Machine Learning Lab1

Alari Varmann and Octave Mariotti

February 2016

Introduction

All Training MSE graphs are on linear scale

Training Feedforward Neural Network Using Gradient Descent

The network is created using the following code:

```
net = newff(p, t, [2], {'tansig' 'logsig'}, 'traingd', ...
'', 'mse', {}, {}, '')
```

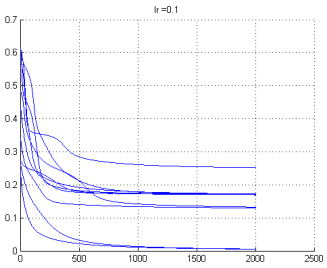
We see that in the hidden node, the activation function is the hyperbolic tangent and the in the output layer, it is the logistic sigmoid.

Question 1: As you have seen, the error (performance) sometimes converges to a value greater than zero. Explain why this happens.

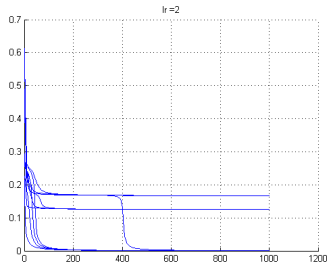
It is known that the gradient descent uses only order 1 derivative information from the Taylor expansion, so it doesn't give the fastest convergence. Every time before training, the network weights are randomised. Depending on the initial weights and error tolerance (training stopping criterion – low gradient value, RMSE or number of iterations), the network can get stuck in different local minima, which don't necessarily coincide with the global minimum. Since the stopping criterion is satisfied, the training will stop and the error won't be decreased any further. That's it.

Question 2: How does the learning rate affect the training of the network? What are the effects of using a too low value? What are the effects of using a too high value?

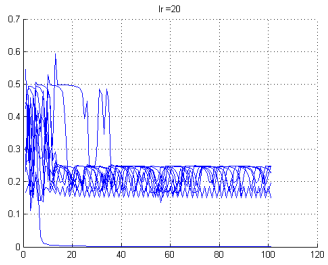
Very low learning rate makes the learning possibly very slow, since the weight changes are relatively small. Higher learning rates cause the network to consecutively jump back and forth /zigzag over the minimum. Some heuristic should be utilized (line search, different strategies) to make the network adapt to the change in the gradient magnitude.



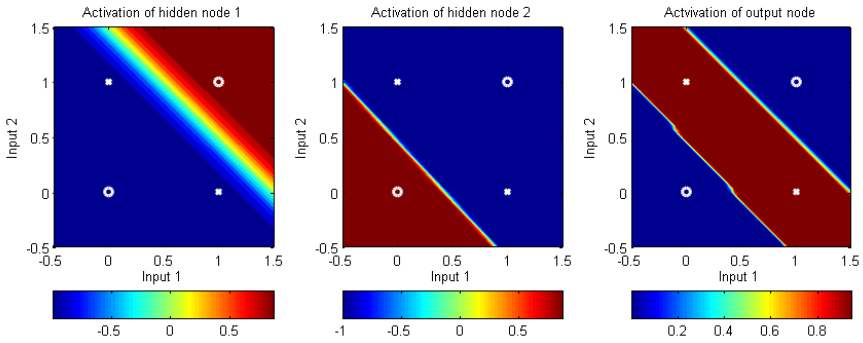
Backprop training MSE, learning rate 0.1



Backprop training MSE, learning rate 2



Backprop training MSE, learning rate 20



First solution

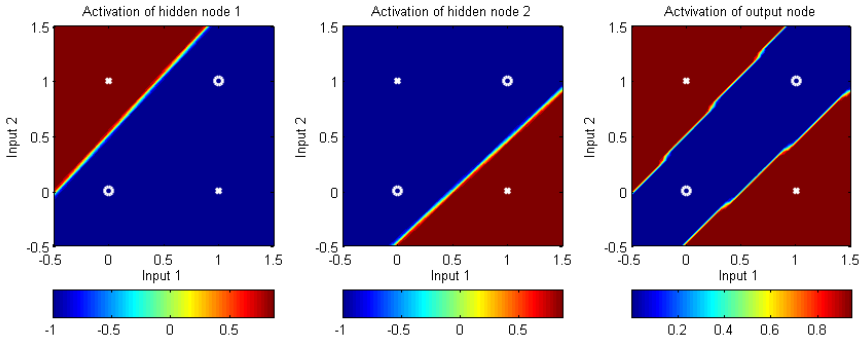
Plot 2: XOR-network results

Question 3: Why are the activations of the hidden nodes in the range $[-1, 1]$, but the output of the network in the range $[0, 1]$?

