Chapter 2

Method descriptions

The problems we are considering in the DALLAS project are either classification type problems (*e. g.* the AstraZeneca application) or regression problems (*e. g.* the EKA Chemicals application). It is therefore suitable to begin with a brief introduction to these fields and some key concepts.

2.1 Regression

Thorsteinn Rögnvaldsson

2.1.1 The regression problem

We are dealing with a "fixed regressor model". That is, we have a data set $\mathcal{X} = \{(\boldsymbol{x}(n), y(n))\}_{n=1,\ldots,N}$ of observation pairs, where $\boldsymbol{x}(n)$ is the input and $y(n)$ is the corresponding output. We assume that the output is generated by the following process

$$y(n) = g(\boldsymbol{x}(n)) + \varepsilon(n) \tag{2.1}$$

where $\varepsilon(n)$ is a zero mean noise process with constant variance σ_ε^2. We refer to g as the "underlying function".

"Model selection" refers to the search for g by picking candidate functions $f(\boldsymbol{x}; \boldsymbol{w})$ from a model family $\mathcal{F}$, where $\boldsymbol{w}$ denotes the parameters of the function. We select from the model family $\mathcal{F}$ the function $f(\boldsymbol{x}; \boldsymbol{w})$ that has the minimum "distance" $E(f(\boldsymbol{x}; \boldsymbol{w}); y)$ to our observed data y (the "distance" is often referred to as the error function).

The modeling process consists of selecting both an appropriate model family $\mathcal{F}$ and the best function in this family.

Examples of model families

Some examples of model families are:

$\mathcal{F} = \{$all linear models$\}$, and $\mathcal{F} = \{$all polynomial models of order $p\}$.

Examples of error functions (distance measures)

The summed square error (SSE):

$$E = \text{SSE} = \sum_{n=1}^{N} \left(f(\boldsymbol{x}(n); \boldsymbol{w}) - y(n)\right)^2 = \sum_{n=1}^{N} e^2(n) \tag{2.2}$$

The "maximum likelihood" (ML) measure: (the negative log likelihood because it is more convenient to work with)

$$E = -\ln \mathcal{L}(\mathcal{X}|\boldsymbol{w}) = -\ln\left(\prod_{n=1}^{N} p(\boldsymbol{x}(n), y(n)|\boldsymbol{w})\right) = -\sum_{n=1}^{N} \ln p(\boldsymbol{x}(n), y(n)|\boldsymbol{w}) \tag{2.3}$$

where $p(\boldsymbol{x}(n), y(n)|\boldsymbol{w})$ is the likelihood for the observation $\{\boldsymbol{x}(n), y(n)\}$ given the parameter values $\boldsymbol{w}$. The most common assumption is the Gaussian likelihood in which case the negative log likelihood is equal to the SSE.

The Bayesian measure:
Maximizing the likelihood is somewhat strange. Why maximize the likelihood for the observations given the model parameters (although we do this by changing the model parameters)? What we really would like to do is to maximize the model parameters, given the observations. Bayes' theorem tells us how we should do. The probability for the model parameters, given the observations, is expressed as

$$p(\boldsymbol{w}|\mathcal{X}) = \frac{p(\mathcal{X}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{X})} = \frac{\mathcal{L}(\mathcal{X}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{X})} \tag{2.4}$$

where $p(\boldsymbol{w})$ is our "prior" for the model parameters $\boldsymbol{w}$. Just as in the case of the ML cost, it is more convenient to minimize the negative likelihood, which gives us

$$\begin{aligned} E = -\ln p(\boldsymbol{w}|\mathcal{X}) &= -\ln \mathcal{L}(\mathcal{X}|\boldsymbol{w}) - \ln p(\boldsymbol{w}) + \ln p(\mathcal{X}) \\ &\rightarrow -\ln \mathcal{L}(\mathcal{X}|\boldsymbol{w}) - \ln p(\boldsymbol{w}) \end{aligned} \tag{2.5}$$

since the third term does not depend on the model parameters $\boldsymbol{w}$.

The Bayesian error measure is even more general than the ML error. The ML error is equal to the special case of a uniform prior in the Bayesian picture.

The Bayesian error is important in the context of overfitting.

2.1.2 Overfitting and the bias vs. variance trade-off

The training data is, unfortunately, only a sample of the real world and it is surprisingly easy to overemphasize the importance of the training data, at the cost of worse performance on new test data (*i. e.* worse generalization performance).

To understand why, we can look at what is commonly referred to as "model bias" and "model variance". Model bias is a measure of how well we can model the underlying function g with our model family $\mathcal{F}$. If the underlying function can be modeled perfectly with a model from our family, *i. e.* if the underlying model is a member of our family, $g \in \mathcal{F}$, then we say that our model family has zero model bias. If the underlying function is not a member of the model family, $g \notin \mathcal{F}$, then we say that our model family is biased. Model variance is a measure of how much our models vary when we train them with different training sets. If the model family $\mathcal{F}$ is very small then there will be small differences between models trained with different training sets and we say that the model variance is small. On the other hand, if the model family is large then there can be (will be, according to Murphy's law) large differences between models trained with different training sets and we say that the model variance is large.

Examples:

Suppose that the underlying function g is linear.
$\mathcal{F} = \{$ all linear models $\}$ has zero model bias and small model variance.
$\mathcal{F} = \{$ all polynomial models of order 3 $\}$ also has zero model bias, but a significantly larger model variance.

Suppose that the underlying function g is cubic.
$\mathcal{F} = \{$ all linear models $\}$ has a significant model bias and small model variance.
$\mathcal{F} = \{$ all polynomial models of order 3 $\}$ has zero model bias and a large model variance.

Whenever we are constructing a model, we should remember that the ultimate goal is to minimize the expected generalization error. Generalization error is the sum of model bias (squared) and model variance. Thus, minimizing expected generalization error necessarily means weighting the model bias against the model variance. This may mean that it is a bad idea to choose a model family $\mathcal{F}$ such that it is guaranteed that $g \in \mathcal{F}$, because the accompanying model variance cancels the benefits from having zero bias.

2.1.3 Classical statistical methods for regression

The most well-known methods for regression in statistics are linear regression (LR), principal components regression (PCR), and partial least squares (PLS). All these methods are linear but they do not produce the same result in general.

Linear regression amounts to using a linear model and minimizing the summed square error (SSE). Principal components regression is also a linear model, but the variables are projected onto the principal axes of the data covariance matrix to transform them to new more informative variables (where fewer variables are needed to solve the problem). Partial least squares is also a PCR-like method where the variables are projected onto the principal axes of the data covariance matrix. However, in PLS one does also consider the variance in the output.

2.2 Classification

Thorsteinn Rögnvaldsson

To *classify* means that an object or event is ordered into one out of several classes, *i. e.* a mapping from a feature space $\mathbf{X}^D$ to a category space $\mathbf{C}^K$

$$f : \mathbf{X}^D \to \mathbf{C}^K \tag{2.6}$$

where $\mathbf{X}^D \subset \mathbf{R}^D$ and $\mathbf{C}^K = \{0, 1\}^K$.

2.2.1 Statistical decision theory

Classification is a decision, one decides to categorize an observation into a category. The final decision of course depends on the consequences of the decision and not just the probability that an observation $\mathbf{x}(n)$ belongs to a given category c_k. Medical applications are excellent examples of this.

Statistical decision theory tells us how we should proceed to make an optimal decision, given that we know the costs associated with our decisions and the probabilities for the different categories. The optimal decision strategy, called the *Bayes classifier*, is the strategy that always chooses the decision that minimizes the expected conditional risk

$$R(\alpha_i|\mathbf{x}) = \sum_{k=1}^{K} \lambda(\alpha_i|c_k) p(c_k|\mathbf{x}). \tag{2.7}$$

where α_i is an action, c_k is a category, and $\lambda(\alpha_i|c_k)$ is the cost for taking action α_i if the object belongs to category c_k. Thus, making the right decision means having to know the *a posteriori* probability $p(c_k|\mathbf{x})$ and the conditional costs for different actions. The *a posteriori* probabilities can of course be estimated from the conditional probabilities by using Bayes' rule

$$p(c_k|\mathbf{x}) = \frac{p(c_k)p(\mathbf{x}|c_k)}{p(\mathbf{x})}. \tag{2.8}$$

It is common to group classifiers into three groups, depending on the philosophy behind their construction:

1. **A posteriori classifiers:** Model the a posteriori probabilities $p(c_k|\mathbf{x})$.
2. **Probability density classifiers:** Model the conditional probabilities $p(\mathbf{x}|c_k)$ and combine them with Bayes' rule to get at the *a posteriori* probabilities.
3. **Decision boundary classifiers:** Construct only discrimination functions.

2.2.2 Parametric and non-parametric models

When modeling, it is common to make a distinction between *parametric* and *non-parametric* models. Parametric models are models where one has made an assumption about the probability density (or a posteriori probability). Non-parametric models are models where no assumption is made, so-called *general approximators* are used instead.

The distinction is somewhat artificial, since all models have parameters. It is more correct to speak of models with many free parameters (non-parametric), and models with few free parameters (parametric).

The advantage with parametric models is that they are simple and quick to construct. One can afford to try many different setups. The drawback of parametric models is that one may have assumed the "wrong" parametric family, in the sense that the Bayes classifier (the optimal classifier) is not a member of the hypothesis family. This leads to a model *bias*, meaning that we will never be able to model the Bayes optimal classifier.

The benefit of non-parametric classifiers is that they are general and that we run little risk of having a model bias. The drawback, however, is that they take a lot of effort to construct, there will be little time for experimenting, and they are likely to "overtrain' and have a large model *variance*. This means that the resulting classifier will depend very much on the set of observations used to construct it, and if we change one or a few observations then the resulting classifier will also change significantly. A classifier with a large model variance is unlikely to be able to generalize well to new observations.

It is therefore a good idea to work in the intermediate area between non-parametric (many free parameters) and parametric (few free parameters) classifiers. In this region it is important to do a trade-off between model bias and model variance, because both contribute to the generalization error. Artificial neural networks are an example of a classification method in the "twilight zone" between parametric and non-parametric classifiers.

2.2.3 Classical statistical methods for classification

The most common classifiers in statistics are: The Gaussian classifier (can be made both linear and quadratic), k-nearest neighbor classifiers, and linear discriminants.

Gaussian classifiers are examples of parametric probability density classifiers. The idea is to assume that the data is normally distributed (the parametric assumption) and estimate the mean and variance of this distribution.

The classic non-parametric classifier is k-nearest neighbors which is extremely appealing for its simplicity and speed. The k-nearest neighbors classifier simply classifies a new observation as belonging to the most common category among previous observations that are similar to it.

2.2.4 How to estimate the generalization error

The real goal when modeling is to generalize to new data, not just perform well on the training data set that is presented during training.

In general, the error on the training data will be a biased estimate of the generalization error. To be specific, it will tend to be smaller than the generalization error if we select our model such that it minimizes the training error. We can therefore not use the training error as our selection criteria.

One way to estimate the generalization error is to do cross-validation. This means using a test data set, which is a subset of the available data (typically 25-35%) that is removed before any training is done, and which is not used again until all training is done. The performance on this test data will be an unbiased estimate of the generalization error, provided that the data has not been used in any way during the modeling process. If it has been used, *e. g.* for model validation when selecting hyperparameter values, then it will be a biased estimate.

If there is lots of data available then it may be sufficient to use one test set for estimating the generalization error. However, if data is scarce then it is necessary to use more data-efficient methods. One such method is the K-fold cross-validation method.

The central idea in K-fold cross-validation is to repeat the cross-validation test K times. That is, divide the available data into K subsets, here denoted by $\mathcal{D}_k$, where each subset contains a sample of the data that reflects the data distribution (*i. e.* you must make sure that one subset does not contain *e. g.* only one category in a classification task). The procedure then goes like this:

1. Repeat K times, *i. e.* until all data subsets have been used for testing once.
    1.1 Set aside one of the subsets, $\mathcal{D}_k$, for testing, and use the remaining data subsets $\mathcal{D}_{i \neq k}$ for training.
    1.2 Train your model using the training data.

1.3 Test your model on the data subset $\mathcal{D}_k$. This gives you a test data error $E_{test,k}$.

2. The estimate of the generalization error is the mean of the K individual test errors: $E_{gen.} = \frac{1}{K} \sum_k E_{test,k}$.

One benefit with K-fold cross-validation is that you can estimate an error bar for the generalization error by computing the standard deviation of the $E_{test,k}$ values.

Note: The errors $E_{test,k}$ are often approximately log-normally distributed. At least, $\log E_{test,k}$ tends to be more normally distributed than $E_{test,k}$. It is therefore more appropriate to use the mean of the logs as an estimate for the log generalization error. That is

$$\log E_{gen.} = \frac{1}{K} \sum_{k=1}^{K} \log E_{test,k} \tag{2.9}$$

$$\Delta \log E_{gen.} = 1.96 \sqrt{\frac{1}{K-1} \sum_{k=1}^{K} [\log E_{test,k} - \log E_{gen.}]^2} \tag{2.10}$$

where the lower row is a 95% confidence band for the log generalization error.

