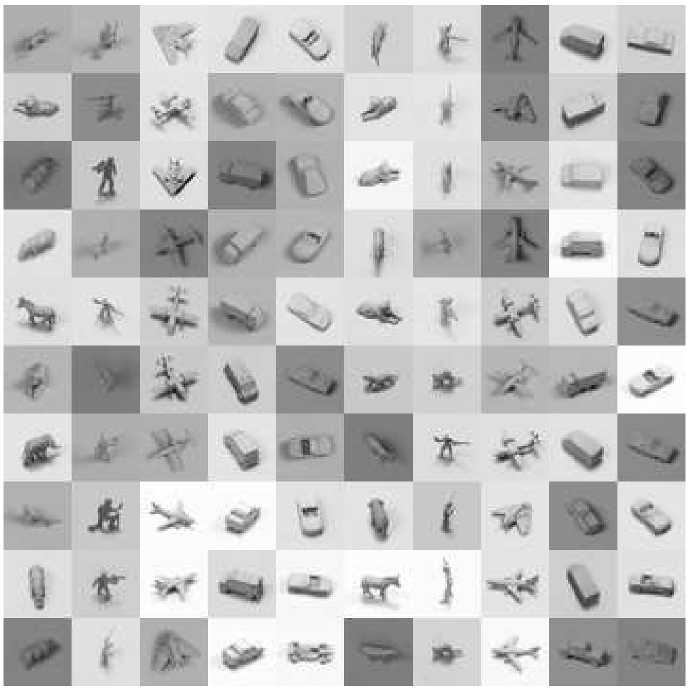
Lecturer: Matthias Seeger
Assistants: Vassilis Kalofolias, Young Jun Ko, Willem A. M. Wybo, Giuseppe Chindemi

Miniproject

March 26, 2013

1 Introduction

The aim of this miniproject is to provide you with experience in implementing some widely used machine learning methods and applying them to a real world problem: object recognition, i.e. classifying an input image according to which object is depicted. The dataset that you will work with is the so called small NORB 3D dataset(reference), an excerpt of which is shown below, that contains stereo images of objects from 5 categories taken under varying illumination conditions from different angles. A detailed description of the dataset can be found in section 3.



You will solve two classification tasks, requiring the implementation of multi-layer perceptrons, logistic- and linear regression. The main aim of this exercise for you is to gain hands-on experience with some widely used pattern classification methods, to understand how they work by implementing them in a clean, elegant way, and above all, to provide a thorough and statistically sound state-of-the-art evaluation of this methodology on real data. Make sure to keep your code as clean and simple as possible and focus on understanding and elegance rather than fine-tuning the bits. We are interested only in you understanding how the algorithms work, that your implementation works and is well tested, and in the resulting evaluation. Excessive optimization is not rewarded whereas a good design that facilitates the reusability of components can save you a vast amount of time.
The main deliverables that the grading of the project will be based on are:

- project report: provide a concise description of your approach and the details the evaluation of the methods you tried.
- fraud detection session: each of you will be interviewed about your understanding of the code you produced.

2 Project Overview

There will be two tasks:

1. binary classification: implement, train and evaluate a Multi-Layer Perceptron(MLP) to discern two kinds of objects.
2. multi-way classification: extend your MLP implementation for the simple binary case to the 5 classes. Implement linear classifiers to compare against.

We encourage you to use `MATLAB` or `Python/Scipy`, which are the major development tools used by machine learners. You may also use C++, Java or other languages, but in that case we may be less able to help with specific questions.
The most important rule: **You must not use any kind of foreign code** for this project, neither any piece of code from another group. In fact, you will have to demonstrate convincingly that you implemented all relevant algorithms by yourself. Copying even small pieces of code from elsewhere is considered **plagiarism**: a serious offense. We will automatically match your submitted code against code submitted by other groups, and against code out there in the web, so **do not copy**. The following exceptions apply (please check with us when in doubt):

- In case you do not use `MATLAB`, you are allowed to use libraries for basic linear algebra (vectors, matrices, matrix multiplication, ...). In fact, you are **discouraged** from implementing this for yourself.
- You are allowed to use foreign code in order to load or store data, or to plot results (we provide I/O code for `MATLAB` and `Python`).

