Backpropation learning: Exercise 5

This exercise is based on C.M.Bishop: *Pattern Recognition and Machine Learning*, Chapter 5.

The objective of this exercise is to get a feel for the issues related to initialization, step size and stopping criteria for gradient descent learning of neural networks, and finally become familiar with basic aspects of second order optimization.

Print and comment on the figures produced by the software main5a.m, main5b.m, main5c.m as outlined below at the **Checkpoints**.

Neural network model

Let $\mathbf{y}(\mathbf{x})$ be a function of the vector $\mathbf{x}$, where $\mathbf{x} = (x_1, \ldots, x_d)^\top$. The functional form of $\mathbf{y}(\mathbf{x})$ is a neural network (NN) with one hidden layer as shown in Figure 1. We have a data-set, $\mathcal{D} = \{(\mathbf{x}^n, \mathbf{t}^n)\}$, $n = 1, \ldots, N$ of N corresponding values of inputs $\mathbf{x}^n$ and targets $\mathbf{t}^n$. We wish to use the training data to learn the mapping $\mathbf{t} = \mathbf{y}(\mathbf{x}) + \epsilon$, where ϵ is the residual which we usual interpret as noise.

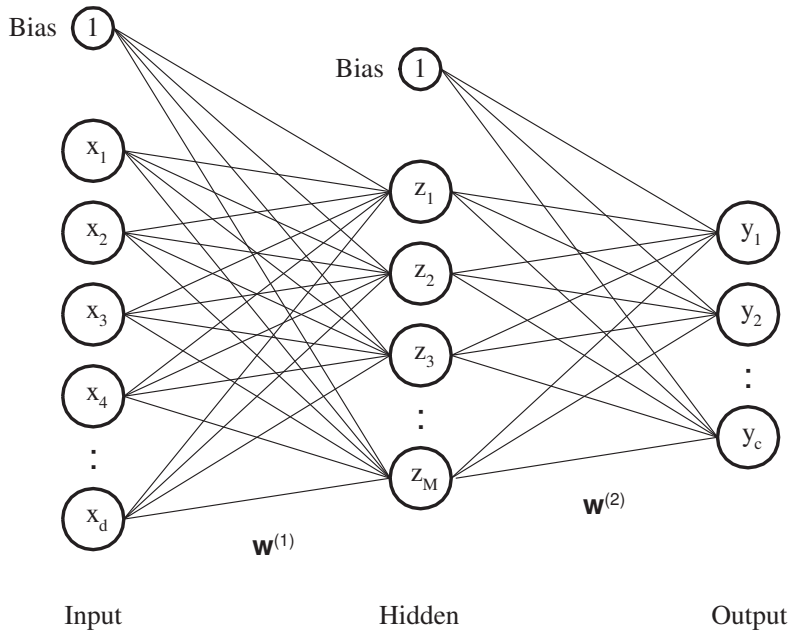

Figure 1: Neural network with one hidden layer.

The function is given by

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \quad , \quad z_0 \equiv 1 \tag{1}$$

$$z_j = g\left(\sum_{i=0}^{d} w_{ji}^{(1)} x_i \right) \quad , \quad x_0 \equiv 1 \ , \tag{2}$$

where y_k is the k'th output of the model, z_j is the the j'th output of hidden units and $g(\cdot)$ is a nonlinear sigmoid function. The parameters that we a going to estimate are usually called weights. They are given as $\mathbf{w}^{(1)}$ for the hidden layer and $\mathbf{w}^{(2)}$ for the output layer. It should be noted that a bias is included in the output and hidden layer by having z_0 and x_0 set to one. In matrix notation this can also be expressed as

$$\mathbf{y} = (\mathbf{w}^{(2)})^\top \mathbf{z} \ , \quad z_0 \equiv 1 \tag{3}$$

$$\mathbf{z} = g\left((\mathbf{w}^{(1)})^\top \mathbf{x}\right) \ , \quad x_0 \equiv 1 \ . \tag{4}$$