The activations of the hidden nodes are given by the hyperbolic tangent function

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \rightarrow \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

with range $[-1, 1]$, and visualized on $\mathbb{R}^2$ it looks basically like a logistic sigmoid ex-



Second solution

tended to the lower half plane. The logistic sigmoid on the other hand is a more suitable function in a statistical sense since it can be seen as a density function of the weighted sum of hidden node activations, its range is $[0, 1]$.

Generative vs Discriminative Models.

As we know: "Discriminative models, as opposed to generative models, do not allow one to generate samples from the joint distribution of x and y. However, for tasks such as *classification and regression (that do not require the joint distribution)*, **discriminative models can yield superior performance**. On the other hand, *generative models are typically more flexible* than discriminative models in expressing dependencies in complex learning tasks. In addition, most discriminative models are inherently supervised and cannot easily be extended to unsupervised learning." [2]

Logistic Regression

Logistic regression is one type of a discriminative probabilistic model, more specifically a type of generalized linear (in this case linear of hidden layer activation transformation) regression used for predicting binary or categorical outputs (also known as maximum entropy classifiers).

Consider the case of binary classification in which one has a single target variable t such that $t = 1$ denotes class C_1 and $t = 0$ denotes class C_2. The activation $y = \sigma(a)$. We can interpret $y(\mathbf{x}, \mathbf{w})$ as $p(C_1|\mathbf{x})$ with $p(C_2|\mathbf{x})$ given by $1 - y(\mathbf{x}, \mathbf{w})$. The conditional distribution of targets given inputs is a Bernoulli distribution:

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t (1 - y)(\mathbf{x}, \mathbf{w})^{1-t}.$$

.

Logistic Regression General Idea

Limit to only one hidden layer here (no deep network). In general, **logistic regression** is a technique used for **generalized classification problems** – the output result can be regarded as a vector of posterior probabilities of sigmoidal class activations for each class (in the general case given by the **softmax** activation function) given a nonlinear input data dependent transformation(s) ϕ, which correspond to the activations of the hidden layer. If we had K separate binary classifications to perform, use K-fold binary classification with Bernoulli targets $t_k \in \{0, 1\}$. Then a network having K output nodes is used ($k = 1, \ldots, K$), each of which has a logistic sigmoid activation function. The k'th output node activation would be:

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\big(\sum_{j=0}^{K} w_{kj}^{(2)} \phi\big[\sum_{i=0}^{M} w_{ji}^{(1)} x_i\big]\big).$$

Our example – XOR Implementation

The logistic sigmoid specifies the activation distribution for each class. Given training data $m = 1, \ldots, M = 4$, $\{\phi(\mathbf{x}_m), \mathbf{T}\} = \{\tanh(\mathbf{x}_m), \mathbf{t}\}, \mathbf{T} = (0, 1, 1, 0)^T$, we need to find the weight vectors $\mathbf{w^{(1)}}, \mathbf{w^{(2)}}$ of the hidden and output layer that maximize the likelihood function l of getting the target output XOR $\mathbf{T} = \mathbf{t}$. The Bernoulli targets for XOR $t_m \in \{0, 1\}$ and denote $\mathbf{w} = (\mathbf{w}^{(1)}, \mathbf{w}^{(2)})$, where 1 corresponds to the hidden layer and 2 the output layer. Assume that the class labels are independent, given the input vector, then the posterior distribution of the target vector component t_n is: $p(t_n|\mathbf{x_n}, \mathbf{w}) = y_n(\mathbf{x_n}, \mathbf{w})^{t_n}\{1 - y_n(\mathbf{x_n}, \mathbf{w})\}^{1-t_n}$. In our case we have only one output node, $m = 1, \ldots, M = 4$ and since

$$p(t_m|\mathbf{w}, \mathbf{x_m}) = \sigma\big(\mathbf{w}^{(2)T}\phi(\sum_{i=0}^{2} w_{ji}^{(1)} x_{mi})\big)^{t_m}\big(1 - \sigma(\mathbf{w}^{(2)T}\phi(\sum_{i=0}^{2} w_{ji}^{(1)} x_{mi}))\big)^{1-t_m},$$

thus the likelihood of getting XOR is :

$$l = p(\mathbf{t} = (0, 1, 1, 0)^T|\mathbf{x}, \mathbf{w}) = \prod_{m=1}^{4} p(t_m|\mathbf{w}, \mathbf{x_m})$$

The hidden layer performs nonlinear feature extraction, and sharing of features between different outputs can save on computation and lead to improved generalization.