The assignment consists of the following steps (this is more of a checklist, details are provided in subsequent sections):

1. **Register your group (max. 2 persons) on Moodle.**
   - Choose a slot according to the programming language which you are planning to use (this influences your assignment to a TA).
   - Each group is assigned a TA who is responsible for answering your questions and monitoring your progress.
   - We will also assign training and test sets to each group.
2. **Download the data from Moodle(Section 3).**
   - Retrieve and examine the test and training sets for both tasks from Moodle(section 3.1).
   - For the first task, you will work on randomly drawn subsets corresponding to binary subproblems of the full 5-class dataset, i.e. you are given patterns corresponding to two different objects (e.g. cars and airplanes) encoded as $+1$ and -1
   - For the second task, you will work on randomly drawn subsets corresponding to the full 5-class dataset, i.e. you are given patterns corresponding to all five different classes encoded as $\{0, \dots, 4\}$
   - Split the training data into a training and a validation set. Use the former for training, the latter for model selection and early stopping(section 3.2).
   - Preprocess the data(section 3.3).
3. **Task 1: Implement a MLP for binary classification(Section 4).**
   - Train a MLP by stochastic gradient-descent optimization (with momentum term) based on the error backpropagation rule as discussed in the lecture.

- Use the validation set for model selection (number of hidden units) and adjustment of the algorithm parameters(learning rate, momentum term).
- Re-train the MLP on the whole training set with your choice of parameters and evaluate the final performance on the test set.

4. **Task 2: Tackle the multi-way classification problem(Section 5).**
   - Implement linear methods as baseline:
     - Train a linear classifier using the squared loss and Tikhonov regularization. As described in the lecture notes, this has an analytic solution. Use cross validation to select the regularization parameter.
     - Train a linear classifier using the logistic loss. There is no analytic solution for this. Use stochastic gradient descent as for the MLP.
   - Extend your MLP implementation to the 5 class setting, but use the squared error as loss function. For training, validation and testing, proceed as in task 1.
5. **Write and submit a short report (Maximum 7 pages).**
   - Compare the algorithms. Report your findings not only in quantitative but also in qualitative terms.
   - **Hand in your source code as well. Include comments, clearly marking the core of your implementation in your code, that is, the most important lines of code**.
   - Be prepared for in-depth questions about the workings of your code.

3 The Simplified NORB 3D Dataset – Object Recognition

3.1 Downloading the Dataset

To get you started quickly, we provide two `.mat` files containing the training and test sets for the two classification tasks. The format is native to MATLAB and you can load it directly. For Python users we provide a small code snipped `load_norb.py` that illustrates how to load the files.
Once you downloaded the database and set up the code to load it, verify that the input works for you. Load the database, plot a histogram of the labels and plot some images (e.g. use `imagesc` and `reshape` in `MATLAB` to do that – or likewise `imshow` and `reshape` in Python).
The NORB dataset consists of images of 5 different classes of objects acquired with a stereo vision system, i.e. two cameras positioned next to each other. Hence, each sample $(\boldsymbol{x}_i, t_i)$ consists of two images as input:

$$\boldsymbol{x}_i = \{\boldsymbol{x}_{L,i}, \boldsymbol{x}_{R,i}\} \text{ where } \boldsymbol{x}_{L,i}, \boldsymbol{x}_{R,i} \in \mathbb{R}^{24\times 24}$$

one from each camera, and the class label ($t_i \in \{-1, +1\}$ for the binary problem and $t_i \in \{0, \ldots, 4\}$ for the multi-class problem). For convenience, the inputs are stored in two matrics $\boldsymbol{X}_L, \boldsymbol{X}_R \in \mathbb{R}^{576\times N}$, whose columns containing all the N images of the respective camera in vectorized form.
In the first contact meeting with your assigned TA, you should demonstrate that you can load the data and know how to get started. You should also demonstrate how you split the training part into a training set and a validation set. Please communicate with your assigned TA early to get help. Do not wait until the end of the term!

3.2 Split the Data Set

The `.mat` files contain training and test sets. **The test set is to be used only to determine the final performance of your implementation.** All other operations related to training, such as tuning parameters, etc., has to be performed on the samples of the training set only. For that we ask you to use different validation strategies.
For the MLP you have to reserve a part of the training set for validation (1/3 of cases) and perform the actual training on the remaining 2/3 of the cases. Do this at random (in `MATLAB`, use `randperm`). Now, you have three datasets:

- A training dataset
- A validation dataset, for selecting between different parameters and for early stopping
- A test dataset, which you use to report final results only

It is easier to do this split once and then store the subsets, especially during the earlier stages of your implementation, to make the outcome of your algorithm reproducible for debugging. Later you could check that the randomness of the split does not affect your results (too much). In section 5 we also ask you to perform cross validation for one of the methods, which again needs to be performed on the training data only by randomly dividing it into the required number of folds.
Do not touch the test dataset **at any time** during training, model selection and parameter tweaking, certainly not for early stopping. This set has a single purpose only: to evaluate the performance (a single number, the test set error) of the very final trained classifier you commit to at the very end. You must not go back and tweak things after this final evaluation. Otherwise, the test error as estimate for the true risk of your classifier is overly optimistic and will not be useful. Also, the NORB data is particularly difficult in that the test set contains instances of the different classes that are not present in the training set (e.g. in the class "four legged animals“ the training set contains only images dogs whereas the test set contains images of horses). A good model is therefore one, that generalizes well, by learning the important characteristics of each class. As this is the real challenge in Machine Learning, you should not be surprised to find a much higher test error in the end. Notice also that there are no bonus points whatsoever for the best performance in absolute terms: you get points for a clean and statistically sound evaluation.
In the first contact meeting with your assigned TA, you should demonstate how you split the training part of the data into a training and a validation set.

