Introduction to Artificial Neural Networks and Deep Learning (FYTN14/EXTQ40/NTF005F)

Mattias Ohlsson and Patrik Edén
Computational Biology and Biological Physics
Department of Astronomy and Theoretical Physics
Lund University

Fall 2021

Preface

This course is intended to give an introduction to artificial neural networks and deep learning. Artificial neural networks (ANN) represents a technology that has connections to many disciplines such as neuroscience, computer science, mathematics, statistics and physics. This course will provide you with enough knowledge to use neural networks and deep learning in practical applications.

We will study ANNs from both a theoretical and a computational/practical point of view. The course material is divided into different parts:

1. Introduction
2. Feed-forward neural networks
3. Generalization to unseen data
4. CNN, Autoencoder and GAN
5. Recurrent neural networks
6. Self-organizing neural networks

There are three computer exercises in this course. The exercises are application oriented, meaning that we will focus on the methods/application and not so much on the programming part. The amount of programming required is therefore limited.

These notes are meant as a complement to the lectures. The lectures will also use material from the "Deep Learning Book" (DLB) found at www.deeplearningbook.org. Additional material, such as lecture videos and math problems, will be provided during the course.

Chapter 1

Introduction

Read the introduction chapter of the "Deep Learning Book" (DLB)[1].

1.1 Introduction to machine learning

From chapter 5 in the DLB book, read the following sections:

- "Intro"
- 5.1 Learning Algorithms
- 5.2 Capacity, Overfitting and Underfitting (pages 108-114)
- 5.11.1 The Curse of Dimensionality

One of the most important features of artificial neural networks (ANN) is the ability to learn from data. The DLB talks about three ingredients, the experience E, the task T and the performance measure P.

- The task. There are many tasks in machine learning, such as *classification, regression, sequence prediction, imputation of missing values* etc, where classification problems are very popular.
- The performance measure. We need to evaluate the performance of the machine learning algorithm. The performance measure is often specific to the task. One also can have different performance measures during training and when the model is actually used. During training one often uses a "loss function" instead of a performance measure.

[1] www.deeplearningbook.org

- The experience. To simplify we can say that the experience often comes as a dataset, *i.e.,* a collection of examples.

Learning can broadly be divided into two categories, **unsupervised** and **supervised** learning. The data representing the "experience" is usually a set of data points. Sometimes each data point comes with a label (eg. an image of a handwritten digit can have label "five"). For supervised learning we make use of both data points and labels, but for unsupervised learning we only use the data points, so the data need not provide labels.

There are other learning processes, such as semi-supervised learning, multi-instance learning, one-shot learning, reinforcement learning. In this course we will only look at supervised and unsupervised learning.

1.2 What is an artificial neural network?

Artificial neural networks (ANN) is a collective name for many algorithms that (almost) all are inspired by the construction and functioning of the human brain. Haykin[2] defines an artificial neural network as:

A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experimental knowledge and making it available for use. It resembles the brain in two aspects:

1. *Knowledge is acquired by the network from its environment through a learning process.*
2. *Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.*

Note that an ANN is by no means a simulation of the whole or some part of the brain. Still, a comparison of the biological neuron and the "node" of an ANN illustrates how biology has inspired the field of ANNs.

1.2.1 The biological neuron

The neuron can be seen as the elementary computational node of our brain. There are many different types of neurons, where a typical so called pyramidal cell is shown in figure 1.1.

[2]Simon Haykin, "Neural Networks: A Comprehensive Foundation", (1998)

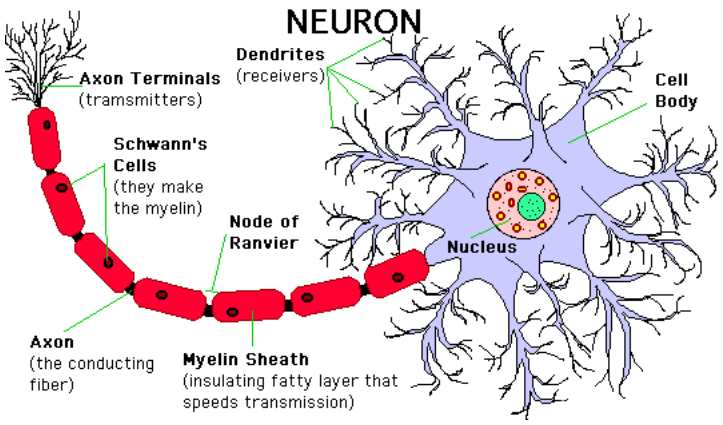


Figure 1.1: Illustration of a *pyramidal* cortical neuron.

This neuron consists of several parts. In a simple description they are:

- The cell body: Here is the cell nucleus located.
- Dendrites: Tree like constructions are connected to the cell body. They can be viewed as receivers of signals from other neurons.
- Axon: Extending from the cell body is a long nerve fiber called the axon. The axon can be viewed as the transmitter of generated signals from the cell body.
- Synaptic terminals (synapses): The axon eventually branches off and forms a tree of transmitter ends called synaptic junctions or synapses. The connections to other receivers are dendrites or the cell bodies themselves. The strength of the synapses can vary and the connections can be both inhibitory and excitatory.

When the neuron receives strong enough signals (within some small amount of time) from its dendrites, a reaction starts that leads to a signal being sent out through its axon, influencing other neurons.

The fact that

- we have about 10^{11} neurons in our brain
- there are about 10^3 connections per neuron giving a total of about 10^{14} synapses in our brain
- signals travel up to 120 m/s in humans

makes our brain is a highly complex, non-linear and parallel information-processing system!

1.2.2 The ANN node – models of a neuron

Figure 1.2 shows the basic element of an ANN, the artificial neuron. The output of the

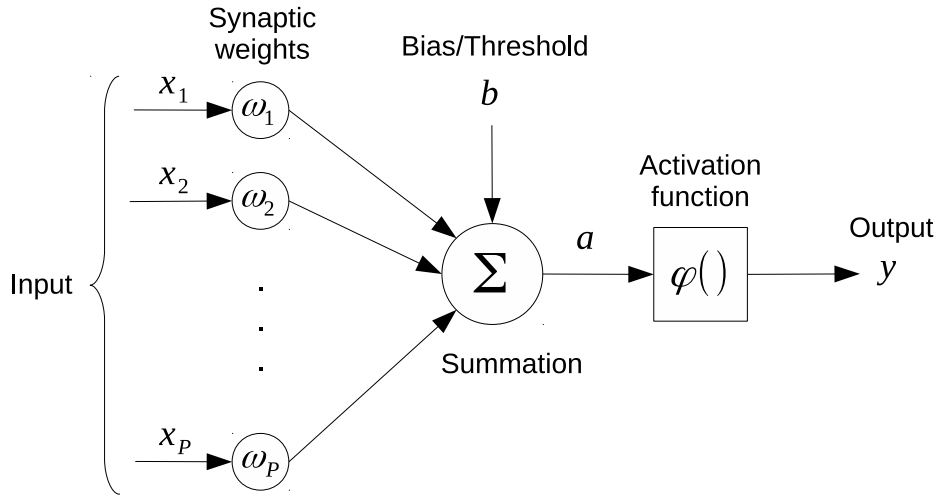


Figure 1.2: The basic element of the ANN, the neuron

neuron is calculated as

$$a = \sum_{k=1}^{K} \omega_k x_k + b$$
$$y = \varphi(a)$$

Neurons do a weighted summation of its incoming signals and outputs a new signal. The weights ω_k can be both positive and negative. The function $\varphi(a)$ is called activation function, but also has other names, *e.g.,* transfer function or node function. The function is often inspired by biology, where increased input gives larger output, and where there is a threshold behaviour for some critical input level. Fig. (1.3) shows some popular choices. They are:

Linear,	$\varphi(a) = a$
Threshold or "step",	$\varphi(a) = \theta(a) = \begin{cases} 0, & a < 0 \\ 1, & a \geq 0 \end{cases}$
Logistic,	$\varphi(a) = \frac{1}{1+e^{-a}}$
Hyperbolic tangent,	$\varphi(a) = \tanh(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)}$
Rectifier function,	$\varphi(a) = \max(0, a)$
Softplus,	$\varphi(a) = \log(1 + e^a)$

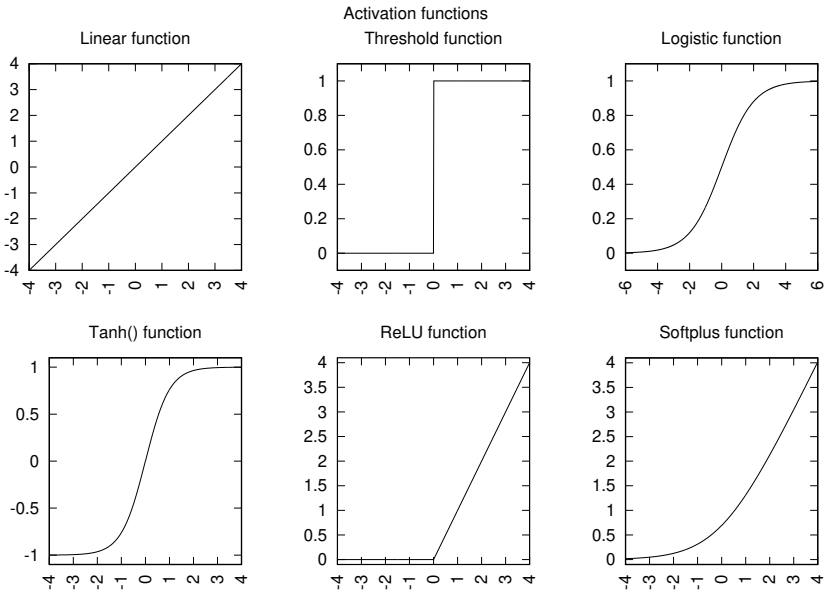


Figure 1.3: Examples of activation functions.

1.2.3 Network architectures

The way we connect nodes together is called the network architecture, or network topology. There are two major groups:

1. Feed-forward architectures
2. Feed-back architectures

Feed-forward networks

In feed-forward networks there are no closed loops of signal transfer. Feed-forward networks are often strictly structured to let input values propagate forward through layers of nodes to an output layer, as illustrated in fig. (1.4). We can view the input as a "question", and the output as the ANN "answer".

The structure does not have to be so strict. There can be partially connected layers, instead of the fully connected shown in fig. (1.4), and there can be "skip-layer links", directly connecting nodes of distant layers. As long as there are no feed-back loops, it is a feed-forward architecture.

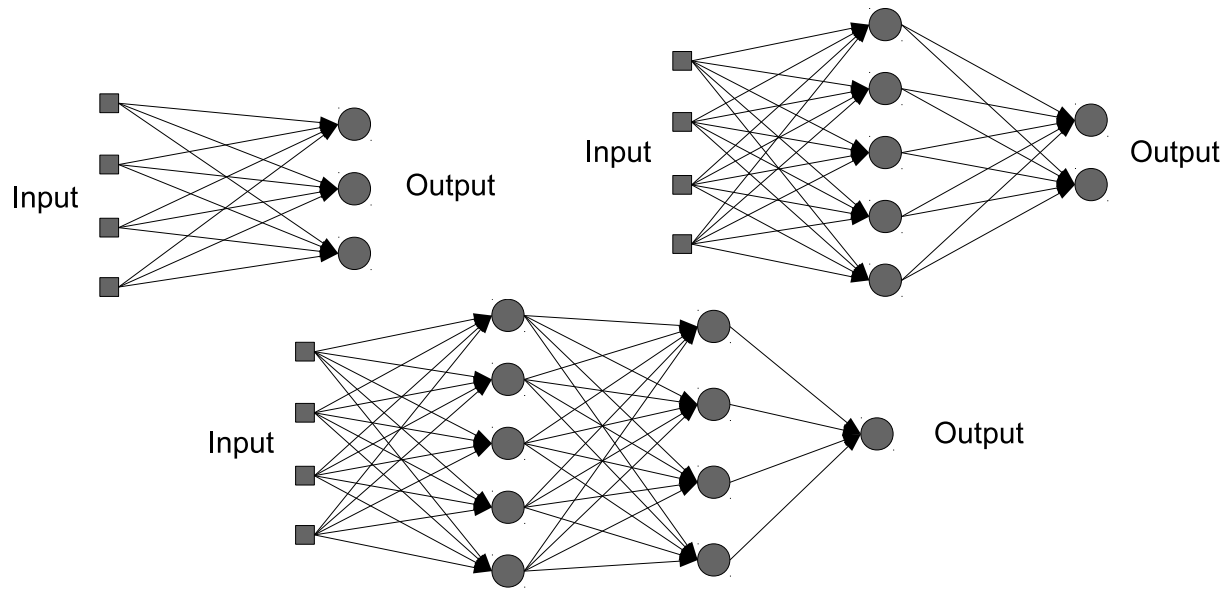


Figure 1.4: Feed-forward networks with one, two and three layers of computational nodes, respectively.

Recurrent (feed-back) networks

Fig. (1.5) shows examples of networks with feed-back connections. They are often used to model sequence data. In a fully recurrent network there are no clear inputs and outputs. Instead, the initial values of the nodes can be viewed as the "question", and the steady state that the network eventually reaches becomes the "answer".

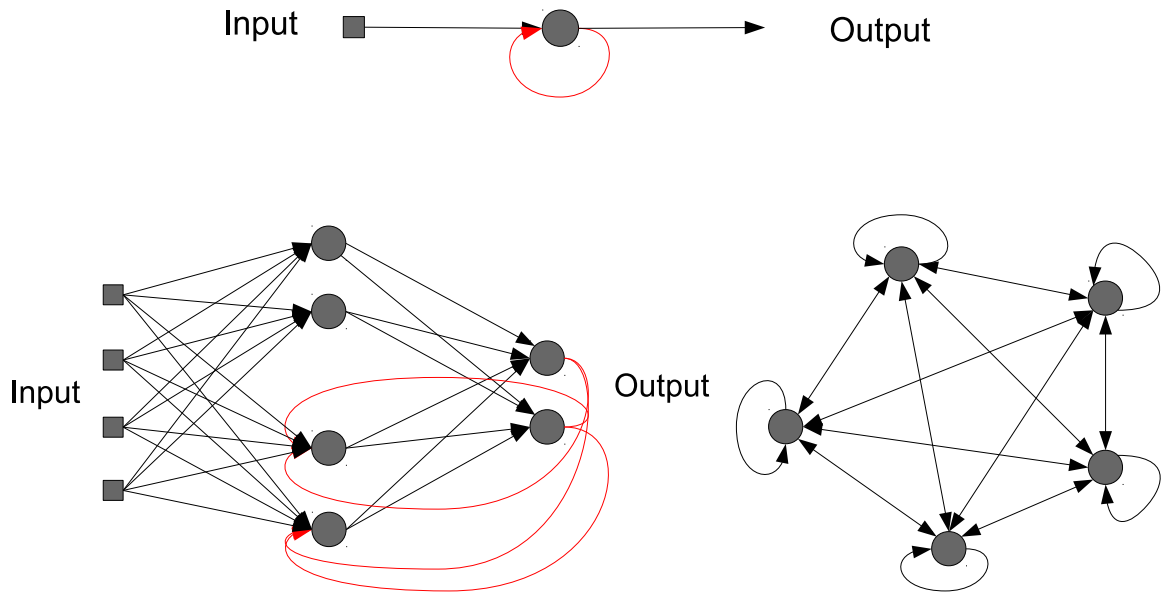


Figure 1.5: (Top) A single node with a feed-back connection (red). (Lower left) A neural network with mainly feed-forward and a few feed-back connections (red). The bottom two nodes in the first layer receives inputs from the two output nodes. (Lower right) A fully recurrent network with 5 nodes.

Chapter 2

Feed-forward Neural Networks

2.1 The typical tasks

Feed-forward networks are designed to provide an output function $y(\boldsymbol{x})$ of inputs $\boldsymbol{x}$. The task is typically represented by a dataset of patterns n, with inputs $\boldsymbol{x}_n$ and target d_n. The goal of the network is to get the outputs equal to the targets $y(\boldsymbol{x}_n) = d_n, \forall n$.

The output y also depends on all the weights and biases for different nodes. We represent all those parameters with a symbol $\boldsymbol{\omega}$. The goal is to find the $\boldsymbol{\omega}$ that solves the task, or at least solves it "as well as possible". This vague goal is represented mathematically by a so-called *loss function* $E(\boldsymbol{\omega})$ that compares outputs $y_n = y(\boldsymbol{\omega}, \boldsymbol{x}_n)$ to targets d_n.

We have great freedom in selecting a loss function that represents our own opinion about "good" and "bad" solutions, but there are standard choices that have many benefits.

2.1.1 Regression

If the targets are samples from a continuous distribution, it is customary to have a linear activation function,

$$\varphi(a) = a,$$

since it is simple and un-bounded, and combine it with a loss based on the mean squared error (MSE),

$$E(\boldsymbol{\omega}) = \frac{1}{2N} \sum_{n=1}^{N} (y_n - d_n)^2. \tag{2.1}$$

Here, N is the total number of patterns in the dataset, and the sum runs over all patterns. Minimizing this loss is then the same as solving a least squares regression problem. Since $y_n = y(\boldsymbol{\omega}, \boldsymbol{x}_n)$, the loss function depends on the network parameters $\boldsymbol{\omega}$.

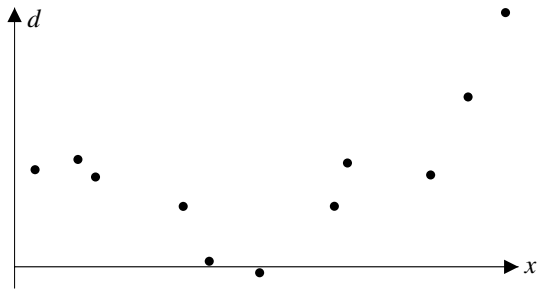


Figure 2.1: A regression task: data with a continuous target d as function of an input x.

Every now and then, you will hear and read $E(\boldsymbol{\omega})$ referred to as the "mean squared error", or MSE, desipte the factor 1/2. That factor has no effect on minimization.

2.1.2 Binary classification

In binary classification, the task is to classify each input pattern n into one of two classes C_1 or C_2, as shown in fig. (2.2). We can then define a target d_n as

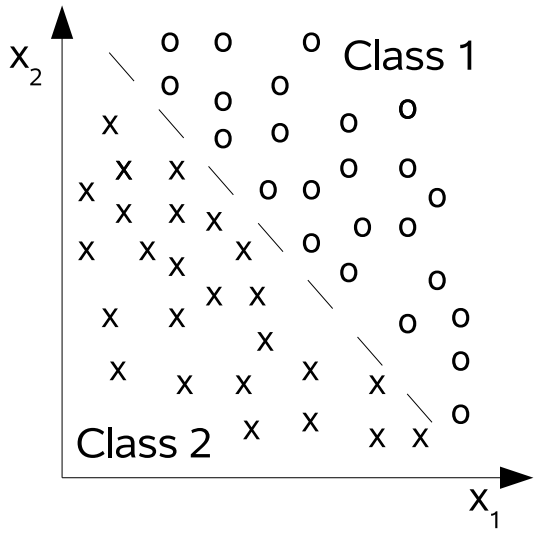


Figure 2.2: Data points represented by two coordinates (x_1 and x_2). There are two classes in this classification problem, denoted by 'o' and 'x' symbols.

$$d_n = \begin{cases} 1, & n \in C_1 \\ 0, & n \in C_2 \end{cases} \tag{2.2}$$

Historically, the first attempts to solve classification tasks with ANNs used the threshold activation function. However, to handle the situation where the ANN is unable to solve the task perfectly, it is preferable to use the logistic function $\varphi(a) = 1/(1 + \exp(-a))$. The output y_n is then confined to the range $(0, 1)$, We will later see that if we also use the "cross entropy error" (CEE) as loss function,

$$E(\boldsymbol{\omega}) = -\frac{1}{N} \sum_n \left[d_n \ln(y_n) + (1 - d_n) \ln(1 - y_n) \right], \tag{2.3}$$

then the output can be interpreted as the probability that $d_n = 1$. We can already now see that the loss represents "bad solution": each pattern n contributes with a $-\ln(y_n)$ if $d_n = 1$, and $-\ln(1 - y_n)$ if $d_n = 0$. So, if all y_n are correct, the loss is zero, and the loss will increase greately if any y_n is close to the wrong target value.

2.1.3 Multiple targets

Multi-class problems are also common. A bench-mark dataset is MNIST, which provides images of hand-written digits. Each image is 28×28 pixels, which gives 784 input variables, and belongs to one of 10 classes, represented by the digits 0-9. We will discuss suitable activation functions and loss functions for multiclass problems later on.

The dataset could in principle provide multiple targets that are not a single multi-class task. Perhaps d_{n1} and d_{n3} are two continuous properties for pattern n, while d_{n2} specifies a binary class. This is fairly rare, and it is possible to simply create one network for each target, but you can also have multiple outputs, each with an activation function suitable for its target, and simply add suitable loss functions together. That involves a subjective weighting of losses for different targets. Maybe it is crucial to get d_{n1} right while d_{n3} is less important?

2.2 The perceptron

The simple perceptron has no hidden layer, only an input layer and one output node. It is illustrated in fig. (1.2). Given an input vector $\boldsymbol{x}$ we can then compute the output y,

$$y(\boldsymbol{x}, \boldsymbol{\omega}, b) = \varphi_o(\sum_{k=1}^{K} \omega_k x_k + b) = \varphi_o(\boldsymbol{\omega}^T \mathbf{x} + b). \tag{2.4}$$

2.2.1 How does the perceptron work?

The argument a to φ_o depends on the input vector $\mathbf{x}$ only through a scalar product $\boldsymbol{\omega}^T\mathbf{x}$. Thus, the contour surfaces of a become parallel hyperplanes in $\mathbf{x}$-space. The direction of $\boldsymbol{\omega}$ defines the orientation of the contour surfaces, and the magnitude of $\boldsymbol{\omega}$ determines the distance between surfaces for two different values of a. Finally, the bias b determines which contour surface that corresponds to $a = 0$. This is illustrated for a simple 2-dimensional case in fig. (2.3).

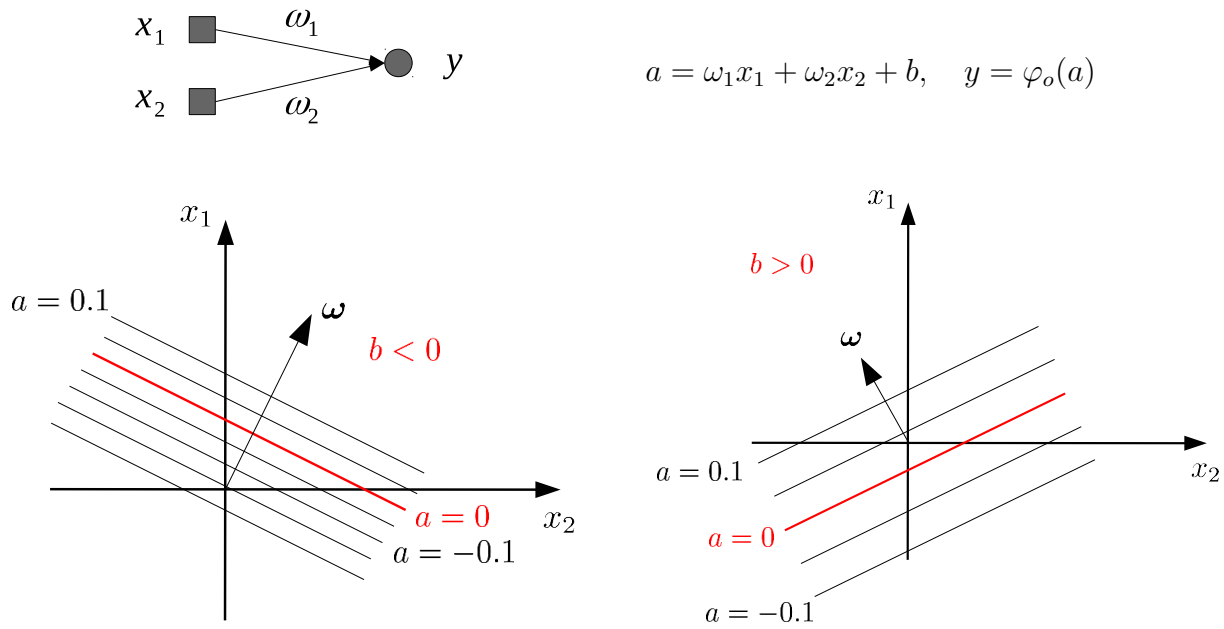


Figure 2.3: The countour surfaces for the output argument a. Large $\boldsymbol{\omega}$ and negative bias to the left, small $\boldsymbol{\omega}$ and positive bias to the right.

This is of course not the only possible way to define contour surfaces for a (see for example the part on "radial basis functions" later). However, one appealing property of neural networks is that in order to solve more complex tasks, you do not need to make your nodes more complicated – instead you add more nodes. We will therefore restrict our discussion to what very simple perceptrons can do.

It is common to add an extra "bias input" $x_0 = 1$ to all samples, and represent the bias parameter $b = \omega_0$ as a weight, as shown in fig. (2.4). If so, ω_0 and x_0 are absorbed into the vecctors $\boldsymbol{\omega}$ and $\mathbf{x}$, respectively, and the output function is written

$$y(\mathbf{x}, \boldsymbol{\omega}) = \varphi_o(\sum_{k=1}^{K} \omega_k x_k + \omega_0) = \varphi_o(\sum_{k=0}^{K} \omega_k x_k) = \varphi_o(\boldsymbol{\omega}^T\mathbf{x}). \tag{2.5}$$

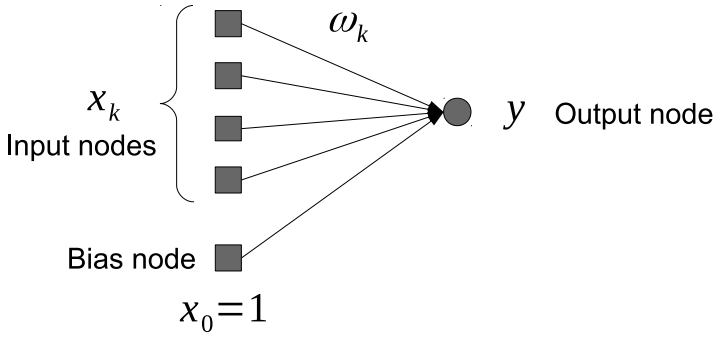


Figure 2.4: Including the bias term as an extra weight ω_0. All patterns n have $x_{n0} = 1$.

Be mindful that the vectors $\mathbf{x}$ and $\boldsymbol{\omega}$ therefore may have slightly different meaning in different context, for example if bias terms are explicitly added as arguments to activation functions or not.

Linear regression

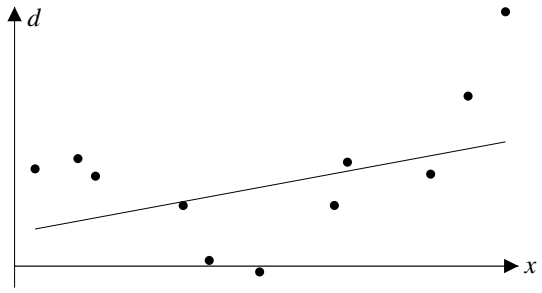


Figure 2.5: Illustration of a fit with linear regression.

For a regression problem, with output activation $\varphi_0(a) = a$, the perceptron output is simply a linear combination of the inputs, $y = \boldsymbol{\omega}^T\mathbf{x}$ (including bias). Minimizing the sum of squares in eq. (2.1) corresponds to *linear regression*. Before we discuss how our perceptron could solve this, we note that there is an exact solution. The loss has a stagnation point if all $\frac{\partial E}{\partial \omega_k} = 0$, which implies

$$0 = \frac{1}{N}\sum_{n=1}^{N}(y(\mathbf{x}_n) - d_n)\frac{\partial y(\mathbf{x}_n)}{\partial \omega_k} = \frac{1}{N}\sum_{n=1}^{N}(\sum_{i=0}^{K} x_{ni}\omega_i - d_n)x_{nk}, \quad \forall k. \tag{2.6}$$

If we treat x_{nk} as elements in a matrix $\mathbf{X}$, while d_n and ω_k are elements in column matrices $\mathbf{d}$ and $\boldsymbol{\omega}$, respectively, this can be written as $0 = \mathbf{X}^T\mathbf{X}\boldsymbol{\omega} - \mathbf{X}^T\mathbf{d}$. The solution can most

often be found as

$$\boldsymbol{\omega} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{d}.$$

Remark 1: The matrix $\frac{1}{N}\mathbf{X}^T\mathbf{X}$ is called the *correlation matrix* for the inputs $\mathbf{x}_n$.
Remark 2: $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ is called the *pseudo inverse* of $\mathbf{X}$. Typically, $\mathbf{X}$ itself does not have an inverse (it need not even be square).
Remark 3: If $\mathbf{X}^T\mathbf{X}$ lacks an inverse, there are still solutions, but they are not unique. Finding one of them typically involves diagonalizing $\mathbf{X}^T\mathbf{X}$.

Linearly separable classification

In binary classification, the target values d_n only take on values 0 or 1, and the threshold function or logistic function are two suitable output activations. The contour surfaces for these functions are illustrated in fig. (2.6). In the limit of infinite $|\boldsymbol{\omega}|$, the logistic function becomes the threshold function, with the sensitive region compressed to a single decision plane.

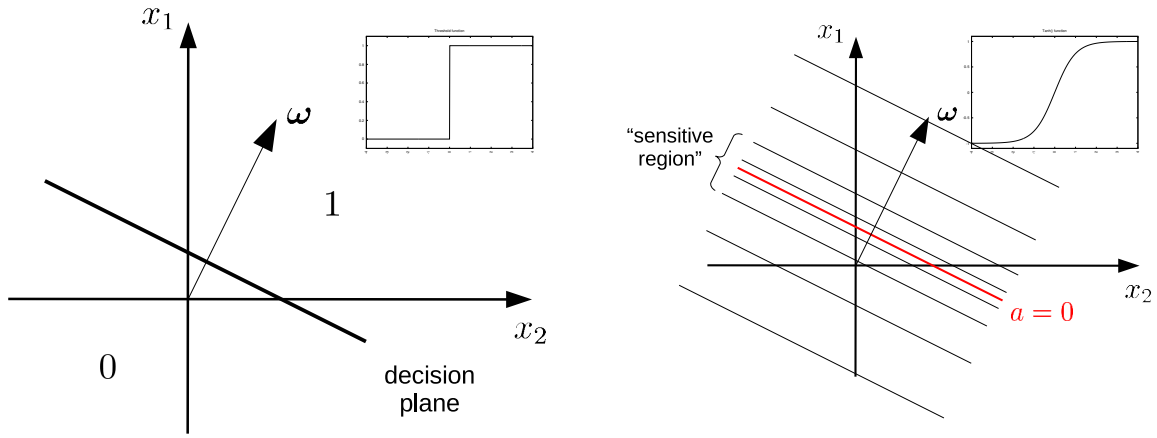


Figure 2.6: Left: For a threshold function, the are no countour surfaces, but a "decision plane" at $a = 0$. Right: The contour surfaces for the logistic node are close to each other for $a \approx 0$, where the function is sensitive to the argument. Further away the output is essentially constant.

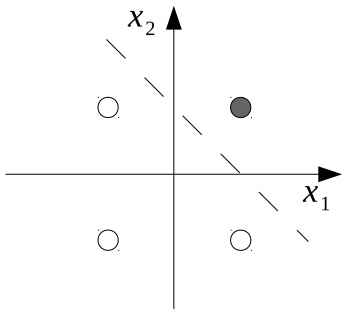
We say that a classification problem is *linearly separable* if we can find a weight vector $\boldsymbol{\omega}$ such that the threshold function $\theta(a)$ solves

$$\theta(\boldsymbol{\omega}^T\mathbf{x}_n) = d_n \quad \forall n$$

which can be rephrased as: A problem is linearly separable if we can find a hyperplane that separates the two classes.

A classical example of what the simple perceptron can handle is the 2-dimensional AND problem. In two dimensions it is defined by

x_1	x_2	d
1	1	1
1	-1	0
-1	1	0
-1	-1	0



A perceptron that solves this problem is given by $(\omega_1, \omega_2, \omega_0) = (1, 1, -1)$ (see fig. (2.7)).

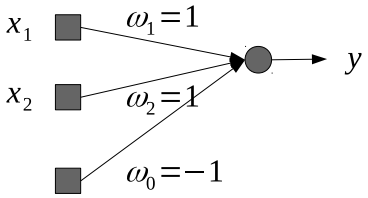
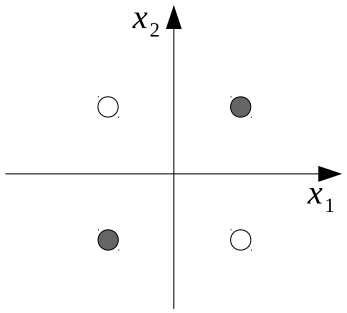


Figure 2.7: A perceptron that solves the 2-dimensional AND problem.

A problem that *cannot* be solved by a single perceptron is the XOR problem.

x_1	x_2	d
1	1	0
1	-1	1
-1	1	1
-1	-1	0



If we try a threshold function $y = \theta(x_1\omega_1 + x_2\omega_2 + \omega_0)$, we get the following set of equations

d		
0	$\omega_1 + \omega_2 + \omega_0 < 0$	(1)
1	$\omega_1 - \omega_2 + \omega_0 > 0$	(2)
1	$-\omega_1 + \omega_2 + \omega_0 > 0$	(3)
0	$-\omega_1 - \omega_2 + \omega_0 < 0$	(4)

Adding expression (2) and (3) results in $\omega_0 > 0$ but the sum of expressions (1) and (4) gives $\omega_0 < 0$. So, the XOR problem is not linearly separable.

2.2.2 Gradient descent learning

With our loss function $E(\boldsymbol{\omega})$, the task has become an optimization problem. We want to find weights (and biases) $\boldsymbol{\omega}$ that minimize the loss. If we abandon the threshold function, it is possible to differentiate all proposed losses and activation functions. Then we can find the derivatives

$$\frac{\partial E}{\partial \omega_k} = \sum_n \frac{\partial E}{\partial y_n} \cdot \frac{\partial y_n}{\partial a_n} \cdot \frac{\partial a_n}{\partial \omega_k} = \sum_n \frac{\partial E}{\partial y_n} \varphi'(a_n) x_{nk}. \tag{2.7}$$

One strategy is therefore to start with randomly initialized weights and improve by taking small steps in the negative gradient direction $\Delta\boldsymbol{\omega} = -\eta \nabla_{\boldsymbol{\omega}} E$, as illustrated in fig. (2.8). The "learning rate" η is a positive parameter, that we need to define.

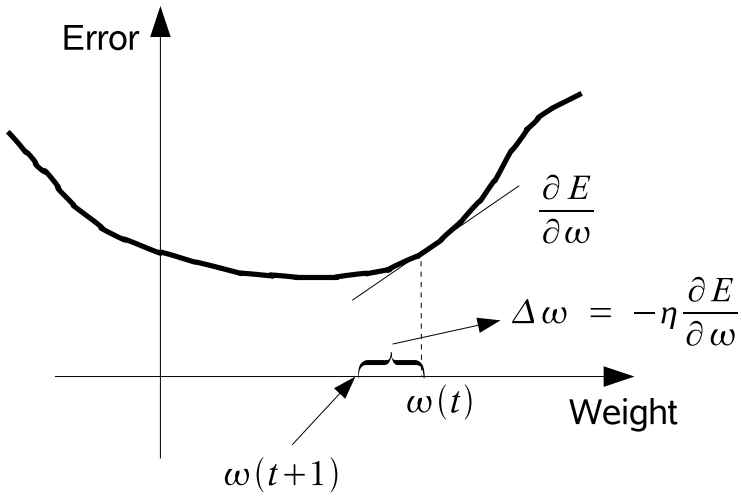


Figure 2.8: Illustration of the gradient descent method.

This *gradient descent* algorithm can be written as in fig. (2.9)

As seen, the derivatives of activation functions are needed, and since the node values are already known after calculation of the output, it is useful to express the derivatives in terms of the function value itself. For example, for the logistic function we have $\varphi'(a) = \ldots = \varphi(a)[1 - \varphi(a)]$. In that case, we get $\varphi'_o(a_n) = y_n(1 - y_n)$.

When the loss is of the form $E = \frac{1}{N}\sum_n E_n$, then $\frac{\partial E}{\partial y_n}$ can be easily calculated as soon as y_n is known, since $\frac{\partial E}{\partial y_n} = \frac{1}{N}\frac{\partial E_n}{\partial y_n}$ is independent of all other patterns. This is the case for the

1. Initiate all ω_k with small random numbers.
2. Define a small "learning rate" η.
3. Repeat until convergence
   (a) For each pattern n, compute the output
   $y_n = y(\mathbf{x}_n)$
   and the derivative
   $\delta_n = -\frac{\partial E}{\partial a_n} = -\frac{\partial E}{\partial y_n}\varphi_0'(a_n)$.
   (b) Update the weights ω_k according to
   $\omega_k \to \omega_k + \eta \sum_n \delta_n x_{nk}$.

Figure 2.9: Gradient descent learning.

MSE-based loss for the cross entropy loss.

Another nice feature of the standardloss and activation functions is that the derivatives $\frac{\partial E}{\partial a_n}$ actually become the same:

	Regression	Classification
E	MSE-based	cross entropy
$\frac{\partial E}{\partial y_n}$	$\frac{1}{N}(y_n - d_n)$	$-\frac{1}{N}\left[\frac{d_n}{y_n} - \frac{1-d_n}{1-y_n}\right] = \frac{1}{N}\frac{y_n - d_n}{y_n(1-y_n)}$
output function	linear	logistic
$\frac{\partial y_n}{\partial a_n} = \varphi_o'(a_n)$	1	$y_n(1-y_n)$
$\delta_n = -\frac{\partial E}{\partial y_n}\frac{\partial y_n}{\partial a_n}$	$\frac{1}{N}(d_n - y_n)$	$\frac{1}{N}(d_n - y_n)$

Thus, if we stick to the standard functions, the quantity δ_n in the gradient descent method above will always be $\frac{1}{N}(d_n - y_n)$.

2.2.3 Data pre-processing and perceptron initialization

To get started with gradient descent training, we must decide how to initialize random weights, and what learning rate we should use. The best strategy may depend on the task,

and therefore we will give a suggestion that involves some simple data transformation.

Input normalization

Usually we want to normalize the input values. Strictly this is not necessary since we can always rescale the weights to deal with different input ranges, but practically it is always a good idea. Consider the inputs x_1 and x_2 and their distributions in figure 2.10.

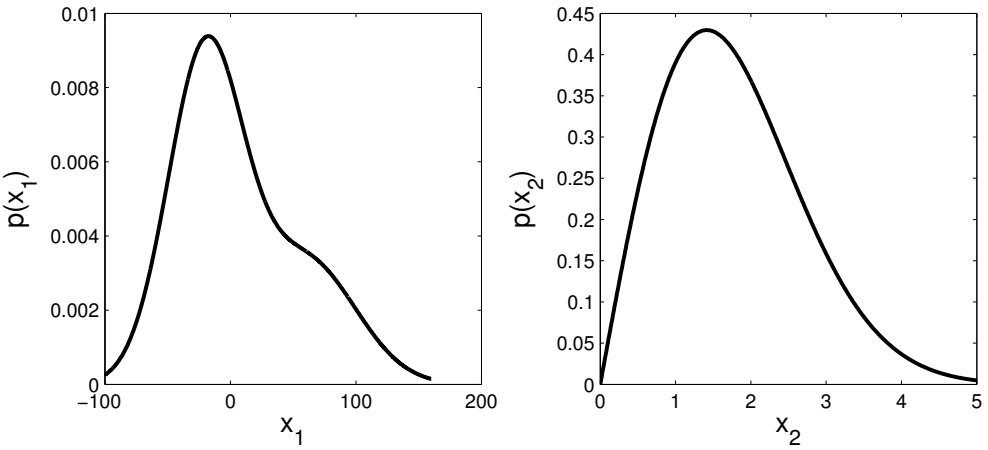


Figure 2.10: Distribution of input vectors x_1 and x_2.

If we start with small random weights and then minimize the error function it will be difficult to see the effect of the x_2 inputs since, they will "drown" in the x_1 input. To avoid having to initiate the weights different for different inputs it is always better to normalize the inputs and have a common procedure for the weight initialization. There are many ways of doing input normalization, but the most common one is rescaling to zero mean and unit standard deviation for each input variable:

Compute the sample mean and (some estimate of) the standard deviation

$$\mu_k = \frac{1}{N}\sum_{n=1}^{N} x_{nk}$$

$$\sigma_k = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(x_{nk} - \mu_k\right)^2}$$

Do the transformation

$$x_{nk} \leftarrow \frac{x_{nk} - \mu_k}{\sigma_k} \quad \forall\ n, k$$

For binary inputs we make sure the two values are small integers, typically 0 and 1, or ± 1. We avoid strange values like 100 and 2000, even if they represent something for the creator of the data set.

Initial weight values

If we have performed a linear mapping of inputs to zero mean and unit variance, it is a good idea to initalize the weights from a distribution with zero mean and variance $1/K$, where K is the number of inputs. We then have typical magnitudes $|x_{nk}| \sim 1$ and $|\omega_k| \sim 1/\sqrt{K}$, so that $\sum_{k=1}^{K} \omega_k x_{nk}$ can be seen as a random walk with terms of typical size $1/\sqrt{K}$. Since we add K such terms, the typical sum is about 1.

If we now let the distribution for a random initial bias term b to have mean 0 and variance about 1, we can expect that many samples will have an argument to the output function, $a_n = \sum_{k=1}^{K} \omega_k x_{nk} + b$, that will be of order 1.

Target normalization and learning rate

It is a good idea to also normalize continuous target values to zero mean and unit variance, and to let binary target values take on values 0 and 1. Then, we can expect a good solution to the task to have $\mathcal{O}(1)$ outputs. As can be seen in fig. (1.3), all the output functions we consider will then need $\mathcal{O}(1)$ arguments, which we have achieved with our input transformation and weight selection.

With all these decisions, it seems that weight updates $\Delta\omega_k$ noticeably smaller than 1 are a good idea, while gradient terms $\delta_n x_{nk}$ can be expected to (at least in the beginning) be $\mathcal{O}(1)$. A good learning rate should then be smaller than 1, say in the range 0.01-0.1, or so. This may need to be checked with some early trial-and-error. If the error during Gradient Descent wobbles drastically between updates, the learning rate seems high. If the error reduction is steady but annoyingly slow, it seems low. In either case, we can redo training with a new learning rate.

Non-linear transformations

If some data distributions are heavily skewed, data values may become large even after transformation to zero mean and unit variance. Then, one may consider a non-linear transformation. Fig. (2.11) shows an example of the effect of a simple log transformation of a skewed input variable.

Notice that if you use a perceptron for regression, but perform nonlinear transformations first, you are formally no longer performing linear regression on the original data!

