TEKNILLINEN KORKEAKOULU
Teknillisen fysiikan
koulutusohjelma

ERIKOISTYÖ
T-61.195 Informaatiotekniikan erikoistyö I
July 5, 2004

Feature set selection for inferring relevance from eye movements

Ilkka Kudjoi
58117T

Contents

1 Introduction **1**

2 Experimental Arrangements **1**

2.1 Experimental setup 1

2.2 Measuring equipment 1

2.3 Feature extraction 1

3 Learning a neural network **2**

3.1 Multi-Layer Perceptron neural network 2

3.1.1 Evaluating perceptron values 3

3.1.2 Activation function 3

3.2 Learning MLP 3

3.2.1 Error function 3

3.2.2 Back-propagation 3

3.2.3 Maximum Likelihood 4

3.2.4 Cross-Validation 4

3.3 Bayesian learning 4

3.3.1 Bayes' rule 4

3.3.2 Selection of a prior distribution 5

3.3.3 Automatic Relevance Determination (ARD) -prior . . . 5

3.4 Markov Chain Monte Carlo -integration 6

3.4.1 Gibbs Sampling 6

3.5 The hybrid Monte Carlo algorithm 6

3.5.1 The Metropolis algorithm 7

3.5.2 Energy formulation 7

3.5.3 Hamiltonian mechanics 7

3.5.4 Leapfrog discretization 8

3.5.5 Hybrid Monte Carlo method description 8

4 Experiments **8**

4.1 Radford Neal's Software for Flexible Bayesian Modeling 8

4.2 Data analysis 9

4.3 Results and discussion 9

5 Acknowledgments **9**

A Eye movement features **11**

B Documentation for use of FBM **12**

B.1 Dividing data into validation sets 12

B.2 Command syntax 12

B.3 Analyzing the results 13

1 Introduction

Assume that a person is reading a text from a computer display and she browses through words and rows in search of something interesting from the text. To follow interesting information, the person must click or check out another page. This and many other intentions of a reader could be inferred from reader's eye movements, i.e. we assume that the the scan path of the eye of the reader is different when she notices something which she finds interesting.

> If we were able to infer human intentions from eye movements in general, we could ultimately create personal assistants.

When reading, reader's eyes don't move smoothly along the rows. They stop (fixate) on some words. After a few hundred milliseconds the eye rapidly jumps (makes a saccade) to another point. The behavior is due to the fact that the field of accurate vision spans only a small area in the central fovea. When the eye makes a saccade, hardly any observations are made.

To study the eye movements, the movements are tracked in a setup where the test person is asked to find certain information of twelve shown document or news titles. We can extract tens of different variables from the eye movement data including pupilla diameter, saccade length, position relative to the words and lines and different counts.

Combining the variables with relevance data we will learn a system that predicts the most relevant title among others. The learning system in this work will be Multi-Layer Perceptron Network (MLP) [3, 12, 8, 1], and it will be learned with Bayesian methods [5, 11, 2] and Markov Chain Monte Carlo (MCMC) -integration [2, 10]. The methods and data retrieval will be discussed in the next two sections.

The work was carried out within Proactive Information Retrieval [7, 6] by Adaptive Models of User's Attention and Interests (PRIMA) research project. The project is part of the proactive computing (PROACT) research programme funded by the Academy of Finland.

The data used in this experiment was acquired through the PRIMA project by co-operating with CKIR (Helsinki School of Economics: Center for Knowledge and Innovation Research).

2 Experimental Arrangements

2.1 Experimental setup

In this setup each of three subjects was shown a question and twelve titles. The subjects were asked to pick the number of the title that handled the same topic as the question. One of the titles included the answer to the question, three of them had some information about the question and the last eight titles were irrelevant.

In the meanwhile, when the subjects were answering the questions, together 20 tasks, the subjects' eye movements were measured with a gaze tracking system. The tracking system measured the location of the pupilla and corneal with frequency of 50Hz, and the raw data was then segmented to a sequence of saccades and fixations by the software from the manufacturer of the tracking system.

2.2 Measuring equipment

The measuring equipment consists of a headset (Figure 1) with a camera and a semi-transparent mirror. The headset was iView's gaze tracking system from Sensomotoric Instruments Gmbh. When using the headset the subject looks through a semi-transparent mirror at a computer screen. The mirror reflects the image of the subject's eye to the camera.

The eye was observed in two different ways. One detector operates in visible light and extracts the location of the pupilla, and the second one finds the location of corneal reflex of an infrared led. Also pupilla diameter is measured. Accuracy of the measurements is about one degree, so we can determine the word which is being looked at. Figure 2 shows an example eye scan path.

2.3 Feature extraction

From the raw data we can extract many variables like eye coordinates, saccade and fixation lengths and regressions (gaze returns to previous location). Total 21 of variables (Appendix A) are extracted. In this work the data consisted of 379 data samples.

The data should be classified to three classes, *non-relevant*, *relevant but not answer* and *the answer*. The class is the target variable. We want

Figure 1: The head-mounted eye tracker device

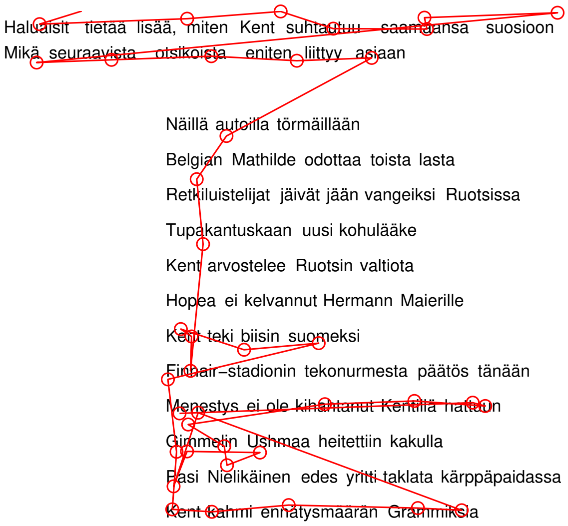

Figure 2: An example eye scan path. Circles are fixation points.