The optimal weights $\mathbf{w}$ are found by minimization of an error function. Here we shall use the sum-of-squares error function augmented by a weight-decay term

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}\{y_k(\mathbf{x}^n;\mathbf{w}) - t_k^n\}^2 + \frac{1}{2}\alpha|\mathbf{w}|^2 \ . \tag{5}$$

The weight decay term, parametrized by α, lowers the complexity of the model by shrinking the magnitude of the weights. Larger absolute weights makes the neural network mapping more non-linear and thus able to fit more different datasets including noise in the data. Adding the weight decay term can therefore help avoid overfitting.

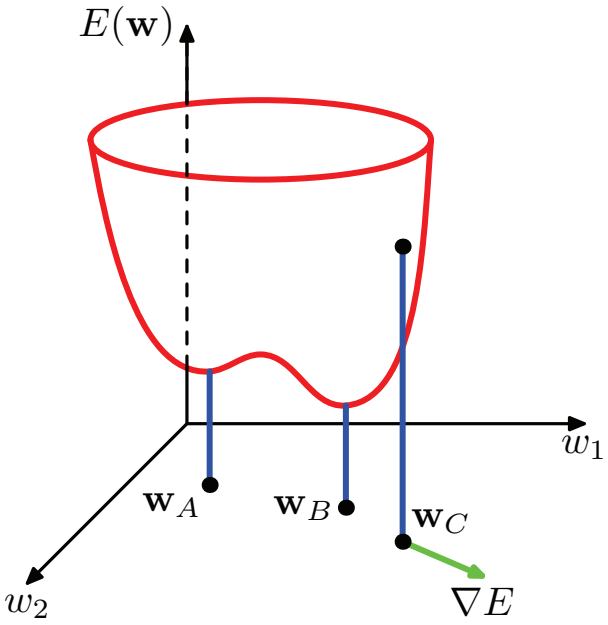

Figure 2: The weight update is done by gradient descent going the opposite direction of the cost functions gradient.

Finding the optimal weights minimizing Eq. (5) cannot be find analytically for the neural network model. An iterative scheme

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \triangle\mathbf{w} \tag{6}$$

is used instead. One choice for $\triangle\mathbf{w}$ is to move the weights in the opposite direction of the gradient of the error function Eq. (5):

$$\triangle\mathbf{w} = -\eta\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \ , \tag{7}$$

where η is the *step size* parameter controlling how long a step $\triangle\mathbf{w}$ should be. This method is called *gradient descent* and is illustrated in Figure 2. Differentiating (5) with respect to the weights in the output and hidden layer gives

$$\triangle w_{kj}^{(2)} = -\eta \left(\sum_{n=1}^{N} (y_{nk} - t_{nk}) z_{nj} + \alpha w_{kj}^{(2)} \right) \tag{8}$$

$$\triangle w_{ji}^{(1)} = -\eta \left(\sum_{n=1}^{N} \left((1 - z_{nj}^2) \sum_{k=1}^{c} w_{kj}^{(2)} (y_{nk} - t_{nk}) \right) x_{ni} + \alpha w_{ji}^{(1)} \right) \tag{9}$$

using that the nonlinear function $g(a) = \tanh(a)$ shown in Figure 3.

In the following sections we want to investigate some of the parameters involved in the optimization.

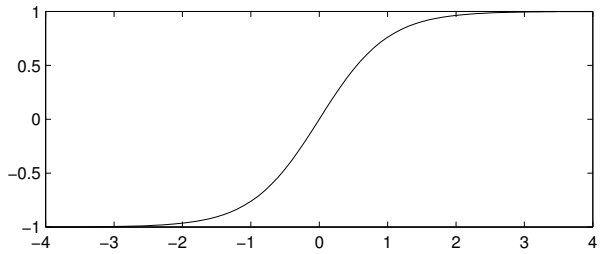

Figure 3: The nonlinear function $g(\cdot) = \tanh(\cdot)$.

