Advanced Machine Learning

Jacobs University Bremen

http://minds.jacobs-university.de/teaching/MLFall11

Herbert Jaeger

Lecture Notes

Version status:
Mar 29, 2013: corrected some URLs
Sep 6, 2011: created by copy from the 2009 version
Sep 26, 2011: corrected math typos in Section 2.9
Oct 4, 2011: extended and corrected Sections 4.4.1, 4.4.2
Oct 7, 2011: improved notation and explanations in 4.4.2
Oct 9, 2011: thorough workover and extension of entire 4.4
Oct 13, 2011: added paragraphs on linear regression in 4.4
Nov 1, 2011: small add-ons and improvements in beginnings of HMM section
Nov 3, 2011: replaced "measure space" by "observation space" throughout
Nov 11, 2011: added Section 10.6 on sequence model comparison
Nov 24, 2011: corrected and improved equations 8.10, 8.11
Jan 11, 2012: corrected typos in Sections 1 -- 5

1. Introduction

1.1 What this lecture is – unfortunately – NOT about

What do you see?

(From an ad for "Fiddler on the Roof", www.addaclevenger.org/show/tophat.jpg)

Don't behave like an image processing algorithm. I think you can "see" much more:

What's sticking out in the left lower corner?
What's the expression on the face of that man?
What will be his next movement?
How does the hat smell?
Other questions you could answer...

How did your answers get there where they come from?

Human learning: uncomprehensively rich and complex, involving aspects of

- body growth
- brain development
- motion control
- exploration, curiosity, play
- creativity
- social interaction
- drill and exercise and rote learning

- reward and punishment, pleasure and pain
- evolution
- dreaming
- remembering
- forgetting
- & 1000 more ...
-

and all is integrated into your person, makes up your individual personality

You are what you learnt.

What must be in place for you to become yourself through learning?

- The universe, the earth, the atmosphere, water, food, caves
- Your body, brain, sensor & motor apparatus
- Physiology and neurophysiology
- Evolution
- Other people, living
- Other people, long dead
- Machines, tools, buildings, toys
- Words and sentences
- Concepts and meanings
- Letters and books
- Traditions
- Schools

Your and your learning are part of the world's development

By the way, does evolution learn?
Is, what is learnt by an individual, affecting evolution? Can the results of learning be passed on to siblings genetically? The Baldwin effect says, yes it can, albeit indirectly.

Excerpt from http://www.geocities.com/Athens/4155/edit.html:

At the turn of the century, it was unclear whether Darwin's theory or Lamarck's better explained evolution. Lamarck believed in direct inheritance of characteristics acquired by individuals during their lifetime. Darwin proposed that natural selection coupled with diversity could largely explain evolution. Darwin himself believed that Lamarckian evolution might play a small role in life, but most Darwinians rejected Lamarckism. One potentially verifiable difference between the two theories was that Darwinians were committed to gradualism (evolution in tiny, incremental steps), while Lamarckians expected occasional rapid change. Lamarckians cited the gaps in the fossil record (which are now associated with punctuated equilibria) as supporting evidence.

Lamarckism was a viable theory until August Weismann's (1893) work was widely accepted. Weismann argued that higher organisms have two types of cells, germ cells that pass genetic information to offspring and somatic cells that have no direct role in reproduction. He argued that there is no way for information acquired by somatic cells to be transmitted to germ cells.

In the context of this debate, James Mark Baldwin (1896) proposed "a new factor in evolution", whereby acquired characteristics could be indirectly inherited. Morgan (1896) and Osborn (1896) independently proposed similar ideas. The "new factor" was phenotypic plasticity: the ability of an organism to adapt to its environment during its lifetime. The ability to learn is the most obvious example of phenotypic plasticity, but other examples are the ability to tan with exposure to sun, to form a callus with exposure to abrasion, or to increase muscle strength with exercise. Baldwin (1896) pointed out that, among other things, the new factor could explain punctuated equilibria.

The Baldwin effect works in two steps. First, phenotypic plasticity allows an individual to adapt to a partially successful mutation, which might otherwise be useless to the individual. If this mutation increases inclusive fitness, it will tend to proliferate in the population. However, phenotypic plasticity is typically costly for an individual. For example, learning requires energy and time, and it sometimes involves dangerous mistakes. Therefore there is a second step: given sufficient time, evolution may find a rigid mechanism that can replace the plastic mechanism. Thus a behavior that was once learned (the first step) may eventually become instinctive (the second step). On the surface, this looks the same as Lamarckian evolution, but there is no direct alteration of the genotype, based on the experience of the phenotype. This effect is similar to Waddington's (1942) "canalization".

The Baldwin effect came to the attention of computer scientists with the work of Hinton and Nowlan (1987). The Baldwin effect may arise in evolutionary computation when a genetic algorithm is used to evolve a population of individuals that also employ a local search algorithm. Local search is the computational analog of phenotypic plasticity in biological evolution. In computational terms, in the first step of the Baldwin effect, local search smooths the fitness landscape, which can facilitate evolutionary search. In the second step, as more optimal genotypes arise in the population, there is selective pressure for reduction in local search, driven by the intrinsic costs associated with the search.

1.2 What this lecture is about

This lecture is about *inductive learning*: given observations / experiences / examples, derive a model / description / concept / rule (we'll stick to *model*)

Most important examples:

- Categorization (I see a *man* who wears a *hat*)
- Second-to-second action result expectations (if I let go of this pen, it will fall down)
- Skills, professional and everyday
- Everyday theories ("plants grow because rain falls and sun shines")
- Models in the natural sciences

Some might claim that induction *is* learning *is* induction and that Science is The Big Learning, but we have learnt otherwise a few minutes ago.

This lecture is about inductive learning performed by algorithms: given a data set ("training data", "sample"), condense it into a data structure that can be used for various purposes, such as simulation, prediction, filtering (de-noising, transformation), classification, failure monitoring, pattern completion.

Examples:

- Given a set of plant descriptions with dozens to hundreds anatomical details per plant, derive an efficient classification scheme (as you find it in botanic field guides)

- Given a sample of profile data records of customers, classify them into target groups for marketing.

- Given a sample of profile data records of startup companies, predict the risk of insolvency for a new startup company, together with predicting the inaccuracy of your prediction.

- Given 100,000 spam emails and 1,000 non-spam emails, create a spam filter.

- Given audiorecordings of properly working gearboxes and of gearboxes shortly before failure, develop a monitoring system that warns you of imminent failure.

- Given some thousand samples of hand-written postal codes, train an automated postal code recognizer.

- Given a corrupted radio signal with echo and noise and amplifier distortion, distil a de-distortion filter (an *equalizer*) which undoes these effects and returns the "clean" signal; and, do this within milliseconds (your cellphone does it).

- Read a long text aloud into a microphone. From that recording, train a personal dictating system that transforms your speech into text. When the system makes mistakes, speak the mistyped word a few times and let the system adapt.

- Given test data from a combustion engine run under many different working conditions, learn an automated control device that minimizes fuel consumption.

- Given a team of soccer robots and a lot of test game recordings, develop a behavior control module that lets the team win.

In this lecture, you will learn techniques that enable you to start coping with most of the examples.

You see, it's useful. But it's a far cry from the totality of human learning. And it involves statistics... Don't be afraid, though, it's so powerful that statistics turns into a good feeling.

1.3 What this lecture isn't about either

This lecture is mostly about inductive learning of *numerical* models. The methods we will see have mostly been developed in the fields of pattern recognition, signal processing, neural networks and statistical decision making. The books of Bishop and Duda/Hart/Stork are representative (see references list on course homepage).

This is one big portion of the field of machine learning. Another big portion is concerned with learning *symbolic* models, for instance, deriving sets of logical rules or even little computer programs from training data. Such techniques often have a typical "artificial intelligence" flavour. Here the book of Mitchell is a good source.

Finally, there are two further big fields of machine learning that we will not touch: genetic algorithms / evolutionary optimization, and reinforcement learning. The former installs artificial evolutionary processes to finding (sometimes amazingly "creative") solutions to complex optimization tasks, the latter deals with leraning optimal action policies for autonomous agents when during training only scarce reward / punishment signals are provided. Mitchell's book gives short intros for both.

1.4. Technical remarks

The course homepage is at http://minds.jacobs-university.de/teaching/MLFall11. There you can find this script, references, exercise sheets, software etc.

2. A first tour of topics

2.1. Introducing the Digits example

The digits dataset and all Matlab routines used in this section are available at http://www.faculty.jacobs-university.de/hjaeger/courses/MLFall04/OnlineGuide.html. *If you want to play around with this example, go ahead!*

In order to get a feeling for the issues of ML, in the first few sessions we will informally consider a simple learning task and learn about some crucial themes of that will turn up again and again throughout the course.

Assume you want to train an algorithm to distinguish the written symbols "1" and "0". Part of the training material might look like the first two lines in the following figure:

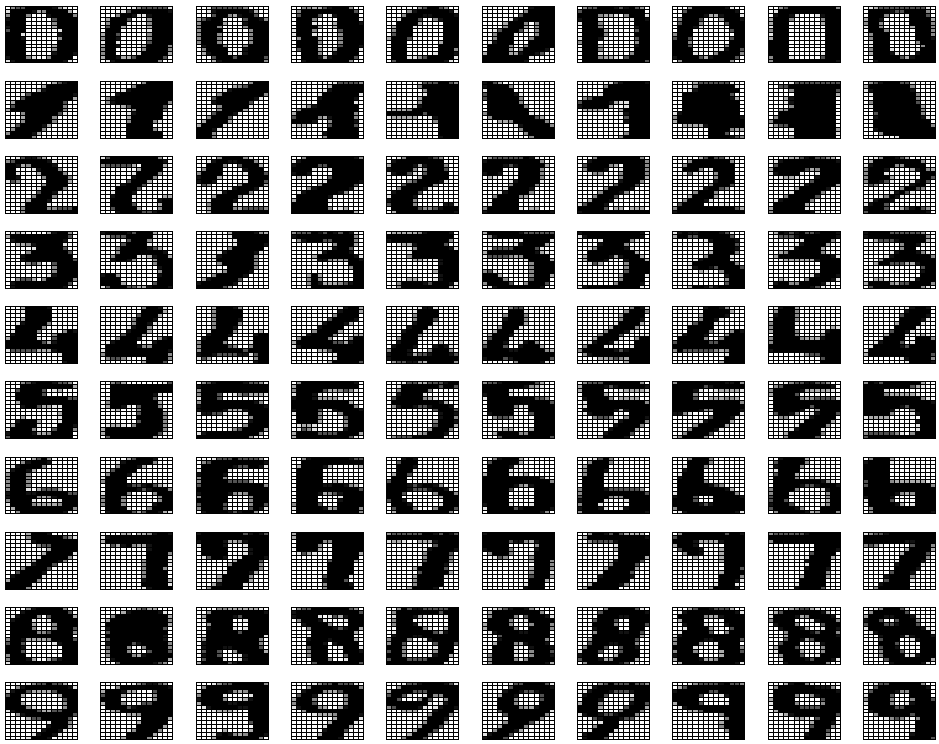


Figure 2.1: part of the digits training sample (created from a benchmark dataset donated by Robert Duin, orginally retrieved from http://ftp.ics.uci.edu/pub/ml-repos/machine-learning-databases/mfeat/mfeat-pix, now also at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/mfeat-pix.txt and http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/mfeat.info.txt.

Technically speaking, these samples are two-dimensional arrays of size 15x16, or alternatively vectors $\mathbf{x}_i$ of length 240, where $i = 1, ..., N$ and N is the size of the training set. The values of the vector components indicate greyscale values. In our digits dataset, these

values are integers ranging from 0 to 6, and for "zero" and "one" patterns this data set contains 200 samples, so here $N = 400$.

In addition, this training set contains the information whether some input vector is a zero or a one. We code this by introducing a class label $y_i \in \{1, 2\}$, where $y_i = 1$ if $\mathbf{x}_i$ represents a "zero" and $y_i = 2$ if $\mathbf{x}_i$ represents a "one". Thus, your training data is a set of *labelled samples*:

(2.1) $$(\mathbf{x}_i, y_i)_{i = 1,..., N}$$

What in the end you want to have is an algorithm that accepts as input some (new) alternatively vectors $\boldsymbol{x}$ and produces as output a "1" or a "2" according to whether the input vector represents a one or a zero. Formally, such an algorithm implements a binary *decision function*

(2.2) $$y = \hat{f}(\mathbf{x}), \text{ where } y \in \{1, 2\}.$$

We generally use the hat, ^, to mark functions or variables that are *learnt*, or – as statisticians would say with equal right – *estimated* from data. In writing $y = \hat{f}(\mathbf{x})$, we indicate that we believe that there is a *correct* (but unknown) decision function *f*, of which $\hat{f}$ is an estimate obtained through some learning method (or even by inspired hand design).

You might find it easy to explicitly write a little algorithm, inspired by insight into the nature of 0's and 1's, that implements some reasonable $\hat{f}$. The zero and one pictures have some very distinctive characteristics that you might be able to exploit. But you might find it much more difficult to come up with a program that distinguishes between ones and twos, or between ones an sevens!

Even humans are sometimes unsure about a classification. Look at the last "one" example in Figure 2.1. If you wouldn't know it is a "one", you might be unsure whether it might not also be a "zero".

Facing this intrinsic uncertainty of classifications, a more adequate type of classification algorithm would not return on input $\boldsymbol{x}$ a single class label, but should rather return a *hypothesis vector*: $(P(y = 1 \mid \mathbf{x}), P(y = 2 \mid \mathbf{x}))$. Because $P(y = 2 \mid \mathbf{x}) = 1 - P(y = 1 \mid \mathbf{x})$, one of the two hypothesis components is redundant, and it is enough if the probability $p = P(y = 1 \mid \mathbf{x})$ is returned. That is, such an algorithm would implement a *hypothesis function*

(2.3) $$p = \hat{f}(\mathbf{x}), \text{ where } p \in [0,1].$$

The explicit design of a classification algorithm soon becomes tedious if not impractical, and it would be hard to write an algorithm that returns reasonable hypotheses instead of just a binary decision.

Bright idea: write a *learning algorithm* instead, that reads the training samples (2.1) and automatically generates a function *f* of the kind (2.2) or (2.3)!

2.2. The curse of dimensionality

We will now try to design an ad hoc, brute-force learning algorithm L for a decision function f of type (2.2).

Since in the dataset we are using, vectors x coding for digits have integer values ranging between 0 and 6, the input space for f is a 240-dimensional, integer-valued cube X with edge length 6. Our (first, crude) approach to design L is learning by *frequency counts*:

- Partition X into a number of subcubes X_j, where $j = 1, ..., d$.
- For each subcube X_j, L considers all training samples $(\mathbf{x}_i, y_i)$ with $\mathbf{x}_i \in X_j$ and counts how many of them are "zeros" (= $N_0(j)$) and how many are "ones" (= $N_1(j)$).
- Store all these $N_0(j)$, $N_1(j)$.
- $\hat{f}(\mathbf{x})$ is then defined as follows:
    - Determine index j of the subcube with $\mathbf{x} \in X_j$.
    - If $N_0(j) > N_1(j)$ return 1.
    - If $N_0(j) < N_1(j)$ return 2.
    - If $N_0(j) = N_1(j)$ return ?.

Hopeless! Why?

The coarsest reasonable partition of X into subcubes X_j would halve the edges of X and create subcubes of edge length 3. This would give 2^{240} subcubes! That is, only in a vanishingly small number of subcubes would we find a training sample at all. In the vast majority of subcubes, $N_0(j)$ and $N_1(j)$ would be 0 and f wouldn't be defined. Even if we had a number N of training samples amounting to the number of atoms in the universe, the situation would hardly improve! (plus, we would run into storage problems).

So our naive approach fails completely.

This situation is common to many ML tasks and is known as the

Curse of dimensionality: Training samples $(\mathbf{x}_i, y_i)$ that are high-dimensional vectors populate their input vector space X so thinly that it is meaningless to use them directly to define a decision or hypothesis function $\hat{f}$ over X.

2.3 Using features

A way to escape from the curse of dimensionality: use a few *features* instead of original input vectors. The function of features is to reduce the dimensionality of the input space.

Technically, a feature F is a one-dimensional function over the original input vectors $\boldsymbol{x}$. Examples in the digit domain:

- $F_1(\mathbf{x})$ = sum of components of $\mathbf{x}$ / 240 = $\mathbf{1}^T \mathbf{x}$ / 240, where $\mathbf{1}$ is the 240-vector of all ones and ^{T} denotes transpose (in this script, all vectors are assumed to be column vectors unless noted otherwise). This is the average brightness of the digit picture. Might be useful to distinguish between "ones" and "zeros" because "zeros" should use more ink than "ones".

- $F_2(\mathbf{x})$ = 0 if in the middle row of the digit picture there are some black pixels followed by some white pixels followed again by some black pixels, and = 1 else. Might single out "zeros" from "ones" because $F(\mathbf{x}) = 0$ seems typical for "zeros". (But look at the fourth "one" image in Fig. 2.1).

- Construct F_3 as follows. First, create a pixel image with a clean, clear, "one". Transform this image into a vector $\mathbf{x}_{best1}$. Put $F(\mathbf{x}) = < \mathbf{x}_{best1}, \mathbf{x} > = \mathbf{x}_{best1}^T \mathbf{x}$. This is the inner product between $\mathbf{x}$ and $\mathbf{x}_{best1}$. Intuitively, F_3 measures the overlap of a pattern $\mathbf{x}$ with the *prototype* $\mathbf{x}_{best1}$, and thus should be large for patterns $\mathbf{x}$ representing "ones" and small for patterns representing "zeros".

There are very many ways to define useful features, and we will learn about more of them.

Features are a tool for *dimensionality reduction*. If you have some few (hopefully relevant) features $F_1, ..., F_m$, rewrite your high-dimensional input patterns $\mathbf{x}$ into low-dimensional *feature vectors* $F(\mathbf{x}) = (F_1(\mathbf{x}), ..., F_m(\mathbf{x}))$. This transformation is called *feature extraction*. Use $(F(\mathbf{x}_i), y_i)_{i = 1,..., N}$ as training material. The learning task is then to distil from this material a function $y = \hat{f}(F(\mathbf{x}))$ that gets as input feature vectors and returns classification labels.

Finding "good" features for dimensionality reduction is a key factor in many ML tasks. It is also something of an art, because the "good" features are often task-specific and require some insight into the "nature" of the data (compare F_2 from the list above). Unfortunately, there is no known universal way to define the "best" features for a given learning task (but there are good working solutions that are quite generally applicable).

Using low-dimensional features, our ad hoc approach of learning by frequency counts suddenly makes sense. In the simplest case, we only use a single feature. The space X on which f operates is now one-dimensional (the dimension of the feature value).

Using feature F_3 and 10 subcubes (= subintervals here) we get the following histogram for $N_0(j)$, $N_1(j)$:

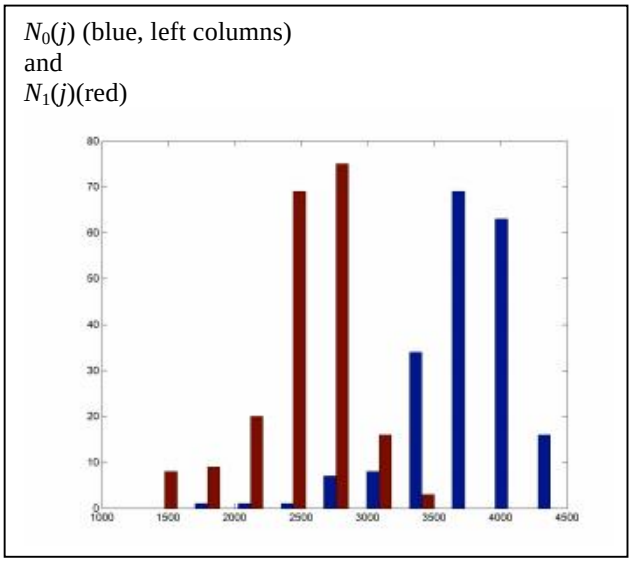


Figure 2.2. Frequency counts for "zero" and "one" training patterns w.r.t. feature F_3

It becomes clear from this figure that a decision function $\hat{f}$ based on this feature alone would classify the majority of the *training* examples correctly but would also yield some misclassifications. In fact, 5.25 per cent of the training samples would be misclassified – apparently more misclassifications than a human would make. We will learn to do much better.

2.4 Regression and time series prediction. Introducing the Mackey-Glass attractor example.

The digit recognition task is a classification task. The training patterns for a classification task are of the kind $(\mathbf{x}_i, y_i)$, where y_i is a discrete-valued *class label*. "Discrete-valued" here means that there are only finitely many possible values y_i can take.

If the y_i in the training patterns are allowed to take "analog", real-number values, then the training patterns $(\mathbf{x}_i, y_i)$ are best seen as argument-value pairs of some *function*, that is, $y_i = f(\mathbf{x}_i)$, and the learning task becomes one of estimating the function from training examples, that is, to obtain some $y = \hat{f}(\mathbf{x})$. Estimating real-valued functions from training data is called *regression.* In a sense, classification is just a special case of regression, where the function f is intended to take only discrete (class label) values.

Often, the training patterns come from *noisy* measurements. Figure 2.3 shows a noisy "measurement" of a linear and the square function; the crosses mark the $(\mathbf{x}_i, y_i)$ training patterns.

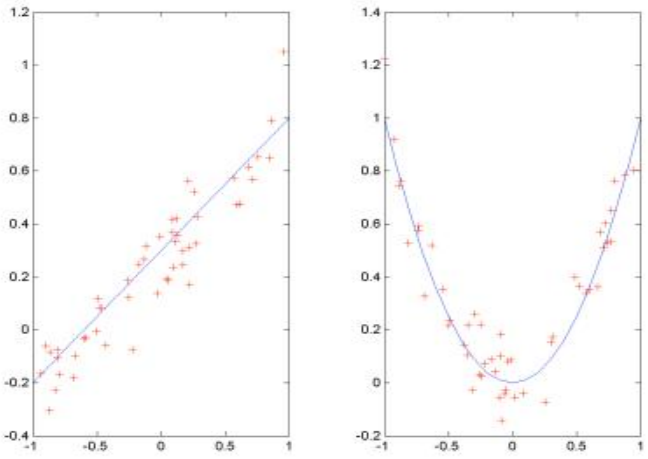


Figure 2.3: A linear and the square function (solid lines), represented by "noisy observations" (crosses).

If one wishes to obtain a linear function (as in the left panel of Fig. 2.3), one speaks of a linear regression; otherwise, of a nonlinear regression. The term "regression" typically means that one deals with noisy training patterns, but sometimes it is also used for noise-free training samples. (While we are at it: the term "sample" may refer to a single training instance (one red cross in Fig. 2.3), but is likewise used to denote the complete training set – don't blame *me* for this confusion.)

Regression tasks abound in practical applications. Here are some examples:

- Predict the time a cancer patient will live. Training samples $(\mathbf{x}_i, y_i)$: $\mathbf{x}_i$ is a vector containing diagnostic measurements of patient i, y_i is the time this patient lived after diagnosis.
- Assess the quality of a production batch leaving a chemical factory. In training samples $(\mathbf{x}_i, \mathbf{y}_i)$, $\mathbf{x}_i$ is a vector of measurements made during the production process, $\mathbf{y}_i$ is a vector of quality criteria describing the end product. This is not only interesting to predict the quality of an end product currently being processed, but also (applied in reverse) for finding "manufacturing tuning parameters" $\mathbf{x}$ that will yield a product with certain desired characteristics $\mathbf{y}$.
- Image restoration: Training samples $(\mathbf{x}_i, \mathbf{y}_i)$ are pairs of a corrupted image $\mathbf{x}_i$ and its non-corrupted form $\mathbf{y}_i$. Here, both $\mathbf{x}_i$ and $\mathbf{y}_i$ are grayscale vectors.

Regression is also key in a type of problem where one would not, at first sight, expect it: time series prediction. We will see now how time series prediction can be seen as a regression task.

By way of example, we consider a time series which is generated by a deterministic equation, namely, the Mackey-Glass (MG) equation. The MG equation was first introduced as a model for the onset of leukaemia[1]. It describes a chaotic system and has been used extensively as a benchmark system in time series prediction research, so there is a big number of articles available that treat the prediction task for the MG system and which allow one to compare one's own prediction performance with the state of the art. Originally, the MG equation is a

[1] J. McNames, J. A. K. Suykens, J. Vandewalle, *Int. J. of Bifurcation and Chaos* **9**, 1485 (1999)

so-called delay differential equation. When it is discretized with stepsize δ (that is, changed from continuous time *t* to discrete time *n*), one gets an approximate update equation that looks like this:

$$x(n+1) = x(n) + \delta\left(\frac{0.2\,x(n-\tau/\delta)}{1+x(n-\tau/\delta)^{10}} - 0.1\,x(n)\right) \tag{2.4}$$

This equation describes how the value $x(n+1)$ can be computed from previous values. The *delay* parameter τ is usually set to 17 or to 30, that is, $x(n+1)$ depends on the previous value $x(n)$ and the value 17 update steps before *n*. The larger τ , the more chaotic the resulting time series. Fig. 2.4 shows how Eq. (2.4) evolves for τ = 17 and τ = 30 (a stepsize of δ = 0.1 was used). For values of τ less than 17, the resulting time series is not chaotic but only periodic.

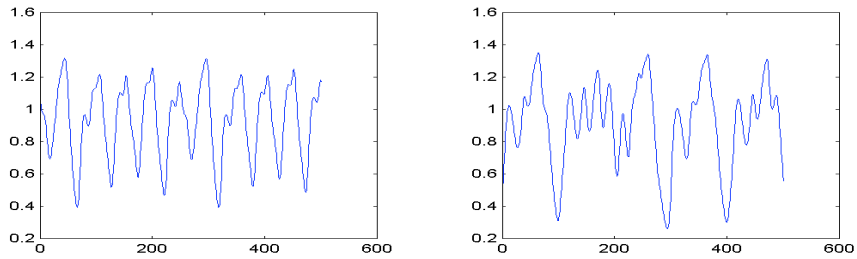


Figure 2.4: Evolution in time of the MG system for delays τ = 17 and τ = 30.

Time series prediction tasks. In the case of discrete time *n*, and a one-dimensional time series $(x_n)_{n=1,2,3,\ldots}$, a time series prediction task is set up as follows:

- Given: an initial sequence $x_1, x_2, \ldots, x_N$. generated by some dynamical system (in our case, Eq. (2.4)).
- NOT given: knowledge of the generating system!
- Wanted: an estimated continuation of this sequence $\hat{x}_{N+1}$, $\hat{x}_{N+2}$, ...

For the MG example, the initial sequence might for instance consist in the first 400 points of Fig. 2.4 (left), and the prediction task would be to generate the next points. Here are some other important time series prediction tasks:

- Predict the currency exchange rate, given the currency charts up to NOW.
- Predict the weather.
- Predict the next move of your opponent in a game (for instance, chess or real-life warfare – yes, that's being tried!)
- Predict where a comet is heading (will it hit us?)

- Predict the next sounds in a speech utterance (this is important for automated speech processing systems, because such predictions are required to "filter" hypotheses about what has actually been said).

- This is a learning task in the sense that in order to create the continuation $\hat{x}_{N+1}$, $\hat{x}_{N+2}$, ...one has to induce (= "learn") from the given initial sequence $x_1, x_2, ..., x_N$ a model M of the (unknown!) generating system, and then use M to compute the continuation.

What kind of mathematical thing is M?

There are many, many possibilities. In fact, M might take the form of *any* mathematical/algorithmical object capable of generating a time series. For instance, M might be some kind of deterministic automaton, or a random system like a Markov chain, or a game strategy, or simply an update equation of the form $x(n+1) = \hat{f}(x(n))$. A good deal of ingenuity and intuitive insight is typically required to settle for a "suitable" type of algorithmical object for a given time series prediction task.

Now we will see how the task of time series prediction can be cast as a regression task. The idea is to establish M as a *regression function* $\hat{f}$ (which has to be learnt from the training data $x_1, x_2, ..., x_N$). Consider a short section of the MG series (Fig. 2.5) for an illustration: the idea is to learn a function $\hat{f}$ which takes as arguments some previous, already known values of the time series and computes the next value from those.

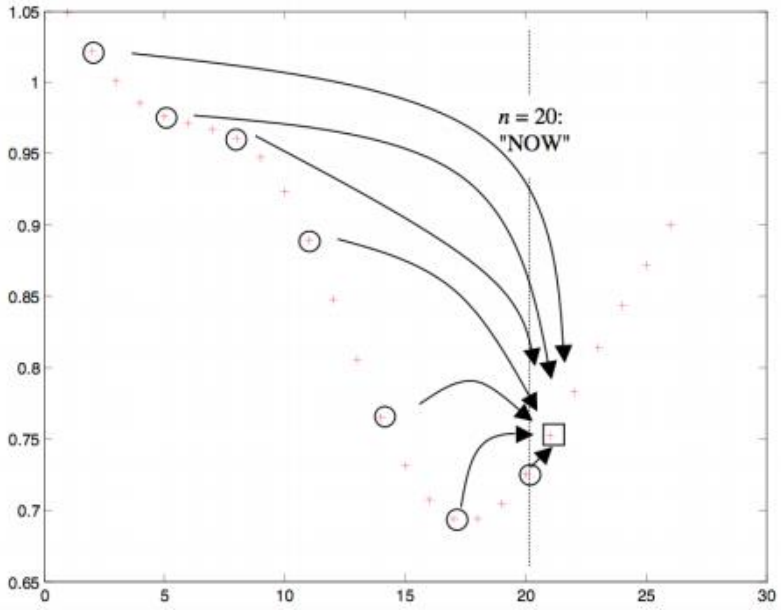


Figure 2.5: One way to specify a mechanism M capable of creating a time series: compute next value (square) as a function (arrows) of some previous values (circles).

Typically, when one uses this approach, one takes a relatively small number of preceding points (order of 3 to 10), which are spaced at regular intervals (in Fig. 2.5., with 2 points in between). Formally, M is an update equation $\hat{f}$ of the following kind:

(2.5) $$x(n+1) = \hat{f}\,(x(n), x(n-d), x(n-2d), \ldots x(n-kd))$$

At this point we make a little digression and describe a curious and extremely useful and relatively recent mathematical insight – the Takens theorem[2].

Consider some dynamical system that is governed by a differential equation of the form

(2.6) $$\dot{\mathbf{x}} = h(\mathbf{x}),$$

where **x** is a vector from a possibly high-dimensional vector space X. Assume further that the dynamics described by this equation is confined to a compact submanifold A of X**.** Basically, this means that the dynamics does not drive the system state to infinite values.This is typically given for real-life physical systems. Furthermore, let the dimension of A be d_A. A typical occurence in very-high-dimensional physical systems is that $\dim(X) >> d_A$. This is due to the fact that often the very many componentes variables of **x** are *coupled* through the dynamics (2.6), a condition which makes them change in time in a coordinated fashion, which effectively shrinks the dimension of the manifold A of X where the action $\mathbf{x}(t)$ happens. This would for instance be expected in brain states or in many biomechanical systems. But still, even with the dimensionality reduction afforded by mutual couplings of variables, d_A still may be quite large in many systems of interest.

In many cases where researchers face complex dynamical systems of the kind (2.6), they face a situation where the following two facts are at odds:

1) The researcher wants to understand the dynamics of the system – for instance, whether it is periodic, chaotic (and if so, how "badly" chaotic), whether it has stable equilibria and so forth – in short, s/he wants to know about the qualitative properties of h.
2) The researcher cannot measure the high-dimensional state vector **x,** nor some lower-dimensional transform of it that covers the dynamics on A. (Imagine, for instance, that **x** contains the zillions of neural activations of a brain state, -- even given that many neurons act in some coordinated way, d_A still would be quite large; or imagine the hundreds of concentrations of reactants in a complex chemical reaction).

In many cases, all that the researcher has in her hands is just the value of one single measurement (or observation) variable $y(\mathbf{x}(t))$ which is a function of the underlying high-dimensional physical state. (For instance, $y(\mathbf{x}(t))$ might be a voltage reading of an extracranial electrode for measuring brain activity, or it might be the concentration of a single, easy-to-measure reactant in a chemical reaction).

[2] **Original paper:** F. Takens (1991), *Detecting strange attractors in turbulence*. In Dynamical Systems and Turbulence (Rand, D.A. and Young, L.-S., eds.), Springer LN Mathematics 898, 366-381. **More recent extensions:** Stark, J., Broomhead, D.S., Davies, M.E., Huke, J., *Takens embedding theorems for forced and stochastic systems*. Nonlinear Analysis, Theory, Methods & Applications 30 (8), 1997, 5303-5314 **Websites:** try "Takens theorem" on Google and you will immediately find pages with a rigorous statement of Takens theorem.

Now the Takens theorem comes to your help! It says that under certain conditions, the dynamics $\mathbf{x}(t)$ of the original state vector is essentially identical to the dynamics of a *delay embedding* of the observation variable, that is, to the dynamics of a vector of the kind

(2.7) $$\mathbf{y}(t) = (y(t), y(t - \tau), y(t - 2\tau), ..., y(t - (k-1)\tau)).$$

Some explanations:

- The phrase, "the dynamics is essentially identical", intuitively means that the dynamics of $\mathbf{x}(t)$ and of $\mathbf{y}(t)$ exhibit the same qualitative properties, for instance same number of equilibrium points, same number of possible oscillation patterns, etc. Technically speaking, the dynamics of $\mathbf{y}(t)$ evolves on some compact manifold $y(A)$, and there is a differentiable embedding of $y(A)$ into A which maps the dynamics of $\mathbf{y}(t)$ on the dynamics of the original states $\mathbf{x}(t)$.
- The dimension k of $\mathbf{y}(t)$ is called the *embedding dimension*. Takens theorem states that if k is chosen to be at least $2\, d_A + 1$, one obtains "essentially identical dynamics".
- Since Takens' pioneering paper, numerous embedding theorems of the same flavour have been found. Specifically, a theorem that goes back to Sauer et al[3] states that if the dynamical system in question is a chaotic attractor of (fractal) dimension d, then the system can be reconstructed generically in delay embedding coordinates of dimension $k > 2d$.
- The value of the delay τ must be chosen with some care to get good results in practical applications.
- For many dynamical systems (even when the system equation is known), it is not possible to compute the embedding dimension analytically. Approximate numerical methods to estimate this dimension from one-dimensional observation sequences $y(t)$ have been developed; they are often somewhat hairy and need a good deal of mathematical understanding to yield reliable estimates.

> Takens theorem can be re-phrased as follows: If you observe a high-dimensional dynamical system over time through a single measurement variable, you can reach full knowledge about the "hidden" high-dimensional dynamics if you take a delay embedding of your observation variable instead of the original state.

Takens theorem has been mis-used in the last decade in sometimes extreme ways, which disregarded the "small print" of the technical prerequisites for its application, and often disregarded the malicious influence of noise in observations (the theorem holds strictly only for noise-free observations). It is a seductive but also dangerous theorem!

And this is the connection of Takens/Sauer's theorem with our update function $\hat{f}$ (Eq. (2.5)) for the Mackey-Glass system. The original MG equation is a so-called *delay differential equation*. Technically, this implies that the dynamics governed by the MG equation uses an *infinite-dimensional state space X*! However, after the dynamics converges on the final MG attractor (that is what we see in Fig. 2.4), the (fractal) dimension of the attractor is small (about ~3 for a delay of $\tau = 30$). Because in Eq. (2.4) we took a discrete-time approximation to the original continuous-time MG equation, the dimension of X in this discrete-time approximate system is not infinite but equals the delay, that is, it is $\dim(X) = 17$ or 30. If we

[3] T. Sauer, J. Yorke, and M. Casdagli, Embedology, J. Stat. Phys. 65, 579 (1991). Preprint at http://math.gmu.edu/~tsauer/pre/embedology.pdf

consider the dynamics only on the final attractor, it again is a fractal of dimension ~3 for τ = 30.

Although Eq. (2.4) superficially looks like a one-dimensional difference equation of the type $x(n+1) = h(x(n))$, it actually specifies a high-dimensional system (of dim 17 or 30) due to the delay that appears in it. The variable x of Eq. (2.4) should actually be seen not as a state variable, but as an observation variable of the kind $y(\mathbf{x}(t))$ as described above. For the MG attractor, embedding dimensions of k = 7 or 8 are often used.

The reason why it works out to implement M as a function $\hat{f}$ that takes some previous values to compute the next, as in Figure 2.5., now becomes clear: the previous values provide a delay embedding state that is "essentially equivalent" to the true (unknown) ~3-dimensional attractor state. So it is no coincidence that in Fig. 2.5, the number of previous points is 7: it is actually the minimal number required (for τ = 30).

Tricky question: Equation (2.4) uses only 2 previous points to compute the next. So why is the embedding dimension not just 2? Assuming τ = 30 and δ = 1, the form of Eq. (2.4) is $x(n+1) = g(x(n), x(n-30))$. So why shouldn't one be able to learn some $\hat{g}\,(x(n), x(n-30))$, that is, why shouldn't an embedding dimension of 2 be sufficient? Answer: yes, it is possible to learn $\hat{g}\,(x(n), x(n-30))$ from training data. BUT one has to divine the correct delay value of 30 first in order to make this work! Eq. (2.4) has no equivalent version that uses a different delay value. By contrast, using the 7-dimensional delay embedding suggested by Takens' theorem is in principle insensitive to the choice of the embedding delay duration (although some delays work more efficiently than others). Because there is no known way of guessing the correct original delay from training data, one has to apply Takens' theorem.

2.5 Analytical vs. blackbox modelling

Things are relatively simple if one knows, from insight or analysis, that the given initial time series has been generated by a mathematical object of a particular type. For instance, in predicting the trajectory of celestial objects the type of governing equations is known since Newton's days. If one knows the form of the equation, the learning task boils down to fix the values of the few "free parameters" in the equation (for instance, the comet's mass). Or in a chess game prediction, one might settle for one's favourite game-theoretic formalism to write up a template for M; this would then have to be filled with the current opponent's "personality parameters", which would have to be induced from the known training data. In many engineering and chemical processing tasks – and in weather prediction – , a well-guided guess about the form of the governing equations can be made. In such cases where the basic form of M is known, one speaks of *analytical models* (or *physical* models). Learning boils down to fitting a small number of parameters to the individual case at hand.

In many cases, however, analytical models are ruled out. For instance, it is close to impossible to invent "suitable" equation templates for stock market developments, or for brain or cardial recordings (medical and biological time series in general), -- generally, biological, economical, social systems are hard to press into an equation. In other cases, analytical models might be obtainable, but would be too complex for any practical dealings. This is a typical situation when one is confronted with complete, complex technical systems, like

power plants, combustion engines, manufacturing lines and the like. Besides the sheer complexity of such systems, analytical models would present difficulties because they would have to *integrate* several *types* of formalisms: For instance, modeling a combustion engine would require an amalgamation of chemical, thermodynamic, and mechanical formalisms.

If analytical models are inaccessible, the model *M* has to be established in some "generic" formalism that is capable of *mimicking* basically *any* kind of dynamical system, without paying respect to the underlying physical mechanism. The only requirement about *M* here is that *M* can produce the same input-output time series as the original generator. This is the approach of *blackbox modeling*: the original system is seen as an intransparent ("black") device whose internal workings cannot be understood; only the externally observable measurements are available, and only they are "modeled" by *M*.

The best known type of blackbox model *M* is probably neural networks, and we will learn a lot about them. However, there are other important types of blackbox models, which I will at least name here: support vector machines (very fashionable these days), mixtures of Gaussians, Fourier decompositions, Taylor expansions, Volterra expansions, Markov chains and hidden Markov models. All of these models can accomodate to a wide range of systems, and all of them typically require a very large number of free parameters to be fixed through the learning process.

The general picture of a blackbox modeling task is shown in Fig. 2.6.

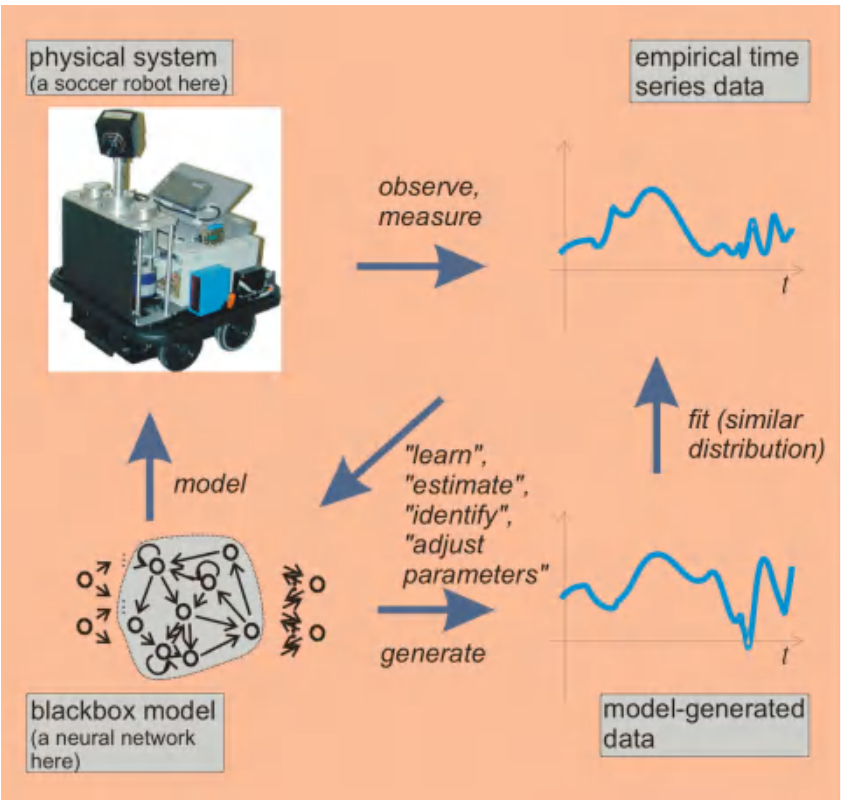


Figure 2.6: General scheme of blackbox modeling (here, for time series).

2.6 The bias-variance dilemma

(Adapted from Bishop Sec. 1.5)

We have encountered two elementary learning tasks,

- classification tasks with training data $(\mathbf{x}_i, y_i)$, where $y_i \in \{1, ..., n\}$ are taken from a finite set of class labels and a classification function $\hat{f}$ (x) is sought,
- regression tasks with training data $(\mathbf{x}_i, \mathbf{y}_i)$, where $\mathbf{y}_i$ is a real-valued vector and a regression function $\hat{f}$ (**x**) is sought.

Both types of tasks have a similar basic structure, and both types of tasks have to face the same fundamental problem of machine learning: the bias-variance dilemma. We will introduce it with the simple regression task of polynomial curve fitting.

Let's consider a one-dimensional input, one-dimensional output regression task of the kind where the training data are of form (x_i, y_i). Assume that there is some systematic relationship $y = f(x)$ that we want to recover from the training data. We consider a simple artificial case where the x_i range in [0, 1] and the to-be-discovered relationship is $y = \sin(2 \pi x)$. The training data, however, contain a noise component, that is, $y_i = \sin(2 \pi x_i) + \nu_i$, where ν_i is drawn from a normal distribution with zero mean and standard deviation σ. Fig. 2.7 shows a sample (x_i, y_i), where eleven x_i are chosen equidistantly.

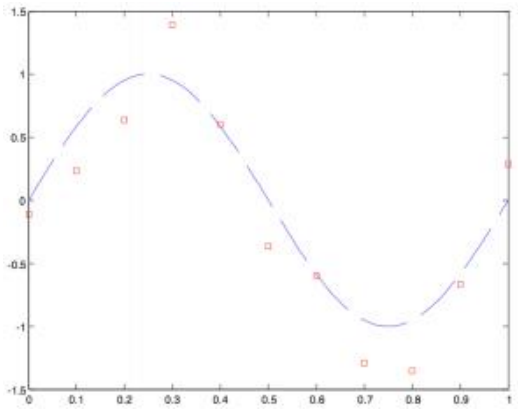


Fig. 2.7: An example of training data (red squares) obtained from a noisy observation of an underlying "correct" function $\sin(2 \pi x)$ (dashed blue line).

We now want to solve the task of learning a good approximation $\hat{f}$ for f from the training data (x_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 M-th order polynomial

$$p(x) = w_0 + w_1 x + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j . \tag{2.8}$$

We want to approximate the function given to us via the training data (x_i, y_i) by a polynomial, that is, we want to find ("learn") a polynomial $p(x)$ such that $p(x_i) \approx y_i$. More precisely, we want to minimize the *mean square error* on the training data

$$\text{MSE}_{\text{train}} = \frac{1}{N}\sum_{i=1}^{N}(p(x_i) - y_i)^2 \tag{2.9}$$

by a good choice of $p(x)$ (here N is the number of training samples, in our example $N = 11$). If we assume that the order M of the polynomial is given, minimizing (2.9) boils down to finding polynomial coefficients w_j such that

$$\text{MSE}_{\text{train}} = \frac{1}{N}\sum_{i=1}^{N}(\sum_{j=0}^{M} w_j x^j - y_i)^2 \tag{2.10}$$

is minimized. 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 training data (x_i, y_i) and polynomial order M, find the polynomial coefficients that minimize (2.10). Fig. 2.8 shows the polynomials found in this way for $M = 1, 3, 10$.

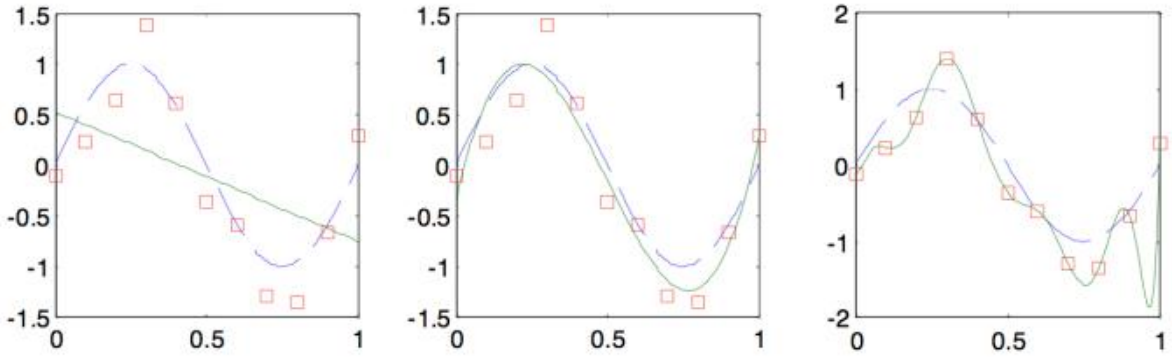


Fig. 2.8: Fitting polynomials (green lines) for polynomial orders 1, 3, 10 (from left to right)

If we compute the MSE's (2.10) for the three orders 1, 3, 10, we get $\text{MSE}_{\text{train}} = 0.4852$, 0.0703, 0.0000. Some observations:

- If we increase the order M, we get increasingly better $\text{MSE}_{\text{train}}$.
- For $M = 1$, we get a linear polynomial, which apparently does not represent our original **sine** function well.
- For $M = 3$, we get a polynomial that hits our target **sine** apparently quite well.
- For $M = 10$, we get a polynomial that perfectly matches the training data, but apparently misses the target **sine** function.

We have stumbled over a phenomenon that will haunt us for the rest of this lecture: we apparently do NOT get the optimal estimate $\hat{f} = p(x)$ for f if we try to optimally fit the training data. This is our first encounter with the *bias-variance dilemma*, one of the fiercest enemies of machines in machine learning. We take a closer look at it.

First we discuss what actually makes the $M = 3$ solution "look better" than the other two. If we knew the correct target function $f(x) = \sin(2\pi x)$, it would be clear what "better" means: a

solution $\hat{f} = p(x)$ is better if it is closer to the target function. However, in real-life applications the correct target function is in general not accessible. In real life, what we desire of a "good" solution to a learning problem derived from training data (x_i, y_i) is that it *generalizes* well to new *test data* not contained in the training data set. Formally, we want the following: if (x'_j, y'_j) are new data coming in from noisy observations $y'_j = \sin(2 \pi x'_j) + \nu_j$, then the expected discrepancy (in the mean square error sense) between the actual new observations y'_j and the model predictions $p(x'_j)$ is minimal, that is,

(2.11) $$\mathrm{MSE}_{\mathrm{test}} = E((y' - p(x'))^2)$$

becomes minimal. Because the empirical observations y' are scattered around the correct target function f, (2.11) essentially amounts to our original intuition that a "good" solution is close to the target function.

We can estimate $\mathrm{MSE}_{\mathrm{test}}$ by creating a large set of test data (x'_j, y'_j) (where $j = 1, ..., K$) and approximate (2.11) through

(2.12) $$\hat{\mathrm{MSE}}_{\mathrm{test}} = \frac{1}{K} \sum_{j=1}^{K} (y_j' - p(x_j'))^2 .$$

Using 101 equally spaced x'_j, computing (2.12) for learnt polynomials of orders 1 through 10 and comparing it with the training MSEs (2.10), we get the following picture:

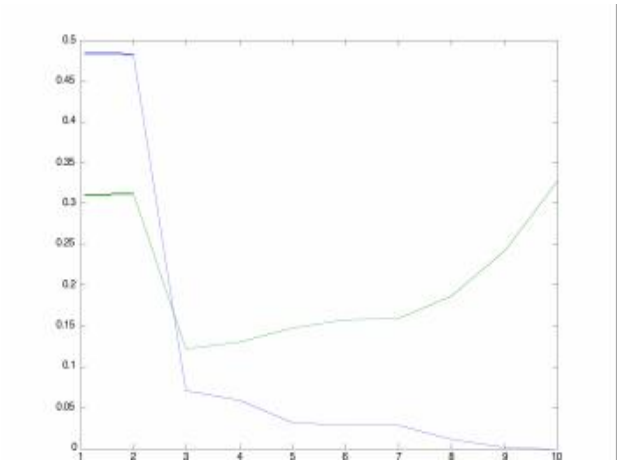


Fig. 2.9: Training (blue solid line) vs. testing errors (green dashed line) for polynomial fits of orders 1 through 10.

We find in Fig. 2.9 a pattern that is highly representative for machine learning tasks: if we train models of increasing complexity (here: polynomials of increasing order), the training error goes down (it here even reaches zero because a 10th order polynomial can fit 11 training points perfectly). On the other hand, the test error first goes down but then up again. When it goes up again, we say that the training data are *overfitted* by the learnt function $\hat{f}$. Intuitively, overfitting means that the training procedure manages to fit even the noise component in the training data – and in doing so, generalizes poorly to new test data which have other noise components.

Abstracting from our simple polynomial fit example, the following situation is common in machine learning:

- When trying to fit the training data with some model (here: a polynomial), we can choose from models of different complexity (here: order of polynomial; another example would be to choose between neural networks of different size).
- More complex models have a larger number of free parameters that have to be estimated through the learning process (here: the polynomial weights). The more complex a model, the more free parameters can be utilized to fit training data, and the smaller the MSE_{train}.
- The test error, however, has a minimum at some intermediate level of model complexity.
- Much of the art of machine learning lies in the choice of an appropriate model complexity. This is tricky because usually one does not have, at the time of learning, independent test data which one could use to find the appropriate model complexity. All one has is the training data. We will learn about several strategies to cope with this situation.

The conflict between choosing models of low complexity (which might under-exploit the information in the training data) and choosing models of high complexity (which might generalize poorly due to overfitting) is known as the *bias-variance dilemma*. This terminology will become clearer when we treat Bayesian model estimation techniques (coming soon), but intuitively it means the following. If we choose a low-complexity model, what we actually do is to impose restrictions on the possible solutions – for instance, if we choose an order-1 polynomial, we restrict ourselves to linear solutions. But imposing restrictions is just another way of stating that we enforce certain preconceptions on the possible solutions – we are "biased" about the possible solutions. In contrast, high-complexity models hook up into the model noise components – as a consequence, if you would repeatedly estimate such models from respectively fresh training data, the resulting models would exhibit a high inter-model variance (because each models the fresh noise in the fresh data).

The bias-variance dilemma, or synonymously the problem of overfitting, arises in classification tasks, too. Consider Fig. 2.10. It shows a training data set where two-dimensional vectors $\mathbf{x}_i = (x_1, x_2)_i$ come in two classes (black cross vs. red circle). One way of specifying a classification function $\hat{f}$ is to provide a *decision boundary* (solid black line). Again, if we allow only low-complexity decision boundaries (such as the linear boundary in the left panel), we get a high training error (number of misclassifications is 4). Conversely, with a high-complexity decision boundary (right panel) we get a low training error (zero misclassifications in Fig. 2.10). We might suspect that the best model should look something like the one in the middle panel (intermediate complexity, number of training errors is 2). But in the absence of fresh test data, we actually can't assess which one of the models is best! it might be the case that the underlying correct decision boundary is in effect linear – then the low-complexity model from the left panel would be appropriate, and the high number of misclassifications is due to a high level of observation noise. But it might just as well be the case the high-complexity model is best – namely, if we have low-noise observations. Unfortunately, it is not easy to guess from the training data what the noise level is.

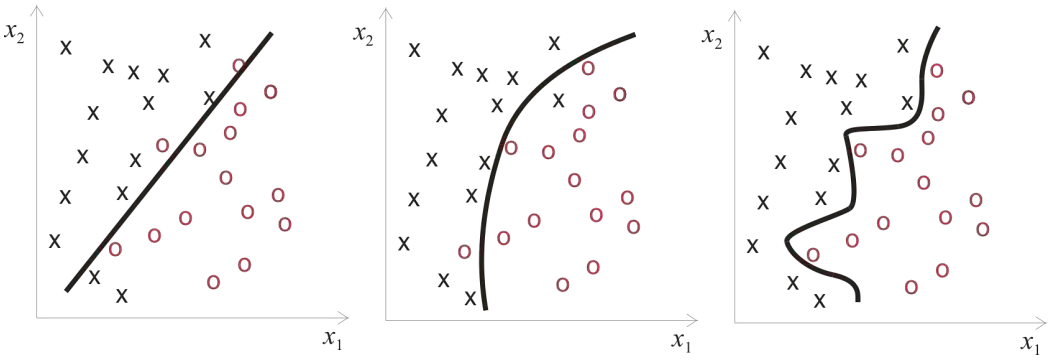


Fig. 2.10: The bias-variance dilemma in a classification task.

It might seem that these are academic considerations, but in fact, our simple examples let the situation appear simpler than it is. If we have high-dimensional training data combined a small number of training samples – an unfortunate condition that is all too often found in practice – the bias-variance dilemma becomes a dominating source of trouble.

There are many approaches to deal with the bias-variance dilemma, some of them very sophisticated. However, in daily practice when quick (but possibly "dirty") solutions are demanded, two simple approaches are often used: regularizers and cross-validation schemes. We will briefly introduce them.

2.7 Using regularization terms to control overfitting

We have seen that minimizing the training MSE alone leads into overfitting. The idea of regularizers is to minimize an additive combination of the training MSE plus a "penalty" term that grows when models exhibit some undesired property. This obviates the need to search explicitly for a good model complexity (e.g., an appropriate polynomial order). For instance, in the polynomial fit task one might consider only 10th order polynomials but punish the "oscillations" seen in the right panel of Fig. 2.8, that is, favour such 10th order polynomials that exhibit only weak oscillations. The degree of "oscillativity" can be measured, for instance, by the integral over the (square of the) second derivative of the polynomial,

$$\Omega = \int_0^1 \left(\frac{d^2 p(x)}{dx^2} \right)^2 dx \tag{2.13}$$

Using this measure of oscillation, high-oscillation solutions are suppressed by adding a penalty term to the error function that is minimized. That is, instead of minimizing the $\mathrm{MSE}_{\mathrm{train}}$ given in (2.10) we seek a polynomial that minimizes

$$\mathrm{LF}_{\mathrm{train}} = \frac{1}{N} \sum_{i=1}^{N} (p(x_i) - y_i)^2 + c\Omega, \tag{2.14}$$

where c is a constant suitably chosen. $c\Omega$ is called a *penalty term* or *regularizer*. The larger c, the stronger we favour low-oscillation solutions. The quantity (2.14) that we want to minimize is called a *loss function* – this is the gerneric term in such optimization tasks (the MSE is just a particular loss function). Implicitly, the model complexity is kept in limits – not by

restricting the polynomial order but by restricting the search space within 10th order polynomials to "smooth" ones.

One difficulty with using regularization is that finding a solution that minimizes LF_{train} might be more difficult to compute. Another, more difficult difficulty is that it is not obvious how to define a "good" regularizer, or how to weigh it by a regularization weight c. A regularizer implements just another kind of preconception about what makes a solution "good" and thus is just another case of introducing bias – the larger the regularization coefficient c, the stronger the influence of our bias.

2.8 Using cross-validation to overcome overfitting

We saw that independent test data can be used to assess the degree of overfitting. The idea of cross-validation is to split the training data set X artificially into two subsets $X_1 = (x_i, y_i)$ and $X_2 = (x'_j, y'_j)$. The first subset is used to learn several models of different complexity. Their generalization performance is then tested on the second subset (x'_j, y'_j). The model complexity that gives best generalization is then chosen, and a model of this complexity is re-trained on the original, complete training data set X.

In this simplistic version, cross-validation might suffer from ill fortune in splitting the training data – the data sets X_1 and X_2 used for training and testing might by chance exhibit peculiarities which systematically favour models that are less complex or more complex than appropriate. Specifically, if the noise components in X_1 and X_2 happen to be positively correlated, models of higher complexity than appropriate are favoured. One common way to sidestep this possibility is to split X into many subsets X_i of equal size ($i = 1, ..., k$). For each model complexity, k learning trials are carried out, the first using $X_1 \cup X_2 \cup ... \cup X_{k-1}$ for training and X_k for testing, the second trial using all X_i except X_{k-1} for training and X_{k-1} for testing, etc. Then, the average testing error across all of these k trials is taken as an indicator for the generalization performance of this particular complexity. After this has been carried out for all complexities that one wants to consider, the complexity that gave the best average generalization performance is chosen and a model of this complexity is finally learnt on the complete training data set. In the extreme, all X_i contain just a single data point – we then speak of *leave-one-out cross-validation*. If the X_i each contain n samples, we speak of *leave-n-out cross-validation.* Another terminology is to speak of *k-fold cross validation*, where k is the number of "batches".

A clever way to perform cross-validation is to start with low-complexity models, assess their generalization capabilities through cross-validation, and increase the complexity until the generalization performance starts to go down. The complexity level reached at that point is likely to be about right.

The advantage of cross-validation techniques is their conceptual simplicity, the disadvantage is the high computational cost resulting from many repeated training trials. If the computational cost of a single learning trial is already significant (as it unfortunately often is – machine learning algorithms are often *very* expensive), elaborate cross-validation may not be a viable option.

2.9 Bayes' theorem and optimal decision boundaries

Let's return to the digits classification task. Assume you are getting a new data vector **x** representing a 15 by 16 pixel image of a "zero" or a "one" – and this pixel image is of very bad quality (as the last one in the second row of Fig. 2.1), so you really can't tell very well whether it's a "0" or a "1". The feature value of $F_3(\mathbf{x})$ is 3000, which in Fig. 2.2 puts this **x** into the sixth bin where there are about twice as many "ones" as "zeros".

But assume you know that **x** originated from a financial report, and you know that in the world of finance, the digit "0" occurs 4 times more frequently than the digit "1" (is there some empirical truth to this?). What should you decide **x** to represent, a zero or a one? By which probability?

Here we face the task to combine two sources of information. The first source is your prior knowledge about the base rates of zeros vs. ones in financial databases, which assigns to **x** being a "1" a *prior* probability of 0.2. The second source of information comes from analysing your sample **x** – which apparently favours the hypothesis that **x** is a "one" by a factor of two.

The proper way of combining these two sources of information makes use of *Bayes' theorem*, a fundamental (and simple) theorem of statistics. To explain this, we redraw Fig. 2.2. in a slightly different way.

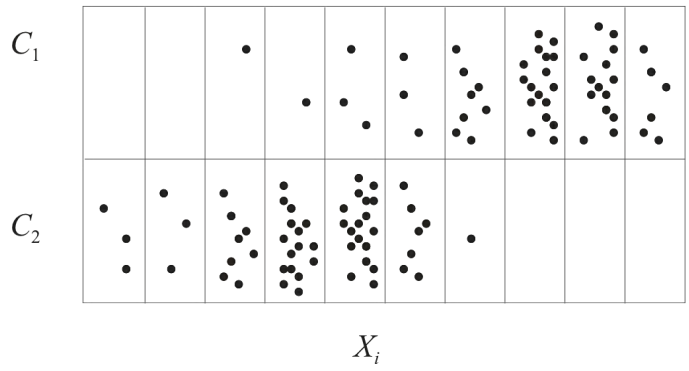


Fig. 2.11: Figure 2.2 redrawn (schematic): Binning the training data in bins distinguished by class (the two rows) and feature values (the columns). Horizontal axis: the ten subsegments ("bins") of the $F_3(\mathbf{x})$ feature. Vertical: the two classes C_1 (containing the "zero" samples) and C_2 (containing the "one" samples). In the plotted example, an equal number of instances from each class was given as training data.

