Machine Intelligence Summary

Stephan Gabler

I wrote this summary to summarize all the content for myself and I wanted to describe all the ideas in complete sentences, because this was what I missed most in Obermayers script. It should be a more intuitive approach to all the techniques he introduces. If you find any mistakes or have suggestions for changes, please don't hesitate to write me: stephan.gabler@gmail.com

MI 1

Inductive Learning

Inductive learning is the key thing we do all the time in modern machine learning. The idea is to come up with a general rule (hypotheses) via extrapolation of examples. It is learning a generalization from data instead of programming of rules. The model selection step we always do in machine learning is done via inductive learning, because we try to approximate the generalization error by extracting the essence/underlying structure of the data. A few keywords:

- data driven, adaptive system
- learning and self organization instead of deduction and programming
- learning by example
- induce a general rule from a set of observed instances

Basic properties of Artificial Neural Network (ANN) architectures

Artificial Neural Networks are inspired by the connectionists idea that complex phenomena can be described by interconnected networks of simple and often uniform units. A major property of such a network is that the data are stored and processed by rules which are not explicitly stated as in a formal deduction system (e.g. prolog). There is no separation between data and algorithm and the computation is solely done by changing the weights of connections between the simple units. This idea gives a powerful computational framework with a distributed representation of information, with highly parallelized processing, which is robust and consists only of very simple elements. The negative aspects of ANNs are the low precision and the implicit and distributed representation of the learned algorithm. It is often difficult to analyse the learned solution.

Three main learning paradigms

Usually people distinguish between three major paradigms in machine learning, which are discriminated by the type of the training data and the feedback which is given to the learning algorithm. However it is a phenomenological characterization as the same or similar mathematical methods are used for all three concepts. For example the principle of empirical risk minimization (approximation of a distribution) and gradient descent are used for all kind of learning problems. Gradient descent e.g. for MLP learning or ICA

supervised

- learning generalization from examples
- classification/regression
- training data is given in form of tuples containing the input AND the desired output
- can be modelled by a conditional density

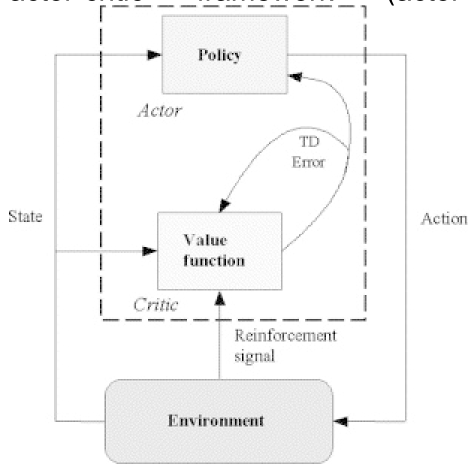
unsupervised

- detect structure of data without further information (no labels, desired output)
- dimensionality reduction, preprocessing, assign data points to clusters/categories, source separation
- can be modelled as an unconditional density in feature space

reinforcement learning

- no examples, but feedback is given
- strategy learning → learn the best action for a given situation

- actor-critic framework (actor → policy; critic → value function)



Connectionist neuron (description and formula)

Mathematically it is a linear filter with a subsequent (static) nonlinearity. The operation of the linear filter is called *feature extraction* and and the nonlinearity can be seen as an *evaluation* of the filters result. The filter is e.g. a projection on the most informative direction in the input space. The nonlinearity could also be a linear function (e.g. in case of an output neuron), but in general it is nonlinear because a network of linear neurons only has the computational power of a single neuron. typical transfer functions are:

- linear (complex linear feature extraction or in the output layer for regression)
- signum (for classification)
- sigmoid or tanh (as nonlinearity in lower layers -> both computationally equivalent)

Recurrent Neural Networks (RNN)

Throughout the MI course we mostly dealt with feed-forward networks as they are much easier to train and analyse. But they are of course also less powerful and biologically less plausible. Recurrent Neural Networks can be used to model complex dynamic systems, they are applied to spatio-temporal pattern analysis (pattern recognition on time-series) and can implement associative memory and pattern completion. Two examples for RNNs are:

- **hopfield networks** (binary neurons, symmetric weights, hebbian learning rule, no self-projections)
- **boltzmann machines** (binary (with extension to real) variables, symmetric weights, no self projections, minimization of energy, stochastic update rule, successful only when connections between hidden layer neurons are forbidden (restricted boltzmann machines).

Feed-Forward Neural Networks (FFNN)

Feed-Forward Neural Networks are much easier to train and analyse then RNNs, yet they are still computationally powerful systems. There is a proof from the 90ies that a 3 layered neural network, with a non-constant, continuous, bounded and monotonously increasing transfer function, can approximate any continuous, real valued function. FFNNs are mostly used for the prediction of attributes or class labels, the association between variables in static situations without dynamics. These networks are only characterized by their common structure (e.g.: MLP, RBF, SVM) but use completely different learning algorithms (MLP → backpropagation; RBF →

to stage process; SVM → structural risk minimization). The parameters that have to be set and learned for a FFNN can be distinguished between architectural parameters (number of nodes, number of edges, number of layers) and the parameters of a certain model including its weights and thresholds.

Error Measures

Different error measures can be used for optimization problems. For differentiable error measures usually the gradient with respect to its weights is computed and used for gradient based methods. The choice of the error function has a big impact on the outcome of the training. The *squared error* is very common (for regression it is the maximum likelihood estimate when the errors belong to a normal distribution.), but also functions with bounded penalty, linear penalty or tolerated small errors could be used to e.g.: ignore or emphasize the influence of outliers.

Generalization Error (E**R)

In one short and simple sentence: It is the average cost per prediction according to the mathematical expectation (true distribution). The generalization error is given for a learning problem where the distribution of the data and the supervisor output (labeling) is known. This is the value that describes the quality of your model but it cannot be computed as neither the data distribution nor the correct labeling is known. If it would be know you wouldn't need any machine learning. The generalization error is approximated by the training error. Statistical learning theory deals with the question when this approximation works. The true generalization error can be estimated via test sets or better: cross-validation.

Empirical Risk Minimization (ERM)

ERM is simply to use the empirical average instead of mathematical expectation. This step is done to approximate the generalization error which cannot be computed due to the missing underlying distributions. Using the empirical average gives us the training error that estimates the performance of a model on a given dataset. The problem is that a minimization of the training error might fit the model only to the given data set and performs bad on new/unseen data. This is why the model that has been trained by minimizing the training error (model selection) has to be evaluated by a test set or via a cross-validation loop. The training is done until a certain criterion is fulfilled, which is usually the number of iterations, a certain amount of time, no (small) changes in error value, certain training error reached or the validation criterion reached.

Gradient Descent

The gradient descent method needs an error function which can be differentiated with respect to the weights of the model or the gradient has to be approximated numerically. Gradient descent is a very common local optimization method (in contrast to global methods like simulated annealing it only looks at local information). Gradient descent allows to modify the weights in a way that decreases the value of the error function. Graphically speaking it looks for a (local) minimum in the multidimensional landscape given by the error function. One method to compute the derivative with respect to the model weights is for example the backpropagation algorithm for MLPs. The difficulty of many gradient based methods is to choose the correct step size on the gradient. When the step size is too small the algorithm converges very slowly and when it is too large the algorithm often jumps over the deepest point of the valley which leads to oscillations. A problem of local optimization methods (like gradient descent) is that they easily get stuck in local minima. Local optima and zig-zagging can be avoided by impulse terms and variable step sizes (like for example in rprop where the step size is increased when the direction of gradient did not change and drastically decreased when it changes. Another problem can appear when the model is not regularized on the magnitude of the weight values. Gradients become small for big weights when the transfer functions saturate.

Backpropagation

Backpropagation is a method to use gradient descent also for multi layered neural networks The main trick is to apply the chain rule in order to compute the derivative of the error function with respect to the weights. It is an efficient algorithm for message passing (who caused the error) in feed-forward neural networks and has complexity of O(n) → linear in the number of weights and thresholds.
forward pass → compute error → compute error w.r.t. weights → update weights

Validation of model selection

This is the same question as: is our training error a reasonably good approximation of the generalization error? Typically two methods are used. The first is the **test set method** which simply splits the data in a training and a test set → trains on the training set, tests on the test set. This method is only possible when the data set is big. And usually, it is not big enough! It gives a generalization estimate for a certain trained model, not including the model selection procedure. The second method is **cross-validation** which is more powerful and normally used as validation procedure. It splits the data in a training and a test set, trains and tests the model on this set. (typical n is 5-10, but could also be # of patterns, which is called *leave one out* validation (computationally expensive)). Here the algorithm in a few simple steps:

- split data set in n subsets
- train the model on all patterns but with one subset missing
- evaluate on the subset
- repeat and average over results from different subsets
- when hyperparameters for a model have to be selected, this should also be done within the cross validation loop

Cross-validation is computationally more expensive as the training as to be repeated over and over, but its big advantage is that it gives a validation estimate for the whole learning procedure and not only for the certain dataset it was once trained on.

Online Learning

In contrast to batch learning the weight update is performed after each training pattern, it is “learning while doing”. There are several reasons for applying on-line learning. Either because the whole data set is not accessible in the beginning and the data points are just collected during the training. Then on-line learning is the only possibility and it has the advantage that the serial presentation of the training data helps to adapt in the case of a nonstationary environment.
But also if the whole dataset is available from the beginning, there are reasons of doing online learning. The serial presentation of training patterns (randomly selected) makes the updates stochastic, helps to avoid local minima and has a smaller memory footprint. The disadvantage is that there is no general convergence proof for MLP online learning and that improved gradient descent methods (impulse terms, adaptive stepsize) are not applicable for online learning. Although there is no general convergence proof for *stochastic approximation* there are some results that can guarantee convergence under conditions. Robbins and Monro (1951) developed an algorithm which works for convex optimization (according to the script). Could not find anything short on the theorem of bottou, but this wikipedia article is very well written.

Improved gradient-descent

Gradient descent is a *local optimization method* (in contrast to e.g. stochastic optimization) and like other methods of this type, it has the problem of getting stuck in local minima. But this is a general problem which is hard to overcome in a non-convex multi-dimensional *error landscape*. Other problems are slow convergence (when learning rate is too small) or oscillations when the learning rate is too large. There are two common approaches to deal with this problems. Both methods can only be used with batch learning!

impulse terms

Don't completely change your mind all the time. Always add the old update term (delta_w) (scaled down by a certain factor) two your new direction determined by the gradient. This is a running average the focuses on the current gradient but still considers its last decision.

adaptive step size

When you already walk since a while and it got better and better, you can speed up constantly. But if you realise its getting worse, slow down immediately. This is the rprop algorithm explained in the oby script. Two different factors are used for for the updates of the step-size. The one to increase the value is typically around 1.2 and the one to decrease around 0.5.

Conjugate Gradient method (CG)

This is the most popular gradient descent method for unconstrained optimization problems. It is iteratively applied in two steps and converges after a maximum of N steps (N -> dimensions of input data). The first step is computing a for this step optimal gradient and the second step to compute the optimal step-size on this previously chosen gradient via line search.

conjugate gradient

Two vectors are conjugate when they are orthogonal after rotation by the matrix A. (<x, Ay> = 0). The conjugate gradient method finds a base of gradient vectors which are conjugate to each other. In each step, one of the conjugate search directions is processed and will not be searched again, since the minimum can be expressed in a basis of conjugate vectors. Hence, it is guaranteed that CG converges in n steps. This method resembles the Grahm-Schmidt process of orthonormalization in the way that in each iteration a direction is chosen. A projection on this direction will then be subtracted from all following results. Via the Pollach-Ribiere Rule all we have to compute is the derivative of the error function with respect to its weights (via backprop!?) to compute the conjugate gradient. Here the formulas, d_t+1 is the new direction:

the gradient:
$$\underline{\mathbf{g}}_{t+1} := \left.\frac{\partial E^T}{\partial \underline{\mathbf{w}}}\right|_{\mathbf{w}_{t+1}}$$

the update rule:
$$\underline{\mathbf{d}}_{t+1} = -\underline{\mathbf{g}}_{t+1} + \beta_t \underline{\mathbf{d}}_t$$
$$\beta_t = \underbrace{\frac{\underline{\mathbf{g}}_{t+1}^T (\underline{\mathbf{g}}_{t+1} - \underline{\mathbf{g}}_t)}{\underline{\mathbf{g}}_t^T \underline{\mathbf{g}}_t}}_{\text{Pollach-Ribiere rule}}$$

*determine the optimal step size via **line search***

Line search is an iterative method to find the local minimum of a multi dimensional function on a line (single dimension) of the objective function.

- fit parabola through three points
- calculate location of minimum
- stop if criterion fulfilled
- change points and start over

Over- and Under-fitting

Over- and underfitting are problems that arise when the model complexity is chosen wrongly or no regularization is applied to bias the model towards the simplest solution that describes the data. **Overfitting** can be detected when a model performs well on the training data but has bad generalization. This usually happens on small datasets or when the model complexity is chosen to high. Another reason could be that the training error does not vary strongly over a large volume of parameter space. The "free model complexity" is then used to adapt to noise in the data instead of meaningful interpolation (generalization). I just read a paper that this also highly depends on the used optimization procedure. Backpropagation seems to refine its non-linear approximation in difficult areas without putting its free complexity into boring linear segments. Only the very efficient methods like CG tend to overfit in boring areas.[1]
Underfitting happens if the model complexity too low because structure and properties of the data cannot be captured by the simple model model. An example is to try fitting a line to data coming from a parabola with noise.

Bias and Variance of estimators

The training error can be seen as an estimator of the generalization error and the analysis of its bias and variance helps to explain the over- and underfitting problems. Bias and Variance can be computed by taking many data sets of equal size, drawn iid from the same distribution. The expected value of the loss, over the different data sets, can be decomposed into bias and variance.

$$\langle(\hat{y}-y)^2\rangle_{\text{all datasets}} = \underbrace{(\langle\hat{y}\rangle - y)^2}_{\text{bias}^2} + \underbrace{\langle(\hat{y}-\langle\hat{y}\rangle)^2\rangle}_{\text{variance}}$$

The bias of the training error (generalization error estimator) shows how systematically it over or underestimates the real generalization error and the variance of the estimator tells something like how often the estimate is wrong. There is usually a trade-off between low bias and low variance of an estimator and as a good estimate of the generalization error we want both values to be low. This means a correct estimate of the true generalization error with little mistakes. A key aspect of many supervised learning methods is that they are able to adjust this trade off between bias and variance (either automatically or by providing a regularization parameter that the user can adjust).