An error bar from cross-validation includes the model variation due to both different training sets and different initial conditions.

2.3 Multilayer Perceptrons and Error Backpropagation

Thorsteinn Rögnvaldsson

2.3.1 The multilayer perceptron – general

A "multilayer perceptron" (MLP) is a hierarchical structure of several so-called "simple" perceptrons (with smooth transfer functions). For instance, a "one hidden layer" MLP with a logistic output unit looks like

$$\hat{y}(\mathbf{x}) = \frac{1}{1+\exp[-a(\mathbf{x})]} \tag{2.11}$$

$$a(\mathbf{x}) = \sum_{j=0}^{M} v_j h_j(\mathbf{x}) = \mathbf{v}^T \mathbf{h}(\mathbf{x}) \tag{2.12}$$

$$h_j(\mathbf{x}) = \sum_{k=0}^{D} \phi(w_{jk} x_k) = \phi(\mathbf{w}_j^T \mathbf{x}) \tag{2.13}$$

where the transfer function, or activation function, $\phi(z)$ typically is a sigmoid of the form

$$\phi(z) = \tanh(z), \tag{2.14}$$

$$\phi(z) = \frac{1}{1+e^{-z}}. \tag{2.15}$$

The former type, the hyperbolic tangent, is the more common one and it makes the training a little easier than if you use a logistic function.

The logistic output unit (2.11) is the correct one to use for a classification problem.

If the idea is to model a function (*i. e.* nonlinear regression) then it is common to use a linear output unit

$$\hat{y}(\mathbf{x}) = a(\mathbf{x}). \tag{2.16}$$

2.3.2 Training an MLP – Backpropagation

The perhaps most straightforward way to design a training algorithm for the MLP is to use the gradient descent algorithm. What we need is for the model output $\hat{y}$ to be differentiable with respect to all the parameters w_{jk} and v_j. We have a training data set $\mathcal{X} = \{\mathbf{x}(n), y(n)\}_{n=1,\ldots,N}$ with N observations, and we denote all the weights in the network by $\mathbf{W} = \{\mathbf{w}_j, \mathbf{v}\}$. The batch form of gradient descent then goes as follows:

1. Initialize $\mathbf{W}$ with *e. g.* small random values.
2. Repeat until convergence (either when the error E is below some preset value or until the gradient $\nabla_W E$ is smaller than a preset value), t is the iteration number
   - 2.1 Compute the update
     $\Delta\mathbf{W}(t) = -\eta \nabla_W E(t) = \eta \sum_{n=1}^{N} e(n,t) \nabla_W \hat{y}(n,t)$
     where $e(n,t) = (y(n) - \hat{y}(n,t))$
   - 2.2 Update the weights $\mathbf{W}(t+1) = \mathbf{W}(t) + \Delta\mathbf{W}(t)$
   - 2.3 Compute the error $E(t+1)$

As an example, we compute the weight updates for the special case of a multilayer perceptron with one hidden layer, using the transfer function $\phi(z)$ (*e. g.* $\tanh(z)$), and one output unit with the transfer function $\theta(z)$ (*e. g.* logistic or linear). We use half the mean square error

$$E = \frac{1}{2N}\sum_{n=1}^{N}\left[y(n)-\hat{y}(n)\right]^2 = \frac{1}{2N}\sum_{n=1}^{N}e^2(n), \tag{2.17}$$

and the following notation

$$\hat{y}(\mathbf{x}) = \theta\left[a(\mathbf{x})\right], \tag{2.18}$$

$$a(\mathbf{x}) = v_0 + \sum_{j=1}^{M} v_j h_j(\mathbf{x}), \tag{2.19}$$

$$h_j(\mathbf{x}) = \phi\left[b_j(\mathbf{x})\right], \tag{2.20}$$

$$b_j(\mathbf{x}) = w_{j0} + \sum_{k=1}^{D} w_{jk} x_k. \tag{2.21}$$

Here, v_j are the weights between the hidden layer and the output layer, and w_{jk} are the weights between the input and the hidden layer.

For weight v_i we get

$$\begin{aligned}
\frac{\partial E}{\partial v_i} &= -\frac{1}{N}\sum_{n=1}^{N} e(n)\frac{\partial \hat{y}(n)}{\partial v_i} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right]\frac{\partial a(n)}{\partial v_i} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] h_i(n)
\end{aligned} \tag{2.22}$$

$$\Rightarrow \Delta v_i = -\eta\frac{\partial E}{\partial v_i} = \eta\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] h_i(n) \tag{2.23}$$

with the definition $h_0(n) \equiv 1$. If the output transfer function is linear, *i. e.* $\theta(z) = z$, then $\theta'(z) = 1$. If the output function is logistic, *i. e.* $\theta(z) = [1+\exp(-z)]^{-1}$, then $\theta'(z) = \theta(z)[1-\theta(z)]$.

For weight w_{il} we get

$$\begin{aligned}
\frac{\partial E}{\partial w_{il}} &= -\frac{1}{N}\sum_{n=1}^{N} e(n)\frac{\partial \hat{y}(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right]\frac{\partial a(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\frac{\partial h_i(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\phi'\left[b_i(n)\right]\frac{\partial b_i(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\phi'\left[b_i(n)\right] x_l
\end{aligned} \tag{2.24}$$

$$\Rightarrow \Delta w_{il} = -\eta\frac{\partial E}{\partial w_{il}} = \eta\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\phi'\left[b_i(n)\right] x_l(n) \tag{2.25}$$

with the definition $x_0(n) \equiv 1$. If the hidden unit transfer function is the hyperbolic tangent function, *i. e.* $\phi(z) = \tanh(z)$, then $\phi'(z) = 1 - \phi^2(z)$.

This gradient descent method for updating the weights has become known as the "backpropagation" training algorithm. The motivation for the name becomes clear if we introduce the notation

$$\delta(n) = e(n)\theta'[a(n)], \quad (2.26)$$

$$\delta_i(n) = \delta(n)v_i\phi'[b_i(n)], \quad (2.27)$$

which enables us to write

$$\Delta v_i = \eta \frac{1}{N}\sum_{n=1}^{N} \delta(n)h_i(n), \quad (2.28)$$

$$\Delta w_{il} = \eta \frac{1}{N}\sum_{n=1}^{N} \delta_i(n)x_l(n), \quad (2.29)$$

which is very similar to the good old LMS algorithm. Expression (2.27) corresponds to a propagation of $\delta(n)$ backwards through the network.

The gradient descent learning algorithm corresponds to backprop in its batch form, where the update is computed using all the available training data. There is also an "on-line" version where the updates are done after each pattern $\mathbf{x}(n)$ without averaging over all patterns.

Backpropagation is, in general, a very slow learning algorithm – even with momentum – and there are many better algorithms which we discuss below. However, backpropagation was very important in the beginning of the 1980:ies because is was used to demonstrate that multilayer perceptrons can learn things.

2.3.3 RPROP

A very useful gradient based learning algorithm is the "resilient backpropagation" (RPROP) algorithm. It uses individual adaptive learning rates combined with the so-called "Manhattan" update step.

The standard backpropagation updates the weights according to

$$\Delta w_{il} = -\eta \frac{\partial E}{\partial w_{il}}. \quad (2.30)$$

The "Manhattan" update step, on the other hand, uses only the sign of the derivative (the reason for the name should be obvious to anyone who has seen a map of Manhattan), *i. e.*

$$\Delta w_{il} = -\eta \text{sign}\left[\frac{\partial E}{\partial w_{il}}\right]. \quad (2.31)$$

The RPROP algorithm combines this Manhattan step with individual learning rates for each weight, and the algorithm goes as follows

$$\Delta w_{il}(t) = -\eta_{il}(t)\text{sign}\left[\frac{\partial E}{\partial w_{il}}\right], \quad (2.32)$$

where w_{il} denotes any weight in the network (*e. g.* also hidden to output weights).

The learning rate $\eta_{il}(t)$ is adjusted according to

$$\eta_{il}(t) = \begin{cases} \gamma^+\eta_{il}(t-1) & \text{if } \partial_{il}E(t)\cdot\partial_{il}E(t-1) > 0 \\ \gamma^-\eta_{il}(t-1) & \text{if } \partial_{il}E(t)\cdot\partial_{il}E(t-1) < 0 \end{cases} \quad (2.33)$$

where γ^+ and γ^- are different growth/shrinking factors ($0 < \gamma^- < 1 < \gamma^+$). Values that have worked well for me are $\gamma^- = 0.5$ and $\gamma^+ = 1.2$, with limits such that $10^{-6} \leq \eta_{ij}(t) \leq 50$. I have used the short notation $\partial_{il} E(t) \equiv \frac{\partial E(t)}{\partial w_{il}}$. The RPROP algorithm is a batch algorithm, since the learning rate update becomes noisy and uncertain if the error E is evaluated over only a single pattern.

The RPROP algorithm is implemented in the MATLAB neural network toolbox.

2.3.4 Second order learning algorithms

Backpropagation, *i. e.* gradient descent, is a *first order* learning algorithm. This means that it only uses information about the first order derivative when it minimizes the error. The idea behind a first order algorithm can be illustrated by expanding the error E in a Taylor series around the current weight position $\mathbf{W}$

$$E(\mathbf{W} + \Delta\mathbf{W}) = E(\mathbf{W}) + \nabla_W E(\mathbf{W})^T \Delta\mathbf{W} + \mathcal{O}\left(\|\Delta\mathbf{W}\|^2\right). \tag{2.34}$$

The vector $\mathbf{W}$ contains all the weights w_{jk} and v_j (and others if we are considering other network architectures) and we require that $\Delta\mathbf{W}$ is small. The notation $\mathcal{O}\left(\|\Delta\mathbf{W}\|^2\right)$ denotes all the terms that contains the small weight step $\Delta\mathbf{W}$ multiplied by itself at least once, and by "small" we mean that $\Delta\mathbf{W}$ is so small that the gradient term $\nabla_W E(\mathbf{W})\Delta\mathbf{W}$ is larger than the sum of the higher order terms. In that case we can ignore the higher order terms and write

$$E(\mathbf{W} + \Delta\mathbf{W}) \approx E(\mathbf{W}) + \nabla_W E(\mathbf{W})^T \Delta\mathbf{W}. \tag{2.35}$$

Now, we want to change the weights so that the new error $E(\mathbf{W} + \Delta\mathbf{W})$ is smaller than the current error $E(\mathbf{W})$. One way to guarantee this is to set the weight update $\Delta\mathbf{W}$ proportional to the negative gradient, *i. e.* $\Delta\mathbf{W} = -\eta\nabla_W E(\mathbf{W})$, in which case we have

$$E(\mathbf{W} + \Delta\mathbf{W}) \approx E(\mathbf{W}) - \eta\|\nabla_W E(\mathbf{W})\|^2 \leq E(\mathbf{W}). \tag{2.36}$$

However, this of course requires that $\Delta\mathbf{W}$ is so small that we can motivate (2.35).

We can extend this and also consider the second order term in the Taylor expansion. That is

$$E(\mathbf{W} + \Delta\mathbf{W}) = E(\mathbf{W}) + \nabla_W E(\mathbf{W})^T \Delta\mathbf{W} + \frac{1}{2}\Delta\mathbf{W}^T H(\mathbf{W})\Delta\mathbf{W} + \mathcal{O}\left(\|\Delta\mathbf{W}\|^3\right), \tag{2.37}$$

where

$$H(\mathbf{W}) = \nabla_W \nabla_W^T E(\mathbf{W}) \tag{2.38}$$

is the Hessian matrix with elements $H_{ij}(\mathbf{W}) = \frac{\partial^2 E(\mathbf{W})}{\partial w_i \partial w_j}$. The Hessian is symmetric (all eigenvalues are consequently real and we can diagonalize H with an orthogonal transformation).

If we can ignore the higher order terms in (2.37) then we have

$$E(\mathbf{W} + \Delta\mathbf{W}) \approx E(\mathbf{W}) + \nabla_W E(\mathbf{W})^T \Delta\mathbf{W} + \frac{1}{2}\Delta\mathbf{W}^T H(\mathbf{W})\Delta\mathbf{W}. \tag{2.39}$$

We want to change the weights so that the new error $E(\mathbf{W} + \Delta\mathbf{W})$ is smaller than the current error $E(\mathbf{W})$. Furthermore, we want it to be as small as possible. That is, we want to minimize $E(\mathbf{W} + \Delta\mathbf{W})$ by choosing $\Delta\mathbf{W}$ appropriately. The requirement that we end up at an extremum point is

$$\begin{aligned} \nabla_W E(\mathbf{W} + \Delta\mathbf{W}) &= \mathbf{0} \\ \Rightarrow \nabla_W E(\mathbf{W}) + H(\mathbf{W})\Delta\mathbf{W} &= \mathbf{0}, \end{aligned} \tag{2.40}$$

which yields the optimum weight update as

$$\Delta\mathbf{W} = H^{-1}(\mathbf{W})\nabla_W E(\mathbf{W}). \tag{2.41}$$

To guarantee that this is a minimum point we must also require that the Hessian matrix is positive definite. This means that all the eigenvalues of the Hessian matrix must be positive. If any of the eigenvalues are zero then we have a saddle point and $H(\mathbf{W})$ is not invertible. If any of the eigenvalues of $H(\mathbf{W})$ are negative then we have a maximum point for at least one of the weights w_j and (2.41) will actually move away from the minimum!