Some terminology:

- The *joint* probability $P(C_k, X_i)$ is the probability that some sample **x** belongs to class C_k **and** has feature value X_i. This corresponds to the probability that some sample falls into a particular cell of Fig. 2.11.
- The *conditional* (or *class-conditional*) probability $P(X_i \mid C_k)$ specifies the probability that some observation has feature value X_i **given** that it belongs to class C_k. In Fig. 2.11, this corresponds to the fraction of all samples in row C_k which fall in column X_i.

- The *total* (or *prior*) probability $P(C_k)$ is the probability that some sample falls into row C_k. This corresponds to the fraction of all samples that fall into row C_k. In our financial database world, we would have $P(C_1) = 0.8$ and $P(C_2) = 0.2$.
- The probability $P(X_i)$ is the probability that some sample **x** has feature value X_i. This probability has no special name.
- The *posterior* probability $P(C_k \mid X_i)$ is the probability that a sample **x** belongs to class C_k given that feature value X_i was observed. In Fig. 2.11, it is the fraction of all samples in column X_i which fall in row C_k, *provided that* the training sample correctly reflects the prior probabilities – which it does not do here (because we assumed that in the world of finance there are four times as many 0's as 1's – so the training data plotted in this figure over-represent the 1's).

Obviously it is the posterior probability that we want to know when confronted with a sample **x** that we want to classify. We get at it through the Bayes theorem, which we will now derive.

Using the generally valid formula for joint probabilities, $P(A, B) = P(A \mid B)\, P(B)$, we can write the joint probability in two ways:

$$P(C_k, X_i) = P(X_i \mid C_k)\, P(C_k) = P(C_k \mid X_i)\, P(X_i). \tag{2.15}$$

Combining the two expressions on the rhs., we get

$$P(C_k \mid X_i) = \frac{P(X_i \mid C_k) P(C_k)}{P(X_i)}, \tag{2.16}$$

which is (one form of) the theorem of Bayes. It allows us to compute the posteriori probability from the class-conditional probability $P(X_i \mid C_k)$ and the prior probability $P(C_k)$. The denominator $P(X_i)$ plays the role of a normalization term, and it can itself be expressed in terms of class-conditional probabilities and prior probabilities by

$$P(X_i) = \sum_k P(X_i \mid C_k) P(C_k)\,. \tag{2.17}$$

Why is Bayes' theorem so important?

- In many practical problems, the prior probabilities are known (or can be reasonably guessed), and the class-conditional probabilities can be estimated from observations / experiments. Consider for example a situation where we want to design a medical screening test for distinguishing between normal (= class C_1) and tumor (= class C_2) tissue, by exploiting the information we can get from some tissue diagnostic feature F with values X_i. From epidemiological statistics we know that tumor tissue probes occur very rarely, say with 0.01 %. If we would try to assess our statistics for $P(C_k \mid X_i)$ by evaluation data counts of the kind shown in Fig. 2.11, we would need, say, data from about 1000 tumor patients to get good estimates of the probabilities in the C_2 (= tumor) row. But this would mean that the C_1 row would contain samples from 1,000,000 patients! We would instead want to analyse probes from only another 1000 healthy patients in order to calibrate a decision function based on feature F. Bayes' theorem tells us how.

- More generally speaking, Bayes' theorem shows us how to do valid *abductive* reasoning. An abductive argument is of the following type: given knowledge about the consequences of some causal effect, what is the probability of a certain cause? This is the standard situation of diagnostic reasoning, for instance in medicine or machine fault monitoring. The consequences X_i are observable – for instance, the outcomes of a medical diagnostic probe. The cause C_k is hidden – for instance, a disease. Bayes tells us how to arrive at valid diagnostic conclusions. In fact, naive humans are prone to make gross errors in this respect. For instance, if a tumor test returns value X_2, and the probability of getting this value in case of a tumor is 90%, this is not necessarily bad news for the patient. Assume, for instance (not unrealistically) that the prior probability of tumors C_2 in the population (of patients going to doctors for a broad-band health check) is 1%, and that the class-conditional probability of the tumor test to yield X_2 in non-tumor patients is 5%. Then Bayes teaches us that $P(\textit{tumor} \mid X_2) = 0.9 * 0.01 / P(X_2)$ and $P(\textit{no tumor} \mid X_2) = 0.05 * 0.99 / P(X_2)$, so the ratio $P(\textit{tumor} \mid X_2) / P(\textit{no tumor} \mid X_2)$ is only $(0.9 * 0.01) / (0.05 * 0.99) \approx 0.18$. We also see here that the denominator X_i need not be computed if we want to obtain a hypothesis vector $(P(C_1 \mid X_i), P(C_2 \mid X_i))$. And we see why in fact it is part of the professional training of doctors to learn about Bayes' theorem (there is even a complete research field on *medical decision making*).

Let's briefly finish our digit decision problem. The frequencies in Fig. 2.2 and Fig. 2.11 actually were obtained from an equal number of class C_1 and class C_2 samples (as in the hypothetical medical tumor screening example), so these figures do not represent the unequal distribution of "zeros" vs. "ones" in our financial database. Using Bayes' theorem, and the probability estimates from those figures, we would classify our pattern **x** as a "zero" vs. a "one" with a probability ratio of $P(C_1 \mid X_6) / P(C_2 \mid X_6) = 1 * 0.8 / 2 * 0.2 = 2$.

So far we have considered raw high-dimensional observation vectors **x**, from which we distilled low-dimensional feature vectors $F(\mathbf{x})$ taking values in some feature space X, which we partitioned into discrete bins X_i., that is, we considered discrete-valued feature vectors. From now on, we will take the feature extraction for granted (it is done in virtually every machine learning task) and denote by **x** the feature vector directly. Furthermore we will consider the more general (and more convenient) case where we do not discretize the (feature) observation space X into bins but consider continuous-valued observation vectors **x**. In order to lift our treatment of Bayes' theorem to the continuous-valued case, we quickly recapitulate how to work with continuous probability densities.

The probability that an observation **x** falls into some region $\mathcal{R} \subseteq X$ is given by

$$P(\mathbf{x} \in \mathcal{R}) = \int_{\mathcal{R}} p(\mathbf{x}) d\mathbf{x}\,, \tag{2.18}$$

where $p(\mathbf{x})$ is the *probability density function* of the distribution of **x**. Observe that $\int_X p(\mathbf{x}) d\mathbf{x} = 1$. If $Q: X \to \mathbb{R}$ is a numerical function of observations, the *expectation* of Q is given by

$$E[Q] = \int_X Q(\mathbf{x})\, p(\mathbf{x}) d\mathbf{x} \tag{2.19}$$

which can be approximated by

(2.20) $$E[Q] \approx \frac{1}{N} \sum_{i=1}^{N} Q(\mathbf{x}_i)$$

if we have N samples $\mathbf{x}_i$ randomly drawn from the distribution of $\mathbf{x}$. Note that we use smallcap letter p to denote densities and capital letter P to denote probabilities. In the continuous domain, Bayes' theorem becomes

(2.21) $$P(C_k \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid C_k) P(C_k)}{p(\mathbf{x})},$$

where the unconditional density $p(\mathbf{x})$ is given by

(2.22) $$p(\mathbf{x}) = \sum_k p(\mathbf{x} \mid C_k) P(C_k),$$

which ensures that the posterior probabilities sum to unity,

(2.23) $$\sum_k P(C_k \mid \mathbf{x}) = 1.$$

In a decision problem, we are given an observation $\mathbf{x}$ and want to find the class C_k such that the probability of misclassification becomes minimal. Intuitively, it is clear that the proper class choice is to select that class C_k that makes

(2.24) $$p(C_k \mid \mathbf{x}) > p(C_j \mid \mathbf{x}) \quad \text{for all } j \neq k.$$

We will soon justify (2.24) but simply take it as granted for the time being. If we fill (2.21) into (2.24), we find that the comparison ">" is independent of the denominator $p(\mathbf{x})$, so for purposes of comparing posterior probabilities we can drop it and use

(2.25) $$p(\mathbf{x} \mid C_k)\, P(C_k) > p(\mathbf{x} \mid C_j)\, P(C_j) \quad \text{for all } j \neq k$$

instead of (2.24). The borders in X where $p(\mathbf{x} \mid C_k)\, P(C_k) = p(\mathbf{x} \mid C_l)\, P(C_l)$ for some $l \neq k$ and $p(\mathbf{x} \mid C_k)\, P(C_k) = p(\mathbf{x} \mid C_l)\, P(C_l) > p(\mathbf{x} \mid C_j)\, P(C_j)$ for all $j \neq k, l$, are called the *decision boundaries* of the decision problem. Our class decision would change when the observation $\mathbf{x}$ passes over the some decision boundary. The decision boundaries partition X into regions $\mathcal{R}_i$, where within each region $\mathcal{R}_i$ the value of $p(\mathbf{x} \mid C_i)\, P(C_i)$ is maximal among all $p(\mathbf{x} \mid C_j)\, P(C_j)$. A region $\mathcal{R}_i$ need not be connected, convex, or in any other way easy to describe.

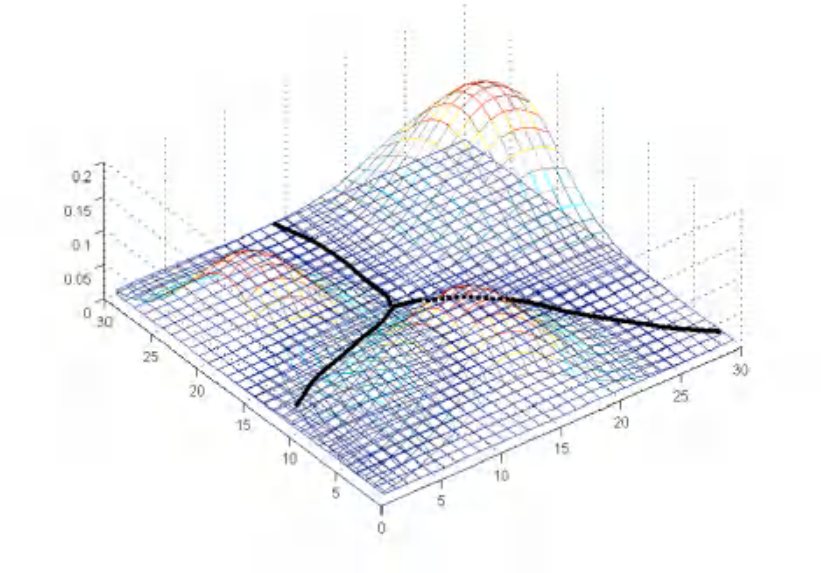


Fig. 2.12. Decision boundaries separating three classes. The "hills" are pdfs $p(\mathbf{x} \mid C_j)\, P(C_j)$ for some two-dimensional observations $\mathbf{x}$.

The decision criteria (2.24) (or (2.25)) certainly look plausible – but why are they in fact the best one can do? What we want is to minimize the probability P(error) of making a misclassification. For simplicity we restrict ourselves to the case of only two classes C_1 and C_2. Assume that we have divided X into two regions $\mathcal{R}_1$ and $\mathcal{R}_2$ (not necessarily the ones given by criteria (2.24) or (2.25)), such that we decide for class C_i whenever $\mathbf{x} \in \mathcal{R}_i$ (i = 1, 2). We make a classification error when we assign a new sample $\mathbf{x}$ to class C_1 when it rightfully belongs to C_2, and vice versa. Then P(error) is the probability of making an error of either kind:

$$
\begin{aligned}
(2.26) \qquad P(\text{error}) &= P(\mathbf{x} \in \mathcal{R}_1, \mathbf{x} \in C_2) + P(\mathbf{x} \in \mathcal{R}_2, \mathbf{x} \in C_1) \\
&= P(\mathbf{x} \in \mathcal{R}_1 \mid C_2)\, P(C_2) + P(\mathbf{x} \in \mathcal{R}_2 \mid C_1)\, P(C_1) \\
&= \int_{\mathcal{R}_1} p(\mathbf{x} \mid C_2)\, P(C_2)\, d\mathbf{x} + \int_{\mathcal{R}_2} p(\mathbf{x} \mid C_1)\, P(C_1)\, d\mathbf{x}
\end{aligned}
$$

Thus, if $p(\mathbf{x} \mid C_2)\, P(C_2) > p(\mathbf{x} \mid C_1)\, P(C_1)$ for a given $\mathbf{x}$, we should choose the regions $\mathcal{R}_i$ such that $\mathbf{x} \in \mathcal{R}_2$ because that minimizes the integrals' contribution to P(error). We recognize this as the criteria (2.24) or (2.25).

So, we base our decision rightfully on the relative magnitude of the functions $p(\mathbf{x} \mid C_i)\, P(C_i)$. Apparently our decision would not change if we base it on the relative magnitude of some monotonic function g of $p(\mathbf{x} \mid C_i)\, P(C_i)$, that is, on the relative magnitude of some *discriminant functions*

$$
(2.27) \qquad y_i(\mathbf{x}) = g(p(\mathbf{x} \mid C_i)\, P(C_i)).
$$

A much-used choice for g is the logarithm. We would neither change the decision regions $\mathcal{R}_i$ nor the outcome of our decision criteria (2.24) or (2.25), if we replaced (2.25) by choosing class k whenever

(2.28) $\ln p(\mathbf{x} \mid C_k) + \ln P(C_k) > \ln p(\mathbf{x} \mid C_j) + \ln P(C_j)$ for all $j \neq k$.

In the case of two-class decision problems, a slightly different version of discriminant functions is often used. Instead of using two discriminant functions $y_1(\mathbf{x})$ and $y_2(\mathbf{x})$, one introduces a single function $y(\mathbf{x}) = y_1(\mathbf{x}) - y_2(\mathbf{x})$ and assign $\mathbf{x}$ to class C_1 iff $y(\mathbf{x}) > 0$. From what we have just seen it follows that we can use various forms of $y(\mathbf{x})$ including

(2.29) $y(\mathbf{x}) = p(C_1) - p(\mathbf{x} \mid C_2)$
$[= (p(\mathbf{x} \mid C_1)\, P(C_1) - p(\mathbf{x} \mid C_2)\, P(C_2)) / (p(\mathbf{x} \mid C_1) + p(\mathbf{x} \mid C_2)),$
use that $P(C_1) + P(C_2) = 1$!]

and

(2.30) $$y(\mathbf{x}) = \ln \frac{p(\mathbf{x} \mid C_1)}{p(\mathbf{x} \mid C_2)} + \ln \frac{P(C_1)}{P(C_2)}$$
$[= \ln (p(\mathbf{x} \mid C_1)\, P(C_1)) - \ln(p(\mathbf{x} \mid C_2)\, P(C_2))$!]

Discriminant functions are sometimes easier to use than the original probabilities from (2.24) or (2.25), because it is often possible to determine suitable discriminant functions without going through the intermediate step of probability density estimation. Furthermore, one often works with log's of probabilities directly, without ever computing probabilities, because this helps to avoid numerical underflow problems with very small probabilities.

3 A refresher on essential probability theory and statistics – classical and Bayesian

The aim of this section is to make you acquainted with a number of notions from probability theory and statistics that constitute a required background for this course.

3.1 A handful of basic concepts

It is possible to become a reasonably good modelling practician without really knowing what probabilities are – you can use equations like the Bayes formula or decision criteria "mechanically" – but it is not possible to become a really creative in this field without this knowledge. Therefore we devote some time to a more rigorous (re-) introduction of the basic concepts of probability theory and statistics.

A fine webpage is http://www.probability.net – you can find there an online tutorial and a dictionary of all important definitions.

We will in some detail consider a simple standard task, namely, estimating the probabilities of symbols from a sample. If the sample is small, this task becomes surprisingly subtle. A typical situation in bioinformatics is the following. Proteins are sequences of amino acids, of which there are 20 different that occur in proteins. They are standardly tagged by 20 capital symbols, as A, G, H, ..., all intimately familiar to biologists. Proteins come in families. Some protein in one species has typically close relatives in other species. Related proteins differ in detail but generally can be *aligned*, that is, corresponding sites in the amino acid string can be detected,

put side by side, and compared. For instance, consider the following short section of an alignment of 7 amino acids from one family[4]:

```
...GHGK...
...AHGK...
...KHGV...
...THAN...
...WHAE...
...AHAG...
...ALGA...
```

Fig. 3.1: Seven aligned protein sniplets from one protein family (here: of globulines).

A basic task that bioinformatics faces is: for each column in such a *family alignment*, estimate the probability distribution of the amino acids in this column, as you would expect it to be in the total population of all proteins belonging to this family. This task is, on the one hand, important: because such distribution estimates are the basis for deciding whether some newly found protein belongs into the family. On the other hand, this task is apparently rendered difficult by the fact that the sample of aligned proteins used to estimate this distribution is typically quite small – here we have only 7 individuals in the sample. As can be seen in Fig. 3.1, some columns have widely varying entries (e.g. the last column `K K V N E G A`). In contrast, the family of related proteins is huge: in every animal species one would expect at least one family member; typically many more. So how can one derive "good" estimates for the distribution of symbols in a large population, from very small samples?

For this task of estimating a probability distribution (and all other such tasks) there are two major types of approaches:

1. The "classical", "frequentist", "objective", where probabilities are defined in terms of limits of frequency counts. This is the kind of probability theory and statistics that has dominated mathematics and statistics in the last centuries and it is the approach taught in most university courses on statistics. In this view, a probability of a symbol in a population is defined to be its frequency *in the limit of infinite population size*. The probability *P*(A) of "amino acid A occurs in the population" is objectively defined – at least in ideal theory (assuming the population is infinitely large). The classical approach gives a clear picture of things when one has access to large samples but has difficulties in dealing with small samples.
2. The "Bayesian", "subjective" approach where a probability is defined as a subjective degree in belief that a newly observed symbol would be of some particular kind. Here the probability *P*(A) need not be defined objectively. But this does not mean Bayesian statistics is not a rigorous mathematical field. Bayesian theory is not concernd with what probability *is* but with how rational people should *correctly reason about* probabilities. Bayesian statisticians ask (and answer) questions like: If someone believes some things about some probabilities in some population, what can this person formally deduce from his starting assumptions? The Bayesian approach is better suited than the classical one when it comes to drawing conclusions from small samples. In the words of E.T. Jaynes, a fierce proponent of Bayesian statistics: *"Scientific inference is concerned, necessarily, not with empty assertions of 'objectivity' but with information processing; how to extract the best conclusions possible from the incomplete information available to us."* Because such

[4] Example and some parts of this Section taken from: R. Durbin, S. Eddy, A. Krogh, G. Mitchinson: *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University press 2000

questions have recently found to be of extreme practical relevance – not the least in bioinformatics and in Artificial Intelligence – Bayesian statistics has seen a surge of interest in the last two decades, and has become very important for practical machine learning techniques.

Until recently there was something like a "war of believers" between the two approaches. The belligerent atmosphere is reflected in the most prominent, original textbook on Bayesian statistics (4 MB) by E.T. Jaynes, called not very modestly "Probability theory: the Logic of Science". Click on http://www.quackgrass.com/roots/0796rts.html for a short intro to Bayesian logic / probability theory. Today the aggressive tone has largely vanished and both approaches to probability are considered as valid, if alternative, perspectives.

A condensed, detailed, rigorous writeup of the basic definitions of probability theory has been produced by Dr. Mingjie Zhao as reference for a probability primer accompanying course in 2007. This document is available online, and I suggest that you download and print it.

Both approaches share the definitions (but not the interpretation) of some elementary concepts, which we will now revise.

Event space, probability space. Symbol: Ω. This concept, which is the fundament of statistics and probability theory, is unfortunately very hard to understand. The reason for this conceptual difficulty is that Ω has a dual nature: (i) as a real-world entity, which cannot be formally specified but needs everyday language to be described, and (ii) as a mathematical object that can be formally specified... Event spaces are, so to speak, the interface between the real world and mathematical (probabilistic) models. If one looks at some event space from the real-world side, one sees a real-world thing, which can be described only with real-world language, i.e. plain English. If one looks at some Ω from the formal side, from maths, one sees a mathematical object (a set, to be more precise – an object of set theory). Confused? you should be...

As a real-world entity, a good way to understand event spaces is to start from scientific paper writing (in the empirical sciences, like physics, biology, experimental psychology etc). Such papers typically describe an experimental setup where certain measurements are taken, or they specify (e.g. in the geosciences or botany) a location on this earth where observations have been made (e.g. a mountain range where rock samples have been taken, or a wildforest area where plant specimen were collected). The essence of empirical science is that other groups than the one who first did the experiment / expedition must be enabled to *reproduce* the findings. The outcomes of reproducing experiments are comparable to the originally reported results only to the extent that the reproducing experimenter reconstructs the original experimental setup (or goes to the same mountains or forests). Only if this similarity is warranted, the data collected by the second experimenter can be assumed to have the same "distribution" (we will soon explain that concept, for the time being your intuition must suffice) as the data sampled in the original investigation. An event space is what is specified (in a scientific paper, for instance) as a setup / context / location / experimental condition with respect to its role as a source of potential data. In papers in physics / chemistry / biosciences, this specification is usually done in a section called "methods". In papers in psychology / medicine /sociology it would typically be the specification of the population of human subjects that were investigated ("As subjects for our study we used undergraduate students from psychology, balanced in gender and with an age between 19 and 23..."). In sum, specifying an event space amounts to specifying particular conditions for collecting data.

As a formal entity, an event space is just seen as a set – and almost always denoted by Ω. In the light of what I said before, the elements ω of this set Ω should be considered as all the *potential acts of measurements* that *could* be made in experiments / expeditions of a certain type. A sometimes used terminology is to speak of ω as *realizations* of Ω. Each such realization is a source of measurement data. For example, if (in the real-world perspective) Ω is "undergraduate students from psychology, balanced in gender and with an age between 19 and 23...", then whenever in some university a particular undergraduate psychology student Mr. A with age between 19 and 23 is chosen from a gender-balanced sample, this Mr. A would be considered an $\omega \in \Omega$. However, mathematicians don't care about this real-world interpretation of the $\omega \in \Omega$. What they care about is that once Ω is fixed, the measurement data that can be obtained from $\omega \in \Omega$ have a well-defined statistical distribution. Mathematicians, then, care about the mathematical apparatus needed to equip (arbitrary) sets with the requisite add-on mathematical structure that enables them to handle such statistical distriubutions of measurement data obtained from $\omega \in \Omega$.

We will now describe this mathematical structure. Unfortunately, it is not simple (and has taken mathematics centuries to develop – only completed by the work of Andrey Kolmogorov in the early 1930's).

Events are subsets $A \subseteq \Omega$. In the extreme case, an event is an *elementary event* $\{\omega\} \subseteq \Omega$, but in general an event is a larger subset. In our example, "proteins belonging to a particular family" would be an event, or "proteins of family *X*, which have amino acid G in position 110".

σ-algebra, σ-field, event field: The set of all events in Ω. Typical symbol: F or $\mathfrak{A}$, $\mathfrak{B}$, ... (if you have Latex with the AMS package, use \mathfrak font!). We have $F \subseteq \mathrm{Pot}(\Omega)$. If Ω is finite, typically $F = \mathrm{Pot}(\Omega)$. With infinite Ω, F is typically *much* smaller than Pot(Ω). Not any subset of Pot(Ω) qualifies as a σ-algebra. A σ-algebra must adhere to certain structural axioms. Here is the definition of σ-algebras:

Definition 3.1: $F \subseteq \mathrm{Pot}(\Omega)$ is a σ-algebra if

1) $\Omega \in F$,
2) $A \in F \Rightarrow A^C \in F$ (closure under complement),
3) for every sequence $(A_n)_{n = 1, 2, \ldots}$ in F, the set $\bigcup_{n=1}^{\infty} A_n$ is in F (closure under countable union).

This definition reflects how we would like to be able to reason about events. Condition 1) says that the "all-event" is an event, that is, the event "we observe *some* individual from Ω". Condition 2) requires that if we have some event A, then we can also talk about the event "not A". Finally, condition 3) fixes that if we have a (countable) number of events A_n, then the event "we observe something from one of the A_n" is also a valid event.

σ-algebras are the fundamental concept of probability theory, of measure theory, and the theory of (Lebesgue) integration.

A **measureable space** is an event space equipped with some σ-algebra, written (Ω, F).

A **probability measure** P is defined as follows:

Definition 3.2: Let (Ω, F) be a measurable space. A function $P: F \to [0, 1]$ is a probability measure on (Ω, F), if

4) $P(\Omega) = 1$,
5) for every sequence $(A_n)_{n = 1, 2, \ldots}$ of pairwise disjoint events it holds that

$$P(\bigcup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} P(A_n) \qquad (\sigma\text{-additivity})$$

Remarks:

a) Conditions 1) through 5) are the **Kolmogorov axioms** of probability theory. In classical statistics, these axioms are the *foundation* of probability theory. In Bayesian statistics, these laws are *derived* from other axioms.

b) The "Bayesians" often admit in 5) only finite sequences.

c) The triple (Ω, F, P) is called a **probability space**.

d) Very often it holds that $P(\omega) = 0$ or even $P(A) = 0$ for nonempty A. Then A is a **null set**. For instance, if Ω is the set of all infinite sequences generated by some random number generator, then $P(\pi) = 0$: the chance of obtaining the digit sequence belonging to π is zero.

e) Here are some elementary properties of probability spaces:

i) $A, B \in F \Rightarrow A \cap B \in F, A \cup B \in F, A \setminus B \in F$.

ii) $\varnothing \in F$.

iii) For every sequence $(A_n)_{n = 1, 2, \ldots}$ of events it holds that $\bigcap_{n=1}^{\infty} A_n$ is in F.

iv) $A \subseteq B \Rightarrow P(A) \leq P(B)$.

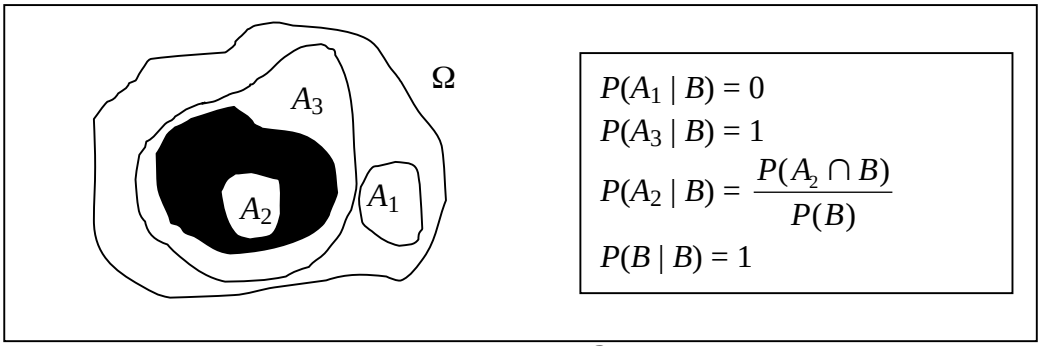
Conditional probability. Let (Ω, F, P) be a probability space, $B \in F$ an event with $P(B) > 0$. The function

(3.1) $P(\cdot \mid B)$: $F \to [0, 1]$

$$A \mapsto \frac{P(A \cap B)}{P(B)}$$

is again a probability measure on (Ω, F), the *conditional probability under hypothesis B*.

Here is a graphical display that explains how to think about conditional probabilities:



Here are some rules for computing with conditional probabilities:

1) $P(A \cap B) =: P(A, B) = P(A \mid B)\, P(B)$

2) Let $\Omega = \bigcup_{i \in I} B_i$ with pairwise disjoint B_i. Then for every $A \in F$,

$$P(A) = \sum_{i \in I} P(B_i) P(A \mid B_i),$$

the *formula of total probability*.

3) Let $\Omega = \bigcup_{i \in I} B_i$ with pairwise disjoint B_i. Then

$$P(B_n \mid A) \quad = \quad \frac{P(B_n) P(A \mid B_n)}{P(A)} \quad = \quad \frac{P(B_n) P(A \mid B_n)}{\sum_{i \in I} P(B_i) P(A \mid B_i)},$$

which is *Bayes formula*.

Baysian statistics *starts* from rules like $P(A \cap B) =: P(A, B) = P(A \mid B)\, P(B)$, which are justified as "rational", "intuitively correct" laws of reasoning about probabilities. (To be more precise, a Baysian statistician would write $P(A \cap B \mid M) =: P(A, B \mid M) = P(A \mid B,M)\, P(B \mid M)$ instead – Baysians *always* include a formal reference M to some prior knowledge about the relevant domain of reality into their formulas, reflecting the fact that all intuitive reasoning about probabilities must start from some prior assumptions about the world one is reasoning about.)

Further elementary concepts...

Observation space. Typical symbol: $(E, \mathfrak{B})$. This concept is closely related to, but fundamentally different from the measurable space (Ω, F). The basic intuition is that the "things" in Ω are "just there as they are there" (a Kantian philosopher might think of the *Ding an und für sich*, the "things as they are for themselves", not being understood or observed by humans). In order to get access to a thing $\omega \in \Omega$, one has to *observe* or *measure* it. The outcome of the observation is an object $a \in E$. In our protein example, ω might be a physical gene coding a protein in a biochemist's sequencing apparatus, and after sequencing the gene, a might be the formal sequence of amino acid symbols corresponding to the protein. Thus E would be created from Ω through the measurement operation "run a gene ω through a sequencer, transform the nucleic acid sequence into an amino acid sequence, and output the sequence of its symbols". With another observation operation one gets another observation space E. Taking up our example, with the measurement operation "run a gene ω through a sequencer, transform the nucleic acid sequence into an amino acid sequence, and output the 110th symbol that you get", one would only observe genes/proteins at the 110th position. The set of possible observation outcomes then would be the set $E = \{\mathsf{A}, ..., \mathsf{Y}\}$, which has 20 elements. In sum, the observation space builds on a set E that contains *all possible outcomes of observing elements* $\omega \in \Omega$ under a given observation procedure.

It is very important to keep the observation space apart from the underlying probability space with its measurable space (Ω, F). In our last example, the event space Ω of the underlying probability space (Ω, F, P) would have as many elements as there are proteins of the family – potentially (given the open-endedness of evolution) infinitely many. In contrast, E contains just 20 elements.

An observation space is also equipped with a σ-algebra, $\mathfrak{B}$. In our last example, since E is finite, we would typically take $\mathfrak{B} = \mathrm{Pot}(E)$. The pair $(E, \mathfrak{B})$ is thus a measurable space.

Random variables, typical symbol: X. If you thoroughly understand the concept of a random variable, nothing can happen to you in the remainder of this lecture! Formally, a random variable is a mapping $X: \Omega \to E$, also written as $X: (\Omega, F, P) \to (E, \mathfrak{B})$. Intuitively, a random variable describes a measurement or observation procedure. To each elementary event $\omega \in \Omega$, a random variable assigns an observation, or measurement outcome, $X(\omega) = x \in E$. In our example, X would assign to every protein the amino acid symbol detected at the 110th position.

Now comes a subtle and powerful idea. A random variable $X: (\Omega, F, P) \to (E, \mathfrak{B})$ "transports" the probability measure P from the underlying probability space (Ω, F, P) into the observation space, creating there another probability measure, the *induced measure* P_X, by the prescription

$$\forall B \in \mathfrak{B}: \; P_X(B) = P(X^{-1}(B)) \tag{3.2}$$

In our example, for instance, we would have

$$P_X(\text{"symbol A is observed"}) = P(\text{"all globulines that show A at position 110"})$$

Instead of $P_X(B)$, the notation $P(X \in B)$ is also used.

The observation space $(E, \mathfrak{B})$ is typically much smaller and has a simpler structure than the underlying probability space (Ω, F, P). This reflects the loss of information usually incurred by any measurement process!

Distribution. The induced probability measure P_X on the observation space is called the *distribution of the random variable X*.

In real-world modelling tasks, the underlying probability space (Ω, F, P) is usually an object that we cannot directly access or model mathematically (think of how difficult it would be to model "the set of all globulines in all organisms of past, present and future"). However, the much simpler observation spaces with their induced probability measures (distributions) *can* be analysed. Therefore,

the main object of study in probability theory and statistics is distributions.

All we have said so far is just the long, hidden story behind the simple distributions that we are used to work with. It is the story of how axiomatic probability theory tries to come to terms with the concept of probability. The probabilities we mostly speak of are distributions,

which are "borrowed" from the underlying (but ignored) probability spaces by virtue of Eqn. (3.2).

A cautionary remark. The distinction between the underlying probability space and distributions is sometimes obscured in introductory textbooks that try to make life (too) easy for the reader. In these books, the distinction between Ω and E is not made. For instance, one may find that the set {1, 2, 3, 4, 5, 6} of possible outcomes of throwing a die is called a probability space, and the set of probabilities of these outcomes (all 1/6 for a fair die) are called a probability measure (used interchangeably with distribution). Technically speaking this is admissible: because the events $\omega \in \Omega$ are not formally defined but represent one's pre-mathematical choice of the "piece of reality" one wants to model, one is in principle free to choose *anything* for Ω, including E. But if the distinction is dropped, the intuitive interpretation of random variables as measurement/observation operators is lost; furthermore, some themes of advanced probability theory become impossible to treat (for instance, the question of whether the zigzag trajectories of Brownian motion can be assumed to be continuous).

In finite observation spaces, a distribution is most conveniently expressed by a table or a bar chart giving the values of $P_X(x) = P(X = x)$ for all $x \in E$. In continuous-valued observation spaces, a distribution is often represented by a probability density function (*pdf* is a much-used abbreviation). For instance, the normal distribution density function with mean μ and standard deviation σ is given by

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma}\ e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.3}$$

We use smallcap p to denote pdf's and capital P to denote probability measures or distributions in general. For the event B = "measurement lies between 0.5 and 1.2" (note $B \in \mathfrak{B}$) we can use (3.3) to calculate a probability

$$P(X \in B) = \int_{0.5}^{1.2} p(x)dx\,. \tag{3.4}$$

Note that B does not refer to a single measurement "action" – it refers to the class of all individual measurement actions that return a value between 0.5 and 1.2. In probability theory, when talking about distributions, the concept "event" refers not to individual observation actions (those would be the measurements $X(\omega) \in E$), but to classes of measurements, defined by ranges of their outcomes, that is, the concept "event" refers to the $B \in \mathfrak{B}$.

When it is clear from the context that one is dealing with distributions (and not with the underlying probability space), often the induced probability measure P_X is simply written as P. In fact, most *"P"* symbols that you will encounter in statistics should actually be interpreted as P_X.

Numerical random variables and expectation. A random variable is *numerical* if the observation space E is numerical, that is, integer-, real-, or complex-valued. The expectation of a numerical random variable X is its "average value" and for integer-valued X is given by

$$E[X] = \sum_{i=-\infty}^{i=\infty} i\ P(X = i), \tag{3.5}$$

and for the case of real-valued variables with pdf p is given by

(3.6) $$E[X] = \int_{x=-\infty}^{x=\infty} x\ p(x)dx\,.$$

Given a numerical random variable X, one obtains a random variable X' that is *normalized to zero mean* by putting $X' = X - E[X]$.

Independent and uncorrelated random variables. Let X: $(\Omega, F, P) \to (E, \mathfrak{B})$ and Y: $(\Omega, F, P) \to (F, \mathfrak{C})$ be two random variables in arbitrary observation spaces. They are called *independent* if for all $A \in \mathfrak{B}$, $B \in \mathfrak{C}$ it holds that

(3.7) $$P(X \in A, Y \in B) := P(X^{-1}(A) \cap Y^{-1}(B)) = P(X \in A)\, P(Y \in B).$$

If X and Y are numerical RVs, then X and Y are *uncorrelated* if

(3.8) $$E[X\ Y] = E[X]\ E[Y],$$

or equivalently, if their *covariance*

(3.9) $$\mathrm{cov}(X, Y) = E[(X - E[X])\ (Y - E[Y])]$$

is zero. Only random variables with numerical values can be uncorrelated, but random variables with values in any arbitrary observation space can be independent. Independent numerical random variables are always uncorrelated, but uncorrelated numerical random variables are not necessarily independent. Thus independence is a (much) stronger notion than uncorrelatedness. Unfortunately, in analytical and computational investigations, independence is also much more difficult to prove or use.

Side remark. A modern and fashionable field within machine learning is *blind source separation*. Given n statistically independent signal sources (e.g., speakers in a room) and n measurements which each pick up a different mixture of the source signals (e.g. microphones placed at different positions in the room), one can use the fact that the sources are statistically independent to learn from a training sequence of the mixed measurements a filtering device that re-separates the signal mixtures into their independent components. The quality of the separated signals is sometimes astounding. Applications (besides speech processing): picking out a unborn baby's heartbeat from the "noise" signals generated inside a mother; detecting individual signal sources in EEG mixtures of signals. Check http://web.media.mit.edu/~paris/ica.html for pointers to people, papers, labs and striking audio-demos. The results obtained from independence analysis with the modern techniques of blind source separation are often much stronger than results obtained with the more traditional and easier methods of classical linear signal analysis and filtering, which rely merely on uncorrelatedness.

Joint and marginal distributions. Often a probability space (Ω, F, P) is observed/described by several random variables $\mathbf{X} = (X_i)_{i\,=\,1,\,\ldots,\,n}$ simultaneously. These variables may take values in different observation spaces $(E_i, \mathfrak{B}_i)$. Think of this as describing a complex piece of reality in terms of a number of different measurables, observables, concepts. One may glue the individual random variables together in a single *product* random variable

$\bigotimes_{i\in\{1,...,n\}} X_i = X_1 \times \cdots \times X_n$ that takes values in the product space $\bigotimes_{i\in\{1,...,n\}} E_i = E_1 \times \cdots \times E_n$, so the values taken by the product random variable is an *n*-tuple of the individual variables' values:

(3.10) $$(\bigotimes_{i\in\{1,...,n\}} X_i)(\omega) = (X_1(\omega),..., X_n(\omega)) = (x_1,..., x_n)\,.$$

The distribution $P_{\bigotimes_{i\in\{1,...,n\}} X_i}$ of this product random variable is called the *joint distribution* of the random variables $(X_i)_{i = 1, ..., n}$. We also write $P(\bigotimes_{i\in\{1,...,n\}} X_i)$ or simply $P(\mathbf{X})$ for the joint distribution. Notice that the joint distribution is likely to be a very unwieldy beast. To see why, consider the simplest possible case, where all the concerned random variables are binary (you may conceive them as Boolean observations, indicating the presence or absence of an observation). Then the distribution $P(X_i)$ of any individual variable is just a histogram over the values 0 and 1. However, the joint distribution would assign a probability value to each possible combination of the *n* binary observations, which makes this a histogram over 2^n arguments. In naive words, joint distributions are "exponentially more complex" than the individual distributions. If the individual distributions are themselves not as simple as just a binary distribution, it soon becomes practically impossible even to write down some closed formula for characterizing the joint distribution – and the computational and computer-based methods for handling complex distributions that we will learn about in this lecture will be needed.

The joint distribution of a descriptive ensemble $(X_i)_{i = 1, ..., n}$.comprises the *complete* probabilistic information about the piece of reality that one is modelling by $(X_i)_{i = 1, ..., n}$. Any specific question that one might ask about this piece of reality can be derived from $P(\bigotimes_{i\in\{1,...,n\}} X_i)$. For instance, one may wish to ignore some of the descriptors and ask for the distribution of one or a few selected observables only. Such "ignore the rest" distributions are called *marginal distributions*. They can be computed, in principle, by integrating away the others. For instance, if the joint distribution is characterized by an *n*-dimensional pdf *g*, we could recover the pdf g_1 of the marginal distribution of X_1 by

(3.11) $$g_1(x_1) = \int_{\Re^{n-1}} g(x_1,..., x_n)\, dx_2 dx_3 ... dx_n\,,$$

or in the discrete case, where each X_i takes values in $E_i = \{a_1^i,..., a_{m_i}^i\}$, the marginal probabilities of *X1* would be obtained by summing over all combinations of the other variables' values:

(3.12) $$P(X_1 = a_x^1) = \sum_{a^2\in E_2} ... \sum_{a^n\in E_n} P(X_1 = a_x^1, X_2 = a^2,..., X_n = a^n)\,.$$

Marginal distributions of more than a single variable can be computed by integrating/summing away the remaining ones in a similar way. While thus the marginal distributions can be recovered from the joint distribution, conversely the joint distribution can be constructed from the marginal distributions only if the $(X_i)_{i = 1, ..., n}$.are independent. Then (and only then) it holds that the joint pdf *g* is

(3.13) $$g(x_1,..., x_n) = g_1(x_1)\cdot ... \cdot g_n(x_n)$$

for distributions with pdf's, and

(3.14) $$P(X_1 = a^1, X_2 = a^2, ..., X_n = a^n) = \prod_{i=1,...,n} P(X_i = a^i) .$$

The joint distribution $P(\bigotimes_{i\in\{1,...,n\}} X_i)$ can be *factorized* into a product of conditional distributions by

(3.15) $$P(\bigotimes_{i\in\{1,...,n\}} X_i) = P(X_1)\,P(X_2 \mid X_1)\,P(X_3 \mid X_1, X_2) \ldots P(X_n \mid X_1, ..., X_{n-1}) .$$

The proof is a homework exercise.

Samples. The underlying event space Ω contains all *possible* individual measurable events ω (or elementary events). In life's reality, only a small fraction $\omega_i \in \Omega$ (where $i \in I$ and I is by life's necessity finite) of all possible measurable events is realized in concrete observations. Such a set $\{ \omega_i \mid i \in I \}$ of actually realized measureable events is called a *sample*. At least, this is the strict definition of the term.

However, we have noted that statisticians prefer not to talk about measurable events ω_i but rather like to think in terms of their distributions. This is reflected in another, related usage of the word *sample*, which also reflects a typical situation in the experimental sciences, namely, that some experiment or measurement is repeated many times in order to obtain an as precise as feasible estimate of some quantity of interest (for instance, by taking the mean over reapeated measurements).

To model this situation the following approach is taken in statistics. The elements ω of the underlying probability space (Ω, *F*, *P*) are taken to be the *sequences* of repeated experiments – and for mathematical convenience, it is assumed that one such "repeat-experiment-session" ω comprises infinitely many repetitions of the experiment. Next, a sequence $(X_i)_{i \in \mathbb{N}}$ of random variables is considered, where $X_i(\omega)$ refers to the *i*-th measurement outcome in the repeat-experiment session ω. This is of course an idealized picture: in practice, an experiment cannot be repeated infinitely many times. What one has in real-life, is the outcomes of *n* many measurements of one repeat-experiment session ω, that is, the data that one has really available are comprised in a vector $\mathbf{X}(\omega) = (X_1(\omega), ..., X_n(\omega)) = (x_1, ..., x_n) \in E^n$. Such data vectors are then called *samples*.

Although this might appear a bit contrived, it gives a faithful account of how research in the empirical sciences should be carried out: In some Lab A, some quantity of interest is derived with an as great as possible precision (implying repeated measurements) – this is, $(X_1(\omega), ..., X_n(\omega))$ is used by Lab A's statistician to estimate the quantity of interest. Another Lab B may want to contest or improve on Lab A's result. They will also carry out a repeat-experiment session ω', obtain a sample $(X_1(\omega'), ..., X_{n'}(\omega'))$, and infer something about the quantity of interest from *this* sample. Typically, their results will somewhat deviate from Lab A's results. The question, then, is how the conclusions obtained in Lab A from the sample $(X_1(\omega), ..., X_n(\omega))$ can be *compared* with the conclusions obtained in Lab B from the sample $(X_1(\omega'), ..., X_{n'}(\omega'))$ – for instance, if $n' > n$, to what extent are the conclusions drawn by Lab B's

statistician more reliable than the findings in Lab A? Such considerations lie at the heart of statistics and the theory of statistical estimation (of quantities of interest from samples).

This strict understanding of a sample as $(X_1(\omega'), ..., X_{n'}(\omega'))$ is not easy to grasp, especially because there is also a "naïve" setup of a probability model where one has only a single random variable X. An example will be helpful to sort these subtle concepts out.

Consider an article in a medical journal where it is stated that patients with a particular form of cancer have, with a probability of 0.1, a particular antibody A in their blood. The most natural probability model would introduce the following items:

"Natural" model:

Ω: set of all patients with this type of cancer (suitably restricted, e.g. all patients *in Germany* who *come to hospital* – depends on the data source used for the journal article)
X: measuring whether a patient carries antibody A. This would typically be effected by a binary indicator X, i.e. the measure space E is $\{0,1\}$ and for a patient ω, $X(\omega) = 1$ iff the patient carries antibody A.

This natural model contrasts with the model that professional statisticians would use. They would set up their probability space and random variables as follows:

"Professional" model:

Ω: set of all sequences of tests for antibody A that would be carried out for one study (the original study of the journal, or some confirmation studies, or hypothetical studies of the same sort that *could* be done). One $\omega \in \Omega$ would be the suite of all such measurements done for one study. (Again, suitable restrictions would apply, e.g. to all such studies in Germany, or studies carried out in a particular year)
X_i: for $i = 1, 2, ...$, $X_i(\omega)$ is the i-th measurement of the sample for the study ω. Again, a standard choice for the measure space would be the indicator values $\{0,1\}$.

The variables $X_1, X_2, ...$ of the "professional" model would be assumed to be i.i.d., and they would have the same distribution as the RV X from the "natural" model. Both the "naïve" and the "professional" type of model are mathematically correct and conceptually legitimate (and either of the two could be used for answering exam questions…).

The more complicated (and I admit: somewhat less intuitive) "professional" type of model becomes a necessity when it comes to build the theory of statistical estimation – that is, to understand how one can extract an estimate of the distribution P_X from a sample $(X_1(\omega), ..., X_n(\omega)) = (x_1, ..., x_n)$. This is the core task of statistics (classical and Bayesian):

Basic task of statistics. Given a sample $(x_1, ..., x_n)$, find out something about the underlying distribution P_X – typically, give an estimate of P_X.

Parametrized distributions. Concretely, P_X is often to be represented by some (few) parameters. For instance, a normal distribution P_X is characterized by its pdf, which in turn is characterized by its mean μ and its standard deviation σ, that is, by two parameters. In our amino acid example, the distribution P_X of amino acid symbols at location 110 would be represented by 20 probability values of the various possible symbols, that is, by 20 parameters. A common symbol for the set of parameters characterizing a distribution is θ. With parametrized distributions, the basic task of statistics then spells out like this:

> **Basic task of statistics, formulated as parameter estimation task.** Given a sample $(x_1, ..., x_n)$, give an estimate $\hat{\theta}$ of the parameters of the distribution.

Note that there are other, "parameter-free" ways of characterizing a distribution – we will soon meet some.

Estimators. Formally, the task estimating the parameters θ of a distribution from a sample can be expressed in terms of a function T_n which assigns to each sample $(X_1(\omega), ..., X_n(\omega))$ of size n a set $\hat{\theta}$ of parameters. Such functions T_n: $(X_1(\omega), ..., X_n(\omega)) \mapsto \hat{\theta}$ are called *estimators* or *estimation functions*. Note that $T_n(X_1(\omega), ..., X_n(\omega))$ is fully determined by ω, so we might also write $T_n(\omega)$ – that is, estimators are themselves random variables.

The art and science of statistics is to find "good" estimators. The art and science of (much of) algorithmical modelling is to find "good" ways of describing pdf's – an analytic expression being rather the exception than the rule, because one mostly is confronted with high-dimensional, badly-behaved distributions for which finding an analytic pdf is all but hopeless. (And the art and science of *machine learning* is to find good estimators for a kind of pdf representation that the modellers wish to use, with a little more emphasis than in "ordinary" statistics on efficient algorithms – $T((x_1, ..., x_n))$ must be efficiently computable, that is, one looks for fast learning algorithm).

While the notion of an estimator typically refers to parametrized distributions, you may also use it in a more loosely fashion for any method that creates a characterization of a distribution from a sample.

> **One basic task of (the statistical branch of) algorithmical modelling.** Given a piece of world (POW) that one wishes to capture in a (statistical) model, find a way to represent its distribution(s) such that
>
> - these distributions are complex enough to capture essential aspects of the POW, and adapted to the particulars of the POW (e.g., a normal distribution would be incapable to express the most interesting aspects of a stock market index)
> - these distributions are simple enough to be manipulated with efficient algorithms (aspects of memory demands, "sampling", computing probabilities of events)
> - these distributions can be learned from data (the machine learning part of algorithmical modelling)
>
> Unlike in traditional statistics, you are allowed to use a lot of raw computational power, heuristics, and borrow ideas from other fields such as physics, neurobiology, psychology, evolutionary biology, or any other.

3.2 Maximum-likelihood estimators

One of the most common approaches to design estimators is the maximum-likelihood approach. It is conceptually transparent, it is a typical "frequentist" approach, and it works well when the sample size is not too small. We will explain it with our amino acid example.

We use abbreviation *D* ("data") for the sample. The distribution estimation (or learning) task is the following:

- *Given:* a sample *D* of *n* observations of amino acids in some location in *n* representatives of some protein class. In Fig. 3.1, we would have $n = 7$ and for instance D = H H H H H H L (in the location shown in the second column in Fig. 3.1) or D = K K V N E G A (last column). Another, equivalent way to write *D* is as a count vector $D = (n_1, ..., n_{20})$ where n_i is the number of counts of the *i*-th amino acid symbol in the sample.
- *Wanted:* an estimate $\hat{\theta} = (\hat{\theta}_1,...,\hat{\theta}_{20}) = (\hat{P}_X(\mathsf{A}),...,\hat{P}_X(\mathsf{V}))$ of the 20 parameters describing the amino acid distribution in some location over all proteins in a family.

Approach: estimate $\hat{\theta} = (\hat{\theta}_1,...,\hat{\theta}_{20})$ such that the $P(D \mid \theta)$ is maximized over all θ, that is, put

(3.16) $$T(D) = \theta^{\mathrm{ML}} = \mathrm{argmax}_\theta\, P(D \mid \theta).$$

$P(D \mid \theta)$ is called the *likelihood* of θ given *D*, and often written as $\mathcal{L}(\theta)$. The notion of likelihood must not be confused with the notion of probability – they are dual concepts. $P(D \mid \theta)$ is the *probability* of *D* given θ, and it is the *likelihood* of θ given *D*.

For simple frequency counts as in our example, the ML-estimator θ^{ML} can be analytically shown to be

(3.17) $$\theta^{\mathrm{ML}} = (\theta_1^{\mathrm{ML}},...,\theta_{20}^{\mathrm{ML}}) = (\frac{n_1}{N},...,\frac{n_{20}}{N}),$$

where *N* is the sample size (here $N = 7$) and n_i is the count number of the *i*-th amino acid symbol in the sample.

This is beautifully simple and apparently convincing – but very inadequate for small sample sizes. Consider D = H H H H H H L. The maximum-likelihood estimator would assign zero probabilities to all amino acids except H and L. But every geneticist worth his/her salt would expect that in the protein family at large, every other amino acid would also occur in this location in some protein, albeit maybe rarely. But if we really assign zero probabilities to them, we would be forced to exclude every such protein from the family, which is not something we want to happen.

ML estimators of conditional distributions with Gaussian noise. There exists an intimate connection between ML estimators and the "method of least mean square (LMS) errors". We first recapitulate from High School the essentials of the LMS method. It applies in *regression tasks* where one wants to recover a deterministic input-output relationship from noisy observations of the outputs. Assume a situation where a researcher manipulates some experimental setup by subjecting it to inputs $\mathbf{x}_i \in \mathbb{R}^m$, where $i = 1, ..., N$. The researcher obtains scalar measurements y_i as a result. An example would be a psychological experiment where a graphical pattern is flashed on a screen at position $\mathbf{x}_i = (x^1_i, x^2_i)$, and a response time y_i of the subject is measured; an other example would be a medical survey where each $\mathbf{x}_i$ describes a patient by a vector of diagnostic variables, and y_i would be the remaining lifetime after diagnosis. Galileo did do the same thing when he let a heavy object fall from different heights x_i ($m = 1$ in this case) and recorded the falling times y_i (I did not check the history books of modern physics – I just guess that Galileo did something like this). Well, this is just **the** most standard situation in the empirical sciences. Now assume that the researcher knows that there is a law of nature which deterministically establishes a function $f: \mathbb{R}^m \rightarrow \mathbb{R}$, that is, on input $\mathbf{x}_i$ the "true" outcome would be $y_i = f(\mathbf{x}_i)$. The researcher even knows the nature of this function – it comes from a family parametrized by parameters $\theta \in \mathbb{R}^d$. So the researcher knows that $f = f(\theta^{\text{true}})$ for a particular parameter vector θ^{true}. But, the researcher does not know θ^{true}, and wants to estimate these parameters from his experimental data. For example, Galileo (or later, Newton) might have known that the falling time y is equal to $y = \text{sqrt}(2\, x / g)$, where g is the constant of gravitation, which would be the unknown parameter θ^{true} which he wanted to estimate from his falling experiments.

In such situations, the LMS method is to estimate the sought-after θ^{true} as the parameter vector which minimizes the mean square error of the observations, i.e. to calculate

$$\hat{\theta}^{\text{LMS}} = \arg\min_\theta \sum_{i=1}^{N} (f(\theta)(\mathbf{x}_i) - y_i)^2. \tag{3.18}$$

As a justification for the LMS method, you may remember from High School statements like, "we want an estimate that punishes larger deviations from the predicted true outcome more strongly than smaller deviations" – at least, that was how I was taught the LMS principle. However, there is a better and more rigorous justification for the LMS method than this. This runs as follows.

We assume that the measurement process is subject to Gaussian noise, that is, if the effective parameter is θ, then upon input $\mathbf{x}_i$ the observation y_i will be drawn from a Gaussian distribution centered on $f(\theta)(\mathbf{x}_i)$:

$$p(y \mid \mathbf{x}_i, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(f(\theta)(\mathbf{x}_i) - y)^2}{2\sigma^2}\right), \tag{3.19}$$

where p is the pdf of the distribution of the y_i and σ is the standard deviation of the Gaussian (which we assume is the same for all possible inputs $\mathbf{x}$). Assuming that the observations y_i are independent, and calling the ensemble of all outcomes y_i our data D, then

$$p(D \mid \{\mathbf{x}_i\}, \theta) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(f(\theta)(\mathbf{x}_i) - y_i)^2}{2\sigma^2}\right) \tag{3.20}$$

is the likelihood of θ, or more conveniently,

$$\mathcal{L}(\theta) = \frac{N}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(f(\theta)(\mathbf{x}_i) - y_i)^2 \tag{3.21}$$

its log likelihood. Maximizing the likelihood of θ amounts to finding

$$\hat{\theta}^{\mathrm{ML}} = \arg\max_\theta -\sum_{i=1}^{N}(f(\theta)(\mathbf{x}_i) - y_i)^2, \tag{3.22}$$

which is identical to (3.18). We thus find that under an assumption of Gaussian measurement noise, the LMS estimate of the true parameters is the maximum likelihood solution. A note on terminology: When statisticians speak of a "regression problem", they typically refer to exactly this situation, where the parameters of a *regression function f*(θ) are computed by a LMS calculation, with the tacit understanding that this is also the ML estimate to the extent that a Gaussian measurement noise assumption is valid.

In (very) many cases, one does not know the nature of *f*(θ). Then, one often resorts to the least-committing assumption that *f* is linear, that is, $f(\mathbf{x}) = f(\mathbf{w})(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$, where $^{\mathrm{T}}$ is matrix/vector transpose. In this context the parameters θ are typically denoted by **w**, and called the (linear) *regression weights*. The LMS/ML solution $\hat{\mathbf{w}}^{\mathrm{ML}}$ can then be analytically computed in closed form via the following derivation. First observe that

$$\hat{\mathbf{w}}^{\mathrm{ML}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_i - y_i)^2 = \arg\min_{\mathbf{w}} \sum_{i=1}^{N}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_i)^2 - 2\mathbf{w}^{\mathrm{T}}\mathbf{x}_i y_i + y_i^2. \tag{3.23}$$

At the argmin, the gradient w.r.t. **w**

$$\nabla_{\mathbf{w}} = \sum_{i=1}^{N}(2\mathbf{w}^{\mathrm{T}}\mathbf{x}_i\mathbf{x}_i^{\mathrm{T}} - 2\mathbf{x}_i^{\mathrm{T}}y_i) = 2\mathbf{w}^{\mathrm{T}}\sum_{i=1}^{N}\mathbf{x}_i\mathbf{x}_i^{\mathrm{T}} - 2\sum_{i=1}^{N}\mathbf{x}_i^{\mathrm{T}}y_i = \mathbf{0}^{\mathrm{T}} \tag{3.24}$$

must be the all-zero row vector. Transposing this equation, and introducing the input data matrix $\Phi = (\mathbf{x}_1 \ldots \mathbf{x}_N)^{\mathrm{T}}$ and an output vector $\mathbf{y} = (y_1 \ldots y_N)^{\mathrm{T}}$, (3.24) can be written as

$$\mathbf{0} = \Phi^{\mathrm{T}}\Phi\mathbf{w} - \Phi^{\mathrm{T}}\mathbf{y}, \tag{3.25}$$

which resolves to **w** as

$$\mathbf{w} = (\Phi^{\mathrm{T}}\Phi)^{-1}\Phi^{\mathrm{T}}\mathbf{y}. \tag{3.26}$$

The matrix $(\Phi^{\mathrm{T}}\Phi)^{-1}\Phi^{\mathrm{T}}$ is known as the (left) *pseudo-inverse* of Φ. It generalizes the usual matrix inverse, which is defined only for full-rank square matrices, to full-column-rank rectangular matrices of size $a \times b$, where $a \geq b$. Indeed, it is obvious to check that $[(\Phi^{\mathrm{T}}\Phi)^{-1}\Phi^{\mathrm{T}}]\Phi = \mathbf{I}_{b\times b}$. Formula (3.26) indicates one way to compute solutions to linear regression problems: first compute $(\Phi^{\mathrm{T}}\Phi)^{-1}$, then multiply with $\Phi^{\mathrm{T}}\mathbf{y}$. This is fast but prone to numerical instability when $(\Phi^{\mathrm{T}}\Phi)^{-1}$ is not well-conditioned. If you call in Matlab the

routine *pinv* (for pseudo-inverse), another algorithm is used which is slower but more stable because it avoids to explicitly compute $(\Phi^T\Phi)^{-1}$.

3.3 The bias-variance dilemma

We have just seen how a maximum-likelihood estimator can yield clearly unsatisfactory results. The problem we stumbled across is known as the *overfitting* problem, or the *bias-variance dilemma*. In fact, it is a *general* problem that *always* raises its ugly head when it comes to statistical model estimation. In a nutshell, the best model of a probability distribution that one can get, given empirical data, is *not* the model yielded by maximum-likelihood methods – because ML methods try to come as close as possible to the empirical distribution represented by the training data; as a result, the model also "models" the purely random fluctuations of the training data. A thorough treatment of the bias-variance dilemma has in the last two decades been started in a modern branch of statistics called *statistical learning theory*. I will here only give a traditional account of the problem, which also explains why it is called "bias-variance" dilemma.

We consider only a special case here, which is enough to demonstrate the concept. Assume that you possess an estimator T_n: $(X_1(\omega), ..., X_n(\omega)) \mapsto \hat{\theta}(\omega)$, where $\hat{\theta} \in \mathbb{R}^d$. We ask the question, how much does the estimate $\hat{\theta}(\omega)$ deviate, in the mean square error sense, from the true value θ? That is, we ask for the value of $E[(\hat{\theta} - \theta)^2]$. We can compute this as follows:

$$\begin{aligned} E[(\hat{\theta} - \theta)^2] &= E[(\hat{\theta} - E[\hat{\theta}] + E[\hat{\theta}] - \theta)^2] \\ &= E[(\hat{\theta} - E[\hat{\theta}])^2] + E[(E[\hat{\theta}] - \theta)^2] + 2E[(\hat{\theta} - E[\hat{\theta}])(E[\hat{\theta}] - \theta)] \\ &= E[(\hat{\theta} - E[\hat{\theta}])^2] + (E[\hat{\theta}] - \theta)^2. \end{aligned} \tag{3.27}$$

The third term in the second line vanishes because $2(E[\hat{\theta}] - \theta)E[(\hat{\theta} - E[\hat{\theta}])] = 2(E[\hat{\theta}] - \theta)(E[\hat{\theta}] - E[E[\hat{\theta}]]) = 2(E[\hat{\theta}] - \theta) \cdot 0 = 0.$ Among the remaining two terms, the first term, $E[(\hat{\theta} - E[\hat{\theta}])^2]$, gives the *variance* of the estimates, and the second, $(E[\hat{\theta}] - \theta)^2$, gives the systematic (averaged) squared amount by which the estimates deviate from the correct value. The quantity $E[\hat{\theta}] - \theta$ is the *bias* of the estimator.

Thus we have seen that the error inherent in an estimator can be split into two parts, a variance part that captures how much the estimates scatter around the mean estimate, and a bias part that quantifies how much the mean estimate differs from the correct value. The lessons taught by statistical learning theory is that there is a tension between the two: within a given class of estimators (say, neural networks) one can tune models either towards a low bias error (by data (over-)fitting, using larger networks) or towards a low variance error (by introducing a bias, e.g. small networks), but it is intrinsically impossible to optimize both simultaneously.