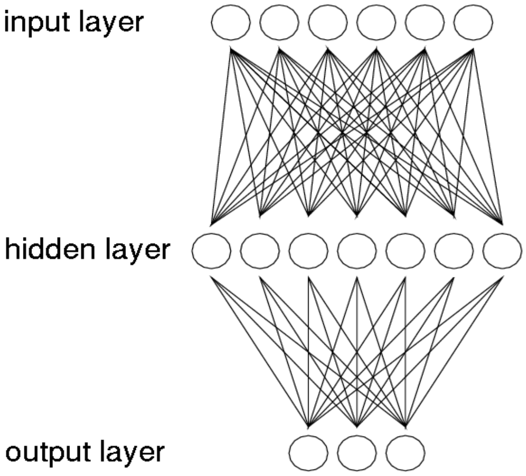

Figure 3: An example MLP network with six input, seven hidden and three output nodes.

to predict the class of a data sample by learning a Multi-Layer Perceptron network to classify the eye movement data. The method will be discussed more in detail in the next section.

3 Learning a neural network

3.1 Multi-Layer Perceptron neural network

Multi-Layer Perceptron network (MLP) [3, 12, 8, 1] is often used in *supervised learning*, like classifying tasks. In supervised learning the preferred output of the model is used in the learning procedure.

The MLP model has a biological origin, since *perceptron*, which is a building block of the MLP, is crude model of how a neuron operates. That is why MLP is called a *neural network*. From a mathematician's point of view, MLP is nice because it is possible to approximate any continuous function infinitely exactly with it.

The perceptrons [3, page 98.] take one or more inputs, make a weighted sum of them and nonlinearize the sum with an activation function. The perceptrons in the network form layers and the layers may then be connected to other layers, forming then *a multi-layer* structure (Figure 3).

3.1.1 Evaluating perceptron values

Usually one layer is used as an input layer, where each input data variable is associated with one input layer node. The input nodes are then connected to nodes in other layers, usually to a hidden layer. The layer is called a hidden layer because the values of hidden layer nodes are not studied while learning the network. A connection between nodes mean that variable values or distributions are passed from a node to another along the connection. Each connection is associated with a weight and the passed variable will be multiplied with the weight.

In each hidden node all the connected input will be summed together and possibly a bias value is added to the sum. The sum is then passed to a *activation function* f (see Section 3.1.2). The result is the value of a single perceptron and a hidden node.

$$H_j = f(\sum_{i=1}^{L} I_i w_{ij}^I + a_j). \tag{1}$$

The perceptron and the hidden node value is evaluated from input values I_i, weights w_{ij}^I and from bias values a_j. i is index that iterates through input nodes and j hidden nodes.

The values of hidden layer perceptrons are considered input values for an output layer or another hidden layer. Usually there are one or more hidden layers. In output nodes values are again summed and passed again to an activation function.

$$O_k = f(\sum_{j=1}^{M} H_j w_{jk}^O + b_k). \tag{2}$$

Hidden node values H_j, output weights w_{jk}^O and output biases b_k are used when evaluating output node value O_k. Again there is also an activation function f, but in the output layer the function may also be linear or e.g. *a Softmax activation function* discussed in section 3.1.2.

The previous formula with H_j extracted from (1)

$$O_k = f(\sum_{j=1}^{M} f(\sum_{i=1}^{L} I_i w_{ij}^I + a_j) w_{jk}^O + b_k).$$

The weights and biases are together *the parameters of the network.*

3.1.2 Activation function

In the hidden layer the activation function is usually a nonlinear function that scales the sum so that the value of the function is in (-1,1) or (0,1). The most popular nonlinear functions used are sigmoid (3) or tanh -functions.

$$f(x) = \frac{1}{1+e^{-x}}. \tag{3}$$

However later introduced software (FBM) uses plain tanh (hyperbolic tangent) as a hidden layer activation function. And especially in this classification task a *Softmax activation function* [10] is used as the **output** layer activation function. A Softmax activation function is

$$p(O = \ell | I) = \frac{e^{o_\ell}}{\sum_k e^{o_k}} \tag{4}$$

where o_k:s are the same as the argument of f in (2).

3.2 Learning MLP

3.2.1 Error function

Initially the weights are set to a random state. In *supervised* learning the output data is known and it is used to adjust the weights so that they give correct answers. The problem is, how to adjust the weights to get the correct answer. Lets introduce an error function that we should minimize:

$$E = \sum_{k=1}^{N} (D_k - O_k)^2, \tag{5}$$

where O_k is k:th output and D_k is the correct output. This is called minimum squared error (MSE).

3.2.2 Back-propagation

There is no single Back-propagation method, the main idea of back-propagation methods is to minimize the error function. For thorough introduction to back-propagation, please see [3, pages 140-148.]. Here we introduce the *gradient descent* method as a simple back-propagation method for an MLP.

If we had only 2-2-2 (input-hidden-output) MLP, the error function could be imagined as a two-dimensional surface in three-dimensional space.

Then we could minimize error by moving the weights toward the minima of the error function. The same applies also to higher dimensional cases, and we can minimize the error by moving the weights in the direction of the *negative gradient*. This way we can iterate the weights toward the correct ones. On each iteration the gradient is multiplied by a step parameter η and the step is added to the previous state. The formulation is

$$\begin{aligned} \Delta w^{(\tau)} &= -\eta \nabla E|_{w^{(\tau)}} \\ w^{(\tau+1)} &= w^{(\tau)} + \Delta w^{(\tau)}, \end{aligned}$$

where $w^{(\tau)}$ denotes the model parameters.

In gradient descent, the weights won't necessarily converge and there might also exist local minimas. One way to solve these problems is to start over with different initial weights, but still the global minima can be unreachable.