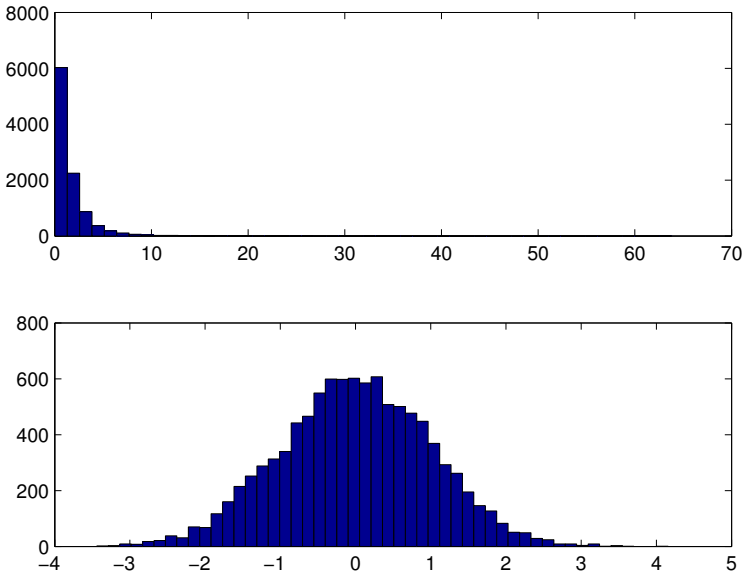


Figure 2.11: Upper plot. The original input variable distribution. Lower plot. The new distribution after a log() operation.

Non-linear preprocessing of input data is **very** problem dependent. Sometimes preprocessing is a very large part of working with a machine learning problem, sometimes it is enough with a simple normalization step.

Missing values

A very common problem with real world data is *missing values*. This means that some of the input variables are missing for a part of the data set. Since the network need input values for each input node, one needs to replace the missing value with something new. The procedure of replacing missing values is called *imputation*. There are many different ways of doing missing data imputation. Here are a few alternatives:

- Mean or median value: Missing values are constructed using only the variable itself. For binary variables, the mean may not make much sense, while the median simply becomes the majority value.
- Random imputation: Missing values are replaced by random picks from the known values of the variable itself. With this approach we can actually create many different datasets, each with a specific random imputation of missing values. That can allow

estimates of how sensitive the result is to imputation, and may for example lead to a decision to omit the input variable instead.

- More elaborate estimate: This means that the missing values is estimated using information from other input variables. One approach is kNN-impute (k nearest neighbour impute)[1], where the imputed value is the mean of the k patterns "closest" to the pattern with the missing value. Distance can be euclidean, or based on correlations. Another possibility is to use an autoencoder network to impute missing values. (We will talk more about autoencoders later.) Many elaborate estimates need complete data as well, so all "NaN" are replaced with a simple imputation, *e.g.*, mean values. Then the elaborate method is used to update the values. If computational time allows, the process can be iterated to convergence.

2.3 The multilayer perceptron (MLP)

2.3.1 The MLP architecture

The multilayer perceptron is constructed by adding one or more hidden layers of nodes. Before Deep Learning became popular, one or possibly two hidden layers was most common (see figure 2.12).

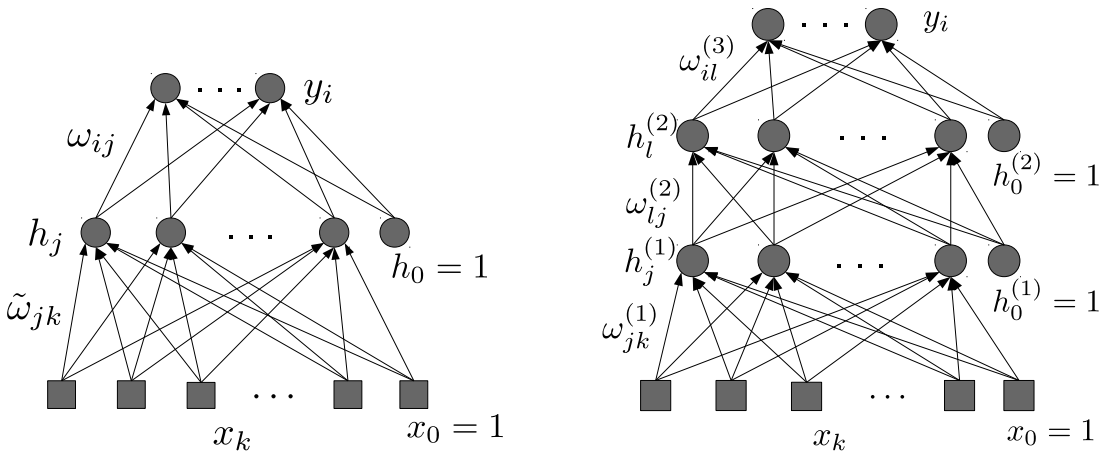


Figure 2.12: (left) A standard MLP with one hidden layer. (right) Two hidden layers.

[1] Troyanskaya et al., Missing Value Estimation Methods for DNA Microarrays, Bioinformatics 17(6) p 520 (2001)

The activation functions for the hidden nodes should be non-linear. Otherwise, the argument to the output would become a linear combination of linear combinations, which is just one big linear combination, possible to represent with a simple perceptron. For networks with a few number of hidden layers the $\varphi(a) = \tanh(a)$ is often used, while the rectifier function $\varphi(a) = \max\{0, a\}$ is the most popular for deeper networks. Most common is to have the same activation function for all nodes in the same layer. However, if there are multiple output nodes, each node has the activation function adapted to its type of target, as discussed for the perceptron.

It is common to add a bias node to each hidden layer, with value 1 for all patterns. Then the bias parameters can be represented as weights on a link between two adjacent layers (see figure 2.12).

Assuming a single hidden layer, the output $y_i(\mathbf{x}_n)$, given an input pattern $\mathbf{x}_n$, is calculated through a forward pass

$$\begin{aligned} y_i(\mathbf{x}_n) &= \varphi_o\Big(\sum_j \omega_{ij} h_{nj}\Big) = \varphi_o\Big(\boldsymbol{\omega}_i^T \mathbf{h}_n\Big) \quad \text{where} \\ h_{n0} &= 1, \\ h_{nj} &= \varphi_h\Big(\sum_k \tilde{\omega}_{jk} x_{nk}\Big) = \varphi_h\Big(\tilde{\boldsymbol{\omega}}_j^T \mathbf{x}_n\Big), j > 0. \end{aligned}$$

In one expression,

$$y_i(\mathbf{x}_n) = \varphi_o\Big(\sum_{j=1} \omega_{ij} \varphi_h\Big(\sum_{k=0} \tilde{\omega}_{jk} x_{nk}\Big) + \omega_{i0}\Big)$$

2.3.2 How does the MLP work?

The hidden nodes, in the first hidden layer, act as a kind of feature detectors. Each hidden node has its own decision plane or sensitive region in input space. The argument to a node in the nex layer will then depend on different hidden nodes in different regions, allowing for non-flat contour surfaces to the output. See figure 2.13 for an illustration and figure 2.14 for two numerical examples.

2.3.3 An XOR solution

As an example, fig. (2.15) shows a small one-hidden-layer MLP that solves the XOR problem for any positive value of the parameter ω. In this simple case, we can use the threshold activation both in the output node and in the hidden nodes. The two hidden nodes then implement the AND and the OR logic, respectively, resulting in the final XOR logic.

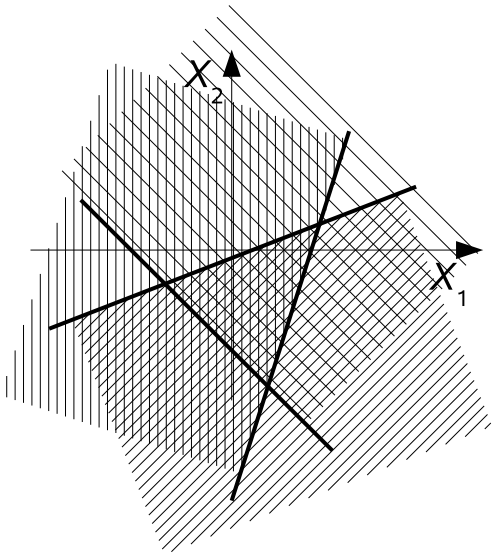


Figure 2.13: Illustration of the role played by the hidden nodes. Each thick line represent a hyperplane where the argument to the node is 0. This is the decision plane or the center of the sensitive region for that node. The shaded regions show an example of where a hidden node has positive output.

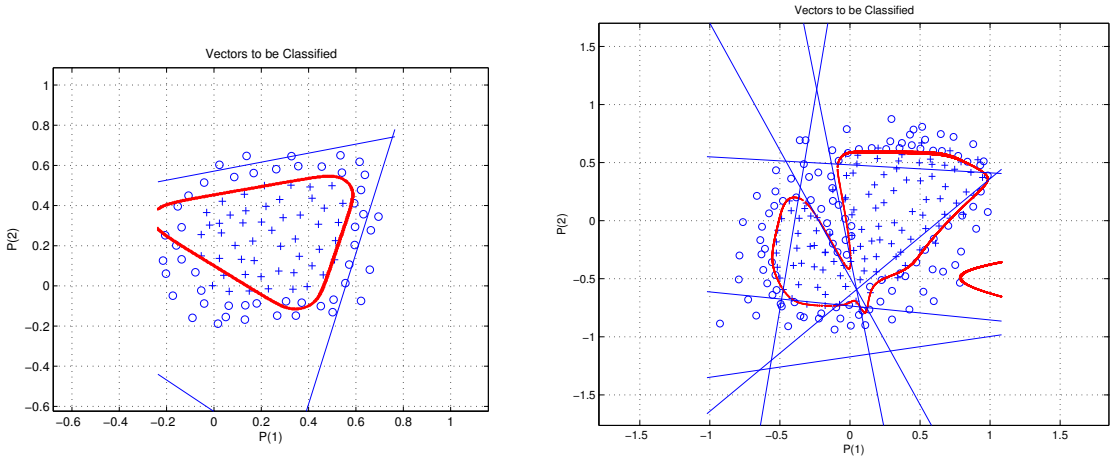
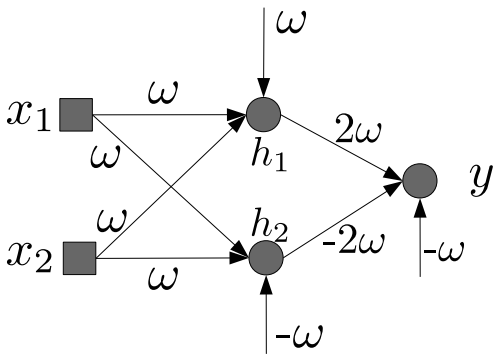


Figure 2.14: Examples of the role played by the hidden nodes. The straight line represents "hyperplanes" implemented by different hidden nodes, while the curved line is the output boundary. (Left) A 3-hidden node network. Here we see how the 3 nodes can create a closed boundary in a 2-dimensional space. (Right) A more complicated problem with more hidden nodes.



$$h_1 = \theta(\omega x_1 + \omega x_2 + \omega)$$
$$h_2 = \theta(\omega x_1 + \omega x_2 - \omega)$$
$$y = \theta(2\omega h_1 - 2\omega h_2 - \omega)$$

x_1	x_2	d	h_1	h_2	y
+1	+1	0	1	1	0
+1	-1	1	1	0	1
-1	+1	1	1	0	1
-1	-1	0	0	0	0
			$\overbrace{OR}$	$\overbrace{AND}$	

Figure 2.15: An MLP that solves the XOR problem.

2.3.4 Learning algorithm: Back-propagation

Gradient descent learning of an MLP is called "back-propagation". Compared to the gradient descent for a simple perceptron, we now have to keep track of more indices and exploit the chain rule even further. Other than that, there is nothing new. As before we will need a training dataset,

$$\mathcal{D} = \{\mathbf{d}_n, \mathbf{x}_n\}_{n=1\cdots N}\,.$$

We allow for multiple targets $\mathbf{d}_n$, and have the corresponding number of output nodes. Each input pattern $\mathbf{x}_n$ gives an output $\mathbf{y}_n$, where $y_{ni} = y_i(\boldsymbol{\omega}, \mathbf{x}_n)$. We also need a differentiable loss function $E(\boldsymbol{\omega})$ to minimize. It could for example be a function based on the sum of mean squared errors for all outputs,

$$E(\boldsymbol{\omega}) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{i}\left(d_{ni} - y_i(\boldsymbol{\omega}, \mathbf{x}_n)\right)^2. \tag{2.8}$$

Here, we use $\boldsymbol{\omega}$ to represent all weights and biases in the MLP, not only for one single node.

For an MLP with a single hidden layer, we need to update two sets of weights, namely

$$\text{input-to-hidden weights} \qquad \Delta\tilde{\omega}_{jk} = -\eta\frac{\partial E}{\partial\tilde{\omega}_{jk}}$$
$$\text{hidden-to-output weights} \qquad \Delta\omega_{ij} = -\eta\frac{\partial E}{\partial\omega_{ij}}$$

It is convenient to work with the arguments to nodes for each patten n, $a_{ni} = \sum_j \omega_{ij}h_{nj}$ for output node i and $\tilde{a}_{nj} = \sum_k \tilde{\omega}_{jk}x_{nk}$ for hidden node j. To relate to the single perceptron

algorithm in fig. (2.9) we introduce

$$\delta_{ni} = -\frac{\partial E}{\partial a_{ni}} = -\frac{\partial E}{\partial y_{ni}}\varphi_o'(a_{ni}) \tag{2.9}$$

$$\tilde{\delta}_{nj} = -\frac{\partial E}{\partial \tilde{a}_{nj}} = -\frac{\partial E}{\partial h_{nj}}\frac{\partial h_{nj}}{\partial \tilde{a}_{nj}} = -\left(\sum_i \frac{\partial E}{\partial a_{ni}}\frac{\partial a_{ni}}{\partial h_{nj}}\right)\frac{\partial h_{nj}}{\partial \tilde{a}_{nj}} = \left(\sum_i \delta_{ni}\omega_{ij}\right)\varphi_h'(\tilde{a}_{nj}) \tag{2.10}$$

With our standard choices of loss and activation, the hidden-to-output result is $\delta_{ni} = \frac{1}{N}(d_{ni} - y_{ni})$. Apart from the extra index i which specifies output node, this is exactly what we had for the single perceptron.

Once all δ_{ni} are known, they are used in eq. (2.10) to calculate all $\tilde{\delta}_{nj}$. Similarly the $\tilde{\delta}_{nj}$ values can be used to find updates in the next, even earlier layer. In this way, the deltas are "passed backwards" through the network to calculate weight updates, which is why the method is called *back-propagation*, In the literature, back-propagation is often the standard method for training MLP's.

Considering many patterns n, we get as a final expression for the weight updates

$$\Delta\omega_{ij} = \eta\sum_n \delta_{ni}h_{nj} \tag{2.11}$$

$$\Delta\tilde{\omega}_{jk} = \eta\sum_n \tilde{\delta}_{nj}x_{nk} \tag{2.12}$$

2.3.5 Data pre-processing and MLP initialization

The recipe motivated for a perceptron can be inherited for an MLP, and can be summarized:

- It is often a good idea to linearly transform all continuous inputs and outputs to zero mean and unit variance.
- If some data distributions are heavily skewed, non-linear transformations could be considered.
- Let binary variables take on values 0 and 1, or ± 1.
- For initialization of weights into a node with K input weights (sometimes called "fan-in" K), a random distribution with zero mean and variance roughly $1/K$ is suitable.
- For initialization of bias parameters, a random distribution with zero mean and about unit variance is suitable.
- Trial-and-error may be needed to find a good learning rate, but a value smaller than 1, but not vanishingly small, could be a good starting guess.
- Missing values have to be handled by discarding variables or selecting an imputation scheme.

2.4 General feed forward networks

It is not necessary to have an MLP with well-defined, fully connected layers. Here, we briefly discuss training of a general feed forward network.

2.4.1 A possible representation

The general feed forward network can be represtented by a long list of nodes, where node zero is a bias node common for the entire network and nodes 1 through K represent the K input nodes. Every node j has an activation function φ_j (wich is unused for the input nodes), an array of inputs $\mathbf{I}_j$ and an array of weights $\mathbf{W}_j$. The two arrays have identical lengths. A weight on the connection from node i to node j is then $\omega_{ji} = W_{jc}$, where $I_{jc} = i$. For example, the MLP solving XOR presented in fig. (2.15) could be represented as:

node	0	1	2	3		4		5	
φ	-	-	-	θ		θ		θ	
	$\mathbf{I}_0$ $\mathbf{W}_0$	$\mathbf{I}_1$ $\mathbf{W}_1$	$\mathbf{I}_2$ $\mathbf{W}_2$	$\mathbf{I}_3$	$\mathbf{W}_3$	$\mathbf{I}_4$	$\mathbf{W}_4$	$\mathbf{I}_5$	$\mathbf{W}_5$
	-	-	-	0	ω	0	$-\omega$	0	$-\omega$
	-	-	-	1	ω	1	ω	3	2ω
	-	-	-	2	ω	2	ω	4	-2ω

A feed forward network has the property that $I_{jc} < j$, $\forall c$. The input nodes to j are all before j in the network.

For the bias and input nodes, $\mathbf{I}_k$ has zero components. For all other nodes the bias node ($i = 0$) can be stored as the first input, $I_{j0} = 0$.

2.4.2 Feed forward

For a sample n with inputs x_{nk}, we want to create all node values h_{nj}. We need the outputs to calculate losses, and all other h-values for back-propagation later.

Initialize $\mathbf{h}$, with the length of the number of nodes. Initialize $h_0 = 1$ (bias), and $h_k = x_{nk}$ for all inputs k from 1 to K.

- For each node j in increasing order, starting at $K + 1$:
  - Set $a = 0$.
  - For each component c of $\mathbf{I}_j$:
    - Set $i = I_{jc}$.
    - Add $h_i W_{jc}$ to a
  - Set $h_j = \varphi_j(a)$

2.4.3 Back-propagation

We assume the loss E to separate into sample-specific terms, $E = \frac{1}{N}\sum_n E_n$. Also, all activation functions in the network must be differentiable. (So, the θ-functions in the example in section 2.4.1 need to be replaced by, *e.g.,* tanh functions.)

- For each node j, create a gradient vector $\boldsymbol{g}_j$, of the same length as $\mathbf{I}_j$. Set all $g_{jc} = 0$.
- For each sample n:
  - Use inputs $\mathbf{x}_n$ and the feed forward algorithm to get all node values $\mathbf{h}_n$.
  - Find loss E_n, and derivatives $\partial E_n/\partial h_{nl}$ for all output nodes l.
  - Initialize a node gradient vector $\mathbf{G}$, of the same length as $\mathbf{h}_n$, with all output components $G_l = \partial E_n/\partial h_{nl}$, and all other components $G_j = 0$.
  - For each node j in reverse order (ending at $K+1$):
    - Calculate $A_j = G_j\varphi'_j$.
      If all derivatives φ' are expressed in terms of φ, each φ'_j can be found from h_{nj}.
    - For each component c of $\mathbf{I}_j$:
      - Set $i = I_{jc}$.
      - Add $W_{jc}A_j$ to G_i.
      - Add $h_i A_j/N$ to g_{jc}.
- Use all $\boldsymbol{g}_j$ for weight update. For example, gradient descent with a learning rate η would result in the updates $W_{jc} = W_{jc} - \eta g_{jc}$.

2.5 Target distributions

2.5.1 The universal approximation theorem

The following theorem states that a MLP can approximate any continuous function with a compact definition range. It is important since it means that there is no fundamental constraint built into the MLP. It is an proof-of-existence, meaning that no information is given concerning the details of the MLP. We only state the theorem here and leave the proof to a more rigorous mathematics course.

Let $\varphi(\cdot)$ be a non-polynomial function and let $f(\mathbf{x})$ be a continuous function defined over the m-dimensional hypercube $I_m = [0,1]^m$. For any $\varepsilon > 0$ there exists an integer N_h and a set of real constants: $\omega_j, \tilde{\omega}_{jk}, b_j$ $(j = 1, \cdots, N_h,\ k = 1, \cdots, m)$ such that

$$|F(\mathbf{x}) - f(\mathbf{x})| < \varepsilon \quad \forall \mathbf{x} \in I_m$$

where

$$F(\mathbf{x}) = \sum_j^{N_h} \omega_j \varphi\left(\sum_{k=1}^{m} \tilde{\omega}_{jk} x_k + b_j\right).$$

Remark 1: $F(\mathbf{x})$ is a 2-layer network with $\varphi(\cdot)$ as the activation function for the hidden layer and a linear output node.

Remark 2: The theorem does not tell me how many nodes that I need for a specific problem.

Remark 3: The theorem also applies for classification problems.

Remark 4: While an MLP with a large enough hidden layer can solve a complicated task presented by training data, the results of the MLP outside the range of training patterns can be counter-intuitive. As an example, notice the extra decision boundary outside availiable data in the right plot in fig. (2.14)!

2.5.2 Target means

Right after we have learned about the universal approximation theorem, we should note that our data with may not represent a well-defined function $d(\mathbf{x})$. The extreme case would be that we have many patterns with identical inputs $\mathbf{x}_{n_1} = \mathbf{x}_{n_2} = \mathbf{x}^*$ but different targets $d_{n_1} \neq d_{n_2}$. We take a look at what a large network, able to fit a complicated function, would give in this case.

Our training will be completed if for every weight ω we have

$$0 = \frac{\partial E}{\partial \omega} = \sum_n \frac{\partial a_n}{\partial \omega} \frac{\partial E}{\partial a_n} = \frac{1}{N} \sum_n \frac{\partial a_n}{\partial \omega} (y_n - d_n), \tag{2.13}$$

where we have used the fact that the derivative of the loss with respect to the argument to the output node, $\frac{\partial E}{\partial a}$, equals $\frac{1}{N}(y - d)$ for the standard setup of both regression and classifation, as we noted in the end of section 2.2.2.

With the network large enough to fit all targets except the conflicting ones, the only non-zero terms come from patterns with the same input $\mathbf{x}^*$ but different targets. The loss will then be minimized when

$$0 = \frac{1}{N} \frac{\partial a^*}{\partial \omega} \sum_m (y^* - d_m), \tag{2.14}$$

where each $\mathbf{x}_m = \mathbf{x}^*$. This is solved when the output equals the sample mean of targets, $y^* = \overline{d(\mathbf{x}^*)}$. For large datasets, the sample mean approaches the conditional expectation value

$$\overline{d(\mathbf{x})} \to \langle d|\mathbf{x} \rangle, \quad N \to \infty. \tag{2.15}$$

Thus, while we cannot be sure that there is a function $d(\mathbf{x})$ for a network to find, we can always expect there to be a conditional probability distribution $p(d|\mathbf{x})$, The standard choices for loss and activation then give the network the task to find the conditional expectation value $\langle d|\mathbf{x} \rangle$. This is illustrated in figure 2.16.

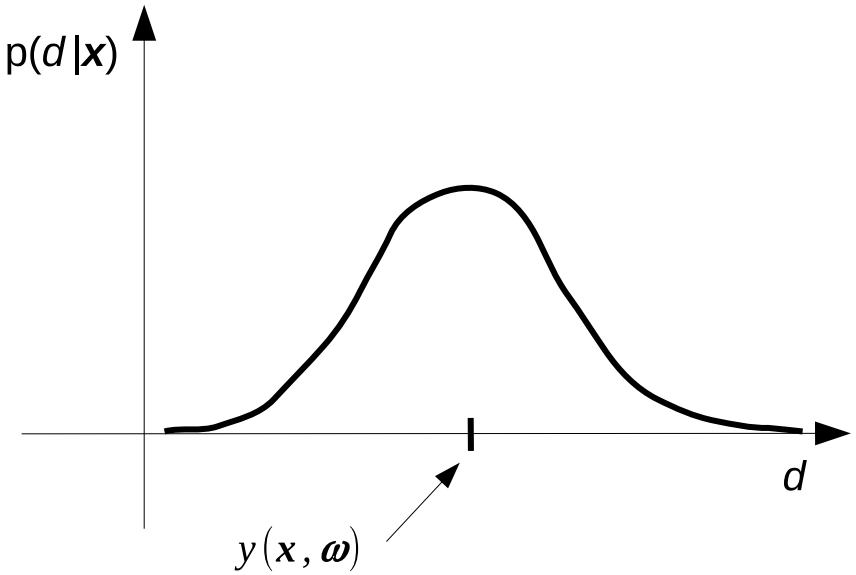


Figure 2.16: Illustration of the conditional distribution $p(d_i|\mathbf{x})$ and the corresponding network mapping found by the weights $\boldsymbol{\omega}$ that minimize the loss function.

2.5.3 Loss and maximum likelihood

In rare cases, such as the 2-dimensional AND and XOR problems discussed earlier, the training data is an exhaustive list of all possible inputs $\mathbf{x}$, but it is more common that the training data set,

$$\mathcal{D} = \{\mathbf{x}_n, d_n\}_{n=1\cdots N},$$

can be seen as a sample from an unknown distribution $p(\mathcal{D})$. Assuming independent samples, this distribution can be written

$$p(\mathcal{D}) = \prod_n p(d_n, \mathbf{x}_n) = \prod_n p(d_n|\mathbf{x}_n)p(\mathbf{x}_n). \tag{2.16}$$

Here $p(d_n, \mathbf{x}_n)$ is the *joint* distribution for inputs $\mathbf{x}_n$ and target d_n, while $p(d_n|\mathbf{x}_n)$ is the *conditional* distribution for target d_n, given inputs $\mathbf{x}_n$.

A neural network produces an output $y(\mathbf{x}, \boldsymbol{\omega})$ that we interprete as a model for the conditional average $\langle d|\mathbf{x}\rangle$. This means that the parameters $\boldsymbol{\omega}$ also become part of a model of the distribution $p(\mathcal{D})$. The probability of getting our sample $\mathcal{D}$ according to the model represented by $\boldsymbol{\omega}$, is called the *likelihood* for $\mathcal{D}$ given $\boldsymbol{\omega}$. We denote that likelihood $p(\mathcal{D}|\boldsymbol{\omega})$, and get

$$p(\mathcal{D}|\boldsymbol{\omega}) = \prod_n p(d_n, \mathbf{x}_n|\boldsymbol{\omega}) = \prod_n p(d_n|\mathbf{x}_n, \boldsymbol{\omega})p(\mathbf{x}_n). \tag{2.17}$$

Our weights $\boldsymbol{\omega}$ only influence the relation between inputs and targets, so $p(\mathbf{x}_n)$ is independent of $\boldsymbol{\omega}$.

One way to train the network is to look for the $\boldsymbol{\omega}$ that maximizes the likelihood $p(\mathcal{D}|\boldsymbol{\omega})$. This *Maximum Likelihood* approach is very common in statistical analyses. Instead of maximizing the likelihood we can minimize its negative logarithm,

$$-\ln p(\mathcal{D}|\boldsymbol{\omega}) = -\sum_{n=1}^{N} \ln p(d_n|\mathbf{x}_n, \boldsymbol{\omega}) - \underbrace{\sum_{n=1}^{N} \ln p(\mathbf{x}_n)}_{\text{Independent of } \boldsymbol{\omega}} \tag{2.18}$$

Omitting the term independent of $\boldsymbol{\omega}$, we want to minimize a loss proportional to the remaining term

$$E(\boldsymbol{\omega}) \propto -\sum_{n} \ln p(d_n|\mathbf{x}_n, \boldsymbol{\omega}) \tag{2.19}$$

Regression

In regression problems, we can always define a residual error ε that relates the observation d to the expectation value $\langle d \rangle$, as

$$d(\mathbf{x}) = \langle d|\mathbf{x}\rangle + \varepsilon. \tag{2.20}$$

Thus, ε represents the noise that makes it possible for d to differ from $\langle d \rangle$. A common and simple model is a Gaussian noise distribution with constant width σ,

$$p(\varepsilon) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\varepsilon^2}{2\sigma^2}\right). \tag{2.21}$$

With $y(\mathbf{x}, \boldsymbol{\omega})$ as our model for $\langle d|\mathbf{x}\rangle$ we then get $\varepsilon = d(\mathbf{x}) - y(\mathbf{x}, \boldsymbol{\omega})$ and

$$-\ln p(d|\mathbf{x}, \boldsymbol{\omega}) = \frac{(d - y(\mathbf{x}, \boldsymbol{\omega}))^2}{2\sigma^2} + \frac{1}{2}\ln 2\pi\sigma^2. \tag{2.22}$$

Minimizing this expression is equivalent to minimizing the loss

$$E(\boldsymbol{\omega}) = -\frac{1}{2N}\sum_{n}(d_n - y_n)^2 \tag{2.23}$$

Here, we multiplied the log likelihood by σ^2/N to get rid of the unknown constant σ and to help us find a good learning rate η for gradient descent training. Without the $1/N$ factor, a good η would strongly depend on the size of the dataset.

In summary, the combination of linear output and an MSE-based loss will give outputs y that can be interpreted as models for $\langle d|\mathbf{x}\rangle$, in a simple model assuming Gaussian noise.

Binary classification

For binary classification, we have

$$\langle d|\mathbf{x}\rangle = 1 \cdot p(1|\mathbf{x}) + 0 \cdot p(0|\mathbf{x}) = p(1|\mathbf{x}). \tag{2.24}$$

Since $y_n = y(\mathbf{x}_n, \boldsymbol{\omega})$ is a model for $\langle d|\mathbf{x}_n\rangle$, it then also models $p(d_n|\mathbf{x}_n, \boldsymbol{\omega})$ as

$$p(d_n|\mathbf{x}_n, \boldsymbol{\omega}) = \left\{ \begin{array}{rl} y_n, & d_n = 1 \\ 1 - y_n, & d_n = 0 \end{array} \right\} = y_n^{d_n}(1 - y_n)^{1-d_n} \quad \text{(Bernoulli distribution)}. \tag{2.25}$$

The loss motivated by eq. (2.19) is then

$$E(\boldsymbol{\omega}) = -\frac{1}{N} \sum_n \left[d_n \ln(y_n) + (1 - d_n) \ln(1 - y_n) \right], \tag{2.26}$$

which is just the cross-entropy error! As with regression, we have included a factor $1/N$ to get make the learning rate independent of the size of the dataset.

Multiple classes

With the likelihood framework in place, we are ready to formulate a strategy for classification tasks with $c > 2$ classes. If we tried a single output node for this task, we would have to let the target d be an integer, where $d = 1$ for class C_1, $d = 2$ for class C_2, etc. However, we do not want to impose any sorting of classes. For example, if the classes are "Cat", "Boat", "Fork", there is no reason to say that one of those is between the others.

Instead, the preferred coding scheme for the classes is called "one-hot encoding" and is given by

$$d_{ni} = \left\{ \begin{array}{ll} 1, & n \in C_i \\ 0, & n \notin C_i \end{array} \right.$$

Below is an example with four classes,

1 0 0 0	class 1
0 1 0 0	class 2
0 0 1 0	class 3
0 0 0 1	class 4

We let the representation be redundant, with c targets. We could use $c - 1$ targets and let one class be "none of the above", but we do not want any class to be handled differently from the others in the network.

(Side note: if one of our inputs x_k is also a multi-class variable with p classes, it is a good idea to turn it into a p-dimensional one-hot vector during data pre-processing.)

As before, we want our MLP to model $\langle d_i|\mathbf{x}\rangle = p(C_i|\mathbf{x})$, i.e.

$$y_{ni} = y_i(\mathbf{x}_n, \boldsymbol{\omega}) = P(C_i|\mathbf{x}_n, \boldsymbol{\omega}) \tag{2.27}$$

Each pattern now has a target vector $\mathbf{d}_n$, with one component $d_{nn^*} = 1$,

$$d_{ni} = \begin{cases} 1, & i = n^* \\ 0, & i \neq n^* \end{cases} \tag{2.28}$$

The target vector specifies that the pattern n belongs to class n^*. Therefore

$$p(\mathbf{d}_n|\mathbf{x}_n) = p(C_{n^*}|\mathbf{x}_n) = y_{nn^*}. \tag{2.29}$$

Making use of eq. (2.28), a loss motivated by $-\ln p(\mathbf{d}|\mathbf{x}) = -\sum_n \ln y_{nn*} = \sum_{ni} d_{ni} \ln y_{ni}$ is the *categorical cross-entropy error*

$$E(\boldsymbol{\omega}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{c} d_{ni} \ln y_{ni}. \tag{2.30}$$

What output activation function should we have for multiple classes? If we want to be able to interpret the output as probabilities we want $y_i(\mathbf{x}_n) \in [0, 1]$ and $\sum_i y_i(\mathbf{x}_n) = 1$. A generalization of the logistic function, called softmax or smooth winner-takes-all, turns out to be a solution,

$$y_i(\mathbf{x}_n) = \frac{e^{a_{ni}}}{\sum_{i'} e^{a_{ni'}}}$$

where a_{ni} is the argument to the node i activation function for pattern n.

Notice that softmax rather is a function for the entire output layer, not for individual nodes. The output y_{ni} depends on all arguments $a_{ni'}$. In the big chain-rule exercise to find gradients $\nabla_{\boldsymbol{\omega}} E$, we among other things need to calculate all $\frac{\partial E}{\partial a_{ni}}$. The only term in the categorical cross-entropy error that depends on a_{ni} is $-\frac{1}{N}\ln(\mathbf{d}_n|\mathbf{x}_n)$ which according to eq. (2.29) is $-\frac{1}{N}\ln y_{nn^*}$. In other words, we need to calculate

$$\frac{\partial \ln y_{nn^*}}{\partial a_{ni}} = \frac{\partial a_{nn^*}}{\partial a_{ni}} - \frac{\partial}{\partial a_{ni}} \ln\left(\sum_{i'} \exp(a_{ni'})\right) = \frac{\partial a_{nn^*}}{\partial a_{ni}} - \frac{\exp(a_{ni})}{\sum_{i'} \exp(a_{ni'})} = d_{ni} - y_{ni},$$

where the last step follows from

$$\frac{\partial a_{nn^*}}{\partial a_{ni}} = \left\{ \begin{array}{ll} 1, & i = n^* \\ 0, & i \neq n^* \end{array} \right\} = d_{ni}$$

according to eq. (2.28).

At the end of all calculations, we get $\frac{\partial E}{\partial a_{ni}} = \frac{1}{N}(y_{ni} - d_{ni})$, just as for the binary cross entropy error!

Summary

Training a network can be seen as both trying to solve $y_n = \langle d_n | \mathbf{x}_n \rangle$ and maximizing the likelihood $\prod_n p(d_n|\mathbf{x}_n)$ if we make the following choices:

Problem	# of classes	φ_{output}	Loss function
regression	-	linear	mean square error
classification	2	logistic	cross entropy (binary)
classification	> 2	softmax	cross entropy (categorical)

2.5.4 Bayesian learning of neural networks

We will very briefly discuss how the maximum likelihood approach can be extended to a Bayesian framework.

Before we start training, we have no real clue what the weights $\boldsymbol{\omega}$ should be, but even so, we always start with some restrictions. With the size of our network, we claim that a longer $\boldsymbol{\omega}$-vector, representing more nodes, is not needed. With our weight initialization, we introduce some guess of the typical size of a weight value. In the Bayesian framework, such "beliefs" are represented by a *prior distribution* $p(\boldsymbol{\omega})$. This idea can seem bold, but one argument is that whenever we start to build a model, we implicitly introduce a lot of assumptions about $\boldsymbol{\omega}$, and the prior distribution $p(\boldsymbol{\omega})$ helps us to keep track of the consequences.

Given a prior and a dataset $\mathcal{D}$, we can calculate the conditional probability $p(\mathcal{D}|\boldsymbol{\omega})$, and also the total probability for the set $\mathcal{D}$,

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\omega}) p(\boldsymbol{\omega}) d\boldsymbol{\omega}.$$

However, what we really want to formulate is how the dataset $\mathcal{D}$ influences our belief in different $\boldsymbol{\omega}$: we want a *posterior distribution* $p(\boldsymbol{\omega}|\mathcal{D})$. Bayes' theorem allows us to write

$$p(\boldsymbol{\omega}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\omega}) p(\boldsymbol{\omega})}{p(\mathcal{D})} \tag{2.31}$$

So in order to compute $p(\boldsymbol{\omega}|\mathcal{D})$ we need to...

formulate the prior	$p(\boldsymbol{\omega})$
compute the likelihood	$p(\mathcal{D}|\boldsymbol{\omega})$

Note that for a large dataset $\mathcal{D}$, the likelhood $p(\mathcal{D}|\boldsymbol{\omega})$ peaks so strongly near the maximum likelhood value $\boldsymbol{\omega}^*$ that the both the width around $\boldsymbol{\omega}^*$ and the effects of the prior can

be neglected. The maximum likelihood method can therefore be seen as an approximate Bayesian strategy.

With our network models, we focus on the relations between inputs and targets. Changing notation from dataset $\mathcal{D}$ to targets $\mathbf{D}$ and inputs $\mathbf{X}$, where $\mathbf{X}$ are independent of $\boldsymbol{\omega}$ so that $p(\mathbf{X}|\boldsymbol{\omega}) = p(\mathbf{X})$, we can write Bayes' relation in eq. (2.31) as

$$p(\boldsymbol{\omega}|\mathcal{D}) = \frac{p(\mathbf{D}|\mathbf{X},\boldsymbol{\omega})p(\boldsymbol{\omega})}{p(\mathbf{D}|\mathbf{X})}, \tag{2.32}$$

where the normalization in the denominator is

$$p(\mathbf{D}|\mathbf{X}) = \int p(\mathbf{D}|\mathbf{X},\boldsymbol{\omega})p(\boldsymbol{\omega})d\boldsymbol{\omega}.$$

Our goal is to find $\boldsymbol{\omega}$ with a large posterior, so we can decide to minimize

$$-\ln p(\boldsymbol{\omega}|\mathcal{D}) = -\ln p(\mathbf{D}|\mathbf{X},\boldsymbol{\omega}) - \ln p(\boldsymbol{\omega}) + \ln p(\mathbf{D}|\mathbf{X}) \tag{2.33}$$

The last logarithm can be omitted, since it is independent of $\boldsymbol{\omega}$. With indepedent samples, $p(\mathbf{D}|\mathbf{X},\boldsymbol{\omega}) = \prod_n p(\mathbf{d}_n|\mathbf{x}_n,\boldsymbol{\omega})$, we then get to minimize

$$S(\boldsymbol{\omega}) = -\sum_n \ln p(\mathbf{d}_n|\mathbf{x}_n,\boldsymbol{\omega}) - \ln p(\boldsymbol{\omega}) \tag{2.34}$$

Adding terms to the loss that only depend on $\boldsymbol{\omega}$ without any data dependence, can be seen as representing a prior distribution. We will later discuss how this can be exploited to improve network models.

Example: Gaussian noise model

The Gaussian prior is a very common way to represent an expectation of some typical weight values:

$$p(\boldsymbol{\omega}) \propto \exp\left(-\frac{1}{2\sigma_\omega^2}\sum_i \omega_i^2\right).$$

This leads to the so-called "L2-regularization term" in the error function (see chapter 3.5.2):

$$S(\boldsymbol{\omega}) = \sum_n E_n(\boldsymbol{\omega}) + \frac{1}{2\sigma_\omega^2}\sum_i \omega_i^2. \tag{2.35}$$

In practical applications, the unknown width σ_ω allows us to introduce a free parameter $\alpha = \frac{1}{N\sigma_\omega^2}$.

The weight vector $\boldsymbol{\omega}_{MP}$ that maximizes the posterior $p(\boldsymbol{\omega}|\mathcal{D})$ is found by minimizing $S(\boldsymbol{\omega})$. Due to the prior, it differs from the maximum likelihood weight vector $\boldsymbol{\omega}_{ML}$ that minimizes $\sum_n E_n(\boldsymbol{\omega})$. Figure 2.17 illustrates the difference between maximum likelihood and maximum a posteriori.

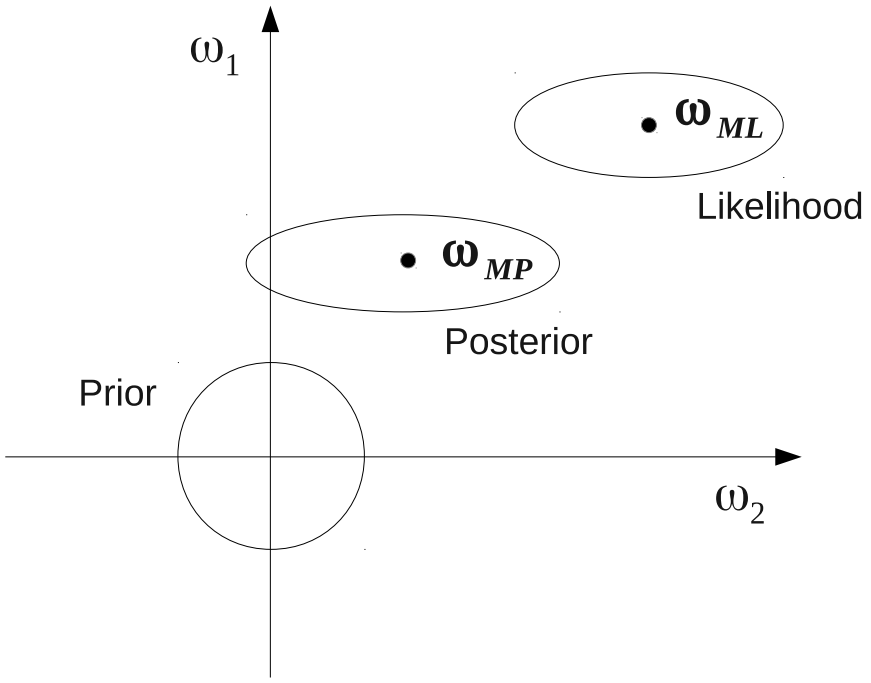


Figure 2.17: Illustration of the difference between maximum likelihood and maximum a posteriori. The posterior distribution is now a combination of both the likelihood and the prior, changing the obtained weight vector from $\boldsymbol{\omega}_{ML}$ to $\boldsymbol{\omega}_{MP}$.

2.5.5 Distribution of network outputs

We can take the Bayesian picture one step further. Once the posterior $p(\boldsymbol{\omega}|\mathcal{D})$ is found, we can use it to predict the target distribution for any $\mathbf{x}$, even those not present in the dataset $\mathcal{D}$. Using the rules of probability we can write

$$p(d|\mathbf{x}, \mathcal{D}) = \int p(d|\mathbf{x}, \boldsymbol{\omega}) p(\boldsymbol{\omega}|\mathcal{D}) d\boldsymbol{\omega} \tag{2.36}$$

We can see this formula as a weighting of all possible models $p(d|\mathbf{x}, \boldsymbol{\omega})$, where the weighting is taken care of by the posterior weight distribution $p(\boldsymbol{\omega}|\mathcal{D})$.

We saw previously that the MLP gives us the conditional average $\langle d|\mathbf{x}, \mathcal{D}\rangle$. The above eq. (2.36) gives us a distribution of outputs which we could use to, *e.g.,* assign confidence intervals to our prediction.

