

Chapter 2

A Brief Introduction to Probabilistic Machine Learning and Its Relation to Neuroscience

Thomas P. Trappenberg

Abstract My aim in this chapter is to give a concise summary of what I consider the most important ideas in modern machine learning, and relate to one another different approaches, such as support vector machines and Bayesian networks, or reinforcement learning and temporal supervised learning. I begin with general comments on organizational mechanisms, then focus on unsupervised, supervised and reinforcement learning. I point out the links between these concepts and brain processes such as synaptic plasticity and models of the basal ganglia. Examples for each of the three main learning paradigms are also included to allow experimenting with these concepts.

1 Evolution, Development and Learning

Development and learning are two crucial ingredients for the success of natural organisms, and applying those concepts to artificial systems might hold the key to new breakthroughs in science and technology. This chapter is an introduction to machine learning that illustrates its links with neuroscientific findings. There has been much progress in this area, in particular by realizing the importance of representing uncertainties and the corresponding usefulness of a probabilistic framework.

1.1 Organizational Mechanisms

Before focusing on the main learning paradigms that dominate much of our recent thinking in machine learning, I would like to briefly outline some of my views on

Available at <http://projects.cs.dal.ca/hallab/MLreview2013>.

T. P. Trappenberg (✉)
Dalhousie University, Halifax, Canada
e-mail: tt@cs.dal.ca

the close relationships that exist among the organizational mechanisms discussed in this volume. It seems to me that at least three levels of these mechanisms contribute to the success of living organisms: evolutionary mechanisms, developmental mechanisms and learning mechanisms. *Evolutionary mechanisms* focus on the long-term search for suitable architectures. This search takes time, usually many generations, to establish small modifications that are beneficial for the survival of a species, and even longer to branch off new species that can exploit niches in the environment. Evolution is by essence adaptive, as it depends on the environment, the physical space, and other organisms. A good basic organization and good choices by an organism ultimately determine the survival of the individuals, hence the species in general.

While evolution works on the general architectural level of the population, a precise architecture has to be realized in individuals, too. This is where *development* comes into play. The genetic code is used to grow specific organisms from a master plan (the genome) and environmental conditions. Thus, this mechanism is also adaptive since the environment can influence the specific decoding of the master plan. For example, the shape and metabolism of the sockeye salmon can change drastically when environmental conditions allow migration from a freshwater environment to the ocean—whereas this fish remains small and adapted to fresh water if prevented from migrating, or if food sources are sufficient in the river. The ability to grow specific architectures in response to the environment gives organisms a considerable advantage, and these external stimuli seem to continually influence genetic expression.

Having grown a specific architecture, the resulting organisms can continue to respond to environmental conditions by *learning* about specific situations and how to take appropriate actions. Learning is another type of adaptation of a specific architecture that can take several forms. For example, it can be supervised by other individuals, such as parents teaching their offspring behavioural patterns that they find advantageous, or the organisms can learn from more general environmental feedback by receiving reinforcement signals such as food reward or the accuracy of anticipated outcomes. This chapter will focus for the most part on such learning mechanisms.

The three different adaptive frameworks outlined above are somewhat abstract at this level and it is important to be more precise about their meaning by showing specific implementations. However, this is also when distinctions between these mechanisms become somewhat blurred. For example, the emergence of receptive fields (e.g. in the visual cortex) during the critical postnatal period is definitely an important event at the developmental level, yet we will discuss such mechanisms as a special form of “learning” in this chapter. For the sake of this volume it might be useful to think about the learning processes described here as *fine-tuning* a system to specific environmental conditions, as they can be experienced by an individual during its lifetime. Other mechanisms discussed in this volume are aimed at developing better learning systems in the long term, or growing specific individuals in response to the environment.

While I will try to draw lines between development and learning, mainly to discuss approaches from different scientific camps, it is debatable that such distinctions could

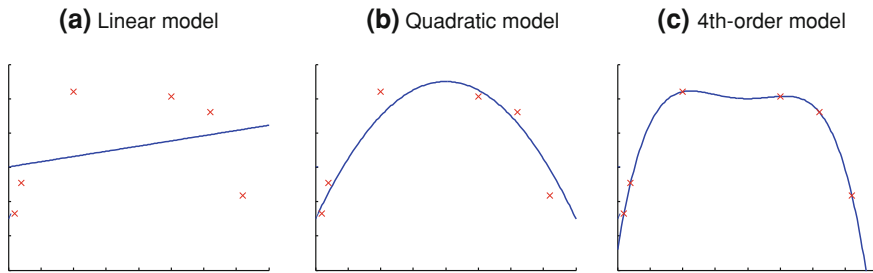


Fig. 1 Examples of underfitting (a) and overfitting (c)

even be made in the first place since, ultimately, all model implementations have to be reflected by some *morphological changes* in the system. Thus it is quite appropriate to bring together the modeling of different biological views into this volume.

1.2 Generalization

The general goal of the learning systems described here is to predict *associations*, or “labels”, for future unseen data. The examples given during the learning phase are used to choose the parameters of a model that represents certain hypotheses so that a specific realization of this model can later make good predictions. The quality of generalization from training data depends crucially on the complexity of the model that is hypothesized to describe the data, as well as the number of training samples.

This is illustrated by Fig. 1. Let us think about describing the six data points shown there with a linear model: the corresponding regression curve is shown in the left-hand graph, while the other two graphs show the regression of a quadratic model and a fourth-order polynomial. Certainly, the linear model seems too low-dimensional since the data points deviate systematically, with the points in the middle trending above the curve and the points at both ends laying below the curve. Such a systematic *bias* is a clear indication that the model complexity is too low. In contrast, the curve on the right fits the data perfectly. Indeed, we can always achieve a perfect fit for a finite number of training points if the number of free parameters (one for each order of the polynomial, in this example) approaches the number of training points. But this could be *overfitting* the data in the light of possible noise. To evaluate whether we are overfitting, we need additional validation examples. An indication of overfitting is when the *variance* of this validation error grows with an increasing model complexity.

What we just discussed, called the *bias-variance tradeoff* when choosing between different potential hypotheses, is summarized in the left-hand graph of Fig. 2. Many advances in machine learning have been made by addressing ways to choose good models. While the bias-variance tradeoff has been well appreciated in the machine learning community for some time now [1], many methods are still based on general learning machines that have a large number of parameters. For such machines it is

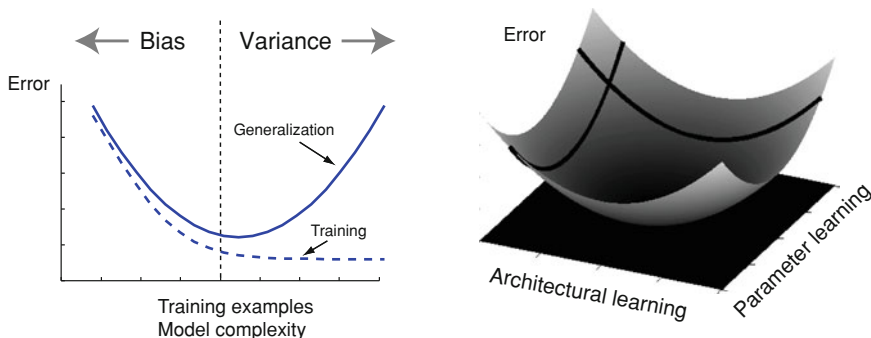


Fig. 2 Bias-variance tradeoff and explorative learning. While a training error can always decrease with increasing model complexity, minimizing the generalization error is what we are seeking. To find the smallest possible generalization error we need to search in hypothesis space and optimize in parameter space

now common to use *meta-learning* methods to address the bias-variance tradeoff, such as cross-validation where some of the training data is used to evaluate the generalization ability of the model.

We also need to consider if the model takes into account all the necessary factors that influence the outcome. How about including new features not previously considered such as a temporal domain? I believe that genetic and developmental mechanisms can address these issues by exploring a hypothesis space by ways of different model architectures. Of course, the exploration of a hypothesis space (developmental learning) must be accompanied by parameter optimization (behavioural learning) to find the best possible generalization performance. Several of the contributions in this volume represent good examples of this approach.

In summary, for the discussion in this volume it is useful to draw a distinction between two main processes:

- **Architectural exploration:** This process explores the hypothesis space in terms of *global structures*, such as what kind of features are relevant to build appropriate models and what model structures (parameterized functions) can be used.
- **Parameter optimization:** This process is about finding solutions (appropriate values of the parameters) within a specific architecture (a parameterized function).

Naturally, these processes are ultimately entwined and can be covered by common mechanisms. It remains that both aspects need to be included to find good predictive systems, as illustrated in the right-hand graph of Fig. 2.

1.3 Learning with Uncertainties

Machine learning has recently revolutionized computer applications such as autonomous car driving or information searching. Two major ingredients have contributed to this recent success. The first was building into the system the ability to *adapt*

to *unforeseen events*. In other words, we must build “machines that learn”, since the traditional method of encoding appropriate responses for all future situations is impossible. Like humans, machines should not be static entities that can only blindly follow orders, which might be outdated by the time real situations are encountered. Although learning machines have been studied for at least half a century, often inspired by human capabilities, the field has matured considerably in recent years through more rigorous formulations of the systems and the realization of the importance of predicting previously unseen events rather than only memorizing former events. Machine learning is now a well established discipline within artificial intelligence.

The second ingredient for the recent breakthroughs was the acknowledgment that there were *uncertainties* in the world. Thus, rather than only following the most likely explanation for a given situation, keeping an open mind and considering other possible explanations has proven to be essential in systems that have to work in a real-world environment, in contrast to a controlled lab environment. The language of describing uncertainty, that of probability theory, has proven to be elegant and tremendously simplify arguing in such worlds. This chapter is dedicated to an introduction to the *probabilistic formulation* of machine learning.

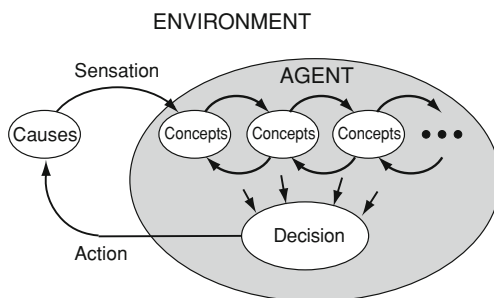
In the following sections I outline a contemporary view of learning theories that includes unsupervised, supervised and reinforcement learning. I begin with unsupervised learning since it is likely less known and relates more closely to certain developmental aspects of an organism. Then, I briefly review supervised learning in a probabilistic framework. Finally, I present reinforcement learning as an important generalization of supervised learning. In addition, I discuss some relations of these learning theories with biological analogies. This includes the relations of unsupervised learning with the development of filters in early sensory cortical areas, synaptic plasticity as the physical basis of learning, and research that relates the basal ganglia to reinforcement learning theories.

I thought important to include supervised, unsupervised and reinforcement learning in a form that would correspond to an advanced treatment of these topics in a course on machine learning. While there are now many good publications that focus on specific approaches in machine learning (such as kernel methods or Bayesian models), my aim is to link together and contrast several popular learning approaches. Most discussions of machine learning start with supervised learning, but I opted here for an initial discussion on unsupervised learning instead, as it logically precedes supervised learning and is generally less known.

1.4 Predictive Learning

Since my main research focus is neuroscience, I would like to first clarify how machine learning relates to this field. Machine learning can actually help neuroscience in many ways, one of which certainly concerns *data analysis*, as learning methods constitute the foundations of most advanced data mining techniques. Another application area, and the one examined here, is to understand the main

Fig. 3 The “anticipating brain” contains a hierarchical generative model of concepts and a decision system that guides behavior with the help of an anticipatory world model



problems and solutions in machine learning that can guide our understanding of biological learning systems. That is, we can ask what essential methods for solving learning problems are available, in a way somewhat reminiscent of Marr and Poggio’s view of a computational, and possibly algorithmic, level of neuroscience. Similarly, many models discussed here can be construed as models of the brain on a more abstract level. Within computational neuroscience, there are also models that represent more mechanistic levels with specific representations and physical implementations. The implementation of learning via synaptic plasticity, and a more system-level model of the basal ganglia are a few of the examples mentioned later in this chapter.

If pressed to summarize what the brain does, I would say that it is an organ that represents a sophisticated decision system based on an adaptive world model. The goal of learning as it is described here is anticipation, or *prediction*. A predictive model can be used by an organism to make appropriate decisions to reach some goals. I believe that increasingly complex nervous systems evolved to make increasingly sophisticated predictions that could give them survival and evolutionary advantages.

A possible architecture of a predictive learning system resembling my high-level view of the brain is outlined in Fig. 3. An agent must interact with the environment from which it learns and receives a reward. This interaction has two sides: sensation and action. The state of the environment is conveyed by sensations that are caused by specific situations in the environment. A comprehension of these sensations requires hierarchical processing in deep-learning systems. The hierarchical processes are bidirectional so that the same structure can be used to generate expectations that should ultimately yield appropriate actions. These actions have to be guided by a decision system that itself needs to learn from the environment. This chapter reviews the principal components of such a learning system.

2 Unsupervised Learning

2.1 Representations

An important requirement for a natural or artificial agent is to decide on an appropriate course of action given specific circumstances, mainly the encountered environment.

We can treat the environmental circumstances as cues given to the agent. These cues are communicated by sensors that specify the values of certain features. Let us represent these *feature values* as a vector \mathbf{x} . The goal of the agent is then to calculate an appropriate response

$$y = f(\mathbf{x}). \quad (1)$$

In this review we use a probabilistic framework so that we can address uncertainties, or different possible responses. The corresponding statement of the deterministic function approximation of Eq. (1) is then to find a probability density function

$$p(y|\mathbf{x}). \quad (2)$$

A common example is object recognition where the feature values might be RGB values of pixels in a digital image and the desired response might be the identity of a person in this image. A learning machine for such a task is a model that is presented with specific examples of feature vectors \mathbf{x} and their corresponding desired *labels* y . Learning under these circumstances mainly consists of adjusting the model's parameters based on the given examples. A trained machine should be able to *generalize* by predicting the appropriate labels of previously unseen feature vectors, where the “appropriateness” usually depends on the task. Since this type of learning is based on specific training examples with known labels, it is called *supervised*. We discuss specific algorithms of supervised learning and corresponding models in the next section. We start here with unsupervised learning since it is a more fundamental task that precedes supervised learning.

As stated above, the aim of learning is to find a mapping function $y = f(\mathbf{x})$ or probability density function $p(y|\mathbf{x})$. An important insight that we explore in this section is that finding such relations is much easier if the representation of the feature vector is chosen carefully [1]. For example, it is very challenging to use raw pixel values to infer the content of a digital photo such as the recognition of a face. In contrast, if we possess useful descriptions of faces, such as the distance between the eyes or other landmarks, the hair colour, nose length, and so on, it becomes much easier to classify photographs into specific target faces. Finding a useful representation of a problem is key to a successful application. When we use learning techniques for this task we talk about *representational learning*. Representational learning mostly exploits statistical characteristics of the environment without the need for labeled training examples. This is therefore an important area of *unsupervised learning*.

