1 Representing Information

1.1 Elements of Representation

- features
- mappings
- dynamical systems
- gaussian processes
- graphical models

1.2 Feature Space Expansion

High-dimensional feature spaces can simplify computations, e.g. enable linear separability of class regions.

Standard methods for feature space expansion:

- **polynomial expansion**

  $$(x_1..x_d) \rightarrow (x_1..x_d, .., x_i x_j .., x_i x_j x_k)$$

  $(\frac{d}{k})$ k-multinomials, can increase dimensionality up to 2^d.

- **time history:** $\boldsymbol{x}_t \in \mathbb{R}$ is transformed into

  $$(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}..\boldsymbol{x}_{t-k}) \in \mathbb{R}^{d\cdot(k+1)}$$

- **filters** provide a similar effect by generating features through convolution with filter kernels:

  $$x_i(t) = \int K_i(t, t')x(t')dt'$$

  where $K_i(t, t')$ is a suitable set of filter kernels.

1.3 Kernel Trick

Provides non-linear and high-dimensional features $\phi_k(\boldsymbol{x})$ for linear regressors or perceptrons. With features $\boldsymbol{x} \in X$, a linear classifier can always be written as a superposition of scalar products

$$\bar{y}(\boldsymbol{x}) = \sum_{k=1}^{D} \bar{w}_k \bar{\boldsymbol{x}}_k \cdot \boldsymbol{x} \tag{1.1}$$

Mapping $\boldsymbol{x} \in \mathbb{R}^d$ into features $\boldsymbol{\phi}(\boldsymbol{x}) \in V$ in some higher-dimensional feature space and performing the derivation of the classifier directly in this space leads to

$$y(\boldsymbol{x}) = y\left(\boldsymbol{\phi}(\boldsymbol{x})\right) = \sum_{k=1}^{D} w_k \boldsymbol{\phi}(\boldsymbol{x}_k) \cdot \boldsymbol{\phi}(\boldsymbol{x}) \tag{1.2}$$

which is analogous to Eq. (1.1) (with possibly different weights and "support points", indicated by omitting the bar).

This can be written as

$$y(\boldsymbol{x}) = y\left(\boldsymbol{\phi}(\boldsymbol{x})\right) = \sum_{k=1}^{D} w_k K(\boldsymbol{x}_k, \boldsymbol{x}) \tag{1.3}$$

where we have written the scalar product as

$$\boldsymbol{\phi}(\boldsymbol{u}) \cdot \boldsymbol{\phi}(\boldsymbol{v}) = K(\boldsymbol{u}, \boldsymbol{v}) \tag{1.4}$$

This can be reversed: whenever we have a "kernel" $K(\boldsymbol{u}, \boldsymbol{v})$ for which we know that a representation (1.4) exists, we can write the "linear" classifier directly in the form (1.2), even if we don't know the functions $\phi(\boldsymbol{x})$ explicitly.

Such substitution can be made always. However, it becomes very useful if also the computation of to $\boldsymbol{w}_i$ and the $\boldsymbol{x}_i$ required for Eq. (1.3) can be entirely formulated in terms of scalar products $\boldsymbol{\phi}(\boldsymbol{u}) \cdot \boldsymbol{\phi}(\boldsymbol{v})$ (which includes the case for the support vector machine). Then, these scalar products can be replaced by kernels $K(\boldsymbol{u}, \boldsymbol{v})$ as well, *making any explicit access to the transformed features $\boldsymbol{\phi}(\boldsymbol{x})$ completely unnecessary*: this is the **kernel trick**. It allows to benefit from the linearization of the non-linear equation (1.3) through feature-space expansion *without the need of explicitly computing the linearizing transformation $\boldsymbol{x} \mapsto \boldsymbol{\phi}(\cdot)$ explicitly*.

The existence of a representation (1.4) is ensured by

Mercer's condition:

$K(x, y)$ is symmetric and for any square-integrable function $\phi(x)$

$$\int \phi(x) K(x, y) \phi(y) dx dy \geq 0$$

Frequently used kernels that fulfil Mercer's condition include:

- Gaussian kernel $K(\boldsymbol{x}, \boldsymbol{y}) = \exp(-\lambda(\boldsymbol{x} - \boldsymbol{y})^2)$
- polynomial kernel $K(\boldsymbol{x}, \boldsymbol{y}) = (1 + \boldsymbol{x} \cdot \boldsymbol{y})^\lambda$

It may (and usually does) happen that the dimensionality D of the generated, implicit feature space is not finite, which, however, does not prevent the kernel trick from working well if Mercer's condition is met.