How do we in practice compute $p(d|\mathbf{x}, \mathcal{D})$ according to eq. (2.36)? The are two main alternatives:

1. Monte Carlo approximation: This means that we approximate

$$p(d|\mathbf{x}, \mathcal{D}) \approx \frac{1}{K} \sum_{k=1}^{K} p(d|\mathbf{x}, \boldsymbol{\omega}_k)$$

where $\boldsymbol{\omega}_k$ are drawn from the posterior weight distribution $p(\boldsymbol{\omega}|\mathbf{D})$. Practically this is accomplished using various Markov Chain Monte Carlo (MCMC) methods.

2. Approximation by Gaussians: Here we approximate the posterior weight distribution $p(\boldsymbol{\omega}|\mathcal{D})$ by a Gaussian distribution $q(\boldsymbol{\omega})$. This can be done by Taylor expanding the expression for $S(\boldsymbol{\omega})$ (eq. (2.35)) up the second order. Another popular method is "Variational Inference", where we treat the parameters $\boldsymbol{\theta}$ of the Gaussian (the means and widths of all ω_i) as trainable, and adjust them so that the approximate parameterization $q(\boldsymbol{\omega}|\boldsymbol{\theta})$ becomes as "similar" to the posterior distribution as possible. One measure of simliarity is the Kullback-Leibler divergence $\int \mathrm{d}\boldsymbol{\omega} q(\boldsymbol{\omega}) \ln \frac{p(\boldsymbol{\omega})}{q(\boldsymbol{\omega})}$.

In summary, the Bayesian approach to learning offers some new important features:

- It motivates "regularization terms" $-\ln p(\boldsymbol{\omega})$ in the loss function.
- For regression problems, confidence intervals and error bars can be assigned to the predictions generated by the network.
- We do not expand on it here, but the relative importance of different inputs can be determined using the Bayesian technique of ARD (Automatic Relevance Determination).

2.6 Loss minimization issues

Here we discuss how to improve the efficiency of the gradient descent method, and also briefly introduce more advanced minimization methods.

The gradient descent method has the following basic updating formula

$$\Delta\omega_i = -\eta \frac{\partial E}{\partial \omega_i}$$

It is easy to implement and generally fast, but there are still some concerns.

1. It requires a reasonably informed setting of learning rate η.
2. If there is a plateau in the loss function, $\nabla_{\boldsymbol{\omega}} E$ is small and it takes a long time to leave the region.

3. It is possible to get stuck in a local minimum.
4. If the dataset is large, calculating $\nabla_{\boldsymbol{\omega}} E = \frac{1}{N} \sum_n \nabla_{\boldsymbol{\omega}} E_n$ can become very slow.

The points 2 and 3 are illustrated in fig. (2.18).

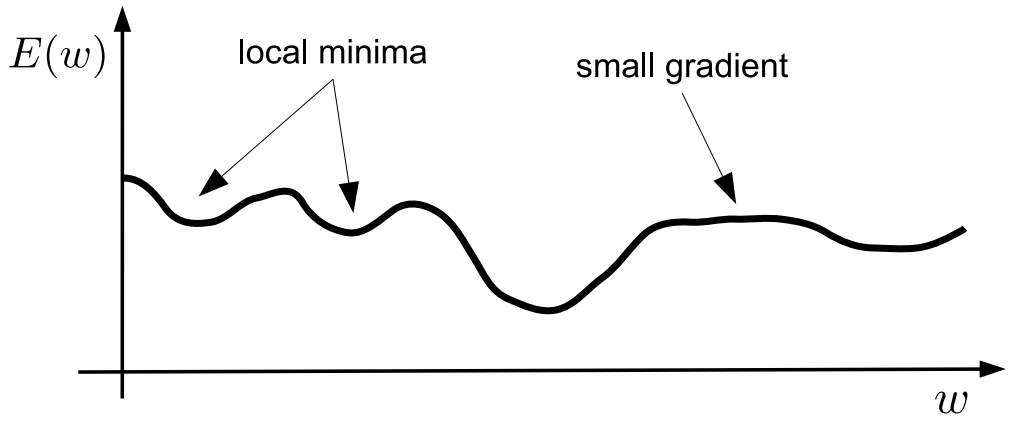


Figure 2.18: Illustration of the problem of local minima and small gradients.

2.6.1 Stochastic Gradient Descent (SGD)

A crucial improvement is *Stochastic Gradient Descent* (SGD). It is an efficient way to solve the time issue (point 4), but also helps to escape local minima. It exploits the fact that the proposed loss functions are just means of losses E_n for individual patterns,

$$E = \frac{1}{N} \sum_n E_n,$$

where $E_n = \frac{1}{2}(y_n - d_n)^2$ for the mean squared error and $E_n = -d_n \ln(y_n) - (1 - d_n) \ln(1 - y_n)$ for the cross entropy error. This means that it is possible to update the weights using only a so-called *minibatch* of P patterns in the training dataset,

$$\Delta\omega_k = \frac{1}{P} \sum_{p=1}^{P} \Delta\omega_{pk}, \qquad \text{with} \qquad \Delta\omega_{nk} = -\eta \frac{\partial E_n}{\partial \omega_k}.$$

Here, P is the *minibatch size*. The word "stochastic" highlights that we make a random partitioning of samples into different minibatches.

Every time we have used one minibatch to update the weights, we have performed one *iteration*.

Typically all patterns are partitioned into minibatches so that all patterns are included in precisely one part. When all patterns have been used for weight updates once, we have peformed one *epoch*. After each epoch, a new random partitioning is usually made.

The special case of $P = N$, so that all data is used for one update (one iteration is also one epoch), is called *batch updating*. In this case, we are actually doing just gradient descent, without anything "stochastic". The special case of $P = 1$, when weights are updated after each pattern, is called *online updating*. We could introduce stochasticity by randomly order the patterns within an epoch.

Even though a minibatch only gives an approximation of the true gradient, SGD is usually much quicker than batch updating, since we perform many reasonable updates in a single epoch. Furthermore, if P is small enough for the gradient to become strongly dependent on pattern selection, we may "shake out" of a local minimum or even a plateau. A typical minibatch size P is 10 to 50.

We may worry that SGD prevents us from finding the perfect solution to $\nabla_{\boldsymbol{\omega}} E = 0$, since stochastic selection of minibatches will cause our weights to wobble around in the neighbourhood of the correct solution. Later, when we discuss the problem of overtraining, we will learn that this need not be an issue, and may even be an asset!

2.6.2 Gradient descent enhancements

Many graident descent enhancements use information from previous iterations. We will look at different ways to relate the loss $E(t)$, gradient $\boldsymbol{g}(t) = \nabla_{\boldsymbol{\omega}} E(t)$ and weight update $\Delta\boldsymbol{\omega}(t)$ at iteration t.

Momentum term

The momentum term adds a part of the previous update to the current one,

$$\Delta\boldsymbol{\omega}(t+1) = -\eta\boldsymbol{g}(t+1) + \alpha\Delta\boldsymbol{\omega}(t)$$

where usually $0.5 < \alpha < 1$. In a plateau-like region, when several updates in a row have the same sign, it will increase the minimzation process. Near an optimum, there will be a form of damped oscillation. In a small local minimum the momentum may take the weight update over the next "crest" in the loss landscape, thus escaping the minimum.

To help select parameters η and α, we can note that the gradient at iteration t will also influence the future updates. Collecting all the $\boldsymbol{g}(t)$-dependent terms after many updates

gives

$$\begin{aligned}\sum_{t'\geq t}\Delta\boldsymbol{\omega}(t') &= \Delta\boldsymbol{\omega}(t)+\Delta\boldsymbol{\omega}(t+1)+\ldots= \\ &= -\eta\boldsymbol{g}(t)-\eta\alpha\boldsymbol{g}(t)-\eta\alpha^2\boldsymbol{g}(t)+\text{other terms}\approx \\ &\approx -\frac{\eta}{1-\alpha}\boldsymbol{g}(t)+\text{other terms.}\end{aligned}$$

Thus, with some knowledge of a good learning rate η_0 without momentum, one can assume a reasonable combination of η and α satisfies $\eta/(1-\alpha)\approx\eta_0$.

Before listing other alternatives, we should note that a well-tuned stochastic gradient descent with momentum is still a very popular and useful choice.

Dynamical learning rate

The following idea (called "bold driver") modifies the learning rate based on the error changes.

$$\eta(t+1)=\begin{cases}\eta(t)\cdot(1-\epsilon) & \text{if } E(t+1)\geq E(t)\\ \eta(t)\cdot(1+\frac{\epsilon}{10}) & \text{otherwise}\end{cases} \tag{2.37}$$

where $0<\epsilon\ll 1$.

There are better algorithms based on dynamical learning rates. We will discuss RPROP and Adam.

Individual and dynamical learning factors - RPROP

One way to escape a plateau is to replace $\frac{\partial E}{\partial\omega_k}$ with $\text{sgn}\left(\frac{\partial E}{\partial\omega_k}\right)$, so that the step $\Delta\boldsymbol{\omega}$ does not vanish. One method based on this idea is RPROP (Resilient PROPagation). The method introduces individual dynamical learning rates η_k, which are initialized to some selected value, $\eta_k(t=0)=\eta_0$. In each iteration, weights are updated by

$$\Delta\omega_k=-\eta_k\text{sgn}(g_k)$$

and the learning rates are updated by

$$\eta_k(t)=\begin{cases}\gamma^+\eta_k(t-1) & \text{if} \quad g_k(t)\cdot g_k(t-1)>0\\ \gamma^-\eta_k(t-1) & \text{if} \quad g_k(t)\cdot g_k(t-1)<0\end{cases} \tag{2.38}$$

where the constraints $0<\gamma^-<1<\gamma^+$ are motivated as follows: When the update of ω_k changes sign, one may assume that we have passed its optimum, so that we are close to it,

and want to reduce the future learning rate. Otherwise, we seem to be far away from an optimum and want to increase the learning rate. RPROP introduces new parameters γ^- and γ^+. A suggestion is $\gamma^- = 0.5$ and $\gamma^+ = 1.2$.

However, when combined with minibatch update, the sign of $\frac{\partial E}{\partial \omega_k}$ starts to change *before* we have reached the optimum, because of fluctuations between mini-batches. This can cause the individual learning rates to decrease too soon, and make it harder to avoid local minima. Even though this need not be a problem in many tasks, the method is more or less superceded by a method called Adam.

Adam

If you have experimented with training networks before, you may have used the Adam (short for Adaptive moment estimation) "minimizer". Let's now describe this method!

It is based on a running average of the gradient, just as momentum, but also by a running average of the square of derivatives. We present the algorithm first and discuss the rationale for it afterwards.

In Adam, we have three learning parameters, η, β_1 and β_2. During training, we keep running averages $\boldsymbol{m}$ and $\boldsymbol{v}$ as:

$$\begin{aligned} m_k(t+1) &= \beta_1 m_k(t) + (1-\beta_1) g_k(t) \\ v_k(t+1) &= \beta_2 v_k(t) + (1-\beta_2)[g_k(t)]^2 \end{aligned}$$

With $\hat{m}_k(t+1) = \frac{m_k(t+1)}{1-\beta_1^t}$ and $\hat{v}_k(t+1) = \frac{v_k(t+1)}{1-\beta_2^t}$, weights are then updated by

$$\omega_k(t+1) = \omega_k(t) - \eta \cdot \frac{\hat{m}_k(t+1)}{\sqrt{\hat{v}_k(t+1)} + \epsilon}$$

where ϵ is a small number just to avoid division by zero problems at the beginning.

We note that $\hat{m}_k(t+1) = \frac{g_k(t)+\beta_1 g_k(t-1)+\beta_1^2 g_k(t-2)+\ldots}{1+\beta_1+\beta_1^2+\ldots}$, is a weighted average of derivates, where $0 < \beta_1 < 1$ gives largest weight to the latest value. Similarly, $\hat{v}_k(t+1) = \frac{v_k(t+1)}{1-\beta_2^t}$ is a weighted average of g_k^2.

If g_k does not change much over iterations, which could be the case in a plateau-like region, we would get $|\hat{m}_k| \sim \sqrt{\hat{v}_k}$ and $\omega_k(t+1) \approx \omega_k(t) - \eta \cdot \mathrm{sgn}(g_k)$. However, near an optimum we can expect g_k to change sign every now and then, while of course g_k^2 does not, so that $|\hat{m}_k| \ll \sqrt{\hat{v}_k}$, and the size of weight updates decreases. In this way, Adam is designed to behave as the best of SGD and RPROP in different situations.

Adam was presented with the suggestion $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$. The special case $\beta_1 = 0$, where there is no momentum term to the gradient but still a weigthed sum of squared

gradients, is called RMSPROP (Root Mean Square Propagation), and was introduced before Adam.

2.6.3 Line minimization methods

In a line minimization procedure, we decide on a direction $\mathbf{p}(t)$ for weight updates, and then try to find the step size that minimizes E in that direction. In "Steepest Descent", we let $\mathbf{p} = -\boldsymbol{g}$, and use some iterative search to find the λ that minimizes $E(t+1)$ when

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) - \lambda \boldsymbol{g}(t)$$

However, since steepest descent minimizes E in each step, we get

$$0 = \frac{\partial}{\partial \lambda} E\left(\boldsymbol{\omega}(t) + \lambda \boldsymbol{g}(t)\right) = \nabla_{\boldsymbol{\omega}} E\left(t+1\right) \cdot \boldsymbol{g}(t) = \boldsymbol{g}(t+1) \cdot \boldsymbol{g}(t),$$

and we see two consecutive steps are perpendicular: steepest descent follows a "zigzag" path, as illustrated in fig. (2.19). That seems a bit inefficient, and in *conjugate gradient* methods,

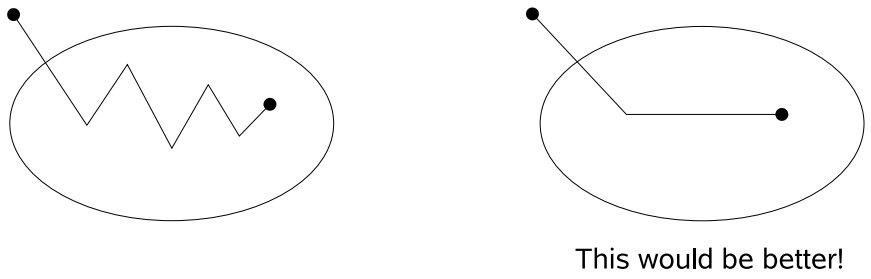


Figure 2.19: (Left) Steepest descent follows a "zigzag" path. (Right) With conjugate gradients, we aim for a more clever choice of new direction.

the step direction $\mathbf{p}$ is modified to

$$\mathbf{p}(t+1) = -\boldsymbol{g}(t+1) + \beta \mathbf{p}(t).$$

To avoid a zig-zag path, the direction remembers the previous one to some extent. There are a few different suggestions for β, for example the so called Polak-Ribiere rule,

$$\beta = \frac{[\boldsymbol{g}(t+1) - \boldsymbol{g}(t)] \cdot \boldsymbol{g}(t+1)}{(\boldsymbol{g}(t))^2}$$

Notice that even though there is an analytical expression for β, the step size λ along $\mathbf{p}$ is still determined iteratively.

For the specific case of a MSE loss, we can use the so-called Levenberg-Marquardt method to minimize an approximation of the next error, instead of numerically finding the step size λ. With target-to-output errors $e_n = y_n - d_n$ as components in a vector $\mathbf{e}$, we get

$$e_n(t+1) \approx e_n(t) + \sum_k \frac{\partial y_n(t)}{\partial \omega_k(t)} \cdot (\omega_k(t+1) - \omega_k(t)).$$

Since $E \propto \sum_n e_n^2 = \mathbf{e}^2$, the approximate function to minimize can be written

$$\tilde{E} = \frac{1}{2}\|\mathbf{e}(\boldsymbol{\omega}_{old}) + \mathbf{Z}(\boldsymbol{\omega}_{new} - \boldsymbol{\omega}_{old})\|^2$$

where $\frac{\partial y_n}{\partial \omega_k}$ is element Z_{nk} in the matrix $\mathbf{Z}$. This a linear regression problem, and the solution is

$$\boldsymbol{\omega}_{new} = \boldsymbol{\omega}_{old} - \left(\mathbf{Z}^T\mathbf{Z}\right)^{-1}\mathbf{Z}^T\mathbf{e}(\boldsymbol{\omega}_{old})$$

However, if the linear approximation is bad, the step size can become unreasonably large. To overcome this one may minimize a modified error,

$$\begin{aligned} \tilde{E} &= \frac{1}{2}\|\mathbf{e}(\boldsymbol{\omega}_{old}) + \mathbf{Z}(\boldsymbol{\omega}_{new} - \boldsymbol{\omega}_{old})\|^2 + \frac{1}{\eta}\|\boldsymbol{\omega}_{new} - \boldsymbol{\omega}_{old}\|^2 \\ &\Rightarrow \\ \boldsymbol{\omega}_{new} &= \boldsymbol{\omega}_{old} - \left(\mathbf{Z}^T\mathbf{Z} + \frac{1}{\eta}\mathbf{I}\right)^{-1}\mathbf{Z}^T\mathbf{e}(\boldsymbol{\omega}_{old}) \end{aligned}$$

If $\eta \ll 1$ we get back the gradient descent method with learning rate η.

Note: Line minimizataion methods are all batch algorithms, i.e. they use all training data to compute ∇E. In order to exploit the direction of the gradient as much as these methods do, that gradient had better be a very reliable estimate!

2.6.4 Second order methods

Second order methods typically use a local quadratic approximation to the error function. Assume that we Taylor expand $E(\boldsymbol{\omega})$ around some point $\boldsymbol{\omega}_o$,

$$E(\boldsymbol{\omega}) \approx E(\boldsymbol{\omega}_o) + (\boldsymbol{\omega} - \boldsymbol{\omega}_o)^T\nabla E|_{\boldsymbol{\omega}_o} + \frac{1}{2}(\boldsymbol{\omega} - \boldsymbol{\omega}_o)^T\mathbf{H}(\boldsymbol{\omega} - \boldsymbol{\omega}_o) \tag{2.39}$$

where the elements of the *Hessian* matrix $\mathbf{H}$ is defined as

$$H_{ij} = \left.\frac{\partial^2 E}{\partial \omega_i \partial \omega_j}\right|_{\boldsymbol{\omega}_o}$$

Now suppose that $\boldsymbol{\omega}_o$ is a local minimum of E, then

$$\nabla E|_{\boldsymbol{\omega}_o} = 0$$

This will then reduce eqn. 2.39 to

$$E(\boldsymbol{\omega}) \approx E(\boldsymbol{\omega}_o) + \frac{1}{2}(\boldsymbol{\omega} - \boldsymbol{\omega}_o)^T \mathbf{H} (\boldsymbol{\omega} - \boldsymbol{\omega}_o)$$

Thus a local approximation of the gradient around the minimum can be written

$$\nabla E \approx \mathbf{H}(\boldsymbol{\omega} - \boldsymbol{\omega}_o)$$

In the so called Newton method we solve for $\boldsymbol{\omega}_o$ in this expression to obtain

$$\boldsymbol{\omega}_o = \boldsymbol{\omega} - \mathbf{H}^{-1}\boldsymbol{g} \tag{2.40}$$

where we have defined $\boldsymbol{g} = \nabla E|_{\boldsymbol{\omega}}$. Because the local quadratic approximation is not always a good approximation one usually have to use this method in an iterative fashion. The iterative mathod must adress some potential problems:

- It can be computationally demanding to compute $\mathbf{H}$.
- $\mathbf{H}$ must be inverted, which may be numerically difficult.
- If $\mathbf{H}$ is not positive definite ($\boldsymbol{\omega}^T \mathbf{H} \boldsymbol{\omega} > 0 \quad \forall\, \boldsymbol{\omega}$) then the Newton step can point away from the local minimum. This can be the case if we are not close enough to the local minimum.

One simple and somewhat drastic solution is to use a diagonal approximation of the Hessian. This would result in the following update equation for the weights,

$$\Delta \boldsymbol{\omega}_i = -\left(\left| \frac{\partial^2 E}{\partial \boldsymbol{\omega}_i^2} \right| + \lambda \right)^{-1} \cdot \frac{\partial E}{\partial \boldsymbol{\omega}_i}$$

where λ is a small, positive constant.

Quasi Newton

The quasi Newton method builds up an approximation to the inverse Hessian over a number iterations. It has the following nice properties,

- One generates $\mathbf{G}(t)$ matrices that are better and better approximations of $\mathbf{H}^{-1}$.

- The method only uses gradient information to compute $\mathbf{G}(t)$.
- $\mathbf{G}(t)$ is always positive definite.

A common method for computing $\mathbf{G}(t)$ is the so called BFGS (Broyden-Fletcher-Goldfarb-Shanno) method,

$$\mathbf{G}(t+1) = \left(\mathbb{1} - \frac{\mathbf{v}\mathbf{p}^T}{\mathbf{p}^T\mathbf{v}} \right) \mathbf{G}(t) \left(\mathbb{1} - \frac{\mathbf{p}\mathbf{v}^T}{\mathbf{p}^T\mathbf{v}} \right) + \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{p}^T\mathbf{v}}, \tag{2.41}$$

where

$$\begin{aligned} \mathbf{p} &= \boldsymbol{\omega}(t+1) - \boldsymbol{\omega}(t) \\ \mathbf{v} &= \boldsymbol{g}(t+1) - \boldsymbol{g}(t) \end{aligned}$$

The update rule for the quasi Newton method is then,

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \alpha(t)\mathbf{G}(t)\boldsymbol{g}(t)$$

where $\alpha(t)$ is determined by a line minimization.

2.6.5 General considerations regarding minimization methods

There are many different minimization methods when it comes to training neural networks. Stochastic gradient descent (SGD) can be seen as a baseline, with momentum and Adam as popular expansions. More advanced methods, involving line minimization and/or second order derivatives, may not be computationally feasible to use if the dataset or the network is large. That is typically the case in deep learning. For small datasets and shallow networks, advanced methods may be very efficient. (On the other hand, for smaller datasets and easier tasks, training may not need that much optimizaiton...)

2.7 Radial basis function networks

This brief description of radial basis function (RBF) networks are provided mainly for historical reasons and to show that alternatives to the MLP architecture exists. It is probably fair to say that RBF networks are not used so much today.

2.7.1 The interpolation problem

The so-called *radial basis function technique* interpolates between known data points $\mathbf{x}_n$, each with a target value d_n, by introducing weights $\boldsymbol{\omega}$ and a class of *radial basis functions*

φ, and creating the interpolation

$$F(\mathbf{x}) = \sum_n \omega_n \varphi(|\mathbf{x} - \mathbf{x}_n|).$$

Micchelli's theorem states that a large class of radial basis functions can be used. The most commonly used in ANN applications is the Gaussian.

$$\varphi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right)$$

2.7.2 RBF neural networks

We get a RBF neural network by introducing a set of modifications to the above idea:

1. The number of basis functions is usually smaller than the number of data points.
2. The centers $\tilde{\boldsymbol{\omega}}_j$ of the basis functions need not be data points. Determination of suitable centers is part of the training procedure.
3. $\sigma \to \sigma_j$, meaning that we introduce individual widths for each basis function.
4. A linear output node represents $F(\mathbf{x})$, but a bias parameter is included in the linear sum.

This leaves us with a RBF network with a single hidden layer, where each node represents a radial basis function. For pattern n, the value of hidden node j is

$$h_j(\mathbf{x}_n) = \varphi(|\mathbf{x}_n - \tilde{\boldsymbol{\omega}}_j|/\sigma_j).$$

Contour surfaces to φ are hyper-spheres around the center $\tilde{\boldsymbol{\omega}}_j$. Since $\tilde{\boldsymbol{\omega}}_j$ specifies where the argument to φ is 0, we do not need a bias $\tilde{\omega}_{j0}$. Instead, we need a scale parameter σ_j to determine the spacing between the contour surfaces.

RBF networks are designed for regression tasks, so the output node is typically a "normal" node with linear activation function and a bias,

$$y_n = y(\mathbf{x}_n) = \sum_j \omega_j h_j(\mathbf{x}_n) + b.$$

2.7.3 Learning strategies

One option is to simply train all parameters using gradient descent, using an MSE-based loss function

$$E(\boldsymbol{\omega}) = \frac{1}{N}\sum_n (y_n - d_n)^2 .$$

With

$$y_n = \sum_j \omega_j \exp\left(-\frac{||\mathbf{x}_n - \tilde{\boldsymbol{\omega}}_j||^2}{2\sigma_j^2}\right) + b_i$$

The parameters are updated according to

$$\begin{aligned}
\Delta\omega_{ij} &= -\eta_1 \frac{\partial E}{\partial \omega_{ij}} \\
\Delta\tilde{\omega}_{jk} &= -\eta_2 \frac{\partial E}{\partial \tilde{\omega}_{jk}} \\
\Delta\sigma_j &= -\eta_3 \frac{\partial E}{\partial \sigma_j}
\end{aligned}$$

A drawback with this learning strategy is speed. It usually takes much longer to train this way.

Another option is therefore to first determine centers $\tilde{\boldsymbol{\omega}}_j$ as random selections of data points, or as centers appearing in some clustering method (*e.g.*, K-means clustering). The widths of the basis fucntions must be chosen so that they are not too peaked or too flat. One suggestion is $\sigma_j = \frac{1}{M}\max_{j'}(|\tilde{\boldsymbol{\omega}}_j - \tilde{\boldsymbol{\omega}}_{j'}|)$, where M is the number of hidden nodes. Once the centers and widths have been selected, the values of hidden nodes h_{nj} can be seen as elements in a matrix $\mathbf{H}$, and the hidden-to-output weights can be determined by exactly solving the linear regression task $0 = \mathbf{H}\mathbf{H}^T\boldsymbol{\omega} - \mathbf{H}^T\mathbf{d}$. We discussed more details when presenting the simple perceptron.

2.7.4 RBF versus MLP

- Each node in a RBF is very local (it influences the output mainly for $\mathbf{x} \approx \tilde{\boldsymbol{\omega}}_j$). The argument $\tilde{\boldsymbol{\omega}}_j^T\mathbf{x} + \tilde{b}_j$ to an MLP node is more global.
- The MLP can have a complicated structure (many hidden layers), where RBF has a simple 1-hidden layer structure.
- For MLP:s all weights are determined in one learning session, but for the RBF the weights are typically determined at different stages.
- RBF suffers more from the "curse of dimensionality" than the MLP does.

Chapter 3

Generalization

3.1 Introduction

The central challenge of machine learning is to perform well on previously unseen data, not part of the training. A model with such ability is said to *generalize*. One obviously have to define the precise meaning of "new" data. Without going into mathematical details it is reasonable to require that new data must come from the same process that generated the training data.

3.2 Overfitting

One complication that can limit the generalization performance of an ANN model is *overfitting*. We can say that overfitting is a result of having a model that conforms, in too much detail, to the training set. Figure 3.1 is an illustrations of the concept of overfitting.

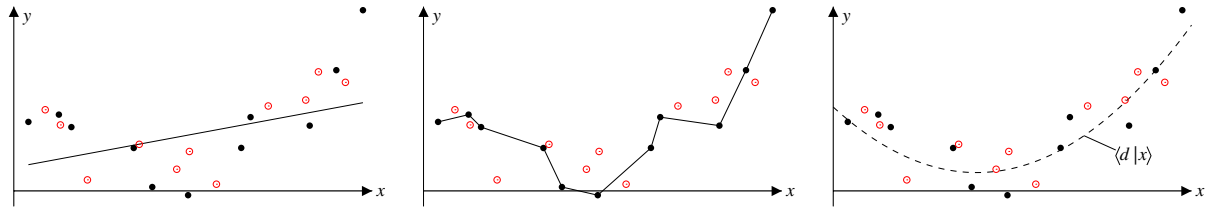


Figure 3.1: How training on black data can fit differently to new (red) data. Left: an underfitted model fits poorly to its own data, and also new data. Middle: an overfitted model fits very well to training data, but may fail on new data. Right: ideally, the ANN should be able to find $\langle d|x\rangle$ from training data, to optimize the fit to new data.

3.3 Estimating generalization: Validation

It is very important to estimate generalization performance, but how do we do that without any "new" data available? There are some different ways.

3.3.1 Holdout method

Simply split the dataset into two parts, a training set and a validation set. A model is trained

Figure 3.2: The holdout method. One part is for training and one is for validation.

on the training set and the generalization performance is estimated using the validation set.

This is (still) a very common method, but it is a bit unreliable if the total amount of data is not large enough.

- If you set off too little data for validation, the validation result is unreliable.
- If you set off too much data for validation, training is performed on a small dataset, and is not representative of what you can accomplish with all your data.

When there is plenty of data avialbale a simple hold-out method still works, and typically ∼1/3 is set aside for validation. However, there are other (more time consuming methods) that allow you to train on more data and still get a reliable generalization estimate. We present K-fold cross validation and Bootstrap.

3.3.2 K-fold cross validation

Split (randomly) the dataset into K parts of (approximate) equal size. We will now train K models on K slightly different training sets and validate on K different validation sets. Figure 3.3 illustrates how the different training and validation sets are defined. The final estimate

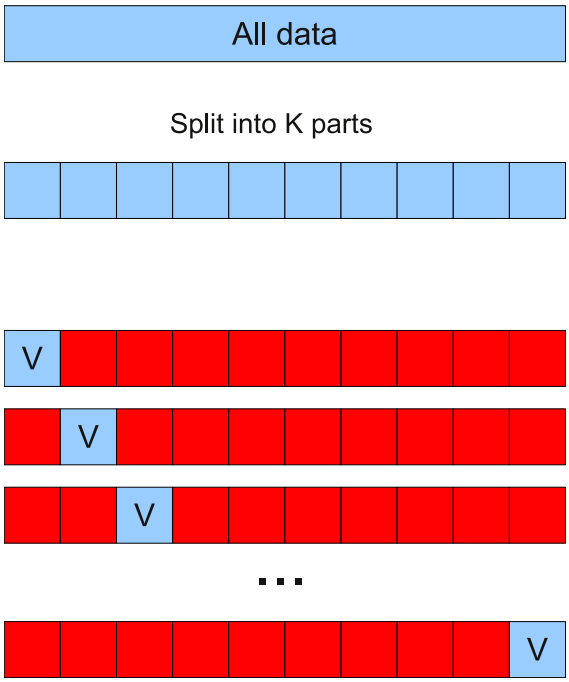


Figure 3.3: The K-fold cross validation method. Each of the K validation parts (V) are used as a validation set when training on the remaining parts.

of the generalization performance is given by the average of the K validation results. Let P_i denote the performance measured on validation set i when training on the corresponding training set i. Then $\hat{P}$, the estimate of the true P is given by,

$$\hat{P} = \frac{1}{K} \sum_{i=1}^{K} P_i$$

Often the procedure is repeated for C "cycles" with different K-fold splits each cycle, in order to further enhance the estimation. The estimation then becomes,

$$\hat{P} = \frac{1}{CK} \sum_{c=1}^{C} \sum_{i=1}^{K} P_{ic}$$

where P_{ic} being the validation performance on split i for cycle c.

In this way, you train on a large fraction of your data, so that the validated models are representative of what you can accomplish with the data set. The average performance over many validation sets becomes a reliable estimate of generalization performance.

A typical procedure would do 3-fold to 10-fold cross validation, but when the data set is small, you sometimes do "leave-one-out" cross validation, where K equals the number of samples. In that special case there is no randomness in the K-fold splitting, so there is no point in doing it for more than one cycle.

Stratified partitioning

If there are very few samples of one class, some validation sets might be completely without it. Even worse, training could be without representatives for the rare class. To avoid that, one can make *stratified* partitioning, where the balance between classes is preserved in each fold.

3.3.3 Bootstrap

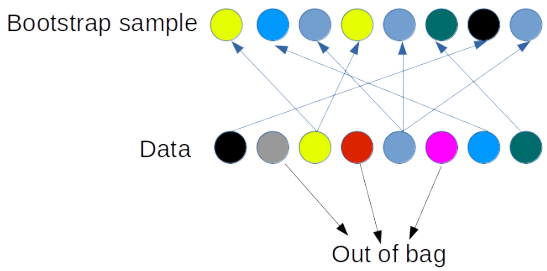


Figure 3.4: A bootstrap sample is generated by randomly selecting from data with replacement, allowing repeated occurance of the same sample, while other samples are not included at all ("out-of-bag").

A bootstrap sample of the dataset (of size N) means resampling the data set, *with replacement*, to produce a new dataset of size N. There will hence be duplicate points in the

bootstrap sample. Let P_b^* be the performance measured on the samples *not* part of the b:th bootstrap sample (the "out-of-bag" samples). The bootstrap estimate is then given by

$$\hat{P}^{boot} = \frac{1}{B}\sum_{b=1}^{B} P_b^*$$

One can argue the this estimate is too pessimistic and an adjustment has been suggested, which is called the *bootstrap 632* rule, where

$$\hat{P}^{632} = 0.368\hat{P}^a + 0.632\hat{P}^{boot}$$

where $\hat{P}^a$ is the training performance averaged over all bootstrap samples. The 632 rule have some problems with very overfitted models where it often gives a to optimistic estimation. The .632 factor comes from the fact that for large N this is the probability that a given data point is part of the bootstrap sample (see exercise).

3.4 Performance Measures

We have proposed a set of loss functions for ANN training, and they can of course be used to measure the performance on a validation set. However, it is not necessary. There is no need to take the derivative of a validation result, and it can be a good idea to select a performance measure that is closer to the intended application of the ANN.

3.4.1 Regression

In regression problems, the mean squared error is still a very reasonable measure of performance. In order to become independent of an arbitrary scaling of target values, it is a good idea to compare MSE to the variance of target values, and use the *Coefficient of determination*, $R^2 = 1-\text{MSE/Var}$, as performance measure. If R^2 is much smaller than 1, it reveals that the ANN has a hard time finding any useful information in the inputs. This can be understood as follows: Imagine there is no useful signal in the input data, so that the best ANN will produce the same output for all patterns. When the ANN minimizes the MSE-based error $E \propto \sum_n (d_n - y_n)^2$ it will then train towards $y_n = \overline{d}$ and we get MSE equal to a variance estimate, so that $R^2 = 0$. It is even possible to get negative R^2, revealing severe overtraining issues.

Notice that MSE and Var must be calculated with precisely the same data, in this case the validation set. Even if we pre-processed our training data to unit variance, the variance in the validation set could be slightly different.

3.4.2 Classification

In binary classification, our standard ANN design will give countinous outputs y_n, constrained to be between 0 and 1. The assignment to classes must depend on a threshold c, so that all patterns with $y_n > c$ are assigned to class 1. However, the cross-entropy error used during training does not need to specify c, so the threshold remains to be determined when the network model is completed.

Once a threshold is set, it is possible to write down the *confusion matrix* that compares predicted classes to actual classes

	Predicted Positive $y_n > c$	Predicted Negative $y_n \leq c$
Actual Positive $d_n = 1$	TP	FN
Actual Negative $d_n = 0$	FP	TN

Table 3.1: The confusion matrix, displaying the number of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN).

More or less every combination of these numbers you can invent has been investigated and has at least one name. Some examples are

- Sensitivity TP/(TP+FN) (fraction of actual positivies that were correctly predicted).
- Specificity TN/(TN+FP) (fraction of actual negatives that were correctly predicted).
- Accuracy (TP+TN)/(TP+FN+TN+FP) (fraction of correct predictions).
- Odds ratio (TP·TN)/(FN·FP) (the "odds" you get a positive right divided by the odds you get a negative wrong).

These are often used as validation performance measures. There are standard ways of using these measures in hypothesis testing, to generate p-values.

Instead of deciding on a threshold, it is possible to decide on a certain confusion matrix result, and then use another one as performance measure. For example, it could be decided beforehand that a 90% sensitivity is desired. Then, for every ANN model considered, the threshold is first set to achieve 90% sensitivity, and models could be compared by their specificity.

It is possible to measure performance without a threshold, and still get a measure that is more interpretable than the cross entropy error. One option is the "Area under the Curve" (AUC). It uses the "Receiver Operator Characteristic" (ROC) curve, which plots sensitivity versus

1−specificity. (As we change the threshold, both sensitivity and specificity will change. It is then possible to plot them against each other.) The AUC is the probability that a randomly selected sample from class 1 has a higher output y than a randomly selected sample from class 0. If the AUC is 1, then the samples are perfectly sorted by the output, and there exists a range of thresholds that give both sensitivity and specificity 1. If the AUC is 0.5, it is not sorting the samples better than a random guess. The AUC is mathematically equivalent to the *Wilcoxon rank sum*.

For categorical classification it common to assign pattern n to the class i with the largest output y_{ni}. However, the outputs could first be weighted by class prevalence. If you desire to avoid determining thresholds, the AUC can be generalized for multiple classes, using the *Kruskal-Wallis test*.

3.5 Improving generalization

3.5.1 Bias-Variance tradeoff

We start this section by looking at the bias-variance tradeoff which will motivate a class of methods designed to prevent overfitting without instead becoming underfitted.

Consider a dataset $\mathcal{D}$ of size N. Also assume that we have many such N-sized datasets $\mathcal{D}$. A measure of how close the ANN function is to the target at input $\mathbf{x}$ is

$$(y_{\mathcal{D}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2$$

where $y_{\mathcal{D}}(\mathbf{x})$ denotes the ANN function one obtains using dataset $\mathcal{D}$. To avoid the dependence on a particular dataset $\mathcal{D}$ we take the expectation value over all possible $\mathcal{D}$'s, which we denote

$$\mathrm{E}\left[(y_{\mathcal{D}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2\right].$$

We can investigate this expectation value using the well-known formula for variance, $\mathrm{Var}[z] = \mathrm{E}[z^2] - \mathrm{E}[z]^2$, with let $z = y_{\mathcal{D}} - \langle d\rangle$. Since $\langle d\rangle$ is independent of dataset $\mathcal{D}$, so that $\mathrm{E}[y_{\mathcal{D}} - \langle d\rangle] = \mathrm{E}[y_{\mathcal{D}}] - \langle d\rangle$ and $\mathrm{Var}[y_{\mathcal{D}} - \langle d\rangle] = \mathrm{Var}[y_{\mathcal{D}}]$, we get

$$\mathrm{E}\left[(y_{\mathcal{D}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2\right] = \underbrace{\Big(\mathrm{E}[y_{\mathcal{D}}(\mathbf{x})] - \langle d|\mathbf{x}\rangle\Big)^2}_{\mathrm{bias}^2} + \mathrm{Var}[y_{\mathcal{D}}(\mathbf{x})]. \tag{3.1}$$

- The bias term measures how much the average of different ANN models differs from the target function $\langle d|\mathbf{x}\rangle$.
- The variance term measures how sensitive $y_{\mathcal{D}}(\mathbf{x})$ is to the datasets $\mathcal{D}$.

The above expressions for the bias and the variance still have an $\mathbf{x}$ dependence. To get the overall tradeoff we need to integrate over $\mathbf{x}$, taking into account a distribution $p(\mathbf{x})$ generating the inputs of the datasets,

$$\begin{aligned} \text{bias}^2 &= \int \Big(\mathrm{E}\left[y_{\mathcal{D}}(\mathbf{x})\right] - \langle d|\mathbf{x}\rangle \Big)^2 p(\mathbf{x}) d\mathbf{x} \\ \text{variance} &= \int \mathrm{Var}\left[y_{\mathcal{D}}(\mathbf{x})\right] p(\mathbf{x}) d\mathbf{x} \end{aligned}$$

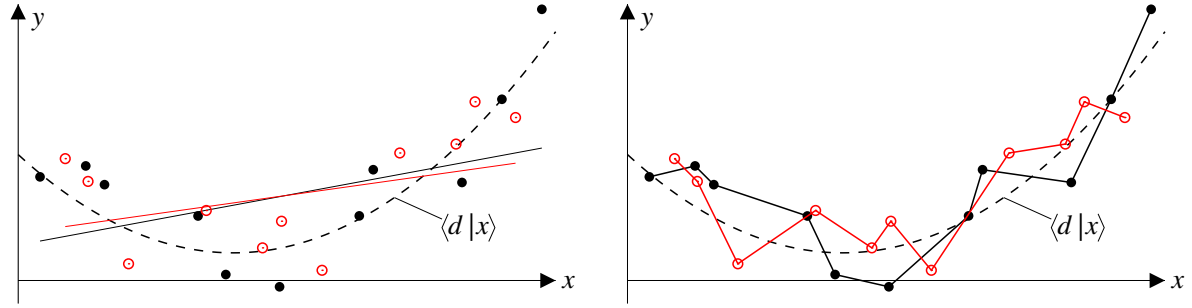


Figure 3.5: Illustration of the bias-variance tradeoff. Left: the linear models for different datasets are essentially the same, so there is a low variance. However, the average of models is nowhere near the target function $\langle d|\mathbf{x}\rangle$, so the bias is high. Right: The average of models for many datasets (the black and red and many others) will eventually approach the target function $\langle d|\mathbf{x}\rangle$ so the bias is low, or even 0. However, the models for different datasets are drastically different, so there is a high variance, which means that none of the models on their own are good at describing other data sets.

The bias-variance tradeoff is illustrated in figure 3.5. For ANN models this usually means that in order to have low bias one needs a flexible model, e.g. a large number of hidden nodes. A flexible model however means that we can fit the ANN model very well to a specific dataset $\mathcal{D}$, i.e. the variance increases. In order to have a low variance we can restrict the ANN model by e.g. having very few hidden nodes, but this also means that we may increase the bias since the ANN model may be too limited.

Another way to put the bias-variance tradeoff is in terms of function curvature (second order derivatives). Each hidden node can provide curvature only in a limited range of input space (near the region where the argument to the activation function is zero). In order to fit a target function $\langle d|\mathbf{x}\rangle$ with a small curvature everywhere, many nodes are needed, but the danger is then that the ANN creates a much too large curvature in some input regions. In the next section we will try to control the bias-variance tradeoff using *regularization*, which allows for many nodes but suppresses their ability to create large curvatures.

3.5.2 Regularization terms

The first types of regularization we are going to look at modifies the loss function according to,

$$\tilde{E}(\boldsymbol{\omega}) = E(\boldsymbol{\omega}) + \alpha\Omega(\boldsymbol{\omega})$$

where Ω is the regularization term and the *regularization strenght* α controls the amount of regularization. From section 2.5.4, we remember that we can interpret the regularization function Ω in terms of a Bayesian prior $\Omega \propto -\ln p(\boldsymbol{\omega})$.

Notice that the bias term b in the node output $\varphi(\sum_k \omega_k x_{nk} + b)$ (so: not "generalization bias", but "node activation argument bias") does not contribute to output curvature, and can be expected to be governed by a different prior than the weights in the network. Therefore, Ω is usually only a function of the weights on links between nodes.

We are going to look at three Ω terms.