Model Class Complexity

The model class complexity tells us the "power" of the model, how complex the data relations can be and still be captured by the model. model complexity too low leads to high training error rates and high validation error because the essential relationship of the data is not captured by the model. if it is too high, it leads to overfitting -> low training error, high validation error. The complexity of a model can for example be measured by its VC-Dimensions which are minimized

[1] Caruana, Rich, Steve Lawrence, and Lee Giles. "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping." *Advances in neural information processing systems* (2001): 402-408.

during SRM. Bayesian Model selection automatically prefers simpler model if they are equally good in explaining the data.

Regularization

Regularization is a soft restriction of the model complexity class. This allows to choose a model class which is definitively complex enough to capture the relation in the data, but tends to find a solution as simple as possible. The previous error term is replaced by a risk function that consists of the old error term and a regularization term. The influence of the regularization term can be selected by the hyperparameter lambda:

$$E^T + \lambda E^R \overset{!}{=} \min$$

The correct value of the regularization parameter lambda has to be estimated in an additional step → nested cross-validation.

Regularization techniques are for example weight pruning, optimal brain damage or weight decay. Weight decay corresponds to an implicit smoothness assumption and keeps the model simple by restricting the neurons to their linear regime. Dimensions which are not supported by data decay to zero. Weight decay is a whole family of regularization methods, depending on the norm imposed on the weights. L2 norm leads to generally smaller weights, L1 to sparse coding (also other L_q normes might be used). Regularization techniques allow to add previous knowledge to the model which corresponds to the prior specification in a bayesian formulation. Other regularizations could also be to bias the model class towards monotonous ore symmetric functions. This can be done for example with a penalty term that increases with non symmetry of the model output.

nested Cross-Validation

The choice of regularization parameters has to be made within a model selection loop. This is called a nested cross-validation which can be used to select the correct regularization parameter. This is simply done by the following pseudocode:

```
for lambda in lambdas:
    perform n-fold cross-validation on data
pick optimal lambda and train model on the whole data
```

To assess the quality (estimate of the generalization error) of the whole regularized model, a nested n-1 fold cross-validation has to be done in order to estimate the average performance when the best lambda is picked. This is done according to the following pseudocode:

```
data = split_in_folds(data)
for outer_fold in data:
    for lambda in lambdas:
        for inner_fold in data - outer_fold:
            model = train_model(data- outer_fold - inner_fold, lambda)
            compute_error(model, inner_fold)
    best_lambda = pick_best_lambda()
    model = train_model(data - outer_fold, best_lambda)
    compute_error(model, outer_fold)
```

multi-class classification

Many classifiers can only distinguish between two classes (e.g. SVM, LDA, etc). There are several ways to deal with this problem. Some techniques (like the MLP) can be extended for multiple classes and and for others ensemble techniques have to be used. For the ensemble

solution either several classifiers are trained as *one-vs-the-rest* classifiers and then decide in a winner-takes-all strategy or the classifiers are trained as one-vs-one class classifiers and then all vote for class assignment. For MLPs the prediction of class-labels is not possible with the old machinery because gradients cannot be computed on ordinal or binary class attributes. Luckily a simple trick helps to overcome this problem → class labels are seen as real valued class probabilities. This is done by using a *1 out of c coding* of the ordinal data. For example the ordinal variable of car manufacturers would be translated in a vector $y = karman, y \in \{karman, vw, astonmartin\} \Rightarrow y = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ where each entry gives the probability of belonging to a certain class. The output layer of the MLP would now consist of several output neurons, each giving the probability of belonging to a certain class y_k(x,w) := P(C_k | x,w). This output needs a normalization to fulfill the laws of probability (summing up to 1). This normalization is done by the softmax function which is used because it is a smooth, differentiable approximation of the maximum function which is even biologically plausible (lateral inhibition).
But also some other part of the previous method has to be changed. Often the quadratic error was used with the assumption of gaussian noise in the residual values. But this does not make sense for probability values. The *Kullbach Leibler Divergence* is a *pseudo*-metric (non symmetric) for the distance between two distributions. Plugging in the KL-Divergence as cost function, parts of it can be removed because they don't depend on the model parameters and are therefore not interesting for optimization, the error function simplifies to measuring the *cross entropy* between the network output and the label vector. The empirical version of this formula looks like:

$$E^T = -\frac{1}{p} \sum_{\alpha=1}^{p} \sum_{k=1}^{c} y_{Tk}^{(\alpha)} \ln y_{k(\underline{\mathbf{x}}^{(\alpha)};\underline{\mathbf{w}})}$$

Softmax multi-class networks can also be validated by the test-set or cross-validation method. For the actual prediction of class labels additional factors might be considered, e.g. whether all terms are equally costly (is confusing a banana with an apple maybe worse than confusing an orange with an apple).

RBF Networks

The RBF network is a structurally really simple network that consists of one nonlinear feature extraction layer with a subsequent linear evaluation. In contrast to a two layer MLP the input layer consists of neurons implementing radial basis functions instead of linear filters with sigmoid evaluation functions. Geometrically this major difference can be seen as moving from a discriminative analysis (separating hyperplanes) to a prototype based analysis (clusters) focusing on categories. It is a more local approach of looking at the data. As an example: being on the same side of a separating hyperplane does not yet say much about having many things in common. You have to draw hyperplanes along many directions to get only data points that really have something in common. But being close to each other in space (especially when there is a cluster in this area) might give you valuable information about having things in common. A big advantage is that RBF networks can be trained in a two-step procedure where the first step is unsupervised and can therefore **use unlabeled data**! This first step involves cluster detection (analysis of input space) to place the prototypes in the right areas of the space. Only the second step, learning the weights of the output layer, depends on labels. This makes RBF networks suitable for low-dimensional data or data with a pronounced cluster structure.

pros:

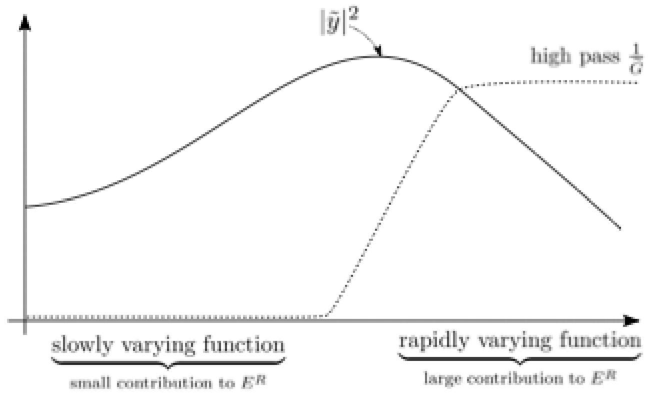
- fast convergence during the training, because of the local approximation property (only a few rbf kernels close to a data point)
  - only a few params have to be changed per training point
  - *credit assignment* is simple (which weight caused the error)
- second learning step can be solved analytically

cons:

- curse of dimensionality when high dimensional and no clusters (when no clustering step is applied and the rbf functions are uniformly spread over the input space the number of required rbf functions increases exponentially to the number of dimensions.)

RBF and regularization

When the model is of too high complexity, the learning problem is often ill-posed which leads to overfitting and an estimator with high variance. One way of regularization in function approximators is to filter them in fourier space. Usually regularization is done by adding a smoothness constraint to the function approximator in order to reduce the complexity of the model. This can be done by punishing for high frequencies (e.g. wiggly polynomial) through a high pass filter in the fourier domain. ! We don't filter the signal, we just increase the regularization part of the risk function with high frequencies. The role of the filter can be seen in the following picture:



This reasoning adds the following regularization term $E^R = \frac{1}{2}\int d\underline{\mathbf{k}} \frac{|\tilde{y}_{(\mathbf{k})}|^2}{\tilde{G}_{(\mathbf{k})}}$ to the original error function: $\int d\underline{\mathbf{k}} e^{(i\underline{\mathbf{k}}^T \underline{\mathbf{x}})} \tilde{y}_{(\mathbf{k})}$. The minimization of the whole risk term leads to the following model: $y_{(\underline{\mathbf{x}})} = \sum_{\alpha=1}^{p} \mathrm{w}_\alpha G_{(\underline{\mathbf{x}} - \underline{\mathbf{x}}^{(\alpha)})}$ which is something like a weighted density estimate. The prior knowledge e.g. smoothness constraint (high-pass) determines the basis functions (gaussian for high-pass). This gives an analytical solution of how to compute the weights and with this yet another model selection procedure for RBF networks. This method does not scale well as the number of basis functions equals the number of data-points. Previous description of this part of the script: Models of the type of RBF networks (feature mapping → linear evaluation) can also be regularized. They are a model of all continuous and differentiable functions, but this *power* of the model often leads to an ill-posed learning problem (overfitting) as many functions might fit the same training data. In this case it is a common strategy to use the simplest/smoothest function that is able to model the data (with a reasonably small training error). In MLPs this smoothness constraint was introduced via weight decay. Here an error term is added which punishes high frequencies in the fourier transform of the approximated function, which is also an implicit smoothness constraint, since high frequencies in the estimated model correspond to a 'rough' output function of the model. Minimizing such a regularized model results in a change of basis functions; namely, the model is essentially based on the inverse fourier transform of the filter applied in the fourier space.

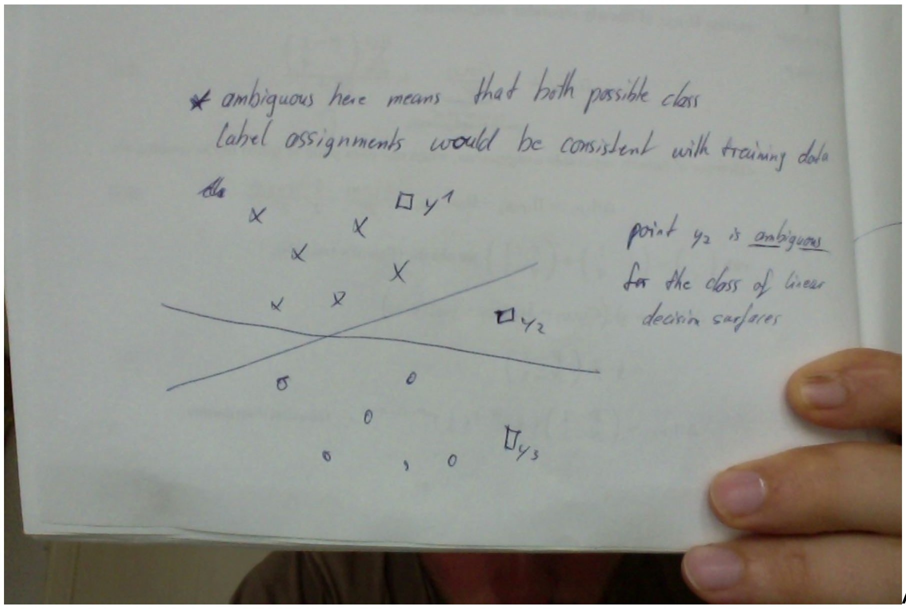
Statistical learning theory (SLT)

The main question of SLT is: under which constraints and conditions does ERP work. Which constraints on the data, amount of data and the model class have to be fulfilled in order to make ERM (replacement of the mathematical expectation by the empirical mean) work? Statistical learning theory is the basis for the development of SVMs. Basically it deals with the question: when does the minimum of the training error result in a reasonably small generalization error. What we are looking for is a predictor, a model defined by its parameters w, which converges to the optimal predictor, at least for the case of infinite samples. This can be stated by the following formula: $\lim_{p \to \infty} P\left\{ \left| R_{(\underline{\mathbf{w}}_p)} - R_{(\underline{\mathbf{w}}_0)} \right| \geq \eta \right\} = 0 \text{ for all } \eta > 0$ and this can be achieved by finding conditions for which the empirical risk converges to the true risk: $\lim_{p \to \infty} P\left\{ \left| R_{emp}(w_p) - R(w_o) \right| \geq \eta \right\} = 0$. But what are the conditions for this convergence? Here we come to the **Key Theorem** of learning theory which states that when the risk function is bound, the empirical risk converges to the true risk even for the worst model (necessary for strict consistency which implies that ERM works). This tells us that we always have to do a *worst case analysis* in order to show that ERM works. The key theorem as a formula (for bound R_(w)):

$$\lim_{p \to \infty} P\left\{ \sup_{\underline{\mathbf{w}} \in \Lambda} \left(R_{(\underline{\mathbf{w}})} - R^{(p)}_{\mathrm{emp}(\underline{\mathbf{w}})} \right) > \eta \right\} = 0 \text{ for all } P_{(\underline{\mathbf{z}})} \in \Pi$$

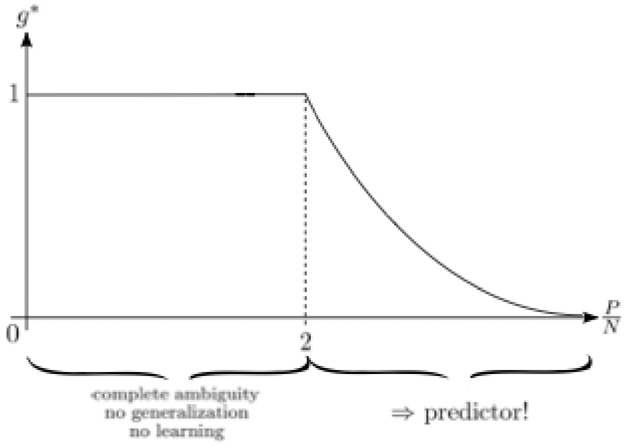
The convergence of the empirical risk to the true risk is related to the law of large numbers which states that a *sequence of means* converges to the *expectation* of a random variable as the number of samples increases. The problem is that this law only works for a countable number of elements and most of the model classes we use are infinite sets (like a MLP with real numbers as weights). To overcome this problem the notion of a capacity measure was introduced that characterizes sets of predictors into equivalence classes w.r.t their capacity. The capacity measure used for SLT is the VP-Dimension. But before this is introduced we should motivate why to restrict the power of a model at all. Simply, because a model too powerful can learn any distribution and does not generalize at all. For example an extremely high degree polynomial could be that wiggly that it correctly separates two classes of a very noisy distribution. And this does not help to make any predictions on an unseen data point. Only when the polynomial is restricted to a smoother shape an unseen data point becomes more likely to belong to one or another class. The models that can learn anything are called *optimal predictors*. To investigate the point from which a model is not optimal anymore and starts generalizing we have to look at the number of *ambiguous assignments*[2]. These are the assignments which assign labels to p-1 data points but still would be consistent for both labelings of the p-th point.