We have 2 classes, although the decision boundary for XOR is disconnected. Knowing the boolean solution for the XOR problem, we would then look for the two hyperplanes defined by the weights for the hidden and output layer – that would linearly separate the outputs into the 2 classes. Since we seek to implement XOR and we know that the decision surface, which corresponds to constant input to the output layer, is a linear function of *hidden layer activation functions*, we will have two corresponding hyperplanes.

Because we have a **discriminative model**, we don't model the inputs, so instead the cross-entropy error function –the negative logarithm likelihood function is:

$$E(\mathbf{w}) = -\ln(l) = -\sum_{m=1}^{M} \{t_m \ln y_m + (1 - t_m) \ln(1 - y_m)\}$$

and taking gradient w.r.t. $\mathbf{w}$ yields: $\nabla E(\mathbf{w}) = \sum_{m=1}^{M}(y_m - t_m)\phi_m$, which is basically the sum-of-squares error function for a linear regression model. In **Backprop and Rprop**, these gradients are computed via (recursive) use of the chain rule, combined with reuse of information that is needed for more than one gradient.That is the basis for the generalized delta rule $\Delta w_{ji} = \eta\delta_j x_i$ as well (that uses squared loss function as well).

Here a 2-layer neural network logistic tanh and sigmoid activation function is trained to find the weight vectors. Since y depends on $\mathbf{w}$ non-linearly, there is no closed form solution – **thus the neural networks to find those weight vectors.** The algorithms used are gradient-based optimization algorithms backpropagation (backprop) and resilient backpropagation (Rprop). Rprop converges much faster than the backprop in our problems.

Question 4: Question 4: Why does the training not always end up with the same solution, even when the same training parameters are used?

Note that actually the training could end up exactly with the same solution, unless we **initialize** the network. The randomness comes into our neural network through the random initialization of the weights, which is done before training. First we tried training without prior initialization, and we got exactly the same results. Given that all the parameters maintain constant, gradient descent is a deterministic algorithm, there is no chance in there.

Question 5: Write down the boolean functions H_1 ,H_2 and O for each of the two different solutions identified above with plot ***XOR***. Use only the logical connectives and, or and not.

We see that the first network actually implements :

$$\mathcal{O} = \overline{H_1 \wedge H_2}, \text{ with } \begin{cases} H_1 = I_1 \wedge I_2 \\ H_2 = \overline{I_1 \vee I_2} \end{cases}$$

And the second :

$$\mathcal{O} = H_1 \vee H_2, \text{ with } \begin{cases} H_1 = \overline{I_1} \wedge I_2 \\ H_2 = I_1 \wedge \overline{I_2} \end{cases}$$

Question 6: Verify (e.g., with a truth table) that the functions actually implement xor when combined

The truth Table for the first solution is :

I_1	0	0	1	1
I_2	0	1	0	1
$H_1 = I_1 \wedge I_2$	0	0	0	1
$H_2 = \overline{I_1 \vee I_2}$	0	1	1	1
$O = \overline{H_1 \wedge H_2}$	0	1	1	0

And for the second :

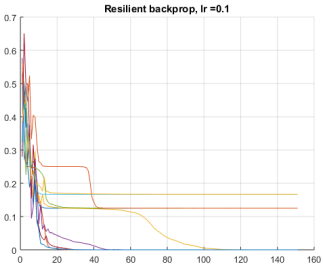
I_1	0	0	1	1
I_2	0	1	0	1
$H_1 = \overline{I_1} \wedge I_2$	0	1	0	0
$H_2 = I_1 \wedge \overline{I_2}$	0	0	1	0
$O = H_1 \vee H_2$	0	1	1	0

Resilient backpropagation

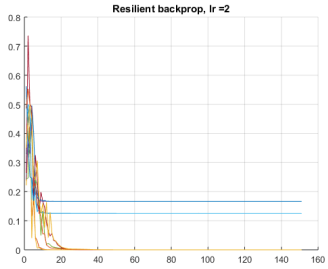
We used the same learning rates for comparison. The parameters we used were:

