Supervised Learning and Generalisation

James O'Reilly
james.oreilly@student.kuleuven.be

Introduction

This report follows my first coursework as part of the 'Neural Networks and Deep Learning' course at KU Leuven. This coursework focuses on supervised learning in simple feed-forward networks. The report is split into three parts:

- Comparing the performance of different learning algorithms for a simple curve fitting problem, and testing how well these algorithms generalise to noisy data.
- Approximating a 2D function using a feed-forward network. This section focuses on how data is divided before training, as well as selecting network hyper-parameters.
- Finally, following from section two, we look at the Bayesian treatment of neural networks and Bayesian inference of network hyper-parameters.

1 Comparison of Different Learning Algorithms

A number of neural networks were trained for a simple function approximation task with different learning algorithms. The underlying function is given by

$$y = sin(x^2), \quad 0 \leq x \leq 3\pi$$

Data points were sampled at intervals of 0.05. A list of the different training algorithms that were compared is given below:

- Gradient descent (GD)
- Gradient descent with adaptive learning rate (GDA)
- Conjugate gradient backpropogation with Fletcher-Reeves updates (CGF)
- Conjugate gradient backpropagation with Polak-Ribiére update (CGP)
- BFGS quasi-Newton backpropogation (BFG)
- Levenberg-Marquardt backpropogation (LM)

1.1 Network Configuration and Architecture

When comparing the learning algorithms, network hyper-parameters such as the number of neurons, the number of layers, initialisation weights, and transfer function were kept constant. The network has a single dense hidden layer with 30 neurons, each with a `tanh` activation function. It will be clear later that 30 neurons is not sufficient to accurately approximate the function, however limiting the neurons, and therefore the performance, allows for a much clearer comparison of the algorithms. If 50 or 60 neurons are used in the hidden layer, then the quasi-Newton and LM algorithms both reach a very low error on the test set, and it becomes difficult to see the difference in performance. The data set was divided randomly with 70% used for training and 15% for both validation and testing. Early stopping is implemented by stopping training if the MSE on the validation set increases for six epochs consecutively.

1.2 Comparison

A number of metrics were used to compare the performance of the learning algorithms:

- Mean-squared error (MSE) on the test set
- Regression between the network outputs and targets
- Time taken to converge (number of epochs). Convergence is determined by the best performance on the validation set

The performance metrics were measured for networks trained for 1, 14, and 100 epochs. Each metric was measured by running the network 50 times and then averaging the results.

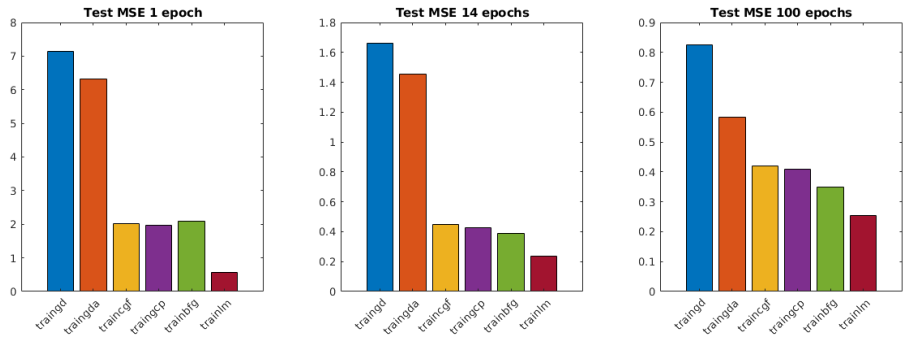

Figure 1: Test set MSE for different numbers of epochs. Note the different y-axis ranges.

It is clear that basic gradient descent performs the worst across the range of epochs, while Levenberg-Marquardt consistently outperforms all other learning algorithms. GDA gives only a slight improvement over standard GD. CGF, CGP, and BFG far outperform GD across the range of epochs. The performance of BFG relative to CFG and CGP improves as the number of epochs increases.