For an elementary demonstration of the bias-variance theme, consider a situation where $\theta = (\mu_1, \mu_2)$ is comprised of the two coordinate means of some distribution over $\mathbb{R}^2$. That is, the observation space is $E = \mathbb{R}^2$, and measurement values $X_i(\omega)$ are vectors $\mathbf{x}_i = (x_1, x_2)^T$,

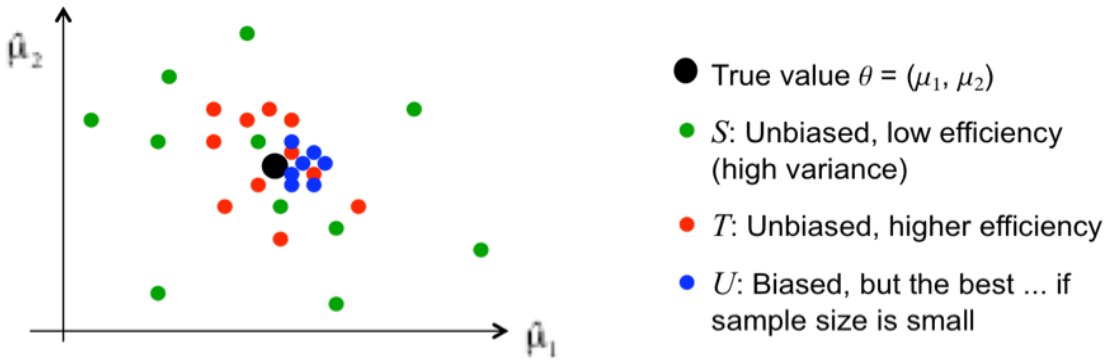
where superscript ^{T} denotes transpose. Now consider the following three estimators for $\theta = (\mu_1, \mu_2)$:

$$S : (\mathbf{x}_1, ..., \mathbf{x}_N) \mapsto (\mathbf{x}_1 + \mathbf{x}_2)/2$$

$$T : (\mathbf{x}_1, ..., \mathbf{x}_N) \mapsto (\mathbf{x}_1 + \ldots + \mathbf{x}_N)/N$$

$$U : (\mathbf{x}_1, ..., \mathbf{x}_N) \mapsto 1/2 \cdot \begin{pmatrix} m_1 \\ m_2 \end{pmatrix} + (\mathbf{x}_1 + \ldots + \mathbf{x}_N)/2N$$

where $(m_1, m_2)^T$ is an informed guess about true (but not precisely known) mean $(\mu_1, \mu_2)^T$. The following figure shows typical outcomes of applying these estimators to samples $(\mathbf{x}_1, ..., \mathbf{x}_N)$. It turns out that the *U* estimator would fare best in the sense of yielding the lowest expected error, although it is not unbiased – its estimates will be centered not around $(\mu_1, \mu_2)^T$ but around $((m_1, m_2)^T + (\mu_1, \mu_2)^T) / 2$. Intuitively speaking, it is superior because (and if) our guess $(m_1, m_2)^T$ comes close to the true value. This is a simple instance of a general principle in designing estimators: whenever one has *some* prior insight in the nature of the true parameters θ, and one finds a way to insert this knowledge into the estimator, then one may reasonably hope that the resulting estimator is better than another estimator where this prior knowledge has not been inserted. Since this prior knowledge will usually not exactly hit the correct θ, it will however introduce a *bias* into the estimator. In the next subsection 3.4, we will see how one can insert such prior information into an estimator in a principled fashion, such that prior information in which we only weakly trust has a lower impact than prior information in which we put much trust.



3.4 An estimator with Bayesian priors

In the ML-approach, the problematic zero probability estimates occurred because the estimator *exclusively* used the information given by the sample. The background knowledge that every protein expert has, namely, that every amino acid may (albeit possibly rarely) occur at every position, was ignored. This knowledge is crucial for getting an estimator that really makes sense, and it is the starting point in a Bayesian analysis: start from the assumption ("prior") $P(\theta \mid M)$ about *the distribution of parameters*.

This needs two bits of explanation.

- The first explanation is simple: *M* does not refer to a measurable event [like the *B* in the "classical" expression $P(A \mid B)$] but simply is the Bayesian way to make explicit that some background knowledge, or model, *M* is involved. *M* need not (and usually cannot) be formalized; it is a pointer to what biologists know *a priori* about distributions of amino acids in families of proteins.
- The second explanation is not so simple. The distribution $P(\theta \mid M)$ is a *hyperdistribution*: it describes how distributions (which are characterized by the various possible settings of θ) are distributed. Syntactically, it is just a distribution of numerical values (namely, the possible values of θ), but semantically, it is a distribution of distributions, because each possible value of θ represents a distribution. In our protein example, the prior wisdom that any amino acid might occur at a given site could be reflected in a choice of $P(\theta \mid M)$ which would assign a relatively high (and nonzero!) pdf value to the distribution parameter θ = (1/20, ..., 1/20).

With a Bayesion prior information $P(\theta \mid M)$, the biologist's background knowledge enters the parameter estimation as follows, through Bayes' formula:

$$P(\theta \mid D, M) = \frac{P(D \mid \theta, M)\, P(\theta \mid M)}{P(D \mid M)} = \frac{P(D \mid \theta)\, P(\theta)}{P(D)} = \frac{P(D \mid \theta)\, P(\theta)}{\int P(D \mid \theta) dP(\theta)}, \tag{3.28}$$

where the two rightmost terms are understood by Bayesians as a shorthand for the middle term. $P(\theta \mid D, M)$ is the *posterior distribution* (of parameters) and $P(\theta \mid M)$ is the prior distribution (of distributions...) or simply the prior.

Notice that $P(\theta \mid D, M)$ is a (hyper)distribution over parameters – but the target of model estimation is some estimate value $\hat{\theta}$ of parameters, not a distribution over candidate values. Therefore, Bayesian model inference must conclude with a final step where from the distribution $P(\theta \mid D, M)$ a specific value $\hat{\theta}$ is obtained. The usual approach here is to take the mean value of θ over this distribution, that is, calculate the *mean posterior estimate*

$$\hat{\theta} = \theta^{\mathrm{PME}} = \int \theta\, P(\theta \mid D, M)\, d\theta\,. \tag{3.29}$$

We will now concretely compute (3.29) step by step for our amino acid distribution problem, where $\theta = (\theta_1, ..., \theta_{20})$.

To start, we remark that with true $\theta = (\theta_1, ..., \theta_{20})$ the sample statistics for *D* should follow a multinomial distribution, that is, the probability to obtain a sample $D = (n_1, ..., n_{20})$ is

$$P(D \mid \theta) = \frac{N!}{n_1! \cdots n_{20}!} \prod_{i=1}^{20} \theta_i^{n_i}\,. \tag{3.30}$$

Next we try to fix how the prior $P(\theta \mid M)$ should look like. This is a subjective decision! For reasons that will soon become clear we (and most proteinologists) opt for the Dirichlet

distribution $P(\theta \mid M) = \mathcal{D}(\theta \mid \alpha)$ with parameters $\alpha = \alpha_1, ..., \alpha_{20}$. The pdf of $\mathcal{D}(\theta \mid \alpha)$ is given by

$$\text{(3.31)} \qquad \mathrm{D}(\theta \mid \alpha) = \frac{1}{Z(\alpha)} \cdot \prod_{i=1}^{20} \theta_i^{\alpha_i - 1} \cdot \delta\left((\sum_{i=1}^{20} \theta_i) - 1 \right).$$

Some comments will help to make this formula look less frightening. The factor $1/Z(\alpha)$ is just there to ensure that the integral over $\mathcal{D}(\theta \mid \alpha)$ is unity, that is it holds that

$$\text{(3.32)} \qquad Z(\alpha) = \int \prod_{i=1}^{20} \theta_i^{\alpha_i - 1} \cdot \delta\left((\sum_{i=1}^{20} \theta_i) - 1 \right) \quad d(\theta).$$

This integral has an explicit solution

$$\text{(3.33)} \qquad Z(\alpha) = \frac{\prod_i \Gamma(\alpha_i)}{\Gamma(\sum_i \alpha_i)},$$

where Γ is the gamma function. We don't have to understand Γ because it will later cancel out. The δ in (3.31) is the Dirac delta function which is defined by

$$\text{(3.34)} \qquad \delta(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{x} = 0 \\ 0, & \text{if } \mathbf{x} \neq 0 \end{cases} \quad \text{and} \quad \int_{\Re^n} \delta(\mathbf{x})\, d\mathbf{x} = 1.$$

Equipped with (3.30) and (3.31) we return to (3.28), which we now can calculate as a pdf:

$$\begin{aligned} p(\theta \mid D, M) &= \frac{P(D \mid \theta, M)\ p(\theta \mid M)}{P(D \mid M)} \\ &= \frac{1}{P(D \mid M)} \frac{N!}{n_1! \cdots n_{20}!} \prod_{i=1}^{20} \theta_i^{n_i}\ Z^{-1}(\alpha) \cdot \prod_{i=1}^{20} \theta_i^{\alpha_i - 1} \cdot \delta\left((\sum_{i=1}^{20} \theta_i) - 1 \right) \\ &= \frac{1}{P(D \mid M)} \frac{N!}{n_1! \cdots n_{20}!} \frac{Z(D+\alpha)}{Z(\alpha)} \mathrm{D}(\theta \mid D + \alpha) \end{aligned} \qquad \text{(3.35)}$$

where $D + \alpha = (n_1+\alpha_1, ..., n_{20}+\alpha_{20})$. Because $p(\theta|D,M)$ and $\mathcal{D}(\theta \mid D+\alpha)$ are probability distribution functions, the first three multiplicative terms in the last line of (3.35) must evaluate to unity, whereby we find

$$\text{(3.36)} \qquad p(\theta|D,M) = \mathcal{D}(\theta \mid D+\alpha).$$

Thus we have the posterior distribution of θ. In order to arrive at the posterior mean estimator, we integrate over the posterior distribution (of distributions!):

$$
\begin{aligned}
\theta_i^{\mathrm{PME}} &= \int \theta_i \; \mathrm{D}(\theta \mid D+\alpha)\, d\theta \\
&= Z^{-1}(D+\alpha) \int \theta_i \cdot \prod_{j=1}^{20} \theta_j^{n_j+\alpha_j-1} \cdot \delta\left((\sum_{i=1}^{20} \theta_i) - 1 \right) \, d\theta \\
&= \frac{Z(D+\alpha+e_i)}{Z(D+\alpha)} \\
&= \frac{\prod_i \Gamma(n_i+\alpha_i+e_i)}{\Gamma(\sum_i n_i+\alpha_i+e_i)} \cdot \frac{\Gamma(\sum_i n_i+\alpha_i)}{\prod_i \Gamma(n_i+\alpha_i)} \\
&= \frac{\Gamma(n_i+\alpha_i+e_i)}{\Gamma(n_i+\alpha_i)} \cdot \frac{\Gamma(\sum_i n_i+\alpha_i)}{\Gamma(\sum_i n_i+\alpha_i+e_i)} \\
&= \frac{n_i+\alpha_i}{N+A,}
\end{aligned}
\qquad (3.37)
$$

where by $D + \alpha + e_i$ we mean $(n_1+\alpha_1+e_1\ , ..., n_{20}+\alpha_{20}+e_{20})$; $e_i(n) = 1$ if $i = n$, else 0; $A = \alpha_1 + ...+\alpha_{20}$ and in the last step we exploit $\Gamma(x+1) = x\Gamma(x)$.

We see that the posterior mean estimator $\theta_i^{\mathrm{PME}} = \frac{n_i+\alpha_i}{N+A}$ is rather similar to the maximum likelihood estimator $\theta_i^{\mathrm{ML}} = \frac{n_i}{N}$, and we can see how the parameters α_i of the Dirichlet distribution can intuitively be interpreted as "pseudo-counts". That is, the prior knowledge is entered into the game here by augmenting the empirical counts n_i with extra pseudo-counts α_i. These pseudo-counts reflect the subjective intuitions of the biologist, and there is no rigorous rule of how to set them correctly. There are two limiting cases: if we don't add any pseudo counts, the Bayesian approach reduces the the maximum-likelihood case, that is, only the empirical information enters the estimation. This would drive us to the "far right" side in the bias-variance dilemma, that is, we run danger of overfitting. If we add, on the contrary, very large pseudo-counts, the final "estimate" will just replay the prior information with almost no influence from the empirical information; this would put us to the far left side in the bias-variance-dilemma, that is, we would just get our bias (the Bayesian prior) back. So the Bayesian approach does not solve the bias-variance dilemma; it only makes it transparent and forces the researcher to take his/her stand.

The outcome of (3.37) can be seen in yet another way, which indicates another way of how one may work in one's personal bias into a parameter estimation. Assume that according to your personal insight (before seeing the data) you expect that the parameters $\theta = (\theta_1,..., \theta_{20})$ have true values $\theta^{prior} = (\theta_1{}^{prior},..., \theta_{20}{}^{prior})$. This θ^{prior} does not incorporate any information from D and thus marks the extreme left (bias) end of the bias-variance dilemma. (Note that θ^{prior} is not a proper Bayesian prior – a proper Bayesian prior would be a distribution of distributions θ!). You compute the maximum-likelihood estimator $\theta^{\mathrm{ML}} = (\theta_1^{\mathrm{ML}},..., \theta_{20}^{\mathrm{ML}}) = (\frac{n_1}{N},..., \frac{n_{20}}{N})$ from D. θ^{ML} does not reflect any prior information, fits the data perfectly and thus marks the extreme right end of the bias-variance-range. Now, in order to settle at some compromise between the two extremes, construct "blended" estimators

(3.38) $\theta^{\text{post}} = q\,\theta^{prior} + (1 - q)\,\theta^{\text{ML}}$,

where $0 \le q \le 1$. Writing θ^{prior} as $\theta^{prior} = (\frac{\alpha_1}{A},...\frac{\alpha_{20}}{A})$ and putting $q = A / (N + A)$ and $(1 - q) = N / (N + A)$ yields the same result as (3.37). Note however that this procedure of linearly blending the parameters of a "personally expected" distribution with the parameters of a ML distribution does not universally work – not all types of parametrizations θ of distributions allow linear blending.

The biosequence analysis textbook of Durbin et al., from which this example is taken, some thought is given to how one should properly select the pseudo-counts. The proper exploitation of such "soft" knowledge makes all the difference in real-life machine learning problems.

Here is a summary of Bayesian approaches to parameter estimation for parametrized distributions:

1. Carefully choose a prior $P(\theta \mid M)$ which reflects your a-priori expert belief about how distributions θ should be distributed. If you don't know much beforehand, $P(\theta \mid M)$ should be close to uniform; if you have strong preferences for particular θ, make $P(\theta \mid M)$ peak strongly around the preferred values.
2. Make your measurements and think of a proper type of distribution (here: polynomial distribution) to obtain $P(D \mid \theta, M)$.
3. Use Bayes formula to obtain the posterior distribution of distributions, $P(\theta \mid D, M)$.
4. Integrate over $P(\theta \mid D, M)$ to find the final posterior distribution $\hat{\theta}$.

Always do step 1 before making measurements! If your choice of the prior would be influenced by what you empirically observe, the Bayesian approach becomes thoroughly flawed!

3.5 Some general remarks on estimation theory

We have seen that even in a situation as simple as estimating the probabilities of 20 symbols from a sample, several estimators T: $(x_1, ..., x_n) \mapsto \hat{\theta}$ can be considered, which have each their pro's and con's. This situation is typical and has spurred the development of a complete subbranch of statistics, estimation theory. Estimation theory is concerned in defining general quality criteria for estimators, thus helping to compare the various estimators one might think of in a given situation. The field was pioneered by Sir Ronald Aylmer Fisher in the first half of the 20th century.

The general approach in estimation theory is to investigate the behavior of an estimator T as the number N of observations grows, that is, consider T as a sequence of related estimators T_n: $(x_1, ..., x_n) \mapsto \hat{\theta}$. Note that an estimator T_n is a random variable.

Let θ_0 denote the true distribution. Here are the most important quality criteria for estimators:

1. Unbiasedness. T is *unbiased* if for all n, $E[T_n] = \theta_0$. In our example, θ_i^{ML} was unbiased but θ_i^{PME} was not.

2. Asymptotic unbiasedness. T is *asymptotically unbiased* if $\lim_{n\to\infty} E[T_n] = \theta_0$. θ_i^{PME} is an example.

For the next quality criteria we need to consider a probability space (Ω, F, P) where each $\omega \in \Omega$ is an "experiment" in which we carry out an infinite sequence $x_1, x_2,... = X_1(\omega), X_2(\omega)...$ of measurements (so X_i is the random variable "carry out the ith measurement within such an experiment"). $T_n(\omega)$ is then $T_n(x_1, ..., x_n)$ for $\omega = x_1, x_2,....$Then we can define:

3. Strong consistency. T is *strongly consistent* if T_n converges to θ_0 *P-almost-surely*, that is, if(3.39)

(3.39) $$P(\lim_{n\to\infty} T_n(\omega) = \theta_0) = 1$$

θ_i^{ML} and θ_i^{PME} are strongly consistent (by the law of large numbers). Explanation: The strong law of large numbers isn't actually a law but a property of a sequence $X_1, X_2, ...$ of numerical random variables. Such a sequence obeys the strong law of large numbers if

(3.40) $$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} (X_i - E(X_i)) = 0 \quad P\text{-almost surely.}$$

It holds, for instance, if all X_i are integrable, independent, and identically distributed (a fundamental theorem of Kolmogorov). For θ_i^{ML}, we can use the law of large numbers to show (3.39) as follows. For a sample of size N, $\theta_{i,N}^{\mathrm{ML}}(\omega) = \frac{1}{N} \sum_{j=1}^{N} X_j(\omega)$, where $X_j(\omega) = 1$ if the j-th protein sequence in our sample has the i-th amino acid symbol in the location of interest, else $X_j(\omega) = 0$. The X_j are integrabel, independent, and identically distributed, so the strong law applies. The expectation of X_j is θ_i for all j. So we can conclude:

$$\begin{aligned} P(\lim_{N\to\infty} \theta_{i,N}^{ML}(\omega) = \theta_i) &= P(\lim_{N\to\infty} \frac{1}{N} \sum_{j=1}^{N} X_j(\omega) = \theta_i) \\ &= P(\lim_{N\to\infty} (\frac{1}{N} \sum_{j=1}^{N} X_j(\omega)) - \theta_i = 0) \\ &= P(\lim_{N\to\infty} (\frac{1}{N} \sum_{j=1}^{N} X_j(\omega) - \theta_i) = 0) \\ &= 1 \end{aligned}$$

where the last equality is justified by the strong law. For θ_i^{PME} a similar argument can be used.

A background note. As we have just seen, the strong law justifies that (and how, namely with probability 1) we may interpret the limit of relative frequency counts, $\lim_{N\to\infty} \frac{1}{N} \sum_{j=1}^{N} X_j(\omega)$,

as the probability of a discrete observation outcome. This is, on the one hand, the intuitive foundation of the frequentist approach to probability, but on the other hand, it is also a *derived* result within that theory. Therefore, the law of large numbers (especially the fundamental theorem of Kolmogorov) is a pillar in the frequentist theory of probability.

4. Weak consistency. *T* is *weakly consistent* if T_n converges to θ_0 *in probability*, that is, if

(3.41) for all $\varepsilon > 0$, $\lim_{n\to\infty} P(\{\omega \in \Omega \mid 7\ T_n(\omega) - \theta_0\ 7 > \varepsilon\}) = 0$.

Weak consistency follows from strong consistency, so our two estimators θ_i^{ML} and θ_i^{PME} are weakly consistent, too. Many estimators of great practical significance in machine learning have none of the properties 1. – 4. This is likely to happen if the estimator incorporates a nonlinear optimum finding subroutine, which for instance is the case in most neural network (widely used in pattern recognition) and hidden Markov model (widely used in speech recognition) based estimators.

5. Efficiency. These critiera 1. – 4. are all-or-none, that is, an estimator either has that property or has it not. Another kind of quality criterium asks for the relative *efficiency* of an estimator, that is, how efficiently it makes use of the information contained in a sample. The general idea is that an estimator T_n (which should be unbiased to start with) is efficient if it has small variance $\sigma^2(T_n)$, that is, if the estimates $\hat{\theta}$ returned by T_n are scattered narrowly around θ_0. An unbiased estimator S_n is more efficient than an unbiased estimator T_n, if $\sigma^2(S_n) > \sigma^2(T_n)$.

6. Sufficiency. (Here I roughly follow the book from Duda/Hart/Stork, Section 3.6). Yet another angle on judging the quality of estimators starts from the question whether the choice θ of parameters is appropriate in the first place. For our amino acid example (distribution of discrete symbols in classes) the set of class probabilities makes an obviously adequate set of parameters; when one wants to characterize a normal distribution, one chooses $\theta = (\mu, \sigma)$. But what about cases where one does not have a well-founded intuition about how one should characterize the unknown distribution with a few parameters? Assume that the unknown distribution would rightfully be described by parameters θ, but you don't know which kind of θ. This is a standard situation in practice, where you meet "wild" distributions that cannot be expected to be of any known, simple kind. So you devise of a vector **s** of parameters that you can estimate from data instead of θ, hoping that **s** contains all the relevant information about the underlying distribution. Such a set **s** of parameters that you estimate from data is called *a statistic*. Technically, a statistic is just some (possibly vector-valued) function $\mathbf{s} = \varphi(D)$. A statistic is called *sufficient* if indeed it contains all the relevant information about the underlying distribution, that is, about θ.

Intuitively, one would define **s** to be sufficient if

(3.42) $P(\theta \mid \mathbf{s}, D) = P(\theta \mid \mathbf{s})$,

that is, if **s** extracts from the data *D* all that is relevant for learning about θ. However, this would imply that θ is taken as a random variable, a perspective not common for "classical" statisticians, who therefore defined sufficiency in another way: a statistic **s** is said to be

sufficient for θ if $P(D \mid \theta, \mathbf{s})$ is independent of θ, that is, $P(D \mid \theta, \mathbf{s}) = P(D \mid \mathbf{s})$. The two ways of defining sufficiency are equivalent. To see this, first assume the classical definition $P(D \mid \theta, \mathbf{s}) = P(D \mid \mathbf{s})$. Use the Bayesian formula to spell out $P(\theta \mid \mathbf{s}, D)$ by

$$P(\theta \mid \mathbf{s}, D) = \frac{P(D \mid \mathbf{s}, \theta) P(\theta \mid \mathbf{s})}{P(D \mid \mathbf{s})}, \tag{3.43}$$

where the r.h.s. cancels to $P(\theta \mid \mathbf{s})$ with $P(D \mid \theta, \mathbf{s}) = P(D \mid \mathbf{s})$, yielding the Bayesian-style definition. Conversely, if you assume $P(\theta \mid \mathbf{s}, D) = P(\theta \mid \mathbf{s})$, you get $P(D \mid \theta, \mathbf{s}) = P(D \mid \mathbf{s})$ by a mirrored argument (you need the extra condition $P(\theta \mid \mathbf{s}) \neq 0$.).

A fundamental theorem characterizes sufficient statistics as those statistics **s**, where $P(D \mid \theta)$ can be factorized into a part that depends only on **s** and θ, and another part that depends only on *D*:

Theorem 3.1 (factorization theorem): A statistics **s** is sufficient for θ if there exist functions *g* and *h* such that $P(D \mid \theta) = g(\mathbf{s}, \theta)\, h(D)$.

For an intuitive grasp, here is a sloppy version of a proof for the ⇒ direction. Assume that **s** is sufficient, and formally write $P(D \mid \theta) = P(D, \mathbf{s} \mid \theta) = P(D \mid \mathbf{s}, \theta)\, P(\mathbf{s} \mid \theta)$, which is equal to $P(D \mid \mathbf{s})\, P(\mathbf{s} \mid \theta)$ by sufficiency of **s**. The first factor $P(D \mid \mathbf{s})$ is a function of *D*, namely, $P(D \mid \mathbf{s}) = P(D \mid \varphi(D))$. Put $h(D) = P(D \mid \mathbf{s})$. The second factor is the desired $g(\mathbf{s}, \theta) = P(\mathbf{s} \mid \theta)$. Thus, $P(D \mid \theta) = g(\mathbf{s}, \theta)\, h(D) = P(\mathbf{s} \mid \theta)\, P(D \mid \varphi(D))$.

A more detailed proof for the case of discrete distributions can be found in Duda/Hart/Stork (p. 104). The importance of the factorization theorem lies in the fact that when we want to check whether a statistic **s** is sufficient, we can restrict our analysis to the distribution $P(D \mid \theta)$ instead of having to deal with $P(D \mid \theta, \mathbf{s})$. If $D = (x_1, ..., x_n)$, and the individual measurements are statistically independent, $P(D \mid \theta)$ takes the simple form of

$$P(D \mid \theta) = \prod_{k=1}^{n} P(x_k \mid \theta). \tag{3.44}$$

Specifically, the factorization theorem teaches us that a sufficient statistic depends only on the probabilties $P(x_k \mid \theta)$ and not on (inaccessible) assumptions on a prior $P(\theta)$.

Exponential distributions. The factorization theorem and Eq. (3.44) can be applied particularly well if we are dealing with parametric probability distributions from the *exponential* familiy. This family includes most of the standard textbook distributions, for instance the normal, exponential, Poisson, Gamma, Beta, Bernoulli, binomial and multinomial distributions. Exponential distributions are characterized by a pdf of the form

$$p(\mathbf{x} \mid \theta) = \alpha(\mathbf{x}) \exp\left[\mathbf{a}(\theta) + \mathbf{b}(\theta)^{\mathsf{T}}\, \mathbf{c}(\mathbf{x})\right], \tag{3.45}$$

where **a**, **b**, **c** are linear functions.

For exponential distributions, one gets a sufficient statistic by

$$\mathbf{s} = \frac{1}{n}\sum_{k=1}^{n}\mathbf{c}(x_k), \tag{3.46}$$

and the two factorizing functions as

$$g(\mathbf{s},\theta) = \exp[n(\mathbf{a}(\theta) + \mathbf{b}(\theta)^T\mathbf{s})], \quad h(D) = \prod_{k=1}^{n}\alpha(x_k). \tag{3.47}$$

A table containing an overview of all these expressions for a dozen or so much-used distributions is shown in Duda/Storck/Hart p. 108-109.

Once one has a sufficient statistic $\mathbf{s} = \varphi(D)$ and the factorization functions, given a sample $\mathbf{x} = (x_1, ..., x_n)$ one can find the maximum likelihood estimator θ^{ML} through

$$\begin{aligned}\theta^{ML}(\mathbf{x}) &= \arg\max_\theta P(\mathbf{x}\,|\,\theta)\\ &= \arg\max_\theta g(\mathbf{s},\theta)h(\mathbf{x})\\ &= \arg\max_\theta g(\mathbf{s},\theta)\\ &= \theta'^{ML}(\mathbf{s}).\end{aligned} \tag{3.48}$$

For the various exponential distributions, the sufficient statistics and the factorization functions g are significantly simpler than the original formulae for the pdfs. Because very many distributions of practical relevance are exponential, the tools of sufficient statistics and factorization are of great practical importance.

4. Linear discrimination and single-layer neural networks

In this section we will treat a special case of two-class classification, namely, *linear discrimination*. Together with the maths we will introduce a particular conceptual / graphical notation, namely, cast the classification algorithm as a *neural network*. Linear discrimination is the basis for understanding more advanced techniques that we will treat later: adaptive Wiener filters, multilayer neural networks, support vector machines. I follow closely chapter 3 of Bishop's book.

4.1. Linear discrimination: two classes

Recall. Toward the end of Section 2 we introduced discriminant functions as monotonically increasing functions f

(4.1) $y_i(\mathbf{x}) = f(p(\mathbf{x} \mid C_i)\, P(C_i))$

of the class-conditional probability times the prior. For the case of binary discrimination we mentioned that one can introduce a single discrimination function

(4.2) $y(\mathbf{x}) = y_1(\mathbf{x}) - y_2(\mathbf{x})$

and decide that $\mathbf{x}$ falls into class 1 whenever $y(\mathbf{x}) > 0$. We remarked at the end of that section that it is sometimes easier to learn discriminant functions directly from the training data, than first estimate the class-conditional distributions first and construct the discriminant function from those distributions in the second place.

This is the approach we will take in this section: we forget (for a while) the connection of discriminant functions with distributions and start directly from a given functional form of the two-class discriminant function (4.2), namely, *linear discriminants* of the form

(4.3) $y(\mathbf{x}) = \mathbf{w}^{\mathsf{T}}\mathbf{x} + w_0,$

where $\mathbf{w}$ is a *weight* (column) vector and w_0 is a *bias* (this usage of the term "bias" is historical and not identical to the Bayesian notion of a bias we met in connection with the bias-variance dilemma). For the case of two and three-dimensional features $\mathbf{x} = (x_1, x_2)$ or $\mathbf{x} = (x_1, x_2, x_3)$, linear discriminants can be visualized as in Fig. 4.1. See figure caption for a geometrical interpretation.

In a *neural network* interpretation, a linear discriminant corresponds to a network with $M + 1$ input neurons, where M is the dimension of the features $\mathbf{x}$, and a single output neuron, where the output $y(\mathbf{x})$ of the discriminant is read from. The first input neuron x_0 receives constant input 1, the remaining input neurons receive input $\mathbf{x} = (x_1, ..., x_M)$. The "synaptic" *network weights* are $w_0, w_1, ..., w_M = (w_0, \mathbf{w}^{\mathsf{T}}) = \tilde{\mathbf{w}}^{\mathsf{T}}$. The output in this network is computed in the output neuron by summing up the inputs, weighted by the weights, which gives

(4.4) $y(\mathbf{x}) = \tilde{\mathbf{w}}^{\mathsf{T}}\tilde{\mathbf{x}} = w_0 + \mathbf{w}^{\mathsf{T}}\mathbf{x}.$

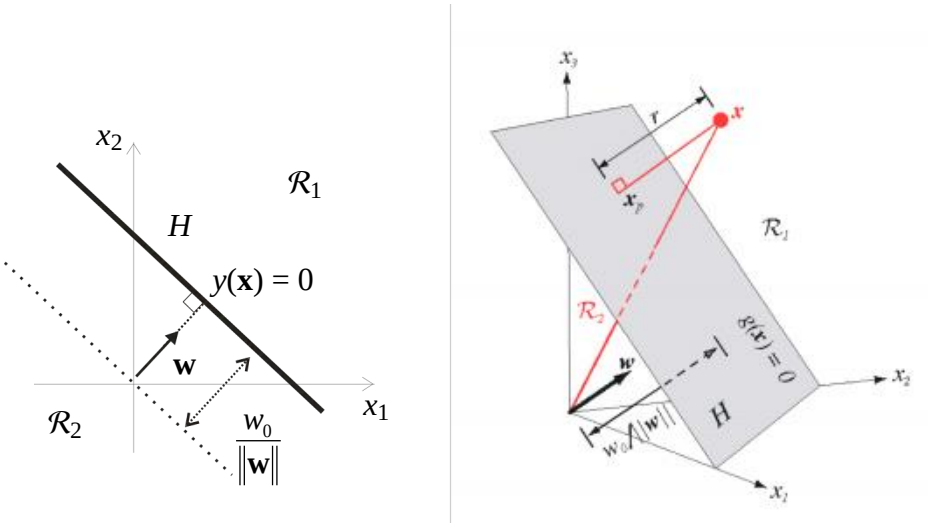


Fig. 4.1 Geometrical interpretation of two-class linear discriminant $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$ for two-dimensional (left) and three-dimensional features $\mathbf{x}$. A hyperplane H defined by $y(\mathbf{x}) = 0$ separates the feature space into two decision regions $\mathcal{R}_1$ and $\mathcal{R}_2$. The hyperplane has orientation perpendicular to $\mathbf{w}$ and distance $w_0 / \|\mathbf{w}\|$ to the origin. (Left figure after the book from Bishop, right figure was taken from the online supplements to the book of Duda/Hart/Stork).

The notation $y(\mathbf{x}) = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$ is often more convenient than (4.3). The network representation of the discriminant is shown in Fig. 4.2.

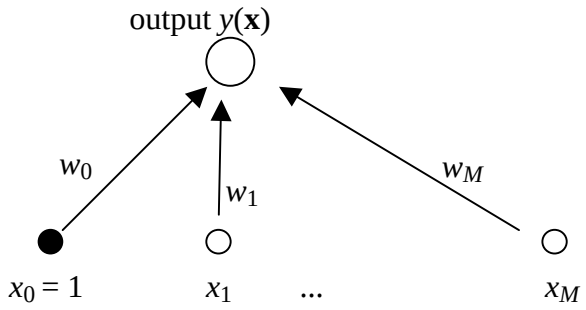


Figure 4.2: A network representation of a two-class linear discriminant.

4.2 Linear discrimination: several classes

The two-class case can be extended to n classes by introducing linear discriminant functions y_k for each class:

(4.5) $\quad y_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + w_{k0},$

assigning an input pattern $\mathbf{x}$ to class k if $y_k(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq k$. Because $y_k(\mathbf{x}) > y_j(\mathbf{x})$ if $y_k(\mathbf{x}) - y_j(\mathbf{x}) > 0$, the decision boundary between classes k and j are given by

(4.6) $y_k(\mathbf{x}) - y_j(\mathbf{x}) = (\mathbf{w}_k - \mathbf{w}_j)^\mathsf{T}\mathbf{x} + (w_{k0} - w_{j0}) = 0.$

The network representation of (4.5) is sketched in Fig. 4.3:

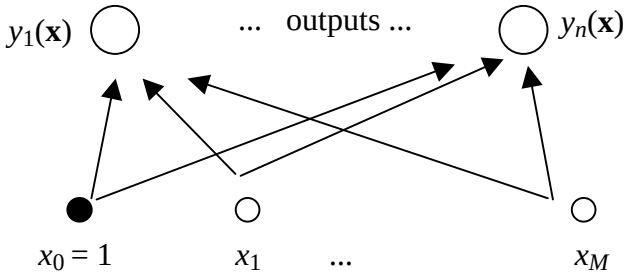


Figure 4.3: Representation of multiple linear discriminant functions.

As before in (4.4), we cover the bias by an additional constant input x_i of unit size and thus may re-write (4.5) as

(4.7) $y_k(\mathbf{x}) = \sum_{i=0}^{M} w_{ki}\ x_i\ .$

The decision regions are now regions in $\mathbb{R}^{M+1}$. They have linear hyperplanes as boundaries, as can be seen from (4.6). Furthermore, the decision regions are connected and convex. To see this, consider two points $\mathbf{x}^A$ and $\mathbf{x}^B$, which both lie in region $\mathcal{R}_k$. Any point $\hat{\mathbf{x}}$ that lies on a line between $\mathbf{x}^A$ and $\mathbf{x}^B$ can be written as $\hat{\mathbf{x}} = \alpha\mathbf{x}^A + (1-\alpha)\mathbf{x}^B$ for some $0 \le \alpha \le 1$. From the linearity of the discriminant functions it follows that $y_k(\hat{\mathbf{x}}) > y_j(\hat{\mathbf{x}})$ for all $j \ne k$. Therefore all $\hat{\mathbf{x}}$ between $\mathbf{x}^A$ and $\mathbf{x}^B$ are in class $\mathcal{R}_k$, too. This is schematically shown in Fig. 4.4.

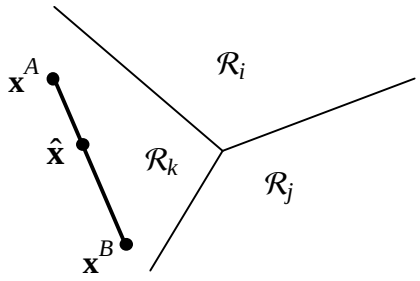


Figure 4.4 Convexity and connectedness of linear decision regions.

4.3 Computing logistic discriminants for Gaussian class-conditional distributions

We have not yet addressed the obvious question of how the weights $\mathbf{w}$ and w_0 can be computed in order to yield optimal classifications. In special (and important) cases, explicit solutions can be given. In this subsection we address the case where the class-conditional probabilities $p(\mathbf{x} \mid C_i)$ are normal distributed. The closed-form solution in this case also

reveals a connection of the discriminant function with the underlying class-conditional probabilities $p(\mathbf{x} \mid C_i)$.

We have seen earlier in this lecture (end of Section 2) that decision regions are not affected by wrapping the outcome of a discrimination function with a monotonically increasing function *f*. Instead of using $y_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + w_{k0}$, as in (4.5), in an *n*-class classification network as shown in Fig. 4.3 we may just as well use

(4.8) $$y_k(\mathbf{x}) = f(\mathbf{w}_k^T\mathbf{x} + w_{k0}).$$

In the neural network metaphor, *f* describes how the output of the output neuron is re-shaped after the simple summation of the incoming signals. Because in neural networks the output values of neurons are called activations, *f* is called an *activation function* in this context. Since the decision boundaries generated by (4.8) are still linear, this setup is still regarded as a case of linear discrimination. The case where *f* is the identity is also referred to as the *linear activation function.* We will later see that linear learning techniques that estimate weights for linear activation functions can be adapted to likewise linear learning techniques that estimate weights for nonlinear activation functions.

A common choice for a nonlinear *f* is the *logistic* function

(4.9) $$f(a) = \frac{1}{1+e^{-a}}.$$

Figure 4.5 shows a plot.

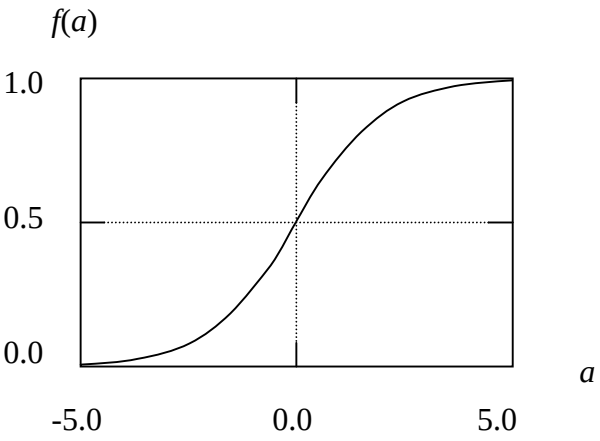


Figure 4.5 The logistic function.

The logistic function has a value range between 0 and 1; for large arguments it "saturates" at 1, which is an abstract version of the biological fact that biological neurons cannot become activated (in the sense of average firing frequency) above a certain saturation value. Thus, there is a biological motivation for using the logistic. The logistic function is S-shaped ("sigmoid"). There is another sigmoid function in common usage in neural networks, namely *f* = tanh. This function ranges from –1 to +1 and has no good biological justification (what would be negative activations?).

However, there is also a mathematical reason for using the logistic function. Consider a two-class problem where the class-conditional densities are given by Gaussian distributions with equal covariance matrices $\Sigma_1 = \Sigma_2 = \Sigma$:

$$(4.10) \qquad p(\mathbf{x} \mid C_k) = \frac{1}{(2\pi)^{M/2}\ \det(\Sigma)^{1/2}}\ \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^{\mathsf{T}}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right),$$

where $k = 1, 2$. (A more detailed discussion of multidimensional Gaussians will be given in Section 4.4.2).

Using Bayes' theorem, we find that the posterior probability of membership in class C_1 is given by

$$\begin{aligned} p(C_1 \mid \mathbf{x}) &= \frac{p(\mathbf{x} \mid C_1)P(C_1)}{p(\mathbf{x} \mid C_1)P(C_1) + p(\mathbf{x} \mid C_2)P(C_2)} \\ &= \frac{1}{1+e^{-a}} \qquad (4.11) \\ &= f(a), \end{aligned}$$

where

$$(4.12) \qquad a = \ln\frac{p(\mathbf{x} \mid C_1)P(C_1)}{p(\mathbf{x} \mid C_2)P(C_2)}.$$

If we substitute (4.10) into (4.12) we obtain [by multiplying out the expressions $-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^{\mathsf{T}}\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)$]

$$(4.13) \qquad a = \mathbf{w}^{\mathsf{T}}\mathbf{x} + w_0,$$

where

$$\begin{aligned} \mathbf{w} &= \Sigma^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \\ w_0 &= -\frac{1}{2}\boldsymbol{\mu}_1{}^{\mathsf{T}}\Sigma^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2{}^{\mathsf{T}}\Sigma^{-1}\boldsymbol{\mu}_2 + \ln\frac{P(C_1)}{P(C_2)}. \end{aligned} \qquad (4.14)$$

Thus we see that $p(C_1 \mid \mathbf{x}) = f(a) = f(\mathbf{w}^{\mathsf{T}}\mathbf{x} + w_0)$, that is, the output of the first output neuron in a two-class network (set up with two output neurons, like in the *n*-class network in Fig. 4.3) can be interpreted directly as a posterior probability for class 1 if we use the weights given in (4.14). As we have seen in Section 2.9, this gives us optimal decisions in the sense of minimizing the probability of misclassifications.

Supplementary background information. Our findings can be generalized in several directions.

First, using a logistic activation function together with the right weight vector gives us correct posterior pdfs not only for class-conditional probabilities that are normal distributed, but also

for any other exponential distribution. This is described in Section 6.7.1 in Bishop's book; a special case (Bernoulli distributed class-conditional probabilities) is detailed out in Section 3.1.4 in Bishop's book.

Second, our finding can be generalized from the two-class classification task to *n*-class classification. Let the summed input to the *k*-th output unit be given by

(4.15) $a_k = \mathbf{w}_k^T \mathbf{x} + w_{k0}$.

Define the activation of the *k*-th output neuron by

(4.16) $y_k(\mathbf{x}) = \frac{e^{a_k}}{\sum_{i=1}^{n} e^{a_i}}$.

This is known as the *softmax* "activation function". Note that the *k*-th output neuron must "know" the inputs to the other output neurons to compute its own output, so the simple network metaphor shown in Figure 4.3 does not really hold any longer, and softmax is actually not a single-unit activation function. The softmax activation function enforces that the sum of activations of all output units is 1, and each is nonnegative, so the vector of all outputs is a probability vector. Using softmax output activations, for class-conditional distributions from the exponential family one can derive closed-form solutions for the weights $\mathbf{w}_k$ and w_{k0} (as we did in Eqn. (4.14) for the two-class Gaussian case) such that the outputs become the posterior distributions (Section 6.9 in Bishop's book).

Third, when in the binary classification task we drop the assumption of equal covariance matrices, now having covariance matrices Σ_1 and Σ_2 for the two class-conditional distributions (4.10), we end up with a quadratic function

(4.17) $a = \mathbf{x}\, \mathbf{W}\, \mathbf{x} + \mathbf{w}^T \mathbf{x} + w_0$,

with different $\mathbf{w}$ and w_0 than before. The decision boundaries are then quadratic hyperplanes (hyperparaboloids).

Perceptrons. Historically, the first "neural networks" (not called like that then) for classification tasks used another kind of activation function, namely, binary threshold functions $f(a) = -1$ if $a < 0$, $f(a) = 1$ if $a \geq 0$. These networks were introduced by Rosenblatt in the early 60ies and named *Perceptrons*. Perceptrons were biologically inspired in a context of visual pattern classification from pixel images. Another characteristic of perceptrons is that they come with a particular type of feature extraction, that is, their input neurons correspond to a particular kind of features extracted from pixel images. Figure 4.6. (redrawn from Bishop's book) shows the setup of a perceptron. There exists a learning rule for perceptrons that incrementally adapts network weights for maximal discrimination rates; this rule can be proven to converge. Perceptrons are still rather popular.

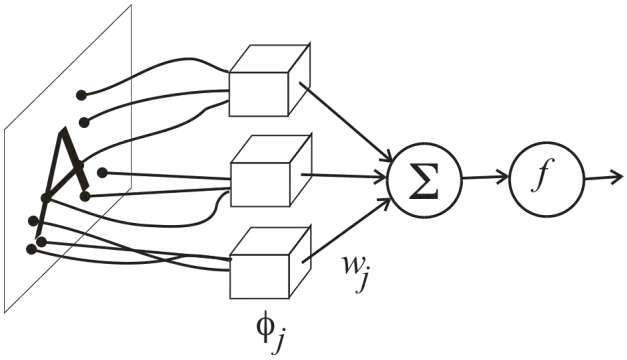


Figure 4.6: The perceptron's input neurons ϕ_j are patched to the input pattern by (random) links. They typically compute their outputs by a threshold function from the sum of the signals received through these links. Input neuron outputs are weighted, summed, and passed through another threshold function *f* whose output indicates whether the pattern belongs to class 1 or class 2 (binary classification).

4.4 The powers of, and maths behind, linear data transformations

So far we have only described linear discrimination networks and have seen that weights exist which yield optimal decisions, if the class-conditional distributions are from the exponential family. This is reassuring from a theoretical perspective, but does not help too much in practice, because strong assumptions about the form of the class-conditional distributions are needed to justify using the closed-form weight computations. In this subsection we will show another case, of greater practical relevance, where the network weights have closed-form solutions, without making assumptions about the form of the class-conditional distributions.

Before we embark on this topic, we will quickly refresh some linear algebra (Section 4.4.1) and facts about multi-dimensional normal distributions (Section 4.4.2). This material is adapted from Appendix A of the Bishop book and Appendix A.5.1 of Duda/Hart/Stork.

4.4.1 The eigenmagic of symmetric matrices

Definition. A matrix **A** is *symmetric* if its columns and rows permute, that is, $\mathbf{A} = \mathbf{A}^\mathsf{T}$.

Example. We will encounter symmetric matrices very often through covariance and correlation matrices. Given *n* numerical random variables $\mathbf{X} = (X_1, ..., X_n)^\mathsf{T}$, their *covariance matrix* $\Sigma = (\sigma_{ij})$ is given by

(4.18) $\sigma_{ij} = \mathrm{Cov}(X_i, X_j) = E[(X_i - E[X_i])\,(X_j - E[X_j])]$

and their *correlation matrix* $\mathbf{R} = (r_{ij})$ by

(4.19) $r_{ij} = E[X_i\, X_j]$, or equivalently, $\mathbf{R} = E[\mathbf{X}\mathbf{X}^\mathsf{T}]$

Inverse. The inverse of a symmetric matrix is symmetric.

A commutative property. For symmetric matrices **A** and vectors **u**, **v** we have

(4.20) $\mathbf{u}^\mathsf{T}\mathbf{A}\mathbf{v} = \mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{u}$.

Eigenvector equations of a matrix. The eigenvectors **u** of an $n \times n$ matrix **A** (not necessarily symmetric) by definition satisfy

(4.21) $\mathbf{A}\,\mathbf{u} = \lambda\mathbf{u}$,

where λ is an eigenvalue of **A**. In matrix notation, (4.21) can be rewritten as

(4.22) $(\mathbf{A} - \lambda\mathbf{I})\,\mathbf{u} = \mathbf{0}$,

where **I** is the $n \times n$ identity matrix and **0** the $n \times 1$ null vector. To prevent the trivial solution $\mathbf{u} = \mathbf{0}$, the matrix $\mathbf{A} - \lambda\mathbf{I}$ has to be singular, that is,

(4.23) $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$.

Remember that the determinant of an n by n matrix **M** is given by

(4.24) $$\det(\mathbf{M}) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} M_{i,\sigma_i},$$

where S_n is the set of all permutations of $\{1, ..., n\}$, and $\operatorname{sgn}(\sigma)$ is $+1$ if σ is even, and $\operatorname{sgn}(\sigma)$ is -1 if σ is odd. Equation (4.23) is the *characteristic equation* of the matrix **A**. When expanded into the form (4.24), it is an n-th order polynomial in the unknown λ. The roots of this polynomial, which may be called $\lambda_1, ..., \lambda_n$, are the eigenvalues of **A**. When a root λ_i has multiplicity 1 (that is, there is only a single root with this value), there exists a unique (up to scaling) eigevector **u** to that eigenvalue. If a root λ_i has multiplicity $k > 1$, the eigenvectors to that eigenvalue span a k'-dimensional subspace $V(\lambda_i)$ whose dimension is at most k. This is often stated as "the geometric multiplicity [i.e., $\dim(V(\lambda_i))$] of λ_i is at most as large as the algebraic multiplicity of λ_i".

Note that the roots of a polyonomial with real coefficients may be complex numbers. Thus, even when **M** is real, it may have complex eigenvalues and complex eigenvectors of the form $\mathbf{u} + i\mathbf{v}$. This does however not happen when **M** is symmetric:

Real eigenvalues. The eigenvalues of a real symmetric matrix **A** are real. Proof: Consider the eigenvalue equation $\mathbf{Au} = \lambda\mathbf{u}$. Premultiplication by $\bar{\mathbf{u}}^{\mathsf{T}}$ yields $\bar{\mathbf{u}}^{\mathsf{T}}\mathbf{Au} = \lambda\bar{\mathbf{u}}^{\mathsf{T}}\mathbf{u}$. By multiplying out $\bar{\mathbf{u}}^{\mathsf{T}}\mathbf{Au}$, it is easy to see that $\bar{\mathbf{u}}^{\mathsf{T}}\mathbf{Au}$ is real. Likewise, $\bar{\mathbf{u}}^{\mathsf{T}}\mathbf{u}$ is real. Therefore, λ must be real.

Real eigenvectors suffice. The eigenvectors of a real symmetric matrix may be chosen real. Proof: Consider $\mathbf{A}(\mathbf{u} + i\mathbf{v}) = \lambda(\mathbf{u} + i\mathbf{v})$. Since **A** and λ are real, we have $\mathbf{Au} = \lambda\mathbf{u}$ and $\mathbf{Av} = \lambda\mathbf{v}$. This means that the subspace $V(\lambda)$ in $\mathbb{C}^n$ is spanned by real eigenvectors of λ.

Orthonormal eigenvectors in symmetric matrices. Without proof I mention that an n-dimensional symmetric matrix **A** has n eigenvectors spanning $\mathbb{R}^n$. Furthermore, if **A** is real and symmetric, the eigenvectors can be chosen to be real and orthonormal (that is, orthogonal and of unit length). To see this, for real eigenvectors $\mathbf{u}_j$, $\mathbf{u}_k$ (with eigenvalues λ_j, λ_k) consider

(4.25) $\mathbf{u}_j^\mathsf{T} \mathbf{A}\, \mathbf{u}_k = \lambda_k\, \mathbf{u}_j^\mathsf{T} \mathbf{u}_k$ and $\mathbf{u}_k^\mathsf{T} \mathbf{A}\, \mathbf{u}_j = \lambda_j\, \mathbf{u}_k^\mathsf{T} \mathbf{u}_j$,

which follows from (4.21). Subtracting these two equations and exploiting (4.20) we get $0 = \mathbf{u}_j^\mathsf{T} \mathbf{A}\, \mathbf{u}_k - \mathbf{u}_k^\mathsf{T} \mathbf{A}\, \mathbf{u}_j = (\lambda_k - \lambda_j)\, \mathbf{u}_k^\mathsf{T} \mathbf{u}_j$, from which it follows that for $\lambda_k \neq \lambda_j$, $\mathbf{u}_k$ and $\mathbf{u}_j$ must be orthogonal. If $\lambda_k = \lambda_j$, any linear combination of $\mathbf{u}_k$ and $\mathbf{u}_j$ is also an eigenvector; this can be used to create a selection of pairwise orthogonal eigenvectors for the eigenvalue λ_k. This gives us an altogether orthogonal set of eigenvectors, which can be then each normalized to unit length to yield orthonormal eigenvectors, that is,

(4.26) $\mathbf{u}_k^\mathsf{T} \mathbf{u}_j = \delta_{kj}$.

This has two immediate consequences. First, the matrix $\mathbf{U} = [\mathbf{u}_1 \ldots \mathbf{u}_n]$ may be assumed to be orthonormal, that is,

(4.27) $\mathbf{U}^\mathsf{T}\mathbf{U} = \mathbf{U}\mathbf{U}^\mathsf{T} = \mathbf{I}$, or equivalently, $\mathbf{U}^{-1} = \mathbf{U}^\mathsf{T}$.

Second, the eigenvalues can be obtained from the eigenvectors by

(4.28) $\mathbf{u}_k^\mathsf{T} \mathbf{A}\, \mathbf{u}_k = \lambda_k$,

which follows from (4.21) and (4.26).

Geometric interpretation. Considering that symmetric real matrices have orthonormal eigenvectors with real-valued eigenvalues, we see that geometrically, such a matrix describes a linear transformation of $\mathbb{R}^n$ which shrinks/expands the orthonormal coordinate system given by the eigenvectors by the amounts given by the eigenvalues. If all eigenvalues are nonnegative, the unit sphere in $\mathbb{R}^n$ is mapped to an ellipsoid whose central axes point in the directions of the $\mathbf{u}_k$ and have length λ_k:

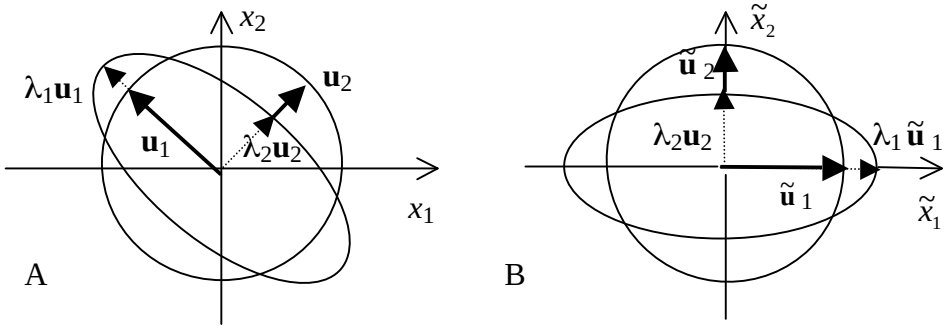


Figure 4.7: A. Effects of a symmetric matrix **A** (with non-negative eigenvalues) transforming the unit sphere in $\mathbb{R}^2$. B. Effect of applying $\mathbf{\Lambda}$ in the transformed coordinates $\tilde{\mathbf{x}} = \mathbf{U}^\mathsf{T}\mathbf{x}$.

Eigensystem of $\mathbf{A}^{-1}$. If we multiply (4.21) by $\mathbf{A}^{-1}$ we find $\mathbf{A}^{-1}\, \mathbf{u}_k = \lambda_k^{-1}\mathbf{u}_k$, that is, the eigenvectors of $\mathbf{A}^{-1}$ are the same as the eigenvectors of **A**, and the eigenvalues of $\mathbf{A}^{-1}$ are the λ_k^{-1}. Geometrically this means "undoing" the expansion/shrinking in the direction of the eigenvectors.

Diagonalization and coordinate transformation. A symmetric matrix **A** can be diagonalized using **U**. Left-multiplication of $\mathbf{AU} = [\lambda_1 \mathbf{u}_1 \ldots \lambda_n \mathbf{u}_n]$ by $\mathbf{U}^\mathsf{T}$, observing (4.26) yields

$$\mathbf{U}^\mathsf{T}\mathbf{A}\mathbf{U} = \mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}. \tag{4.29}$$

Left-multiplication by $\mathbf{U}^\mathsf{T}$ maps eigenvectors $\mathbf{u}_k$ on the unit vectors of $\mathbb{R}^n$, and more generally any $\mathbf{x} \in \mathbb{R}^n$ on

$$\tilde{\mathbf{x}} = \mathbf{U}^\mathsf{T}\mathbf{x}. \tag{4.30}$$

With respect to the new coordinate system, $\mathbf{\Lambda}$ acts like **A** in $\mathbb{R}^n$ (see Figure 4.7 B). The effect of $\mathbf{U}^\mathsf{T}$ is a rigid rotation of $\mathbb{R}^n$.

Positive definiteness. A matrix **A** is *positive* [*semi-*]*definite* if $\mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} > 0$ [$\mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} \geq 0$, respectively] for all nonzero vectors **v**. Correlation [and hence, covariance] matrices **R** are positive semidefinite. To see this, we use $\mathbf{R} = E[\mathbf{X}\mathbf{X}^\mathsf{T}]$ from (4.19) and get

$$\mathbf{v}^T\mathbf{R}\mathbf{v} = \mathbf{v}^T E[\mathbf{X}\mathbf{X}^T]\mathbf{v} = E[\mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v}] = E[\left\|\mathbf{v}^T\mathbf{X}\right\|^2] \geq 0. \tag{4.31}$$

All eigenvalues of a positive [semi-]definite matrix are positive [nonnegative]. This follows from (4.28).

General quadratic forms. Consider a quadratic function F on $\mathbb{R}^n$ of the form

$$F(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}, \tag{4.32}$$

where $\mathbf{A} = (a_{ij})$ is any real matrix. Replacing **A** by the symmetric matrix $\mathbf{A}' = (a'_{ij}) = ((a_{ij} - a_{ji})/2)$ doesn't change F, so without loss of generality **A** can be assumed to be symmetric. F can be computed using the matrix **U** of orthogonal eigenvectors of **A**, as follows:

$$\begin{aligned} F(\mathbf{x}) &= \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} \\ &= \mathbf{x}^\mathsf{T}\mathbf{U}\mathbf{U}^\mathsf{T}\mathbf{A}\mathbf{U}\mathbf{U}^\mathsf{T}\mathbf{x} \\ &= \mathbf{x}^\mathsf{T}\mathbf{U}\mathbf{\Lambda}\mathbf{U}^\mathsf{T}\mathbf{x} \\ &= \tilde{\mathbf{x}}^\mathsf{T}\mathbf{\Lambda}\tilde{\mathbf{x}} \\ &= \sum_{i=1}^{n} \lambda_i \tilde{x}_i^2, \end{aligned} \tag{4.33}$$

where we used (4.26), (4.29) and (4.30). Elementary geometry tells us that the surfaces of constant $F(\mathbf{x})$ are hyperellipsoids, with principal axes having lengths proportional to $\lambda_k^{-1/2}$.

Notice that in a symmetric matrix **A** need not be positive definite or semi-definite. (4)(4.33) shows that a symmtetric matrix **A** is positive (semi-)definite iff all its eigenvalues are positive (non-negative). All symmetric matrices map the unit sphere to an ellipsiod; when **A** has negative eigenvalues, the mapping includes a *mirroring* along the associated principal directions; when **A** has zero eigenvalues, the ellipsiod is degenerate (squashed to zero thickness) in the corresponding directions.

Multi-dimensional normal distributions. Now we are equipped to understand the nature of multi-dimensional normal distributions. Consider a vector-valued random variable $\mathbf{X}(\omega) = (X_1(\omega), ..., X_n(\omega))^\mathsf{T}$. If the outcome of the i-th measurement X_i can be considered as an additive effect of many independent physical causes – a not too unrealistic asssumption in many cases – then **X** will be (approximately) distributed according to the n-dimensional normal distribution (this is a rough statement of the *central limit theorem*). Its pdf is given by

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}\det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)\right), \tag{4.34}$$

where the mean $\boldsymbol{\mu}$ is the (vector) mean $E[(X_1, ..., X_n)^\mathsf{T}]$ and Σ is the $n \times n$ covariance matrix of the random variables $X_1, ..., X_n$. The prefactor ensures that this pdf integrates to unity; we will ignore it in our discussion. From our discussion in 4.4.1. we know that the argument of the exponential is a quadratic form on $\mathbb{R}^n$, in the *centered* coordinates $\mathbf{x} - \boldsymbol{\mu}$. Therefore, the constant level lines of $p(\mathbf{x})$ are hyperellipsoids with principal axes having lengths proportional to $\lambda_k^{1/2}$, where λ_k are the eigenvalues of the covariance matrix Σ. These hyperellipsoids are centered on $\boldsymbol{\mu}$. Furthermore, the principal axes fall in the directions of orthonormal eigenvectors $\mathbf{u}_k$ of Σ. Figure 4.8 shows some level curves of a two-dimensional Gaussian with a nondiagonal Σ.

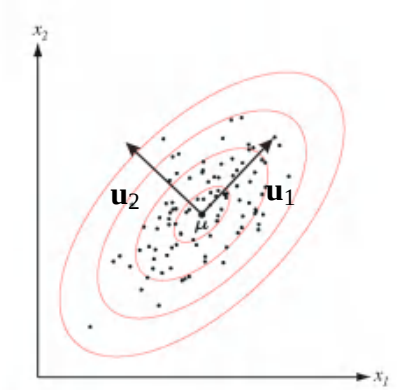


Figure 4.8: A two-dimensional Gaussian with a nondiagonal Σ. Samples lie in a cloud centered on $\boldsymbol{\mu}$.

