Chapter 27

Neural networks for solving ODEs

Prerequisites: *Chapters 7, 8 18*

27.1 ▪ Introduction

The schematic diagram in Figure 27.1 depicts a neural network consisting of four input units, two hidden layers of three and four units each, and a single output unit. A general neural network may have any number of hidden layers, and the number of units within the hidden, input, and output layers may vary. The units are interconnected and influence each other in the directions of the arrows depicted in the figure.

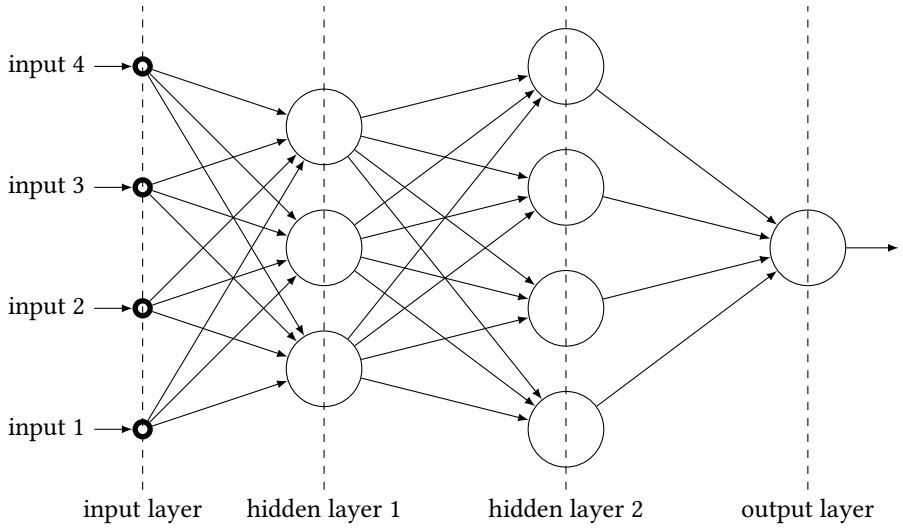

Figure 27.1: The schematic of a neural network of four inputs, two hidden layers, and one output.

The details of a single unit, let's call it the unit i, are shown in Figure 27.2. It receives inputs $x_1, x_2, \ldots, x_n$ from the units in the preceding layer and forms the weighted sum

$$z_i = u_i + \sum_{j=1}^{n} w_{ij} x_j,$$

where the weights w_{ij}, and the *bias* u_i are specific to this unit. Then it passes the sum through a function σ, called the *activation function*, and produces the output $\sigma(z_i)$. A typical choice for σ is the sigmoidal function[95]

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{27.1}$$

It is also typical to use the same activation function for all units, with the exception of the output unit whose activation function is often the identity function $\sigma(x) = x$.

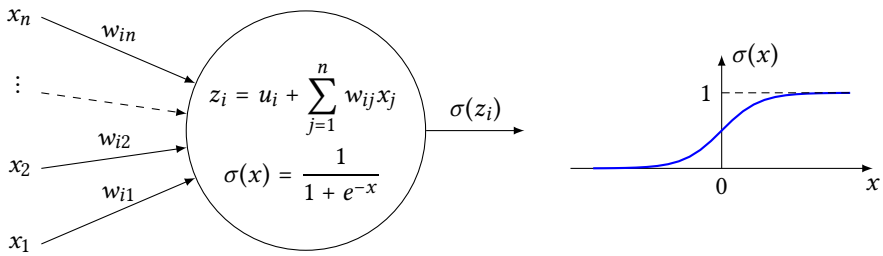

Figure 27.2: The ith neural network unit receives inputs $x_1, x_2, \ldots$ and produces output $\sigma(z_i)$. The activation function σ is typically the sigmoid (27.1).

27.2 ▪ A single-layer neural network

In these notes we focus on a simple neural network consisting of n inputs, one hidden layer of q units, and one output, as shown in Figure 27.3. The unit i in the hidden layer combines the inputs $x_1, x_2, \ldots, x_n$ from the input layer into $z_i = u_i + \sum_{j=1}^{n} w_{ij}x_j$, and produces the output $\sigma(z_i)$, where σ is as in (27.1). The output unit computes and outputs the weighted sum $N(\boldsymbol{x}) = \sum_{i=1}^{q} v_i\sigma(z_i)$. The activation function of the output unit is the identity $\sigma(x) = x$.

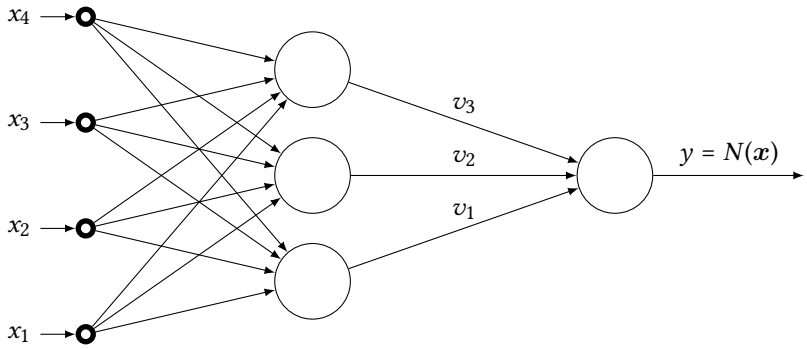

Figure 27.3: A simple neural network of $n = 4$ input units, a single hidden layer of $q = 3$ units, and one output unit.