The above is completely analogous to the "spectral expansion" of a positive symmetric matrix $\boldsymbol{K}$ with matrix elements K_{uv}, using its eigenvectors $\boldsymbol{e}_k$ and eigenvalues λ_k:

$$K_{uv} = \sum_{k=1}^{D} \lambda_k (\boldsymbol{e}_k)_u (\boldsymbol{e}_k)_v = \sum_{k=1}^{D} (\lambda_k^{1/2} \boldsymbol{e}_k)(u)(\boldsymbol{\lambda}^{1/2} \boldsymbol{e}_k)(v) = \boldsymbol{\phi}(u) \cdot \boldsymbol{\phi}(v)$$

i.e., the values $\sqrt{\lambda_k}(\boldsymbol{e}_k)(u)$ of the the scaled eigenvectors at a fixed index position u are seen to take an analogous role as the $\phi_k(\boldsymbol{u})$ in equation (1.4).

Numerous articles and books exist on the subject. For a compact introduction to more details, see eg. [MüllerEtAl01].

1.4 Dynamical Systems and Filters

Temporal filters can be seen as *dynamical systems* which compute at each time step a "state" s_t that is some function of the state(s) s_{t-k}, the current input x_t and the input(s) x_{t-l} at the previous time step(s):

$$s_t = F(s_{t-1}, s_{t-2}, \ldots s_{t-k}, x_t, x_{t-1}, x_{t-2}, \ldots, x_{t-l})$$

In the simplest case, the function F is linear in its arguments, leading to the well-known recursive filters

$$s_t = \sum_{i=1}^{K} a_i \cdot s_{t-i} + \sum_{j=0}^{L} b_j \cdot x_{t-j}$$

that allow, e.g., to selectively damp/enhance specifiable frequency bands of the input time sequence. The very simplest example is a smoothing filter

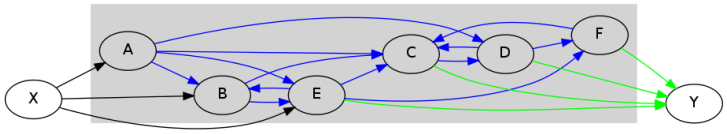
$$s_t = (1 - \gamma) \cdot s_{t-1} + \gamma \cdot x_t$$

which computes a time-locally weighted average of the time series x_t. In all cases, s_t and x_t can be multidimensional vectors and the coefficients suitable matrices to allow for more complex processing.

Yet, combining linear filters always leads back to a linear filter. Richer processing can only occur when non-linearities are included. In the following, we briefly consider nonlinear filters arising from recurrent neural networks that have led to an approach called

1.5 Reservoir Computing

The generation of non-linear dynamic features with recurrent networks has been an essential idea of the so-called "reservoir networks" or "echo state networks" that have enabled a major break-through in learning with recurrent networks: a fixed recurrent network provides a "dynamic feature generator" (a "dynamic reservoir") in the form of a dynamical system whose features $\boldsymbol{v}(t)$ can then be mapped into the output $\boldsymbol{y}$ by a trainable read-out stage



echo state network: blue connections are fixed and create recurrent dynamics of reservoir (nodes in grey area). Trainable readout connections are green, input connections black.

Formally,

$$v_i(t) = (1-\lambda)v_i(t-1) + \lambda \tanh\left(\sum_{j=1}^{D} a_{ij}v_i(t-1) + \sum_{k=1}^{d} b_{ik}x_k\right)$$

$$y_i(t) = \sum_{j=1}^{D} w_j v_i(t)$$

Here, $\lambda > 0$ determines the time scale of the dynamics and a_{ij} and b_{ik} are fixed (e.g., random). Since the output weights w_j occur only linearly, they can be computed by a variety of algorithms, such as on-line perceptron training or linear least squares etc.

