the conditions of storing information in a neural network.

In the remainder of this subsection I document these findings in some more detail.

5.4.1 Bit errors in fundamental memories

If the storing formula (43) places a training pattern ξ *exactly* at a local minimum, it becomes a *point attractor* under the update rule (42). This means that if one cues the trained HN with a test input pattern $\mathbf{u} = \xi'$ which is equal to ξ except for a small number of flipped bits — that is, ξ' is located in a *neighborhood* of ξ — then the energy minimization induced by (42) will let the sequence of patterns end up in ξ, as we would wish.

However, often the storing formula (43) leads to a slight misplacement of the local minimum associated with a training pattern ξ. The created local minimum $\hat{\xi}$ will be very near to, but not exactly equal to, the original pattern. As a consequence, if the HN is cued with the exact pattern ξ, the state update dynamics will move it a little, going down the "energy valley", until it ends in $\hat{\xi}$. Some bits in the original pattern ξ will become flipped — these bits are *unstable*.

How many bits in a pattern will be unstable, that is, how precisely can a HN recall the training patterns? This depends probabilistically on the load n/L. Here is the main result:

Proposition 5.1 *If N patterns are stored in an L-dimensional HN using the formula* (43)*, the probability that a given bit i among the L bits of a training pattern ξ is flipped when the HN is cued with ξ is*

$$P_{\mathsf{flip}}(\textit{bit } i \textit{ is unstable} \mid \textit{load is } N/L) = \Phi(-\frac{1}{\sqrt{N/L}}).$$

Here $\Phi : \mathbb{R} \to \mathbb{R}$ is the so-called *error function* associated with the one-dimensional Gaussian distribution, visualized in Figure 31.

This proposition does not imply that, if the trained HN is cued with ξ, the state update will come to rest at a pattern $\hat{\xi}$ close to ξ. It may occur that after ξ is used as input, the flipping of some bits in ξ triggers a *bit flip avalance* and the state update leads far away from ξ. The training pattern is entirely unstable and is not located near a local minimum of the energy landscape. In the worst case, all bits in ξ become randomly flipped, resulting in a maximal $P_{\mathsf{flip}} = 0.5$.

How bad this avalanching becomes depends on the load. In the limit of large L, at a load of $N/L \approx 0.138$, every training pattern becomes maximally unstable. Avalanches start to occur for loads $n/L > 0.05$. Figure 32 shows the growth of the avalance effect with the load.

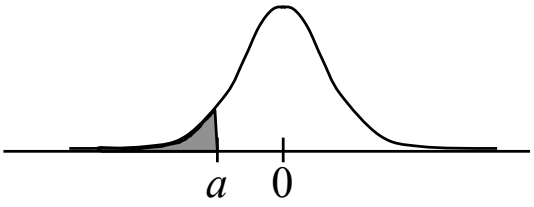

Figure 31: The error function $\Phi(a)$ gives the area under the pdf of the standard normal distribution on the left-hand side of a.

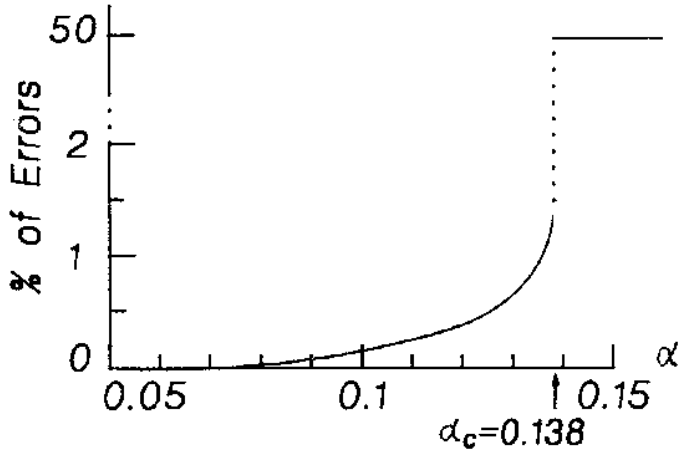

Figure 32: Percent of changed pattern bits under *iterated* network update, vs. load (here denoted by). Figure and result from Amit et al. (1985).

The result illustrated in Figure 32 is an example of a *phase transition* — you remember that from the dynamical systems primer.

5.4.2 Spurious states

The textbook of Haykin (1999) describes in detail a computer experiment where the patterns were 10 × 12 black/white pixel images (the patterns shown in the right panel of Figure 30 were copied from that section in the Haykin book). This makes $L = 120$. A small set of training patterns consisting of merely $N = 8$ images was used. It contained pixelized versions of the digits 0,1,2,3,4,6,9, plus a pattern showing a 5 × 6 sized black square in the upper left corner (Figure 33).

When testing the trained HN with about 43,000 different cues obtained by randomly flipping one fourth of the pixels in one of the training patterns, the network frequently ended in a local energy minimum (attractor) state which was not one of the 8 training patterns. Such "wrong memories" are called *spurious states* in the HN terminology. Figure 34 shows 108 such spurious states. Most

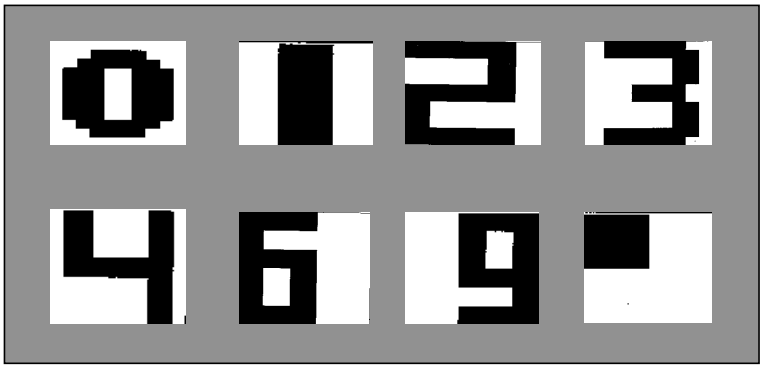

Figure 33: The 8 training patterns from a HN computer demo presented in Haykin (1999). The slight jitter visible at some of the black-white boundaries are artefacts from my postprocessing of photocopies from the Haykin book.

likely there are many more — it is not possible to exhaustively search a pattern space of 2^{120} many different patterns! Haykin cites without further explanation findings of Amit (1989) to the effect that there are three kinds of spurious states (see Figure 34), one of which is always present: namely, the sign-inverted training patterns always become local energy minima if the original training patterns do so (why? that's an easy one).

5.4.3 Summary of imperfections related to load

Here is an overview of what goes wrong or right in HNs depending on the load. I collected these findings from Amit et al. (1985).

- For all loads N/L: stable spin glass states exist.
- For $N/L > 0.138$: spin glass states are the only stable ones.
- For $0 < N/L < 0.138$: stable states close to desired fundamental patterns exist.
- For $0 < N/L < 0.05$: pattern-related stable states have lower energy than spin glass states.
- For $0.05 < N/L < 0.138$: spin glass states dominate (some of them have lower energy than pattern-related states).
- For $0 < N/L < 0.03$: additional mixture states exist, with energies not quite as low as the pattern-related states.

In summary, a HN works really well only for a load $0.03 < N/L < 0.05$.

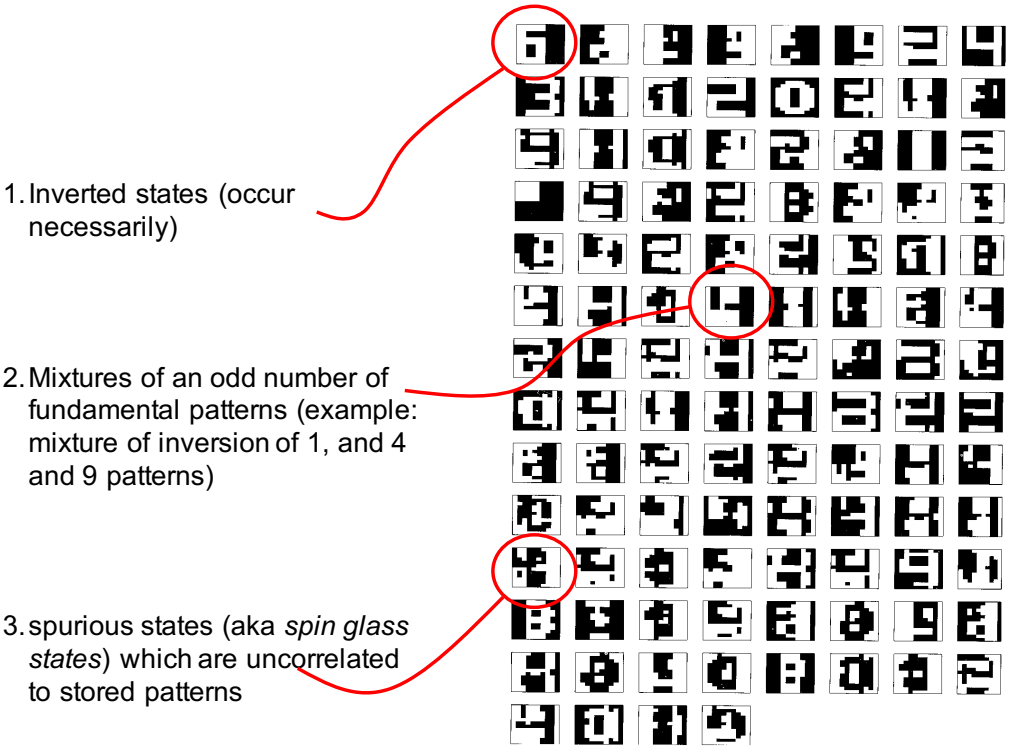

Figure 34: Some spurious states found in the demo in the Haykin book.

5.5 Miscellaneous notes

HNs have been deeply explored, and many more interesting facts could be related about them. I conclude this section with some ad hoc comments.

Local stochastic vs. deterministic global update. We used a stochastic local update: in (42) a single bit was randomly selected to consider whether it would be flipped in a state update. Alternatively, one could consider an update where *all* bits of a pattern would be subjected to (42) in one global update. Interestingly, this is not equivalent to the local stochastic update. A dramatically different update dynamics would result, and the local minima of the energy landscape would fall into different positions.

Such a decisive difference between local stochastic and global deterministic state update rules is not confined to HNs, but is a common finding in all kinds of dynamical systems that have states made of discrete bits.

Specifically, digital computer chips use a global deterministic update rule. This necessitates a *global clock* which on your notebook box typically runs at a Gigahertz clock rate. In contrast, in the emerging field of *neuromorphic microchips* (I will say more about them in the final lecture), no global clock is available and the update dynamics is necessarily local and stochastic, driven

by local physical laws. This is one of the reasons why the computational mathematics in such unconventional microchips is entirely different, and much less understood, than the mathematical theory for classical digital computing — which every student of computer science learns in lectures on theoretical CS.

Variations. The HN model that I described in this section is but one specific version among many other formalizations. For instance, one could use state values $\{0, 1\}$ instead of $\{-1, 1\}$. This is a minor variation which leads to an entirely equivalent theory. A much stronger modification is to admit continuous values of the neurons, say from the continuous interval $[0, 1]$. The quality of findings is similar to what we saw in this section, though details and formalism differ. The Haykin book discusses both discrete and continous HNs.

Importance of symmetric synaptic connections. The symmetry of the weight matrix $\mathbf{W}$ is crucial for HN theory. If one would admit asymmetric connections $w_{ij} \neq w_{ji}$, energy functions could no longer be defined; the intuition of state updates reducing energy would evaporate; besides stable fixed points, other kinds of attractors (for instance cycles) would emerge and the "use case" of storing patterns as point attractors would collapse. It is a characteristic of the larger family of energy-based models of neural computation to have symmetric weights.

Heteroassociative networks. HNs are alse referred to as *autoassociative networks*. This terminology is motivated by the fact that the HN update dynamics "associates" a fundamental memory ξ with itself: $\xi(n + 1) = \xi(n)$. In heteroassociative networks, one wants to learn associative *sequences* of patterns. The cognitive modeling motivation to consider such sequences is the idea of a flow of thought: if you think of a rose, the next moment you might be thinking of love, then of tears, and so on. If your brain were a HN, once you think of a rose, you will continue to think of the rose over and over again...

Designing and training a heteroassociative network, one would start with a training dataset that is made of sequences of patterns. In a simple case that would be a circular sequence $\xi^{(1)}, \ldots, \xi^{(n)} = \xi^{(1)}$. The learning formula (45) would be replaced by

$$\mathbf{W}(k) = \mathbf{W}(k-1) + \lambda\, \xi^{(i+1)}\, \xi^{(i)\prime}, \tag{46}$$

and (if convergence is achieved) the weight matrix would become non-symmetrical. A substantial literature on such heteroassociative networks exists (of course)

HN pro's. Many things are good and influential and insightful about HNs:

- It is a simple model, can be mathematically almost fully analysed
- HNs are not biologically immediately unrealistic.
- HNs have strongly influenced how neuroscientists think about memory.
- Deep connections to other field of computational physics exist (spin glass models).
- HNs are robust against "brain damage" (not discussed in this section), that is, if in a trained HN some weights are modified, the overall performance only gradually deteriorates.
- HNs have historically helped to salvage neural network research, and J. J. Hopfield was traded over several years as a Nobel prize candidate.

HN con's. A number of aspects of HNs are maybe not so good:

- The memory capacity is small. It is however unclear whether the capacity of biological brains is larger, or how it can be quantified in the first place.
- All the "mathematically nice" results only hold for *uncorrelated* fundamental patterns — an unrealistic assumption for real-world patterns. A HN would have difficulties to store two rather similar patterns separately.
- Various imperfections — pointed out above in Section 5.4.
- HNs are not technically useful.
- HNs have strongly influenced how neuroscientists think about memory. This may be bad if biological brains turn out to work entirely different.

6 Moving toward Boltzmann machines

One of the central models of a "cognitive" neural network is the *Boltzmann machine* (BM). The Boltzmann machine is a must-know architecture for machine learners, AI afficionados, cognitive scientist and neuroscientists alike because

- it gives a computational model of memory, concept representation, learning and reasoning in hierarchical cognitive systems,
- it can be used in machine learning as a universal learning "machine" for almost any probability distribution,
- it can be understood as a universal statistical *inference* ("reasoning") device because it admits to compute conditional probabilities of any sort ("what is the probability I see a horse given that this is the input picture?", "what is the probability that tomorrow it will rain given that today it was cold and windy?" — for a conditional image completion task watch `https://www.youtube.com/watch?v=tk9FTdKOL5Q`),

- it connects neural networks to statistical physics in a deep and insightful way,
- it is the main and first representative of a large class of computational architectures in machine learning and theoretical physics, the class of *energy based models* of neural computing,
- it helped paving the way for deep learning,
- it is mathematically very transparent, and looks so natural and elegant that nobody (at least, no mathematician or theoretical physicist) can remain untouched by its charms.

This sounds as if there *must* be a drawback. Indeed, there is. Training and using a Boltzmann machine is computationally very expensive. This has barred a widespread use in machine learning (except for the early years of deep learning, say between 2006 and 2012, when the Boltzmann machine was competing with MLPs and CNNs for realizing deeply layered learning systems). Although in those exciting years the nascent DL community was using a computationally streamlined version, the *restricted Boltzmann machine* (RBM), ultimately the computuational costs (among other reasons) led to a depreciation of BM/RBMs in machine learning — though there is still some research on them in the DL world. However, biological brains can do the computations which are so expensive (namely, *sampling* operations) at almost zero cost. Therefore, the BM and related models retain their fascination for theoretical neuroscience and the theory of cognitive computing.

In order to understand BMs, one must bring to the table two formal concepts from statistics and statistical physics, namely the concept of a *Boltzmann distribution* (which gave the Boltzmann machine its name) and the notion of a *sampling algorithm*. These concepts are also forming the basis for the general, large class of energy-based models. It is thus a well-invested effort to make friends with these two concepts. The class of energy-based models include, for instance,

in theoretical physics: so-called *spin-glass* or *Ising* models, which describe how solid-state materials change their properties under the influence of external controls like temperature, electric or magnetic fields;

in statistics and classical pattern recognition, especially computer vision: so-called *Markov random fields*, a 2D or 3D generalization of Markov processes, which can be used, for instance, for photographic image processing;

in the cognitive neurosciences: various concrete computational models for explaining hierarchical information processing in brains, where bottom-up sensor data processing and top-down attention and expectation mechanisms interact – in fact, one of the current leading paradigms in cognitive science is rooted in energy-based formalism, namely the *free energy principle* of learning in intelligent agents;

in optimization theory / operations research: the *simulated annealing* algorithm for finding a good local minimum in complex performance landscapes – besides evolutionary search methods ("genetic algorithms") this is a last-resort, general-purpose optimization algorithm when all others falter in the face of hypercomplex performance landscapes;

in machine learning: a major branch of machine learning, namely *Bayesian networks* and the more general class of *graphical models* uses energy-based mechanisms to implement learning architectures and mechanisms for rational reasoning on the basis of stochastic sensor input / stochastic input data. These models have a wide range of uses which is orthogonal to the use cases of deep learning. This line of modeling is therefore not dimmed by the deep learning revolution, a fate from which so many other fields in machine learning have suffered.

Before we can start enjoying the Boltzmann machine, and before we can take a closer look at some of these other kinds of models, we must equip ourselves with a fair understanding of the Boltzmann distribution and the idea of sampling algorithms. This is what I will try to give you in this section.

6.1 The Boltzmann distribution

The Boltzmann distribution is a classical concept from the field of statistical physics. So, first question: what is statistical physics?

In statistical physics (SP), the general objective is to explain *macroscopic* phenomena from the interaction of very large "ensembles" of *microscopic* particles. "Macroscopic" means that something can be measured with instruments of everyday size: you can measure the temperature in a pot of water with a thermometer, you can measure electric fields with a voltmeter, etc. These instruments have sizes that you will find them again after you have put them away on a shelf – they belong to the macroscopic world. "Microscopic", in contrast, means that one cannot measure it because it's too small — at least, one cannot measure it readily (e.g. the current velocity of a water molecule); and sometimes one cannot measure it at all (quantum effects submitting to uncertainty principles).

Two examples of how statistical physics connects the microscopic to the macroscopic levels of description:

- The temperature (macroscopic) is explained by / reduced to a the average kinetic energies of atomic particles (microscopic) contained in a vessel or block of solid matter.

- The magnetic strength (macroscopic) of a magnet is explained by / reduced to the average spatial alignment of the spins (microscopic) of the atoms in the crystal lattice of the magnet.

Macroscopic properties of materials have been a subject of research in physics since the beginnings of that field. In particular, in a classical branch of physics called *Thermodynamics*, the macroscopic observables temperature, pressure, volume, viscosity and many others have been investigated, and many laws of how they depend on each other in different materials have been found. These laws are particularly interesting and nontrivial when it comes to *phase transitions* — sudden changes of macroscopic observables, for instance when water freezes or dynamite explodes.

These laws could be found, formally stated and experimentally verified in classical thermodynamics, but they could not be *explained*. This is what statistical thermodynamics (and subsequently, the more general discipline of statistical physics) strives to achieve: mathematically deduce the macroscopic laws of thermodynamics from assumed mechanisms of how microscopic particles interact in large numbers, such that the macroscopic observables and laws can be derived as *statistical* distributions.

Brains are also macroscopic lumps of matter in which the interactions of large numbers of microscopic "particles" (the neurons) give rise to macroscopic observables (for instance, the words coming out of your mouth). It is a naturally inviting idea to describe brains with the tools of statistical physics. The Boltzmann machine is one way of doing exactly this.

A (rather, *the*) founding father of statistical thermodynamics was Ludwig Boltzmann (1844-1906, `https://en.wikipedia.org/wiki/Ludwig_Boltzmann`), an Austrian professor of physics and philosophy (!).

In order to start getting familiar with the Boltzmann distribution, let us consider the textbook example of a vessel filled with water.

The macroscopic description of this system is simple: volume, temperature and pressure are enough to characterize it. These can be measured with macroscopic instruments.

The microscopic description is based on the notion of a *microstate*. A microstate is (in a first approximation, treating water molecules as point masses) a specification of all the 3D position coordinates and 3D velocity components of all individual H_2O molecules in that vessel. This is a very high-dimensional real-valued vector, say of dimension d, which completely characterizes the molecular-level state of affairs inside the vessel at given moment in time.

Boltzmann asked, and answered, the following fundamental question: *what is the probability distribution of these microstates?* — and from that distribution he would infer the values and laws of macroscopic observables.

First let us understand that this distribution is not uniform. Some microstates are more probable to occur than others. The water molecules bounce against each other and the vessel's walls, exchanging impulses, in a wild stochastic dance. Boltzmann assumed that the vessel is submerged in a *heat bath*, which you can visualize as an infinite ocean of water that has the same temperature as the water in the vessel and the vessel's walls. Ocean molecules bounce against the vessel

walls, vessel wall molecules transmit these impulses to water molecules inside the vessel, and conversely there are stochastic transmissions of molecular impulses from the inside to the outside. Just be stochastic coincidences, at some moments large amounts of impulse energy will have found their way from the outside to the inside; and at other moments, less of it. A microstate's (kinetic) *energy* is the sum of all the kinetic energies of the individual water molecules. We don't have to understand how the kinetic energy of a water molecule is defined (maybe you remember from high-school physics). The bottom line is that the energy of microstates is wildly fluctuating all the time.

Side remark: this energy fluctuation is due to the embedding of the vessel in a heath bath. If the vessel would be perfectly isolated, then the energy of all possible microstates would be constant due to the law of conservation of energy.

Let us denote microstates by $\mathbf{s}$ and the energy of a microstate by $E(\mathbf{s})$. Now we have put our foot on the doorstep of one of the grandest and most far-reaching principles of modern physics. Boltzmann reasoned that the probability of a microstate depends *only* on the microstate's energy $E(\mathbf{s})$ and the temperature T of the vessel/heatbath system. Note that physicists measure *absolute* temperature (in Kelvin), where zero is the lowest possible temperature. The probabilities of microstates are described by a simple formula which today is called the *Boltzmann distribution*. Since the position&velocity vector microstates are continuous-valued vectors, this distribution is written down as a pdf which assigns a probability density value $p(\mathbf{s})$ to each microstate:

$$p(\mathbf{s}) = \frac{1}{Z} \exp(\frac{-E(\mathbf{s})}{T}). \tag{47}$$

In this equation, Z is the normalizing factor that ensures that the pdf p integrates to unity:

$$Z = \int_{\mathbf{s}\in S} \exp(\frac{-E(\mathbf{s})}{T}) ds, \tag{48}$$

where S is the space of all possible microstates. I am not a physicist but I would think that $S = \mathbb{R}^d_{\geq 0}$ comprises all non-negative real-valued vectors of dimension d.

Note that both p and Z depend on the temperature T, so we sometimes write $p(\mathbf{s}, T)$ and $Z(T)$. Z is called the *partition function* in statistical physics ("function" because it depends on the temperature). The partition function plays a central role in physics. In most cases it cannot be calculated analytically. One needs supercomputing power to approximately estimate it, and the possibility to actually do this has changed the face of modern physics (computing Z is one of the reasons why physicists need supercomputing facilities). Computing estimates of Z is also of importance in certain applications of machine learning, deep learning in particular. The entire Chapter 18 of the deep learning "bible" (Goodfellow et al., 2016) is devoted to estimation algorithms for Z. Luckily, in many applications, among them the Boltzmann machine, Z cancels out and need not be computed.

The Boltzmann distribution can also be defined on spaces of *discrete* microstates (finitely or countably many). Then the pdf from (47) turns into a probability mass function (pmf) and the integral in (48) into a sum:

$$P(\mathbf{s}) = \frac{1}{Z} \exp(\frac{-E(\mathbf{s})}{T}), \tag{49}$$

$$Z = \sum_{\mathbf{s} \in S} \exp(\frac{-E(\mathbf{s})}{T}). \tag{50}$$

The shape of the pdf (or pmf) changes quite dramatically when the temperature parameter is varied. Figure 35 shows an example. There are two noteworthy extreme cases. When the temperature is very large, the terms $\exp(\frac{-E(\mathbf{s})}{T})$ uniformly approach 1, which (after normalization by division with Z) gives a uniform distribution. At high temperatures, all microstates become equally probable. When the system is cooled down toward zero, the probabilities of the low-energy states grows relative to the probability of the high-energy states. Ultimately, the distribution converges to a point distribution where the (single) lowest-energy state has a probability of 1. Cooling a physical system down toward zero will localize its distribution at the lowest energy state! It should be noted however that this cooling down must be done veeery slowly, called *adiabatic* cooling — in the limit, even infinitely slowly — in order to see this effect in real physics experiments.

The fact that a (sufficiently slow) cooling of a Boltzmann system leads to the *global* minimum of the energy landscape is the basis of the *simulated annealing* algorithm. This is a general-purpose optimization algorithm which can find, in principle, the global minimum of any performance surface. In this respect it is vastly more powerful than gradient-descent optimization algorithms which we discussed in Section 2.2.2. This algorithm is really good stuff to know and I will explain it in more detail in Section 6.4.

Two facts about the Boltzmann distribution worth knowing:

- If a energy function $E : S \to \mathbb{R}$ gives rise to a Boltzmann distribution P, and $E' : S \to \mathbb{R}, \mathbf{s} \mapsto E(\mathbf{s}) + C$ is the same energy function shifted by some constant C, then the Boltzmann distribution P' you get from the shifted energy function is the same as you had before: $P' = P$.