The update step (2.41) is usually referred to as a "Newton-step", and the minimization method that uses this update step is the Newton algorithm.

Some problems with "vanilla" Newton learning (2.41) are:

- The Hessian matrix may not be invertible, *i. e.* some of the eigenvalues are zero.
- The Hessian matrix may have negative eigenvalues.
- The Hessian matrix is expensive to compute and also expensive to invert. The learning may therefore be slower than a first order method.

The first two problems are handled by regularizing the Hessian, *i. e.* by replacing $H(\mathbf{W})$ by $H(\mathbf{W}) + \lambda\mathbf{I}$. This effectively filters out all eigenvalues that are smaller than λ. The third problem is handled by "Quasi-Newton" methods that iteratively try to estimate the inverse Hessian using expressions of the form $H^{-1}(\mathbf{W} + \Delta\mathbf{W}) \approx H^{-1}(\mathbf{W}) + \text{correction}$.

The Levenberg-Marquardt algorithm

The Levenberg-Marquardt is a very efficient second order learning algorithm that builds on the assumption that the error E is a quadratic error (which it usually is), like half the mean square error. In this case we have

$$H_{ij} = \frac{1}{2}\frac{\partial^2 \text{MSE}}{\partial w_i \partial w_j} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial \hat{y}(n)}{\partial w_i}\frac{\partial \hat{y}(n)}{\partial w_j} + \frac{1}{N}\sum_{n=1}^{N} e(n)\frac{\partial^2 \hat{y}(n)}{\partial w_i \partial w_j}. \tag{2.42}$$

If the residual $e(n)$ is symmetrically distributed around zero and small then we can assume that the second term in (2.42) is very small compared to the first term. If so, then we can approximate

$$\begin{aligned} H_{ij} &\approx \frac{1}{N}\sum_{n=1}^{N} \frac{\partial \hat{y}(n)}{\partial w_i}\frac{\partial \hat{y}(n)}{\partial w_j} \\ H(\mathbf{W}) &\approx \frac{1}{N}\sum_{n=1}^{N} \mathbf{J}(n)\mathbf{J}^T(n), \end{aligned} \tag{2.43}$$

where we have used the notation

$$\mathbf{J}(n) = \nabla_W \hat{y}(n) \tag{2.44}$$

and we will refer to $\mathbf{J}$ as the "Jacobian". This approximation is not as costly to compute as the exact Hessian, since no second order derivatives are needed.

The fact that the Hessian is approximated by a sum of outer products $\mathbf{J}\mathbf{J}^T$ means that the rank of H is at most N. That is, there must be at least as many observations as there are weights in the network (which intuitively makes sense).

This approximation of the Hessian is used in a Newton step together with a regularization term, so that the Levenberg-Marquardt update is

$$\Delta\mathbf{W} = \left[\frac{1}{N}\sum_{n=1}^{N} \mathbf{J}(n)\mathbf{J}^T(n) + \lambda\mathbf{I}\right]^{-1} \nabla_W E(\mathbf{W}). \tag{2.45}$$

The Levenberg-Marquardt update is a very useful learning algorithm, and it represents a combination of gradient descent and a Newton step search. We have that

$$\Delta\mathbf{W} \to \begin{cases} \frac{1}{\lambda}\nabla E(\mathbf{W}) & \text{when } \lambda \to \infty \\ \left[\frac{1}{N}\sum_n \mathbf{J}(n)\mathbf{J}^T(n)\right]^{-1}\nabla E(\mathbf{W}) & \text{when } \lambda \to 0 \end{cases}, \tag{2.46}$$

which corresponds to gradient descent, with $\eta = 1/\lambda$, when λ is large, and to Newton learning when λ is small.

2.3.5 Interpretation of the MLP

Classification

If the multilayer perceptron will be used for classification then it should have a logistic output (in the two-class case, in multi-class cases we would use a generalization of the logistic function). If we have a single hidden layer MLP, the output is (c.f. equations (2.18) – (2.21))

$$\hat{y}(\mathbf{x}) = \left\{1 + \exp\left[v_0 + \sum_j v_j h_j(\mathbf{w}_j, \mathbf{x})\right]\right\}^{-1} \tag{2.47}$$

which is of the general form

$$\hat{y}(\mathbf{x}) = \frac{1}{1 + \exp\left[f(\mathbf{x})\right]} \tag{2.48}$$

where $f(\mathbf{x})$ is a nonlinear function of $\mathbf{x}$ (actually, it is a function of projections of $\mathbf{x}$ onto directions $\mathbf{w}_j$). If we compare this to the classical classification methods, we see that we are dealing with a generalization of the logistic regression, a nonlinear logistic regression model. That is, we are modeling the a posteriori probability $p(c|\mathbf{x})$, but using a nonlinear decision boundary.

Regression

We use a linear output in the regression case. The MLP function, using a single hidden layer, is then (generally speaking)

$$\hat{y}(\mathbf{x}) = v_0 + \sum_{j=1}^{M} v_j h_j(\mathbf{w}_j^T\mathbf{x}) \tag{2.49}$$

where $h_j(\mathbf{w}^T\mathbf{x})$ are nonlinear functions of the projections $\mathbf{w}_j^T\mathbf{x}$. Models of this form are often referred to as *projection pursuit regression* (PPR) models in statistics, since projections $\mathbf{w}_j^T\mathbf{x}$ are used as arguments. (To be exact, PPR refers to a specific method of minimizing the error and choosing projections but there are strong similarities between the MLP and PPR.)

2.4 A guide to recurrent neural networks and backpropagation

Mikael Bodén

This section provides guidance to some of the concepts surrounding recurrent neural networks. Contrary to feedforward networks, recurrent networks can be sensitive, and be adapted to past inputs. Backpropagation learning is described for feedforward networks, adapted to suit our (probabilistic) modeling needs, and extended to cover recurrent networks. The aim of this brief text is to set the scene for applying and understanding recurrent neural networks.

2.4.1 Introduction

It is well known that conventional feedforward neural networks can be used to approximate *any* spatially finite function given a (potentially very large) set of hidden nodes. That is, for functions which have a *fixed* input space there is always a way of encoding these functions as neural networks. For a two-layered network, the mapping consists of two steps,

$$y(t) = G(F(x(t))). \tag{2.50}$$

We can use automatic learning techniques such as backpropagation to find the weights of the network (G and F) if sufficient samples from the function is available.

Recurrent neural networks are fundamentally different from feedforward architectures in the sense that they not only operate on an input space but also on an internal *state* space – a trace of what already has been processed by the network. This is equivalent to an Iterated Function System (IFS; see [Barnsley, 1993] for a general introduction to IFSs; [Kolen, 1994] for a neural network perspective) or a Dynamical System (DS; see *e. g.* [Devaney, 1989] for a general introduction to dynamical systems; [Tino et al., 1998; Casey, 1996] for neural network perspectives). The state space enables the representation (and learning) of temporally/sequentially extended dependencies over unspecified (and potentially infinite) intervals according to

$$y(t) = G(s(t)) \tag{2.51}$$

$$s(t) = F(s(t-1), x(t)). \tag{2.52}$$

To limit the scope of this text and simplify mathematical matters we will assume that the network operates in discrete time steps (it is perfectly possible to use continuous time instead). It turns out that if we further assume that weights are at least rational and continuous output functions are used, networks are capable of representing *any* Turing Machine (again assuming that any number of hidden nodes are available). This is important since we then know that all that can be computed, can be processed[1] equally well with a discrete time recurrent neural network. It has even been suggested that if real weights are used (the neural network is completely analog) we get super-Turing Machine capabilities [Siegelmann, 1999].

2.4.2 Some basic definitions

To simplify notation we will restrict equations to include two-layered networks, *i. e.* networks with two layers of nodes excluding the input layer (leaving us with one 'hidden' or 'state' layer, and one 'output' layer). Each layer will have its own index variable: k for output nodes, j (and h) for hidden, and i for input nodes. In a feed forward network, the input vector, $\mathbf{x}$, is propagated through a weight layer, $\mathbf{V}$,

$$y_j(t) = f(net_j(t)) \tag{2.53}$$

[1] I am intentionally avoiding the term 'computed'.

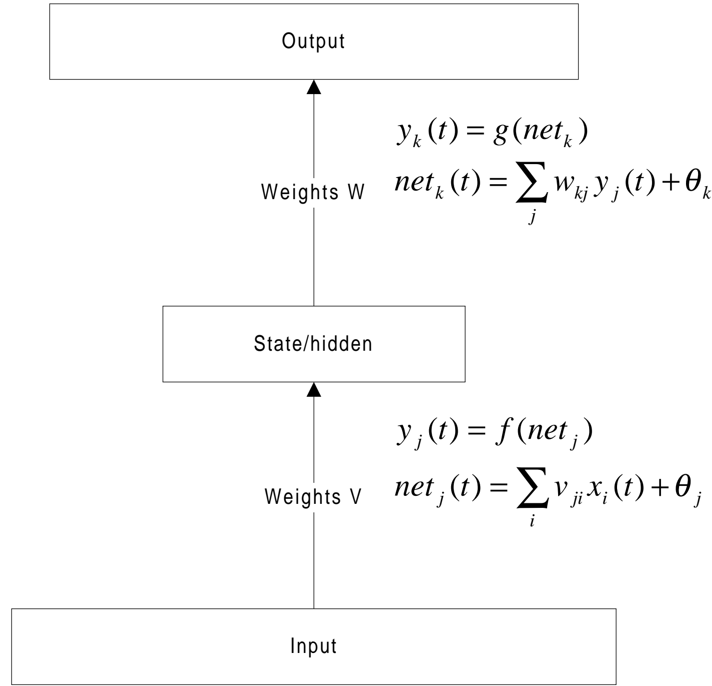

Figure 2.1: A feedforward network.

$$net_j(t) = \sum_i^n x_i(t) v_{ji} + \theta_j \tag{2.54}$$

where n is the number of inputs, θ_j is a bias, and f is an output function (of any differentiable type). A network is shown in Figure 2.1.

In a simple recurrent network, the input vector is similarly propagated through a weight layer, but also combined with the previous state activation through an additional *recurrent* weight layer, **U**,

$$y_j(t) = f(net_j(t)) \tag{2.55}$$

$$net_j(t) = \sum_i^n x_i(t) v_{ji} + \sum_h^m y_h(t-1) u_{jh}) + \theta_j \tag{2.56}$$

where m is the number of 'state' nodes.

The output of the network is in both cases determined by the state and a set of output weights, **W**,

$$y_k(t) = g(net_k(t)) \tag{2.57}$$

$$net_k(t) = \sum_j^m y_j(t) w_{kj} + \theta_k \tag{2.58}$$

where g is an output function (possibly the same as f).

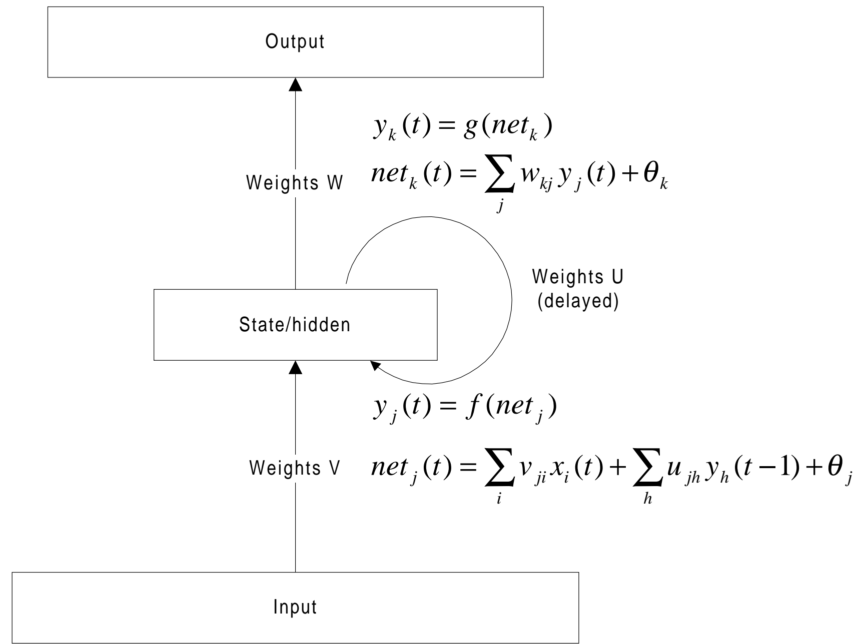

Figure 2.2: A simple recurrent network.

2.4.3 The principle of backpropagation

Any network structure can be trained with backpropagation when desired output patterns exist and each function that has been used to calculate the actual output patterns is differentiable. As with conventional gradient descent (or ascent), backpropagation works by, for each modifiable weight, calculating the gradient of a cost (or error) function with respect to the weight and then adjusting it accordingly.