3.2.3 Maximum Likelihood

Maximum Likelihood (ML) [3] is a simple training method. The idea of ML is simply to maximize the likelihood that the input data matches the correct output by altering the model parameters. This could be done by maximizing the likelihood function

$$\begin{aligned} \mathcal{L} &= \prod_n p(x^n, y^n) \qquad (6) \\ &= \prod_n p(y^n|x^n)p(x^n), \end{aligned}$$

where x^n is the input and y^n the output vector.

Often it is more convenient to minimize the negative logarithm of the likelihood function

$$\begin{aligned} E &= -\log \mathcal{L} \\ &= -\sum_n \log p(y^n|x^n) - \sum_n \log p(x^n) \\ &\propto -\sum_n \log p(y^n|x^n) \qquad (7) \end{aligned}$$

This is also called an error function.

The second term does not depend on the model parameters, so it remains an additive constant which can be dropped away from the error function. The idea of Maximum Likelihood method is then to minimize an error function defined by (7).

ML is very straightforward training method but it comes with dangers of overfitting. Overfitting means that the model works so well with the training data, that it would give false results with other data that isn't included in the training set.

3.2.4 Cross-Validation

Overfitting of ML and other training methods may be avoided easily with Cross-Validation (CV) [3] Idea of CV is to divide the test sample into two sets, to a training set and a test set. Usually the test set is relatively small compared to the training set. The MLP is learned with the training set and tested with the test set. Then another test set is chosen and learning is repeated for all set combinations. This is called Leave-One-Out-Cross-Validation (LOOCV) [3].

If we used all data for training only, the model would fit too tightly to the data and external data would propably be misclassified. By excluding part of the data this is better avoided. Unfortunately Cross-Validation often increases calculation time so immensely, that compromises have to be carried out.

3.3 Bayesian learning

In this learning task we decided to try to use Bayesian approach. The philosophy of the Bayesian model is different from other models. In Bayesian approach we must have some knowledge of the model parameters *a priori*. In spite of this fact Bayesian approach is considered efficient and much more versatile model than others. Benefits of Bayes methods are e.g. that vague or missing variables can be regenerated from posterior probabilities.

Good introductions to Bayesian learning can be found from [10, 11, 5].

3.3.1 Bayes' rule

The simple rule behind Bayesian methods is the Bayes' formula of conditional probability. To use Bayesian methods, we must define a *prior* distribution over the model parameters according to our initial beliefs about the model. In Bayesian methods point estimates of parameters become distributions.

When the variables are observed a *posterior* distribution is acquired from Bayes' rule.

$$p(\theta|X) = \frac{p(X|\theta)p(\theta)}{p(X)} \tag{8}$$

Here $p(\theta)$ is the prior distribution, θ are the model parameters, input data is X and the denominator is a normalization factor called marginalized likelihood or model evidence (9). $p(\theta|X)$ is the probability distribution of the model parameters given the data X, the likelihood of the parameters $p(X|\theta)$, and the prior distribution $p(\theta)$. The marginalization principle is

$$p(X) = \int p(X|\theta)p(\theta)\mathrm{d}\theta. \tag{9}$$

We may also take the output data Y in consideration.

$$p(\theta|X,Y) = \frac{p(X|\theta,Y)p(\theta,Y)}{p(X,Y)}$$

A formula for predictive distribution of output value is

$$\begin{aligned}&y_i \in Y, \quad x_i \in X,\\ &p(y^{n+1}|x^{n+1},(x^1,y^1),\ldots,(x^n,y^n)) =\\ &\int p(y^{n+1}|x^{n+1},\theta)p(\theta|(x^1,y^1),\ldots,(x^n,y^n))\mathrm{d}\theta\end{aligned} \tag{10}$$

and a formula for expected value of the distribution:

$$\widehat{y}_k^{n+1} = \int O_k(x^{n+1},\theta)p(\theta|(x^1,y^1),...,(x^n,y^n))\mathrm{d}\theta \tag{11}$$

where O_k is the k:th output function and $p(\theta|\ \ )$ is the posterior distribution.

The idea of Bayesian methods lies in these formulae. The introduction of a prior distribution is a crucial step that allows us to go from a likelihood function to a probability distribution.

The need of a prior distribution is often criticized in Bayesian methods, but the bayesist (e.g. Harri Valpola, [11, page 16.]) argue that learning cannot start without any prior assumptions. In Bayesian learning the prior distributions are stated explicitly. Another argument is that distributions are safer than the point estimates provided by e.g. the Maximum Likelihood (see Section 3.2.3) method. That is, Bayesian learning avoids better overfitting of the parameters.

3.3.2 Selection of a prior distribution

Initially, if only little is known of the prior distribution of the parameters, a broad prior distribution should be selected. Vice-versa, if we have strong beliefs about the parameters values, then we can choose narrow distributions to describe our assumptions.

The superiority of a prior distribution against another can be determined easily. For each iteration we get a new posterior distribution that can be used as a prior distribution for the next iteration run. If the prior selection was correct, we get narrower probability distributions, which converge to "true" values of the weights.

3.3.3 Automatic Relevance Determination (ARD) -prior

Prior selection can have a significant effect on the result. Different priors do not necessarily converge to the same distribution. The prior distribution should be broad enough, but if they were too broad, the results would be vague. With narrow prior distribution, however, you are more likely to get false results. More information about priors in general can also be found in [12, 2, 10].

In ARD model each connection weight is associated with a hyperparameter that controls the variance of the weight. The weight hyperparameters of the same input node should then have a common prior distribution. When learning the network, if the variance of some input becomes narrow, the expectated value of them would more likely to be small and they could be pruned away. In contrary to narrow distributions broad ones means that that input are more significant. Pruning irrelevant features away from the model allows us to make classification more faster and efficient.