Representational learning itself can be viewed as a mapping problem, for example the mapping from raw pixel values to more direct features of a face. This is illustrated in Fig. 4: the raw input feature vector \mathbf{x} is represented by a layer of nodes at the bottom, which we will call the *input layer*, while the feature vector \mathbf{h} supporting higher order representations is represented by nodes in the upper layer of this network, which we will call the *representational layer* or *hidden layer*. The connections between the nodes represent the desired transformation between input layer and hidden layer. In line with our probabilistic framework, each node represents a random

Fig. 4 A restricted Boltzmann machine is a probabilistic two-layer network with bidirectional symmetric connections between the input layer and the representational (hidden) layer

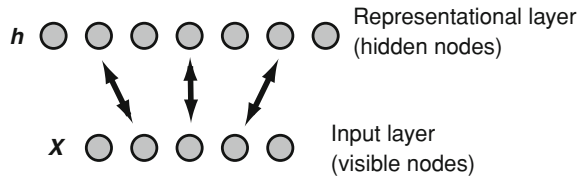
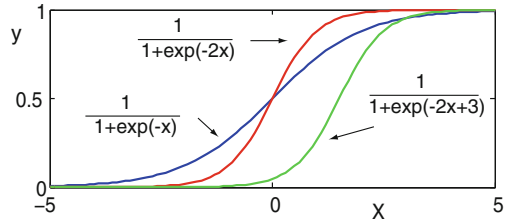


Fig. 5 Logistic function with different slopes and offsets



variable. The main idea behind the principle that we will employ to find “useful” representations is that these representations should be useful inasmuch as they can help reconstructing the input.

Before we discuss different variants of hidden representations, let us make the functions of the model more concrete. Specifically, we consider binary random variables for illustration purposes. Given the values of the input nodes (indexed by j), we choose to calculate the value of the hidden nodes (indexed by i), or more precisely their probability of having a certain value, via the logistic function shown in Fig. 5:

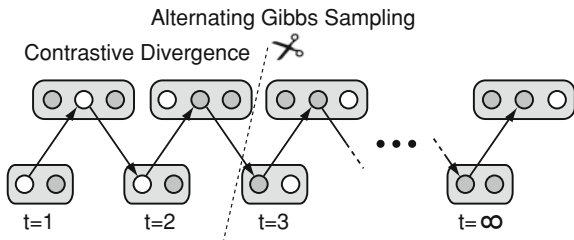
$$p(h_i = 1|\mathbf{x}) = \frac{1}{1 + e^{-\frac{1}{T}(\mathbf{w}_i \mathbf{x} + b_i^h)}}, \quad (3)$$

where T is a “temperature” parameter controlling the steepness of the curve, \mathbf{w} are the weight values of the connections between the input and hidden layers, and b_i^h is the offset of the logistic function, also called the *bias* of the hidden node. In this model, which is called a “restricted Boltzmann machine” (RBM) [2], there are no connections among the hidden nodes, so these nodes represent random variables that are conditionally independent when the inputs are observed. In other words, the joint density function with fixed inputs factorizes as follows:

$$p(\mathbf{h}|\mathbf{x}) = \prod_i p(h_i|\mathbf{x}). \quad (4)$$

The connections here are bidirectional and symmetric, meaning that $w_{ij} = w_{ji}$, therefore this kind of model also represents an “undirected Bayesian network”, which is a special case of the Bayesian networks that will be discussed later. Thus the state of the input nodes can be generated by hidden activities according to

Fig. 6 Alternating Gibbs sampling and the approximation of contrastive divergence



$$p(x_j = 1 | \mathbf{h}) = \frac{1}{1 + e^{-\frac{1}{T} \sum_i w_{ij} h_i + b_j^v}}$$

$$p(\mathbf{x} | \mathbf{h}) = \prod_j \frac{1}{1 + e^{-\frac{1}{T} \sum_i w_{ij} h_i + b_j^v}} \quad (5)$$

where b_j^v are the biases for each visible (input) node.

The remaining question is: how can we choose the parameters, specifically the weights and biases of the model? Since our aim is to reconstruct the observed world, we can formulate the answer in a probabilistic framework by minimizing the distance between the world's distribution (the density function of the visible nodes when set to unlabeled examples from the environment) and the generated model of the world when sampled from hidden activities. The difference between distributions is often measured by the Kullback-Leibler divergence, denoted by D_{KL} , and minimizing this objective function with a gradient method leads to a Hebbian-type learning rule:

$$\Delta w_{ij} = \eta \frac{\partial D_{\text{KL}}}{\partial w_{ij}} = \eta \frac{1}{2T} (\langle h_i v_j \rangle_{\text{clamped}} - \langle h_i v_j \rangle_{\text{free}}). \quad (6)$$

The angular brackets $\langle . \rangle$ denote sample averages, either in the clamped mode where the inputs are fixed or in the free running mode where the input nodes' activities are determined by the hidden nodes. Unfortunately, in practice this learning rule suffers from the long time it takes to produce an unbiased average from sequentially sampled time series. However, it turns out that learning still works for a few steps in the Gibbs sampling as illustrated in Fig. 6. This learning rule, which has finally made Boltzmann machines applicable, is called *contrastive divergence* [3] (see also [4]).

An example of a basic restricted Boltzmann machine is given in Table 1. This RBM has $n_h = 100$ hidden nodes and is trained for $n_{\text{epochs}} = 150$ epochs, where one epoch consists of presenting all images once. The network is trained with contrastive divergence in the next block of code. The training curve, which shows the average error of recall of patterns, is shown on the left in Fig. 7. After training, 20% of the bits of the training patterns are flipped and presented as input to the network, then the program plots the patterns after repeated reconstructions as displayed on the right side of Fig. 7. Only the first 5 letters are shown here, but this number can be increased to inspect more letters.

Table 1 Basic restricted Boltzmann machine for learning letter patterns

```

clear; nh = 100; nepochs = 150; lrate = 0.01;
%load data from text file and rearrange into matrix
load pattern1.txt;
letters = permute(reshape(pattern1, [12 26 13]), [1 3 2]);

%train rbm for nepochs presentations of the 26 letters
input = reshape(letters, [12*13 26])
vb = zeros(12*13, 1); hb = zeros(nh, 1); w = .1*randn(nh, 12*13);

figure; hold on;
xlabel 'epoch'; ylabel 'error'; xlim([0 nepochs]);
for epoch = 1:nepochs;
    err = 0;
    for i = 1:26
        %sample hidden units given input, then reconstruct
        v = input(:, i);
        h = 1./(1 + exp(-(w*v + hb))); %sigmoidal activation
        hs = h > rand(nh, 1); %probabilistic sampling
        vr = 1./(1 + exp(-(w'*hs + vb))); %input reconstruction
        hr = 1./(1 + exp(-(w*vr + hb))); %hidden reconstruction

        %contrastive divergence rule: dw = h*v - hr*vr
        dw = lrate*(h*v'-hr*vr'); w = w + dw;
        dvb = lrate*(v - vr); vb = vb + dvb;
        dhb = lrate*(h - hr); hb = hb + dhb;
        err = err + sum((v-vr).^2); %reconstruction error
    end
    plot(epoch, err/(12*13*26), '.'); drawnow; %figure output
end

%plot reconstructions of noisy letters
r = randomFlipMatrix(round(.2*12*13)); % (20% of bits flipped)
noisy_letters = abs(letters - reshape(r, [12 13 26]));
recon = reshape(noisy_letters, 12*13, 26); %put data in matrix
recon = recon(:, 1:5); %plot only first 10
figure; set(gcf, 'Position', get(0, 'screensize'));

for i = 0:3
    for j = 1:5
        subplot(3 + 1, 5, i*5 + j);
        imagesc(reshape(recon(:, j), [12 13])); %plot
        colormap gray; axis off; axis image;

        h = 1./(1 + exp(-(w*recon(:, j) + hb))); %compute hidden
        hs = h > rand(nh, 1); %sample hidden
        recon(:, j) = 1./(1 + exp(-(w'*hs + vb))); %compute visible
        recon(:, j) = recon(:, j) > rand(12*13, 1); %sample visible
    end
end

function r = randomFlipMatrix(n);
%return matrix with components 1 at n random positions
r = zeros(156, 26);
for i = 1:26
    x = randperm(156);
    r(x(1:n), i) = 1;
end

```

This network is used to learn digitized letters of the alphabet that are provided in the file pattern1.txt at <http://www.cs.dal.ca/~repository/MLintro2012> together with the other programs of this chapter

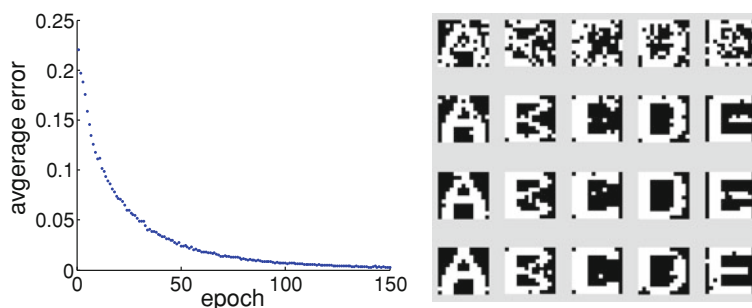


Fig. 7 Output of the example program for a restricted Boltzmann machine. *Left* learning curve showing the evolution of the average reconstruction error. *Right* reconstructions of noisy patterns after training

2.2 Sparse and Topographic Representations

In the previous section we reviewed a basic probabilistic network that implements representational learning based on the reconstruction of inputs. There are many other unsupervised algorithms that can achieve representational learning, such as non-probabilistic recurrent networks (for example, see Rebecchi et al. in this volume). Also, many other representational learning algorithms originate from signal processing, such as Fourier transform, wavelet analysis, or independent component analysis (ICA). Indeed, most advanced signal processing methods include steps to re-represent or decompose a signal into basis functions. For example, the Fourier transform decomposes a signal into sine waves with different amplitudes and phases. The original signal can then be reconstructed from the sum of individual sine waves weighted by their amplitude parameters. An example is shown in Fig. 8. The signal in the upper left is made out of three sine waves as revealed by the power spectrum on the right, which plots the square of the corresponding coefficients.

The Fourier transform has been very useful in describing periodic signals, but one problem with this representation is that an infinite number of basis functions are needed to represent a signal that is localized in time. An example of a square signal localized in time is shown in the lower left panel of Fig. 8 together with its power spectrum on the right. In the case of the time-localized signal, the power spectrum shows that a continuous interval of frequencies is necessary to accurately represent the original signal. Thus, a better choice for applications with localized features would be basis functions that are localized in time. Examples are wavelet transforms [5] or the Huang-Hilbert transform [6]. The usefulness of a specific transformation depends of course on the nature of the signals. Periodic signals with few frequency components, such as the rhythm of the heart or yearly fluctuations of natural events, are well represented by Fourier transforms, while signals with localized features, such as objects in a visual scene, are often well represented with wavelets. The main reason for calling a representation “useful” is that the original signal can be represented with

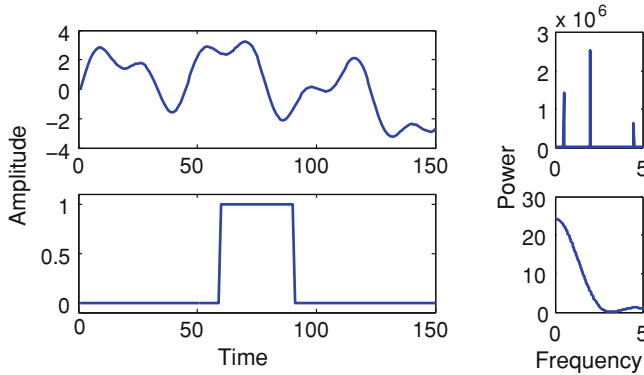


Fig. 8 Decomposition of signals into sine waves. The example signals are shown on the *left side*, and the corresponding description of the power spectrum on the *right*. The power spectrum shows the square of the amplitude for each contributing sine wave with specified frequency

only a small number of basis functions—in other words, when only a small number of coefficients have significantly large values. Therefore, even if the dictionary is large, each example of a signal from the specific environment can be represented with a small number of components. Such representations are called *sparse*.

The importance of sparse representations in the visual system has long been pointed out by Horace Barlow [7], and one of the best and probably first examples that demonstrate such mechanisms was give by his student Peter Földiák [8] (see also [9]). Another very influential article by Olshausen and Field [10] demonstrated that sparseness constraints are essential in learning basis functions that resemble receptive fields in the primary visual cortex, and similar concepts should also hold for higher-order representations in deep-belief networks [11]. It is now argued that such unsupervised mechanisms resemble receptive fields of simple cells.

The major question is then how to find good (sparse) representations for specific environments. One solution is to learn representations by unsupervised training as demonstrated above with the example of a Boltzmann machine. To learn sparse representations we now add additional constraints that force the learning of specific basis functions. In order to do this we can keep track of the mean activation of the hidden nodes by setting

$$q_i(t) = (1 - \lambda)q_i(t - 1) + \lambda h_i(t), \quad (7)$$

where parameter λ determines the averaging window. We then add to the learning rule the constraint of minimizing the difference between the *desired sparseness* ρ and the *actual sparseness* q , expressed by

$$\Delta w_{ij} \propto v_j(h_i + \rho - q_i) - v_j^r h_i^r. \quad (8)$$

This works well in practice and has the extra advantage of preventing *dead nodes* [4].