Initialization range

The gradient descent process starts from an initial set of values for the weights. The role of the initialization range is important for the convergence of the parameters. If the range is making the initial weights too small compared to the signal, the gradient will be small and learning slow initially. On the other hand, setting the range to high forces the nonlinear function close to the step function and again the gradients become small and learning slow.

Checkpoint 5.1:

Use the program main5a.m to create an NN training-set with a 4-dimensional input variable, 5-dimensional hidden variable and a 1-dimensional output variable ($4 \times 5 \times 1$). Observe the effects of setting the range of the initial weights to: 0, 0.5 and 10000. Use Eqs. (8,9) and Figure 3 to explain the effects you see.

Step size parameter

The step size parameter η can be chosen in various ways. In this exercise we use a fixed value throughout the iteration scheme. This is not optimal, and we shall look at better schemes in the last Checkpoint in this exercise. When the step size is large the NN can 'overshoot'. To avoid this one might choose a small step size value, but this tends to make the NN converge more slowly.

Checkpoint 5.2:

Use the program main5a.m to create an NN training-set with a 4-dimensional input variable, 5-dimensional hidden variable and a 1-dimensional output variable ($4 \times 5 \times 1$). Set the step size to different values and observe the behavior. What is a good step size for the data-set?

Stopping criteria

Various methods can be used to stop the iteration process. The stopping criterion depends on the application. Here we will consider two possible candidates.

Checkpoint 5.3:

Use the program main5b.m to plot the test and training errors, the length of the gradient vector, and the training error difference between iterations. Comment on the plots and how they could be used as a stopping-criteria.

Improved optimization procedures

Non-linear optimization is a very active area of research and there exist many algorithms non of which are uniformly superior. The best choice depends upon the problem at hand. Although this topic is not a part of the curriculum, a bit of background knowledge about this topic is important to have in your toolbox. A subset of important algorithms are shown in Table 1. Most of the algorithms in Table 1 use an iterative scheme where the parameters $\mathbf{w}$ are initialized

1st	2nd	Name
−	−	Amoebe / simplex / Nelder-Mead
+	−	**Gradient descent**
+	−	Gradient descent with momentum
+	−	Natural gradient
+	−	Conjugate gradient algorithm
		— **Hestenes-Stiefel**
		— **Fletcher-Reeves**
		— **Polak-Ribiere**
		— Scaled conjugate gradient
+	−	Quasi-Newton
		— Davidson-Fletcher-Powell (DFP)
		— Rank-one-formula
		— Broyden-Fletcher-Goldfarb-Shanno (BFGS)
+	(+)	**Pseudo(-Gauss)-Newton**
+	(+)	Gauss-Newton
+	+	Levenberg-Marquardt
+	+	Newton(-Ralphson)

Table 1: Some optimization algorithms.

to some values and then a step is taken to a new place

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \ . \tag{10}$$

The usual *gradient descent* (used int eh previous Checkpoints) is the negative gradient multiplied by a suitable learning rate η:

$$\Delta\mathbf{w} = -\eta \ \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \ . \tag{11}$$

One can develop the second order algorithms (Newton, Levenberg-Marquardt, Gauss-Newton, pseudo-Gauss-Newton) from a Taylor expansion up to the second order term of the cost function E around $\hat{\mathbf{w}}$:

$$E(\mathbf{w}) = E(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})'\mathbf{g}_{\hat{\mathbf{w}}} + \frac{1}{2}(\mathbf{w} - \hat{\mathbf{w}})'\mathbf{H}_{\hat{\mathbf{w}}}(\mathbf{w} - \hat{\mathbf{w}}) + \dots \ . \tag{12}$$

$\mathbf{g}_{\hat{\mathbf{w}}}$ is the first order derivative (gradient) of the cost function in $\hat{\mathbf{w}}$ and $\mathbf{H}_{\hat{\mathbf{w}}}$ is the second order derivative—the Hessian—of the cost function in $\hat{\mathbf{w}}$. The first order derivative with respect to $\mathbf{w}$ is:

$$\nabla E(\mathbf{w}) = \mathbf{g}_{\hat{\mathbf{w}}} + \mathbf{H}_{\hat{\mathbf{w}}}(\mathbf{w} - \hat{\mathbf{w}}) \ . \tag{13}$$