- A Boltzmann distribution arises from an energy function at a given temperature T. Conversely, if $P(\mathbf{s})$ is any (discrete) probability distribution which is nonzero for all $\mathbf{s} \in S$, for any temperature T the energy function $E(\mathbf{s}) = -T \log(P(\mathbf{s}))$ makes $P(\mathbf{s})$ the Boltzmann distribution associated with $E(\mathbf{s})$. Every (globally nonzero) probability distribution can be written as a Boltzmann distribution. As we will see in Section 7, this makes BMs universal approximators of (discrete) probability distributions.

While the Boltzmann distribution has been found and explored in physics, its ideas and maths transfer to any other application domain where there are

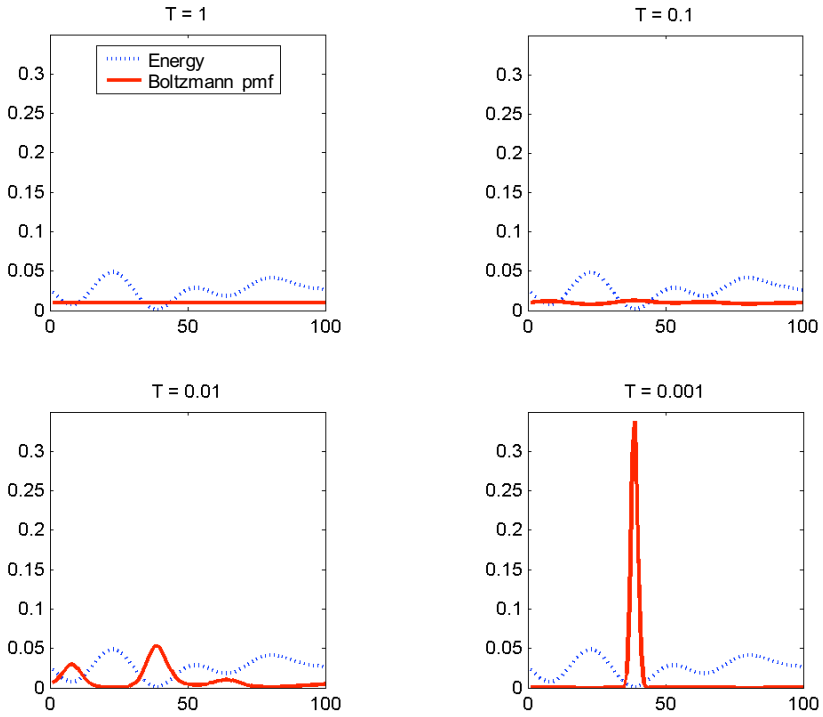

Figure 35: A Boltzmann distribution over a finite space of 100 microstates. The distribution is given by its pmf (red). The underlying energy function, which is not changing with temperature, is rendered by a blue broken line.

macroscopic systems which can switch between large numbers of microstates, each of which has an "energy". In particular, neural networks (artificial ones and real brains) can be regarded as macroscopic systems, with vectors of activations of all neurons being the microstates. If one defines some sort of "energy" for the activation vectors, the Boltzmann machinery can be launched. The "energy" which one defines can be *any* function from microstates to the reals; physical considerations can be ignored; it even need not be non-negative.

6.2 Sampling algorithms

In this subsection I will first explain in plain English what a sampling algorithm is, then why they are important, and finally in formal terms how the specific sampling algorithm works that is used for the Boltzmann machine.

6.2.1 What is that?

A sampling algorithm, or for short a *sampler*, is a computational procedure which generates "random" examples from a given distribution. Samplers there are many. Some are highly specialized and can only generate random examples from a single distribution, others are generic and can "sample from" any distribution which is given by a pdf or pmf.

You all know a sampling algorithm that samples from the uniform distribution over the interval $[0, 1]$. It comes as a primitive function with all programming languages that I know, including MS Word. It is typically named rand. Every time you evaluate this function, it generates a new "random" number from the interval $[0, 1]$.

In fact, the algorithms that sit behind rand are deterministic, and the outputs from rand are only *pseudorandom* numbers. Many algorithms are known which deterministically generate numbers between $[0, 1]$ in a way that can hardly be distinguished from true randomness by statistical tests. It is quite a sophisticated corner of maths where such clever "pseudorandom generators" are thought out. We just use them as if they generated truly random outputs and don't think twice about the math miracles behind the curtain which turns digital determinism into (almost) randomness. If you want to get true randomness, for instance for unbreakable codes, you'd need to build a non-digital microchip that gets its randomness from quantum fluctuations.

Formalizing the concept of a "sampler" in mathematical rigor requires some stochastic processes theory and is beyond the scope of our course. But I think the idea is intuitively clear. Consider a pdf as a "landscape" with hills and valleys (as the two-hill landscape mapped in Figure 36). A sampler is a mechanism which, each time it is executed, lets fall down a grain of sand on a plane that was flat and empty at the beginning. In the long run, these grains of sand should pile up to a landscape whose profile is the same as the pdf landscape.

Note that a sampler need not "jump around" wildly. The modern algorithms that are used for generic sampling tasks over arbitrary pdfs or pmfs indeed have some kind of memory. They linger for a while in the vicinity of the place where the last grain of sand was dropped: the next grain will typically fall down somewhere close to the previous one. Such samplers perform a *random walk* over the sampling space, dropping a grain at each step, and the average progression rate of that walk can be slow. The entire landscape is covered only in the long run. The sampler that we will be using for the Boltzmann machine is of this kind. The general theory behind the design of such samplers is called *Markov Chain Monte Carlo* (MCMC) sampling. Once powerful computing hardware became available, MCMC changed the face first of physics, then of other sciences, because these methods (and *only* these methods) made it possible to simulate complex stochastic systems in physics, chemistry, biology and the social and economics sciences. We cannot and need not dig deeper here. If you are interested: a classical tutorial is Neal (1993), and in my legacy lecture notes on "Algorithmic and statistical modeling" I

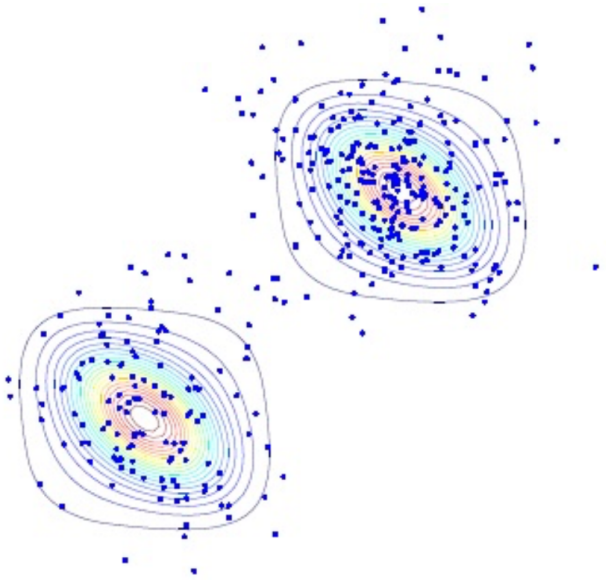

Figure 36: Sampling from a 2-dimensional pdf. The pdf is here rendered by a contour line plot; it consists of two "hills" separated from each other by a "valley". Each blue dot is one output of a sampling algorithm. Graphic taken from the online course notes of IEOR E4703: Monte-Carlo Simulation, Columbia University `https://martin-haugh.github.io/files/MonteCarlo/MCMC_Bayes.pdf` where they were taken from the book *Bayesian Reasoning and Machine Learning* by D. Barber.

give an outline (Section 4.5 in `http://minds.jacobs-university.de/uploads/teaching/lectureNotes/LN_AlgMod.pdf`).

6.2.2 What are they good for?

Probability distributions are the raw material of all scientific research or practical or economical exploits when it comes to dealing with uncertainty in data. Often these distributions are defined over very high-dimensional state spaces, like brain states (activation vectors of all neurons), epidemiological states (healthy / ill assignments to the citizens of the world), or the spatial configurations of a folding protein. It is impossible to "write down", or plot, or even imagine the global geometric shape concerned pdf or pmf. It is a very misleading experience which students of statistics or machine learning get from their textbooks, where all graphics show only very low-dimensional pdfs or pmfs - only one- or two-dimensional examples can be readily visualized! This textbook reading experience falsely leaves the student with the impression that seriously real-life distributions can be described in terms of intuitively understandable geometric shapes, like the

Gaussian bell curve. This is, in general, not possible! And worse yet, it is not only impossible for humans to get an intuitive visual idea of the shape of a complex distribution. It is, as per today, likewise impossible for mathematical formulism to characterize the overall "geometry" of most real-life pdfs or pmfs.

The *only* thing that *can* be computed is usually just the pdf or pmf values of a given point in the state space of the distribution. That is, for any point $\mathbf{s}$ in a high-dimensional state space S, it is possible to compute the corresponding pdf value $p(\mathbf{s})$ or the pmf value $P(\mathbf{s})$.

Oh... no. Not even that is usually possible. The enemy here is numerical underflow. For a demonstration, consider a probability distribution over the set S of possible binary health states of the world population. Assume for simplicity that 10 billion humans live on our planet. Each of them can be ill or healthy — 1 or 0. A global health status vector is thus a binary vector of dimension 10e10. This makes for a large but finite state space $S = \{0, 1\}^{10,000,000,000}$. Since S is finite, a probability distribution over this space has to be described by a pmf. The probability $P(\mathbf{s})$ of a state $\mathbf{s}$ is, on average, $2^{-10,000,000,000}$. Written to base 2, which is what digital computers do internally, this would be written as $0.0 \ldots 01$, with $10,000,000,000$ digits after the dot. But the machine precision on your 64-bit computer allows only for 53 bits of precision. Any number smaller than 2^{-53} is treated as zero by your programs.

A standard escape from such numerical underflow issues to always work not with the raw probabilities $P(\mathbf{s})$ but with their log values $\log(P(\mathbf{s}))$. In base 2, the average log probability then is $\log_2(P(\mathbf{s})) = -10,000,000,000$, an order of magnitude (rather, minitude) that is within convenient precision reach of your machine. If you read math-oriented papers in machine learning you will see log probabilities all over the place.

Ok., let us return to our main thread. I said above that the only thing that can be computed for a distribution given by a pdf or pmf are the values $p(\mathbf{s})$ or $P(\mathbf{s})$, or rather their logs. Let us return to the global health status vector example. An epidemiologist might want to know, *what is the probability Q that more than one tenth of the world population is ill?*. This turns out to be a computational show stopper. Let $S_Q \subset S$ be the set of all health state vectors that have more than 10 percent 1's entries. The mathematical formula that defines Q is

$$Q = \sum_{s \in S_Q} P(\mathbf{s}).$$

This is not computable for two reasons: numerical underflow, and the gigantic size of S_Q.

But now assume you had a way to get a "fair sample" of manageable size, drawn from the distribution P. Say, you have "drawn" 1,000 examples $\mathbf{s}_1, \ldots, \mathbf{s}_{1000}$. You count how many of these example vectors have more than 10 percent 1's in them. Your count number is q. Then, thinking about it, you see that q is an *estimate* of Q! This estimate will become more precise if you collect larger samples, converging to the correct probability Q as your sample size goes to infinity.

This example demonstrates one of two major things that sampling is good for: sampling can be used to get estimates of probabilities for events that a researcher is interested in. Similarly, sampling can be used to get estimates of other statistical quantities, like expectations, variances, partition functions, or all sorts of integrals over functions over the sample space. And there is no other way to get access to these quantities of interest besides sampling. I think you can divine what a game-changer in the sciences it was to afford of both general-purpose sampling algorithms and computing hardware with enough bandwidth.

There is an angle to this success story which is worth knowing and thinking about. Both the mathematical development of general-purpose sampling algorithms, and the computational exploit of them on the first powerful enough digital computers, were done in the Los Alamos Labs in the context of developing the hydrogen bomb. See the webpage authors (2014) for a historical outline and Metropolis et al. (1953) for the landmark scientific publication (42K Google Scholar cites).

The other major thing that sampling is good for: it's just the sampling itself. It can be used to *generate examples* from a statistical model of some interesting part of the world. When the Boltzmann machine is used after it has been trained, this is the way that it is used. In cognitive terms: the random walk of an artificial brain state sampler creates a "stream of thought" which in the long visits all the places and themes that the brain knows about. The technical term for this process, used both in psychology and machine learning, is *confabulation*.

6.3 The Metropolis algorithm

Today many general-purpose sampling algorithms are in use. Their common ancestor is the *Metropolis-Hastings algorithm*, often named just "Metropolis algorithm". It was developed by a joint effort of eminent mathematicians and nuclear physicists. The classical reference is Metropolis et al. (1953). Metropolis sampling works with a particular elegance in conjuction with Boltzmann distributions, and it is the sampler that is used in the Boltzmann machine. In this subsection I describe this sampler in detail.

The Metropolis sampler is applicable in very general situations. All that is needed is some set (or "space") S of possible "states" of some modeled system, and for each state $\mathbf{s} \in S$ a computable pdf value $p(\mathbf{s})$ (for continuous state spaces) or a computable pmf value $P(\mathbf{s})$ (for discrete spaces). In fact, even less need be given: it is enough to have the pdf or pmf only up to some unknown normalization factor. The Metropolis sampler only needs the ratios $p(\mathbf{s})/p(\mathbf{s}')$ or $P(\mathbf{s})/P(\mathbf{s}')$ to run, and these ratios remain the same if the pdf or pmf is scaled by some constant factor. This makes it unnecessary to compute partition functions.

I will present the Metropolis sampler for the case of a given pmf, because that is the situation we will meet in the Boltzmann machine. The pdf case is entirely analoguous and you can easily translate the pmf recipes into pdf recipes.

So, here is the scenario. We are given a finite state space S and a non-negative function $F : S \to \mathbb{R}^{\geq 0}$ whose sum $\sum_{\mathbf{s}\in S} F(\mathbf{s})$ is finite. I call this function a *proto-pmf* because it could be turned into a pmf P by scaling it with $1/Z = 1/\sum_{\mathbf{s}\in S} F(\mathbf{s})$, but this normalization is not needed (and often not feasible), so we stick with the proto-pmf F.

The task is to generate a potentially endless sequence $\mathbf{s}^1, \mathbf{s}^2, \ldots$ such that, in the long run, this sequence of *sampling points* would re-model the pmf landscape in the intuitive sense of our "grain dropping" metaphor.

I do not give the mathematical derivation of why the Metropolis sampler does its job, but just describe the algorithmic procedure. If you are interested in the mathematical derivation, you can find it in Section 4.7 in my legacy lecture notes on Algorithmical and Statistical modeling, online at `http://minds.jacobs-university.de/uploads/teaching/lectureNotes/LN_AlgMod.pdf`, or the MCMC tutorial of Neal (1993).

The Metropolis algorithm generates the sample point sequence $\mathbf{s}^1, \mathbf{s}^2, \ldots$ by implementing a Markov process mechanism, that is, every newly generated point $\mathbf{s}^n$ depends (only) on the predecessor $\mathbf{s}^{n-1}$ (*note:* a Markov *process* generalizes discrete, finite-state Markov *chains*, as we saw them in the dynamical systems tutorial, to continuous state spaces). In order to get the whole sequence started, you have to "guess" the initial point $\mathbf{s}^1$. You can pick it arbitrarily.

Thus, assume $\mathbf{s}^n$ has already been computed. In order to compute $\mathbf{s}^{n+1}$, the Metropolis algorithm executes two steps, each of which contains a cheap-to-compute random decision:

Step 1: randomly propose a candidate $\mathbf{s}^*$ for $\mathbf{s}^{n+1}$. For this step one needs to design and implement a mechanism to sample from a *proposal distribution*. Each instance of a Metropolis algorithm comes with such a mechanism. Mathematically, a proposal distribution is a conditional distribution over S which gives the probability to choose $\mathbf{s}^*$ given $\mathbf{s}^n$. Let us write $P_{\mathsf{prop}}(\mathbf{s}^* \mid \mathbf{s}^n)$ for the probability to pick $\mathbf{s}^{n+1}$.

The proposal distribution should be chosen such that one can sampled from it cheaply. For instance, it could be the uniform distribution on a small-scale hypercube centered on $\mathbf{s}^n$, or a multidimensional Gaussian distribution centered on $\mathbf{s}^n$. Efficient samplers exist for such elementary distributions.

Step 2: accept or reject the candidate $\mathbf{s}^*$. If $\mathbf{s}^*$ is *accepted*, it becomes the next output of the sampler, that is $\mathbf{s}^{n+1} = \mathbf{s}^*$. If $\mathbf{s}^*$ is *rejected*, it is discarded and the sampler repeats the previous value in its next output, that is $\mathbf{s}^{n+1} = \mathbf{s}^n$.

This requires a decision-making subroutine, whose choice is again random. Several such decision-making procedures, called *acceptance functions*, are known which result in sampling point sequences $\mathbf{s}^1, \mathbf{s}^2, \ldots$ which asymptotically re-model the "landscape" of the proto-pmf F. Here are the two most common and famous ones:

The Boltzmann acceptance function computes an *acceptance probability* by the formula

$$P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n) = \frac{F(\mathbf{s}^*)}{F(\mathbf{s}^*) + F(\mathbf{s}^n)}. \tag{51}$$

The Metropolis acceptance function computes the acceptance probability by

$$P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n) = \begin{cases} 1, & \text{if } F(\mathbf{s}^*) \geq F(\mathbf{s}^n) \\ \frac{F(\mathbf{s}^*)}{F(\mathbf{s}^n)} & \text{if } F(\mathbf{s}^*) < F(\mathbf{s}^n). \end{cases} \tag{52}$$

that is, the Metropolis acceptance function accepts the proposed candidate with certainty if its proto-pmf is larger than the one of the previous value, and it accepts it with probability $F(\mathbf{s}^*)/F(\mathbf{s}^n)$ if the F-value of the candidate is lower than the one of the previous sample point.

Both acceptance functions can be computed solely on the basis of the ratio $r = \frac{F(\mathbf{s}^*)}{F(\mathbf{s}^n)}$. The Boltzmann acceptance function can be re-written as

$$P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n) = \frac{r}{r+1},$$

and the Metropolis acceptance function as

$$P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n) = \begin{cases} 1, & \text{if } r \geq 1 \\ r & \text{if } r < 1. \end{cases}$$

The computation of $P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n)$ by an acceptance function is followed by a random decision that says "yes, accept, please" with the acceptance probability $P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n)$. This weighted random decision can be implemented by drawing a number a from the uniform distribution on $[0, 1]$, using the elementary sampler rand, and returning "yes" if $a \leq P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n)$.

I conclude this subsection with a few remarks to round of the picture.

- In order to guarantee that the Metropolis algorithm, as outlined above, indeed yields a sampling point sequence which correctly re-shapes the F-landscape in the long run, some additional conditions must be satisfied.
  - A *necessary* condition is that the "random walk" process which yields the sampling point sequences is *ergodic*. This is an involved concept from information theory which is beyond the scope of this course. In intuitive terms this means that every point in S can be reached by some sample point sequence at some time. This condition would be violated, for instance, if the F landscape is zero everywhere except at two "hills" H_1, H_2 which are separated from each other by a zero-F "lowland" of

width w, and the proposal distribution is chosen such that it always proposes a candidate $\mathbf{s}^*$ which is closer to $\mathbf{s}^n$ then w. If the sampling sequence is started in the hill H_1, it can never cross the flatland to H_2. Checking whether a Metropolis sampler that one has designed has the ergodicity property is nontrivial and there is no general recipe to assure this property.

  - Not every proposal distribution will yield a valid Metropolis sampler. Assuming ergodcitiy, a *sufficient* condition to obtain a proposal distribtution that makes a valid Metropolis sampler for F is to require that P_{prop} is symmetric, that is $P_{\mathsf{prop}}(\mathbf{s}^* \mid \mathbf{s}^n) = P_{\mathsf{prop}}(\mathbf{s}^n \mid \mathbf{s}^*)$. The two simple proposal distributions that I mentioned in Step 1 have this property.

- The proposal distribution is the heart of a Metropolis algorithm. If it is designed poorly, the sampling process will take a long time to "cover the grounds" of the probability landscape. Some thinking is needed to design a good proposal distribution. If $P_{\mathsf{prop}}(\mathbf{s}^* \mid \mathbf{s}^n)$ is allowed to often propose candidates $\mathbf{s}^*$ that are far away from $\mathbf{s}^n$, the danger is that one lands in a low-probability zone of the landscape, which means that the candidate is rarely accepted and the sample point sequence has many repeated points. The net effect is slow coverage and long required sampling times. If the proposal distribution mostly suggests candidates from the close vicinity of the previous point, the resulting random walk will also be too slow. Thus, the art lies in finding a proposal distribution which jumps far — but mostly hits candidates that have a high probability. Finding such a proposal distribution needs insight into the nature of the probability distribution that one wants to sample from.

- Often one wishes to create a sampler which produces an *independently, identically distributed* (i.i.d.) sequence of sample points $\mathbf{s}^1, \mathbf{s}^2, \ldots$. That is, the choice of point $\mathbf{s}^{n+1}$ should be statistically independent of the previous point $\mathbf{s}^n$ — the sampling point sequence should have no "memory". But the proposal / acceptance mechanism of Metropolis sampling (and any other MCMC technique) makes the choice of $\mathbf{s}^{n+1}$ depend on $\mathbf{s}^n$. The solution for this problem is to *subsample* the sequence generated by the sampler, that is, instead of using the original sequence $\mathbf{s}^1, \mathbf{s}^2, \ldots$, retain only each h-th point, recording only $\mathbf{s}^h, \mathbf{s}^{2h}, \ldots$.

- A commonly used special variant of Metropolis sampling (and other MCMC samplers) is called *Gibbs* sampling. It can be employed when the states $\mathbf{s} \in S$ are vectors $\mathbf{s} = (s_1, \ldots, s_d)'$. In Gibbs sampling, the proposal distribution changes only one component s_i of $\mathbf{s}$ at a time, cycling through the indices. That is, for $\mathbf{s}^n = (s_1^n, \ldots, s_d^n)'$, and change index i, the proposed candidate is of the form $\mathbf{s}^* = (s_1^n, \ldots, s_{i-1}^n, s_i^*, s_{i+1}^n, \ldots, s_d^n)'$.

- The Metropolis sampler with the Metropolis acceptance function harmonizes nicely with the Boltzmann distribution. Recall that a Boltzmann pmf is given by $P(\mathbf{s}) = \frac{1}{Z} \exp(-E(\mathbf{s})/T)$. It is easy to see that if $E(\mathbf{s}^*) \leq E(\mathbf{s}^n)$, the candidate $\mathbf{s}^*$ is accepted with certainty. If $E(\mathbf{s}^*) > E(\mathbf{s}^n)$, the ratio $r = \frac{F(\mathbf{s}^*)}{F(\mathbf{s}^n)}$ in (52) becomes

$$r = \frac{F(\mathbf{s}^*)}{F(\mathbf{s}^n)} = \frac{\exp(-E(\mathbf{s}^*)/T)}{\exp(-E(\mathbf{s}^n)/T)} = \exp(E(\mathbf{s}^n) - E(\mathbf{s}^*))^{1/T}. \tag{53}$$

  The acceptance probability depends only on the energy difference between the previous and the proposed microstate (and the temperature). That is, in order to carry out Metropolis sampling over a Boltzmann distribution defined by an energy E, all one needs to know is the energy; one can forget about the probabilistic framework around it.

6.4 Simulated annealing

It would be a waste to introduce the Boltzmann / Metropolis framework only for the purpose of discussing the Boltzmann machine. Metropolis sampling has many important applications other than that, and some of these are *very* important indeed. From the listing that I gave at the end of the introduction to this section, I pick simulated annealing.

Simulated annealing is a general-purpose optimization algorithm. An optimization task, in the most general setting, consists of a search space S and a cost function $R : S \to \mathbb{R}$. The goal is to solve the cost minimization problem

$$\mathbf{s}_{\mathsf{opt}} = \operatorname*{argmin}_{\mathbf{s} \in S} R(\mathbf{s}). \tag{54}$$

This is (please remember!) the form of the neural network trianing objective in supervised learning, where the states $\mathbf{s}$ would be parameter vectors θ of a NN and the cost function R would be the (empirical) risk.

But the minimization problem (54) is absolutely general and could, for instance, mean the task to minimize financial loss in stock market transactions; or to find a protein folding which minimizes the energy of the resulting 3D molecular structure (which is what nature does, and which gives rise to the proteins you are made of, and which is a major task in biochemstry research).

Solving (54) analytically is out of the question in many real-world optimization tasks. If the cost function is differentiable one can try to solve (54) by gradient descent, which is computationally often quite feasible but which can spot only local cost minima. If one wants to find the *global* minimum, one has to "search" through the entire search space S. A systematic grid search is infeasible if the states $\mathbf{s}$ are high-dimensional vectors because the number of grid points in a search sub-volume of $\mathbb{R}^d$ explodes exponentially with d. In this situation one needs a "clever" random search method which explores the search space in a way that low-cost candidates

are (much) more often tried than high-cost ones, while making sure that the search does not become trapped in some subvolume of the search space S but will visit all parts of it.

