1 Introduction

Standard implementations of neural networks use only $\ell_2$-regularisation and thus the regularisation does not force coefficients to zero—does not prune the connections in the network. In the eight practice session, you learnt that regression with $\ell_1$-penalty term forces sparse solutions. The task of this project is to implement $\ell_1$-regularisation to neural network package \texttt{nnet}. Note that you do not have to shoehorn $\ell_1$-regularisation into back-propagation algorithm. Instead, you should use $\ell_1$-regularisation individually for each neuron.

1.1 Pruning Unnecessary Neurons

Since \texttt{nnet} package implements only one hidden layer, you can first train the neural network and then consider the linear regression problem $y \sim f_1 + \cdots + f_k + 1$ separately, where $f_1, \ldots, f_k$ are the outputs on hidden neurons. For that, you have to compute the outputs of hidden layer network for each input data. By solving this task with \texttt{lars} package you can enforce sparse solution. You can try different search strategies: \texttt{lasso}, \texttt{lars} and \texttt{stagewise}.

After you have identified the right number of nonzero components (you have to invent your own method for this) then you can prune all neurons with zero weights. As their outputs are not used they can be eliminated. Then retrain the network—but keep the weights and structure for the remaining nodes.

1.2 Pruning Unnecessary Inputs

Second option is to consider individual hidden layer networks. After the initial training the neuron has hopefully learnt a concept that is useful in predicting. However, the training algorithm does not assure that all inputs are necessary for computing the concept. Hence, we can use $\ell_1$-regularisation on the individual neuron level to remove all unnecessary inputs for this particular neuron. Let $f_i$ be the outputs of the trained hidden layer neuron before pruning and $a_i$ the aggregation values that are sent through sigmoid function to get $f_i$. Then we have to solve linear regression task $a_i \sim x_1 + \cdots + x_n + 1$ to find the coefficients of the hidden layer neuron. Again, you can force most coefficients to zero by applying $\ell_1$-regression.
Now determining how many inputs to keep is a more complex task, since a good approximation of \( a_i \) is not the main aim. You should measure how much does the overall prediction error increase because of pruning.

After pruning some inputs, you should retrain the network starting form the modified solution. Note that you can do training in two ways. First, you can retrain the network each time you prune some hidden layer neuron. Second, you can prune all hidden layer neurons and then retrain the network. Note that in both cases you can force zeroes by setting appropriate mask in the nnet call.

1.3 Alteration of Basic Optimisation Steps

I do not know whether you should start form neuron or from connection pruning so the right pruning order is up to you. However, I guess that you should do several normal training faces interleaved with pruning phases. Also, the stopping criterion for pruning should be evaluated on separate validation set that is different from training set. Whether you use hold-out, cross-validation or bootstrap design for this goal is up to you.

2 Performance Evaluation

2.1 Sanity Checks

You should first test your method on the standard one- and two-dimensional testbeds, we used in the sixth exercise session. It should get the correct number of neurons for one-dimensional cases and I would be really glad if it would discover the correct representation of 2D humps and tilted checker-board patterns.

2.2 Real-world Performance

Test your method also on some standard regression data set which contains many data points, compare with other methods and report results.

3 Extra Points for Packaging

You get extra points if you package the code into a package together with examples and appropriate documentation (help files).