Karhunen-Loéve transform. If we transform the vector of the centered random variable $\mathbf{X}_0 = (X_1 - E[X_1], ..., X_n - E[X_n])^\mathsf{T}$ by premultiplication with $\mathbf{U}^\mathsf{T}$ (compare (4.30)) we get a new random variable $\mathbf{X}' = (X_1', ..., X_n')^\mathsf{T}$:

$$\mathbf{X}' = \mathbf{U}^\mathsf{T} \mathbf{X}_0, \tag{4.35}$$

whose components X_k' are pairwise uncorrelated. This can be seen as follows:

$$
\begin{aligned}
E[\mathbf{X}'\mathbf{X}'^{\mathsf{T}}] &= \mathbf{U}^{\mathsf{T}} E[\mathbf{X}_0 \mathbf{X}_0^{\mathsf{T}}]\mathbf{U} \\
&= \mathbf{U}^{\mathsf{T}}\Sigma\mathbf{U} \\
&= \mathbf{U}^{\mathsf{T}}\mathbf{U}\Lambda\mathbf{U}^{\mathsf{T}}\mathbf{U} \\
&= \Lambda,
\end{aligned}
\tag{4.36}
$$

where $\mathbf{\Lambda}$ is the diagonal matrix with the eigenvalues of Σ and we observed $\Sigma = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathsf{T}}$, which follows from (4.29). Because $\mathbf{\Lambda}$ is diagonal form, $E[X_i' X_j'] = 0$ for $i \neq j$. The transformation (4.35) is not constrained to normal distributed $\mathbf{X}$, but can generally be used to transform n random variables $\mathbf{X}$ into another set of n random *uncorrelated* variables, by first normalizing $\mathbf{X}$ to zero mean variables $\mathbf{X}_0$, then using the covariance matrix Σ and its eigenvector matrix $\mathbf{U}$ to apply (4.35). This transformation is called the *Karhunen-Loéve transform* in the area of signal processing; in other areas it has no special name. It is used in many techniques of signal processing and machine learning; often filtering or learning techniques work better when the original observations / inputs $\mathbf{X}$ are first *decorrelated* in this way.

Whitening: A very practical routine for data preprocessing. After you have decorrelated a centered random variable $\mathbf{X}_0 = (X_1 - E[X_1], ..., X_n - E[X_n])^{\mathsf{T}}$ via (4.35), obtaining $\mathbf{X}'$, you can carry out an additional step of *normalizing* the n signals in $\mathbf{X}_0$ to unit variance, by computing

$$
\mathbf{X}'' = \begin{pmatrix} \lambda_1^{-1/2} & & \\ & \ddots & \\ & & \lambda_n^{-1/2} \end{pmatrix} \mathbf{X}' = \Lambda^{-1/2}\,\mathbf{X}',
\tag{4.37}
$$

where $\Lambda^{-1/2}$ is the diagonal matrix of the inverse square roots of the eigenvalues of Σ. This operation (4.37) scales the decorrelated signals $\mathbf{X}'$ such that they reach unit variance, which can be checked as follows:

$$
E[\mathbf{X}''\mathbf{X}''^{T}] = E[\Lambda^{-1/2}\mathbf{X}'\mathbf{X}'^{T}\,\Lambda^{-1/2}] = \Lambda^{-1/2}E[\mathbf{X}'\mathbf{X}'^{T}]\Lambda^{-1/2} = \Lambda^{-1/2}\Lambda\Lambda^{-1/2} = \mathbf{I}.
$$

The overall transformation from some n-dimensional random variable $\mathbf{X}$, through its centered version $\mathbf{X}_0$ and decorrelation and normalization, up to $\mathbf{X}''$ is called *whitening*, sometimes also *sphering*.

Beware of the difference between true and estimated Σ. In machine learning applications one often starts from a sample of observations $\mathbf{x}^k = (x^k{}_1, ..., x^k{}_n)^{\mathsf{T}}$, where $k = 1, ..., N$. These observations are typically registered row-wise in a data collection matrix $\mathbf{M} = (x^i{}_j)_{i=1,...,N;\, j=1...n}$. From $\mathbf{M}$ one can compute an *estimate* $\hat{\Sigma} = 1/N\ \mathbf{M}_0^T\mathbf{M}_0$ of the true correlation matrix Σ of the centered random variable $\mathbf{X}_0 = (X_1 - E[X_1], ..., X_n - E[X_n])^{\mathsf{T}}$ which gave rise to the samples $\mathbf{x}^k$. Here $\mathbf{M}_0$ is obtained from $\mathbf{M}$ by subtracting the column mean vector $\mu = 1/N\ \mathbf{1}_N{}^{\mathsf{T}}\ \mathbf{M}$ from each row in $\mathbf{M}$; $\mathbf{1}_N$ denotes the vector of N ones. Using this estimate $\hat{\Sigma}$ and its eigenvectors and eigenvalues $\hat{\mathbf{U}}$ and $\hat{\Lambda}$, one can orthogonalize the columns of $\mathbf{M}$ by putting

$$
\mathbf{M}' = \mathbf{M}\ \hat{\mathbf{U}},
\tag{4.38}
$$

and if one wishes, further normalize the columns of **M**' to unit norm by putting

(4.39) $\mathbf{M}'' = \mathbf{M}' \hat{\Lambda}^{-1/2} = \mathbf{M} \hat{\mathbf{U}} \hat{\Lambda}^{-1/2}$.

It is important to stay aware of the fact that this operation only decorrelates / normalizes the particular data set that you have in **M**; if you would continue to use the estimates $\hat{\Sigma}$ etc. which you obtained from **M** on further (test) data not contained in **M**, say in a data collection matrix **K**, then the columns of **K** would not be perfectly decorrelated / normalized by $\hat{\mathbf{U}}$ and $\hat{\Lambda}$.

Principal component analysis. Principal component analysis (PCA) is another close relative in this family of basic algebraic data manipulations. Assume again that you are handling data which are obtained from a centered random variable $\mathbf{X}_0 = (X_1^{\text{raw}} - E[X_1], ..., X_n^{\text{raw}} - E[X_n])^{\mathsf{T}}$, obtained from the raw data $\mathbf{X}^{\text{raw}}$ by subtracting the expectation $E[\mathbf{X}] =: \boldsymbol{\mu}$, and that you know the covariance matrix Σ with its associated eigenvectors **U** and eigenvalues Λ (or know at least estimates thereof). Assume furthermore that the eigenvalues are ordered to be monotonically decreasing, i.e. $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$. This means that the variance of the *n*-dimensional signal $\mathbf{X}_0$ is greatest in the direction of $\mathbf{u}_1$, then the next greatest orthogonal signal direction is aligned with $\mathbf{u}_2$, etc. See Figure 4.8 for a graphical impression (note that we do not require $\mathbf{X}_0$ to be normal distributed; PCA does not need this assumption). The eigenvectors $\mathbf{u}_i$ are also called the *principal components*[5] of the distribution $P_{\mathbf{X}0}$.

Now let $\mathbf{X}_0(\omega) = \mathbf{x} = (x_1, ..., x_n)$ be some data point. Call the projection of **x** on $\mathbf{u}_i$, i.e. the inner product $\mathbf{x}^{\mathsf{T}} \mathbf{u}_i$, the *loading* of **x** on the *i*-th principal component. Then it holds that

(4.40) $\mathbf{x} = \sum_{i=1}^{n} (\mathbf{x}^T \mathbf{u}_i) \mathbf{u}_i$, or respectively $\mathbf{x}^{\text{raw}} = \mu + \sum_{i=1}^{n} (\mathbf{x}^T \mathbf{u}_i) \mathbf{u}_i$,

which is just a way to state that **x** (or $\mathbf{x}^{\text{raw}}$, respectively) can be expressed in the orthonormal ($\boldsymbol{\mu}$-shifted, respectively) coordinate system of the $\mathbf{u}_i$. Now furthermore assume that *n* is big – say, in the order of hundreds or even tens of thousands; this situation easily occurs e.g. when **x** is a pixel value vector of an image. Now consider what error in accuracy of representing **x** is incurred if we use only a first few *k* of the principal components to reconstruct **x**, i.e. if we consider

(4.41) $\tilde{\mathbf{x}} = \sum_{i=1}^{k} (\mathbf{x}^T \mathbf{u}_i) \mathbf{u}_i \quad or \quad \tilde{\mathbf{x}}^{\text{raw}} = \mu + \sum_{i=1}^{k} (\mathbf{x}^T \mathbf{u}_i) \mathbf{u}_i \quad (k << n)$.

The expected squared norm error between **x** and $\tilde{\mathbf{x}}$ is obtained by

[5] beware of the wrong spelling "principle components"!

$$E[\|\tilde{\mathbf{X}} - \mathbf{X}\|^2] = E[\left\|\sum_{i=k+1}^{n}(\mathbf{X}^T\mathbf{u}_i)\mathbf{u}_i\right\|^2] = E[\sum_{i=k+1}^{n}\|(\mathbf{X}^T\mathbf{u}_i)\mathbf{u}_i\|^2]$$

$$= E[\sum_{i=k+1}^{n}(\mathbf{X}^T\mathbf{u}_i)^2] = \sum_{i=k+1}^{n}E[(\mathbf{X}^T\mathbf{u}_i)^2]$$

(4.42)
$$= \sum_{i=k+1}^{n}E[\mathbf{u}_i{}^T\mathbf{X}\mathbf{X}^T\mathbf{u}_i] = \sum_{i=k+1}^{n}\mathbf{u}_i{}^T\Sigma\mathbf{u}_i = \sum_{i=k+1}^{n}\mathbf{u}_i{}^T\lambda_i\mathbf{u}_i$$

$$= \sum_{i=k+1}^{n}\lambda_i$$

It is a general observation about real-world data **x** that the eigenvalues of the associated covariance matrix decrease very quickly (often roughly exponentially), so the reconstruction error (4.42) will likely be small even when only a few leading principal components are used.

PCA has very many applications. For our purposes, the most important ones are data compression and preprocessing. Instead of storing a data point **x** in its raw form, i.e. as a high-dimensional vector, one can store instead only a few loadings of **x** on a small number of leading principal components. Then, an approximate version $\tilde{\mathbf{x}}$ (or $\tilde{\mathbf{x}}^{\text{raw}}$) of **x** can be reconstructed from these loadings via (4.41). The savings in storage space can be enormous. Furthermore, this is a simple yet powerful way to **escape from the curse of dimensionality**. To break away from cursed high-dimensional raw data, use as a low-dim feature vector the loads of each raw data sample on some leading principal components. For splendid examples, check out Google images for "eigenfaces reconstruction" (for instance, http://www.cse.iitk.ac.in/users/amit/courses/768/00/vamsi/ links to a nice student project which also points out problems).

4.4.2 Generalized linear discriminants

Now we introduce a type of linear discrimination networks whose weights we will learn from data, using a closed-form computation.

We will consider networks for *n*-class discrimination with linear output activation functions. However, we will use a more general form for such networks, namely, allow that the input patterns **x** are pre-processed by bank of *M* preprocessing filters ϕ_j, which may be nonlinear. That is, there are *M* input neurons which may perform arbitrary filter functions on the patterns **x**. One example of such a *generalized linear discriminant* is the Perceptron [except for its nonlinear output activation function], where the input neurons each first pick some pixels and then do some thresholding. The general form of such networks is

(4.43) $$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x}) + w_{k0}\,,$$

where $k = 1, ..., n$ is the index for the classes to be discriminated (= number of output neurons), and $j = 1, ..., M$ is the index for the input filters (= number of input neurons). Again, we may wish to represent the contribution of bias w_{k0} by a constant dummy input $\phi_0(\mathbf{x}) \equiv 1$, which would give us the following variant of (4.43):

(4.44) $$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x}).$$

Example: Radial basis function networks. A very popular type of such networks is *radial basis function networks* (RBF networks). If $\dim(\mathbf{x}) = d$, each filter ϕ_j is a symmetric ("radial"), typically unimodal function on $\mathbb{R}^d$ with center $\boldsymbol{\mu}_j$. Gaussian density functions are a typical choice. For Gaussians, the output $\phi_j(\mathbf{x})$ of filter ϕ_j ($j > 0$) is

(4.45) $$\phi_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right).$$

Note that (i) we do not normalize the Gaussian function here to integral 1, and (ii) although we have an d-dimensional Gaussian, we do not have to care about a covariance matrix Σ because we restrict ourselves to radially symmetric Gaussians.

RBF networks offer the possibility to place many fine-grained filters ϕ_j into regions of the input space X where we need a fine-tuned discrimination, and to be more generous in "less interesting" regions where we plant only a few broad filters. Figure 4.9. shows an example where X is one-dimensional and where we want a high discriminiation precision around the origin and around 1.

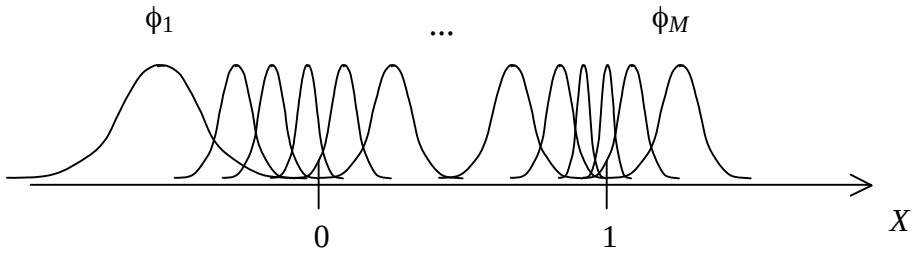


Figure 4.9: Radial basis functions example.

Two background notes:

Remark 1: The performance of RBF networks obviously depends on the proper sizing and placement of the basis functions ϕ_j. These are often optimized by *unsupervised* training schemes in a data-driven way. In Section 5 we will introduce such an algorithm that is often used with RBF networks.

Remark 2: *Any* desired input-output mapping F_k from X to the output unit y_k can be achieved with perfect precision with networks of the kind specified by Eq. (4.43). This is trivially clear because you may just use $M = n$ and $\phi_j = F_k$ and $w_{kj} = \delta_{kj}$. However, more interesting results state that any desired input-output mapping F_k can be approximated arbitrarily well with radial basis functions of a given simple class, for instance Gaussians. The art of designing RBF networks is to achieve good performance with as few as possible basis filters – because the fewer filters you have, the fewer training data points you need for estimating the network weights (another instance of the bias-variance dilemma!).

Let's return to the general equation (4.43). If such a network would be working perfectly, on input $\mathbf{x}_i \in C_k$ it would return an output vector $t_i = (0 \ldots 0\ 1\ 0 \ldots 0)^\mathsf{T}$ of length n, with a single 1 at position k. One natural way to specify a good set of network weights $\mathbf{W} = (w_{kj})_{k=1,\ldots,n;\ j=0,\ldots,M}$ is to demand that the squared error

$$SE_{\text{train}}(\mathbf{W}) = \frac{1}{2}\sum_{i=1}^{N}\left\|\mathbf{y}(\mathbf{x}_i;\mathbf{W}) - t_i\right\|^2 \tag{4.46}$$

is minimal, where i is the index for training patterns $\mathbf{x}_i$, and $\mathbf{y}(\mathbf{x}_i;\mathbf{W})$ is the output vector of a network with weights $\mathbf{W}$ on input $\mathbf{x}_i$. Formally, we want to find network weights $\mathbf{W}_{\text{opt}}$ such that

$$\mathbf{W}_{\text{opt}} = \operatorname*{argmin}_{\mathbf{W}} SE_{\text{train}}(\mathbf{W})\,. \tag{4.47}$$

$\mathbf{W}_{\text{opt}}$ can be calculated analytically – given the general approximation property of Eq. (4.43) mentioned in remark 2 above, this is a piece of powerful good news.

But before we derive the solution, it is instructive to give a geometrical interpretation of the least-squares problem in a simple case (Fig. 4.10). Consider the case where we have only two filters ϕ_0, ϕ_1 ($M = 1$), a single output unit y ($n = 1$), and three training samples $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ ($N = 3$). The three values of ϕ_1 yield a three-dimensional vector $\boldsymbol{\phi}_1 = (\phi_1(\mathbf{x}_1)\ \phi_1(\mathbf{x}_2)\ \phi_1(\mathbf{x}_3))^\mathsf{T}$, the three target values t_i yield a likewise three-dimensional vector $\boldsymbol{t}$, and the three identical values of the dummy filter become $\boldsymbol{\phi}_0 = (1,1,1)^\mathsf{T}$. The network outputs $y(\mathbf{x}_i) = \sum_{j=0}^{1} w_j \phi_j(\mathbf{x}_i)$ become another three-dimensional vector $\boldsymbol{y}(\mathbf{W}) = (y(\mathbf{x}_1)\ y(\mathbf{x}_2)\ y(\mathbf{x}_3))^\mathsf{T}$, which of course depends on the weights w_j. The two vectors $\boldsymbol{\phi}_0$ and $\boldsymbol{\phi}_1$ span a 2-dimensional subspace S in $\mathbb{R}^3$. Now, the least squares solution for the weights w_j yields a vector of network outputs $\boldsymbol{y}$ which is the orthogonal projection of $\boldsymbol{t}$ on S. Thus, the least squares solution gives us the network output which has smallest Euclidean distance to the teacher data $\boldsymbol{t}$.

Why is this so? Observe that in our simple case, the error equation (4.46) can be rewritten as

$$\begin{aligned} SE_{\text{train}}(\mathbf{W}) &= \frac{1}{2}\sum_{i=1}^{N}\left\|\mathbf{y}(\mathbf{x}_i;\mathbf{W}) - t_i\right\|^2 = \frac{1}{2}\sum_{i=1}^{3}\left(\sum_{j=0}^{1} w_j \phi_j(\mathbf{x}_i) - t_i\right)^2 = \\ &= \frac{1}{2}\left\|\sum_{j=0}^{1} w_j \boldsymbol{\phi}_j - \boldsymbol{t}\right\|^2 = \frac{1}{2}\left\|\mathbf{y}(\mathbf{W}) - \boldsymbol{t}\right\|^2. \end{aligned} \tag{4.48}$$

Thus, SE_{train} is the square of the Eucledian distance between $\boldsymbol{t}$ and $\boldsymbol{y}(\mathbf{W})$. Because $\boldsymbol{y}(\mathbf{W})$ lies in S, the orthogonal projection of $\boldsymbol{t}$ on S minimizes the distance, and thereby the square of the distance, which is the error SE_{train}.

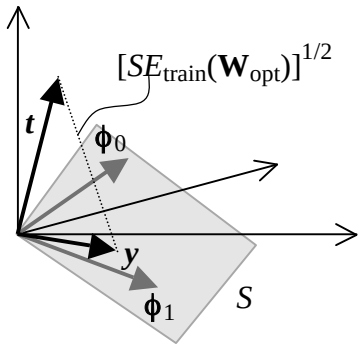


Figure 4.10: Illustrating the geometry of least squares weights. (Redrawn from Bishop's book).

Now we proceed to finding a numerical solution for the least squares problem. First we write (4.46) out in full detail:

$$SE_{\text{train}}(\mathbf{W}) = \frac{1}{2}\sum_{i=1}^{N}\left\|\mathbf{y}(\mathbf{x}_i;\mathbf{W}) - t_i\right\|^2 = \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{n}\left(\sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x}_i) - t_i^k\right)^2 , \tag{4.49}$$

where t_i^k is the k-th component of the target outputs on input $\mathbf{x}_i$. At a minimum of $SE_{\text{train}}(\mathbf{W})$, the derivatives of $SE_{\text{train}}(\mathbf{W})$ w.r.t. the weights must be zero. This gives us the *normal equations* for the least squares problem:

$$\frac{\partial}{\partial w_{kj}}\frac{1}{2}\sum_{i=1}^{N}\sum_{k'=1}^{n}\left(\sum_{j'=0}^{M} w_{k'j'}\phi_{j'}(\mathbf{x}_i) - t_i^{k'}\right)^2 = \sum_{i=1}^{N}\left(\sum_{j'=0}^{M} w_{kj'}\phi_{j'}(\mathbf{x}_i) - t_i^k\right)\phi_j(\mathbf{x}_i) = 0 . \tag{4.50}$$

Assembling all these n times M equations into a matrix equation yields

$$(\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi})\mathbf{W}^{\mathsf{T}} = \boldsymbol{\Phi}^{\mathsf{T}}\mathbf{T}, \tag{4.51}$$

where $\boldsymbol{\Phi}$ has dimension $N \times M$ and elements $\phi_j(\mathbf{x}_i)$ – that is, contains in its rows the filtered input vectors to the network for the training data patterns –, and $\mathbf{T}$ has dimension $N \times n$ and contains the desired target outputs for the training samples in its rows. The matrix $(\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi})$ is square of dimension $M \times M$. Provided it is nonsingular, we obtain the following solution for the weight matrix $\mathbf{W}$:

$$\mathbf{W}^{\mathsf{T}} = (\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathsf{T}}\mathbf{T} =: \boldsymbol{\Phi}^{\dagger}\mathbf{T}, \tag{4.52}$$

where $\boldsymbol{\Phi}^{\dagger}$ is known as the *pseudo-inverse* of $\boldsymbol{\Phi}$. Note that $\boldsymbol{\Phi}$ is in general a non-square matrix and thus does not have an inverse. The name "pseudo-inverse" derives from the fact that $\boldsymbol{\Phi}^{\dagger}\boldsymbol{\Phi} = (\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi} = \mathbf{I}$ (but note that not in general $\boldsymbol{\Phi}\boldsymbol{\Phi}^{\dagger} = \mathbf{I}$).

A direct calculation of $(\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathsf{T}}$ is prone to suffer from numerical instability, namely, when $\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi}$ is close to singular. Then, numerical roundoff error or statistical data noise contained in $\boldsymbol{\Phi}$ becomes largely magnified through the $^{-1}$ operation. This can be avoided by

calculating the pseudo-inverse via the *singular value decomposition* of $\mathbf{\Phi}$. Actually, that is how most (if not all) professional software tools do it, including Matlab and Mathematica. If you want to see a truly beautiful introduction to singular value decompositions, check the online tutorial of Todd Will at http://www.uwlax.edu/faculty/will/svd/index.html. Going through this tutorial will give you perfect insight into basic matrix theory within one hour – you should absolutely do it.

Linear regression. If one has N n-dimensional data vectors $\mathbf{x}_i$ assembled row-wise in a data collection matrix $\mathbf{M}$ of size $n \times N$, and a target vector $\mathbf{t}$ of size N, the task to find weights $\mathbf{W}$ which minimize the square error

$$SE(\mathbf{W}) = \left\| \mathbf{W}\mathbf{M}^T - \mathbf{t}^T \right\|^2 \tag{4.53}$$

is known as the *linear regression* task; and its solution according to (4.52),

$$\mathbf{W}_{opt} = \arg\min_{\mathbf{W}} SE(\mathbf{W}) = (\mathbf{M}^\dagger \mathbf{t})^T \tag{4.54}$$

is called the *regression weights*.

A cautionary remark. The least mean square solution for learning network weights from data is easy to compute and does not require much thinking about the class-conditional distributions of the input features $\mathbf{x}_i$. That's good. However, neither is linear discrimination appropriate for all problems (they might be nonlinear), nor is it easy to find good preprocessing filters ϕ_j if you want to tackle nonlinear classification problems (you will need *unsupervised* learning techniques to optimize them), nor – even if you have found good ϕ_j – is the least mean square approach necessarily the best you can do for training classificators (because it tends to over-represent extreme or even outlier inputs; you may land far from the optimal weights that would be yielded by a probabilistic approach where you first estimate the posterior class distributions). So there is ample room for further improvements. This all said, in practice a linear discriminant trained by minimizing square error often is a quite accurate and certainly a very simple way to learn a classificator.

4.4.3 Slow Feature Analysis

Slow Feature Analysis (SFA) is a recently found method of machine learning which is a beautiful illustration of the power and elegance of the basic linear algebra transformations that we have been considering throughout this section. In my treatment I follow (often verbatim) the 2002 article[6] from Laurenz Wiskott and Terry Sejnowski where this method was first presented. A tutorial slideshow by Laurenz is online at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/TutorialWiskott2008.pdf. In these lecture notes I only outline the basic algorithm; the tutorial is rich in amazing examples.

The data on which SFA learns is a time series signal $\mathbf{x}(t) = (x_1(t), ..., x_I(t))^T$, where $t = 0, ..., N$. The goal is to find an input-output function $g(\mathbf{x}) = (g_1(\mathbf{x}), ..., g_J(\mathbf{x}))^T$ generating a J-dimensional output signal $\mathbf{y}(t) = (y_1(t), ..., y_J(t))^T$, with $y_j(t) := g_j(\mathbf{x}(t))$, which are "slow" in the following sense. Let $\dot{y}_j(t) := y_j(t+1) - y_j(t)$ be the discrete version of the "time derivative"

[6] L. Wiskott, T. J. Sejnowski (2002): Slow Feature Analysis: Unsupervised Learning of Invariances. Neural Computation 14, 715-770

of the signal $y_j(t)$. Let $\langle y_j \rangle = 1/N \sum_{t=1,...,N} y_j(t)$ denote the temporal average of $y_j(t)$. Then we desire that for all j, the average squared time derivative

(4.55) $\Delta_j := \Delta(y_j) := \langle \dot{y}_j^2 \rangle$

is minimal under the constraints

(4.56) $\langle y_j \rangle = 0$ (zero mean)
(4.57) $\langle y_j^2 \rangle = 1$ (unit variance)
(4.58) $\langle y_{j'} y_j \rangle = 0$ for all $j' < j$ (decorrelation).

Equation (4.55) expresses the primary objective of minimizing the temporal variation of the output signal. Constraints (4.56) and (4.57) help avoid the trivial solution $y_j(t) = \text{const}$. Constraint (4.58) guarantees that different output signal components carry different information and do not simply reproduce each other. It also induces an order, so that $y_1(t)$ is the optimal (slowest) output signal component, while $y_2(t)$ is a less optimal one, since it obeys the additional constraint $\langle y_1 y_2 \rangle = 0$. Thus, $\Delta(y_{j'}) \leq \Delta(y_j)$ if $j' < j$.

A general solution to this slowness optimization would try to optimize over all nonlinear candidate functions for the g_j. This task is difficult and would require methods from variational calculus. However, there is a workaround that is used in several situations in machine learning – in fact, it is the same trick that we used in radial basis function networks. Namely, we first project the raw data $\mathbf{x}(t)$ into a rich (and possibly high-dimensional) feature space, by using a fixed, pre-determined bank of K (nonlinear) filters $\phi_k: \mathbb{R}^l \rightarrow \mathbb{R}$, where $k = 1, ..., K$. A typical choice in SFA is to use all polynomials of the raw x_i of order up to 2. For example, if $l = 2$, we would use $\phi_1 = x_1$, $\phi_2 = x_2$, $\phi_3 = x_1 x_1$, $\phi_4 = x_1 x_2$, $\phi_5 = x_2 x_2$. Applying $\boldsymbol{\phi} = (\phi_1, ..., \phi_K)^T$ to the input signal yields a nonlinearly expanded, K-dimensional signal $\tilde{\mathbf{z}}(t) = \boldsymbol{\phi}(\mathbf{x}(t))$. As a next step, we whiten the signals $\tilde{\mathbf{z}}(t)$ using the method from Section 4.4.2, obtaining a new version $\mathbf{z}(t) = (z_1(t), ..., z_K(t))^T$ of nonlinear transforms which satisfy

(4.59) $\langle \mathbf{z} \rangle = \mathbf{0}$ and
(4.60) $\langle \mathbf{z}\, \mathbf{z}^T \rangle = \mathbf{I}$,

and which we will use hereafter instead of the raw $\mathbf{x}$.

Just as we did in the case of RBF networks, after the nonlinear expansion (and whitening) we will treat the optimization problem as linear in the expanded signal components $z_k(t)$. That is, we restrict our optimization search to functions $y_j(t) = g_j(\mathbf{x}(t)) = \mathbf{w}_j^T \mathbf{z}(t)$, where the weight vectors $\mathbf{w}_j = (w_{j1}, ..., w_{jK})^T$ are subject to learning. Now, in this setting, the objective (4.55) to minimize $\langle \dot{y}_j^2 \rangle$ reduces to find $\mathbf{w}_j$ such that

(4.61) $\Delta(y_j) := \langle \dot{y}_j^2 \rangle = \mathbf{w}_j^T \langle \dot{\mathbf{z}}\, \dot{\mathbf{z}}^T \rangle \mathbf{w}_j$

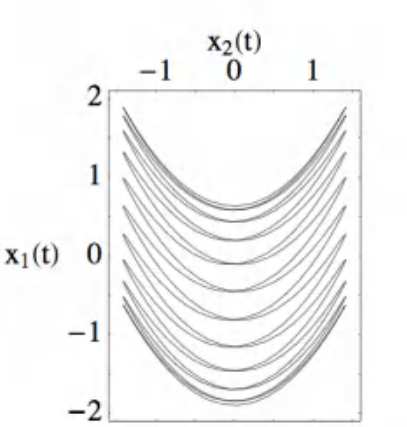
becomes minimal, subject to the constraints (4.56) – (4.58). Constraint (4.57) and $y_j(t) = \mathbf{w}_j^T \mathbf{z}(t)$ and (4.60) imply that $\|\mathbf{w}_j\| = 1$, so we may restrict our search for optimal weights to unit norm weight vectors.

Observe that $\mathbf{w}_j^T < \dot{\mathbf{z}}\dot{\mathbf{z}}^T > \mathbf{w}_j$ is a quadratic form in $\mathbf{w}$, with $\dot{\mathbf{z}}\dot{\mathbf{z}}^T$ being a symmetric (covariance) matrix. Let $\lambda_1 \leq ... \leq \lambda_J$ be the J smallest eigenvalues of $\dot{\mathbf{z}}\dot{\mathbf{z}}^T$, and $\mathbf{u}_1$, ..., $\mathbf{u}_J$ the associated unit norm eigenvectors. If you think about Figure 4.7A, you will find that $\mathbf{w}^T < \dot{\mathbf{z}}\dot{\mathbf{z}}^T > \mathbf{w}$ is smallest when $\mathbf{w} = \mathbf{u}_1$. So we choose $\mathbf{w}_1 = \mathbf{u}_1$, which gives us $y_1 = \mathbf{w}_1^T \mathbf{z}$. Furthermore, the next smallest solution which is orthogonal to y_1 (constraint (4.58)!) is obtained by choosing $\mathbf{w}_2 = \mathbf{u}_2$, etc. We thus end up with using for the $\mathbf{w}_j$ the J norm-1 eigenvectors of $\dot{\mathbf{z}}\dot{\mathbf{z}}^T$ that correspond to the smallest eigenvalues.

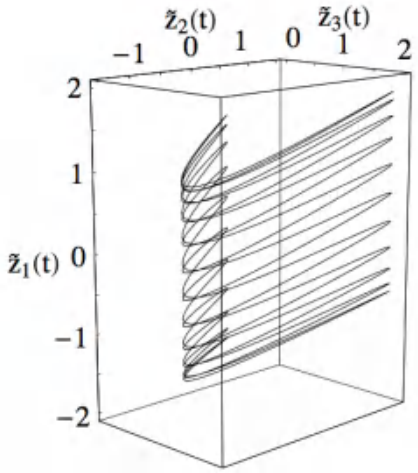
Summing up, the SFA algorithm takes us through the following steps:

1. Nonlinear expansion: transform the raw data $\mathbf{x}$ to (K-dimensional) $\tilde{\mathbf{z}}(t) = \boldsymbol{\phi}(\mathbf{x}(t))$, where $\boldsymbol{\phi}$ is some predetermined bank of nonlinear transforms (often polynomials).
2. Whiten $\tilde{\mathbf{z}}$ to obtain $\mathbf{z}$.
3. Choose the J smallest-eigenvalue eigenvectors of $\dot{\mathbf{z}}\dot{\mathbf{z}}^T$ and declare them as the weights $\mathbf{w}_1$, ..., $\mathbf{w}_J$.
4. Then, $y_j = \mathbf{w}_j^T \mathbf{z}$ will be the desired solutions that solve the constrained optimization problem (4.55) – (4.58).

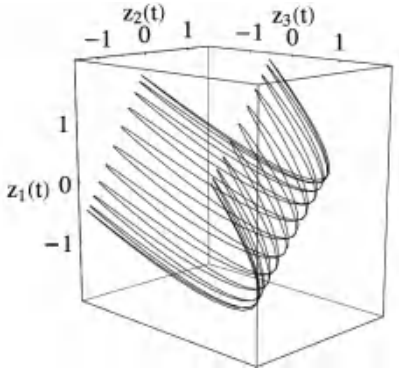
A didactic example. The following simple, synthetic example (taken from the Wiskott & Sejnowski paper) illustrates the working of SFA. For explanation see figure caption.



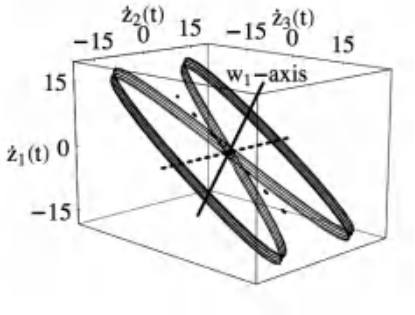
a) input signal $\mathbf{x}(t)$



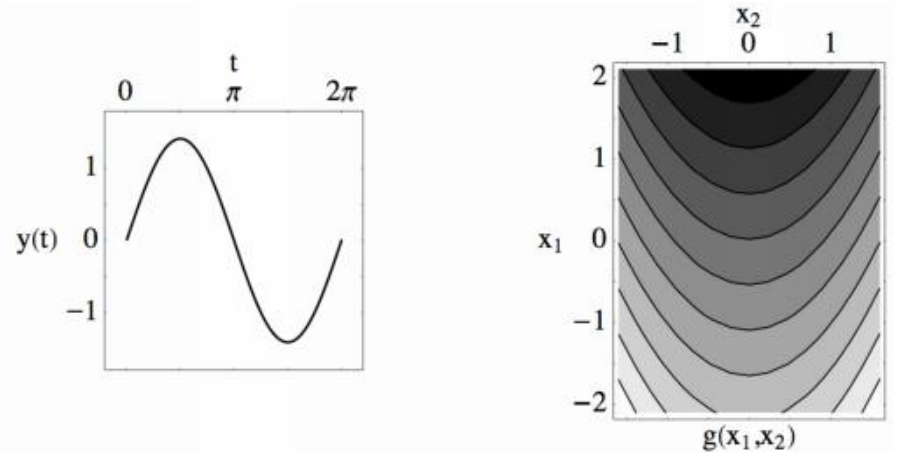
b) expanded signal $\tilde{\mathbf{z}}(t) = \boldsymbol{\phi}(\mathbf{x}(t))$. Three of five components are shown.



c) whitened ("sphered") $\mathbf{z}(t)$



d) time derivative signal $\dot{\mathbf{z}}(t)$



e) output signal $y(t)$ f) input-output function $g_1(\mathbf{x})$

Figure 4.11 (taken from Wiskott & Sejnowski 2002). Illustration of the learning algorithm by means of a simplified example. (a) Input signal $\mathbf{x}(t)$ is given by $x_1(t) := \sin(t) + \cos^2(11\ t)$, $x_2(t) := \cos(11\ t)$, t œ [0, 2], where $\sin(t)$ constitutes the slow feature signal. It results in a 2-D trajectory where a fast parabola is evolving while slowly moving up and down. (b) Expanded signal $\tilde{\mathbf{z}}(t)$ is defined as $\tilde{z}_1(t) := x_1(t)$, $\tilde{z}_2(t) := x_2(t)$, and $\tilde{z}_3(t) := x_2^2(t)$. Components $x_1^2(t)$ and $x_1(t)\ x_2(t)$ are left out for easier display. (c) Whitened signal $\mathbf{z}(t)$ has zero mean and unit covariance matrix. Its orientation in space is algorithmically determined by the principal axes of $\tilde{\mathbf{z}}(t)$ but otherwise arbitrary. (d) Time derivative signal $\dot{\mathbf{z}}(t)$. The direction of minimal variance determines the weight vector $\mathbf{w}_1$. This is the direction in which the whitened signal $\mathbf{z}(t)$ varies most slowly. The axes of next higher variance determine the weight vectors $\mathbf{w}_2$ and $\mathbf{w}_3$, shown as dashed lines. (e) Projecting the whitened signal $\mathbf{z}(t)$ onto the $\mathbf{w}_1$-axis yields the first output signal component $y_1(t)$, which is the slow feature signal $\sin(t)$. (f) The first component $g_1(x_1, x_2)$ of the input-output function derived by the steps a) to e) is shown as a contour plot.

This simple example does not reveal the true powers of SFA. One nice example that was worked out in a partner lab of mine (Reservoir Lab, University of Gent) used SFA to train speaker and spoken digit recognizers. The training data consisted in (suitably pre-processed) audiorecordings of utterances of the digits "zero" to "nine", spoken by different speakers. When SFA was used on a training audiostream where first all digits were spoken by one speaker, then by the next speaker, etc., a speaker voice recognizer resulted because the slowest feature was "speaker". When the single recordings were ordered in time such first all "zero" utterances, across speakers, were displayed, then all "one" recordings etc., a spoken digit recognizer came out – because now the slowest source of variance was the kind of digit. The main application area of SFA seems to be in visual (video) image processing; SFA also has been proposed as a model of how animals learn positions of landmarks (these are persistent, i.e. slow, while the animal moves) and register them in a brain region known as hippocampus.

Take-home messages from Section 4

- In this section we "forgot" about the probabilistic approach to the classification problem and considered the "shortcut" to estimate generalized linear discriminant functions $y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x}) + w_{k0}$ directly.
- Such discriminant functions can be seen as single-layer neural networks ("layer" refers to a layer of weights, not neurons).
- With logistic output activation functions, the probabilistic interpretation of discriminant functions, $y_k(\mathbf{x}) = p(C_k \mid \mathbf{x})$, can be recovered if the class-conditional probability distributions come from the exponential family.
- Using linear output activation functions, explicit solutions for the network weights can be calculated that minimize the training error in the least square error sense. This also works when the input neurons are arbitrary preprocessing filters ϕ_j.
- In SFA the same trick is used, namely, first use pre-defined nonlinear filters ϕ_j to expand an input signal into a *higher-dimensional* feature space, and then use linear methods to solve the task, which in SFA was to identify the slowest characteristics in the input signal.
- If you know how to handle eigenvectors and eigenvalues of covariance matrices you can get awfully far in an easy-going way – using Karhunen-Loéve, whitening, PCA to start with and (not) ending in a variety of machine learning methods that boil down to linear regression performed on the basis of nonlinearly transformed data, as in RBF networks or in SFA. The method of Echo State Networks, treated in a later section of these lecture notes, are another example of this trick.

5 An unsupervised intermezzo: K-means clustering

In this lecture we almost exclusively treat *supervised* learning problems. Such problems are characterized by the fact that each training sample comes with a correct target value (for instance a class label or a regression value). The task is then to learn from the training data an algorithm that should approximate the known target values on the training data, and generalize well to samples not seen during training.

In *unsupervised* learning, no correct target values are supplied – only the input patterns $\mathbf{x}_i$. The learning task is here *to discover structure* in the data.

For instance, the $\mathbf{x}_i$ might be vectors of customer data of some mail order warehouse. The company running this business is probably interested in grouping their customers into groups that have similar customer profiles, in order to facilitate group-specific advertisment campaigns. More generally, discovering *clusters* within an a priori unstructured sample set is often a first step in exploratory data analysis in the natural and social sciences. The guiding idea is to find K clusters such that the training samples $\mathbf{x}$, $\mathbf{x}'$ that are assigned to the same cluster G should lie closely together, that is, $\|\mathbf{x} - \mathbf{x}'\|$ should be small. Conversely, if $\mathbf{x}$, $\mathbf{x}'$ are assigned to two different clusters G, G', then $\|\mathbf{x} - \mathbf{x}'\|$ should be large. This is easy if samples $\mathbf{x}$ are just numerical feature vectors. When samples come in a heterogeneous data format – for instance, $\mathbf{x}$ is a description of a customer involving numerical data *and* class data *and* symbolic descriptions – then finding a distance measure $\|\cdot\|$ in the first place is a challenging task. Solving this task often requires some ingenuity, and the distance measure found may not satisfy all the mathematical requisites of a metric. In such cases, there are many specific clustering techniques that work with different kinds of pseudo-distances.

There are many other unsupervised data-structuring tasks besides simple clustering of real-valued sample vectors $\mathbf{x}$:

- In a time series that is generated by alternating, different generators, one might want to discover the number of such generators and when they start and end generating a time series (unsupervised time series segmentation). A nice example for this is the discovery of different sleep phases in EEGs of human subjects. A common approach is to train a set of K signal predictor devices (e.g., neural networks) in mutual competition. In such *mixture of experts training*, at each time step only one of the K predictors is allowed to adjust its parameters – namely the one that could best predict the next signal value. Starting from K randomly initialized predictors, this setup leads to a competitive differentiation of predictors, each of which learns to specialize on the prediction of a particular generating mode in the time series.
- In an auditory signal that is an additive mixture of generators, one might wish to single out the generating signals. Surprisingly enough, this is possible if the original generators are statistically independent. Check Paris Smaragdis' FAQ page on *blind signal separation* and *independent component analysis* at http://web.media.mit.edu/~paris/ica.html for compelling audio demo tracks where a raw signal that is an overlay of several speakers or muscial instruments is separated into almost crystal-clear audiotracks of the individual speakers or instruments.
- In symbolic machine learning and data mining, there are numerous unsupervised learning techniques that aim at distilling concise symbolic descriptions (e.g., context-free grammars or automata) from (huge) symbolic datasets.

- Another field that is, technically speaking, a case of unsupervised learning is *data compression.* Here the task is to detect regularities (= redundancies) in a large dataset that can be used to rewrite the data in a condensed form.

All in all, the field of unsupervised machine learning is as large, as important and as fascinating as the field of supervised learning. It is sad that a semester is short and enforces concentration on only one branch of ML. All we will do is to describe a particularly simple technique for clustering, called *K*-means clustering. Without some such unsupervised technique, one cannot really use RBF networks – the determination of well-placed and well-shaped input filters ϕ_i is a typical unsupervised learning task.

We can be brief, because *K*-means clustering is almost self-explanatory. Given: a training data set $(\mathbf{x}^i)_{i=1,...,N} = ((x^i_1, \ldots, x^i_n))_{i=1,...,N}$ of real-valued feature vectors, and a number *K* of clusters that one maximally wishes to obtain. The algorithm goes like this:

Initialization: randomly assign the training samples to *K* sets S_j ($j = 1, ..., K$).

Repeat: For each set S_j, compute the mean $\boldsymbol{\mu}_j = \frac{1}{|S_j|}\sum_{\mathbf{x}\in S_j}\mathbf{x}$. Create new sets S'_j by putting each sample x into the set S'_j where $\|\boldsymbol{\mu}_j - \mathbf{x}\|$ is minimal. Dismiss empty S'_j and reduce *K* to *K'* by subtractring the number of dismissed empty sets (this happens rarely). Put $S_j = S'_j$ (for the nonempty sets) and $K = K'$.

Termination criterium: Stop when in one iteration the sets remain unchanged.

It can be shown that at each iteration, the error quantity

$$J = \sum_{j=1}^{K}\sum_{\mathbf{x}\in S_j}\|\mathbf{x} - \boldsymbol{\mu}_j\|^2 \tag{5.1}$$

will not increase. The algorithm typically converges quickly and works well in practice. It finds a local minimum or saddle point of *J*. The final clusters S_j may depend on the random initialization. The clusters are bounded by straight-line boundaries; each cluster forms a *Voronoi cell*. Thus, *K*-means cannot find clusters defined by nonlinear boundaries. Figure 5.1 shows an example of a clustering run using *K*-means.

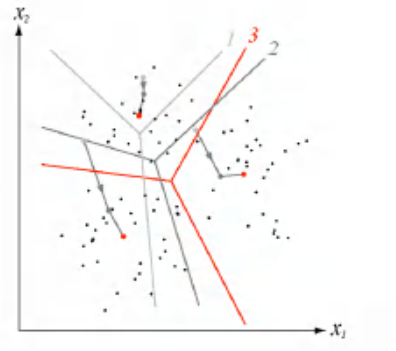


Figure 5.1: Running *K*-means with $K = 3$ on two-dimensional samples **x**. Dots mark cluster means $\boldsymbol{\mu}_j$, lines mark cluster boundaries. The algorithm terminates after three iterations. (Picture taken from chapter 10 of the Duda/Hart/Stork book).

It is clear how *K*-means clustering may be used in conjunction with an RBF network. This is how:

Problem statement: given *d*-dimensional training data $\mathbf{x}_i$ (i = 1,..., N, may be preprocessed), find a "good" number M of "good" RBF filters filters ϕ_j (j = 1,..., M) to train a RBF network $y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x}) + w_{k0}$ for n classes k = 1, ..., n.

Idea: Cluster the $\mathbf{x}_i$ by *K*-means clustering, create one filter per cluster, use Gaussian RBFs with covariance matrix Σ determined by cluster properties to represent data of each cluster.

Algorithm:

1) Use intuition (or trial and error with cross-validation...) to fix a desired target clustering number M. Put $M = K$.
2) Run *K*-means clustering on training data $\mathbf{x}_i$, which partitions the training samples into M' (maybe $M' < K = M$, but most likely $M' = K$) D_1, ..., D_M with means $\boldsymbol{\mu}_1$, ..., $\boldsymbol{\mu}_{M'}$.
3) For cluster j (j = 1, ..., M') let $\mathbf{x'}^j_1,...,\mathbf{x'}^j_{N_j}$ be the normalized (i.e., cluster mean subtracted) samples in D_j, and let $\mathbf{X'}^j$ be the matrix that contains $\mathbf{x'}^j_1,...,\mathbf{x'}^j_{N_j}$ as columns. Compute the covariance matrix $\Sigma_j = \mathbf{X'}^j (\mathbf{X'}^j)^T$. Put $\phi_j(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_j)^T \Sigma_j^{-1}(\mathbf{x}-\boldsymbol{\mu}_j)\right)$.

Notes:

The RBF filters filters ϕ_j created in this way should accomodate to the shape and orientation of their cluster due to Σ_j. Some experimentation with Σ_j might however further improve the overall classificator performace: one might for instance try to flatten out the filters ϕ_j for clusters with only few members, by using $\kappa_j \Sigma_j$ instead of Σ_j, where $\kappa_j > 1$.

Strictly speaking, we don't have a *radial* basis function network any more, because the filters ϕ_j are not radially symmetric. However, one still speaks of RBF networks.

The RBF filters are derived from pdf's of multivariate Gaussians. However, they are not strictly pdfs because our ϕ_j don't integrate to unity. For the purposes of using the ϕ_j as input filters in a RBF network, that does not matter (the weights of the network will be adjusted automatically to make up for different scalings of the filter functions).

What we did here is related to the task of approximating a probability distribution by a *mixture of Gaussians*. What is that? Let p be some pdf over a d-dimensional sample space, possibly shaped in a badly nonlinear and "bumpy" way. Let ($\boldsymbol{\mu}_j$, Σ_j) [j = 1,..., M] be d-dimensional Gaussian distributions with pdfs p_j, and let $p' = \sum_{j=1,...,M} \alpha_j p_j$, where $\sum_{j=1,...,M} \alpha_j = 1$, be a mixture pdf made from the Gaussian distributions with non-negative mixture coefficients α_j. The task of finding a "good" mixture of Gaussians is to find parameters ($\boldsymbol{\mu}_j$, Σ_j) such that p' becomes as similar to p as possible (there are several ways to specify what "similar" means). Our method of finding ϕ_j via *K*-means clustering would lead to reasonably good such mixtures of Gaussians if we would use the true (integrating to unity) pdfs ($\boldsymbol{\mu}_j$, Σ_j) for our ϕ_j,

and would weigh them by the relative sizes of the clusters, that is, put $\alpha_j = N_j/N$. However, there are better (but more complex and computationally more expensive) ways to find good (indeed, optimal) mixture of Gaussian approximations to a target distribution (using the EM algorithm – which we will meet later in this lecture in a different context – see Section 2.6 in the Bishop book).

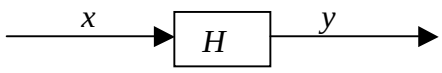
6 The adaptive linear combiner

Overview. In this section we will consider a linear regression task (as opposed to the linear classification tasks from Section 4) on time series data (as opposed to the static samples **x** from Section 4), training an online adaptive filter that incrementally adapts its weights as new data come in (as opposed to the "batch" offline least squares solutions from Section 4). The background is signal processing / electrical engineering. The assumption that the dynamic system we want to learn is indeed linear is often satisfied in signal processing – wired and wireless communication channels, which are a main type of system that require signal processing techniques, are indeed typically quite linear.

Throughout this section, I am loosely guided by the book from B. Farhang-Boroujeny, *Adaptive Filters: Theory and Applications*, Wiley & Sons 1999, [IRC: TK7872.F5 F37] which I would recommend to purchase to those students who aim at a career in signal processing.

6.1 The adaptive linear combiner, a special Wiener filter

I start with a refresher on systems and signals terminology. A discrete-time, real-valued *signal* is a left-right infinite sequence $x = \{x(n)\} = (x(n))_{n \in \mathbb{Z}} \in \mathbb{R}^{\mathbb{Z}}$. (We will only consider discrete-time, real-valued signals here.) Note that in this section, x and $\{x(n)\}$ refer to a complete sequence of values; if we want to single out a particular signal value, we write $x(n)$. A *system* (or *filter*) H is a mapping from signals to signals, written $y = H(x)$ or in graphical notation



The signal x is the input signal and y is the output signal of H. A system is *linear* if for all complex constants a, b and signals x_1, x_2 it holds that

(6.1) $$H(a\,x_1 + b\,x_2) = H(a\,x_1) + H(b\,x_2).$$

A system is *shift-invariant* if

(6.2) $$\forall \{x(n)\}\ \forall k \in \mathbb{Z}\ \ H(\{x(n-k)\}) = \{H(\{x(n)\})(n-k)\}.$$

A shift-invariant, linear system is called an LSI system for short. We will be concerned with LSI systems exclusively.

- The *unit impulse* $\delta(n)$ is a signal that is defined to be 1 for $n = 0$ and zero elsewhere. Let $H(\delta) = h$ be the *impulse response* of a system H. For an LSI H, we get the system response on input x by convolving x with the impulse response:

(6.3) $$\{y(n)\} = \left\{\sum_{k=-\infty}^{\infty} x(k)\,h(n-k)\right\} = \left\{\sum_{k=-\infty}^{\infty} h(k)\,x(n-k)\right\} = \{x(n)\} \otimes \{h(n)\}.$$

A system is *causal* if its current output does not depend on future inputs, or equivalently, if $h(n) = 0$ for $n < 0$.

An LSI filter is a *finite impulse response filter* (FIR filter) if h has a finite carrier, that is, h is zero except at a finite number of points.

We will restrict ourselves to time-domain signal representations in this section; a frequency-domain treatment of adaptive filters is also possible but seems less common (Farhang-Boroujeny book, Section 7).

In a causal FIR filter, the output $y(n)$ is a linear function of a finite number of previous M inputs $x(n)$, $x(n-1)$, ..., $x(n-M+1)$, that is,

$$(6.4) \quad y(n) = \sum_{j=1}^{M} w_j \, x(n-j+1).$$

Engineers call this equation a *transversal filter*, and the engineer's (and Simulink's) way of graphically representing it is shown in Figure 6.1.

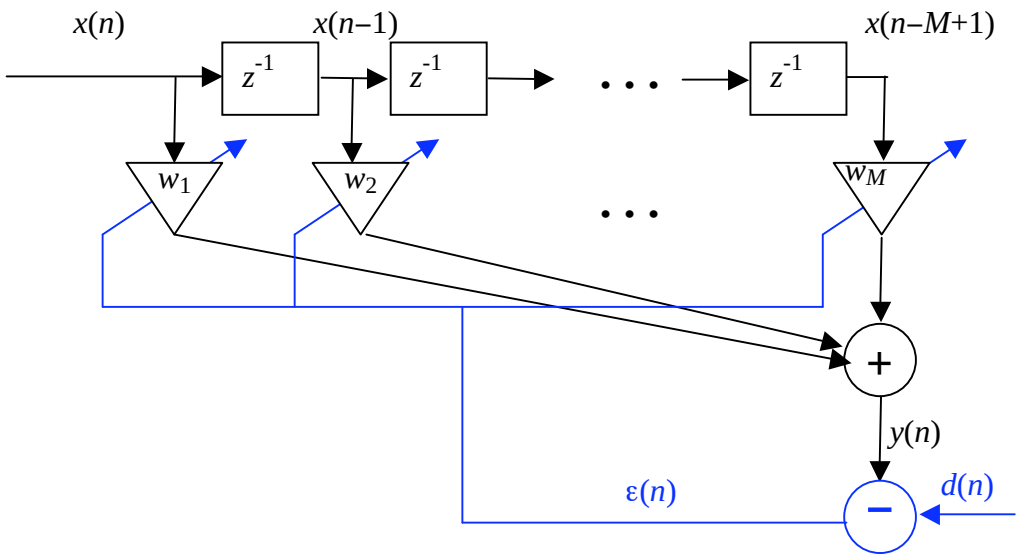


Figure 6.1: A transversal filter (black) and an adaptive linear combiner (black plus blue). The z^{-1} boxes are unit delay elements.

We will consider the task where the filter output $y(n)$ should be made to follow a *desired* teacher signal $d(n)$ as well as possible in the mean square error sense, by adapting the weights w_j. Engineers sometimes call this setup an *adaptive linear combiner* (blue parts in Fig. 6.1), a special case of *Wiener filters* (Wiener filters in general have a version of Eq. (6.4) where on the lhs there are also weighted terms of the form $w_i \, y(n\text{-}k)$, that is, the current output depends on previous inputs and previous outputs. If such terms are included, the impulse response of a filter generally attains infinite length, and one has *infinite impulse response* (IIF) filters). Another name for transversal filters is *tapped delay line*, and the filter weights are sometimes called *tap weights*.

Formally, we want to find optimal weights $\mathbf{w}_{\text{opt}} = (w_{\text{opt}1}, ..., w_{\text{opt}M})^{\mathsf{T}}$ such that

(6.5) $\mathbf{w}_{\text{opt}} = \arg\min_{\mathbf{w}} E[\varepsilon_{\mathbf{w}}(n)^2] = \arg\min_{\mathbf{w}} E[(d(n) - y_{\mathbf{w}}(n))^2] = \arg\min_{\mathbf{w}} E[(d(n) - \mathbf{w}^\top \mathbf{x}(n))^2]$,

where $\mathbf{x}(n) = (x(n), x(n-1), ..., x(n-M+1))^\top$.

We could frame this in the spirit of Section 4.4 as a single-layer neural network with a single output unit y and M input units (with no extra bias input). The training patterns would be the $\mathbf{x}(n)$ and the targets would be $d(n)$. We could use Eq. (4.52) directly to obtain an estimate for $\mathbf{w}_{\text{opt}}$ from a finite training data set comprising filter inputs $x(1)$, ..., $x(N)$ and desired outputs $d(M)$, $d(M+1)$, ..., $d(N)$.

However, here we want to derive an online, *adaptive* algorithm that updates the weights incrementally as new training data come in. Such an adaptive procedure maintains a set of weights $\mathbf{w}(n)$ at every time, and should yield the correct optimal weights in the limit of infinite time, that is, $\lim_{n\to\infty} \mathbf{w}(n) = \mathbf{w}_{\text{opt}}$. This reflects the temporal nature of our training data, and the common situation in signal processing that a filter should be able to track time-varying systems online.

Wiener-Hopf equation. In order to prepare the grounds for an online learning algorithm, we derive a variant of Eq. (4.52), the *Wiener-Hopf equation*, from scratch. Let $\xi(\mathbf{w}) = E[(\varepsilon_{\mathbf{w}}{}^2(n))]$ denote the mean square error, and rewrite it as follows:

$$
\begin{aligned}
\xi(\mathbf{w}) &= E[(d(n) - \mathbf{w}^\top \mathbf{x}(n))\ (d(n) - \mathbf{w}^\top \mathbf{x}(n))] \\
&= E[(d(n))^2] - 2\,\mathbf{w}^\top E[\mathbf{x}(n)d(n)] + \mathbf{w}^\top E[\mathbf{x}(n)\mathbf{x}^\top(n)]\mathbf{w} \\
&= E[(d(n))^2] - 2\mathbf{w}^\top \mathbf{p} + \mathbf{w}^\top \mathbf{R}\mathbf{w},
\end{aligned}
\tag{6.6}
$$

where we introduced the $M \times 1$ cross-correlation vector of the tap inputs with the desired signal

$$\mathbf{p} = E[\mathbf{x}(n)d(n)] = (E[x(n)d(n)] \ldots E[x(n-M+1)d(n)])^\top, \tag{6.7}$$

and the $M \times M$ correlation matrix

$$\mathbf{R} = E[\mathbf{x}(n)\mathbf{x}^\top(n)]. \tag{6.8}$$

Eq. (6.6) is a quadratic function in $\mathbf{w}$. Because $\xi(\mathbf{w})$ cannot be negative, $\xi(\mathbf{w})$ must have the shape of a hyperparabeloid which is opened upwards. Figure 6.2 shows this function for the case of two-dimensional $\mathbf{w}$.

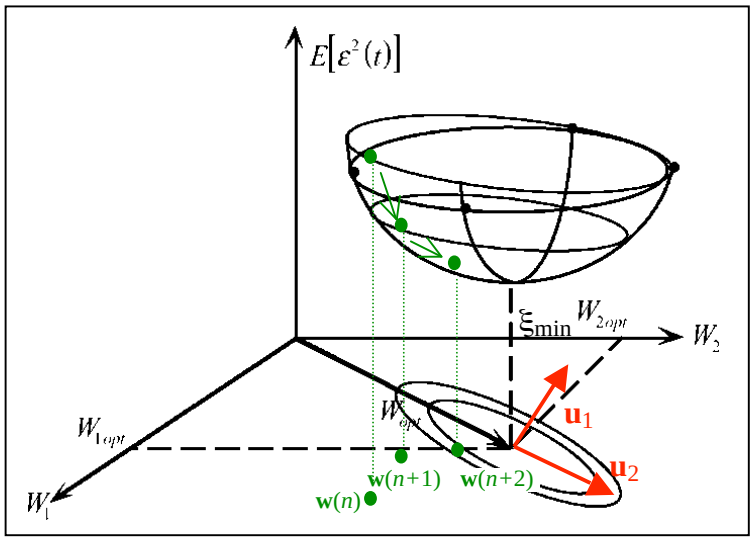


Fig. 6.2: The performance surface, case of two-dimensional weight vectors (black parts of this drawing taken from drip.colorado.edu/~kelvin/links/Sarto_Chapter2.ps). An adaptive algorithm for weight determiniation would try to determine a sequence of weights ... $\mathbf{w}(n)$, $\mathbf{w}(n+1)$, $\mathbf{w}(n+2)$,... that moves toward $\mathbf{w}_{opt}$ (green). The eigenvectors $\mathbf{u}_i$ of the correlation matrix lie on the central axes of the hyperellipsiod given by the level curves of the performance surface (red).

The function shown in Fig. 6.2 is called the *performance surface* of an adaptive linear combiner. It has a minimum at $\mathbf{w}_{opt}$, which is the unique weight value where the gradient of ξ vanishes. This gradient can be computed (by expanding (6.6), for a complete derivation see Farhang-Boroujeny p. 53) as

$$\nabla\xi = \left(\frac{\partial\xi}{\partial w_1} \cdots \frac{\partial\xi}{\partial w_M} \right)^T = 2\mathbf{R}\mathbf{w} - 2\mathbf{p}. \tag{6.9}$$

Putting this to zero gives us the *Wiener-Hopf* equation

$$\mathbf{R}\,\mathbf{w}_{opt} = \mathbf{p}, \tag{6.10}$$

which yields the optimal weights by $\mathbf{w}_{opt} = \mathbf{R}^{-1}\mathbf{p}$.

Note that (6.10) can be seen as a version Eq. (4.51), which was $(\mathbf{\Phi}^T\mathbf{\Phi})\mathbf{W}^T = \mathbf{\Phi}^T\mathbf{T}$, for the special case where we have only a single network output unit. There are two differences. The unimportant one is that (6.10) concerns the case of a single output, whereas (4.51) described several outputs. The important difference is that (4.51) described weights that gave least mean square error *on a finite set of training data*, whereas (6.10) describes the weights for the least mean square error in the average over *all possible signals* from a probability space, which gives rise to **R**.

Principle of orthogonality. In passing, we derive a fundamental property of optimally tuned transversal filters, the *principle of orthogonality*, which states that the residual error is uncorrelated to all tap inputs. While we will not use this principle in the sequel, it provides a

deeper insight into such filters and is often exploited in the analysis and design of optimal filters.

Using $\xi = E[(\varepsilon^2(n))]$ we have

(6.11) $$\frac{\partial \xi}{\partial w_i} = E\left[2\varepsilon(n)\frac{\partial \varepsilon(n)}{\partial w_i}\right] \quad \text{for } i = 1, ..., M.$$

Since $\varepsilon(n) = d(n) - y(n)$ and $d(n)$ does not depend on w_i, it holds that

(6.12) $$\frac{\partial \varepsilon(n)}{\partial w_i} = -\frac{\partial y(n)}{\partial w_i} = -x(n - i + 1).$$

Inserting this into (6.11) yields

(6.13) $$\frac{\partial \xi}{\partial w_i} = -2E[\varepsilon(n)x(n - i + 1)].$$

For optimal weights these gradients are zero, that is,

(6.14) $$0 = E[\varepsilon_{\text{opt}}(n)x(n - i + 1)] \quad \text{for } i = 1, ..., M,$$

which is the *principle of orthogonality*. Intuitively, it can be re-phrased like this: "As long as there is any correlation between a tap input and the current error, one can reduce the error further by subtracting away this correlation through a suitable tuning of the weights".