I know of only two main families of such "clever" stochastic search techniques, *evolutionary optimization* with the popular special case of *genetic algorithms*, and *simulated annealing*. Both approaches are inspired by nature:

Evolutionary optimization mimics how natural evolution finds "solutions" (= species) that are highly adapted to their ecological niches. The cost function here is (the inverse of) biological *fitness*. The main principle of evolutionary optimization is to compute a sequence of "generations" $G^1, G^2, \ldots$, where a generation is a set of individual solution candidates (animals or plants in the biological world, network parameter vectors θ for us) $G^n = \{s_1^n, \ldots, s_N^n\}$. The next generation is derived from the previous generation by some "procreation" mechanism which favors parents that have a high fitness and introduces some random variation in the offspring generation. See `https://en.wikipedia.org/wiki/Evolutionary_computation` for an introduction.

Simulated annealing (SA) mimics the behavior of a lump of material which is slowly cooled down from its gaseous phase until it crystallizes. The cost function is the energy of microstates. If you take another look at Figure 35 you will see that at low temperatures the Boltzmann distribution concentrates around the global minimum of the energy (cost) landscape. The core idea of SA is to consider the cost function as an energy function, then run an extended Metropolis sampling from the Boltzmann distribution associated with that energy function, starting with a high temperature (which facilitates an exploration of the entire state space), then slowly cooling down which will nudge the search toward the global minimum. The naming of this procedure, simulated *annealing*, comes from metallurgy where a slow cooling of heated metals leads to the formation of large crystals which harden the material.

I'll now give a more detailed description of SA, again for the case of discrete search spaces where the Boltzmann distribution is characterized by a pmf. Here is an outline of the complete process:

1. Identify the points of the search space S with microstates $\mathbf{s}$ of an artificial "thermodynamical" system.

2. Identify the cost function, of which a global minimum should be found, with the energy function $E(\mathbf{s})$.

3. Start with some medium or high temperature T_0 and consider the Boltzmann distribution $P(\mathbf{s}, T_0)$, which will be close to uniform (as in the first panel

of Figure 35). Start sampling from this distribution with the Metropolis algorithm. The sequence of created samples will cover the search space almost uniformly.

4. Now lower the temperature gradually, thereby obtaining a sequence of Boltzmann distributions $P(\mathbf{s}, T_0), P(\mathbf{s}, T_1), P(\mathbf{s}, T_2), \ldots$ which more and more concentrates around the microstates that have low cost/energy values. Continue sampling all the time. The sequence of samples should thus concentrate more and more on low-cost microstates.

5. Continue until $T_n \approx 0$. The hope is that then the cooling process has guided you toward the global minimum and that the samples that you now get are closely scattered around that global minimum.

A natural question at this point is, why not start immediately at low temperatures (e.g. in a situation like that shown in the 4th panel of Figure 35), wouldn't that just save the time of "bouncing aimlessly" around in the search space at high temperatures, and instead directly lead you to the desired minimum, which is well pronounced at low temperatures? The answer is, if one starts at low temperatures — or, for that matter, if one cools to rapidly — one is likely to get "frozen" in a very suboptimal local minimum far from the best one, from which one cannot escape. This becomes intuitively clearer if we re-interpret the Metropolis sampling of $P(\mathbf{s}, T_n)$ in physical terms of "jumping around" in the energy landscape $E(\mathbf{s})$ directly.

To see this, we consider the two cases when (A) the Metropolis algorithm accepts with certainty, and (B) when it accepts with probability $P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n) = \frac{P(\mathbf{s}^*,T)}{P(\mathbf{s}^n,T)}$, and see how we can translate this into energies.

Case A: This case occurs when $P(\mathbf{s}^*, T) \geq P(\mathbf{s}^n, T)$. This is equivalent to the condition $E(\mathbf{s}^*) \leq E(\mathbf{s}^n)$. Thus, whenever the energy of the proposed state is lower than the energy of the previous state, accept with certainty.

Case B: If $P(\mathbf{s}^*, T) < P(\mathbf{s}^n, T)$, then $P_{\mathsf{accept}}(\mathbf{s}^* \,|\, \mathbf{s}^n) = \frac{P(\mathbf{s}^*,T)}{P(\mathbf{s}^n,T)}$. Rewriting the log of this acceptance probability in terms of energy gives

$$\begin{aligned} P_{\mathsf{accept}} &= \frac{P(\mathbf{s}^*, T)}{P(\mathbf{s}^n, T)} = \frac{\exp(-E(\mathbf{s}^*)/T)}{\exp(-E(\mathbf{s}^n)/T)} \\ &= \exp((E(\mathbf{s}^n) - E(\mathbf{s}^*))/T) \\ &= \exp(\Delta E/T), \end{aligned} \tag{55}$$

where $\Delta E < 0$ is the energy difference between $\mathbf{s}^n$ and $\mathbf{s}^*$.

Summarizing we see that in terms of energy, a new proposed microstate is accepted with certainty if its corresponding energy jump goes "downhill", and if it goes uphill, it is accepted with probability $P_{\mathsf{accept}} = \exp(\Delta E/T)$. That is,

the greater the energy increase, the accept (exponentially) more unlikely is such a step taken; however, this may be compensated by a proportional increase in temperature. In other words, on the average, at higher temperatures we may take higher jumps uphill.

Equipped with this re-interpretation of the Metropolis algorithm in terms of an energy-based acceptance function, we can better understand why slow cooling is important for a final landing in a good local minimum of the energy landscape. We can now intuitively interpret the SA search process as a random jump sequence of a "search ball" in the energy landscape $E(\mathbf{s})$, where the temperature determines the ability of the ball to (randomly) climb uphill and in this way overcome "energy barriers". Figure 37 illustrates the different behavior of the SA search process at different temperatures.

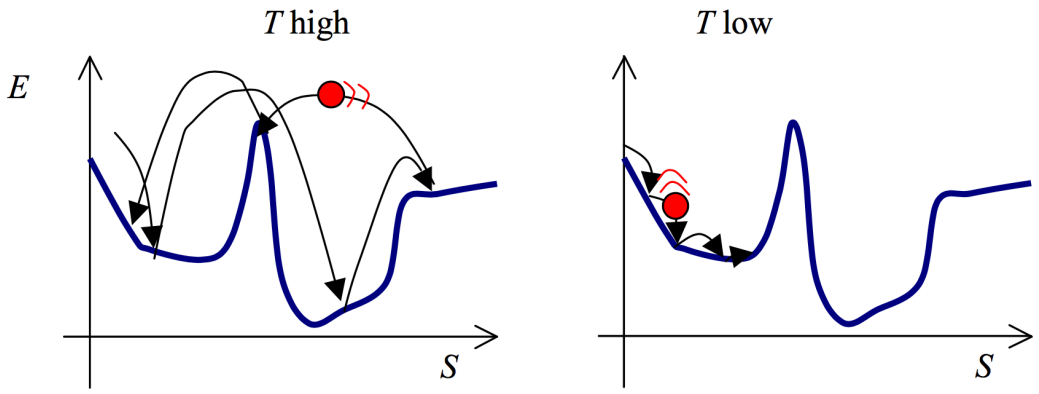

Figure 37: SA seen as an energetic ball game. At high temperatures, the ball can more easily jump high and overcome energy barrier than at low temperatures.

We are now aware that the cooling process is important for the success of running an SA algorithm. A widely used, quick and dirty cooling scheme is *exponential cooling*: Put $T_{n+1} = k\,T_n$ for some $k < 1$ close to 1. Update T after every single sample point. Clearly the size of k is important a typical way to determine it would be just to experiment.

However, such simple cooling schemes, although widely used, may yield unsatisfactory results. During a SA search run, one may encounter periods where a particularly slow cooling is required, while at other periods, one may cool faster. I will first explain this fact intuitively and then give a formal account.

Here are two intuitive examples that illustrate the necessity for slower-than-others cooling periods. The temperatures where particularly slow cooling is required are associated with *phase transitions*, like when water freezes at temperature $T = 273.15$ Kelvin.

The original physical metaphor: simulated annealing. In metallurgy and

chemistry, "annealing" refers to a process of carefully cooling a liquid into crystallisation. A general observation is that if a liquid is cooled very quickly, the resulting solid will consist of many fine-granular microcrystals. By contrast, if the cooling is done slowly, large crystals or even a single solitary crystal result. Specifically, the cooling must be slow (heat being withdrawn from the liquid at a low rate) at the solidification temperature, because it is at this temperature that the final crystal structure is determined. You might know from own experience with deep-freezing condiments or producing ice-cream that fast cooling across the solidification temperature produces a substrate structured into many small crystals, whereas slow cooling results in fewer and larger crystals. This is important in both ways in many applications: for instance, when deep-freezing biological specimens (seeds, live cells) it is crucial to shock-freeze the material very rapidly, avoiding the growth of larger ice crystals which would destroy cell membranes; or in the industrial production of silicon wafers needed for microchip production, an almost or perfectly monocrystalline block of solid silicon is very slowly pulled out of the melted mass. In terms of energy landscape: large crystals corresponds to microstates of low energy (with a monocrystalline block corresponding to globally minimal-energy states). Therefore, slow cooling around the critical solidification temperature is a prerequisite for low-energy final products. When the molten material is still significantly hotter than the solidification point, it can be cooled fast; similarly, once it is solidified, further cooling will change the crystal structure only a extremely slow timescales which make further cooling practically meaningless.

Naive human problem solving. When solving some magazine puzzle question or a math homework, you will probably have experienced something similar to a phase transition. After an initial thinking phase where you have no clue of how to solve the problem and think of many possible approaches (= a high temperature search phase), the inklings of a solution appear on the horizon (= close to the phase transition), and suddenly! you lock in to a particular approach (= beyond the transition) from which it would require quite some effort (= mental re-heating) to escape. If you cooled to quickly (= decided for a solution strategy too early, too quickly) your approach is likely to fail (= lead to a quite suboptimal minimum); if you spent time to consider different solution options (= hover around the critical temperature) and then sloooowly decided, your chances of hitting a good solution strategy are much higher.

There is a physical / mathematical indicator of such phase transition temperatures when it is important to cool very slowly. I use concepts from thermodynamics without further explanation. The remainder of this section is optional reading.

The *free energy* of a system at temperature T is

$$F(T) = E_T[E] - T\,\mathcal{S}(T),$$

where $E_T[E]$ is the average energy at temperature T,

$$E_T[E] = \sum_{\mathbf{s} \in S} E(\mathbf{s}) \ \frac{1}{Z} \ \exp(-E(\mathbf{s})/T)$$

and $\mathcal{S}(T)$ is the *entropy* of the system at temperature T,

$$\mathcal{S}(T) = -\sum_{\mathbf{s} \in S} P(\mathbf{s}, T) \ \log(P(\mathbf{s}, T)).$$

Thus the free energy relates the average energy at temperature T with the entropy. Intuitively, the free energy of a system is the "useable" part of its energy, energy that could be exploited at a macroscopic scale (for example, the free energy of a volume of gas would be the energy that one could exploit by expanding the gas in a piston, plus the energy that one might gain from cooling the volume). Phase transitions are defined in physics as discontinuities in the free energy (or one of its derivatives) as temperature (or another macroscopic variable) passes a critical value (check out `https://en.wikipedia.org/wiki/Phase_transition`). For instance, as a volume of water is cooled from some ε value above zero Celsius to some ε value below, such that it freezes, one has to extract a certain amount of energy (the melting heat) from that volume one may in fact exploit this energy; it is part of the free energy of the volume of water. Therefore, the free energy of the volume of water just above zero jumps discontinuously to a lower value as the water is cooled to a temperature just below freezing.

Similarly, when running SA for an optimization problem, one can in principle compute, at every step n, the free energy F_n of the system, and make the cooling rate depend on the development of the free energy: cool slowly when F_n shows signs of changing rapidly, or in other words, cool in a fashion such F_n decreases smoothly.

The free energy can be computed from the partition function Z by $F(T) = -T \log(Z(T))$. So the question is, how can one compute the partition function, which is a gigantic sum (for discrete systems) or an intractable integral (for continuous systems)? The brutal answer is: use sampling (!) for an approximate evaluation of this integral. There are a number of specialized sampling procedures for the partition function, surveyed in Neal (1993) and Chapter 18 of Goodfellow et al. (2016). This method of steering the cooling is obviously very expensive: within an SA run (which may have millions of steps), we repeatedly have to squeeze in complete auxiliary sampling runs.

It can be mathematically shown that if the cooling is done slower, on average across different SA runs one ends in lower-energy minima of the energy landscape. In the limit of infinitesimally slow cooling, SA is guaranteed to find a global minimum.

I conclude this section with two examples. The first one is in many ways representative for many SA applications in *combinatorial optimization* tasks; the second is just for fun.

6.4.1 Optimizing computer hardware layout: circuit partitioning subtask

The first example is taken from the pioneering paper on SA Kirkpatrick et al. (1983) (47K Google Scholar cites). According to Kirkpatrick et al., whose paper I closely follow in this subsection, in computer hardware layout one is confronted with a number of subproblems that build on each other, with *circuit partitioning* being the most elementary (the article also treats the subsequent optimizations of metrical placement and wiring). In the example described in Kirkpatrick et al. (1983), the partitioning task is to distribute a set of about 5000 elementary computational circuits (together forming a complete CPU architecture) over two microchips such that (i) the number of pins is low and (ii) the circuits are distributed approximately in equal numbers across the microchips.

Formally, this task can be specified through describing possible distributions of the 5000 circuits on the two chips by microstates $\mathbf{s}$, where each microstates $\mathbf{s}$ is 5000-dimensional binary vector with entries $\{-1, +1\}$. The i-th entry s_i is set to -1 if the i-th circuit is assigned to the first chip, and it is set to $+1$ if it is assigned to the second chip. The cost function must reflect the two requirements (i) and (ii). Expanding on the treatment in the original article (it provides no detail), this can be done as follows:

For (i), consider a symmetric 5000×5000 matrix (a_{ij}) with 0-1 entries, a value of $a_{ij} = 1$ indicating that circuits i and j directly exchange a signal (and hence, if placed on different chips, require a pin at each chip). For a given partitioning-encoding microstate $\mathbf{s}$, the number of signals that must cross between the two chips is

$$\sum_{i>j} \frac{a_{ij}}{4} \, (s_i - s_j)^2 = -\sum_{i>j} \frac{a_{ij}}{2} s_i \, s_j + \sum_{i>j} \frac{a_{ij}}{2}.$$

The second sum term is independent of the circuit placement and can be dropped from the energy function, because it does not affect the location of its minima.

For (ii), the objective function should grow with the degree of imbalance of circuits assigned to the two chips. The squared imbalance score $(\sum_i s_i)^2$ is equal to $2 \sum_{i>j} s_i \, s_j + \sum_i s_i^2$. Again, the second term is independent of the placement and can be dropped.

Assembling these two cost contributions and replacing the constant 2 by a weighting factor λ one gets a cost/energy function of the form

$$E(\mathbf{s}) = \sum_{i>j} \lambda \, s_i \, s_j - \sum_{i>j} a_{ij} \, s_i \, s_j.$$

The SA scheme used in Kirkpatrick et al. (1983) involved a proposal distribution that simply flipped the assignment of a randomly chosen circuit. The temperature was lowered with a factor of 0.9 from one temperature to the next lower one. Starting at $T = 10$, at each temperature in the order of 500,000 flips

were executed, until a temperature of 0.1 was reached (from which I infer a total runlength of about 20,000,000 updates). Figure 38 summarizes the distributions of number of pins obtained at different temperatures. As expected, the average number of pins sampled at decreasing temperatures decreases, as does the variance of that number. At the lowest temperature, the sampling has frozen into apparently a single (or very few) solution(s).

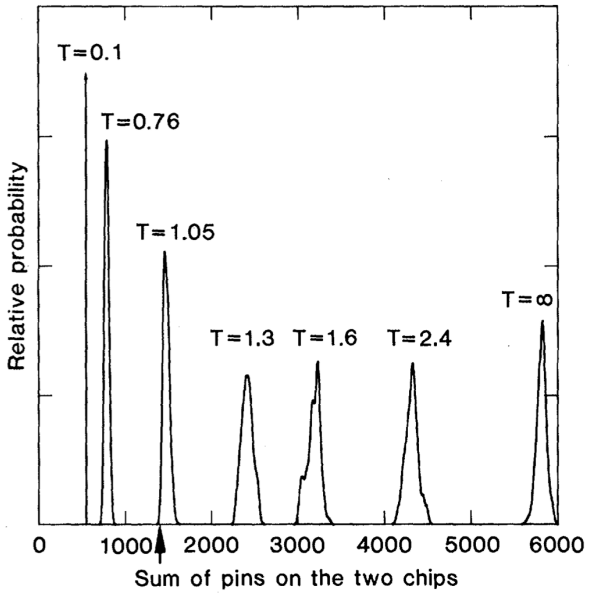

Figure 38: Distributions of total number of pins at various temperatures. The arrow inserted at the x-axis indicates the average number of pins obtained by a greedy search algorithm (Metropolis sampling with T = 0). Figure taken from the Kirkpatrick et al paper.

6.4.2 Laying a jigsaw puzzle

In my 2012 course "Statistical and Algorithmical Modeling", a miniproject that I gave was to use SA to re-assemble the fragments of a shattered photographic image into the original photo (not knowing the original!). Figure 39 shows a solution from Ivaylo Enchev and Corneliu Prodescu. The two key design ingredients to set up SA were to find a suitable energy function and a good proposal distribution. All students used some measure of geometric / color agreement between the edge regions of neighboring "tiles" as a basis for the energy function (graphical mismatch = high energy). For the proposal distribution, the Enchev/Prodescu team opted for a weighted random choice of a pair of tiles which then would be swapping places. The weighting encouraged to choose swapping candidates that had a significant graphical mismatch with their neighbors.

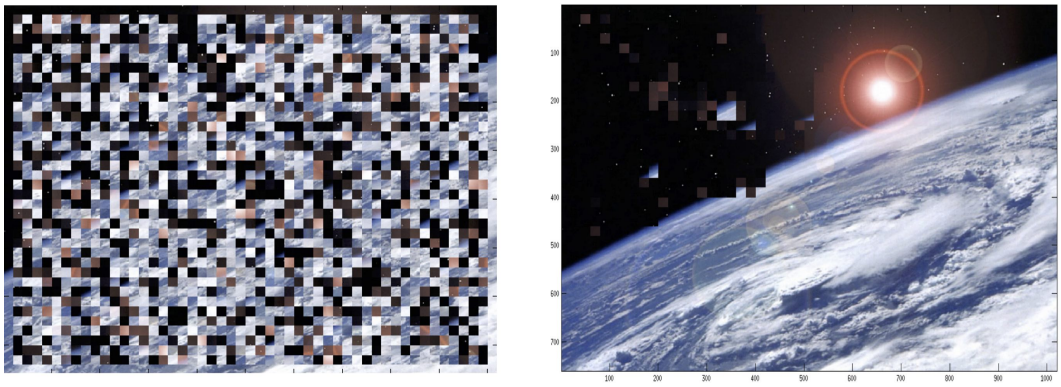

Figure 39: Unscrambling a photo that was shattered into 1938 fragments. Left: the mess that I gave out to students. Right: a low-energy re-ordering found by running SA for 24 hrs on a PC. Right picture taken from the project report of Ivaylo Enchev and Corneliu Prodescu (unpublished).

7 The Boltzmann machine

The Boltzmann Machine (BM), introduced in 1983 by Hinton and Sejnowski (Hinton and Sejnowski (1983), didactic introduction in Ackley et al. (1985)), is not actually a machine but a neural network model for a representation of complex distributions — or, stated in the terms of cognitive science, which is the appropriate background: a model of a contents-addressable, associative, generative long-term memory.

Before going into the technical aspects, I want to explain the background intuitions and the intended use of BMs. A good way to approach BMs is to see them as an abstract model of human memory, including mechanisms for storing, retrieving, associating between stored items, and completing corrupted inputs. Let us consider the question what is the nature of a human's long-term memory of the written digit pattern for the number "four" (in the Times bold font: **4**). Numerous and diverse answers to this question have been given in the fields of cognitive science and neuroscience, for instance:

- The digit-pattern-4-memory is a stored *prototype*, that is, we have a conceptual/neural representation of some ideal, prototypical pattern, – let's say, something very clean and clear that looks like this:

  

  In order to recognize new incoming instances of the digit "four", like this one:

the stored prototype is matched against the new input, which is classified as "four" if the match is close enough. The prototype view is one of the classical models of memory in cognitive science.

- The digit-pattern-4-memory is a set of processing rules, which specify how low-level features which are extracted from a visual input can (and must) be combined in order to be recognized as a "four" pattern. This is another classical model, especially in AI and pattern recognition.

- The digit-pattern-4-memory is a huge set of contextual expectations (or *anticipations* or *affordances*) which specify in which contexts one should expect the pattern "four" to appear, and when it appears, which further actions or perceptions are likely to occur. This would be the approach of the rather recent and flourishing school of thought of *anticipatory representations*.

This is only a selection among numerous other models of the nature of long-term memory. The most confusing part of this picture is that for each kind of model there is good empirical evidence from psychological or neurophysiological studies.

The BM should be seen on this background of a multitude of models of long-term memory (LTM), because it adds another such model. The fundamental assumption of the BM is that memory is a *generative model of a probability distribution*. Coming back to the pattern "four" example, our memory of this pattern should be seen as a distribution over possible variations of that pattern. A sample from this distribution might look like in Figure 40.

Figure 40: A sample from a distribution of the pattern "four" (taken from the widely used MNIST digit benchmark dataset).

The BM model of memory is *generative*. In technical terms this means that the BM comes complete with a sampling algorithm, which allows it to produce sample items from the memorized distribution. In more intuitive terms one could say that the BM can be run in a mode of "hallucination" or "dreaming" the technical term that is mostly used is to say that the BM can *confabulate* pattern samples.

7.1 Architecture

A BM is a recurrent NN whose neurons are all binary, that is, every neuron can have an activation of 0 or an activation of 1. Neurons can be either *visible* or they can be *hidden*. As we will shortly see, the visible neurons can be very flexibly used in various ways (in the same BM) for input and output, whereas the hidden units add internal computing power to the achievable input/output mappings.

We will use the following notation. For a BM with L visible units, $\mathbf{v} = (v_1, \ldots, v_L)' \in \{0,1\}^L$ is an activation vector of the visibles; and if the BM has M hidden units, $\mathbf{h} = (h_1, \ldots, h_M)' \in \{0,1\}^M$ is the activation vector of the hidden units. Often we do not want to distinguish between these two sorts and write $\mathbf{s} = (v_1, \ldots, v_L, h_1, \ldots, h_M)' =: (s_1, \ldots, s_{L+M})' \in \{0,1\}^{L+M}$ for the entire network state. Always the first L members of an entire network state will be reserved for the visible units.

In a BM, there is an *undirected* synaptic link between any two visible and/or hidden units s_i, s_j. Each link has a real-valued *weight* $w_{ij} = w_{ji} \in \mathbb{R}$. Special case: a zero weight $w_{ij} = 0$ amounts to "no link between units i and j". Self-connections $w_{ii} \neq 0$ are not allowed.

Thus, in summary, a BM is fully characterized (i) by its symmetric weight matrix $\mathbf{W}$ of size $(L+M)\times(L+M)$ with a zero diagonal and (ii) by the specification which of the units are visible, that is by the number L.

A BM can be used for many purposes. Among others, it can be used for the same tasks as MLPs, namely the supervised training of pattern classification tasks. I now work this use case out a little more.

In the supervised learning setting, the training data are $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\ldots,N}$ where the $\mathbf{u}_i \in \{0,1\}^{L_\mathbf{u}}$ are binary pattern vectors and the teacher outputs $\mathbf{y}_i \in \{0,1\}^{L_\mathbf{y}}$ are binary classification vectors in one-hot encoding. The training objective is the same as we know it from MLPs: Upon input of a new testing pattern $\mathbf{u}_{\mathsf{test}}$ from pattern class j, the j-th output unit should become activated to a value of 1 and the other output units should stay at zero activation. The inputs $\mathbf{u}_i$ and outputs $\mathbf{y}_i$ are assigned to the visible units of the BM, which means that $L = L_\mathbf{u} + L_\mathbf{y}$.

And here's the first amazing new thing about BMs that makes them so different from MLPs. A BM that was trained on a pattern classification task can also be run backwards in a generative confabulation mode. For instance, after it has been trained to classify the ten classes of handwritten digits $1, 2, \ldots, 9, 0$, one can "clamp" the ten classification neurons in one class, for instance in the class of the digit 4, by fixing the output neurons to the state (0, 0, 0, 1, 0, 0, 0, 0, 0, 0). In other words, the ten classification units are now used as input units. Then, examples of the digit 4 patterns will be *generated* on the input layer. This generation of examples will be driven by Metropolis sampling, that is, you will see a random sequence of various patterns "4" appearing on the "input" layer. This sequence of patterns "4" is a (Metropolis generated) sample from the learnt distribution of this pattern class. The examples shown in Figure 40 might well have been collected from the input layer while the classification layer was clamped to

(0, 0, 0, 1, 0, 0, 0, 0, 0, 0).

The "input" layer can thus be interpreted/used sometimes as an input "retina" when the BM is run in classification mode, or at other times as an output projection screen when it is run in confabulation mode. Likewise, the classification layer with its ten class-coding neurons can be seen as output layer in classification mode, and as input layer in the confabulation mode. This is why it is common thinking and terminology in the BM world to drop the distinction between input and output layers. Instead, the neurons in these layers are just called visible neurons. Figure 41 gives a schematic view of the structure of a BM that is structured as indicated in this digit classification / generation set-up.