3.3 Preprocess the Data

Preprocessing is in general an important part of any machine learning application. This is especially important in the context of MLPs where the non-linear sigmoids can easily saturate and the optimization stoppes making progress. In the exercise here, a basic normalization suffices (if you are done with everything else, you are encouraged to try other ideas as well). Please shift and rescale the data to have 0 mean and unit variance across samples. For that denote

$$\begin{aligned}
\hat{\boldsymbol{\mu}} &= \frac{1}{n}\sum_i \boldsymbol{x}_i \\
\sigma_j &= \sqrt{\frac{1}{n}\sum_i (x_{i,j} - \hat{\boldsymbol{\mu}}_j)^2} \\
\boldsymbol{\sigma}_- &= \left[\sigma_j^{-1}\right]
\end{aligned}$$

and apply the following transformation to all inputs($\circ$ denotes componentwise multiplication):

$$\boldsymbol{x}_i \leftarrow \boldsymbol{\sigma}_- \circ (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})$$

Apply this normalization to all patterns of the dataset, training and test. We recommend that you store the preprocessed data and work on it exclusively henceforth.
For the NORB data, you can of course do this separately for $\boldsymbol{X}_L, \boldsymbol{X}_R$.
Note that $\hat{\boldsymbol{\mu}}, \boldsymbol{\sigma}_-$, are computed on the training part of the data only. This is because you *never* touch the test set for anything, not even for preprocessing.

4 Task 1: Implement a multi-layer perceptron for binary classification (Start Now!!)

As preparation for dealing with the multi-way classification problem, your first task is to implement and evaluate stochastic gradient descent training of a MLP for the binary classification problems we provide, i.e. distinguish between cars and airplanes.
As the miniproject takes substantial time to complete, not only the coding, but also the evaluation, it is important that you start now – do not wait until the end of the term. We first describe the setup, then give comments about implementation and debugging, and finally explain how the evaluation should be done, and what you should provide in your final report (6). Make sure to study chapter 3 ("The multi-layer perceptron") in the course notes, we will use the same notation adapted to the specific MLP architecture to be used here.

4.1 Setup

You will implement a three-layer MLP, with two hidden and one output layer, as shown in figure 4.1.
For readibility, we drop the index for the training sample. The **first layer** is symmetric for left and right input images and consists of an affine map followed by a tanh transfer functions. Thus the forward equations are for $i = 1, \dots, H_1$, the number of hidden units in layer 1:

$$\begin{aligned}
\boldsymbol{a}^{(1)}_{L,i} &= (\boldsymbol{w}^{(1)}_{L,i})^T \boldsymbol{x}_L + b^{(1)}_{L,i} \\
\boldsymbol{z}^{(1)}_{L,i} &= g^{(1)}(\boldsymbol{a}^{(1)}_{L,i}) \\
g^{(1)}(a) &= \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} = 2\sigma(2x) - 1
\end{aligned}$$

The **second layer** combines both sides of the image. Each unit has three activations as input: $a^{(2)}_L$, an affine function of left side inputs; $a^{(2)}_R$, of right side inputs; $a^{(2)}_{LR}$ of both left and right side inputs. The transfer function for this layer is derived from the standard sigmoid function. Thus the forward equations are for $i = 1, \dots, H_2$, the number of hidden units in layer 2:

$$\begin{aligned}
\boldsymbol{a}^{(2)}_{L,i} &= (\boldsymbol{w}^{(2)}_{L,i})^T \boldsymbol{z}^{(1)}_L + b^{(2)}_{L,i} \\
\boldsymbol{a}^{(2)}_{R,i} &= (\boldsymbol{w}^{(2)}_{R,i})^T \boldsymbol{z}^{(1)}_R + b^{(2)}_{R,i} \\
\boldsymbol{a}^{(2)}_{LR,i} &= (\boldsymbol{w}^{(2)}_{LR,i})^T \boldsymbol{z}^{(1)}_{LR} + b^{(2)}_{LR,i}, \text{ where } \boldsymbol{z}^{(1)}_{LR} = [(\boldsymbol{z}^{(1)}_L)^T, (\boldsymbol{z}^{(1)}_R)^T]^T \\
\boldsymbol{z}^{(2)}_i &= g^{(2)}(\boldsymbol{a}^{(2)}_{L,i}, \boldsymbol{a}^{(2)}_{R,i}, \boldsymbol{a}^{(2)}_{LR,i}) \\
g^{(2)}(a_L, a_R, a_{LR}) &= a_{LR}\sigma(a_L)\sigma(a_R) = \frac{a_{LR}}{(1 + e^{-a_L})(1 + e^{-a_R})}
\end{aligned}$$