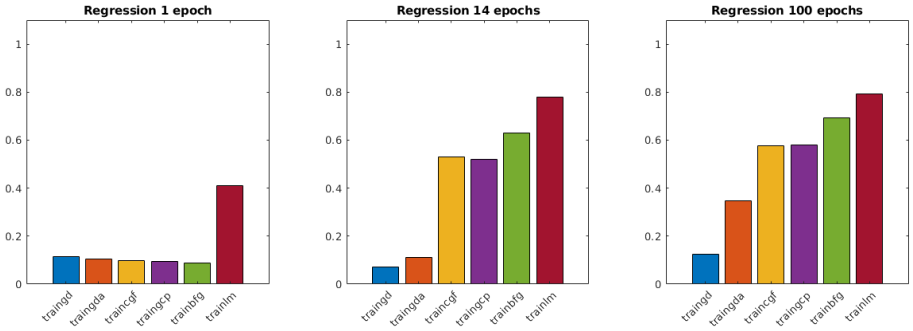

Figure 2: Regression between approximation and target for different numbers of epochs. Note the different y-axis ranges.

As would be expected, the regression values reflect the MSE results for the relative performance of the different learning algorithms. It is interesting to look at the average time taken to converge for the algorithms. This is given by the epoch at which the MSE on the validation set is minimised.

Note that LM converges most rapidly, while BFG is slower to converge on average. Naturally, Levenberg-Marquardt will converge faster than gradient descent in most cases as the varying dampening parameter μ allows it to interpolate between Gauss-Newton method and gradient descent. Note also that as LM converges rapidly and consistently within the maximum number of epochs, it is possible to increase the maximum value of the dampening parameter μ as we can afford slower convergence to the optimum. Increasing the upper range of μ could potentially give better performance, at the cost of speed. Both GD and GDA are slow to converge, with GDA converging due to the adaptive learning rate.

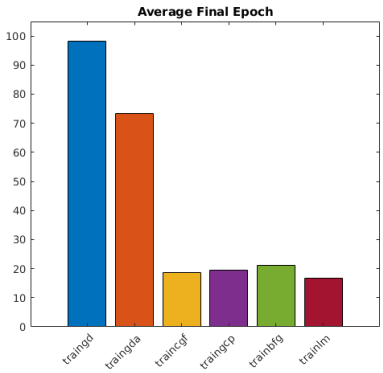

Figure 3: Number of epochs until convergence for each learning algorithm. The maximum number of epochs is 100.

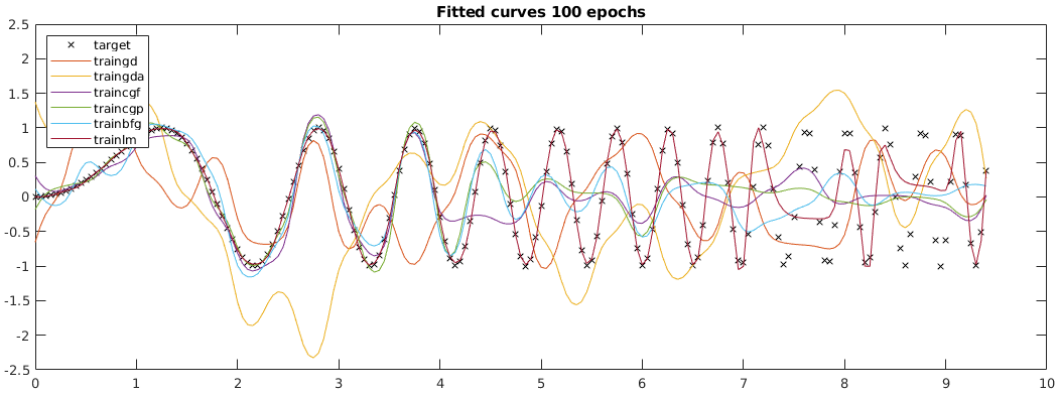

Figure 4: The fitted curves for 100 epochs.

1.3 Noisy Data and Generalisation

To test the performance of the learning algorithms with noisy data, additive white Gaussian noise was added to the original function with a signal-to-noise ratio of 10. The networks were then trained with under the same conditions as before and the average MSE, correlation, and time to convergence was recorded for each of the algorithms.