An example ARD-prior [8] is

$$\begin{aligned}\omega_{ij} &\sim N(0,\alpha_j),\\ \alpha_j &\sim \mathit{Inv\text{-}Gamma}(\alpha_{ave},\nu_\alpha),\\ \alpha_{ave} &\sim \mathit{Inv\text{-}Gamma}(\alpha_0,\nu_{\alpha,ave}),\end{aligned}$$

Where ω_{ij} is some weight in the network, α_j is the common variance parameter for weights connected to the same node, α_{ave} is the expectation hyperparameter for α_j and ν_α and $\nu_{\alpha,ave}$ are some fixed parameters for the distribution variances. Moreover N is a Gaussian distribution and *Inv-Gamma* is an inverse Gamma distribution which has the density function

$$p(x) = \frac{1}{b^a \Gamma(a)} x^{1-a} e^{-\frac{1}{bx}}.$$

The *Inv-Gamma*-distribution is used because it is always positive. (Variance must be non-negative).

3.4 Markov Chain Monte Carlo - integration

The objective in Bayesian learning is to compute posterior distributions for the model parameters. We can acquire distributions from the Equation 10 or single values by calculating expected values of the equation. In both tasks we need to evaluate the expectation with respect to the posterior distribution $Q(\theta)$ (8) for model parameters. The expected value of some function $a(\theta)$ is then

$$E[a] \quad = \quad \int a(\theta) Q(\theta) \mathrm{d}\theta$$

If we insert $a(\theta) = O_k(x^{n+1}, \theta)$, we get Equation 11. The integral can be approximated by *Monte Carlo* integration [10] methods, which give a sample of values from distribution Q,

$$E[a] \quad \approx \frac{1}{N} \sum_{t=1}^{N} a(\theta^{(t)}). \tag{12}$$

Here $\theta^{(1)}, \ldots, \theta^{(N)}$ are the samples from Q. The main problem with MCMC-integration is generating samples from the distributions. The samples should be independent, but it is often impossible in practice. The sum (12) will nevertheless converge to the expected value if the dependence is not too great. *A Markov Chain* [4], having Q as its stationary distribution would be a good way to generate such samples [10].

3.4.1 Gibbs Sampling

Gibbs sampling [1, 10] is a way to draw samples from multi-dimensional distributions. According to Neal [10], Gibbs Sampling is also known as the *heat bath method* especially in physics literature. Gibbs Sampling is a part of *hybrid Monte Carlo* (Section 3.5) -method.

The idea of Gibbs Sampling is that often the distribution $Q(\theta)$ is too complex, that it is impossible to draw samples of it directly. Although $Q(\theta)$ can be too hard to sample, *conditional* distributions $Q(\theta_i | \{x_j\}_{i \neq j})$ (where x is a parameter) can be tractable. The Gibbs algorithm gives us $\theta^{(t+1)}$ from $\theta^{(t)}$ as follows ($\theta^{(0)}$ should be selected wisely.):

$$\begin{aligned}
&\text{Pick } \theta_1^{(t+1)} \text{ from } Q(\theta_1 | \theta_2^{(t)}, \ldots, \theta_n^{(t)}) \\
&\text{Pick } \theta_2^{(t+1)} \text{ from } Q(\theta_2 | \theta_1^{(t+1)}, \theta_3^{(t)}, \ldots, \theta_n^{(t)}) \\
&\qquad\quad \vdots \\
&\text{Pick } \theta_n^{(t+1)} \text{ from } Q(\theta_n | \theta_1^{(t+1)}, \ldots, \theta_{n-1}^{(t+1)})
\end{aligned}$$

Note that new values $\theta_j^{(t+1)}$ are used immediately when drawing $\theta_{j+1}^{(t+1)}$.

3.5 The hybrid Monte Carlo algorithm

Neal argues in his thesis [10] that MCMC is the only feasible method to approximate the posterior distribution in the Bayes' rule (8), because does **not** take any assumptions about the distribution. In some methods the distribution might be approximated e.g. with a Gaussian distribution (For example by Laplace approximation [9]). In addition, MCMC's performance doesn't depend on the dimension of the distribution being sampled.

Nevertheless, Neal also claims that MCMC is too slow and that *a hybrid Monte Carlo* (HMC) -method would be better because its *random walk behavior* is more restricted. The restriction feature is gained by modeling the chain as a physical process and by taking the benefits of the Metropolis algorithm and Gibbs sampling while avoiding the weaknesses of the methods.

In HMC, the state of the Markov Chain will have location and momentum attributes and they will be changed following later described dynamics and energy conservation law.

If the algorithm was exact, the total energy in the system wouldn't change and only part of the states would be visited. But because this is an approximate discretization of a physical process, the total energy in the system may change, and all states will be visited sooner or later. Sometimes phase state changes are rejected based on energy changes. That should eliminate the bias produced by the inexact simulation. The details of the algorithm will be explained in the next sections.

3.5.1 The Metropolis algorithm

New state candidates in Monte Carlo type algorithms, for example in *Gibbs sampling* are produced with the *Metropolis algorithm* [10]. In Metropolis a new state $\theta^{(t+1)}$ is generated from the previous state $\theta^{(t)}$ using a specified *proposal* distribution that must be symmetrical,

$$S(\theta'|\theta) = S(\theta|\theta')$$

The example proposal distribution can be for example a Gaussian distribution having mean $\theta^{(t)}$ and variance selected so that the probability of the candidate state being accepted is reasonably high relative to the distribution Q, which is the desired stationary distribution of the Markov chain.

The next state is picked using the proposal distribution as follows:

1. Draw a *candidate* state, θ^* from a proposal distribution $S(\theta^*|\theta^{(t)})$.
2. If $Q(\theta^*) \geq Q(\theta^{(t)})$, accept the candidate state. Otherwise accept it with probability $Q(\theta^*)/Q(\theta^{(t)})$.
3. If the candidate state is accepted, let $\theta^{(t+1)} = \theta^*$. Else leave the state unchanged, $\theta^{(t+1)} = \theta^{(t)}$.