The most frequently used cost function is the summed squared error (SSE). Each pattern or presentation (from the training set), p, adds to the cost, over all output units, k.

$$C = \frac{1}{2} \sum_p^n \sum_k^m (d_{pk} - y_{pk})^2 \tag{2.59}$$

where d is the desired output, n is the total number of available training samples and m is the total number of output nodes.

According to gradient descent, each weight change in the network should be proportional to the negative gradient of the cost with respect to the specific weight we are interested in modifying.

$$\Delta w = -\eta \frac{\partial C}{\partial w} \tag{2.60}$$

where η is a learning rate.

The weight change is best understood (using the chain rule) by distinguishing between an error component, $\delta = -\partial C / \partial net$, and $\partial net / \partial w$. Thus, the error for output nodes is

$$\delta_{pk} = -\frac{\partial C}{\partial y_{pk}} \frac{\partial y_{pk}}{\partial net_{pk}} = (d_{pk} - y_{pk}) g'(y_{pk}) \tag{2.61}$$

and for hidden nodes

$$\delta_{pj} = -(\sum_k^m \frac{\partial C}{\partial y_{pk}} \frac{\partial y_{pk}}{\partial net_{pk}} \frac{\partial net_{pk}}{y_{pj}}) \frac{\partial y_{pj}}{\partial net_{pj}} = \sum_k^m \delta_{pk} w_{kj} f'(y_{pj}). \tag{2.62}$$

For a first-order polynomial, $\partial net/\partial w$ equals the input activation. The weight change is then simply

$$\Delta w_{kj} = \eta \sum_p^n \delta_{pk} y_{pj} \tag{2.63}$$

for output weights, and

$$\Delta v_{ji} = \eta \sum_p^n \delta_{pj} x_{pi} \tag{2.64}$$

for input weights. Adding a time subscript, the recurrent weights can be modified according to

$$\Delta u_{jh} = \eta \sum_p^n \delta_{pj}(t) y_{ph}(t-1). \tag{2.65}$$

A common choice of output function is the logistic function

$$g(net) = \frac{1}{1+e^{-net}}. \tag{2.66}$$

The derivative of the logistic function can be written as

$$g'(y) = y(1-y). \tag{2.67}$$

For obvious reasons most cost functions are 0 when each target equals the actual output of the network. There are, however, more appropriate cost functions than SSE for guiding weight changes during training [Rumelhart et al., 1995]. The common assumptions of the ones listed below are that the relationship between the actual and desired output is probabilistic (the network is still deterministic) and has a known distribution of error. This, in turn, puts the interpretation of the output activation of the network on a sound theoretical footing.

If the output of the network is the mean of a *Gaussian* distribution (given by the training set) we can instead minimize

$$C = -\sum_p^n \sum_k^m \frac{(y_{pk} - d_{pk})^2}{2\sigma^2} \tag{2.68}$$

where σ is assumed to be fixed. This cost function is indeed very similar to SSE.

With a Gaussian distribution (outputs are not explicitly bounded), a natural choice of output function of the output nodes is

$$g(net) = net. \tag{2.69}$$

The weight change then simply becomes

$$\Delta w_{kj} = \eta \sum_{p}^{n} (d_{pk} - y_{pk}) y_{pj}. \tag{2.70}$$

If a *binomial* distribution is assumed (each output value is a probability that the desired output is 1 or 0, *e. g.* feature detection), an appropriate cost function is the so-called cross entropy,

$$C = \sum_{p}^{n} \sum_{k}^{m} d_{pk} \ln y_{pk} + (1 - d_{pk}) \ln(1 - y_{pk}). \tag{2.71}$$

If outputs are distributed over the range 0 to 1 (as here), the logistic output function is useful (see Equation 2.66). Again the output weight change is

$$\Delta w_{kj} = \eta \sum_{p}^{n} (d_{pk} - y_{pk}) y_{pj}. \tag{2.72}$$

If the problem is that of "1-of-n" classification, a *multinomial* distribution is appropriate. A suitable cost function is

$$C = \sum_{p}^{n} \sum_{k}^{m} d_{pk} \ln \frac{e^{net_k}}{\sum_q e^{net_q}} \tag{2.73}$$

where q is yet another index of all output nodes. If the right output function is selected, the so-called softmax function,

$$g(net_k) = \frac{e^{net_k}}{\sum_q e^{net_q}}, \tag{2.74}$$

the now familiar update rule follows automatically,

$$\Delta w_{kj} = \eta \sum_{p}^{n} (d_{pk} - y_{pk}) y_{pj}. \tag{2.75}$$

As shown in [Rumelhart et al., 1995] this result occurs whenever we choose a probability function from the exponential family of probability distributions.

2.4.4 Tapped delay line memory

The perhaps easiest way to incorporate temporal or sequential information into a training situation is to make the temporal domain spatial and use a feedforward architecture. Information available back in time is inserted by widening the input space according to a fixed and pre-determined "window" size, $\mathbf{X} = \mathbf{x(t)}, \mathbf{x(t-1)}, \mathbf{x(t-2)}, ..., \mathbf{x(t} - \omega)$ (see Figure 2.3). This is often called a tapped delay line since inputs are put in a delayed buffer and discretely shifted as time passes.

It is also possible to manually extend this approach by selecting certain intervals "back in time" over which one uses an average or other pre-processed features as inputs which may reflect the signal decay.

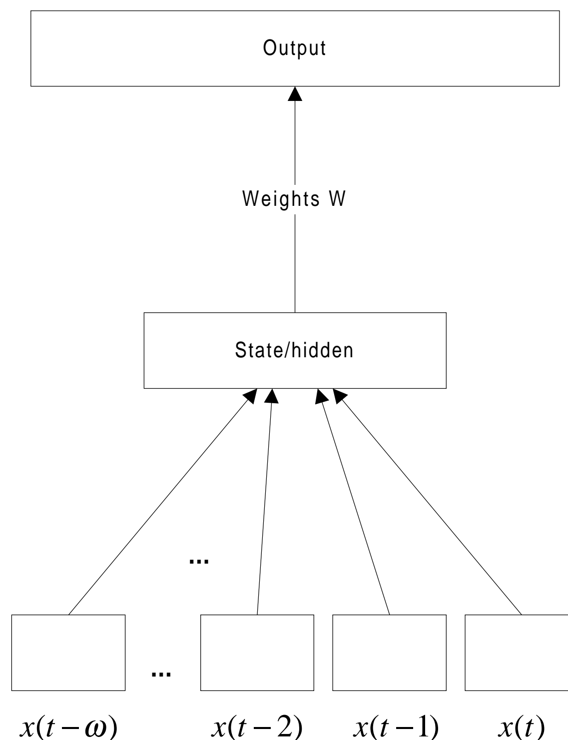

Figure 2.3: A "tapped delay line" feedforward network.

The classical example of this approach is the NETtalk system [Sejnowski and Rosenberg, 1987] which learns from example to pronounce English words displayed in text at the input. The network accepts seven letters at a time of which only the middle one is pronounced.

Disadvantages include that the user has to select the maximum number of time steps which is useful to the network. Moreover, the use of independent weights for processing the same components but in different time steps, harms generalization. In addition, the large number of weights requires a larger set of examples to avoid over-specialization.

2.4.5 Simple recurrent network

A strict feedforward architecture does not maintain a short-term memory. Any memory effects are due to the way past inputs are re-presented to the network (as for the tapped delay line).

A simple recurrent network (SRN; [Elman, 1990]) has activation feedback which embodies short-term memory. A state layer is updated not only with the external input of the network but also with activation from the previous forward propagation. The feedback is modified by a set of weights as to enable automatic adaptation through learning (*e. g.* backpropagation).

Learning in SRNs: Backpropagation through time

In the original experiments presented by Jeff Elman [Elman, 1990] so-called truncated backpropagation was used. This basically means that $y_j(t-1)$ was simply regarded as an additional input. Any error at the state layer, $\delta_j(t)$, was used to modify weights from this additional input slot (see Figure 2.4).

Errors can be backpropagated even further. This is called backpropagation through time (BPTT; [Rumelhart et al., 1986]) and is a simple extension of what we have seen so far. The basic principle of BPTT is that of "unfolding." All recurrent weights can be duplicated spatially for an arbitrary number

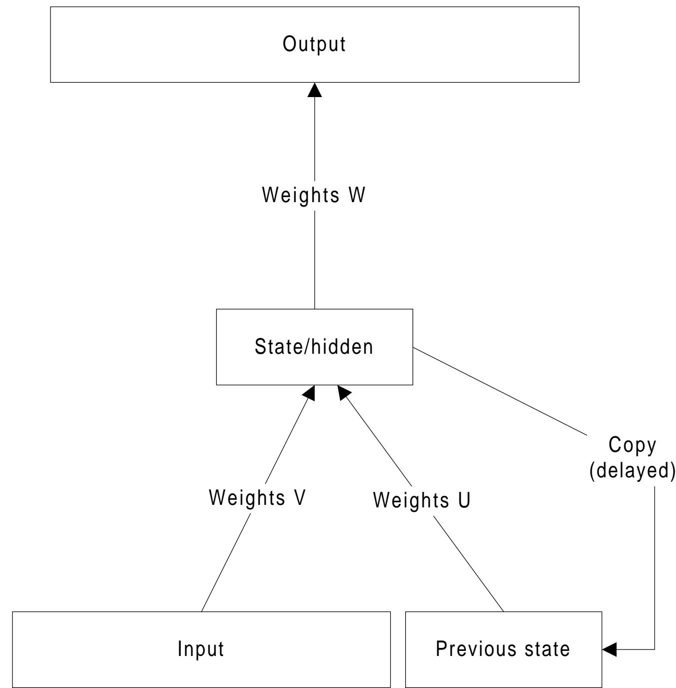

Figure 2.4: A simple recurrent network.

---

of time steps, here referred to as τ. Consequently, each node which sends activation (either directly or indirectly) along a recurrent connection has (at least) τ number of copies as well (see Figure 2.5).

In accordance with Equation 2.62, errors are thus backpropagated according to

$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(y_{pj}(t-1)) \tag{2.76}$$

where h is the index for the activation receiving node and j for the sending node (one time step back). This allows us to calculate the error as assessed at time t, for node outputs (at the state or input layer) calculated on the basis of an arbitrary number of previous presentations.

It is important to note, however, that after error deltas have been calculated, weights are folded back adding up to one big change for each weight. Obviously there is a greater memory requirement (both past errors and activations need to be stored away), the larger τ we choose.

In practice, a large τ is quite useless due to a "vanishing gradient effect" (see *e. g.* [Bengio et al., 1994]). For each layer the error is backpropagated through the error gets smaller and smaller until it diminishes completely. Some have also pointed out that the instability caused by possibly ambiguous deltas (*e. g.* [Pollack, 1991]) may disrupt convergence. An opposing result has been put forward for certain learning tasks [Bodén et al., 1999].

2.4.6 Discussion

There are many variations of the architectures and learning rules that have been discussed (*e. g.* so-called Jordan networks [Jordan, 1986], and fully recurrent networks, Real-time recurrent learning [Williams and Zipser, 1989] etc). Recurrent networks share, however, the property of being able to internally use and create states reflecting temporal (or even structural) dependencies. For simpler tasks (*e. g.* learning grammars generated by small finite-state machines) the organization of the state space straightforwardly reflects the component parts of the training data (*e. g.* [Elman, 1990; Cleeremans et al., 1989]). The

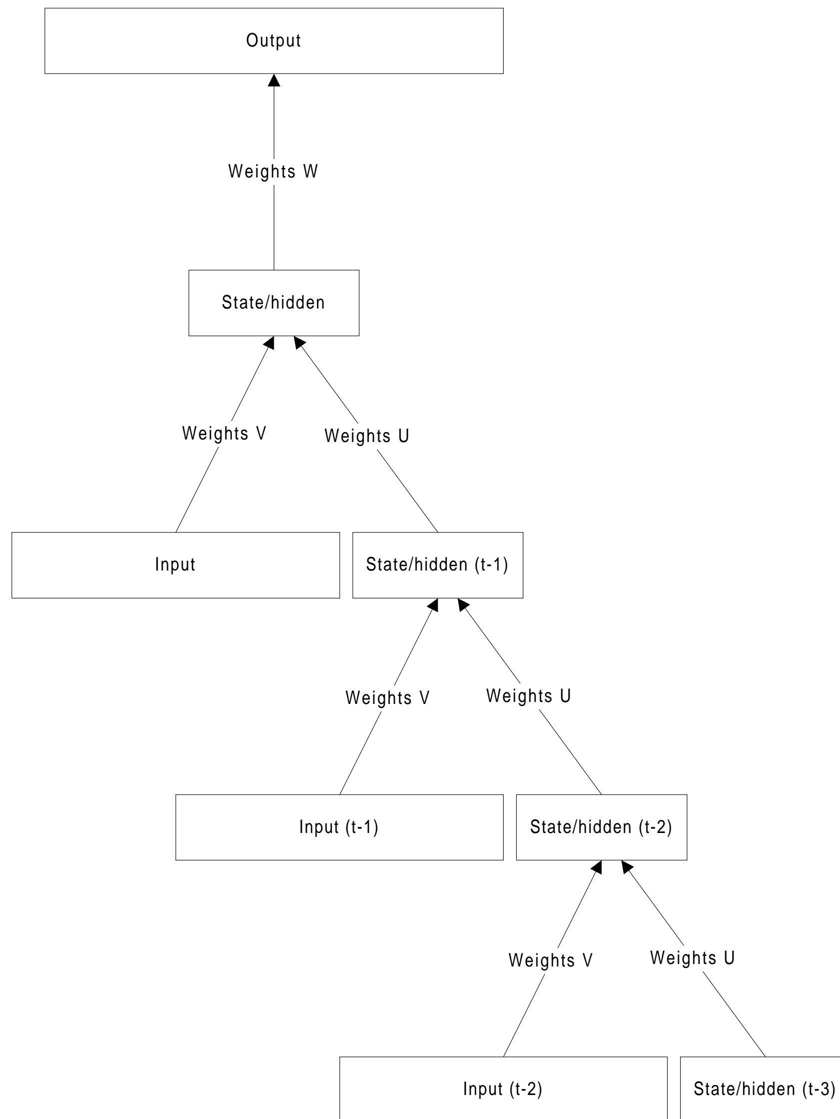

Figure 2.5: The effect of unfolding a network for BPTT ($\tau = 3$).