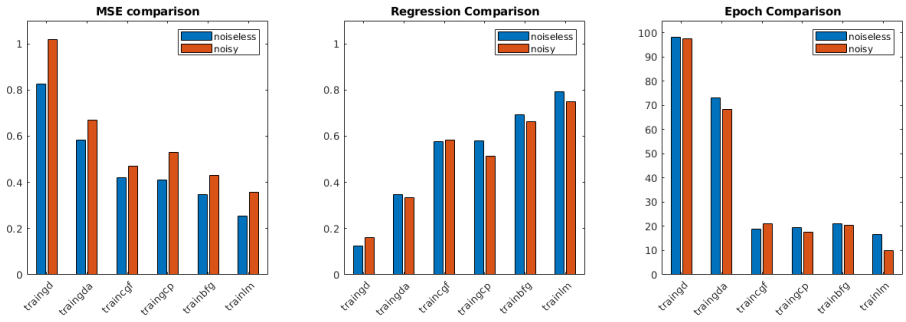

Figure 5: Comparison of performance on noiseless vs noisy data. These results are averaged over 50 runs.

Comparing the results of the noisy and noiseless function approximation show that in general both the test MSE and the regression were slightly worse for the noisy data, as would be expected. This was the case for every learning algorithm except for CGF, which had a better regression score with the noisy data. On average, the networks converged faster when trained on the noisy data, again with the exception of CGF.

2 Approximating a 2D Function using a Feed-forward Network

Here we approximate a 2D function using a multi-layer feed-forward network. This section focuses on data division and how network hyper-parameters are selected for.

2.1 Data Division

The data was sampled randomly from the dataset without replacement and split into three evenly-sized sets for training, validation and testing. What is the rationale behind random division? It is instructive to look at the alternative: dividing the data into contiguous blocks. The issue with dividing the data into contiguous blocks is that if the structure of the data is not consistent throughout the entire dataset, the network will not generalise well. The data is sampled without replacement because if there is an overlap between the training set and the validation or test set, the performance metrics for the validation and test set will be biased as they will be measured with data points on which the metric was trained. See Figure 6 for a visualisation of the training and test set used.

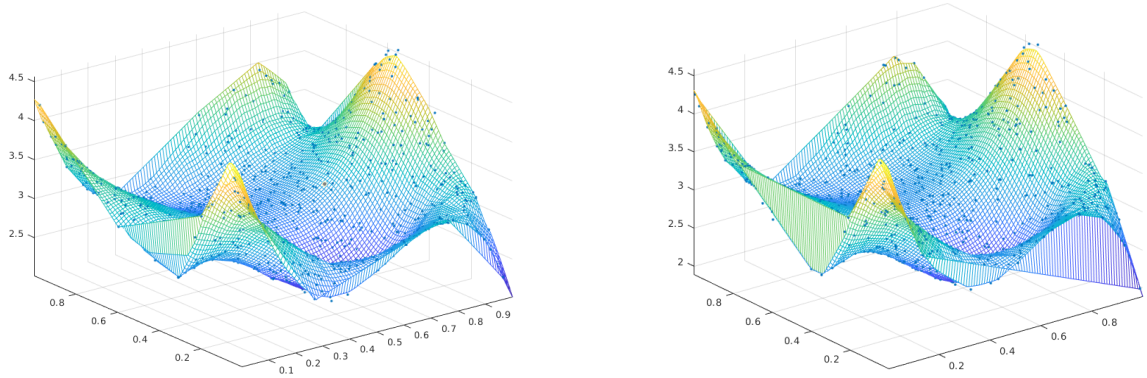

Figure 6: A visualisation of the training set (left) and test set (right) surfaces. The actual data points which were used to form the mesh are shown as blue dots. The roughness at the edge of the plot is from the interpolation for forming the mesh, and not from the dataset itself.

2.2 Building the Network and Choosing the Hyper-parameters

Based off the results from the function approximation in the previous section, I decided to use Levenberg-Marquardt as the learning algorithm. Before training the network, it's also important to consider the nature of the input data and the function that we are trying to approximate. The function has only positive values for both inputs and outputs, with inputs X_1 and X_2 both in the range $[0, 1]$, and so data-scaling is not needed as the input data is already properly scaled between 0 and 1.

2.2.1 Activation Function

For the activation function, I decided to use ReLU. Generally the two main benefits of using ReLU are that it avoids the vanishing gradient problem and is computationally more efficient. As there are only a few layers in the network (discussed below), the vanishing gradient problem is not a huge concern and the primary reason for using ReLU is computational efficiency. The decision must then be made between using standard ReLU or some ReLU variant such as leaky ReLU, parametric ReLU, or ELU (exponential linear unit). This decision is determined by whether or not one is concerned with the 'dying ReLU' problem. The aforementioned extensions to ReLU each relax the non-linear output of the function to allow small negative values in some way, and therefore solve this problem. It was decided to use standard ReLU and to instead try to initialise weights and biases in a way which avoids the dying ReLU problem as much as possible.