As a whole, this neural network acts as a function $N : R^n \longrightarrow R$, where

$$N(\boldsymbol{x}) = \sum_{i=1}^{q} v_i\sigma(z_i) = \sum_{i=1}^{q} v_i\sigma\Big(u_i + \sum_{j=1}^{n} w_{ij}x_j\Big). \tag{27.2}$$

[95] This sigmoidal function acts as a cut-off—its output is zero when the input is small, and is one when the output is large.

In Section 27.3 we will see that solving an ODE through neural networks leads to an optimization problem involving $N(\boldsymbol{x})$ and its derivatives of up to order m with respect to $\boldsymbol{x}$, where m is the order of the ODE. Thus, we proceed to calculate those derivatives now. We have:

$$\frac{\partial z_i}{\partial x_k} = \frac{\partial}{\partial x_k}\Big(u_i + \sum_{j=1}^{n} w_{ij}x_j\Big) = \frac{\partial}{\partial x_k}\big(w_{i1}x_1 + w_{i1}x_2 + \cdots + w_{in}x_n\big) = w_{ik},$$

and therefore

$$\frac{\partial}{\partial x_k}\sigma(z_i) = \sigma'(z_i)\frac{\partial z_i}{\partial x_k} = w_{ik}\sigma'(z_i).$$

We conclude that

$$\frac{\partial}{\partial x_k}N(\boldsymbol{x}) = \frac{\partial}{\partial x_k}\sum_{i=1}^{q} v_i\sigma(z_i) = \sum_{i=1}^{q} v_i w_{ik}\sigma'(z_i).$$

Higher order derivatives of N may be computed in the same way. Here is the λ_kth derivative with respect to x_k:

$$\frac{\partial^{\lambda_k}}{\partial x_k^{\lambda_k}}N(\boldsymbol{x}) = \sum_{i=1}^{q} v_i w_{ik}^{\lambda_k}\sigma^{(k)}(z_i),$$

where $\sigma^{(k)}$ is the kth derivative of σ. This extends in the obvious way to mixed derivatives of any order. Thus, consider the *multi-index* $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$, where each λ_k is a nonnegative integer, and let's write $|\lambda| = \sum_{k=1}^{n}\lambda_k$. Then

$$\frac{\partial^{\lambda}N(\boldsymbol{x})}{\partial x_1^{\lambda_1}\partial x_2^{\lambda_2}\cdots\partial x_n^{\lambda_n}} = \sum_{i=1}^{q} v_i w_{ii}^{\lambda_i} w_{i2}^{\lambda_2}\cdots w_{in}^{\lambda_n}\sigma^{(|\lambda|)}(z_i),$$

which we may write in the compact notation

$$\nabla^{\lambda}N(\boldsymbol{x}) = \sum_{i=1}^{q} v_i P_{\Lambda,i}\sigma^{(|\lambda|)}(z_i), \tag{27.3}$$

where

$$P_{\lambda,i} = w_{ii}^{\lambda_i} w_{i2}^{\lambda_2}\cdots w_{in}^{\lambda_n} = \prod_{k=1}^{n} w_{ik}^{\lambda_k}. \tag{27.4}$$

The derivatives calculated in (27.3) are all that is needed for applying a gradient-free method (such as the Nelder–Mead algorithm) to solve the optimization problem that arises in solving ODEs. If, however, we wish to apply a gradient-based optimization algorithm (such as a conjugate gradient method), then we will also need the first derivatives of (27.3) with respect to the parameters u, v, and w.[96] For completeness, here we present the calculation of those derivatives, although in our implementation, which is based on the Nelder–Mead algorithm, these are not needed.

Derivatives with respect to u_k and v_k calculated in a straightforward way by applying the chain rule to (27.3). We have:

$$\frac{\partial}{\partial u_k}\nabla^{\lambda}N(\boldsymbol{x}) = v_k P_{\Lambda,k}\sigma^{(|\lambda|+1)}(z_i), \tag{27.5a}$$

$$\frac{\partial}{\partial v_k}\nabla^{\lambda}N(\boldsymbol{x}) = P_{\Lambda,k}\sigma^{(|\lambda|)}(z_i). \tag{27.5b}$$

[96] There are q components of u, q components of v, and qn components of w, and therefore a total of $2q + qn = (2 + n)q$ parameters.

Calculating the derivative with respect to w_{pq} is a little bit more tedious. Here we give the details.