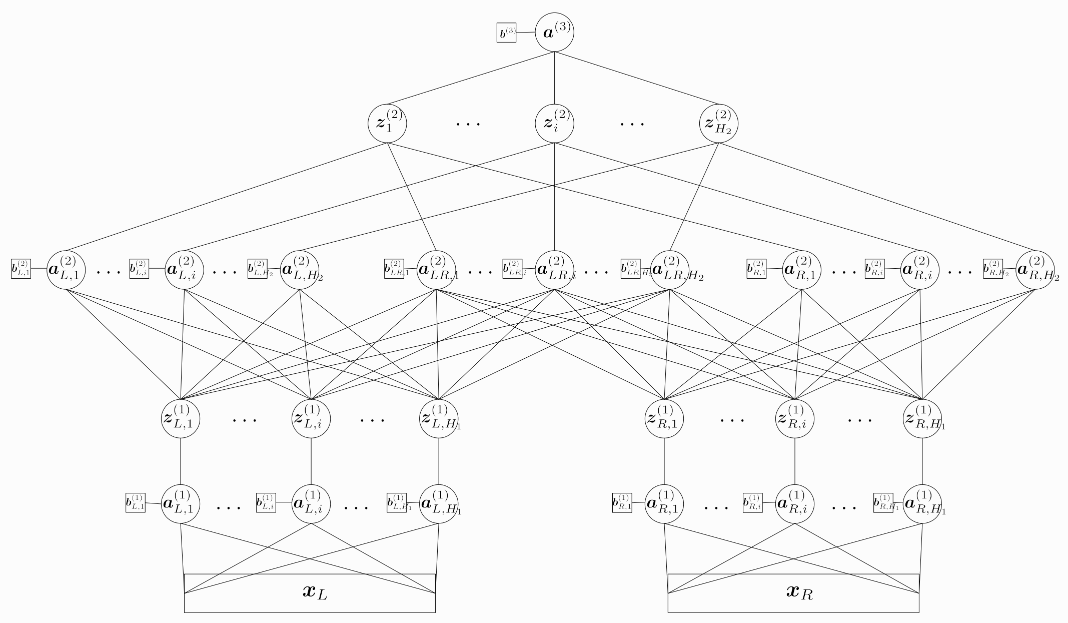

$\boldsymbol{b}^{(3)}$
$\boldsymbol{a}^{(3)}$
$\boldsymbol{z}_1^{(2)}$ … $\boldsymbol{z}_i^{(2)}$ … $\boldsymbol{z}_{H_2}^{(2)}$
$\boldsymbol{a}_{L,1}^{(2)}$ … $\boldsymbol{a}_{L,i}^{(2)}$ … $\boldsymbol{a}_{L,H_2}^{(2)}$ $\boldsymbol{a}_{LR,1}^{(2)}$ … $\boldsymbol{a}_{LR,i}^{(2)}$ … $\boldsymbol{a}_{LR,H_2}^{(2)}$ $\boldsymbol{a}_{R,1}^{(2)}$ … $\boldsymbol{a}_{R,i}^{(2)}$ … $\boldsymbol{a}_{R,H_2}^{(2)}$
$\boldsymbol{z}_{L,1}^{(1)}$ … $\boldsymbol{z}_{L,i}^{(1)}$ … $\boldsymbol{z}_{L,H_1}^{(1)}$ $\boldsymbol{z}_{R,1}^{(1)}$ … $\boldsymbol{z}_{R,i}^{(1)}$ … $\boldsymbol{z}_{R,H_1}^{(1)}$
$\boldsymbol{a}_{L,1}^{(1)}$ … $\boldsymbol{a}_{L,i}^{(1)}$ … $\boldsymbol{a}_{L,H_1}^{(1)}$ $\boldsymbol{a}_{R,1}^{(1)}$ … $\boldsymbol{a}_{R,i}^{(1)}$ … $\boldsymbol{a}_{R,H_1}^{(1)}$
$\boldsymbol{x}_L$
$\boldsymbol{x}_R$

Figure 1: The MLP architecture.