L2 norm regularization (weight decay)

The weight decay term is as follows,

$$\Omega = \frac{1}{2}\sum_i \omega_i^2$$

where the sum runs over all weights in the network (except biases). This corresponds to a Gaussian prior $p(\boldsymbol{\omega}) \propto \exp\left(-\frac{1}{2\sigma_\omega^2}\sum_i \omega_i^2\right)$.

Modified L2 norm (weight elimination)

The modified L2 norm is

$$\Omega = \frac{1}{2}\sum_i \frac{(\omega_i/\omega_o)^2}{1 + (\omega_i/\omega_o)^2}$$

where ω_o is a new parameter often set to 1. A plot of this term is shown in figure 3.6. This allows for some large weights. Formally, it does not correspond to a normalizeable prior $p(\boldsymbol{\omega})$, but the Bayesian interpretation is not necessary.

L1 norm regularization (lasso)

The lasso regularization is as follows,

$$\Omega = \sum_i |\omega_i|,$$

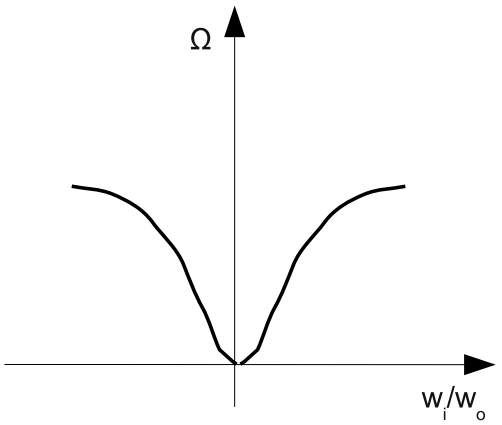


Figure 3.6: The weight elimination regularization term.

corresponding to an exponential prior $p(\boldsymbol{\omega}) \propto \prod_i \exp(-|\omega_i|/\beta)$, where the unkown β is represented by the regularization strength α multiplied to Ω. A plot of this term is shown in figure 3.7. A difference compared to weight decay is that the derivative of the lasso term is constant and does not go to zero as the weight approaches zero. This means that weights that are not needed will be forced to zero. As we will discuss in section 3.5.7, Lasso can therefore be used as *feature selection* method!

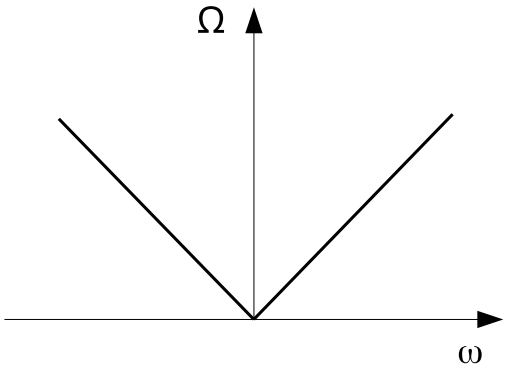


Figure 3.7: The lasso regularization term.

Max-norm regularization

The max-norm method puts a constraint on individual node weight vectors. Denote by $\boldsymbol{\omega}_j$ the vector of all weights feeding into node j. Max-norm then introduces the constraint,

$$|\boldsymbol{\omega}_j| \leq c$$

where c is a parameter, typically of size 2-4. The constraint is enforced during training after each weight update, by simply multiply $\boldsymbol{\omega}_j$ with $c/|\boldsymbol{\omega}_j|$ if the constraint is violated.

Max-norm is not implemented via a function Ω, so it does not quite fit into this section about "regularization terms", but the procedure is actually related to a prior $p(\boldsymbol{\omega})$ which for each node j is uniform within the hypersphere of $|\boldsymbol{\omega}_j| \leq c$ and zero elsewhere. Admittedly, it is very unusual to introduce a prior which is explicitly zero for a possible range of values, so the connection to Bayesian formalism should not be taken too seriously.

3.5.3 Ensemble averaging

One way to smooth the output function $y(\boldsymbol{\omega}, \mathbf{x})$ is to combine many networks to form an *ensemble of networks* (or sometimes called committee machines). We can then treat the average of all model outputs as an ensemble output:

$$y_{\text{ens}}(\mathbf{x}) = \frac{1}{M}\sum_m y_m(\mathbf{x}), \tag{3.2}$$

where there are M members in the ensemble.

To examine generalization, we are interested in how well outputs agree with the target expectation value, and we can study the squared error $(y(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2$. Hiding the $\mathbf{x}$-dependence for a while, we then see that each member's generalization ability could be measured by $(y_m - \langle d\rangle)^2$ and the mean generalization ability is

$$\begin{aligned}\frac{1}{M}\sum_m (y_m - \langle d\rangle)^2 &= \frac{1}{M}\sum_m (y_m - y_{\text{ens}} + y_{\text{ens}} - \langle d\rangle)^2 = \\ &= \ldots = \frac{1}{M}\sum_m (y_m - y_{\text{ens}})^2 + (y_{\text{ens}} - \langle d\rangle)^2,\end{aligned} \tag{3.3}$$

where the final results comes from the fact that $\sum_m y_m = M y_{\text{ens}}$, and that neither y_{ens} nor $\langle d\rangle$ depends on m.

Re-introducing the $\mathbf{x}$-dependence, we can formulate the following important decomposition

of the ensemble error

$$(y_{\text{ens}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 = \tag{3.4}$$

$$= \frac{1}{M}\sum_m (y_m(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 - \frac{1}{M}\sum_m (y_m(\mathbf{x}) - y_{\text{ens}}(\mathbf{x}))^2. \tag{3.5}$$

The error made by the ensemble is the mean error made by the individual networks **minus** the (approximate) variance of the ensemble members. The last term is often called the "diversity term".

Since the relation holds for all $\mathbf{x}$ and datasets $\mathcal{D}$, it will remain after we integrate over $\mathbf{x}$ with an input distribution $p(\mathbf{x})$, and after we take and expectation value $\mathrm{E}[\cdot]$ with respect to datasets $\mathcal{D}$.

Actually, it is quite common that the ensemble performs better than its best member! One reason is that some members may overestimate $\langle d\rangle$, while others underestimate it, so that y_{ens}, which is the mean of all y_m, is closer to $\langle d\rangle$ than any y_m. Another reason is that overall performance involves integration over inputs. Unless the same model m is best for all inputs $\mathbf{x}$, it can easily happen that

$$\min_m \int (y_m(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 p(\mathbf{x}) d\mathbf{x} \; > \; \int (y_{\text{ens}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 p(\mathbf{x}) d\mathbf{x}.$$

To conclude: For the ensemble to work well we want a set of *accurate* ensemble members that *disagree* as much as possible. With this in mind, we can actively introduce extra randomness in our training, to create more diverse (but still reasonably accurate) ensemble members!

Generating an ensemble

Here are a few approaches to create different ensemble members:

1. Different initial conditions: Just train M networks with different (random) initial values of the weights. This may lead to finding different local minima of the loss function, which may produce slightly different networks. Also, with stochastic gradient descent, the networks may end in slightly different places, even if they find the same minimum.

   - This is very easy to implement, but networks are rarely diverse enough.

2. Bagging: Each network in the bagging ensemble is trained on a bootstrap sample of the original training data set ("bagging" stands for bootstrap aggregating). This means that we resample the original training data set *with replacement*. The size of the bootstrap sample is the same as the original one.

- Creates better diversity than random initialization, but the diversity cannot be tuned. It may be too small for very large data sets.
- The variance of ensemble member outputs become estimates of the variance in new datasets of size N.

3. K-fold cross splitting: In this approach we divide the training data set into K parts of (approximately) equal size (as in K-fold cross validation). Ensemble member k is trained on the training set you obtain when leaving out part k. The procedure is illustrated in figure 3.8. The whole procedure can be repeated C cycles with different

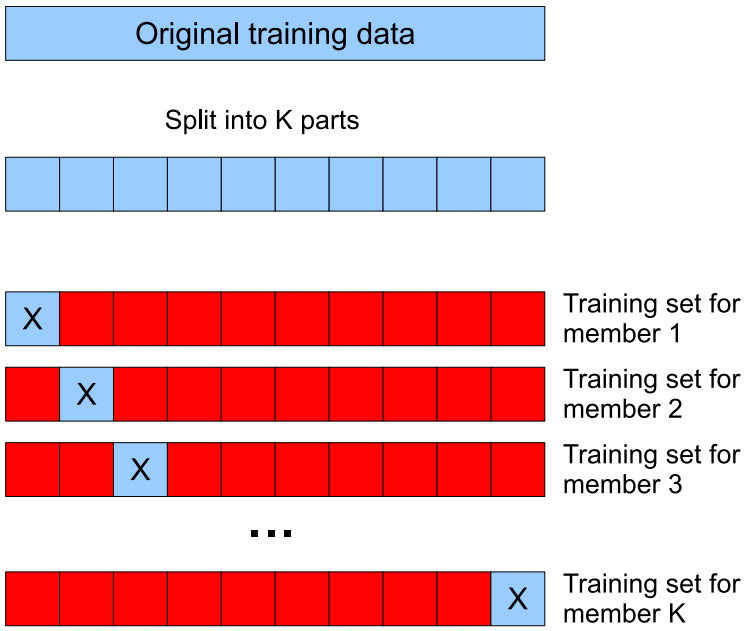


Figure 3.8: The K-fold splitting method for training ensemble members. For each of K splits you remove one part of the data, thereby having K training datasets that are similar, but not identical. If K is small the dataset becomes more different and the diversity should increase.

random split into K parts. As a result we have an ensemble of size $C \cdot K$.

- Slightly faster than bagging since we train each member on fewer data.
- We can select K to adjust the diversity. Maximal diversity is given for $K = 2$.

4. K-fold mini-batches: This is the same approach as K-fold cross-splitting, but here we only train on the k:th part in each K-fold cross split. Looking at figure 3.8 this approach would correspond to train on each blue square with the 'X' and disreagard

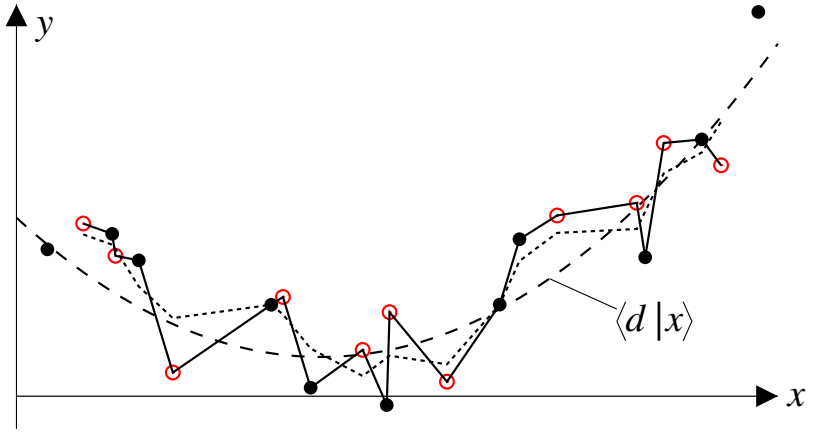


Figure 3.9: Illustration of the ensemble machine. Data is split into to parts (black and red). The average (dotted line) of two models overfitted to separate parts of data can be a better solution than a model trained on all data (solid line).

the rest. Again, the whole procedure can be repeated C cycles with different random split into K parts. As a result we have an ensemble of size $C \cdot K$.

- This approach will generate diverse members and is suitable for large datasets.
- The smallest diversity we can get is with $K = 2$.

With bagging or some K-fold approach, we deliberately reduce the dataset for each ANN in order to improve diversity, as is illustrated in fig. (3.9).

Ensemble cross-splitting versus cross-validation

The K-fold partition method has now been presented twice, which can cause some confusion and blur the distinction between ensembles and cross-valdiation. To clarify:

The purpose of cross-validation is to *detect* overfitting.
The purpose of ensembles is to *reduce* overfitting.

In cross-validation, the omitted fold is used to calculate a validation performance. After that, the trained model is discarded.
In ensemble creation, the trained model is kept, while the validation result on the omitted fold is not even calculated.

3.5.4 Elaborate ensemble methods

In order to regularize networks with ensembles, we have all we need now. In particular, the simple mean of ensemble members will do as ensemble output. However, if there is little risk of overfitting, more elaborate weightings can be considered. We will briefly discuss *optimal weighting* and *boosting*. They are rarely used for regularization: they add a computational effort and actually introduce a risk that the optimized ensemble gets overfitted.

Optimal ensemble weighting

We start with general weights α_i to create a weighted ensemble average,

$$y_{\text{ens}}(\mathbf{x}) = \sum_i^M \alpha_i y_i(\mathbf{x}), \quad \sum_i \alpha_i = 1. \tag{3.6}$$

Introducing errors

$$\epsilon_i(\mathbf{x}) = y_i(\mathbf{x}) - \langle d|\mathbf{x}\rangle,$$

we get the squared ensemble error

$$(y_{\text{ens}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 = (\sum_i \alpha_i\epsilon_i)(\sum_j \alpha_j\epsilon_j) = \sum_{i,j} \alpha_i\alpha_j\epsilon_i\epsilon_j \tag{3.7}$$

The expectation value over many datasets is then

$$\mathrm{E}\left[(y_{\text{ens}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2\right] = \sum_{i,j} \alpha_i\alpha_j \mathrm{E}\left[\epsilon_i(\mathbf{x})\epsilon_j(\mathbf{x})\right] = \sum_{i,j} \alpha_i\alpha_j C_{ij}, \tag{3.8}$$

where the last step defines elemens in the error correlation matrix $\mathbf{C}$.

We now want find the α_i that minimizes this double sum. The normalization contstraint $\sum_i \alpha_i = 1$ can be included using a Lagrange multiplier, and the solution turns out to be

$$\alpha_i = \frac{\sum_j (\mathbf{C}^{-1})_{ij}}{\sum_k \sum_j (\mathbf{C}^{-1})_{kj}}$$

Unfortunately, we don't know C_{ij} exactly. We need to approximate it using an external dataset of size N'

$$C_{ij} \approx \frac{1}{N'} \sum_{n'=1}^{N'} \left(y_i(\mathbf{x}_{n'}) - d_{n'}\right)\left(y_j(\mathbf{x}_{n'}) - d_{n'}\right)$$

So, optimal weighting has the disadvantage of having parameters that should be trained using an extra dataset different from the training dataset to avoid overfitting. Since the extra dataset "trains" the weights α_i, it cannot be used for validation.

Another problem with this approach is that we may get solutions with large positive and negative weighting factors α_i. To overcome this, we can introduce yet another constraint on the α's:

$$\sum_i \alpha_i = 1 \qquad \text{and} \qquad \alpha_i > 0 \; \forall i,$$

but then the optimization problem is more difficult and cannot be solved by just including a Lagrange multiplier.

Boosting

Boosting is a general idea that can be applied to many machine learning algorithms that do supervised learning.

We will look at the well-known AdaBoost (Adaptive Boosting) (meta)-algorithm, which can be used to improve the performance of other learning algorithms. It can be regarded as an ensemble learning algorithm since it based on the predictions of many individual members, called "learners". It constructs the ensemble by putting more and more emphasis on certain data points.

We focus on a binary classification problem, and we let our output be binary. We could for example use a threshold function for the output, or turn a continuous output into a binary variable using a threshold value.

AdaBoost

1. Initialize a set P_1 of uniform sample weights, i.e.

$$P_{1n} = \frac{1}{N} \quad \forall n$$

2. Do the following for $t = 1, 2, ..., T$, where T is a predifined max number of iterations

   (a) Call the learning algorithm for your learner using P_t. For a neural network, we could introduce P_{tn} as weights in the loss function, $E(\boldsymbol{\omega}) \propto \sum_n P_{tn} E_n(\boldsymbol{\omega})$. Then, the trained weights $\boldsymbol{\omega}_t$ will depend on P_t and the outputs $y_t(\mathbf{x})$ will also depend on P_t.

   (b) Calculate the weighted accuracy a_t according to:

$$a_t = \frac{\sum_{n:y_t(\mathbf{x}_n)=d_n} P_{tn}}{\sum_n P_{tn}}$$

(c) Update the sample weights P_{tn}

$$P_{t+1,n} = \begin{cases} P_{tn}, & y_{tn} = d_n \\ P_{tn}\frac{a_t}{1-a_t}, & y_{tn} \neq d_n \end{cases}$$

If the accuracy a_t is larger than 0.5, we increase the weight of misclassified samples, and give the next learner a harder task. If the accuracy is below 0.5 the next learner gets an easier task. A plausible and appealing scenario is that the accuracy starts above 0.5 (already the first model should do better than average) and as more and more emphasis is put on the difficult samples, the weighted accuracy approaches 0.5 from above. The changes of sample weights then get smaller and the weight vector P_t converges towards a steady state with "optimal" emphasis on difficult patterns.

(d) Set the learner weight

$$\alpha_t = \log\left(\frac{a_t}{1-a_t}\right).$$

A learner with higher accuracy a_t gets a higher weight α_t.

3. The output from the final classifier is given by

$$y_{\text{AdaBoost}}(\mathbf{x}) = \arg\max_d \sum_{t:y_t(\mathbf{x})=d} \alpha_t$$

So the output of $y_{\text{AdaBoost}}(\mathbf{x})$ is the label d that maximizes the sum of learner weights α_t.

AdaBoost is not the most popular boosting algorithm. Today *Gradient Boosting* and *Extreme Gradient Boosting (XGBoost)* are more used, where the latter is the current state-of-the-art boosting method. It is beyond the scope of this course to give a more detailed description of them.

Ensembles with dynamic structures

In the ensembles we have presented, several predictors (in our case, ANNs) are combined in way that is *not* influenced by the inputs. These are referred to as *static structures*. It is also possible to create *dynamic structures*, where the input signal is used in the mechanism that combines the outputs of the individual predictors into the final output. Without presenting them, we just mention two possible approaches, which are *Mixture of experts* and *Hierarchical mixture of experts*.

Bayesian ensembles

As discussed in section 2.5.4, a Bayesian ensemble aims at predicting the entire conditional target distribution $p(d|\mathbf{x})$, and if successful, it will provide a good generalizing estimate of the expectation value $\langle d|\mathbf{x}\rangle$. Creating a Bayesian ensemble can however be computationally expensive and require a large data set.

3.5.5 Dropout regularization

The dropout method was introduced 2013, by Hinton et al. It can be seen as an efficient regularization technique to avoid overtraining. It can also be seen as a huge ensemble of many different networks.

The term dropout refers to temporarily removing nodes in a network, so that all weights feeding into and out from the node are also temporarily removed, as shown in fig. (3.10). The nodes to drop are selected at random, usually with a fixed parameter p that is the

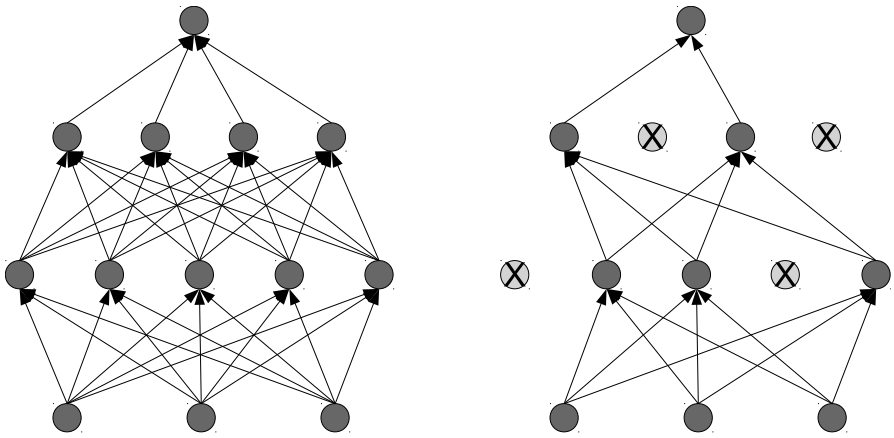


Figure 3.10: (left) An MLP with two hidden layers. (right) The same network but this time a few of the hidden nodes have been (temporarily) removed along with all of the associated weights.

probability to *keep* nodes. So if $p = 0.8$ it means that, in average, 1/5 of the nodes are dropped. We can use dropout in all layers of the network, except the output layer. It is common to have one p parameter for each layer in the network.

Training with dropout Each time we consider a new pattern during training we will apply the dropout procedure, meaning that each node will be subject to a removal with a probability of $1 - p$. This results in a thinned network that will be used during the forward and backpropagation steps. A new thinned network will be sampled for each new pattern

used during training. One can summarize the dropout procedure for stochastic gradient descent as,

1 Set mini-batch size M for SGD

2 Select M random patterns for the mini-batch.

3 For each of the M patterns do,

   a Sample a thinned network by dropping random nodes according to p.

   b Use this thinned network to compute all weight updates for this pattern and add to the total weight update. Weights that are not used because any of its associated nodes are dropped are omitted.

4 Average the weight update by the number of times a given weight was part of a thinned network and perform the update. Go back to item 2.

Using a network trained with dropout The dropout technique is only used during training of the network, when testing the network all of the nodes are kept as usual.

We can see dropout as training a collection of thinned networks with extensive weight sharing. With P droppable nodes in the network, there are 2^P possible thinned networks, but the dropout probabilities p make some of them much more likely than others. For new data, it turns out to be unnecessary to generate an ensemble of thinned neworks. Instead, we use a re-weighted version of the entire network. If a node was present during training with probability p, then all of the weights feeding out from this node are multiplied with p, as illustrated in figure 3.11. In this way, the node provides its average contribution to all its outgoing nodes, and the final output becomes the output of an "average" thinned network.

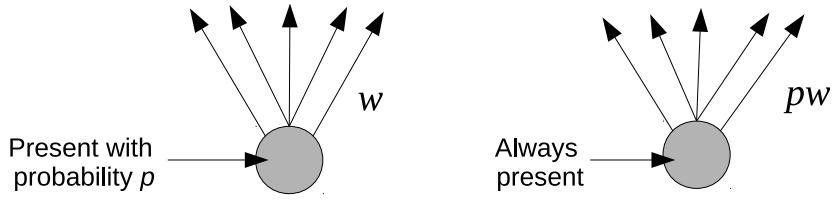


Figure 3.11: (left) During training a node is present with probability p. (right) During testing the node is always present, and to make the expected output to agree in both cases the outgoing weights are multiplied by p.

An example Figure 3.12 shows two examples of varying the number of hidden nodes for the Pima Indians classification problem. In the left graph we see the expected behavior of an increasing validation error as we increase the number of hidden nodes. At the same time the training error goes down. A typical example of overtraining! In the right graph we have the same network setup except we are now using $p = 0.5$ dropout on the hidden layer. First note the different scale and the fact that regardless of the number of hidden nodes the validation error approximately stays the same.

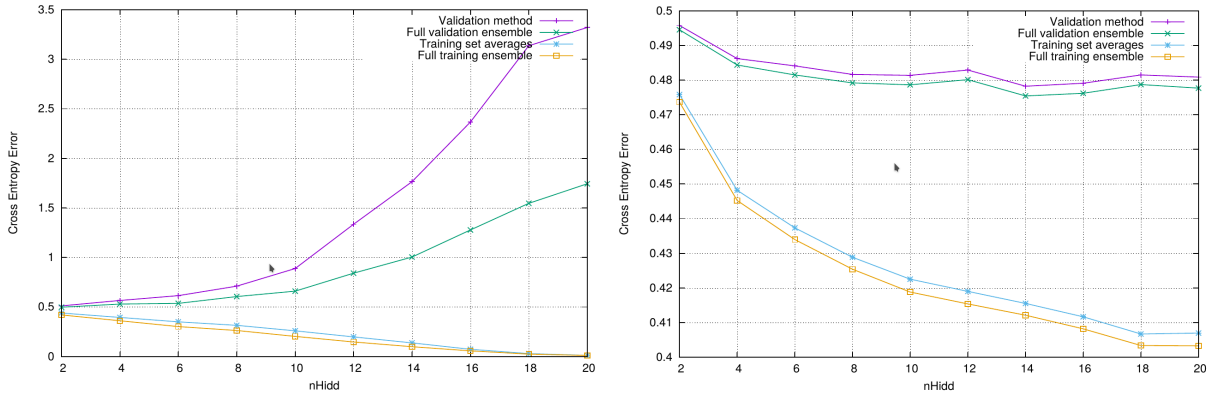


Figure 3.12: Training and validation error as a function of the number of hidden nodes for the Pima Indians classification problem. The left experiment is without dropout and the right is with dropout ($p = 0.5$) for the hidden layer. There are four curves, two validation curves and two training curves, representing an ensemble or no ensemble. We can see in the left graph that the ensemble prevents some overtraining, but not all of it.

Why does dropout work? One way to understand how dropout works is to consider an overfitted model, that fits to an outlier by introducing very high curvatures in a region near the outlier. This requires *at least two nodes* with large weights. One node sends the output function away to the outlier, the other node brings the output back to more typical values. With dropout, no two nodes are trained together all the time, and they cannot combine their weights to fit to an outlier. This becomes an indirect suppression of large weights and high curvature.

When we perform dropout on the input layer, we can view it as an ensemble: one way to create diverse ensemble members is to let then train on different subsets of input variables. (A very succesful machine learning method exploiting this idea is the *random forest*.)

3.5.6 Early stopping

Another alternative to regularization terms as a way of controlling the complexity of a network is the procedure of *early stopping*. A typical behavior during the minimization process can be seen in fig. (3.13). If we estimate the generalization error using a separate

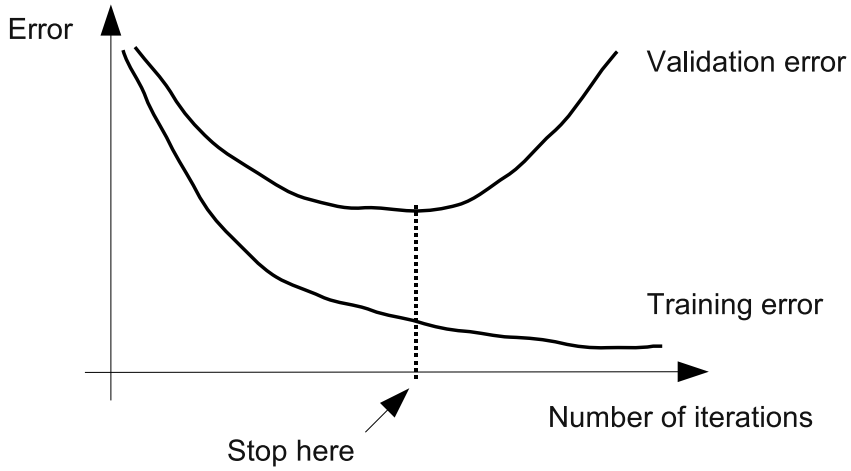


Figure 3.13: The training and the validation error during the minimization process. The "early stop" procedure halts the training when the validation error have reached a minimum.

validation set we can stop the training when the validation error starts to increase.

With early stopping we let the validation data strongly influence our training. There is a risk that our model becomes overfitted to the validation set. This will be discussed more in section 3.6.

3.5.7 Pruning and feature selection

Pruning

Pruning (the gardening term for cutting away branches on a tree) means removing some nodes or links after training, if they are considered unimportant. This can help avoid overfitting. It can also make the network model more comprehensible for a human, and it slightly speeds up the use of the network on new data (since it has become a bit smaller).

Essential in pruning is to decide which links and nodes to remove. One way is to introduce a *very small* threshold c and remove all links k with weight magnitudes $|\omega_k| < c$. It is popular to combine this with the lasso regularization term $\Omega = \alpha \sum_k |\omega_k|$, since it not only reduces weight values, but drives them towards zero.

A different approach is "optimal brain damage" (OBD), which assumes that the network is optimized so that all loss derivatives $\frac{\partial E}{\partial \omega_k}$ are zero. The leading term in a Taylor expansion that measures how E depends on ω_k would then be the second order derivative. As discussed before, creating the full Hessian matrix $\mathbf{H}$ with $H_{ij} = \frac{\partial^2 E}{\partial \omega_i \partial \omega_j}$ is a daunting task. A radical simplification is to look only for diagonal elements $\frac{\partial^2 E}{\omega_k^2}$ and back-propagate them ignoring all cross terms, $\frac{\partial^2 E_n}{\omega_{ij}^2} \approx \frac{\partial^2 E_n}{a_{nj}^2} h_{ni}^2$, where the link with weight ω_{ij} connects node i with value h_{ni} for pattern n to node j with argument a_{nj} for pattern n. Despite its approximations, this measure of how the loss depends on weights can work as well as, or better than, weight magnitude as a pruning criterion.

If all links to or from a node are pruned, the entire node can be removed. To increase the chances of this simplification, one can introduce a "group lasso" regularization. For the N_j weights ω_{ij} in a group j (typically, all inputs to a node j), create the root-mean-square $\omega_j^{\text{rms}} = \sqrt{\frac{1}{N_j}\sum_i \omega_{ij}^2}$ and add a regularization term $\Omega_j = N_j \omega_j^{\text{rms}}$ to the loss. Then $\frac{\partial \Omega}{\partial \omega_{ij}} = N_j \frac{1}{2\omega_j^{\text{rms}}} \frac{\partial \omega_j^{\text{rms}}}{\partial \omega_{ij}} = \frac{\omega_{ij}}{\omega_j^{\text{rms}}} = \text{sgn}(\omega_{ij}) \frac{|\omega_{ij}|}{\omega_j^{\text{rms}}}$. This is very similar to the lasso regularization $\frac{\partial \Omega}{\partial \omega_{ij}} = \text{sgn}(\omega_{ij})$, but the larger weights in group j will be more heavily regularized than the smaller ones. This helps bring all weights of a less needed node to zero, and the entire node can be removed during pruning.

Supervised feature selection

If we prune nodes in the input layer, we perform *feature selection*. One recurring theme in ANN training is to handle the curse of dimensionality, and therefore reducing the number of inputs can help avoid overfitting.

In supervised feature selection, we first create a model (a trained network) and try to estimate the importance of an input in that model. One measure is "saliency"

$$S_k = \frac{1}{N} \sum_{n=1}^{N} \left| \frac{\partial y(\mathbf{x}_n)}{\partial x_{nk}} \right| .$$

The absolute value in the sum reflects that we are only interested in the magnitude of output changes, not the signs. We can consider inputs with smallest saliency to be the least important, and perhaps unimportant enough to remove.

Another way to measure input importance is to corrupt the input somehow. For example, all inputs x_{nk} for patterns n could be replaced by the mean $\overline{x}_k = \frac{1}{N}\sum_n x_{nk}$, or by a randomly selected $x_{n'k}$ from available values. The change in network performance (measured by the loss function, or something more interpreteable like accuracy or AUC) is calculated. The input with largest negative change could be considered most important. It is likely that all

inputs are useful for the training performance, so it is most interesting to look at the change in validation performance. If input k has more noise than signal, it can happen that the validation performance increases when x_k is corrupted. Then it is easy to decide to remove that input!

A more costly way to select input features is to train many models with different features omitted, and keep the models that validate best.

A very cheap way to select input features is to start with *univariate* analysis, where each features is used alone. The best features are then kept for the ANN. However, this goes a bit against the ANN goal of finding subtle non-linear patterns between many input features in the data.

Here we have discussed examples of *supervised* feature selection, where validation targets are used to influence the selection of inputs to use. One danger with these approaches is that the selection itself can become overfitted to data. We will return to this when we discuss model selection in section 3.6.

3.5.8 Dimensionality reduction

Some data preprocessing can also help to avoid overfitting, in particular if it reduces the dimension of the input space.

Many datasets have inputs with similar expected noise. In that case, a variance filter can be applied, where only features with largest variance are used as inputs to the ANN. The hypothesis is then that there is noise unrelated to target values in all features, and that features with the smallest variance only display that noise.

A popular way to reduce dimensionality, without explicitly removing features, is PCA = *principal component analysis* (see section 4.3). Briefly, find a coordinate system that diagonalizes the *covariance matrix* of inputs and select the *principal components* with the most variance. The assumption is that components with small variance mainly contain noise.

Both variance filter and principal component selection are examples of *unsupervised* dimensional reduction. The target labels d_n do not influence the selection. That reduces the risk of becoming overtrained in the feature selection process.

3.5.9 Other generalization methods

There exists other regularization techniques and to mention a few:

- Training with noise
- Soft weight sharing

- Data augmentation

3.6 Model selection: optimized generalization

We have seen that there are plenty of *hyper-parameters* involved when training a neural network. There is no strictly defined distinction between parameter and hyper-parameter for ANN training, but we refer to hyper-parameters as all those that need to be defined before training starts, e.g. number of hidden nodes, learning rate, possible value of the momentum parameter, weight decay parameter, binary variable for the decision to create an ensemble, the data partition strategy for ensemble members, etc. More or less all parameters except weights and biases are hyper-parameters.

Most of these hyper-parameters are difficult to optimize before training. The procedure of selecting all parameters that define the neural network and the construction of it is usually called *model selection*.

Model selection is often performed by the simple but time consuming procedure of just trying different models and keeping the best. For example, the α parameter that controls the weight decay regularization term we can be optimized by trying many different values and produce a plot like the one in fig. (3.14). Here the validation performance is apparently an "error",

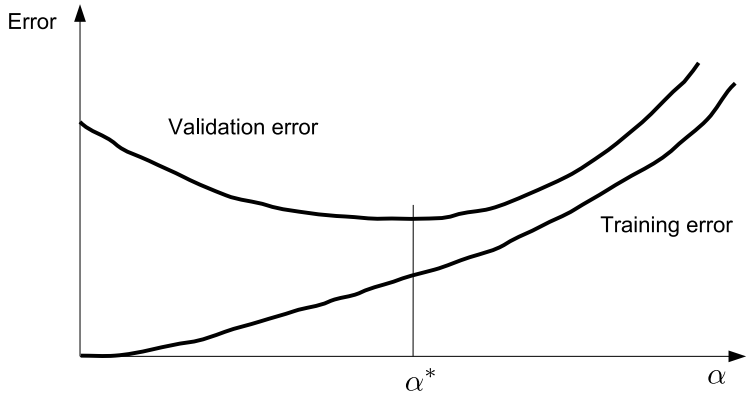


Figure 3.14: Let α be a regularization parameter, e.g. the weight decay parameter. We train and validate a number of networks, each with a different value of α. Since the validation error is an estimate of the generalization error, we pick the α that minimizes the validation error.

which could be the loss function without regularization terms (just the $-\ln p(d|\mathbf{x})$). To discover overfitting, it is illuminating to plot the same "error" for the training set.

We expect the training error to be lowest, and whenever the gap gets large, we have signs

of overfitting. However, our goal is to optimize the validation performance, so we are more interested in minimizing the validation error, than to require validation and training errors to be similar.

Once many models have been considered, and one has been selected using the validation performance, that performance is no longer an unbiased estimate of generalization performance: the validation data has been part in the "training" of hyper-parameters! A common way to get an generalization estimate is to to introduce yet another dataset, often called an independent test set.

Some hyperparameters are more likely to bias the validation result than others. For example, early stopping investigates a wide range of reasonable stopping values, and the validation result can depend strongly on the choice. This means that the validation result is at risk of becoming biased. Similarly, supervised feature selection involves a huge number of input combinations, and too elaborate optimization can lead to overfitting. Other parameters, like regularization strength, may have a rather extended region of near-optimal values, and then the test performance often turns out to be fairly similar to the optimized validation performance.

Nested loops

If there is not enough data to set aside a large independent test set, one possibility is to have two cross-partitioning loops, one outer "cross-testing" loop for estimating the generalization performance and one inner "cross-validation" loop that takes care of model selection. Figure 3.15 is illustrating this. Here each "construction" dataset is subject to a model section M-fold cross validation where model parameters for the construction data is determined. Once that is finished a new model is trained on all construction data and that model is then used on the corresponding test dataset from the outer most K-fold cross "testing" loop. The test sets in the K-fold cross "testing" loop are never part of any training of the networks, hence we have a unbiased estimate of the generalization performance.

This heavy calculation of nested loops is not always feasible. Even though fig. (3.15) illustrates two loops, there is also a loop over considered models. A naive "grid search" of some predefined values for number of nodes, regularization strength, etc., can rapidly create an overwhelmingly large pool of models. It can be more efficient to do a random search among reasonable values for hyper-parameters.

Notice that this entire process does not give us a final model to use on future data! Each construction set may give its own winning model (two winning models could for example have different regularization strength). To get a final model, we need to redo the loops using the entire data as construction data. We can then use our earlier comparision between test performances and optimized validation performances to estimate what the generalization performance of this final model could be.

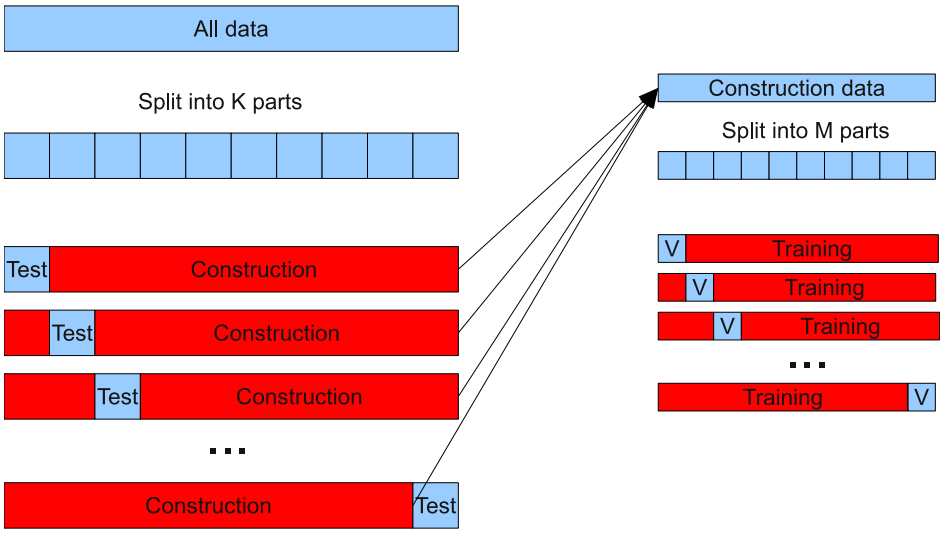


Figure 3.15: Procedure for estimating the generalization performance using two cross validation loops.

Chapter 4

CNN, Autoencoder and GAN

Early attempts at training deep MLPs with many hidden layers were not very successful, and they typically failed to outcompete an MLP with a single but large hidden layer. In other words, "deep learning" is more than just training deep networks: the design and training methods are adapted to avoid overtraining in a deep setting, and to exploit the depth to find useful features in the data that passed forward to the rest of the deep network.

Convolutional Neural Networks (CNNs), Autencoders and Generative Adverserial Networks (GANs), are three forms of feed-forward networks that are designed to perform well when they become deep.

4.1 Rectified linear unit (ReLU)

The rectifier activation function

$$\varphi(a) = \max(0, a)$$

is a popular choice for deep neural networks, especially for convolutional neural networks. It gives fast computations and reduces the risk of vanishing gradients. It could happen that the argument a_n becomes negative for almost all pattterns n, so that the node essentially "dies". This could be accepted and even appreciated as a pruning mechanism, but it can be avoided by using the smoother softplus activation, $\varphi(a) = \ln(1 + e^a)$, at the expense of slower computations.

4.2 Convolutional neural network (CNN)

(See also chapter 9 in the Deep Learning Book).

CNNs are mostly used for image analysis. We can of course think of images in both 1D and in 3D, but the most applications work with "ordinary" 2D images.

CNNs are designed to handle *spacial relations* between inputs, where inputs can be neighbours in one dimension or another. As a contrasting example, consider inputs "age", "smoker status" and "BMI" in a medical dataset. Those input columns can be presented in any order in training data, as long as the same order is used on new data. The MLP links all inputs to all hidden nodes and does not treat any inputs as neighbours.

CNNs focus heavily on *feature detection* in multiple layers. The first layer discovers small features like lines, arcs, brightness. These are merged to larger features in the next layer, then larger still, etc.

CNNs are to some extent *translationally invariant*. If there is a car and a tree in a picture, it may not be that important exactly where they are.

We will represent 2D images as a set of pixels with spacial relations. For a black and white image the value $I(i,j)$ of pixel (i,j) is a binary scalar. In a gray-scale image each pixel it could be considered a continuous value, though in practice it is often an integer in the range$[0,255]$. For colour images each pixel is represented by more than one value, where a common scheme is the RGB encoding, hence each pixel is a vector $\mathbf{I}(i,j)$.

4.2.1 Convolution

The basic tool of a CNN is the *convolution*,

$$H(y) = (I * K)(y) = \int K(y-x)I(x)\mathrm{d}x,$$

which integrates a function $I(x)$ with a *kernel* $K(y-x)$. This has many physical and mathematical applications: if $I(x)$ is a charge density and $K(y-x)$ is the force between charges a distance $y-x$ apart, then $H(y)$ is the total force in point y. If $I(x)$ is the distribution for a stochastic variable x and $K(z)$ is a distribution for z, then $H(y)$ is the distribution for $y = x+z$.

The convolution can of course be multi-dimensional, with vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$. For our 2D images, it would be 2D vectors. If the input $\mathbf{I}$ is a vector itself (perhaps representing an RGB image), then the kernel $\mathbf{K}$ is a matrix, creating a new vector $\mathbf{H}$ as the convolution result. The dimensions of $\mathbf{H}$ and $\mathbf{I}$ need of course not be the same.