We want to find a local minimum $\mathbf{w} = \mathbf{w}_0$. The gradient should be zero there: $\nabla E(\mathbf{w}_0) = 0$, which means that we can now isolate $\mathbf{w}_0$:

$$\mathbf{w}_0 = \hat{\mathbf{w}} + (-\mathbf{H}_{\hat{\mathbf{w}}}^{-1}\mathbf{g}_{\hat{\mathbf{w}}}) \ . \tag{14}$$

Taking this step is the (full) Newton algorithm. The other second order methods (and the quasi-Newton methods) use some kind of approximation to the Hessian, e.g., the *pseudo-Gauss-Newton* uses only the diagonal of the Hessian (here for each variable w_i in $\mathbf{w}$):

$$\Delta w_i = -\frac{\partial E}{\partial w_i} \Big/ \frac{\partial^2 E}{\partial w_i^2} \ . \tag{15}$$

Conjugate gradient algorithms construct a series of *conjugate* directions $\mathbf{d}$. These are direction that satisfy the following condition:

$$\mathbf{d}'_{j+1}\mathbf{H}\mathbf{d}_j = 0 \ . \tag{16}$$

There are three classic conjugate gradient algorithms, Hestenes-Stiefel, Fletcher-Reeves and Polak-Ribiere, which construct the series of conjugate directions using the following equations ($\mathbf{g}$ is the gradient):

$$\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \frac{\mathbf{g}'_{j+1}(\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{d}_j{}'(\mathbf{g}_{j+1} - \mathbf{g}_j)}\mathbf{d}_j \tag{17}$$

$$\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \frac{\mathbf{g}'_{j+1}\mathbf{g}_{j+1}}{\mathbf{g}'_j\mathbf{g}_j}\mathbf{d}_j \tag{18}$$

$$\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \frac{\mathbf{g}'_{j+1}(\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{g}_j{}'\mathbf{g}_j}\mathbf{d}_j \ . \tag{19}$$

The conjugate gradient algorithms usually require that the cost function is minimized along the conjugate direction, thus a *line search* is performed.

Optimization in the neural regression toolbox

The neural regression toolbox implements five different optimization algorithms presently: Gradient descent, pseudo-Gauss-Newton and three conjugate gradient algorithms: Hestenes-Stiefel (HS), Fletcher-Reeves (FR) and Polak-Ribiere (PR). The function `nr_trainx` implements them all and `nr_train` only implements the two first. The gradient descent algorithm chooses the (negative) gradient as its direction and the step size (Bishop: learning rate η) is determined by line search using a sort of bisection: The gradient decent starts with a step that is as long as the gradient (i.e., $\eta = 1$) and then halves it until the cost function is decreasing. The pseudo-Gauss-Newton starts with a length determined by equation 15. The line search for both of these algorithms is implemented in `nr_linesear`. The pseudo-Gauss-Newton presently starts with 10 gradient descent steps so that it (hopefully!) will get into a region where the second order derivative is well-behaved.

The conjugate gradient algorithms use another type of line search algorithm: quadratic (parabolic) and cubic interpolation and extrapolation. The line search is inexact and the stopping criterion for it is the so-called Wolfe-Powell condition.

This kind of line search is implemented with `nr_linesearch` (not the same as `nr_linesear`!).

The neural network is run on the sunspot data. There are 12 inputs plus a bias unit, 3 hidden units and one output.

Checkpoint 5.4

Use the program `main5c.m` to investigate the speed of convergence for the different algorithms. The program runs the algorithms a couple of times (each with a different seed for the initialization of the weights) and computes the average of the cost function value. The evolution of the cost functions is available in the variable `E`.

Determine which is the best algorithm and comment on the shapes of the curves.