Note, that the weight vector $\boldsymbol{w}_{LR,i}^{(2)}$ is twice as long as $\boldsymbol{w}_{L,i}^{(2)}$ and $\boldsymbol{w}_{R,i}^{(2)}$.
The **third layer** is simply an affine mapping from.

$$\boldsymbol{a}^{(3)} = (\boldsymbol{w}^{(3)})^T \boldsymbol{z}^{(2)} + b^{(3)}$$

Suppose the binary training dataset is $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \ldots, n\}$, where $t_i \in \{-1, +1\}$. Your code should minimize the *logistic error function*, which evaluated over the whole training set is

$$E_{\log}(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{N} \log\left(1 + e^{-t_i a^{(3)}(\boldsymbol{x}_i)}\right).$$

This error function is discussed in Section 8.2 ("Logistic Regression") of the course notes. Here, $\boldsymbol{w}$ is a placeholder for **all** the weights and biases of the network. Notice that this is **not** the "usual" squared error function, and also that the output layer activation $a^{(3)}(\cdot)$ is not pushed through another transfer function: its value goes directly into the logistic error function.
As this setup deviates from what is detailed on in the course notes, you will have to derive the backpropagation equations. **Please do this early, and then take the opportunity to show your solution to your assigned TA, who will help you spotting mistakes.**
Some tips:

- Rewriting the transfer functions in terms of the sigmoid function can help you to find derivatives.
- Prepare for vectorization in your implementation by writing the forward equations as matrix-vector or matrix-matrix multimplications and prepare also to support mini-batches.
- Figure out which matrix dimension in terms of the parameters of your network (H_1, H_2, etc.) you would expect. This will help you a lot during the implementation to stay on top of things and avoid mistakes.

4.2 Implementation

Before you start hacking away, make sure to understand the structure of what you want to do. Your goal is to minimize the logistic error function $E_{\log}(\boldsymbol{w})$ w.r.t. all free parameters of the network, collectively called $\boldsymbol{w}$. To this end, you choose an *optimization algorithm*: gradient descent minimization with a momentum term. It is recommended that you use *stochastic gradient descent*, updating on single patterns or small batches. The optimizer is driven by the gradient $\nabla_{\boldsymbol{w}} E_{\log}(\boldsymbol{w})$, which you compute using the error backpropagation rule. The main computational backbone of your code will be this latter part. Since this is also the most complex part of your code, this should be the first part to check for bugs.
Here some hints on the implementation. As discussed in detail in the course notes, MLPs consists of layers interleaving *linear* (inner product with weight vectors, adding biases) and *componentwise nonlinear* operations (evaluating transfer functions). For the linear parts, **it is very inefficient, error-prone and not elegant** to use `for` loops. You avoid `for` loops by using matrices and vectors, then coding the linear parts by way of matrix-vector or matrix multiplications. Use built-in language facilities:

- Matrix-vector or matrix-matrix multiplication $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$: `y = a*x` in `MATLAB` .
- Pointwise multiplication: $\boldsymbol{z} = \boldsymbol{x} \circ \boldsymbol{y}$, meaning that $z_i = x_i y_i$: `z = x.*y` in `MATLAB` .
- Inner (or scalar) product: $\alpha = \boldsymbol{x}^T \boldsymbol{y} = \sum_i x_i y_i$: `alpha = x'*y` in `MATLAB` .

- Outer product: $\boldsymbol{A} = \boldsymbol{x}\boldsymbol{y}^T$ (if $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$, then $\boldsymbol{A} \in \mathbb{R}^{n\times m}$): `a = x*y'` in `MATLAB` .

The golden rule: matrix-vector multiplication is better than a loop over vector-vector multiplication (inner product), and matrix-matrix multiplication is better than a loop over matrix-vector multiplication. For example, while it is admissable to run a `for` loop over training patterns to compute the full error function and gradient, computing forward and backward pass for each point separately, it is more elegant and faster to process all training cases together, propagating matrices through the network instead of vectors.
Another important point about error backpropagation is to get the bookkeeping between forward and backward pass right. Sit down and discuss this with your group partner, using hints given in the course notes.
Here some general hints on getting your code to work (specific debugging hints are given below):

- The network architecture you have to implement is fairly complex. We highly recommend that you debug your gradient code (backpropagation) on a much simpler standard two-layer MLP architecture, say with $h_1 = 4$ hidden units. Play around with an artificial XOR problem (Section 3.1 in the course notes) with $\boldsymbol{x} \in \mathbb{R}^2$ and just four datapoints. Use stochastic gradient descent, as described in Section 4.3. Test your gradient code (see below). Once all this works (gradient correct, XOR solved), move on and debug your code on the final architecture.