[2] http://neuron.eng.wayne.edu/tarek/MITbook/chap1/1_5.html


∗ ambiguous here means that both possible class label assignments would be consistent with training data

point y_2 is ambiguous for the class of linear decision surfaces

Apparently, I can’t find the proof, this number of ambiguous assignments is given by C(P, N-1) where C(P, N) would be the number of possible assignments for P data points in an N dimensional space. The limits of the fraction (g*) of the number of ambiguous assignments to the number of all possible assignments shows that the training_patterns / dimensions ratio has to exceed a certain threshold before the model gets any predictive power. The graph is annotated with P,N → infinity, but the ratio being constant. I think this only means that the ratio is valid for large P, N, but most important is still the ratio between the two.



This capacity of a model is further formalized by the introduction of the VP-dimension and the definition of a growth function which finally helps to answer our question from the beginning: when does ERM work? The first step is to define a cost/indicator function Q(z, w) for a data point and a model. It yields 0 for correct and 1 for incorrect classification. These individual cost values are combined in a binary cost vector, each dimension is a data point from a given set. The VP-dimension (capacity measure of a model class) is simply the largest number p (number of data points) on which (at least for one set) 2**P different loss vectors can be assigned ⇒ the number of data points for which the model class is completely ambiguous. The growth function is defined as the logarithm of the supremum of number of possible assignments for p points with a certain model class. The supremum is taken over the points which means it chooses the *most ambiguous* data points. $G^{\Lambda}_{(p)} = \ln\left(\sup_{\mathbf{z}^{(1)},\dots,\mathbf{z}^{(p)}} N^{\Lambda}_{(\underline{\mathbf{z}}^{(1)},\dots,\underline{\mathbf{z}}^{(p)})} \right)$ This growth function increases linearly up to the VC-dimension and is from there on bound by an exponential which guarantees generalization when the number of training patterns exceeds the VP-dimension of the model. And back to the main question it says that: Inductive learning using the ERM method is only guaranteed (for large enough sample size) if d_VC is finite. And there are even tighter bounds that can be achieved by using capacity measures which are more sophisticated than a simple d_VC. All this results are for the number of sample points going to infinity. But there are also results for a finite sample size which allow to compute bounds for epsilon and eta.

SVM and structural risk minimization (SRM)

The results of the previously introduced statistical learning theory show that the complexity class of the models has to be restricted, especially in high-dimensional spaces, in order to generalize. The formalization of this problem led to an approach to trading off empirical error with hypothesis space complexity which became known as Structural Risk Minimization[3]. Instead of minimizing the training error (and maybe regularizing by some prior knowledge), SRM directly minimizes the model complexity while keeping the training error on an acceptable level. A learning algorithm directly employing SRM is the Support Vector Machine (SVM) learning method. SVMs have the big advantage that the VC dimension of large margin hyperplanes doesn't need to depend on the dimensionality of the feature space.

Introduction of the SVM algorithm

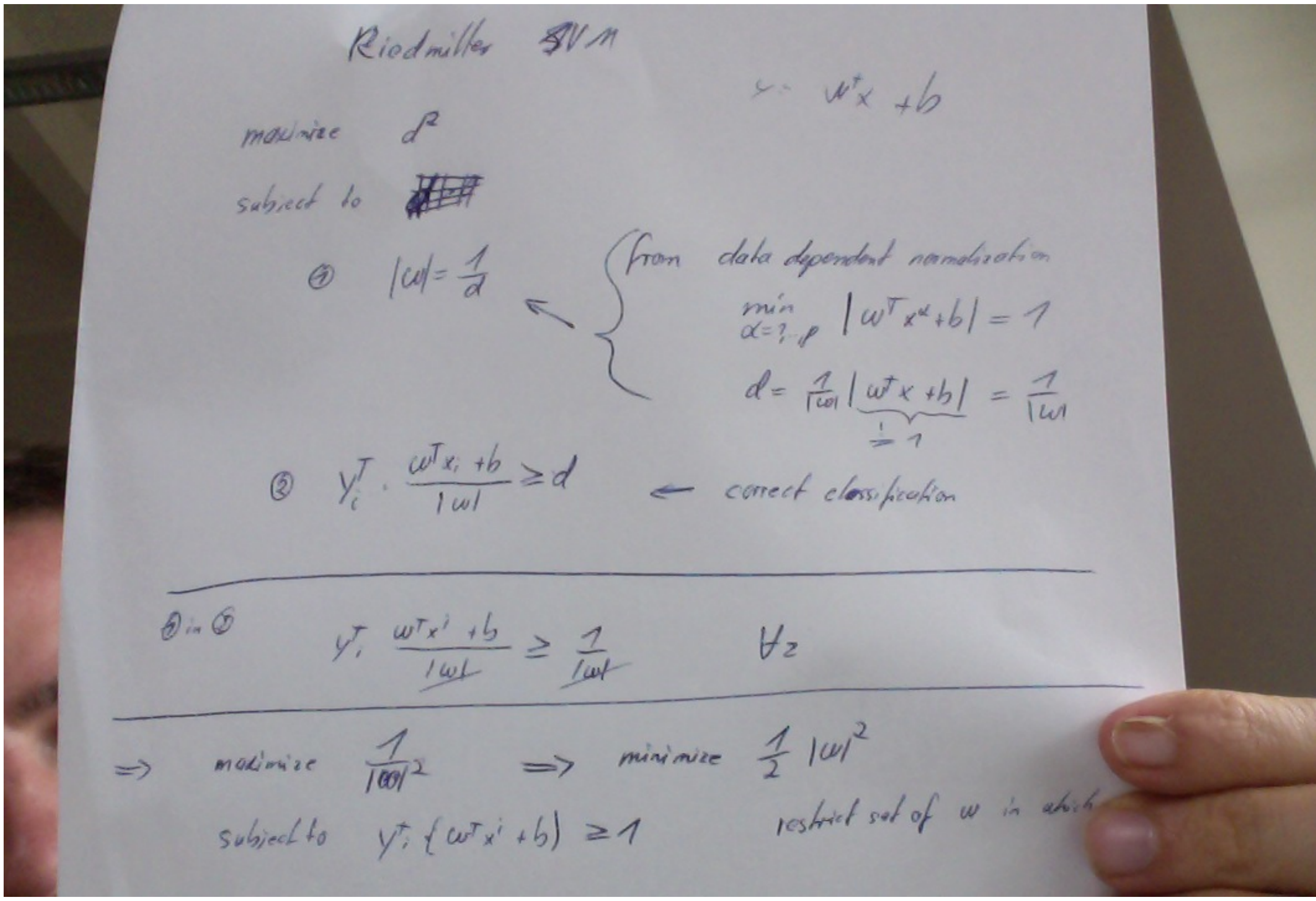
The idea of the SVM algorithm is based on the result for the d_VC of a *large margin classifier*:

Let $|\underline{\mathbf{x}}| \leq R$ (bounded range of $\underline{\mathbf{x}}$ for which $P_{(\underline{\mathbf{x}})} \neq 0$), then:

$$\mathrm{d_{VC}} \leq \min\left(\underbrace{\left[\frac{R^2}{\mathrm{d}_{\mathrm{w}}^2}\right]}_{\text{New!}}, N\right) + \underbrace{1}_{\substack{\mathrm{d_{VC}}\text{of the}\\ \text{connectionist}\\ \text{neuron}}}$$

The VC-dimension is still limited, even for an infinite or huge feature space, if the margin is large enough. The size of the margin (d) has direct influence on the model capacity, which is easily understood when the effect of a larger margin is investigated geometrically. Because of this, the SVM basically doesn't do much more than maximizing the margin while still correctly classifying the training patterns. The problem can be stated as an optimization problem with constraints:

[3] http://www.svms.org/srm/


Riedmiller SVM

$y = w^T x + b$

maximize d^2

subject to

① $|w| = \frac{1}{d}$ $\leftarrow$ $\left\{ \begin{array}{l} \text{from data dependent normalization} \\ \min_{\alpha=1..p} |w^T x^\alpha + b| = 1 \\ d = \frac{1}{|w|} |\underbrace{w^T x + b}_{\overset{!}{=} 1}| = \frac{1}{|w|} \end{array} \right.$

② $y_i^T \cdot \frac{w^T x_i + b}{|w|} \geq d$ $\leftarrow$ correct classification

① in ② $\quad y_i^T \frac{w^T x^i + b}{\cancel{|w|}} \geq \frac{1}{\cancel{|w|}} \quad \forall i$

$\Rightarrow$ maximize $\frac{1}{|w|^2}$ $\Rightarrow$ minimize $\frac{1}{2}|w|^2$

subject to $y_i^T (w^T x^i + b) \geq 1$ restrict set of w in which

The reformulation above shows that the minimization of the norm of the weights is equivalent to the maximization of the margin. This gives us a way to control complexity by a nested set of models (with decreasing complexity for larger margins).

Because the constraints are inequality constraints we have to apply the Theorem of Karush, Kuhn and Tucker which states the necessary conditions for using inequality constraints in a Lagrangian. These conditions become sufficient for a quadratic objective function with linear constraints, and yeah this is what we have so we are doing convex optimization with only **one global optimum**. Furthermore the KKT theorem states the conditions necessary for solving the dual instead of the primal problem. $\min_{\mathrm{x} \in A} L_{(\mathrm{x}, \{\lambda_k^*\})} = L_{(\mathrm{x}^*, \{\lambda_k^*\})} = \max_{\lambda_k \geq 0} L_{(\mathrm{x}^*, \{\lambda_k\})}$ Applying the theorem and reformulation into the dual:

- state the lagrangian: $$L = \frac{1}{2}|\underline{\mathbf{w}}|^2 - \sum_{\alpha=1}^{p} \lambda_\alpha \left\{ y_T^{(\alpha)} \left(\underline{\mathbf{w}}^T \underline{\mathbf{x}}^{(\alpha)} + b \right) - 1 \right\}$$ $\underline{\mathbf{w}}, b$: "primal variables" $\sim$ dimension of feature space

  λ_α : "dual variables" $\sim$ number of training data
- analytically minimize w.r.t.: w and b (derivative, set to 0)
- insert the results back into the Lagrangian
  - w can now be represented by a linear combination of data points

    - b vanishes when its derivative set to 0, but still stays as a constraint (important when doing sequential minimal optimization (SMO)).
    - ⇒ all vector operations become dot products! ⇒ important for the kernel trick.
- This result is called the *dual formulation* of the optimization problem and helps us gaining significant insight into the structure of the problem in combination with the KKT conditions → e.g.: that the optimization surface is only constrained by the Support Vectors (the ones closest to the margin).

This dual problem can now be solved by applying the method of Sequential Minimal Optimization (SMO), which is a modification of the coordinate search algorithm. This original algorithm keeps all but one parameter (the lagrangian lambdas in our case) fixed and then only optimizes the single free parameter. Unfortunately this is not possible for the dual as setting the derivative with respect to b to zero imposes the following constraint:

$$\frac{\partial L}{\partial b} = -\sum_{\alpha=1}^{p} \lambda_\alpha y_T^{(\alpha)} \overset{!}{=} 0$$

The lambdas cannot be optimized individually without breaking this constraint. Therefore SMO uses always a pair of parameters which are picked by certain heuristics (pick lambda that violates a KKT condition, pick the second one to make a big step) in order to speed up the search. SMO is an efficient implementation of the SVM learning algorithm. After learning the parameters lambda, the SVM can be applied for classification:

$$\begin{aligned} y &= \operatorname{sign}\left(\underline{\mathbf{w}}^T \underline{\mathbf{x}} + b\right) \\ &= \operatorname{sign}\left\{ \sum_{\text{SV}} \lambda_\alpha y_T^{(\alpha)} \underbrace{\left(\underline{\mathbf{x}}^{(\alpha)}\right)^T \underline{\mathbf{x}}}_{\circledast} + b \right\} \end{aligned}$$

Kernel Trick

In general the kernel trick is to replace all dot products by kernel functions. According to Mercer's Theorem each positive definite kernel function corresponds to a scalar product in some metric feature space. This trick can be applied to many linear methods and has two advantages:

1. a computational advantage because the feature mapping doesn't have to be computed explicitly. (O(n**d) might change into O(n))
2. the implicit feature mapping can be nonlinear and because of this the linear SVM might learn nonlinear decision boundaries.

There are different kernel functions that proved to be useful:

- polynomial kernel (image processing → pixel correlations)
- RBF kernel (infinite dimensional feature space → preferred kernel) SVMs can deal with infinite dimensions because of the SRM principle. So the SVM with a RBF kernel does classification by a weighted sum of kernel functions. This is exactly the same as a RBF-network, but the learning algorithm is completely different. And the SVM has better control for overfitting.
- tanh - neural network kernel (not positive definite)
- plummer kernel (scale invariant kernel)

And there are other methods that can be kernelized

- fisher discriminant analysis
- PCA
- K-means and self organizing maps

SVM variants

C-SVM

The SVM has the big advantage that it automatically reduces model complexity. But it also attempts to do a classification without any mistakes which might lead to overfitting in the case of overlapping classes or outliers. To overcome this problems, training patterns in the C-SVM are now permitted to have (functional) margin less than 1, and if an example has functional margin 1 − ξi (with ξ > 0), we would pay a cost of the objective function being increased by Cξi. This leads to a re-formulation of the basic optimization problem: $\frac{1}{2}|\underline{\mathbf{w}}|^2 + \frac{C}{p}\sum_{\alpha=1}^{p} \varphi_\alpha \overset{!}{=} \min$ (minimize weights and punish for mistakes) under the constraints:

$$y_T^{(\alpha)}\left(\underline{\mathbf{w}}^T \underline{\mathbf{x}}^{(\alpha)} + b\right) \geq 1 - \varphi_\alpha \quad \text{correct classification of all data points with margin } |\mathbf{w}| \text{ for } \varphi{=}0$$

$$\varphi_\alpha \geq 0 \quad \text{"margin errors" for } \varphi \neq 0$$

Interestingly this results in only a minor change of the dual formulation. The constraint on the parameter lambda becomes: $0 \leq \underbrace{\lambda_\alpha \leq \frac{C}{p}}_{\text{difference to linearly separable case}}$

Oldschool AI, first order logic in the real world

In the more traditional AI approach, knowledge was usually modelled by propositions and conditionals for deduction. But there are many shortcomings of this approach in a complex and stochastic environment. A rule might be almost but not always true, an effect might have an almost unlimited list of causes, there is often no complete theory of the domain, etc. A new approach is required to model knowledge where we are hardly ever 100% sure, that it is true. The solution is to only state the ***degree of belief*** in a certain proposition and treat this degrees of belief as if they were random variables on a certain domain and obey the laws of probability theory. Only if the *beliefs* of an agent follow the laws of probability theory, optimal decisions can be made in a multi-agent environment (see Betting Agents, 1931).

To model a certain situation/world in such a framework, the random variables (degrees of belief in an event) and their domains have to be stated (e.g variable: weather; domain: {sunny, rainy}, p(weather=sunny) = 0.3). Now the whole *world* can be modelled by atomic events which are each a complete specification of the state of the world.