From (27.4) we get:

$$\ln P_{\lambda,i} = \sum_{k=1}^{n} \lambda_k \ln w_{ik},$$

and therefore

$$\frac{1}{P_{\lambda,i}} \frac{\partial P_{\lambda,i}}{\partial w_{pq}} = \sum_{k=1}^{n} \lambda_k \frac{1}{w_{ik}} \frac{\partial w_{ik}}{\partial w_{pq}} = \sum_{k=1}^{n} \lambda_k \frac{1}{w_{ik}} \delta_{ip}\delta_{kq} = \lambda_q \frac{1}{w_{iq}} \delta_{ip},$$

and therefore

$$\frac{\partial P_{\lambda,i}}{\partial w_{pq}} = \lambda_q \frac{P_{\lambda,i}}{w_{iq}} \delta_{ip}.$$

Additionally, we have

$$\frac{\partial z_i}{\partial w_{pq}} = \frac{\partial}{\partial w_{pq}} \Big(u_i + \sum_{j=1}^{n} w_{ij} x_j \Big) = \sum_{j=1}^{n} \frac{\partial w_{ij}}{\partial w_{pq}} x_j = \sum_{j=1}^{n} \delta_{ip}\delta_{jq} x_j = \delta_{ip} x_q,$$

and therefore

$$\frac{\partial}{\partial w_{pq}} \sigma^{(|\lambda|)}(z_i) = \sigma^{(|\lambda|+1)}(z_i) \frac{\partial z_i}{\partial w_{pq}} = \sigma^{(|\lambda|+1)}(z_i) \delta_{ip} x_q.$$

Now we are ready to calculate the derivative of (27.3):

$$\begin{aligned}
\frac{\partial}{\partial w_{pq}} \nabla^{\lambda} N(\boldsymbol{x}) &= \sum_{i=1}^{q} v_i \frac{\partial P_{\lambda,i}}{\partial w_{pq}} \sigma^{(|\lambda|)}(z_i) + \sum_{i=1}^{q} v_i P_{\lambda,i} \frac{\partial}{\partial w_{pq}} \sigma^{(|\lambda|)}(z_i) \\
&= \sum_{i=1}^{q} v_i \lambda_q \frac{P_{\lambda,i}}{w_{iq}} \delta_{ip} \sigma^{(|\lambda|)}(z_i) + \sum_{i=1}^{q} v_i P_{\lambda,i} \sigma^{(|\lambda|+1)}(z_i) \delta_{ip} x_q. \\
&= v_p \lambda_q \frac{P_{\lambda,p}}{w_{pq}} \sigma^{(|\lambda|)}(z_p) + v_p P_{\lambda,p} \sigma^{(|\lambda|+1)}(z_p) x_q.
\end{aligned}$$

For cosmetic reasons, we replace the pq indices with ij:

$$\frac{\partial}{\partial w_{ij}} \nabla^{\lambda} N(\boldsymbol{x}) = v_i P_{\lambda,i} \sigma^{(|\lambda|+1)}(z_i) x_j + v_i \lambda_j \frac{P_{\lambda,i}}{w_{ij}} \sigma^{(|\lambda|)}(z_i). \tag{27.5c}$$

Note that $\frac{P_{\lambda,i}}{w_{ij}}$ that appears on the right-hand side is not actually a fraction since $P_{\lambda,i}$ has a factor of $w_{ij}^{\lambda_j}$ which cancels the denominator.

Equations (27.5a), (27.5b), (27.5c), along with (27.3), supply all the derivatives that are needed in applying a gradient-based optimization algorithm to solve ODEs.

27.3 ▪ Solving ODEs through neural networks

Consider the boundary value problem for 2nd order ODE

$$F\Big(x, u(x), u'(x), u''(x)\Big) = 0, \quad a < x < b, \tag{27.6a}$$

$$u(a) = u(b) = 0. \tag{27.6b}$$

We seek a solution of the form $u(x) = \phi(x)N(x)$, where $N(x)$ is the transfer function of a suitably tuned neural network, and $\phi(x)$ is a function which we pick, a priori, to enforce the boundary conditions. The function $\phi(x) = (b - x)(x - a)$ is an obvious choice since it is positive in the interval (a, b) and vanishes at the boundary points. Our main task is to design a neural network whose transfer function $N(x)$ is so that $u(x) = \phi(x)N(x)$ is a good approximation to the solution of the boundary value problem (27.6).

Our neural network will have only a single input, x and a single output, $N(x)$. There is quite a bit of flexibility in designing the hidden layers. In this introductory exposition, we choose to have only one hidden layer consisting of q units, and we will experiment by varying q. A schematic of the $q = 3$ case is shown in Figure 27.4.

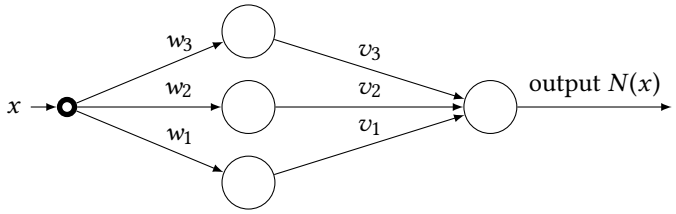

Figure 27.4: A neural network of one hidden layer consisting of $q = 3$ units, for solving the boundary value problem (27.6).

Ideally, the function $u(x) = \phi(x)N(x)$ will satisfy the boundary value problem (27.6) exactly, that is, the *residual at x*

$$R(x) = F\Big(x,\ \phi(x)N(x),\ \big(\phi(x)N(x)\big)',\ \big(\phi(x)N(x)\big)''\Big) \tag{27.7}$$

will be zero for all x. Our neural network produces only an approximation to the solution, and therefore $R(x)$ will be small but not necessarily zero.