Echo property. A major issue is the choice of the fixed weights to make the reservoir generate good features. The reservoir should obey the so called

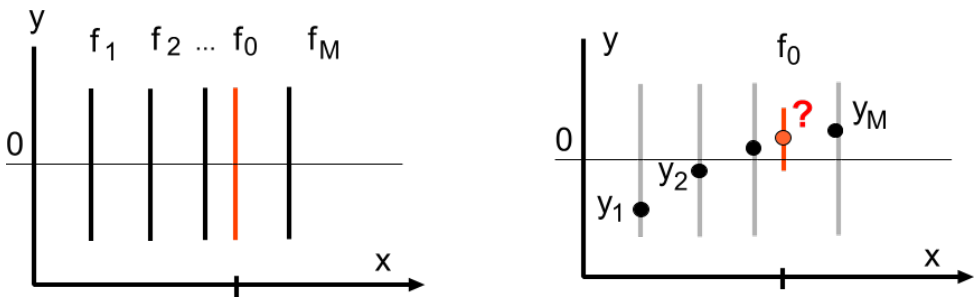
"echo property": this requires that without external excitation (all $x_k = 0$) all activities of the reservoir will decay slowly to zero. A criterion for this is that the *spectral radius* (the largest eigenvalue of $\boldsymbol{A}^T \boldsymbol{A}$) is less than 1.

However, even when the echo property is fulfilled, the activities may decay too fast or too slowly. A major subject of echo state network research is to tune the reservoir such that its dynamics generates sufficiently "rich" features that match the time scales of the application task.

For more details and examples, see e.g. the survey article [LukJaeg09].

1.6 Gaussian Processes

generalize deterministic mappings:[1]



Intuitive idea: create distribution of functions $f(x)$ such that any finite tuple of values $(f(x_1), f(x_2)..f(x_M))$ is distributed according to a M-variate Gaussian distribution. Formally,

$$(f(x_1), f(x_2)..f(x_M)) \sim N\left(0, \boldsymbol{\Sigma}(x_1..x_M)\right)$$

Here, $\boldsymbol{\Sigma}(x_1..x_M)$ is a $M \times m$ correlation matrix that depends on the chosen point tuple $(x_1, ..x_M)$. (for simplicity, all expectation values are chosen as zero)

Example: Let $w_i, i = 1..K$ be normally distributedrandom variables with zero mean[2] and covariance matrix $\boldsymbol{\sigma}$, and $g_i(x), i = 1..K$ arbitrary functions. Then

$$f(x) = \sum_{i=1}^{K} w_i g_i(x)$$

is a gaussian process: indeed, considering a tuple $(f(x_1), f(x_2)..f(x_M))$ we see that each element is a superposition of gaussian random variables, making the entire tuple a sample of a gaussian distribution.

[1] the following section follows closely [RasWil06], Chap. 2.2

[2] this is not essential

Consistency condition: $\mathbf{\Sigma}(x, x')$ has to be chosen such that for any sub-tuple the corresponding correlation matrix coincides with the corresponding submatrix of the matrix of the larger tuple.

Correlation function: Any Gaussian distribution is fully specified by its mean and its pair correlation function. Therefore, with mean fixed at 0, the Gaussian process is fully specified when

$$\langle f(x) f(x') \rangle = K(x, x')$$

A well-known relationship for Gaussian distributions is that the "kernel" $K(x, x')$ is the correlation matrix (note that conceptually the arguments x, x' act like vector indices in a very high (actually infinite) dimensional vector space, $\boldsymbol{K}$ is a matrix in this space, and $(\boldsymbol{K}^{-}1)_{x,x'}$ is the xx' element of the inverse matrix).

For the above example, we obtain

$$\mu(x) = \rangle f(x) \rangle = \pm \sum_{i=1}^{K} \langle w_i \rangle g_i(x) = 0$$

$$K(x, x') = \langle f(x) f(x') \rangle = \sum_{i=1}^{K} \sum_{j=1}^{K} \langle w_i w_j \rangle g_i(x) g_j(x) = \sum_{i=1}^{K} \sum_{j=1}^{K} \Sigma_{ij} g_i(x) g_j(x)$$

To prevent values at neighboring points $f(x), f(x'), ||x - x'||$ small, from being very different, they should be highly correlated. This can be achieved by choosing the correlation function $K(x, x')$ to be "peaked" at $x = x'$, e.g. as

$$K(x, x') = \exp\left(-\frac{1}{2} D(x, x') \right)$$

where $D(x, x')$ is some suitable distance function. A straightforward choice is the scaled euclidean square distance $(x - x')/L^2$ with a scaling parameter L specifying a characteristic distance beyond which correlations get weak, but it is not required that $K(x, x')$ itself has the form of a Gaussian: more complex prescriptions for the structure of the desired correlations are possible, as long as $K()$ remains a symmetric and positive definite kernel. This makes the method rather flexible and allows its adaptation to a variety of situations, e.g., use it also for non-vectorial input data (such as symbol strings), as long as one can define a suitable kernel for these.