- atomic events are mutually exclusive
- the set of atomic events must be exhaustive (all values of the domain have to be used)
- a proposition is a disjunction of atomic events
- $cavity = true$ is equivalent to:
  $$(cavity = true \wedge toothache = false \wedge weather = sunny) \vee$$
  $$(cavity = true \wedge toothache = true \wedge weather = rainy) \vee \ldots$$

Two different kinds of probabilities have to be defined in order to do interesting computations (inference) with the propositions.

- **prior probability**: simply the probability of a certain event, in absence of any other information (e.g. P(cavity=true) = 0.01, P(weather=sunny, country=germany) = 0.000001)
- **conditional probabilities** which is also called the **product rule**: (e.g. P(cavity=true| toothache=true)): ones knowledge about the world, given a (not necessarily complete) set of observations → the evidence. The conditional probability of the event X, given the event Y can be computed as the joint probability of X and Y, divided by the probability of Y.

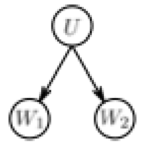
$$P(\underbrace{C}_{\text{variable}} \mid \underbrace{t}_{\text{value}}) = \frac{\overbrace{P(C,t)}^{\text{joint probability}}}{P(t)}$$

$$P(C,t) = P(C|t)P(t)$$

Together with the process of **marginalization** or **sum rule** of probability theory, inference can be done on a complete table of atomic events (joint probabilities). To query the probability of X, given the evidence E, we only have to marginalize over the unobserved variables.

$$P(x|\underline{\mathbf{e}}) = \underbrace{\frac{P(x,\underline{\mathbf{e}})}{P(\underline{\mathbf{e}})}}_{\text{def. of cond. probability}} = \underbrace{\alpha P(x,\underline{\mathbf{e}})}_{\text{normalization}} = \underbrace{\alpha \sum_{\underline{\mathbf{y}}} P(x,\underline{\mathbf{e}},\underline{\mathbf{y}})}_{\text{marginalization}}$$

Note that P(e) does not have to be computed explicitly because of the normalization of probabilities to 1. But the **big problem** here is that modelling the *world* by joint probabilities is very inefficient and does not scale. Storage and Inference (marginalization) are both computationally very expensive using joint probabilities. This can be overcome by using **conditional independence** for efficient representations of the world knowledge. It is defined as P(X, Y| Z) = P(X | Z) * P(Y | Z) and reads that X and Y are conditionally independent given Z, which of course does not mean that X and Y are independent!!! They might even be completely dependent, but the conditional independence means that the don't contain any further information about each other, when the variable they both depend on (Z) is already given. Conditional independence statements allow a clever and efficient decomposition of the knowledge base in a graph structure. Here the graphical representation of conditional independence:



$$P(W_1, W_2|U) = P(W_1|U)P(W_2|U)$$

In a larger graph of causal relations, there are several statements of conditional independence:

- a node is conditionally independent *of its non-descendants*, given its parents
- a node is conditionally independent *of all other nodes in the network* given its Markov Blanket (parents, children and childrens parents)

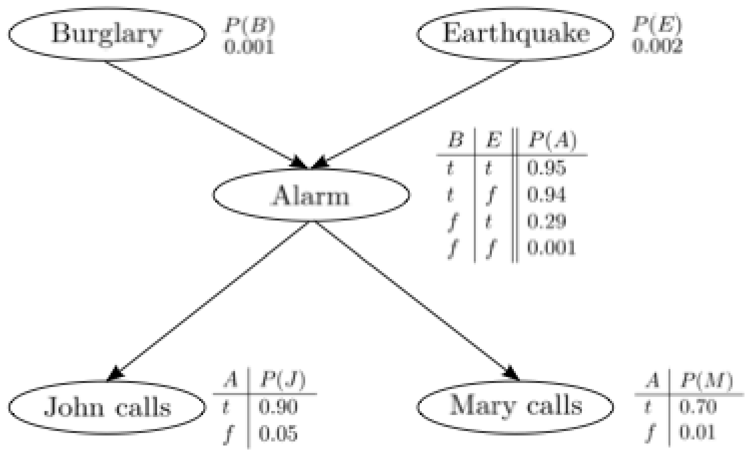
Bayes Theorem

Is based on the product rule for probabilities and helps to relate conditional probabilities to each other: $$P(U|W) = \frac{P(W|U)P(U)}{P(W)} = \alpha P(W|U)P(U)$$ It helps to express one kind of conditional probability by another one, for example a diagnostic rule by a causal one, which is then much easier to update in case of change in the prior. The following example is very illustrative I think: We could come up with a **diagnostic knowledge** base with the probability of having Meningitis (M) when the symptom is a stiff neck (S). This rule P(M | S) can simply be constructed from the hospital records. But how could we update this if the prior (base rate of meningitis cases) changes in case of an epidemic? Using bayes theorem makes it easy to model the diagnostic rule by a **causal knowledge base** which is easy to update P(M | S) = P(S | M) P(M).

bayesian networks

A bayesian network is a graphical model using a directed acyclic graph to encode the factorization of a joint distribution into a collection of conditional independence statements. Each node corresponds to a random variable (annotated with its conditional independence) while directed links between the nodes model the causal relationships between the variables (model topology). When the nodes of the graph are *well ordered* (index of a node x_i should always be larger than the indices of all of its parents) the causal flow is represented in the structure of the graph. The previous joint probability is now efficiently factorized into conditional probabilities $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | parents(x_i))$. This is an efficient representation of knowledge about causal relationships between variables, although unfortunately the model is **not efficient for inference.** A nice illustration of this principle is given by the *californian example* and also inference can easily be performed on it, although not very efficient. Inference algorithm:

- fill in the evidence
- marginalize over the rest
- normalize outcomes to sum up to 1



B	E	P(A)
t	t	0.95
t	f	0.94
f	t	0.29
f	f	0.001

A	P(J)
t	0.90
f	0.05

A	P(M)
t	0.70
f	0.01

$$P(J, M, A, B, E) = P(J|A)P(M|A)P(A|B, E)P(B)P(E)$$

Inference in bayesian networks

Unfortunately the straightforward to construct and memory wise efficient representation as a directed acyclic graph is not good for inference. Therefore the DAG has to be transformed into its second representation that allows efficient inference. To give a concise description of the graph transformation, we need the following definitions:

- **complete graph:** every pair of nodes is connected by an edge
- **separator:** Three sets of nodes A, B, C are part of a graph and not necessarily disjoint. C separates A and B, if every path from an arbitrary node from A(x1 ∊ A) to an arbitrary node from B(xn ∊ B) passes through at least one node from C. A separator is a node which is a hub between two other graphs, i.e. all connections go through the separator.
- **proper decomposition:** A, B, C are subsets of nodes of a given graph which are non-empty and disjoint. It is a proper decomposition if C is a complete graph that separates the sets A and B.
- **decomposable graph:** is a graph which is either complete or there exists a proper decomposition A, B, C such that both subgraphs are proper decomposable. Such a graph represents a valid probability distribution.
- **clique:** maximally complete subgraphs

So the second representation of a directed acyclic graph is a **undirected decomposable graph** which is a decomposition into maximally complete subgraphs. ⇒

1. A is conditionally independent of B given C ⇐>A, B, C is a proper decomposition of the graph G
2. $$P(\mathbf{x}) = \frac{\prod_{\text{cliques } C} P_C(\mathbf{x}_C)}{\prod_{\text{separators } S} P_S(\mathbf{x}_S)} \qquad \begin{array}{c}\text{decomposable graph}\\ \longleftrightarrow \text{ factorization into}\\ \text{marginal distributions}\end{array}$$

A decomposition of a graph into its cliques and separators, corresponds to the statement that 2 cliques are conditionally independent from each other, given their separator. Such a factorization is useful for calculating marginals and for inference. The complexity of a full marginalization is O(2**n) with n the cardinality of the largest clique. In general potentials (unnormalized probability distributions) are used instead of probabilities and the results are later normalized.

$$P(\mathbf{x}) \sim \frac{\prod_{\text{cliques } C} \psi_C(\mathbf{x}_C)}{\prod_{\text{separators } S} \psi_S(\mathbf{x}_S)} \qquad \begin{array}{c}\text{not unique!}\\ \text{shifting factors between}\\ \text{numerator and denominator}\\ \rightsquigarrow \text{will become important for inference}\end{array}$$

How to create such a junction tree

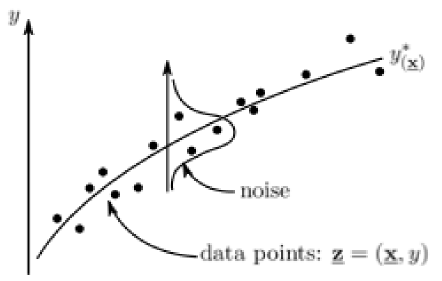
1. definition of the random variables and construction of the bayesian network (DAG)
    a. create the topology of the DAG by analysis of the causal relationships (structure learning) or by the analysis of an expert
    b. topological sorting → simple causal flow
    c. annotation of the nodes with conditional probabilities (expert knowledge, extraction of fractions from a database)
2. transformation of the DAG into the inference machineary
    a. construction of the related moral graph
        i. connect all parents of a node with undirected links
        ii. replace all directed links with undirected links
    b. construction of a undirected decomposable graph
        i. add undirected links such that all cycles of length four and larger contain a chord → chordal graph is always decomposable

    ii. chordal graph is not unique and the construction of a chordal graph with cliques of minimal size is NP hard (done by triangulation algorithm)
  c. construction of a related junction tree
    i. The junction tree is constructed by taking the cliques and separators of the chordal graph, and thus turns the graph into a chain.
    ii. nodes of the junction tree are maximal cliques of the decomposable graph
    iii. links of the junction tree: connect neighboring maximal cliques annotated by the corresponding separator.
3. inference
  a. initialize the clique potentials with the corresponding probability of the according directed graph, i.e. a clique involving nodes A,B,C is initialized with a probability which includes all of the nodes. Note that each probability which is known from the database is only used for one clique. Obviously, such an initialization is not unique! The separators can be initialized by 1, as they are just normalization factors and the cliques are initialized with probabilities per se.
  b. modification of clique potentials by observed evidence (not just when calculating the prior)
  c. message passing (belief propagation)
  d. final marginalization with relevant clique

generative models

We are given a set of observations $\underline{\mathbf{z}}^{(\alpha)} = (\underline{\mathbf{x}}^{(\alpha)} , \underline{\mathbf{y}}_T^{(\alpha)}), \alpha = 1, \ldots, p$ from an underlying distribution P(z) = P(y | x) P(x). Previously (in the first part of the script) we used parametrized classes of *deterministic* predictors y(x,w) and the inference was based on a selected (optimal) predictor y(x, w*). This deterministic predictor accounts (should account) for the real structure of the data which is usually hidden by overlayed noise. A **generative model** in contrast tries to give a complete description of the data generation process, modelled as a conditional probability p(x |x, w). This means that a model of the noise is included in the predictor. This means that the generative model can also be used to generate predictions or *fantasies*.

$$y_{(\underline{\mathbf{x}})} = \underbrace{\widehat{y}_{(\underline{\mathbf{x}};\underline{\mathbf{w}})}}_{\substack{\text{model of a deterministic} \\ \text{relationship} \\ \leadsto\text{parametrized function} \\ \leadsto\text{e.g. neural network}}} + \underbrace{\widehat{\eta} \text{ and } \widehat{p}_{(\widehat{\eta};\underline{\sigma})}}_{\substack{\text{model of the noise} \\ \leadsto\text{here: additive noise} \\ \text{(could also be multiplicative)} \\ \leadsto\text{here: parametrized distribution}}}$$


y
$y^*_{(\underline{\mathbf{x}})}$
noise
data points: $\underline{\mathbf{z}} = (\underline{\mathbf{x}}, y)$

To sample from a conditional probability p(y|x), the y and x would be set to a certain value and then with the resulting probability the point at (x, y) would be drawn.

Bayesian Learning

Already when *degrees of belief* were introduced we talked about cause, effect, causal rules and diagnostic rules. I want make a few more comments on this notions to make the relation to the current section more clear.

- **cause:** is the model hypothesis; the goal is to model the influence of some variables on the output → the effect
- **effect:** this is the observed evidence; the output of the underlying system and the interactions of its variables
- **causal rule:** this is the rule that describes which cause generates which effect; it models the effect of a cause. The previously introduced **generative model** can be seen as such a causal rule as it relates the input variables to the distribution of the output. (P(S|M) in the meningitis example → when someone has meningitis, what is the probability that it *causes* a stiff neck?)
- **diagnostic rule:** is what we use for evaluating the given evidence: which cause generated the observed evidence. The diagnostic rule is the *model evaluation* that finds the underlying structure of the observations. (P(M|S) in the meningitis examples → you try to diagnose meningitis, what is the probability of having meningitis when someone has a stiff neck.

The biggest difference between the bayesian and the frequentist method is that it not only estimates a parameter, but computes the probability distribution over this parameter. Furthermore it gives a nice and formal way to incorporate knowledge in form of the prior for models or its parameters. In practice (outside some distribution with special properties - from the exponential family) computation of the posterior distribution via bayes theorem is often computationally very challenging or simply not feasible. To deal with this problem, the posterior is assumed to have a maximum and the **maximum a posteriori (MAP)** method is used to compute a point estimate of the parameter with the highest posterior, without computing the whole distribution. When no prior is specified, this method is called **maximum likelihood (ML)**. All these methods can be used either to learn the parameters of a specific model or to select the best model.

$$P(\theta_i|\mathcal{D},\mathcal{M}_i) = \frac{P(\mathcal{D}|\theta_i,\mathcal{M}_i)P(\theta_i|\mathcal{M}_i)}{P(\mathcal{D}|\mathcal{M}_i)}; \quad P(\mathcal{D}|\mathcal{M}_i) = \int d\theta_i \, P(\mathcal{D}|\theta_i,\mathcal{M}_i)P(\theta|\mathcal{M}_i)$$

Parameter learning: $P(\mathcal{D}|\mathcal{M}_i)$ is called the marginal likelihood or evidence for $\mathcal{M}_i$. It is proportional to the posterior probability model $\mathcal{M}_i$ being the one that generated the data.