We will shortly need the expanded form of (27.7), so we might as well do it now:

$$R(x) = F\Big(x, \phi(x)N(x), \phi'(x)N(x) + \phi(x)N'(x), \\ \phi''(x)N(x) + 2\phi'(x)N'(x) + \phi(x)N''(x)\Big). \tag{27.8}$$

For the purpose of "training" the network, we pick ν equally spaced points with coordinates $x_k = a + \frac{b-a}{\nu+1}k$, $k = 1, 2, \ldots, \nu$, in the interval (a, b). We evaluate the residual (27.8) at those points and calculate the sum of the squares:

$$\begin{aligned} E &= \sum_{i=1}^{\nu} R(x_i)^2 \\ &= \sum_{i=1}^{\nu} \Big[F\Big(x_i, \phi(x_i)N(x_i), \phi'(x_i)N(x_i) + \phi(x_i)N'(x_i), \\ &\qquad \phi''(x_i)N(x_i) + 2\phi'(x_i)N'(x_i) + \phi(x_i)N''(x_i)\Big)\Big]^2. \end{aligned} \tag{27.9}$$

The *residual error* E is immediately computable since ϕ is given, and we have explicit expressions for N and its derivatives in (27.2) and (27.3). These, of course, depend on the neural network's parameters u, v, and w. We obtain the best choice of the parameters by minimizing E with respect to those parameters. A gradient-based minimization algorithm

requires the derivatives of E with respect to those parameters. These may be calculated with the aid of the equations (27.5a)–(27.5c). In our implementation we apply the Nelder–Mead minimization algorithm which is gradient-free, and therefore we have no use for equations (27.5a)–(27.5c), but we still need the differentiation formula (27.3) since (27.9) involves derivatives of N with respect to x.

Finally, let us note that the ideas outlined above generalize to higher order ODEs. We limit our implementation to second order ODEs for the sake of the exposition's transparency.

27.4 ▪ An overview of the program

Our implementation of the neural networks for solving ODEs is in the file *neural-net-ode.c*. The header file *neural-net-ode.h* provides the application's interface.

Additionally, the program relies on the *xmalloc* module of Chapter 7) to allocate memory, the *Nelder Mead* module of Chapter 18 to minimize the objective function, and the *array.h* header file of Chapter 8 to construct vectors and matrices. Therefore, following the recommendations of Chapters 2 and 6, the program's directory will look like this:

```
$ cd neural-nets
$ ls -F
Makefile     nelder-mead.c@     neural-nets-ode.h   xmalloc.c@
array.h@     nelder-mead.h@     plot-with-maple.c   xmalloc.h@
demo-ode1.c  neural-nets-ode.c  plot-with-matlab.c
```

The primary task in *neural-net-ode.c* is to provide the infrastructure to encode the residual error function E (equation (27.9)) for the generic second order boundary value problem (27.6). The file *demo-ode1.c* defines a concrete instance of (27.6)—see Section 27.7 for the details—and then calls the *Nelder Mead* module to minimize E by tuning the neural network's parameters. With the network thus trained, the solution $u(x)$ of the boundary value problem may be evaluated at any desired point x.

The files *plot-with-maple.c* defines a function which applies the trained neural network to produce a sequence of pairs $\big(x_i, u(x_i)\big)$, $i = 0, 1, \ldots, n$, and writes the result in the form of a Maple script, which when loaded into Maple, produces a graph of the solution. The files *plot-with-matlab.c* does the same, but writes a script suitable for loading into Matlab. Download these files from the book's website.

Here is the transcript of a session on executing the compiled *demo-ode1*:

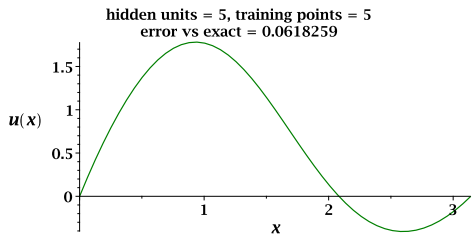

Figure 27.5: The graph of the solution $u(x)$ of the boundary value problem (27.10), as rendered in MAPLE.

```
$ ./demo-ode1
Usage:
    ./demo-ode1 q nu
    q   : number of units in the hidden layer (q ≥1)
    nu  : number of training points (nu ≥1)
```

We see that when *demo-ode1* is invoked without additional arguments, it prints a help message and exits. The message indicates the need to specify two arguments. The first argument is the number q of units in the hidden layer. The second argument is the number ν of training points. These are distributed uniformly as $x_k = a + \frac{b-a}{\nu+1}k$, $k = 1, 2, \ldots, \nu$ in the interval (a, b).