For images, we replace the convolution integral with a sum. It is also conventional to re-define the kernel $K(y-x) \to K(x-y)$ and work with the so-called *cross-correlation*

$$H(y) = \int K(x-y)I(x)\mathrm{d}x = \int K(x)I(y+x)\mathrm{d}x, \tag{4.1}$$

where the last step is just a change of integration variable $x \to x + y$. The corresponding 2D sum is

$$H(i,j) = (I * K)(i,j) = \sum_{m,n} I(i+m, j+n)K(m,n)$$

Here is a small numerical example where a 4x4 image matrix is convoluted (using cross-correlation) with a 2x2 kernel.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} * \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} (-1+2+0+6) & (-2+3+0+7) & (-3+4+0+8) \\ (-5+6+0+10) & (-6+7+0+11) & (-7+8+0+12) \\ (-9+10+0+14) & (-10+11+0+15) & (-11+12+0+16) \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 8 & 9 \\ 11 & 12 & 13 \\ 15 & 16 & 17 \end{bmatrix}$$

4.2.2 Filters

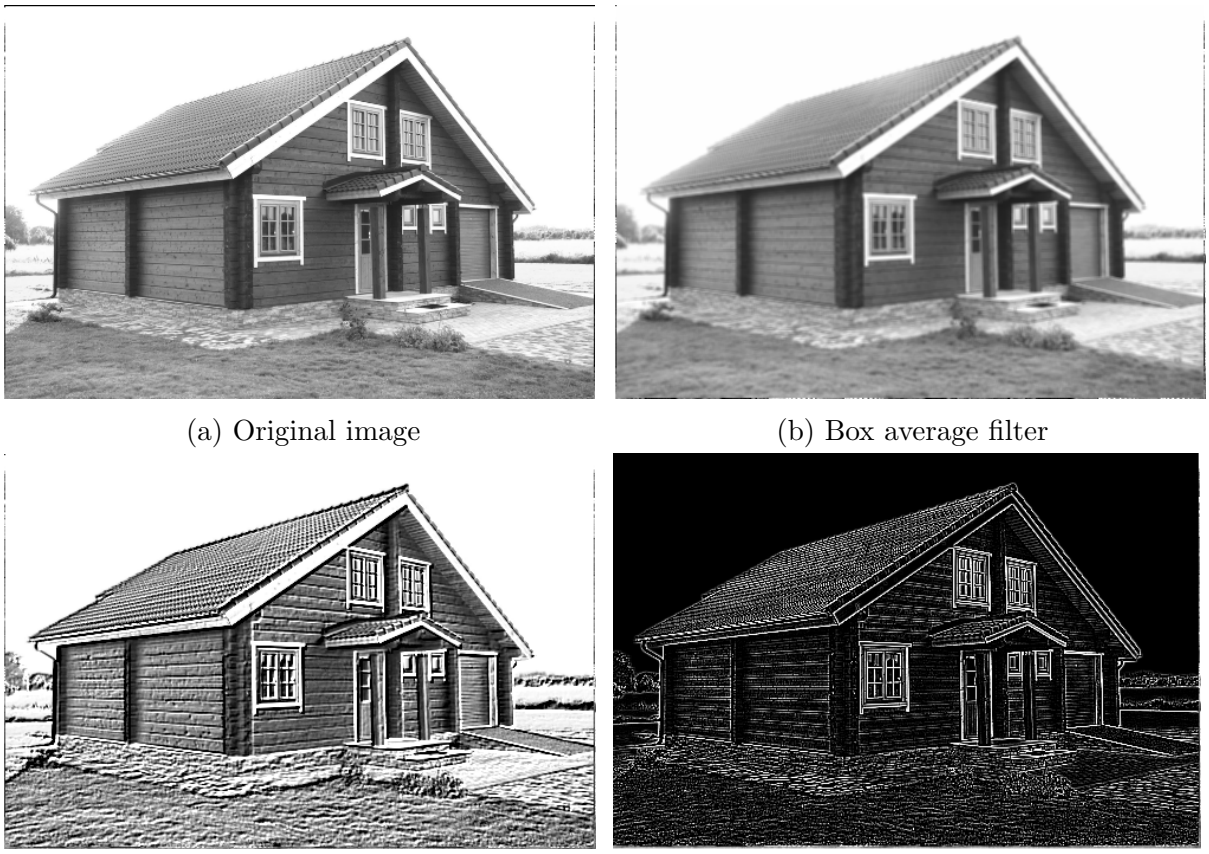
In CNNs there is a hierarchical search for features by repeating convolution in many layers. However, just using convolutions will not be of much help. A series of convolutions can be summarized as a single convolution, $K_2 * (K_1 * I) = (K_2 * K_1) * I$. The proof is prettiest in integral form

$$\begin{aligned} & \int K_2(y'-y)\left\{\int K_1(x-y')I(x)\mathrm{d}x\right\}\mathrm{d}y' = \\ = & \int\left\{K_2(y'-y)K_1(x-y')\mathrm{d}y'\right\}I(x)\mathrm{d}x = \\ = & \int\left\{K_2(x-y-y'')K_1(y'')\mathrm{d}y''\right\}I(x)\mathrm{d}x = \\ = & \int (K_2 * K_1)(x-y)I(x)\mathrm{d}x \end{aligned}$$

Just as we have a non-linear activation function in an MLP to make use of a hidden layer, we introduce a non-linear activation function φ to the convolution result, with a bias b. The combination of kernel and activation is called a *filter* and it gives an output

$$H(i,j) = \varphi\left(b + \sum_{m,n} I(i+m, j+n)K(m,n)\right)$$

Figures 4.1 and 4.2 illustrate the effect of some standard 3×3 filters. They illustrate that the result $H(i,j)$ typically gets largest in regions where input I equals the kernel K. Thus, the kernel itself reveals what features it detects.



(a) Original image

(b) Box average filter

(c) Emboss filter

(d) Outline filter

Figure 4.1: A selection of three different filters applied on the top left image.

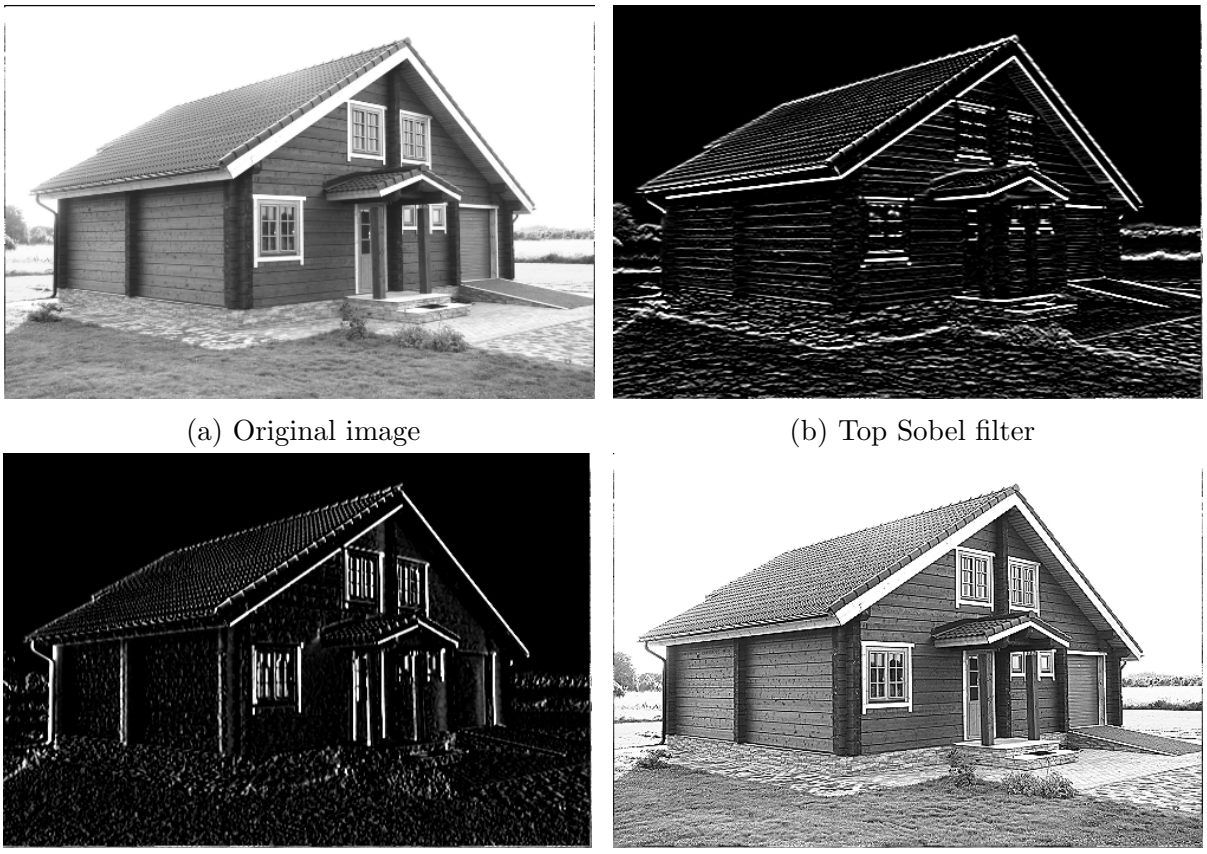
The kernels in figures 4.1b–4.1d are

$$\text{box average} = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}, \qquad \text{emboss} = \begin{bmatrix}-2 & -1 & 0\\ -1 & 1 & 1\\ 0 & 1 & 2\end{bmatrix}, \qquad \text{outline} = \begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}.$$

The box average blurs the picture. The emboss finds gradients from top left to bottom right, but since the sum of kernel weights are 1, it also preserves some weighted average of brightness. The outline filter gives zero (black) if all nine pixels are equal, but enhances a pixel that is brighter than its surrounding.

The kernels in figures 4.2b–4.2d are

$$\text{top Sobel} = \begin{bmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix}, \qquad \text{left Sobel} = \begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}, \qquad \text{sharpen} = \begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$$



(a) Original image

(b) Top Sobel filter

(c) Left Sobel filter

(d) Sharpen filter

Figure 4.2: A selection of three different filters applied on the top left image.

The Sobel filters find edges in different directions. The sharpen filter enhances a bright center pixel, similarly to the outline filter. However, the sum of weights is 1 instead of 0, so it preserves the value if nearby pixels are equal.

Importantly, we will not need to learn what a good filter is and select kernels for our CNN, that is part of the CNN training! The weights in the kernels are treated as tunable parameters and will be trained, *e.g.,* with back-propagation. We show some standard filters here to illustrate how they can be interpreted as feature detectors.

4.2.3 Sparse connectivity and parameter sharing

A key feature in CNNs is to have a fairly *local* kernel K. Very popular is 3×3 as with the standard examples presented above. In this way, small features are found wherever they are.

To combine smaller features into larger ones, we build a deep CNN with multiple layers, which we describe in more detail later.

A CNN with a local kernel corresponds to sparce network with shared weights. Let's illustrate this using an ordinary MLP. Figure 4.3 shows the two versions of the input-to-hidden layer of an MLP. The top network is fully connected the usual way. Here a hidden node is connected to all of the input nodes.

The lower network in figure 4.3 shows the same input-to-hidden network, but with fewer connections. The hidden node h_3 is only receiving input from inputs (x_2, x_3, x_4), etc. The number of weights is independent of the size of the input layer, but still grows with the size of the hidden layer.

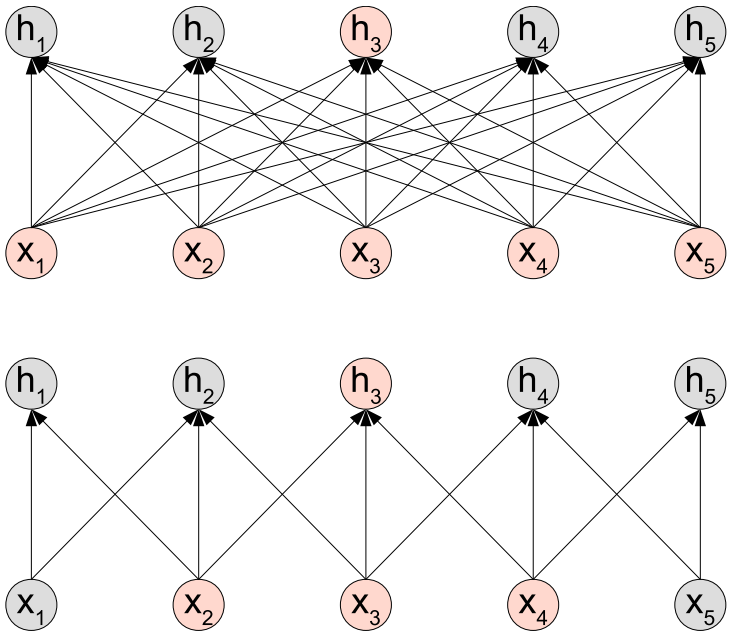


Figure 4.3: (Top) A fully connected input-to-hidden layer in an MLP. The highlighted hidden node (h_3) is receiving input from all input nodes. It has a large "receptive field". (Bottom) The same MLP layer, but with fewer weights. The h_3 hidden node is now only receiving input from three input nodes. It has a smaller receptive field.

Figure 4.4 shows the same 1D example as before, but now with a lot of weights being shared across the layer. All weights with the same colour are being shared between the different hidden nodes. This means that e.g. h_2 is computing a weighted sum of (x_1, x_2, x_3) exactly the same way as e.g. h_4 is weighting together (x_3, x_4, x_5). Also sharing the bias reduces the number of trainable parameters to 4, independently of the size of the layers.

The sparce network with shared weights is doing exactly the same as a 1D filter acting on a 1D image. The hidden nodes represent the filtered image. Figure 4.5 shows an illustration

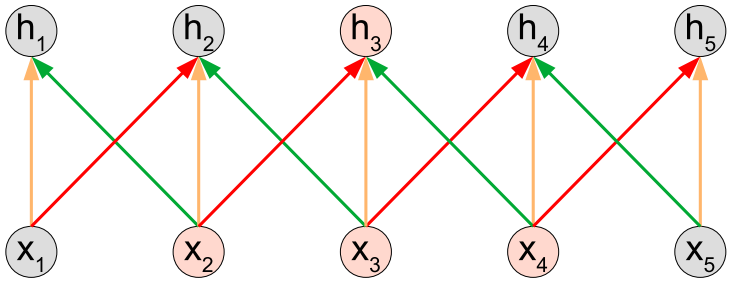


Figure 4.4: Sparse connectivity and **shared** weights. Weights with the same colour are shared between the hidden nodes.

of a corresponing filter of a 2D input layer into a 2D hidden layer. Independently of the size of the input layer, the number of tunable parameters is 10 (3×3 weights and one bias).

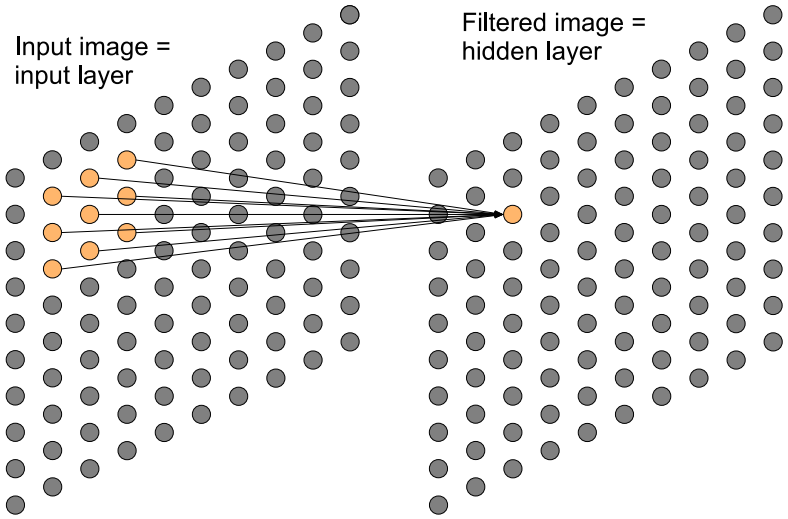


Figure 4.5: The 2D arrangement of the input layer is filtered into a 2D hidden layer, representing the filtered image. Each hidden node is only connected to nine inputs in the input layer (the "receptive field" of the hidden node). This is visualized by the orange hidden and input nodes. The kernel in the convolution process used when computing the filtered image is here represented by the weight values of a given hidden node. Note: each hidden node share the weights with all other hidden nodes.

4.2.4 Channels

The input layer may come with multiple *channels*, *e.g.,* an RGB colour image with 3 channels. We expect all channels in a layer to have the same size. We can view the image as having

a certain *channel depth*. A filter will have the same depth as the number of channels in the layer it acts upon. This means that for RGB colour images all filters in the first convolutional layer have depth three. This is illustrated in figure 4.6 where a 10x10x3 image is filtered using a 3x3x3 filter. The mathematical operation of the filter with multiple channels c can be written

$$H(i,j) = \varphi\left(b + \sum_c \sum_{m,n} I_c(i+m, j+n)K(m,n,c)\right).$$

Here the sums over n and m run over the size of the kernel, while the sum over c runs over all input channels.

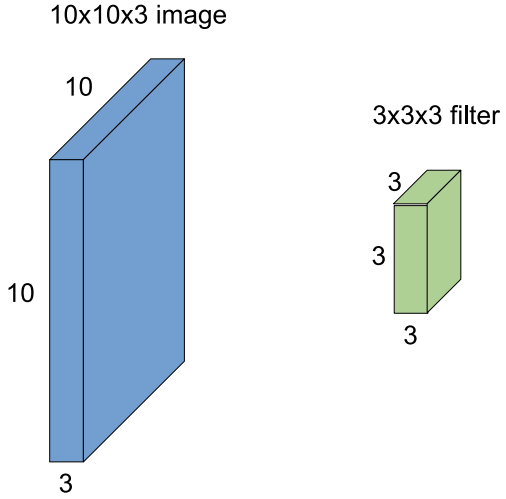


Figure 4.6: A 10x10x3 images and a filter of size 3x3x3. The important thing to remember is that filters always extend to the full channel depth of the input image.

To inteprete an image, we may need to find many different features and then discover key combinations. A corner could for example be found as a horizontal edge and a vertical edge in a specific relation to each other. Therefore, we apply multiple filters to the input image and create many channels in the hidden layer. This is illustrated in figure 4.7, where five filters are applied to an RGB input, to create five channels in the next layer. The mathematical operation for each filter f can be written

$$H_f(i,j) = \varphi_f\left(b_f + \sum_c \sum_{m,n} I_c(i+m, j+n)K_f(m,n,c)\right).$$

Each of the filters will detect different features useful for the task in questions, for example horizontal or vertical transitions from one colour to another.

Notice that the number of trainable parameters grows rapidly with the number of channels — different channels do not share weights.

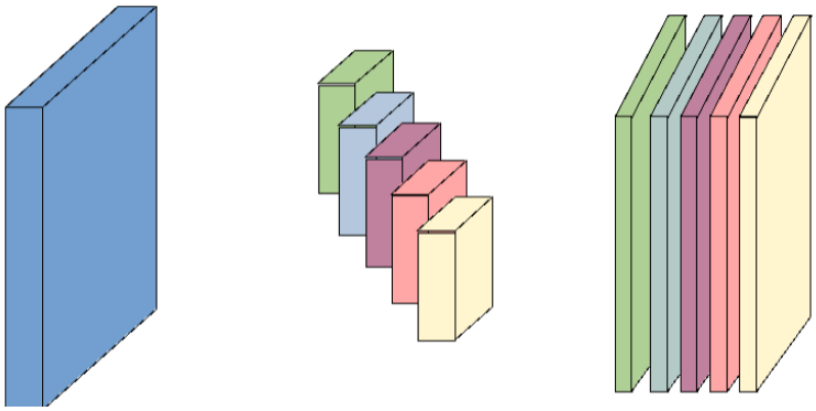


Figure 4.7: Here we have an input image of depth 3 (e.g. RGB image). Using five different filters will result in an filtered image of depth five (five channels).

4.2.5 Padding

When we do convolution, we have to decide on how to handle the outskirts of an image, and also how large the filtered image should be. In the numerical example of a 4x4 input and a 2x2 kernel presented above, the kernel was simply forced to fit into the image, creating a smaller 3x3 result. The effect is larger if the kernel is big: a 5x5 kernel would reduce a 6x6 image to a 2x2 result.

If we want to preserve the image size, and if we are afraid of losing information in the outskirts of the image, we can include *padding*. Fig. (4.8) and fig. (4.9) compare the result of a 3x3 kernel with and without a zero padding. Zero padding is very common, but

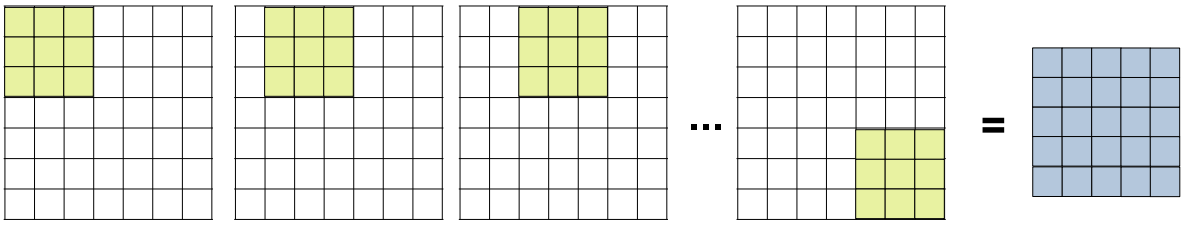


Figure 4.8: The image is 7x7 pixels and the kernel (green) is 3x3. Moving the kernel (fully inside) the image will produce a 5x5 filtered images (blue).

copy padding (where padded pixels get the same value as the nearest real pixel) is also a reasonable option sometimes.

In many libraries it is common to write *padding='same'*, to indicate that the filtered image should have the same dimensions as the input image.

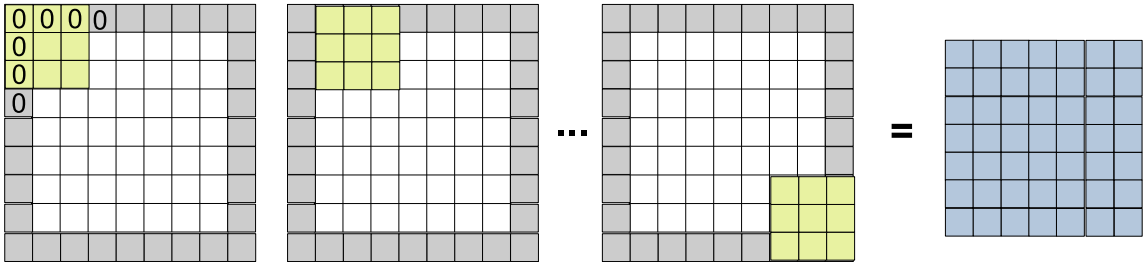


Figure 4.9: Zero-padding the images with a boundary of zeros will make sure that the filtered image has the same size as the original image.

4.2.6 Stride (data reduction)

So far, we have placed our kernel above every possible region in the input image, creating a filtered image of (almost) the same size. This is not necessary. We can sample fewer points in the input image by introducing a *stride*. Figure 4.10 illustrates a 3x3 kernel with stride 2.

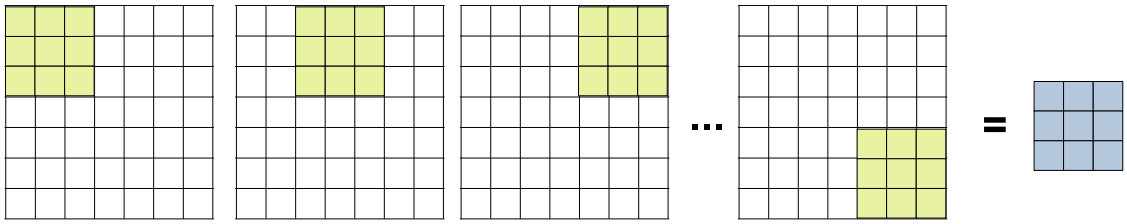


Figure 4.10: The image is 7x7 pixels and the kernel (green) is 3x3. Moving the kernel (fully inside) the image, but now with stride=2, will produce a 3x3 filtered images (blue).

For certain combinations of image sizes, kernel sizes and strides, it can be necessary to add padding for the kernel to fit everywhere. Consider for example a 7x7 image, a 3x3 kernel and stride 3.

A large stride performs data reduction and is mainly used together with a special kind of filter called a *pooling* filter.

4.2.7 Pooling

A pooling filter makes a fixed mapping of input values: the weights in the pooling filter are not trained. The two most common, *max pooling* and *average pooling* are shown in fig. (4.11). Note that max pooling is not of the form $h = \varphi(\sum \omega_k x_k)$, but $h = \max_k\{x_k\}$. The pooling filter has depth 1 and is applied to all channels independently, creating the same number of channels. The pooling filters make the representation approximately invariant to small

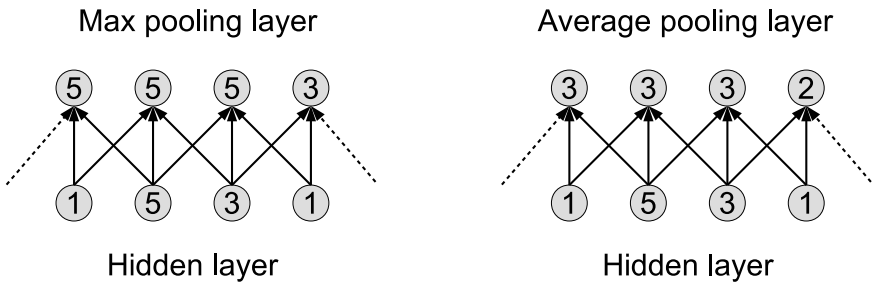


Figure 4.11: Illustration of two different pooling stages. The left graph is showing max pooling with a receptive field of three nodes. (The dotted line at the border indicates that other hidden nodes may be part of the receptive field, but they are not part of the calculations). The right graph is showing average pooling with also a receptive field of three nodes.

translations of the input: if the input is moved by a small amount, the values after pooling stay approximately the same. This is useful if we are interested in whether a specific feature exists, and not so much exactly where it is. Consider how much you can move around a upward arc $\frown$, downard arc $\smile$ and a cricle $\circ$ and still say that the these three features form an "eye"!

Data reduction using large stride fits very well with pooling, where we anyway abandon exact information about the location of features. This is illustrated in fig. (4.12).

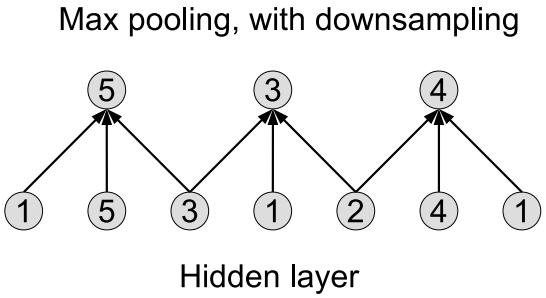


Figure 4.12: Max pooling with downsampling using stride=2, reducing seven hidden nodes to three.

4.2.8 Building a CNN

Convolutional layers

Following the terminology used in the Deep Learning Book, a convolutional layer consists of three blocks (see figure 4.13), the convolution stage, the activation stage and the pooling stage. The first two are vital, but the pooling stage can many times be omitted. You can

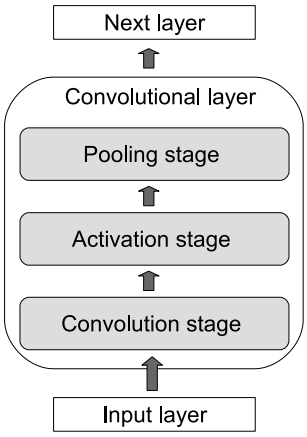


Figure 4.13: The three building blocks of a convolutional layer in a CNN. The convolutional stage, the activation stage and the pooling stage. The first two are fundamental, but the pooling stage can sometimes be omitted.

find more detailed discussions on convolutional- and pooling layers in chapter 9.1 - 9.8 in the Deep Learning Book.

It is quite common to combine many convolutional layers into a deep CNN. Currently ReLU is the dominant choice for the ativation step, but layers can vary a lot in most other aspects: the size of kernels (3×3 or larger?), the number of channels, the use of pooling, and data reduction by stride. It is of course possible to play around with asymmetric shapes, *e.g.,* a 2x5 kernel or different strides in different dimensions. However, this rapidly enlarges the number of hyper-parameters in a way that is rarely needed.

Flattening and MLP

When several convolutional layers have been used to detect fairly large features, it may not be so important to know their relative position. It can then be efficient to "flatten" the final channels into a long 1-dimensional vector, and treat it as an input layer to an ordinary MLP. The output layer of the MLP is selected to match the task of the CNN (categorical classification of images is very common).

Figure 4.14 shows the architecture of one of the first successful models, LeNet, used for classification of handwritten digits. This paper[1] was published 1998.

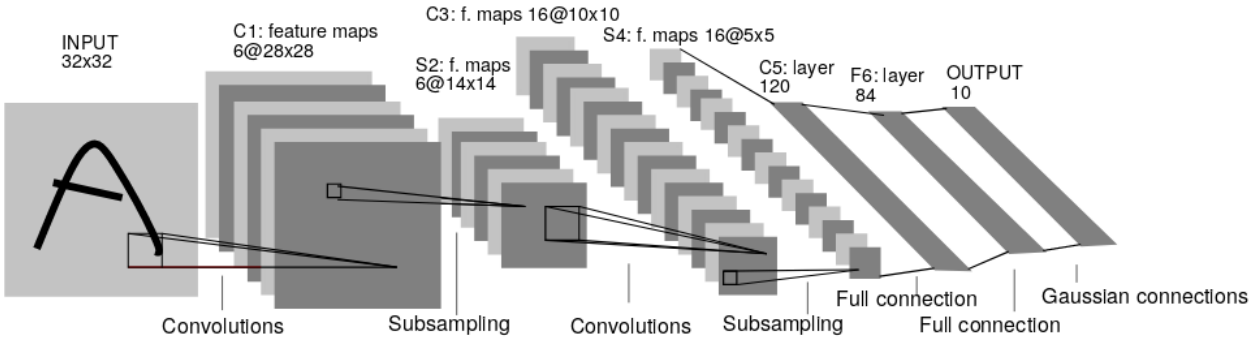


Figure 4.14: The CNN model of LeNet-5. This architecture was used to classify digits, hence the 10 output nodes. The term "subsampling" refers to pooling.

4.2.9 Further reading

There are many, many ways of building a CNN model and they can be used for many different problems. There is also a lot of research activity around CNNs, meaning that new models and new tools for understanding them are produced as we speak.

There are also many sites that provide good explanations of how CNN models work and how to use them. This site [2] provides an good overview of a few important CNN models.

4.3 The autoencoder

An autoencoder, visualized in figure 4.15, contains at least one hidden layer, and the size of the output layer is the same as the input layer. In fact, as indicated in the figure, the output equals the input. This is a trivial problem to solve if we allow for the hidden layer to be at least as large as the input layer. We therefore the require that the middle layer is a "bottleneck", with fewer nodes than the input layer.

If the output fits the input, despite the bottleneck, then the values in the bottleneck layer are an efficient representation of the data: we have performed dimensional reduction! We call

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, november 1998.

[2] https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

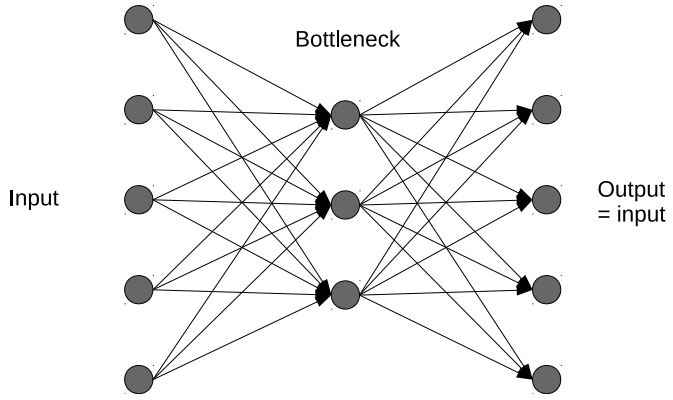


Figure 4.15: This special MLP, called an autoencoder, has the same number of output nodes as the inputs. The hidden layer, called bottleneck, always have a smaller size compared to the input size.

the first half of the network an "encoder", where the input is encoded by the hidden layer. The second part of the network is then the "decoder", where the hidden representation is decoded into the original input.

To highlight the purpose of the autoencoder, we will call inputs $\mathbf{x}_n$, outputs $\mathbf{x}'_n$ and reserve $\mathbf{y}_n$ for the bottleneck values. The loss sums up losses of all inputs i and all patterns n,

$$E(\boldsymbol{\omega}) = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{P} E_{ni}(\boldsymbol{\omega}) \tag{4.2}$$

where N is the number of data in the training dataset and P is the number of inputs. There is little need to abandon our standard loss functions, so that $E_{ni} = \left(x'_i\left(\mathbf{x}_n, \boldsymbol{\omega}\right) - x_{ni}\right)^2$ for continous inputs x_i, and a cross entropy error for binary inputs. Typically, the error $E(\boldsymbol{\omega})$ is minimized using some standard stochastic gradient descent approach.

Note that the autoencoder does not use any labels of the data!

PCA

We have previously argued that there is no point in having linear activation functions in hidden layers, but because of the bottleneck property, a linear middle layer can be used in an autoencoder. With linear output functions and a squared error loss function, the autoencoder will then perform *principal component analysis*. For the interested reader, we describe it here in some detail.

We call input vectors $\mathbf{x}_n$, hidden vectors $\mathbf{y}_n$ and output vectors $\mathbf{x}'_n$. Hidden biases form a vector $\mathbf{y}_0$ and output biases a vector $\mathbf{b}$. With P inputs and a bottleck of size M, we have that $\mathbf{x}_n$, $\mathbf{x}'_n$ and $\mathbf{b}$ are P-dimensional, while $\mathbf{y}_n$ and $\mathbf{y}_0$ are M-dimensional.

To relate to matrix conventions in PCA, we represent each vector with a column matrix. (And must remember that most ANN code packages expect input vector $\mathbf{x}_n$ to be a row in a data file.) For each hidden node k and pattern n we have the relation $y_{kn} = \tilde{\boldsymbol{\omega}}_k^\top \mathbf{x}_n + y_{k0}$, which can be summarized as

$$\mathbf{y}_n = \tilde{\boldsymbol{W}}\mathbf{x}_n + \mathbf{y}_0, \tag{4.3}$$

where row k in $\tilde{\boldsymbol{W}}$ is $\tilde{\boldsymbol{\omega}}_k^\top$. Similarily, we can write

$$\mathbf{x}'_n = \boldsymbol{W}\mathbf{y}_n + \mathbf{b}, \tag{4.4}$$

where each row i in $\boldsymbol{W}$ is the weights $\boldsymbol{\omega}_i^\top$ to output node i.

There is a lot of redundancy which we can exploit: A change $\mathbf{y}_0 \to \mathbf{y}_0 + \delta\mathbf{y}$ can be compensated with $\mathbf{b} \to \mathbf{b} - \boldsymbol{W}\delta\mathbf{y}$, and with an invertible matrix $\boldsymbol{A}$ we can make the changes $\boldsymbol{W} \to \boldsymbol{W}\boldsymbol{A}$, $\tilde{\boldsymbol{W}} \to \boldsymbol{A}^{-1}\tilde{\boldsymbol{W}}$, $\mathbf{y}_0 \to \boldsymbol{A}^{-1}\mathbf{y}_0$. Therefore, we can restrict ourselves to solutions where the hidden vectors satisfy

$$\sum_n \mathbf{y}_n = 0, \tag{4.5}$$

and where the columns in $\boldsymbol{W}$ are orthonormal,

$$\boldsymbol{W}^\top \boldsymbol{W} = \mathbb{1}_{M\times M}. \tag{4.6}$$

A column k in W corresponds is a vector $\mathbf{u}_k$ of weights pointing *out* from hidden node k. Notice that $\boldsymbol{W}$ is not square, so we get $\boldsymbol{W}\boldsymbol{W}^\top \neq \mathbb{1}_{P\times P}$

We want to minimize the sum of squared errors between $\mathbf{x}'_n$ and $\mathbf{x}_n$, so we work with a loss

$$E \propto \sum_n |\mathbf{x}_n - \mathbf{x}'_n|^2. \tag{4.7}$$

The derivative with respect to some parameter θ is

$$\frac{\partial E}{\partial \theta} \propto \sum_{ni} \frac{\partial x'_{in}}{\partial \theta}(x'_{in} - x_{in}). \tag{4.8}$$

With $\theta = b_j$ we get an extremum value when

$$\begin{aligned} 0 &= \sum_n \left(\sum_k W_{jk} y_{kn} + b_j - x_{jn} \right) \Rightarrow \\ \mathbf{b} &= \frac{1}{N}\sum_n \mathbf{x}_n - \frac{1}{N}\boldsymbol{W}\underbrace{\Big(\sum_n \mathbf{y}_n\Big)}_{=0} = \frac{1}{N}\sum_n \mathbf{x}_n. \end{aligned} \tag{4.9}$$

The bias vector $\mathbf{b}$ should be the mean of inputs!

With $\theta = y_{lm}$ we get an extremum value when

$$\begin{aligned} 0 &= \sum_i W_{il} \left(\sum_k W_{ik} y_{km} + b_i - x_{im} \right) \Rightarrow \\ \boldsymbol{W}^\top \boldsymbol{W} \mathbf{y}_n &= \boldsymbol{W}^\top (\mathbf{x}_n - \mathbf{b}) \Rightarrow \\ \mathbf{y}_n &= \boldsymbol{W}^\top (\mathbf{x}_n - \mathbf{b}). \end{aligned} \tag{4.10}$$

Thus, under the constraints on $\mathbf{y}_0$ and $\boldsymbol{W}$, the solution for the autoencoder is $\mathbf{b} = \frac{1}{N} \sum_n \mathbf{x}_n$, $\tilde{\boldsymbol{W}} = \boldsymbol{W}^\top$ and $\mathbf{y}_0 = -\boldsymbol{W}^\top \mathbf{b}$.

It remains to find the columns in $\boldsymbol{W}$. To shorten notation, we introduce mean-subtracted vectors

$$\mathbf{z}_n = \mathbf{x}_n - \mathbf{b}, \quad \mathbf{z}'_n = \mathbf{x}'_n - \mathbf{b}, \tag{4.11}$$

to find $\mathbf{z}'_n = \boldsymbol{W}\boldsymbol{W}^\top \mathbf{z}_n$ and

$$\mathbf{x}_n - \mathbf{x}'_n = \mathbf{z}_n - \mathbf{z}'_n = \left(\mathbb{1} - \boldsymbol{W}\boldsymbol{W}^\top \right) \mathbf{z}_n. \tag{4.12}$$

Since $\boldsymbol{W}\boldsymbol{W}^\top$ is symmetric and $(\mathbb{1} - \boldsymbol{W}\boldsymbol{W}^\top)^2 = \mathbb{1} - 2\boldsymbol{W}\boldsymbol{W}^\top + \boldsymbol{W}\boldsymbol{W}^\top \boldsymbol{W}\boldsymbol{W}^\top = \mathbb{1} - \boldsymbol{W}\boldsymbol{W}^\top$, this gives

$$|\mathbf{x}_n - \mathbf{x}'_n|^2 = \mathbf{z}_n^\top \left(\mathbb{1} - \boldsymbol{W}\boldsymbol{W}^\top \right)^\top \left(\mathbb{1} - \boldsymbol{W}\boldsymbol{W}^\top \right) \mathbf{z}_n = \mathbf{z}_n^\top \left(\mathbb{1} - \boldsymbol{W}\boldsymbol{W}^\top \right) \mathbf{z}_n. \tag{4.13}$$

We want to sum this over n and minimize, but to include the constraints that all columns in $\boldsymbol{W}$ should be orthonormal, we introduce Lagrange multipliers $F_{kk'}$ for every scalar product $\sum_i W_{ik} W_{ik'}$ between two columns, and add $\sum_{ikk'} W_{ik} W_{ik'} F_{kk'} = \mathrm{Tr}\left[\boldsymbol{W}^\top \boldsymbol{W} \boldsymbol{F} \right]$ to the optimization problem. Also introducing the *covariance matrix*

$$\boldsymbol{R} = \frac{1}{N} \sum_n (\mathbf{x}_n - \mathbf{b})(\mathbf{x}_n - \mathbf{b})^\top = \frac{1}{N} \sum_n \mathbf{z}_n \mathbf{z}_n^\top, \tag{4.14}$$

we see that we want to minimize

$$\mathrm{Tr}\left[\left(\mathbb{1} - \boldsymbol{W}\boldsymbol{W}^\top \right) \boldsymbol{R} \right] + \mathrm{Tr}\left[\boldsymbol{W}^\top \boldsymbol{W} \boldsymbol{F} \right] \tag{4.15}$$

Taking the derivative with respect to an element W_{jl} and going through some book-keeping, shows that we get extremum values when

$$\boldsymbol{R}\boldsymbol{W} = \boldsymbol{W}\boldsymbol{F}^{(s)}, \tag{4.16}$$

The result exploits the fact that $\boldsymbol{R}$ is symmetric by its definition, and introduces $\boldsymbol{F}^{(s)} = \frac{1}{2}(\boldsymbol{F} + \boldsymbol{F}^\top)$ as the symmetric part of $\boldsymbol{F}$. This means that $\boldsymbol{F}^{(s)}$ can be diagonalized by some orthogonal matrix $\boldsymbol{B}$, so that

$$\boldsymbol{R}\boldsymbol{W}\boldsymbol{B} = \boldsymbol{W}\boldsymbol{B}\boldsymbol{B}^\top \boldsymbol{F}^{(s)} \boldsymbol{B} = \boldsymbol{W}\boldsymbol{B}\boldsymbol{\Lambda}, \tag{4.17}$$

where $\boldsymbol{\Lambda}$ is diagonal. Furthermore, since $(\boldsymbol{WB})^\top(\boldsymbol{WB}) = \boldsymbol{B}^\top\boldsymbol{W}^\top\boldsymbol{WB} = \boldsymbol{B}^\top\boldsymbol{B} = \mathbb{1}$, the constraint on orthogonal columns holds for $\boldsymbol{WB}$. We can therefore absorb $\boldsymbol{B}$ into $\boldsymbol{W}$ to get $\boldsymbol{RW} = \boldsymbol{W\Lambda}$. Since $\boldsymbol{\Lambda}$ is diagonal, this separates into one equation for each column in $\boldsymbol{W}$:

$$\boldsymbol{Ru}_k = \lambda_k \boldsymbol{u}_k. \tag{4.18}$$

The columns in $\boldsymbol{W}$ should be *eigenvectors* to the correlation matrix $\boldsymbol{R}$! To minimize the error, we should select the eigenvectors $\boldsymbol{u}_k$ with the largest eigenvalues λ_k. They are called the *principal components*.

The simple linear projection to a space that minimizes the sum of squared errors corresponds to a projection to a space that maximizes the preserved variance among patterns.

When selecting how many principal components that should be considered in a PCA, it is common to study the fraction of *explained variance*, which is the sum of included eigenvalues λ_k divided by the sum of all eigenvalues λ_k. Fig. (4.16) shows some 2-D data sets and the corresponding eigenvalues.

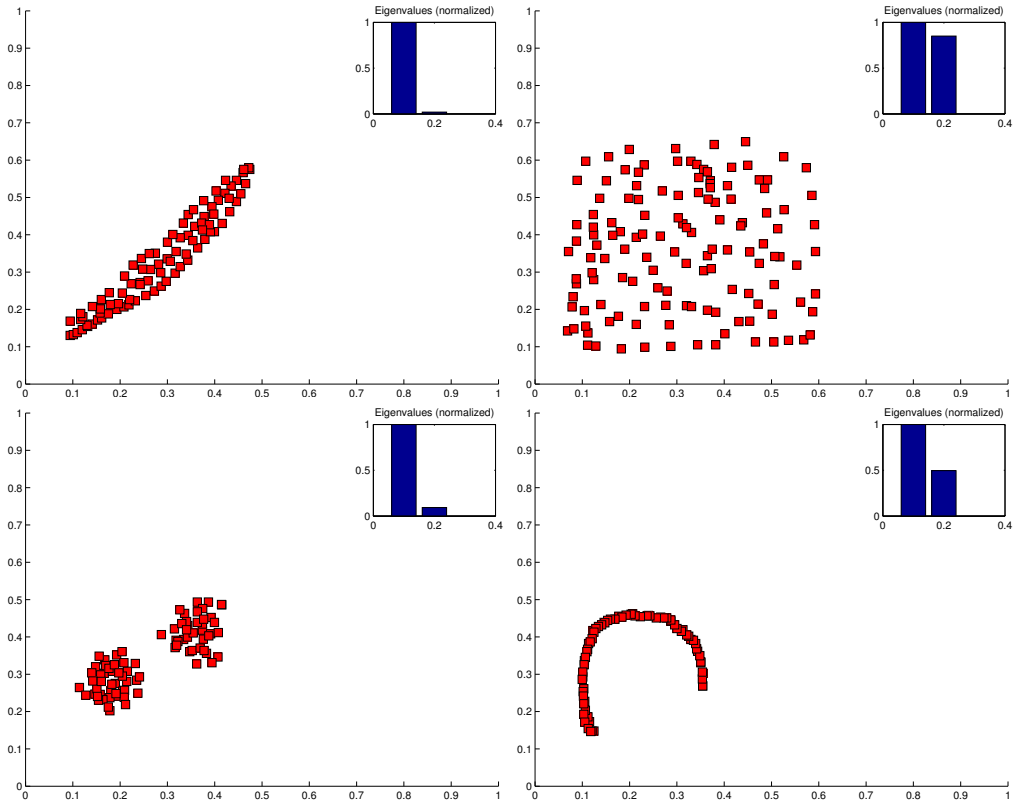


Figure 4.16: Eigenvalues for some 2-D data sets.