Model selection:

$$P(\mathcal{M}_i|\mathcal{D}) = \frac{P(\mathcal{D}|\mathcal{M}_i)P(\mathcal{M}_i)}{P(\mathcal{D})}$$

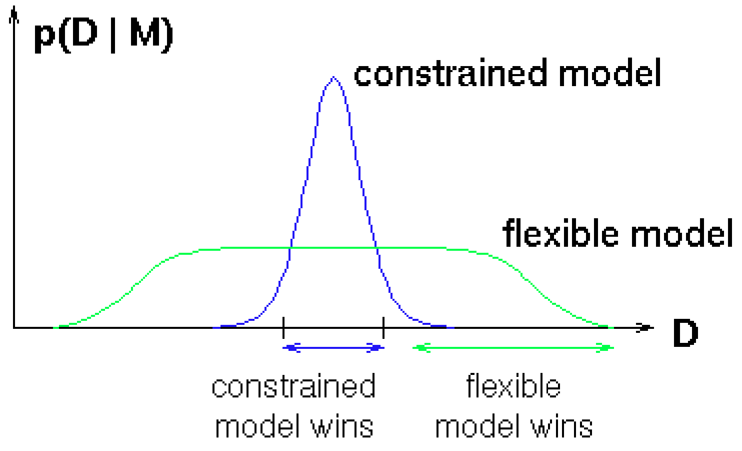
To make predictions on unseen data we can now either choose the model with the highest posterior or keep a whole ensemble of models and let them all vote for a certain result, weighted by their posterior. For the selection process we are given a set of disjunct models {M_i} and some random observed evidence E. The likelihood P(E | M_i) that model M_i has generated the data can be computed via a generative model ⇒ product over data points to compute how likely it is that the model M_i produced the given evidence E. To compute the likelihood of a whole model we need to integrate over all its parameters, where we run into 2 major problems.

$$p(D|M) = \int_\theta p(D|\theta)p(\theta|M)d\theta$$

1. How to choose the correct prior over the parameters: There are several approaches

   a. choose the least informative prior (maximum entropy) with some constraints → regularization
   b. use Jeffrey's Prior (based on fisher information)
   c. ect..
2. integration over all parameters is very expensive ⇒ MAP assumption that prior over the parameters is peaky which allows to simply use the parameter with the highest likelihood.

The computation of this evidence gives us the *simplest* model which is most likely to have generated the given data. Why a simpler model is preferred becomes obvious from the following illustration:



this is why bayesian model selection automatically selects the simpler (more constrained model) when it is equally supported by the data. In the above plot, when all the data we have is located in the area below the blue curve, the more constrained model wins because it has more of its probability mass in this volume.

How to make predictions by a committee of models → compute weighted average:

$$\text{observations } E \xrightarrow{\substack{\text{fundamental problem}\\ \text{of prediction}}} \substack{\text{degree of belief } P_{(e|E)}\\ \text{into a new event } e}$$

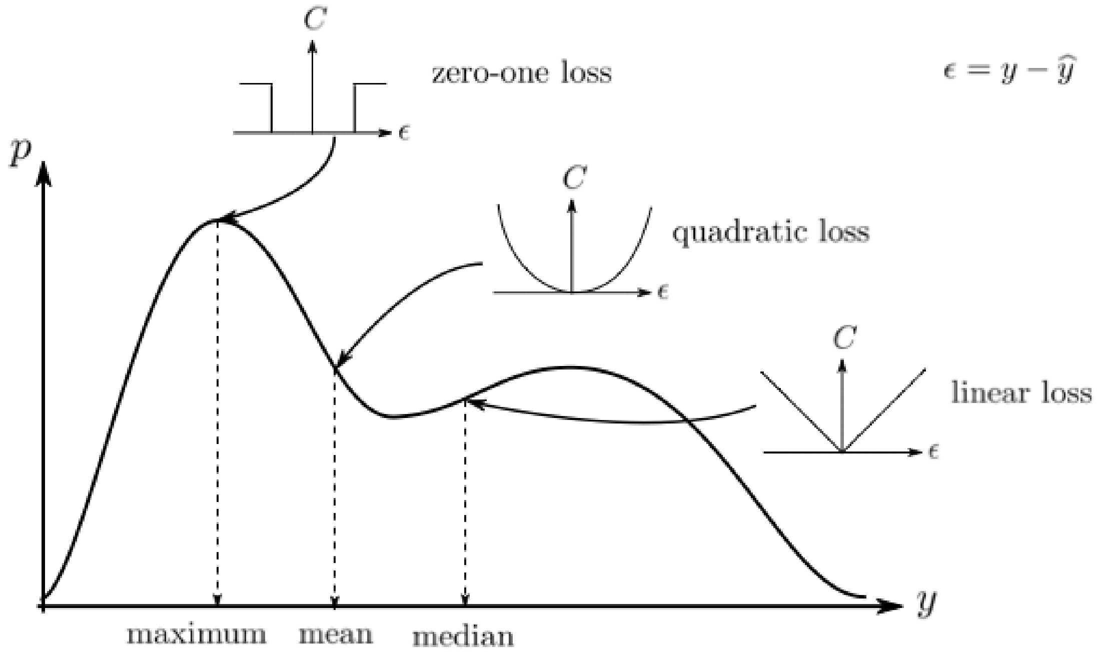
$$\begin{aligned} P_{(e|E)} &= \sum_i P_{(e,M_i|E)} && \text{marginalization} \\ &= \sum_i P_{(e|M_i,E)} P_{(M_i|E)} && \text{def. of conditional probability} \\ &\overset{!}{=} \sum_i P_{(e|M_i)} P_{(M_i|E)} && \text{conditional independence assumption} \end{aligned}$$

This prediction is only correct if the true model is in the set of the assumed models.

To improve the predictions made by the model we choose (in case of model selection, no committee) we can furthermore assign a loss function to the problem This function influences the prediction to not only choose the most probable new event, but to also incorporate consequences of the error made for a certain value. Different loss functions can be chosen and then its value is minimized according to the following function which weights the loss according to the probability of its occurrence:

$$\widehat{y}_{(\underline{\mathbf{x}})} = \underset{\bar{y}}{\operatorname{argmin}} \int dy C_{(y,\bar{y})} P_{(y|x;Y,X)}$$

The choice of the loss function strongly influences which model we finally choose according to their posteriors.



A few comments on the weight decay example for MLPs in the Oby script:

- noise model chosen from the form of the training error?
- likelihood of a parameter depends on the training error
- a prior can be constructed via the maximum entropy method under constraints (from prior knowledge). The optimization is done via lagrangian multipliers and the constraints can be incorporated e.g. via a regularization term.
- There is prior knowledge also already involved via:
    - the choice of the model class
    - the choice of the noise model
- the more data-points the less important the prior

More on Bayesian model selection from a paper I found

Paper: Bayesian Model Selection and Model Averaging by Larry Wasserman

To decide which of two models is better in describing a set of given data, the bayes factor between the two models, given the data can be computed. This factor is a measure of evidence for M_i vs. M_j. When assuming equal probability for both models (flat prior) this bayes factor is given by:

The "Bayes factor" for $\mathcal{M}_i$ versus $\mathcal{M}_j$ is defined to be

$$B_{ij} = \frac{Pr(\mathcal{M}_i|Y^n = y^n)}{Pr(\mathcal{M}_j|Y^n = y^n)} = \frac{m_i}{m_j} = \frac{\int \mathcal{L}_i(\theta_i)p_i(\theta_i)d\theta_i}{\int \mathcal{L}_j(\theta_j)p_j(\theta_j)d\theta_j}.$$

The whole model comparison thingy includes the following ingredients:

- a set of models that each can be parametrized
- a likelihood function for each model → computed by integrating (marginalizing) over the parameters of the model
- prior for each model (1/k in the paper)
- prior over parameters for each model

A single posterior is computed by:

$$Pr(\mathcal{M}_j|Y^n = y^n) = \frac{p(y^n|\mathcal{M}_j)Pr(\mathcal{M}_j)}{\sum_r p(y^n|\mathcal{M}_r)Pr(\mathcal{M}_r)}.$$

The computation of the Bayes factor involves two major problems:

1. prior for the parameters is not known and a wrong choice for the prior renders the Bayes factor meaningless ⇒ can be solved by using some special priors
2. integration over all parameters is very expensive ⇒ use maximum likelihood and the peaky distribution assumption

MI 2

Preliminaries

First we have to introduce random variables on which we can apply density estimation. Random variables can be discrete or continuous and can assume values from their *domain* with certain probabilities, assigned to a value in the discrete case or a volume from the domain space for the continuous case. For the discrete case, probability values for certain values/events are between 0 and 1 and the sum over the probabilities over all events has to yield 1. The same normalization is valid for continuous variables, the integral over the whole space has to yield one. The difference is that the probability density can be larger than 1 for a certain value of the domain. The probability is the integral of the density function over a volume.

Probability density estimation

Density estimation is a typical example for inductive learning, the probability density of a random variable should be estimated from only a few observations, which are drawn iid from the underlying distribution.

Kernel density estimation (KDE)

This is a very simple non-parametric method which is based on a convolution of the observations with a Kernel function. This basically places a kernel above each observation and sums them up → weighted sum of data points. When a simple rectangular kernel is chosen, KDE yields a histogram. The Kernel function is usually parametrized by a variable h for its bandwidth. The choice of this hyperparameter is crucial for approximation of the true underlying density. h too small → high variance of the estimator, h too large → high bias. The correct hyperparameter can be found via validation methods. There are also several approaches to analytically calculate the true bandwidth.

Parameterized density estimation

This method creates a generative model of a data source, by learning the (locally-) optimal parameters of a certain model. There are several ways to derive the same method and I'll show two here. The first is an ERM version of the minimization of the Kullbach-Leibler divergence (D_kl) between the model and the true density and the second is using the maximum likelihood method.
The first method is based on the idea of minimizing the distance between the true density and its approximation by the model. This distance is measured by the D_kl which is a kind of metric between two densities. $\mathrm{D_{KL}} = \int d\underline{\mathbf{x}} P(\underline{\mathbf{x}}) \ln \frac{P(\underline{\mathbf{x}})}{\widehat{P}(\underline{\mathbf{x}};\underline{\mathbf{w}})}$ We then split the integral into two parts by the quotient rule for logarithms and remove the part of the term which does not depend on the parameter w. This shows that minimizing D_kl is equivalent to minimizing the negative cross entropy between the two distributions: $-\int d\underline{\mathbf{x}} P(\underline{\mathbf{x}}) \ln \widehat{P}(\underline{\mathbf{x}};\underline{\mathbf{w}})$. Because P(x) is not known, otherwise we probably wouldn't do all this stuff, we approximate the above expression by an empirical version. This is the typical ERM step of approximating the generalization error by the empirical training error. Here the result: $E^T = -\frac{1}{p}\sum_{\alpha=1}^{p} \ln \widehat{P}\big((\underline{\mathbf{x}}^{(\alpha)};\underline{\mathbf{w}}\big) \overset{!}{=} \min_{(\underline{\mathbf{w}})}$
The minimal w can be found via standard gradient methods, batch and online versions are available and also the validation can be done via normal validation methods (test set, cross-validation).
The second method starts by looking at the likelihood of the parameter w. How likely is it that a certain parameter w (and the model resulting from it) created the whole dataset that we observed? The likelihood for a certain dataset is given by:

$$\widehat{P}\big(\{\underline{\mathbf{x}}^{(\alpha)}\};\underline{\mathbf{w}}\big) = \prod_{\alpha=1}^{p} \widehat{P}\big(\underline{\mathbf{x}}^{(\alpha)};\underline{\mathbf{w}}\big)$$

Maximizing this likelihood w.r.t the parameter w is equivalent to minimizing the negative log-likelihood (log taken for computational reasons, the optimization is equivalent but derivatives of sums are much easier to compute than derivatives of products) and leads to the same result as the first method: $= -\sum_{\alpha=1}^{p} \ln \widehat{P}\big(\underline{\mathbf{x}}^{(\alpha)};\underline{\mathbf{w}}\big)$
Minimizing the entropy the dataset with respect to the model makes sense when thinking about it. We want the model to assign high probability to data-points that are frequent and very low probability to data-points that hardly ever appear.

maximum likelihood and estimation theory

Estimation theory deals with the existence and quality of an estimator of the parameters of a model. If the true model is a member of the model class, the true distribution can be written as:

$P\big(\{\underline{\mathbf{x}}^{(\alpha)}\};\ \underbrace{\underline{\mathbf{w}}^*}_{\substack{\text{true}\\ \text{parameter}\\ \text{value}}}\ \big) \equiv P$ and model selection would then mean to select this true parameter w* from the observed data estimator w_hat. This estimator for a given dataset $\underline{\widehat{\mathbf{w}}} = \underline{\widehat{\mathbf{w}}}\big(\{\underline{\mathbf{x}}^{(\alpha)}\}\big)$ is a random variable because the x_alpha are random variables. In MI 1 we discussed the bias - variance tradeoff which gives us bias and variance as quality criteria of an estimator. An optimal estimator would be one with no bias (only possible if true model within model class) and minimum variance (smallest deviations from w*). The lower bound of the variance of an unbiased estimator is given by the Cramer-Rao bound, which is the inverse of the fisher information matrix, evaluated at the true parameter w*. It gives the best/lowest variance we can get for a certain dataset and model. Given this measure of the CR-bound, two new definitions arise:

- efficient estimator: b = 0 and variance = CR-bound
- unbiased minimum variance estimator: b = 0 and difference between variance and CR_bound is minimal.

both estimators might not exist or might be hard to find/compute.
Using this results for the maximum likelihood estimator shows that the ML estimator is asymptotically (for p → infty) efficient: $\underline{\widehat{\mathbf{w}}} \sim \mathcal{N}\big(\underline{\mathbf{w}}^*, \underline{\mathbf{M}}^{-1}_{(\underline{\mathbf{w}}^*)}\big)^{\substack{\text{asymptotically}\\ \text{Gaussian}\\ \text{distributed}}}$
For a finite number of observations the ML-estimator is efficient if an efficient estimator exists.

Two more sentences on consistency that I found in another paper:
Under weak assumptions, the MLE estimator has many useful properties. First, the MLE is consistent, meaning that theta* converges to the true value of theta with probability 1. Second it has, asymptotically in sample size, a normal distribution.

fisher information matrix

The fisher information gives the amount of information that an observed random variable x carries about an unknown parameter theta and is useful to evaluate data representations (by a certain model). The likelihood function $f(X;\theta)$ describes the probability that we observe a given sample *x given* a known value of *θ*. If *f* is sharply peaked with respect to changes in θ, it is easy to intuit the "correct" value of θ given the data, and hence the data contains a lot of information about the parameter. If the likelihood *f* is flat and spread-out, then it would take many, many samples of *X* to estimate the actual "true" value of *θ*. Therefore, we would intuit that the data contain much less information about the parameter.

projection methods

The goal of projection methods can be summarized by one sentence: find interesting directions (informative features) in the data. These features can be used for dimensionality reduction for pre-processing (decorrelation, whitening) and visualization or for finding different sources in a signal → unmixing.