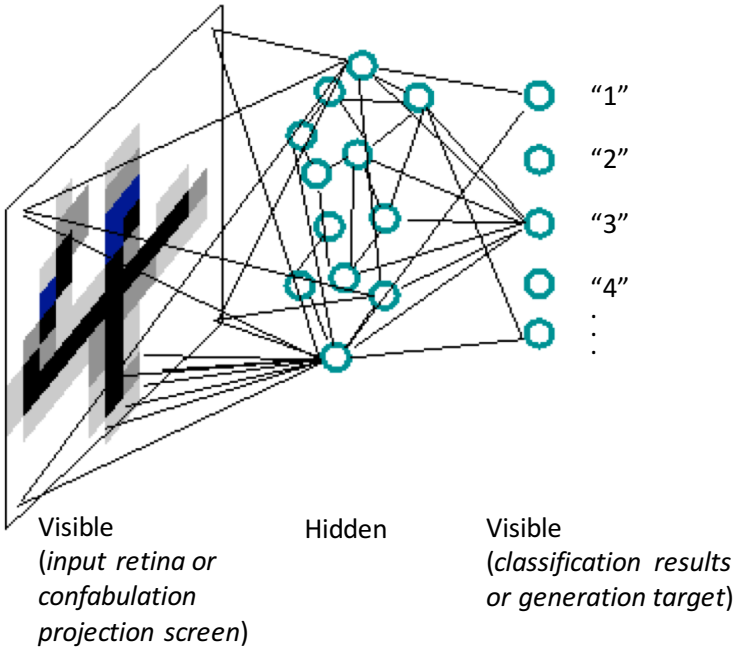

Figure 41: Schematic of a Boltzmann machine.

7.2 The stochastic dynamics of a BM

A BM "runs" by sampling states with the Metropolis sampler. That is, the temporal evolution of the BM state is staged in the format of a random sequence $\mathbf{s}^1, \mathbf{s}^2, \ldots$ of binary network states which is generated by an application of Metropolis sampling.

Metropolis sampling needs an energy function. In a BM, the energy of a state $\mathbf{s} = (s_1, \ldots, s_{L+M})'$ is defined by

$$E(\mathbf{s}) = -\sum_{i<j} w_{ij}\, s_i\, s_j. \tag{56}$$

In plain English, this energy function gives low values if pairs of neurons s_i, s_j which both have an activation of 1 are connected by large positive weights. Conversely, the energy is high when such s_i, s_j are connected by strong negative weights. Any unit s_i that has zero activation does not contribute to the energy of the network state.

If a unit s_i jumps from an activation of 0 to an activation of 1, while all other units s_j retain their activation, the energy (56) changes by adding the amount

$$-\Delta E_i = -\sum_j w_{ij}\, s_j. \tag{57}$$

This energy function gives rise to the Boltzmann distribution over the set $S = \{0,1\}^{L+M}$ of all possible states, which has the pmf

$$P(\mathbf{s}) = \frac{1}{Z}\, \exp(\frac{-E(\mathbf{s})}{T}). \tag{58}$$

In most usages of a BM, the temperature is not changed and can be fixed at an arbitrary value, typically $T = 1$, which simplifies (58) to $P(\mathbf{s}) = 1/Z\ \exp(-E(\mathbf{s}))$.

Thus each particular fixed setting of the weights defines a particular energy landscape and hence, a particular probability distribution over the microstates. We write $E_{\mathbf{W}}, P_{\mathbf{W}}$ for the energy / probability distribution induced by a weight matrix $\mathbf{W}$.

Sampling from the distribution $P_{\mathbf{W}}$ is achieved with a Gibbs version of the Metropolis sampler using the Boltzmann acceptance function. The sampler cycles through the components s_i of the states $\mathbf{s}$, updating only the selected component to get a new sample point. This ansatz results in the following update rule:

When unit s_i is chosen for update from iteration n to $n+1$, set it to a value of 1 in generation $n+1$, regardless of its value at generation n, with the probability

$$P(s_i^{n+1} = 1 \,|\, \mathbf{s}^n) = \frac{1}{1 + \exp(-\Delta E_i/T)}, \tag{59}$$

where $-\Delta E_i$ is the energy increment from (57). Proving that this rule is an instantiation of the Metropolis acceptance function is a recommended, elementary exercise (*hint:* start from considering the conditional probabilities $P(s_i = 1 \,|\, \text{state of all other units}), P(s_i = 0 \,|\, \text{state of all other units})$; note that the ratio of these two probabilities is the same as the ratio $P(\mathbf{s}')/P(\mathbf{s})$ where $\mathbf{s}'$ is the same as $\mathbf{s}$ except for unit i where $\mathbf{s}'$ has a value of 1 and $\mathbf{s}$ has a value of 0; exploit that $P(s_i = 1 \,|\, \text{state of all other units}) + P(s_i = 0 \,|\, \text{state of all other units}) = 1$.)

7.3 The learning task

A BM is trained to learn a probability distribution $P_0(\mathbf{v})$ over the visible units. The training data consist in a sample $S = (\mathbf{v}_i)_{i=1,\ldots,N}$ of patterns sampled from that target distribution. In formal terms, the training objective is to find weights

$\mathbf{W}$ such that, if the trained network is run with the Metropolis sampler, the distribution of the patterns that can be read off the visible units is a good approximation of the target distribution:

$$P_{\mathbf{W}}(\mathbf{v}) \approx P_{\mathsf{target}}(\mathbf{v}).$$

An interesting special case occurs when the L-dimensional training data vectors $\mathbf{v}$ are split into an $L_{\mathbf{u}}$-dimensional "input" part and an $L_{\mathbf{y}}$-dimensional "output" part, as in pattern classification tasks. The training data vectors $\mathbf{v} = (\mathbf{u}', \mathbf{y}')'$ are then composed of two parts, an "input" part $\mathbf{u}$ and an "output" part $\mathbf{y}$. The BM learns a distribution $P_{\mathbf{W}}(\mathbf{v})$ over all visibles which approximates the joint distribution of the input and output vectors. After training, such a BM can be run in two ways:

1. The input part of the visibles can be fixed ("frozen", "clamped") to a test input $\mathbf{u}_{\mathsf{test}}$. When the Metropolis sampler is then launched, it is not allowed to change the activations of the "input" units, but it does its cyclical update job on all other units, including the "output" units. In our exemplary digit classification scenario, there would be ten such "output" units. Assume (again!) that a handwritten instance of the digit "4" is clamped on the input units. As the Metropolis sampling proceeds, monitoring the sequence of 10-dimensional output states should reveal that the fourth output unit is active much more often than the other ones. The relative frequency of activations of the fourth output unit indicates the "belief" that the BM thinks the input is from class "4".

2. Conversely (staying with the digits example), the ten classfication units can be clamped to a classification vector, say to (0, 1, 0, 0, 0, 0, 0, 0, 0, 0) (for a change, not for class "4"). The sampling process should then lead to a sequence of random variations of handwritten-like "2" patterns on the "retina" part of the visible units.

A beautiful movie demonstration of exactly this kind of digit classifier / confabulator can be found at Geoffrey Hinton's webpage at `http://www.cs.toronto.edu/~hinton/digits.html`. The architecture used there has two more hidden "layers" than sketched out in Figure 41 but is otherwise similar. The gray level rendering of the retinal visible units in these online movies is obtained by not plotting states (which would be binary black-and-white), but instead a suitably normalized version of the unit's energy contribution ΔE_i.

For a well-defined learning algorithm one needs to have a well-defined loss function. In the BM scenario, the loss function should measure how close a BM distribution $P_{\mathbf{W}}(\mathbf{v})$ is to the target distribution $P_{\mathsf{target}}(\mathbf{v})$. This measure should be zero if the two distributions are identical, and it should be greater than zero otherwise. The standard choice in statistical modelling of a measure that compares two discrete probability distributions $P, \hat{P}$ is the *Kullback-Leibler distance*

(or *Kullback-Leibler divergence*, mostly referred to by its acronym just as *KL divergence*:

$$KL(P, \hat{P}) = \sum_{\mathbf{s} \in S} P(\mathbf{s}) \ \log \frac{P(\mathbf{s})}{\hat{P}(\mathbf{s})}. \tag{60}$$

The KL distance is not actually a true distance because it is not symmetric. It is always nonnegative and it is zero if and only if $P = \hat{P}$.

Thus, the learning task becomes to solve the problem of minimizing the empirical risk associated with this loss function: find

$$\mathbf{W}_{\mathsf{opt}} = \underset{\mathbf{W} \in \mathbb{R}^{(L+M)\times(L+M)}}{\operatorname{argmin}} KL(P_{\mathsf{target}}, P_{\mathbf{W}}). \tag{61}$$

7.4 The learning algorithm

Given a training sample $S = (\mathbf{v}_i)_{i=1,\ldots,N}$ and a BM architecture with the right number of visible units, a weight matrix $\mathbf{W}_{\mathsf{opt}}$ must be computed which solves the optimization problem (61). Since the KL divergence $KL(P_{\mathsf{target}}, P_{\mathbf{W}})$ is differentiable with respect to the weights w_{ij}, gradient descent optimization can be used.

The beauty of BMs lies in the circumstance that the formula that gives us the gradients is very simple. A little non-deep maths (Ackley et al., 1985) reveals that

$$\frac{\partial\, KL(P_{\mathsf{target}}(\mathbf{v}), P_{\mathbf{W}}(\mathbf{v}))}{\partial\, w_{ij}} = -\frac{1}{T}(p_{ij} - q_{ij}), \tag{62}$$

where p_{ij} is the average (over training samples) probability that the units i and j are both active (that is, $s_i = s_j = 1$) when the visible units are clamped to the training data point $\mathbf{v}$, and q_{ij} is the probability that these two units are simultanously active in a "free-running" sampling mode with no external clamping. This yields the following update rule for weights:

$$w_{ij}(n+1) = w_{ij}(n) + \lambda\,(p_{ij} - q_{ij}), \tag{63}$$

where $w_{ij}(n)$ is the value of the weight wij at the n-th step of gradient descent and λ is a learning rate.

A single weight update $w_{ij}(n) \rightarrow w_{ij}(n+1)$ thus involves the following operations:

1. Estimation of p_{ij}: for each training sample point $\mathbf{v}_k$, clamp the visible units to $\mathbf{v}_k$. While the visible units remain clamped to $\mathbf{v}_k$, run the BM Metropolis sampler (59) until a reasonably representative sample of network states under this clamping condition has been collected. Use this collection to estimate the probability p_{ij}^k of co-activity of units i and j in this clamping condition. Do this for all $k = 1, \ldots, N$. Finally, set p_{ij} to the average of all of these p_{ij}^k. All of this is sometimes called the "wake" phase of the BM learning algorithm (when it "sees" the visible input).

2. Estimation of q_{ij}: Similar, only without clamping the visible units. This is sometimes called the "sleep" phase (the BM has its eyes closed).

3. Weight update: apply (63).

It is clear what the catch is: for a single weight update step, one has to run as many complete sampling runs as there are training patterns! At face value, this is prohibitive. In their original paper, Ackley et al. (1985) introduce a number of simplifications which allowed them to learn some demo examples, even with the computers of the mid 80'ies. The main simplifications are the following:

1. Instead of (63), use $w_{ij}(n+1) = w_{ij}(n) + \lambda\,\mathsf{sign}(p_{ij} - q_{ij})$, where sign is the signum function. This reduces computational load because determining the sign of $(p_{ij} - q_{ij})$ needs a less accurate sampling than estimating the size of this difference.

2. For each sampling run in steps 1 or 2 above, use a two-phase procedure. In the first phase, start from a higher temperature than the agreed T and carry out a simulated-annealing like cooling from the higher temperature to T. In the second phase, sample at the target temperature T. The initial annealing phase is intended to prevent that the (randomly generated) starting state remains stuck in some narrow, untypical local energy minimum.

Despite these simplifications and heuristics, the intrinsic computational challenge of a very large number of sampling runs needed to determine weight updates is not fundamentally dissolved. This is probably a good enough reason to explain that BMs were never used in practical applications.

7.5 The restricted Boltzmann machine

The maths underlying the classical BM is both so simple and so potentially powerful that Geoffrey Hinton and other academic researchers continued to research these architectures in the decades since 1985. I probably wouldn't have elected BMs for this lecture nonetheless, if not many researchers in machine learning (including myself) have freshly become excited about BMs. This stir was triggered by the Science paper Hinton and Salakuthdinov (2006), where several developments which started from BMs were combined into a strikingly powerful architecture for learning complex distributions, now named *restricted Boltzmann machines* (RBMs) or *deep belief networks* (DBNs):

- DBNs are layered neural networks, where each layer corresponds to one BM. The hidden units of one such BM make the visible units of the next-higher BM.

- The connectivity of the participating layer BMs is very much reduced: there are no within-layer connections, only connections between adjacent layers exist. This led to the name "restricted" Boltzmann machines.

- The learning is done in a divide-and-conquer fashion layer by layer.
- Each sampling subroutine for the estimation of the probabilities p_{ij}, q_{ij} is condensed to only two state updates of one layer BM, using a shortcut approximate algorithm called *contrastive divergence*.

These innovations together have brought BMs back on stage with a flourish, which in turn has triggered the deep learning revolution. I can't possibly describe DBNs better (nor more concisely) than Hinton & Salakhutinov did in their celebrated Science paper, so if you are interested in digging deeper into the deep roots of deep learning, that paper is a must-read for you (only 3 pages).

Hinton and Salakhutinov did not foresee or plan what is now called deep learning. Their paper concentrates on a use of RBMs for data compression. Only in a 14-line sideline paragraph they mention that they could use RBMs for pre-training (initializing) the weights of a multilayer perceptron, giving a good starting point for the subsequent application of the standard backpropagation algorithm. The rest is (already) history.

DBNs remained fashionable for some years after 2006, and there were many sophisticated attempts to lift them to a generally useful, stand-alone learning approach for real-world modeling tasks. In fact, — at least, in my personal recollection — the entire machine learning community was thrilled about DBNs at that time (check out the 2007 youtube video `https://www.youtube.com/watch?v=AyzOUbkUf3M` to see Hinton himself presenting the BM in a Google talk). In the end, however, the successes of deep learning schemes which used cheaper initialization schemes than DBNs won over.

8 Reservoir computing

Reservoir computing (RC) is an umbrella term for a number of closely related approaches for designing and training recurrent neural networks. RC methods are in many ways complementary to the RNN methods which we met in Section 4. While the computational and learning algorithms differ, the *tasks* solved by RC networks are however the same as those that are solved by the RNNs and LSTM networksfrom Section 4: supervised timeseries-in, timeseries-out learning tasks of all sorts and for all kinds of applications. I repeat the basic set-up from Section 4 for convenience. The generic formulas for RNNs and supervised learning tasks are:

Training data: One or several, long or short pairs of discrete-time input and output timeseries $S = (\mathbf{u}(n), \mathbf{y}(n))_{n=1,\ldots,n_{\mathsf{max}}}$, where $\mathbf{u}(n) \in \mathbb{R}^K$ is the input and $\mathbf{y}(n) \in \mathbb{R}^M$ is the output vector at time n.

Network equations:

$$\mathbf{x}(n+1) = \sigma(\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{\mathsf{in}}\,\mathbf{u}(n+1) + \mathbf{b}) \quad (64)$$

$$\mathbf{y}(n) = f(\mathbf{W}^{\mathsf{out}}\,\mathbf{x}(n)) \quad (65)$$

Re-read Section 4.1.1 if you are unsure about what the symbols in these equations mean.

Learning task: Find weights $\mathbf{W}, \mathbf{W}^{\mathsf{in}}, \mathbf{W}^{\mathsf{out}}, \mathbf{b}$, all lumped together in one global parameter vector θ, such that some loss function L defined to measure the mismatch between the teacher output signal $\mathbf{Y}^{\mathsf{train}} = (\mathbf{y}(n))_{n=1,\ldots,n_{\mathsf{max}}}$ and the RNN output $\hat{\mathbf{Y}}^{\mathsf{train}}_{\theta} = (\hat{\mathbf{y}}(n))_{n=1,\ldots,n_{\mathsf{max}}}$ of a network with weights θ is minimized, that is, solve the minimization problem

$$\theta_{\mathsf{opt}} = \operatorname*{argmin}_{\theta} L(\hat{\mathbf{Y}}^{\mathsf{train}}_{\theta}, \mathbf{Y}^{\mathsf{train}}). \quad (66)$$

The big difference between training RNNs in the "normal" way by gradient descent on the loss landscape $L(\hat{\mathbf{Y}}^{\mathsf{train}}_{\theta}, \mathbf{Y}^{\mathsf{train}})$ and how training is done in RC is that in the "normal" way, *all* parameters in $\theta = \{\mathbf{W}, \mathbf{W}^{\mathsf{in}}, \mathbf{W}^{\mathsf{out}}, \mathbf{b}\}$ are optimized, whereas in RC, *only the readout weights* $\mathbf{W}^{\mathsf{out}}$ are trained. The other weights contained in $\{\mathbf{W}, \mathbf{W}^{\mathsf{in}}, \mathbf{b}\}$ are randomly set at network design time and are then kept fixed forever. Figure 42 highlights this difference.

The recurrent neural network that lies between the input and the output neurons is called the *reservoir* in this field, and the weights $\mathbf{W}^{\mathsf{out}}$ from the reservoir to the output neurons are called the *readout weights* or just the *readouts*.

Only training the readouts seems to be a very strong simplification and one might expect that it will lead to a drastic reduction in achievable performance. There are a number of reasons why reservoir computing methods are nonetheless being explored and utilized besides the BPTT training schemes which dominate in deep learning:

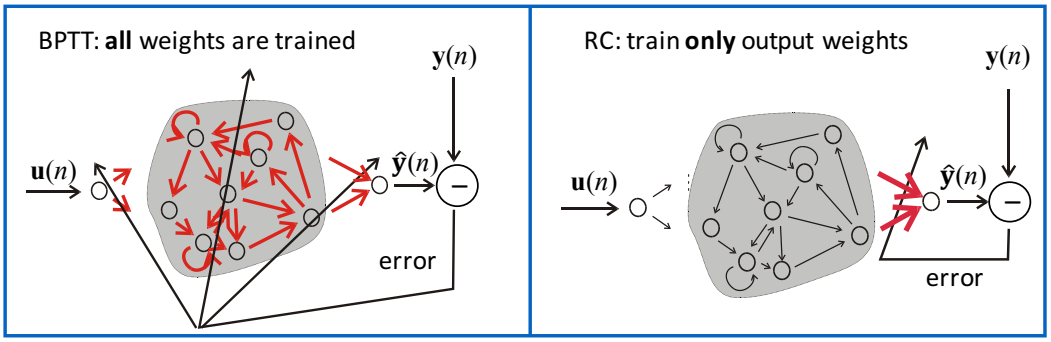

Figure 42: Highlighting the difference between the "normal" BPTT training and RC training of an RNN. Arrows shown in red are trainable.

1. The computational cost of RC training is only a tiny fraction of what is claimed by BPTT – seconds on a PC compared to days on a supercomputing cluster.

2. The training algorithm is numerically robust and there are no local minima problems.

3. For many tasks RC networks yield solutions that are on a par with what one gets from "normal" BPTT-trained RNNs. In some applications, RC even systematically outperforms BPTT-based RNN training schemes.

4. RC is biologically plausible. Neuroscientists have identified several circuits in real brains that might implement RC.

5. RC can work with RNNs whose equations are not differentiable, which precludes BPTT training. In fact, one does not need a "neural network" between input and output at all — any kind of nonlinear dynamical system works. Specifically, one can use exotic, non-digital and even non-electronic microchips as stand-ins for the RNN. This has made RC a leading computational paradigm in recent research in optical computing and other physical substrates which use non-electric nanoscale phenomena.

A note on history and significance: The core idea of RC, namely to use a fixed, non-trainable RNN and only train readouts, has been independently discovered a few times. The earliest publication known to me is Kirby (1991), a contribution to a local low-key AI conference, which was immediately and totally forgotten, followed after 1994 by a series of papers from Peter F. Dominey, a cognitive neuroscientist who identified RC-like circuits in the human brain (for instance, Dominey (1995)). He used a biologically inspired learning algorithm with low statistical efficiency, such that this work was not taken up in the machine learning quarters. In the year 2001 the RC principle was again re-discovered (by myself)

within a machine learning context, this time with an efficient learning algorithm, and branded under the name of *echo state networks* (ESNs) Jaeger (2001). At the same time it was also independently re-disovered by Wolfgang Maass in a theoretical neuroscience context, based on biologically detailed, spiking neuron models, and published under the name of *liquid state machines* (LSMs) Maass et al. (2002a). In those years, BPTT training of RNNs was not very practical because numerical instabilities and vanishing gradient problems were not yet under control. ESNs became popular in those years in machine learning, especially after the publication of Jaeger and Haas (2004) where ESNs achieved accuracy levels on benchmark tasks of that time which were up to five orders of magnitude better than the state of the art.

The term "reservoir computing" established itself as an umbrella term for ESNs, LSMs and some variants. In machine learning contexts, the term "echo state networks" is still common. When it is used, it is implied that simple neural networks with equations like 64 are used. When the word "liquid state machine" is used, this usually means that the author treats a neuroscience modeling topic and uses more involved, biologically motivated network models with spiking neurons. I will concentrate on the machine learning aspects of RC and therefore use the word "echo state network" in the next subsections.

In the decade until about 2015, the successful harnessing of BPTT in the deep learning field diminshed the interest in RC in the machine learning community, while it continued to be explored in neuroscience.

Since about 2015, both in academia and industry one could witness a quickly growing interest in developing "brain-inspired" computing microchips. Digital computing technologies will soon hit ultimate limits in miniaturization, and furthermore the energy demands of digital IT technologies are becoming prohibitive — it is estimated that today 10% of the world's energy budget is eaten up by digitial computing hardware. The biological brain is estimated to have an energy efficiency that is four orders of magnitude better than what can be realized with classical digital microchip designs. This is a strong economical and ecological motif to explore non-digital, "brain-like" *neuromorphic* hardware solutions. The rise of neuromorphic computing research has pulled RC back into the focus of attention.

8.1 A basic demo

The easiest way to understand RC is to go through a simple demo example. Let us consider a two-class timeseries classification task. The training data $S = (\mathbf{u}(n), \mathbf{y}(n))_{n=1,\ldots,n_{\mathsf{max}}}$ is a pair of input and output timeseries, both one-dimensional, where the input $\mathbf{u}(n)$ contains randomly alternating sections of two patterns of type C_1, C_2, and the desired output $\mathbf{y}(n)$ is a binary indicator signal which is 1 while the input pattern is of class C_1 and is 0 when the input pattern is of type C_2. In our demo, class C_1 is a rectangular wave while class C_2 is a sinewave. Figure 43 shows a portion of the training data.

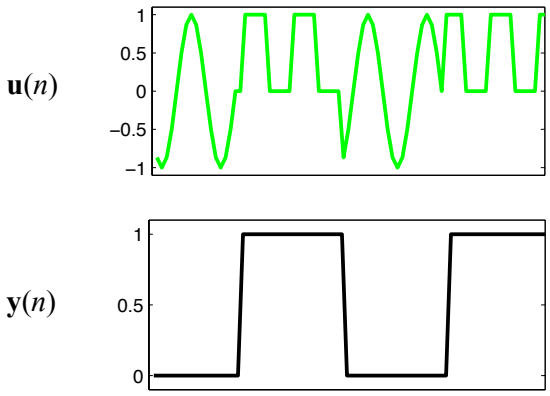

Figure 43: A simple binary temporal pattern classification task. A segment of the training data is shown. Top: input, bottom: desired output. The duration shown comprises 80 discrete timesteps (not shown on horizontal axis for cleaner graphical impression). Note that we are dealing with discrete-time signals; subsequent points are connected by a line for better visualization.

In this demo I use a reservoir made of $L = 100$ neurons. Since the input and output signals are one-dimensional ($K = M = 1$), there is one input neuron and one output neuron. The $L \times L$ sized reservoir-internal weight matrix $\mathbf{W}$, the $L \times 1$ input weight matrix $\mathbf{W}^{\text{in}}$ and the L-dimensional bias vector $\mathbf{b}$ from Equation 64 are filled with random values sampled from uniform distributions around 0, that is, positive and negative values occur roughly equally often.

8.1.1 Formal statement of the learning objective

The learning goal is to compute an $1 \times L$ sized output matrix $\mathbf{W}^{\text{out}}$ such that, when the network is driven by a test input of the same kind as used in training, the network output $\hat{\mathbf{y}}(n)$ approximates the binary teacher output $\mathbf{y}(n)$. For the activation function f in (65) I use the identity. The network output signal will thus be $\hat{\mathbf{y}}(n) = \mathbf{W}^{\text{out}}\ \mathbf{x}(n)$.