`epochs = 150, delta0 = 2, deltamax = 8, delta_inc = 2, delta_dec = 0.5`

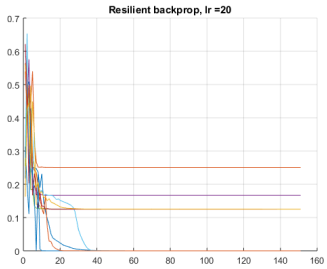
The training error outcomes were the following:



Rprop training MSE, learning rate 0.1

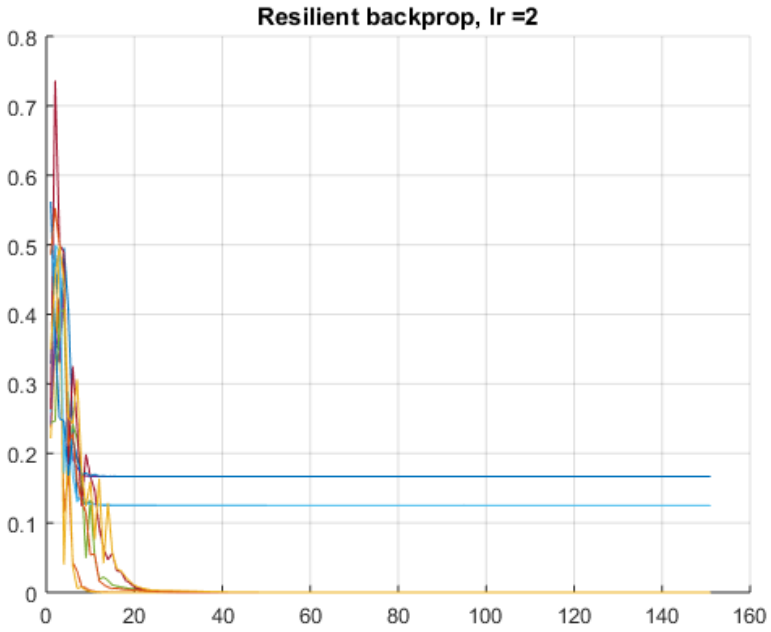


Rprop training MSE, learning rate 2



Rprop training MSE, learning rate 20

We decided to choose the one with learning rate 2.



Plot 3:Learning rate 2.

Task 2: Function Approximation

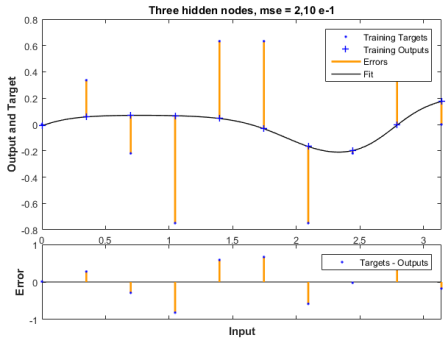
Question 13: Differences between backpropagation and resilient backpropagation

Plot 3

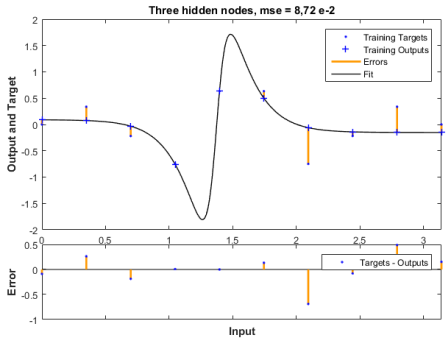
Parameters used	
Epochs	5000
η	0.01
Δ_0	1
Δ_{max}	10
Δ_{inc}	0.8
Δ_{dec}	1.2

Question 8

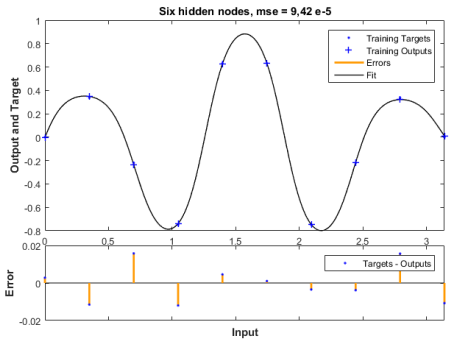
The training error is minimum with 20 hidden nodes. It seems that it is easier to achieve a low training error with a high number of hidden nodes