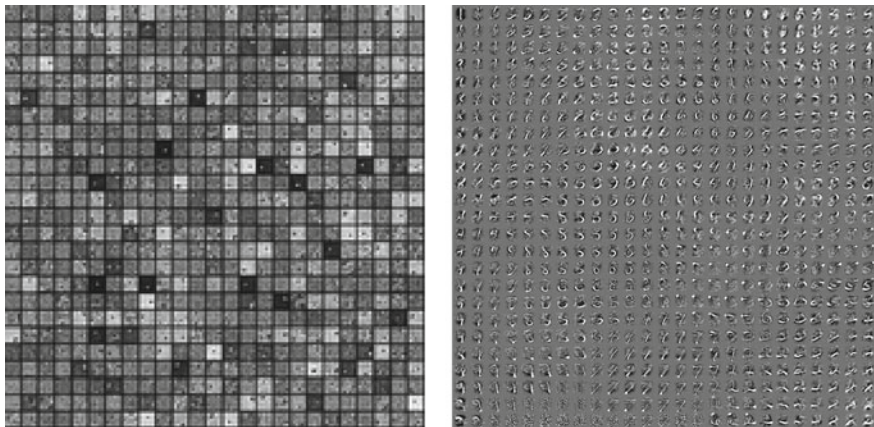


Fig. 9 Examples of learned receptive fields of a RBM without (*left*) and with (*right*) sparse and topographic constraints

In addition to the typical form of receptive fields, many brain areas show some *topographic organization* in that neurons with adjacent features of receptive fields are located in adjacent tissues. An example of unsupervised topographic representations are “self-organizing projections” [12, 13] or “self-organizing maps” (SOMs) [14]. Topographic self-organization can be triggered by lateral interactions with local facilitation and distant competition, as can be implemented with pairwise local excitation and distant inhibition between neurons. Such interactions also promote sparse representations. Along these lines, my student Paul Hollensen together with my collaborator Pitoyo Hartono and myself proposed to include lateral interactions within the hidden layer [15] as follows:

$$p(\hat{h}_i|\mathbf{v}) = \sum_j \mathcal{N}_{ij} p(h_j|\mathbf{v}), \quad (9)$$

where i, j both represent hidden units here, and \mathcal{N}_{ij} is a kernel such as a shifted Gaussian or a Mexican-hat function centered on hidden node i . For binary hidden units the natural measure of the difference in distributions is the cross entropy, for which the derivative with respect to the weights is simply $(\hat{h}_i - h_i) \cdot \mathbf{v}$. Combining this with the contrastive divergence update yields

$$\Delta w_{ij} \propto v_j h_i - v_j^r h_i^r + v_j (\hat{h}_i - h_i) = v_j \hat{h}_i - v_j^r h_i^r. \quad (10)$$

Figure 9 presents examples of receptive fields learned with (right) and without (left) sparse topographic learning.

While purely bottom-up driven SOMs have dominated the thinking in this field, it is also important to consider models with top-down guidance of self-organized feature representations. An excellent example is the Adaptive Resonance Theory

(ART) of Stephen Grossberg [13, 16], which is most relevant in a biological context and even addresses the stability-plasticity dilemma. Further aspects of top-down control in SOMs are discussed in [17].

2.3 Hierarchical Representations and Deep Learning

Before leaving our discussion about representational learning, I would like to mention at least briefly the importance of hierarchical representations. So far we have only considered one layer of internal representations that we called the hidden layer. However, it is widely believed that representations that allow abstractions at different levels are essential to enable the cognitive abilities displayed by humans.

An obvious example consists of stacking Boltzman machines so that the hidden layer of one Boltzman machine becomes the input layer to the next Boltzman machine. This already has the advantage that more complex filters can be built from filters learned in previous levels. For example, if a first layer represents edges in a visual scene, a higher level could represent corners or more elaborate combinations of edges.

However, just obtaining more elaborate filters might not be the only advantage derived from hierarchical representations. In order to enable more advanced cognitive abilities, such as exploiting more general concepts or making higher-level plans, we need to enable more abstract representations of concepts. Such *deep learning* algorithms are the subject of much recent research in machine learning, and the chapter by Joshua Bengio is an excellent discussion of some of the challenges in this area.

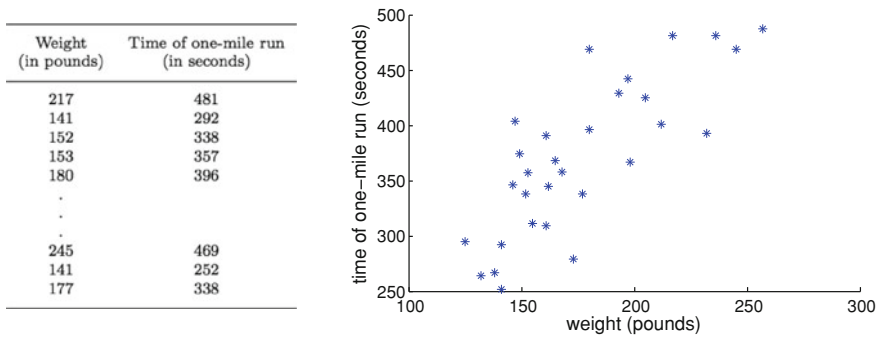
Deep learning structures also resemble better the situation of the brain as a learning machine. We have mentioned above that filters in the early sensory areas and higher levels of the cortex, such as neurons in the inferotemporal cortex [18], are known to respond to more complex patterns. But it is also known that the prefrontal cortex contributes to high-level cognition functions such as planning and other executive functions that are often based on abstract concepts.

3 Supervised Learning

3.1 Regression

Representational learning is about learning a mapping function that transforms a signal (input vector) into a new signal (hidden vector):

$$f_h: \mathbf{x} \rightarrow \mathbf{h} \quad (\text{given unlabeled examples and constraints}). \quad (11)$$

**Fig. 10** Health data

The unsupervised learning of this mapping function typically exploits statistical regularities in the signals, and therefore depends on the nature of the input signals. This learning process is also guided by principles such as good reconstruction abilities, sparseness and topography. Supervised learning, on the other hand, is about learning an unknown mapping function from labeled examples:

$$f_y: \mathbf{h} \rightarrow \mathbf{y} \quad (\text{given labeled examples}). \quad (12)$$

We have indicated in the formula above that supervised learning takes the hidden representation of examples, \mathbf{h} and maps them to a desired output vector \mathbf{y} . This assumes that representational learning is somewhat completed during a developmental learning phase, which is then followed by supervised learning with a teacher that supplies desired labels (output values) for given examples. It may be argued that in natural learning systems these learning phases are not as strictly separated as discussed here, but for the purpose of this tutorial it is useful to make a distinction between these two major learning components.

In our discussion of strictly supervised learning for this section, let us follow the common nomenclature in denoting input values by \mathbf{x} and output values by \mathbf{y} . In supervised learning we consider training data that consists of example inputs and corresponding labels, that is, pairs of values $(\mathbf{x}^{(e)}, \mathbf{y}^{(e)})$, where $e = 1, \dots, m$ indexes the m training examples. For instance, Fig. 10 presents a partial list and plot of the running records of 30 employees who were regular members of a company's health club [19]. Specifically, the data shows the relationship between the weight of these persons and their time in a one-mile run.

Looking at the plot seems to reveal a systematic relation between the weights and running times, with a trend for heavier individuals to be slower at running, although this is not true for everyone. Moreover, the trend appears linear. This hypothesis can be quantified as a parameterized function

$$h(x; \boldsymbol{\theta}) = \theta_0 + \theta_1 x. \quad (13)$$

This notation means that hypothesis h is a family of functions of the quantity x that includes all possible straight lines, where each line can have a different offset θ_0 (intercept with the y -axis) and slope θ_1 . We typically collect parameters in a *parameter vector* denoted by θ . We only considered a single input feature x above, but we can easily generalize this to higher-dimensional problems where more input *attributes* are given. For example, there might be the amount of exercising each week that might impact the results of running times. If we make the hypothesis that this additional variable has also a linear influence on the running time, independently from the other attribute that adds or reduces the time, we can express this new hypothesis with

$$h(\mathbf{x}; \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2. \quad (14)$$

A useful trick to enable a compact notation in higher dimension with n attributes is to introduce $x_0 = 1$. We can then write the linear equations as

$$h(\mathbf{x}; \theta) = \theta_0 x_0 + \dots + \theta_n x_n = \sum_j \theta_j x_j = \theta^T \mathbf{x}. \quad (15)$$

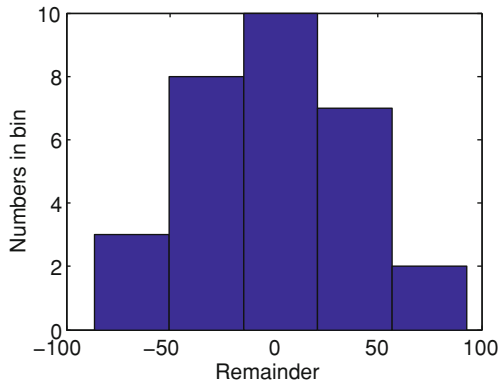
where vector θ^T is the transpose of vector θ .

At this point it would be common to fit the unknown parameters θ with methods such as a least mean squares (LMS) regression. However, I would like to frame this problem right away in a more modern probabilistic framework. The data already shows that the relations between the weight and the running time is not strictly linear, thus the main question is how we should interpret the differences. We could introduce a more complicated nonlinear hypothesis to obtain a better fit. However, this could lead to conclusions such as: increasing your weight from 180 to 200 pounds will make you run faster. While we might wish this conclusion were true, it is most certainly unwarranted. Thus, instead of making the hypothesis function more complex, we should consider other possible sources that influence this data. One is certainly that the ability to run does not only depend on the weight of a person but also on other physiological factors. However, this data does not include information about such other factors, and the best we can do (other than collecting more information) is to treat these deviations as *uncertainties*.

There are many possible sources of uncertainties such as *irreducible indeterminacy* or *epistemological limitations*. Irreducible indeterminacy might be called “true noise”, as it comes from system limitations such as time constraints on measurements, other sensors’ limitations, or simply laziness for collecting more information. For us, it is actually not important where these uncertainties originate; rather, we must only acknowledge the uncertain nature of the data. In this type of thinking, we treat sampled data from the outset as fundamentally stochastic, that is, sensory data can be different even in situations that we deem identical.

To model the uncertainties in this data, we look at the deviations from the mean. Figure 11 shows a histogram of the differences between the actual data and the hypothesized regression line. This histogram looks a bit like one sampled from

Fig. 11 Histogram of the differences between the data points and the fitted hypothesis, $(y - \theta_0 - \theta_1 x)$



Gaussian data, which is a frequent finding in many situations though not necessarily the only one. In any case, let us just make this additional assumption that there is noise in the data. With this conjecture, we should revise our hypothesis in a probabilistic framework. More precisely, we acknowledge that we can only give a probability of finding certain values. Specifically, we assume here that the data follows a certain trend $h(\mathbf{x}; \boldsymbol{\theta})$ with an *additive noise* denoted η ,

$$p(y|\mathbf{x}; \boldsymbol{\theta}) = h(\mathbf{x}; \boldsymbol{\theta}) + \eta, \quad (16)$$

where the random variable η comes from a Gaussian (normal) distribution \mathcal{N} in the above example, i.e.,

$$p(\eta) = \mathcal{N}(\mu, \sigma). \quad (17)$$

We can then also write the probabilistic hypothesis in the above example as a Gaussian model with a mean that depends on the variable \mathbf{x} :

$$\begin{aligned} p(y|\mathbf{x}; \boldsymbol{\theta}) &= \mathcal{N}(\mu = h(\mathbf{x}; \boldsymbol{\theta}), \sigma) \\ &= \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y - \boldsymbol{\theta}^T \mathbf{x})^2}{2\sigma^2}\right). \end{aligned} \quad (18)$$

This function defines the probability of an y value, given an input \mathbf{x} and parameters $\boldsymbol{\theta}$. We have here treated the variance σ^2 as given, although it, too, could be part of the model parameters that need to be estimated. Specifying a model with a density function is an important step in modern modeling and machine learning.

We have thus far made a parameterized hypothesis underlying the nature of the data. We now need to estimate values for the parameters to make real predictions. Therefore, let us consider again the examples of input-output pairs, i.e. our training set $\{(\mathbf{x}^{(e)}, y^{(e)}); e = 1, \dots, m\}$ (in 1D). The important principle that we will follow now is to choose the parameter $\boldsymbol{\theta}$ so that the examples we have are most likely covered by

the model. This is called *maximum likelihood estimation*. To formalize this principle, we need to think about how to combine probabilities for several observations. If the observations are independent, then the joint probability of several observations is the product of the individual probabilities:

$$p(Y_1, Y_2, \dots, Y_m | X_1, X_2, \dots, X_m; \theta) = \prod_{e=1}^m p(Y_e | X_e; \theta). \quad (19)$$

Note that the Y_i 's are still random variables in the above formula. We now use our training examples as specific observations (point estimates) for each of these random variables, and introduce the *likelihood function*

$$L(\theta) = \prod_{e=1}^m \hat{p}(\theta; y^{(e)}, x^{(e)}). \quad (20)$$

Here, on the right-hand side, \hat{p} is not a density function but a regular function of parameter θ (with the same functional form as our parameterized hypothesis p) for the given values $y^{(e)}$ and $x^{(e)}$. Instead of evaluating this large product, however, it is common to use the logarithm of the likelihood function, so that we can use the sum over the training examples:

$$l(\theta) = \log L(\theta) = \sum_{e=1}^m \log(\hat{p}(\theta; y^{(e)}, x^{(e)})). \quad (21)$$

Since the log function is strictly monotonically increasing, the maximum of L is also the maximum of l . The maximum (log-)likelihood estimate (MLE) of the parameter can thus be calculated from the examples by

$$\theta^{\text{MLE}} = \arg \max_{\theta} l(\theta). \quad (22)$$

In some cases, we can calculate this analytically or we can use a search algorithm to find an approximation.

Let us now apply this strategy to the regression of a linear function with Gaussian noise as discussed above. The log-likelihood function for this example is given by

$$\begin{aligned} \hat{p}(\theta; y^{(e)}, x^{(e)}) &= \frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(y^{(e)} - \theta x^{(e)})^2}{2\sigma^2}\right) \\ \Rightarrow l(\theta) &= -\frac{m}{2} \log 2\pi\sigma - \sum_{e=1}^m \frac{(y^{(e)} - \theta x^{(e)})^2}{2\sigma^2}. \end{aligned} \quad (23)$$

Since the first term on the right-hand side of Eq. (23) is independent of θ , and since we considered here a model with a given variance σ^2 for the data, maximizing the log-likelihood function is equivalent to minimizing a quadratic error term

$$E = \frac{1}{2}(y - h(\mathbf{x}; \boldsymbol{\theta}))^2 \Leftrightarrow p(y|\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(y - h(\mathbf{x}; \boldsymbol{\theta}))^2}{2}\right) \quad (24)$$

(switching the notation back to the multidimensional case). Thus, the MLE of a Gaussian dataset corresponds to minimizing a quadratic cost function, as it was commonly used in LMS regression. LMS regression is well motivated for Gaussian data, but our derivation also shows that data with non-Gaussian noise should be fitted with different cost functions. For example, a *polynomial error function* corresponds more generally to a density model of the form

$$E = \frac{1}{p} \|y - h(\mathbf{x}; \boldsymbol{\theta})\|^p \Leftrightarrow p(y|\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{2\Gamma(1/p)} \exp(-\|y - h(\mathbf{x}; \boldsymbol{\theta})\|^p). \quad (25)$$

Later we will mention the ε -insensitive error function, where errors less than a constant ε do not contribute to the error measure:

$$E = \|y - h(\mathbf{x}; \boldsymbol{\theta})\|_\varepsilon \Leftrightarrow p(y|\mathbf{x}; \boldsymbol{\theta}) = \frac{p}{2(1 - \varepsilon)} \exp(-\|y - h(\mathbf{x}; \boldsymbol{\theta})\|_\varepsilon). \quad (26)$$

Since we already acknowledged that we expected noisy data, it is logical not to count some amount of deviation from the expectation as error. It also turns out that this last error function is often more robust than other error functions, especially for datasets that contain outliers.