When *demo-ode1* is supplied with the requisite arguments, here is what we see:

```
$ ./demo-ode1 5 4
weights before training:
  0.340  -0.106   0.283   0.298   0.412  -0.302  -0.165   0.268
-0.222   0.054  -0.023   0.129  -0.135   0.013   0.452
Nelder-Mead: Converged after 1097 function evaluations
Nelder-Mead: Neural network's residual error = 3.56089e-13
weights after training:
  0.280  -0.369   2.554   0.786   0.422  -0.682  -0.856   2.103
0.022  -0.256   0.409  -1.578  -1.636   0.308  -0.288
Error versus the ODE's exact solution = 0.0842691
```

Here we see the results of solving the boundary value problem with five units in the hidden layer, and four training points. The Nelder–Mead algorithm minimizes the residual error E after 1097 function evaluation. The minimum value of E is of the order 10^{-13}, which is quite good. The discrepancy between the calculated and exact solutions is small, and can be made smaller by increasing the number of training points.

Not visible in that transcript are the two script files, generated silently, for plotting the solution in MAPLE and MATLAB. The scripts are written to files whose names are specified by the user in *demo-ode1.c*. Figure 27.5 shows one such graph, plotted in MAPLE.

27.5 ▪ The interface

The contents of the header file *neural-nets-ode.h* is shown in Listing 27.1. It declares a structure **`struct`** `Neural_Net_ODE` that holds the necessary data for defining a boundary value problem for an ODE and a neural network to solve it. Let us begin by examining the structure's details. Line numbers refer to those in Listing 27.1.

Line 5: The member `ODE` of the **`struct`** `Neural_Net_ODE` points to a user-defined

Listing 27.1: The header file *neural-nets-ode.h*.

```
1  #ifndef H_NEURAL_NET_ODE_H
2  #define H_NEURAL_NET_ODE_H
3
4  struct Neural_Net_ODE {
5      double (*ODE)(double x, double u, double u_x, double u_xx);
6      double a;                          // left endpoint of the interval
7      double b;                          // right endpoint of the interval
8      int q;                             // number of units in the hidden layer
9      int nu;                            // the number of training points
10     double *training_points;           // the array of training points
11     double (*exact_sol)(double x);     // NULL if no exact solution available
12
13     // no user modifiable parts beyond this point
14     int nweights;                      // 3 × q
15     double *weights;                   // the array of u, v, w
16     double sigma[3];                   // array to hold σ, σ′, σ″
17     double phi[3];                     // array to hold φ, φ′, φ″
18     double N[3];                       // array to hold N, N′, N″
19 };
20
21 void Neural_Net_init(struct Neural_Net_ODE *nn);
22 void Neural_Net_end(struct Neural_Net_ODE *nn);
23 void Neural_Net_eval(struct Neural_Net_ODE *nn, double x);
24 void Neural_Net_phi(struct Neural_Net_ODE *nn, double x);
25 void Neural_Net_plot_with_maple(struct Neural_Net_ODE *nn, int n,
26         char *outfile);
27 void Neural_Net_plot_with_matlab(struct Neural_Net_ODE *nn, int n,
28         char *outfile);
29 double Neural_Net_residual(double *weights, int nweights,
30         void *params);
31 double Neural_Net_error_vs_exact(struct Neural_Net_ODE *nn, int n);
32
33 #endif /* H_NEURAL_NET_ODE_H */
```

function that defines the differential equation to be solved, which is, in effect, the function F in (27.6a). See the description of the file *demo-ode1.c* for an example.

Lines 6 and 7: These define the interval (a, b) of over which we solve the ODE.

Line 8: q is the number of units in the hidden layer.

Line 9: ν is the number of training points.

Line 10: Pointer to an array of the x coordinates of the training points.

Line 11: Pointer to a function that returns the exact solution to the problem. This is used to test the program's correctness. When no exact solution is available, we set this pointer to `NULL`.

Line 14: As seen in Section 27.3, our neural network is characterized by $3q$ parameters (weights) u_i, v_i, w_i, $i = 1, \ldots, q$, where q is the number units in the hidden layer.

The value of `weights` is set to $3q$ in the function `Neural_Net_init()`. It is not intended to be set by the user.

Line 15: `weights` points to an array of length $3q$ which holds the $3q$ parameters u, v and w, in that order. Thus, for $i = 0, 1, \ldots, q-1$, we have u_i = `weights[i]`, v_i = `weights[q+i]`, w_i = `weights[2*q+i]`.

Lines 16–18: The array `sigma` will hold the values of $\sigma(x)$, $\sigma'(x)$, $\sigma''(x)$ at varying values of x. Similarly, `phi`, and `N` will hold the values of $\phi(x)$ and $N(x)$, and their first and second derivatives.

Lines 21 to end: The function declared here are the public functions exported by our module. The are described in the next section.

27.6 ▪ The implementation

Listing 27.2 presents an outline of the implementation file *neural-nets-ode.c*. It contains a few private functions (marked **`static`**) and several public function which appear in *neural-nets-ode.h* in Listing 27.1. I will describe the purposes of the various functions in the following subsections.

27.6.1 ▪ The function `sigmoid()`

The function `sigmoid()` receives a value x and evaluates the sigmoidal function $\sigma(x)$ of (27.1) and its derivatives $\sigma'(x)$ and $\sigma''(x)$, and stores them in `sigma[0]`, `sigma[1]`, `sigma[2]`, respectively.