In the following, we simplify the notation and abbreviate $(f(x_1), f(x_2)..f(x_M))$ by $(f_1, f_2, \ldots f_M)$.

Gaussian Progress Regression. The underlying idea is captured in the following three steps:

- given M "training points" $(x_1..x_M)$, we know the distribution of the associated values $(f(x_1), f(x_2)..f(x_M))$ of the Gaussian Process is Gaussian with mean zero and known covariance.
- we embed an additional M+1-th "query point" $x = x_0$: this again leads to a Gaussian distribution, now of the M+1-tuple $\boldsymbol{f} = (f_0, f_1..f_M)$. Let us denote this distribution by $P(f_0, f_1..f_M)$.
- we finally force the random values $f_1..f_M$ to coincide with the training data $y_1..y_M$ and ask how this constrains the distribution $P(f_0) = P(f_0, y_1..y_M)$ at the query location.

The expectation value $\langle f_0 P(f_0) \rangle$ is the output of the Gaussian Process regression.

The computational steps:

$$P(f_0, f_1..f_M) = C \cdot \exp\left(-\frac{1}{2}\boldsymbol{f}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{f}\right)$$

To make the f_0 dependency explicit, we split the inverse correlation matrix $\boldsymbol{A} = \boldsymbol{\Sigma}^{-1}$ into blocks:

$$\boldsymbol{A} = \begin{pmatrix} a & \boldsymbol{w}^T \\ \boldsymbol{w} & \boldsymbol{Q} \end{pmatrix}$$

This and setting $f_i = y_i$ for $i > 0$ leads to

$$\begin{aligned} P(f_0, y_1..y_M) &= C \cdot \exp\left(-\frac{1}{2}(f_0^2 \cdot A_{00} + f_0 \sum_{i=1}^{M} A_{0i} y_i + \sum_{i=1}^{M} y_i A_{i0} f_0 + \sum_{i,j=1}^{M} y_i A_{ij} y_j)\right) \\ &= C \cdot \exp\left(-\frac{1}{2}(a \cdot f_0^2 + 2 f_0 \boldsymbol{w} \cdot \boldsymbol{y} + \quad \text{terms with } i,j > 0)\right) \\ &= C \cdot \exp\left(-\frac{a}{2}(f_0^2 + 2 f_0 \boldsymbol{w} \cdot \boldsymbol{y}/a) + \quad \text{terms with } i,j > 0\right) \\ &= C' * \exp\left(-\frac{a}{2}(f_0 + \boldsymbol{w} \cdot \boldsymbol{y}/a)^2\right) \end{aligned}$$

where have only kept terms containing f_0 in the exponent and all other terms moved into a changed constant C'.

From this we read off that the value f_0 at the query point follows a gaussian distribution with mean

$$y^*(x) = -\boldsymbol{w} \cdot \boldsymbol{y}/a$$

and variance $1/a$.

To get explicit values for $\boldsymbol{w}$ and a, we need to express the elements of $\boldsymbol{A} = \boldsymbol{\Sigma}^{-1}$ in terms known elements of $\boldsymbol{\Sigma}$. Writing

$$\boldsymbol{\Sigma} = \begin{pmatrix} s & \boldsymbol{k}^T \\ \boldsymbol{k} & \boldsymbol{K} \end{pmatrix}$$

we know that $s = \langle f_0 f_0 \rangle = K(x_0, x_0)$, $k_i = \langle f_0 f_i \rangle = K(x_0, x_i)$ and $K_{ij} = \langle f_i f_j \rangle = K(x_i, x_j)$. We must have

$$\begin{aligned} \mathbf{1} = \boldsymbol{\Sigma A} &= \begin{pmatrix} s & \boldsymbol{k}^T \\ \boldsymbol{k} & \boldsymbol{K} \end{pmatrix} \begin{pmatrix} a & \boldsymbol{w}^T \\ \boldsymbol{w} & \boldsymbol{Q} \end{pmatrix} \\ &= \begin{pmatrix} sa + \boldsymbol{w} \cdot \boldsymbol{k} & s\boldsymbol{w}^T + \boldsymbol{k}^T \boldsymbol{Q} \\ a\boldsymbol{k} + \boldsymbol{K}\boldsymbol{w} & \boldsymbol{k}\boldsymbol{w}^T + \boldsymbol{K}\boldsymbol{Q} \end{pmatrix} \end{aligned}$$