PCA

PCA is a quite simple but useful method, often used as a preprocessing step in for supervised methods (whitening, dimensionality reduction), for compression or visualization. It finds directions in the features space along which the data has maximum variance. It works on centered data, the dimensions should be comparable and it is unfortunately susceptible to outliers. Some definition before the derivation of the algorithm:

- e_x: are basis vectors in the original feature space (dimensions of measurements)
- e_a: basis vectors with interesting directions in feature space.
- both basis vectors are normalized to 1
- feature value: v_a = e_a**T x (projection on feature space basis vector)

The covariance matrix of the data is given by: $C_{ij} = \frac{1}{p}\sum_{\alpha=1}^{p} \mathrm{x}_i^{(\alpha)}\mathrm{x}_j^{(\alpha)}$ where entries with i = j give the variance along elementary features and entries i != j the covariances between variables. A few steps of calculation show that the moments of the data along the complex features are given by:

- transformed mean: m_a = e_a**T m
- transformed variance: $\sigma_a^2 = \underline{\mathbf{e}}_a^T \underline{\underline{\mathbf{C}}}\underline{\mathbf{e}}_a$

These two results lead to a result on which the whole PCA idea is based: The covariance matrix determines the variance of the data along every possible direction.

And from there we come back to the beginning by asking, how can we formally define the *interesting directions* we are looking for ⇒ directions in feature space along which the variance is maximal!
This leads to the following formulation of our problem:

$$\boxed{\sigma_a^2 = \max\left(\underline{\mathbf{e}}_a\right)}$$

optimisation
under constraints

$$\boxed{\underline{\mathbf{e}}_a^2 = 1}$$

Which is a simple convex optimization under linear equality constraints that we of course solve by a Lagrangian with the following formulation:

$$\underbrace{\underline{\mathbf{e}}_a^T \underline{\underline{\mathbf{C}}}\underline{\mathbf{e}}_a}_{\text{objective}} - \underbrace{\lambda}_{\substack{\text{Lagrange}\\ \text{multiplier}}} \underbrace{\left(\underline{\mathbf{e}}_a^2 - 1\right)}_{\text{constraints}} \overset{!}{=} \max$$

Setting the derivative to 0, etc.. leads to the eigenvalue problem that is known as PCA:

$$\boxed{\underline{\underline{\mathbf{C}}}\underline{\mathbf{e}}_a = \lambda \underline{\mathbf{e}}_a}$$

- e_a is an eigenvector of the covariance matrix
- lambda its eigenvalue
- e_a**2 == 1 can be fulfilled by normalization
- the eigenvalues give the variances along principal component e_a
- the eigenvectors form an orthonormal basis
- the principal components are uncorrelated
- can be used for optimal dimensionality reduction to subspace R**M with M < N → mean squared approximation error in representing the observations by components with largest eigenvalues is minimal
- scree plots of eigenvalue spectrum might give a reasonable value for dimensionality reduction → identify the # of relevant dimensions

- pca is often used as a whitening transform → scale variance along all directions to one. and remove covariances. By this transform the covariance becomes a diagonal matrix with the variances of the principal components on the diagonal.

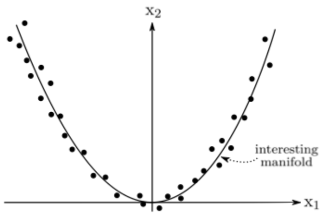
hebbian learning for linear neurons and relation to PCA (on-line PCA)

Interestingly a simple linear perceptron when trained using the hebbian learning rule computes the first principal component of the provided data set. There are even algorithms for small single layer perceptron networks that are capable of doing a real PCA with not only the first component. The algorithm is simply:

```
initialize weight vector to small values
loop
    chose a random data point x**alpha
    delta w_i = epsilon * y(x,w) * x_i**alpha
end
```

In the learning step of the algorithm (delta w) weights increase if input and output of the perceptron are correlated and therefore the weight vector will converge to the direction of largest variance which is the first principal component. It can be used as an online PCA with adaptive tracking of the direction of largest variance. The formulation of a batch version of the algorithm gives insight into the result of the procedure. Averaging over all patterns with a small learning rate epsilon gives: $\Delta \underline{\mathbf{w}} = \varepsilon \underline{\mathbf{C}\mathbf{w}}$ which is an analysis of the correlation matrix of the data. But there is a problem with this method when pure Hebbian Learning is used. The magnitude of the weight vector approaches infinity. This can be avoided when Oja's Rule (variant of hebbian learning) is used for the weight updates. The weight will still converge to the first PC but its magnitude will be normalized to 1. In simple words, this is achieved by relative depression of all weights if the value of one is increased.

Kernel PCA

Kernel PCA is basically a normal PCA in a feature space (which does not have to be computed explicitly). It is useful when the *interesting feature* is a nonlinear combination of elementary features. 

So the idea is to transform the data to a space where such a nonlinear feature would be found by PCA. This could for example be applied to image data where the interesting structure is hidden in correlation between pixels. The data is transformed by a certain feature mapping $\underline{\phi} : \underline{\mathbf{x}} \rightarrow \underline{\phi}_{(\mathbf{x})}$ from R**N to R**M with M usually >> N. But looking at all variations of higher moments might result in an extremely high dimensional space and this is why usually the, tadaaaaaaaa: Kernel Trick is applied. But to apply this famous and useful trick, a few preparations have to be done. Similar as the weights in the SVM derivation, this time the eigenvectors are expanded in a linear combination of data points. This is always possible because the PCs lie in the subspace of the data. $\underline{\mathbf{e}}_k = \sum_{\beta=1}^{p} a_k^{(\beta)} \underline{\mathbf{x}}^{(\beta)}$ Insertion of this expansion into the eigenvalue equation $\underline{\mathbf{C}}\underline{\mathbf{e}}_k = \lambda_k \underline{\mathbf{e}}_k$ leads to $\underline{\mathbf{K}}^2 \underline{\mathbf{a}}_k = p\lambda_k \underline{\mathbf{K}}\underline{\mathbf{a}}_k$ Because K is positive semi-definite this can be reduced to the final reformulation of the eigenvalue problem $\boxed{\underline{\mathbf{K}}\underline{\mathbf{a}}_k = p\lambda_k \underline{\mathbf{a}}_k}$with a_k the principal component represented in the (probably overcomplete) basis given by the data points {x_alpha}. The a_k form a vector in the space spanned by {x_alpha} for alpha = 1,...,p. But this principal components have to be normalized to 1 in order to fulfill the constraint for the previous optimization (lagrangian) that lead to the formulation as an eigenvalue problem $\underline{\mathbf{a}}_k^{\text{norm.}} = \frac{1}{\sqrt{p}\sqrt{\lambda_k}|\underline{\mathbf{a}_\mathbf{k}}|} \underline{\mathbf{a}}_k$. The final **projection** of the data points (or new data) on the principal components is then given by:

$$
\begin{aligned}
u_k &= \underline{\mathbf{e}}_k^T \cdot \underline{\mathbf{x}} \\
&= \sum_{\beta=1}^{p} a_k^{(\beta)} \underbrace{\left[\left(\underline{\mathbf{x}}^{(\beta)} \right)^T \cdot \underline{\mathbf{x}} \right]}_{\text{scalar product}}
\end{aligned}
$$

.

Now we are at a stage where everything is expressed in terms of dot products which allows to apply the kernel trick to avoid explicit transformation into the feature space. One assumption for the original PCA was that the data was scaled to zero mean. This cannot be guaranteed to still be true after the transformation, but can be accounted for in an additional normalization step. The whole K-PCA method can be summarized in a few steps:

1. calculate the kernel matrix and normalize it to zero mean (in feature space)
2. solve the eigenvalue problem
3. normalize eigenvalues to unit length

4calculate projections of (new) data points x_delta onto eigenvectors

$$u_{k(\underline{\mathbf{x}}^\delta)} = \sum_{\beta=1}^{p} a_k^{(\beta)} K_{\beta\delta} \leftarrow \begin{array}{c}\text{use normalized}\\\text{matrix element!}\end{array}$$

And a few comments on that procedure:

- the # of principal components can exceed the # of dimensions in the original space
- the expansion of PCs into data-points is **not** sparse ⇒ the projection is computationally expensive. One approach to deal with this problem is to approximate the PCs by a subset of the data-points or by using prototypical data-points (centers of clusters). Approximation by $\widehat{\underline{\mathbf{e}}}_k = \sum_{\gamma=1}^{q} \widehat{a}_k^{(\gamma)} \underline{\phi} \underbrace{\left(\underline{\mathbf{z}}^{(\gamma)}\right)}_{\substack{\text{new data}\\\text{point}}}$ with q < p. components of the approximated eigenvectors should be chosen to minimize the squared error $\varphi = \left(\underline{\mathbf{e}}_k - \widehat{\underline{\mathbf{e}}}_k\right)^2$ which can be done by standard gradient based methods.

the ica problem

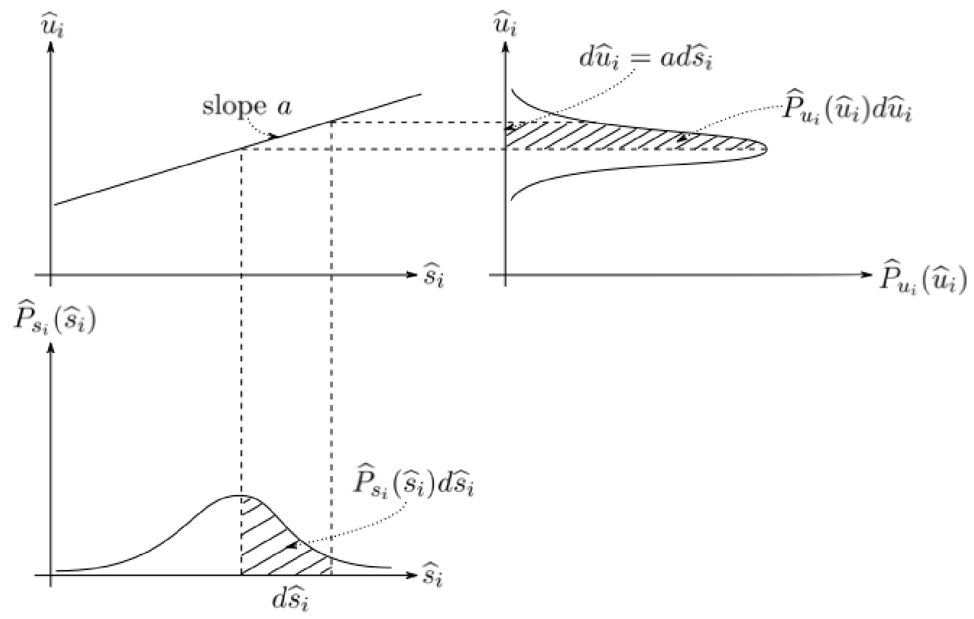
ICA is a family of algorithms that are usually used to solve *blind source separation* problems without prior knowledge on the mixing matrix (A) or the source signals (s), hence the blind in source separation. The only input are observations (x) which are assumed to be generated by noise free mixing x = As. To achieve this magical unmixing, the input sources are assumed to be **independent** and having a **non-gaussian distribution**. There are efficient algorithms which solve this problem up to a few theoretical limitations. The original amplitude and the correct order of the signals can not be recovered from the observations without further knowledge. The independence assumption can be stated as: $P_{\underline{\hat{s}}}(\underline{\hat{s}}) = \prod_{i=1}^{N} P_{s_i}(\hat{s}_i)$
The result of the algorithm are statistically independent components. Note that independence is a much stronger statement than mere decorrelation achieved by linear methods.

The infomax approach to ICA

The infomax approach uses an information theoretic objective function which maximizes the mutual information between input and output of an artificial neural network with nonlinear transfer functions. These nonlinearities allow to find independent components instead of mere decorrelation like in a linear network. The objective function we maximize in the end is the entropy of the output variables H(Y) and why maximizing this value can be derived in several ways. I show the version of Bell and Sejnowski (BS) because it is very short and concise and I don’t get the point in obermayer’s derivation where he assumes the transformed variables in the objective function (independence assumption in K_dl) to be uniformly distributed although this is what we are optimizing for in the end. BS simply state the mutual information that we want to maximize I(X, Y) - the information that output Y contains about input X - and its relation to individual and conditional entropy: $I(Y, X) = H(Y) - H(Y|X)$ Because H(Y | X) is constant in the case of noise free mixing (input X completely determines the output Y) it is sufficient to maximize the entropy of the output signal Y → max H(Y). Maximizing the information (entropy) contained in the output layer involves maximizing the entropy of the individual units while minimizing the mutual information between them (making them independent): $H(y_1, y_2) = H(y_1) + H(y_2) - I(y_1, y_2)$ The maximum entropy of the individual components is achieved by calibrating the network nonlinearities to the detailed higher-order moments of the input density function.

Derivation

BS start their derivation for a single perceptron for which the outputs entropy should be maximized. When the input is transformed by a monotonic function (the perceptron transfer function - sigmoid), the output pdf is given as a function of the input pdf. This scaling is to conserve the probability mass and is nicely depicted by the following graph form oby’s script.