state space is, in most cases, real-valued. This means that subtleties beyond the component parts, *e. g.* statistical regularities may influence the organization of the state space (*e. g.* [Elman, 1993; Rohde and Plaut, 1999]). For more difficult tasks (*e. g.* where a longer trace of memory is needed, and context-dependence is apparent) the highly non-linear, continuous space offers novel kinds of dynamics (*e. g.* [Rodriguez et al., 1999; Bodén and Wiles, 2000]). These are intriguing research topics but beyond the scope of this introductory text. Analyses of learned internal representations and processes/dynamics are crucial for our understanding of what and how these networks process. Methods of analysis include hierarchical cluster analysis (HCA), and eigenvalue and eigenvector characterizations (of which Principal Components Analysis is one).

2.4.7 References

Barnsley, M. (1993). *Fractals Everywhere.* Academic Press, Boston, 2nd edition.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bodén, M. and Wiles, J. (2000). Context-free and context-sensitive dynamics in recurrent neural networks. *Connection Science*, 12(3):197–210.

Bodén, M., Wiles, J., Tonkes, B., and Blair, A. (1999). Learning to predict a context-free language: Analysis of dynamics in recurrent hidden units. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 359–364, Edinburgh. IEE.

Casey, M. (1996). The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6):1135–1178.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381.

Devaney, R. L. (1989). *An Introduction to Chaotic Dynamical Systems.* Addison-Wesley.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, 48:71–99.

Giles, C. L., Miller, C. B., Chen, D., Chen, H. H., Sun, G. Z., and Lee, Y. C. (1992). Learning and extracted finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405.

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Conference of the Cognitive Science Society.*

Kolen, J. F. (1994). Fool's gold: Extracting finite state machines from recurrent network dynamics. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems*, volume 6, pages 501–508. Morgan Kaufmann Publishers, Inc.

Pollack, J. B. (1991). The induction of dynamical recognizers. *Machine Learning*, 7:227.

Rodriguez, P., Wiles, J., and Elman, J. L. (1999). A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40.

Rohde, D. L. T. and Plaut, D. C. (1999). Language acquisition in the absence of explicit negative evidence: How important is starting small? *Cognition*, 72:67–109.

Rumelhart, D. E., Durbin, R., Golden, R., and Chauvin, Y. (1995). Backpropagation: The basic theory. In Chauvin, Y. and Rumelhart, D. E., editors, *Backpropagation: Theory, architectures, and applications*, pages 1–34. Lawrence Erlbaum, Hillsdale, New Jersey.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by back-propagating errors. *Nature*, 323:533–536.

Sejnowski, T. and Rosenberg, C. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168.

Siegelmann, H. T. (1999). *Neural Networks and Analog Computation: Beyond the Turing Limit*. Birkhäuser.

Tino, P., Horne, B. G., Giles, C. L., and Collingwood, P. C. (1998). Finite state machines and recurrent neural networks – automata and dynamical systems approaches. In Dayhoff, J. and Omidvar, O., editors, *Neural Networks and Pattern Recognition*, pages 171–220. Academic Press.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280.

2.5 Inductive Logic Programming

Lars Asker and Henrik Boström

2.5.1 Introduction

Virtual Predict is a system for induction of rules from pre-classified examples. It is based on recent developments within the field of machine learning, in particular inductive logic programming. In this section, we first give a brief description of the field and then point out the main features of Virtual Predict.

2.5.2 Inductive Logic Programming

Inductive Logic Programming (ILP) is a research area in the intersection of machine learning and computational logic whose main goal is the development of theories of and practical algorithms for inductive learning in first-order logic representation formalisms. From inductive machine learning, ILP inherits its goal: to develop tools and techniques for inducing hypotheses from observations (examples) or to synthesize new knowledge from experience. By using computational logic as the representation formalism for hypotheses and observations, ILP can overcome the two main limitations of classical machine learning techniques (such as decision tree learners): the use of a limited knowledge representation formalism (essentially propositional logic), and the difficulties to use substantial background knowledge in the learning process.

The first limitation is important because many domains of expertise can only be expressed in first-order logic, or a variant of first-order logic, and not in propositional logic. The use of domain knowledge is also crucial because one of the well-established findings of artificial intelligence (and machine learning) is that the use of domain knowledge is essential for achieving intelligent behavior. From computational logic, ILP inherits not only its representational formalism but also its theoretical orientation and various well-established techniques. Indeed, in contrast to many other approaches to inductive learning, ILP is also interested in properties of inference rules, in convergence (*e. g.* soundness and completeness) of algorithms and the computational complexity of procedures. Because of its background, it is no surprise that ILP has a strong application potential in inductive learning. Strong applications exist in drug-design, protein engineering, medicine, mechanical engineering, etc. The importance of these applications is clear when considering that, for example, in the case of drug-design and protein-engineering the results were published in the biological and chemical literature, the results were obtained using a general purpose ILP algorithm and they were transparent to the experts in the domain.

2.5.3 Virtual Predict

Virtual Predict can be viewed as an upgrade of standard decision tree and rule induction systems in that it allows for more expressive hypotheses to be generated and more expressive background knowledge to be incorporated in the induction process. The major design goal has been to achieve this upgrade in a way so that it should still be possible to emulate the standard techniques with lower expressiveness (but also lower computational cost) within the system if desired. As a side effect, this has allowed the incorporation of several recent methods that have been developed for standard machine learning techniques into the more powerful framework of Virtual Predict.

2.5.4 Strategy

There are two main strategies for generating rules from an example file and a theory file: Divide-and-Conquer and Separate-and-Conquer. The former strategy is the same as used by decision-tree learners, allowing most techniques developed within that field to be upgraded to the ILP framework (see following sections). The second strategy is the one adopted by most previous ILP systems. The first strategy works in time linear in the number of examples, while the second works in quadratic time (in the worst case).

The latter may however be more effective than the first in cases where the target is highly disjunctive (see [Boström and Idestam-Almquist, 1999; Boström and Asker, 1999] for further details and a comparison of the two strategies).

2.5.5 Measure

The strategies for generating rules use a measure for choosing among several candidate rules. Methods that use the Divide-and-Conquer strategy can use either the information gain measure [Quinlan, 1986] or adaptive coding measure [Quinlan and Rivest, 1989], while methods that use Separate-and-Conquer can choose between weighted information gain [Quinlan, 1990] or a measure based on the hypergeometric distribution [Boström and Asker, 1999].

2.5.6 Probability estimate

When estimating the probability that an example that is covered by a particular rule belongs to a particular class, two different probability measures may be used by the methods: the La Place estimate and the m estimate (see [Cestnik and Bratko, 1991] for details).

2.5.7 Structure cost

The minimum description length principle according to [Quinlan and Rivest, 1989] may optionally be used both in divide-and-conquer and separate-and-conquer, penalizing extensive search for hypotheses at the cost of information gain according to the chosen measure.

2.5.8 Pruning methods

Some kind of pruning is often necessary in order to avoid the problem of over-fitting the training data. The pruning methods that have been incorporated in Virtual Predict are pre-pruning, post-pruning and incremental reduced error pruning.

Pre-pruning may be used optionally for Divide-and-Conquer and is sometimes desired in order to speed up the induction process and avoid over-fitting. However, it should be used with some care since it may cause the search to stop prematurely.

Post-pruning may be used optionally for Divide-and-Conquer, and it will after having terminated the initial tree structured search select the nodes in the tree that correspond to the highest information gain (*i. e.* possibly considering structure cost). Optionally a fraction of the training examples will not be used when growing the initial set of rules, but will only be used as a validation set for estimating the information gain.

Incremental reduced error pruning can be used optionally for Separate-and-Conquer. This strategy prunes a rule immediately after a search path has been terminated, resulting in a very efficient induction process (c.f., [Cohen, 1995]). The pruning criterion can be set to one of the following: accuracy on the training set (using the probability estimate), accuracy on a separate validation set (fraction of the training examples to be used for this is set by the user) and information gain (using structure cost).

2.5.9 References

Boström H. and Asker L. (1999). Combining divide-and-conquer and separate-and-conquer for efficient and effective rule induction. In *Proc. of the Ninth International Workshop on Inductive Logic Programming, LNAI Series 1634*, pp. 33–43. Springer.

Boström H. and Idestam-Almquist P. (1999). Induction of logic programs by example-guided unfolding. *Journal of Logic Programming* **40**: 159–183.

Cestnik B. and Bratko I. (1991). On estimating probabilities in tree pruning. In *Proc. of the Fifth European Working Session on Learning*, pp. 151–163. Springer.

Cohen W. W. (1995). Fast effective rule induction. In *Machine Learning: Proc. of the 12th International Conference*, pp. 115–123. Morgan Kaufmann.

Quinlan J. R. (1986). Induction of decision trees. *Machine Learning* **1**: 81–106.

Quinlan J. R. (1990). Learning logical definitions from relations. *Machine Learning* **5**: 239–266.

Quinlan J. R. and Rivest R. L. (1989). Inferring decision trees using the minimum description length principle. *Information and Computation* **80**: 227–248.

2.6 The Bayesian modeling tools

Anders Holst

2.6.1 Introduction

The Bayesian modeling tools used at SICS consists of a number of statistical models that can be combined with each other, and used to model a variety of different domains in a very generally applicable way. The methods are mainly the same as are used in a Bayesian neural network [Lansner and Ekeberg, 1989; Kononenko, 1989; Holst, 1997], although they are here used separate from the neural network structure. This makes it possible to build more general models.

The original purpose of the models built is to calculate the probability of some attribute given the other attributes. However, the same models can also be used for prediction, clustering, and likelihood calculations.

The techniques that are used and combined are probabilistic graphical models, mixture models, and Markov models. Bayesian statistics is used throughout to estimate the parameters. The resulting model family includes as special cases such standard methods as the naive Bayesian classifier, the quadratic (or Gaussian) classifier, and a kind of linear regression.

2.6.2 Theoretical background

The original purpose of the models is to calculate probabilities. The probability of some attribute or class y given a vector of attributes $\boldsymbol{x}$ can be written as:

$$P(y \mid \boldsymbol{x}) = \frac{P(y)P(\boldsymbol{x} \mid y)}{P(\boldsymbol{x})} \propto P(y)P(\boldsymbol{x} \mid y) \tag{2.77}$$

Since the denominator $P(\boldsymbol{x})$ is the same for all classes, and the probabilities over all classes has to sum to 1, the rightmost expression can be used by normalizing over the classes. The main objective here is therefore to estimate the distribution $P(\boldsymbol{x} \mid y)$ for each class y as accurately as possible.

If y represents a continuous variable instead of a class, and the model is to be used for prediction of that variable, it is more convenient to estimate the joint distribution $P(\boldsymbol{x}, y)$ instead. The known vector $\boldsymbol{x}$ can then be inserted, and the marginal distribution y calculated from this. Depending on what the result should be used for, one can either calculate the mean and variance of this distribution, or make some other more advanced operation on it.

Now, if the distribution of $\boldsymbol{x}$ is high dimensional or complicated, $P(\boldsymbol{x} \mid y)$ (or $P(\boldsymbol{x}, y)$) can not be estimated directly. The number of degrees of freedom increases exponentially with the number of attributes, and the available data used for training will soon be insufficient. Also if the attributes are continuous valued, some model distribution must be assumed before the estimation, and it should be noted that all distributions are not Gaussian. The idea here is to use the available structure of the domain to break down the distribution in several subdistributions, each of which are easier to estimate.