Non-linear projection

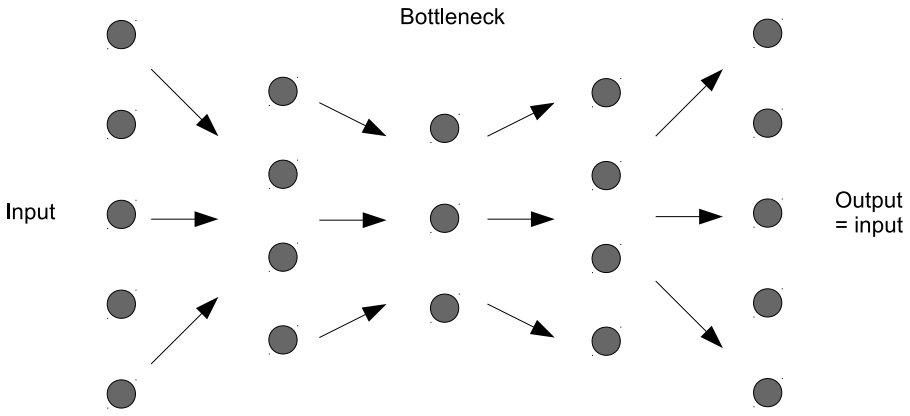


Figure 4.17: An example of an autoencoder with 3 hidden layers.

PCA may be a poor choice if there are complex relations between input dimensions, as in the bottom right graph in fig. (4.16). Introducing a non-linear activation function in the bottleneck, such as the tanh() or the sigmoid, will make the encoder a bit more complex, but what really improves mapping capabilities are more hidden layers (with non-linear activation functions). Figure 4.17 shows an autoencoder with 3 hidden layers. It is an example of a the very common *butterfly* design, where there is an odd number of hidden layers, and the size of the layers are "mirrored" around the bottleneck, so that the first hidden layer is the same size as the last, and so on.

Figure 4.18 shows an example where 2D data nicely follows a one-dimensional curve. However, we cannot use a PCA approach since the curve is not linear. Instead, the figures show results for autoencoders with three hidden layers. The error function was MSE and training used standard SGD. The output from the autoencoder is shown as "reconstructed" curve. The left graph used layer sizes $2-2-1-2-2$ with tanh() for the hidden layers, and we can see that it is almost being able to reconstruct the input curve. In the right graph we have an $2-4-1-4-2$ autoencoder. Now the output fits perfectly to the input.

Note here that in both cases, the bottleneck layer only has one hidden node. This means that we effectively represent the 2D input by a single value.

In conclusion, we can view the autoencoder as a very efficient tool for finding good representations of data, without the need for labels.

Applications of the autoencoder

Looking at the error $|\mathbf{x}_n - \mathbf{x}'_n|$ for each pattern n is a way to do *outlier detection*. If the error is extremely large for one pattern, that pattern could be suspected to be an error in

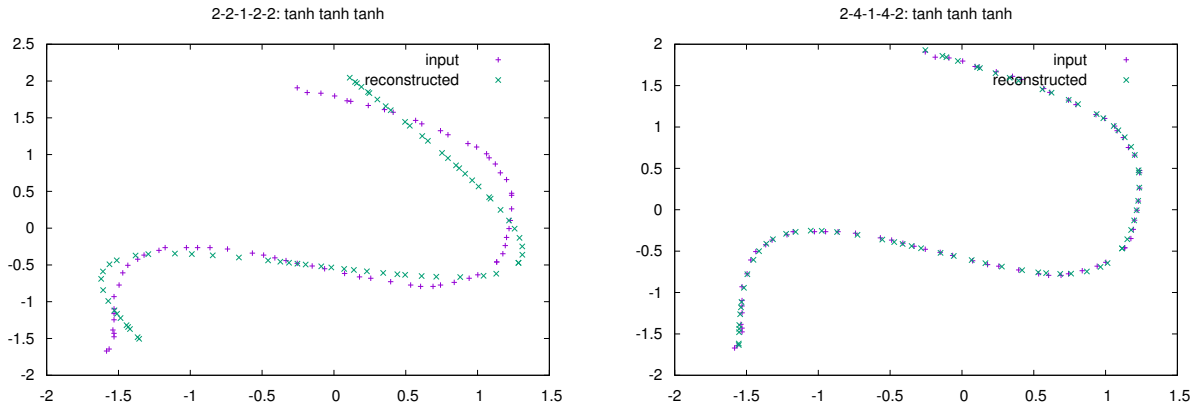


Figure 4.18: Training an autoencoder on a 2D dataset. (left) The input "curve" is feed into an autoencoder of size (2-2-1-2-2) and the output is the curve marked "reconstructed". (right) Here 4 nodes where used in the first and last hidden layer. Now the output is perfectly matching the input.

the provided data.

Another application is *missing value imputation*. If a pattern m has some missing values x_{mi}, they can first be given some simple replacement, for example a randomly selected known value x_{ni}, or the mean of known values x_{ni}. Then the vector $\mathbf{x}_m$ can be fed to the autoencoder and the output $\mathbf{x}'_m$ contains a new value x'_{mi} that often is a better estimate of the missing value.

4.3.1 Stacked denoising autoencoder

The stacked denoising autoencoder (SdAE) is modern version (i.e. deep learning version) of the autoencoder described above. It introduces two new concepts, one that you are already familiar with and one that we have not yet talked about. They are,

- Denoising: This simply means that we train with dropout on the input layer. This is a regularization technique that avoids overtraining on noise.
- Pre-training: This we have not used before! It means that we do not train the full network from scratch (i.e. random initialization of the weights). Instead each layer is pre-trained one by one and in the end a fine-tuning is performed using standard methods.

Let's look at the "stacked" pre-training in a bit more detail. Each layer l in the encoder

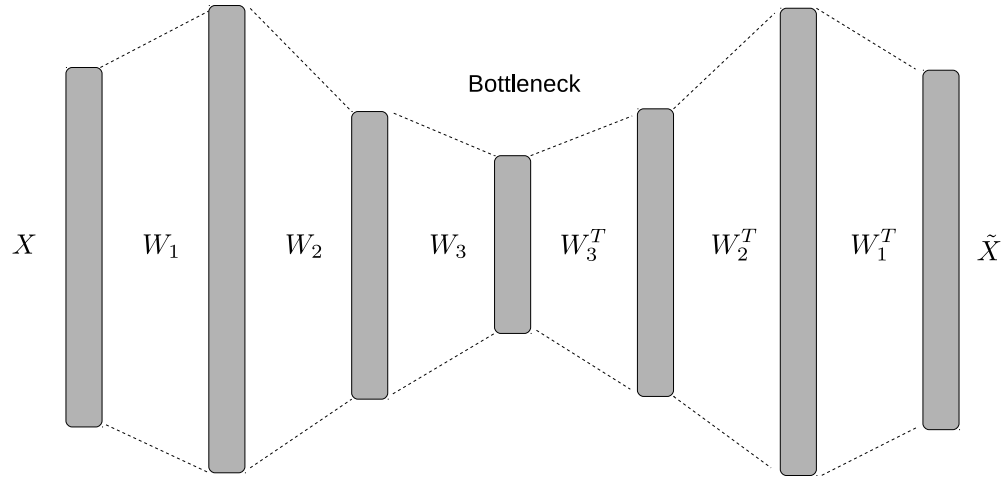


Figure 4.19: The SdAE network architecture. The equally sized encoder and decoder part share the bottleneck layer. Tied weights are used indicated by the set of weights $(W_1, W_1^\top)$, $(W_2, W_2^\top)$ etc. The output layer $\tilde{\mathbf{x}}$ is the autoencoder reconstruction of the input X.

performs a mapping from layer input $\mathbf{h}_{l-1}$ to layer output $\mathbf{h}_l$ with parameters $\mathbf{W}_l$ and $\mathbf{b}_l$,

$$\mathbf{h}_l = f_l(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l),$$

where $f_l(\cdot)$ is the mapping (or activation) function. The weight matrix $\mathbf{W}$ and the bias vector $\mathbf{b}$ are trainable parameters for each layer of the encoder. The SdAE typically uses tied weights, where the decoders weights are the transpose of the encoder weights, as shown in fig. (4.19). For instance if $\mathbf{W}_L$ denotes the last weight matrix of the decoder, then $\mathbf{W}_L = \mathbf{W}_1^\top$. This is in part inspired by the PCA solution for a simple, linear autoencoder. It reduces the number of tunable paratemers in the deep network, and it also prevents large weights to some extent, since the same large weights would have to be useful in two layers with very different tasks.

Instead of training $\mathbf{x} \to \mathbf{h}_1 \to \mathbf{h}_2 \to \ldots \to \mathbf{x}'$ all at once, we start with a small autoencoder with a single hidden layer, which implements $\mathbf{x} \to \mathbf{h}_1 \to \mathbf{x}'$. Minimizing an error between $\mathbf{x}$ and $\mathbf{x}'$, this will find good values for the weights and biases for the first layer, $\mathbf{W}_1$ and $\mathbf{b}_1$ as well as for the last, $\mathbf{W}_L$ and $\mathbf{b}_L$, possibly with the constraint of tied weights, $\mathbf{W}_L = \mathbf{W}_1^\top$

When pre-trained $\mathbf{W}_1$ and $\mathbf{b}_1$ are found, we run the entire dataset through the small encoder, to get vectors $\mathbf{h}_{1n}$ for each pattern n. This is then used as input to a new, small autoencoder, which implements $\mathbf{h}_1 \to \mathbf{h}_2 \to \mathbf{h}_1'$. Training searches for $\mathbf{W}_2$, $\mathbf{b}_2$, $\mathbf{W}_{L-1}$ and $\mathbf{b}_{L-1}$ that minimize the error between $\mathbf{h}_1$ and $\mathbf{h}_1'$. And so on!

When all layers have been pre-trained, the entire autoencoder is put together, which results in a butterfly design. The pre-trained values are used as initial values, and training of the entire network continues as usual. In this way, the initialized values are close to a good

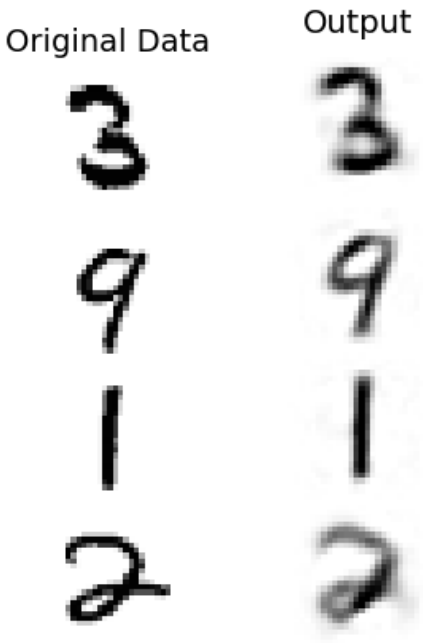


Figure 4.20: An example of original (left) and reconstructed images from the MNIST database, using a SdAE of size [784] − [1000] − [400] − [20] − [400] − [1000] − [784].

solution, which speeds up and improves the final training.

If the butterfly is not strictly reducing layer size in each step, some small networks would get so large bottle-necks, that they get overtrained on their input. As an example, consider $\mathbf{x} \rightarrow \mathbf{h}_1 \rightarrow \mathbf{x}'$ for the network in fig. (4.19). This is solved with "denoising", which means dropout on the input layer to each small network when it is trained.

MNIST example

Figure 4.20 shows original and reconstructed images of the MNIST database using a SdAE. Each image in the MNIST database is represented as a single numerical vector of size 784. There are 60 000 images in the training dataset. The architecture used in this example was,

$$[784] - [1000] - [400] - [20] - [400] - [1000] - [784]$$

and training was run for 50 epochs with a batch size of 500 images. Note that the bottleneck only contains 20 hidden nodes, meaning that each image is represented by a numerical vector of size 20. The left graph in fig. (4.20) shows 4 digits and the right graph shows the corresponding reconstructed images. The reconstruction is not perfect, but the bottleneck is on other hand rather small.

Applications of the SdAE

Just as for smaller autoencoders, the SdAE can be used for outlier detection and missing value imputation. Furthermore, a deep SdAE can do unsupervised dimensional reduction for other deep networks. For example, in a complex classification task with high-dimensional inputs, an SdAE can be used to represent data in a small bottle-neck. Then, the decoder part can be omitted, and the bottleneck result can be fed into another network that performs the classification. In fact this was one of the first important applications of the autoencoder.

4.4 Autoencoders with CNNs

In the example in fig. (4.20), the MNIST images were immediately flattened to long input vectors to an autoencoder. It would be tempting to use the CNN architecture instead. It is designed to do well on images and with some stride it typically reduces the initial image to a small "bootleneck" image.

One method to decode a convolution is called "transposed convolution" and is illustrated in fig. (4.21), based on gifs presented at the web page "An Introduction to Different Types of Convolutions in Deep Learning".[3] A convolution of the input image creates an "encoded image", which then is decoded to a "reconstructed image". The procedure is:

- The encoder and decoder kernels have the same size.
- The stride during decoding is 1.
- If stride>1 was used during encoding, zero-nodes are *interspersed* to separate the encoded nodes by the encoding stride.
- The encoded image is padded to give the reconstructed image the correct size.

In fig. (4.21), the two first images illustrate a convolution with a 3x3 kernel and stride 2. Before decoding, the nodes are set a distance 2 apart (the encoder stride) and the image is padded to size 7x7. Then, 3x3 convolution with stride 1 creates a 5x5 image.

[3] https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d

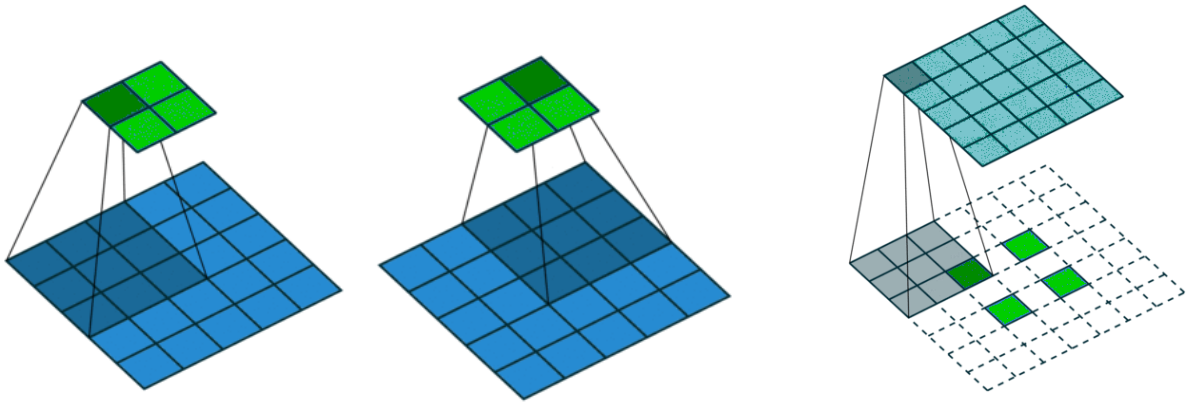


Figure 4.21: The two left pictures illustrate convolution with a 3x3 kernel and stride 2. The right picture shows how interspersing and padding is added so that a 3x3 kernel with stride 1 reconstructs a 5x5 image.

4.5 The variational autoencoder

The variational autoencoder (VAE) introduces an explicit uncertainty to the projection onto the bottleneck. Most importantly, the VAE is our first meeting with a network that not only analyses data, but that can generate new data! This will be further investigated with generative adverserial networks (GANs) in the next chapter.

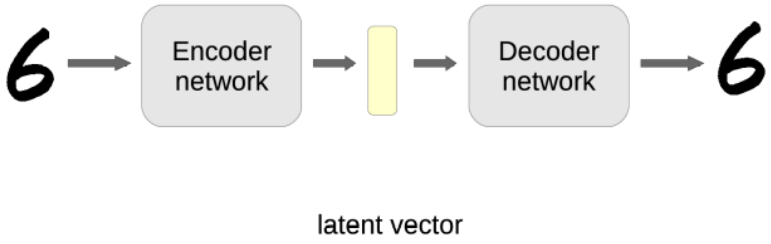


Figure 4.22: The general design of an autoencoder. There is an encoder part, followed by the bottleneck, here called latent vector. The decoder is then taking the latent vector and reconstructs the input.

Figure 4.22 shows the standard autoencoder design, where an input $\mathbf{x}_n$ is being encoded by the encoder part of the network ending up with a representation given by the bottleneck, here called the *latent vector* $\mathbf{z}_n$. The decoder part is then taking the latent vector and reconstructs the input as well as possible.

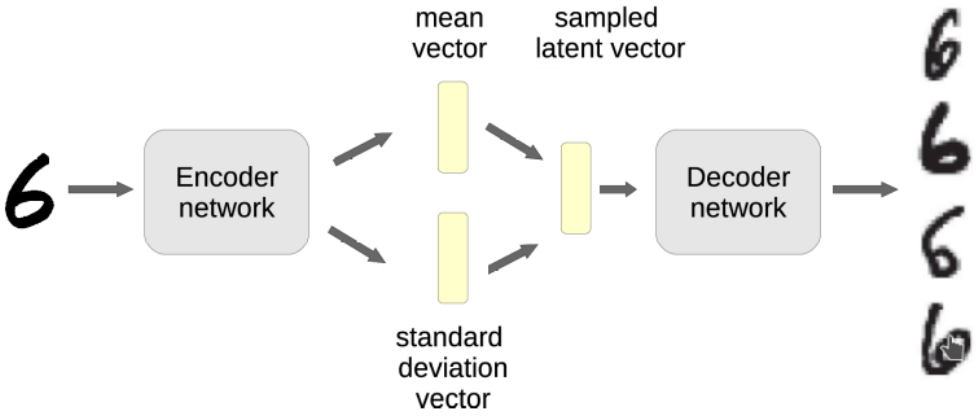


Figure 4.23: The general design of a variational autoencoder. The bottleneck is now divided into two vectors, the mean and the standard deviation vectors. These are specifying a normal distribution that can be sampled from. Each sample is then used as the input to the decoder, that is now *generating* a new output.

The VAE doubles the latent vector into one *mean* vector $\boldsymbol{\mu}_n$ and one *standard deviation* vector $\boldsymbol{\sigma}_n$. To use the decoder, the bottleneck vectors are used to generate a random, latent vector $\mathbf{z}_n$. Typically, each component z_{ni} is picked from a normal distribution with mean μ_{ni} and standard deviation σ_{ni}. The resulting sample is fed to the decoder to generate the output. Fig. (4.23) shows an example where four different $\mathbf{z}_n^{(s)}$ are sampled from the distribution determined by the input $\mathbf{x}_n$, resulting in four different images that all to some extent resemble the initial image.

How do we train a VAE? For each sample $\mathbf{z}_n^{(s)}$, we can define a loss $E_n^{(s)}$ that compares inputs $\mathbf{x}_n$ to the output $\mathbf{x'}_n^{(s)}$. Using backpropagation, we can train the decoder and also find derivatives $\frac{\partial E_n^{(s)}}{\partial z_{ni}^{(s)}}$. To train the encoder, we must find $\frac{\partial E_n^{(s)}}{\partial \mu_{ni}}$ and $\frac{\partial E_n^{(s)}}{\partial \sigma_{ni}}$. We can write $z_{ni}^{(s)} = \mu_{ni} + \sigma_{ni} t^{(s)}$, where $t^{(s)}$ is picked from a unit normal. Therefore $\frac{\partial z_{ni}^{(s)}}{\partial \mu_{ni}} = 1$ and $\frac{\partial z_{ni}^{(s)}}{\partial \sigma_{ni}} = t^{(s)} = (z_{ni}^{(s)} - \mu_{ni})/\sigma_{ni}$. This so-called "reparametrization trick" allows us to find gradients with respect to the mean and standard deviation vectors, and then continue back-propagation into the encoder.

If the loss only compares output to input, most patterns n would likely train $\boldsymbol{\sigma}_n$ to zero: once the best projection point $\boldsymbol{\mu}_n$ is found, any variation is to the worse. To avoid that, we can introduce a penalty for small σ in the loss function. The penalty can represent a *Bayesian prior* that we expect some variance around $\boldsymbol{\mu}$. The least informative prior (introducing as few unmotivated assumptions as possible) is that all patterns have the same standard deviation $\sigma_{ni} = \sigma^*$. Since the coordinate system in the latent space is arbitrary – whatever the encoder happens to create the decoder can train to interprete – we can without loss of

Figure 4.24: The leftmost image is the input to a VAE trained on the MNIST database. The four images to the right are then generated by the VAE.

generality set $\sigma^* = 1$. A popular loss term, motivated by a chi-squared prior for the variance, is $-\ln p(\sigma_{ni}^2) = \sigma_{ni}^2 - \ln \sigma_{ni}^2$, where constant terms are neglected. This has a maximum at 1 and strongly penalizes small values.

Generating data

When we have ensured that $\boldsymbol{\sigma}_n$ survives training, we can use the VAE as a *generative* model. A generative model produces both inputs and outputs. In our VAE case, the latent vector is the input, which we generate by sampling from a distribution, and the result of the decoder is the output.

Fig. (4.24) shows examples where the mean and standard deviation vectors from an image are used to generate many similar images. The architecture of this model was

$$[784] - [400] - [20] - [3+3] - [20] - [400] - [784]. \tag{4.19}$$

The next goal for a generative model is to generate reasonable data without trying to resemble one particular input. In the examples of fig. (4.23) and fig. (4.24), we still needed an input image to find $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. What if we just want to generate "any digit"? Could we do that with a trained decoder, without knowing any input images? To get there, we introduce a prior for the mean vectors $\boldsymbol{\mu}_n$. Again, the only reasonable prior is that all $\boldsymbol{\mu}_n$ are equal, and again we can exploit the arbitrariness of the latent space coordinate system to assume a unit normal prior. This gives a very simple loss term, where the only μ-dependent term in $-\ln p(\mu)$ is $\mu^2/2$.

After training, the encoder will create separate posteriors $\boldsymbol{\mu}_n$, to help the decoder identify and reconstruct different images, but with the chosen priors, the distribution of trained vectors $\boldsymbol{\mu}_n$ are likely to resemble a sample from a unit normal. To illustrate this, we use the architecture in eq. (4.19). Since its latent space is 3-dimensional, we can actually plot where the different patterns lie. Figure 4.25 shows a random subset of the images from the test MNIST data. The colours of the different points represent the 10 digits, as shown by the colourbar to the right. We can see how the different digits separate in the plot, even though there is overlap between the "classes". We can also see that the entire subset of data forms a rather Gaussian-like cloud, in agreement with the prior for $\boldsymbol{\mu}$.

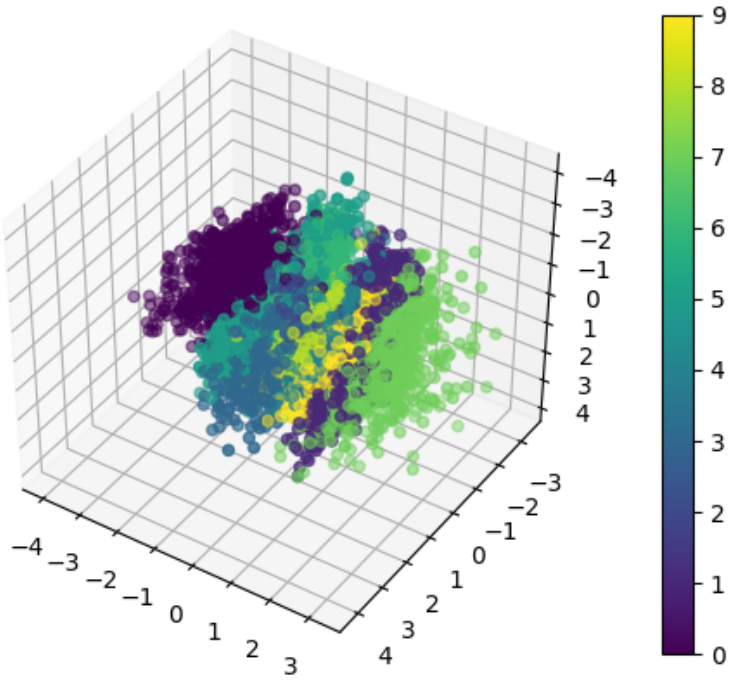


Figure 4.25: A visualization of the latent space for a VAE trained on the MNIST database. Each point in the figure corresponds the the mean vector on the VAE. The colour of the points denote the digit of the input image, as explained by the colourbar to the right.

Now we can use the decoder to generate a reasonably pretty "digit", by simply picking a latent vector $\mathbf{z}$ from a unit normal and feed into the decoder! It could of course create some very fuzzy images, perhaps some strange hybrids between different digits, and it could miss some kind of digits that were pushed far away from $\boldsymbol{\mu} = 0$ during training, but accepting such shortcomings, we have created a program that generates what seems to be hand-written digits! If we do not accept such shortcomings, we need to be more clever, which brings us to the next chapter.

4.6 Generative adversarial networks

The Generative adversarial network (GAN), invented 2014 by Goodfellow et al., belong to the category of generative models. Creating a generative model means here using training samples from some unknown distribution p_d to come up with an estimate p_m of this unknown distribution. We can do this by explicitly stating what p_m looks like or we can provide a model that can generate data from p_m. The main usage of the GAN is to be able to generate samples.

The GAN actually consists of two networks, called the *generator* network and the *discriminator* network. The purpose of the generator network is to create samples that look like they are coming from the training dataset. The discriminator network is trying to separate generated samples from "real" samples, i.e. it is just a binary classification network. Figure 4.26 shows the design of the GAN model. Both the generator and the discriminator can be simple MLPs, think of the generator as the decoder part of an autoencoder and the discriminator as being an ordinary MLP for binary classification. However, GANs are mostly used in connection with images and then both the generator and the discriminator are CNN type of networks.

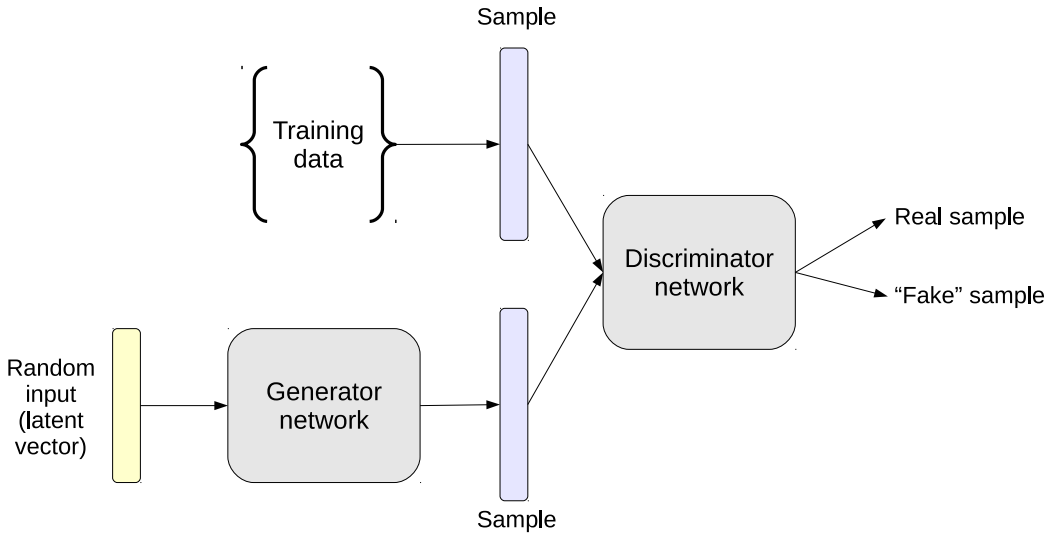


Figure 4.26: An illustration of the GAN model. Using a random input vector (latent vector) the generator is producing a sample with the aim of being similar to samples coming from the training dataset. The discriminator network on the other hand is taking both real and generated samples as inputs with the task of separating them from each other.

Training the GAN One can think of GAN training in terms of playing a game. The generator is trying fool the discriminator by making as good "fake" samples as possible, while the discriminator network is trying to become as good as possible to tell whether it is a fake sample or not. In the original problem formulation, training the GAN is presented as a max-min problem,

$$\min_{\mathrm{G}} \max_{\mathrm{D}} V(D, G)$$

where V is defined as

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]. \tag{4.20}$$

Here $D(x)$ represents the discriminator output and $G(z)$ the generator output. The output $D(x)$ is the probability that x is a real sample. The random variable z is drawn from the distribution $p_z(z)$ that generates inputs to the generator network. And x are samples drawn from the $p_{data}(x)$, represented by our training dataset. If we look at the max of V with respect to D we see that it is our usual cross entropy "error", but now being maximized since it is positive. For the first term all data comes from the training dataset and maximizing $\log D(x)$ means that $D(x)$ should be one. For the second term all data are coming from the generator network, hence $D(G(z))$ should be zero, i.e. maximizing $\log(1 - D(G(z)))$. Now, the min of V with respect to G does the opposite. It wants to make generated samples look like the real samples, meaning that the discriminator should classify a generated sample as real, i.e. minimize,

$$\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

Note that the first term of eqn. 4.20 is independent of G and is therefore just a constant with respect to the minimization of V. It turns out that training a GAN model is not a trivial process. Both the generator and the discriminator need to improve at a similar pace, otherwise one can for example end up in a local minimum where the discriminator is perfect and the generator looses its generative ability and just outputs some fixed average sample.

Some example of GAN models There is a lot of research going on at the moment concerning GAN models. How to improve the training procedure and how to evaluate them are such topics. Below are a few titles where GAN models are being used in applications:

- Generative Adversarial Text to Image Synthesis
- Image-to-Image Translation with Conditional Adversarial Networks
- Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

4.6.1 Cycle GAN

There are many extensions to the original GAN model presented above. The GAN Zoo contains a list of some of these extensions and modifications. We will look at one extension called *Cycle GAN*. This model is suitable for image-to-image translation. Suppose we want to transfer a landscape image taken a summer day into the same landscape, but now on a winter day. We can imagine that this problem could be solved using a large set of paired images, e.g. a lot of landscapes taken both on a summer day and a winter day. Then we could train a CNN to do the mapping of summer to winter landscape. The main limitation with this approach comes from the requirement of having a paired dataset. This can be difficult, and in some cases even impossible, to obtain. The cycle GAN model do not have that requirement and can work with a set of unpaired images. For the cycle GAN we can use a set of summer landscapes images and a unrelated set om winter landscape images. Figure 4.27 shows an example of paired and unpaired image sets.

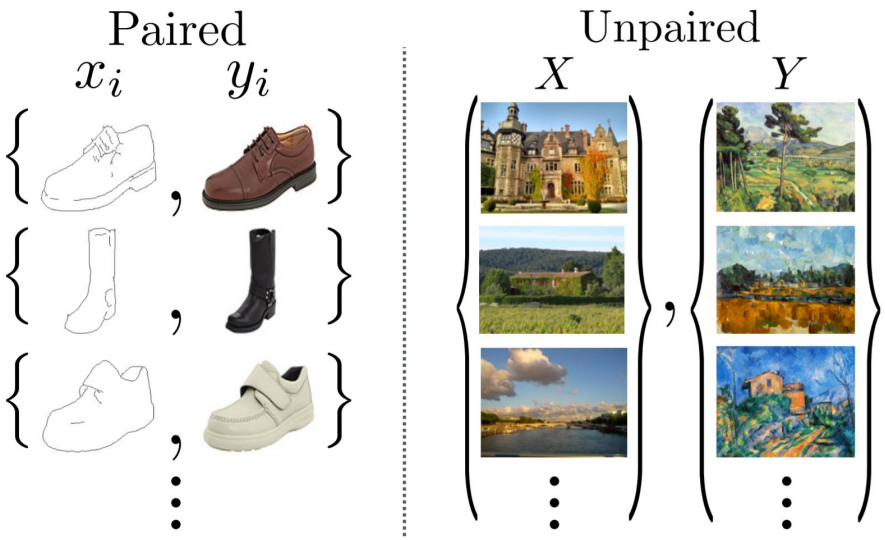


Figure 4.27: (Left) A set of paired images where there exists a correspondence between each pair of images. (Right) A set of unpaired images, where no correspondence exists. (This figure is taken from the original cycle GAN paper.)

Assume we have training data from the domain X and one set of training data from the domain Y. The cycle GAN uses two generators $G : X \to Y$ that convert images from domain X to domain Y, and $F : Y \to X$ that converts from Y to X. We also have two discriminators D_X and D_Y where D_X discriminates x from $F(y)$ and D_Y distinguishes y from $G(x)$. The process is illustrated in Figure 4.28. There is an additional image generation process for the

cycle GAN, each generated image is also fed into the reverse generator as to "cycle" back to the original image.

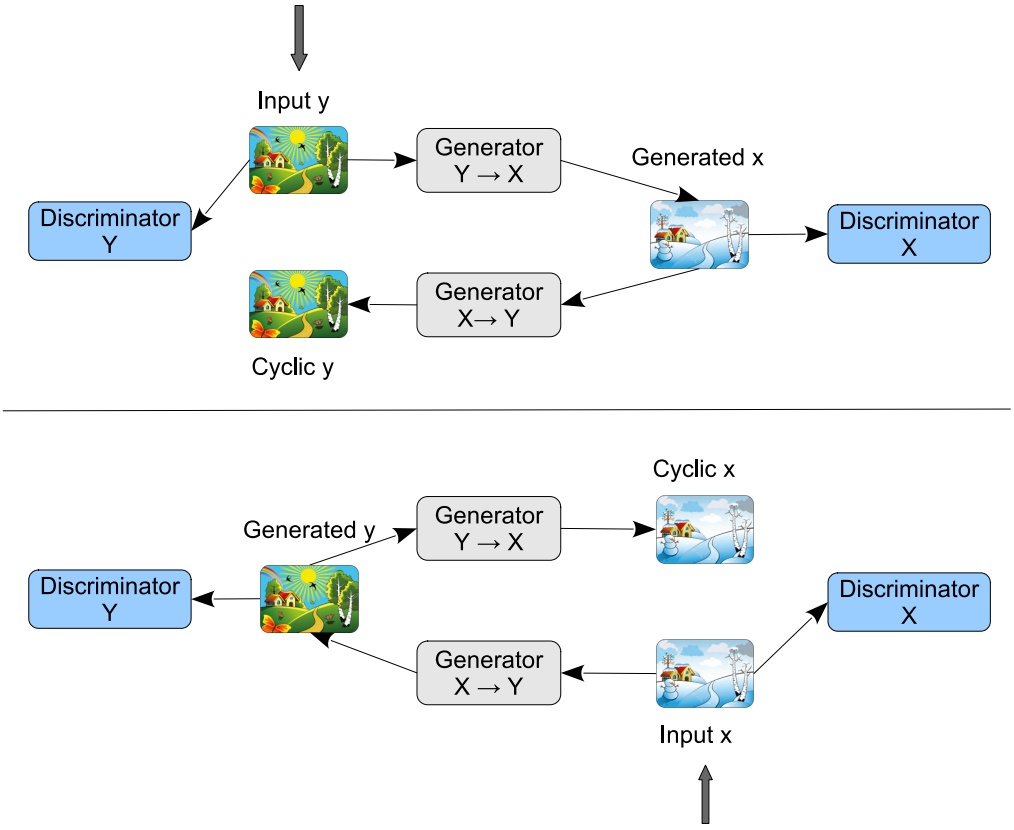


Figure 4.28: An illustration of the Cycle GAN model. (Top) An input y (summer landscape) is translated into an x (winter landscape) and then cycled back into the original image. Discriminator D_Y is taking the y image and separates that one from the generated y image (from bottom). (Bottom) An input x (winter landscape) is translated into an y (summer landscape) and then cycled back into the original image. Discriminator D_X is taking the x image and separates that one from the generated x image (from top).

The objective function for the cycle GAN consists of two parts. The first part is the same as the objective for the original GAN (*adversarial loss*), but we need one for each pair of generator and discriminator. For the G mapping and its discriminator D_Y we have the objective,

$$V_{\text{GAN}}(G, D_Y) = \mathbb{E}_{y \sim p_{data}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{data}(x)}[\log(1 - D_Y(G(x))] \tag{4.21}$$

where G tries to generate images $G(x)$ that look similar to images from domain Y, while D_Y will try to distinguish between translated samples $G(x)$ and real samples y. We have

similar objective for the F mapping and D_X,

$$V_{\text{GAN}}(F, D_X) = \mathbb{E}_{x \sim p_{data}(x)}[\log D_X(x)] + \mathbb{E}_{y \sim p_{data}(y)}[\log(1 - D_X(F(y))] \tag{4.22}$$

Training the cycle GAN would then translate into the following max-min problem,

$$\min_{\text{G,F}} \max_{D_y, D_x} V_{\text{GAN}}(G, D_Y) + V_{\text{GAN}}(F, D_X)$$

There is however a problem with this approach. There is nothing that "forces" the translated images to look the same. A winter landscape of a lake could easily be translated to a summer landscape of a forest. To address this problem the authors introduced the *cycle consistency* loss. This loss ensures that translating a image from X to Y and then translating it back to X again should result in the same starting image, i.e. $F(G(x)) \approx x$ and $G(F(y)) \approx y$. The suggested loss function was,

$$V_{\text{cyc}}(G, F) = \mathbb{E}_{x \sim p_{data}(x)}[||F(G(x)) - x||_1] + \mathbb{E}_{y \sim p_{data}(y)}[||G(F(y)) - y||_1] \tag{4.23}$$

The final objective function is a combination of adversarial losses (Eqn. 4.21 and 4.22) and the cycle consistency loss (Eqn. 4.23),

$$V(G, F, D_X, D_Y) = V_{\text{GAN}}(G, D_Y) + V_{\text{GAN}}(F, D_X) + \lambda V_{\text{cyc}}(G, F) \tag{4.24}$$

In their orginial paper λ was set to 10. The training then results in two generators G^* and F^* as found by,

$$G^*, F^* = \min_{\text{G,F}} \max_{D_y, D_x} V(G, F, D_Y, D_X)$$

Here are a couple of examples from the original cycle GAN paper.

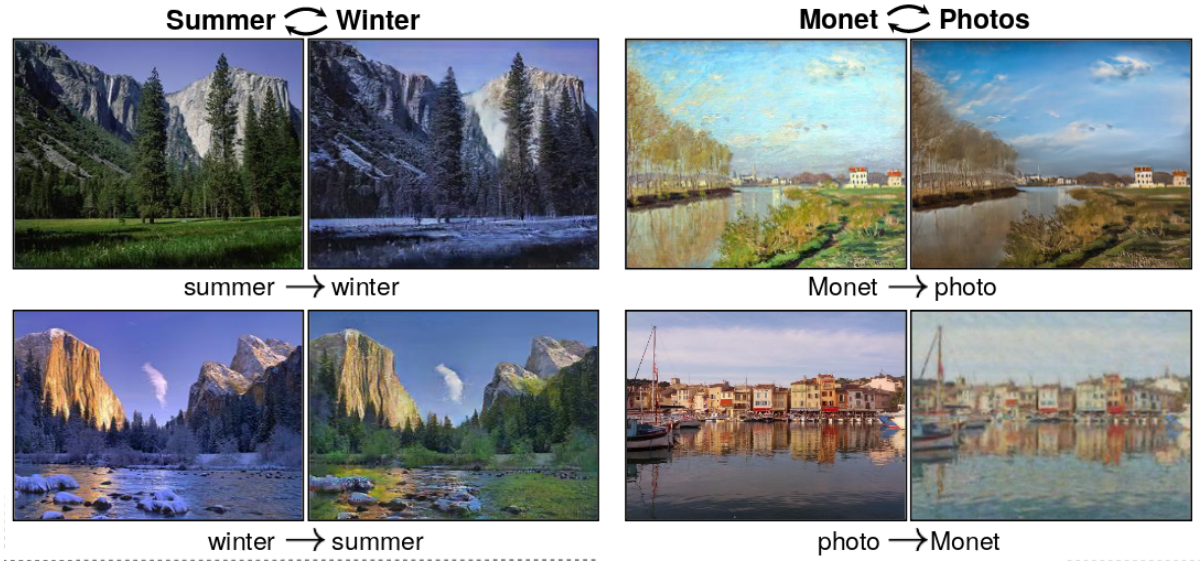


Figure 4.29: Translation of summer and winter images and taking a photo and turning it into a Monet painting.

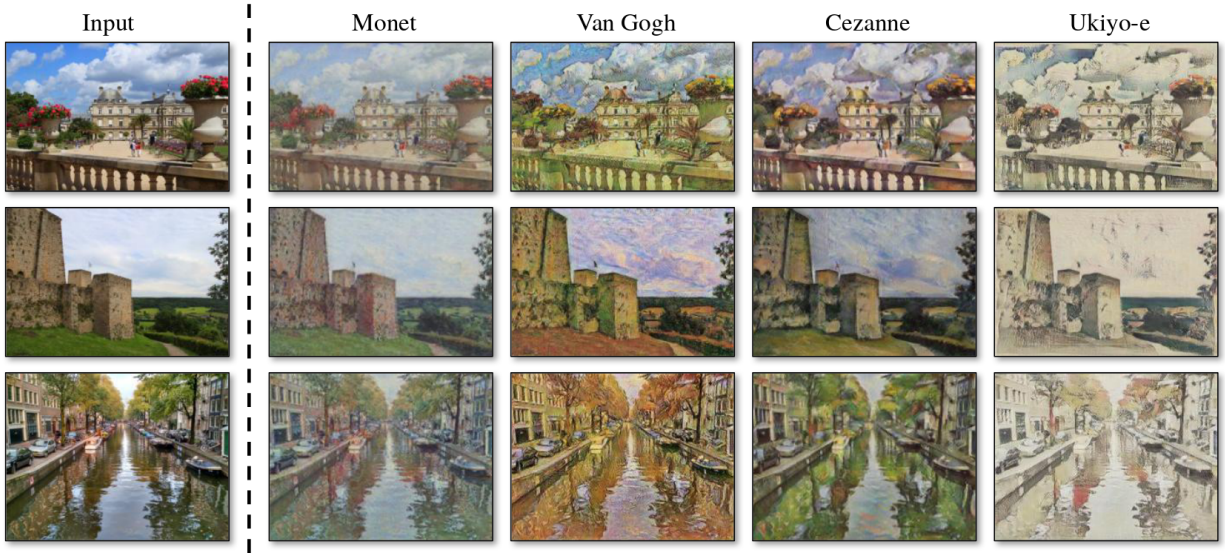


Figure 4.30: More of images to paintings.