$\widehat{u}_i$
slope a
$\widehat{s}_i$
$\widehat{u}_i$
$d\widehat{u}_i = ad\widehat{s}_i$
$\widehat{P}_{u_i}(\widehat{u}_i)d\widehat{u}_i$
$\widehat{P}_{u_i}(\widehat{u}_i)$
$\widehat{P}_{s_i}(\widehat{s}_i)$
$\widehat{P}_{s_i}(\widehat{s}_i)d\widehat{s}_i$
$\widehat{s}_i$
$d\widehat{s}_i$

in the terminology of the BS paper they write formula (1) $f_y(y) = \frac{f_x(x)}{\partial y/\partial x}$ and mention that f_x(x) is approximated by the training set. The empirical entropy of the output (what we want to maximize) is given by (2) $H(y) = -E\left[\ln f_y(y)\right]$. Putting (1) into (2) leads to $H(y) = E\left[\ln \frac{\partial y}{\partial x}\right] - E\left[\ln f_x(x)\right]$ where the second term can be neglected in the optimzation as it does not depend on w. Calculating the derivative for this term leads to update rules for w and the bias w_0. BS also give intuitive explanations for what these update rules do:

- when the input pdf is peaky → the w_0 rule centers the sigmoid around the peak
- the w update rule scales the slope of the sigmoid to match the variance of f_x(x)

These two rules produce a output pdf close to a flat uniform with max entropy. One graph from the BS paper helps to understand the effect of this transformation (it spreads the area which contains most of the probability mass on a wider interval and condenses the sparsely populated area of the probability volume on a small interval. A linear transformation - like in the oby diagram - only makes a gaussian wider or narrower, but could not make it flat):

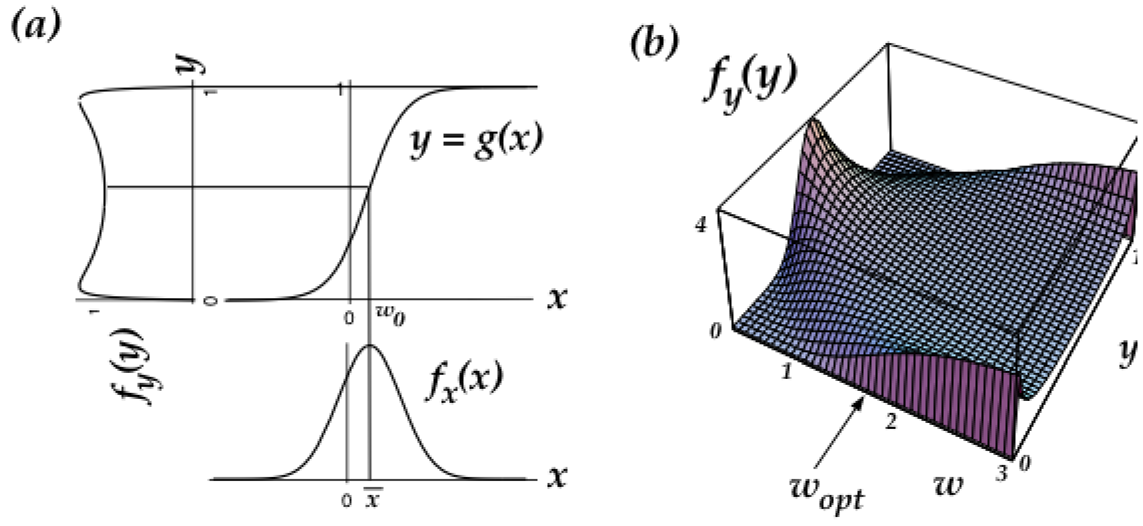


Figure 1: (a) Optimal information flow in sigmoidal neurons (Schraudolph et al 1992). Input x having density function $f_x(x)$, in this case a gaussian, is passed through a non-linear function $g(x)$. The information in the resulting density, $f_y(y)$ depends on matching the mean and variance of x to the threshold and slope of $g(x)$. In (b) $f_y(y)$ is plotted for different values of the weight w. The optimal weight, w_{opt} transmits most information.

The learning rules derived from maximizing entropy of the output might also make sense in a biological context where neurons must position the gain of their non-linearity to a level appropriate to the average value and size of input fluctuations. Maximum information transmission can be achieved when the sloping part of the sigmoid is optimally lined up with the high- density parts of the inputs.

The solution for the multidimensional case is derived in exactly the same way which leads to similar learning rules. When looking at the update rule for individual entries in the weight matrix, it becomes visible how the minimization of mutual information between output variables (making them independent) emerges from the update rule. $\Delta w_{ij} \propto \frac{\text{cof } w_{ij}}{\det \mathbf{W}} + x_j(1 - 2y_i)$ The determinant in the denominator takes care that not all neurons learn the same. When weights become too similar, W becomes degenerate, the determinant becomes close to 0 and this will introduce a larger change in the weight update factor. Obermayer mentions problems with the BS update rule because the matrix inversion in the update rule is computationally expensive. This computation can be sped up by using the natural gradient method which gives the direction of steepest ascent with a normalized step size and avoids the matrix inversion. And a few more comments from the script:

- sigmoid transfer function is chosen because a peaky input distribution is assumed
- ICA is also robust against the choice of the wrong transfer function
- true independent source signals are always a fixpoint of the algorithm, independent of choice of the transfer function
- if in doubt and enough data available, use parameterized approach for the transfer function

cost function based source separation

The previous ICA approach (infomax) used the assumption of **no noise** in the mixing process, and the following approach is designed to work with noisy observations. One problem for this is that the estimation of higher order moments (e.g. kurtosis) is often unreliable for noisy data. But the estimation of second order moments (correlation) is easy also for noisy data, although additional knowledge about the source signals is necessary to do something interesting and noise robust with the observations.

⇒ for many signals (audio, image, video) it is ok/reasonable to assume *finite length autocorrelations*

When finite length autocorrelations are assumed and the signals are uncorrelated ⇒ all cross correlation functions should vanish. Here an outline of this ICA approach:

1. Do PCA sphering as preprocessing. This step makes the dimensions to be equally scaled and leaves only one additional orthogonal transformation to be done in order to solve the ICA problem. $\underline{\mathbf{u}} = \underline{\mathbf{M}}_0\mathbf{x}$ M_0 is the sphering matrix and x the observations
2. do the remaining orthogonal transformation s = B u (the script shows here that the remaining transformation has to be orthogonal)
3. determine the orthogonal transformation B. This is done by diagonalization of a time-shifted cross-correlation matrix
   a. solve the eigenvalue problem:

$$\underline{\mathbf{C}}_u^{(\tau)}\underline{\mathbf{e}}_k = \lambda_k\underline{\mathbf{e}}_k \text{ with } \left[\underline{\mathbf{C}}_u^{(\tau)}\right]_{ij} = \left\langle u_{i(t)}u_{j(t-\tau)}\right\rangle_t$$

   b. transform into the eigenbasis: $\underline{\widehat{\mathbf{s}}} = \underbrace{\mathbf{E}_\tau}_{\text{matrix of eigenvectors}} \underline{\mathbf{u}}$

The joint diagonalization (QDIAG, FFDIAG) of of multiple cross correlation matrices gives a noise robust ICA algorithm.

Projection Pursuit

The last group of algorithms for performing ICA is based on the Central Limit Theorem which states that *the sum of random variables is more gaussian than the original variables*. So to solve the ICA problem we have to maximize the non-gaussianity of the output variables. Different measures exist for non-gaussianity:

1. kurtosis: For sphered data kurtosis is defined as $\text{kurt}(u) = \langle u^4\rangle_{P_u(u)} - 3$ and is > 0 for super gaussian data (e.g. exponentially distributed) and < 0 for sub gaussian data (bulky distribution, no outliers). The optimization problem is to maximize kurtosis while keeping the data sphered.
2. negentropy: is a theoretically well founded measure, but is computationally very

$$J_{(u)} := \underbrace{H_{(u)}^{\text{Gauss}}}_{\substack{\text{entropy of a Gaussian} \\ \text{distribution (with} \\ \text{some variance } \sigma^2)}} - \underbrace{H_{(u)}}_{\substack{\text{entropy of true} \\ \text{distribution} \\ (\text{variance } \sigma^2)}}$$

   expensive.

3. negentropy approximations. negentropy can be approximated by a weighted sum over some contrast functions.
    a. $$J_{(u)} \approx \sum_{i=1}^{l} \underbrace{k_i}_{\substack{\text{some}\\ \text{constant}}} \left\{ \left\langle G_{(u)} \right\rangle_{\underbrace{P_u(u)}_{\substack{\text{true}\\ \text{density}}}} - \left\langle G_{(u)} \right\rangle_{\underbrace{\text{Gauss}}_{\substack{\text{reference:}\\ \text{Gaussian}\\ \text{density with}\\ \text{some variance}}}} \right\}$$

        also the contrast functions should be indexed by i in the above formula. kurtosis could be used as a contrast function.

weight vector update rules can be derived from these objective functions

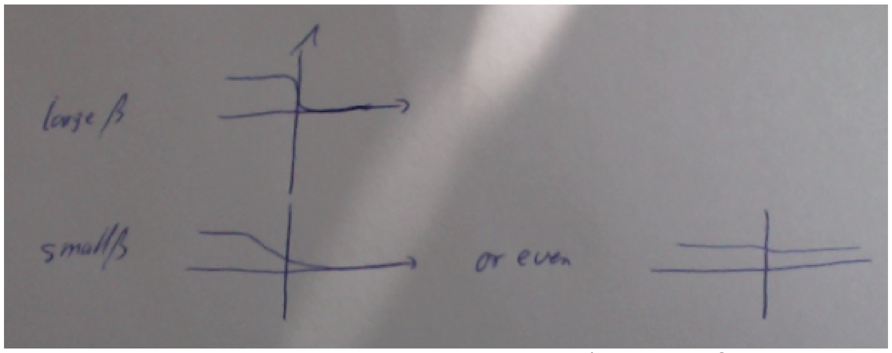
Stochastic Optimization with simulated annealing

Stochastic optimization can be used in huge and usually discrete search spaces. But how to do optimization in such a space, previously we mainly used gradient based methods which don't work in a discrete space. The idea is a method that starts as an almost random walk through the search space (exploration), which then develops a preference for the downhill direction and does deterministic steepest descent in the end. It is often used in statistical mechanics and the name comes in analogy to *natural optimization* for example during freezing or crystallization. In this cases optimization is done with respect to the energy level of a physical configuration which can be optimized by heating up some material with subsequent slow cooling. This is also why the parameter in simulated annealing is called the temperature. The temperature T (or mostly a noise parameter beta which is the inverse of the temperature) is decreased during the optimization process. The optimization is done according to the following algorithm:

```
initialize S_0, beta_0 and tao          beta_0 must be low in the beginning (high T)

t = 0
begin
       t ← t + 1
       choose a new state S                    often similar to old state
       calculate difference in cost delta_E = E_S - E_S_old
       keep S as new S_t with probability  $W_{(\underline{S}_{t-1} \to \underline{S})} = \frac{1}{1+\exp(\beta_t \Delta E)}$
       otherwise: keep the old state
       beta_t ← tao * beta_t-1
end
```

Here an illustration of the influence of beta:



- in the beginning almost random walk (change of error has a very small influence)
- after annealing (high beta, low T) decisions are only made according to the sign of delta_E ⇒ deterministic steepest descent
- the convergence to the global optimum is guaranteed for the annealing schedule beta_t = ln(t) which is very slow and in practice usually replaced by an exponential annealing schedule beta_t+1 = tao * beta_t with tao = 1.01, ..., 1.3
- this method is a very robust optimization procedure

Stochastic Optimization with Mean Field Annealing (MFA)

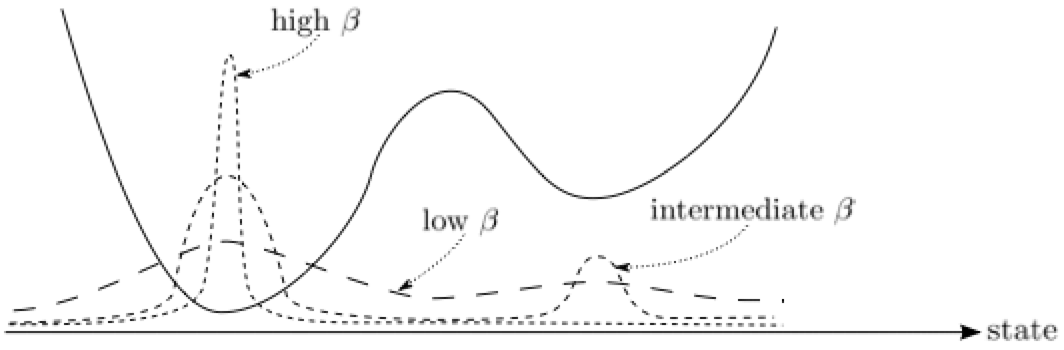
Gibbs Distribution

The Gibbs distribution is the basis of MFA as it gives a probability measure for the distribution of the states of the system → the probability of the system X being in state x. The simulated annealing model was stochastic and we had state changes also for a fixed temperature, but in the end we are interested in the states the system converges to. The idea of mean field annealing is to look at equilibrium points for given temperatures and this is also how the Gibbs distribution can be derived. Let **PI_s,t** be the probability distribution across states at a given time for a given temperature. Now lets see what happens to this distribution for a given

$$\Pi_{(\underline{S},t)} \to \underbrace{P_{(\underline{S})}}_{\text{stationary density}} \quad \text{for } t \to \infty \text{ (and constant } T, \beta)$$

temperature when t → infinity.

This stationary density can be calculated by looking at the *balanced situation* where the probability of being in S and changing to S' equals the probability of being in S' and changing to S. Formalized this can be written as $\overbrace{P_{(\underline{S})} W_{(\underline{S}\to\underline{S}')}}^{\text{probability of transition } \underline{S}\to\underline{S}'} \underbrace{=}_{\text{"determitted balances" condition}} \overbrace{P_{(\underline{S}')} W_{(\underline{S}'\to\underline{S})}}^{\text{probability of transition } \underline{S}'\to\underline{S}}$ and when we plug in the previous energy function it reduces to $\exp(\beta\Delta E)$ which is fulfilled for the Gibbs distribution $P_{(\underline{s})} = \frac{1}{Z}\exp(-\beta E)$ with normalization constant $Z = \sum_{\underline{S}} \exp(-\beta E)$



The solid line is the energy function and the dashed lines are the probabilities of certain states, depending on the noise parameter beta.

mean field annealing

Mean field annealing is used for stochastic optimization because exact algorithms like simulated annealing often have very slow convergence. What we would like to have is the Gibbs distribution P(S) of the system for different temperatures which gives for each state the probability that the system in equilibrium ends up in this particular state ⇒ We would like to evaluate P(S), but usually the maxima of P(S) are equally hard to obtain than the minima of the energy function. Also its moments (e.g. its mean) can usually not be calculated analytically and the solution to this problem is to approximate the Gibbs distribution by a computationally tractable approximation. Mean field annealing (MFA) is a method with an origin in statistical mechanics to analyse physical systems with multiple bodies and is a deterministic approximation of simulated annealing. A many-body system with interactions is generally very difficult to solve exactly. The n-body system is replaced by a 1-body problem with a chosen good external field (the mean behavior/field of the other particles). The external field replaces the interaction of all the other particles to the selected particle. In our case each body corresponds to one dimension in state space.

One comment on what this can be used for (and is used for later on in the script): Clustering is mainly about the assignment variables (assigning data-points to their prototypes/clusters)! The prototypes are only computed after the assignment and for pairwise clustering we don't even compute prototypes. It is really about assignment variables and they form a discrete (k-means) or continuous (soft-k-means) state space for which we want to know the *annealed state*).