A geometric interpretation of the principle of orthogonality is maybe more enlightening than this rote derivation. A signal source for signals x can be modeled (under certain conditions) as a sequence of random variables X_n, where a particular observed sequence $\{x(n)\}$ is modelled by a sequence $(X_n(\omega))_{n \in \mathbb{Z}}$. Such a sequence of random variables $(X_n)_{n \in \mathbb{Z}}$ is an example of a *stochastic process*. Random variables X_n, Y_m, etc., can be linearly combined and thus can be conceived as vectors in a suitable vector space V (these vectors are numerical functions from Ω to $\mathbb{R}^{\mathbb{Z}}$). Such vector spaces are typically infinite-dimensional. The correlation $E[X\,Y]$ induces an inner product on such spaces, thereby a norm $\|X\| = E[X^2]^{1/2}$ and thus a metric $d(X,Y) = \|X - Y\|$, plus a notion of orthogonality: $X \perp Y$ iff $E[X\,Y] = 0$, that is, two such random variables are orthogonal if they are uncorrelated. In short, we get all the conveniences of a (pre-)Hilbert space – that is, intuitively, you can work with random variables as with vectors of an Euclidean vector space. Now, let's reconsider our tapped delay line. The inputs $x(n)$, $x(n - 1)$, ..., $x(n - M+1)$ turn into random variables X_n, X_{n-1}, ..., X_{n-M+1}, as does the teacher signal $d(n)$ which becomes D_n. All of these are vectors in V. The vectors X_n, X_{n-1}, ..., X_{n-M+1} span an M-dimensional subspace S in V. Typically, D_n is not contained in this subspace. The task of finding optimal weights, in this view, boils down to combine the filter input vectors into a filter output vector Y_n via $Y_n = w_1X_n + ... + w_M X_{n-M+1}$., such that the error signal achieves minimal norm, that is, such that $\|D_n - Y_n\| = E[(D_n - Y_n)^2]^{1/2} = \xi_n^{1/2}$.

Geometrically this amounts to finding the orthogonal projection of D_n on the subspace S. The error signal $\varepsilon = d - y$ becomes the vector $E_n = D_n - Y_n$. We can simply re-use Figure 4.10 with

different vector names to illustrate this. It becomes clear from Figure 6.3 that the optimal weights lead to an error vector that is orthogonal to all the signals X_n, X_{n-1}, ..., X_{n-M+1} – but this is just the principle of orthogonality.

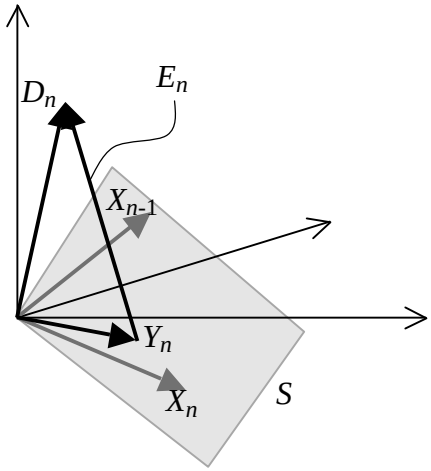


Figure 6.3: Illustration of the principle of orthogonality if $x(n)$ is treated as a random variable.

6.2 Basic applications of adaptive linear combiners

In the previous subsection we considered the following general situation. A time series $x = \{x(n)\} = (x(n))_{n \in \mathbb{Z}}$ of some filter inputs is given, together with a desired filter output $d = \{d(n)\} = (d(n))_{n \in \mathbb{Z}}$. We started to address the task to train a filter that on the same input x produces an output $y = \{y(n)\} = (y(n))_{n \in \mathbb{Z}}$ that matches d as closely as possible in the mean square error sense. We considered tapped delay line filters, but other, more complicated filter designs are of course also possible. Before we proceed with learning algorithms for this task, we will briefly present some standard application situations where this task arises. In the signal processing field, one often finds four basic applications: system identification, inverse system identification, adaptive noise cancelling, and beamforming (design of antenna arrays).

6.2.1 System identification

This is the most basic task: reconstruct from x and d a filter ("model system", "system model", "identification model") y that approximates d. This kind of task is called *system identification*. A schematic block diagram for this kind of application looks as follows:

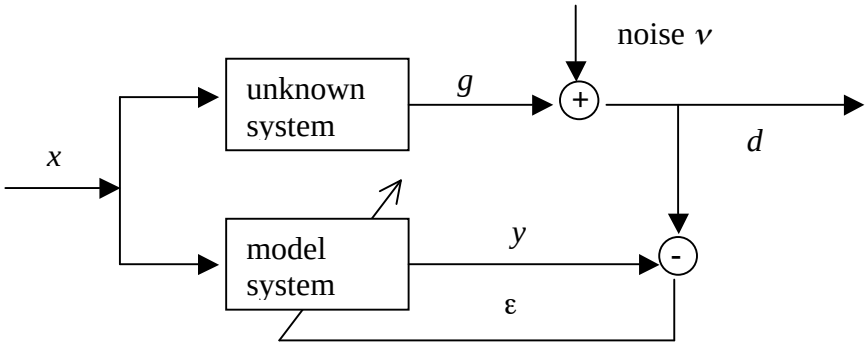


Figure 6.4: Schema of the system identification task.

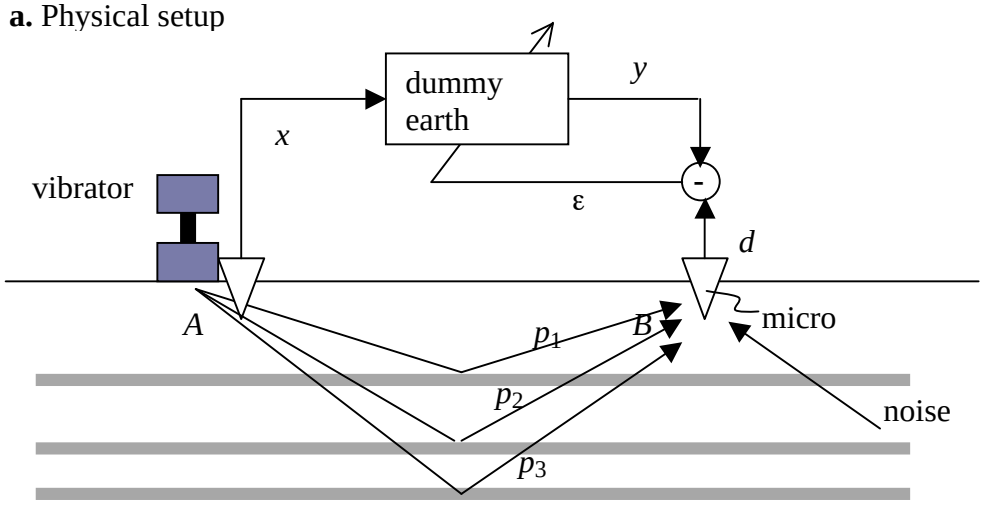
Notes:

- The randomness that is inherent in most real-life systems is modeled by white noise ν that is added to a deterministic system output *g*. This is a highly simplifying assumption ("system + noise" model). Other models of randomness might for instance have systems whose parameters vary randomly – which leads to much more complicated maths.
- The graphical representation with the diagonal ε-arrow through the model system should be read as "adjust model parameters such that $E[|\varepsilon|^2]$ is minimized".
- If the unknown system is shift-invariant ("stationary"), the system identification means to find a model of *the* system. If however the unknown system is non-stationary, that is, its parameters vary (slowly) over time, the system identification task means that one wants to *track* the unknown system, that is, over time the model system should *follow* the unknown system as closely as possible.

Examples (taken from Farhang-Boroujeny).

1. Geological exploration. At one point *A*, the earth surface is excited by a strong acoustic signal *x* (explosion or large vibrating mass). An earth microphone is placed at a distant point *B*, picking up a signal *d*. A model *M* ("dummy earth") is learnt. After *M* is obtained, one may analyse the impulse response *r* of *M*. The peaks of *r* give indications about reflecting layers in the earth crust between *A* and *B*, which correspond to different delayed responses p_i of the input signal *x*.

a. Physical setup



b. Analysis of impulse response

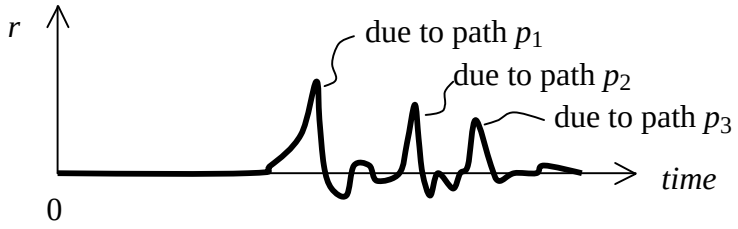


Figure 6.5: Geological exploration via impulse response of learnt earth model.

2. Adaptive open-loop control. In general terms, an open-loop (or *direct* or *inverse* or *feedforward*) controller is a device that generates an input signal u into a system (or *plant*) such that the system output y follows a *reference* (or *target*) trajectory r as closely as possible. In linear systems theory, the system is characterized by a transfer function $H(\omega) = Y(\omega)/U(\omega)$ in the frequency domain (where U, Y are the frequency transforms of the input and output signals of the system, respectively). If the controller has a transfer function $H^{-1}(\omega) = U(\omega)/Y(\omega)$, and the controller is serially connected to the plant, the two transfer functions cancel out and $r = y$ is obtained. One way to obtain H^{-1} is to identify H online as an adaptive linear combiner and compute H^{-1} analytically, as shown in Fig. 6.6:

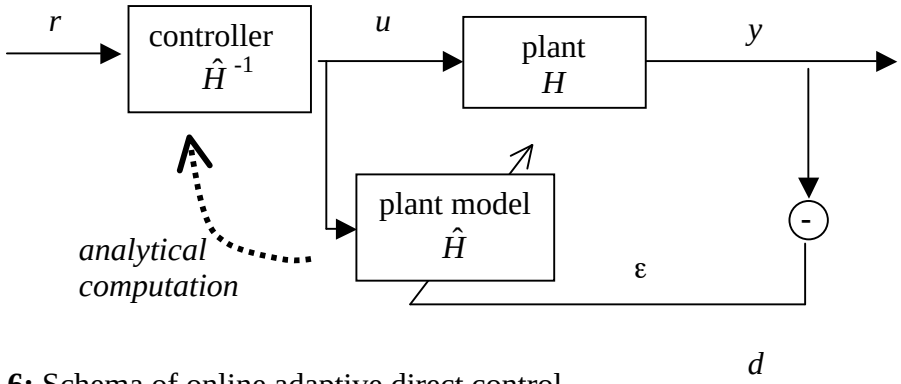


Figure 6.6: Schema of online adaptive direct control.

6.2.2 Inverse system identification

This is the second most basic task: given an unknown system that on input d produces output x, learn an inverse system that on input x produces output d [note the reversal of variable roles and names]. A typical setup is shown in Figure 6.7.

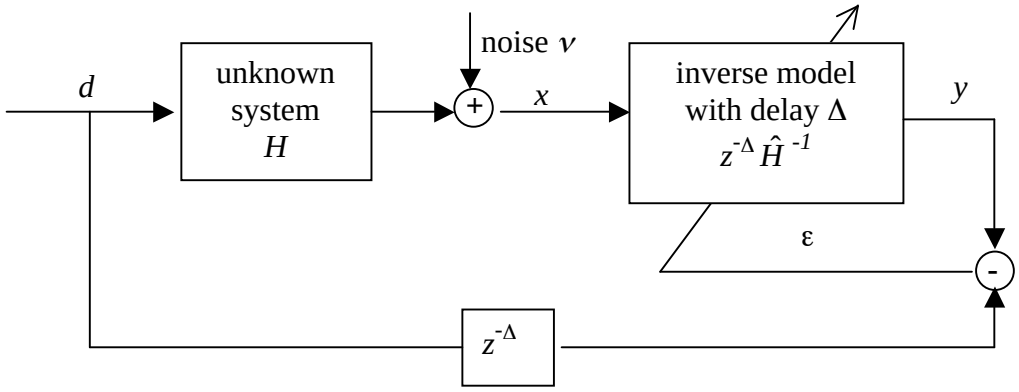


Figure 6.7: Schema of inverse system identification.

Introducing the delay $z^{-\Delta}$ is not always necessary but typically improves stability of the learnt system. Inverse system identification is also referred to as *deconvolution* because the original system *H* transforms its input *d* by convolving it with its impulse response *h*.

Examples.

Equalization of a communication channel (from Farhang-Boroujeny). A prime application of inverse system modelling is in telecommunication, where a binary signal *s* is distorted when it is passed through a noisy channel *H*, and should be un-distorted ("equalized") by passing it through an equalizing filter with system transfer function H^{-1}. In order to train the equalizer, the correct signal *s* must be known by the receiver, where the equalizer is trained. But of course, if *s* would be already known, one would not need the communication in the first place... this hen-and-egg problem is often solved by using a predetermined training sequence *s* = *d*. From time to time (especially at the initialization of a transmission), the sender transmits *s* = *d*, which is already known by the receiver and enables it to estimate an inverse channel model. But also while useful communication is taking place, the receiver can continue to train its equalizer, as long as the receiver is successful in restoring the binary signal *s*: in that case, the correctly restored signal *s* can be used for continued training. The overall setup is sketched in Figure 6.8

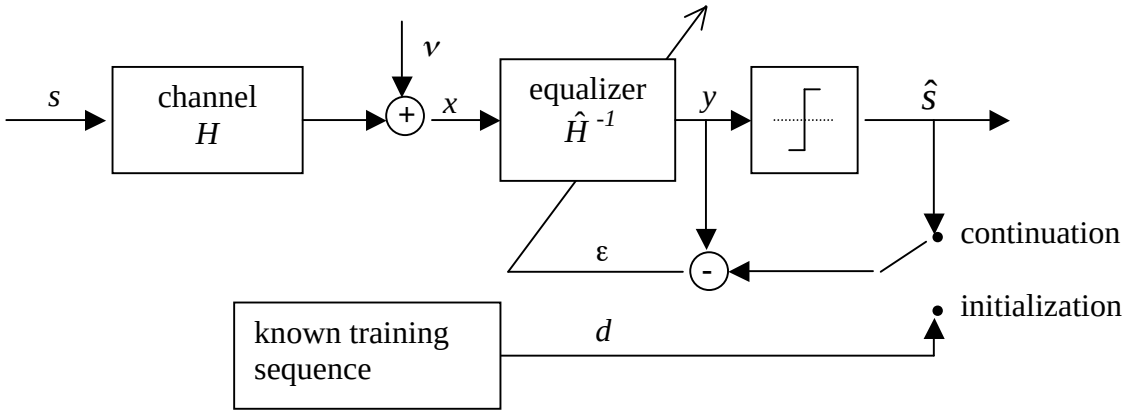


Figure 6.8: Schema of adaptive online channel equalization. Delays are omitted.

Feedback error learning for a composite direct / feedback controller. Pure open-loop control cannot cope with external disturbances to the plant. Furthermore, the simple setup from Fig. 6.6 requires that for training the plant is driven by specially prepared training input,

a condition not desirable in true online applications where the controller has to adapt to the plant continuously while the entire system is operating. The following scheme (proposed by Michael Jordan in a nonlinear control context, using neural networks[7]) trains an open-loop inverse controller in conjunction with the operation of a fixed feedback-controller. The architecture is shown in Fig. 6.9.

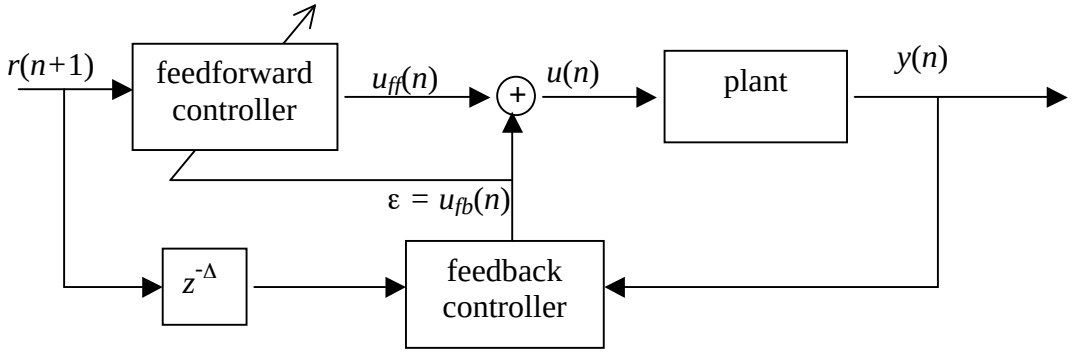


Figure 6.9: Schema of feedback error learning for a composite control system.
Some explanations on this ingenious architecture:

- The control input $u(n)$ is the sum of the outputs $u_{fb}(n)$ of the feedback controller and $u_{ff}(n)$ of the feedforward controller.
- If the feedforward controller works perfectly, the feedback controller detects no discrepancy between the reference r and the plant output y and therefore produces a zero output $u_{fb}(n)$ – that is, the feedforward controller sees zero error ε and does not change.
- If the feedforward controller does not work perfectly, its output $u_{fb}(n)$ acts as an error signal for further adaptation of the feedforward controller. The feedforward controller tries to minimize this "error" – that is, it changes its way to generate output $u_{ff}(n)$ such that the feedback controller's output is minimized, that is, such that $r - y$ is minimized, that is, such that the control improves.
- When the plant characteristics change, or when external disturbances set on, the feedback controller sets on again – as does the further adaptation of the feedforward controller. Thus, situations that cannot be handled by a pure feedforward controller are coped with by the composite architecture, which is always operative.

6.2.3 Interference cancelling, "denoising" (from Farhang-Boroujeny)

Assume that there is a signal $s + \nu_0$ that is an additive mixture of a useful signal s and a noise component ν_0. You want to cancel the interfering component ν_0 from this mixture. Assume further that you also have another signal source ν_1 that correlates strongly with ν_0 but weakly with s. In this situation you may use a denoising scheme as shown in Figure 6.10.

[7] M. I. Jordan, Computational Motor Control, in M.S. Gazzaniga (ed.), The Cognitive Neurosciences, MIT Press 1995, 597-612)

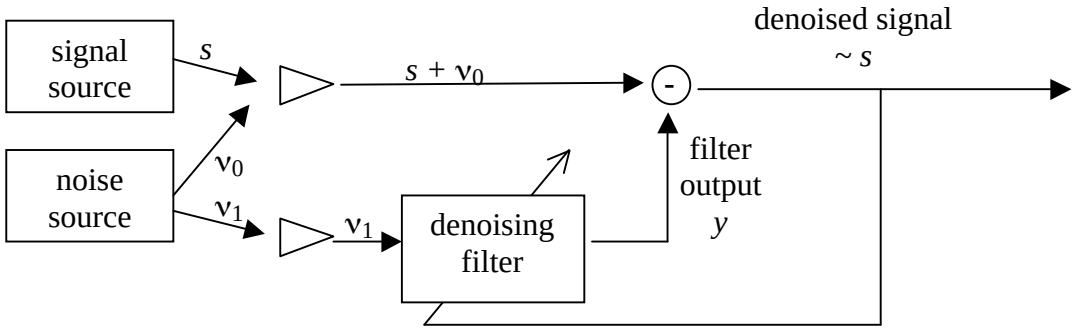


Figure 6.10: Schema of denoising filter.

Some explanations:

- The "error" that the adaptive denoising filter tries to minimize is $s + \nu_0 - y$.
- The only information that the filter has to achieve this is its input ν_1. Because this input is (ideally) not correlated with s, but highly correlated with ν_0, all that the filter can do is to subtract from $s + \nu_0$ whatever it finds correlates in $s + \nu_0$ with ν_1. Ideally, this is ν_0. Then, the residual "error" ξ_{min} would be just s.
- Note that the working principle behind this architecture is just an application of the principle of orthogonality.
- This scheme is interesting (and not just a trivial subtraction of ν_1 from $s + \nu_0$) because the correlation between ν_1 and ν_0 may be complex, involving superposition of delayed versions of ν_0.
- Applications include cleaning up EKG signals (the ν_1 signal corresponds to electrodes that are planted on you at distant positions from the heart), distinguishing the child's heartbeat from the mother's in prenatal diagnosis, cancelling the 50Hz background noise found in many biological recordings, denoising of speech signals. Interference cancelling as explained here is a traditional technique. Today, one might want to employ the more advanced techniques of blind signal separation for similar purposes. But I would not be surprised if most EKG recording devices sold today still use this traditional approach.

6.2.4 Beamforming

I will only briefly mention the fourth traditional application area of adaptive filters. If one has an array of M omnidirectional antennas, at which a mixture of M radio signals x_i arrives, all of the same frequency but coming from different directions, it is desirable in many telecommunication applications to pick out *one* of M incoming signals from all the others, say x_1. This can be done by postprocessing the M antenna signals by a filter that basically cancels the interfering signals x_2, ..., x_M. What is different here as compared to the other applications of adaptive filters considered so far is that here the data vector x_1, ..., x_M used as input to the filter is not temporal but spatial. However, the mathematics remain the same. The name *beamforming* illustrates that by adaptation of its filter, the antenna array forms a "lobe" or "beam", that is an angular segment in the compass circle from which it effectively receives signals while suppressing signal input from directions outside the lobe.

6.3 Gradient descent for finding optimal weights in online adaptive tasks

The solution of the Wiener-Hopf equation provides an offline algorithm to compute optimal weights from a fixed training time series. In practice, however, one often desires an *online* algorithm that incrementally improves the weights. Specifically this is the case when the system that one wishes to model is varying over time. Then, *adaptive* algorithms are needed. The terms "online" and "adaptive" have slightly different meanings. "Adaptive" refers to the circumstance that the target system is time-varying and the model has to *track* the target system. This is typically done by using online algorithms but could, in principle, be done with an offline algorithm, too – over time, one would have to collect training sequences into a memory, and recompute the model from scratch on new training sequences. "Online" refers to algorithms that adapt their weights incrementally using each new data point as it comes in while the filter is being used. That is, at every time n, a set of weights $\mathbf{w}(n)$ is computed, and typically $\mathbf{w}(n) \neq \mathbf{w}(n+1)$. For adaptive system identification tasks, online methods are in most cases more natural, more elegant, computationally cheaper, and more precise.

In this subsection we provide and introduction to the simplest kind of online algorithms. They rest on the idea of *gradient descent*: at each time n, go "downhill" on the performance surface a little bit in the steepest direction, just like a tired mountaineer. We will first treat this task from a theoretical perspective, assuming that the gradient is perfectly known (Subsection 6.3.1), and then describe a practical algorithm that estimates this gradient online. This algorithm, variously known as the *LMS*-algorithm ("least mean square", this name is common in signal processing), as *stochastic gradient descent* (common in machine learning) or as the [*Widrow-Hoff*-] *delta rule* (in the biologically oriented neural network community). This multitude of names indicates that this algorithm has been re-discovered independently many times in different contexts, and it is certainly the simplest and likely the most widely used algorithm in adaptive signal processing. (I re-discovered it myself when I started to work my way into machine learning...) I lean on the treatment given in Farhang-Boroujeny, but any other book on neural networks, pattern recognition or adaptive signal processing will treat this subject, too.

6.3.1 Principles of gradient descent on quadratic performance surfaces

Further properties of the performance surface; normalized coordinates. Our goal in this section is to find online adaptive algorithms that incrementally adapt the weights $\mathbf{w}(n)$ such that the error decreases. Such algorithms (of which there are many) exploit the geometry of the performance surface. Therefore, next we investigate this geometric object more closely.

First we use (6.6) and the Wiener-Hopf equation (6.10) to write in various ways the *expected residual error* $\xi_{\min}$ that we are left with when we have found $\mathbf{w}_{\text{opt}}$:

$$\begin{aligned} \xi_{\min} &= E[(d(n))^2] - 2\mathbf{w}_{\text{opt}}^{\mathsf{T}}\mathbf{p} + \mathbf{w}_{\text{opt}}^{\mathsf{T}}\mathbf{R}\mathbf{w}_{\text{opt}} \\ &= E[(d(n))^2] - \mathbf{w}_{\text{opt}}^{\mathsf{T}}\mathbf{p} = E[(d(n))^2] - \mathbf{w}_{\text{opt}}^{\mathsf{T}}\mathbf{R}\mathbf{w}_{\text{opt}} = E[(d(n))^2] - \mathbf{p}^{\mathsf{T}}\mathbf{R}^{-1}\mathbf{p}. \end{aligned} \tag{6.15}$$

Next we present an alternative version of the error function ξ. Observing that the paraboloid is centered on $\mathbf{w}_{\text{opt}}$, that is has "elevation" $\xi_{\min}$ over the weight space, and that the shape of the paraboloid itself is determined by $\mathbf{w}^{\mathsf{T}}\mathbf{R}\mathbf{w}$, we find that we can rewrite (6.6) as

$$\begin{aligned}\xi &= \xi_{\min} + (\mathbf{w} - \mathbf{w}_{\text{opt}})^{\mathsf{T}} \mathbf{R} (\mathbf{w} - \mathbf{w}_{\text{opt}}) \\ &= \xi_{\min} + \mathbf{v}^{\mathsf{T}} \mathbf{R} \mathbf{v},\end{aligned} \tag{6.16}$$

where we introduced shifted weight coordinates $\mathbf{v} = \mathbf{w} - \mathbf{w}_{\text{opt}}$. Differentiating (6.16) w.r.t. $\mathbf{v}$ yields

$$\frac{\partial \xi}{\partial \mathbf{v}} = \left(\frac{\partial \xi}{\partial v_1} \cdots \frac{\partial \xi}{\partial v_M} \right)^{\mathsf{T}} = 2\mathbf{R}\mathbf{v}. \tag{6.17}$$

From our discussion in Section 4.4 we obtain immediately the following insights. Since $\mathbf{R}$ is symmetric and positive semi-definite, we can write $\mathbf{R} = \mathbf{U}^{\mathsf{T}}\mathbf{D}\mathbf{U} = \mathbf{U}\mathbf{D}\mathbf{U}^{-1}$, where $\mathbf{U}$ contains a set of orthonormal real eigenvectors in its columns and $\mathbf{D}$ is a diagonal matrix containing the corresponding eigenvalues, which are likewise real, and non-negative. Furthermore, the eigenvectors $\mathbf{u}_i$ of $\mathbf{R}$ lie on the central axes of the hyperellipsoid formed by the contour lines of the performance surface (see Fig. 6.2, red arrows). By left-multiplication of the shifted coordinates $\mathbf{v} = \mathbf{w} - \mathbf{w}_{\text{opt}}$ with $\mathbf{U}^{\mathsf{T}}$ we get new *normal coordinates* $\tilde{\mathbf{v}} = \mathbf{U}^{\mathsf{T}}\mathbf{v}$. The coordinate axes of the $\tilde{\mathbf{v}}$ system are in the direction of the eigenvectors of $\mathbf{R}$, and equation (6.17) becomes

$$\frac{\partial \xi}{\partial \tilde{\mathbf{v}}} = 2\mathbf{D}\tilde{\mathbf{v}} = 2(\lambda_1 \tilde{v}_1 \cdots \lambda_M \tilde{v}_M)^{\mathsf{T}}, \tag{6.18}$$

from which we get the second derivatives

$$\frac{\partial^2 \xi}{\partial \tilde{\mathbf{v}}^2} = 2(\lambda_1 \cdots \lambda_M)^{\mathsf{T}}, \tag{6.19}$$

that is, the eigenvalues of $\mathbf{R}$ are (up to a factor of 2) the curvatures of the performance surface in the direction of the central axes of the hyperparabeloid. We will shortly see that the most natural and simple adaptive learning algorithm, the LMS algorithm, depends in its efficiency critically on these curvatures.

The basic formula for taking a small step downhill along the gradient, thereby adapting $\mathbf{w}(n)$ to $\mathbf{w}(n+1)$, is

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \nabla\xi(\mathbf{w}(n)), \tag{6.20}$$

where μ is a *stepsize parameter* and $\nabla\xi(\mathbf{w}(n))$ is the gradient of the performance surface at point $\mathbf{w}(n)$. In typical cases, μ is set to values of 1/100 to 1/1000 – we will later learn to optimize this. We now analyze the convergence properties of the update rule (6.20). We will operate in the normal coordinates $\tilde{\mathbf{v}} = \mathbf{U}^{\mathsf{T}}\mathbf{v}$ (remember $\mathbf{v} = \mathbf{w} - \mathbf{w}_{\text{opt}}$ and $\mathbf{U}^{\mathsf{T}}$ was the matrix containing orthonormal eigenvectors of $\mathbf{R}$; further recall that $\mathbf{R} = \mathbf{U}^{\mathsf{T}}\mathbf{D}\mathbf{U}$ and $\mathbf{D}$ contains the eigenvalues λ_j of $\mathbf{R}$ on its diagonal). By some elementary transformations [use (6.18)] (6.20) turns into

$$\tilde{\mathbf{v}}(n+1) = (\mathbf{I} - 2\mu\mathbf{D})\tilde{\mathbf{v}}(n). \tag{6.21}$$

Because $\mathbf{I} - 2\mu\mathbf{D}$ is diagonal, this can be split up into the components of $\tilde{\mathbf{v}}$, yielding

(6.22) $\tilde{v}_j(n+1) = (1 - 2\mu\lambda_j)\tilde{v}_j(n)$ (j = 1, ..., M).

This is a geometric sequence. If started in $\tilde{v}_j(0)$, one obtains

(6.23) $\tilde{v}_j(n) = (1 - 2\mu\lambda_j)^n \tilde{v}_j(0)$.

The sequence $\mathbf{w}(n)$ converges to $\mathbf{w}_{\text{opt}}$ if $\tilde{v}_j(n)$ converges to zero for all j. (6.23) implies that this happens if and only if $|1 - 2\mu\lambda_j| < 1$ for all j. These inequalities can be re-written as $-1 < 1 - 2\mu\lambda_j < 1$ or equivalently,

(6.24) $0 < \mu < 1/\lambda_j$.

Specifically, we must make sure that $0 < \mu < 1/\lambda_{\max}$, where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{R}$. Depending on the size of μ, the convergence behavior of (6.23) can be grouped in four classes which may be referred to as *overdamped*, *underdamped*, and two types of *unstable*. Figure 6.11 illustrates how $\tilde{v}_j(n)$ evolves in these four classes.

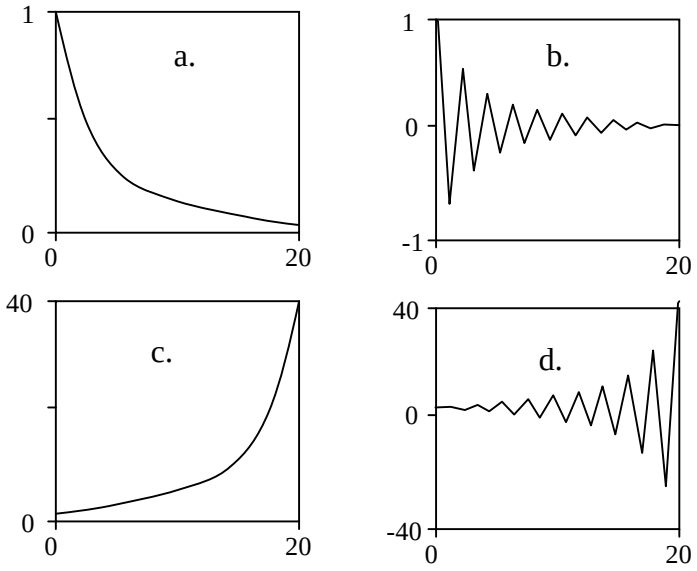


Figure 6.11: The development of $\tilde{v}_j(n)$ [plotted in the y-axis] vs. n [x-axis]. The qualitative behaviour depends on the stepsize parameter μ. **a.** Overdamped case: $0 < \mu < 1/(2\lambda_j)$. **b.** Underdamped case: $1/(2\lambda_j) < \mu < 1/\lambda_j$. **c.** Unstable with $\mu < 0$ and **d.** unstable with $1/\lambda_j < \mu$. All plots start with $\tilde{v}_j(0) = 1$.

We can find an explicit representation of $\mathbf{w}(n)$ if we observe that $\mathbf{w}(n) = \mathbf{w}_{\text{opt}} + \mathbf{v}(n) = \mathbf{w}_{\text{opt}} + \sum_{j=1}^{M} \mathbf{u}_j \tilde{v}_j(n)$, where the $\mathbf{u}_j$ are the orthonormal eigenvectors of $\mathbf{R}$. Inserting (6.23) gives us

(6.25) $$\mathbf{w}(n) = \mathbf{w}_{\text{opt}} + \sum_{j=1}^{M} \tilde{v}_j(0)\, \mathbf{u}_j (1 - 2\mu\lambda_j)^n .$$

This representation reveals that the convergence of $\mathbf{w}(n)$ toward $\mathbf{w}_{\text{opt}}$ is governed by an additive overlay of M exponential terms, each of which describes convergence in the direction of the eigenvectors $\mathbf{u}_j$ and is determined in its convergence speed by λ_j and the stepsize parameter μ. One speaks of the *M modes* of convergence with geometric ratio factors $1 - 2\mu\lambda_j$. If all eigenvalues are roughly equal, convergence rates are roughly identical in the M directions. If however two eigenvalues are very different, say $\lambda_1 << \lambda_2$, and μ is small compared to the eigenvalues, then convergence in the direction of $\mathbf{u}_1$ will be much slower than in the direction of $\mathbf{u}_2$ (see Figure 6.12).

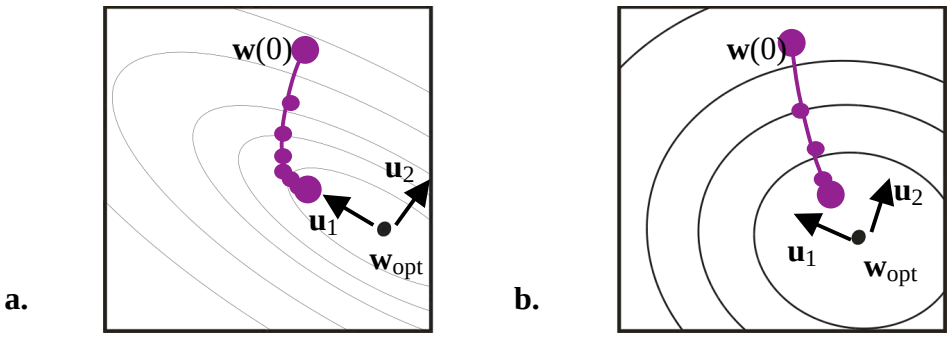


Figure 6.12: Two quite different modes of convergence (**a.**) vs. rather similar modes of convergence (**b.**). Plot shows contour lines of performance surface for two-dimensional weights $\mathbf{w} = (w_1, w_2)$. Violet dotted lines indicate some initial steps of weight evolution, starting from $\mathbf{w}(0)$.

Next we turn to the question how the error ξ evolves over time. Recall from (6.16) that $\xi = \xi_{\text{min}} + \mathbf{v}^{\mathsf{T}}\mathbf{R}\mathbf{v}$, which can be re-written as $\xi = \xi_{\text{min}} + \tilde{\mathbf{v}}^{\mathsf{T}}\mathbf{D}\tilde{\mathbf{v}}$. Thus the error in the n-th iteration is

(6.26) $$\xi(n) = \xi_{\text{min}} + \tilde{\mathbf{v}}^{\mathsf{T}}(n)\mathbf{D}\tilde{\mathbf{v}}(n) = \xi_{\text{min}} + \sum_{j=1}^{M} \lambda_j (1 - 2\mu\lambda_j)^{2n} \tilde{v}_j(0)^2 .$$

For suitable μ (see (6.24)), $\xi(n)$ converges to ξ_{min}. Plotting $\xi(n)$ yields a graph known as *learning curve*. (6.26) reveals that the learning curve is the sum of M decreasing exponentials (plus ξ_{min}). Figure 6.13 shows a three-mode learning curve for the case $\xi_{\text{min}} = 0$, where in **a.** $\xi(n)$ is plotted on a linar scale and in **b.** in a logarithmic scale.

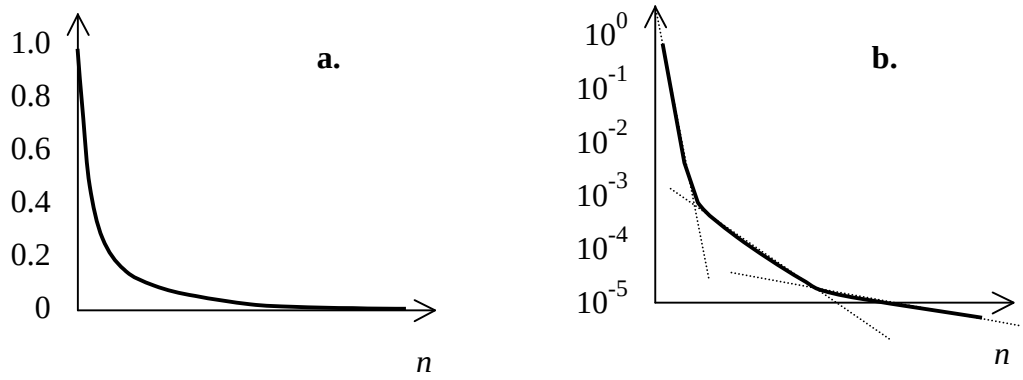


Figure 6.13: a learning curve (case $\xi_{min} = 0$) with three modes of convergence.

Each of the terms $(1-2\mu\lambda_j)^{2n}$ is characterized by a time constant τ_j according to

(6.27) $$(1-2\mu\lambda_j)^{2n} = e^{\frac{-n}{\tau_j}}$$

If $2\mu\lambda$ is close to zero, $\exp(2\mu\lambda)$ is close to $1+2\mu\lambda$ and thus $\ln(1-2\mu\lambda) \approx -2\mu\lambda$. Using this approximation, solving (6.27) for τ_j yields for the *j*-th mode a time constant of

(6.28) $$\tau_j \approx \frac{1}{4\mu\lambda_j}.$$

That is, the convergence rate (i.e. the inverse of the time constant) of the *j*-th mode is proportional to λ_j *for very small* μ.

However, this analysis is meaningless for larger μ. If we want to maximize the speed of convergence, we should use significantly larger μ, as we will presently see. As can be seen from Fig. 6.13 **b.**, the final rate of convergence is dominated by the slowest mode of convergence, which is characterized by the geometrical ratio factor

(6.29) $$\max\{|1-2\mu\lambda_j| \;|\; j=1,...,M\} = \max\{|1-2\mu\lambda_{max}|, |1-2\mu\lambda_{min}|\}.$$

In order to maximize convergence speed, the learning rate μ should be chosen such that (6.29) is minimized. Some elementary considerations reveal that this minimum is attained at $|1-2\mu\lambda_{max}| = |1-2\mu\lambda_{min}|$, which is equivalent to

(6.30) $$\mu_{opt} = \frac{1}{\lambda_{min}+\lambda_{max}}.$$

For this optimal learning rate, $1-2\mu_{opt}\lambda_{min}$ is positive and $1-2\mu_{opt}\lambda_{max}$ is negative, corresponding to the overdamped and underdamped cases shown in Figure 6.11. However, the two modes converge at the same speed (and all other modes are faster). Concretely, the optimal speed of convergence is given by the geometric ratio facto

(6.31) $$\beta = 1 - 2\mu_{opt}\lambda_{\min} = \frac{\lambda_{\max}/\lambda_{\min} - 1}{\lambda_{\max}/\lambda_{\min} + 1},$$

where the last term is found by substituting (6.30). This has a value between 0 and 1. There are two extreme cases: if $\lambda_{\max} = \lambda_{\min}$, then $\beta = 0$ and we have convergence in a single step. As the ratio $\lambda_{\max}/\lambda_{\min}$ increases, β approaches 1 and the convergence slows down toward stillstand. The ratio $\lambda_{\max}/\lambda_{\min}$ thus plays a fundamental role in limiting the convergence speed of steepest descent algorithms. It is called the *eigenvalue spread.*

The eigenvalue spread is closely related to the spectral properties of the input process *x*. We can only sketch the connection here. Recall that for a stationary stochastic process $\{x(n)\}$, $\phi_{xx}(k) = E[x(n)\, x(n-k)]$ is the *autocorrelation function* and $\Phi_{xx}(\omega) = \sum_{k=-\infty}^{k=\infty} \phi_{xx}(k) e^{-i\omega k}$ is its *power spectral density* (or simply *power spectrum* or just *spectrum*). For each frequency $-\pi \le \omega < \pi$, $\Phi_{xx}(\omega)$ gives the squared contribution of that frequency (the *energy* of that frequency) to *x*. It can be shown (details in Farhang-Boroujeny p. 97*ff*) that

(6.32) $$\begin{aligned} \lambda_{\min} &\ge \min_{\omega} \Phi_{xx}(\omega), \\ \lambda_{\max} &\le \max_{\omega} \Phi_{xx}(\omega). \end{aligned}$$

Thus, if *x* has a flat power spectrum (i.e., $\min_{\omega} \Phi_{xx}(\omega) \approx \max_{\omega} \Phi_{xx}(\omega)$), then $\lambda_{\max}/\lambda_{\min} \approx 1$ and we can expect fast convergence in steepest descent algorithms – and conversely, if *x* has a very uneven power distribution, steepest descent algorithms are likely to perform poorly. For this reason, it helps to speed up convergence if the input signal *x* is first passed through a *whitening filter* that flattens its power spectrum, before it is used as input to an adaptive filter.

6.3.2 The LMS algorithm

The update formula (6.20) for steepest gradient descent, $\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \nabla\xi(\mathbf{w}(n))$, is not useful in practice because the gradient $\nabla\xi(\mathbf{w}(n))$ is not known. Remember that $\xi = E[\varepsilon^2]$ is the expected squared error of filter output *y* vs. teacher *d*. Given filter weights $\mathbf{w}(n)$, we need to estimate the expected squared error $\xi(\mathbf{w}(n))$ of the filter output generated by the filter with weights $\mathbf{w}(n)$ vs. the teacher *d.* At first sight, what one needs to estimate an expected squared error is *time* – namely, to observe the ongoing filtering with weights $\mathbf{w}(n)$ for some time and then approximate $\xi(\mathbf{w}(n)) = E[\varepsilon^2(\mathbf{w}(n))]$ by averaging over the errors seen in this observation interval. But we don't have this time – because we want to update $\mathbf{w}(n)$ at every time step *n.* One ruthless way out of this impasse is to just use the *momentary* squared error as an approximation to its *expected* value, that is, use

(6.33) $$\xi(\mathbf{w}(n)) \approx \varepsilon^2(\mathbf{w}(n)) = (d(n) - \mathbf{w}^{\mathsf{T}}(n)\, \mathbf{x}(n))^2.$$

Using this most brutal possible approximation, the update formula (6.20) for steepest gradient descent becomes

(6.34) $$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \nabla\varepsilon^2(\mathbf{w}(n)),$$

We can compute $\nabla\varepsilon^2(\mathbf{w}(n))$ as follows:

$$
\begin{aligned}
\nabla\varepsilon^2(\mathbf{w}(n)) &= 2\varepsilon(\mathbf{w}(n))\,\nabla\varepsilon(\mathbf{w}(n)) = \\
&= 2\varepsilon(\mathbf{w}(n))\left[\frac{\partial\varepsilon(\mathbf{w}(n))}{\partial w_1}\cdots\frac{\partial\varepsilon(\mathbf{w}(n))}{\partial w_M}\right]^{\mathrm{T}} \\
&= -2\varepsilon(\mathbf{w}(n))\left[\frac{\partial y(n)}{\partial w_1}\cdots\frac{\partial y(n)}{\partial w_M}\right]^{\mathrm{T}} \quad [\textit{use } \varepsilon(n) = d(n) - y(n)] \\
&= -2\varepsilon(\mathbf{w}(n))\left[x(n)\cdots x(n-M+1)\right]^{\mathrm{T}} \\
&= -2\varepsilon(n)\,\mathbf{x}(n)
\end{aligned}
\tag{6.35}
$$

where in the last step we simplified the notation $\varepsilon(\mathbf{w}(n))$ to $\varepsilon(n)$. Inserting this into (6.34) gives

$$
\boxed{\mathbf{w}(n+1) = \mathbf{w}(n) + 2\,\mu\,\varepsilon(n)\,\mathbf{x}(n),}
\tag{6.36}
$$

which is the weight update formula of the LMS algorithm. This formula can hardly be beaten in simplicity and computational efficiency! For completeness, here are all the computations needed to carry out one full step of online filtering & weight adaptation with the LMS algorithm:

(4) read in input and compute output: $y(n) = \mathbf{w}^{\mathrm{T}}(n)\,\mathbf{x}(n)$,
(4) compute current error: $\varepsilon(n) = d(n) - y(n)$,
(4) compute weight update: $\mathbf{w}(n+1) = \mathbf{w}(n) + 2\,\mu\,\varepsilon(n)\,\mathbf{x}(n)$.

One fact about the LMS algorithm should always be kept in mind: being a stochastic version of steepest gradient descent, the LMS algorithm inherits the problems connected with the power spectrum of the input process x. It this power spectrum is very unevenly distributed, the LMS algorithm is likely not to work satisfactorily. (As an aside, in my working with neural networks, I tried out learning algorithms related to LMS. But the input signal to this learning algorithm had an eigenvalue spread of 10^{14} to 10^{16}, so the beautifully simple LMS algorithm was entirely useless.)

Because of its eminent usefulness (*if* the input signal has a reasonably flat power spectrum), the LMS algorithm has been analysed in minute detail. We conclude this section by reporting the most important insights without mathematical derivations. At the same time we introduce some of the standard vocabulary used in the field of adaptive signal processing.

We assume that x and d are stationary processes. The evolution $\mathbf{w}(n)$ of weights is now also a stochastic process, because the LMS weight update depends on the stochastic vector $\mathbf{x}(n)$. One interesting question is how fast the LMS algorithm converges in comparison with the ideal steepest gradient descent "algorithm" $\tilde{\mathbf{v}}(n+1) = (\mathbf{I} - 2\mu\mathbf{D})\tilde{\mathbf{v}}(n)$ from (6.21). Because we now have a stochastic update, the vectors $\tilde{\mathbf{v}}(n)$ become random variables and one can only speak about their *expected* value $E[\,\tilde{\mathbf{v}}(n)\,]$ at time n. [Intuitively, this value would be obtained if many (infinitely many in the limit) training runs ω of the adaptive filter would be carried out and in each of these runs, the value of $\tilde{\mathbf{v}}(n)$ at time n would be taken, and an average would be formed over all these $\tilde{\mathbf{v}}(n)$.]. The following can be shown (using some additional assumptions, namely, that μ is small and that the signal x has no substantial autocorrelation for time spans larger than M):

(6.37) $$E[\tilde{\mathbf{v}}(n+1)] = (\mathbf{I} - 2\mu\mathbf{D})\, E[\tilde{\mathbf{v}}(n)]\,.$$

Rather to our surprise, if the LMS algorithm is used, the weights converge – on average – as fast to the optimal weights as when the ideal algorithm (6.21) is employed. Figure 6.14 depicts an overlay of the deterministic development of weights according to (6.21) (grayish pink line) with one run of the stochastic gradient descent according to the LMS algorithm.

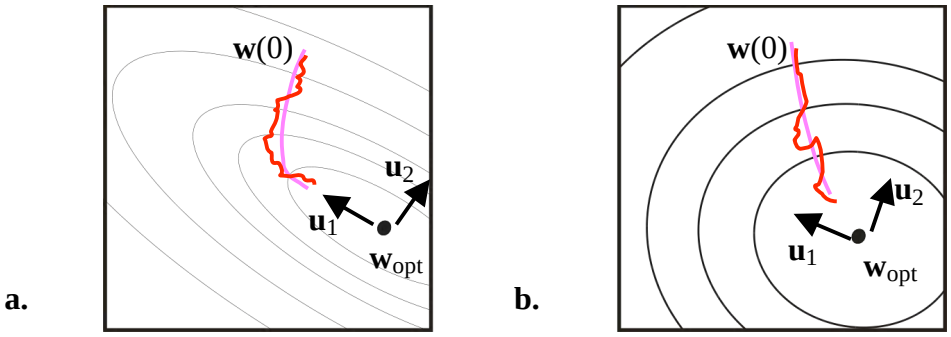


Figure 6.14: Illustrating the similar performance on average of deterministic (pink) and stochastic gradient descent.

The fact that on average the weights converge to the optimal weights (cf. (6.38)) by no means implies that $\xi(n)$ converges to $\xi_{\min}$. To see why, assume that at some time n, the LMS algorithm actually would have found the correct optimal weights, that is, $\mathbf{w}(n) = \mathbf{w}_{\text{opt}}$. What would happen next? Well, due to the random weight adjustment, these optimal weights would become misadjusted again in the next time step! So the best one can hope for asymptotically is that the LMS algorithms lets the weights $\mathbf{w}(n)$ jitter randomly in the vicinity of $\mathbf{w}_{\text{opt}}$. But this means that the effective best error that can be achieved by the LMS algorithm in the asymptotic limit is not $\xi_{\min}$ but $\xi_{\min} + \xi_{\text{excess}}$, where ξ_{excess} comes from the random scintillations of the weight update. It is intuitively clear that ξ_{excess} depends on the stepsize μ – the larger μ, the larger we expect ξ_{excess} to become. The absolute size of the excess error

ξ_{excess} is not so interesting as is the ratio $\mathcal{M} = \xi_{excess}/\xi_{min}$, the relative size the excess error w.r.t. the minimal error. The quantity $\mathcal{M}$ is called the *misadjustment* and describes what fraction of the *residual error* $\xi_{min} + \xi_{excess}$ can be attributed to the random oscillations effected by the stochastic weight update [i.e., ξ_{excess}], and what fraction is inevitably due to inherent limitations of the filter itself [i.e., ξ_{min}]. Notice that ξ_{excess} can in principle be brought to zero by tuning down μ to zero – however, that would be at odds with the objective of fast convergence.

Under some assumptions (notably, small $\mathcal{M}$) and using some approximations (cf. Farhang-Boroujeny, Section 6.3), the misadjustment can be approximated by

(6.39) $$\mathcal{M} \approx \mu \, trace(\mathbf{R}),$$

where the *trace* of a matrix is the sum of its diagonal elements. The misadjustment is thus proportional to the stepsize and can be steered by setting the latter, if *trace*(**R**) is known. Fortunately, *trace*(**R**) can be estimated online from the sequence $x(n)$ simply and robustly [how? – easy exercise].

This is an important insight if one wishes to track a nonstationary system adaptively while maintaining a given misadjustment. In this situation, one commits oneself to a fixed level of misadjustment, maintains an online estimate of *trace*(**R**), and uses $\mu = \mathcal{M} / trace(\mathbf{R})$.

Another issue that one has always to be concerned about in online adaptive signal processing is stability. We have seen in the treatment of the ideal case (Section 6.3.1) that the stepsize μ must not exceed $1/\lambda_{max}$ in order to guarantee convergence. But this result does not directly carry over to our stochastic version of gradient descent, because it does not take into account the stochastic jitter of the gradient descent, which is intuitively likely to be harmful for convergence. Furthermore, the value of λ_{max} cannot be estimated robustly from few data points in a practical situation. Using again middle-league maths and several approximations, in the book of Farhang-Boroujeny the following upper bound for μ is derived:

(6.40) $$\mu \leq 1 / (3 \, trace(\mathbf{R}))$$

If this bound is respected, the LMS algorithm converges stably.

In practical applications, one often wishes to achieve an initial convergence that is as fast as possible: this can be done by using μ close to the stabilty boundary from (6.40). After some time, when a reasonable degree of convergence has been attained, one wishes to optimize the mismatch; then one switches into a control mode where μ is adapted dynamically according to (6.39).

The LMS algorithm is since 40 years the workhorse of adaptive signal processing and numerous refinements and variants have been developed. Here are some:

4) An even simpler stochastic gradient descent algorithm than LMS uses only the sign of the error in the update, i.e. uses $\mathbf{w}(n+1) = \mathbf{w}(n) + 2 \mu \, \text{sign}(\varepsilon(n)) \, \mathbf{x}(n)$. If μ is a power of 2, this algorithm does not need a multiplication (a shift does it then) and is suitable for very high throughput hardware implementations. There exist yet other "sign-simplified" versions of LMS [cf. Farhang-Boroujeny p. 169]

5) Online stepsize adaptation: at every update use a locally adapted stepsize $\mu(n) \approx 1/(\mathbf{x}^T(n)\,\mathbf{x}(n))$. This is called "Normalized LMS" or "NLMS". In practice this pure NLMS is apt to run into stability problems; a safer version is $\mu(n) \approx \mu_0/[\mathbf{x}^T(n)\,\mathbf{x}(n) + \psi]$, where μ_0 and ψ are hand-tuned constants [Farhang-B. p. 172]. In my own experience, normalized LMS sometimes works wonders in comparison with standard LMS.
6) Include a whitening mechanism into the update equation: $\mathbf{w}(n+1) = \mathbf{w}(n) + 2\,\mu\,\mathbf{R}^{-1}\,\varepsilon(n)\,\mathbf{x}(n)$. This "Newton-LMS" algorithm has a single mode of convergence, but a problem is to obtain a good estimate of $\mathbf{R}^{-1}$. [Farhang-B. p. 210]
7) Block implementations: for very long filters (say, $M > 10{,}000$) and high update rates, even LMS may become too slow. Various computationally efficient "block LMS" algorithms have been designed in which the input stream is partitioned into blocks, which are processed in the frequency domain and yield weight updates after every block only ["block LMS", cf. Farhang-B. p. 247ff].

To conclude this section, it should be said that besides LMS algorithms there is another major class of online adaptive algorithms for tapped delay line filters, namely, *recursive least squares* (RLS) filters. RLS algorithms are not steepest gradient-descent algorithms; in fact, the background metaphor of RLS is not to minimize ξ but to minimize the error $\zeta(n) = \sum_{i=1}^{n}(d(i) - y(i))^2$, so the performance surface we know from LMS plays no role for RLS. The main advantages and disadvantages of LMS vs. RLS are:

6) LMS has computational cost $O(M)$, where M is filter length; RLS has $O(M^2)$. Also the space complexity of RLS is an issue for long filters because it is $O(M^2)$.
7) LMS is numerically robust, RLS is plagued by numerical stability problems.
8) RLS has a single mode of convergence and converges faster than LMS, *much* faster when the input signal is highly coloured.
9) RLS is more complicated than LMS and thus more difficult to implement.
10) In applications where fast tracking of highly nonstationary systems is required, LMS may have better tracking performance than RLS.

The RLS class of algorithms has been boosted by the development of *fast RLS* algorithms which reach a linear time complexity in the order of $O(20\,M)$ [Farhang-B. Section 13].

7 A closer look at the bias-variance dilemma

In this short section we will give a formal treatment of the bias-variance theme. First we will see how the relatively young *statistical learning theory*[8] (SLT) addresses this problem. Then we will take a more traditional stance and see how the generalization error of any learning method is made of two components, the *approximation error* (squared "bias") and an *estimation error* ("variance"). The approximation error captures how close the best possible model in some model class comes to the target function; the variance measures how strongly the learnt models adapt themselves to the random variations of individual training data sets. I use material from Section 9.1 of the Bishop book and from the texts indicated in the footnotes.

We consider the following situation of learning a regression model. We are given a probability space (Ω, F, P), a random variable X with values $\mathbf{x}$ in $\mathbb{R}^k$ and a random variable D with values d in $\mathbb{R}$. Pairs $(X(\omega), D(\omega)) = (\mathbf{x}, d)$ are argument-value pairs of some stochastic function which we want to learn from such pairs. From an eagle's perspective, the task of statistical learning is to estimate a function $\hat{f}$ (the *model* that we learn) which provides the smallest possible value of the average error R that we make when we use functions f to predict d from $\mathbf{x}$,

$$(7.1) \qquad R(f) = \int_{\Omega} (f(X(\omega)) - D(\omega))^2 \, dP(\omega) = \int_{\Re^k \times \Re} (f(\mathbf{x}) - d)^2 \, dP_{X \times D}(\mathbf{x}, d)$$

$R(f)$ is called the *risk* in statistical learning theory. The risk can be interpreted as a generalization error or expected test error. It is not possible to use (7.1) for minimizing the risk because the joint distribution $P_{X \times D}$ is unknown. All that is available for learning is a finite sample of N instances of pairs $Data = ((\mathbf{x}_i, d_i))_{i = 1,\ldots,N}$ – well known to us as training data. A straightforward way to estimate $\hat{f}$ is to minimize the training error $R_{\text{emp}}(f)$, which is called *empirical risk* in statistical learning theory:

$$(7.2) \qquad R_{\text{emp}}(f) = \frac{1}{N} \sum_{i=1}^{N} (f(\mathbf{x}_i) - d_i)^2$$

We know from all our experience that a brute-force minimization of $R_{\text{emp}}(f)$ is likely to succeed perfectly, yielding a model $\hat{f}$ that has zero empirical risk – but generalizes poorly because we just (over)fitted the training data. SLT is a rigorous mathematical account of this situation. One main result gives bounds on the risk that connect the sample size N to the model complexity of f (for the time being, think of model complexity as the number of parameters available to tune f).

SLT assumes that models f are selected (by learning) from some family – for instance, the family of linear neural networks, the family of linear neural networks with k neurons, or whatever. Formally, such a family of potential models is written as $(f_\alpha)_{\alpha \in \Lambda}$. The risk of a

[8] Recommended textbook: Vladimir N. Vapnik, The Nature of Statistical Learning Theory (Second Edition). Springer Verlag 1999[2]. Recommended introductory paper (online via IRC): V. N. Vapnik, An Overview of Statistical Learning Theory. IEEE Transactions on Neural Networks 10(5), 1999

particular function f_α from such a family is also written as $R(\alpha)$, the empirical risk as $R_{emp}(\alpha)$. A fundamental idea in SLT is to characterise such a family of candidate functions by their "expressiveness", giving a single characteristic quantity that captures how "rich" the family $(f_\alpha)_{\alpha\in\Lambda}$ is. There are several such characteristics, but the most widely used is the *Vapnik-Chervonenkis(VC)-dimension*. Here is the definition of the VC dimension for a family of functions, for two cases: when the functions are used for binary classification tasks and when they are used for regression tasks.

Definition 7.1 (VC dimension for indicator functions). Let $(f_\alpha)_{\alpha\in\Lambda}$ be a family of indicator functions (that is, 0-1-valued functions) on some vector space V. Then the VC-dimension of $(f_\alpha)_{\alpha\in\Lambda}$ is the maximal number h of vectors $z_1, ..., z_h$ that can be *shattered* by functions from $(f_\alpha)_{\alpha\in\Lambda}$, that is, for each of the 2^h possible ways of assigning the vectors to two classes C_1 and C_2, there exists one function from the family that assigns a value of 0 to the vectors in C_1 and a value of 1 to the vectors in C_2. If any number of vectors can be shattered, $h = \infty$.

Definition 7.2 (VC dimension for real-valued functions). Let $(f_\alpha)_{\alpha\in\Lambda}$ be a family of real-valued functions on some vector space V. Consider the family of indicator functions $(I_{\alpha,\beta})_{\alpha\in\Lambda,\,\beta\in\tilde{\mathbb{N}}}$ obtained from $(f_\alpha)_{\alpha\in\Lambda}$ by putting $I_{\alpha,\beta}(\mathbf{x}) = 0$ if $f_\alpha(\mathbf{x}) - \beta < 0$, else 1. Then the VC-dimension h of $(f_\alpha)_{\alpha\in\Lambda}$ is the VC-dimension of $(I_{\alpha,\beta})_{\alpha\in\Lambda,\,\beta\in\tilde{\mathbb{N}}}$.

Example 1. If $(f_\alpha)_{\alpha\in\Lambda}$ is the family of lines in the plane [more precisely, the family of indicator functions that are 0 on one side of a line], then $h = 3$, because 3 points in the plane can be separated into all possible two-class partitions by lines (Fig. 7.1 left) whereas this is not possible for 4 points (Fig. 7.1 right).

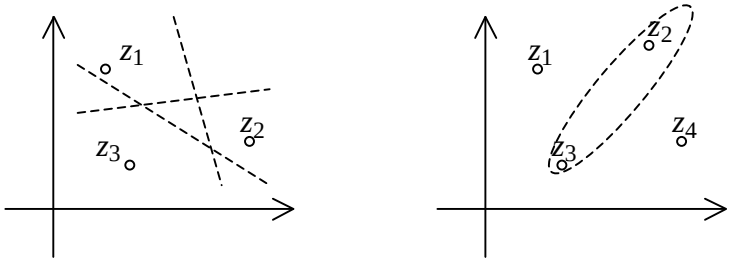


Figure 7.1: Three points z_1, z_2, z_3 can be shattered in the plane by lines, but with four points there are always two (here: z_2 and z_3) that cannot be separated from the other two by lines.