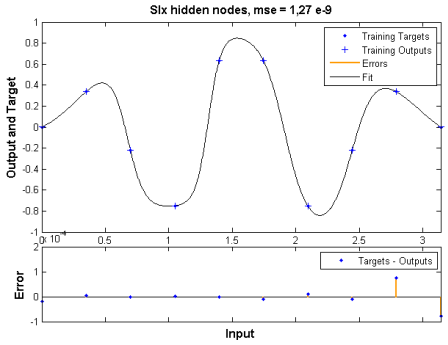
(a) Three Hidden nodes - backprop



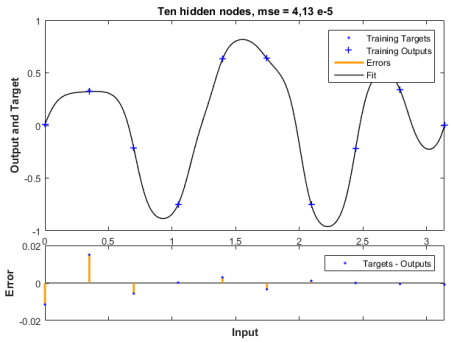
(b) Three Hidden nodes - resilient backprop



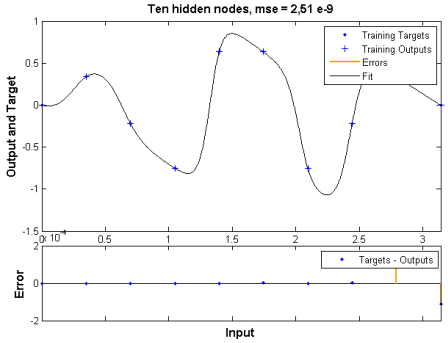
(a) Six Hidden nodes - backprop



(b) Six Hidden nodes - resilient backprop



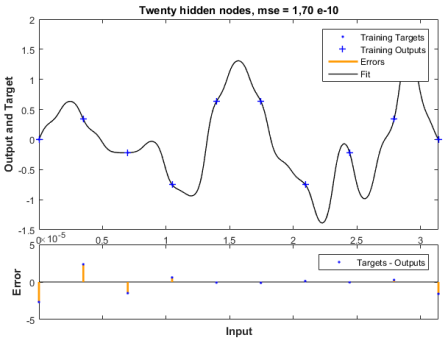
(a) Ten Hidden nodes - backprop



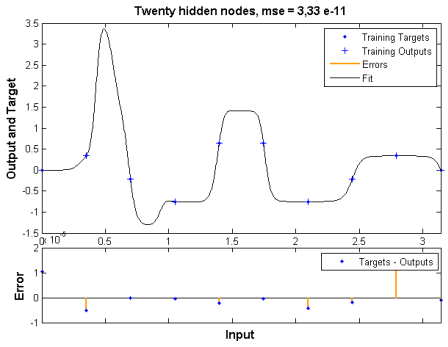
(b) Ten Hidden nodes - resilient backprop

Question 9

The best approximation of the function $f : x \mapsto sin(x) \cdot sin(5x)$ seems to be achieved with six hidden nodes : the implemented function "looks" the closest to f. Here, the MSE does not seem to be relevant, as it is a discrete measure of error, and we want



(a) Twenty Hidden nodes - backprop



(b) Twenty Hidden nodes - resilient backprop

to approximate a continuous set. We should actually look at something like a function distance, for instance : $\max_{x\in[0,\pi]} |f(x) - g(x)|$ where g is the function implemented by the network. this of course is hard to calculate, hence the importance here of a validation set.

Question 10

If we have too few hidden nodes — three, for instance — the network will not be able to approximate the function at all. This is because the number of hidden nodes can be directly seen as the number of monotonic part of the implemented function. In this case, the function is too much restricted to achieve a decent approximation of f.

Question 11

On the contrary, if we have too many hidden nodes, the network is not bounded enough. This result in overtraining : the network tries to approach the discrete set of points it is given, trying to minimize error as much as possible to the detriment of the rest of the function domain.

Question 12

As stated before, the number of hidden nodes is the number of monotonic parts in the function implemented by the network. In this case, we can see that the slope of the target function changes sign five times, giving six monotonous sections. It would then be sound to think that a number close to six gives the best results. This is exactly what happens here, as the network having six hidden nodes displays the best results.