3.2 Classification as a Logistic Regression

We have grounded supervised learning in probabilistic function regression and maximum likelihood estimation. An important special instance of supervised learning is *classification*, and the simplest case is binary classification which corresponds to data that has only two possible labels, such as $y \in \{0, 1\}$.

More formally, let us consider a random number that takes value 1 with probability ϕ and value 0 with probability $1 - \phi$. Such a random variable is called a *Bernoulli distribution*. Tossing a coin is a good example of a process that generates a Bernoulli random variable, and we can use maximum likelihood estimation to estimate the parameter ϕ from such trials. For example, if we consider m tosses of a coin, the log-likelihood of finding h heads ($y = 1$) and $m - h$ tails ($y = 0$) is

$$\begin{aligned} l(\phi) &= \log(\phi^h (1 - \phi)^{m-h}) \\ &= h \log(\phi) + (m - h) \log(1 - \phi). \end{aligned} \quad (27)$$

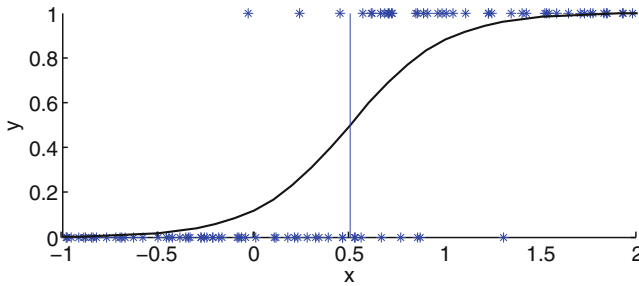


Fig. 12 Binary random numbers (stars) drawn from the density $p(y = 1) = 1/(1+\exp(-\theta_0-\theta_1 x))$ (solid line) with offset $\theta_0 = -2$ and slope $\theta_1 = 4$

To find the maximum of l with respect to ϕ , we set the derivative of l to zero:

$$\begin{aligned}\frac{dl}{d\phi} &= \frac{h}{\phi} - \frac{m-h}{1-\phi} = 0 \\ \Rightarrow \phi &= \frac{h}{m}.\end{aligned}\tag{28}$$

As you might have expected, the MLE of parameter ϕ is the fraction of heads in m trials.

Let us now discuss the case when the probability of observing a head or tail, the parameter ϕ , depends on some attribute x , as usual in a stochastic way. An example is illustrated in Fig. 12 with 100 examples plotted as star symbols. The data suggests that it is far more likely that the class is $y = 0$ for smaller (possibly negative) values of x , and $y = 1$ for larger values of x . They also show that the transition between the low and high probability region is smooth. We can qualify this hypothesis by a parameterized density function p known as a *logistic* (sigmoidal) function:

$$p(y = 1) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}.\tag{29}$$

As before, we can then treat this density as a function of the parameters $\boldsymbol{\theta}$ for the given data values (likelihood function), and apply MLE to estimate the values of the parameters for which the data is most likely.

How can we use the knowledge (estimate) of the density function to perform classification? The obvious choice is to predict the class with the highest probability, given the input attribute. This *Bayesian decision point*, denoted by \mathbf{x}_d , is characterized by

$$\begin{aligned}p(y = 1|\mathbf{x}_d) &= p(y = 0|\mathbf{x}_d) = 0.5 \\ \Leftrightarrow \boldsymbol{\theta}^T \mathbf{x}_d &= 0,\end{aligned}\tag{30}$$

where the last expression is called the *dividing hyperplane*.

We looked here at binary classification with linear decision boundaries as a logistic regression, but we could also generalize this method to problems where hypotheses have different functional forms, creating nonlinear decision boundaries. However, coming up with specific functions for boundaries is often difficult in practice, and we will discuss more practical methods for binary classification later in this chapter.

3.3 Multivariate Generative Models and Probabilistic Reasoning

We have so far only considered very simple hypotheses appropriate for the low dimensional data given in the above examples. An important issue that has to be considered in machine learning is generalizing to more complex nonlinear data in high-dimension, that is, when many factors interact in a complicated way. This topic is probably one of the most important when applying machine learning to real world data. This section discusses a useful way of formulating more complicated stochastic models with causal relations and how to use such models to argue, i.e. do inference.

Let us consider high-dimensional data and the corresponding supervised learning problem which is simply a generalization of our discussions above. In the probabilistic framework, this means making a hypothesis of joint density function for the problem:

$$p(y, \mathbf{x}) = p(y, x_1, x_2, \dots | \boldsymbol{\theta}), \quad (31)$$

where y, x_1, \dots are random variables and $\boldsymbol{\theta}$ represents the parameters of the model. With this joint density function we could argue about every possible situation in the environment. For example, we could request classification or object recognition by calculating the conditional density function

$$p(y|\mathbf{x}) = p(y|x_1, x_2, \dots; \boldsymbol{\theta}). \quad (32)$$

Of course, the general joint density function and even this conditional density function for high-dimensional problems typically have many free parameters that we need to calculate with MLE. Thus it is useful to make more careful assumptions of causal relations that would restrict the density functions.

The object recognition formulation above is sometimes called a *discriminative approach* to object recognition because it tries to discriminate labels given the feature values. Another approach is to consider modeling the inverse conditional density

$$p(\mathbf{x}|y) = p(x_1, x_2, \dots | y; \boldsymbol{\theta}). \quad (33)$$

This is called a *generative model* as it can generate examples from a class, given its label. To use generative models in classification or object recognition we can apply Bayes' rule and calculate a discriminative model. It means relying on *class priors* (the relative frequencies of the classes) to calculate the probability that an item with features \mathbf{x} belongs to a class y :

$$p(y|\mathbf{x}; \boldsymbol{\theta}) = \frac{p(\mathbf{x}|y; \boldsymbol{\theta})p(y)}{p(\mathbf{x})}. \quad (34)$$

While using generative models for classification seems to be much more elaborate, there are several reasons that make generative models attractive for machine learning. For example, in many cases, features might be conditionally independent given a label, i.e. they verify

$$p(x_1, x_2, \dots | y) = p(x_1 | y) p(x_2 | y) \dots \quad (35)$$

where the indication of the parameter vector was dropped to make the formula less cluttered. Even if the independence does not strictly hold, this *naive Bayes assumption* is often useful and drastically reduces the number of parameters that must be estimated. This can be seen by factorizing the full joint density function with the chain rule

$$\begin{aligned} p(x_1, x_2, \dots, x_n | y) &= p(x_n | y, x_1, \dots, x_{n-1}) p(x_1, \dots, x_{n-1} | y) \\ &= p(x_n | y, x_1, \dots, x_{n-1}) \dots p(x_2 | y, x_1) p(x_1 | y) \\ &= \prod_{j=1}^n p(x_j | y, x_{j-1}, \dots, x_1). \end{aligned} \quad (36)$$

But what if the naive Bayes assumption is not appropriate? Then we need to build more elaborate models, or *causal models*. This particular challenge has been greatly simplified with *graphical methods* that specify the conditional dependencies between random variables using graphs [20]. A well known example from one of the inventors of graphical models, Judea Pearl, is shown in Fig. 13. In graphical models, the nodes represent random variables, and the links between them represent causal relations with conditional probabilities. In the case shown here, there are arrows on the links and the graph contains no loops, which makes it an example of *directed acyclic graph* (DAG). In contrast, the RBM discussed previously was an example of undirected Bayesian network.

In Fig. 13, each of the five nodes stands for a random binary variable: Burglary $B = \{\text{yes}, \text{no}\}$, Earthquake $E = \{\text{yes}, \text{no}\}$, Alarm $A = \{\text{yes}, \text{no}\}$, JohnCalls $J = \{\text{yes}, \text{no}\}$, MaryCalls $M = \{\text{yes}, \text{no}\}$. In general, a joint distribution of several variables can be factorized in various ways following the chain rule mentioned before Eq. (36), for example:

$$p(B, E, A, J, M) = p(B|E, A, J, M) p(E|A, J, M) p(A|J, M) p(J|M) p(M). \quad (37)$$

In this case, with binary random variables we need $2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 31$ parameters to specify the full joint density function. However, the model of Fig. 13 restricts causal relations between the random variables to represent only a subset of the factorization of the joint probability function, namely

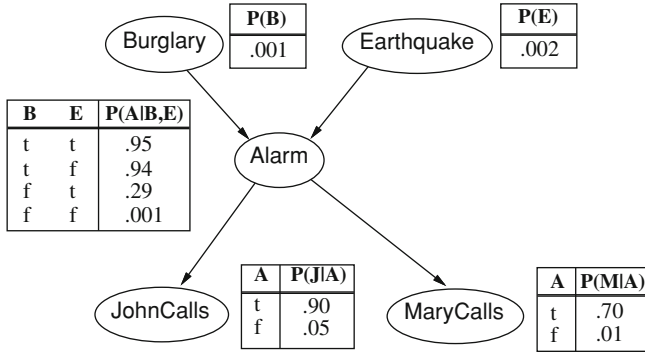


Fig. 13 Example of causal model with a two-dimensional probability density function (pdf) and a few other marginal pdf's

$$p(B, E, A, J, M) = p(B)p(E)p(A|B, E)p(J|A)p(M|A). \quad (38)$$

Therefore, we only need $1 + 1 + 2^2 + 2 + 2 = 10$ parameters to specify all the knowledge in the system. Example parameters for a specific case are displayed in the *conditional probability tables* (CPTs), which define the conditional probabilities represented by the links between the nodes. The graphical representation makes is very convenient to represent the particular hypotheses about causal relations.

The graph structure of the model also makes it easier to do inference (draw conclusions) on specific questions. For example, say we want to know the probability that there was no earthquake or burglary when the alarm rings and both John and Mary call. This is expressed by

$$\begin{aligned} p(B = f, E = f, A = t, J = t, M = t) \\ &= p(B = f)p(E = f)p(A = t|B = f, E = f)p(J = t|A = t)p(M = t|A = t) \\ &= 0.999 * 0.998 * 0.001 * 0.9 * 0.7 \\ &= 0.00063 \end{aligned}$$

where f stands for false and t for true. Although we have a causal model where parent variables influence the outcome of child variables, we can also use evidence from child variables to infer possible values for the parent variables. For example, let us calculate the probability that the alarm rings given that John calls, $p(A = t|J = t)$. For this we should first calculate the probability that the alarm rings as we will need this later. It is given by

$$\begin{aligned} p(A = t) &= p(A = t|B = t, E = t)p(B = t)p(E = t) \\ &\quad + p(A = t|B = t, E = f)p(B = t)p(E = f) \\ &\quad + p(A = t|B = f, E = t)p(B = f)p(E = t) \\ &\quad + p(A = t|B = f, E = f)p(B = f)p(E = f) \end{aligned}$$

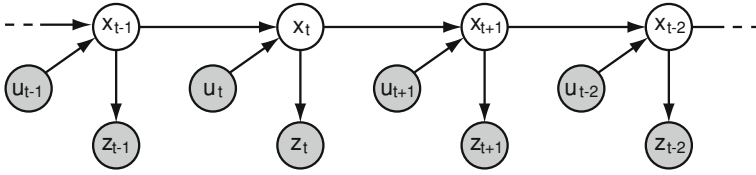


Fig. 14 A temporal Bayesian network called a Hidden Markov Model (HMM), with hidden states x_t , observations z_t , and external influences u_t

$$\begin{aligned}
 &= 0.95 * 0.001 * 0.002 + 0.94 * 0.001 * 0.998 \\
 &\quad + 0.29 * 0.999 * 0.002 + 0.001 * 0.999 * 0.998 \\
 &= 0.0025.
 \end{aligned}$$

We can then use Bayes' rule to calculate the required probability:

$$\begin{aligned}
 p(A = t | J = t) &= \frac{p(J = t | A = t)p(A = t)}{p(J = t | A = t)p(A = t) + p(J = t | A = f)p(A = f)} \\
 &= \frac{0.9 * 0.0025}{0.9 * 0.0025 + 0.05 * 0.9975} \\
 &= 0.043
 \end{aligned}$$

We can similarly apply the rules of probability theory to calculate other quantities, but these calculations can get cumbersome with larger graphs. It is therefore better to resort to numerical tools for the inference, for example a Matlab toolbox for Bayesian networks.¹

I already mentioned the importance of learning in temporal sequences (anticipatory systems), and Bayesian networks are easily extended to this domain, where they are called *dynamic Bayesian networks* (DBN). An important example of DBN is a *hidden Markov model* (HMM), as shown in Fig. 14. In this model, a state variable x_t is not directly observed and is called a *hidden* or *latent* random variable. The “Markov condition” in this model means that each state only depends on the previous state (or states), which can include external influences denoted here by u_t . A typical example is robot localization, where a robot is driven with some motor command u_t and the goal is to estimate the new state of the robot. We can use some knowledge about the influence of the motor command on the system to calculate a new expected location, and can also combine this in a Bayesian optimal way with sensor measurements denoted by z_t . Such Bayesian models are essential in many robotics applications.

¹ Available at <http://code.google.com/p/bnt/>, and used to implement Fig. 13; file at www.cs.dal.ca/~tt/repository/MLintro2012/PearlBurglary.m.

3.4 Nonlinear Regression and the Bias-Variance Tradeoff

While graphical models are great to argue about situations (doing inference), the role of supervised learning is to determine the parameters of the model. We have only considered binary models where each Bernoulli variable is characterized by a single parameter ϕ . However, the density function can be much more complicated than that and introduce many more parameters. Therefore, a major problem in practice is to have enough labeled training examples to restrict useful learning appropriately. This is one important reason for unsupervised learning, as we usually have a lot of unlabeled data that can be used to learn how to represent the problem appropriately in order to simplify the task. But we still need to understand the relations between free parameters and the amount of training data.

We already discussed the bias-variance tradeoff in the first section. Finding the right function that describes nonlinear data is one of the most difficult tasks in modeling, and there is no single algorithm that can give us the answer. This is why more general learning machines, which we will discuss in the next section, are popular. To evaluate the generalization performance of a specific model, it is helpful to split the training data into a *training set*, which is used to estimate the parameters of the model, and a *validation set*, which is used to study the *generalization performance* on data that has not been included during the training of the model.