This results in four equations that can be solved easily. For computing $\boldsymbol{w}$ and a it suffices to consider the top-left and bottom-left elements in the matrix, yielding

$$\begin{aligned} \boldsymbol{w} &= -a\boldsymbol{K}^{-1}\boldsymbol{k} \\ 1 &= sa - a\boldsymbol{k}^T\boldsymbol{K}^{-1}\boldsymbol{k} \Rightarrow a = \frac{1}{s - \boldsymbol{k}^T\boldsymbol{K}^{-1}\boldsymbol{k}} \end{aligned}$$

Thus

$$y^*(x) = \boldsymbol{k}^T\boldsymbol{K}^{-1}\boldsymbol{y} \quad \text{with variance } a^{-1} = s - \boldsymbol{k}^T\boldsymbol{K}^{-1}\boldsymbol{k} \tag{1.5}$$

If the training data are assumed to be noisy with variance σ^2, a similar reasoning leads to the result

$$y^*(x) = \boldsymbol{k}^T(\boldsymbol{K} + \sigma^2)^{-1}\boldsymbol{y} \quad \text{with variance } s - \boldsymbol{k}^T(\boldsymbol{K} + \sigma^2)^{-1}\boldsymbol{k}$$

i.e., $\boldsymbol{K}$ is simply replaced by $\boldsymbol{K} + \sigma^2$.

These equations express the Gaussian regressor output in terms of matrices whose dimensionality is the number M of training points. Requiring their inversion, the method has computational complexity $O(M^3)$.

Main input to the algorithm are the given training pairs (x_i, y_i), along with a symmetric, positive definite kernel $K(\cdot, \cdot)$ that maps input space point pairs (x_i, y_i) into correlation strengths $\boldsymbol{K}_{ij} = K(x_i, x_j) = \langle y_i, y_j \rangle$ for corresponding pairs of output values. Since at no point we have made use of the fact that x_i are scalars, everything works in the same manner *for arbitrary objects* x_i (including vectors, strings, lists, trees etc.) as long as a suitable kernel $K()$ can be defined for these[3]

Eq. (1.5) then has a simple and straightforward interpretation: the output value $y^*(x)$ is a weighted sum of the given output values $y_1..y_M$ with weighting coefficients given by the correlations $k_i = K(x, x_i)$ for the query location x

[3] However, there is no such freedom on the output side, since the very concept of a Gaussian distribution is based on real numbered values (with straighforward generalization to real numbered vectors)

with the the training locations x_i, and an extra linear rescaling by the inverse of the correlation matrix $\boldsymbol{K}$. This latter scaling accounts for any statistical dependencies between the outputs and makes the final result independent of the overall scaling of $K()$.

More details can be found in [RasWil06], which is the standard reference on the subject.

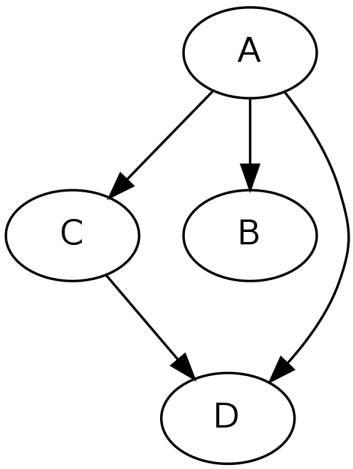
1.7 Graphical Models

Not all probability densities can be well described by Gaussians. Graphical models offer a different way of working with structured PDFs such that computational simplifications become possible:

- A general PDF with k n-ary variables requires $n^k - 1$ parameters for its complete specification.
- Graphical models offer a scheme to describe structured PDFs that require fewer parameters for the same number of variables.
- the scheme is based on a graph expressing dependencies among variables leading to a factorization of the PDF into lower-parameter factors

The graph is formed by representing each variable of the PDF as a node receiving arrows from those other variables from which it is "directly affected".