Let $\hat{\mathbf{Y}}^{\text{train}}_{\mathbf{W}^{\text{out}}} = (\hat{\mathbf{y}}(n))_{n=1,\ldots,n_{\text{max}}}$ be the network output when the network is driven by the training input and has output weights $\mathbf{W}^{\text{out}}$, and let $\mathbf{Y}^{\text{train}} = (\mathbf{y}(n))_{n=1,\ldots,n_{\text{max}}}$ be the teacher output. Note that $\hat{\mathbf{Y}}^{\text{train}}_{\mathbf{W}^{\text{out}}}$ and $\mathbf{Y}^{\text{train}}$ are vectors of length n_{max}. Using the quadratic loss (which is the most common choice in RC), the learning objective is to solve

$$\mathbf{W}^{\text{out}}_{\text{opt}} = \underset{\mathbf{W}^{\text{out}}}{\text{argmin}}\ \|\hat{\mathbf{Y}}^{\text{train}}_{\mathbf{W}^{\text{out}}} - \mathbf{Y}^{\text{train}}\|^2. \tag{67}$$

8.1.2 Step 1: state harvesting

Solving (67) is done in two steps. In the first step, the network (which has been randomly created) is driven by the teacher input, that is, for a duration of n_{max} steps. While it is being driven, we record the activations $x_i(n)$ of each of the reservoir neurons. This gives L timeseries of length n_{max}. Figure 44 illustrates this step, which is sometimes called the *harvesting* of reservoir states.

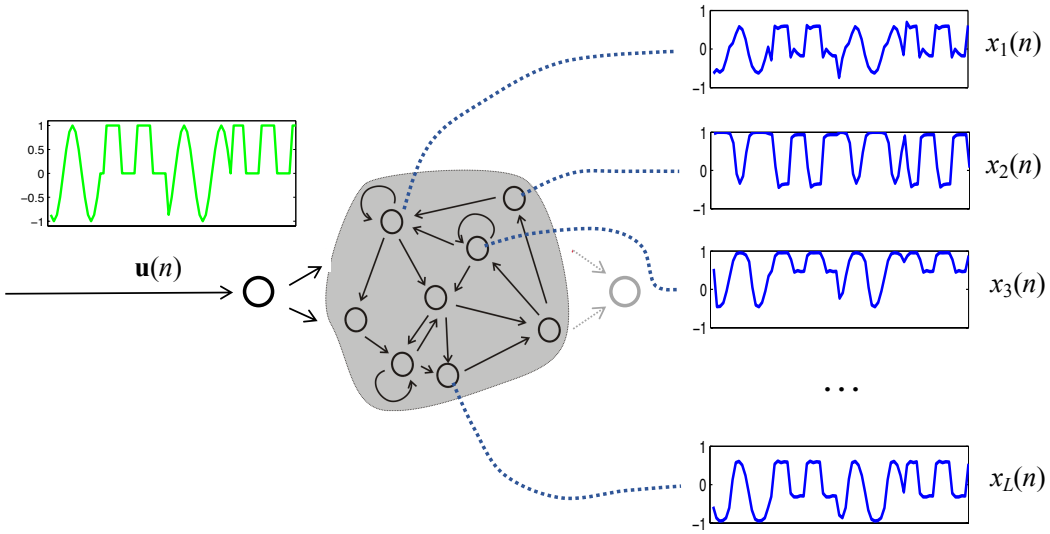

Figure 44: State harvesting — the first step in training an RC network. The network is driven by the training input $\mathbf{u}(n)$ (green) and the L reservoir neuron activation traces are recorded (blue traces on the right). Note that the readout weights are not known in this first step (indicated by the light gray coloring of the readout) and no network output is generated.

8.1.3 Step 2: compute readouts

In step 2 the optimal readout weights $\mathbf{W}^{\mathsf{out}}_{\mathsf{opt}}$ are computed. This calculation is based on the L harvested state sequences $\mathbf{x}_i = (x_i(1), \ldots, x_i(n_{\mathsf{max}})'$. Notice that these state sequences are n_{max}-dimensional vectors. If the readout weights are $\mathbf{W}^{\mathsf{out}} = (w_1^{\mathsf{out}}, \ldots, w_L^{\mathsf{out}})$, then the network output signal would be the n_{max}-dimensional vector $\hat{\mathbf{Y}}^{\mathsf{train}}_{\mathbf{W}^{\mathsf{out}}} = \sum_{i=1}^{L} w_i^{\mathsf{out}} \mathbf{x}_i$. The learning task (67) thus can be re-written as

$$\mathbf{W}^{\mathsf{out}}_{\mathsf{opt}} = \underset{\mathbf{W}^{\mathsf{out}}}{\mathrm{argmin}} \, \| \sum_{i=1}^{L} w_i^{\mathsf{out}} \mathbf{x}_i - \mathbf{Y}^{\mathsf{train}} \|^2. \tag{68}$$

In words: the L vectors $\mathbf{x}_i$ have to be linearly combined such that the combination sum best approximates the teacher vector $\mathbf{Y}^{\mathsf{train}}$ in the least mean square

error sense. This is just a case of computing a linear regression. Every mathematical or statistical programming toolbox offers a choice of ready-made algorithms for computing a linear regression.

If you are not familiar with the concept of linear regression, I can recommend Section 3.1 in my lecture notes for the Machine Learning course. (In fact mandatory reading if you need to refresh your understanding of linear regression. Nobody should leave a neural networks course without knowing what linear regression is!)

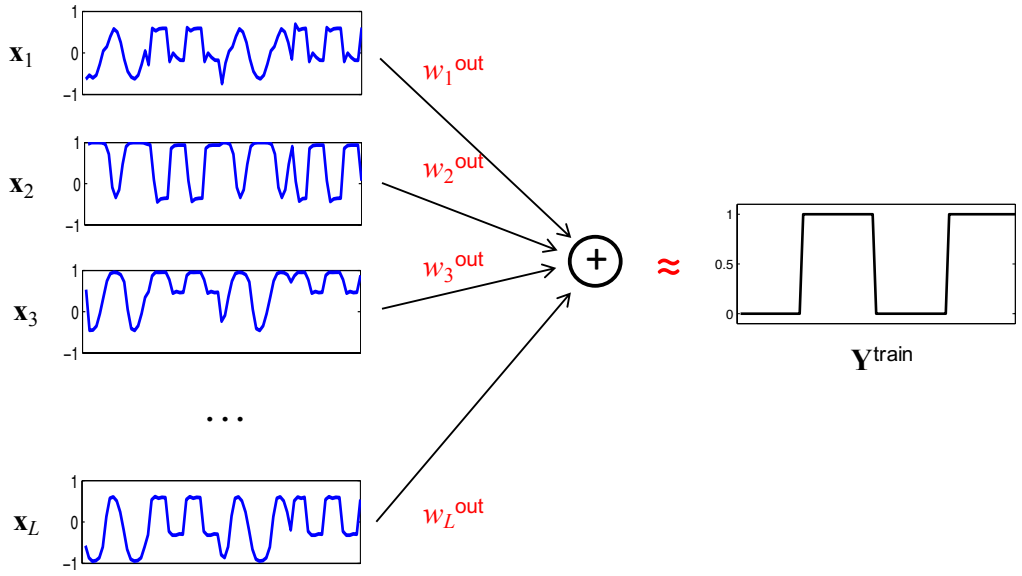

Figure 45: Step 2 in the RC training scheme: compute optimal readout weights w_i^{out} which give the best mean-square error approximation to the teacher $\mathbf{Y}^{\text{train}}$.

Figure 45 gives a graphical impression of this situation. After this step 2, the training is finished. The found weights $\mathbf{W}_{\text{opt}}^{\text{out}}$ are now inserted into the network architecture, the training is finished and the network ready for testing and use.

8.1.4 Testing

For testing, the trained network is driven with fresh input data $\mathbf{u}^{\text{test}}(n)$ for which a correct reference output signal $\mathbf{y}^{\text{test}}(n)$ is known, and the network-generated output $\hat{\mathbf{y}}^{\text{test}}(n)$ can be compared to the reference signal. Figure 46 shows this for our little demo example.

8.1.5 Computational cost

I conclude this demo with a summary of the computational cost. Assuming that the number K of input channels is less than the reservoir size L (which is typically the case), the random initialization of all the fixed weights comes at a cost of

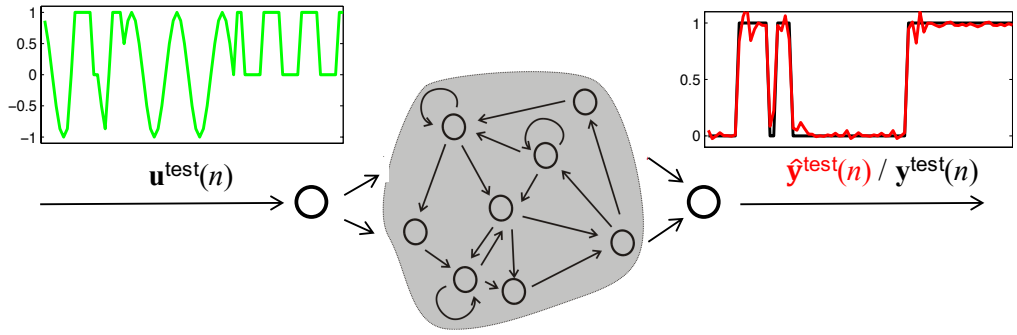

Figure 46: Testing the trained network (which now has the readout weights installed) with fresh input data. The reference output $\mathbf{y}^{\text{test}}(n)$ is shown in black and the network output in red.

$O(L^2)$. The state harvesting needs n_{max} network updates, each of which has a cost of $O(L^2)$. The linear regression costs $O(L^3)$. All of these together give a total cost of $O(L^2+n_{\text{max}}\,L^2+L^3) = O(n_{\text{max}}\,L^2+L^3)$. Normally one should have more training data time points than reservoir neurons (because if one has more neurons than training data points, one will get a zero-error solution from the linear regression, which would mean overfitting in all except the most trivial tasks). Then the cost is dominated by the harvesting phase and becomes $O(n_{\text{max}}\,L^2)$. That is, the cost of training is essentially the same as running the network once with the training input data. It cannot be cheaper.

8.2 RC in practice

The demo example illustrated all that needs to be done in training a RNN in reservoir computing style. Seems simple... But if you want to squeeze good performance out of ESNs, a number of design decisions need to be made appropriately, and this is not so simple. After all, we are dealing with high-dimensional nonlinear dynamical systems, and these are never easy to handle. In my experience it takes some months for a RC novice, working full-time, to gain the insight and routine necessary to handle RC techniques adequately. The "tricks of the trade" are explained in the detailed practical RC tutorial Lukosevicius (2012). Here I mention some things that need to be considered. This summary account cannot replace reading Lukosevicius (2012) if you seriously want to get started with RC in practice.

Discard washouts. For state harvesting, the reservoir must be started in an initial state $\mathbf{x}(0)$. This state is arbitrary and unrelated to the learning task, and traces of it will remain in the next few networks states. Figure 47 shows this "initial state memory" effect. Our demo ESN was driven twice with the same input, but started from two different, randomly chosen initial

network states. The top panel in Figure 47 shows the initial 10 step traces of four reservoir neurons, with the set of traces from the first initial state in solid lines and the traces arising fro the other initial state in broken lines. One can see that for the first few steps these state sequences differ from each other, but converge toward each other. The rate of convergence is typically exponential on average, as can be seen in the bottom plot. This plot was computed as follows. Let $\mathbf{x}(n), \mathbf{x}^*(n)$ denote the two reservoir state sequences. The bottom plot shows the development of the Euclidean distance between $\mathbf{x}(n)$ and $\mathbf{x}^*(n)$ in log10 scaling, that is, it plots $\log_{10} \|\mathbf{x}(n) - \mathbf{x}^*(n)\|$. The initial state differences are "forgotten", in RC terminology: they are *washed out*. How fast this washing-out happens depends on many factors; it can be much slower than in this demo where the differences become invisible in the top plot after three or four steps already.

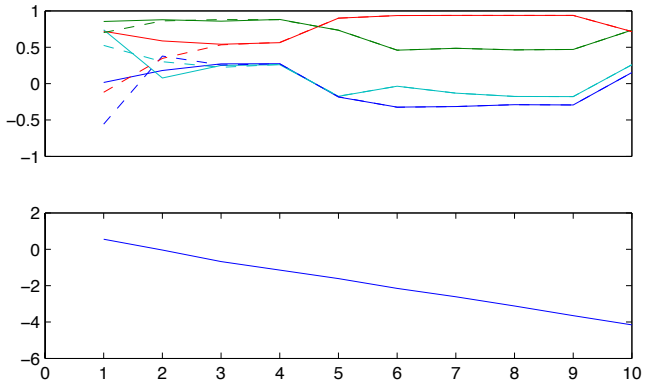

Figure 47: The initial state memory effect. For explanation see text.

In order not to pollute the linear regression in step 2 of the RC training procedure, one simply starts the harvesting after an initial washout period. An appropriate length of this washout time is most easily determined by computing a diagnostic plot like our Figure 47.

Scaling of initial weights. This is the most important point. The geometrical and dynamical properties of the harvested state sequences depend crucially on the average absolute sizes of the weights in $\mathbf{W}^{\mathsf{in}}, \mathbf{W}$ and the bias $\mathbf{b}$. Figure 48 illustrates this.

Concretely, in this scaling exercise I always started from the same reference matrices $\mathbf{W}^{\mathsf{in}}_{\mathsf{ref}}, \mathbf{W}_{\mathsf{ref}}, \mathbf{b}_{\mathsf{ref}}$. The entries of $\mathbf{W}^{\mathsf{in}}_{\mathsf{ref}}$ and $\mathbf{b}$ were sampled from the uniform distribution in the range $[-0.5, 0.5]$. In order to get $\mathbf{W}_{\mathsf{ref}}$, another method was used which is typical for practical work with ESNs. A preliminary version of $\mathbf{W}_{\mathsf{ref}}$ was created again by sampling weights from $[-0.5, 0.5]$. Then the absolute value $|\lambda_{\mathsf{max}}|$ of the largest eigenvalue of that preliminary matrix was calculated. This number, which can be computed for any square

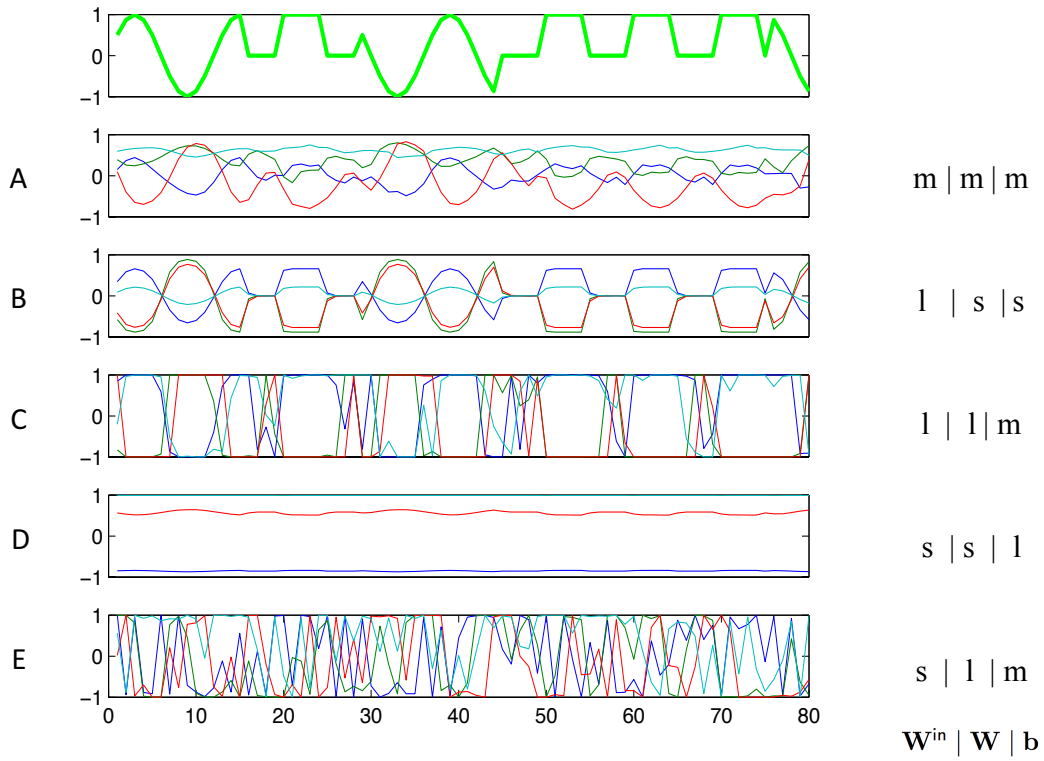

Figure 48: The effects of weight scaling. Top panel: the input signal. The other plots A – E show traces of four neurons in runs with that input, where the weights were scaled with different scaling combinations. From top to bottom the three weightable items $\mathbf{W}^{\text{in}}$ | $\mathbf{W}$ | $\mathbf{b}$ were scaled large (l), medium (m) or small (s) as indicated on the right hand side.

matrix, is called the *spectral radius* of the matrix. It plays an important role in the theory and practice of RC. The preliminary matrix was scaled by $1/|\lambda_{\text{max}}|$ to give $\mathbf{W}_{\text{ref}}$, which thus had a spectral radius of 1.

Given these reference matrices $\mathbf{W}^{\text{in}}_{\text{ref}}, \mathbf{W}_{\text{ref}}, \mathbf{b}_{\text{ref}}$, I scaled each of them with a factor that was either small (order of 0.1) or medium (order of 1.0) or large (order of 10). The exact scalings are not important here — I finetuned them a little to get visually appealing graphics in Figure 48.

Here are some observations and comments. I refer to the cases A – E in that figure:

Case A: When all weights are scaled to an intermediate range, the network states behave in a way that normally works well in RC: their amplitudes span much of the possible value range $[-1, 1]$; they are clearly influenced by the driving input but the reservoir-internal interactions make them markedly different from each other. Note that such a behavior is not universally the best for all sorts of tasks.

Case B: When the input weights are large and all other weights are small, the activation of a reservoir neurons will be dominated by the input term in (64) and (almost) all neurons will exhibit activation traces that look like scaled versions of the input. This is not a desirable behavior in most applications, except sometimes when the desired output $\mathbf{y}(n)$ at time n depends only on the input $\mathbf{u}(n)$ at the same time (no memory effects needed). But in such cases, a feedforward network would be a better choice than an RNN.

Case C: If both input weights and reservoir-internal weights are large, the activations of neurons will typically be pushed toward the $-1, +1$ limits of the tanh sigmoid and an almost binary "switching" dynamics inside the reservoir results. This may be appropriate in extremely nonlinear, quasi Boolean input-output learning tasks.

Case D: If the bias yields the dominating weight components, the reservoir dynamics degrades toward constant values in each neuron. I cannot imagine any interesting task where this would be beneficial.

Case E: If the reservoir-internal weights are large and the input and bias not, then there is danger that the network dynamics falls prey to a wild recurrent self-excitation which is no longer modulated in a useful way by the input: *chaotic* dynamics (in the strict mathematical sense) emerge. This is certainly useless because the "echo state property", to which I will devote a separate subsection below, is violated.

In summary, you see that the absolute and relative scalings of $\mathbf{W}^{\mathsf{in}}_{\mathsf{ref}}, \mathbf{W}_{\mathsf{ref}}, \mathbf{b}_{\mathsf{ref}}$ exert a dramatic influence on the reservoir dynamics, and getting those scalings right is crucial for a high-quality RC performance.

Unfortunately there is no general rule of how to set these scalings optimally. It depends on the task and also on the dimensions K and L of the input and reservoir. Like so often in practical work with neural networks, the beginner has to spend a lot of time experimenting, and experienced users will benefit from their well-honed intuitions. In any case, you should always create indicative plots of state dynamics as in Figure 48 to get a "feeling" for what is happening inside your reservoir. It's like a doctor doing an X-ray.

Output feedback for signal generation tasks. Some tasks demand that the trained network should *generate* an output signal. The basic example is to train a network that has no input and a single output unit which should yield a generated signal, for instance a sinewave oscillation. Such signal generation tasks require that the generated output signal is fed back into the reservoir. Using a linear output unit, the network update equations are

$$\mathbf{x}(n+1) = \sigma(\mathbf{W}\,\mathbf{x}(n) + \mathbf{W}^{\mathsf{fb}}\,\mathbf{y}(n) + \mathbf{b}), \tag{69}$$

$$\mathbf{y}(n+1) = \mathbf{W}^{\mathsf{out}}\,\mathbf{x}(n+1). \tag{70}$$

The weights $\mathbf{W}, \mathbf{W}^{\text{fb}}, \mathbf{b}$ are fixed at design time, the readouts $\mathbf{W}^{\text{out}}$ are trained. Figure 49 illustrates the set-up.

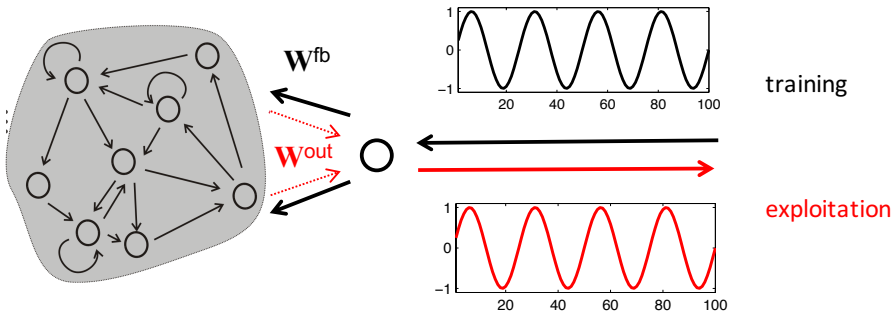

Figure 49: Training an ESN as a sinewave oscillator. At training time the output weights, shown in dotted red lines, are not there yet. For explanation see text.

The teacher signal $\mathbf{y}(n)$ is a sample of the desired generated signal, here a sinewave (black signal in Figure 49). For state harvesting, the teacher signal is written into the output node in a version that is delayed by one timestep, that is in update cycle n the value written into the output unit is $\mathbf{y}(n-1)$. This time lag accounts for the relative time difference of the $\mathbf{y}$ signal that you witness in (69) versus (70). The feedback weights $\mathbf{W}^{\text{fb}}$ assume a role as input weights in the state harvesting phase.

After the output weights are computed as usual with linear regression, the network is ready for use. If all worked out well, started from a random initial state the network will settle into an oscillation mode which after an initial transient, where the random initial reservoir state is washed out, settles into the desired oscillation. Figure 50 shows this.

One can also add separate input signals which *modulate* the generated output signals. Figure 19 (Section 4.1) shows an example where the task was to generate a sinewave output whose frequency is set by an input signal. When the input signal has a high value, the generated sine should have a high frequency. The demo illustrated in Figure 19 was based on an ESN.

ESNs are particularly well suited for signal generation tasks. Specifically, they are unrivalled when it comes to generate *chaotic* signals. The paper which popularized ESNs (Jaeger and Haas, 2004) included demos of generating chaotic signals whose precision was essentially machine precision. In the last few years, the surprising performance of RC in modeling chaotic systems has attracted the attention of mathematicians and physicists, who exploit this phenomenon for the study of chaotic systems (for example spatial wavefront dynamics, Pathak et al. (2018)).

Leaky integrator neurons. We have seen in Figure 48 that reservoir dynamics

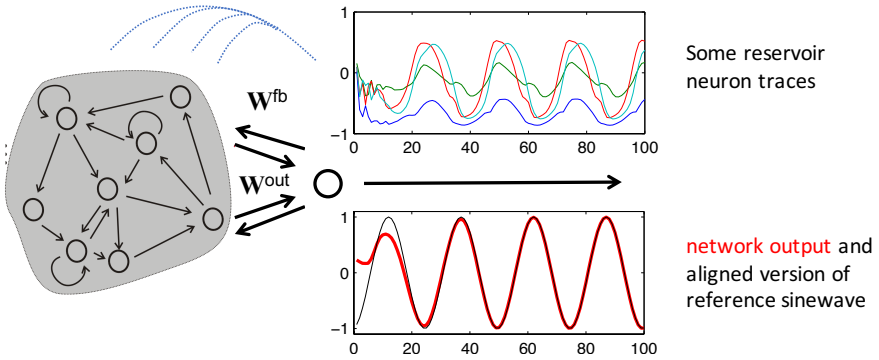

Figure 50: After training, the network functions as an oscillator. The output unit will (after an initial washout resulting from the arbitrary initial reservoir state) generate the kind of signal that was used as teacher. The network output (red) is overlaid with a suitably phase-shifted version of the training sine (thin black line). The reservoir neurons display all kinds of oscillations whose shapes result from the nonlinear interactions of the reservoir neurons (four traces shown).

can have quite different properties. For good RC results, the dynamical and geometrical properties of the reservoir should match the properties of the task. Such properties which should match are, for example, the degree of nonlinearity, the length of memory timespans, or the sheer complexity of the task.

One of the most important properties is something that one could call the "speed" of the system from which the training data come. There are fast systems and there are slow systems. A gigahertz computer clock is faster than the tides of the ocean. While it is intuitively clear that there are fast and slow systems, there exists no universal best mathematical way to define or measure the "speed" of a dynamical system. I will not dig deeper here although I would love to do so — comint to terms with timescales is the core of my current research in the European Project *MeM-Scales* ("Memory Technologies with multi-scale time constants for neuromorphic architectures", `memscales.eu`).

In practical applications of RC a crucial factor for success is to adjust the reservoir's "speed" to the task's "speed". To make this possible, one needs a way to design slow or fast reservoirs.