- Numerical errors: MLP training is fairly uncritical to implement, but still beware of numerical errors, in particular for the nonlinear componentwise steps in forward and backward passes. For example, terms of the form A/B, where either $A, B \approx 0$, or A, B very large, are critical: try to rewrite them into a new form C/D, where either C or D are order of unity (not too large, not too close to 0). A simple example is
$$\frac{e^{-x}}{1+e^{-x}} \quad \Rightarrow \quad \frac{1}{1+e^{x}}.$$
The term on the left may be critical for x large negative. The term on the right is the same in exact arithmetic, but it is numerically uncritical.
Another example: Computing $\log(1+e^x)$ directly does not work for large positive x, but notice that
$$\log(1+e^x) = x + \log(1+e^{-x}).$$
Use the function `log1p`, available both in `MATLAB` and `C++`, to evaluate $\log(1+e^x)$ for $x < 0$. In short, to evaluate $[\log(1+e^{x_i})]$ for a vector $[x_i]$, you could make a case distinction between $x_i < 0$ and $x_i \geq 0$. You do not need a `for` loop for that in `MATLAB` : the expression `find(x>=0)` gives you the positions in $\boldsymbol{x}$ where $x_i \geq 0$, so you can compute the two parts separately. If in doubt about such points, please contact your assigned TA and show him/her your code.

- Choose small learning rates (such as 0.01). Experiment with different values and with a momentum term.

- Since the objective that we optimize is highly non-convex, a good initialization is crucial for the algorithm to converge to a good solution. Initialize all weights to small values drawn at random from a Gaussian $N(0, \sigma^2)$ where a good heuristic for the variance σ^2 is the inverse of the number of inputs to the correspinding hidden node, e.g. for a weight on the first layer it should be $\sigma^2 = \frac{1}{d}$ where d is the number of input dimensions. Play around with these settings.

- Run your code several times from different initial settings. Do you always obtain similar results?

Debugging

Methods based on numerical optimization, in particular non-convex minimization as for MLPs, can be difficult to debug. As an important lesson to be taken away, you should implement the following measures (please do show that you have done so to your TA, and ask questions if something is not clear).
The most important lesson is to **test the gradients** which your backpropagation code produces. Gradient testing is discussed in Section 3.4.1 ("Gradient descent optimization in practice") in the course notes. Once you have established the correctness of your gradient code, a second property to test is whether your gradient descent optimization decreases the criterion value in every step. If this does not happen, try to reduce the learning rate and play with the momentum term. If problems persist, there is usually a bug.
One general comment. Do not debug your code on the full training set, choose smaller tasks to speed up bug-finding:

- Test the gradient on single data points (a few different ones), do not average over all the training data. You will use stochastic gradient descent anyway.
- Choose a small number h_1 of hidden units, first $h_1 = 1$, but then also $h_1 > 1$.
- NORB images are vectors in $\mathbb{R}^{c576}$. To save time, it makes sense to create a debugging dataset of a few patterns by downsampling the bitmaps. In `MATLAB` , the relevant command is `imresize`. Choose a resolution of 12×12 or even 8×8.

4.3 Model Evaluation

Now is a good time to read the beginning of Section 10.1 in the course notes, on evaluating models by way of an independent validation set. If you have not already done so, split your training dataset into a training part (2/3) and an evaluation part (1/3). You do your gradient descent training on the training part only, the validation set is reserved for:

- Early stopping (Section 7.2.1 of the course notes): Evaluate the classification error on the validation set after each run over all training points (this is called an epoch). Plot both the logistic error function on the training set and the validation error. Stop training once the latter starts increasing.
- Model selection: Compare different network architectures by their error on the validation set. In this case you have only one free parameter, that is, the number of neurons in the first and second layer.

Here some general advice on how to run your method, given you established that the implementation is correct. Notice that most people working with MLPs use a host of "tricks of the trade" which they learn by experience and playing around, so this is an integral part of this exercise.

- It is recommended[1] that you use *stochastic (online) gradient descent* (Section 2.4.2 in the course notes). Each epoch (run over all training points) consists of two phases. First, iterate over the training points in random ordering and update $\boldsymbol{w}$ based on the stochastic gradient of $E_{\log}(\boldsymbol{w})$ evaluated on single points. Second, evaluate the *full* logistic error function over *all* training points (as well as the validation error for early stopping), this is what you plot.

[1] It is admissable to use gradient descent on the criterion evaluated on the whole training set as well, but you will find that stochastic gradient descent runs faster.

Play around with the learning rate and the momentum term in order to improve convergence speed. Theory predicts that in order to get stochastic gradient descent to converge, the learning rate has to be decreased eventually, but you should stick with a fixed rate initially for faster decrease.

- To get a feeling of how learning rate and momentum term affect convergence, use a small training subset (say, 50 patterns) and compare plots. However, be aware that you will have to optimize these parameters for the final training set size eventually.