Unfortunately Metropolis algorithm won't necessarily give an ergodic Markov chain if the details of Q aren't adequate (the distribution is too complex) or the proposal distribution isn't properly selected (for example, a Gaussian would be a good choice). Highly dependent states and *random walk behavior* are also unfavorable properties of Metropolis algorithm.

The Metropolis-Hastings algorithm [3] is an improved version, where the *proposal distribution* does not have to be symmetrical.

3.5.2 Energy formulation

The random walk behavior in *hybrid Monte Carlo* is avoided with formulating the chain in terms of a physical model. In this formulation the distribution being sampled has to be in *canonical* form. A canonical distribution is defined by

$$P(q) \quad \propto \quad e^{-E(q)} \tag{13}$$

where $E(q)$ is an "energy function". Fortunately almost any positive distribution can be formulated this way, by simply defining $E(q) = -logP(q) - logZ$ for some Z.

Next we replace $E(q)$ with a *Hamiltonian function* $H(q,p)$, introducing so called *momentum variables*, p_i. We assume that the Hamiltonian function, *total energy* of the system, can be expressed in form

$$P(q,p) \quad \propto \quad e^{-H(q,p)} \quad = \quad e^{-E(q)-K(p)}, \tag{14}$$

where the *kinetic energy*, $K(p)$ can be conveniently defined by

$$K(p) \quad = \quad \sum_{i=1}^{n} \frac{p_i^2}{2m_i}$$

A good selection of masses m_i can improve algorithm performance, but for moment the may be assumed to be one.

3.5.3 Hamiltonian mechanics

After energy formulation the state phases will be changed following *Hamiltonian dynamics formulae* combined with the Metropolis algorithm. Now the distribution being sampled is the energy distribution (14). In a fictitious time τ, the state evolves

$$\frac{dq_i}{d\tau} \quad = \quad \frac{\partial H}{\partial p_i} \quad = \quad \frac{p_i}{m_i} \tag{15}$$

$$\frac{dp_i}{d\tau} \quad = \quad -\frac{\partial H}{\partial q_i} \quad = \quad \frac{\partial E}{\partial q_i} \tag{16}$$

Due to the *energy conservation law* the Hamiltonian function H must be constant, that is

$$\begin{aligned} \frac{dH}{d\tau} &= \sum_i \Big[\frac{\partial H}{\partial q_i}\frac{dq_i}{d\tau} + \frac{\partial H}{\partial p_i}\frac{dp_i}{d\tau}\Big] \\ &= \sum_i \Big[\frac{\partial H}{\partial q_i}\frac{\partial H}{\partial p_i} - \frac{\partial H}{\partial p_i}\frac{\partial H}{\partial q_i}\Big] \quad = \quad 0 \end{aligned}$$

Secondly, the volumes of the regions of the phase state must preserve. So the divergence of motion in phase space must also be zero:

$$\begin{aligned}&\sum_i \Big[\frac{\partial}{\partial q_i}\Big(\frac{dq_i}{d\tau}\Big) + \frac{\partial}{\partial p_i}\Big(\frac{dp_i}{d\tau}\Big)\Big] \\ = \quad &\sum_i \Big[\frac{\partial H}{\partial q_i \partial p_i} - \frac{\partial H}{\partial p_i \partial q_i}\Big] \quad = \quad 0\end{aligned}$$

The third feature of Hamiltonian dynamics is that the phase system is reversible. The original state can be recovered by following the dynamics backward.

Together these properties imply that any transition following Hamiltonian dynamics leaves the canonical distribution (13) invariant for a fixed value of H due to the energy conservation law. This actually means that those transitions will eventually explore the whole region of phase space for a fixed value of H.

An ergodic Markov chain can be obtained by making dynamical transitions with the Gibbs Sampling (3.4.1) updates for the momentum variable. By combining momentum and location and making the momentum change its direction on every step, it is more likely that we get an ergodic chain, in contrary to ordinary Gibbs sampling.

A new value for the momentum variable p can be drawn from a distribution proportional to $e^{-K(p)}$. The updates allow the total energy H to remain constant and we have an ergodic Markov chain that explores the entire phase space where H is constant. We still need to explore the entire phase state for different values of H. That is why we need *Leapfrog discretization*.

3.5.4 Leapfrog discretization

In (15) and (16) we introduced a fictitious time, τ. Since the Hamiltonian formulae cannot be followed exactly, the time must be discretized. The length of the time step then becomes an adjustable parameter of the stochastic dynamics. Using a time step which is long enough is better, because it results in large changes of q. The chain then better avoids the random walk behavior.

In *Leapfrog discretization* new $\widehat{q}(\tau+\epsilon)$ and $\widehat{p}(\tau+\epsilon)$ can be obtained from old $\widehat{q}(\tau)^n$ and $\widehat{p}(\tau)^n$ following these equations:

$$\widehat{p}_i(\tau + \frac{\epsilon}{2}) \quad = \quad \widehat{p}_i(\tau) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\widehat{q}(\tau)) \qquad (17)$$

$$\widehat{q}_i(\tau+\epsilon) \quad = \quad \widehat{q}_i(\tau) + \epsilon\frac{\widehat{p}_i(\tau+\frac{\epsilon}{2})}{m_i} \qquad (18)$$

$$\widehat{p}_i(\tau+\epsilon) \quad = \quad \widehat{p}_i(\tau+\frac{\epsilon}{2}) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\widehat{q}(\tau+\epsilon)) \qquad (19)$$

The momentum variable step is made in two half-steps.

To follow the dynamics for a time Δt, a convenient value for the leapfrog step ϵ must be chosen. Then equations (17)-(19) are followed for $L = \frac{\Delta t}{\epsilon}$ times.

The leapfrog discretization *preserves the phase space volume* because the changes of p and q don't depend on the components, not on theirselves. The system is still *reversible*, but the energy H is **not** constant. Therefore there will be a systematic error that decreases if the leapfrog step is shortened.