Example:



Example of a graphical model for four random variables A,B,C,D

$$P(A, B, C, D) = P(A)P(B|A)P(C|A)P(D|A, C)$$

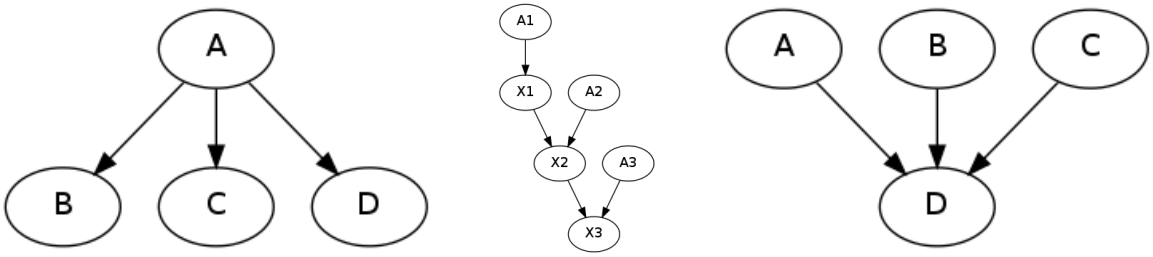
While a general PDF with four n-ary variables would require $n^4 - 1$ parameters, a PDF of the dependency structure expressed in the above model requires only $n + 2n^2 + n^3$ parameters (which is significantly less, e.g. 63 vs. 18 in the binary case k=2)

General approach: Let G be a acyclic directed graph with nodes labelled $x_1, ..x_N$ and let U_i be the subset of variables whose nodes are connected by a directed link to the node labelled x_i. Then we associate with G the PDF

$$P(x_1, ..x_N) = \prod_{i=1}^{N} P(x_i|U_i)$$

A graph specifying a PDF in this manner is also called a *Bayes Net*.

Examples:



Left: "Single Cause:" $P(A)P(B|A)P(C|A)P(D|A)$ Middle: Chain: $P(A1)P(X1|A1)P(X2|A2, X1)P(X3|A3, X2)$, Right: "Multiple Causes:" $P(A)P(B)P(C)P(D|A, B, C)$

Usage: As a means to concisely specify structured PDFs in terms of lower-parameter factors, Bayes Nets help to simplify the following basic learning tasks (in ascending order of computational complexity):

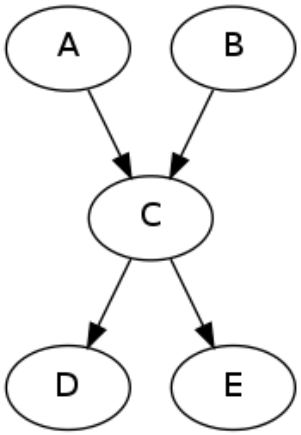
inference: given values for some nodes in the graph, what is the PDF of the remaining nodes?

parameter learning: in this case, the factor PDFs are parametrized and the task is to find optimal parameter values, given some data.

model selection: here, one tries parameter learning for a number of competing graph structures and chooses the model that gives the maximal likelihood (minus some complexity penalty to avoid overfitting)

model inference: here, one wishes to infer the graphical model structure

from the given data. To be feasable, this usually requires additional constraints, with model selection as a maximal simplification.



In the following, we only consider an example of **inference**. Consider a pdf structure as depicted in the left Figure.
What is the probability of E, given B and D? The first step is to use Bayes Rule to express the desired conditional pdf $P(E|B,D)$ in terms of the complete pdf $P(A,B,C,D,E)$:

$$\begin{aligned} P(E|B,D) &= \sum_{A,C} P(A,C,E|B,D) \\ &= \sum_{A,C} \frac{P(A,B,C,D,E)}{P(B,D)} \\ &= \sum_{A,C} \frac{P(A,B,C,D,E)}{\sum_{A,C,E} P(A,B,C,D,E)} \end{aligned}$$

The second step uses the Bayes Net to substitute the full pdf $P(A,B,C,D,E)$ by a simpler expression:

$$P(A,B,C,D,E) = P(A)P(B)P(C|A,B)P(D|C)P(E|C)$$

This allows to carry out the remaining summations in terms of "low-dimensional" factors (the detailed computation is not very interesting and omitted here).

This direct approach is only feasable for small graphs and/or state spaces. For larger problems there exist methods to use the graph for computational approximations, this is, however, beyond the scope of this lecture.

1.8 References

[MüllerEtAl01] Klaus-Robert Müller et al. (2001) *An Introduction to Kernel-Based Learning Algorithms.* IEEE Trans. Neural Networks 12(2) pp. 181ff

[LukJaeg09] Lukoševičius M., Jaeger H. (2009) *Reservoir computing approaches to recurrent neural network training* Computer Science Review Vol.3(3) pp.127-149

[RasWil06] Rasmussen C.E. and Williams C.K.I. Gaussian Processes for Machine Learning. MIT Press 2006