In machine learning, evaluation of a method (or a set of methods, for a comparative study) is a two-stage process. First, you play around and compare many different options, in order to find a configuration which works best. You train on the training set and use the validation set in order to compare choices. If you have a lot of data, it is best to use *different* validation sets to make different choices, so to reduce the risk of overfitting. Of course, **you never touch the test set at all**. In the second stage, you fix a setting which you are happy with and do a final training run with early stopping, then report test set performance of the final classifier. Here are some ideas for your first stage exploration (requirements for the second stage are given in Section 6.1):

- Compare curves for different numbers of hidden units (say, between 10 and 100), inspect both the training and the validation error. You should witness overfitting at some point (in order to force overfitting, you can also decrease the training set size).
- Compare different choices of learning rate and momentum term parameter. Do they only affect convergence speed or also the absolute error values at the end? What happens with too large a learning rate? With too little momentum?
- Run the algorithm several times from different initializations (all drawn at random, see above). Do you always get a similar final performance?
- Look at the misclassified patterns, in particular those for which the value $t_i a^{(2)}(\boldsymbol{x}_i)$ is largest negative, versus some for which $t_i a^{(2)}(\boldsymbol{x}_i)$ is close to zero, but negative. Any interpretation?

5 Task 2: Multi-way classification

In this task you are asked to classify images that come from all 5 different classes of the small NORB dataset.

5.1 Linear Classifiers

As a baseline for comparison your task is to implement linear methods for classification using

1. the squared error (ch. 4.2) and a Tikhonov regularizer (ch. 7.2)
2. the logistic error (no regularization, ch. 8.2)

For this part you should concatenate the inputs of the left and right camera to form a single input vector $\boldsymbol{x}_i = [\boldsymbol{x}_{L,i}^T, \boldsymbol{x}_{R,i}^T]^T$.

Squared Error

Recall from the course notes (ch. 5.2), that a linear classifier for K classes is of the form

$$f(\boldsymbol{x}) = \underset{k \in \{1,\dots,K\}}{\operatorname{argmax}} \; \boldsymbol{y}_k(\boldsymbol{x})$$

where for a linear model $\boldsymbol{y}_k(\boldsymbol{x}) = \boldsymbol{w}_k^T \boldsymbol{x}$ and the constant feature is added to the input to absorb the bias into the weights (ch. 2.2). Let $\boldsymbol{W} = [\boldsymbol{w}_k]^T$. Then $\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$.
A common way to learn this classifier is to use the squared error which is defined as

$$E_2(\boldsymbol{W}) = \frac{1}{2} \sum_{i=1}^{N} \|\boldsymbol{y}(\boldsymbol{x}_i) - \tilde{\boldsymbol{t}}_i\|^2 \tag{1}$$

where the vector $\tilde{\boldsymbol{t}}_i$ is the *1 of K* encoding of the target t_i as described in the course notes, ch. 8.3.1.
The Tikhonov regularized squared error function (ch. 7.2) is

$$E_{reg}(\boldsymbol{W}) = E_2(\boldsymbol{W}) + \nu \sum_{k=1}^{K} \|\boldsymbol{w}_k\|^2 \tag{2}$$

Optimization of this error function has an analytic solution as described in the course notes.

Cross validation

Perform model selection, i.e. determining the regularization constant ν, by using 10-fold cross validation (ch. 10).
Try at least 20 different values of ν, including $\nu = 0$. There are different cross-validation criteria (ch. 10.2) that you can choose from and should mention in your report.
Once you have chosen the value ν, use it to train on the whole **training** dataset.

Logistic Error

Another loss function is the multi-way logistic loss, with its conditional likelihood interpretation (ch. 8.3). It is given as:

$$E_{\log}(\boldsymbol{W}) = \sum_{i=1}^{N} \operatorname{lsexp}(\boldsymbol{y}(\boldsymbol{x}_i)) - \tilde{\boldsymbol{t}}_i^T \boldsymbol{y}(\boldsymbol{x}_i)$$

Unfortunately, there is no analytic solution to this error minimization problem, but you can use the whole machiniery that you implementated for task 1 (stochastic gradient descent with momentum and early stopping) to solve it, especially since gradient computations are simple compared to the MLP. Still, testing the gradient is highly recommended.