A important question then becomes how much data we should keep for validation vs. training. If we use too much data for validation, then we might end up with too little data for accurate learning in the first place. On the other hand, if we have too little data for validation, then it might not be very representative. In practice, we often use some *cross-validation* technique to minimize the tradeoff, i.e. we use most of the data for training but repeat the selection of the validation data several times to make sure that the validation was not just a result of outliers. The repeated division of the data into a training set and a validation set can be done in different ways. For example, in *random subsampling* we merely use random subsamples for each set and repeat the procedure with other random samples. More common is *k-fold cross-validation*: in this technique, we divide the data set into k subsamples and use $k - 1$ subsamples for training and 1 subsample for validation. In the next round, we use another subsample to validate the training. A common choice for the number of subsamples is $k = 10$. By combining the results for the different runs we can often reduce the variance of our prediction while utilizing most data for learning.

We can sometimes help the learning process further. In many learning examples it turns out that some data is easy to learn while other data is much harder. In particular techniques called *boosting*, data that is hard to learn is oversampled in the learning set so that the machine has more opportunities to learn these examples. A popular implementation of such an algorithm is *AdaBoost* (adaptive Boosting).

Before proceeding to general nonlinear learning machines, I would like to outline a point that was eloquently made by Doug Tweed in a course module that we shared in the summer of 2012, part of a computational neuroscience program in Kingston, Canada. As discussed above, supervised learning is best phrased in terms

of regression and many applications are nonlinear in nature. It is common to make a nonlinear hypothesis under the form $y = h(\theta^T \mathbf{x})$, where θ is a parameter vector and h is a nonlinear function. A common example of such a model is an artificial perceptron with a sigmoidal transfer function in 1D such as $h(x; \theta) = \tanh(\theta x)$. However, as stressed by Doug, there is no reason to make the functions nonlinear *in the parameters*, which would result in a nonlinear optimization problem. Support Vector Machines (SVM; reviewed next) are a good example where the optimization error is simply quadratic in the parameters. The corresponding convex optimization has none of the local minima that plague multilayer perceptrons.

In summary, these different strategies can be expressed through the following optimization functions:

$$\text{Linear Perceptron: } E \propto (y - \theta^T \mathbf{x})^2 \quad (39)$$

$$\text{Nonlinear Perceptron: } E \propto (y - h(\mathbf{x}; \theta))^2 \quad (40)$$

$$\text{Linear in Parameters (LIP): } E \propto (y - \theta^T \phi(\mathbf{x}))^2 \quad (41)$$

$$\text{Linear SVM: } E \propto \alpha_i \alpha_j y_i y_j \mathbf{x}^T \mathbf{x} + \text{constraints} \quad (42)$$

$$\text{Nonlinear SVM: } E \propto \alpha_i \alpha_j y_i y_j \phi(\mathbf{x})^T \phi(\mathbf{x}) + \text{constraints} \quad (43)$$

The LIP model is more general than a linear model in that it considers functions of the form $y = \theta^T \phi(\mathbf{x})$ involving some mapping function $\phi(\mathbf{x})$. In light of this review, the transformation $\phi(\mathbf{x})$ can thus be seen as re-coding a sensory signal into a more appropriate form using unsupervised learning methods as discussed above.

3.5 General Learning Machines

Before we leave this discussion of basic supervised learning, I would like to mention some methods that are very popular and often used in machine learning applications. In the previous section we discussed the formulation of specific hypothesis functions. However, finding an appropriate hypothesis function requires considerable domain knowledge. To some extent, this is the “hard problem” in machine learning.

Finding general learning machines has long been on the minds of researchers, and this area has been especially inspired by human abilities and the brain itself as a learning machine. A good example are artificial neural networks, in particular multilayer perceptrons, which became popular in the 1980’s although they had been introduced much earlier. Boltzmann machines (discussed above) and support vector machines, which I briefly describe in this section, are also examples of this category. The overall concept behind these learning machines is to provide a very general function with many parameters that are adjusted through learning. Of course, the real problem then becomes to avoid “overfitting” the data with the model. This can

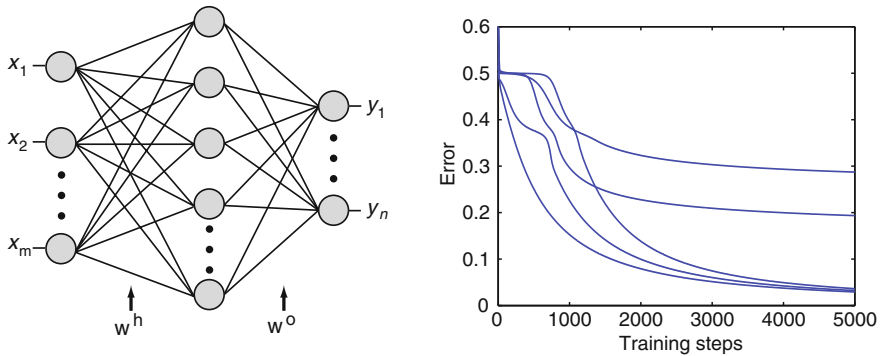


Fig. 15 Multilayer perceptron with one hidden layer. The parameters are called weights \mathbf{w} . The graph on the *right* shows example training curves when trained on XOR data

be done by applying appropriate restrictions and making the learning efficient enough so that it can be used for a larger problem size.

Scientists who design different models specially tailored to the different cognitive functions and their applications point out that a general learning machine is always at a disadvantage. There is “no free lunch”, they argue, meaning that we need to create specific models for specific problems. While this is true in principle, general learning machines can still be successful by providing answers where other methods are not known. In fact, these methods are currently experiencing something like a renaissance as they are now applied to massive data sets, whose size also help alleviate overfitting issues (see [21] for a recent example).

Let us begin with a multilayer perceptron (MLP) as shown in Fig. 15. Each node represents a simple calculation. The input layer relays the inputs, while the hidden (resp. output) layer multiplies each input channel x_j (resp. h_i) by an associated weight w_{ij}^h (resp. w_{ki}^o), sums these net inputs, then passes them through a transfer function, generally nonlinear, often the sigmoid curve of the logistic function. This reads:

$$y_k = g \left(\sum_i w_{ki}^o g \left(\sum_j w_{ij}^h x_j \right) \right). \quad (44)$$

A network of this type is a graphical representation of nested nonlinear functions with parameters \mathbf{w} . Applying a particular input results in a particular output \mathbf{y} , which can be compared to a desired output \mathbf{y}' in supervised learning. The parameters can then be adjusted as usual in LMS regression, by minimizing the least square error $E = \|\mathbf{y} - \mathbf{y}'\|^2$, typically via a gradient descent:

$$w \leftarrow w + \alpha \frac{\partial E}{\partial w}, \quad (45)$$

Table 2 A multilayer perceptron with backpropagation for solving the XOR problem

```

clear;
N_i = 2; N_h = 2; N_o = 1;
w_h = randn(N_h, N_i); w_o = randn(N_o, N_h);

%training vectors (XOR)
r_i = [0 1 0 1 ; 0 0 1 1];
r_d = [0 1 1 0];

%Updating and training network with sigmoid activation function
for trial = 1:5000;
    r_h = 1./(1 + exp(-w_h*r_i));
    r_o = 1./(1 + exp(-w_o*r_h));

    %error over all pattern
    d(trial) = 0.5*sum((r_o-r_d).^2);

    %training
    d_o = (r_o.*(1-r_o)).*(r_d-r_o);
    d_h = (r_h.*(1-r_h)).*(w_o'*d_o);
    w_o = w_o + 0.7*(r_h*d_o)';
    w_h = w_h + 0.7*(r_i*d_h)';
end
plot(d)

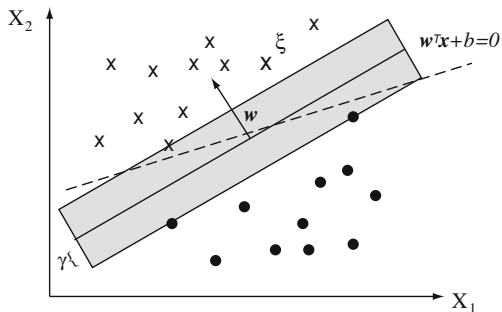
```

where α is a learning rate. Since \mathbf{y} is a nested function of the parameters, this requires the application of the chain rule. The resulting equations appear to be “propagating back” an error term $\mathbf{y} - \mathbf{y}'$ from the output to the previous layers, and for this reason this algorithm has been termed *error-backpropagation* [22]. An example program of an MLP that learns to represent the Boolean logic XOR function is shown in Table 2, and training curves in Fig. 15.

It is easy to see that such networks are universal approximators [23], i.e. the error of the training examples can be made as small as desired by increasing the number of parameters. This can be achieved by adding hidden nodes. However, the aim of supervised learning is to make predictions, that is to minimize the generalization error and not the training error. Thus, choosing a smaller number of hidden nodes might be more appropriate for this goal. The bias-variance dilemma [1] reappears here in this specific graphical model, and years of research have been investigated in solving this puzzle. Good practical methods and research directions have been proposed to counter overfitting, such as early stopping [24], weight decay [25] or Bayesian regularization [26]. Also, transfer learning [27, 28] can be seen as biasing models beyond the current data set.

A more recent general learning machine for classification are support vector machines, which were introduced by Vapnik, Guyon and Boser in 1992 [29, 30]. These machines are fundamentally based on minimizing the estimated generalization (called the “empirical error” in this community). The main idea behind SVMs for binary classification is that the best linear classifier for a separable binary classification problem is the one that maximizes the margin between a separating classification

Fig. 16 Illustration of linear support vector classification



line (separating hyperplane in higher dimensions) and the nearest data points [31]. Since there are many lines that can separate the data, as shown in Fig. 16, the most robust line is expected to be positioned as far from any data point as possible, since we also expect new data to be more likely to fall near the clusters of the training data—if the training data is indeed representative of the general distribution. In the end, the separating line is determined only by a few close points that are the ones called *support vectors*.

Vapnik’s important contributions did not stop there. He also formulated the margin maximization problem in a form such that the formulas are quadratic in the parameters and only contain dot products of training vectors, $\mathbf{x}^T \mathbf{x}$ by solving the dual problem cast in a Lagrangian formalism [30]. This has several important benefits. The problem becomes a convex optimization challenge, which avoids the local minima that have crippled MLPs. Furthermore, since only dot products between example vectors appear in these formulations, it is possible to apply a so-called “kernel trick” to efficiently generalize these approaches to nonlinear functions.

Let me illustrate the idea behind using kernel functions for dot products. To do this, it is important to distinguish attributes from features as follows. Attributes are the raw measurements, whereas features can be made up by combining attributes. For example, the attributes x_1 and x_2 could be combined into a tentative feature vector $(x_1, x_2, x_1 x_2, x_1^2, x_2^2)^T$. This is a bit like trying to guess a better representation of the problem, one that should be “useful” as discussed above in the part about structural learning. Let us now denote this transformation by a function $\phi(\mathbf{x})$. The interesting part in Vapnik’s formulation is that we actually do not even have to calculate this transformation explicitly, but we can replace the corresponding dot products by a *kernel function*

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z}), \quad (46)$$

which is often much easier to calculate. For example, a Gaussian kernel function formally corresponds to an infinite-dimensional feature transformation ϕ . There are some arguments from structural learning [30, 32] why SVMs are less prone to overfitting, and extensions have also been made to problems with overlapping data in the form of soft margin classification [31]. These ideas have also been generalized to regression problems [33], but their performance is often not as satisfactory. We will

Table 3 Using LibSVM for classification

```

clear; close all; figure; hold on; axis square

%training data and training SVM
r1 = 2 + rand(300, 1); a1 = 2*pi*rand(300, 1); polar(a1, r1, 'bo');
r2 = randn(300, 1); a2 = .5*pi*rand(300, 1); polar(a2, r2, 'rx');

x = [r1.*cos(a1), r1.*sin(a1); r2.*cos(a2), r2.*sin(a2)];
y = [zeros(300, 1); ones(300, 1)];
model = svmtrain(y, x);

%test data and SVM prediction
r1 = 2 + rand(300, 1); a1 = 2*pi*rand(300, 1);
r2 = randn(300, 1); a2 = .5*pi*rand(300, 1);
x = [r1.*cos(a1), r1.*sin(a1); r2.*cos(a2), r2.*sin(a2)];
yp = svmpredict(y, x, model);

figure; hold on; axis square
[tmp, I] = sort(yp);
plot(x(1:600-sum(yp), 1), x(1:600-sum(yp), 2), 'bo');
plot(x(600-sum(yp) + 1:600, 1), x(600-sum(yp) + 1:600, 2), 'rx');

```

not dive more into the theory of Support Vector Machine but show instead an example using the popular LibSVM [34] implementation. This implementation includes interfaces to many programming languages, such as MATLAB and Python. SVMs are probably currently the most successful general learning machines for classification.

Table 3 gives an example of applying the LibSVM library to the data displayed in Fig. 17. The plot on the left is the training data, which is produced from sampling two distributions: the points in the first class (blue circles) are chosen within a ring of radius 2.0–3.0, while the points in second class (red crosses) are distributed across two quadrants. The examples are provided with their corresponding labels to the training function `svmtrain`. Similarly, the plot on the right of Fig. 17 is test data. The corresponding class labels are given to the function `svmpredict` only for the purpose of calculating the cross-validation error. For true predictions, this vector can be set to arbitrary values. The performance of this classification is around 97 % with the standard parameters of the LibSVM package. However, it is advisable to tune these parameters, for example with search methods [35].

While SVMs have had a large impact in application-oriented machine learning, more recent research works are combining ideas from SVMs (in particular kernel methods), Bayesian networks, and good old-fashioned neural networks. Such hybrid methods are now taking off, too, and starting to have another great impact—not only in research but also in many industrial domains.

4 Reinforcement Learning

As discussed above, a basic form of supervised learning is function approximation, relating input vectors to output vectors, or more generally finding density functions

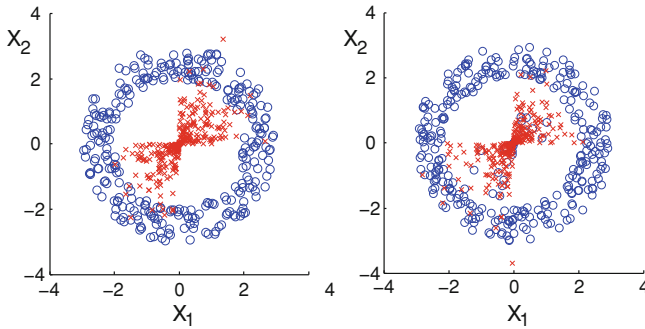


Fig. 17 Example of using training data on the *left* to predict the labels of the test data on the *right*

$p(\mathbf{y}, \mathbf{x})$ from examples $(\mathbf{x}^{(e)}, \mathbf{y}^{(e)})$. However, in many applications we do not have a teacher to tell us exactly at any time the appropriate response to a specific input. Rather, feedback from a teacher is often delayed and given only in the form of general feedback such as ‘good’ or ‘bad’, instead of a detailed explanation about what the learner should have done.