Example 2. Let H be the Heaviside step function (recall: $H(x) = 0$ if $x < 0$, else 1). Consider the set of *linear indicator functions*

$$f_\alpha((x_1,...,x_k)) = H\left[\sum_{i=1}^{k}\alpha_i x_i + \alpha_0\right] \tag{7.3}$$

on $\mathbb{R}^k$. This set of functions has VC dimension $h = n+1$.

Example 3. The VC dimension of the set of linear functions

(7.4) $$f_\alpha((x_1,...,x_k)) = \sum_{i=1}^{k} \alpha_i x_i + \alpha_0$$

on $\mathbb{R}^k$ also has VC dimension $h = n+1$.

Note that in these examples, the number of free parameters $\alpha_0,..., \alpha_k$ equals the VC-dimension, in accordance with our old intuition that the complexity, or expressiveness, of a class of models scales with the number of tuneable parameters. In general, however, this correspondence need not hold:

Example 3. Let Λ be the set $1\{0,1\}^*$ of all binary strings starting with a 1, interpreted as integer numbers written in base 2. For $\alpha \in \Lambda$, consider the function $f_\alpha: \mathbb{R} \to \{0,1\}$ defined by

(7.5) $$f_\alpha(x) = \begin{cases} 0, & \text{if the remainder of } \alpha / x \text{ (in base 2) starts with a 0} \\ 1, & \text{if the remainder of } \alpha / x \text{ starts with a 1} \end{cases}$$

With this set of indicator functions, we can shatter k points $z_1 = 10$, $z_2 = 100$, ..., $z_k = 2^k$, (in binary representation), for any k. To see why, consider an example where $k = 4$ and we want to find an indicator function that assigns z_1, z_2 and z_4 to class 2, z_3 to class 1. We choose $\alpha = 1011$ and find $\alpha/z_1 = 101.1$, $\alpha/z_2 = 10.11$, $\alpha/z_3 = 1.011$, $\alpha/z_4 = 0.1011$, that is, $f_\alpha(z_1) = f_\alpha(z_2)$ $f_\alpha(z_4) = 1$ and $f_\alpha(z_3) = 0$. It becomes clear from this example how we just exploit a binary shift operation to code arbitrary class memberships. Because this works for any k, the VC dimension of this family is infinite – although we only have a single free parameter, α.

The VC dimension is called called the *capacity* of a family of models. An important contribution of SLT is that by the VC dimension / capacity it has found a rigorous and productive method to quantify what we have earlier in the lecture called the complexity (or expressiveness) of a class of models, and what we intuitively related to the number of tuneable parameters. One lesson of SLT is that the sheer number of free parameters in a model family is not always an appropriate measure of the family's modelling capacity.

Equipped with the capacity h, we can now state a fundamental result of SLT, which gives an upper bound on the total risk $R(\alpha)$:

Theorem 7.1 (structural risk minimization principle[9]). The total risk $R(\alpha)$ is bounded by

(7.6) $$R(\alpha) \le R_{\text{emp}}(\alpha) + \phi\left(\frac{h}{N}, \frac{\log(\eta)}{N}\right)$$

with a probability of at least $1-\eta$, where the *confidence term* ϕ is defined by

(7.7) $$\phi\left(\frac{h}{N}, \frac{\log(\eta)}{N}\right) = \sqrt{\frac{h\left(\log\frac{2N}{h} + 1\right) - \log(\eta/4)}{N}}\,.$$

[9] given here as presented in B. Schölkopf, Support Vector Learning, GMD-Bericht Nr. 287, R. Oldenbourg Verlag 1997

Here, N is the size of the training data set and h is the VC-dimension of the family $(f_\alpha)_{\alpha\in\Lambda}$.

The bound in (7.6) deserves some comment. It is intended as a guide in situations where we have small training samples, where "small" means that the ratio N/h is small, say, $N/h < 20$. In this condition, the confidence term increases with h. If we wish to control the risk we can adjust two quantities, the empirical risk and the confidence term. The empirical risk becomes smaller when we fit our training data better – which we can achieve by increasing h, that is, use more complex models. However, by increasing h at the same time we increase the confidence term. Thus we fare best at some compromise value of h.

Concretely, SLT proposes to do the following. Instead of considering a single family $(f_\alpha)_{\alpha\in\Lambda}$, a sequence of families $(f^n_\alpha)_{\alpha\in\Lambda_n}$ is considered, such that the corresponding capacities h_n form an increasing sequence $h_1 \le h_2 \le \ ... \le h_n \le ...$. This could, for instance, be achieved by considering families of neural networks with increasing numbers of neurons. The capacity is used as a control parameter to optimize the final risk, that is, to minimize the generalization error. One considers the sum of the empirical risk and the confidence term, according to (7.6), and selects that h_i which makes this minimal. Figure 7.2 shows the error curves.

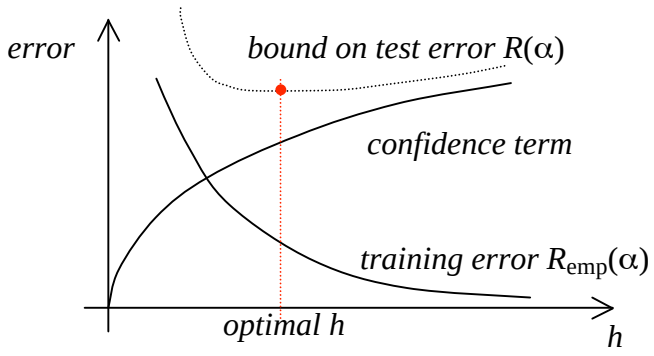


Figure 7.2 Optimal model capacity as a compromise between conflicting demands of small training error vs. small confidence term.

This principle for defining and finding an optimal tradeoff between small training error and a small confidence term is called the *principle of structural risk minimization* in SLT. It can be regarded as one way to deal with the bias-variance dilemma. SLT offers numerous concrete techniques for various types of learning problems to implement the principle of structural risk minimization. The most conspicuous contribution of SLT is, however, that it gives a theoretical foundation for *support vector machines*, a relatively recent technique for learning classification functions. From a SLT perspective, support vector machines are distinguished by the fact that they have a huge number of tuneable parameters but a small VC dimension (just the contrary of our binary shift example 3 from above!). The huge number of tuneable parameters brings with it a small training error, which is however not bought at the expense of a bad generalization error, because the confidence term can be kept small due to the small h.

After this glimpse on SLT we re-address the bias-variance dilemma from a more traditional angle. We will finally explain the origin of the term "bias-variance"!

Without proof we note the following, intuitively plausible fact. Among all functions f that we may consider, the risk $R(f) = \int (f(\mathbf{x}) - d)^2 \, dP_{X\times D}(\mathbf{x}, d)$ is minimized by the function

(7.8) $$f_{\text{minrisk}}(\mathbf{x}) = E_P[D \mid X = \mathbf{x}] = \int_{\Re} d \; dP_{D|X=\mathbf{x}}(d),$$

that is, the expected value of d under condition $\mathbf{x}$. Here $P_{D|X=\mathbf{x}}$ is the conditional distribution of d under hypothesis $\mathbf{x}$. We will use shorthand $\langle d \mid \mathbf{x} \rangle$ for $E_P[D \mid X = \mathbf{x}]$. Note that $\langle d \mid \mathbf{x} \rangle$ is just a function of $\mathbf{x}$. We use subscript P in E_P to indicate that the expectation is computed w.r.t. a conditional probability measure that is derived from the probability space (Ω, F, P).

Let us analyse the learning situation. Statistically, a learning algorithm is an estimator that gets $Data = ((\mathbf{x}_i, d_i))_{i=1,\ldots,N}$ as input and returns an estimate $\hat{f}$. We can consider this estimator as a random variable from a probability space (Ω', F', P'), whose elements ω' are events of drawing a sample $((\mathbf{x}_i, d_i))_{i=1,\ldots,N}$, so we should correctly write $((\mathbf{x}_i, d_i))_{i=1,\ldots,N}(\omega')$ or $((\mathbf{x}_i(\omega'), d_i(\omega')))_{i=1,\ldots,N}$. The estimates $\hat{f}$ are also random variables over this probability space, and we should correctly write $\hat{f}(\omega')$ to denote the function obtained from learning, and $\hat{f}(\omega')(\mathbf{x})$ to denote the value on argument $\mathbf{x}$ of this function.

Now let us fix some $\mathbf{x}$ and ask by how much $\hat{f}(\mathbf{x})$ deviates, on average and in the sqared error sense, from the theoretical optimal function $\langle d \mid \mathbf{x} \rangle$. This expected error is

(7.9) $$E_{P'}[(\hat{f}(\mathbf{x}) - \langle d \mid \mathbf{x} \rangle)^2] = \int_{\Omega'} (\hat{f}(\omega)(\mathbf{x}) - \langle d \mid \mathbf{x} \rangle)^2 \, dP'.$$

We can learn more about this error if we re-write $(\hat{f}(\mathbf{x}) - \langle d \mid \mathbf{x} \rangle)^2$ as follows:

(7.10)
$$\begin{aligned}(\hat{f}(\mathbf{x}) - \langle d \mid \mathbf{x} \rangle)^2 &= (\hat{f}(\mathbf{x}) - E_{P'}[\hat{f}(\mathbf{x})] + E_{P'}[\hat{f}(\mathbf{x})] - \langle d \mid \mathbf{x} \rangle)^2 \\ &= (\hat{f}(\mathbf{x}) - E_{P'}[\hat{f}(\mathbf{x})])^2 + (E_{P'}[\hat{f}(\mathbf{x})] - \langle d \mid \mathbf{x} \rangle)^2 \\ &\quad + 2(\hat{f}(\mathbf{x}) - E_{P'}[\hat{f}(\mathbf{x})])(E_{P'}[\hat{f}(\mathbf{x})] - \langle d \mid \mathbf{x} \rangle).\end{aligned}$$

If we now take the expectation $E_{P'}$ on both sides, we see that the third term on the r.h.s. vanishes and we get

(7.11) $$E_{P'}[(\hat{f}(\mathbf{x}) - \langle d \mid \mathbf{x} \rangle)^2] = \underbrace{(E_{P'}[\hat{f}(\mathbf{x})] - \langle d \mid \mathbf{x} \rangle)^2}_{(\text{bias})^2} + \underbrace{E_{P'}[(\hat{f}(\mathbf{x}) - E_{P'}[\hat{f}(\mathbf{x})])^2]}_{\text{variance}}.$$

The two components of this error are conventionally named the bias and the variance contribution to the error $E_{P'}[(\hat{f}(\mathbf{x}) - \langle d \mid \mathbf{x} \rangle)^2]$. The bias measures how on average the learning result $\hat{f}(\mathbf{x})$ differs from the optimal value $\langle d \mid \mathbf{x} \rangle$. The bias measures how strongly the average learning result deviates from the optimal value; thus is indicates a systematic error component. The variance measures how strongly the learning results $\hat{f}(\mathbf{x})$ vary around their mean $E_{P'}[\hat{f}(\mathbf{x})]$; thus this is an indication of how strongly the particular training data sets induce variations on the learning result. Note that the bias and variance shown in (7.11) are

functions of **x**. By integrating over **x**, one can obtain the average values for the bias and variance:

$$\begin{aligned} (\text{bias})^2 &= E_P[E_{P'}[\hat{f}(\mathbf{x})] - \langle d \mid \mathbf{x} \rangle)^2]] = \int_{\Re^k} E_{P'}[\hat{f}(\mathbf{x})] - \langle d \mid \mathbf{x} \rangle)^2 \, dP_X \\ \text{variance} &= E_P[E_{P'}[(\hat{f}(\mathbf{x}) - E_{P'}[\hat{f}(\mathbf{x})])^2]] = \int_{\Re^k} E_{P'}[(\hat{f}(\mathbf{x}) - E_{P'}[\hat{f}(\mathbf{x})])^2] dP_X . \end{aligned} \tag{7.12}$$

8 Multilayer feedforward networks

We saw in Section 5 that single-layer neural networks can only compute linear decision boundaries (unless they are equipped with preprocessing filters). In this section we will introduce multi-layer neural networks. We will first discuss their representational capacity and then describe a famous gradient-descent learning algorithm for weight-optimization in such networks, the *backpropagation* algorithm. I lean heavily on Section 4 of the Bishop book.

8.1 Structure and representational capacity of multilayer networks

One way to boost the power of our single-layer networks from Section 5 is to add more "hidden" layers of neurons between the layer of input neurons and the layer of output neurons. Fig. 7.1 shows such a general *layered* architecture.

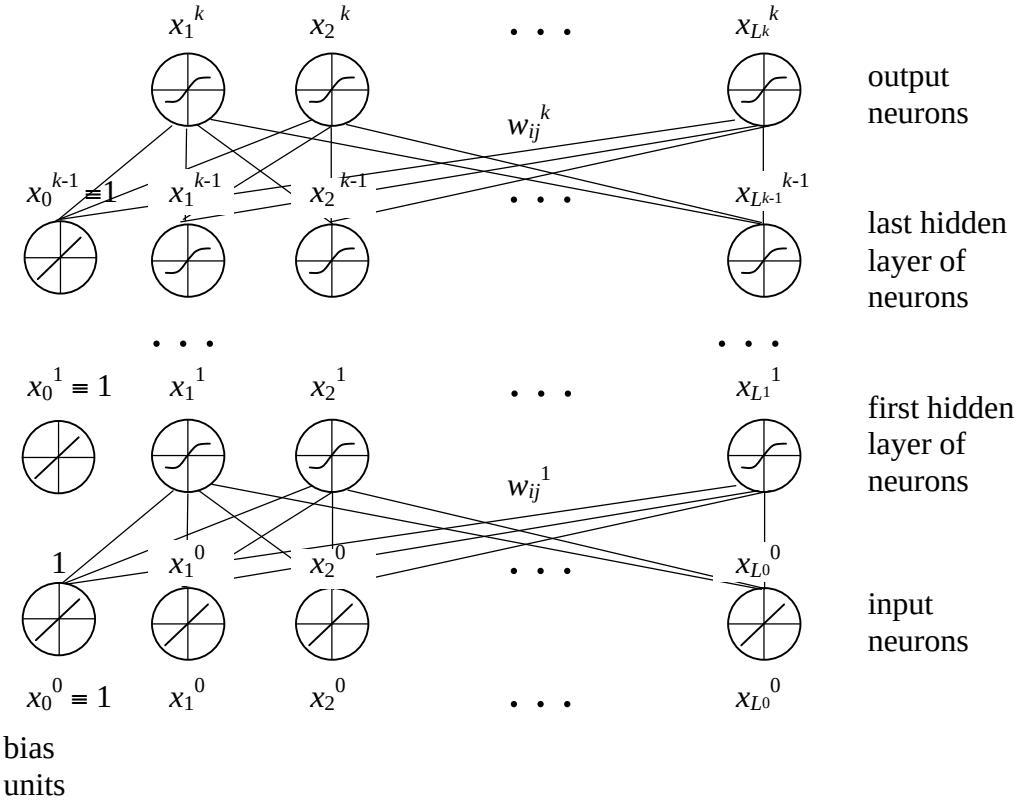


Figure 8.1: Schema of a multi-layer network with k–1 hidden layers of neurons. Layers of neurons are numbered 0, ..., k+1, where layer 0 contains the input units and layer k+1 the output units. The number of input units is L^0, of output units L^k, and of units in hidden layer m is L^m. The connection weight between the j-th unit in layer m and the i-th unit in layer m+1 is denoted by w_{ij}^m. The activation of the i-th unit in layer m is x_i^m (for m = 0 this is an input value, for $m = k$+1 an output value). In this figure, the units with activations x_0^0, ..., x_0^{k-1} are dummy inputs responsible for feeding a bias term into the next upper layer of units.

In "network talk", one speaks of a single-layer network when there is a single layer of *connection weights*, that is, when we have a linear network as we saw in Section 5. Correspondingly, a two-layer network has two layers of connections and one layer of "hidden" units between the input and output layer of units.

The network architecture shown in Fig. 8.1 has an orderly layered structure. It is also possible to extend the powers of single-layer networks by adding more units in a "disordered" way without a clear layer topology. However, this is rarely done. Layered networks of the kind shown in the Figure are also often called *multi-layer perceptrons* (MLPs).

From a mathematical perspective, a MLP implements a function $\mathbf{y} = f_{\mathrm{MLP}}(\mathbf{u})$, where $\mathbf{u} = (x_1^0, \ldots, x_{L^0}^0)^\mathsf{T}$ is the vector of inputs to the network and $\mathbf{y} = (x_1^k, \ldots, x_{L^k}^k)^\mathsf{T}$ is the output vector. The network computes **y** by passing the input **u** through its internal layers. To make this formal, first consider the circumstance that both in training and in exploitation we will use the network for many different inputs, which we denote by **u**(*n*), where *n* is an index marking different exemplars of input, not time. Then **y**(*n*) is the output obtained on that input, $x_i^m(n)$ are the internal activations, etc.

Formally, a *k*-layer MLP has the following components. The activation of input units is just the input, with the first input unit (index 0) set up as a dummy to contribute a constant bias input to the next higher layer:

$$\mathbf{x}^0(n) = (1, \mathbf{u}^\mathsf{T}(n))^\mathsf{T} = (x_0^0(n), x_1^0(n), \ldots, x_{L^0}^0(n))^\mathsf{T} = (1, x_1^0(n), \ldots, x_{L^0}^0(n))^\mathsf{T} \tag{8.1}$$

The activations of the dummy units $x_0^m(n)$ [where $1 \le m \le k-1$] is always fixed at 1. The activation $x_i^m(n)$ of the *i*-th non-dummy unit in a non-input unit layer *m* (where $i, m \ge 1$) is computed from the activations of the next lower layer by

$$x_i^m(n) = g^m\Big(\sum_{j=0,\ldots,L^{m-1}} w_{ij}^m x_j^{m-1}(n)\Big). \tag{8.2}$$

Thus, the activation $x_i^m(n)$ is computed by first taking a linear combination of the activations of the units from one layer below, and passing this through the unit's *activation function* g^m. (also called *output function*). The activation function may change across layers. We will consider here cases where g^m is a sigmoid function for hidden layers of units. The activation function of the output layer may be a sigmoid too; but sometimes it is more convenient to use linear activation functions on the output units. If one uses sigmoids on the output layer, one can only implement functions whose value range is within [0 1] (for the logistic sigmoid) or within [–1 1] (for the tanh sigmoid). If one uses linear output units, the implementable function range is unbounded.

Typical choices for sigmoid *g* are

$$g_1(a) = \frac{1}{1+e^{-1}}, \text{ the "logistic" sigmoid, and } g_2(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}. \tag{8.3}$$

Figure 8.2 shows these two. Note that g_2 differs from g_1 only through linear pre- and postprocessing transformations. Specifically, it holds that $g_2(a) = 2\, g_1(2a) - 1$. Thus, any network that uses g_2 as an activation function for hidden units can be replaced by an

equivalent network using g_1 but having different weights. Empirically it is often found that networks set up with the tanh sigmoid exhibit faster convergence in training algorithms than when the logistic sigmoid is used.

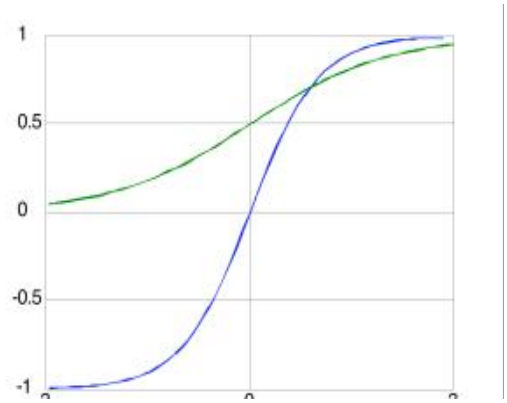


Figure 8.2: The two most commonly used sigmoids. Blue: tanh, green: the logistic $1/(1+e^{-a})$.

The weights $w_{ij}^{\,m}$ are trainable in an MLP. The objective is to find weights such that for a given set of training input-output data,

(8.4) $\mathbf{u}(n) = (x_1^0(n), \ldots, x_{L^0}^0(n))^\mathsf{T}$, $\mathbf{d}(n) = (d_1^k(n), \ldots, d_{L^k}^k(n))^\mathsf{T}$ $\quad [n = 1,\ldots, N]$

the network outputs $\mathbf{y}(n) = (x_1^k(n), \ldots, x_{L^k}^k(n))^\mathsf{T}$ become a good approximation of the teacher output $\mathbf{d}(n)$ in the sense of a small training sum of squared errors:

(8.5) $$SE_{\text{train}} = \sum_{n=1,\ldots,N} \left\|\mathbf{d}(n) - \mathbf{y}(n)\right\|^2 = \sum_{n=1,\ldots,N} E(n)\,.$$

The following two facts explain why MLPs have become so popular in the last thirty years:

- Any smooth function $f\colon \mathcal{I}^{L^0} \to \mathbb{R}^{L^k}$ from the unit hypercube to $\mathbb{R}^{L^k}$ can be approximated arbitrarily well by two-layer MLPs with linear output units (*universal approximation* property).
- There exists an relatively efficient learning algorithm to find weights that represent a local minimum of SE_{train}.

Taken together, these facts recommend MLPs as powerful and computationally sufficiently efficient black-box modelling devices for nonlinear function approximation and classification tasks.

We will outline a simple proof of the universal approximation property. We consider the case of f being a mapping from the two-dimensional unit square to $\mathbb{R}$. We know that f can be Fourier-approximated with arbitrarily small sum-of-squares error by

$$
\begin{aligned}
(8.6)\quad & f(x_1, x_2) \approx \\
& \sum_s \alpha_s(x_1)\cos(sx_2) + \beta_s(x_1)\sin(sx_2) \\
&= \sum_{s,t} (\alpha_{st}\cos(tx_1) + \alpha'_{st}\sin(tx_1))\cos(sx_2) + (\beta_{st}\cos(tx_1) + \beta'_{st}\sin(tx_1))\sin(sx_2) \\
&= \sum_{s,t} \alpha_{st}\cos(tx_1)\cos(sx_2) + \alpha'_{st}\sin(tx_1)\cos(sx_2) + \beta_{st}\cos(tx_1)\sin(sx_2) + \beta'_{st}\sin(tx_1)\sin(sx_2) \\
&= \frac{1}{2}\sum_{s,t} \alpha_{st}(\cos(tx_1 + sx_2) + \cos(tx_1 - sx_2)) + \alpha'_{st}(\sin(tx_1 + sx_2) - \sin(tx_1 - sx_2)) + \\
&\qquad \beta_{st}(\sin(sx_2 + tx_1) - \sin(sx_2 - tx_1)) + \beta'_{st}(\cos(tx_1 - sx_2) - \cos(tx_1 + sx_2)) \\
&= \frac{1}{2}\sum_{s,t} \alpha_{st}(\cos(z_{st}) + \cos(z'_{st})) + \alpha'_{st}(\sin(z_{st}) - \sin(z'_{st})) + \\
&\qquad \beta_{st}(\sin(z_{st}) + \sin(z'_{st})) + \beta'_{st}(\cos(z'_{st}) - \cos(z_{st})) \\
&= \frac{1}{2}\sum_{s,t} \gamma_{st}\cos(z_{st}) + \gamma'_{st}\cos(z'_{st}) + \delta_{st}\sin(z_{st}) + \delta'_{st}\sin(z'_{st}),
\end{aligned}
$$

where $\alpha_s(x_1)$ and $\beta_s(x_1)$ are functions of x_1, and we used elementary trigonometric identies like $\cos(\alpha)\cos(\beta) = 1/2\cos(\alpha + \beta) + 1/2\cos(\alpha - \beta)$ and substitutions $z_{st} = tx_1 + sx_2$ and $z'_{st} = tx_1 - sx_2$ to show that $f(x_1, x_2)$ can be approximated by a linear combination of trignonometric functions sin or cos of linear combinations of the arguments x_1, x_2. Now consider any one of these trigonometric terms, for instance $\cos(z_{st})$. It can itself be approximated to arbitrary precision (in the mean squared distance sense) on $[z_{st_{min}}, z_{st_{max}}]$ by a superposition of unit step functions H (recall: this is the Heaviside step function $H(x) = 0$ if $x < 0$, else 1):

$$
(8.7)\quad \cos(z_{st}) \approx \cos(z_{st_{min}}) + \sum_{i=0}^{K} (\cos(z_{i+1}) - \cos(z_i))H(z - z_i),
$$

where $z_{st_{min}} = z_0 < z_1 < \ldots < z_K = z_{st_{max}}$ is some sequence of intermediate arguments.

Note finally that the Heaviside function can be approximated arbitrarily well by the logistic sigmoid, by linear scaling of the argument of this sigmoid.

Taking all this together, we can approximate $f(x_1, x_2)$ to arbitrary accuracy by a two-layer MLP, when we

5. use the first weight layer to transform the input x_1, x_2 into the linear transform variables substitutions $z_{st} = tx_1 + sx_2$ and $z'_{st} = tx_1 - sx_2$,
6. use (8.7) to approximate $\cos(z_{st})$ by the combined output of K+1 hidden units,
7. use (8.6) to approximate $f(x_1, x_2)$ as the output from the output layer of units.

The core of this proof idea is that we can transform a product of trigonometric functions, e.g. $\cos(\alpha)\cos(\beta)$, into a linear combination of single trigonometric functions. This is also possible for longer products of trigonometric functions, e.g. it holds that $\cos(\alpha)\cos(\beta)\cos(\gamma) = 1/4[\cos(\alpha + \beta - \gamma) + \cos(-\alpha + \beta + \gamma) + \cos(\alpha - \beta + \gamma) + \cos(\alpha + \beta + \gamma)]$, etc. Thus the idea of this proof carries over to higher-dimensional input.

Numerous mathematical articles on the representational power of MLPs have appeared in the early 90-ies, showing that MLPs can approximate smooth, differentiable functions arbitrarily well, including their derivatives, in various norms for measuring the distance between network output and the correct target functions. These results have been important to establish the reputation of MLPs as universal approximation machines. However, such results are of little practical importance, because the network solutions that these theorems find for a given approximation task are typically very generous w.r.t. the size of the networks. Practical learning algorithms typically find much smaller networks for the same required approximation accuracy.

Little is known about the minimal size of a network that is required for a given degree of approximation in a given task, or about the best structure of the network (e.g., number of hidden layers). Three-layer networks are often preferred over the two-layer networks that are theoretically sufficient. Experience, personal taste and patience are asked for from the MLP designer!

A much cited result by Barron[10] gives the following bound on the risk of a two-layer MLP f_{MLP} that is trained to approximate a function (with one-dimensional target domain) $y = f(\mathbf{u})$:

$$R(f_{\mathrm{MLP}}) \le O\left(\frac{C_f^2}{L^1}\right) + O\left(\frac{L^1 d}{N}\log(N)\right), \tag{8.8}$$

where L^1 is the number of hidden neurons, N is the size of training data, d is the dimension of the input space (that is, $d = L^0 - 1$), and C_f is a certain spectral measure of complexity of the function f which is to be approximated. The first term in (8.8) is a bound on the (squared) bias (cf. Eq. (7.12)) while the second term is a bound on the variance. Note that the number of adjustable weights is $O(L^1 d)$. A consequence of (8.8), also obtained by Barron in the same paper, gives a bound on the risk that is obtained when the network size L^1 is optimized for minimal risk depending on the training sample size N. The risk is then

$$R(f_{\mathrm{MLP}}) \le O(C_f ((d/N) \log N)^{1/2}) \tag{8.9}$$

The bound (8.9) states that the rate of risk convergence with optimally selected network sizes, as a function of sample size is of order $(d/N)^{1/2}$ (times a logarithmic factor), where the exponent 1/2 is independent on the input dimension d.

The traditional way to construct approximations to nonlinear functions is by a linear combination of a fixed set of n basis functions (e.g. polynomials in Taylor expansions, multinomials in Volterra expansions, or sines in Fourier expansions). Putting learning aside, and considering the ideal case where the linear combination is set to yield the minimal risk, Barron[11] showed that there are functions in class C_f where the risk is at least

[10] Andrew R. Barron, *Approximation and Estimation Bounds for Artificial Neural Networks*, Machine Learning 14 (1994), 115-133

[11] Barron, A. R., Universal Approximation Bounds for Superpositions of a Sigmoidal Function, IEEE Trans. Inf. Theory 39(3), 1993, 930-945

(8.10) $$R(\textit{best linear model from n components}) \geq \kappa \frac{C_f}{d}\left(\frac{1}{n}\right)^{1/d},$$

where κ is a universal constant. This indicates the presence of a learning-independent version of the curse of dimensionality, because the risk scales exponentially with *d*. In contrast, for 2-layer MLPs the corresponding ideal risk for a model with *n* hidden units is (Barron 1994)

(8.11) $$R(\textit{best n-hidden-unit MLP}) \leq \frac{(2C_f)^2}{n}.$$

If one looks at the ratio of the convergence rates w.r.t. *n* of (8.10) and (8.11) for fixed *d*, i.e. at

(8.12) $$\kappa \frac{C_f}{d}\left(\frac{1}{n}\right)^{1/d} / \frac{(2C_f)^2}{n} = K\frac{n^{1-1/d}}{d},$$

one sees that this is (up to a constant *K*) approximately equal to *n*/*d* for not too small *d*. That is, if one wishes to approximate *f* better and better by increasing the number of basis functions (in (8.10)) or the number of hidden units (in (8.11)), this race for better risk is won by the neural network approach – the risk ratio between the two approaches grows roughly linearly with *n*. The underlying reason for this superiority of MLPs over fixed basis functions combinations is that the MLP can *shape* the functions represented by the hidden units, which then are linearly combined into the output layer. With linear combinations of *fixed* basis functions, one cannot adapt to the particulars of the given target function *f*.

Another result by Koiran and Sontag[12] that also adresses the risk convergence of MLPs, but from the perspective of statistical learning theory, states that for MLPs with at least one hidden layer of units and logistic sigmoid activation functions, the VC dimension is $h = O(|W|^2)$, where |W| is the total number of adjustable weights. If we insert this into the expression of the confidence term (cf. Eq. (7.7)) in the SLT risk bound, we get

(8.13) $$\phi\left(\frac{h}{N}, \frac{\log(\eta)}{N}\right) = \sqrt{\frac{|W|^2\left(\log\frac{2N}{|W|^2}+1\right) - \log(\eta/4)}{N}} \leq \frac{1}{\sqrt{N}}\left(|W| + |W|\log(\frac{\sqrt{2N}}{|W|}) + \sqrt{-\log(\eta/4)}\right).$$

Both (8.8) and (8.13) reveal a linearly bounded relationship between the number of adjustable weights and the generalization error of MLPs, which is described through the variance term in (8.8) and through the confidence term in (8.13). Thus, for MLPs, our rather vague intuition that "the danger of overfitting grows with the number of adjustable parameters" is here justified.

8.2 Training MLPs with the backpropagation algorithm

[12] Koiran, P. and E.D. Sontag, *Neural networks with quadratic VC dimension,* in: Advances in Neural Information Processing Systems (NIPS), vol. 8, MIT Press (1996). Cited after: S. Haykin, Neural Networks: A Comprehensive Foundation, Second Edition. Prentice-Hall 1999 (page 97).

Because MLPs are nonlinear functions with a large number of adjustable variables (the weights), a closed-form solution (like the Wiener-Hopf equation for linear systems) for the weights that give the smallest training error is not available. Instead, one uses iterative gradient-descent methods like we know them from adaptive linear combiners. However, now the performance surface has no longer the simple shape of a bowl with a unique minimum. Rather you should think of it as a rugged landscape with many local minima. If one starts the gradient descent from a particular set of starting weights, one ends up in the nearest local minimum – which may or may not give a training error close to the possible minimal error. We will discuss later how to deal with this situation.

The gradient of the training error w.r.t. the weights is a vector made from the partial derivatives of the training error w.r.t. the weights. These are (compare (8.5)):

$$\frac{\partial SE_{train}}{\partial w_{ij}^m} = \frac{\partial \sum_{n=1,...N} E(n)}{\partial w_{ij}^m} = \frac{\partial \sum_{n=1,...N} \left\| \mathbf{d}(n) - \mathbf{y}(n) \right\|^2}{\partial w_{ij}^m}, \tag{8.14}$$

where $m = 1, ..., k$ and $j = 0, ..., L^{m-1}$ and $i = 1, ..., L^m$. Note that for computing this gradient for a single iteration step "downhill", we have to use the entire training data set! However, because

$$\frac{\partial \sum_{n=1,...N} E(n)}{\partial w_{ij}^m} = \sum_{n=1,...N} \frac{\partial E(n)}{\partial w_{ij}^m}, \tag{8.15}$$

we only need to compute the gradient on single instances of the training data and may then sum up these values. Once we have solved the problem of computing (8.15), we obtain a weight update algorithm through

$$new\ w_{ij}^m = w_{ij}^m - \gamma \frac{\partial E}{\partial w_{ij}^m}, \tag{8.16}$$

where γ is a small learning rate. Naive implementations for computing the derivatives $\partial E(n) / \partial w_{ij}^m$ for all weights need $O(W^2)$ operations, where W is the number of weights. Considering that for a single gradient-downhill-step, we need N $O(W^2)$ operations, and that both N and W can easily be of the order of 1000, naive implementations quickly run out of steam. The *backpropagation* algorithm is a computationally efficient method to evaluate $\partial E(n) / \partial w_{ij}^m$ at cost $O(W)$. Only with this algorithm MLPs became widely useful. The backpropagation algorithm was made popular in a paper by Rumelhart, Hinton and Williams[13]. It appeard in 1986 in a famous two-volume collection of articles that laid the foundations of what today can be considered mainstream neural network techniques. The algorithm had precursors in work by Werbos[14] and Parker[15]. We now derive this algorithm.

[13] Rumelhart, D.E., G.E. Hinton, R.J. Williams (1986): Learning internal representations by error propagation. In D.E. Rumelhart, J.L. McClelland, and the PDP Research Group (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, pp. 318-362. Cambridge, MA: MIT Press

[14] Werbos, P.J. (1974): Beyond regression: new tools for prediction and analysis in the behavioural sciences. PhD thesis, Harvard Univ., Boston, MA.

[15] Parker, D.B. (1985): Learning Logic. Technical Report TR-47, MIT, Cambridge, MA.

The overall structure of the algorithm is as follows. The network is first initialized with randomly chosen (typically small) weights. Then the backprop algorithm is used iteratively. At each iteration, all weights are updated, going "downhill" the performance surface a small step, using the information from all training samples. Each such iteration is called an *epoch.* The procedure terminates if the output error falls under a pre-set threshold, or if the error improvement per step falls under another pre-set threshold, or if the number of epochs reaches a pre-set maximum.

Now we describe what happens in one epoch. At the beginning, weights $w^m{}_{ij}$ are given from the previous epoch. From (8.15) we see that we need to consider only a single training sample, *n*. In order to compute all $\partial E(n)/\partial w_{ij}^m$, the first step is to present the input pattern **u**(*n*) to the network and compute its output **y**(*n*). This is called the *forward pass*. In the forward pass, for each unit we compute the activation $x^m{}_i$; and in order to obtain this activation, we also compute the following quantity:

(8.17) $$a_i^m(n) = \sum_{j=0,\ldots,L^{m-1}} w_{ij}^m x_j^{m-1}(n),$$

which is sometimes referred to as the *potential* or the *internal state* of the unit. Applying the chain rule we have

(8.18) $$\frac{\partial E(n)}{\partial w_{ij}^m} = \frac{\partial E(n)}{\partial a_i^m}\frac{\partial a_i^m}{\partial w_{ij}^m}.$$

Define

(8.19) $$\delta_i^m = \frac{\partial E(n)}{\partial a_i^m}.$$

Using (8.17), we can write

(8.20) $$\frac{\partial a_i^m}{\partial w_{ij}^m} = x_j^{m-1}.$$

Inserting (8.19) and (8.20) into (8.18), we get

(8.21) $$\frac{\partial E(n)}{\partial w_{ij}^m} = \delta_i^m x_j^{m-1}.$$

Thus, in order to calculate the derivatives, we only need to compute the values of δ_i^m for each hidden and output unit. For output units, this is straightforward. From the definition (8.19), we have

(8.22) $$\delta_i^k = \frac{\partial E(n)}{\partial a_i^k} = g'(a_i^k)\frac{\partial E(n)}{\partial y_i} = g'(a_i^k)\frac{\partial (d_i(n)-y_i(n))^2}{\partial y_i} = 2(d_i(n)-y_i(n))g'(a_i^k).$$

To evaluate the δ_i^m for the hidden units, we again make use of the chain rule,

(8.23) $$\delta_i^m = \frac{\partial E(n)}{\partial a_i^m} = \sum_{l=1,\ldots,L^{m+1}} \frac{\partial E(n)}{\partial a_l^{m+1}} \frac{\partial a_l^{m+1}}{\partial a_i^m},$$

which is justified by the fact that the only path by which a_i^m can affect $E(n)$ is through the potentials a_l^{m+1} of the next higher layer, that is, $E(n)$ is a function of the a_l^{m+1}. If we now substitute (8.19) into (8.23) and observe (8.17), we get

(8.24)
$$\begin{aligned}
\delta_i^m = \frac{\partial E(n)}{\partial a_i^m} &= \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} \frac{\partial a_l^{m+1}}{\partial a_i^m} \\
&= \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} \frac{\partial \sum_{j=0,\ldots,L^m} w_{lj}^{m+1} g^m(a_j^m(n))}{\partial a_i^m} \\
&= \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} \frac{\partial w_{li}^{m+1} g^m(a_i^m(n))}{\partial a_i^m} \\
&= g^m{}'(a_i^m(n)) \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} w_{li}^{m+1}
\end{aligned}$$

This formula describes how the δ_i^m in a hidden layer can be computed by "back-propagating" the δ_l^{m+1} from the next higher layer. The formula can be used to compute all δ_i^m, starting from the output layer (where (8.22) is used) as a basis, and then working backwards through the network in the *backward pass* of the algorithm to compute the δ_i^m using the values already found in the next higher layer.

If the logistic sigmoid g_1 is used for the g^m, the computation of $g_1'(a_i^m(n))$ takes a particularly simple form, observing that for this sigmoid

(8.25) $$g_1'(a) = g_1(a)(1 - g_1(a)),$$

which leads to $g^m{}'(a_i^m(n)) = x_i^m(1 - x_i^m)$.

Although simple in principle, and readily implemented, using the backprop algorithm appropriately is something of an art. Here is only place to point out some difficulties:

- The stepsize γ in (8.16) 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. It is however not possible to provide an analytical treatment of how to set the stepsize optimally. Generally, one uses larger stepsizes in early epochs.
- Gradient descent on nonlinear surfaces may sometimes be very slow in areas where the gradient is small in some directions. By consequence, the backpropagation algorithm may sometimes need in the order of thousand(s) iterations to settle near a local minimum.
- Like all gradient-descent techniques on error surfaces, backpropagation finds only a local error minimum. This problem can be addressed by various measures, e.g. adding noise during training (simulated annealing approaches) to avoid getting stuck in poor minima, or by repeating the entire learning from different initial weight settings, or by using task-

specific prior information to start from an already plausible set of weights. All of these counter-measures (except the last one) are computationally expensive. Some authors claim that the local minimum problem is overrated.

- Finally, finding a network structure (number of units, number of layers) that is appropriate for a given task is not trivial. A decent amount of experimentation and cross-validation exploration may be needed.

These difficulties are not unique to MLPs trained by the backpropagation algorithm. The same problems surface with all methods for learning nonlinear regression models; they are a consequence of nonlinearity. The field of estimating nonlinear systems is difficult and rich in problems and techniques, it requires a lot of experience, and it has great importance for practical applications. By the way, our own brain is in many ways an exceedingly good learning apparatus for nonlinear (dynamical) systems, but nobody comes anywhere close to understanding how it functions.

9. Recurrent neural networks

This section is a slightly revised version of: H. Jaeger, *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network approach*, GMD Report 159, GMD – German National Research Institute for Information Technology, Sankt Augustin, 2002.

9.1 First impressions

9.1.1 Recurrent vs. Feedforward networks

There are two major types of neural networks, feedforward and recurrent. As we have seen in the previous section, in feedforward networks, activation is "piped" through the network from input units to output units (from left to right in left drawing in Fig. 9.1):

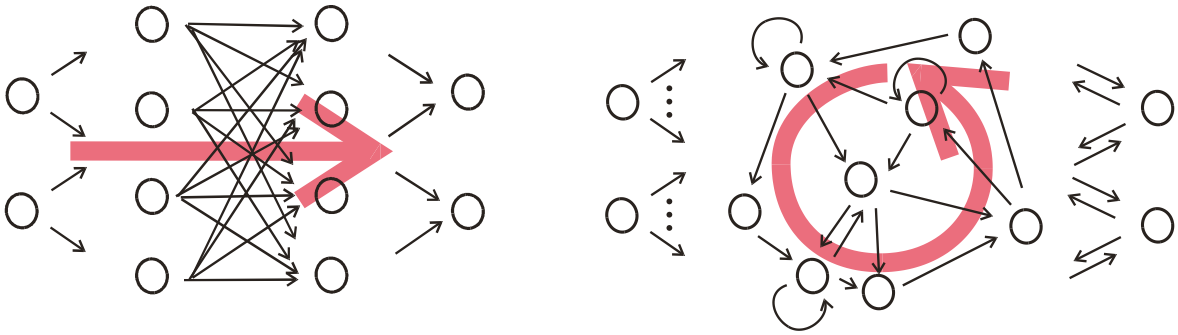


Figure 9.1: *Typical structure of a feedforward network (left) and a recurrent network (right).*

For the sake of contrast, here are some characteristic properties of feedforward networks:

> Typically, activation is fed forward from input to output through "hidden layers" (as in MLPs), though many other architectures exist.
> Mathematically, they implement static input-output mappings (functions).
> Basic theoretical result: MLPs can approximate arbitrary (term needs some qualification) nonlinear maps with arbitrary precision ("universal approximation property").
> Most popular supervised training algorithm: backpropagation algorithm.
> Huge literature, 95 % of neural network publications concern feedforward nets (my estimate).
> Have proven useful in many practical applications as approximators of nonlinear functions and as pattern classificators.

By contrast, a recurrent neural network (RNN) has (at least one) cyclic path of synaptic connections. Basic characteristics:

> Virtually all biological neural networks are recurrent.
> Mathematically, RNNs implement dynamical systems. While feedforward networks are used with "static" data (input-output pairs), RNNs are always used with time series data (signals in, signals out).

Basic theoretical result: RNNs can approximate arbitrary (term needs some qualification) dynamical systems with arbitrary precision ("universal approximation property").
Several types of training algorithms are known, no clear winner.
Theoretical and practical difficulties by and large have prevented practical applications so far.
Not covered in most neuroinformatics textbooks, absent from engineering textbooks.

Types of tasks for which RNNs can, in principle, be used:

system identification and inverse system identification,
filtering and prediction ,
dynamic pattern classification,
stochastic sequence modelling,
associative memory,
data compression.

Some relevant application areas:

telecommunication,
control of chemical plants,
control of engines and generators,
fault monitoring, biomedical diagnostics and monitoring,
speech recognition,
robotics, toys and edutainment,
video data analysis,
man-machine interfaces.

9.1.2 Time series data

At first glance, there is no big difference in the kind of training data that we use with RNNs vs. with feedforward networks. The training samples still essentially look like $(\mathbf{x}_i, \mathbf{d}_i)_{i = 1,...,N}$, only in the context of RNNs we will use a slightly other convention, namely use **u** to denote inputs, *n* to denote the sample index, and **x**(*n*), **d**(*n*) to denote the *n*-th sample item:

(9.1) *Training data:* $(\mathbf{u}(n), \mathbf{d}(n))_{n = 1,...,N}$

The reason for using this notation is to agree with the conventions in signal processing and control, where the symbol **u** is typically reserved for inputs, and – because one always deals with time series data – one writes **u**(*n*), **d**(*n*) to indicate that the input and teacher values are functions of time *n*.

9.1.3 Formal description of RNNs

Exactly like in feedforward networks, the elementary building blocks of a RNN are neurons (we will often use the term *units*) connected by synaptic links (*connections*) whose synaptic strength is coded by a *weight*. One typically distinguishes *input units*, *internal* (or *hidden*) *units*, and *output units*. At a given time, a unit has an *activation*.We denote the activations of input units by *u*(*n*), of internal units by *x*(*n*), of output units by *y*(*n*). Sometimes we ignore the input/internal/output distinction and then use *x*(*n*) in a metonymical fashion.

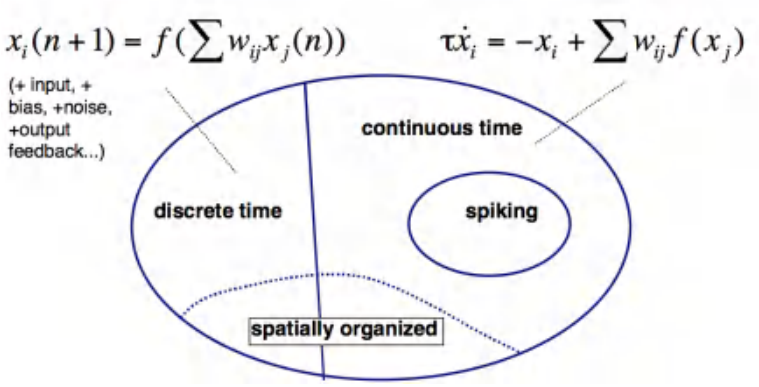


Figure 9.2: *A typology of RNN models (incomplete).*

There are many types of formal RNN models (see Fig. 9.2). Discrete-time models are mathematically cast as maps iterated over discrete time steps n = 1, 2, 3, ... Continuous-time models are defined through differential equations whose solutions are defined over a continous time t. Especially for purposes of biological modeling, continuous dynamical models can be quite involved and describe activation signals on the level of individual action potentials (*spikes*). Often the model incorporates a specification of a spatial topology, most often of a 2D surface where units are locally connected in retina-like structures.

In this tutorial we will only consider a particular kind of discrete-time models without spatial organization. Our model consists of K input units with an activation (column) vector

(9.2) $$\mathbf{u}(n) = (u_1(n), \ldots, u_K(n))^T,$$

of N internal units with an activation vector

(9.3) $$\mathbf{x}(n) = (x_1(n), \ldots, x_N(n))^T,$$

and of L output units with an activation vector

(9.4) $$\mathbf{y}(n) = (y_1(n), \ldots, y_L(n))^T,$$

The input / internal / output connection weights are collected in N x K / N x N / L x (K+N) weight matrices

(9.5) $$\mathbf{W}^{in} = (w_{ij}^{in}), \quad \mathbf{W} = (w_{ij}), \quad \mathbf{W}^{out} = (w_{ij}^{out}).$$

The output units may optionally project back to internal units with connections whose weights are collected in a N x L backprojection weight matrix

(9.6) $$\mathbf{W}^{back} = (w_{ij}^{back}).$$

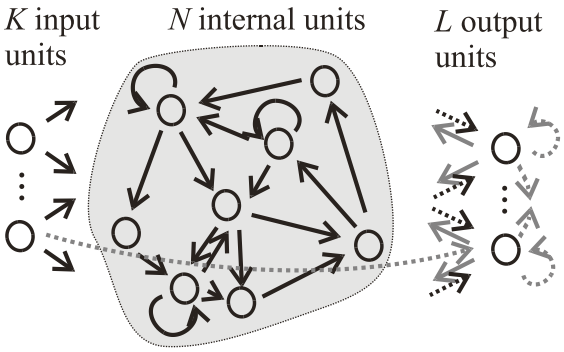


Figure 9.3: Our basic RNN architecture. Shaded arrows indicate optional connections. Dotted arrows mark connections which are trained in the "echo state network" approach (in other approaches, all connections can be trained).

A zero weight value can be interpreted as "no connection". Note that output units may have connections not only from internal units but also (often) from input units and (rarely) from output units.

The activation of internal units is updated according to

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)), \quad (9.7)$$

where $\mathbf{u}(n+1)$ is the externally given input, and $\mathbf{f}$ denotes the component-wise application of the individual unit's *transfer function, f* (also known as activation function, unit output function, or squashing function). We will mostly use the sigmoid function $f = \tanh$ but sometimes also consider linear networks with $f = 1$. The output is computed according to

$$\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1)), \quad (9.8)$$

where $(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n))$ denotes the concatenated vector made from input, internal, and output activation vectors. We will use output transfer functions $f^{out} = \tanh$ or $f^{out} = 1$; in the latter case we have linear output units.

9.1.4 Example: a little timer network

Consider the input-output task of *timing*. The input signal has two components. The first component $u_1(n)$ is 0 most of the time, but sometimes jumps to 1. The second input $u_2(n)$ can take values between 0.1 and 1.0 in increments of 0.1, and assumes a new (random) of these values each time $u_1(n)$ jumps to 1. The desired output is 0.5 for $10 \cdot u_2(n)$ time steps after $u_1(n)$ was 1, else is 0. This amounts to implementing a timer: $u_1(n)$ gives the "go" signal for the timer, $u_2(n)$ gives the desired duration.

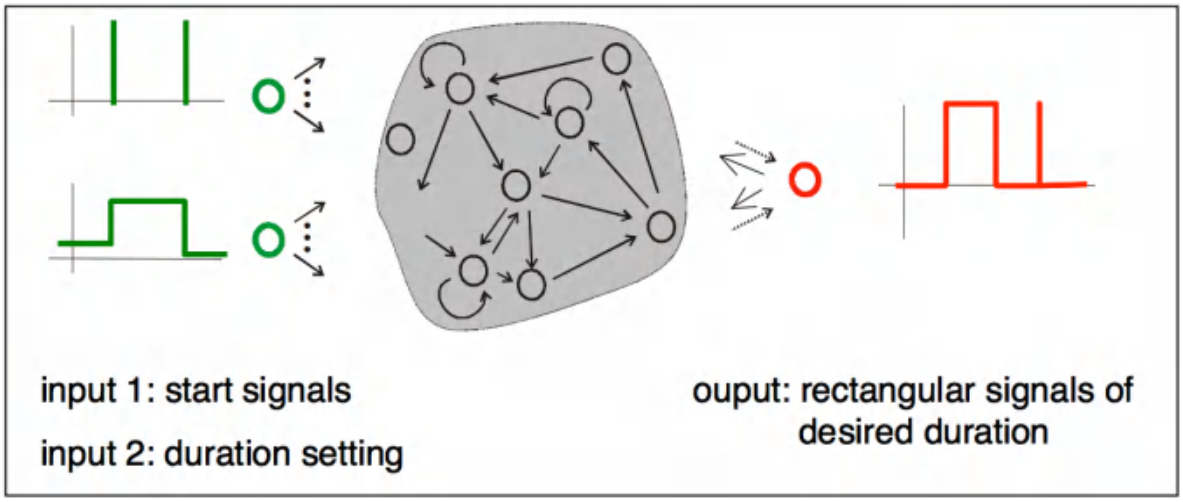


Figure 9.4: Schema of the timer network.

The following figure shows traces of input and output generated by a RNN trained on this task according to the ESN approach:

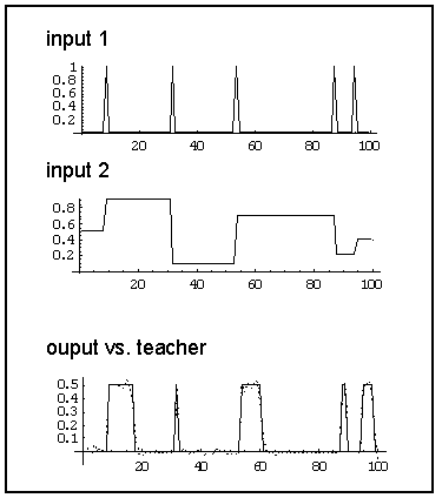


Figure 1.7: Performance of a RNN trained on the timer task. Solid line in last graph: desired (teacher) output. Dotted line: network output.

Clearly this task requires that the RNN must act as a memory: it has to retain information about the "go" signal for several time steps. This is possible because the internal recurrent connections let the "go" signal "reverberate" in the internal units' activations. Generally, tasks requiring some form of memory are candidates for RNN modeling.

9.2 Standard training techniques for RNNs

During the 1990's, several methods for supervised training of RNNs have been explored, which today already might be considered "classics". All of these rely on gradient descent methods for training error minimization. Since 2000, an altogether different approach to RNN training was found, which will be presented later. In this subsection we review the most

important "classical" trianing methods: *backpropagation through time* (BPTT), *real-time recurrent learning* (RTRL), and *extended Kalman filtering based techniques* (EKF). BPTT is probably the most widely used, RTRL is the mathematically most straightforward, and EKF is (arguably) the technique among the classics that gives best results – if used by experts.

9.2.1 Backpropagation revisited

BPTT is an adaptation of the well-known backpropagation training method known from feedforward networks.

We start with a recap of this notation, as introduced in section 8.1. We consider a multi-layer perceptron (MLP) with k hidden layers of neurons.Together with the layer of input units and the layer of output units this gives k+2 layers of units altogether, which we number by 0, ..., k+1 (as in figure 8.1). The number of input units is L^0, of output units L^{k+1}, and of units in hidden layer m is L^m. The weight of the j-th unit in layer m and the i-th unit in layer m+1 is denoted by w_{ij}^m. The activation of the i-th unit in layer m is x_i^m (for $m = 0$ this is an input value, for $m = k$+1 an output value).

The training data for a feedforward network training task consist of T input-output (vector-valued) data pairs

$$(9.9) \quad \mathbf{u}(n) = (x_1^0(n),...,x_{L^0}^0(n))^T, \ \mathbf{d}(n) = (d_1^{k+1}(n),...,d_{L^{k+1}}^{k+1}(n))^T,$$

where n denotes training instance, not time. The activation of non-input units is computed according to

$$(9.10) \quad x_i^{m+1}(n) = f\Big(\sum_{j=1,...,N^m} w_{ij}^m x_j(n)\Big).$$

(Standardly one also has bias terms, which we omit here). Presented with teacher input $\mathbf{u}(t)$, the previous update equation is used to compute activations of units in subsequent hidden layers, until a network response

$$(9.11) \quad \mathbf{y}(n) = (x_1^{k+1}(n),\ldots,x_L^{k+1}(n))'$$

is obtained in the output layer. The objective of training is to find a set of network weights such that the summed squared error

$$(9.12) \quad E = \sum_{n=1,...,T} \|\mathbf{d}(n) - \mathbf{y}(n)\|^2 = \sum_{n=1,...,T} E(n)$$

is minimized. This is done by incrementally changing the weights along the direction of the error gradient w.r.t. weights

$$(9.13) \quad \frac{\partial E}{\partial w_{ij}^m} = \sum_{t=1,...T} \frac{\partial E(n)}{\partial w_{ij}^m}$$

using a (small) learning rate $\boldsymbol{\gamma}$:

(9.14) $new\ w_{ij}^m = w_{ij}^m - \gamma \frac{\partial E}{\partial w_{ij}^m}$.

This is the formula used in *batch learning mode*, where new weights are computed after presenting all training samples. One such pass through all samples is called an epoch. Before the first epoch, weights are initialized, typically to small random numbers. A variant is *incremental learning*, where weights are changed after presentation of individual training samples:

(9.15) $new\ w_{ij}^m = w_{ij}^m - \gamma \frac{\partial E(n)}{\partial w_{ij}^m}$.

The central subtask in this method is the computation of the error gradients $\frac{\partial E(n)}{\partial w_{ij}^m}$, which is affected by the backpropagation algorithm which we described in Section 8.2.

9.2.2. Backpropagation through time

The feedforward backpropagation algorithm cannot be directly transferred to RNNs because the error backpropagation pass presupposes that the connections between units induce a cycle-free ordering. The solution of the BPTT approach is to "unfold" the recurrent network in time, by stacking identical copies of the RNN, and redirecting connections within the network to obtain connections between subsequent copies. This gives a feedforward network, which is amenable to the backpropagation algorithm.

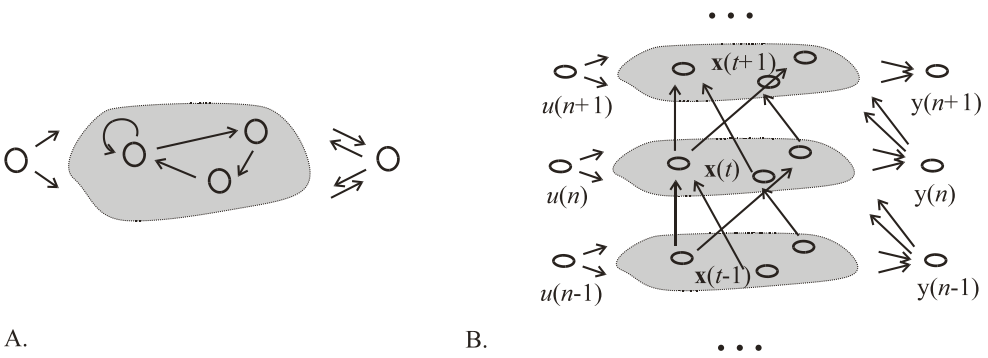


Figure 9.6: Schema of the basic idea of BPTT. A: the original RNN. B: The feedforward network obtained from it. The case of single-channel input and output is shown.

The weights w_{ij}^{in}, w_{ij}, w_{ij}^{out}, w_{ij}^{back} are identical in/between all copies. The teacher data consists now of a single input-output time series

(9.16) $\mathbf{u}(n) = (u_1(n),\ldots,u_K(n))'$, $\mathbf{d}(n) = (d_1(n),\ldots,d_L(n))'$ $\quad n = 1,\ldots T$.

The forward pass of one training epoch consists in updating the stacked network, starting from the first copy and working upwards through the stack. At each copy (= time) n input $\mathbf{u}(n)$ is read in, then the internal state $\mathbf{x}(n)$ is computed from $\mathbf{u}(n)$, $\mathbf{x}(n\text{-}1)$ [and from $\mathbf{y}(n\text{-}1)$ if nonzero w_{ij}^{back} exist], and finally the current copy's output $\mathbf{y}(n)$ is computed.

The error to be minimized is again (like in (9.12))

$$(9.17)\quad E = \sum_{n=1,\ldots,T} \left\|\mathbf{d}(n) - \mathbf{y}(n)\right\|^2 = \sum_{n=1,\ldots,T} E(n),$$

but the meaning of *n* has changed from "training instance" to "time". The algorithm is a straightforward, albeit notationally complicated, adaptation of the feedforward algorithm:

Input: current weights w_{ij}, training time series.

Output: new weights.

Computation steps:

1. Forward pass: as described above.
2. Compute, by proceeding backward through $n = T,\ldots,1$, for each time n and unit activation $x_i(n)$, $y_j(n)$ the error propagation term $d_i(n)$

$$(9.18)\quad \delta_j(T) = (d_j(T) - y_j(T)) \left.\frac{\partial f(u)}{\partial u}\right|_{u=z_j(T)}$$

for the output units of time layer T and

$$(9.19)\quad \delta_i(T) = \left[\sum_{j=1}^{L} \delta_j(T) w_{ji}^{out}\right] \left.\frac{\partial f(u)}{\partial u}\right|_{u=z_i(n)}$$

for internal units $x_i(T)$ at time layer T and

$$(9.20\quad \delta_j(n) = \left[(d_j(n) - y_j(n)) + \sum_{i=1}^{N} \delta_i(n+1) w_{ij}^{back}\right] \left.\frac{\partial f(u)}{\partial u}\right|_{u=z_j(n)}$$

for the output units of earlier layers, and

$$(9.21)\quad \delta_i(n) = \left[\sum_{j=1}^{N} \delta_j(n+1) w_{ji} + \sum_{j=1}^{L} \delta_j(n) w_{ji}^{out}\right] \left.\frac{\partial f(u)}{\partial u}\right|_{u=z_i(n)}$$

for internal units $x_i(n)$ at earlier times, where $z_i(n)$ again is the potential of the corresponding unit.

3. Adjust the connection weights according to

$$
\begin{aligned}
&new\ w_{ij} = w_{ij} + \gamma \sum_{n=1}^{T} \delta_i(n)\, x_j(n-1) \quad [use\ x_j(n-1) = 0\ for\ n = 1] \\
&new\ w_{ij}^{in} = w_{ij}^{in} + \gamma \sum_{n=1}^{T} \delta_i(n)\, u_j(n) \\
&new\ w_{ij}^{out} = w_{ij}^{out} + \gamma \times \begin{cases} \sum_{n=1}^{T} \delta_i(n)\, u_j(n), & if\ \ j\ refers\ to\ input\ unit \\ \sum_{n=1}^{T} \delta_i(n)\, x_j(n-1), & if\ \ j\ refers\ to\ hidden\ unit \end{cases} \\
&new\ w_{ij}^{back} = w_{ij}^{back} + \gamma \sum_{n=1}^{T} \delta_i(n)\, y_j(n-1) \quad [use\ y_j(n-1) = 0\ for\ n = 1]
\end{aligned}
\tag{2.22}
$$

Warning: programming errors are easily made but not easily perceived when they degrade performance only slightly.