2.6.3 The naive Bayesian classifier

The first step is to assume independence between the individual attributes in $\boldsymbol{x}$ (given each class y). Then the complete distribution can be expressed as a product of the probabilities of the individual attributes:

$$P(y \mid \boldsymbol{x}) \propto P(y)P(\boldsymbol{x} \mid y) = P(y) \prod_{i=1}^{n} P(x_i \mid y) \tag{2.78}$$

The distribution for each attribute given a specific class, $P(x_i \mid y)$, is significantly easier to estimate. For example, for n binary attributes and two classes, there are only $4n$ probabilities to be estimated, as opposed to 2^{n+1} for the complete distribution. This independence assumption is what is used in the

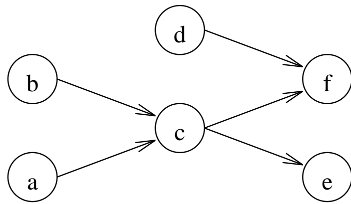

Figure 2.6: A directed dependency tree

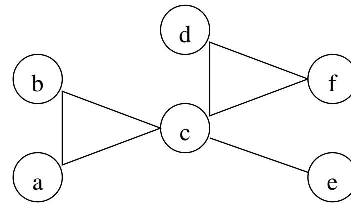

Figure 2.7: A non-directed hyper-graph

---

naive Bayesian classifier. It actually often gives surprisingly good results, in spite of the simplifying assumption that is usually only approximately fulfilled.

The way to think of this classifier is that each input attribute contributes with its evidence for or against each class, and then all the individual evidence is weighted together to the final result. It can not properly handle cases where the combination of two attributes are more important than the sum of considering them separately. In general, it does not account for the dependence between different attributes. Since in most domains there are some dependences between the attributes, this may be a too big simplification.

2.6.4 Probabilistic graphical models

In the situations where the naive Bayesian classifier is to simple, and the correlations between attributes has to be accounted for, one can instead use a *probabilistic graphical model* [Chow and Liu, 1968]. The graph describes how the attributes depends on each other: each node in the graph represents one attribute and each edge represents a dependency between two attributes. (I general a "hyper-graph" would be required, which can contain edges that can each connect three or more nodes, thus representing higher order dependencies between the corresponding attributes.) A dependency graph and can be built by searching for strong correlations in the data. Using such a graph it is again possible to write the complete distribution as a product of simpler distributions, *i. e.* the joint distributions of attributes that are directly dependent on each other according to the graph.

This is the same technique that is used in *Bayesian belief networks* [Pearl, 1988; Lauritzen and Spiegelhalter, 1988; Heckerman, 1995], but the way it is used here is slightly different. For example, here the output attribute y is kept outside of the graph (or rather, all probabilities are conditional on y), whereas in Bayesian belief networks the output attribute is part of the graph. We claim that it is computationally advantageous to keep the class outside the graph, since there is no need to iterate probabilities through the graph in our case. It should also be more robust, since probabilities are calculated in "parallel" rather than in "series", and thus noise will cancel out rather than accumulate.

The product expressions here are somewhat more complicated than for the naive Bayesian classifier. They are best exemplified with an example.

If there are six attributes with dependencies between them as in figure 2.6, it is possible to write the joint probability as:

$$P(\boldsymbol{x}) = P(a)P(b)P(c \mid ab)P(d)P(e \mid c)P(f \mid cd)$$

By rewriting the conditional probabilities as fractions, and using that different parts of the tree are independent, this can be rewritten as:

$$\begin{aligned} P(\boldsymbol{x}) = & P(a)P(b)P(c)P(d)P(e)P(f) \cdot \\ & \cdot \left(\frac{P(abc)}{P(a)P(b)P(c)}\right)\left(\frac{P(ce)}{P(c)P(e)}\right)\left(\frac{P(cdf)}{P(c)P(d)P(f)}\right) \end{aligned}$$

This expression corresponds to the undirected hyper graph in figure 2.7. Every individual attribute, plus every hyper edge in the graph corresponds to one factor in the product.

2.6.5 Markov models

A special case of the above dependency graphs occurs for sequential data. If each attribute is sampled at a number of different times, it is reasonable to expect a strong correlation between successive sample points of each attribute. Then it is natural to assume a Markov chain of some order for each attribute. Again this allows that the distribution over a whole sequence can be written as a product (and fraction) of simpler distributions. In the case of a first order Markov model, this will include joint distributions for consecutive samples from the sequence:

$$P(\boldsymbol{x}) = P(x_n \mid x_{n-1})P(x_{n-1} \mid x_{n-2}) \cdots P(x_2 \mid x_1)P(x_1) = \frac{P(x_n, x_{n-1})P(x_{n-1}, x_{n-2}) \cdots P(x_2, x_1)}{P(x_{n-1})P(x_{n-2}) \cdots P(x_2)} \tag{2.79}$$

This can also be generalized to several dimensions, by using *Markov grids*. It can also be combined with graphs between different attributes.

2.6.6 Continuous valued attributes

If all attributes are discrete the above models can be used directly. However, if one or more attributes are continuous valued this must be handled first. This is done in these tools by starting with Gaussian distributions, and combining these in various ways to build up the appropriate distributions.

The simplest case is when the continuous attributes can be considered as normally distributed directly. If all attributes are Gaussians (including the output), the entire space can be modeled as one multivariate Gaussian distribution. When this model is used for prediction, it is equivalent to the normal linear regression. However, if there are many attributes and too few data, there might be severe over-fitting in this case. Therefore an alternative is to separately model the relation between each input attribute and the output, and then combine these models with the naive Bayesian classifier. It is still a linear models, but less prone to over-fitting, due to the reduced number of degrees of freedom inherent in the independence assumption.

If all attributes except the output are continuous, one simple model is the Gaussian classifier, which models each class (*i. e.* each possible output value) with a Gaussian distribution.

2.6.7 Mixture models

If the continuous attributes have mode complicated distributions than Gaussians, they can be modeled with a *Mixture model* [McLachlan and Basford, 1988]. This means that the distribution is approximated by a sum of a number of simpler distributions, *e. g.* Gaussians (see figure 2.8):

$$P(\boldsymbol{x}) = \sum_{i=1}^{n} P(v_i)P(\boldsymbol{x} \mid v_i) = \sum_{i=1}^{n} \pi_i \cdot f_i(\boldsymbol{x}; \theta_i) \tag{2.80}$$

Here $f_i(\boldsymbol{x}; \theta_i)$ is the ith component of the mixture, with parameters θ_i, and the weight of the component π_i.

Formally any distribution can be approximated sufficiently well by a sum of Gaussians, provided they are many enough. In practice the number of Gaussians used should be small, and the typical situation when this is useful, is when the data consist of a number of clusters, each of which is approximately Gaussian.

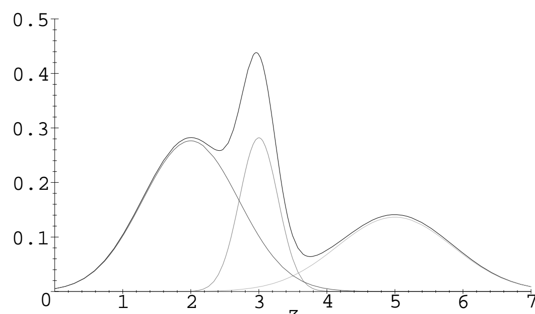

Figure 2.8: A mixture of three Gaussian distributions

To estimate the parameters of a mixture model, an iterative method called *Expectation Maximization* [Dempster *et al.*, 1977] is used. It works by alternately partitioning the data between the components in the mixture and then estimating the parameters of the component from the assigned data.

Mixture models are not limited to Gaussian distribution, but can also combine distributions over discrete attributes, or more complex distributions like graphs and products.

2.6.8 Gamma functions

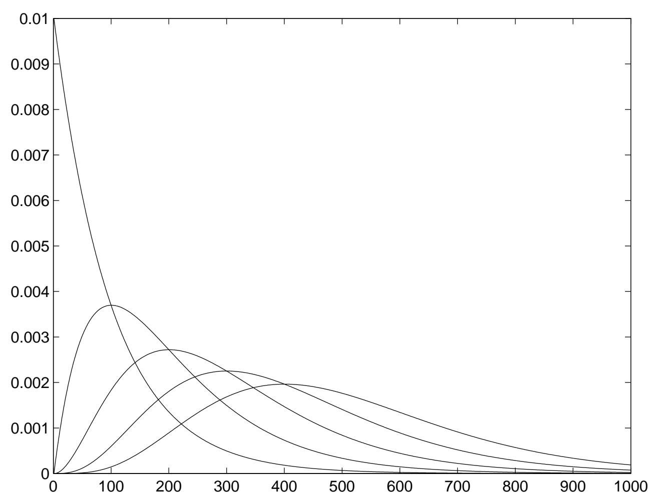

Figure 2.9: The response from basis functions of different time scales

A technique using aspects of both mixture models and Markov models can be used to handle sequential data with information in various time scales. By introducing a number of running average filters, coupled in series, the effect will be that of a number of *Gamma functions*, each sensitive for the information at a certain time scale (see figure 2.9. These can then be combined using a Markov model, and included in the rest of the model.

2.6.9 Bayesian statistics

Throughout these tools, *Bayesian statistics* [Cox, 1946; Jaynes, 1986] are used to estimate the probability distributions. This means that the influence of the data is moderated by priors, which acts as regularizing factors. The resulting models are less prone to over-training, and especially when there are few data the estimates will be less noise sensitive and more reliable than with classical statistics.

The main difference between Bayesian and classical statistics in this context, is how the parameters of the distributions are estimated. If the parameter to estimate is p, and the data is denoted by D, the objective in classical estimation is to find the p that maximizes $P(D \mid p)$, the probability of obtaining the data if that were the true parameter value. This is also called the *likelihood* of the parameter. In Bayesian statistics the goal is instead to find $P(p \mid D)$, the probability distribution for p given the data D. The expectancy of p can be used as the estimation, and the variance of the distribution is a measure of the precision of this estimation.

The relation between the (classical) likelihood and the (Bayesian) probability distribution of the parameter is again given by Bayes theorem:

$$P(p \mid D) \propto P(p)P(D \mid p) \tag{2.81}$$

$P(p)$ is the *apriori distribution* over p, the distribution of it before any data is observed. Typically this can be a uniform distribution, saying that all probabilities are equally likely before any indication from data. When the apriori distribution is multiplied with the likelihood and normalized, the *posteriori distribution* is obtained, *i. e.* the distribution over p given the data.

For example, in the case of a binary variable x, the classical estimate for $p = P(x)$ is:

$$\hat{p} = \frac{n_x}{N} \tag{2.82}$$

where n_x is the number of cases with x out of N cases in total. The Bayesian estimate is:

$$\hat{p} = \frac{n_x + \alpha/2}{N + \alpha} \tag{2.83}$$

where α is a factor determining how much effect the prior should have. When the number of data is large, the effect of α is negligible and the two estimates are almost equal. However, for few data, the Bayesian estimate will avoid the extremes zero and one as probability estimates, because they probably just signal that something is too unusual to have happened yet in the limited data, rather than impossible as the classical estimate would suggest.

Similarly, when a multivariate Gaussian distribution is estimated, the classical estimate tends to over-fit the data and produce very "thin" Gaussians along some axis. The Bayesian prior tends to "span out" the distribution, by using a prior assumption of independence between the attributes.

When a mixture is used, the prior for each component is based on all data, making each component tend to the whole distribution. This prevents individual components from completely disappearing or locking on to single data points, as is otherwise a risk when classical estimates are used in these mixtures.

Bayesian statistics is also useful when two models are to be compared. The likelihood says which model fits the data the best, but a more complex model can usually fit the data better at expense of generalization. The Bayesian prior adds a penalty for the complexity of the model, thus giving an advantage to simple models that explain the data.

2.6.10 Summary

Together these techniques comprise a very powerful toolbox for modeling. Starting with the basic distributions like Bernoulli and Gauss, more complex distributions can be built up by combining these in products, graphs, mixtures, and Markov models. The combination can be done hierarchically: you can for example create mixtures of graphs, or graphs of mixtures. This gives a very high flexibility and the possibility to construct models that can handle complex distributions of very high dimension and with mixed types of variables.

2.6.11 References

Chow C. K. and Liu C. N. (1968). Approximating discrete probability distributions with dependency trees. *IEEE Trans. Information Theory* **14**: 462–467.

Cox R. T. (1946). Probability, frequency and reasonable expectation. *American Journal of Physics* **14**: 1–13.

Dempster A. P., Laird N. M., and Rubin D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society B* **39**: 1–38.

Heckerman D. (1995). A tutorial on learning Bayesian networks. Technical report TR-95-06, Microsoft Research, Redmond, WA.

Holst A. (1997). *The Use of a Bayesian Neural Network Model for Classification Tasks*. PhD thesis, dept. of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden.

Jaynes E. T. (1986). Bayesian methods: General background. In Justice J. H. (ed.), *Maximum Entropy and Bayesian Methods in Applied Statistics*, pp. 1–25. Cambridge University Press, Cambridge, MA. Proc. of the fourth Maximum Entropy Workshop, Calgary, Canada, 1984.

Kononenko I. (1989). Bayesian neural networks. *Biological Cybernetics* **61**: 361–370.

Lansner A. and Ekeberg Ö. (1989). A one-layer feedback, artificial neural network with a Bayesian learning rule. *Int. J. Neural Systems* **1**: 77–87.

Lauritzen S. L. and Spiegelhalter D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Statistical Society B* **50**: 157–224.