Question 13

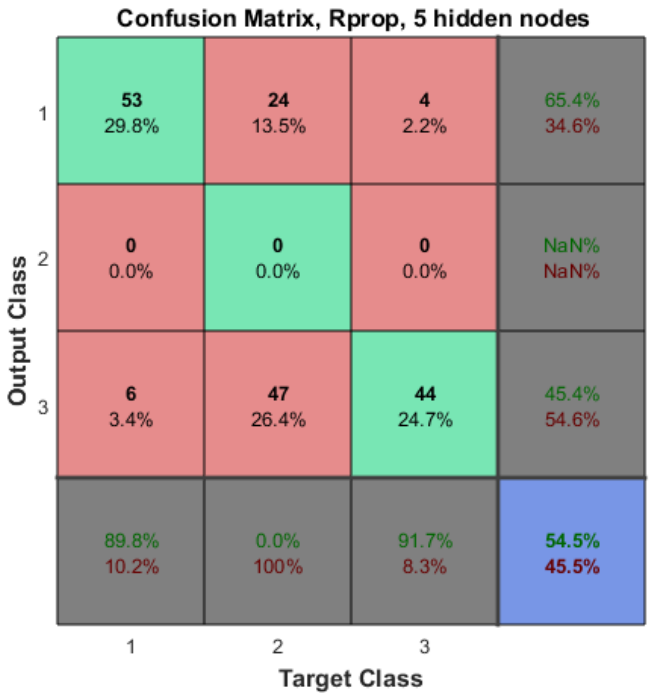
The functions obtained with regular backpropagation seem somewhat smoother than those obtained with Rprop. This is not necessarily a good thing, in particular when having too few hidden nodes (cf three hidden nodes network), but with a correct size of the network, the result looks closer.

Of course, this is only regarding this problem. It is likely that Rprop approximates some other function better than backpropagation.

Task 3 – Wine Data Classification

Question 14: What settings did you use to get good results? What was the smallest number of hidden nodes that you needed? Approximately how many percent of the wines are placed in the right class after training this network?

```
net4 = newff(p, t, [5], {'tansig' 'logsig'}, 'trainrp', ...
'', 'mse', {}, {}, '')
net4 =init(net4);
net4.trainParam.epochs = 1000
net4.trainParam.delta0 = 0
net4.trainParam.deltamax = 7
net4.trainParam.delta_inc = 1
net4.trainParam.delta_dec = 0.3
net.trainParam.min_grad = 0
```



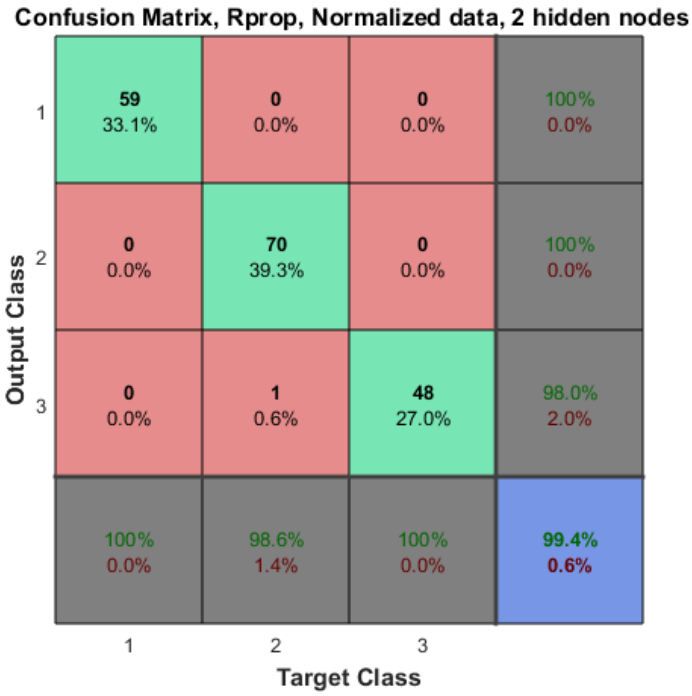
Best achieved Confusion Matrix.

Approximately 45.5 per cent of the wines were misclassified by this trained Rprop neural network. The network can also achieve over 30 per cent performance with only 1 hidden node, but we obtained a better result with 5 hidden nodes.

```
In general we realize that to get better than random chance (p=1/3) results with unnormalized input
```

Question 15: Did the normalization have any significant impact on the results of the training? How many hidden nodes did you need now in order to get good results?

```
Minimum Gradient                 min_grad: 1e-05
   Maximum Validation Checks        max_fail: 6
   Initial Delta                      delta0: 0.07
   Delta Increase                   delt_inc: 1.2
   Delta Decrease                   delt_dec: 0.5
   Maximum Delta                    deltamax: 50
```



Best result – 99.4 per cent with 5 hidden nodes.