The standard approach is to use a neuron model where each neuron's dynamics is governed by the *leaky integration* update equations. In fact, RC experts will almost always use *leaky integrator neurons*. Such neurons can be slowed down or sped up as one wishes.

They are best explained by starting from a continuous-time neuron model.

Consider a reservoir that runs in continuous time $t \geq 0$. The L-dimensional activation vector at time t is $\mathbf{x}(t)$ (notice that often one uses symbol $n \in \mathbb{Z}$ for discrete time points and $t \in \mathbb{R}$ for continuous time). The activation $x_i(t)$ of neuron i at time t is a continous-time signal. For a leaky integrator neuron, it is governed by the ODE

$$\dot{x}_i(t) = \frac{1}{c}\left(-x_i + \sigma\left(\sum_{j=1}^{L} w_{ij}\, x_j(t) + \sum_{k=1}^{K} w_{ik}^{\mathsf{in}}\, u_k(t) + b_i\right)\right), \qquad (71)$$

where the w_{ij} are the synaptic connection weights of the incoming connections of neuron i, $u_k(t)$ is the k-th component of the input vector $\mathbf{u}(t)$ at time t, the w_{ik}^{in} are the input connection weights leading into neuron i, b_i is this neuron's bias, and — the most interesting bit in this equation — c is the *time constant* of this ODE.

If this ODE is integrated with different settings of the time constant, the obtained trajectory that one could plot in a phase portrait will look the same. Remember that the trajectories follow tangentially the vectors of the vector field given by the right hand side of the ODE. The effect of changing the time constant c only scales the length of these vectors but not their direction. If the time constant grows, the length of the vectors in the vector field shrinks in inverse proportion. This means that the "forward speed" of the point $x_i(t)$ along the trajectory slows down when c grows. By setting c one can create slow or fast continuous-time reservoir dynamics at one's discretion.

This dynamical law is called the *leaky integration* model of a neuron because the term $\sigma\left(\sum_{j=1}^{L} w_{ij}\, x_j(t) + \sum_{k=1}^{K} w_{ik}^{\mathsf{in}}\, u_k(t) + b_i\right)$ integrates the input to this neuron over time, while the term $-x$ always lets the activation $x(t)$ diminish ("leak") at a rate that is proportional to the current activation.

This is, by the way, an effect that is also active in biological neurons: due to elementary electrophysics, their electric potential (measured in millivolt) likewise would dissipate at a rate proportional to its current level, because the cell membrane is not a perfect insulator. Therefore, leaky integrator models are a much better fit to biological neurons than the simple neuron model that we have been using in this section so far. In computational neuroscience one almost always uses leaky integrator models of various sorts.

However, for practical machine learning applications run on digital computers one needs a discrete-time neuron model. Instead of a continuous trajectory $(x_i(t))_{0 \leq t}$ one needs a timeseries which advances in discrete steps with a chosen stepsize Δ, that is one wants to have a discrete-time version $(\tilde{x}_i(n\,\Delta))_{n=0,1,2,\ldots}$ of the continuous trajectory. At times $t = n\,\Delta$ the two trajectories should be (approximately) equal, $x_i(n\,\Delta) \approx \tilde{x}_i(n\,\Delta)$.

Finding discretization methods which allow such a discretization of ODEs with a good balance between approximation accuracy and computational cost is a main subject of numerical mathematics. Whenever you simulate an ODE on a digital machine — which is the daily bread and butter in all the natural sciences — a numerical *ODE solver* is invoked. Every mathematical toolbox offers a choice of such solvers.

The simplest of all ODE solvers is called the *Euler* method (see `https://en.wikipedia.org/wiki/Euler_method` for an easy introduction). After some point $\tilde{x}_i(n\,\Delta)$ has been computed, the next point $\tilde{x}_i((n+1)\,\Delta)$ is computed by following the direction of the vector given by the right-hand side of the ODE for a timespan of Δ:

$$\begin{aligned}\tilde{x}_i((n+1)\,\Delta) \;&=\; \tilde{x}_i(n\,\Delta) \;+ \\ &\Delta\;\tfrac{1}{c}\left(-\tilde{x}_i(n\,\Delta) + \sigma\left(\textstyle\sum_{j=1}^{L} w_{ij}\,\tilde{x}_j(n\,\Delta) + \sum_{k=1}^{K} w_{ik}^{\mathsf{in}}\,u_k(n\,\Delta) + b_i\right)\right).\end{aligned}$$

Using $\Delta = 1$, renaming $1/c$ to a and joining the two $\tilde{x}_i$ terms leads to the simpler looking version

$$\tilde{x}_i(n+1) = (1-a)\;\tilde{x}_i(n) + a\;\sigma\left(\sum_{j=1}^{L} w_{ij}\,\tilde{x}_j(n) + \sum_{k=1}^{K} w_{ik}^{\mathsf{in}}\,u_k(n) + b_i\right), \quad (72)$$

which is the form that you will most commonly find in the RC literature. Now you can control the "speed" of neurons by setting $a \in [0,1]$: the larger this is set, the faster the neuron. In the extreme case $a = 1$ one recovers our accustomed simple update equation (64). The slowest "dynamics" is obtained with $a = 0$: then nothing happens — the network state remains frozen in its initial state. The number a is often called the *leaking rate* of the neuron.

Many real-world physical systems and all truly cognitive systems operate on several timescales simultaneously. For instance, in atmospheric dynamics small whirls of air (like gust eddies between houses) have a typical time constant of a few seconds, while the large whirl of a low over central Europe evolves over several days. Or, for a cognitive dynamics example, while you are reading this lecture notes section, your brain at any moment has to integrate information bits that come just from the preceding syllable (read a few milliseconds before) with information from the beginning of this section (maybe an hour ago). As of today, neither deep learning LSTM networks trained with BPTT, nor ESNs are capable of integrating information across many timescales. Extending the multi-timescale capabilities of RNNs is a major topic of current research. In ESNs, one approach is to design reservoirs with leaky integrator neurons, where different submodules of the reservoir have different values for the inverse time constant a. Typically, fast modules

or processing layers are closer to the input than slow layers. An example is Gallicchio et al. (2018) where "deep" ESNs are constructed in this way for speech recognition and music composition.

Regularization and reservoir size. Like any machine learning method, RC is susceptible to overfitting. In order to cope with this problem, the flexibility of a RC training scheme must be adapted to the available training data volume and task complexity by experimentation in a cross-validation scheme (recall this from Section 1.4). This needs a way to tune the degree of modeling flexibility. The recommended approach with ESNs is

- use a *large* reservoir, choosing a size that would allow overfitting,
- then use a regularized version of linear regression for the readout weight calculation, called *ridge regression*.

Ridge regression, also known as *Tykhonov regularization* (`https://en.wikipedia.org/wiki/Tikhonov_regularization#Determination_of_the_Tikhonov_factor`) should always be used in machine learning when a linear regression has to be carried out — which means, all over the place, not only in a RC context. It is a very valuable thing to know, and I will explain it in a little detail and present an example.

Let us first rehearse the maths of linear regression, framed in an ESN training scenario. Assume we want to compute the $1 \times L$-dimensional readout weight vector $\mathbf{w}$ for a single output neuron from harvested L-dimensional reservoir states $\mathbf{x}(1), \ldots, \mathbf{x}(n_{\mathsf{max}})$ such that the mean squared training error is minimized, that is, we want to solve the problem

$$\mathbf{w}_{\mathsf{opt}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{n_{\mathsf{max}}} (\mathbf{w}\,\mathbf{x}(n) - y(n))^2, \tag{73}$$

where $y(n)$ is the teacher for the output unit (note that $\mathbf{w}$ is a row vector). The analytical solution for this problem is the well-known solution formula for linear regression problems,

$$\mathbf{w}'_{\mathsf{opt}} = (\mathbf{X}'\,\mathbf{X})^{-1}\,\mathbf{X}'\,\mathbf{y}, \tag{74}$$

where $\mathbf{X}$ is the $n_{\mathsf{max}} \times L$ sized matrix containing the state vectors $\mathbf{x}(n)$ in its rows and $\mathbf{y}$ is the $n_{\mathsf{max}} \times 1$ vector $\mathbf{y} = (y(1), \ldots, y(n_{\mathsf{max}}))'$ containing the teacher output values.

In ridge regression, the objective function in (73) is augmented by a regularization term which penalizes the sum of squared weights,

$$\mathbf{w}_{\mathsf{opt}} = \underset{\mathbf{w}}{\operatorname{argmin}} \left(\sum_{n=1}^{n_{\mathsf{max}}} (\mathbf{w}\,\mathbf{x}(n) - y(n))^2 \right) + \alpha\,\mathbf{w}'\mathbf{w}, \tag{75}$$

where $\alpha \geq 0$ weighs the strength of the added regularization. The analytical solution of this problem is

$$\mathbf{w}'_{\text{opt}} = (\mathbf{X}'\,\mathbf{X} + \alpha\,\mathbf{I})^{-1}\,\mathbf{X}'\,\mathbf{y}, \tag{76}$$

where $\mathbf{I}$ is the L-dimensional identity matrix. Ridge regression thus just adds α on the diagonal ("ridge") of $\mathbf{X}'\,\mathbf{X}$, which gave this method its name.

The larger α is chosen, the stronger the regularization and the smaller the resulting entries in $\mathbf{w}_{\text{opt}}$. For $\alpha = 0$ the ordinary regression formula (74) is recovered.

I demonstrate the working with ridge regression on a simple example. The task is to predict the next value of a sinewave signal. The training data $(u(n), y(n))_{n=1,\ldots,n_{\text{max}}}$ consists of a sinewave signal $u(n)$ to which noise was added, and the teacher is the same signal advanced by one step into the future, that is $y(n) = u(n-1)$. The top panel in Figure 52 shows the noisy sinewave used for training.

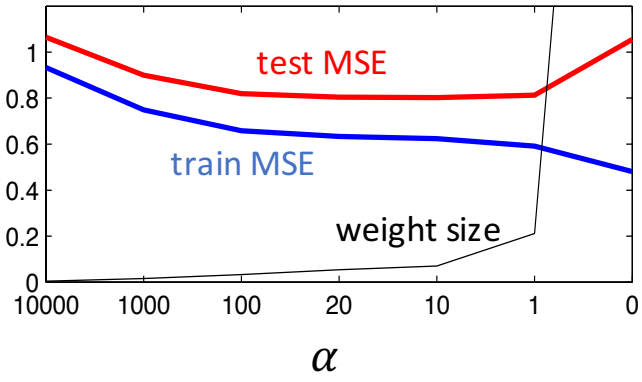

Figure 51: THE paradigmatic behavior of training and testing errors as the vary with the degree of regularization. For explanation see text. The black curve shows how the mean absolute output weights grow as the regularization strength is decreased. For $\alpha = 0$ the average absolute weight size is about 5.2 (not drawn).

The reservoir was made of $L = 100$ leaky integrator neurons with a leaking rate $a = 0.5$. I omit a discussion of how the various weight scaling factors were set — this is not a challenging task and the ESN training works well in a wide range of these scalings. I computed different versions of output weights with the regularization parameter α in (76) chosen differently, namely as 10000, 1000, 100, 20, 10, 1, and 0. I computed training and testing MES's and plotted them against α (Figure 51). This is the same kind of graphic as Figure 6 in Section 1.3.5. You should always draw this plot when carrying out a supervised training task with whatever kind of method! The curves

behave as in a textbook they should: the less regularization is applied, the lower the training error; but the test error is high both with very strong and very low amounts of regularization and has a minimum for some intermediate degree of regularization (here at $\alpha = 20$). The figure also shows how the regularizing term $\alpha\,\mathbf{w}'\mathbf{w}$ in (75) pulls down the resulting weight sizes when α gets larger.

Figure 52 illustrates the performance of the ESN when it is trained in underfitting ($\alpha = 10000$), optimal ($\alpha = 20$) and overfitting ($\alpha = 0$) conditions. Again, this is true proper textbook behavior as it should be.

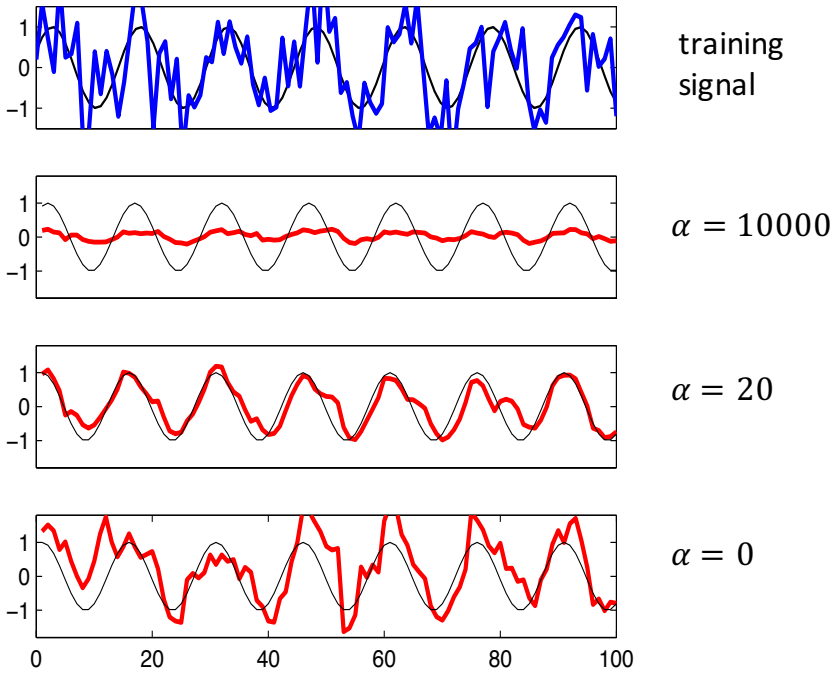

Figure 52: The training signal (top) and the network output (red) in three regularization conditions. The thin black sines drawn into the panels are the clean sinewaves; they are shown for visual intuition only and the training and testing procedures did not use this information.

8.3 Online reservoir training

While the analytical solution (76) will be used in most practical RC applications, it is sometimes necessary to use an iterative stochastic gradient descent method to solve the linear regression task (75). This is mandatory in *online adaptive* learning, where the properties of the input-to-output signal transformation change with time

and the learning system must continually *track* the target system.

For example, if an RNN is set up to control the carburettor valves of a combustion engine for optimal fuel efficiency, the operating conditions of the engine change with temperature, runtime, load, age of the engine, quality of the fuel, and other unforseeable factors. RNNs are indeed employed for this purpose (or at least they have been — I learnt about this from Ford engineers two decades ago). In such a scenario, the RNN must be continuously re-trained as new measurement data (which are used as input) come in. If you are interested, you find an entire section devoted to online adaptive signal processing in my machine learning lecture notes which are online on our Nestor course pages. Furthermore, biological neural systems need to learn in incremental adaptive fashion — biology has no way to invert a matrix, as required be the analytical solution (76).

Again considering, for simplicity, the case where there is only a single output neuron, the training data used in online adaptive learning is a potentially endless stream $(\mathbf{u}(n), y(n))_{n=1,2,\ldots}$. Its input-output transformation properties will slowly change as time runs on, which is why it makes no sense collecting data from a long stretch of this stream and use it for RNN training of any kind: the RNN would "learn" a useless compromise blend of the early and late behavior of the system which produces the training data.

Instead, with ESNs one uses the following online algorithm to continuously adapt the current model to the ongoing data stream $(\mathbf{u}(n), y(n))_{n=1,2,\ldots}$.

- The reservoir is continually fed with the input data stream $\mathbf{u}(n)$, leading to a synchronous reservoir state sequence $\mathbf{x}(n)$.

- Assume that at time n, an output weight vector $\mathbf{w}^{\mathsf{out}}(n)$ is in place which at the time around n produces outputs $\hat{y}(n)$ which give a good approximation (in the mean square error sense) to the teacher signal $y(n)$. In the update $n \to n+1$, the online algorithm uses the next teacher value $y(n+1)$ to adjust $\mathbf{w}^{\mathsf{out}}(n)$ a little, by doing a small step down the error gradient, that is, add a little correction vector to $\mathbf{w}^{\mathsf{out}}(n)$ which leads it in that direction which most strongly reduces the squared error $(y(n+1) - \hat{y}(n+1))^2$. Skipping the maths (derivation is straightforward, if interested you find it in Section 11.3 in my Machine Learning lecture notes, or in hundred other textbooks and online tutorials), this leads to the update equation

$$\mathbf{w}^{\mathsf{out}}(n+1) = \mathbf{w}^{\mathsf{out}}(n) + \lambda \, (y(n+1) - \hat{y}(n+1)) \, \mathbf{x}'(n), \tag{77}$$

  where λ is a small *learning rate*, say $\lambda = 0.01$.

That's it. This very cheap and simple rule sits behind most of the signal processing procedures which make the radio receiver module in your smartphone work; neuroscientists say that it is biologically plausible; and it enables reservoir computing to function well in online adaptive task settings, where deep learning methods are severely challenged. If you have a very good memory of the earlier

parts of these lecture notes you will recognize that the Perceptron learning rule is in fact just a version of this algorithm with $\lambda = 1$.

Like so many powerful ideas in engineering and science, this rule has been discovered independently several times in different disciplines, where it is known under different names. In signal processing and control it is called the *LMS algorithm*, in the neurosciences and sometimes in artificial neural network research it is called the *Widrow-Hoff* rule or the *Delta-rule*, and a plain mathematician would likely refer to it as *stochastic gradient descent on a quadratic error surface*. No introductory course on neural networks would be complete without it, and I seized the opportunity to introduce it in an ESN context.

8.4 The echo state property

Not every design of a reservoir will work. Specifically, when the reservoir weight matrix $\mathbf{W}$ is scaled too large, the reservoir will recurrently excite itself so strongly that its internal, self-generated dynamics will overrule any influence of the input, and the harvested states will become useless. If run twice with the same input but from different initial states, the two state sequences will not converge to each other after a washout as they should do, like visualized in Figure 47. Instead, anything can happen, including unpredictable chaotic dynamics. Figure 53 demonstrates this.

When the network washes out differences of initial states under the influence of an input signal $\mathbf{u}(n)$, the combination of the network and this input is said to possess the *echo state property* (ESP).

The scaling of $\mathbf{W}$ is standardly expressed by the spectral radius ϱ of $\mathbf{W}$. The spectral radius has an important effect on the learning accuracy and one usually has to do some manual experimentation to find a good value for ϱ. One typically starts with a reference weight matrix $\mathbf{W}_{\mathsf{ref}}$ which has a spectral radius of 1, and then re-runs the training with different settings of ϱ, that is, using $\varrho\,\mathbf{W}_{\mathsf{ref}}$ for the reservoir weight matrix.

For every input signal $\mathbf{u}(n)$ and reservoir weight matrix $\mathbf{W}_{\mathsf{ref}}$ there is a unique critical spectral radius ϱ_{crit} such that for $\varrho < \varrho_{\mathsf{crit}}$ the ESP is obtained and for $\varrho > \varrho_{\mathsf{crit}}$ it is lost. This critical value marks a bifurcation in the reservoir dynamics.

There is no known way to predict the critical spectral radius ϱ_{crit} analytically. A lot of effort has been spent and is being spent on the mathematical study of the ESP. There are two reasons why the ESP is attracting so much attention:

- For machine learning applications of RC it is obviously important because RC training only works for input-reservoir combinations that exhibit the ESP.

- In the general research on cognitive neurodynamics, a currently popular hypothesis states that, roughly speaking, the human brain works at its best when it increases its self-excitiation level unit just below the point where it

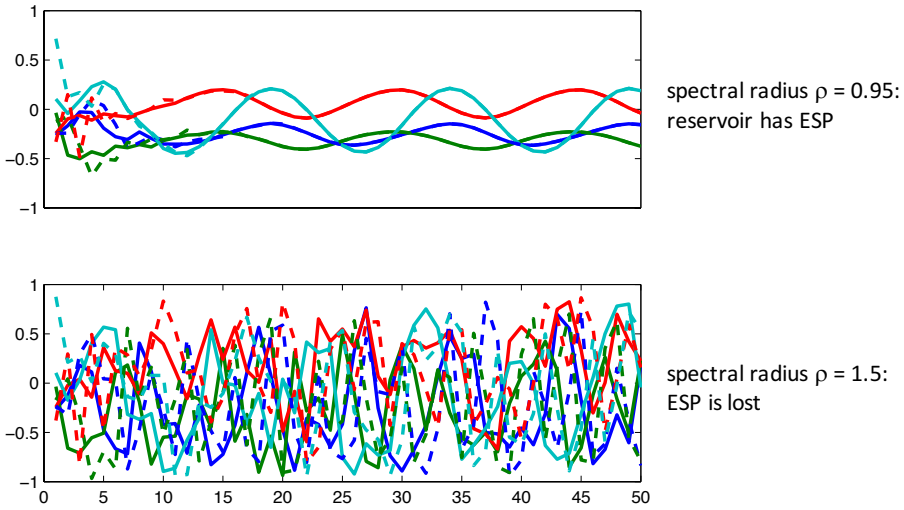

Figure 53: The echo state property: to have or have not. Top: with a scaling of the reservoir weight matrix under a critical value, the ESP is granted and state sequences converge to each other after some washout time, when the reservoir is started from different initial states with the same input. Four reservoir neurons are plotted, with the two runs distinguished by solid / dashed lines. The input was a sinewave (not shown). Bottom: the ESP is lost when the reservoir weight matrix is scaled up beyond a critical value. Here the reservoir engages in a chaotic dynamics that has lost the connection to the input signal.

bifurcates into chaos. Google `‘‘edge of chaos’’ neural network` to get a glimpse of the lively research on this topic, both in machine learning and neuroscience quarters.

8.5 Physical reservoir computing

This subsection is not mandatory reading. It's just exotic, visionary slightly crazy fun stuff — a peek into the future of computing maybe — or maybe not.

Since a few years, interest in reservoir computing has been re-kindled. The reason is that RC is one of the few computational approaches which do not need, in principle, digital computers. Non-digital, "neuromorphic", "brain-inspired", "unconventional" computing microchips are gaining relevance due to the reasons that I briefly mentioned in the last paragraph before subsection 8.1. The Groningen Cognitive Systems and Materials Center (CogniGron) research center (`https://www.rug.nl/research/fse/cognitive-systems-and-materials/`), which was founded at RUG two years ago, strives to become a European pioneer in this field. The principle of reservoir computing, which can be summarized as

use an input signal to drive a nonlinear excitable medium — the "reservoir" — and generate a desired output signal by combining many of the local response signals that can be observed in the medium

can be applied to many kinds of "reservoirs" other than neural networks simulated on digital machines. The recent survey of Tanaka and et al (2019) gives an overview. If the reservoir is a real, physical piece of material which can be excited into interesting dynamical responses by some physical driver input, one speaks of *physical reservoir computing*. The potential benefits are inherent parallelism, a low energy budget (in physical nanoscale devices), high speed, extreme high-dimensionality of the reservoir states (even, in principle, infinite-dimensional states in continuous materials), and, hopefully, low cost. As of today, all of this is still academic research. I will not go into details but just illustrate the flavor of this kind of research with a few examples. I present them by figures with instructive captions, without surrounding explanatory text.

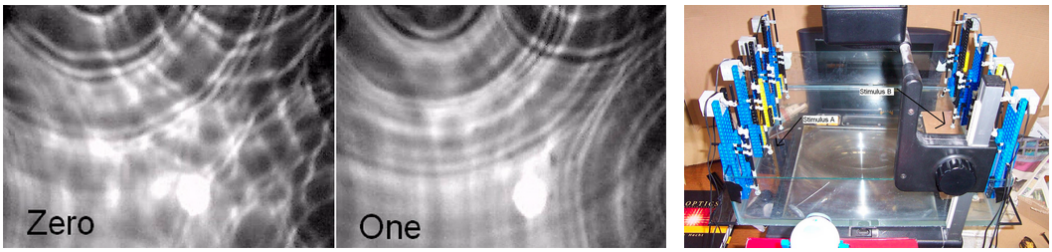

Figure 54: Reservoir computing in a bucket. Wolfgang Maass, the inventor of the "liquid state machine" version of RC, used to call the reservoir "the liquid" (Maass et al., 2002b). This was taken literally by students Chrisantha Fernando and Sampsa Sojakka at the School of Cognitive and Computer Sciences, University of Sussex. They filled a small transparent acryl basin with real water, excited it with eight Lego-made mechanical pushrods whose oscillations were derived from speech signals, optically recorded the states of the water surface ripples, and used these states for RC. Their "liquid brain" could solve the infamous XOR task and classify spoken "Zero" versus "One". Their paper at the ECAL 2003 (Fernando and Sojakka, 2003) won the highest impact paper award. Christian, now a Senior Research Scientist at Google DeepMind, explains it on youtube (`https://www.youtube.com/watch?v=nmxV0FtsOnc`).