We are now turning to these more general learning problems. Specifically, we are interested in learning a sequence of appropriate actions to maximize an expected payoff. More formally, let us learn a temporal density function

$$p(\mathbf{y}(t+1)|\mathbf{x}(t), \mathbf{x}(t-1), \dots, \mathbf{x}(1)). \quad (47)$$

We have already encountered such models in the form of temporal Bayesian networks. We will now discuss this issue further within the realm of *reinforcement learning* or *learning from reward*. While we mainly consider here the prediction of a scalar utility function, most of this discussion can be applied directly to a more graded environmental feedback.

4.1 Markov Decision Processes

Reinforcement learning is best illustrated in a Markovian world.² As discussed before, such a world is characterized by transition probabilities between states, $T(s'|s, a)$, that only depend on the current state $s \in S$ and the action $a \in A$ taken in this state. We now consider feedback from the environment in the form of a reward $r(s)$ and ask what actions should be taken in each state to maximize future reward. More formally, we define the *value function* or *utility function*

$$Q^\pi(s, a) = E[r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + \dots]_\pi, \quad (48)$$

² Markov models are often a simplification or abstraction of a real world. In this section, however, we discuss a “toy world” in which state transitions were designed to fulfill the Markov condition.

as the expected future payoff (cumulative reward) for being in state s , then s_1, s_2 , etc. We introduce here the “discount factor” $0 \leq \gamma < 1$ to express that we value immediate reward over later reward. This is a common treatment to keep the expected value finite. An alternative scheme would be to consider only finite action sequences. The policy $\pi(a|s)$ describes what action can be taken in each state. In accordance with our overall probabilistic world view, we consider the most general case of probabilistic policies, i.e., we want to know with what probability a given action should be chosen. If the policy was deterministic, then taking a specific action would be determined by applying the policy to the current state, and the value function is often denoted by $V^\pi(s)$.³ Our goal is to find the optimal policy π^* , i.e. the one that maximizes the expected future payoff Q^π :

$$\pi^*(a|s) = \arg \max_{\pi} Q^\pi(s, a). \quad (49)$$

This search is called a *Markov Decision Process* (MDP).

MDPs have been studied since the mid 1950s, and Richard Bellman noted that it was possible to calculate the value function for each state, and a given policy π , using a self-consistent equation now named the *Bellman equation*. He also called the corresponding algorithm *dynamic programming*. Specifically, we can separate the expected value of the immediate reward from the expected value of the reward from visiting subsequent states as follows:

$$Q^\pi(s, a) = E[r(s)]_\pi + \gamma E[r(s_1) + \gamma r(s_2) + \gamma^2 r(s_3) + \dots]_\pi. \quad (50)$$

The first expected value on the right-hand side is simply the immediate reward received when reaching state s at this particular point in time. The second expected value is the function of state s_1 . State s_1 is related to state s , since s_1 can be reached from s when taking action a_1 with a certain probability according to policy π (for example by setting $s_1 = s + a_1$, or more generally $s_n = s_{n-1} + a_n$). The state actually reached can also depend on stochastic environmental factors encapsulated in the matrix $T(s'|s, a)$. Incorporating these factors into the equation yields

$$Q^\pi(s, a) = r(s) + \gamma \sum_{s'} \left(T(s'|s, a) \sum_{a'} \left(\pi(a'|s') E[r(s') + \gamma r(s'_1) + \gamma^2 r(s'_2) + \dots]_\pi \right) \right), \quad (51)$$

where s'_1 is the next state after state s' , etc. Thus the expression on the right is the state-value-function of state s' . If we substitute it with the corresponding expression on the left of Eq. (48), we get the *Bellman equation for a specific policy*, namely

$$\textbf{Bellman-stoch-}\pi: \quad Q^\pi(s, a) = r(s) + \gamma \sum_{s'} \left(T(s'|s, a) \sum_{a'} \left(\pi(a'|s') Q^\pi(s', a') \right) \right). \quad (52)$$

³ $V^\pi(s)$ is usually called the *state value function* and $Q^\pi(s, a)$ the *state-action value function*. Note, however, that the value depends in both cases on the states *and* the actions taken.

The variant of this equation for deterministic policies is a bit simpler. Since in this case an action a is uniquely specified by the policy, the value function $Q^\pi(s, a)$ reduces to $V^\pi(s)$ and the equation becomes⁴

$$\textbf{Bellman-det-}\pi: V^\pi(s) = r(s) + \gamma \sum_{s'} (T(s'|s, a) V^\pi(s')). \quad (53)$$

The Bellman equation is a set of N linear equations in an environment with N states, one equation for each unknown value function of each state. The environment being given, i.e. having functions r and T , we can use well-known methods from linear algebra to solve for $V^\pi(s)$. This can be formulated compactly by matrix notation, in which s and s' are the indices:

$$\mathbf{r} = (\mathbf{I} - \gamma \mathbf{T}) \mathbf{V}^\pi, \quad (54)$$

where \mathbf{r} is the reward vector, \mathbf{I} is the identity matrix, and \mathbf{T} is the transition matrix. To solve this equation we have to invert a matrix and multiply this with the reward values,

$$\mathbf{V}^\pi = (\mathbf{I} - \gamma \mathbf{T})^{-1} \mathbf{r}^T, \quad (55)$$

where \mathbf{r}^T is the transpose of \mathbf{r} . We can also use the Bellman equation directly to calculate a state-value-function iteratively. We can start with a guess \mathbf{V} for the value of each state, then calculate from this a better estimate:

$$\mathbf{V} \leftarrow \mathbf{r} + \gamma \mathbf{T} \mathbf{V} \quad (56)$$

and so on, until this process converges. Either way, we get a value function for a specific policy. To find the best policy, the one that maximizes the expected payoff, we have to loop through different policies and find the maximal value function. This can be done in different ways, most commonly by using the *policy iteration*, which starts with a guess policy, iterates a few times the value function for this policy, and then chooses a new policy that maximizes this approximate value function. This process is repeated until convergence.

Table 4 provides an example program for a simple 1D state space consisting of a chain of 10 states, as shown on the left of Fig. 18. The 10th state is rewarded with $r = 1$, while the first state receives a large negative reward, $r = -1$. The intermediate

⁴ This formulation of the Bellman equation for an MDP [36–38] is slightly different from the formulation of Sutton and Barto in [39], as these authors define the value function to be the cumulative reward starting from the next state, not the current state. In their case, the Bellman equation reads $V^\pi(s) = \sum_{s'} T(s'|s, a)(r(s') + \gamma V^\pi(s'))$. This is only a matter of convention about when we consider the prediction: just before getting the current reward or after taking the next step.

Table 4 Program for the chain example using the policy iteration process

```

%chain example: policy iteration

%parameters
clear; N = 10; P = 0.8; gamma = 0.9;

%reward function
r = zeros(1, N) - 0.1; r(1) = -1; r(N) = 1;

%initiality random start policy and value function
policy = ceil(2*rand(1, N)); policy(1) = 2; policy(N) = 1;
Vpi = rand(1, N); Vpi(1) = r(1); Vpi(N) = r(N);

for iter = 1:3
    %estimate V for this policy
    for i = 1:10
        for s = 2:N-1
            snext = s-1 + 2*(policy(s)-1);
            sother = s + 1-2*(policy(s)-1);
            Vpi(s) = r(s) + gamma*(P*Vpi(snext) + (1-P)*Vpi(sother));
        end
    end
    %updating policy
    for s = 2:N-1
        [tmp, policy(s)] = max([Vpi(s-1), Vpi(s + 1)]);
    end
end
plot(Vpi);

```

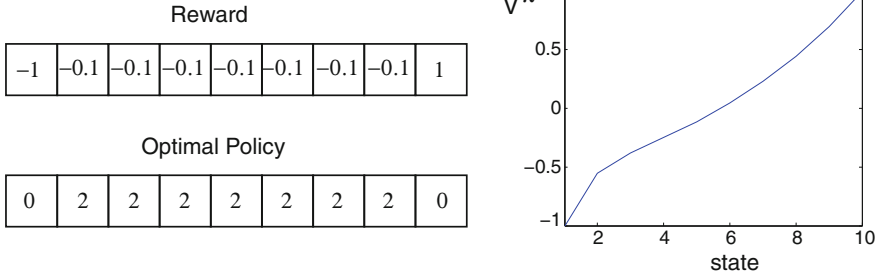


Fig. 18 Example of using policy iteration on a chain of rewarded states. *Left* reward values and optimal policy for each state, where a policy value 1 means “go left” (not present) and a value 2 means “go right”. No further action is taken in the end states

states receive a small negative reward to account for movement costs. After three iterations, the policy reaches the optimal one (bottom left of figure). Actually, the optimal policy is often found within just one or two iterations, so the extra iteration was added to ensure that the value function was properly calculated for this policy.

It is also possible to derive a version of the Bellman equation *for the optimal value function* itself:

$$\textbf{Bellman-det-}^* \quad V^*(s) = r(s) + \max_a \gamma \sum_{s'} T(s'|s, a) V^*(s'). \quad (57)$$

The max function is a little more difficult to implement in the analytic solution, but we can again easily use an iterative method to solve for this optimal value function. This algorithm is called *value iteration*. The *optimal policy* can always be calculated from the optimal value function with

$$\pi^*(s) = \arg \max_a \sum_{s'} T(s'|s, a) V^*(s'). \quad (58)$$

A policy tells an agents what action should be chosen, hence the optimal policy is related to optimal control as long as the reward reflects the desired performance.

The previously discussed policy iteration has some advantages over value iteration. In value iteration we have to try out all possible actions when evaluating the value function, which can be time consuming when there are many possible actions. In policy iteration, we choose only one specific policy, although we then have to iterate over consecutive policies. In practice, it turns out that policy iteration often converges fairly rapidly.

4.2 Temporal Difference Learning

In dynamic programming, we iterate repeatedly over every possible state of the system. This only works if we have complete knowledge of the system. In that scenario, the agent does not even have to ‘perform’ the actions physically, which would be very time consuming. Instead, the agent can just ‘sit’ and calculate the solution during a “planning phase”. However, in many cases we do not know the rewards given in different states, and we usually have to estimate transition probabilities, too, etc. One approach would be to estimate these quantities by interacting with the environment before using dynamic programming. In contrast, the following methods are more direct estimations of the state value function that determines the optimal actions. These online methods assume that we still know exactly in which state the agent is, and they can be generalized to partially observable situations by considering probability maps over the state space.

A general strategy for estimating the value of states is to act in the environment and thereby sample reward. This sampling should be done with some degree of stochasticity to ensure sufficient exploration of the states. These methods are generally called *Monte Carlo* methods. Monte Carlo methods can be combined with the bootstrapping ideas of dynamic programming, and the resulting algorithms are called *temporal difference* (TD) learning, since they rely on the difference between expected reward and actual reward.

We start again by estimating the value function for a specific policy before moving to schemes for estimating the optimal policy. The Bellman equations require the estimation of future reward:

$$\sum_{s'} T(s'|s, a) V^\pi(s') \approx V^\pi(s'). \quad (59)$$

In this equation we introduced an approximation of the sum by the value of the state that is reached in one Monte Carlo step. In other words, we replace the total sum that we could build knowing the environment with a single sampling step. While this approach is only an estimation, the idea is that it will still result in an improvement of the estimation of the value function, and that other trials have the possibility to evaluate other states that have not been reached in this trial. The value function should then be updated carefully, by considering the new estimate only incrementally:

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha[r(s) + \gamma V^\pi(s') - V^\pi(s)]. \quad (60)$$

This is called *temporal difference* or *TD learning*. The constant α is a learning rate and should be fairly small. This policy evaluation can then be combined with policy iteration as already discussed in the section on dynamic programming.

We should now think a little more about what policy to follow. An obvious choice is to take the action that leads to the largest expected payoff, also called *greedy policy*. Applying this policy should be optimal when the value function is exact. However, one problem with purely sticking to this strategy is that we might not be sufficiently “exploring” the state space—as opposed to “exploiting” known returns. We address this *exploration-exploitation dilemma* here by opting for stochastic policies. Thus we need to go back to the notation of the state-action value function (although we will drop the ‘*’ superscript for the optimal value function for convenience). To include randomness in the policy we can, for example, follow the greedy policy most of the time, and only choose another possible action with a small probability denoted by ε . This probabilistic policy is called the ε -*greedy policy* and can be formulated as

$$\pi(a = \arg \max_a Q(s, a)) = 1 - \varepsilon. \quad (61)$$

A more graded approach employs the *softmax policy*, which chooses each action proportionally to a Boltzmann distribution:

$$\pi(a|s) = \frac{e^{\frac{1}{T} Q(s, a)}}{\sum_{a'} e^{\frac{1}{T} Q(s, a')}}. \quad (62)$$

This policy chooses most often the action with the highest expected reward, followed by the second highest, etc., where the temperature parameter T sets the relative probability of these choices.

We can now use these policies to explore the state space and estimate the optimal value function with temporal difference learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r(s) + \gamma Q(s', a') - Q(s, a)], \quad (63)$$

where the actions a' is the action chosen according to the policy. This *on-policy TD algorithm* is called *Sarsa* for state-action-reward-state-action [39]. A variant of this approach uses the stochastic action above only when choosing the next state, but estimates the value function by considering the other possible actions, too:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r(s) + \max_{a'} \gamma Q(s', a') - Q(s, a)]. \quad (64)$$

This is called an *off-policy TD algorithm*, or Q-learning [40]. These algorithms have been instrumental in the success of reinforcement learning in many engineering applications.

4.3 Function Approximation and $TD(\lambda)$

The large number of states in real-world applications makes these algorithms unpractical. This was already noted by Richard Bellman himself, who coined the phrase “*curse of dimensionality*”. We have only considered discrete state spaces, while many applications involve a continuous state space. While discretizing a continuous state space is a common approach, increasing the resolution of the discretization has the consequence of increasing the number of states exponentially. Another major problem in practice is that the environment is not fully, or reliably, observable. Thus we might not even know exactly in which state the agent finds itself when considering the value update. A common approach to a “partially observable Markov decision process” (POMDP) is the introduction of a *probability map*. In the update of the Bellman equation, we need then to consider all possible states that can be reached from the current state, something which will typically increase the number of calculations even further. We will not follow this approach here but rather consider the use of *function approximators* to overcome these problems. A more general discussion of reinforcement learning in continuous state and action spaces is given in [41].