Firstly, a smaller bias input value of 0.1 was used. This makes it likely that the units will be initially active for most inputs in the training set and allow the derivatives to pass through. Secondly, a different weight initialisation scheme should be used when using ReLU. If the weights are initialised to small random values centered on zero, then by default half of the units in the network will output a zero value. One initialisation scheme which could be used is 'Kaiming initialisation'. This initialisation scheme causes each individual ReLU

layer to have a standard deviation of 1 on average. keeping the standard deviation of layers' activations around 1 allows stacking of several more layers in a deep neural network without gradients exploding or vanishing. Ultimately I couldn't effectively implement these weight and bias initialisation schemes in Matlab, but I think its important to mention them nonetheless. However, after training the network multiple times using ReLU layers, I found that with my (likely faulty) implementation it consistently performed worse than with a *tanh* activation function, and so *tanh* is used in the sections that follow on hyper-parameter tuning and Bayesian regularisation.

2.2.2 Determining Neural Network Topology

Determining both the optimal number of hidden layers and neurons is very important. One way to determine these values manually is to perform a grid search over the hyper-parameter space. First the number of neurons was first kept constant and the number of layers was varied. Fixing the number of neurons at 60, networks with one up to five hidden layers were trained. The performance of these networks over many runs was averaged to determine the effective number of layers. Increasing the number of layers improved performance on the validation set, but the performance began to drop once the 60 neurons were split between more than four layers. Splitting the 60 neurons between two or three layers gave the best performance. I chose to use two hidden layers.

No. of Hidden Layers	1	2	3	4
Average Test MSE ($\times 10^{-4}$)	0.1223	0.0468	0.0487	0.0730

Table 1: Average test MSE for different numbers of hidden layers. The results were averaged over 10 runs and the networks were each trained for 200 epochs.

Once then number of hidden layers was decided, this number was kept fixed while the number of neurons was increased. The goal of this was to try to determine at which point the network became over-parametrised and would not generalise well. The performance of networks with two hidden layers were evaluated for each layer having from 10 to 30 neurons. The performance of the network generally increased as the number of neurons increased, however there were outliers, likely due to favourable or unfavourable initialisation of weights and biases. The results of this grid search are given below. The networks performance still improved as the number of neurons increased, and did not appear over-parametrised with 30 neurons in each layer. The final architecture that was used is given below

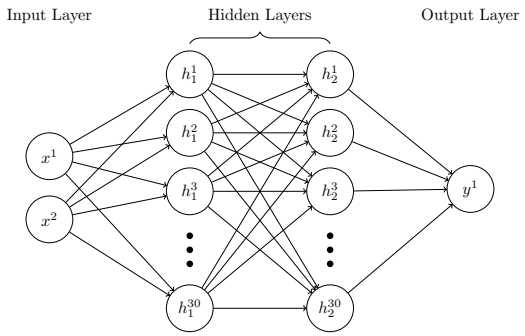

Figure 7: The final architecture for the 2D function fitting network determined through grid search.

Once the architecture was determined, the network was trained for 800 epochs with early stopping. If the MSE on the validation set did not improve for 10 epochs, training would stop. The approximation of the function given by the network is shown in Figure 8.

The error level curves for the network trained with standard Levenberg-Marquardt are given on the left of Figure 9. Looking at these error level curves, the training performance diverges massively from both the test and validation performance, implying that the network has overfitted to a large degree. This overfitting could be avoided by using Bayesian regularisation or dropout regularisation. The second graph in Figure 9 shows the improved performance of the network with Bayesian regularisation. Note also the shape of the error curves, with the test and validation error not diverging as much from the training error.

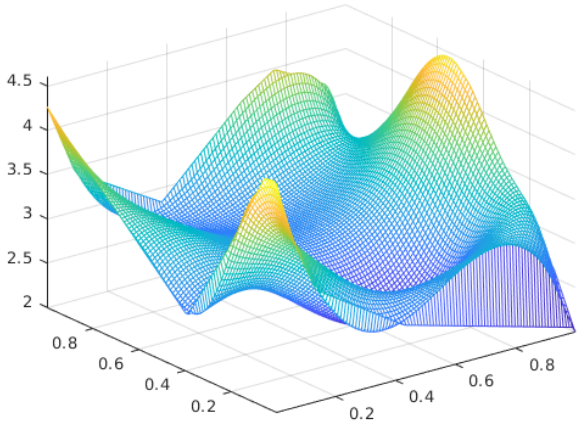

Figure 8: Approximation for the network.