McLachlan G. J. and Basford K. E. (1988). *Mixture Models: Inference and Applications to Clustering.* Marcel Dekker, New York.

Pearl J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Francisco, CA.

2.7 Self-organizing feature maps

Mikael Hall and David Martland

2.7.1 Introduction

Clustering data can be done for a number of reasons, but the act of clustering can be seen as a way to make a smaller description of the data, than that given by the actual collection of samples. Thus it is possible to view clustering as simply being data compression. Perhaps we want to compress data images, or we may want to make some kind of analysis of the sample set. Whatever our motifs are, we need a framework in which the description is given. The crucial difference between say, describing the data by its statistics alone and clustering is that the statistics is only fruitful if we already have a context which give them meaning. We have to have some valid way of knowing when the statistics are good or bad and so on. Merely saying that the variance is one, is not enough.

Suppose we have a data set which in some sense is viewed to be unique so that relations to other sets is complicated or not relevant, like a picture or measurements of a system. Then we must find a framework within the data itself. A histogram is a clustering method, which replace talk about individual samples with properties of bins which are evenly distributed in some portion of the space of vector values representing the samples. Each bin contains some nonnegative number of samples. This way we can talk about groups of samples in terms of bins and compare different groups (bins). The number of bins, location and the width of the bins will consequently determine what is displayed about the data. Clearly it may be beneficial if we in some way can let the things which can be displayed affect the framework by which we view the data, instead of the other way around. In particular we would like the data to determine the width and the location of individual bins, replacing one general design choice regarding the bins, made by us, with parameters unique to individual bins, tuned by the data. Histograms just cut the data into parts and displays whatever can be seen between parts, thus hiding almost everything only describable by differences smaller than the width of the bins and/or if it is out of phase relative the location of the bins.

2.7.2 K-means clustering

K-means clustering differs from histograms in that only the number of bins is determined by the user, assuming the notion of distance is set. The volume of individual bins and their location is determined by the data itself, within the bounds given by the number of bins. Many small bins occur in regions of high sample density. This makes sense in data compression. The algorithm minimize the following cost function:

$$\sum_{j=1}^{K} \sum_{i \epsilon C_j} \|\mathbf{x}_i - \mu_j\|^2,$$

where the N_j samples $\mathbf{x}_i$ belonging to one of the K bins (clusters) C_j is used to calculate the corresponding center (mean, centroid) μ_j of the bins by

$$\mu_j = \frac{1}{N_j} \sum_{i \epsilon C_j} \mathbf{x}_i.$$

We are thus trying to find K means so that the summed (Euclidean) distance, or dissimilarity, between each sample and its nearest mean is minimized. The search procedure is one of two types. In the batch type, shown above, K clusters of samples are formed randomly, new means are calculated and the samples are appropriately rearranged. This continues recursively until the means have settled. In sequential search, K points are randomly chosen and samples are picked one by one attracting the most similar mean.

The driving force behind the scheme is to avoid counting components of the distances to a degree depending on the number K, the data given and the already found sequence of means. We can see this by fixing one sample and its nearest centroid. From the centroid's point of view this sample could be replaced by any points lying on the surface of a hypersphere centered on the mean and with the same radius as the distance to the given sample, since there would be no change in the associated cost. Given $K = 1$, imagine that the samples are symmetrically distributed around a circle and then let one point on the circle represent two samples (add a duplicate). Then the mean would move towards this point, thereby reducing a double cost. To illustrate this further, but also how the history of the search may affect the quality of the solution found, imagine a data set consisting of the corners of a ordinary dice (a unit hypercube of dim 3). If $K = 1$, the center of the cube would be ideal and found (with a total cost of 6), being the center of a hypersphere (of dim 3) and with the corners lying on the surface of that hypersphere, while if $K = 2$, two disks lying on opposite sides of the hypercube, would be ideal bins (with the four corners on the corresponding side on their surfaces), resulting in a total cost of 4. A non-ideal solution would be to choose one corner as the second mean in the ideal solution of case $K = 1$, obtaining a total cost more than 4. If we during the search procedure get to close to the suboptimal solution (when the bipartition of the corners coincide with that of the solution), we will be drawn into it, as into a black hole, since no corner affects any but the nearest centroid.

One way to view the K-mean clustering algorithm would be to regard it as a simple model of a economy with social goals. Each mean would then be an individual, who is trying to exist at a minimal cost, while at the same time forced to share wealth (lack off cost). In an economy without taxes etc, each mean would pick out one sample only and the descriptive task is abandoned. Although economic experiments almost suggest them self, we are not taking this view in order to make political statements. We are trying to make a point regarding the difference between the K-mean and the self-organizing feature map. In a real economy people are trying to maximize their wealth (minimize the cost of living) by doing what they are or feel they are most suited to do. But they are also cooperating with other individuals, not only through the tax they pay, but also directly. They form companies and so on. This is not modelled by the K-mean.

2.7.3 Self-organizing feature maps

As in the previous section, we are viewing the means to be individuals with different skills represented by their prototype vector. During the search which we conduct, we can view the K means as growing up, developing skills, learning to cooperate with others and thus finding a place in life. The data samples are thus rewarding the most fit individual during the search, by lowering its cost. Cooperation, according to the means, could consist in sharing wealth to other means. Now this is not the case. Cooperation means sharing knowledge. The rewarded mean is saying, "follow me, you might be rewarded". And the receivers of this information, being individuals believing in the Bayesian view of statistics, follows.

The above view is very picturesque, but the real motivation behind self-organizing feature maps is not social behavior of people. Instead, it is meant as a model of how neurons in certain regions of the brain cooperate, where stimuli seem to excite groups of neurons which are close to each other in that region of the brain. So instead of allowing the information pathways to change during the course of life (according to the data), they are most often fixed or changed only in a fixed way. The self-organizing feature map can in addition be viewed as introducing the concept of identity, compared to the K-mean. Passing information to others, but not all, assumes an address system. This address system consists of coordinates, which remain unchanged. The information is passed along according to a distance function in the most often two dimensional space of coordinates. The means are placed in this space so that they form a regular grid (hexagonal or other). Before mathematics of the generic kind is presented, we want to point out that there are variants, which do updates the information pathways according to the data.

The mathematics, where we start with the definition of the best-matching unit of the map grid, $\mathbf{m}_b$, is:

$$\|\mathbf{x} - \mathbf{m}_b\| = \min_i\{\|\mathbf{x} - \mathbf{m}_i\|\},$$

where $\mathbf{x}$ is one randomly chosen sample. The learning rule used at each sample presentation t updates every map unit by

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + \alpha(t)h_{bi}(t)[\mathbf{x} - \mathbf{m}_i(t)].$$

Note that these equations exactly describe the sequential K-mean if the neighborhood kernel in the map, h_{bi}, satisfy $h_{bi} = 1$ iff $\mathbf{m}_b = \mathbf{m}_i$ and if it is zero otherwise, for all t.

2.7.4 Common usages

The Self-organizing feature map is used for different reasons. The main areas of use are:

1. Visualization and clustering of data sets, most often in the early stages of correlation hunting.
2. Modelling, either directly or to direct samples to different submodels.
3. Data preparation. This include replacement of missing values, noise reduction and data compression.

These task are achieved in a robust manner, due to the fact that every map unit is a weighted mean of all samples, through the neighborhood kernel. This also produce some disadvantages like, data range contraction and the creation of interpolating units disturbing correlation cues. But all in all, self-organizing feature maps constitute an efficient multipurpose datamining tool.

2.7.5 Further reading

Vesanto, J. (1997). Data mining techniques based on the self-organizing map. Master's thesis, Helsinki University of Technology.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43, 59-69.

Kohonen, T. (1982). *Self-organization and associative memory.* Berlin: Springer-Verlag.

Moody, J. and Darken, C.J. (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation* 1 (2), 281-294.

Ritter, H., Martinetz T. and Schulten, K. (1988). *Neural Computation and Self-Organizing Maps: an Introduction.* Addison-Wesley, Reading, MA.

2.8 Genetic Algorithms

Mikael Hall and David Martland

2.8.1 Introduction

As the name suggest, genetic algorithms mimic and tries to harness the search power of evolution. The success of evolution as a search for solutions may or may not be taken as evident. However, biological differentiation has taken place which often do constitute highly optimal solutions relative to the surrounding environments. This specialization often takes place in relatively few generations, compared to the vast number of conceivable solutions. The relation between evolution and genetic algorithms is a two-way relation, where computer experiments has been made to investigate things like fitness versus sexual appeal. In machine learning, genetic algorithms is suitable when the solutions sought are fairly complex. It has been used to find sets of rules, to design artificial neural networks and even to find and construct computer programs.

The ideas behind genetic algorithms is very simple analogies drawn from evolution, such as fitness, sex and disease. These operations are combined with a simple trial and error methodology. Intuitively genetic algorithms is just a way to test and rank solutions. But they often seem to work much more efficiently than what can be expected from this view, and this can be explained by a simple analysis of the way genetic algorithms work.

2.8.2 Fitness

At the heart of genetic algorithms lies the concept of fitness. The first thing to do when designing a genetic algorithm is therefore to decide in what sense individuals or solutions shall be considered to be fit or good. This choice implicitly induce some ordering, total or partial, of the possible solutions. While trivial at times, this is generally a hard, but crucial step. We must direct the search toward nothing but the intended things, without excluding acceptables. This amounts to specifying our needs in terms of some sort of distance measure-we must be able to measure how far different alternatives are from being an acceptable solution, at least we must recognize improvements.

According to rank or some other function of the chosen fitness, candidate solutions are then allowed to participate in making other solutions in the next generation and to survive or die. Also some random mutations and/or some immigration of solutions are allowed to occur. In this way genetic algorithms may manage to both search for new types of solutions and to improve upon the "good" ones. There will be a crucial trade of between these two principal tasks. One must carefully decide how fast the algorithm is allowed to develop a bias versus maintaining some degree of variance.

2.8.3 Variance and bias

Often solutions consists of smaller parts and in genetic algorithms these parts are combined in various ways. The principal way to combine partial solutions is called *crossover*. This means that the representation of two candidate solutions are each divided into smaller parts and mixed solutions are formed. The degree of bias (or variance) can be seen to be the closely related to the number of partial solutions the algorithm holds on to, the rate at which it sees new ones and the novelty of these solutions. Individual partial solutions have a good chance to contribute to biased search if they are contained in solutions which are ranked high. This effect will depend on the amount of randomness in the algorithm. Randomness is built into genetic algorithms in different ways. The crossover, mutation and survival is made to depend upon random variables and the quality of the fitness function may also introduce some randomness. Through fitness very large partial solutions can survive so that optimal solutions can be found. Partial solutions can also be copied more times if they are small, because the probability of being cut into pieces by crossover or being modified by mutation increase with size; that is, with encoding length. This suggest that the degree to which good solutions are investigated depend upon the design of the crossover operation, thus increasing variance "locally", while increasing bias "globally". The crossover operation

works as a Occam's razor, trying do extract the working parts in fit solutions. The fittest try to sell their solution, while crossover wants to ensure that unnecessary goods aren't bought.

2.8.4 Important dimensions

Although the different aspects of genetic algorithms are clearly highly interconnected, we can separate them at least in the first generation. Then we can view the sequence of generations as being a Bayesian search, where prior knowledge direct the search and in which knowledge is revised in the light of new evidence. The dimensions in which room for development is made, are given by the number of "atomic" parts individuals are made of, as well as the number of individuals competing at each generation and the number of generations. These three dimensions determines the volume investigated. The number of generations determine the pure random nature of the search, while the size of the population determine the degree to which comparison is exhaustive. The genetic operators add additional power with the number of atomic parts, by exploring the revised knowledge in the light of evidence gained by that generation.

2.8.5 Usability

The main reason genetic algorithms are so interesting is that they are capable of exploring different regions simultaneously and they can move abruptly to new areas. Hence local minima won't impair the search, in the same way as gradient descent. However a similar problem can arise if a strong bias is developed too soon. This is called *crowding out* and happens when the "road to success" of the contenders is blocked by the most fittest. One simple way to avoid this is by ranking the solutions and letting this ranking determine the probability of parenthood. This way the degree of superiority do not matter and the less fit are given more room, provided the fitness values has high variability. If not, ranking will give more room to the fittest. The way in which fitness is used in revising the prior knowledge is therefore crucial, as already noted.

The importance and usability of genetic algorithms can be expected to increase with cheap computing power. High dimensional data sets increase the need for automatic datamining tools, as well as for a more diverse set of robust optimization techniques. Genetic algorithms can also easily be parallelized, boosting the applicability further.

2.8.6 Further reading

Booker, L.B., Goldberg, D.E. and Holland, J.H. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, 40, 235-282.

Box, G. (1957). Evolutionary operation: A method for increasing industrial productivity. *Journal of the Royal Statistical Society*, 6(2), 81-101.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, machine learning*. Reading, MA: Addison-Wesley.

Holland, J.H. (1962). Outline for a logical theory of adaptive systems. *Journal of the Association for Computing Machinery*, 3, 297-314.