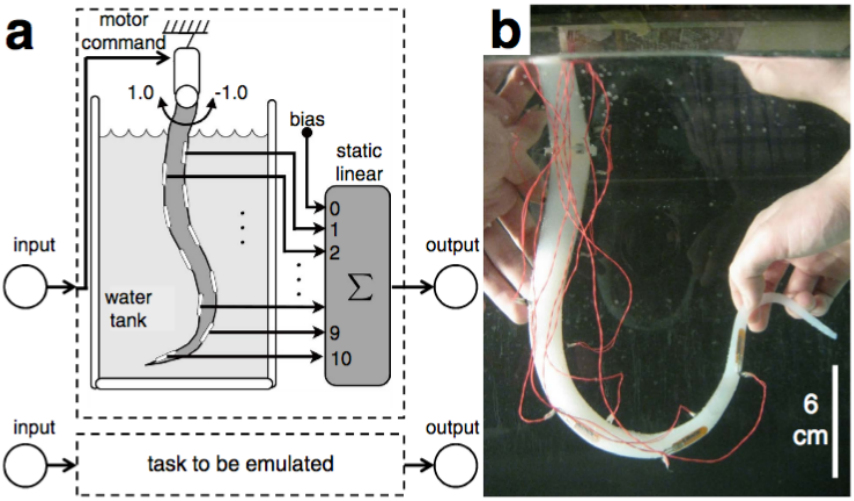

Figure 55: Reservoir computing in a plastic worm. In robotics, the control of body motion needs (among other items) a "forward model" of how the body limbs will react to motor or muscle action. While in classical industrial robotics this forward model can be calculated analytically with high precision, this is not possible with *soft* robot bodies or body parts — snakes, worms, trunks, tongues. One way to get such a model nonetheless is to use the very physical body (part) itself as a reservoir. Its states are observed by sensors placed in or on it. The twofold charm of this approach is, first, that this enables almost delay-less online computing, and second, that this reservoir naturally has exactly the right dynamical properties to "model" itself. At the University of Zurich, a team around Rolf Pfeifer realized this idea with a plastic worm. Their paper Nakajima et al. (2015) comes with a youtube demo (`https://www.youtube.com/watch?v=rUVAWQ7cvPg`).

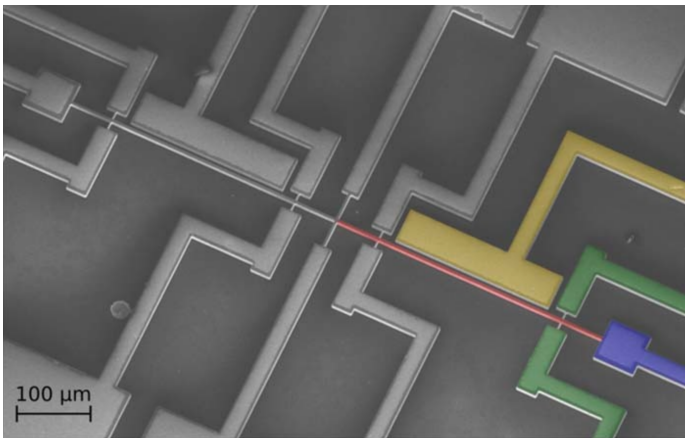

Figure 56: Reservoir computing in mechanical silicon microchips. Julien Sylvestre and his group at the Department of Mechanical Engineering, Université de Sherbrooke, Canada, explores how the *mechanical* oscillatory dynamics of freely suspended microscale silicon beams can be exploited in microchips which combine mechanical sensing with RC signal processing. The microbeams (marked red in image) can be etched into the silicon wafer with standard microchip fabrication technologies (Coulombe et al., 2017). Several such microbeams on the same chip interact nonlinearly with each other by mechanical couplings.

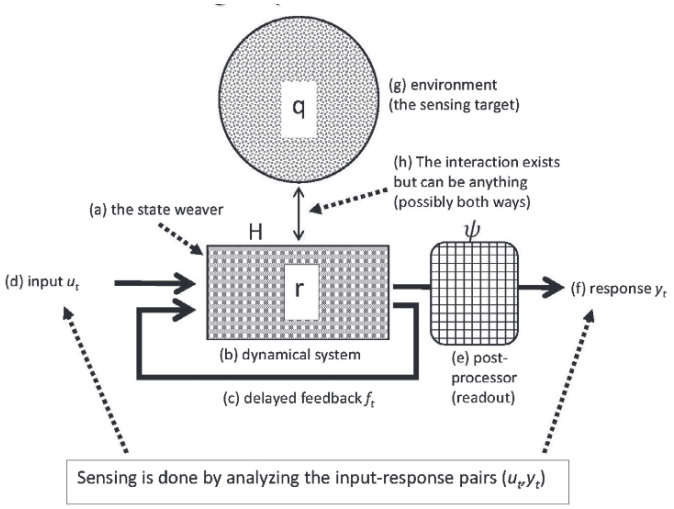

Figure 57: Reservoir sensing. A potentially quite promising future applications for RC is biochemical and environmental sensing. The reservoir is here a carrier plate coated with some chemical or biochemical material or mix of materials which change their properties when the surface is exposed to (traces of) chemical or biological substances whose presence or concentration has to be measured. The property changes induced in the active coating can be amplified and "dynamified" by additional electrical impulses given to the plate. The resulting spatiotemporal dynamics are recorded from the plate in some way, for instance electrically, and used as reservoir states. The desired measurement signal is trained by the RC principles. This line of sensor engineering was explored in a European FET-OPEN project (RECORD-IT, 2015-2018) coordinated by Zoran Konkoli from the Department of Microtechnology and Nanoscience, Chalmers University of Technology, Gothenburg, Sweden. An interim report is Konkoli (2016).

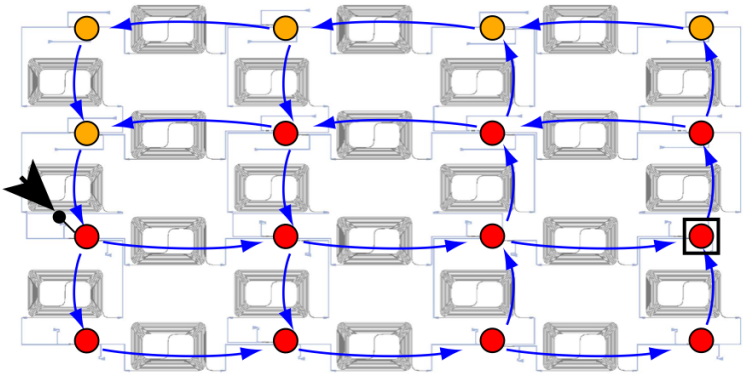

Figure 58: Optical reservoir computing. Optical computing — that is, computing with light instead of with electricity — is a large field of communication engineering, comparable in importance and promises with quantum computing. The potential benefits of optical versus electrical computing are (i) speed: the natural time constants of optical devices are many orders of magnitude smaller than of electronic devices; (ii) 3D wiring: unlike electrical signals, which need wires which in turn lead to headaches in microchip design because they must not cross each other, light beams can cross each other without interference, (iii) potentially extremely low energy consumption. These potential benefits have boosted optical computing research at a large scale, but breakthroughs are still missing — same as in quantum computing. One approach in this field is optical reservoir computing. This has developed into the currently most important branch among the many versions of physical RC (google `optical reservoir computing`). A wide spectrum of optical effects and reservoir architectures is being explored. The image shows a microphotograph of an optical microchip (real size 16 mm^2) developed at the University of Gent which implements an all-optical reservoir with 16 "neurons". It was demonstrated in Vandoorne et al. (2014) that with this reservoir one could realize 5-bit header recognition in internet packages. There are two aspects of this chip which fascinate me. First, the coils that you see in the image are long spirals of silicon waveguides whose function is to *slow down* the reservoir dynamics by inserting lengthy light travel paths between the "neurons". The native processing speed of this chip would be orders of magnitude too fast for feeding and analysing I/O signals with the available electronic lab equipment. Second, this chip is entirely passive: it needs no extra energy besides the energy in the incoming light signals. — Interestingly, very recently, also the quantum computing field has discovered RC as a potential venue for progress.

Appendix

A Elementary mathematical structure-forming operations

A.1 Pairs, tuples and indexed families

If two mathematical objects $\mathcal{O}_1, \mathcal{O}_2$ are given, they can be grouped together in a single new mathematical structure called the *ordered pair* (or just *pair*) of $\mathcal{O}_1, \mathcal{O}_2$. It is written as

$$(\mathcal{O}_1, \mathcal{O}_2).$$

In many cases, $\mathcal{O}_1, \mathcal{O}_2$ will be of the same kind, for instance both are integers. But the two objects need not be of the same kind. For instance, it is perfectly possible to group integer $\mathcal{O}_1 = 3$ together with a random variable (a function!) $\mathcal{O}_2 = X_7$ in a pair, getting $(3, X_7)$.

The crucial property of a pair $(\mathcal{O}_1, \mathcal{O}_2)$ which distinguishes it from the *set* $\{\mathcal{O}_1, \mathcal{O}_2\}$ is that the two members of a pair are *ordered*, that is, it makes sense to speak of the "first" and the "second" member of a pair. In contrast, it makes not sense to speak of the "first" or "second" element of the set $\{\mathcal{O}_1, \mathcal{O}_2\}$. Related to this is the fact that the two members of a pair can be the same, for instance $(2, 2)$ is a valid pair. In contrast, $\{2, 2\}$ makes no sense.

A generalization of pairs is *N-tuples*. For an integer $N > 0$, an N-tuple of N objects $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_N$ is written as

$$(\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_N).$$

1-tuples are just individual objects; 2-tuples are pairs, and for $N > 2$, N-tuples are also called *lists* (by computer scientists that is; mathematicians rather don't use that term). Again, the crucial property of N-tuples is that one can identify its i-th member by its position in the tuple, or in more technical terminology, by its *index*. That is, in an N-tuple, every index $1 \leq i \leq N$ "picks" one member from the tuple.

The infinite generalization of N-tuples is provided by *indexed families*. For any nonempty set I, called an *index set* in this context,

$$(\mathcal{O}_i)_{i \in I}$$

denotes a compound object assembled from as many mathematical objects as there are index elements $i \in I$, and within this compound object, every individual member $\mathcal{O}_i$ can be "addressed" by its index i. One simply writes

$$\mathcal{O}_i$$

to denote the ith "component" of $(\mathcal{O}_i)_{i \in I}$. Writing $\mathcal{O}_i$ is a shorthand for applying the ith projection function on $(\mathcal{O}_i)_{i \in I}$, that is, $\mathcal{O}_i = \pi_i((\mathcal{O}_i)_{i \in I})$.

A.2 Products of sets

We first treat the case of products of a finite number of sets. Let $S_1, \ldots, S_N$ be (any) sets. Then the product $S_1 \times \ldots \times S_N$ is the set of all N-tuples of elements from the corresponding sets, that is,

$$S_1 \times \ldots \times S_N = \{(s_1, \ldots, s_N) \mid s_i \in S_i\}.$$

This generalizes to infinite products as follows. Let I be any set — we call it an *index* set in this context. For every $i \in I$, let S_i be some set. Then the *product set indexed by* I is the set of functions

$$\prod_{i \in I} S_i = \{\varphi : I \to \bigcup_{i \in I} S_i \mid \forall i \in I : \varphi(i) \in S_i\}.$$

Using the notation of indexed families, this could equivalently be written as

$$\prod_{i \in I} S_i = \{(s_i)_{i \in I} \mid \forall i \in I : s_i \in S_i\}.$$

If all the sets S_i are the same, say S, then the product $\prod_{i \in I} S_i = \prod_{i \in I} S$ is also written as S^I.

An important special case of infinite products is obtained when $I = \mathbb{N}$. This situation occurs universally in modeling stochastic processes with discrete time. The elements $n \in \mathbb{N}$ are the points in time when the amplitude of some signal is measured. The amplitude is a real number, so at any time $n \in \mathbb{N}$, one records an amplitude value $a_n \in S_n = \mathbb{R}$. The product set

$$\prod_{n \in \mathbb{N}} S_n = \{\varphi : \mathbb{N} \to \bigcup_{n \in \mathbb{N}} S_n \mid \forall n \in I : \varphi(n) \in S_n\} = \{\varphi : \mathbb{N} \to \mathbb{R}\}$$

is the set of all right-infinite real-valued timeseries (with discrete time points starting at time $n = 0$).

A.3 Products of functions

First, again, the case of finite products: let $f_1, \ldots, f_N$ be functions, all sharing the same domain D, with image sets S_i. Then the product $f_1 \otimes \ldots \otimes f_N$ of these functions is the function with domain D and image set $S_1 \times \ldots \times S_N$ given by

$$\begin{aligned} f_1 \otimes \ldots \otimes f_N : D &\to S_1 \times \ldots \times S_N \\ d &\mapsto (f_1(d), \ldots, f_N(d)). \end{aligned}$$

Again this generalizes to arbitrary products. Let $(f_i : D \to S_i)_{i \in I}$ be an indexed family of functions, all of them sharing the same domain D, and where the image set of f_i is S_i. The product $\bigotimes_{i \in I} f_i$ of this set of functions is defined by

$$\begin{aligned} \bigotimes_{i \in I} f_i : D &\to \prod_{i \in I} S_i \\ d &\mapsto \varphi : I \to \bigcup_{i \in I} S_i \quad \text{given by } \varphi(i) = f_i(d). \end{aligned}$$

B Joint, conditional and marginal probabilities

Note. This little section is only a quick memory refresher of some of the most basic concepts of probability. It does not replace a textbook chapter!

We first consider the case of two observations of some part of reality that have discrete values. For instance, an online shop creating customer profiles may record from their customers their age and gender (among many other items). The marketing optimizers of that shop are not interested in the exact age but only in age brackets, say $a_1 = \mathsf{at\ most\ 10\ years\ old}$, $a_2 = 11-20$ years, $a_3 = 21-30$ years, $a_4 = \mathsf{older\ than\ 30}$. Gender is roughly categorized into the possibilities $g_1 = \mathsf{f}, g_2 = \mathsf{m}, g_3 = \mathsf{o}$. From their customer data the marketing guys estimate the following probability table:

$P(X = g_i, Y = a_j)$	a_1	a_2	a_3	a_4
g_1	0.005	0.3	0.2	0.04
g_2	0.005	0.15	0.15	0.04
g_3	0.0	0.05	0.05	0.01

(78)

The cell (i, j) in this 3×4 table contains the probability that a customer with gender g_i falls into the age bracket a_j. This is the *joint probability* of the two observation values g_i and a_j. Notice that all the numbers in the table sum to 1.

The mathematical tool to formally describe a category of an observable value is a *random variable* (RV). We typically use symbols $X, Y, Z, \ldots$ for RVs in abstract mathematical formulas. When we deal with concrete applications, we may also use "telling names" for RVs. For instance, in Table (78), instead of $P(X = g_i, Y = a_j)$ we could have written $P(\mathsf{Gender} = g_i, \mathsf{Age} = a_j)$. Here we have two such observation categories: gender and age bracket, and hence we use two RVs X and Y for gender and age, respectively. In order to specify, for example, that female customers in the age bracket 11-20 occur with a probability of 0.3 in the shop's customer reservoir (the second entry in the top line of the table), we write $P(X = g_1, Y = a_2) = 0.3$.

Some more info bits of concepts and terminology connected with RVs. You should consider a RV as the mathematical counterpart of a procedure or apparatus to make observations or measurements. For instance, the real-world counterpart of the Gender RV could be an electronic questionnaire posted by the online shop, or more precisely, the "what is your age?" box on that questionnaire, plus the whole internet infrastructure needed to send the information entered by the customer back to the company's webserver. Or in a very different example (measuring the speed of a car and showing it to the driver on the speedometer) the real-world counterpart of a RV Speed would be the total on-board circuitry in a car, comprising the wheel rotation sensor, the processing DSP microchip, and the display at the dashboard.

A RV *always* comes with a set of possible outcomes. This set is called the *sample space* of the RV, and I usually denote it with the symbol S. Mathematically,

a sample space is a set. The sample space for the Gender RV would be the set $S = \{\mathsf{m}, \mathsf{f}, \mathsf{o}\}$. The sample space for Age that we used in the table above was $S = \{\{0, 1, \ldots, 10\}, \{11, \ldots, 20\}, \{21, \ldots, 30\}, \{31, 32, \ldots\}\}$. For car speed measuring we might opt for $S = \mathbb{R}^{\geq 0}$, the set of non-negative reals. A sample space can be larger than the set of measurement values that are realistically possible, but it must contain *at least* all the possible values.

Back to our table and the information it contains. If we are interested only in the age distribution of customers, ignoring the gender aspects, we sum the entries in each age column and get the *marginal probabilities* of the RV Y. Formally, we compute

$$P(Y = a_j) = \sum_{i=1,2,3} P(X = g_i, Y = a_j).$$

Similarly, we get the marginal distribution of the gender variable by summing along the rows. The two resulting marginal distributions are indicated in the table (79).

	a_1	a_2	a_3	a_4	
g_1	0.005	0.3	0.2	0.04	**0.545**
g_2	0.005	0.15	0.15	0.04	**0.345**
g_3	0.0	0.05	0.05	0.01	**0.110**
	0.01	**0.5**	**0.4**	**0.09**	

(79)

Notice that the marginal probabilities of age $0.01, 0.5, 0.4, 0.09$ sum to 1, as do the gender marginal probabilities.

Finally, the *conditional probability* $P(X = g_i \,|\, Y = a_j)$ that a customer has gender g_i given that the age bracket is a_j is computed through dividing the joint probabilities in column j by the sum of all values in this column:

$$P(X = g_i \,|\, Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(Y = a_j)}. \tag{80}$$

There are two equivalent versions of this formula:

$$P(X = g_i, Y = a_j) = P(X = g_i \,|\, Y = a_j) P(Y = a_j) \tag{81}$$

where the righthand side is called a *factorization* of the joint distribution on the lefthand side, and

$$P(Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(X = g_i \,|\, Y = a_j)}, \tag{82}$$

demonstrating that each of the three quantities (joint, conditional, marginal probability) can be expressed by the respective two others. If you memorize one of these formulas – I recommend the second one – you have memorized the very key

to master "probability arithmetics" and will never get lost when manipulating probability formulas.

The factorization (81) can be done in two ways: $P(Y = a_j \,|\, X = g_i)P(X = g_i) = P(X = g_i \,|\, Y = a_j)P(Y = a_j)$, which gives rise to *Bayes' formula*

$$P(Y = a_j \,|\, X = g_i) = \frac{P(X = g_i \,|\, Y = a_j)P(Y = a_j)}{P(X = g_i)}, \tag{83}$$

which has many uses in statistical modeling because it shows how one can revert the conditioning direction.

Joint, conditional, and marginal probabilities are also defined when there are more than two categories of observations. For instance, the online shop marketing people also record how much a customer spends on average, and formalize this by a third random variable, say Z. The values that Z can take are spending brackets, say $s_1 =$ less than 5 Euros to $s_{20} =$ more than 5000 Euros. The joint probability values $P(X = g_i, Y = a_j, Z = s_k)$ would be arranged in a 3-dimensional array sized $3 \times 4 \times 20$, and again all values in this array together sum to 1. Now there are different arrangements for conditional and marginal probabilities, for instance $P(Z = s_k \,|\, X = g_i, Y = a_j)$ is the probability that among the group of customers with gender g_i and age a_j, a person spends an amount in the range s_k. Or $P(Z = s_k, Y = a_j \,|\, X = g_i)$ is the probability that in the gender group g_i a person is aged a_j and spends s_k. As a last example, the probabilities $P(X = g_i, Z = s_j)$ are the marginal probabilities obtained by *summing away* the Y variable:

$$P(X = g_i, Z = s_j) = \sum_{k=1,2,3,4} P(X = g_i, Y = a_k, Z = s_j) \tag{84}$$

So far I have described cases where all kinds of observations were *discrete*, that is, the respective sample spaces S were finite (for example, three gender values) or countably infinite (for example, the natural numbers $1, 2, 3, \ldots$). The function $P : S \to [0, 1]$ which assigns to each possible outcome $s \in S$ its probability $P(s)$ is called a *probability mass function* (pmf) and we denote it with an upper-case P. The sum of the pmf over all possible outcomes is one: $\sum_{s \in S} P(s) = 1$. If one behaves mathematically very correctly, the symbol P is indexed with the random variable that gives rise to the distribution, that is one would write $P_X(s)$. This is often not done out of convenience.

Equally often one faces *continuous* random values which arise from observations that yield real numbers – for instance, measuring the body height or the weight of a person. Since each such RV can give uncountably infinite many different observation outcomes, their probabilities cannot be represented in a table or array, and they cannot be summed up. Instead, one uses *probability density functions* (pdf's) to write down and compute probability values. We denote pdfs by lower-case p.

In order to explain pdfs, let's start with a single RV, say $H =$ Body Height. Since body heights are non-negative and, say, never larger than 3 m, the distribu-

tion of body heights within some reference population can be represented by a pdf $f : [0, 3] \to \mathbb{R}^{\geq 0}$ which maps the interval $[0, 3]$ of possible values to the nonnegative reals (Figure 59). We will be using subscripts to make it clear which RV a pdf refers to, so the pdf describing the distribution of body height will be written f_H.

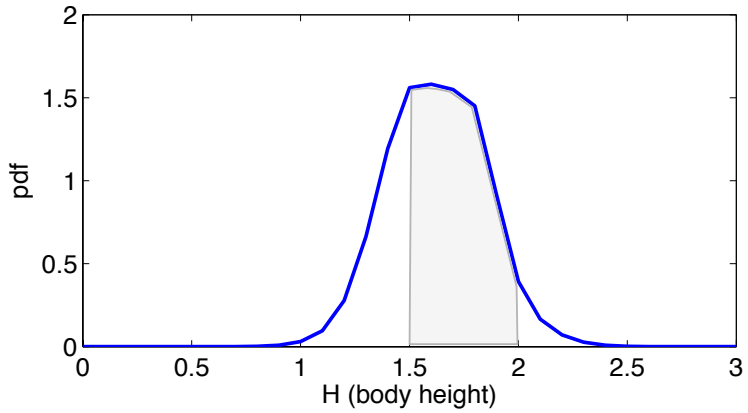

Figure 59: A hypothetical distribution of human body sizes in some reference population, represented by a pdf.

A pdf for the distribution of a continuous RV X can be used to calculate the probability that this RV takes values within a particular interval, by integrating the pdf over that interval. For instance, the probability that a measurement of body height comes out between 1.5 and 2.0 meters is obtained by

$$P(H \in [1.5, 2.0]) = \int_{1.5}^{2.0} f_H(x)dx, \tag{85}$$

see the shaded area in Figure 59. Some comments:

- A probability density function is actually *defined* to be a function which allows one to compute probabilities of value intervals as in Equation 85. For a given continuous RV X over the reals there is exactly one function f_X which has this property, *the* pdf for X. (This is not quite true. There exist also continuous-valued RVs whose distribution is so complex that it cannot be captured by a pdf, but we will not meet with such phenomena in this lecture. Furthermore, a given pdf can be altered on isolated points – which come from what is called a *null set* in probability theory – and still be a pdf for the same distribution. But again, we will not be concerned with such subtelties in this lecture.)

- As a consequence, any pdf $f : \mathbb{R} \to \mathbb{R}^{\geq 0}$ has the property that it integrates to 1, that is, $\int_{-\infty}^{\infty} f(x)dx = 1$.

- Be aware that the values $f(x)$ of a pdf are not probabilities! Pdf's turn into probabilities only through integration over intervals.
- Values $f(x)$ can be greater than 1 (as in Figure 59), again indicating that they cannot be taken as probabilities.

Joint distributions of two continuous RVs X, Y can be captured by a pdf $f_{X,Y} : \mathbb{R}^2 \to \mathbb{R}^{\geq 0}$. Figure 60 shows an example. Again, the pdf $f_{X,Y}$ of a bivariate continuous distribution must integrate to 1 and be non-negative; and conversely, every such function is the pdf of a continuous distribution of two RV's.

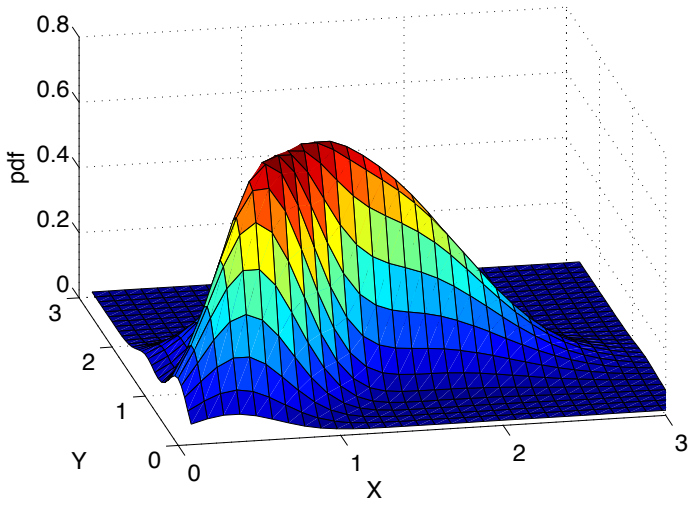

Figure 60: An exemplary joint distribution of two continuous-valued RVs X, Y, represented by its pdf.

Continuing on this track, the joint distribution of k continuous-valued RVs $X_1, \ldots, X_k$, where the possible values of each X_i are bounded to lie between a_i and b_i can be described by a unique pdf function $f_{X_1,\ldots,X_k} : \mathbb{R}^M \to \mathbb{R}^{\geq 0}$ which integrates to 1, i.e.

$$\int_{a_1}^{b_1} \ldots \int_{a_k}^{b_k} f(x_1, \ldots, x_k)\, dx_k \ldots dx_1,$$

where also the cases $a_i = -\infty$ and $b_i = \infty$ are possible. A more compact notation for the same integral is

$$\int_D f(\mathbf{u})\, d\mathbf{u},$$

where D denotes the k-dimensional box $[a_1, b_1] \times \ldots \times [a_k, b_k]$ and $\mathbf{u}$ denotes vectors in $\mathbb{R}^k$. Mathematicians speak of k-dimensional *intervals* instead of "boxes". The

set of points $S = \{\mathbf{u} \in \mathbb{R}^k \mid f_{X_1,\ldots,X_k} > 0\}$ is called the *support* of the distribution. Obviously $S \subseteq D$.