Derivation: first we approximate the Gibbs distribution by a parametrized family of factorizing distributions

$$Q_{(\mathrm{S})} = \frac{1}{Z_Q} \exp\Big\{ -\beta \sum_k \underbrace{e_k}_{\text{parameters}} \mathrm{S}_k \Big\}$$

with e_k being the parameters (mean fields) we are optimizing over. Note that S_k is one dimension in the state space (the state of one assignment variable in the case of clustering). We can now define the means (with respect to each variable) according to the distribution Q, which are usually com putationally

$$\langle S_l \rangle_Q = \frac{\sum_l S_l \exp(-\beta e_l S_l)}{\sum_l \exp(-\beta e_l S_l)}$$

tractable. I think this formula has a typo and the sums should be over all states instead over the dimensions of the state space. To find the right parameters (mean fields) for this approximation we minimize the Kullbach-Leibler Distance between the

$$D_{KL} = \sum_{\underline{S}} Q_{(\underline{S})} \ln \frac{Q_{(\underline{S})}}{P_{(\underline{S})}} \overset{!}{=} \min$$

Gibbs distribution and its approximation. Computing its derivative with respect to the mean fields and setting it to zero gives a P(S) independent equation for the determination of the mean fields. The following deterministic algorithm results from the previous derivation:

```
loop:
      calculate the mean fields
      calculate the moments
      increase beta
```

for beta → infinity the mean <S_k> converges to its optimum

Clustering

Clustering is the unsupervised formation of categories that can be used to detect structure in the data, to compress the data or also for classification (whether a new point belongs to a detected cluster).

k-means

This is a very simple, but often used clustering technique. It is a form of *central* clustering which means that clusters are described by prototypes (in contrast to e.g. hierarchical clustering). It is based on the average euclidian distance between prototypes and observations. Definition of the problem:

- w_q are the prototypes
- m_q**alpha are the assignment variables, defined by the following formula and

$$m_q^{(\alpha)} = \begin{cases} 1, & \text{if } \underline{\mathbf{x}}^{(\alpha)} \text{ belongs to cluster } q \\ 0, & \text{else} \end{cases}$$

  normalized to 1: $\sum_q m_q^{(\alpha)} = 1$ this definition might look a bit *over-formalized* but later generalizes nicely to the definition of self organizing maps or other algorithms with non-binary assignment variables.
- the cost function is the average quadratic distance between prototypes and observations: $$E^T_{[\{m_q^{(\alpha)}\},\{\underline{\mathbf{w}}_q\}]} = \frac{1}{2p} \sum_{q,\alpha} m_q^{(\alpha)} \left(\underline{\mathbf{x}}^{(\alpha)} - \underline{\mathbf{w}}_q\right)^2$$

This definition results in a continuous (prototype locations) and discrete (cluster assignments) optimization process and of course as always E**T is used for model selection. Validation only has to be done when k-means is used for classification. The description of the algorithm is simple:

```
initialize prototypes around data centers and mass
loop
       1) assign each observation to closest prototype
       2) set prototype to the mean of its assigned observations
end
```

- the center of mass assignment in (2) can be shown to be optimal
- k-means always converges to a (local) optimum of E**T

Also an online version of k-means exists which is more robust against local optima. One problem of k-means (clustering in general) is to find the right number of prototypes. This can only be solved by using a prior for this number. One solution for k-means could for example be to set the minimal training error to a certain value and then introduce new prototypes during the iterations (in location of the cluster with largest variance) until the minimum value of E**T is reached.

Pairwise Clustering

Pairwise clustering is done on a set of p objects alpha, whose locations (in a vector space) are not given, but only the distances between the objects in a distance matrix. This distance matrix can be determined by measurements (e.g. in a psychophysical experiment) or the results of an algorithm (e.g. dissimilarity of protein sequences),. Of course they could also come through a distance measure on an underlying vector space or tadaaaaa: the kernel trick. So we are looking again for a set of clusters q which are defined solely by their assignment variables. The **cost function** is given as the average distance between two objects assigned to a cluster:

$$E_{[\{m_q^{(\alpha)}\}]} = \frac{1}{M}\sum_q \frac{\sum_{\alpha\alpha'} m_q^{(\alpha)} m_q^{(\alpha')} d_{\alpha\alpha'}}{\underbrace{\sum_\alpha m_q^{(\alpha)}}_{\substack{\text{number of objects}\\ \text{assigned to cluster } q}}}$$

It can be shown that using euclidian distances for the distance matrix results in k-means.

Mean field approximation

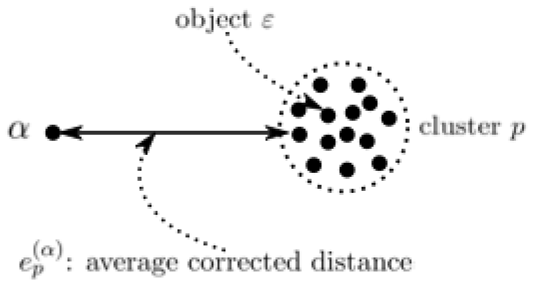
Finding the correct assignment variables for each point and cluster is a complex discrete optimization problem which can be solved by simulated annealing or mean field annealing. Simulated annealing is the stochastic algorithm which converges to the optimal solution (when the correct annealing schedule is used) and mean field annealing is its deterministic approximation. The basic idea again is that we would like to know the probabilities of a system to end up in a certain state, given the current temperature. This distribution is given by the Gibbs function, but its moments are hard to evaluate. Therefore the Gibbs distribution is approximated by a factorizing distribution and solving for the minimum of the difference (d_kl) between the Gibbs distribution and its approximation gives an efficient algorithm to to compute the mean fields of the factorization and via them the first moments of the assignment probabilities which quantify the probability that an object belongs to a cluster $\langle m_p^{(\gamma)} \rangle_Q = \frac{\exp\{-\beta m_p^{(\gamma)} e_p^{(\gamma)}\}}{\sum_\gamma \exp\{-\beta m_\gamma^{(\gamma)} e_\gamma^{(\gamma)}\}}$. For the noise parameter (inverse temperature) approaching infinity the assignment variables converge to {0, 1} and become hard cluster assignments. $\beta \to \infty : \langle m_p^{(\gamma)} \rangle_Q \to \{0,1\}$.

When we assume the distance matrix to be symmetric and have zeros on its diagonal, the computation of the mean fields simplifies and allows for a geometrical interpretation.

$$e_p^{(\alpha)} = \underbrace{\frac{2}{M}\frac{1}{\sum_\gamma \langle m_p^{(\gamma)}\rangle_Q}\sum_\delta \langle m_p^{(\delta)}\rangle_Q \underbrace{\left\{ d_{\delta\alpha} - \frac{1}{2}\frac{1}{\sum_\gamma \langle m_p^{(\gamma)}\rangle_Q}\sum_\varepsilon \langle m_p^{(\varepsilon)}\rangle_Q d_{\varepsilon\delta}\right\}}_{\substack{\text{distance between data objects } \alpha \text{ and } \delta,\\ \text{corrected by the average distance between}\\ \text{objects of the cluster, to which } \delta \text{ belongs (here: } p)}}}_{\substack{\text{average corrected distance between data objects}\\ \alpha \text{ and all objects } \delta \text{ of cluster } p}}$$

comments:

$\Rightarrow$ "metric visualization" of the $e_p^{(\alpha)}$:

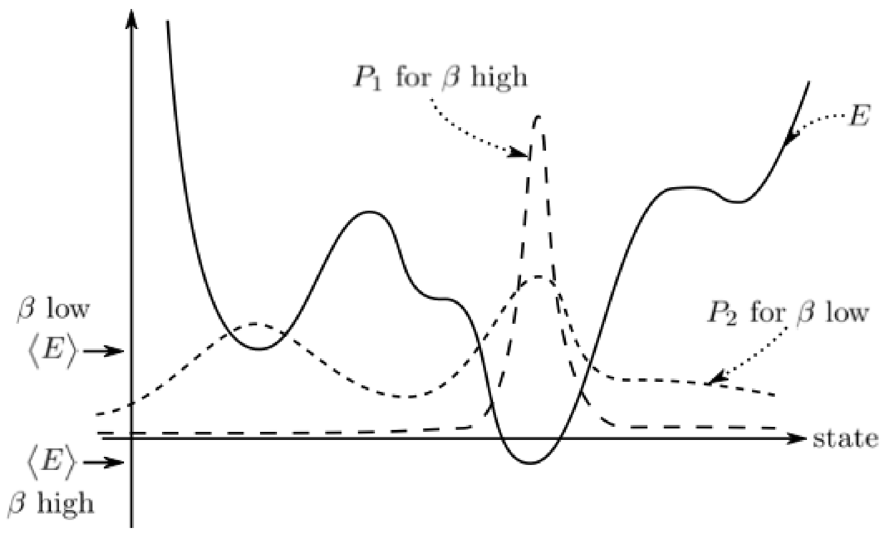


- right part of the equation: make the distance to a more distant but less concentrated cluster equal to a closer and very focused one
- left part: average over all the normalized distances

⇒ mean fields depend on assignment probabilities $\langle m_p^{(\gamma)}\rangle_Q$ rather than on the *hard* binary assignments. This is an effect of the underlying stochastic optimization, although the algorithm is deterministic now. The membership of an object to a cluster is only a *fuzzy* assignment → objects contribute only weighted by their probability of assignment.

⇒ assignment probabilities depend on the average normalized distance between the object alpha under consideration and the objects of cluster p.

For euclidean distances pairwise clustering with MFA becomes a natural extension of k-means with *fuzzy* cluster assignments. The choice of beta corresponds to the resolution (number of clusters) of the algorithm. This can be illustrated by the following plot which shows the influence of the parameter beta on the average energy (average euclidean distances in a cluster). High beta corresponds to a high resolution with a low error, but might be overfitting (number of clusters too large). A decrease of beta implies:



- increase of average cost
- increase of cluster size
- decrease in spatial resolution

⇒ beta controls the complexity of the clustering solution

Pairwise clustering can also deal with **missing data** or might voluntarily dismiss some of the data to reduce the computational complexity. The number of matrix elements in a full distance matrix is p**2 and the computation or measurement of all elements might be computationally too expensive or even infeasible in case of experiments. Furthermore: distance matrices are often redundant, not all matrix entries are needed. Because of the mean field computation, only average distances of one object to the others is used

$$e_p^{(\alpha)} = \left\{ \bar{d}_{p\alpha} - \frac{1}{2} \sum_{\delta} \langle m_p^{(\delta)} \rangle_Q \bar{d}_{p\delta} \right\} \cdot \frac{2}{M}$$

$\Rightarrow$ depend only on the average distances

So the simple heuristics to deal with missing data is to simply perform summations within $\bar{d}_{p\alpha}$ only over the available distances.

Self organizing maps (SOM)

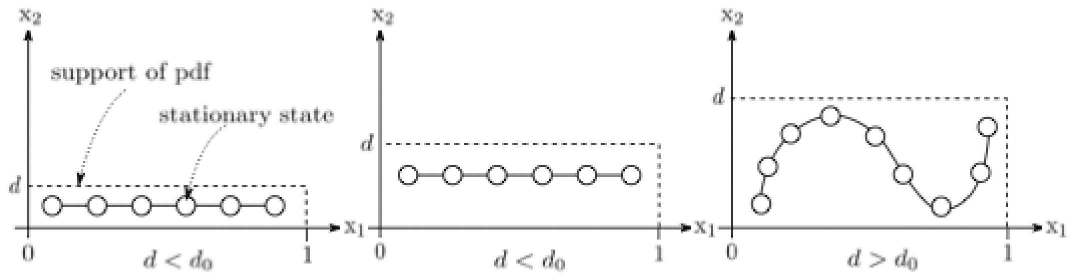
The goal of SOMs is to do clustering and still preserve topological properties of the input space. In simple words, SOMs are neural networks with a geometrical structure → the neurons know where they are and neighboring neurons should represent close by data points (in feature space). The *training* of a SOM is done similar to the k-means algorithm and evaluation is performed in a winner-takes-all fashion, a competitive network where only the neuron closest to the observation in question becomes active. The SOM training results in a 2D (typically 2D is used) map of the data space which can then be used for visualization or dimensionality reduction. The few modifications from k-means break the problem of permutation symmetries and removes *independence* between the neurons. In k-means only the prototype closest to an observation was moved, in SOM all prototypes are moved, weighted by the distance from the current observation. Short description of the SOM algorithm:

```
* choose the # of neurons (partitions)
* choose the layout of the map (1D, 2D, triangular or rectangular grid,
torus)
* init prototypes to be evenly sampled from the subspace spanned by the two
largest principal components.
* loop over the data points x**alpha
      * chose closest prototype
      * change all prototypes according to: $\Delta \underline{\mathbf{w}}_q = \varepsilon \; h_{\underline{\mathbf{qp}}} \; (\underline{\mathbf{x}}^{(\alpha)} \underline{\mathbf{w}}_q^{\text{old}})$
```

h_qp is the neighborhood function

- h_qp → delta_qp results in standard k-means
- similar learning steps for neighboring neurons
- $$h_{\underline{\mathbf{qp}}} = \exp\left\{ - \frac{(\underline{\mathbf{q}} - \underline{\mathbf{p}})^2}{2\sigma^2} \right\}$$
  typical choice for h_qp is a gaussian weighting
    - the annealing parameter sigma is initially large and will be reduced linearly or exponentially (but slow) according to an annealing schedule. If the final value of sigma becomes zero, the solution corresponds to a k-means solution (local optimum of k-means). If sigma is small but finite it results in a better visualization, but non-optimal clustering.

The following illustration shows that the SOM algorithm is able to automatically select the relevant feature dimensions:



SOM for pairwise clustering

Definition like previous pairwise clustering problems, set of objects, distance matrix, binary normalized assignment variables and now a set of clusters q (M in total) with a geometrical structure. The cost function is defined as:

$$E\left[\{m_{\underline{\mathbf{q}}}^{(\alpha)}\}\right] = \frac{1}{M} \sum_{\mathbf{s}} \frac{\sum_{\alpha,\alpha'} \left(\sum_{\underline{\mathbf{q}}} \overbrace{h_{\underline{\mathbf{sq}}}}^{\text{neighborhood function}} \; m_{\underline{\mathbf{q}}}^{(\alpha)} \right) \left(\sum_{\underline{\mathbf{q}}} h_{\underline{\mathbf{sq}}} m_{\underline{\mathbf{q}}^{(\alpha')}} \right) d_{\alpha\alpha'}}{\sum_{\alpha} \left(\sum_{\underline{\mathbf{q}}} h_{\underline{\mathbf{sq}}} m_{\underline{\mathbf{q}}}^{(\alpha)} \right)}$$

which is derived by simply replacing the assignment variables of pairwise clustering by $\sum_{\underline{\mathbf{q}}} h_{\underline{\mathbf{sq}}} m_{\underline{\mathbf{q}}}^{(\alpha)}$ which means that neurons which are neighboring with respect to the distance function contribute to the total average distance within a cluster. And of course, there is also a mean field annealing version of this thing here which is a deterministic approximation to speed up the computation.