The idea behind the following method is to make a hypothesis of the relation between sensor data and expected values in the form of a parameterized function as in supervised learning⁵:

$$V_t = V(\mathbf{x}_t) \approx V(\mathbf{x}_t; \boldsymbol{\theta}), \quad (65)$$

⁵ The same function name is used on both sides of this equation, but these are distinguished by the inclusion of parameters. The value functions all refer to the parametric model, which should be clear from the context.

and to estimate the parameters by maximum likelihood as before. We use here a time index to distinguish state sequences. In principle, one could build very specific temporal Bayesian models for specific problems as discussed above, but in this circumstance I will outline the use of general learning machines. In particular, let us adjust the weights of a neural network using gradient-descent methods on a mean square error (MSE) function:

$$\Delta\theta_j = \alpha \sum_{t=1}^m (r - V_t) \frac{\partial V_t}{\partial \theta_j}. \quad (66)$$

We consider here the total change of the weights for a whole episode of m time steps by summing the errors for each time step. One specific difference between this situation and the supervised learning examples before is that the reward is only received after several time steps in the future, at the end of an episode. One possible approach to manage this situation is to keep a history of our predictions and make the changes for the whole episode only after the reward is received at the end. This is what we have done in Eq. (66) by providing the reward r as supervision signal in each timestep. Another approach is to make incremental (online) updates by following the TD learning philosophy, and replacing the supervision signal for a particular time step by the prediction of the value of the next time step. Specifically, we can write the difference between the received reward $V_{m+1} = r$ at the end of the sequence and the prediction V_t at time t as

$$r - V_t = \sum_{u=t}^m (V_{u+1} - V_u) \quad (67)$$

since the intermediate terms cancel out. Replacing this in Eq. (66) yields

$$\Delta\theta_j = \alpha \sum_{t=1}^m \sum_{u=t}^m (V_{u+1} - V_u) \frac{\partial V_t}{\partial \theta_j} \quad (68)$$

$$= \alpha \sum_{t=1}^m (V_{t+1} - V_t) \sum_{u=1}^t \frac{\partial V_u}{\partial \theta_j}, \quad (69)$$

which can be verified by developing the sums and reordering the terms. Of course, this is only rewriting the original equation, Eq. (66). We still have to keep a memory of all the gradients from the previous time steps, or at least a running sum of these gradients.

While the rules portrayed in Eqs. (66) and (69) are equivalent, Richard Sutton [42] suggested a modified version that multiplied recent gradients by stronger weights than gradients in the more remote past. For this, he introduced a decay factor $0 \leq \lambda \leq 1$. The rule above corresponds to $\lambda = 1$ and is called the *TD(1) rule*, while the more general *TD(λ) rule* is given by

$$\Delta_t \theta_j = \alpha (V_{t+1} - V_t) \sum_{u=1}^t \lambda^{t-u} \frac{\partial V_u}{\partial \theta_j}. \quad (70)$$

It is also interesting to look at the other extreme, when $\lambda = 0$. The *TD(0) rule* is given by

$$\Delta_t \theta_j = \alpha (V_{t+1} - V_t) \frac{\partial V_t}{\partial \theta_j}. \quad (71)$$

While this last rule gives in principle different results from the original supervised learning problem described by TD(1), it has the advantage that it is local in time, does not require any memory, and often still works very well. The TD(λ) algorithm can be implemented in a multilayer perceptron where the error term is back-propagated to hidden layers. A generalization to stochastic networks has also been made within the framework of free-energy formalism [43].

5 Some Biological Analogies

The brain seems to be a very successful learning machine, and it is therefore not surprising that human capabilities have motivated much research in artificial intelligence. Conversely, insights from learning theory are important, too, for our understanding of brain processes. In this last section, I want to mention some interesting relations that neuroscience has with learning theory. I already remarked on the close links between unsupervised learning and receptive fields in the early sensory areas of the cortex, which I believe is a wonderful example of underlying mechanisms behind physiological findings. In the following, I would like to add comments on two other subjects related to supervised learning and reinforcement learning. The first is about *synaptic plasticity*, which appears to be an important mechanism for the physical implementation of learning rules. The second is about the close relation of reinforcement learning with classical conditioning and the *basal ganglia*. Classical conditioning has been a major area in animal learning, and recent recordings in the basal ganglia have helped relating these areas on a behavioural, physiological and learning-theoretical level.

5.1 Synaptic Plasticity

As speculated by the Canadian scientist Donald Hebb [44], the leading theory of the physical implementation of learning is that of synaptic changes, whereby the synaptic efficacy varies in response to causally related pre- and postsynaptic firings. Such correlation rules have first been made concrete by Eduardo Caianiello [45], and have recently been refined in terms of “spike timing-dependent plasticity” (STDP; see for example [46]). The main idea is that when a driving neuron participates in

firing a subsequent neuron, then the connection strength between these neurons will increase—whereas it will decrease in the absence of correlated firing. Many of the learning rules of neural networks have followed this main association rule through increment terms that are proportional to pre- and postsynaptic activity, such as

$$\Delta w_{ij} \propto x_i x_j. \quad (72)$$

Synaptic plasticity is not only a fascinating area in neuroscience but also constitutes an important medical issue, since neurodegenerative disorders, such as Alzheimer's disease and dementia, have synaptic effects and a great number of psychiatric medications exert their action on the synaptic receptors.

There are many mysteries left that need to be understood if we want to make progress in helping with neurological conditions and maybe even make progress in machine learning. One basic fact that seems puzzling is that synapses are not long-lasting compared to the time scale of human memories.⁶ Synapses consist of proteins that have to be actively maintained by protein synthesis. Thus, one may wonder how this maintenance can survive for years and support long-term memory, such as returning to our place of birth after many years of absence, or meeting friends whom we had not seen in ages. These are fundamental questions that, to my knowledge, have not been sufficiently addressed.

While the Hebbian perspective on synaptic plasticity and learning is well established, I would like to outline an aspect of synaptic plasticity that might be less well-known. In particular, I would like to point out the findings of my friend Alan Fine and his colleagues [47], which fit nicely with the probabilistic theme that I have emphasized in this chapter. Fine and colleagues have performed classical plasticity experiments that use high- or low-frequency stimulations of hippocampal slices of rodents to induce measurable changes in synapses. Some of their results are summarized in Fig. 19. To test the strength of the synapses, they stimulated them with two pulses, as paired pulses facilitate synaptic responses (the second pulse makes it easier to elicit a postsynaptic spike). The slices are then activated with high-frequency stimulations inbetween these tests. As shown in Fig. 19a, the electric response of the postsynaptic neuron as measured by the excitatory post-synaptic potential (EPSP) is higher after the high-frequency stimulation. This corresponds to the classical findings by Bliss and Lømo [48] and is called long-term potentiation (LTP), since this enhanced response to a presynaptic stimulus lasts relatively long compared to the usual scale of neuronal dynamics. Of course, the EPSP is a measure that can depend on multiple synapses. But Fine and colleagues also imaged the calcium-related optical luminance signal from individual synapses. This is shown in Fig. 19b. Surprisingly, they observed that this luminance did not change despite the fact the calcium-dependent mechanisms are generally associated with synaptic activity and plasticity. Instead, they found that the probability of eliciting a postsynaptic spike varied nicely. Specifically, the probability of transmitter release increases with high-frequency simulations that are usually associated with LTP. They could also

⁶ Julian Miller made this point nicely at the aforementioned workshop.

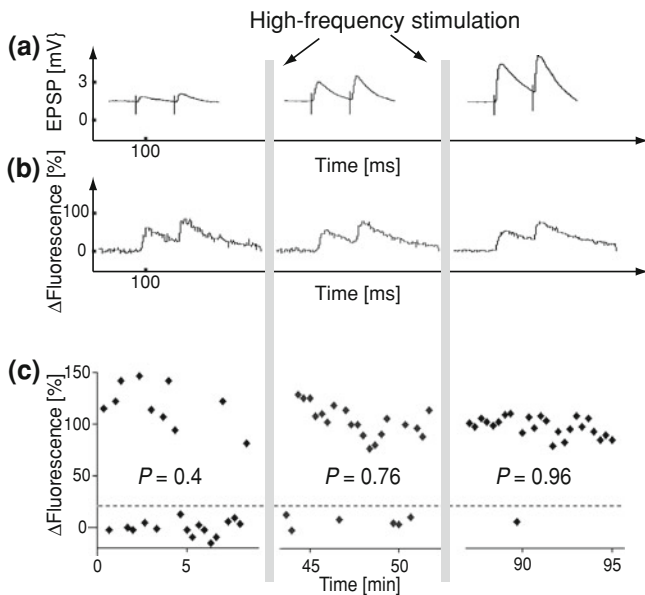


Fig. 19 Plasticity experiment in hippocampal slices in which not only EPSPs were measured, but additionally postsynaptic calcium-dependent fluorescence signals at single synapses were imaged (data courtesy of Alan Fine and Ryosuke Enoki, after [47])

lower the probability of transmitter release with low-frequency stimulus that usually elicits a decrease in EPSPs, called long-term depression (LTD; not shown in the figure).

A manipulation of the probability of transmitter release could explain the increased EPSP in such experiments. If there is a population of synapses that drive that neuron, than a population of synapses with higher likelihood of transmitter release would result in a larger EPSP than a population with smaller likelihood of transmitter release. In this sense, the findings are still consistent with some of the consequences of synaptic plasticity. But these findings also point to additional possibilities also consistent with the view that brain processing might be based on probabilistic computation rather than dealing with point estimates. Thus, the common view of a noisy nervous system with noisy synapses might be misleading. If this is noise in the sense of the “limitations” of a biological implementation, then why could the probability of synaptic responses be modulated reliably?

From a theoretical perspective it is rather difficult for noise to survive thresholding processes. For example, consider a biased random walk to a threshold as shown on the left-hand side in Fig. 20. In this example we add 1 plus a Gaussian noise (mean μ , standard deviation σ) to the signal at each time step, then the signal is reset when crossing the threshold. The noise in the process leads to different times of threshold crossings, and the variation of these times is related to the variations in the signal as shown on the right-hand side of Fig. 20 where the coefficient of variation $C_v = \sigma/\mu$

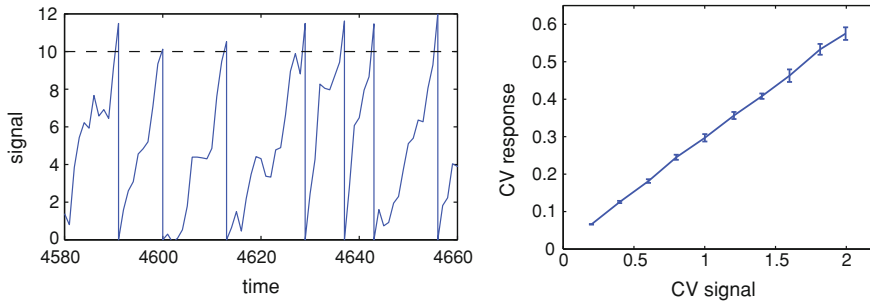


Fig. 20 Demonstration of the relation between variability in signal versus variability in spike timing response. The graph on the left side shows a noisy accumulation toward a threshold. The graph on the right shows how the coefficient of variation (C_V) varies with noise

is plotted. While there is a positive slope between them (higher noise leads to higher variations in firing times), the proportionality factor is only around $1/\sqrt{4\pi}$. Hence, if noise is an issue, then one could use thresholding mechanisms to reduce it and through repeated stages, as in the brain, the noise should become smaller. Or, in other words, if noise is the problem then one should filter it out early in the process and higher processes should be less noisy. In sum, it could be that signal variations in the brain are not all undesirable noise but could play an important information processing role such as representing the likelihood of sensory signals or the confidence in possible actions. This conjecture is consistent with the probabilistic approaches to machine learning.

5.2 Classical Conditioning and the Basal Ganglia

One of the important roles of computational neuroscience is to bridge the gap between behavioural and physiological findings [49]. The following discussion is a good example. Classical conditioning has been intensively studied in the psychological discipline of animal learning at least since the studies by Pavlov. One of the most basic findings of Pavlov is that it is possible to learn the fact that a stimulus is predicting a reward, and that this prediction elicits the same behaviour as the primary reward signal, such as salivation following a tone when the tone predicts food reward. Many similar predictions have been summarized very successfully by the Rescorla-Wagner theory [50]. In terms of the learning paradigms discussed above, this theory relates the change in the value of a state ΔV_i to the reward prediction error $\lambda - V_i$ by the formula

$$\Delta V_i = \alpha_i \beta (\lambda - V_i), \quad (73)$$

where factors α_i and β describe the salencies of the conditioned and unconditioned stimulus, respectively, and λ represents the reward. This model is equivalent to

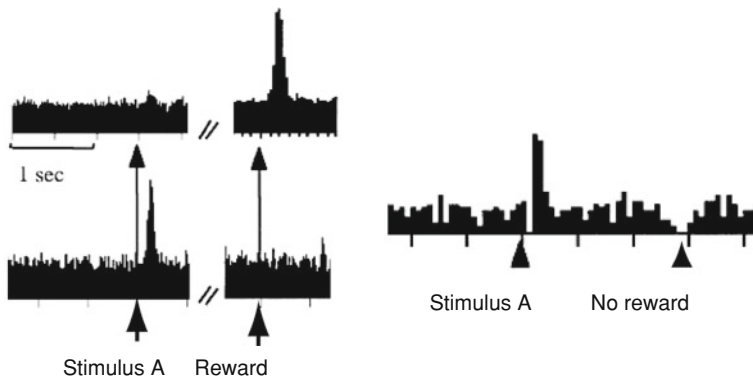


Fig. 21 Recordings by Schultz et al. [51] in a classical conditioning experiment, where a stimulus was presented followed by a reward. Early in the trials the SN neurons responded after the animal received a reward (*top left*), while the neurons responded to the predictor of the reward in later trials (*bottom left*). The neurons even seem to indicate the absence of an expected reward after learning (*right*)

temporal difference learning in a one-step prediction task where the reward follows immediately the stimulus.

The Rescorla-Wagner theory with its essential reliance on the reward prediction error is very successful in explaining behaviour, and it was very exciting when Wolfram Schultz [51] and colleagues discovered neural signatures of reward prediction errors. Schultz found these signals in the *substantia nigra*, which is part of a complex of different nuclei in the midbrain called the basal ganglia. Its name means “black substance”, and the dark aspect of this area is apparently due to a chemical compound related to dopamine, which these neurons transmit to the input area of the basal ganglia and to the cortex, and has been implicated in modulating learning. Some examples of the response of these neurons are shown in Fig. 21.

We can integrate the Rescorla-Wagner theory with these physiological findings in a neural network model, as shown in Fig. 22. The reward prediction error \hat{r} is conveyed by the nigra neurons to the striatum, an input area of the basal ganglia, in order to mediate the plasticity of cortical-striatal synapses. The synapses are thereby assumed to contain an eligibility trace, since the learning rule requires the association with the previous state. Many psychological experiments can be modeled by a one-step prediction task where the actual reward follows a specific condition. The learning rule can then be simplified to a temporal learning rule in which the term in γ can be neglected, corresponding to the model in Fig. 22a. The implementation of the full TD rule would require a fast side-loop as shown in Fig. 22b, which has been speculated to be associated with the subthalamus [52].