3.5.5 Hybrid Monte Carlo method description

Hybrid Monte Carlo updates are made following these steps:

1. Starting from the current state, $(q,p) = (\widehat{q}(0), \widehat{p}(0))$, perform L leapfrog steps with a step size ϵ, resulting in a state $(\widehat{q}(\epsilon L), \widehat{p}(\epsilon L))$.

2. Create a candidate state by negating momentum variables

$$(q^*, p^*) = (\widehat{q}(\epsilon L), -\widehat{p}(\epsilon L))$$

3. Accept the candidate state with probability

$$\min(1, e^{-H(q^*,p^*)-H(q,p)})$$

If the candidate state is rejected the state remains unchanged.

4 Experiments

4.1 Radford Neal's Software for Flexible Bayesian Modeling

The work was done with Radford Neal's software for Flexible Bayesian Modeling (FBM)[1]. The experiment was carried out mainly by following software documentation for three way-classification[2].

[1] http://www.cs.toronto.edu/∼radford/fbm.software.html

[2] http://www.cs.toronto.edu/∼radford/fbm.2003-06-29.doc/Ex-netgp-c.html

The software version used in this experiment was released 29th June 2003.

4.2 Data analysis

More thorough documentation for the software usage can be found in Appendix B. Here is a compact description of the calculations.

To get more reliable results cross-validation (see section 3.2.4) was used in this work. The available data, 379 rows of ASCII data was divided into training and test data by taking 360 rows for training and 19 for testing. Twenty validations sets were created this way.

The MLP (see section 3.1) used eight different hidden node configurations, from three to ten hidden nodes. Each of the 21 input variables had its own input node, and there was three output nodes. The three output nodes were linked to the target value with Softmax function (4). Also the weight priors had to be be introduced in the beginning.

After the program is initialized, the sampling is begun with one hundred steps with a small trajectory. The initially randomly chosen weights are updated once this way.

The step size is next increased slightly and also the hyperparameters are updated on every step. The serious sampling from the posterior distribution is carried out with 1999 iterations. The procedure can be monitored on-line (Figure 5). After sampling the program will give the classification results straight with one command. Results were processed into graphs with Matlab.

4.3 Results and discussion

After cross-validation with twenty training and test sets and eight different hidden node configurations (from three to ten hidden nodes) we acquired relevances shown in Figure 4. Classification results (Table 1) were not so perfect, since from 13 to 34 percent of guesses were wrong. In Figure 4, the sum of all cross-validation case weights with different hidden node configurations and over all configurations is plotted. We can easily see that variables number 1, 2, 13, 18 and 21 are relevant. Also 3, 5, 6 and 16 seem to have some relevance while others seem to be irrelevant. In this point of view, total regression durations to a word (regressDur), the relative duration of the first fixation (timePrct), the total duration to a word (totalFixDur), number of the fixations to a word when first encountered (FirstPassCnt) and the total number of fixations (fixCount) are the most relevant in that order. These acronyms are explained in Appendix A.

Table 1: Classification results with different number of hidden nodes. With two nodes the result was precisely same in all twenty cross-validation sets

Hidden nodes	**Average validation result (%)**	**95% confidence interval (%)**
2	84.21	± 0.00
3	82.63	± 8.92
4	82.37	± 13.50
5	81.58	± 14.78
6	81.31	± 14.77
7	81.84	± 14.39
8	81.31	± 14.39
9	81.31	± 13.99
10	81.58	± 13.60
ALL	82.02	± 12.71

It's difficult to say which hidden node configuration would be the best alternative given these results. It seems though that increasing the hidden node count won't necessarily increase validation result.

5 Acknowledgments

The experiment was mainly devised by the author with strong support of Jarkko Salojärvi. I also want to thank Kai Puolamäki, Eerika Savia and everyone else in the lab who helped me during my assignment.

When devising this work I was a summer student at the lab, and this work will be graded as my first special assignment in Helsinki university of technology.