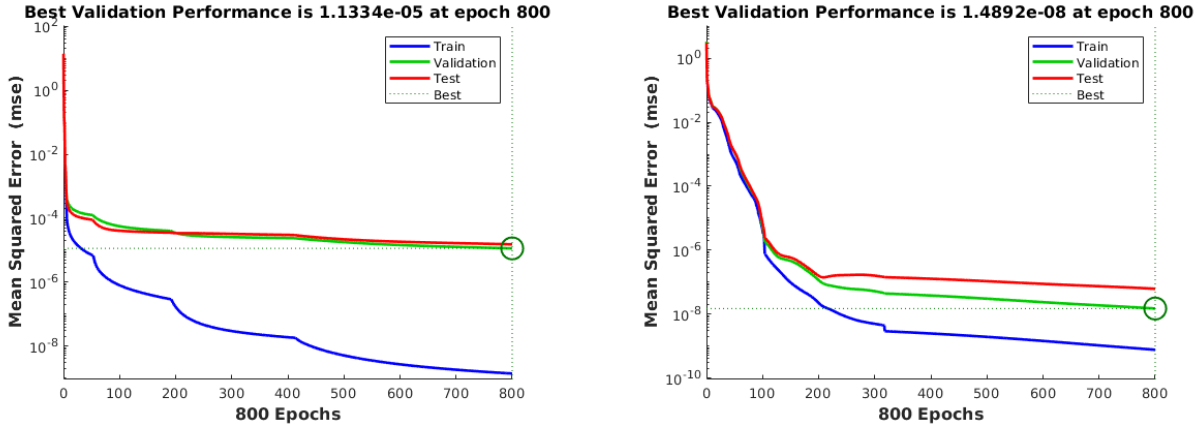

Figure 9: Performance of the network on the training, validation, and test sets. Without regularisation on the left, with regularisation on the right.

3 Bayesian Inference of Network Hyper-parameters

Here we return to the 1D function fitting exercise discussed in the first section. Generalisation can be improved by giving the network hyper-parameters a Bayesian treatment and effectively modifying the prior distribution over the weight-space and bias-space. Doing so can reduce overfitting in over-parametrised networks. To test this, the performance of standard Levenberg-Marquardt (`trainlm`) was compared with Levenberg-Marquardt with Bayesian regularisation (`trainbr`) for a network with a single hidden layer of 30 neurons, both for noisy and noiseless data. In theory, the learning algorithm with Bayesian regularisation should generalise better and reduce overfitting, especially in the case of over-parametrised networks.

Table 2 shows that Levenberg-Marquardt with Bayesian regularisation performs better on the test set than standard Levenberg-Marquardt. In particular, the results show that the network with Bayesian regularisation generalises much better with noisy data, although the effective number of parameters is higher when trained on noisy data. This clearly shows how Bayesian regularisation improves generalisation and performance by reducing overfitting. The results for the same experiment performed with a massively overparametrised network (400 neurons) are shown in Table 3.

In the case of the overparametrised network, the results show the same pattern as before but are just slightly worse. In the overparametrised network, standard Levenberg-Marquardt converges very rapidly.

	Test MSE	Regression	Convergence	Effective # of Parameters
`trainlm`	0.2263	0.7924	14	91
`trainbr`	0.0852	0.9655	100	54
`trainlm` (noisy)	0.3216	0.3216	15	91
`trainbr` (noisy)	0.0766	0.9585	89	65

Table 2: Average test MSE, regression, number of epochs until convergence, and effective number of parameters for Levenberg-Marquardt with and without Bayesian regularisation for both noisy and noiseless data. Maximum number of epochs is 100 and the number of parameters is 91.

	Test MSE	Regression	Convergence	Effective Number of Parameters
`trainlm`	0.1147	0.9679	6	301
`trainbr`	0.04	0.9872	54	110
`trainlm` (noisy)	0.1838	0.9392	4	301
`trainbr` (noisy)	0.1626	0.9563	80	151

Table 3