Only 0.6 per cent of the wines were misclassified by this trained Rprop neural network. **We tried out a few other combinations of parameters as well that yielded worse than chance ($p = 1/3$) results.** This is actually very interesting and indicates

that normalization may help to achieve a lot better training results – but above 99 per cent is probably very much overfitting. Note that in real life, we don't care so much about the training error as compared to the generalization error, which is unknown since we don't have infinite amount of data.

Question 16: In general, will normalization always help in training an MLP? What possible dangers can normalization in this way pose to a supervised learning problem?

No, it might not necessarily always help, but it doesn't complicate training either ("rarely hurts"). If there are a lot of classes, it will be difficult to achieve a high classification accuracy probably in both cases, since the likelihood function is a product of individual posterior probabilities for each class.It is obvious that if there are a lot of classes it will be harder to find the weights that achieve a lot better than chance classification accuracy (low F1-score).

In general, for example for the k-Means algorithm or radial basis function network, where distance measures are used, data normalization is essential since otherwise the feature with high magnitude will dominate in the calculate the metric similarity.

In general,we thought that centering or normalizing it will help to make the data more uniform to a fixed learning rate since the update in weight is implemented through

$$\Delta w_{ji} = \eta \delta_j x_i \quad \text{generalized delta rule,}$$

meaning that the change in weight is proportional to the input – so smaller inputs would always have a smaller influence on the training outcome. This would just mean that the training could be slower due to very different changes in weights?
[1] **"If the input variables are combined linearly, as in an MLP,** then

`it is rarely strictly necessary to standardize the inputs}, at least in theory.`

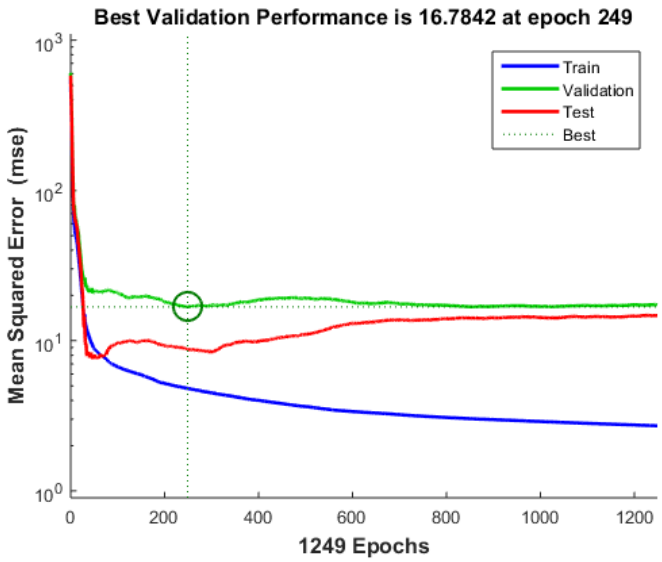
Changing the corresponding weights and biases, leaving one with the exact same outputs as before. However, there are a variety of practical reasons why standardizing the inputs can make training faster and reduce the chances of getting stuck in local optima.

Standardizing inputs removes the problem of scale dependence of the initial weights. Explanation: Assume we have an MLP with one hidden layer applied to a classification problem and are therefore interested in the hyperplanes defined by each hidden unit. *Each hyperplane is the locus of points where the net-input to the hidden unit is zero* and is thus the classification boundary generated by that hidden unit considered in isolation. The connection weights from the inputs to a hidden unit determine the orientation of the hyperplane. The bias determines the distance of the hyperplane from the origin. If the bias terms are all small random numbers, then all the hyperplanes will pass close to the origin. Hence, if the data are not centered at the origin, the hyperplane may fail to pass through the data cloud. If all the inputs have a small coefficient of variation, it is quite possible that all the initial hyperplanes will miss the data entirely. With such a poor initialization, local minima are very likely to occur. It is therefore important to center the inputs to get good random initializations. In particular, scaling

the inputs to $[-1, 1]$ will work better than $[0, 1]$, although any scaling that sets to zero the mean or median or other measure of central tendency is likely to be as good, and robust estimators of location and scale (Iglewicz, 1983) will be even better for input variables with extreme outliers. ”

Moreover, on a much broader level, any kind of transformation on the input data may yield loss of information. If for instance our initial distribution contains value distributed over a large interval, it may be nice to use a logarithmic scale. However, in doing so, the larger values will be gathered in a domain much narrower than before, and the smaller will get scattered. Even linear transformation may cause harm due to rounding errors.