References

[1] *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

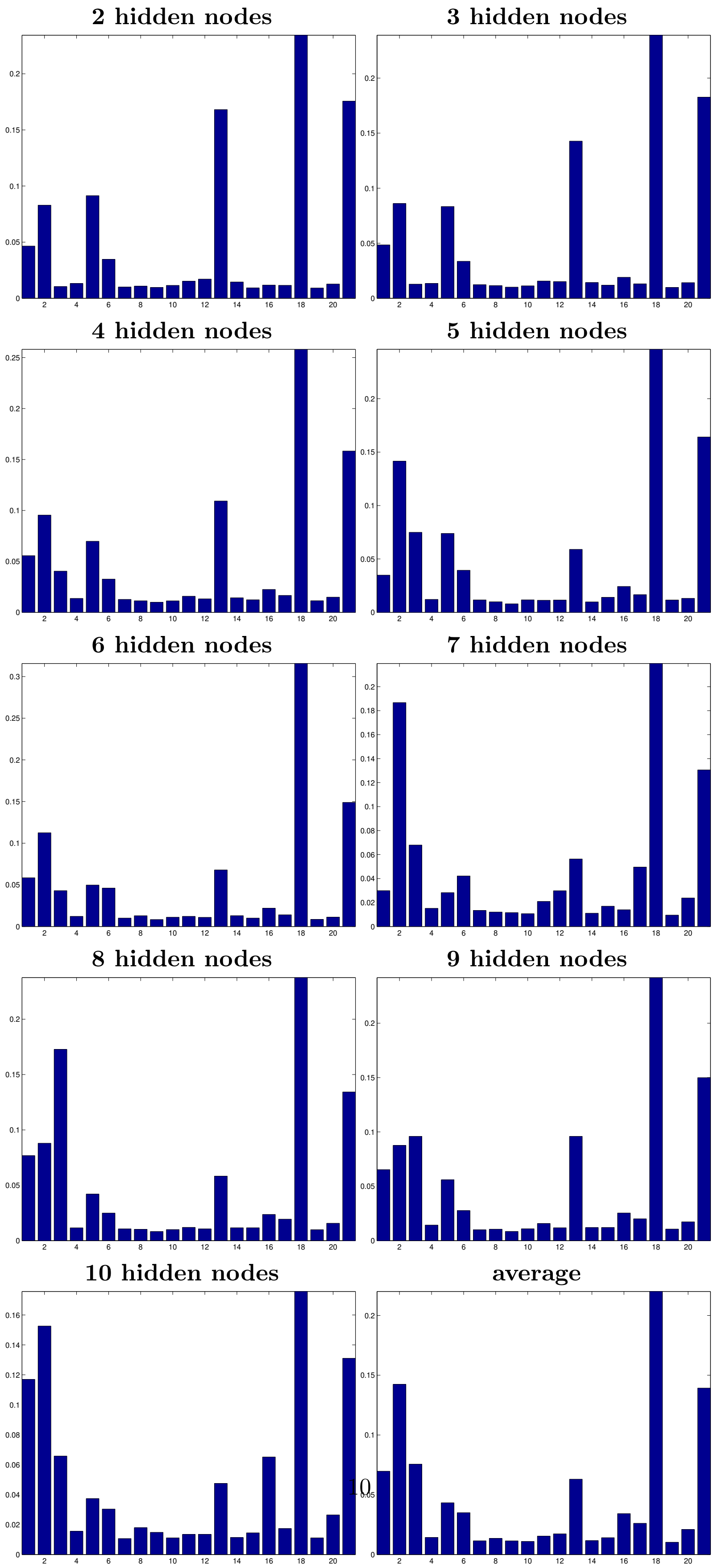

Figure 4: Values of the low-level input-hidden hyperparameters. The bigger pillar, the more relevant variable. Sum of relevances is scaled to one.

[2] Matthew J. Beal. *Variational Algorithms for Approximate Bayesian Inference*. PhD thesis, University of London, 2003.

[3] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press Inc., New York, 1995.

[4] Richard Durrett. *Essentials of Stochastic Processes*. Springer Verlag, July 1999.

[5] J. Karhunen H. Valpola. An unsupervised ensemble learning method for nonlinear dynamic state-space models. *Neural Computation*, 2002.

[6] Kai Puolamäki Jarkko Salojärvi and Samuel Kaski. Relevance feedback from eye movements for proactive information retrieval. Technical report, Helsinki University of Technology, 2003.

[7] Samuel Kaski Jarkko Salojärvi, Ilpo Kojo. Can relevance be inferred from eye movements in iformation retrieval?

[8] Jouko Lampinen and Aki Vehtari. Bayesian approach for neural networks – review and case studies. *Neural Networks*, 2001.

[9] David J. C. MacKay. Choice of basis for laplace approximation. *Machine Learning*, 33(1):77–86, 1998.

[10] Radford M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag New York, 1996.

[11] Harri Valpola. *Bayesian Ensemble Learning for Nonlinear Factor Analysis*. PhD thesis, Helsinki University of Technology, 2000.

[12] Aki Vehtari and Jouko Lampinen. Bayesian input variable selection using posterior probabilities and expected utilities. Technical report, Helsinki University of Technology, Laboratory of Computational Engineering, 2002.

A Eye movement features

The eye movement features used in this paper:

1. **fixCount**: Total number of fixations to the word.
2. **FirstPassCnt**: Number of fixations to the word when the word is first encountered.
3. **P1stFixation**: Did a fixation to a word occur when the sentence that the word was in was encountered for the first time?
4. **prevFixLen**: Duration of the previous fixation when the word is first encountered.
5. **firstFixDur**: Duration of the first fixation when the word is first encountered.
6. **firstPassFixDur**: Sum of durations of fixations to a word when it is first encountered.
7. **nextFixDur**: Duration of the next fixation when the gaze initially moves on from the word.
8. **firstSaccLen**: Distance (in pixels) between the first fixation on the word and the previous fixation.
9. **lastSaccLen**: Distance (in pixels) between the last fixation on the word and the next fixation.
10. **prevFixPos**: Distance between the fixation preceding the first fixation on a word and the beginning of the word (in pixels).
11. **landingPosition**: Distance of the first fixation on the word from the beginning of the word (in pixels).
12. **leavingPosition**: Distance between the last fixation before leaving the word and the beginning of the word (in pixels).
13. **totalFixDur**: Sum of all durations of fixations to the word.
14. **meanFixDur**: Mean duration of the fixations to the word.
15. **nRegressionsFrom**: Number of regressions leaving from the word.
16. **regressLen**: Sum of durations of fixations during regressions initiating from the word.
17. **nextWordRegress**: Did a regression initiate from the following word.

18. **regressDur**: Sum of the durations of the fixations on the word during a regression.

19. **pupilDiamX**: Mean horizontal pupil diameter during fixations on the word minus mean pupil diameter of the subject during the experiment.

20. **pupilDiamStd**: Standard deviation of the pupil horizontal diameter during fixations on the word.

21. **timePrctg**: First fixation duration divided by the total duration of fixations on the display.

B Documentation for use of FBM

Radford's software is based on Delve-environment[3] (*Data for Evaluating Learning in Valid Experiments*). (Radford is himself a member of DELVE's development group.) Here are almost complete instructions for calculations made before in previous section.

B.1 Dividing data into validation sets

In this learning task the data was divided to twenty-fold cross-validation. Each training set consist of 360 samples and each test set consists of 19 samples. Before the creation of validation sets the lines were randomized, because the file originally contained lines in order of subjects.