5.2 Multi-way MLP

Discriminating all 5 classes effectively constitutes a more difficult problem, where we can hope that increasing the complexity of the class of functions that the model can realize to admit non-linear decision boundaries can lead to better performance.
Based on your implementation from task 1 the main goal of task 2 is to train a MLP in the multi-class setting (ch. 8.3).
For that, consider the MLP architecture illustrated in figure 4.1. Extending this architecture designed for the binary case is straightforward: instead of a single output $\boldsymbol{a}^{(3)}$, extend the

network to have $K = 5$ outputs $\boldsymbol{y}_k(\boldsymbol{x}) = \boldsymbol{a}_k^{(3)}$, introducing new weight and bias parameters and leading to the additional forward equations for $k = 1, \ldots, K$:

$$a_k^{(3)} = (\boldsymbol{w}_k^{(3)})^T \boldsymbol{z}^{(2)} + b_k^{(3)}$$

Your task is to train this augmented network by minimizing the squared error (eq. 1). Due to the overlap with the original network, the gradient calculations will be largely the same. Identify which parts of your calculations and your code can be reused and extend the gradient testing to the new setting.
Keep in mind that a more flexible model might be required for this task, which means that you might end up with a larger number of hidden units. This can affect the performance of the backpropagation, if your code is not properly vectorized.
For the evaluation, you should apply the same basic principles as for the binary case.

6 Report

The report must be written in English and must not exceed 7 pages (use 10pt font). Any pages beyond the limit will not be taken into account. Please note, that you are free to consult textbooks, talk to friends, surf the web (but **do not copy code**). You should work in teams of two.

6.1 Requirements for the Report

Writing a good final report is part of the exercise, and you have some freedom how you do this. It is highly recommended **to show (parts of) your report to your TA early on and get feedback**. Here is what you should include:

(I) Introduction Write a short introduction (3-5 sentences only) stating the topic and aim of the project.

(II) Methods Describe what you did and how you did it. This section should suffice for someone (who knows about MLPs and has the course notes) to reproduce your results.

1. Describe your preprocessing if it deviated from the procedure described here (splitting ratio, different preprocessing, etc.)
2. Document and justify your model and parameter choices for all methods for both tasks:
    - Linear regression(task 2): perform 10-fold cross validation to select the regularization parameter ν. To this end, pick a sensible range of initial values (say, 10) that you can further refine. A common practice is to start with a range $\{a2^k | k = 0, \ldots, 9\}$.
    - Logistic regression(task 2) and MLP(task 1 and 2): show comparative plots that illustrate the effects of learning rate, number of hidden units(if applicable), momentum, etc. on (a) overfitting and (b) convergence speed, and comment on them qualitatively.

(III) Results & Discussion To evaluate the performace of you methods you should do the following

1. - Linear regression: Plot the average validation (classification) error for all different values of the normalization constant ν. Also plot the line (mean + standart deviation) above it. On the same plot and for the same values of ν plot the

average training error.
Once you have learned the parameters, use it to classify the images of the test set.

- Logistic regression and MLP: Plot curves of (a) the training set logistic error, (b) the validation set logistic error, (c) the validation set (zero/one) error as a function of the epoch number

$$\frac{1}{m}\sum_{j=1}^{m} \mathrm{I}_{\{t_j a^{(2)}(\boldsymbol{x}_j) \leq 0\}}.$$

2. Annotate each plot separately by the error on the test set of the final classifier and give the standard deviation (but do *not* plot a test error curve – remember that the test set is never used during training). Briefly discuss these findings.
3. Finally, plot examples of patterns which are misclassified, one for large negative $t_i a^{(2)}(\boldsymbol{x}_i)$, another for negative $t_i a^{(2)}(\boldsymbol{x}_i)$ close to zero. Calculate the confusion matrix[2] for the output of your classifiers for the test set. Comment on your findings.

7 Submission and Fraud Detection

There are two submission deadlines for the miniproject. Deadlines are strictly enforced, no exceptions are made. Every member of each team has to attend a *fraud detection* session, where he/she has to discuss details of the submitted code and report with a TA. This is a interview of approximately five minutes, where you should bring your computer and demonstrate that you are the original author of your code and report. **Each student has to attend the session on his/her own. Failure to attend the fraud detection session will result in zero points for the miniproject**.
Teams who submit at the first deadline are free to choose one of the two fraud detection sessions. Clearly indicate your preferred session together with your submission. Teams who submit at the second deadline have to attend the fraud detection session at the second date.

1. Submission deadline 24 May 2012; fraud detection session on 29 May 2012 or on 10 June 2012.

2. Submission deadline 7 June 2012; fraud detection session on 10 June 2012.

Each submission must be clearly marked with:

- Name and Sciper number of each group member

- In case of 24 May deadline: Which fraud detection session will each group member attend?

Note that for organisatorial reasons, **both group members have to submit the report on moodle**. Please submit exactly the same documents and files.

[2]A confusion matrix for a K class problem is a matrix $\boldsymbol{C} \in \mathbb{N}^{K\times K}$ with $\boldsymbol{C}_{i,j} = \sum_{k=1}^{N} \boldsymbol{I}_{\{t_k = i \wedge \hat{t}_k = j\}}$ where $t_k, \hat{t}_k$ are k-th true and estimated targets, respectively. For `MATLAB` users, check `confusionmat`.