Task4 – Approximating House Prices



Plot 5 :Performance plot of Housing.

Question 17: The error of the trained network usually differs for the three sets. Explain why it differs, and what it means for the performance of the network in general

The training error will differ because of two arguments:

- The training data is selected randomly at every training, so the performance graph couldn't possibly be identical if the training data itself is different
- Starting weights initialized randomly

In general, we cannot say anything about the generalization capacity of the network just by the magnitude of the training error. The smaller the training error, possibly the more the network can be prone to overfitting, meaning that the network can become sensitive to the noise in the sample.

Question 18: As the training goes along, the three error curves usually behave differently. Describe these trends, and explain why the curves behave in such a way.

Expected prediction error The expected prediction error $I(f_N)$ of a particular function over all possible values of x and y is

$$I(f_N) = \int_{X \times Y} L(f(x), y)\rho(x, y)\mathrm{d}x\mathrm{d}y,$$

where $\rho(x, y)$ is the joint distribution of the data, which we don't know — the data is modelled using this in probabilistic generative models. Instead of the expected prediction error, we have the **empirical error**, which we can compute from our sample

$$I_S(f_N) = \frac{1}{N}\sum_{i=1}^{N} L(f_N(x_i), y_i)$$

Generalization error is the difference between the expected and empirical error. The algorithm is said to generalize if $\lim_{N\to\infty} I(f_N) - I_S(f_N) = 0$.
The training error decreases in general as a function of epochs because that is the basis for how the gradient-based optimisation algorithms are built. We use the validation/test set to approximate the generalization error of the network, though the test set will be left for the final evaluation. The validation and test set errors usually decrease in the first part of the training, after which they will usually start to increase – that is where the network will start to fine-tune itself to the training data set. We should use the validation data set to determine the training time – i.e. when to stop training. If we really are interested in the generalization capacity of the network, then we should choose the parameters that minimize the validation error or the cross-validation error, if we have a small amount of data.

Question 19: In light of your answer to the question above, what should one have in mind (in terms of training epochs) when training a neural network?

- First do some quick probing to test out training the network with many different parameters to somehow systematically decrease the search domain of parameters.
- Once some decent combinations have been found, then do more large-scale training using those (can run simultaneously) and assess the performance – either based on logical thinking and graphical evaluation (e.g. function fitting) or on (cross) validation error.

- The number of epochs should be chosen such that to minimize the (cross) validation error.
- In general: Choose the complexity of the network based on the preference for bias-variance trade-off and the underlying task. Consider other metrics than squared loss function, that is, consider different regularizations.

-

Question 20 –Own Example

We figured that a network implementing the delta rule to estimate the posterior target probability given sensory input of one or many modalities could be an interesting idea – essentially this could be called a generalized uniclass classification. This process happens in the superior colliculus neurons region of the brain. Training is done on simulated data — we use a probabilistic generative model.
The delta rule would be implementing the famous Bayes' Theorem:

$$p(T=1|V=v_s) = \frac{p(V=v_s|T=1)P(T=1)}{p(V=v_s|T=0)P(T=0) + p(V=v_s|T=1)p(T=1)}$$

to find the posterior probability that the target is present given a certain visual stimulus. The denominator is called the evidence, since we can do measurements with targets present and record how that affects the recording of the stimuli. $p(T=t)$ terms are called the prior, since we have to know the distribution of the targets before we wish to compute the posterior distribution. The terms $p(V=v_s|T=0)$ are called likelihood functions – they measure how the presence of target affects the distribution of the visual stimuli and it is natural to assume that given a target, the stimulus would be normally distributed. To implement the Bayes' rule, first initialize the parameters - like learning rate, bias, iterations, number of inputs and outputs and likelihood function parameters and prior probabilities. Generate the visual input space and initialize the connectivity matrix. Implement the network using the delta-rule(sigmoidal activation). The visual input given the target, as mentioned, are drawn from differently parametrized Normal distributions. When plotting the activation of the output node, it should turn out that it is basically indeed implementing the Bayes' rule to find the posterior distribution for the target – the posterior distribution turns out to be the logistic sigmoid of the input stimulus – the stronger the stimulus, the more likely the target is to be present.

References

[1] Section - Should I normalize/standardize/rescale the [WWW] `http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html`

[2] Discriminative Model [www] `https://en.wikipedia.org/wiki/Discriminative_model`