First we had only one text file containing eye movement data:

```
21
#n fixCount FirstPassCnt P1stFixation
   prevFixLen firstFixLen  firstPassFixLen
   nextFixLen firstSaccLen lastSaccLen
   prevFixPos       landingPosition
   leavingPosition totalFixLen
   meanFixDuration nRegressionsFrom
   regressLength    nextWordRegress
   regressDuration pupilDiamX
   pupilDiamStd     timePrct
-0.166074 0.161154 1.09019 0.850887
 -0.0944882 -0.484063    0.0973345
  0.273784   1.28229     0.312651
  0.262353   0.00648624 -0.211101
 -0.236752  -0.425295   -0.230825
 -0.322011  -0.409793   -0.347073
  0.520749   1.49323     1
```

And similar numerical data yet another 378 lines. The data division was done with Matlab g.e with SOM Toolbox[4]. I used m-file *som_read_data.m* to load the file in Matlab, and then I divided the data and wrote it in multiple files with *som_write_data.m*.

B.2 Command syntax

Then I started to follow the instructions for Radford's Software for Flexible Bayesian Modeling[5]. There exists plenty of different instructions, but there was one exactly designed for my job, *A three-way classification problem*. There is an example, how to classify example test data, but It can easily be modified for my task. On the instruction page there are two alternate examples, but the latter one seems not to have any hidden layer in the network. I tried shortly to use the alternate example, but the classification results were inadequate.

The more successive example is done with these Unix commands:

1. First the MLP must be initialized with command *net-spec*. Here is an example command:

   ```
   net-spec r31 21 3 3 / \
     - x0.2:0.5:0.5 0.05:0.5 \
     - x0.05:0.5 - 0.05:0.5 ;
   ```

   Here *r31* is the name of the log file, *21* is the number of input nodes (variables), first *3* is the number of hidden nodes, second one is the number of output nodes.

   After that follows ARD specification. First input-to-hidden weight priors are specified, as instructions say *"x0.2:0.5:0.5", has two "alpha" values, indicating that there is both a high-level hyperparameter controlling the overall magnitude of input-to-hidden weights.* I also tried to

[3] http://www.cs.toronto.edu/~delve/

[4] http://www.cis.hut.fi/projects/somtoolbox/

[5] http://www.cs.toronto.edu/~radford/fbm.2003-06-29.doc/

modify prior specifications, but neither nothing changed nor things got worse. The latter three priors are for input units, *which control the magnitudes of weights out of each input.* These priors should wipe out inputs that are irrelevant.

2. Secondly we must tell that our problem is a classification problem. This is done with command

   ```
   model-spec r31 class
   ```

3. Third command gives the program the data files:

   ```
   data-spec r31 21 1 3 / \
     ./data/tr201@1:360 . \
     ./data/t201@1:19 .
   ```

   First two arguments are the numbers of input and target values and third argument, *3* followed by / means that output values are "0", "1" and "2". Then follows training data file (after @-sign there are the lines that should be included from the file), and test data file. This command should give output like this:

   ```
   Number of training cases: 360
   Number of test cases: 19
   ```

4. Fourth command initializes the network. It stores the initial phase to the log file and fixes the weight hyperparameters to 0.5.

   ```
   net-gen r31 fix 0.5
   ```

5. Fifth command, *mc-spec* specifies the Markov chain operations to be performed in the initial phase. Following steps are repeated ten times: *A heatbath replacement of the momentum variables, and a hybrid Monte Carlo update with a trajectory 100 leapfrog steps long, using a window of 10, and a step size adjustment factor of 0.2.*

   ```
   mc-spec r31 repeat 10 sample-noise \
     heatbath hybrid 100:10 0.2
   ```

6. Previous command only specifies how the sampling should be done. This *net-mc* command updates the hyperparameters with a single iteration.

   ```
   net-mc r31 1
   ```

7. *The 'mc-spec' command appends a new set of Markov chain operations to the log file, which will override the previous set. These operations are Gibbs sampling for the hyperparameters and, a heatbath update for the momentum variables, and a hybrid Monte Carlo update with a trajectory 100 leapfrog steps long, a window of 10, and a step size adjustment factor of 0.3.* In practice this means that the stepsize is increased slightly.

   ```
   mc-spec r31 repeat 10 sample-sigmas \
     heatbath 0.95 hybrid \
     100:10 0.3 negate ;
   ```

8. And this command does the previously specified sampling for 1999 iterations.

   ```
   net-mc r31 2000
   ```

   These commands were only one example of the scripts I used. I made a script that created scripts that included shell commands for different hidden node configurations and for every cross-validation set. The *net-mc* procedure is very intensive and it takes huge amount of resources and time. Actually I was accused for taking all the processor time. I adjusted efficiency with *nice* and left the machines to calculate chains over a weekend.

B.3 Analyzing the results

1. Hours and days after we can analyze the results. Command *net-plt* allows us to plot the evolution of the low-level input-hidden hyperparameters, even during the simulation run with command

   ```
   net-plt t h1@ r31 | plot
   ```

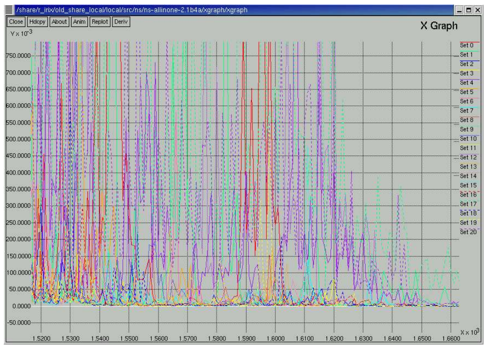

Figure 5: An example plot program output. The hyperparameters behave pretty wildly.

where plot should be a suitable plotting program like *xgraph*[6]. Here (5) is an example plot.

2. Command *net-pred* tells us the estimated classification performance.

   ```
   net-pred ma r31 101:
   ```

   The option “a” means that only the average log is displayed. With option “m” the fraction of wrong guesses are displayed. The number 101 and colon indicates that only iterations beginning from 101 should be taken into consideration. With option “n” the program will print weight means that can be used when assessing relevance.

[6]David Harrison
http://jean-luc.ncsa.uiuc.edu/Codes/xgraph/