Chapter 5

Recurrent Neural Networks

5.1 Introduction

Common for all neural network models we have studied so far is the lack of feedback connections. We will now study networks with such connections! Recurrent networks are typically used when we are dealing with sequence data. It can be text data, speech data or numerical times series data coming from eg. sensors or stock markets. The feedback connections are used to capture the short and long term temporal dependences in the data. We will however also look into ways of dealing with sequence data using ordinary MLPs. Let's start with that!

5.2 Time delay networks

The time delay neural network is an ordinary MLP with no feedback connections. All of the possible temporal dependences are modeled using a fixed number of delayed copies of the sequence data. Let's take an example! Suppose we have a single time series, eg. the yearly sunspot number time series, as shown in figure 5.1. The task is now to train a time delay network that can predict next years sunspot number. Suppose the sunspot data is given in $x(t)$. The approach is to use a fixed number of "previous" values of $x(t)$ in order to predict $x(t+1)$. From $x(t)$ we create the following input-output dataset that will be used to train the MLP.

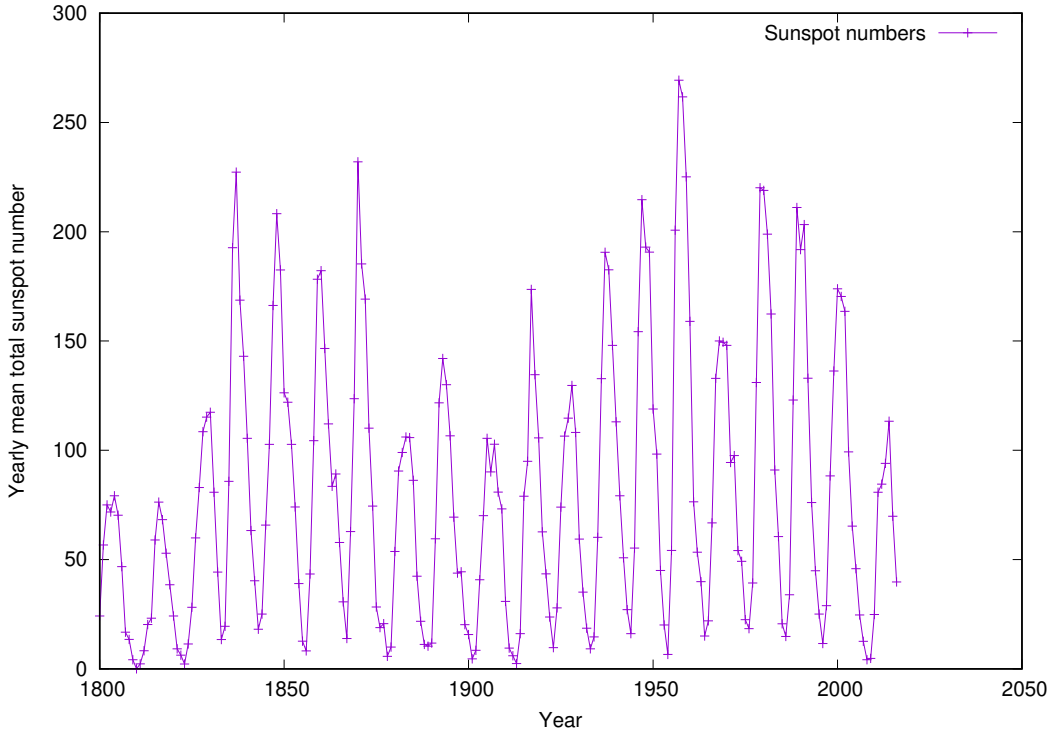


Figure 5.1: The sunspot number timeseries from 1800 to 2016.

Data no	Input	Target
1	$x(t-1), x(t-2), x(t-6), x(t-12)$	$x(t)$
2	$x(t-2), x(t-3), x(t-7), x(t-13)$	$x(t-1)$
3	$x(t-3), x(t-4), x(t-8), x(t-14)$	$x(t-2)$
...	...	...

Here we use four time delays, $t-1, t-2, t-6, t-12$, meaning that the MLP will have four inputs (and a single output). The possible temporal dependences needed to model this time series are static since we have specified the exact nature of this dependence.

Figure 5.2 shows an example when training a time delay neural network for the sunspot data above. Here five inputs were used in an MLP with 4 hidden nodes and a single output for making one year into the future predictions. The inputs were $x(t-1), x(t-2, x(t-3), x(t-6), x(t-12)$ and the corresponding target value was $x(t)$. The data was divided into a training and a test set. Training data consisted of the years between 1700-1930 and test data between 1931-2016. The network was trained without any regularization until convergence using SGD.

The top graph in figure 5.2 shows the training performance as "single step prediction",

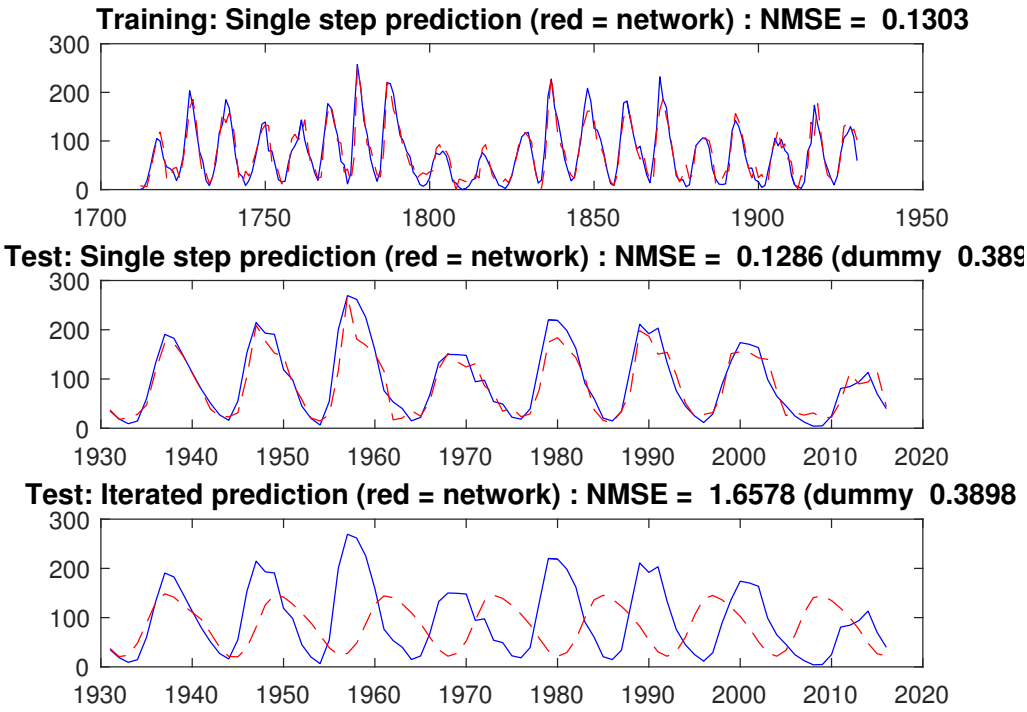


Figure 5.2: The result of training a time delay neural network to predict the yearly sunspot number. This network used 4 hidden nodes and 5 inputs consisting of time delayed inputs at $t-1, t-2, t-3, t-6, t-12$. The "NMSE" refers to the normalized mean square error.

meaning that the network was always fed with correct input values and not predicted ones. The middle graph shows the test performance, again single step prediction. As a comparison we can use the very simple model,

$$y(t)_{\text{dummy}} = x(t-1)$$

meaning that the predicted yearly sunspot number is the same as previous year. The error for this model is referred to in the text as the "dummy" model. The lower graph in the figure shows so called iterated prediction, where test predictions are used as input to the network to predict coming years sunspot number. Here we can see that this is a much more difficult task!

So, one can use time delay networks if the there is a simple "static" dependence on a fixed number of previous time steps for the sequence data. The advantage of using time delay networks when possible, is that they can be simpler to train than recurrent networks.

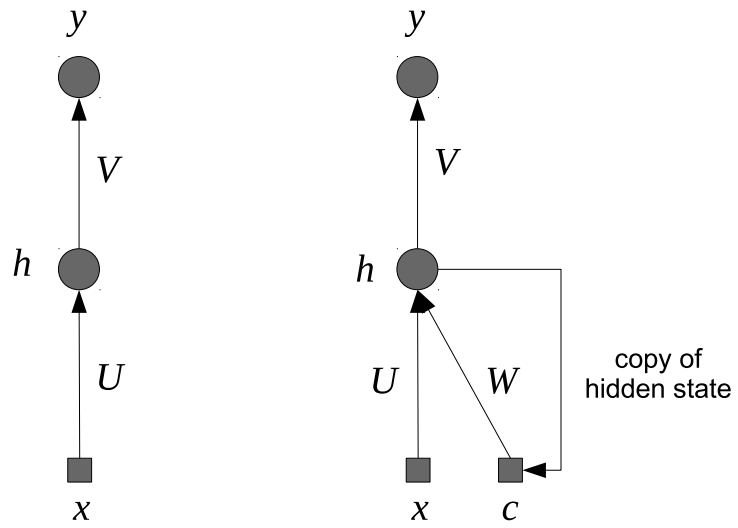


Figure 5.3: (Left) A simple 1-1-1 network. (Right) A simple recurrent network were a copy of the hidden state is used as an input to the network.

5.3 Simple recurrent networks, Elman type

We are now going to look at a class of networks that often goes under the name of "simple recurrent networks". They are also sometimes called *Elman* networks. Figure 5.3 shows such a simple network (right graph). For simplicity the bias node has been omitted. There is a reason for drawing this network in such a way, using the phrase "copy of hidden state". In principle we do not have any trainable feedback weights in this network, it only consists of feed-forward weights that can be trained using *almost* ordinary backpropagation.

So, how does the Elman network work? First we need to introduce a sequence index, let's use t. We have inputs in form a sequence $x(0), x(1), ..., x(T)$. The basic operation for the feedback network in figure 5.3 is,

$$y(t) = g_{\mathrm{o}}\left(Vh(t)\right) \tag{5.1}$$
$$h(t) = g_{\mathrm{h}}\left(Ux(t) + Wc(t)\right) \tag{5.2}$$
$$c(t) = h(t-1) \tag{5.3}$$

where g_{o} and g_{h} are the activation functions for the output and the hidden node respectively. So the input to hidden node is a weighted sum of the input and the **previous** value of the hidden node. Given the input sequence $x(t)$ one can produce an output sequence $y(t)$ according to eqn 5.1 above.

Training the above network can be done in many different ways. The idea presented below follows the original model proposed by Elman (more or less). The key point here is that we

forget that we have a dependence on the history through $c(t)$ and treat the training of this network as training for an ordinary MLP. Let's assume a simple mean square error function of the form,

$$E = \frac{1}{T}\sum_{t=0}^{T} E_t = \frac{1}{T}\sum_{t=0}^{T}(d(t) - y(t))^2$$

where $d(t)$ is the target sequence that we would like to model. The gradient descent approach means that we need, at some point, to compute the derivatives like $\partial y(t)/\partial W$. We have

$$\begin{aligned}
\frac{\partial y(t)}{\partial V} &= g_{\mathrm{o}}'(\cdot)h(t) \\
\frac{\partial y(t)}{\partial U} &= g_{\mathrm{o}}'(\cdot)V g_{\mathrm{h}}'(\cdot)x(t) \\
\frac{\partial y(t)}{\partial W} &= g_{\mathrm{o}}'(\cdot)V g_{\mathrm{h}}'(\cdot)c(t)
\end{aligned}$$

where the notation $g_{\mathrm{o}}'(\cdot)$ simply means the derivative of g_{o} evaluated at the current argument. But did we not miss something here? Since $c(t)$ in principle also depends on W through the relation $c(t) = h(t-1)$, meaning that we should treat the derivative,

$$\frac{\partial}{\partial W}(Wc(t))$$

as a derivative of a product. We will, for this version of the Elman network, simply ignore this dependence. This is why we use the phrase "copy of hidden state" in figure 5.3. We simply treat the network as a feed-forward network. The original Elman training was using "on-line" according to,

1. present an input $x(t)$
2. compute the hidden node value using $x(t)$ and the previous hidden node value $c(t)$. If $t = 0$ then $c(0)$ is typically set to zero
3. compute the output $y(t)$
4. perform all calculations (backpropagation) to compute all weight updates for this input "pattern"
5. update the weights V, U, W
6. go back to item 1, using the next input value in the sequence $(t \to t+1)$

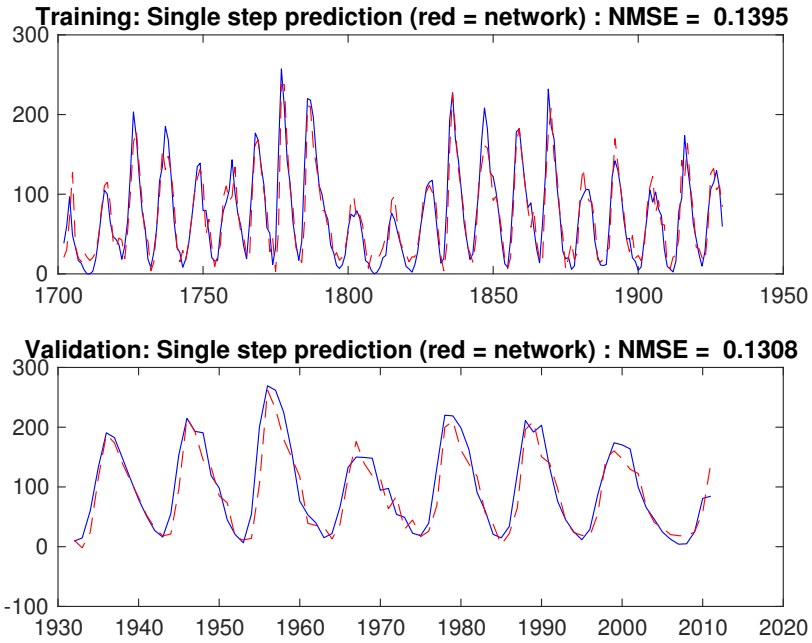


Figure 5.4: The result of training an Elman network for the sunspot number time series.

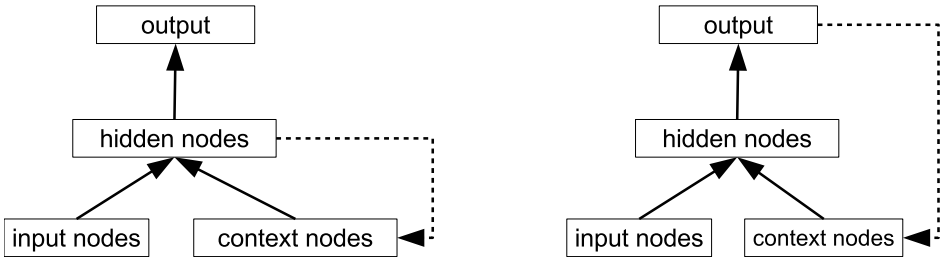


Figure 5.5: (Left) An Elman network. (Right) A Jordan network.

Fig. (5.4) shows the result of training an Elman network (using Matlab and the neural network toolbox). Here 5 hidden nodes were used, meaning that the Elman network had 6 inputs (the single time series input + 5 context nodes). The network was trained using SGD with no regularization.

Figure 5.5 shows the general Elman architecture together with the so called Jordan network where the feedback is coming from the output layer instead of the hidden layer. The extra input nodes are often called *context* nodes.

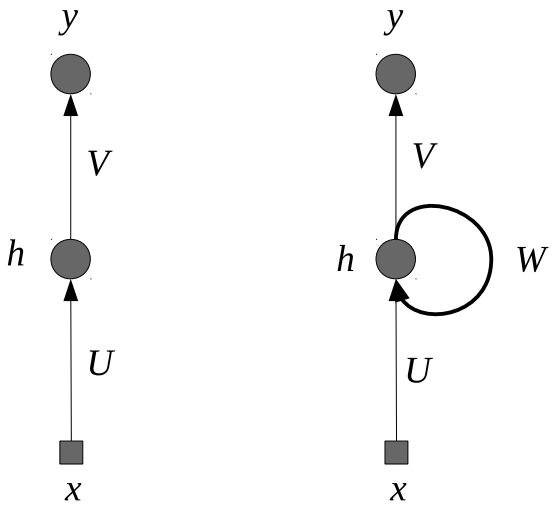


Figure 5.6: (Left) A simple 1-1-1 network. (Right) The same network but with an added feedback weight from the hidden node to itself.

5.4 Simple recurrent networks, general type

We will now start to look at networks were we have "true" feedback connections. Figure 5.6, again shows a very simple network with one input, one hidden node and one output node (left graph). For simplicity the bias weights are not shown in this plot. In the graph to the right we have now added a feedback weight W, feeding from the hidden node to itself. It is similar to the Elman network in figure 5.3, but we are going to treat this feedback weight in a proper way.

Again we have input data in form of a sequence $x(0), x(1), ..., x(T)$, with t being the sequence index. The operation of this network is now,

$$y(t) = g_{\mathrm{o}}\left(Vh(t)\right) \tag{5.4}$$
$$h(t) = g_{\mathrm{h}}\left(Ux(t) + Wh(t-1)\right) \tag{5.5}$$

again, g_{o} and g_{h} are the activation functions for the output and the hidden node respectively. Let's be explicit for a few sequence steps. (To avoid clutter I will use the notation $x(t) \rightarrow x_t$,

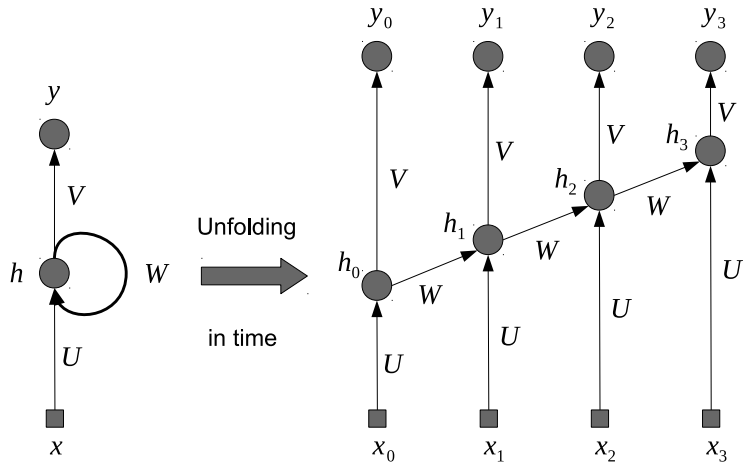


Figure 5.7: Unfolding in time. The simple feedback network to the left is "unfolded" for three time steps. The resulting network looks like a feed-forward network, although with a not so typical structure. Note that the weights are shared between the different layers.

$y(t) \to y_t$ and $h(t) \to h_t$ from now on. Also below, $g_{\mathrm{o}}(\cdot)$ and $g_{\mathrm{o}}[\cdot]$ means the same thing.),

$$
\begin{aligned}
y_0 &= g_{\mathrm{o}}\big[Vh_0\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_0\big)\big] \qquad \text{(using the initial condition } h(-1) = 0) \\
y_1 &= g_{\mathrm{o}}\big[Vh_1\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_1 + Wh_0\big)\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_1 + Wg_{\mathrm{h}}\big(Ux_0\big)\big)\big] \\
y_2 &= g_{\mathrm{o}}\big[Vh_2\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_2 + Wh_1\big)\big] = \\
&= g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_2 + Wg_{\mathrm{h}}\big(Ux_1 + Wh_0\big)\big)\big] = \\
&= g_{\mathrm{o}}\Big[Vg_{\mathrm{h}}\Big(Ux_2 + Wg_{\mathrm{h}}\big(Ux_1 + Wg_{\mathrm{h}}\big(Ux_0\big)\big)\Big)\Big]
\end{aligned}
$$

If we look at the last expression we see that it to some extent looks like an MLP with 3 hidden layers, although not fully connected and with shared weights. The procedure of turning a recurrent network into a MLP like structure is called *unfolding in time*. Figure 5.7 is showing our network unfolded in time for three time steps. Obviously this network will have many layers if we have long sequences.

5.4.1 Backpropagation through time (BPTT)

Let's look at the above simple network and see how it can be trained. First we need a training dataset! Again, assume that we have a an input sequence $x_t, t = 0, ..., T$ and a target sequence $d_t, t = 0, ..., T$. We can define an error function as

$$E(U, W, V) = \sum_{t=0}^{T} E_t(U, W, V)$$

Here E_t can be the usual square difference between y_t and d_t, but we can also imagine other possibilities such that we want to classify the input sequence into different classes (e.g. sequence of words that should be classified). To train using gradient descent we need to compute,

$$\begin{aligned}\frac{\partial E}{\partial V} &= \sum_t \frac{\partial E_t}{\partial V} \\ \frac{\partial E}{\partial W} &= \sum_t \frac{\partial E_t}{\partial W} \\ \frac{\partial E}{\partial U} &= \sum_t \frac{\partial E_t}{\partial U}\end{aligned}$$

The derivative of V is rather simple, since it sits at the topmost level. We have

$$\frac{\partial E_t}{\partial V} = \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial V} = \frac{\partial E_t}{\partial y_t} g_{\mathrm{o}}'(Vh_t)h_t$$

It is however more difficult for both W and U. Let's look at

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial W}$$

Now $h_t = g_{\mathrm{h}}(Ux_t + Wh_{t-1})$ (see eqn 5.5), where also h_{t-1} depends on W and so on, all the way down to the first "layer" (see figure 5.7). The correct way to write this is then,

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W} \tag{5.6}$$

We have a similar expression for the derivative with respect to U. In principle what we have done here is exactly what we would have done to compute the gradients in a MLP with many hidden layer. The key difference is that we here share weights which means that we have to sum up the gradients for e.g. W along the layers (i.e. time steps). This way of training a recurrent network by unrolling it in time and do standard backpropagation is called *backpropagation through time (BPTT)*. This method is common when training simple recurrent networks, however as it stands (eqn 5.6) it has problems with gradients that can become small or become very large, which we refer to as

- Vanishing gradients
- Exploding gradients

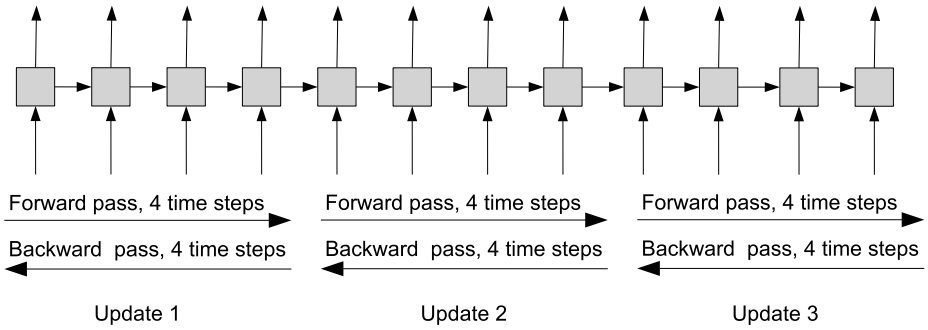


Figure 5.8: Truncated BPTT. Instead of making a full forward pass and a full backward pass, the sequence is divided into several smaller sequences. This will efficiently reduce the problem of vanishing gradients. It is common to remember the state of all nodes before starting the next sub-sequence.

The problem can be understood by looking at the equation 5.6 again. The derivative $\partial h_t/\partial h_k$ is in itself a chainrule. For example,

$$\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}$$

Remember that $h_t = g_{\mathrm{h}}\left(Ux_t + Wh_{t-1}\right)$, leading to,

$$\frac{\partial h_{t+1}}{\partial h_t} = g'_{\mathrm{h}}(\cdot)W$$

So for very long sequences we will get many multiplications of such gradient terms. Two numerical problems can occur: If either or both the weights and the gradients are small then this will lead to the vansishing gradient problem. This causes training to be very slow, especially for sequence data with long time dependences. There is also a risk of exploding gradients if weight starts to grow, as multiplication of many terms can result in very large gradients causing the learning to be unstable. There are a few ways to combat the vanishing and the exploding gradient problem, one can adjust weight initialization, and use rectifier linear units instead of tanh() or sigmoid activation functions. Another very common approach is to truncate the BPTT method. This means that we do not backpropagate for all time steps instead we run the BPTT method for a number of smaller sequences. Figure 5.8 is illustrating truncated BPTT. However, it may still be difficult to fully circumvent the problem. Another method is therefore to change the behavior of the hidden feedback nodes. The next section is looking at two such networks.

As a concluding remark, we have in all examples for this section (5.4) only had one single input and a single hidden node. Of course in practical examples we may have multiple inputs, many hidden nodes and even more than one hidden layer. This means that the

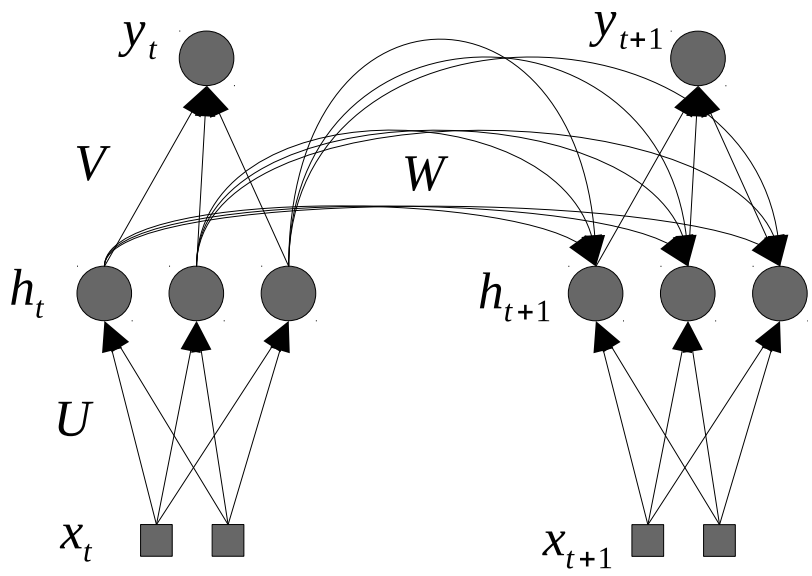


Figure 5.9: A more general recurrent network with three hidden nodes and two input nodes. Note that all hidden nodes have feedback connections to all other hidden nodes. This network is unfolded one time step.

weights U, W, V will be replaces by appropriate sized matrices instead, as illustrated by figure 5.9. However unfolding in time is as before, it is just that the unfolded network is a bit more complex.

5.5 LSTM networks

As we saw in the previous section, the BPTT approach to training a recurrent networks may have problems for long sequences because of the vanishing gradient problem. To handle long time dependences in an efficient way, the Long Short-Term Memory (LSTM) network was introduced (1997 by Schmidhuber).

To prepare for the LSTM network we start by illustating a simple recurrent network as in fig. (5.10). The "U" box represents the hidden node and consists of h_t and x_t feeding into a typical tanh() function. In the LSTM network we are going to replace the "U" box in fig. (5.10) with a LSTM box as in fig. (5.11). Apart from the extra c_t value we can just consider the "LSTM" box as another way of computing the new hidden value h_t. In this respect the LSTM is similar to the simple feedback node, however the computation inside the LSTM box is very different from the "U" box.

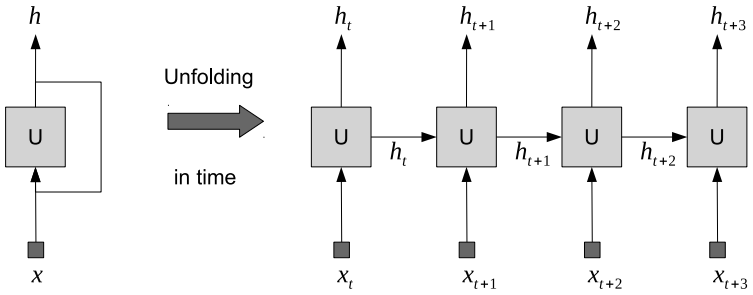


Figure 5.10: Unfolding in time. (Left) A single hidden node with a feedback weight. (Right) This hidden node unfolded in time.

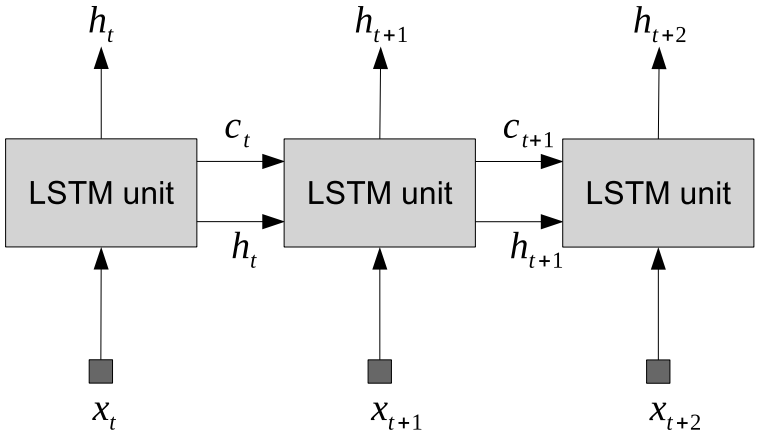


Figure 5.11: A LSTM network is basically the same as simple recurrent network (figure 5.10) where we have replaced the simple "U" box with a new "LSTM" box. We have also added a new value c_t.

In addition to the node value (output from the node) h_t we have an extra value c_t. This is the internal memory of the node. There are also *gates*, denoted f, i, o and called forget, input and output gates, respectively. They are all numbers between 0 and 1, and will be used to "filter" the new values. They all have the same structure,

$$\begin{aligned} i_t &= \sigma(x_t U^i + h_{t-1} W^i) \\ f_t &= \sigma(x_t U^f + h_{t-1} W^f) \\ o_t &= \sigma(x_t U^o + h_{t-1} W^o) \end{aligned}$$

where (U^i, U^f, U^o) and (W^i, W^f, W^o) are new weights and $\sigma(\cdot)$ is the logistic function. Note that where appropriate bias weights should also be added, but we have removed them from the formulas for simplicity. Given all these gates we will decide new values for the internal

memory c_t and for the output h_t. We start by computing a candidate value $\tilde{c}_t$ for the memory,

$$\tilde{c}_t = \tanh(x_t U^c + h_{t-1} W^c)$$

This is exactly the same expression as we used to compute the output value for the simple recurrent networks (see eqn. 5.5). We have just renamed the weights U and W to U^c and W^c. We are now in the position of updating c_t and h_t using the gate values according to,

$$\begin{aligned} c_t &= c_{t-1} f_t + \tilde{c}_t i_t \\ h_t &= \tanh(c_t) o_t \end{aligned}$$

The new memory value c_t is a combination of the previous values "filtered" by the forget gate f, and the candidate value $\tilde{c}$ "filtered" by the input gate i. Finally the new output value of the node h_t is the memory value c_t taken through tanh() to push it between -1 and 1, and filtered by the output gate o. It is the gating mechanism that allows the LSTM network to model long-term dependences. These dependences can be learned by training the various gating weights. For the equations above we have used a simple example of a single hidden node network, i.e one LSTM unit. In practical examples one typically uses many LSTM nodes. This means that the weights in the above equations should be replaced by appropriate matrices. The gates are then effectively vectors of values between 0 and 1.

As final remark let us try to understand why we can avoid vanishing or exploding gradients for the LSTM network. Previously we looked at $\partial h_t / \partial h_{t-1}$ to understand why gradients could vanish or explode. The relevant derivative for the LSTM node becomes $\partial c_t / \partial c_{t-1}$. We then have,

$$\begin{aligned} \frac{\partial c_t}{\partial c_{t-1}} =& c_{t-1} \sigma'(\cdot) W^f o_{t-1} \tanh'(c_{t-1}) \\ &+ \tilde{c}_t \sigma'(\cdot) W^i o_{t-1} \tanh'(c_{t-1}) \\ &+ i_t \tanh(\cdot) W^c o_{t-1} \tanh'(c_{t-1}) \\ &+ f_t \end{aligned}$$

The details of this expression are not important. Similar to previous expressions for the gradient of different weights (see e.g. 5.6), we will multiply many terms like $\partial c_t / \partial c_{t-1}$ above when computing the such gradients. What we can conclude from the above expression is that such terms can be both < 1 and > 1, hence avoiding gradients to vanish or explode. It is in principle possible for the network to learn when certain gradients should vanish.

5.5.1 Gated recurrent network (GRU) variant

The gated recurrent network (GRU) is a simpler version of the LSTM network. It is a rather new and was first mentioned in 2014. The core idea of the GRU network is the same as for

the LSTM, but it does it in a simpler way. While the LSTM have three gates the GRU node only has two, reset gate r and an update gate z. The reset gate determines how to combine the new input with the previous value and the update gate is used to determine how much of the current value to keep. The equations for them are,

$$\begin{aligned} z &= \sigma(x_t U^z + h_{t-1} W^z) \\ r &= \sigma(x_t U^r + h_{t-1} W^r) \end{aligned}$$

We then have a candidate for the output value $\tilde{h}$ and the new output value h_t according to,

$$\begin{aligned} \tilde{h} &= \tanh(x_t U^h + (h_{t-1} r) W^h) \\ h_t &= (1 - z)\tilde{h} + z h_{t-1} \end{aligned}$$

If we compare LSTM and GRU we can see,

- GRU has two gates and LSTM has three
- GRU does not have an extra internal memory c_t as the LSTM has.
- We do not use an extra nonlinear function to compute the output as the LSTM does.

Note if we set the reset gate r to 1 and the update gate z to 0, we end up with a simple recurrent node. Figure 5.12 shows and illustration of the LSTM and the GRU nodes.

As a final remark, we have not talked about how to train the LSTM/GRU networks. One can use the same idea as for simple recurrent networks, backpropagation through time! The problem of vanishing gradients are avoided using the gating mechanism. So we can train the LSTM/GRU networks using SGD and all the possible enhancements that are available to speed up the minimization process.

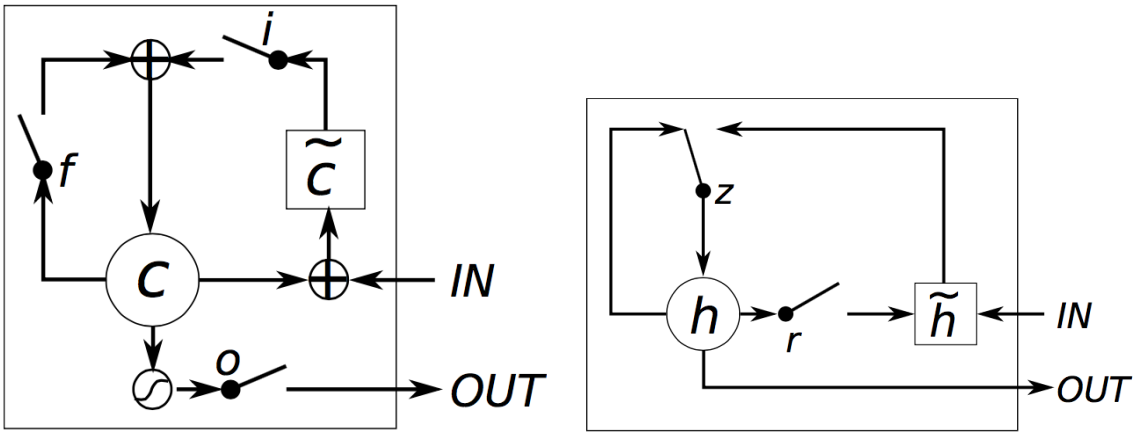


Figure 5.12: Images taken from *GRU Gating. Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." (2014).* (left) The LSTM node where we have three different gates and the extra memory c. (right) The GRU node where we only have two gates and no extra internal memory.

5.6 Recurrent backpropagation

We will now look at a more general formulation of recurrent networks in order to come up with a different approach of how to train a recurrent network. We will consider networks with N units v_i with connections ω_{ij} and activation functions $g(h)$. Some of the units may act as input units and we have input values x_{ni} for pattern n. We set x_{ni} to zero for units i that are not input units. Some units may act as output units with associated target values d_{ni}. See fig. (5.13) We have the following update equation

$$v_i = g(\sum_j \omega_{ij} v_j + x_i) \qquad \text{(Pattern index } n \text{ omitted)}$$

We now make the critical assumption that updating the above equations will lead to a stable fix point. This means that we do not need the concept of sequence index. A given input pattern will lead to a fixed output "pattern". We can define an MSE-based loss (for a single pattern):

$$E = \frac{1}{2}\sum_k e_k^2 \qquad \text{with} \tag{5.7}$$

$$e_k = \left\{ \begin{array}{rl} d_k - v_k & \text{if } k = \text{output unit} \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.8}$$

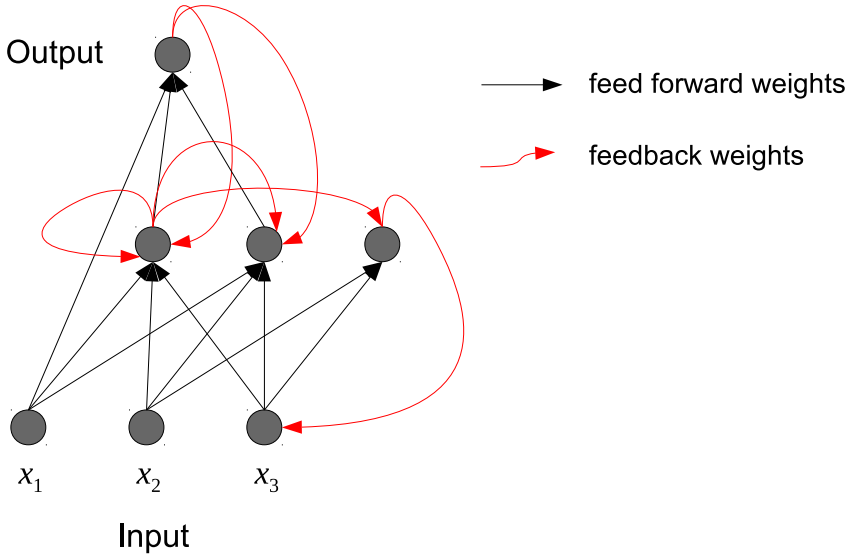


Figure 5.13: Example of a recurrent network. Red arrows denote feedback weights and black arrows are the usual feed-forward weights. Some nodes act as input nodes with external inputs and some are output nodes with known targets.

We will now try to update the weights using gradient descent, as usual,

$$\Delta\omega_{pq} = -\eta\frac{\partial E}{\partial\omega_{pq}} = \eta\sum_k e_k\frac{\partial v_k}{\partial\omega_{pq}}. \tag{5.9}$$

The derivative of the node v_i is

$$\begin{aligned}\frac{\partial v_i}{\partial\omega_{pq}} &= g'(h_i)\left(\delta_{ip}v_q + \sum_j \omega_{ij}\frac{\partial v_j}{\partial\omega_{pq}}\right) \Leftrightarrow \\ \frac{\partial v_i}{\partial\omega_{pq}} - g'(h_i)\sum_j \omega_{ij}\frac{\partial v_j}{\partial\omega_{pq}} &= g'(h_i)\delta_{ip}v_q,\end{aligned} \tag{5.10}$$

where

$$h_i = \sum_j \omega_{ij}v_j + x_i.$$

Now define the matrix $\mathbf{L}$ as

$$L_{ij} = \delta_{ij} - g'(h_i)\omega_{ij}.$$

This gives us the following relation from eqn 5.10

$$\sum_j L_{ij}\frac{\partial v_j}{\partial\omega_{pq}} = g'(h_i)v_q\delta_{ip}$$

This is simply a matrix equation where we can solve for $\partial v_j/\partial \omega_{pq}$, giving

$$\frac{\partial v_k}{\partial \omega_{pq}} = \left(L^{-1}\right)_{kp} g'(h_p) v_q$$

Using this expression in eqn 5.9 gives

$$\Delta\omega_{pq} = \eta \sum_k e_k \left(L^{-1}\right)_{kp} g'(h_p) v_q$$

The structure of this update equation is similar to that of ordinary backpropagation, except that we have to invert a matrix. Note that this derivation was based on a single pattern. We need to average the above update expression over several patterns if we want to use the stochastic gradient descent method. We can view this learning method as a way of using recurrent networks for solving problem where we previously used ordinary MLPs. The question is, did the feedback connections make solving the problem easier? The price to pay is a more complicated learning algorithm.

5.7 Realtime recurrent learning (RTRL)

In this section we will briefly look at what is called *realtime recurrent learning*. We will use the same network structure as in fig. (5.13). But this time we will not assume that there is a fixpoint of the network for a given input. Instead we are interested in learning a target sequence given a number of input sequences, as we did for simple recurrent networks. We therefore introduce a discrete "time" index t. So the basic update equation for a node in the above network is given by,

$$v_i(t) = g(h_i(t-1)) = g\left(\sum_j \omega_{ij} v_j(t-1) + x_i(t-1)\right)$$

Here $x_i(t)$ is an input to node i at time t. If no such input exists then we put $x_i(t) = 0$. Similarly we introduce $d_i(t)$ to be the target for node i at time t, if it exists.

Again we define node error measure $e_k(t)$,

$$e_k(t) = \left\{ \begin{array}{ll} d_k(t) - v_k(t) & \text{if } d_k(t) \text{ is defined at time } t \\ 0 & \text{otherwise} \end{array} \right\}$$

From this we can define the total error for the full sequence $t = 1, ...T$, as,

$$E_T = \sum_{t=1}^{T} E(t) \qquad \text{with} \qquad E(t) = \frac{1}{2} \sum_k e_k(t)^2$$

A gradient descent approach will need to compute gradients of $E(t)$ with respect to any of the weights ω_{pq}. We can define the weight update at time t according to,

$$\Delta\omega_{pg}(t) = -\eta\frac{\partial E(t)}{\partial \omega_{pq}}$$

and

$$\frac{\partial E(t)}{\partial \omega_{pq}} = \sum_k e_k(t)\frac{\partial v_k(t)}{\partial \omega_{pq}}$$

where we have

$$\frac{\partial v_i(t)}{\partial \omega_{pq}} = g'(h_i(t-1))\left(\delta_{ip}v_q(t-1) + \sum_j \omega_{ij}\frac{\partial v_j(t-1)}{\partial \omega_{pq}}\right)$$

The above equation relates $\partial v_i/\partial\omega_{pq}$ at time t to those at time $t-1$. Using the initial condition $\partial v_i(0)/\partial\omega_{pq} = 0$ we can start to compute the derivatives for all other times $t = 1, ..., T$. The online version of this makes an update at each t,

$$\omega_{pq} \to \omega_{pq} + \Delta\omega_{pg}(t) \text{ for } t = 1, ...T$$

Repeat until the model learns the correct sequence associations. Williams and Zipser, who are among the ones that constructed this algorithm, showed that this works for sufficiently small η. It goes under the name *realtime recurrent learning* (RTRL). Still the major problem with this approach and vanilla BPTT is vanishing gradients. Learning takes long time for large problems and where there is "complicated" sequence dependences. It is fair to say that truncated BPTT is more common then RTRL.