27.6.2 ▪ The function `Neural_Net_phi()`

The function `Neural_Net_phi()` receives a value x and evaluates $\phi(x) = (b-x)(x-a)$ and its derivatives $\phi'(x)$ and $\phi''(x)$, and stores them in `phi[0]`, `phi[1]`, `phi[2]`, respectively. The values of a and b are available in $nn->a$ and $nn->b$.

27.6.3 ▪ The function `Neural_Net_init()`

This function called to initialize the module prior to calling any the other functions. Specifically, this sets value of the `nweights` field to $3q$, where q is the number of units in the hidden layer, and then calls `make_vector` to allocate an array of length `nweights` which will hold the neural networks $3q$ parameters u, v, and w.

Each of the $3q$ parameters are assigned random values in the range −0.5 to 0.5. The "training" of the neural network amounts to adjusting those values to minimize the residual error defined in (27.9).[97] For the details of the Standard Library function `rand()`, see Chapter 10.

27.6.4 ▪ The function `Neural_Net_end()`

This function is called at the conclusion of the use of the module to clean up any odds and ends. In the current implementation, this performs only one task—it frees the memory that was allocated on line 13 in 27.2.

[97] Admittedly, the range −0.5 to 0.5 is selected rather arbitrarily. It may be worthwhile to investigate the effect of changing that range on the solver's performance.

Listing 27.2: An outline of the file *neural-nets-ode.c*. Flesh out the parts marked with ▸.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "neural-nets-ode.h"
#include "nelder-mead.h"
#include "array.h"

▸ static void sigmoid(double x, double *sigma) ...
▸ void Neural_Net_phi(struct Neural_Net_ODE *nn, double x) ...
void Neural_Net_init(struct Neural_Net_ODE *nn)
{
    nn→nweights = 3*nn→q;
    make_vector(nn→weights, nn→nweights);
    for (int i = 0; i < nn→nweights; i++)
        nn→weights[i] = (double)rand()/RAND_MAX - 0.5;
}
▸ void Neural_Net_end(struct Neural_Net_ODE *nn) ...
void Neural_Net_eval(struct Neural_Net_ODE *nn, double x)
{
    int q = nn→q;
    double *u = nn→weights;
    double *v = nn→weights + q;
    double *w = nn→weights + 2*q;

    for (int j = 0; j ≤ 2; j++)
        nn→N[j] = 0.0;

    for (int i = 0; i < q; i++) {
        double z = u[i] + w[i]*x;
        sigmoid(z, nn→sigma);
        for (int j = 0; j ≤ 2; j++)
            nn→N[j] += v[i]*pow(w[i],j) * nn→sigma[j];
    }
}
▸ static double residual_at_x(struct Neural_Net_ODE *nn, double x) ...
double Neural_Net_residual(double *weights, int nweights, void *params)
{
    struct Neural_Net_ODE *nn = params;
    int nu = nn→nu;          // the number of training points
    double sum = 0.0;

    for (int i = 0; i < nn→nweights; i++)
        nn→weights[i] = weights[i];

    for (int i = 0; i < nu; i++) {
        double x = nn→training_points[i];
        double r = residual_at_x(nn, x);
        sum += r*r;
    }

    return sum;
}
▸ double Neural_Net_error_vs_exact(struct Neural_Net_ODE *nn, int n)
```

27.6.5 ▪ The function `Neural_Net_eval()`

This function evaluate the output $N(x)$ of the neural network corresponding to the input x, and the derivatives $N'(x)$ and $N''(x)$, according to the formula (27.3). These are stored in `N[0]`, `N[1]`, and `N[2]` slots in the array `N` attached to the structure `nn`.

27.6.6 ▪ The function `residual_at_x()`

This function calculates and returns the residual $R(x)$ at a given x, according to (27.8). The values of ϕ and its derivatives may be obtained by calling `Neural_Net_phi()`. The values of N and its derivatives may be obtained by calling `Neural_Net_eval()`. The function F is available as `nn→ODE`.

27.6.7 ▪ The function `Neural_Net_residual()`

The implementation of this function, shown in full in Listing 27.2, evaluates the residual error E according to (27.9). The function's prototype is set to match exactly that which is required of objective functions in the *Nelder Mead* module. Specifically, since we wish to minimize E as a function of the $3q$ parameters u, v, and w which define the neural network, the first argument of `Neural_Net_residual()` is an array of length $3q$ that holds the values of u, v, and w. The second argument, `nweights`, is set to $3q$, indicating that the minimization takes places over a $3q$-dimensional space. From *Nelder Mead*'s point of view, the argument `weights` represents the coordinates of a vertex of the $3q$-dimensional simplex.

As the Nelder–Mead algorithms performs its downhill march, it repeatedly revises the coordinates in `weights`, and calls `Neural_Net_residual()` to re-evaluate the objective function. In line 43 in Listing 27.2 we copy the trial weights produced by *Nelder Mead* into `nn→weights` in order to keep the weights in **`struct`** `Neural_Net_ODE` in sync with those requested by *Nelder Mead*. This is absolutely necessary, since in line 47 we are calling `residual_at_x()` which in turns calls `Neural_Net_eval()` which needs the current values of the weights in order to correctly calculate the output $N(x)$.

27.6.8 ▪ The function `Neural_Net_error_vs_exact()`

The purpose of this function is to evaluate the now trained neural network at $n+1$ equally spaced points $x_0 = a, x_1, x_2, \ldots, x_n = b$ over the interval $[a, b]$ and calculate and return the largest discrepancy

$$\max_{0 \le i \le n} \left| \phi(x_i) N(x_i) - u_{\text{exact}}(x_i) \right|$$

between the solution produced by our neural network and an exact or target solution u_{exact} supplied by the user. This is useful in testing the accuracy and performance of the implementation, and naturally it is applicable only when such a target solution is available.

27.7 ▪ The file *demo-ode1.c*

The file *demo-ode1.c* provides a demonstration of this module. It solves the boundary value problem

$$u'' + \frac{1}{1+u^2} = f, \quad u(0) = u(\pi) = 0 \tag{27.10a}$$

Listing 27.3: A sketch of the file *demo-ode1.c*. Flesh out the parts marked with ▸.