In analogy to the 1-dim case from Figure 59, probabilities are obtained from a k-dimensional pdf $f_{X_1,\ldots,X_k}$ by integrating over sub-intervals. For such a k-dimensional subinterval $[r_1, s_1] \times \ldots \times [r_k, s_k] \subseteq [a_1, b_1] \times \ldots \times [a_k, b_k]$, we get its probability by

$$P(X_1 \in [r_1, s_1], \ldots, X_k \in [r_k, s_k]) = \int_{r_1}^{s_1} \ldots \int_{r_k}^{s_k} f(x_1, \ldots, x_k)\, dx_k \ldots dx_1. \quad (86)$$

In essentially the same way as we did for discrete distributions, the pdf's of marginal distributions are obtained by *integrating away* the RV's that one wishes to expel. In analogy to (84), for instance, one would get

$$f_{X_1,X_3}(x_1, x_3) = \int_{a_2}^{b_2} f_{X_1,X_2,X_3}(x_1, x_2, x_3)\, dx_2. \quad (87)$$

And finally, pdf's of conditional distributions are obtained through dividing joint pdfs by marginal pdfs. Such conditional pdfs are used to calculate that some RVs fall into a certain multidimensional interval given that some other RVs take specific values. We only inspect a simple case analog to (80) where we want to calculate the probability that X falls into a range $[a, b]$ given that Y is known to be c, that is, we want to evaluate the probability $P(X \in [a, b] \mid Y = c)$, using pdfs. We can obtain this probability from the joint pdf $f_{X,Y}$ and the marginal pdf f_Y by

$$P(X \in [a, b] \mid Y = c) = \frac{\int_a^b f_{X,Y}(x, c)\, dx}{f_Y(c)}. \quad (88)$$

The r.h.s. expression $\int_a^b f_{X,Y}(x, c)\, dx \;/\; f_Y(c)$ is a function of x, parametrized by c. This function is a pdf, denoted by $f_{X \mid Y=c}$, and defined by

$$f_{X \mid Y=c}(x) = \frac{f_{X,Y}(x, c)}{f_Y(c)}. \quad (89)$$

Let me illustrate this with a concrete example. An electronics engineer is testing a device which transforms voltages V into currents I. In order to empirically measure the behavior of this device (an electronics engineer would say, in order to "characterize" the device), the engineer carries out a sequence of measurement trials where he first sets the input voltage V to a specific value, say $V = 0.0$. Then he (or she) measures the resulting current many times, in order to get an idea of the stochastic spread of the current. In mathematical terms, the engineer wants to get an idea of the pdf $f_{I \mid V=0.0}$. The engineer then carries on, setting the voltage to other values $c_1, c_2, ...$, measuring resulting currents in each case, and getting ideas of the conditional pdfs $f_{I \mid V=c_i}$. For understanding the characteristics of this device, the engineer needs to know all of these pdfs.

Conditional distributions arise whenever cause-effect relationships are being modeled. The conditioning variables are causes, the conditioned variables describe effects. In experimental and empirical research, the causes are under the control of an experimenter and can (and have to) be set to specific values in order to assess the statistics of the effects – which are not under the control of the experimenter. In ML pattern classification scenarios, the "causes" are the input patterns and the "effects" are the (stochastically distributed) class label assignments. Since research in the natural sciences is very much focussed on determining Nature's cause-effect workings, and 90% of the applications in machine learning concern pattern classification (my estimate), it is obvious that conditional distributions lie at the very heart of scientific (and engineering) modeling and data analysis.

In this appendix (and in the lecture) I consider only two ways of representing probability distributions: discrete ones by finite probability tables or probability tables; continuous ones by pdfs. These are the most elementary formats of representing probability distributions. There are many others which ML experts readily command on. This large and varied universe of concrete *representations* of probability distributions is tied together by an abstract mathematical theory of the probability distributions themselves, independent of particular representations. This theory is called *probability theory*. It is not an easy theory and we don't attempt an introduction to it. If you are mathematically minded, then you can get an introduction to probability theory in my graduate lecture notes "Principles of Statistical Modeling" (`http://minds.jacobs-university.de/teaching/ln/`). At this point I only highlight two core facts from probability theory:

- A main object of study in probability theory are distributions. They are abstractly and axiomatically defined and analyzed, without reference to particular representations (such as tables or pdfs).
- A probability distribution *always* comes together with random variables. We write P_X for the distribution of a RV X, $P_{X,Y}$ for the joint distribution of two RVs X, Y, and $P_{X\,|\,Y}$ for the conditional distribution (a truly involved concept since it is actually a family of distributions) of X given Y.

C The argmax operator

Let $\varphi : D \to \mathbb{R}$ be some function from some domain D to the reals. Then

$$\operatorname*{argmax}_a \varphi(a)$$

is that $d \in D$ for which $\varphi(d)$ is maximal among all values of φ on D. If there are several arguments a for which φ gives the same maximal value, – that is, φ does not have a unique maximum –, or if φ has no maximum at all, then the argmax is undefined.

D The softmax function

In many applications one wishes a neural network to output a probability vector. If the network has d output units, the d-dimensional output vector should be non-negative and its components should sum to 1. This allows one to treat the network output as a "hypothesis vector", for instance in order to express the network's "belief" in how an input pattern should be classified. However, the outputs of a trained MLP will not usually perfectly sum to 1, and the activations of output neurons may fall outside the range $[0, 1]$ of admissible probability values. In this situation one takes resort to a method for transforming any real-valued, d-dimensional vector $\mathbf{v} = (v_1, \ldots, v_d)' \in \mathbb{R}^d$ into a valid probability vector, by passing $\mathbf{v}$ through the *softmax* function:

$$\text{softmax}(\mathbf{v}) = \frac{1}{Z} \, (\exp(v_1), \ldots, \exp(v_d))', \tag{90}$$

where $Z = \sum_{i=1,\ldots,d} \exp(v_i)$ is the normalization constant.

The softmax is more than just a trick to enforce non-negativity and normalization of some vector. It is the key to an elementary machine learning algorithm called *logistic regression* (`https://en.wikipedia.org/wiki/Logistic_regression`) and has a direct connection to the Boltzmann distribution (compare Section 6.1).

E Expectation, variance, covariance, and correlation of numerical random variables

Recall that a random variable is the mathematical model of an observation / measurement / recording procedure by which one can "sample" observations from that piece of reality that one wishes to model. We usually denote RVs by capital roman letters like X, Y or the like. For example, a data engineer of an internet shop who wants to get a statistical model of its (potential) customers might record the gender and age and spending of shop visitors – this would be formally captured by three random variables G, A, S. A random variable always comes together with a *sample space*. This is the set of values that might be delivered by the random variable. For instance, the sample space of the gender RV G could be cast as $\{\text{m}, \text{f}, \text{o}\}$ – a symbolic (and finite) set. A reasonable sample space for the age random variable A would be the set of integers between 0 and 200 – assuming that no customer will be older than 200 years and that age is measured in integers (years). Finally, a reasonable sample space for the spending RV S could be just the real numbers $\mathbb{R}$.

Note that in the A and S examples, the sample spaces that I proposed look very generous. We would not really expect that some customer is 200 years old, nor would we think that ever a customer spends 10^{1000} Euros – although both values are included in the respective sample space. The important thing about a

sample space is that it must contain all the values that might be returned by the RV; but it may also contain values that will never be observed in practice.

Every mathematical set can serve as a sample space. We just saw symbolic, integer, and real sample spaces. Real sample spaces are used whenever one is dealing with an observation procedure that returns numerical values. Real-valued RVs are of great practical importance, and they allow many insightful statistical analyses that are not defined for non-numerical RVs. The most important analytical characteristics of real RVs are expectation, variance, and covariance, which I will now present in turn.

For the remainder of this appendix section we will be considering random variables X whose sample space is $\mathbb{R}^K$ — that is, observation procedures which return scalars (case $n = 1$) or vectors. We will furthermore assume that the distributions of all RVs X under consideration will be represented by pdf's $f_X : \mathbb{R}^K \to \mathbb{R}^{\geq 0}$. (In mathematical probability theory, more general numerical sample spaces are considered, as well as distributions that have no pdf — but we will focus on this basic scenario of real-valued RVs with pdfs).

The *expectation* of a RV X with sample space $\mathbb{R}^K$ and pdf f_X is defined as

$$E[X] = \int_{\mathbb{R}^K} x \, f_X(x) \, dx, \tag{91}$$

where the integral is written in a common shorthand for

$$\int_{x_1=-\infty}^{\infty} \dots \int_{x_n=-\infty}^{\infty} (x_1, \dots, x_n)' \, f_X((x_1, \dots, x_n)) \, dx_n \dots dx_1.$$

The expectation of a RV X can be intuitively understood as the "average" value that is delivered when the observation procedure X would be carried out infinitely often. The crucial thing to understand about the expectation is that it does not depend on a sample, – it does not depend on specific data.

In contrast, whenever in machine learning we base some learning algorithm on a (numerical) training sample $(x_i, y_i)_{i=1,\dots,N}$ drawn from the joint distribution $P_{X,Y}$ of two RVs X, Y, we may compute the average value of the x_i by

$$\text{mean}(\{x_1, \dots, x_N\}) = 1/N \sum_{i=1}^{N} x_i,$$

but this *sample mean* is NOT the expectation of X. If we would have used another random sample, we would most likely have obtained another sample mean. In contrast, the expectation $E[X]$ of X is defined not on the basis of a finite, random sample of X, but it is defined by averaging over the true underlying distribution.

Since in practice we will not have access to the true pdf f_X, the expectation of a RV X cannot usually be determined in full precision. The best one can do is to *estimate* it from observed sample data. The sample mean is an *estimator* for

the expectation of a numerical RV X. Marking estimated quantities by a "hat" accent, we may write

$$\hat{E}[X] = 1/N \sum_{i=1}^{N} x_i.$$

A random variable X is *centered* if its expectation is zero. By subtracting the expectation one gets a centered RV. In these lecture notes I use the bar notation to mark centered RVs:

$$\bar{X} := X - E[X].$$

The *variance* of a scalar RV with sample space $\mathbb{R}$ is the expected squared deviation from the expectation

$$\sigma^2(X) = E[\bar{X}^2], \tag{92}$$

which in terms of the pdf $f_{\bar{X}}$ of $\bar{X}$ can be written as

$$\sigma^2(X) = \int_{\mathbb{R}} x^2 \, f_{\bar{X}}(x) \, dx.$$

Like the expectation, the variance is an intrinsic property of an observation procedure X and the part of the real world where the measurements may be taken from — it is independent of a concrete sample. A natural way to estimate the variance of X from a sample $(x_i)_{i=1,\ldots,N}$ is

$$\hat{\sigma}^2(\{x_1, \ldots, x_N\}) = 1/N \sum_{i=1}^{N} \left(x_i - 1/N \sum_{j=1}^{N} x_j \right)^2,$$

but in fact this estimator is not the best possible – on average (across different samples) it underestimates the true variance. If one wishes to have an estimator that is *unbiased*, that is, which on average across different samples gives the correct variance, one must use

$$\hat{\sigma}^2(\{x_1, \ldots, x_N\}) = 1/(N-1) \sum_{i=1}^{N} \left(x_i - 1/N \sum_{j=1}^{N} x_j \right)^2$$

instead. The Wikipedia article on "Variance", section "Population variance and sample variance" points out a number of other pitfalls and corrections that one should consider when one estimates variance from samples.

The square root of the variance of X, $\sigma(X) = \sqrt{\sigma^2(X)}$, is called the *standard deviation* of X.

The *covariance* between two real-valued scalar random variables X, Y is defined as

$$\mathrm{Cov}(X, Y) = E[\bar{X} \, \bar{Y}], \tag{93}$$

which in terms of a pdf $f_{\bar{X}\,\bar{Y}}$ for the joint distribution for the centered RVs spells out to

$$\mathrm{Cov}(X,Y) = \int_{\mathbb{R}\times\mathbb{R}} x\,y\,f_{\bar{X}\,\bar{Y}}((x,y)')\,dx\,dy.$$

An unbiased estimate of the covariance, based on a sample $(x_i, y_i)_{i=1,\ldots,N}$ is given by

$$\widehat{\mathrm{Cov}}((x_i,y_i)_{i=1,\ldots,N}) = 1/(N-1)\,\left(x_i - 1/N\,\sum_i x_i\right)\left(y_i - 1/N\,\sum_i y_i\right).$$

Finally, let us inspect the *correlation* of two scalar RVs X, Y. Here we have to be careful because this term is used differently in different fields. In statistics, the correlation is defined as

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sigma(X)\,\sigma(Y)}. \tag{94}$$

It is easy to show that $-1 \leq \mathrm{Corr}(X,Y) \leq 1$. The correlation in the understanding of statistics can be regarded as a normalized covariance. It has a value of 1 if X and Y are identical up to some positive scaling factor, it has a value of -1 if X and Y are identical up to some negative scaling factor. When $\mathrm{Corr}(X,Y) = 0$, X and Y are said to be *uncorrelated*.

The quantity $\mathrm{Corr}(X,Y)$ is also referred to as *(population) Pearson's correlation coefficient*, and is often denoted by the greek letter $\varrho(X,Y) = \mathrm{Corr}(X,Y)$.

In the signal processing literature (for instance in my favorite textbook Farhang-Boroujeny (1998)), the term "correlation" is sometimes used in quite a different way, denoting the quantity

$$E[X\,Y],$$

that is, simply the expectation of the product of the uncentered RVs X and Y. Just be careful when you read terms like "correlation" or "cross-correlation" or "cross-correlation matrix" and make sure that your understanding of the term is the same as the respective author's.

There are some basic rules for doing calculations with expectations and covariance which one should know:

1. Expectation is a linear operator:

   $$E[\alpha\,X + \beta\,Y] = \alpha\,E[X] + \beta\,E[Y],$$

   where $\alpha\,X$ is the RV obtained from X by scaling observations with a factor α.

2. Expectation is idempotent:

   $$E[E[X]] = E[X].$$

3. $$\mathrm{Cov}(X,Y) = E[X\,Y] - E[X]\,E[Y].$$

References

D.H. Ackley, G.E. Hinton, and T.J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

D. J. Amit. *Modeling Brain Function: the World of Attractor Neural Networks*. Cambridge Univ. Press, NY, 1989.

D. J. Amit, H. Gutfreund, and H. Sompolinsky. Storing infinite numbers of patterns in a spin-glass model of neural networks. *Phys. Rev. Lett.*, 55(14):1530 – 1533, 1985.

Atomic Heritage Foundation authors. Computing and the Manhattan project. `https://www.atomicheritage.org/history/computing-and-manhattan-project`, 2014. Accessed: 2019.

F.C. Bartlett. *Remembering: a study in experimental and social psychology*. Cambridge University Press, 1932. Free legal online copies.

Y. Bengio and Y. LeCun. Scaling learning algorithms towards AI. In Bottou L., Chapelle O., DeCoste D., and Weston J., editors, *Large-Scale Kernel Machines*. MIT Press, 2007.

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer Verlag, 2006.

T. P. Castello-Waldow, G. Weston, A. Chenani, Y. Loewenstein, A. Chen, and Al Attardo. Stability of excitatory structural connectivity predicts the probability of CA1 pyramidal neurons to become engram neurons. biorxiv manuscript, `https://doi.org/10.1101/759225`, 2019.

G. Chlebus, A. Schenk, J.H. Moltz, B. van Ginneken, H. K. Hahn, and H. Meine. Automatic liver tumor segmentation in ct with fully convolutional neural networks and object-based postprocessing. *Scientific Reports*, 8:article number 15497, 2018. doi: 10.1038/s41598-018-33860-7.

K. Cho, B. van Berrienboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. manuscript, Universit de Montral, 2014. http://arxiv.org/abs/1409.1259.

J. C. Coulombe, M. C. A. York, and J. Sylvestre. Computing with networks of nonlinear mechanical oscillators. *PLOS ONE*, 12(6), 2017. URL `Https://doi.org/10.1371/journal.pone.0178663`.

S. Das. CNN architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more. Online article `https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5`, 2017. Accessed: 14 Mar 2020.

S. Demyanov. *Regularization Methods for Neural Networks and Related Models*. Phd thesis, Dept of Computing and Information Systems, Univ. of Melbourne, 2015.

P. F. Dominey. Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning. *Biological Cybernetics*, 73:265–274, 1995.

V. Evans, B. K. Bergen, and J. Zinken, editors. *The Cognitive Linguistics Reader*. Equinox Publishing Ltd, 2007. Open access copy at `http://www.academia.edu/download/59443724/COGNITIVE_LINGUISTICS-VYVYAN_EVANS20190529-8486-9kcjsi.pdf`.

B. Farhang-Boroujeny. *Adaptive Filters: Theory and Applications*. Wiley, 1998.

C. Fernando and S. Sojakka. Pattern recognition in a bucket. In W. Banzhaf, T. Christaller, P. Dittrich, J.T. Kim, and J. Ziegler, editors, *Advances in Artificial Life. Proc. 7th European Conference on Artificial Life (ECAL 2003)*, pages 588–597. Springer Verlag, 2003.

W. J. Freeman. Simulation of chaotic EEG patterns with a dynamic model of the olfactory system. *Biological Cybernetics*, 56:139–150, 1987.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

C. Gallicchio, A. Micheli, and L. Pedrelli. Design of deep echo state networks. *Neural Networks*, 108:33–47, 2018.

F. A. Gers and J. Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

I. Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. arXiv manuscript, arXiv:1701.00160v4, 2017.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Open access version at `http://www.deeplearningbook.org`.

N. W. Gouwens and R. I. Wilson. Signal propagation in drosophila central neurons. *The Journal of Neuroscience*, 29(19):6239–6249, 2009.

A. Graves, G. Wayne, and I. Danihelka. Neural Turing Machines. manuscript, http://arxiv.org/pdf/1410.5401v1.pdf, 2014.

A. Graves et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 7626:471–476, 2016.

S. Haykin. *Neural Networks: A Comprehensive Foundation. Second Edition.* Prentice Hall, 1999.

X. He, Sygnowski J., A. Galashov, A. A. Rusu, Y. W. Teh, and R. Pascaanu. Task agnostic continual learning via meta learning. In *Proc. neurIPS 2019 (submitted as arXiv:1906.05201v1)*, 2019.

D. O. Hebb. *The Organization of Behavior*. New York: Wiley & Sons, 1949.

J. Hertz, A. Krogh, and R. G. Palmer, editors. *An Introduction to the theory of neural computation*. Addison-Wesley, 1991.

X. Hinaut, M. Petit, G. Pointeau, and P. F. Dominey. Exploring the acquisition and production of grammatical constructions through human-robot interaction with echo state networks. *Frontiers in Neurorobotics*, 8:article 16, 2014. URL `https://doi.org/10.3389/fnbot.2014.00016`.

G. E. Hinton and R. R. Salakuthdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(July 28):504–507, 2006.

G. E. Hinton and T. J. Sejnowski. Optimal perceptual inference. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition 1983*, pages 448–453, 1983.

J. Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. Diploma thesis, Institut für Informatik, TU München, June 1991. URL `www.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf`.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.

J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci. USA*, 79:2554–2558, 1982.

I. Ilies, H. Jaeger, O. Kosuchinas, M. Rincon, V. Sakenas, and N. Vaskevicius. Stepping forward through echoes of the past: forecasting with echo state networks. URL `http://www.neural-forecasting-competition.com/downloads/NN3/methods/27-NN3_Herbert_Jaeger_report.pdf`. Short report on the winning entry to the NN3 financial forecasting competition, 2007.

H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. GMD Report 148, GMD - German National Research Institute for Computer Science, 2001. URL http://https://www.ai.rug.nl/minds/pubs.

H. Jaeger. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach. GMD Report 159, Fraunhofer Institute AIS, 2002. URL http://minds.jacobs-university.de/pubs.

H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304:78–80, 2004.

H. Jaeger, M. Lukosevicius, D. Popovici, and U. Siewert. Optimization and applications of echo state networks with leaky integrator neurons. *Neural Networks*, 20(3):335–352, 2007.

L. N. Kanal. Perceptrons. In *International Encyclopedia of the Social and Behavioral Sciences*, pages 11218–11221. Elsevier, 2001. URL http://www.iro.umontreal.ca/~kegl/ift3390/2006_1/Lectures/l03_PerceptronKaplan.pdf.

D. P. Kingma, D. J. Rezende, S. Mohamed, and M. Welling. Semi-supervised learning with deep generative models. In *Proc. NIPS 2014*, 2014. arXiv:1406.5298v2.

K. Kirby. Context dynamics in neural sequential learning. In *Proc. Florida AI Research Symposium (FLAIRS)*, pages 66–70, 1991.

S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

T. Kohonen. Self-organized formation of topologically correct feature maps. *Biol. Cybnernetics*, 43:59–69, 1982.

Z. Konkoli. On developing theory of reservoir computing for sensing applications: the state weaving environment echo tracker (sweet) algorithm. *International Journal of Parallel, Emergent and Distributed Systems,*, 2016. URL http://dx.doi.org/10.1080/17445760.2016.1241880.

G. Lakoff. *Women, fire, and dangerous things: What categories reveal about the mind*. University of Chicago, 1987.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf.

B. Litt and J. Echauz. Prediction of epileptic seizures. *The Lancet Neurology*, 1: 22–30, 2002.

M. Lukosevicius. A practical guide to applying echo state networks. In K.-R. Müller, G. Montavon, and G. Orr, editors, *Neural Networks Tricks of the Trade, Reloaded*, LNCS, pages 659–686. Springer Verlag, 2012.

W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002a. URL `http://www.lsm.tugraz.at/papers/lsm-nc-130.pdf`.

W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002b. http://www.cis.tugraz.at/igi/maass/psfiles/LSM-v106.pdf.

M. C. Mackey and L. Glass. Osciallation and chaosin physiological control systems. *Science*, 197(4300):287–289, 1977.

E. Marder and R. L. Calabrese. Principles of rhythmic motor pattern generation. *Physiological Reviews*, 76(3):687–717, 1996.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. of Mathematical Biophysics*, 5:115–133, 1943.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

M. Minsky. *The Society of Mind*. Pan Books, London, originally appeared 1985 in the united states edition, 1987.

M. L. Minsky and S. A. Papert. *Perceptrons*. Cambridge, MA: MIT Press., 1969.

V. Mnih, et al. Human-level control through deep reinforcement learning. *Nature*, 518(February 26):529–533, 2015.

K. Nakajima, H. Hauser, and R. Pfeifer. Information processing via physical soft body. *Scientific Reports*, 5:10487, 2015. URL `DOi:10.1038/srep1048`.

R.M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Dpt. of Computer Science, University of Toronto, 1993.

M. Olazaran. A sociological study of the official history of the perceptrons controversy. *Social Studies of Science*, 26(3):611–659, 1996.

G. I. Parisi, R Kemker, J. L. Part, C. Kanan, and S. Wermter. Continual lifelong learning with neural networks: A review. arxiv preprint arxiv:1802.07569, 2018.

J. Pathak, B. Hunt, M. Girvan, Z. Lu, and E. Ott. Model-free prediction of large spatiotemporally chaotic systems from data: A reservoir computing approach. *Phys. Rev. Let.*, 120:024102, 2018.

F. Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

A. Roudbari and F. Saghafi. Intelligent modeling and identification of aircraft-nonlinear flight. *Chinese Journal of Aeronautics*, 27(4):759–771, 2014.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing Vol. 1*, pages 318–362. MIT Press, 1986. Also as Technical Report, La Jolla Inst. for Cognitive Science, 1985.

D.E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1.* MIT Press, Cambridge, Mass., 1986.

M. A. Savi. Nonlinear dynamics and chaos. In V. Lopes Junior and et al, editors, *Dynamics of Smart Systems and Structures*, pages 93–117. Springer International Publishing Switzerland, 2016.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Preprint: arXiv:1404.7828.

L. Shastri. Advances in Shruti – a neurally motivated model of relational knowledge representation and rapid inference using temporal synchrony. *Artificial Intelligence*, 11:79–108, 1999.

G. Tanaka and et al. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019. preprint in `https://arxiv.org/abs/1808.04962`.

C. van Vreeswijk and D. Hansel. Patterns of synchrony in neural networks with spike adaptation. *Neural Computation*, 13(5):959–992, 2001.

K. Vandoorne, T. Mechet, P. abd Van Vaerenbergh, and et al. Experimental demonstration of reservoir computing on a silicon photonics chip. *Nature Communications*, 5:3541, 2014. URL `DOI:10.1038/ncomms4541`.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proc. ICML*, pages 1096–1103, 2008.

R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.

B. J. Wolf. *Hydrodynamical Imaging with Artificial Intelligence*. Phd thesis, Faculty of Engineering and Science, University of Groningen, 2020.

F. Zenke, E. J. Agnes, and W. Gerstner. Diverse synaptic plasticity mechanisms orchestrated to form and retrieve memories in spiking neural networks. *Nature Communications*, 6, 2014. article nr 6922.