5.8 The Hopfield model

The Hopfield model is fully recurrent network, see figure 5.14, with the following properties:

- The nodes s_i in the network are binary with $s_i \in \{-1, 1\}$.
- We have symmetrical weights, i.e. $\omega_{ij} = \omega_{ji} \ \forall\, i, j$.
- No self feedback weights, i.e. $\omega_{ii} = 0 \ \forall\, i$.

The value of a given node follow the usual update equation for nodes, with a small change. With h_i the net input to node i, $h_i = \sum_j \omega_{ij}s_j$, the update is

$$s_i \to \begin{cases} s_i & \text{if } h_i = 0 \\ \text{sgn}(h_i) & \text{if } h_i \neq 0 \end{cases} \quad \text{where} \tag{5.11}$$

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{5.12}$$

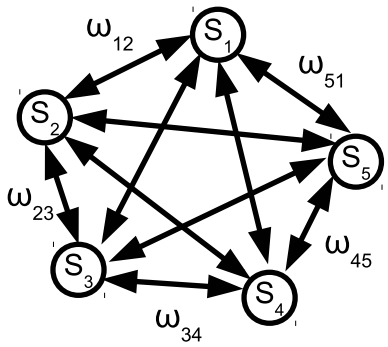


Figure 5.14: A Hopfield model with 5 neurons.

The updating of the nodes can be done in parallel or in serial. The most common mode is to use serial updating of the nodes.

5.8.1 Associative memory

The basic usage of the Hopfield model is to store patterns! What do we mean by that? Let's first introduce the state of the network. Our network has N nodes from which we can define the state vector $\mathbf{s} = (s_1, s_2, ..., s_N)$. A pattern $\boldsymbol{\xi}$ denotes one of the possible 2^N possible state vectors. If we want to separate between different patterns we use the notation $\boldsymbol{\xi}^\mu$ to denote pattern μ. Let also ξ_i^μ denote the i:th binary value of this pattern.

Now a pattern can be stable! By this we mean that if the Hopfield model is initialized with pattern μ ($\mathbf{s} = \boldsymbol{\xi}^\mu$) and we start to update the Hopfield model according to the above update formula (eqn. 5.11), nothing happens. This stability can be written as,

$$\text{sgn}(h_i^\mu) = \xi_i^\mu \qquad (i = 1, ..., N) \tag{5.13}$$

with $h_i^\mu = \sum_j^N \omega_{ij}\xi_j^\mu$. The phrase "associative memory" also means that if some of the "bits" of a stable pattern are changed, then when updating the Hopfield model it should retrieve the correct stable pattern. Next we need a way of training the Hopfield model to actually store patterns, i.e. making a number of patterns stable.

5.8.2 Storing patterns

We can call the process of storing a number of patterns in a Hopfield for training the model. As for other networks, training consists of finding appropriate values of the weights ω_{ij}. We are going to start with one pattern $\boldsymbol{\xi}$. For this case we propose the following,

$$\omega_{ij} = \frac{1}{N-1}\left(\xi_i\xi_j - \delta_{ij}\right) \tag{5.14}$$

where δ_{ij} is the Kronecker delta defined previously. We can easily check that this is working by looking at the stability condition.

$$\begin{aligned} \text{sgn}(h_i) &= \text{sgn}(\sum_j \omega_{ij}\xi_j) = \text{sgn}(\frac{1}{N-1}\sum_j (\xi_i\xi_j - \delta_{ij})\xi_j) = \\ &= \text{sgn}(\frac{1}{N-1}\sum_j \xi_i - \frac{1}{N-1}\xi_i) = \text{sgn}(\frac{N-1}{N-1}\xi_i) = \text{sgn}(\xi_i) = \xi_i \end{aligned}$$

is also easy to realize that if less than half of the bits of the starting pattern $\mathbf{s}$, is not equal to the stored pattern $\boldsymbol{\xi}$, then the update is going to result in $\boldsymbol{\xi}$. Figure 5.15 shows an example

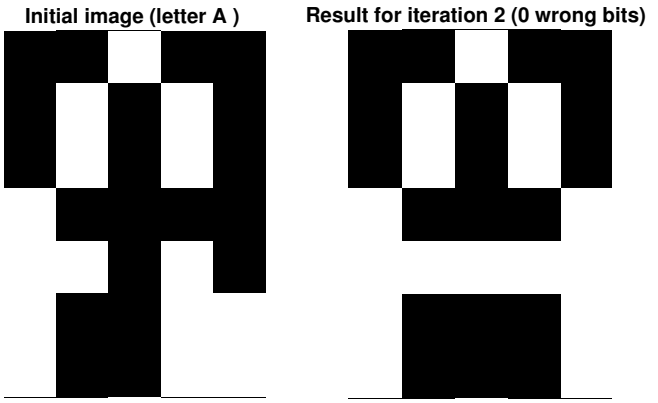


Figure 5.15: (left) A distorted letter "A" was used as the starting state of a Hopfield model that has been trained to store "A". (right) The result after 2 iterations, which is the original stored "A".

where the letter "A" (7x5 pixels) has been stored in a Hopfield model with $7 * 5 = 35$ nodes, giving rise to 35^2 weights. A white pixel corresponds to e.g. the $+1$ state of the nodes and a black pixel to the -1 state. The left figure shows a distorted "A" and in the right figure the result after 2 iterations of the model. As we can see it has recovered the stored image.

Generally we can say that $\boldsymbol{\xi}$ is an attractor, as illustrated in Figure 5.16. We also realize that $-\boldsymbol{\xi}$ is a stable pattern, by checking the stability condition.

To store more than one pattern we simply make a superposition of many patterns. Assume

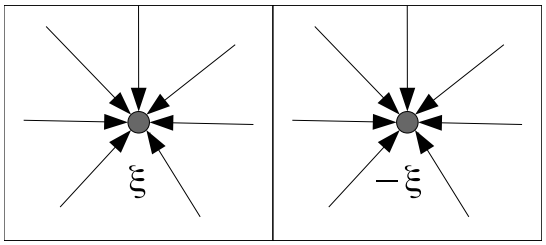


Figure 5.16: An illustration of the retrieving of one stored pattern. Both $\boldsymbol{\xi}$ and $-\boldsymbol{\xi}$ are stable patterns.

we have M patterns ($\boldsymbol{\xi}^\mu, \mu = 1, ..., M$), then we use eqn. 5.14 "many times",

$$\omega_{ij} = \frac{1}{N-1}\left(\sum_{\mu=1}^{M} \xi_i^\mu \xi_j^\mu - M\delta_{ij}\right) \tag{5.15}$$

The hope is now that all these patterns are stable and should act as attractors. It is reasonable to assume that we cannot store an unlimited number of patterns. In fact one can figure out this limit for random and uncorrelated patterns. It turns out to be $P_{\max} \approx 0.14N$, where N is the number of nodes in the Hopfield model.

5.8.3 The energy function

One of the most important contributions in the work of Hopfield was the energy function E of the Hopfield model. It is defined as,

$$E = -\frac{1}{2}\sum_i \sum_j \omega_{ij} s_i s_j \tag{5.16}$$

The energy function is a function of the state of the model $s_1, s_2, ..., s_N$. A central property of this energy function is that,

$$\Delta E \leq 0$$

when the system is updated according to the update rule (see eqn. 5.11). This means that the stored patterns corresponds to local minimum of E. We can imagine the energy landscape as a "hilly" surface where stored patterns represents local minima, see Figure 5.17.

As a final remark, not all local minima are stored patterns, there exists so called spurious patterns that also stable but do not correspond to any of the trained patterns. The energy landscape is however interesting from another perspective, we can imagine that retrieval of stored memories is just an energy minimization process.

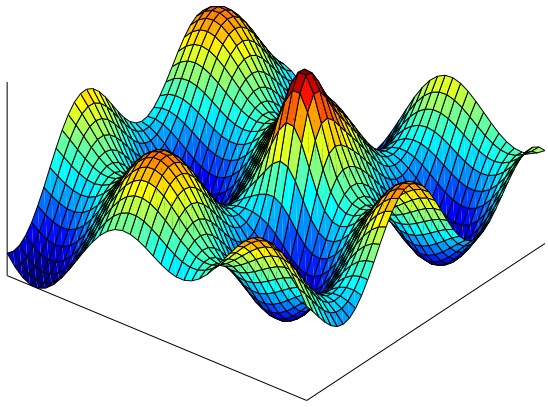


Figure 5.17: The z-axis represents the energy and the x-y plane all corners of the 2^N hyper cube that represents the states of the Hopfield model. A stored pattern often corresponds to a local minimum.

Chapter 6

Self-Organizing Neural Networks

6.1 Introduction

In unsupervised learning there is no teacher! "The purpose of an algorithm for self-organized (or unsupervised) learning is to discover significant features or patterns in the input data and todo the discovery without a teacher".

One often divide the algorithms for unsupervised learning into two subclasses:

- Hebbian learning algorithms. This is basically algorithms that will extract principal components.
- Competitive learning. This is algorithms that are related to various forms of clustering.
  - Competitive networks for vector quantization (clustering)
  - Self-organizing feature maps (SOFM)
  - Learning vector quantization (LVQ)

6.2 Self-organizing networks that do PCA

We have talked about the autoencoder before, and how it can perform a PCA analysis, but also do more complex dimensional reduction. If a simple PCA is the only thing we are after, an autoencoder is not optimal. Even though the encoder projects data onto the space spanned by the first prinicpal components, it can arbitrarily select coordinate system for the projection space, and you need the matching decoder to interprete it.

Much better is to use a powerful matrix manipulation program, such as Matlab, to do PCA. It can also be done with ANNs and, just for completeness, we will present some simple possibilities here.

6.2.1 First Principal Component

We will now look at a 2-layer network with one output using a Hebbian-like update rule at iteration t:

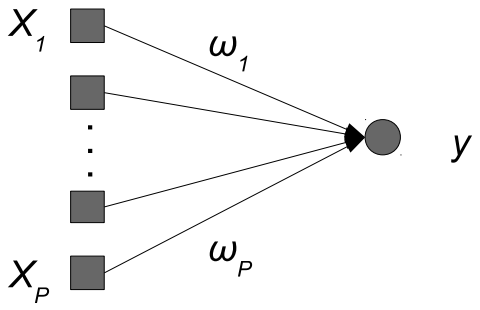


Figure 6.1: Network for extracting the first principal component.

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \eta \frac{1}{N} \sum_n y_n(t) \mathbf{x}_n \tag{6.1}$$

Since $y = \boldsymbol{\omega}^\top \mathbf{x} = \mathbf{x}^\top \boldsymbol{\omega}$ we can use the covariance matrix $\boldsymbol{R}$ from eq. (4.14),to write $\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \eta \boldsymbol{R} \boldsymbol{\omega}(t)$. Here, we assume that our inputs $\mathbf{x}_n$ are alredy normalized to zero mean. If we express $\boldsymbol{\omega}$ in terms of the principal components $\mathbf{u}_k$, defined in eq. (4.18),

$$\boldsymbol{\omega} = \sum_k c_k \mathbf{u}_k, \tag{6.2}$$

we see that the update rule can be written

$$c_k(t+1) = c_k(t) \left[1 + \eta \lambda_k\right], \tag{6.3}$$

where we made use of the fact that each principal component $\mathbf{u}_k$ is an eigenvector to $\boldsymbol{R}$ with eigenvalue λ_k. We do the conventional sorting of vectors so that $\lambda_1 \geq \lambda_2 \geq \ldots$, and see that c_1 grows by a larger factor than any other coefficient c_k. For each update, $\boldsymbol{\omega}$ becomes more and more parallel to $\mathbf{u}_1$. The network finds the first principal component!

However, with this update, all c_k grow unbounded. This can be solved by a brute force normalization in the update

$$\boldsymbol{\omega}(t+1) = \frac{\boldsymbol{\omega}(t) + \eta \frac{1}{N} \sum_n y_n(t) \mathbf{x}_n}{|\boldsymbol{\omega}(t) + \eta \frac{1}{N} \sum_n y_n(t) \mathbf{x}_n|}. \tag{6.4}$$

Another update that is easier to calculate and that also keeps $\boldsymbol{\omega}$ bounded is

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \eta \frac{1}{N} \sum_n \left[y_n(t)\mathbf{x}_n - y_n(t)^2 \boldsymbol{\omega}(t) \right]. \tag{6.5}$$

This is not taken out of the blue: it is what you get if you Taylor expand eq. (6.4) to first order in η, with $|\boldsymbol{\omega}| = 1$. To see how this works, we note that the update for $\boldsymbol{\omega}$ corresponds to an update

$$c_k(t+1) = c_k(t) \left[1 - \eta \frac{1}{N} \sum_n y_n(t)^2 + \eta \lambda_k \right], \tag{6.6}$$

for the coefficients in the expansion eq. (6.2). As long as $1 - \eta \frac{1}{N} \sum_n y_n(t)^2 > 0$, which a small η should ensure, c_1 is modified by the largest factor, and $\boldsymbol{\omega}$ eventually becomes parallel to $\mathbf{u}_1$. Furthermore, since

$$\frac{1}{N} \sum_n y_n(t)^2 = \boldsymbol{\omega}^\top(t) \boldsymbol{R} \boldsymbol{\omega}(t) = \sum_l \lambda_l c_l(t)^2, \tag{6.7}$$

we see that when c_1 dominates, we get the update $c_1(t+1) \approx c_1(t)\left[1 + \eta\lambda_1(1 - c_1(t)^2)\right]$. Thus, $|c_1(t)| \lessgtr 1 \Rightarrow |c_1(t+1)| \gtrless |c_1(t)|$, so $|c_1|$ converges to 1. The weight vector $\boldsymbol{\omega}$ converges to a normalized first prinipal component.

Online updates

Both with the normalized approach eq. (6.4) and the Taylor expanded eq. (6.5), it is possible to use stochastic gradient descent with mini-batches, or even online updates. It corresponds to an approximation of $\boldsymbol{R}$ with $\boldsymbol{R}(t) = \frac{1}{|\mathcal{B}(t)|} \sum_{n \in \mathcal{B}(t)} \mathbf{x}_n \mathbf{x}_n^\top$, where $\mathcal{B}(t)$ is the minibatch selected at iteration t and $|\mathcal{B}|$ is the batch size. For online updates, $\boldsymbol{R}(t)$ is a very crude approximation of $\boldsymbol{R}$, but errors between iterations compensate, and training can still produce a good solution!

6.2.2 Multiple Components

Sanger network

To extract more principal components we can add more output nodes as in fig. (6.2), and update the weights to each output node i using Sanger's rule

$$\boldsymbol{\omega}_i \to \boldsymbol{\omega}_i + \eta \frac{1}{N} \sum_n \left[y_{in}\mathbf{x}_{nk} - y_{in} \sum_{j \le i} y_{jn} \boldsymbol{\omega}_j \right] \tag{6.8}$$

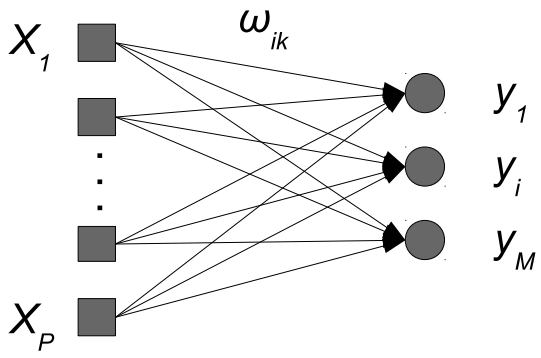


Figure 6.2: Sanger network for extracting the first M principal components.

Here, output node i computes $y_{in}(t) = \boldsymbol{\omega}_i^\top(t)\mathbf{x}_n = \sum_k^P \omega_{ik}(t)x_{nk}$ for pattern n. For node 1, this is just the update in eq. (6.5), so $\boldsymbol{\omega}_1$ will converge to $\mathbf{u}_1$. Inserting $\boldsymbol{\omega}_1 = \mathbf{u}_1$ in updates for the other nodes (which eventually will be a good approximation), and expressing them as $\boldsymbol{\omega}_i = \sum_k c_{ik}\mathbf{u}_k$, we find for the second node

$$\begin{aligned} c_{21} &\rightarrow c_{21}\left[1 - \eta\boldsymbol{\omega}_2^\top \boldsymbol{R}\boldsymbol{\omega}_2\right] \\ c_{2k} &\rightarrow c_{2k}\left[1 - \eta\boldsymbol{\omega}_2^\top \boldsymbol{R}\boldsymbol{\omega}_2 + \eta\lambda_k\right], \quad k > 1 \end{aligned}$$

In other words, c_{22} is modified by the largest factor, so $\boldsymbol{\omega}_2$ converges to $\mathbf{u}_2$. This argument can be iterated to realize that a Sanger network with M output nodes will find the M first principal components.

This network can also be trained with stochastic gradient descent, or even online updating.

There are other networks that find PCA components, for example the APEX network (Adaptive Principal components EXtraction).

6.3 Competitive learning

In competitive learning we have an architecture similar to the PCA networks (see Fig. 6.3). One difference is that we define a "winner" j^* among the C output nodes, usually as

$$j^* = \arg\max_j h(||\mathbf{x} - \boldsymbol{\omega}_j||) \tag{6.9}$$

for some function $h(x)$. A common choice is $h(x) = -x$, meaning that the winning node is simply the one having the weight vector being closest to the input data. Competitive learning is very much used for clustering. Before we present the competitive network, lets describe the standard *k-means* clustering method.

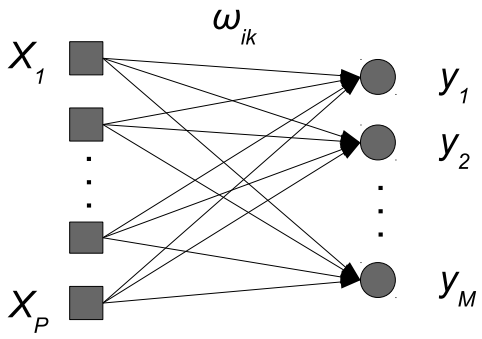


Figure 6.3: The network architecture used for competitive learning.

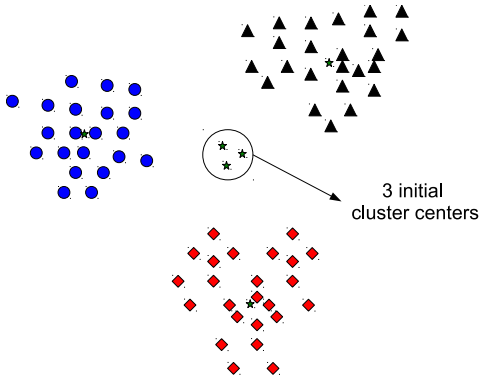


Figure 6.4: Clustering example. Here 3 initial cluster centers are updated according to the k-means procedure, to finally discover the clusters, i.e. move to the center of each cluster.

k-means clustering

1. **Initialization**
   Decide how many clusters to look for (k). Define initial positions for the k clusters (see Fig. 6.4).

   The algorithm now alternates between two steps:

2. **Assignment**
   Each data point is assigned to the nearest cluster. Nearest here often means the Euclidian distance between the data point and the cluster center. If $\mathbf{m}_j^{(n)}$ are the cluster centers at iteration n and $S_j^{(n)}$ is the set of all data points belonging to cluster center j at iteration n. Then we can write the assignment step as:

$$S_i^{(n)} = \left\{ \mathbf{x} : ||\mathbf{x} - \mathbf{m}_i^{(n)}||^2 \langle ||\mathbf{x} - \mathbf{m}_j^{(n)}||^2, j = 1, ..., k \right\}$$

3. **Update**
   Update all cluster centers to be the mean of all data points part of the cluster,

$$\mathbf{m}_j^{(n+1)} = \frac{1}{|S_j^{(n)}|} \sum_{\mathbf{x}_k \in S_j^{(n)}} \mathbf{x}_k$$

The above procedure can be viewed as batch updating. One can also imagine an online procedure where cluster centers are updated after only "presenting" one data point. For this approach to work one need to introduce a learning rate, meaning that a cluster center is moved only a small distance towards the data point. As for neural network training a block update approach is also possible.

6.3.1 Competitive learning for clustering

For the competitive network (see Fig. 6.3) the output nodes are typically set by the following rule,

$$y_j = \begin{cases} 1 & \text{if } j \text{ is the winning node} \\ 0 & \text{otherwise} \end{cases} \tag{6.10}$$

where the winning node is determined according to Eqn. 6.9. Given a training data set $\{\mathbf{x}(1), ..., \mathbf{x}(N)\}$, the training of a competitive network is then accomplished using the following procedure:

1. Determine the size of the network, i.e. the number of output nodes.
2. Initialize all weight vectors $\boldsymbol{\omega}_j$ (cluster centers) randomly, but close to the center of mass of the data set.
3. Randomly pick an input pattern μ from the data set.
   3.1 Find the winning output node j^*.
   3.2 Update the weight vector $\boldsymbol{\omega}_{j^*}$ according to

$$\boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big)$$

4. Repeat step 3 until no more changes occur, including a necessary lowering of the learning rate η.

Again, the above procedure is online learning and can easily be converted into a batch version where all weight updates are averaged over the full data set before doing the actual

weight update. The batch version is identical to k-means clustering if the winning node is determined based on the distance between weight vectors and data points. Once the competitive network has been trained (i.e. the clusters have been determined) it can be used in a test situation where the winning node signals cluster belonging for any new data point.

6.3.2 Learning vector quantization (LVQ)

Since competitive learning is about finding clusters in the data, it is to some extent similar to classification problems. We just have to label the different clusters with appropriate class labels. There is a supervised version of the above competitive learning algorithm, called *Learning Vector Quantization* (LVQ). Here we again have both input data $\{\mathbf{x}_1, ..., \mathbf{x}_N\}$ and target labels $\{\mathbf{d}_1, ..., \mathbf{d}_N\}$. The basic idea of LVQ is to label each of the outputs in the competitive network with a class label and introduce a modified learning rule that depends on whether the class of the winning node is correct or not.

The LVQ learning rule is:

$$\boldsymbol{\omega}_{j^*} \rightarrow \begin{cases} \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to the same class} \\ \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} - \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to different classes} \end{cases}$$

We can summarize the LVQ classification network as follows:

1. Determine the size of the network, i.e. the number of output nodes.
2. Initialize all weight vectors $\boldsymbol{\omega}_j$ (cluster centers) randomly, preferably such that they cover most of the input space.
3. Label all output nodes with the available target class labels. It can be an even distribution or follow the prior class probabilities.
4. Randomly pick an input pattern μ from the data set.

   3.1 Find the winning output node j^*.

   3.2 Update the weight vector $\boldsymbol{\omega}_{j^*}$ according to

$$\boldsymbol{\omega}_{j^*} \rightarrow \begin{cases} \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ “agree”.} \\ \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} - \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ “disagree”.} \end{cases}$$

5. Repeat step 3 until no more changes occur, including a necessary lowering of the learning rate η.

There are modified learning rules for LVQ, called LVQ2.1 and LVQ3, but the underlying idea is as presented above. Two examples of LVQ classification can be seen in Fig. 6.5.

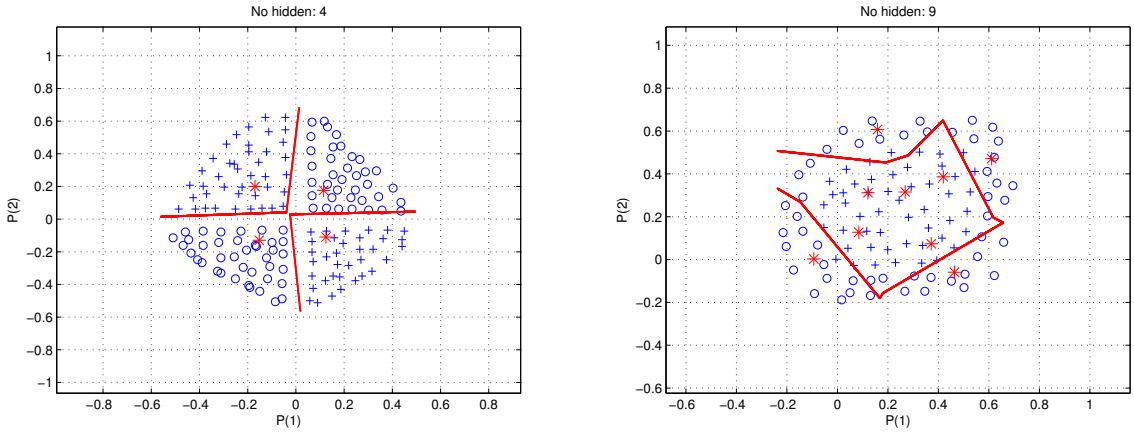


Figure 6.5: Examples of decision boundaries produced by the LVQ network. (Left) The red "*" are the cluster centers found by the LVQ algorithm. Two of then belong to the "+" class and two of the to the "ring" class. (Right) This problem requires many more output nodes, if we compare to an MLP. Again, the red "*" marks the found cluster centers by the LVQ algorithm.

6.4 Self-organizing feature maps

From Haykin: "The principal goal of the self-organizing feature mapping algorithm developed by Kohonen (1982) is to transform an incoming signal pattern of arbitrary dimension into a one- or two-dimensional discrete map, and to perform this transformation adaptively in a topological ordered fashion."

The output nodes are placed in a 1- or 2-dimensional grid where each node has a predefined number of neighbors (see Fig. 6.6). Each of the inputs are connected to each of the output nodes, the same principle for all other feed-forward architectures we have used. Figure 6.7 shows a self-organizing feature map (SOFM) for a 2-dimensional square output grid.

An important difference between competitive networks and SOFM networks is the *neighborhood* function that is used by the SOFM. It is important in order to have a topology preserving map. The neighborhood function Λ_{jj^*} uses the distance between output node j and j^* to derive its output. As for the competitive network a winning node is found among all output nodes, typically the node having the smallest Euclidian distance to the input

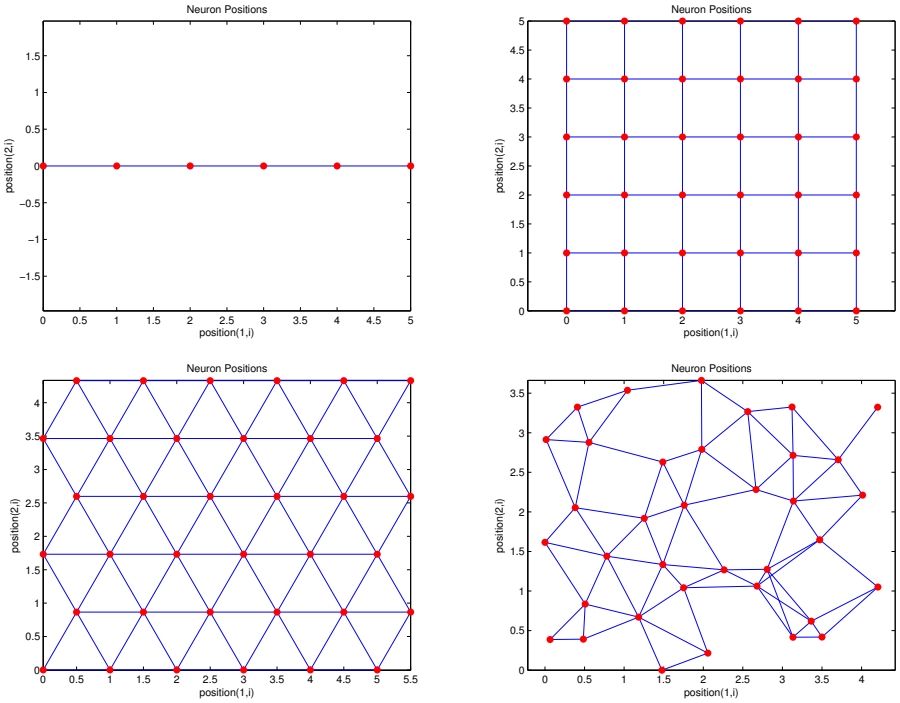


Figure 6.6: Examples of output grids for the self-organizing feature map. A 1-dimensional grid where each node have 2 neighbors, one 2-dimensional square grid where each node has 4 neighbors, a hexagonal grid where each node has 6 neighbors and finally a random grid. (All figures produced in Matlab).

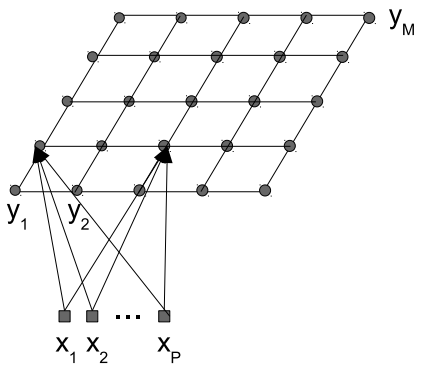


Figure 6.7: An example of a SOFM where a P dimensional input $\mathbf{x}$ is mapped onto a 2-dimensional square grid.

vector (see Eqn. 6.9. The update equation for the SOFM is,

$$\boldsymbol{\omega}_j \rightarrow \boldsymbol{\omega}_j + \eta \Lambda_{jj^*} \big(\mathbf{x}_\mu - \boldsymbol{\omega}_j \big) \tag{6.11}$$

where j^* is the winning node for $\mathbf{x}_\mu$. Note that for the SOFM all weights are updated, not just the winning node. It is the neighborhood function that determines how many (or how mush) of the neighboring nodes that are updated. Examples of neighborhood functions are:

$$(1) \quad \Lambda_{jj'} = \begin{cases} 1 & \text{if } d(j,j') \leq d_o \\ 0 & \text{otherwise} \end{cases} \tag{6.12}$$

$$(2) \quad \Lambda_{jj'} = e^{-d(j,j')^2/2\sigma^2} \tag{6.13}$$

where $d(j,j')$ is the distance between node j and j' on the grid. Again there are several possible choices for $d(j,j')$, such as Euclidian distance or link distance, where the latter is just the smallest number of links connecting j and j' on the grid. We can summarize the SOFM as,

1. Choose the number of output nodes and the layout (1- or 2-dimensional and grid).
2. Initialize all weight vectors $\boldsymbol{\omega}_j$, typically randomly around center-of-mass for the input data.
3. Repeat until all $\boldsymbol{\omega}_j$ have converged:
   - 3.1 Randomly pick an input pattern μ from the data set.
   - 3.2 Find the winning output node j^*.
   - 3.3 Update the weight vector $\boldsymbol{\omega}_{j^*}$ according to
     $$\boldsymbol{\omega}_j \rightarrow \boldsymbol{\omega}_j + \eta \Lambda_{jj^*} \big(\mathbf{x}_\mu - \boldsymbol{\omega}_j \big)$$
   - 3.4 It is often necessary to dynamically change d_o/σ and η to find good "solutions" and ensure convergence.

6.4.1 Properties of the SOFM

Haykin (chapter 9.4) lists an number of properties of the SOFM:

1 **Approximation of the input space**
The feature map Φ, represented by the set of synaptic weight vectors $\{\boldsymbol{\omega}_j\}$ in the output space $\mathcal{A}$, provides a good approximation of input space $\mathcal{X}$.

2 **Topological ordering**
The feature map Φ computed by the SOFM algorithm is topologically ordered in the sense that the spatial location of a neuron in the lattice (grid) corresponds to a particular domain or feature of the input patterns.

3 **Density matching**
The feature map Φ, reflects variations in the statistics of the input distribution: Regions in the input space $\mathcal{X}$ from which sample vectors $\mathbf{x}$ are drawn with a high probability of occurrence are mapped onto larger domains of the output space, and therefore with better resolution than regions in $\mathcal{X}$ from which sample vectors $\mathbf{x}$ are drawn with a low probability of occurrence.

4 **Feature selection**
Given data from an input space, the self-organizing map is able to select a set of best features for approximating the underlying distribution.

The following numerical examples will provide some insight into these properties.

6.4.2 Some SOFM examples

For all examples presented in the figures below (6.8-6.8) the input data is 2-dimensional (continuous) and the output grid of the SOFM is either 1-D or 2-D. The weight vectors (after training) are shown as grey dots and there is a red line between nearest neighbors.

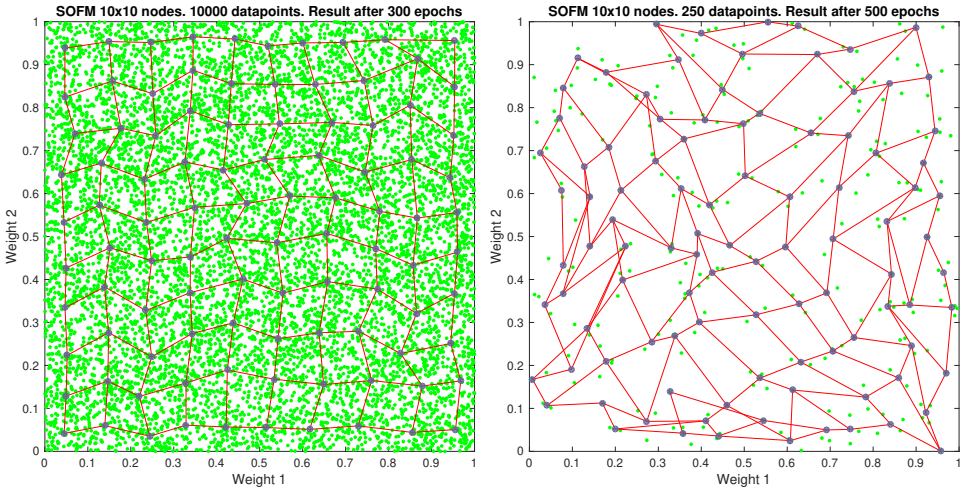


Figure 6.8: (left) Approximation of unit square to a 10x10 square grid. (right) Unit square now sampled with 250 data points, again 10x10 square grid.

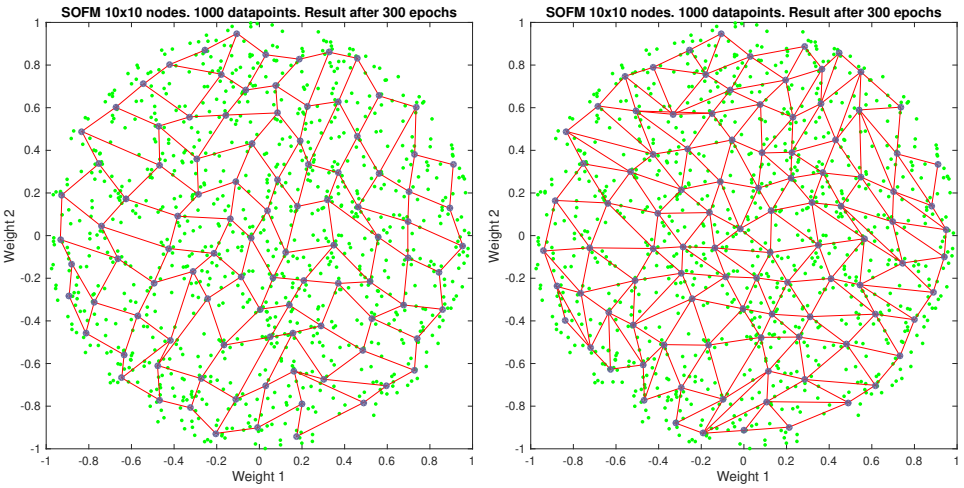


Figure 6.9: (left) Approximation of unit circle to a 10x10 square grid. (right) Approximation of unit circle to a 10x10 hexagonal grid. The unit circle is sampled with 1000 data points.

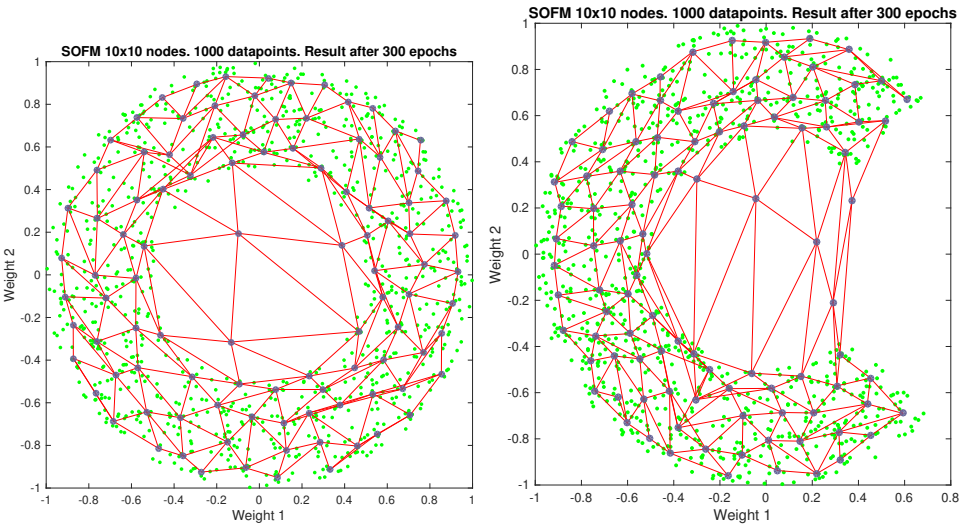


Figure 6.10: Parts of the unit circle approximated by a 10x10 hexagonal grid.

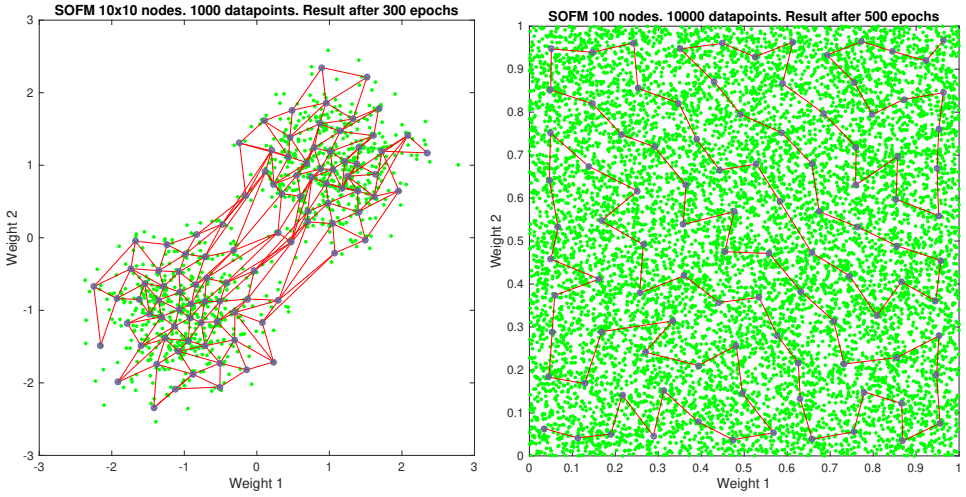


Figure 6.11: (left) Two normal distributed data sets approximated by a 10x10 hexagonal grid. (right) Unit square approximated by 100 nodes in a 1-dimensional SOFM.

6.5 Summary of self-organizing networks

PCA
Expand inputs

$$\mathbf{x}_n = \sum_j a_{nj}\mathbf{u}_j \qquad \text{with} \qquad \boldsymbol{R}\mathbf{u}_j = \lambda_j \mathbf{u}_j$$

dimensionality reduction:

$$(x_{n1}, x_{n2}, ..., x_{nP}) \curvearrowright (y_{n1}, y_{n2}, ..., y_{nM})$$

with $M << P$.

Sanger network for extracting PCA eigenvectors

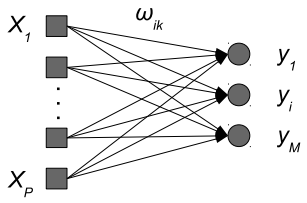


Figure 6.12: Sanger network.

$$y_i(n) = \sum_k^P \omega_{ik} x_{nk}$$

and (online updating)

$$\omega_{ik} \to \omega_{ik} + \eta \left[y_i(\mathbf{x}_n) x_{nk} - y_i(\mathbf{x}_n) \sum_j^i \omega_{jk} y_j(\mathbf{x}_n) \right] \qquad i = 1, .., m$$

Autoencoder (auto-association)

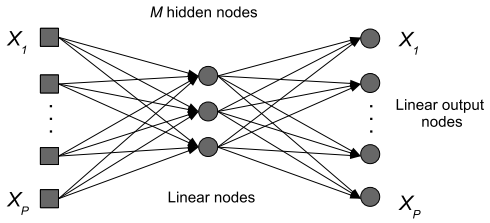


Figure 6.13: Autoencoder network.

$$E(\boldsymbol{\omega}) = \frac{1}{N}\sum_{n}^{N}\sum_{i}^{M}\left(x_i'(\mathbf{x}_n, \boldsymbol{\omega}) - x_{ni}\right)^2$$

$\{\boldsymbol{\omega}_j\}$ span the same space as the first principal components.

Competitive network for clustering

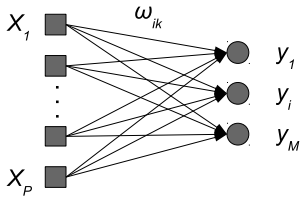


Figure 6.14: Competitive network.

Winning node:

$$j^* = \arg\max_j h(||\mathbf{x} - \boldsymbol{\omega}_j||) \tag{6.14}$$

Output value:

$$y_j = \begin{cases} 1 & \text{if } j \text{ is the winning node} \\ 0 & \text{otherwise} \end{cases} \tag{6.15}$$

Update rule:

$$\boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\left(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\right)$$

LVQ
The LVQ learning rule:

$$\boldsymbol{\omega}_{j^*} \rightarrow \begin{cases} \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to the same class} \\ \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} - \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to different classes} \end{cases}$$

SOFM

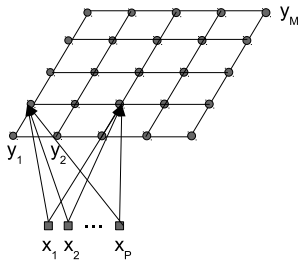


Figure 6.15: SOFM.

Winning node:

$$j^* = \arg\max_j h(||\mathbf{x} - \boldsymbol{\omega}_j||) \tag{6.16}$$

Output value:

$$y_j = \begin{cases} 1 & \text{if } j \text{ is the winning node} \\ 0 & \text{otherwise} \end{cases} \tag{6.17}$$

Update rule:

$$\boldsymbol{\omega}_j \rightarrow \boldsymbol{\omega}_j + \eta\Lambda_{jj^*}\big(\mathbf{x}_\mu - \boldsymbol{\omega}_j\big)$$

Example of Λ_{jj^*}:

$$\Lambda_{jj'} = e^{-d(j,j')^2/2\sigma^2}$$