Jones, A.J. (1993). Genetic algorithms and their applications to the design of neural networks. *Neural Computing and Applications*, 1, 724-735.

Whitley, L.D. and Wose, M.D. (Eds.). *Foundations of genetic algorithms 3*. Morgan Kaufmann.

Zbignew, M. (1992). *Genetic algorithms + data structures = evolution programs*. Berlin: Springer.

2.9 Information theory

Anders Holst

2.9.1 Introduction

Here we will briefly present the information theory concepts that have been used in the DALLAS project. The focus is on how to detect strong correlations between attributes in the domain.

2.9.2 Correlation measures

Consider the task of measuring the correlation between two attributes in the data. That is, we have two series of measurements, and want to know how much they are dependent on each other. If there is a complicated process generating several series of measurements, it may be very useful to find out how they depend on each other, to get a better understanding of the process. Another use is to select the most relevant series when some other attribute of the process is to be predicted.

Unfortunately, there is no single best way to measure the correlation that works in all cases. There are however a number of common measures used in different situations:

Correlation between binary variables:

$$P(x_1, y_1) - P(x_1)P(y_1) \tag{2.84}$$

$$\frac{P(x_1, y_1)}{P(x_1)P(y_1)} \tag{2.85}$$

$$\frac{P(x_1, y_1)P(x_2, y_2)}{P(x_1, y_2)P(x_2, y_1)} \tag{2.86}$$

$$\frac{P(x_1, y_1) - P(x_1)P(y_1)}{\sqrt{P(x_1)P(x_2)P(y_1)P(y_2)}} \tag{2.87}$$

Correlation between continuous variables:

$$\frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{2.88}$$

$$\frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \tag{2.89}$$

The simplest case is for two binary variables x and y (where x_1 and x_2 in the equations above are the two possible outcomes of x). If x and y are independent, then $P(x_1, y_1) = P(x_1)P(y_1)$. This means that a natural test for dependency is to compare $P(x_1, y_1)$ with $P(x_1)P(y_1)$, which is what is done in both (2.84) and (2.85). Equation (2.86) is called the "interplay" between x and y, and is also useful in some contexts. Equation (2.87) is the usual correlation coefficient for binary variables.

When it comes to continuous variables, the most commonly used measure is the linear correlation coefficient, equation (2.88). (Equation (2.88) is of course a special case of this when the variables can only take the values 0 and 1.) If the interesting entity is instead the "slope" of the linear correlation (with y considered as a function of x), the regression coefficient (2.89) is used instead.

One problem with all the above measures is that they only consider the linear part of the correlation. This means that important dependencies can be missed, just because their linear components are too small. The classical example is a number of points lying on a circle. The two coordinates are dependent on each other, but nevertheless the correlation coefficient are zero.

Another problem is what to do with ternary variables, or in general variables without any natural ordering between the outcomes. In such cases there are no natural generalization of the correlation coefficient.

This is where information theory offers a solution.

2.9.3 Information

Information in this context should not be confused with "knowledge" or "meaning". It is merely a measure of the smallest number of bits required to transmit messages. One way of putting it as the smallest average number of bits required to inform someone whether an event has happened or not. An event that always happens (with probability 1) contains no information at all: we already knew that it must have happened before we got the message. A very unusual event on the other hand, which causes much surprise, contains a lot of information.

The formal definition of the information of a stochastic variable A (representing an event that can happen with probability $P(A)$) is:

$$\mathrm{Info}(A) \equiv -\log P(A) \tag{2.90}$$

This definition is indeed very natural: Information should be additive. If there are two independent events A and B, and we first get to know A (giving us $\mathrm{Info}(A)$ bits of information) and then B (giving us another $\mathrm{Info}(B)$ bits), the total information is $\mathrm{Info}(A) + \mathrm{Info}(B)$ bits. If the events are independent, the probability for both A and B to happen is $P(A, B) = P(A)P(B)$. The only function (up to a constant factor) that in this way can make a product into a sum, is the logarithm:

$$\mathrm{Info}(A, B) = -\log P(A, B) = -\log P(A) - \log P(B) = \mathrm{Info}(A) + \mathrm{Info}(B)$$

The minus sign makes the information positive (since the logarithm is negative for numbers less than 1), or zero for probabilities equal to 1. It does not matter which base we use for the logarithm, since this only affects which units of information are used. Base 2 gives the information in bits, but another common choice is to use the natural logarithm, giving the unit "nats".

Another important concept here is the *entropy* of a variable. It is the average information of that variable, *i. e.* the expectation of the information:

$$H(X) \equiv -\sum_k P(x_k) \log P(x_k) \tag{2.91}$$

where x_k is the k:th possible outcome of the variable. For two independent variables it again holds that $H(X, Y) = H(X) + H(Y)$. However, what happens if X and Y are not independent?

If X contains some information about Y, this means that $H(X) + H(Y)$ is greater than $H(X, Y)$, since a part of the information in $H(X, Y)$ is contained in both $H(X$ and $H(Y)$, and therefore is counted twice when they are added. The part of the information that X and Y has in common, is called the *mutual information* between X and Y:

$$\begin{aligned} I(X, Y) &\equiv H(X) + H(Y) - H(X, Y) \\ &= -\sum_{i,j} P(x_i, y_j) \log \frac{P(x_i, y_j)}{P(x_i)P(y_j)} \end{aligned} \tag{2.92}$$

2.9.4 Mutual information used as a correlation measure

As opposed to the correlation coefficient, the mutual information is zero if and only if the variables are independent. It is not fooled by a linear component that is zero. The mutual information can also be generalized to any kind of probability distributions, and is not limited to any special cases. This makes it suitable as a general way to measure the correlation between variables.

There are a few practical considerations though: To calculate the entropy of a variable, its distribution must be known. In the binary case, or more generally a discrete variable with a finite number of outcomes, estimating this distribution is straightforward, since it amounts to estimating the probability of each outcome. The only worry is that when the number of outcomes is large, the number of training data may not be sufficient to make a reliable estimate of the probability for each outcome. However, for continuous valued attributes more care has to be taken when estimating the distribution. Some clue as to what form the distribution has is required, *i. e.* we need a *model*. If a too simple model is selected important aspects of the real distribution can be missed. For example, if a Gaussian distribution is assumed, estimation is again straightforward, but making this assumption also means that we can not detect any nonlinear parts of the correlation. On the other hand, if a too complex model is selected, two variables will always seem correlated due to overfitting, even if they are independent.

A method which does not require any further knowledge of the exact form of the distribution, is to make a histogram. A grid is placed in the joint space for the two variables that should be checked, and the probability of a data point being in each slot is estimated and fed directly into the equation for mutual information. It is still necessary to select a suitably fine grid. Too many slots will again cause overfitting, making the mutual information tend to the logarithm of the number of data points regardless of where they are. Making a too coarse grid again misses important information. (In specific, using just two slots per variable will again only detect a linear part of the correlation). The appropriate number of slots per variable depends on the amount of available data – enough data points must go into each slot to give a reliable estimate.

Another trick can be used here, which makes the method less sensitive to outliers: The slots need not be of the same size. One can use histogram equalization to adjust the slot sizes for each variable until they each get equally many data points. The grid resulting from both variables will for dependent variables not divide the data points evenly, and it is this deviation from an even distribution that is measured by the mutual information in this case.

Just as a final detail, a relation between mutual information and the normal correlation coefficient can be noted for the case where a linear dependency (or more specifically, Gaussian distributed data) can be assumed. The mutual information in this case can be expressed as:

$$I(X, Y) = \frac{-\log(1-r^2)}{2}$$

where r is the correlation coefficient. While the correlation coefficient lies between -1 and 1, the mutual information goes from 0 to $+\infty$, and makes no difference between positive and negative “slopes” of the correlation.

2.9.5 Time series

Finally we will consider how information theory can be used to handle time series. Suppose that we have a sequence of measurements of some variables, *e. g.* from some process, and again we want to know how dependent they are.

The first impulse is to pair values from the same time of the two variables, and calculate the correlation between these pairs as usual, either with mutual information or some other correlation measure. But there are some problems with this approach. First, successive measurements are not independent of each other. If a variable has a high value at some time, it may be more likely to be high also a few time steps before and after this. This magnifies the impact of coincidences in data: If the first variable just happens to be low at some time when the other variable is high, they are likely to remain in that relation for

several time steps. So even if this was just a coincidence between two random fluctuations, there will be a large numbers of pairs of values with this relation contributing to the correlation measure, making it seem more significant than it is.

The second complication is of course that the variables can depend on each other with some time delay. This is usually handled by align the series with different time delays, and select the delay that gives the highest correlation. The problem is that the dependency can be spread over several steps, in which case the total correlation is not found regardless of which delay is selected. In conjunction with the time dependency within the series, this causes the "peaks" in the correllogram to be very wide, and hard to locate with precision.

One solution here is to use the *entropy rate* of a time series. Instead of considering the information in each time step separately, it looks at how much *new* information that is conveyed at any moment in average, *i. e.* the information given the entire series at earlier times:

$$\begin{aligned} &H(X_t \mid X_{t-1}, X_{t-2}, \dots) = \\ &\quad -\sum_k P(x_{k,t} \mid x_{k,t-1}, x_{k,t-2}, \dots) \log P(x_{k,t} \mid x_{k,t-1}, x_{k,t-2}, \dots) \end{aligned} \tag{2.93}$$

This can be generalized to the *mutual information rate*:

$$\begin{aligned} I_{rate}(X,Y) &= H(X_t \mid X_{t-1}, X_{t-2}, \dots) + H(Y_t \mid Y_{t-1}, Y_{t-2}, \dots) - \\ &\quad - H(X_t, Y_t \mid X_{t-1}, Y_{t-1}, X_{t-2}, Y_{t-2}, \dots) \end{aligned} \tag{2.94}$$

Unfortunately, the entropy rate of a time series can not be calculated directly. The probability distributions that is to be used contains an unlimited number of variables (*i. e.* the same variable but at an unlimited number of time steps). This is of course impossible to estimate from a limited amount of data. The solution here is to make a Markov assumption, *i. e.* that the value of a variable depends directly only on the preceding value:

$$P(X_t \mid X_{t-1}, X_{t-2}, \dots) = P(X_t \mid X_{t-1}) \tag{2.95}$$

This is a reasonable assumption for many processes, at least as an approximation. This again leaves us with a distribution over only two variables when calculating the entropy rate.

However, the last term in (2.94) is still a problem. As long as it is conditioned on the entire series, time delays doesn't matter. Even if it takes ten time steps for a change in X to affect Y, ten steps old values of X will be included in the conditioning, and thus taken care of. However, when making a Markov assumption for the joint XY series, we must find the right time delay, or the assumption will not hold. In practice one has to try different time delays d and select the one with lowest $H(X_t, Y_{t-d} \mid X_{t-1}, Y_{t-1-d})$, *i. e.* highest mutual information rate. Dependencies between the two series that are spread out over several time steps will be handled correctly, if the spread is due to the Markov properties (*i. e.* that the value at one point in time depends on the previous value). However, if the series instead interact at two or more different delays (for example changes that have several effects but at different speeds, or feedback loops such that both variables affect each other) this will not be handled, but only the strongest dependency will be considered. In practice the hope is that this will be sufficient though.

Both the mutual information and the mutual information rate has been used to select variables and find the strongest dependencies between variables in this project.

2.10 Ensembles, Boosting and Bagging

Lars Asker and Henrik Boström

2.10.1 Ensemble learning

The generation of ensembles (*i. e.* sets of hypotheses) whose predictions are combined has been demonstrated to improve accuracy in many domains. The ensemble learning methods boosting, bagging and randomisation may be used in Virtual Predict in conjunction with the Divide-and-Conquer strategy. All of them require the user to specify the number of iterations (*i. e.* the number of hypotheses to be generated).

2.10.2 Boosting

Boosting is an ensemble learning method that uses a probability distribution over the training examples, that on each iteration is re-adjusted so that the learning algorithm focuses on those examples that have been incorrectly classified on previous iterations. The method used in Virtual Predict is called AdaBoost (see [Freund and Schapire, 1996] for details). There is an option to tell the system to use so-called stumps only, which allow for faster induction and more compact hypotheses.

2.10.3 Bagging

Bagging is a method that works by creating a number of bootstrap replicates of the training set which are used for generating the hypotheses, and where the entire set of training examples is used as a validation set (see [Breiman, 1996] for details). This strategy has shown to be particularly effective in noisy domains.

2.10.4 Randomization

Randomization works by choosing alternatives according to a probability distribution that is based on the information gain. In this way the hypotheses in the ensemble are slightly varied, and the combined prediction is often more accurate then the prediction made by the single best hypothesis. This strategy may also be used in conjunction with bagging.

Another way of producing ensembles by randomization in Virtual Predict is to generate hypotheses by repeatedly splitting the training examples into a grow set and a validation set (see section on pruning methods).

2.10.5 References

Breiman L. (1996). Bagging predictors. *Machine Learning* **24**: 123–140.

Freund Y. and Schapire R. E. (1996). Experiments with a new boosting algorithm. In *Machine Learning: Proc. of the 13th International Conference*, pp. 148–156. Morgan Kaufmann.