Of course, the anatomy of the basal ganglia is more elaborate than this. My student Patrick Connor and I have suggested a model with lateral interactions in the striatum [53] that has some physiological grounding [54] and can explain a variety of behavioral findings not covered by the Rescorla-Wagner model [55]. Moreover, there

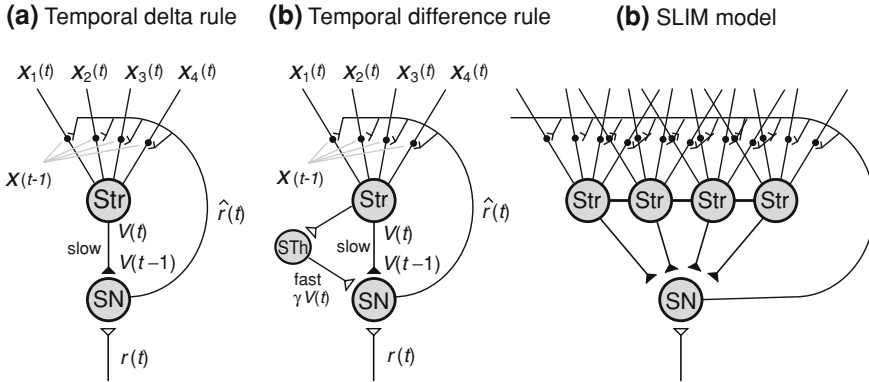


Fig. 22 Implementation of reinforcement learning models through analogies with the basal ganglia. **a** Single state of one-step reinforcement learning model (temporal delta rule) with cortical input, a striatal neuron (Str), and a neuron in the substantia nigra (SN) that conveys the reward prediction error to striatal spines. **b** Implementation of the temporal difference (TD) learning with a fast subthalamic side-loop. **c** Basic version of the striatal-with-lateral-inhibition (SLIM) model

are two main pathways through the basal ganglia, a direct pathway and an indirect one, with intermediate stages in distinct subregions of the basal ganglia (not shown in Fig. 22). The direct pathway has a inhibitory effect on the output neurons of the basal ganglia, while the indirect one has a facilitatory effect. Since the effect of the output of the basal ganglia is itself to inhibit motor areas, it has been speculated that the direct pathway could learn to inhibit non-rewarding actions, whereas the indirect pathway could learn to facilitate rewarding actions. Different alterations of specific pathways have been suggested to relate to different neurological conditions that are known to involve the basal ganglia, such as Parkinson disease, Tourette syndrome, ADHD, schizophrenia and others [56]. Thus, modeling and understanding this learning system has the potential to guide refined intervention strategies.

6 Outlook

Learning is an exciting field that has made considerable progress in the last few years, specifically through statistical learning theory and its probabilistic embedding. These theories have at least clarified what could be expected from ideal learning systems, such as their ability to generalize. Much progress has also been made in unsupervised learning and starting to tackle temporal learning problems. Most excitingly, the advances in this area have enabled machine learning to find its way out of the research labs and into commercial products that have recently revolutionized technologies, such as advanced gaming platforms and smarter recommendation systems. Statistical learning theory has clarified general learning principles, such as optimal generalizability and optimal (Bayesian) decision making in the face of uncertainties.

What are the outstanding questions, then? While machine learning has enabled interesting applications, many of these applications are very focused in scope. The complexity of the environments that humans face still appears far beyond the reach of our models. Scaling up methods even farther is important to enable more applications. Many believe that, to this goal, we require truly hierarchical systems [57], and more specifically systems that process temporal data [58]. While there is exciting progress in this field, learning to map simple features, such as pixels from an image, to high-level abstract concepts, such as objects in a scene, is still challenging.

While Bayesian inference has been instrumental in the maturation of machine learning, there are also severe limitations to such methods. Specifically, truly Bayesian methods have an unbounded requirement for knowledge as we typically have to sum over all possible outcomes with their likelihood of each event in order to faithfully calculate posteriors. This seems not only excessive in its required knowledge and processing demands, but also faces practical limitations in many applications. An alternative approach is *bounded rationality*, which could be underlying a lot of human decision making [59]. Critical for the success of such methods are fast and frugal heuristics that depend on the environment. Thus there is a major role for learning in this domain on many different scales, including developmental and genetic domains. Understanding learning and development is therefore crucial for scientific reasons as well as technological advancements.

In this chapter, I tried to summarize and relate learning systems that sometimes seem to form different camps. While the application of probability theory made a strong impact on our understanding of learning systems in all camps, there has been some divide between Bayesian modelers, on the one hand, and people in “general” learning machine, on the other hand. The first point out that there is no such thing as a “free lunch”, i.e. general learning machines can never become really good compared to specific models for a particular problem. Yet, finding these specific models can also be a major challenge that must be solved by domain experts. What kind of learner does the brain represent? Many aspects of the brain seem to resemble general learning machines such as the astonishing universality of neocortical architecture. On the other hand, the ability of high-level inference seems at this point out of the reach of such learning machines.

I believe that the brain might be somewhat inbetween, as it represents a *biased learning machine* that already encapsulates specific strategies (learned through evolution and development) in the specific environments typically encountered by the organisms. Such restricted learning machines should be able to support the emergence of Bayesian causal models that could be used by humans to argue about the world. Such models would not only enable smarter applications but would also help us in understanding more deeply the nature of cognition and the mind.

Acknowledgments I would like to express my thanks to René Doursat for careful edits, Christian Albers, Igor Farkas, and Stephen Grossberg for useful comments of an earlier draft circulation, and all the colleagues that have provided me with encouraging comments.

References

1. S. Geman, E. Bienenstock, R. Doursat, Neural networks and the bias/variance dilemma. *Neural Comput.* **4**(1), 1–58 (1992)
2. P. Smolensky, Information Processing in Dynamical Systems: Foundations of Harmony Theory, in *Parallel Distributed Processing: Volume 1: Foundations*, ed. by D.E. Rumelhart, J.L. McClelland (MIT Press, Cambridge, MA, 1986), pp. 194–281
3. G. Hinton, Training products of experts by minimizing contrastive divergence. *Neural Comput.* **14**, 1711–1800 (2002)
4. G. Hinton, A Practical Guide to Training Restricted Boltzmann Machines. University of Toronto Technical Report UTML TR 2010–003, 2010
5. A. Graps, *An Introduction to Wavelets*. <http://www.amara.com/IEEEwave/IEEEwavelet.html>
6. N. Huang et al., The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis. *Proc. R. Soc. Lond. A* **454**, 903–995 (1998)
7. H. Barlow (1961) Possible principles underlying the transformation of sensory messages. *Sens. Commun.* 217–234, (1961)
8. P. Földiák, Forming sparse representations by local anti-Hebbian learning. *Biol. Cybern.* **64**, 165–170 (1990)
9. P. Földiák, D. Endres, Sparse coding. *Scholarpedia* **3**, 2984 (2008)
10. B. Olshausen, D. Field, Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* **381**, 607–609 (1996)
11. H. Lee, E. Chaitanya and A. Ng, Sparse deep belief net model for visual area V2, NIPS*2007
12. C. von der Malsburg, Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik* **14**, 85–100 (1973)
13. S. Grossberg, Adaptive pattern classification and universal recoding, I: Parallel development and coding of neural feature detectors. *Biol. Cybern.* **23**, 121–134 (1976)
14. T. Kohonen, *Self-Organizing Maps* (Springer, Berlin, 1994)
15. P. Hollensen, P. Hartono, T. Trappenberg (2011) Topographic RBM as Robot Controller, JNNS 2011
16. S. Grossberg, Adaptive resonance theory: how a brain learns to consciously attend, learn, and recognize a changing world. *Neural Netw.* **37**, 1–47 (2012)
17. T. Trappenberg, P. Hartono, D. Rasmussen, in Top-Down Control of Learning in Biological Self-Organizing Maps, ed. by J. Principe, R. Miikkulainen. *Lecture Notes in Computer Science* 5629, WSOM 2009 (Springer, 2009), pp. 316–324
18. K. Tanaka, H. Saito, Y. Fukada, M. Moriya, Coding visual images of objects in the inferotemporal cortex of the macaque monkey. *J. Neurophysiol.* **66**, 170–189 (1991)
19. S. Chatterjee, A. Hadi, *Sensitivity Analysis in Linear Regression* (John Wiley & Sons, New York, 1988)
20. Judea Pearl, *Causality: Models, Reasoning and Inference* (Cambridge University Press, Cambridge, 2009)
21. D. Cireşan, U. Meier, J. Masci, J. Schmidhuber, Multi-column deep neural network for traffic sign classification. *Neural Netw.* **32**, 333–338 (2012)
22. D. Rumelhart, G. Hinton, R. Williams, Learning representations by back-propagating errors. *Nature* **323**(6088), 533–536 (1986)
23. K. Hornik, Approximation capabilities of multilayer feedforward networks. *Neural Netw.* **4**(2), 251–257 (1991)
24. A. Weigend, D. Rumelhart (1991) Generalization through minimal networks with application to forecasting, ed. by E.M. Keramidis. in *Computing Science and Statistics (23rd Symposium INTERFACE'91, Seattle, WA)*, pp. 362–370
25. R. Caruana, S. Lawrence, C.L. Giles, Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping, in *Proceedings of Neural Information Processing Systems Conference*, 2000. pp. 402–408
26. D.J.C. MacKay, A practical Bayesian framework for backpropagation networks. *Neural Comput.* **4**(3), 448–472 (1992)

27. D. Silver, K. Bennett, Guest editor's introduction: special issue on inductive transfer learning. *Mach. Learn.* **73**(3), 215–220 (2008)
28. S. Pan, Q. Yang, A survey on transfer learning. *IEEE Trans. Knowl. Data Eng. (IEEE TKDE)* **22**(10), 1345–1359 (2010)
29. B.E. Boser, I.M. Guyon, V. Vapnik, A training algorithm for optimal margin classifiers, in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, (ACM, 1992), pp. 144–152
30. V. Vapnik, *The Nature of Statistical Learning Theory* (Springer, Berlin, 1995)
31. C. Cortes, V. Vapnik, Support-vector networks. *Mach. Learn.* **20**, 273–297 (1995)
32. C. Burges, A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Disc.* **2**(2), 121–167 (1998)
33. A. Smola, B. Schölkopf, A tutorial on support vector regression. *Stat. Comput.* **14**(3) (2004)
34. C.-C. Chang, C.-J. Lin, LibSVM: a library for support vector machines (2001), <http://www.csie.ntu.edu.tw/~cjlin/libsvm>
35. M. Boardman, T. Trappenberg, A heuristic for free parameter optimization with support vector machines, WCCI 2006, pp. 1337–1344, (2006). <http://www.cs.dal.ca/boardman/wcci>
36. E. Alpaydim, *Introduction to Machine Learning, 2e* (MIT Press, Cambridge, 2010)
37. S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics* (MIT Press, Cambridge, 2005)
38. S. Russel, P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd edn. (Prentice Hall, New York, 2010)
39. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, Cambridge, 1998)
40. C.J.C.H. Watkins, Learning from Delayed Rewards. Ph.D. thesis, Cambridge University, Cambridge, England, 1989
41. H. van Hasselt, Reinforcement learning in continuous state and action spaces. *Reinforcement Learn.: Adapt. Learn. Optim.* **12**, 207–251 (2012).
42. R. Sutton, Learning to predict by the methods of temporal differences. *Mach. Learn.* **3**, 9–44 (erratum p. 377) (1988)
43. B. Sallans, G. Hinton, Reinforcement learning with factored states and actions. *J. Mach. Learn. Res.* **5**, 1063–1088 (2004)
44. D.O. Hebb, *The Organization of Behaviour* (John Wiley & Sons, New York, 1949)
45. E.R. Caianiello, Outline of a theory of thought-processes and thinking machines. *J. Theor. Biol.* **1**, 204–235 (1961)
46. T. Trappenberg, *Fundamentals of Computational Neuroscience*, 2nd edn. (Oxford University Press, Oxford, 2010)
47. R. Enoki, Y.L. Hu, D. Hamilton, A. Fine, Expression of long-term plasticity at individual synapses in hippocampus is graded, bidirectional, and mainly presynaptic: optical quantal analysis. *Neuron* **62**(2), 242–253 (2009)
48. T. Bliss, T. Lømo, Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *J. Physiol.* **232**(2), 331–56 (1973)
49. D. Heinke, E. Mavritsaki (eds.), *Computational Modelling in Behavioural Neuroscience: Closing the gap between neurophysiology and behaviour* (Psychology Press, London, 2008)
50. R. Rescorla, A. Wagner, in *A Theory of Pavlovian Conditioning: Variations, in the Effectiveness of Reinforcement and Nonreinforcement*, ed. by W.F. Prokasy, A.H. Black, Classical Conditioning, II: Current Research and Theory, (Appleton Century Crofts, New York, 1972), pp. 64–99
51. W. Schultz, Predictive reward signal of dopamine neurons. *J. Neurophysiol.* **80**(1), 1–27 (1998)
52. J. Houk, J. Adams, A. Barto in *A Model of How the Basal Ganglia Generate and Use Neural Signals that Predict Reinforcement*, ed. by J.C. Houk, J.L. Davis, D.G. Breiser. Models of Information Processing in the Basal Ganglia (MIT Press, Cambridge, 1995)
53. P. Connor, T. Trappenberg, in *Characterizing a Brain-Based Value-Function Approximator*, ed. by E. Stroulia, S. Matwin, Advances in Artificial Intelligence LNAI 2056, (Springer, Berlin, 2011), pp. 92–103

54. J. Reynolds, J. Wickens, Dopamine-dependent plasticity of corticostriatal synapses. *Neural Netw.* **15**(4–6), 507–521 (2002)
55. P. Connor, V. LoLordo, T. Trappenberg (2012) An elemental model of retrospective revaluation without within-compound associations. *Anim. Learn.* **42**(1), 22–38
56. T. Maia, M. Frank, From reinforcement learning models to psychiatric and neurological disorders. *Nat. Neurosci.* **14**, 154–162 (2011)
57. Y. Bengio, Learning deep architectures for AI. *Found. Trends Mach. Learn.* **2**, 1–127 (2009)
58. J. Hawkins, *On Intelligence* (Times Books, New York, 2004)
59. G. Gigerenzer, P. Todd and the ABC Research Group, *Simple Heuristics that Make Us Smart* (Oxford University Press, Oxford, 1999)

Growing Adaptive Machines

Combining Development and Learning in Artificial Neural
Networks

Kowaliw, T.; Bredeche, N.; Doursat, R. (Eds.)

2014, VII, 261 p. 82 illus., 14 illus. in color., Hardcover

ISBN: 978-3-642-55336-3