```
1  ▸ the necessary headers here
2  #define PI  4.0*atan(1.0)
3  ▸ static double exact_sol(double x) ...
4  ▸ static double my_ode(double x, double u, double u_x, double u_xx) ...
5  ▸ static void show_usage(char *progname) ...
6  ▸ int main(int argc, char **argv) ...
```

for the unknown $u(x)$ on the interval $0 < x < \pi$, where $f(x)$ is selected as

$$f(x) = -\sin x - 4\sin 2x + \frac{1}{1 + \big(\sin x + \sin 2x\big)^2}. \tag{27.10b}$$

so that the exact solution of the problem is $u(x) = \sin x + \sin 2x$. Note that the boundary value problem in (27.10) is a special case of (27.6) with

$$F\Big(x, u, u', u''\Big) = u'' + \frac{1}{1 + u^2} - f(x), \tag{27.11}$$

and $a = 0$, $b = \pi$.

Listing 27.3 provides an outline of the file *demo-ode1.c*. Here are a few comments on that listing.

Line 2: We set the preprocessor symbol `PI` to take on the value of the mathematical π which we need since the boundary value problem 27.10 is defined on the interval $(0, \pi)$.[98]

Line 3: The function `exact_sol()` evaluates to the exact solution of the boundary value problem, which is $u_{\text{exact}}(x) = \sin x + \sin 2x$ in this case. The name of the function is immaterial; it may be named anything. If you don't have access to an exact solution, then you don't need to define this function at all.

Line 4: The function `my_ode()` implements the function F of equation (27.6a) for the ODE at hand. The F of interest in the current code is that in (27.11). Here is one way of doing it:

```
static double my_ode(double x, double u, double u_x, double u_xx)
{
    double t = sin(x) + sin(2*x);
    double f = -sin(x) - 4*sin(2*x) + 1 / (1 + t*t);
    return u_xx + 1 / (1 + u*u) - f;
}
```

Line 5: The function `show_usage()` is responsible for the usage message in the transcript of the interactive session shown on page 381. It receives the name of the executable file (available in `argv[0]` in `main()`) and prints the message shown.

Line 6: The function `main()` is somewhat long and deserves a listing of its own. Its details are presented in the following subsection.

[98] In modern compilers, such as Gnu C, that macro is evaluated at the preprocessing stage and all occurrences of `PI` are replaced with its numerical value before the compilation begins. Therefore the use of that macro incurs no cost at execution time.

Listing 27.4: The function `main()` in the file *demo-ode1.c* – part 1.