The remarks concerning slow convergence made for standard backpropagation carry over to BPTT. The computational complexity of one epoch is $O(T N^2)$, where N is number of internal units. Several thousands of epochs are typically required.

A variant of this algorithm is to use the teacher output $\mathbf{d}(n)$ in the computation of activations in layer n+1 in the forward pass. This is known as *teacher forcing*. Teacher forcing typically speeds up convergence, or even be necessary to achieve convergence at all, but when the trained network is exploited, it may exhibit instability. A general rule when to use teacher forcing cannot be given.

A drawback of this "batch" BPTT is that the entire teacher time series must be used. This precludes applications where online adaptation is required. The solution is to truncate the past history and use, at time n, only a finite history

$$\mathbf{u}(n-p), \mathbf{u}(n-p+1), \ldots, \mathbf{u}(n), \mathbf{d}(n-p), \mathbf{d}(n-p+1), \ldots, \mathbf{d}(n) \tag{9.23}$$

as training data. Since the error backpropagation terms d need to be computed only once for each new time slice, the complexity is $O(N^2)$ per time step. A potential drawback of such truncated BPPT (or p-BPTT) is that memory effects exceeding a duration p cannot be captured by the model. Anyway, BPTT generally has difficulties capturing long-lifed memory effects, because backpropagated error gradient information tends to "dilute" exponentially over time. A frequently stated opinion is that memory spans longer than 10 to 20 time steps are hard to achieve.

Repeated execution of training epochs shift a complex nonlinear dynamical system (the network) slowly through parameter (weight) space. Therefore, bifurcations are necessarily encountered when the starting weights induce a qualitatively different dynamical behavior than task requires. Near such bifurcations, the gradient information may become essentially useless, dramatically slowing down convergence. The error may even suddenly grow in the vicinity of such critical points, due to crossing bifurcation boundaries. Unlike feedforward backpropagation, BPTT is not guaranteed to converge to a local error minimum. This

difficulty cannot arise with feedforward networks, because they realize functions, not dynamical systems.

All in all, it is far from trivial to achieve good results with BPTT, and much experimentation (and processor time) may be required before a satisfactory result is achieved.

Because of limited processing time, BPTT is typically used with small networks sizes in the order of 3 to 20 units. Larger networks may require many hours of computation on current hardware.

9.2.3. Real-time recurrent learning

Real-time recurrent learning (RTRL) is a gradient-descent method which computes the exact error gradient at every time step. It is therefore suitable for online learning tasks. I basically quote the description of RTRL given in Doya (1995[16]). The most cited early description of RTRL is Williams & Zipser (1989[17]).

The effect of weight change on the network dynamics can be seen by simply differentiating the network dynamics equations (9.7) and (9.8) by its weights. For convenience, activations of all units (whether input, internal, or output) are enumerated and denoted by v_i and all weights are denoted by w_{kl}, with $i = 1,...,N$ denoting internal units, $i = N+1,...,N+L$ denoting output units, and $i = N+L+1,...,N+L+K$ denoting input units. The derivative of an internal or output unit w.r.t. a weight w_{kl} is given by

$$\frac{\partial v_i(n+1)}{\partial w_{kl}} = f'(a_i(n))\left[\left(\sum_{j=1}^{n} w_{ij}\frac{\partial v_j(n)}{\partial w_{kl}}\right) + \delta_{ik} v_l(n)\right] \quad i = 1,\ldots,N+L, \tag{9.24}$$

where k, $l \leq N + L + K$, $a_{ii}(n)$ is again the unit's potential (as in (8.14)), but δ_{ik} here denotes Kronecker's delta (δ_{ik} =1 if $i = k$ and 0 otherwise). The term $\delta_{ik}\, v_l(n)$ represents an explicit effect of the weight w_{kl} onto the unit k, and the sum term represents an implicit effect onto all the units due to network dynamics.

Equation (9.24) for each internal or output unit constitutes an $N+L$-dimensional discrete-time linear dynamical system with time-varying coefficients, where

$$\left(\frac{\partial v_1}{\partial w_{kl}},\ldots,\frac{\partial v_{N+L}}{\partial w_{kl}}\right) \tag{9.25}$$

is taken as a dynamical variable. Since the initial state of the network is independent of the connection weights, we can initialize (9.24) by

$$\frac{\partial v_i(0)}{\partial w_{kl}} = 0. \tag{9.26}$$

[16] Doya, K. (1992), *Bifurcations in the learning of recurrent neural networks.* Proceedings of 1992 IEEE Int. Symp. On Circuits and Systems vol. 6, 1992, 2777-2780

[17] Williams, R.J. and D. Zipser (1989) A learning algorithm for continually running fully recurrent neural networks. Neural Computation 1, 1989, 270-280

Thus we can compute (9.25) forward in time by iterating equation (9.24) simultaneously with the network dynamics (9.7) and (9.8). From this solution, we can calculate the error gradient (for the error given in (9.12)) by the chain rule as follows:

$$(9.27)\quad \frac{\partial E}{\partial w_{kl}} = 2\sum_{n=1}^{T}\sum_{i=N}^{N+L}(v_i(n) - d_i(n))\frac{\partial v_i(n)}{\partial w_{kl}}.$$

A standard batch gradient descent algorithm is to accumulate the error gradient by equation (9.27) and update each weight after a complete epoch of presenting all training data by

$$(9.28)\quad new\ \ w_{kl} = w_{kl} - \gamma\frac{\partial E}{\partial w_{kl}},$$

where γ is is a learning rate. An alternative update scheme is the gradient descent of current output error at each time step,

$$(9.29)\quad w_{kl}(n+1) = w_{kl}(n) - \gamma\sum_{i=1}^{L}(v_i(n) - d_i(n))\frac{\partial v_i(n)}{\partial w_{kl}}.$$

Note that we assumed w_{kl} is a constant, not a dynamical variable, in deriving (9.27), so we have to keep the learning rate small enough. (9.29) is referred to as *real-time recurrent learning.*

RTRL is mathematically transparent and in principle suitable for online training. However, the computational cost is $O((N+L)^4)$ for each update step, because we have to solve the $(N+L)$-dimensional system (9.24) for each of the weights. This high computational cost makes RTRL useful for online adaptation only when very small networks suffice.

9.2.4. Higher-order gradient descent techniques

Just a little note: Pure gradient-descent techniques for optimization generally suffer from slow convergence when the curvature of the error surface is different in different directions. In that situation, on the one hand the learning rate must be chosen small to avoid instability in the directions of high curvature, but on the other hand, this small learning rate might lead to unacceptably slow convergence in the directions of low curvature. A general remedy is to incorporate curvature information into the gradient descent process. This requires the calculation of the second-order derivatives, for which several approximative techniques have been proposed in the context of recurrent neural networks. These calculations are expensive, but can accelerate convergence especially near an optimum where the error surface can be reasonably approximated by a quadratic function. Dos Santos & von Zuben (2000[18]) and Schraudolph (2002[19]) provide references, discussion, and propose approximation techniques which are faster than naive calculations.

[18] Dos Santos, E.P. and von Zuben, F.J. (2000) *Efficient second-order learning algorithms for discrete-time recurrent neural networks*. In: Medsker, L.R. and Jain, L.C. (eds), Recurrent Neural Networsk: Design and Applications, 2000, 47-75. CRC Press: Boca Raton, Florida

[19] Schraudolph, N. (2002). *Fast curvature matrix-vector products for second-order gradient descent.* To appear in Neural Computation. Manuscript online at http://www.icos.ethz.ch/~schraudo/pubs/#mvp.

9.2.5 Extended Kalman-filtering approaches

9.2.5.1 The extended Kalman filter

The extended Kalman filter (EKF) is a state estimation technique for nonlinear systems derived by linearizing the well-known linear-systems Kalman filter around the current state estimate. We consider a simple special case, a time-discrete system with additive input and no observation noise:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{f}(\mathbf{x}(n)) + \mathbf{q}(n) \\ \mathbf{d}(n) &= \mathbf{h}_n(\mathbf{x}(n)) \end{aligned} \quad , \tag{9.30}$$

where $\mathbf{x}(n)$ is the system's internal state vector, $\mathbf{f}$ is the system's state update function (linear in original Kalman filters), $\mathbf{q}(n)$ is external input to the system (an uncorrelated Gaussian white noise process, can also be considered as process noise), $\mathbf{d}(n)$ is the system's output, and $\mathbf{h}_n$ is a time-dependent observation function (also linear in the original Kalman filter). At time $n = 0$, the system state $\mathbf{x}(0)$ is guessed by a multidimensional normal distribution with mean $\hat{\mathbf{x}}(0)$ and covariance matrix $\mathbf{P}(0)$. The system is observed until time n through $\mathbf{d}(0),\ldots, \mathbf{d}(n)$. The task addressed by the extended Kalman filter is to give an estimate $\hat{\mathbf{x}}(n+1)$ of the true state $\mathbf{x}(n+1)$, given the initial state guess and all previous output observations. This task is solved by the following two *time update* and three *measurement update* computations:

$$\begin{aligned} \hat{\mathbf{x}}^*(n) &= \mathbf{F}(\hat{\mathbf{x}}(n)) \\ \mathbf{P}^*(n) &= \mathbf{F}(n)\mathbf{P}(n-1)\mathbf{F}(n)^t + \mathbf{Q}(n) \end{aligned} \tag{9.31}$$

$$\begin{aligned} \mathbf{K}(n) &= \mathbf{P}^*(n)\mathbf{H}(n)[\mathbf{H}(n)^t\mathbf{P}^*(n)\mathbf{H}(n)]^{-1} \\ \hat{\mathbf{x}}(n+1) &= \hat{\mathbf{x}}^*(n) + \mathbf{K}(n)\xi(n) \\ \mathbf{P}(n+1) &= \mathbf{P}^*(n) - \mathbf{K}(n)\mathbf{H}(n)^t\mathbf{P}^*(n) \end{aligned} \tag{9.32}$$

where we roughly follow the notation in Singhal and Wu (1989)[20], who first applied extended Kalman filtering to (feedforward) network weight estimation. Here $\mathbf{F}(n)$ and $\mathbf{H}(n)$ are the Jacobians

$$\mathbf{F}(n) = \left.\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}(n)}, \quad \mathbf{H}(n) = \left.\frac{\partial \mathbf{h}_n(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}(n)} \tag{9.33}$$

of the components of $\mathbf{f}$, $\mathbf{h}_n$ with respect to the state variables, evaluated at the previous state estimate; $\xi(n) = \mathbf{d}(n) - \mathbf{h}_n(\hat{\mathbf{x}}(n))$ $\xi(n) = \mathbf{d}(n) - \mathbf{h}_n(\hat{\mathbf{x}}(n))$ $\xi(n)$ is the error (difference between observed output and output calculated from state estimate $\hat{\mathbf{x}}(n)$), $\mathbf{P}(n)$ is an estimate of the conditional error covariance matrix $E[\xi\xi \mid \mathbf{d}(0),\ldots, \mathbf{d}(n)]$; $\mathbf{Q}(n)$ is the (diagonal) covariance matrix of the process noise, and the time updates $\hat{\mathbf{x}}^*(n), \mathbf{P}^*(n)$ of state estimate and state error covariance estimate are obtained from extrapolating the previous estimates with the known dynamics $\mathbf{f}$.

[20] Singhal, S. and L. Wu (1989), *Training multilayer perceptrons with the extended Kalman algorithm*. In D.S. Touretzky (ed.), Advances in Neural Information Processing Systems 1, 1989, 133-140. San Mateo, CA: Morgan Kaufmann

The basic idea of Kalman filtering is to first update $\hat{\mathbf{x}}(n), \mathbf{P}(n)$ to preliminary guesses $\hat{\mathbf{x}}^*(n), \mathbf{P}^*(n)$ by extrapolating from their previous values, applying the known dynamics in the time update steps (9.31), and then adjusting these preliminary guesses by incorporating the information contained in $\mathbf{d}(n)$ – this information enters the measurement update in the form of $\xi(n)$, and is accumulated in the *Kalman gain* $\mathbf{K}(n)$.

In the case of classical (linear, stationary) Kalman filtering, $\mathbf{F}(n)$ and $\mathbf{H}(n)$ constant, and the state estimates converge to the true conditional mean state $E[\mathbf{x}(n) \mid \mathbf{d}(0),..., \mathbf{d}(n)]$. For nonlinear $\mathbf{f}$, $\mathbf{h}_n$, this is not generally true, and the use of extended Kalman filters leads only to locally optimal state estimates.

9.2.5.2 Applying EKF to RNN weight estimation

Assume that there exists a RNN which perfectly reproduces the input-output time series of the training data

(9.34) $\mathbf{u}(n) = (u_1(n),\ldots,u_K(n))^t,\ \mathbf{d}(n) = (d_1(n),\ldots,d_L(n))^t \qquad n = 1,\ldots T$

where the input / internal / output / backprojection connection weights are as usual collected in N x K / N x N / L x (K+N+L) / N x L weight matrices

(9.35) $\mathbf{W}^{in} = (w_{ij}^{in}),\quad \mathbf{W} = (w_{ij}),\quad \mathbf{W}^{out} = (w_{ij}^{out}),\ \mathbf{W}^{back} = (w_{ij}^{back})$.

In this subsection, we will not distinguish between all these different types of weights and refer to all of them by a weight vector $\mathbf{w}$.

In order to apply EKF to the task of estimating optimal weights of a RNN, we interpret the weights $\mathbf{w}$ of the perfect RNN as the state of a dynamical system. From a bird's eye perspective, the output $\mathbf{d}(n)$ of the RNN is a function $\mathbf{h}$ of the weights and input up to n:

(9.36) $\mathbf{d}(n) = \mathbf{h}(\mathbf{w}, \mathbf{u}(0),..., \mathbf{u}(n))$

where we assume that the transient effects of the initial network state have died out. The inputs can be integrated into the output function $\mathbf{h}$, rendering it a time-dependent function $\mathbf{h}_n$. We further assume that the network update contains some process noise, which we add to the weights (!) in the form of a Gaussian uncorrelated noise $\mathbf{q}(n)$. This gives the following version of (9.30) for the dynamics of the perfect RNN:

(9.37)
$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) + \mathbf{q}(n) \\ \mathbf{d}(n) &= \mathbf{h}_n(\mathbf{x}(n)) \end{aligned}$$

Except for noisy shifts induced by process noise, the state "dynamics" of this system is static, and the input $\mathbf{u}(n)$ to the network is not entered in the state update equation, but is hidden in the time-dependence of the observation function. This takes some mental effort to swallow!

The network training task now takes the form of estimating the (static, perfect) state $\mathbf{w}(n)$ from an initial guess $\hat{\mathbf{w}}(0)$ and the sequence of outputs $\mathbf{d}(0),..., \mathbf{d}(n)$. The error covariance matrix $\mathbf{P}(0)$ is initialized as a diagonal matrix with large diagonal components, e.g. 100.

The simpler form of (9.37) over (9.30) leads to some simplifications of the EKF recursions (9.31) and (9.32): because the system state (= weight!) dynamics is now trivial, the time update steps become unnecessary. The measurement updates become

$$\begin{aligned} &\mathbf{K}(n) = \mathbf{P}(n)\mathbf{H}(n)[\mathbf{H}(n)^t\mathbf{P}(n)\mathbf{H}(n)]^{-1} \\ (9.38)\quad &\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \mathbf{K}(n)\xi(n) \\ &\mathbf{P}(n+1) = \mathbf{P}(n) - \mathbf{K}(n)\mathbf{H}(n)^t\mathbf{P}(n) + \mathbf{Q}(n) \end{aligned}$$

A learning rate η (small at the beginning [!] of learning to compensate for initially bad estimates of $\mathbf{P}(n)$) can be introduced into the Kalman gain update equation:

$$(9.39)\quad \mathbf{K}(n) = \mathbf{P}(n)\mathbf{H}(n)[(1/\eta)\mathbf{I} + \mathbf{H}(n)^t\mathbf{P}(n)\mathbf{H}(n)]^{-1},$$

which is essentially the formulation given in Puskorius and Feldkamp (1994)[21].

Inserting process noise $\mathbf{q}(n)$ into EKF has been claimed to improve the algorithm's numerical stability, and to avoid getting stuck in poor local minima (Puskorius and Feldkamp 1994).

EKF is a second-order gradient descent algorithm, in that it uses curvature information of the (squared) error surface. As a consequence of exploiting curvature, for linear noise-free systems the Kalman filter can converge in a single step. We demonstrate this by a super-simple feedforward network example. Consider the single-input, single-output network which connects the input unit with the output unit by a connection with weight w, without any internal unit

$$\begin{aligned} &w(n+1) = w(n) \\ (9.40)\quad &d(n) = w\,u(n) \end{aligned}$$

We inspect the running EKF (in the version of (9.38)) at some time n, where it has reached an estimate $\hat{w}(n)$. Observing that the Jacobian $\mathbf{H}$(n) is simply d$wu(n)$/d$w = u(n)$, the next estimated state is

$$(9.41)\quad \hat{w}(n+1) = \hat{w}(n) + \mathbf{K}(n)\xi(n) = \hat{w}(n) + \frac{1}{u(n)}(wu(n) - \hat{w}u(n)) = w\,.$$

The EKF is claimed in the literature to exhibit fast convergence, which should have become plausible from this example at least for cases where the current estimate $\hat{\mathbf{w}}(n)$ is already close to the correct value, such that the linearisation yields a good approximation to the true system.

EKF requires the derivatives $\mathbf{H}(n)$ of the network outputs w.r.t. the weights evaluated at the current weight estimate. These derivatives can be exactly computed as in the RTRL algorithm, at cost $O(N^4)$. This is too expensive but for small networks. Alternatively, one can resort to truncated BPTT, use a "stacked" version of (9.37) which describes a finite sequence of outputs instead of a single output, and obtain approximations to $\mathbf{H}(n)$ by a procedure analogous to (9.18) – (9.21). Two variants of this approach are detailed out in Feldkamp et al. (1998)[22]. The cost here is $O(pN^2)$, where p is the truncation depth.

---

[21] Puskorius, G.V. and L. A. Feldkamp (1994) *Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks*. IEEE Transactions on Neural Networks 5(2), 1994, 279-297

[22] Feldkamp, L. A., Prokhorov, D., Eagen, C.F., and F. Yuan (1998) *Enhanced multistream Kalman filter*

Apart from the calculation of $\mathbf{H}(n)$, the most expensive operation in EKF is the update of $\mathbf{P}(n)$, which requires $O(LN^2)$ computations. By setting up the network architecture with suitably decoupled subnetworks, one can achieve a block-diagonal $\mathbf{P}(n)$, with considerable reduction in computations (Feldkamp et al. 1998).

As far as I have an overview, it seems to me that currently the best results in RNN training are achieved with EKF, using truncated BPTT for estimating $\mathbf{H}(n)$, demonstrated especially in many remarkable achievements from Lee Feldkamp's research group (see references). As with BPTT and RTRL, the eventual success and quality of EKF training depends very much on professional experience, which guides the appropriate selection of network architecture, learning rates, the subtelties of gradient calculations, presentation of input (e.g., windowing techniques), etc.

The methods presented so far – BPTT, RTRL, and EKF training – are all gradient-descent-based. A unifying framework for these techniques (and some others) has been given in Atiya and Parlos(2000[23]). They also introduced another learning rule, which combines and gerneralizes insights from BPTT, RTRL, and EKF, now frequently referred to as the *Atiya-Parlos learning rule*.

9.3 Echo state networks

9.3.1 First example: a sinewave generator

In this subsection I informally demonstrate the principles of echo state networks (ESN) by showing how to train a RNN to generate a sinewave.

The desired sinewave is given by $d(n) = 1/2 \sin(n/4)$. The task of generating such a signal involves no input, so we want a RNN without any input units and a single output unit which after training produces $d(n)$. The teacher signal is a 300-step sequence of $d(n)$.

We start by constructing a recurrent network with 20 units, whose internal connection weights $\mathbf{W}$ are set to random values. We will refer to this network as the "dynamical reservoir" (DR). The internal weights $\mathbf{W}$ will not be changed in the training described later in this subsection. The network's units are standard sigmoid units, as in Eq. (1.6), with a transfer function $f = \tanh$.

A randomly constructed RNN, such as our DR, might develop oscillatory or even chaotic acitivity even in the absence of external excitation. We do not want this to occur: The ESN approach needs a DR which is *damped*, in the sense that if the network is started from an arbitrary state $\mathbf{x}(0)$, the subsequent network states converge to the zero state. This can be achieved by a proper global scaling of $\mathbf{W}$: the smaller the weights of $\mathbf{W}$, the stronger the damping. We assume that we have scaled $\mathbf{W}$ such that we have a DR with modest damping. Fig. 9.7 shows traces of the 20 units of our DR when it was started from a random initial state $\mathbf{x}(0)$. The desired damping is clearly visible.

---

training for recurrent networks. In: J.A.K. Suykens and J. Vandewalle (ed.), Nonlinear modeling: advanced black-box techniques, 1998, 29-53. Boston: Kluwer

[23] Atiya, A.F. and Parlos, A.G. (2000), New Results on Recurrent Network Training: Unifying the Algorithms and Accelerating Convergence, IEEE Trans. Neural Networks 11(3), 697-709

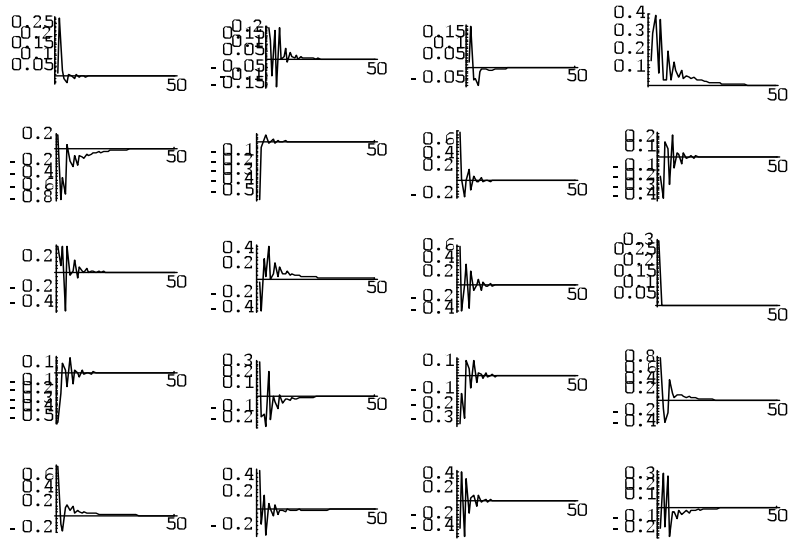


Figure 9.7. The damped dynamics of our dynamical reservoir.

We add a single output unit to this DR. This output unit features connections that project back into the DR. These backprojections are given random weights $\mathbf{W}^{back}$, which are also fixed do not change during subsequent training. We use a linear output unit in this example, i.e. f^{out} = id.

The only connections which *are* changed during learning are the weights $\mathbf{W}^{out}$ from the DR to the output unit. These weights are not defined (nor are they used) during training. Figure 6.2 shows the network prepared for training.

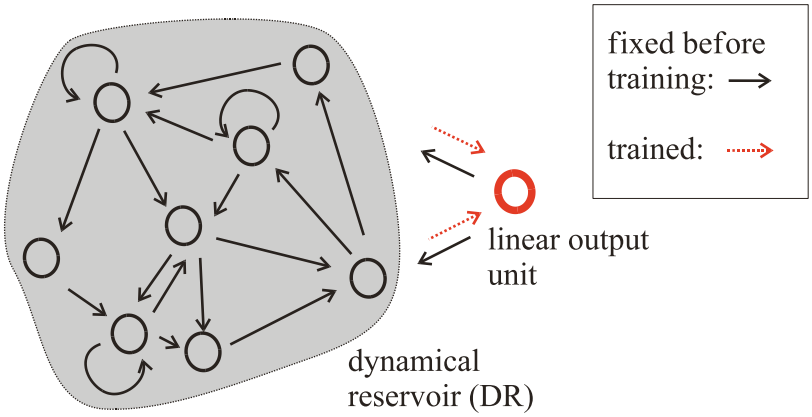


Figure 9.8: Schematic setup of ESN for training a sinewave generator.

The training is done in two stages, *sampling* and *weight computation*.

Sampling. During the sampling stage, the teacher signal is written into the output unit for times n = 1,....,300. (Writing the desired output into the output units during training is often called *teacher forcing*). The network is started at time n = 1 with an arbitrary starting state; we use the zero state for starting but that is just an arbitrary decision. The teacher signal $d(n)$ is pumped into the DR through the backprojection connections $\mathbf{W}^{back}$ and thereby excites an

activation dynamics within the DR. Figure 9.9 shows what happens inside the DR for sampling time steps n = 101,...,150.

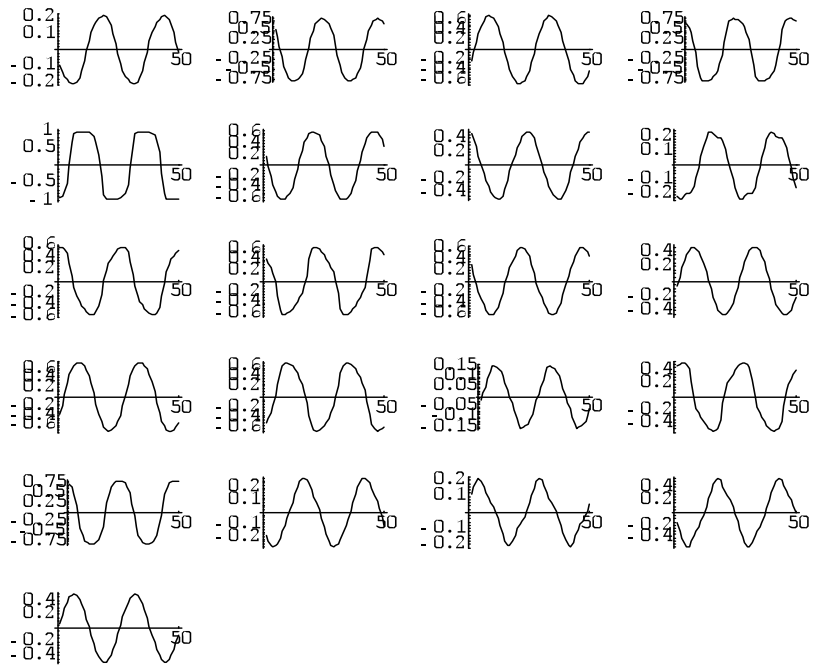


Figure 9.9. The dynamics within the DR induced by teacher-forcing the sinewave $d(n)$ in the output unit. 50-step traces of the 20 internal DR units and of the teacher signal (last plot) are shown.

We can make two important observations:

> The activation patterns within the DR are all periodic signals of the same period length as the driving teacher $d(n)$.
> The activation patterns within the DR are different from each other.

During the sampling period, the internal states $\mathbf{x}(n) = (x_1(n),...,x_{20}(n))$ for n = 101, ..., 300 are collected into the rows of a state-collecting matrix **M** of size 200x20. At the same time, the teacher outputs $d(n)$ are collected into the rows of a matrix **T** of size 200x1.

We do not collect information from times n = 1, ..., 100, because the network's dynamics is initially partly determined by the network's arbitrary starting state. By time n = 100, we can safely assume that the effects of the arbitrary starting state have died out and that the network states are a pure reflection of the teacher-forced $d(n)$, as is manifest in Fig. 9.9.

Weight computation. We now compute 20 output weights w_i^{out} for our linear output unit $y(n)$ such that the teacher time series $d(n)$ is approximated as a linear combination of the internal activation time series $x_i(n)$ by

$$(9.42) \quad d(n) \approx y(n) = \sum_{i=1}^{20} w_i^{out} \, x_i(n) \, .$$

More specifically, we compute the weights w_i^{out} such that the mean squared training error

$$(9.43)\quad \mathrm{MSE}_{train} = 1/200 \sum_{n=101}^{300} (d(n) - y(n))^2 = 1/200 \sum_{n=101}^{300} (d(n) - \sum_{i=1}^{20} w_i^{out} x_i(n))^2$$

is minimized.

From a mathematical point of view, this is a linear regression task: compute *regression weights* w_i^{out} for a regression of $d(n)$ on the network states $x_i(n)$. [n = 101, ..., 300].

From an intuitive-geometrical point of view, this means combining the 20 internal signals seen in Fig. 9.9 such that the resulting combination best approximates the last (teacher) signal seen in the same figure.

From an algorithmical point of view, this *offline* computation of regression weights boils down to the computation of a pseudoinverse: The desired weights which minimize MSE_{train} are obtained by multiplying the pseudoinverse of **M** with **T** (we have derived this method for computing linear regression weights already in section 4.5, equation (4.51)!):

$$(9.44)\quad \mathbf{W}^{\mathrm{out}} = \mathbf{M}^{+}\mathbf{T}$$

In our example, the training error computed by (9.43) with optimal output weights obtained by (9.44) was found to be MSE_{train}.= 1.2e–13.

The computed output weights are implemented in the network, which is then ready for use.

Exploitation. After the learnt output weights were written into the output connections, the network was run for another 50 steps, continuing from the last training network state **x**(300), but now with teacher forcing switched off. The output $y(n)$ was now generated by the trained network all on its own [n = 301, ..., 350]. The *test error*

$$(9.45)\quad \mathrm{MSE}_{test} = 1/50 \sum_{n=301}^{350} (d(n) - y(n))^2$$

was found to be MSE_{test}.= 5.6e–12. This is greater than the training error, but still very small. The network has learnt to generate the desired sinewave very precisely. An intuitive explanation of this precision would go as follows:

> The sinewave $y(n)$ at the output unit evokes periodic signals $x_i(n)$ inside the DR whose period length is *identical* to that of the output sine.
> These periodic signals make a kind of "basis" of signals from which the target $y(n)$ is combined. This "basis" is optimally "pre-adapted" to the target in the sense of identical period length. This pre-adaptation is a natural consequence of the fact that the "basis" signals $x_i(n)$ have been induced *by the target* itself, via the feedback projections.

So, in a sense, the task [to combine $y(n)$ from $x_i(n)$] is solved by means [the $x_i(n)$] which have been formed by the very task [by the backprojection of $y(n)$ into the DR]. Or said in intuitive terms, the target signal $y(n)$ is re-constituted from its own echos $x_i(n)$!

An immediate question concerns the stability of the solution. One may rightfully wonder whether the error in testing phase, small as it was in the first 50 steps, will not grow over time

and finally render the network's global oscillation unstable. That is, we might suspect that the precise continuation of the sine output after the training is due to the fact that we start testing from state **x**(300), which was produced by teacher forcing. However, this is not usually the case. Most networks trained according to the prescription given here can be started from almost any arbitrary nonzero starting state and will lock into the desired sinewave. Figure 9.10 shows an example. In mathematical terms, the trained network is a dynamical system with a single attractor, and this attractor is the desired oscillation. However, the strong stability observed in this example is a pleasant side-effect ot the simplicity of the sinewave generating task. When the tasks become more difficult, the stability of the trained dynamics *is* indeed a critical issue for ESN training.

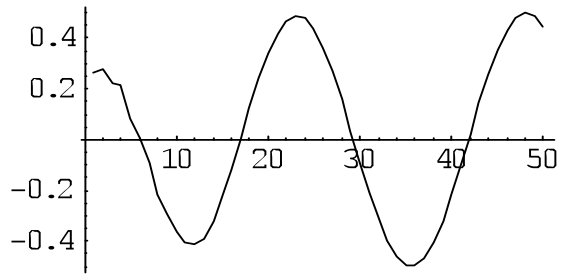


Figure 9.10: Starting the trained network from a random starting state. Plot shows first 50 outputs. The network quickly settles into the desired sinewave oscillation.

9.3.2 Second Example: a tuneable sinewave generator

We now make the sinewave generation task more difficult by demanding that the sinewave be adjustable in frequency. The training data now consists of an input signal $u(n)$, which sets the desired frequency, and an output $d(n)$, which is a sinewave whose frequency follows the input $u(n)$. Figure 9.11 shows the resulting network architecture and a short sequence of teacher input and output.

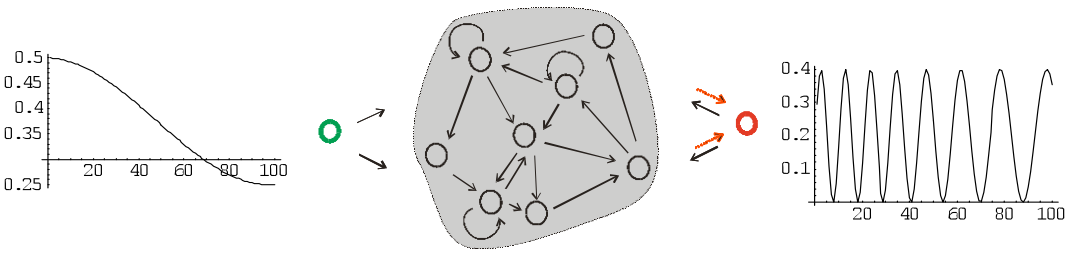


Figure 9.11: Setup of tuneable sinewave generator task. Trainable connections appear as dotted red arrows, fixed connections as solid black arrows.

Because the task is now more difficult, we use a larger DR with 100 units. In the sampling period, the network is driven by the teacher data. This time, this involves both inputting the slow signal $u(n)$, and teacher-forcing the desired output $d(n)$. We inspect the resulting activation patterns of internal units and find that they reflect, combine, and modify both $u(n)$ and $d(n)$ (Figure 9.12).

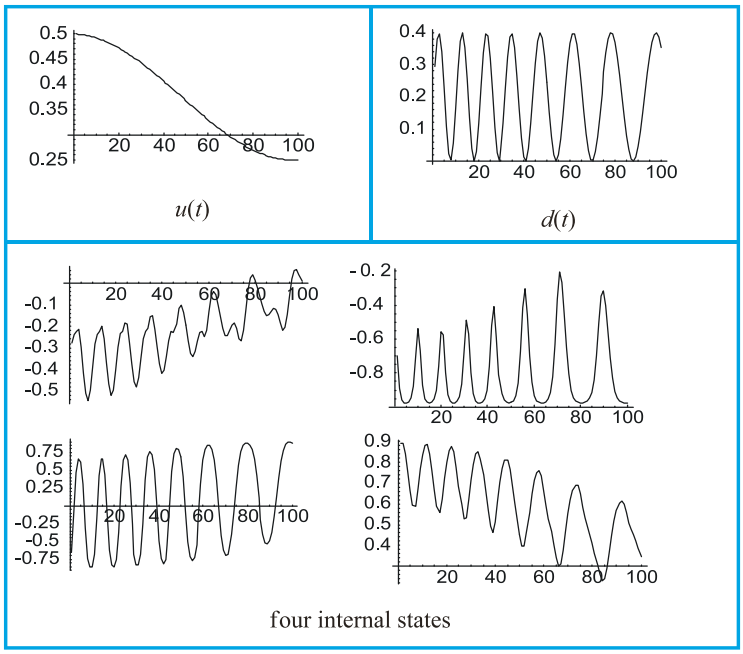


Figure 9.12: Traces of some internal DR units during the sampling period in the tuneable frequency generation task.

In this example we use a sigmoid output unit. In order to make that work, during sampling we collect into **T** not the raw desired output $d(n)$ but the transfer-inverted version $\tanh^{-1}(d(n))$. We also use a longer training sequence of 1200 steps of which we discard the first 200 steps as initial transient. The training error which we minimize concerns the tanh-inverted quantities:

(9.46)

$$\mathrm{MSE}_{train} = 1/1000 \sum_{n=201}^{1000} (\tanh^{-1} d(n) - \tanh^{-1} y(n))^2 = 1/1000 \sum_{n\,201}^{1000} (\tanh^{-1} d(n) - \sum_{i=1}^{100} w_i^{out} x_i(n))^2$$

This is achieved, as previously, by computing $\mathbf{W}^{\mathrm{out}} = \mathbf{M}^{+}\mathbf{T}$. The training error was found to be 8.1e-6, and the test error on the first 50 steps after inserting the computed output weights was 0.0006. Again, the trained network stably locks into the desired type of dynamics even from a random starting state, as displayed in Figure 9.13.

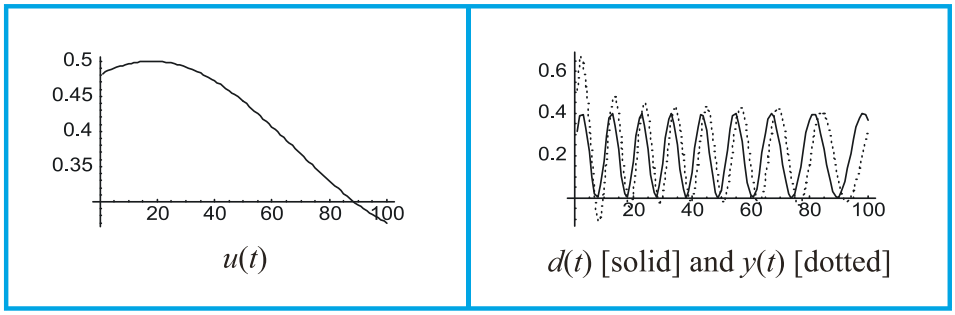


Figure 9.13 Starting the trained generator from a random starting state.

Stability of the trained network was not as easy to achieve here as in the previous example. In fact, a trick was used which was found empirically to foster stable solutions. The trick is to

insert some noise into the network during sampling. That is, during sampling, the network was updated according to the following variant of (9.7):

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n) + \nu(n)), \tag{9.47}$$

where $\nu(n)$ is a small white noise vector of size N.

9.3.3 Mathematics of echo states

In the two introductory examples, we rather vaguely said that the DR should exhibit a "damped" dynamics. We now describe in a rigorous way what kind of "damping" is required to make the ESN approach work, namely, that the DR must have *echo states*.

The key to understanding ESN training is the concept of *echo states*. Having echo states (or not having them) is a property of the network prior to training, that is, a property of the weight matrices $\mathbf{W}^{in}$, $\mathbf{W}$, and (optionally, if they exist) $\mathbf{W}^{back}$. The property is also relative to the type of training data: the same untrained network may have echo states for certain training data but not for others. We therefore require that the training input vectors $\mathbf{u}(n)$ come from a compact interval U and the training output vectors $\mathbf{d}(n)$ from a compact interval D. We first give the mathematical definition of echo states and then provide an intuitive interpretation.

Definition 9.1 (echo states). Assume an untrained network with weights $\mathbf{W}^{in}$, $\mathbf{W}$, and $\mathbf{W}^{back}$ is driven by teacher input $\mathbf{u}(n)$ and teacher-forced by teacher output $\mathbf{d}(n)$ from compact intervals U and D. The network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) has *echo states* w.r.t. U and D, if for every left-infinite input/output sequence ($\mathbf{u}(n)$, $\mathbf{d}(n)$), where n = ..., -2,-1,0, and for all state sequences $\mathbf{x}(n)$, $\mathbf{x}'(n)$ compatible with the teacher sequence, i.e. with

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{d}(n)) \\ \mathbf{x}'(n+1) &= \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}'(n) + \mathbf{W}^{back}\mathbf{d}(n)) \end{aligned} \tag{9.48}$$

it holds that $\mathbf{x}(n) = \mathbf{x}'(n)$ for all $n \leq 0$.

Intuitively, the echo state property says, "if the network has been run for a very long time [from minus infinity time in the definition], the current network state is uniquely determined by the history of the input and the (teacher-forced) output". An equivalent way of stating this is to say that for every internal signal $x_i(n)$ there exists an *echo function* e_i which maps input/output histories to the current state:

$$\begin{aligned} e_i : \quad & (U \times D)^{-\mathbb{N}} \to \mathbb{R} \\ & \left(\ldots,(\mathbf{u}(-1),\mathbf{d}(-2)),(\mathbf{u}(0),\mathbf{d}(-1))\right) \mapsto x_i(0) \end{aligned} \tag{9.49}$$

We often say, somewhat loosely, that a (trained) network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{out}$, $\mathbf{W}^{back}$) is an echo state network if its untrained "core" ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) has the echo state property w.r.t. input/output from *any* compact interval $U \times D$.

Several conditions have been shown to be equivalent to echo states. We provide one for illustration.

Definition 9.2 (uniformly state contracting). With the same assumptions as in Def. 9.1, the network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) is *uniformly state contracting* w.r.t. U and D, if there exists a null sequence $(\delta_n)_{n \geq 1}$, such that for all right-infinite input/output sequences ($\mathbf{u}(n)$, $\mathbf{d}(n\text{-}1)$) $\in U \times D$, where $n = 0,1,2,...$ and for all starting states $\mathbf{x}(0)$, $\mathbf{x}'(0)$ and for all $n > 0$ it holds that $\left| \mathbf{x}(n) - \mathbf{x}'(n) \right| < \delta_n$, where $\mathbf{x}(n)$ [resp. $\mathbf{x}'(n)$] is the network state at time n obtained when the network is driven by ($\mathbf{u}(n)$, $\mathbf{d}(n\text{-}1)$) up to time n after having been started in $\mathbf{x}(0)$, [resp. in $\mathbf{x}'(0)$].

Intuitively, the state forgetting property says that the effects on initial network state wash out over time. Note that there is some subtelty involved here in that the null sequence used in the definition depends on the the input/output sequence.

The echo state property is connected to algebraic properties of the weight matrix $\mathbf{W}$. Unfortunately, there is no known necessary and sufficient algebraic condition which allows one to decide, given ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$), whether the network has the echo state property. We quote here from Jaeger (2001[24]) a sufficient condition for the *non*-existence of echo states.

Proposition 9.1 Assume an untrained network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) with state update according to (9.7) and with transfer functions tanh. Let $\mathbf{W}$ have a spectral radius $|\lambda_{max}| > 1$, where $|\lambda_{max}|$ is the largest absolute value of an eigenvector of $\mathbf{W}$. Then the network has no echo states with respect to any input/output interval U x D containing the zero input/output ($\mathbf{0}$, $\mathbf{0}$).

At face value, this proposition is not helpful for finding echo state networks. However, in practice it was consistently found that when the condition noted in Proposition 9.1 is *not* satisfied, i.e. when the spectral radius of the weight matrix is smaller than unity, we *do* have an echo state network.

Note that in Proposition 9.1 the input and backprojection weights are not used for the claims. It seems that these weights are irrelevant for the echo state property. In practice, it is found that they can be freely chosen without affecting the echo state property. Again, a mathematical analysis of these observations remains to be done.

For practical purposes, the following procedure (also used in the conjecture) seems to guarantee echo state networks:

> Randomly generate an internal weight matrix $\mathbf{W}_0$.
> Normalize $\mathbf{W}_0$ to a matrix $\mathbf{W}_1$ with unit spectral radius by putting $\mathbf{W}_1 = 1/|\lambda_{max}| \, \mathbf{W}_0$, where $|\lambda_{max}|$ is the spectral radius of $\mathbf{W}_0$.
> Scale $\mathbf{W}_1$ to $\mathbf{W} = \alpha \, \mathbf{W}_1$, where $\alpha < 1$, whereby $\mathbf{W}$ has a spectral radius of α.
> Then, the untrained network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) is (or more precisely, has always been found to be) an echo state network, regardless of how $\mathbf{W}^{in}$, $\mathbf{W}^{back}$ are chosen.

The diligent choice of the spectral radius α of the DR weight matrix is of crucial importance for the eventual success of ESN training. This is because α is intimately connected to the intrinsic timescale of the dynamics of the DR state. Small α means that one has a fast DR, large α (i.e., close to unity) means that one has a slow DR. The intrinsic timescale of the task should match the DR timescale. For example, if one wishes to train a sine generator as in the

[24] H. Jaeger (2001), *The "echo state" approach to analysing and training recurrent neural networks*. GMD Report 148, GMD - German National Research Institute for Computer Science, 2001, http://www.faculty.iubremen.de/hjaeger/pubs/EchoStatesTechRep.pdf

example of Subsection 9.3.1, one should use a small α for fast sinewaves and a large α for slow sinewaves.

Note that the DR timescale seems to depend exponentially on 1 - α so e.g. settings of α = 0.99, 0.98, 0.97 will yield an exponential speedup of DR timescale, not a linear one. However, these remarks rest only on empirical observations; a rigorous mathematical investigation remains to be carried out. An illustrative example for a fast task is given in Jaeger (2001, Section 4.2), where very fast "switching"-type of dynamics was trained with a DR whose spectral radius was set to 0.44, which is quite small considering the exponential nature of time scale dependence on α. The sinewave generator and the tuneable sinewave generator from above both used a DR with α = 0.8.

Figure 9.14 gives a plot of the training log error $\log(\mathrm{MSE}_{train})$ of the sinewave generator training task considered in Section 9.3.1 obtained with different settings of α. It is evident that a proper setting of this parameter is crucial for the quality of the resulting generator network.

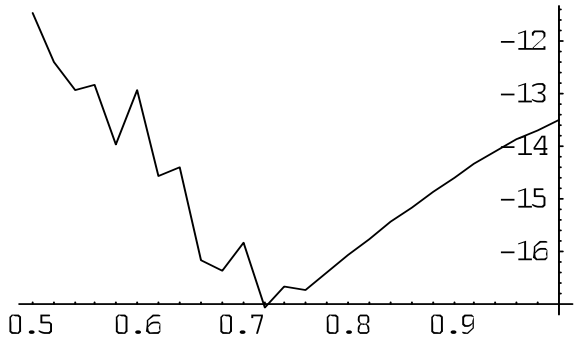


Figure 9.14: $\log_{10}(\mathrm{MSE}_{train})$ vs. spectral radius of the DR weight matrix for the sinewave generator experiment from Section 9.3.1.

9.3.4 Training echo state networks: algorithm

With the solid grasp on the echo state concept, we can now give a complete description of training ESNs for a given task. In this description we assume that the output unit(s) are sigmoid units; we further assume that there are output-to-DR feedback connections. This is the most comprehensive version of the algorithm. Often one will use simpler versions, e.g. linear output units; no output-to-DR feedback connections; or even systems without input (such as the pure sinewave generator). In such cases, the algorithm presented below has to be adapted in obvious ways.

Given: A training input/output sequence ($\mathbf{u}(1)$, $\mathbf{d}(1)$), ..., ($\mathbf{u}(T)$, $\mathbf{d}(T)$).

Wanted: A trained ESN ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$, $\mathbf{W}^{out}$) whose output $\mathbf{y}(n)$ approximates the teacher output $\mathbf{d}(n)$, when the ESN is driven by the training input $\mathbf{u}(n)$.

Notes:

> We can merely expect that the trained ESN approximates the teacher output well after initial transient dynamics have washed out, which are invoked by the (untrained, arbitrary) network starting state. Therefore, more precisely what we want is that the trained ESN approximates the teacher output for times $n = T_0, ..., T$, where $T_0 > 1$.

Depending on network size and intrinsic timescale, typical ranges for T_0 are 10 (for small, fast nets) to 500 (for large, slow nets).
What we *actually* want is not primarily a good approximation of the teacher output, but more importantly, a good approximation of *testing* output data from independent test data sets generated by the same (unknown) system which also generated the teacher data. In practice this means that we have to use cross-validation methods to optimize our model capacity.

Step 1. Procure an untrained DR network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) which has the echo state property, and whose internal units exhibit mutually interestingly different dynamics when excited.

This step involves many heuristics. The way I proceed most often involves the following substeps.

Randomly generate an internal weight matrix $\mathbf{W}_0$.
Normalize $\mathbf{W}_0$ to a matrix $\mathbf{W}_1$ with unit spectral radius by putting $\mathbf{W}_1 = 1/|\lambda_{max}|\ \mathbf{W}_0$, where $|\lambda_{max}|$ is the spectral radius of $\mathbf{W}_0$. Standard mathematical packages for matrix operations all include routines to determine the eigenvalues of a matrix, so this is a straightforward thing.
Scale $\mathbf{W}_1$ to $\mathbf{W} = \alpha\ \mathbf{W}_1$, where $\alpha < 1$, whereby $\mathbf{W}$ obtains a spectral radius of α.
Randomly generate input weights $\mathbf{W}^{in}$ and output backpropagation weights $\mathbf{W}^{back}$.
Then, the untrained network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$) is (or more honestly, has always been found to be) an echo state network, regardless of how $\mathbf{W}^{in}$, $\mathbf{W}^{back}$ are chosen.

Notes:

The matrix $\mathbf{W}_0$ should be sparse, a simple method to encourage a rich variety of dynamics of different internal units. Furthermore, the weights should be roughly equilibrated, i.e. the mean value of weights should be about zero. I usually draw nonzero weights from a uniform distribution over [– 1, 1], or I set nonzero weights randomly to –1 or 1.
The size N of $\mathbf{W}_0$ should reflect both the length T of training data, and the difficulty of the task. As a rule of thumb, N should not exceed an order of magnitude of $T/10$ to $T/2$ (the more deterministic/low-noise the training data, the closer to $T/2$ can N be chosen). This is a simple precaution against overfitting. Furthermore, more difficult tasks require larger N.
The setting of α is crucial for subsequent model performance. It should be small for fast teacher dynamics and large for slow teacher dynamics, according to the observations made above in Section 9.3.3. Typically, α needs to be hand-tuned by trying out several settings.
The absolute size of input weights $\mathbf{W}^{in}$ is also of some importance. Large absolute $\mathbf{W}^{in}$ imply that the network is strongly driven by input, small absolute values mean that the network state is only slightly excited around the DR's resting (zero) state. In the latter case, the network units operate around the linear central part of the sigmoid, i.e. one obtains a network with an almost linear dynamics. Larger $\mathbf{W}^{in}$ drive the internal units closer to the saturation of the sigmoid, which results in a more nonlinear behavior of the resulting model. In the extreme, when $\mathbf{W}^{in}$ becomes very large, the internal units will be driven into an almost pure – 1 / +1 valued, binary dynamics. If one has severyl inputs, one can at this point steer their relative impact on the reservoir dynamics by scaling the input weights individually for the different inputs. For instance, low-amplitude inputs can be emphasized by upscaling their input weights; noisy or not

very relevant inputs can be depreciated by downscaling their input weights. Again, manual adjustment and repeated learning trials will often be required to find the taskappropriate
Similar remarks hold for the absolute size of weights in $\mathbf{W}^{back}$.
More often than not, the training error can be improved by adding a bias input whose value is frozen to 1. Again, its impact on the reservoir dynamics needs to be adjusted by scaling the corresponding input weights.

Step 2. Sample network training dynamics.

This is a mechanical step, which involves no heuristics. It involves the following operations:

Initialize the network state arbitrarily, e.g. to zero state $\mathbf{x}(0) = \mathbf{0}$.
Drive the network by the training data, for times $n = 0, ..., T$, by presenting the teacher input $\mathbf{u}(n)$, and by teacher-forcing the teacher output $\mathbf{d}(n\text{-}1)$, by computing

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{d}(n)) \tag{9.50}$$

At time $n = 0$, where $\mathbf{d}(n)$ is not defined, use $\mathbf{d}(n) = \mathbf{0}$.
For each time larger or equal than an initial washout time T_0, collect the concatenated input and network state $(\mathbf{u}(n), \mathbf{x}(n))^T$ as a new row into a state collecting matrix $\mathbf{M}$. In the end, one has obtained a state collecting matrix of size $(T - T_0 + 1)$ x $(K + N)$.
Similarly, for each time larger or equal to T_0, collect the sigmoid-inverted teacher output $\tanh^{-1}\mathbf{d}(n)$ row-wise into a teacher collection matrix $\mathbf{T}$, to end up with a teacher collecting matrix $\mathbf{T}$ of size $(T - T_0 + 1)$ x L.

Note: Be careful to collect into $\mathbf{M}$ and $\mathbf{T}$ the vectors $\mathbf{u}(n)$, $\mathbf{x}(n)$ and $\tanh^{-1}\mathbf{d}(n)$, not $\mathbf{u}$(n), $\mathbf{x}(n)$ and $\tanh^{-1}\mathbf{d}(n\text{-}1)$!

Step 3: Compute output weights.

Concretely, multiply the pseudoinverse of $\mathbf{M}$ with $\mathbf{T}$, to obtain a $(K + N)$ x L sized matrix $(\mathbf{W}^{out})^T$ whose i-th column contains the output weights from all network units to the i -th output unit:

$$(\mathbf{W}^{out})^T = \mathbf{M}^{-1}\mathbf{T}. \tag{9.51}$$

Every programming package of numerical linear algebra has optimized procedures for computing pseudoinverses.
Transpose $(\mathbf{W}^{out})^T$ to $\mathbf{W}^{out}$ in order to obtain the desired output weight matrix.

Step 4: Exploitation.

The network ($\mathbf{W}^{in}$, $\mathbf{W}$, $\mathbf{W}^{back}$, $\mathbf{W}^{out}$) is now ready for use. It can be driven by novel input sequences $\mathbf{u}(n)$, using the update equations (9.7) and (9.8), which we repeat here for convenience:

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)), \tag{9.52}$$
$$\mathbf{y}(n+1) = \mathbf{f}^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n)). \tag{9.53}$$

If stability problems are encountered when using the trained network, it very often helps to add some small noise during sampling, i.e. to use an update equation

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{d}(n) + \nu(n)), \tag{9.54}$$

where $\nu(n)$ is a small uniform white noise term (typical sizes 0.0001 to 0.01). The rationale behind this is explained in Jaeger 2001.

9.3.5 Why echo states?

Why must the DR have the echo state property to make the approach work?

From the perspective of systems engineering, the (unknown) system's dynamics is governed by an update equation of the form

$$\mathbf{d}(n) = e(\mathbf{u}(n), \mathbf{u}(n-1), ..., \mathbf{d}(n-1), \mathbf{d}(n-2)), \tag{9.55}$$

where e is a (possibly highly complex, nonlinear) function of the previous inputs and system outputs. (9.55) is the most general possible way of describing a deterministic, stationary system. In engineering problems, one typically considers simpler versions, for example, where e is linear and has only finitely many arguments (i.e., the system has finite memory). Here we will however consider the fully general version (9.55).

The task of finding a black-box model for an unknown system (9.55) amounts to finding a good approximation to the system function e. We will assume an ESN with linear output units to facilitate notation. Then, the network output of the trained network is a linear combination of the network states, which in turn are governed by the echo functions, see (9.49). We observe the following connection between (9.55) and (9.49):

$$\begin{aligned} & e(\mathbf{u}(n), \mathbf{u}(n-1), ..., \mathbf{d}(n-1), \mathbf{d}(n-2)) = \\ & \quad = \mathbf{d}(n) \\ & \quad \approx \mathbf{y}(n) \\ & \quad = \sum w_i^{out} x_i(n) \\ & \quad = \sum w_i^{out} e_i(\mathbf{u}(n), \mathbf{u}(n-1), ..., \mathbf{y}(n-1), \mathbf{y}(n-2)) \end{aligned} \tag{9.56}$$

It becomes clear from this equation how the desired approximation of the system function e can be interpreted as a linear combination of echo functions e_i. This transparent interpretatation of the system approximation task directly relies on the interpretation of network states as echo states. The arguments of e and e_i are identical in nature: both are collections of previous inputs and system (or network, respectively) outputs. Without echo states, one could neither mathematically understand the relationship between network output and original system output, nor would the training algorithm work.

9.3.6 Liquid state machines

An approach very similar to the ESN approach has been independently explored by Wolfgang Maass et al. at Graz Technical University. It is called the "liquid state machine" (LSM) approach. Like in ESNs, large recurrent neural networks are conceived as a reservoir (called

"liquid" there) of interesting excitable dynamics, which can be tapped by trainable readout mechanisms. LSMs compare with ESNs as follows:

> LSM research focuses on modeling dynamical and representational phenomena in biological neural networks, whereas ESN research is aimed more at engineering applications.
> The "liquid" network in LSMs is typically made from biologically more adequate, spiking neuron models, whereas ESNs "reservoirs" are typically made up from simple sigmoid units.
> LSM research considers a variety of readout mechanisms, including trained feedforward networks, whereas ESNs typically make do with a single layer of readout units.

An introduction to LSMs and links to publications can be found at http://www.lsm.tugraz.at/.

9.3.7 Short term memory in ESNs

Many time-series processing tasks involve some form of *short term memory* (STM). By short-term memory we understand the property of some input-output systems, where the current output $\mathbf{y}(n)$ depends on earlier values $\mathbf{u}(n\text{-}k)$ of the input and/or earlier values $\mathbf{y}(n\text{-}k)$ of the output itself. This is obvious, for instance, in speech processing. Engineering tasks like suppressing echos in telephone channels or the control of chemical plants with attenuated chemical reactions require system models with short-term memory capabilities.

We saw in Section 9.3.3 that the DR unit's activations $x_i(n)$ can be understood in terms of echo functions e_i which maps input/output histories to the current state. We repeat the corresponding Equation (9.49) here for convenience:

$$(9.57)\qquad \begin{aligned} e_i &: (U \times D)^{-\mathbb{N}} \to \mathbb{R} \\ &\left(\ldots,(\mathbf{u}(-1),\mathbf{d}(-2)),(\mathbf{u}(0),\mathbf{d}(-1))\right) \mapsto x_i(0) \end{aligned}$$

The question which we will now investigate more closely is how many of the previous inputs/output arguments ($\mathbf{u}(n\text{-}k)$, $\mathbf{y}(n\text{-}k\text{-}1)$) are actually relevant for the echo function? or asked in other words, how long is the effective short-term memory of an ESN?

A good intiuitive grasp on this issue is important for successful practical work with ESNs because as we will see, with a suitable setup of the DR, one can control to some extent the short-term memory characteristics of the resulting ESN model.

We will provide here only an intuitive introduction; for a more detailed treatment consult the technical report devoted to short-term memory (Jaeger 2001a).

9.3.9 First example: training an ESN as a delay line

Much insight into the STM of ESNs can be gained when we train ESNs on a pure STM task. We consider an ESN with a single input channel and many output channels. The input $u(n)$ is a white noise signal generated by sampling at each time independently from a uniform

distribution over [–0.5, 0.5]. We consider delays k = 1, 2, ... For each delay k, we train a separate output unit with the training signal $d_k(n) = u(n\text{-}k)$. We do not equip our network with feedback connections from the output units to the DR, so all output units can be trained simultaneously and independently from each each other. Figure 9.15 depicts the setup of the network.

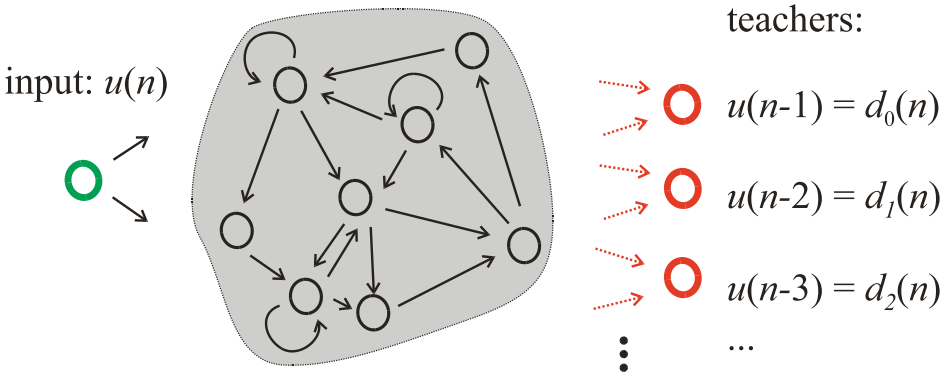


Figure 9.15: Setup of delay learning task.

Concretely, we use a 20-unit DR with a connectivity of 15%, that is, 15% of the entries of the weight matrix **W** are non-null. The non-null weights were sampled randomly from a uniform distribution over [–1,1], and the resulting weigth matrix was rescaled to a spectral radius of α = 0.8, as described in Section 9.3.3. The input weights were put to values of –0.1 or +0.1 with equal probability. We trained 4 output units with delays of k = 4, 8, 16, 20. The training was done over 300 time steps, of which the first 100 were discarded to wash out initial transients. On test data, the trained network showed testing mean square errors of MSE_{test}.= 0.0000047, 0.00070, 0.040, 0.12 for the four trained delays. Figure 9.16 (upper diagrams) shows an overlay of the correct delayed signals (solid line) with the trained network output.

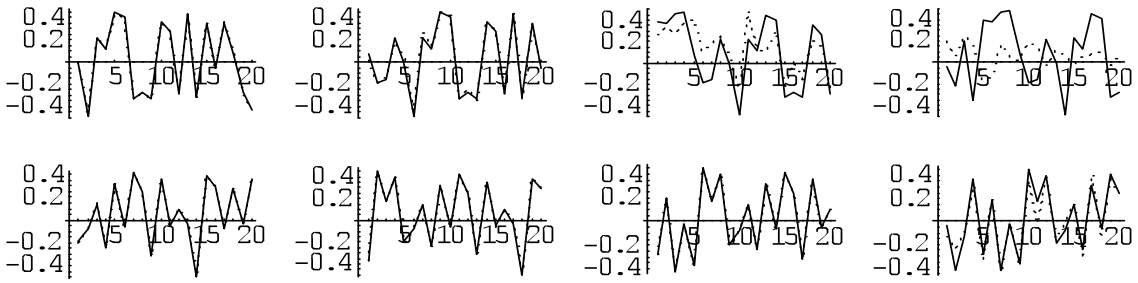


Figure 9.16: Results of training delays k = 4, 8, 16, 20 with a 20-unit DR. Top row: input weights of size –0.1 or +0.1, bottom row: input weights sized –0.001 or +0.001.