```
1  int main(int argc, char **argv)
2  {
3      double a = 0;         // left end of the interval
4      double b = PI;        // right end of the interval
5      double *training_points;
6      char *endptr;
7
8      if (argc ≠ 3) {
9          show_usage(argv[0]);
10         return EXIT_FAILURE;
11     }
12
13     // number of units in the hidden layer
14     int q = strtol(argv[1], &endptr, 10);
15     if (*endptr ≠ '\0' || q < 1) {
16         show_usage(argv[0]);
17         return EXIT_FAILURE;
18     }
19
20     // number of training points
21     int nu = strtol(argv[2], &endptr, 10);
22     if (*endptr ≠ '\0' || nu < 1) {
23         show_usage(argv[0]);
24         return EXIT_FAILURE;
25     }
26
27     // nu equally spaced points inside (a,b)
28     make_vector(training_points, nu);
29     for (int i = 0; i < nu; i++)
30         training_points[i] = a + (b - a) / (nu + 1) * (i + 1);
31
32     struct Neural_Net_ODE nn = {
33         .a                  = a,
34         .b                  = b,
35         .q                  = q,
36         .nu                 = nu,
37         .training_points    = training_points,
38         .ODE                = my_ode,
39         .exact_sol          = exact_sol,
40     };
```

27.7.1 ▪ The details of the function `main()`

Listing 27.4 (continued into Listing 27.5) presents an expanded view of the function `main()` that was noted in the Listing 27.3. I won't comments on the statements in Listing 27.4 since they are quite self-explanatory other than noting that the Standard Library's `strtol()` function is the subject of Chapter 5. What follows are some comments on Listing 27.5.

Line 41: Here we initialize the neural network through calling the function `Neural_Net_init()` which is shown in its entirety in Listing 27.2. Among other things, this initializes the neural network's weights to a set of random values.

Lines 43–52: We set up a structure to pass to the *Nelder Mead* module in order to train our neural network, that is, to determine the parameters u, v, and w that minimize the network's residual error. The values of `h`, `tol`, and `maxevals` are hard-coded here. In a fancier version of the program we may arrange to read those values from the command-line.

Lines 54 and 70: We print the neural network's weights before and after training for the curiosity's sake.

Lines 57–68: We call *Nelder Mead* to train the network, and then print the results to the *stdout*.

Line 73: If an exact solution is provided, we print the maximum discrepancy between the calculated and exact solutions. The argument 50 requests the comparisons to be performed on 51 equally spaced points in the ODE's domain. Adjust as desired.

Lines 76 to end: We produce MAPLE and MATLAB script files for subsequent plotting of the graphs of the solution. The argument 50 requests a graph corresponding to 51 equally spaced points on the ODE's domain. Adjust as needed. Here the file names are specified through their full paths. If you omit the full path specification, the files will be placed in the current directory.

The filename extension `.m` is required in MATLAB. MAPLE does not require a specific filename extension[99] but the extension `.mpl` is recommended.

27.8 ▪ *Project Neural networks: ODEs*

Part 27.1.

Solve the nonlinear boundary value problem for the unknown $u(x)$ on the interval $(0, 1)$:

$$u''(x) + u(x)^3 = f(x), \quad u(0) = 0, \quad u(1) = 0, \tag{27.12}$$

where $f(x) = \frac{352}{9} - \frac{256}{3}\,x + \left(\frac{16}{9}\,x(1-x)(8x-3)\right)^3$. The odd-looking expression for $f(x)$ is reverse-engineered so that the equation has the simple exact solution $u(x) = \frac{16}{9}\,x(1-x)(8x-3)$ shown in Figure 27.6. The solution achieves its maximum value of 1 at $x = 3/4$.

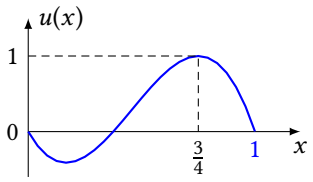

Figure 27.6: The graph of $u(x) = 16/9\,x(1-x)(8x-3)$.

Part 27.2. [optional] The requirement of zero boundary conditions in (27.6b) is too restrictive. Generalizing it to

$$u(a) = \alpha, \quad u(b) = \beta$$

[99] But don't use a filename extension `.m` since that has a special meaning to MAPLE.

requires only minor changes to our program. Instead of looking for solutions of the form $u(x) = \phi(x)N(x)$ as we did in Section 27.3, we look for solutions of the form

$$u(x) = \frac{\beta}{b-a}(x-a) + \frac{\alpha}{b-a}(b-x) + \phi(x)N(x),$$

where $\phi(x) = (b-x)(x-a)$ is as before.

Part 27.3. [optional] Extend Part 27.2 to allow for derivatives in the boundary conditions, as in:

$$u(a) = \alpha, \quad u'(b) = \beta.$$

Part 27.4. [optional] Extend this chapter's module to allow for any number of hidden layers, where the ith hidden layer consists of q_i units.

Listing 27.5: The function `main()` in the file *demo-ode1.c* – part 2.

```
41      Neural_Net_init(&nn);          // initialize the neural network
42
43      struct nelder_mead NM = {
44          .f          = Neural_Net_residual,
45          .n          = nn.nweights,
46          .s          = NULL,
47          .x          = nn.weights,
48          .h          = 0.1,
49          .tol        = 1e-5,
50          .maxevals   = 100000,
51          .params     = &nn,
52      };
53
54      printf("weights before training:\n");
55      print_vector("%7.3f ", nn.weights, nn.nweights);
56
57      int evalcount = nelder_mead(&NM);         // training: minimize the residual
58
59      if (evalcount > NM.maxevals) {
60          printf("Nelder-Mead: No convergence after %d "
61                  "function evaluation\n", evalcount);
62          return EXIT_FAILURE;
63      } else {
64          printf("Nelder-Mead: Converged after %d "
65                  "function evaluations\n", evalcount);
66          printf("Nelder-Mead: Neural network's residual error = %g\n",
67                  NM.minval);
68      }
69
70      printf("weights after training:\n");
71      print_vector("%7.3f ", nn.weights, nn.nweights);
72
73      if (nn.exact_sol ≠ NULL)
74          printf("Error versus the ODE's exact solution = %g\n",
75                  Neural_Net_error_vs_exact(&nn, 50));
76
77      Neural_Net_plot_with_maple(&nn, 50, "/tmp/zz.mpl");
78      Neural_Net_plot_with_matlab(&nn, 50, "/tmp/zz.m");
79
80      Neural_Net_end(&nn);          // end neural network
81
82      free_vector(training_points);
83
84      return EXIT_SUCCESS;
85  }
```