When the same experiment is redone with the same DR, but with much smaller input weights set to random values of –0.001 or +0.001, the performance greatly improves: testing errors MSE_{test}.= 0.000035, 0.000038, 0.000034, 0.0063 are now obtained.

Three fundamental observations can be gleaned from this simple example:

> The network can master the delay learning task, which implies that the current network state $\mathbf{x}(n)$ retains extractable information about previous inputs $u(n\text{-}k)$.
> The longer the delay, the poorer the delay learning performance.

The smaller the input weights, the better the performance.

9.3.10 Theoretical insights on short term memory

I now report some theoretical findings (from Jaeger 2001a), which explain the observations made in the previous subsection.

First, we need a precise version of the intuitive notion of "network performance for learning the *k*-delay task". We consider the correlation coefficient $r(u(n-k), y_k(n))$ between the correct delayed signal $u(n-k)$ and the network ouput $y_k(n)$ of the unit trained on the delay *k*. It ranges between –1 and 1. By squaring it, we obtain a quantity called in statistics the *determination coefficient* $r^2(u(n-k), y_k(n))$. It ranges between 0 and 1. A value of 1 indicates perfect correlation between correct signal and network output,k a value of 0 indicates complete loss of correlation. (In statistical terms, the determination coefficient gives the proportion of variance in one signal explained by the other). Perfect recall of the *k* –delayed signal thus would be indicated by $r^2(u(n-k), y_k(n)) = 1$, complete failure by $r^2(u(n-k), y_k(n)) = 0$.

Next, we define the overall delay recalling performance of a network, as the sum of this coefficient over all delays. We define the *memory capacity MC* of a network by

$$MC = \sum_{k=1}^{\infty} r^2(u(n-k), y_k(n)) \tag{9.58}$$

Without proof, we cite (from Jaeger 2001a) some fundamental results concerning the memory capacity of ESNs:

Proposition 9.2. In a network whose DR has N nodes, $MC \leq N$. That is, the maximal possible memory capacity is bounded by DR size.

Proposition 9.3. In a linear network with N nodes, generically $MC = N$. That is, a linear network will generically reach maximal network capacity. Notes: (i) a linear network is a network whose internal units have a linear transfer function, i.e. $f = \mathbf{id}$. (ii) "Generically" means: if we randomly construct such a network, it will have the desired property with probability one.

Proposition 9.4. In a linear network, long delays can never be learnt better than short delays ("monotonic forgetting")

When we plot the determination coefficient against the delay, we obtain the *forgetting curves* of an ESN. Figure 9.17 shows some forgetting curves obtained from various 400-unit ESNs.

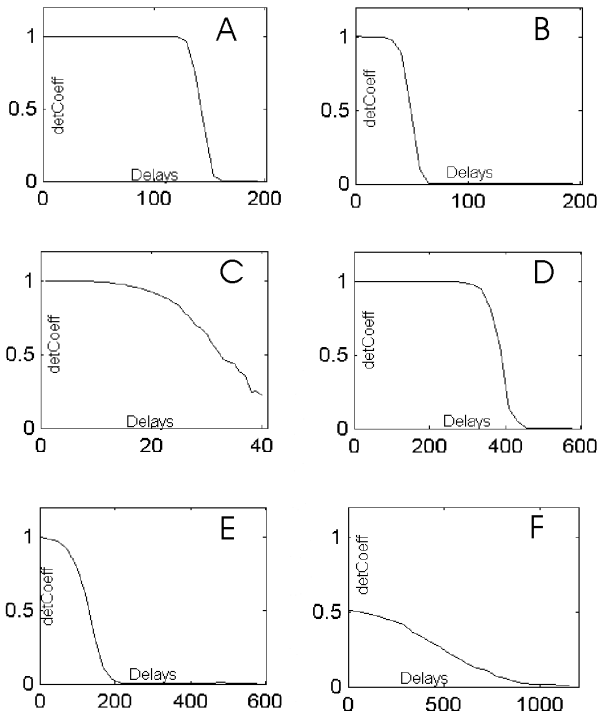


Figure 9.17: Forgetting curves of various 400-unit networks. A: randomly created linear DR. B: randomly created sigmoid DR. C: like A, but with noisy state update of DR. D: almost unitary weight matrix, linear update. E: same as D, but with noisy state update. F: same as D, but with spectral radius $\alpha = 0.999$. Mind the different scalings of the x-axis!

The forgetting curves in Figure 9.17 exhibit some interesting phenomena:

According to theorem 9.3, the forgetting curve in curve A should reflect a memory capacity of 400 (= network size). That is, the area under the curve should be 400. However, we find an area (= memory capacity) of about 120 only. This is due to rounding errors in the network update. The longer the delay, the more severe the effect of accumulated rounding errors, which reduce the effectively achievable memory capacity.

The curve B was generated with a DR made from the same weight matrix as in A, but this time, the standard sigmoid transfer function tanh was used for network update. Compared to A, we observe a drop in memory capacity. It is a general empirical observation that the more nonlinear a network, the lower its memory capacity. This also explains the finding from Section 9.3.9, namely, that the STM of a network is improved when the input weights are made very small. Very small input weights make the input drive the network only minimally around its zero resting state. Around the zero state, the sigmoids are almost linear. Thus, small input weights yield an almost linear network dynamics, which is good for its memory capacity.

Curve C was generated like A, but the (linear) network was updated with a small noise term added to the states. As can be expected, this decreases the memory capacity. What is worth mentioning is that the effect is quite strong. An introductory discussion can be found in Jaeger (2001a).

The forgetting curve D comes close to the theoretical optimum of $MC = 400$. The trick was to use an almost unitary weight matrix, again with linear DR units. Intuitively, this means that a network state $\mathbf{x}(n)$, which carries the information about the current input, "revolves" around the state space $\mathbb{R}^N$ without interaction with succeeding states,

for *N* update steps. There is more about this in Jaeger (2001a) and Bertschinger (2002)[25].
The forgetting curve E was obtained from the same linear unitary network as D, but noise was added to state update. The corruption of memory capacity is less dramatic as in curve C.
Finally, the forgetting curve F was obtained by scaling the (linear, unitary) network from D to a spectral radius $\alpha = 0.999$. This leads to long-term "reverberations" of input. On the one hand, this yields a forgetting curve with a long tail – in fact, it extends far beyond the value of the network size, $N = 400$. On the other hand, "reverberations" of long-time past inputs still occupying the present network state lead to poor recall of even the immediately past input: the forgetting curve is only about 0.5 right at the beginning. The area under the forgetting curve, however, comes close to the theoretical optimum of 400.

For practical purposes, when one needs ESNs with long STM effects, on can resort to a combination of the following approaches:

Use large DRs. This is the most efficient and generally applicable method, but it requires sufficiently large training data sets.
Use small input weights, to work in the almost linear working range of the network. This might conflict with nonlinear task characteristics.
Use linear update for the DR. Again, might conflict with nonlinear task characteristics.
Use specially prepared DRs with almost unitary weight matrices.
Use a spectral radius α close to 1. This would work only with "slow" tasks (for instance, it would not work if one wants to have fast oscillating dynamics with long STM effects).

9.3.11 Tricks of the trade

The basic idea of ESNs for black-box-modeling can be condensed into the following statement:

"Use an excitable system [the DR] to give a high-dimensional dynamical representation of the task input dynamics and/or output dynamics, and extract from this reservoir of task-related dynamics a suitable combination to make up the desired target signal."

Obviously, the success of the modeling task depends crucially on the nature of the excited dynamics – it should be adapted to the task at hand. For instance, if the target signal is slow, the excited dynamics should be slow, too. If the target signal is very nonlinear, the excited dynamics should be very nonlinear, too. If the target signal involves long short-term memory, so should the excited dynamics. And so forth.

Successful application of the ESN approach, then, involves a good judgement on important characteristics of the dynamics excited inside the DR. Such judgement can only grow with the experimenter's personal experience. However, a number of general practical hints can be given which will facilitate this learning process. All hints refer to standard sigmoid networks.

[25] N. Bertschinger (2002), Kurzzeitspeicher ohne stabile Zustände in rückgekoppelten neuronalen Netzen. Diplomarbeit, Informatik VII, RWTH Aachen, 2002 (in German)
http://www.igi.tugraz.at/nilsb/publications/DABertschinger.pdf

Plot internal states

Since the dynamics within the DR is so essential for the task, you should always visually inspect it. Plot some of the internal units $x_i(n)$ during sampling and/or testing. These plots can be very revealing about the cause of failure. If things don't go well, you will frequently observe in these plots one or two of the following misbehaviors:

> **Fast oscillations**. In tasks where you don't want fast oscillations, this observation indicates a too large spectral radius of the weight matrix **W**, and/or too large values of the output feedback weights (if they exist). Remedy: scale them down.
> **Almost saturated network states**. Sometimes you will observe that the DR units almost always take extreme values near 1 or –1. This is caused by a large impact of incoming signals (input and/or output feedback). It is only desirable when you want to achieve some almost binary, "switching" type of target dynamics. Otherwise it's harmful. Remedy: scale down the input and/or output feedback weights.

Plot output weights

You should always inspect the output weights obtained from the learning procedure. The easiest way to do this is to plot them. They should not become too large. Reasonable absolute values are not greater than, say, 50. If the learnt output weights are in the order of 1000 and larger, one should attempt to bring them down to smaller ranges. Very small values, by contrast, do not indicate anything bad.

When judging the size of output weights, however, you should put them into relation with the range of DR states. If the DR is only minimally excited (let's say, DR unit activations in the range of 0.005 – this would for instance make sense in almost linear tasks with long-term memory characteristics), and if the desired output has a range up to 0.5, then output weights have to be around 100 just in order to scale up from the internal state range to the output range.

If after factoring out the range-adaptation effect just mentioned, the output weights still seem unreasonably large, you have an indication that the DR's dynamics is somehow badly adapted to your learning task. This is because large output weights imply that the generated output signal exploits subtle differences between the DR unit's dynamics, but does not exploit the "first order" phenomena inside the DR (a more mathematical treatment of large output weights can be found in Jaeger (2001a)).

It is not easy to suggest remedies against too large output weights, because they are an indication that the DR is generally poorly matched with the task. You should consider large output values as a symptom, not as the cause of a problem. Good doctors do not cure the symptoms, but try to address the cause.

Large output values will occur only in high-precision tasks, where the training material is mathematical in origin and intrinsically very accurate. Empirical training data will mostly contain some random noise component, which will lead to reasonably scaled output weights anyway. Adding noise during training is a safe method to scale output weights down, but is likely to impair the desired accuracy as well.

Find a good spectral radius

The single most important knob to tune an ESN is the spectral radius of the DR weight matrix. The general rule: for fast, short-memory tasks use small α, for slow, long-memory tasks use large α. Manual experimentation will be necessary in most cases. One does not have to care about finding the precise best value for α , however. The range of optimal settings is relatively broad, so if an ESN works well with $\alpha = 0.8$, it can also be expected to work well with $\alpha = 0.7$ and with $\alpha = 0.85$. The closer you get to 1, the smaller the region of optimality.

Find an appropriate model size

Generally, with larger DR one can learn more complex dynamics, or learn a given dynamics with greater accuracy. However, beware of overfitting: if the model is too powerful (i.e. the DR too large), irrelevant statistical fluctuations in the training data will be learnt by the model. That leads to poor generalization on test data. Try increasing network sizes until performance on test data deteriorates.

The problem of overfitting is particularly important when you train on empirical, noisy data. It is not theoretically quite clear (at least not to me) whether the concept of overfitting also carries over to non-statistical, deterministic, 100%-precisely defined training tasks, for example training a chaotic attractor described by a differential equation (as in Jaeger 2001). The best results I obtained in that task were achieved with a 1000-unit network trained on 2000 data points, which means that 1000 parameters were estimated from 2000 data points. For empirical, statistical tasks, this would normally lead to overfitting (a rule of thumb in statistical learning is to have at least 10 data points per estimated parameter).

Add noise during sampling

When you are training an ESN with output feedback from accurate (mathematical, noise-free) training data, stability of the trained network is often difficult to achieve. A method that works wonders is to inject noise into the DR update during sampling, as described in Section 9.3.2, Eqn. (9.47). It is not clearly understood why this works. Attempts at an explanation are made in Jaeger (2001); Bertschinger (2002) provides a more extensive analysis.

In tasks with empirical, noisy training data, noise insertion does not a priori make sense. Nor is it required when there are no output feedback connections.

There is one situation, however, where noise insertion might make sense even with empirical data and without output feedback connections. That is when the learnt model overfits data, which is revealed by a small training and a large test error. In this condition, injection of extra noise works as a *regularizer* in the sense of statistical learning theory. The training error will increase, but the test error will go down. However, a more appropriate way to avoid overfitting is to use smaller networks.

Use an extra bias input

When the desired output has a mean value that departs from zero, it is a good idea to invest an extra input unit and feed it with a constant value ("bias") during training and testing. This bias input will immediately enable the training to set the trained output to the correct mean value.

A relatively large bias input will shift many internal units towards one of the extremer outer ranges of their sigmoids; this might be advisable when you want to achieve a strongly nonlinear behavior.

Sometimes you do not want to affect the DR strongly by the bias input. In such cases, use a small value for the bias input (for instance, a value of 0.01), or connect the bias only to the output (i.e., put all bias-to-DR connections to zero).

Beware of symmetric input

Standard sigmoid networks with the tanh sigmoid are "symmetric" devices in the sense that when an input sequence $u(n)$ gives an output sequence $y(n)$, then the input sequence $-u(n)$ will yield an output sequence $-y(n)$. For instance, you can never train an ESN to produce an output $y(n) = u(n)^2$ from an input $u(n)$ which takes negative and positive values. There are two simple methods to succeed in "asymmetric" tasks:

Feed in an extra constant bias input. This will effectively render the DR an unsymmetric device.
Shift the input. Instead of using the original input signal, use a shifted version which only takes positive sign.

The symmetric-input fallacy comes in many disguises and is often not easy to recognize. Generally be cautious when the input signal has a range that is roughly symmetrical around zero. It almost never harms to shift it into an asymmetrical range. Nor does a small bias input usually harm.

Shift and scale input

You are free to transform the input into any value range $[a, b]$ by scaling and/or shifting it. A rule I work with: the more nonlinear the task, the more extravagantly I shift the input range. For instance, in a difficult nonlinear system identification task (30th order NARMA system) I once got best models with an input range $[a, b] = [3, 3.5]$. The apparent reason is that shifting the input far away from zero made the DR work in a highly nonlinear range of its sigmoid units.

Blackbox modeling generals

Be aware of the fact that ESN is a blackbox modeling technique. This means that you cannot expect good results on test data which operate in a working range that was never visited during training. Or to put it the other way round, make sure that your training data visit all the working conditions that you will later meet when using the trained model.

This is sometimes not easy to satisfy with nonlinear systems. For instance, a typical approach to obtain training data is to drive the empirical system with white noise input. This approach is well-motivated with linear systems, where white noise input in training data generally reveals the most about the system to be identified. However, this may not be the case with nonlinear systems! By contrast, what typically happens is that white noise input keeps the nonlinear system in a small subregion of its working range. When the system (the original system or the trained model) is later driven with more orderly input (for instance, slowly changing input), it will be driven into a quite different working range. A black-box model

trained from data with white noise input will be unable to work well in another working range.

On the other hand, one should also not cover in the training data more portions of the working range than will be met in testing / exploitation. Much of the modeling capacity of you model will then be used up to model those working regions which are later irrelevant. As a consequence, the model accuracy in the relevant working regions will be poorer.

The golden rule is: use basically the same kind of input during training as you will later encounter in testing / exploitation, but make the training input a bit more varied than you expect the input in the exploitation phase.

10 Hidden Markov Models and the EM Learning Algorithm

10.1 Introduction

In this section we will learn about how models of discrete-valued stochastic processes can be learnt from data. Think of a "discrete-valued stochastic process" simply as some mechanism that generates *random symbol sequences* – for instance, sequences of dice throws (the symbols would then be 1,...,6) or ASCII texts (the symbols then would be the ASCII symbols) or amino acid sequences. Our learning task will be the following: given a (long) observed sequence $x(0)\ x(1)\ x(2)...$ generated by an unknown generating mechanism, learn from this sample string a model $\mathcal{M}$ that then can replace the original generator. The model can be used in various way, for instance to generate new sequences (useful for running simulations), or to predict a given sequence into the future, or to classify sequences, and many more.

Specifically, we will consider a type of models known as *hidden Markov models* (HMMs). HMMs have important applications. Here is a choice:

- Virtually every speech recognition system is made from HMMs.
- HMMs are becoming more and more the standard tool for biosequence analysis.
- In communications engineering, HMMs are used to predict subchannel load in order to optimize throughput.
- In robotics, an input-output version of HMMs, called *partially observable Markov decision processes* (POMDPs) is a standard type of "world model" that enables a robot to predict the consequences of its actions.

HMMs are quite powerful – very complicated processes can be modelled by them, and especially, processes that have a *memory*. A process is said to have memory when the probabilities of future observations $x(n + 1)\ x(n + 2)...$ depend not only on the current observation $x(n)$ but also on previous observations $x(n – 1)$, $x(n – 2)$ ... . English texts are processes with a very strong memory component: if you only know that $x(n)$ = `e`, and you would be asked how this process continues, you could hardly make a guess. But if you know that $x(n– 21)$ ... $x(n– 1)\ x(n)$ = `my_mother_and_my_fathe`, then there is an overwhelming probability that $x(n + 1)$ = `r`.

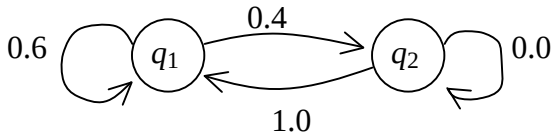
HMMs are equipped with a learning algorithm called the EM-algorithm, or more specifically, with a particular subtype of the EM-algorithm called the Baum-Welch algorithm. EM algorithms are a large family of learning algorithms with applications in many fields of statistics and machine learning. We will learn both the general EM principle and the specific Baum-Welch variety that is tailored to HMMs.

Although hidden Markov processes have been known and investigated in mathematical statistics since at least 50 years, they became popular in applications only in the early 90-ies. In a small number of years, the field of automated speech recognition was completely taken over by HMM techniques. The trigger for the surge of popularity of HMMs was a tutorial text written by L. R. Rabiner in 1989, which is still one of the main teaching texts on HMMs and is cited in almost every article on HMMs. An electronic version of this tutorial can be found at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/RabinerTutorialHMM.pdf.

Hidden Markov processes generalize Markov processes (also called Markov chains), and we first must understand this simpler kind of stochastic process.

A Markov chain describes random sequences from a finite set $S = \{q_1, ..., q_m\}$ of *states*. The evolution of such sequences is governed by a simple probabilistic law: if at some time n the process has generated a state q_i, then the next generated state is chosen from S with fixed conditional probabilities $P(X_{n+1} = q_j \mid X_n = q_i)$ that depend only on q_i (we denote by X_n the random variable that returns the state at time n). Two convenient ways to represent these conditional probabilities are the *transition graph* and the *transition matrix*.

Example 10.1. Here is a simple Markov process with states $S = \{q_1, q_2\}$ represented by its transition graph (left) and transition matrix (right):



$$\begin{bmatrix} 0.6 & 0.4 \\ 1.0 & 0.0 \end{bmatrix}$$

Figure 10.1 A very simple Markov chain.

In words, this processed is characterized by the following facts:

- If the current state is q_1, then the next state is q_1 with probability 0.6 and q_2 with probability 0.4.
- If the current state is q_2, then the next state is q_1 with probability 1.0 and q_2 with probability 0.0.

More formally, the transition matrix for an m-state Markov chain is

(10.1) $$M = (p_{ij})_{i,j=1,...,m} = (P(X_{n+1} = q_j \mid X_n = q_i))_{i,j=1,...,m}$$

It should be clear that the probability labels of all arcs that leave a state node in the graph sum to 1, and that the rows of a transition matrix are non-negative and each sum to 1. Matrices with this property are also called *stochastic matrices* or *Markov matrices*. Since a Markov matrix only specifies probabilities of *next* states given a *previous* state, in order to specify a stochastic process one must additionally know the probabilities of the *first* state. This is a probability vector $\mathbf{w}_0 = (P(X_0 = q_1), ..., P(X_0 = q_m))^T$. A pair $(M, \mathbf{w}_0)$ uniquely specifies a Markov chain. If you are given $(M, \mathbf{w}_0)$, you can generate random sequences as follows:

1. Choose the first state according to the probabilities in $\mathbf{w}_0$.
2. If the n-th state you have generated is q_i, choose the $(n+1)$th state according to the probabilities you find in the i-th row of M.

The probability of a sequence $q_{i_0} q_{i_1} ... q_{i_{N-1}}$ is the product of the individual transition probabilites involved, times the starting probability of q_{i_0}:

$$
\begin{aligned}
&P(X_0 = q_{i_0}, ..., X_{N-1} = q_{i_{N-1}}) = \\
&\quad = \quad P(X_{N-1} = q_{i_{N-1}} | \, X_{N-2} = q_{i_{N-2}}) \, ... \, P(X_1 = q_{i_1} | \, X_0 = q_{i_0}) \, P(X_0 = q_{i_0}) \\
(10.2) \qquad &\quad = \quad P(X_0 = q_{i_0}) \prod_{n=1}^{N-1} P(X_n = q_{i_n} | \, X_{n-1} = q_{i_{n-1}}) \\
&\quad = \quad P(X_0 = q_{i_0}) \prod_{n=1}^{N-1} p_{i_{n-1} \, i_n}
\end{aligned}
$$

Eq. (10.2) is the fundamental equation for Markov chains. If for some process Eq. (10.2) holds, it is a Markov chain and vice versa.

Note that Markov chains have no memory. The probabilities of the next state choices depend only on the current state, not on any that came before.

A note on a rigorous probability-theoretic treatment of stochastic processes (which we don't do). Mathematically, a stochastic process (with disrete time $n = 0, 1, 2, ...$) is a sequence of random variables $(X_n)_{n = 1, 2, ...}$, where all the X_i share the same observation space E (in our Markov chain example, $E = S$ is the finite set of states). The intuitive interpretation of X_i is to consider it as the *observation of the process at time i*. Each $\omega \in \Omega$ is corresponds to one *realization* of the process, that is, the sequence $(X_n(\omega))_{n = 1, 2, ...} \in E^{\mathbb{N}}$ is an observed *path*, or *time series*, of the process.

It is always technically possible to identify the underlying probability space Ω with the set of all paths, that is, one may assume $\Omega = S^{\infty}$. That this representation of Ω is always possible is the essence of a quite nontrivial theorem due to Kolmogorov. If one were 100% correct, one would write $P(X_0(\omega) = q_{i_0}, ..., X_{N-1}(\omega) = q_{i_{N-1}})$ instead of $P(X_0 = q_{i_0}, ..., X_{N-1} = q_{i_{N-1}})$. But instead of being 100% correct, one is rather more often 50% sloppy and writes simply $P(q_{i_0} ... q_{i_{N-1}})$ instead of $P(X_0 = q_{i_0}, ..., X_{N-1} = q_{i_{N-1}})$.

10.3 Hidden Markov Processes

Intuitive description. A hidden Markov model (HMM) [or hidden Markov process] is a generating mechanism for stochastic symbol sequences that consists of two cascaded stochastic mechanisms. First there is a Markov chain with a state set $S = \{q_1, ..., q_m\}$ and associated starting and transition probabilities. However, the states q_i are not observable – when the Markov chain "runs", its states are *hidden*. Instead, a state q_i "emits", when the chain passes through that state, an observable symbol from another alphabet $\Sigma = \{a_1, ..., a_k\}$ according to a probability distribution $P(a_j | \, q_i)$ that is characteristic (and fixed) for that state. Our example 10.1 turns into a HMM if we equip it with such observables:

Example 10.2. This transition graph represents a HMM made from the Markov chain from Example 10.1, with observables $\Sigma = \{a, b\}$:

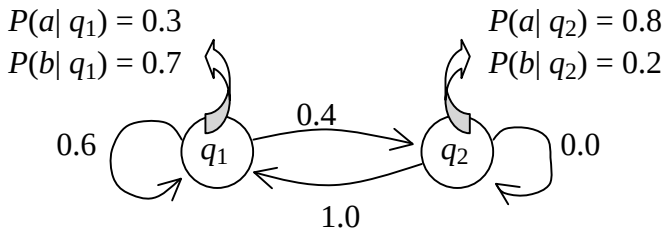


Figure 10.2 The Markov chain from Figure 10.1 emitting observable symbols from its states.

The symbol sequences that are generated by HMMs are sequences from the alphabet of observables, Σ, *not* from the set of states, S. So the HMM from Example 10.2 would generate strings like *abbaab*... .

One intuitive interpretation of HMM processes is that the emitted observables are noisy measurements of their states.

Here is the formal definition:

Definition 10.1 A hidden Markov model is a quintuple $H = (S, \Sigma, M, E, \mathbf{w}_0)$, where $S = \{q_1, ..., q_m\}$ is a finite non-empty set of *hidden states*, $\Sigma = \{a_1, ..., a_k\}$ is an alphabet of *observables*, M is an $m \times m$ stochastic matrix of *state transition probabilities*, E is an $m \times k$ matrix of *emission probabilities*, and $\mathbf{w}_0$ is an m-dimensional probability vector, the *starting vector*. The rows of E must be probability vectors.

Note: the emission matrix E contains in its i-th row the emission probabilities for the k symbols from Σ, that is,

$$E = (e_{ij})_{i=1,...,m;\ j=1,..,k} = (P(a_j|\ q_i))_{i=1,...,m;\ j=1,..,k}\ . \tag{10.3}$$

An HMM $H = (S, \Sigma, M, E, \mathbf{w}_0)$ describes two stochastic processes $(X_i)_{i \in \mathbb{N}}$, $(Y_i)_{i \in \mathbb{N}}$. The first process describes the Markov chain through the states S according to the Markov transition matrix M – that is, the random variables X_i take values in S, and the index i is interpreted as a discrete time. The other process, $(Y_i)_{i \in \mathbb{N}}$, takes values in Σ and describes the sequence of observable symbols. Formally, the joint probability that the process (X_i) goes through a state sequence $q_{i_0}\ q_{i_1}\ ...\ q_{i_{N-1}}$ while the observed symbols are $a_{j_0}\ a_{j_1}\ ...\ a_{j_{N-1}}$, is determined by:

$$\begin{aligned} &P(X_0 = q_{i_0}, ..., X_{N-1} = q_{i_{N-1}}, Y_0 = a_{j_0}, ..., Y_{N-1} = a_{j_{N-1}}) = \\ &= P(Y_0 = a_{j_0} \mid X_0 = q_{i_0})\ P(X_0 = q_{i_0})\ \prod_{n=1}^{N-1} P(Y_n = a_{j_n} \mid X_n = q_{i_n})\ P(X_n = q_{i_n}|\ X_{n-1} = q_{i_{n-1}}) \\ &= e_{i_0 j_0}\ \mathbf{w}_{0\, i_0} \prod_{n=1}^{N-1} e_{i_n j_n}\ p_{i_{n-1} i_n}. \end{aligned} \tag{10.4}$$

One often illustrates the interplay between the "hidden" Markov process (X_i) and the observable process (Y_i) graphically in the following way (see Figure 10.3).

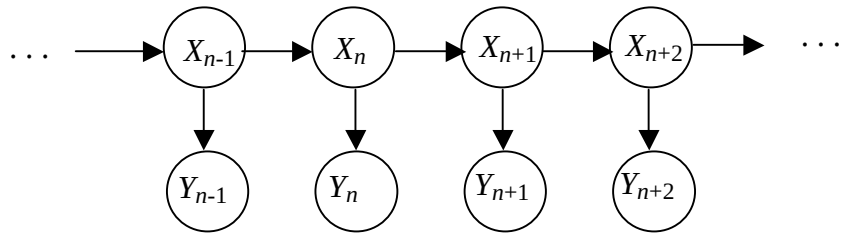
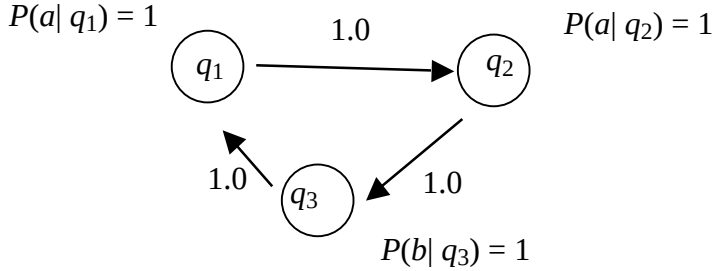


Figure 10.3 A graphical representation of the statistical (in-)dependencies prevailing in an HMM

Each random variable is represented by a node in a directed graph, which are connected by links in a specific way. In such (directed) *graphical models* some random variable X is statistically independent from all its non-descendants[26], given its parents[27]. Thus, for instance, the graph structure of Figure 10.3 tells us that the random variable X_{n+1} is independent of X_{n-1}, given X_n – that is, $P(X_{n+1} \mid X_n,\ X_{n-1}) = P(X_{n+1} \mid X_n)$. Or, it would also tell us that X_{n+1} is independent of Y_n, given X_n. Do not confound graphical representations like in Figure 10.3 (where graph nodes are random variables) with transition graphs as in Figure 10.2 (where nodes are states, that is, values of random variables)! both are quite common, but have a very different interpretation.

The observable process $Y_0, Y_1, Y_2, ...$ is a process with memory! This is at first counter-intuitive, because the underlying hidden process is a Markov chain and has no memory. But recall that for a process $Y_0, Y_1, Y_2, ...$ having *no* memory means that we don't gain any additional information about future probabilities of $Y_{n+1}, Y_{n+2}, ...$ if we learn about previous observations $Y_{n-1}, Y_{n-2}, ...$: all possible information about the future is given with Y_n. But this is not the case with HMMs. We can indeed give a *better* estimate of the future distribution of $Y_{n+1}, Y_{n+2}, ...$ if we are informed about the outcome of $Y_{n-1}, Y_{n-2}, ...$ The reason is that if we learn about $Y_{n-1}, Y_{n-2}, ...,$ then we implicitly gain knowledge about the current hidden state X_n, which in turn sharpens our predictions about the observable future.

Here is a simple demonstration of the memory effect. Consider the following HMM:



[26] The descendants of a node k in a directed graph are all nodes that can be reached by iteratively traversing outgoing links from k in forward direction.
[27] The parents of a node k in a directed graph are all nodes from which a single link leads to k.

This HMM cycles (deterministically) through the states q_1, q_2, q_3 while emitting (likewise deterministically) symbols *a*, *a*, *b*. If one observes a single "*a*", then the next symbol to be observed can either be an *a* or a *b*. If one however observes one symbol before the "*a*", then the next symbol becomes fully determined. For instance, observing "*ba*" allows one to infer that the next symbol must be an "*a*" – the *b* that came before the *a* provided this additional information, which is another way of stating that there is memory in this process.

This memory effect may on a grander scale also be illustrated with our example from the Introduction. Assume that the text string that starts with `my_mother_and` was generated by a human brain, and assume furthermore that the workings of that brain can be described by a Markov chain (with a very large state number). Any physicist worth his/her salt would subscribe to that view. We (= the readers of the string) don't have access to that Markov chain, it is hidden to us. So the text string `my_mother_and`... can be considered as a sequence of observables that was emitted from the hidden brain state sequence. Now if we, the readers, only know that Y_{27} = `e`, we can't infer much about the text producing brain's state, and our predictions about the continuation of the sequence after this `e` are accordingly vague. If however we learn about the previous portion of the sequence, `my_mother_and_my_fathe`, then we learn a lot about the underlying brain state (this brain is currently reasoning about his/her parents), and this helps us (together with general knowledge about English) to predict with almost certainty that the next symbol will be `r`.

Equation (10.4) is not really useful, because it makes explicit use of the hidden state sequence, which is typically not known in HMM processes. In real-life applications, all one has is a sequence of observations $a_{j_0}\, a_{j_1} \ldots a_{j_{N-1}}$. Several questions are of interest:

1. Given $a_{j_0}\, a_{j_1} \ldots a_{j_{N-1}}$, what is the most probable state sequence?
2. Given $a_{j_0}\, a_{j_1} \ldots a_{j_{N-1}}$, what is the probability of this sequence?
3. Given $a_{j_0}\, a_{j_1} \ldots a_{j_{N-1}}$ and a time $n \leq N - 1$, what is the probability that the hidden state at time n is q_i?

There are standard algorithms that provide answers to these basic questions, which we will now treat in turn. Formally, the first question is to find that state sequence $q_{i'_0}\, q_{i'_1} \ldots q_{i'_{N-1}}$ that is the most probable, given the sequence of observations:

$$(10.5)\quad q_{i'_0}\, q_{i'_1} \ldots q_{i'_{N-1}} = \underset{q_{i_0} \ldots q_{i_{N-1}}}{\arg\max}\ P(X_0 = q_{i_0}, \ldots, X_{N-1} = q_{i_{N-1}} \mid Y_0 = a_{j_0}, \ldots, Y_{N-1} = a_{j_{N-1}}).$$

This maximally probable state sequence is called the *Viterbi sequence*. It can be obtained from the observed sequence $a_{j_0}\, a_{j_1} \ldots a_{j_{N-1}}$ by the *Viterbi algorithm*, a straightforward dynamic programming algorithm. It runs as follows. Let $v_k(l)$ be the probability of the most probable state/observation sequence up to time l that ends in state q_k with observation a_{j_l}, that is,

$$(10.6)\qquad v_k(l) = \max_{q_{i_0} \ldots q_{i_{l-1}}} P(X_0 = q_{i_0}, \ldots, X_{l-1} = q_{i_{l-1}}, X_l = q_k, Y_0 = a_{j_0}, \ldots, Y_l = a_{j_l}).$$

For the (l+1)th time step, these probabilities can be computed from the ones from time l by

(10.7) $$v_{k'}(l+1) = e_{k' j_{l+1}} \max_k (v_k(l)\, p_{kk'}).$$

The initialization is given by

(10.8) $$v_k(0) = e_{k j_0} P(X_0 = q_k).$$

After we have computed all $v_k(N-1)$, we can infer that the index i'_{N-1} of the last state of the Viterbi sequence is

(10.9) $$i'_{N-1} = \arg\max_k v_k(N-1).$$

The earlier states of the Viterbi sequence can be found by state backtracking: if we know that the l-th state of the Viterbi sequence has index i'_l, then the ($l-1$)th state can be found by

(10.10) $$i'_{l-1} = \arg\max_k v_k(l-1)\, p_{ki'_l}.$$

Now we answer the second standard question. The correct probability $P(a_{j_0} \ldots a_{j_{N-1}})$ can obviously be obtained by

(10.11) $$P(a_{j_0} \ldots a_{j_{N-1}}) = \sum_{q_{i_0} \ldots q_{i_{N-1}} \in S^N} P(q_{i_0} \ldots q_{i_{N-1}}, a_{j_0} \ldots a_{j_{N-1}}).$$

But this requires a sum over exponentially (in N) many paths which is infeasible. A feasible method is provided by another dynamic programming algorithm that resembles the Viterbi algorithm but has a sum instead of a max operation. We define $f_k(l)$ to be the probability that $a_{j_0}\, a_{j_1} \ldots a_{j_l}$ is observed on any state sequence that ends in q_k, that is,

(10.12) $$f_k(l) = P(X_l = q_k, Y_0 = a_{j_0}, \ldots, Y_l = a_{j_l}).$$

The recursion then is

(10.13) $$f_{k'}(l+1) = e_{k' j_{l+1}} \sum_k f_k(l)\, p_{kk'}.$$

and the initialization is again

(10.14) $$f_k(0) = e_{k j_0} P(X_0 = q_k).$$

Having computed the $f_k(N-1)$ [for $k = 1, \ldots, m$], the desired probability of an observed sequence can be obtained by

(10.15) $$P(Y_0 = a_{j_0}, \ldots, Y_{N-1} = a_{j_{N-1}}) = \sum_k f_k(N-1)$$

This algorithm is known as the *forward algorithm* to compute probabilities of observed sequences in HMM processes. Its computational time complexity is $O(N\,|S|^2)$.

Without proof (easy exercise!) we mention that there is a convenient matrix representation for the forward algorithm, which lends itself to transparent implementations. To this end, for each observable $a_j \in \Sigma$, define an $m \times m$ diagonal matrix O_j which contains the emission probabilities e_{ij} of observing a_j in states $i = 1, ..., m$ on its diagonal. Define $T_j = M^{\mathsf{T}}\, O_j$. Let $\mathbf{1}_m$ be the all-ones vector $(1, ..., 1)^{\mathsf{T}}$ of size m. Then it holds that

(10.16) $$P(Y_0 = a_{j0}, ..., Y_{N-1} = a_{jN-1}) = \mathbf{1}_m^{\mathsf{T}}\, T_{jN-1} \cdots T_{j0}\, \mathbf{w}_0.$$

The third basic task is to infer from an observation sequence $a_{j0}\, a_{j1} \ldots a_{jN-1}$ the hidden state probability at some time l, that is,

(10.17) $$P(X_l = q_k \mid a_{j0} \ldots a_{jN-1}).$$

Examples where this question is of interest:

1. If we inspect a genome sequence at a particular location, do we happen to meet what is called a "CpG-island", that is, a batch within the sequence that due to certain chemical stabilization effects has a different nucleic acid statistics than ordinary DNA?
2. If we analyze a speech signal, did the speaker intend to utter an "a" at a particular place? (that is the crucial question that has to be answered in any speech-to-text recognizer, e.g. in automated dictation systems).
3. In communication systems with noisy channels, at the receiving end one obtains a noisy and/or coded signal Y_n from which the receiver wants to recover the denoised and/or decoded "source signal" X_n such that each recovered $\hat{X}_n$ is the most probable one, given the entire signal $Y_0 \ldots Y_{N-1}$.

While the forward algorithm can be used to compute

(10.18) $$P(X_l = q_k \mid a_{j0} \ldots a_{jl}) \;=\; P(X_l = q_k, a_{j0} \ldots a_{jl}) \,/\, P(a_{j0} \ldots a_{jl})$$

this does not help, because it only gives us a clue about hidden state probabilities using the observable information up to time n. In this situation we employ a mirror version of the forward algorithm, called the *backward algorithm*, to obtain the probability to see $a_{jl+1}\, a_{jl+2} \ldots a_{jN-1}$ if the hidden process is in state q_k at time l:

(10.19) $$b_k(l) \;=\; P(Y_{l+1} = a_{jl+1}, ..., Y_{N-1} = a_{jN-1} \mid X_l = q_k),$$

which can then be combined with (10.12) to get (10.17). $b_k(l)$ can be computed recursively by

(10.20) $$b_{k'}(l-1) \;=\; \sum_k \; p_{k'k}.e_{kil}\; b_k(l),$$

and the initialization is

(10.21) $$b_k(N-1) = 1.$$

We now combine (10.19) with (10.12):

$$
\begin{aligned}
(10.22) \qquad & P(X_l = q_k \mid a_{j_0} \ldots a_{j_{N-1}}) \; = \\
& = P(q_k, a_{j_0} \ldots a_{j_{N-1}}) \,/\, P(a_{j_0} \ldots a_{j_{N-1}}) \\
& = P^{-1}(a_{j_0} \ldots a_{j_{N-1}})\, P(q_k, a_{j_0} \ldots a_{j_l})\, P(a_{j_{l+1}} \ldots a_{j_{N-1}} \mid q_k, a_{j_0} \ldots a_{j_l}) \\
& = P^{-1}(a_{j_0} \ldots a_{j_{N-1}})\, P(q_k, a_{j_0} \ldots a_{j_l})\, P(a_{j_{l+1}} \ldots a_{j_{N-1}} \mid q_k) \\
& = P^{-1}(a_{j_0} \ldots a_{j_{N-1}})\, f_k(l)\, b_k(l).
\end{aligned}
$$

Seen from an abstract perspective, the *posterior distribution* $P(X_l = q_k \mid a_{j_0} \ldots a_{j_{N-1}})$ can be used to *de-noise* or even *decode* a signal $a_{j_0} \ldots a_{j_{N-1}}$.

Notice that a direct implementation of 10.22 on a digital computer is prone to run into numerical underflow problems. While $P(X_l = q_k \mid a_{j_0} \ldots a_{j_{N-1}})$ will be a number of reasonable size (not underflowing machine precision), $P(a_{j_0} \ldots a_{j_{N-1}})$, $f_k(l)$, and $b_k(l)$ are likely too small to be accurately represented. To avoid this, practical implementations of 10.22 make use of dynamic rescaling schemes, where the intermediate quantities that arise during the iterative computations of these quantities are magnified by rescaling factors when needed. In the end, all these rescaling factors cancel out and a rescaled version of the quotient $P^{-1}(a_{j_0} \ldots a_{j_{N-1}})\, f_k(l)\, b_k(l)$ can be reliably evaluated. The Rabiner tutorial describes such a rescaling scheme. In ready-made toolboxes for HMMs these rescalings are integrated (at least they should be) and you don't have to think about them.

10.4 The expectation-maximization (EM) algorithm

The "EM algorithm" is not really an algorithm but rather a general recipe to construct algorithms for maximum likelihood estimators in situations where one has observable (= training) data Y that depend on unobservable (hidden) quantities X. The EM principle was first described in 1977 by Dempster, Laird and Rubin[28] and can be considered a landmark discovery in statistics. Here I first give an abstract account of the general principle and then describe the specific version that is used for HMM learning, called the *Baum-Welch algorithm*. I roughly follow the exposition given in a paper by Sam Roweis and Zoubin Ghahramani, *A unifying review of linear Gaussian models*[29], which also covers other versions of EM for other machine learning problems. I use however another notation than these and I supply more detail.

The general situation is the following. Let Y be (a vector of) *observable* random variables over a probability space (Ω, F, P) and let X be a vector of *hidden* random variables over the same space. Let the observation spaces of X and Y be E_X and E_Y, respectively. In our HMM example, we would have $Y = Y_0, \ldots, Y_{N-1}$ and $X = X_0, \ldots, X_{N-1}$, and $E_X = S^N$ and $E_Y = \Sigma^N$,

[28] Dempster, A.P. , Laird, N.M. and Rubin, D.B. (1977). Maximum likelihood from incomplete data via the EM algorithm (with discussion). Journal of the Royal Statistical Society series B 39, 1-38. **Cited more than 12,500 times in the ISI citation index – I know no other paper that even comes close**

[29] Roweis, S. and Ghahramani, Z., (1999) A unifying review of linear Gaussian models. Neural Computation 11(2), 305-345. Online copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/1616_RoweisGhahramani99.pdf

respectively. Let θ be a set of parameters describing a model for the joint distribution $P_{X,Y}(\theta)$ of X and Y. In the HMM example, θ would consist of all the parameters of a HMM, that is, θ = (M, E, $\mathbf{w}_0$). Now let D be a sample of the observable variables Y – in our example, D would be a sequence $a_{j_0} ... a_{j_{N-1}}$. The objective of an EM algorithm is to determine θ such that the likelihood

(10.23) $$L(\theta) = P(D \mid \theta)$$

is maximized. This is equivalent to finding θ such that the log likelihood

(10.24) $$\mathcal{L}(\theta) = \log P(D \mid \theta)$$

becomes maximal. The difficulty we are facing is that the probability $P(D \mid \theta)$ depends implicitly on the values that the hidden variables take – as becomes clear from Eq. 10.11. Let X and Y have the pdf $p(X, Y)$ w.r.t. the uniform distribution over $E_X \times E_Y$. Then we can re-write (10.24) as

(10.25) $$\mathcal{L}(\theta) = \log P(D \mid \theta) = \log \int_{E_X} p(X, D \mid \theta)\, d\mathbf{x},$$

that is, $P(D \mid \theta)$ is computed by marginalization w.r.t. X. Now let $Q(X)$ be *any* pdf over the hidden variables. Then we can obtain a lower bound on $\mathcal{L}(\theta)$ by

(10.26)
$$\begin{aligned} \log \int p(X, D \mid \theta)\, d\mathbf{x} \;&= \\ &= \log \int Q(X) \frac{p(X, D \mid \theta)}{Q(X)}\, d\mathbf{x} \\ &\geq \int Q(X) \log \frac{p(X, D \mid \theta)}{Q(X)}\, d\mathbf{x} \end{aligned}$$

(10.27)
$$\begin{aligned} &= \int Q(X) \log p(X, D \mid \theta)\, d\mathbf{x} - \int Q(X) \log Q(X)\, d\mathbf{x} \\ &= \mathcal{F}(Q, \theta), \end{aligned}$$

where the inequality is an instance of the *Jensen* inequality for expectations of random variables:

(10.28) $E[q \circ X] \leq q\,(E[X])$ for any concave function q.

Note: the log is a concave function! – and the expectation we use is $E[Z(X)] = \int Z(X)\, Q(X)\, d\mathbf{x}$.

Aside. There is an interesting connection between (10.27) and statistical physics, which I want to mention at this point for the ones who want to dig below the surface (not needed in the remainder of these LNs). If we define the possible joint values (**x**, **d**) of (X, D) as *microstates* of a thermodynamical system, then the *energy* of such a microstate is $-\log p(\mathbf{x}, \mathbf{d} \mid \theta)$, and the expectation under Q of the energy of microstates is $-\int Q(X) \log p(X, D \mid \theta)\, d\mathbf{x}$, i.e. the negative of the first term in (10.27). The second term in (10.27), $-\int Q(X) \log Q(X)\, d\mathbf{x}$, is the *entropy* of Q.

EM algorithms maximize $\mathcal{F}(Q, \theta)$, by alternatingly maximizing $\mathcal{F}(Q, \theta)$ w.r.t. Q and θ, starting from an initial good parameter guess θ_0. That is, the following two operations are carried out in a seesaw fashion:

(10.29) **E**(xpectation) step: $Q_{k+1} = \arg\max_{Q} \mathcal{F}(Q, \theta_k)$

(10.30) **M**(aximization) step: $\theta_{k+1} = \arg\max_{\theta} \mathcal{F}(Q_{k+1}, \theta)$

The maximum in the E-step is obtained when Q is the conditional distribution of X:

(10.31) $Q_{k+1} = p(X \mid D, \theta_k)$,

because then it holds that $\mathcal{F}(Q_{k+1}, \theta_k) = \mathcal{L}(\theta_k)$:

(10.32) $\mathcal{F}(p(X \mid D, \theta_k), \theta_k) =$

$$= \int p(X \mid D, \theta_k) \log p(X, D \mid \theta_k)\, d\mathbf{x} - \int p(X \mid D, \theta_k) \log p(X \mid D, \theta_k)\, d\mathbf{x}$$

$$= \int p(X \mid D, \theta_k) \log \frac{p(X, D \mid \theta_k)}{p(X \mid D, \theta_k)}\, d\mathbf{x} = \int p(X \mid D, \theta_k) \log p(D \mid \theta_k)\, d\mathbf{x}$$

$$= \int \frac{p(X, D \mid \theta_k)}{p(D, \theta_k)} \log p(D \mid \theta_k)\, d\mathbf{x} = \frac{\log p(D \mid \theta_k)}{p(D, \theta_k)} \int p(X, D \mid \theta_k)\, d\mathbf{x}$$

$$= \log p(D \mid \theta_k) = \mathcal{L}(\theta_k).$$

The maximum in the M-step is obtained if the first term in (10.27) is maximized, because the second term does not depend on θ_k:

(10.33)
$$\theta_{k+1} = \arg\max_{\theta} \int Q_{k+1}(X) \log p(X, D \mid \theta)\, d\mathbf{x}$$
$$= \arg\max_{\theta} \int p(X \mid D, \theta_k) \log p(X, D \mid \theta)\, d\mathbf{x}$$

How the M-step is concretely computed depends on the particular kind of model. Because we have $\mathcal{F} = \mathcal{L}(\theta_n)$ before each M-step, and the E-step does not change θ_n, and $\mathcal{F}$ cannot decrease in an EM-double-step, the sequence $\mathcal{L}(\theta_0)$, $\mathcal{L}(\theta_1)$, $\mathcal{L}(\theta_2)$ ... monotonously grows toward a supremum. The computation is stopped when $\mathcal{L}(\theta_n) = \mathcal{L}(\theta_{n+1})$, or more realistically, when a predefined number of iterations is reached or when the growth rate falls below a predetermined threshold. The last parameter set θ that was computed is taken as the outcome of the EM algorithm.

It must be emphasized that EM algorithms steer toward a *local* maximum of the likelihood. If started from another θ_0, another final parameter set may be found. Here is a summary of the EM principle:

> **E-step:** estimate the distribution $p(X \mid D, \theta_n)$ of the hidden variables, given a preliminary model θ_n and data D.

M-step: use the (preliminary, approximate) knowledge of the distribution $p(X \mid D, \theta_n)$ of the hidden variables to obtain a maximum likelihood estimate of θ_{n+1}.

10.5 The EM algorithm for HMMs, "Baum-Welch" algorithm

We proceed to give a concrete instantiation of the EM principle for HMM learning, the "Baum-Welch" algorithm. There are other EM variants for HMM estimation, but the Baum-Welch algorithm seems to be the most widely used.

Let $D = a_{j0} \ldots a_{jN-1}$ be an observed sequence. Furthermore, assume that the number m of hidden states is known. Note: in reality, m is rarely known. Coming up with a good guess for m is not trivial, and typically boils down to trying out various choices of m, comparing the quality of the resulting models (after EM estimation of parameters) with cross-validation schemes – expensive!

The first thing to do is to "guess" a reasonable starter set θ_0 of transition, emission, and start state probabilities. A typical choice (in the absence of prior information about the model) is to use the uniform distribution with some noise on it.

Now we treat the E-step. That is, we have some preliminary estimated set θ_n of HMM parameters. From the knowledge of D and θ_n we have to derive the joint distribution $p(X_0, \ldots, X_{N-1} \mid D, \theta_n)$. Actually for the later use in the M-step we will not need the full information of $p(X_0, \ldots, X_{N-1} \mid D, \theta_n)$ but can make do with the distributions $p(X_l \mid D, \theta_n)$ and $p(X_l, X_{l+1} \mid D, \theta_n)$. Working our way towards these distributions we first note that from the forward and backward algorithms we can get the following probabilities:

(10.34) $$f_k^{(n+1)}(l) = P(X_l = q_k, a_{j0} \ldots a_{jl} \mid \theta_n) \quad \text{and}$$

(10.35) $$b_k^{(n+1)}(l) = P(a_{jl+1} \ldots a_{jN-1} \mid X_l = q_k, \theta_n) \quad \text{and}$$

(10.36) $$P(D \mid \theta_n) = \sum_k f_k^{(n+1)}(N-1).$$

Using these we can compute $P(X_l = q_k, X_{l+1} = q_{k'} \mid D, \theta_n) =: \xi_{kk'}^{(n+1)}(l)$ by

(10.37) $$\xi_{kk'}^{(n+1)}(l) = \frac{f_k^{(n+1)}(l)\, p_{kk'}^{(n)}\, e_{k' j_{l+1}}^{(n)}\, b_{k'}^{(n+1)}(l+1)}{P(D \mid \theta_n)}$$

Note that $p_{kk'}^{(n)}, e_{k' j_{l+1}}^{(n)}$ are part of θ_n. Furthermore, from (10.17) we know how to compute

(10.38) $$\gamma_k^{(n+1)}(l) = P(X_l = q_k \mid a_{j0} \ldots a_{jN-1}).$$

With the $\xi_{kk'}^{(n+1)}(l)$ and $\gamma_k^{(n+1)}(l)$ we have extracted all that we need to know about the distribution of the hidden variables and proceed to the M-step. In that step we derive a new maximum likelihood estimate for θ_{n+1} from $\xi_{kk'}^{(n+1)}(l)$, $\gamma_k^{(n+1)}(l)$ and D. This happens through

the following self-explaining *re-estimation* formulas. For the m components of the starting distribution $\mathbf{w}_0$ we can take

(10.39) $$P(X_0 = q_k)^{(n+1)} = \gamma_k^{(n+1)}(0).$$

The transition probabilities are re-estimated by

(10.40) $$p_{kk'}^{(n+1)} = \frac{\text{expected nr. of transitions } k \to k'}{\text{expected nr. of transitions } k \to \text{any state}} = \frac{\sum_{l=0}^{N-2} \xi_{kk'}^{(n+1)}(l)}{\sum_{l=0}^{N-2} \gamma_k^{(n+1)}(l)}.$$

Finally, the emission probabilities are re-estimated through

(10.41) $$e_{kj}^{(n+1)} = \frac{\text{expected nr. of passes through } q_k \text{, where } a_j \text{ is emitted}}{\text{expected nr. of passes through } q_k} = \frac{\sum_{l=0,\ldots,N-1}^{N-2} \gamma_k^{(n+1)}(l)}{\sum_{l=0}^{N-1} \gamma_k^{(n+1)}(l)}.$$

This finishes the M-step.

10.6 How to compare models for stochastic processes

When estimating HMMs (or any other kind of stochastic sequence model, for that matter) from data, one needs a way to compare the *quality* of different models, for instance in cross-validation schemes. A standard way of doing so is to use the log likelihood of the model on the training data (caution! overfitting!) or on test data (better! in cross-validation schemes, test data are taken from withheld portions of the training data). Concretely, assume first that the *true* model is known. Call it θ_{true}. Let $D = a_{j_{l+1}} \ldots a_{j_{N-1}}$ be the available data for quality checking. Then, the model θ_{true} should enable you to compute the log probability of D as

(10.42) $$LL_{true} = \log P(D \mid \theta_{true})$$

In the case of HMMs, LL_{true} can be conveniently computed with the matrix version of the forward algorithm 10.16, as follows:

1. Initialize $LL_{true} = 0$.
2. for $l = 0$ to N-1 do
    a. compute $\mathbf{v}_l = T_{j_l} \mathbf{w}_l$ (notice that for $l = 0$, $\mathbf{w}_0$ is given as part of the model)
    b. update $LL_{true} = LL_{true} + \log \mathbf{1}_m^{\mathsf{T}} \mathbf{v}_l$
    c. if $l < N$-1, put $\mathbf{w}_{l+1} = \mathbf{v}_l / \mathbf{1}_m^{\mathsf{T}} \mathbf{v}_l$

3. return LL_{true}.

It's a good exercise to work out for yourself why this algorithm indeed computes LL_{true}.

Now, if you have another model θ_{check} besides the true one, you can likewise compute $LL_{check} = \log P(D \mid \theta_{check})$. It is a fact which we mention without proof that always

(10.43) $$LL_{check} \leq LL_{true} .$$

(Well, this is almost true... for short test sequences D, the check model θ_{check} may spuriously give higher LL_{check} than LL_{true} occasionally due to random fluctuations in D; when D grows bigger this will become increasingly improbable). This gives you a way to assess the quality of your model: the closer it comes to LL_{true}, the better it is – and it's optimal, namely equivalent to the true model, when equality is obtained.

Usually however one does not have access to the true model LL_{true}. Then, this procedure can still be used to *compare* two check models θ^1_{check} and θ^2_{check}: if $\theta^1_{check} \leq \theta^2_{check}$, the second model is the better one, and vice versa. You don't know how close you are to the optimal model, but at least you know whether your models get better or worse.

10.7 Miscellaneous Remarks on HMMs

Good news: The Baum-Welch algorithm for HMMs is numerically robust, easy to use, and the only choice for training HMMs. Here are some not so good news:

This algorithm can become computationally expensive. It needs time $O(N |S|^2)$ per iteration, and in the order of 100 iterations are typically needed. For large N (e.g. in speech processing, N easily reaches 100,000) and large $|S|$ (again, may reach half a million in speech processing) this induces minutes or even hours of training time, which render HMMs awkward for adaptive applications (e.g. re-training on the spot for new speakers). State-of-the art HMM-based learning algorithms don't use the raw Baum-Welch method however, but exploit numerous tricks to reduce computation time (and at the same time introduce Bayesian priors for better data exploitation, for instance by prescribing pairwise equality of many of the parameters to be learnt).

Only a local optimum is guaranteed by EM. To make things worse, for complex HMMs (having many states) the likelihood landscape is very rugged, and one may easily land in a globally very subobtimal local optimum. The literature claims that this is not a real problem, but in my own experience I have found that even 3-dimensional HMMs can easily (and repeatedly) get stuck in very bad "optima". To overcome this impasse, one can use extra search methods (e.g., simulated annealing or restarting from different initial guesses of θ_0) that however multiply the computation time.

It is not immediately clear from a training sample what an appropriate model size (= number of states) is. This may force one to add yet another level of computational superstructure, namely, searching for an optimal model size (negotiating for a good compromise in the bias-variance dilemma).

One partial remedy to computational complexity is to use the *Viterbi approximation* to the Baum-Welch algorithm. In this "cheap" version of EM for HMM estimation, the E-step simply computes the Viterbi state sequence.

In spite of these difficulties and challenges, HMM + Baum-Welch is essentially the only available option for learning complex stochastic systems of the "speech or other information-bearing symbol sequence" flavour. The art and technology of HMM design and learning is far advanced, and various public-domain implementations are available. The one that I use was developed in Matlab by Kevin Murphy from MIT and can be downloaded from http://www.cs.ubc.ca/~murphyk/Software/HMM/hmm.html, where also a good choice of tutorial links is provided (this HMM toolbox requires several other public-domain Matlab toolboxes to be installed, as indicated on that webpage. You can download all of them bundled together with Kevin Murphy's HMM toolbox in a single zip file from http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/HMM.zip.

We have assumed in our treatment of HMM + Baum-Welch that the training sample consists of a single (long) sequence. In many applications, the training material consists instead in many short or medium-long sequences (for instance for single-word recognition in speech processing or in biosequence modeling). The Baum-Welch algorithm, as presented above, can be immediately adapted to that kind of training material.

One important way to reduce training time and at the same time to insert prior information intot the HMM that is to be learnt, is to prescribe the structure of the transition graph of the underlying Markov chain. That is, one specifies the number of states and also which state transitions are forbidden (frozen at a zero transition probability). This is of special importance in speech processing and biosequence modelling. A typical hand-coded transition graph for a HMM that models the utterance of a particular target word might look like in Figure 10.4.

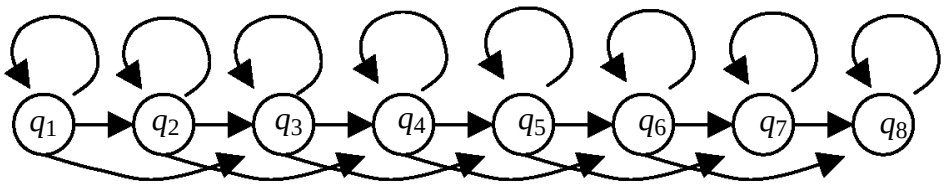


Figure 10.4 A state transition graph for a (finite) Markov chain, as could be obtained in modelling a short vocal utterance.

Many variants of HMMs have been explored together with variants of the EM learning algorithm. A good overview of basic variants is given in Rabiner's tutorial paper. They include:

HMMs that don't just emit from their state symbols from a finite alphabet but instead continuous-valued quantities according to a pdf associated with each state,
HMMs that emit observable events not from states, but from transitions,
HMMs that allow the hidden process to stay in a state for a stochastic time (and repeatedly emit observables from that state during that time) that is specified by additional model parameters.

Finally, I want to point out that HMMs are specific members of a far greater class of stochastic models, termed *Bayesian Networks* or – even more generally still – *Graphical Models.* Graphical models present stochastic models of complex pieces of reality, especially

in applications of diagnostic reasoning (medical expert systems, production line surveillance systems, fault monitoring systems), decision support systems, and image processing (where the random variables correspond to pixels). Such models often comprise thousands or even millions of random variables, whose joint distribution provides the most complete and profound possible description of a modelled piece of reality. However, joint distributions of very many variables cannot be in general learnt from data (curse of dimensionality!), and even if they could be, there is no general way even just to *represent* such joint distributions (how would you represent a complicated, empirical function over $\mathbb{R}^{1000}$?). Graphical models (and their subspecies, Bayesian Networks) exploit that in systems comprising thousands of random variables, most of them are conditionally pairwise independent, a circumstance that can be described by graphs whose nodes are the random variables – just as we did for HMMs in Figure 10.3. Combining advanced graph algorithms and techniques from nonlinear stochastic optimization, using fast computers it becomes in fact feasible to handle modelling tasks of enormous calibre. I give an introduction to graphical models is given in the companion lecture to this Machine Learning course, "Algorithmical and Statistical Modelling".