Chapter 28

Neural networks for solving PDEs

Prerequisites: *Chapters 7, 8 18 27*

28.1 ▪ Introduction

In Chapter 27 we implemented a neural networks methodology for solving boundary value problem (27.6) for general second order ordinary differential equations (ODEs) in one unknown. The purpose of this chapter is to produce a similar solver for boundary value problems for general second order partial differential equations (PDEs) general second order in in two independent variables and one unknown. Writing $u(x, y)$ for the unknown, and u_x for the derivative $\frac{\partial}{\partial x}u(x, y)$, etc., boundary value problem is expressed as

$$F\Big(x, y, u, u_x, u_y, u_{xx}, u_{xy}, u_{yy}\Big) = 0 \qquad x \in \Omega \subset R^2, \tag{28.1a}$$

$$u(x, y) = 0, \qquad (x, y) \in \partial\Omega, \tag{28.1b}$$

where Ω is a domain in R^2, and $\partial\Omega$ is the domain's boundary. As in the case of ODEs, we choose a neural network with a single hidden layer of q units as depicted in Figure 28.1. The inputs marked x_1 and x_2 on the diagram correspond to the independent variables x and y in the formulation (28.1).

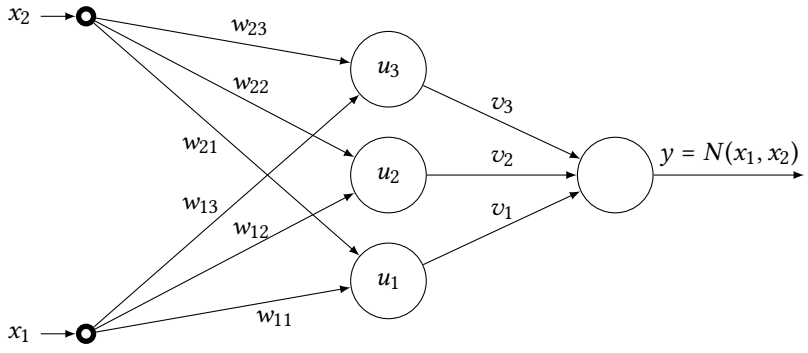

Figure 28.1: A neural network with a single hidden layer of $q = 3$ units, for solving the boundary value problem (28.1).

The units within the hidden layer operate as depicted in Figure 27.2. Thus, the unit i gathers weighted inputs from x_1 and x_2, and adds a bias u_i, as in

$$z_i = u_i + w_{i1}x_1 + w_{i2}x_2,$$

and produces the output $\sigma(z_i)$, where σ is the sigmoid function (27.1). The outputs of the unit are summed with weights v_i at the output unit, which then produces the overall output

$$N(x_1, x_2) = \sum_{i=1}^{q} v_i \sigma(z_i). \tag{28.2}$$

We note that N is defined in terms of $4q$ parameters, u_i, v_i, w_{1i}, w_{2i}, $i = 1, 2, \ldots, v$.

Derivatives of any order of N with respect to the input variables was calculated in (27.3). In the current case the general formula reduces to

$$\frac{\partial N}{\partial x_1} = \sum_{i=1}^{q} v_i w_{i1} \sigma'(z_i), \qquad \frac{\partial N}{\partial x_2} = \sum_{i=1}^{q} v_i w_{i2} \sigma'(z_2),$$

$$\frac{\partial^2 N}{\partial x_1^2} = \sum_{i=1}^{q} v_i w_{i1}^2 \sigma''(z_i), \quad \frac{\partial^2 N}{\partial x_1 \partial x_2} = \sum_{i=1}^{q} v_i w_{i1} w_{i2} \sigma''(z_i), \quad \frac{\partial^2 N}{\partial x_2^2} = \sum_{i=1}^{q} v_i w_{i1}^2 \sigma''(z_i). \tag{28.3}$$

We seek a solution to (28.1) of the form $u(x, y) = \phi(x, y)N(x, y)$, where $N(x, y)$ is the transfer function of a suitably tuned neural network, and $\phi(x, y)$ is a function which we pick, a priori, to enforce the boundary condition (28.1b). Unlike in the case of ODEs, finding a ϕ which is positive in the domain Ω zero on the boundary, and negative in the outside, is generally a nontrivial task. We will have more to say about this in Section 28.8. Other than that, the workflow is pretty much the same as that in the case of ODEs. We evaluate the residual at (x, y):

$$R(x, y) = F\Big(x, y, \phi N, (\phi N)_x, (\phi N)_y, (\phi N)_{xx}, (\phi N)_{xy}, (\phi N)_{yy}\Big), \tag{28.4}$$

and then calculate the *residual error*

$$E = \sum_{i=}^{v} R(x_i, y_y)^2, \tag{28.5}$$

where $(x_i, y_i) \in \Omega$, $i = 1, 2, \ldots, v$ are suitably selected training points. We pass E to *Nelder Mead* module to find the network's parameters u, v, and w that minimize E. Once the network is thus trained, we apply it to calculate the solution at arbitrary inputs $(x, y) \in \Omega$.

In solving ODEs, we distributed the v training points uniformly within the ODE's domain. The equivalent in the case of PDEs would be to enclose the domain Ω within a bounding box, let's say $(a, b) \times (c, d)$, impose a uniform grid on the bounding box, and then pick those grid points that fall inside Ω as training points. For the sake of variety, however, we choose a different approach. We select v points within the bounding box from a random distribution, and pick the subset that falls inside Ω for training.

28.2 ▪ An overview of the program

Our implementation of the neural networks for solving PDEs is in the file *neural-net-pde.c*. The header file *neural-net-pde.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@     plot-with-matlab.c
array.h@                 nelder-mead.h@     xmalloc.c@
demo-L-shaped.c          neural-nets-pde.c  xmalloc.h@
demo-square-with-hole.c  neural-nets-pde.h
demo-square.c            plot-with-maple.c
```

The primary task in *neural-net-pde.c* is to provide the infrastructure to encode the residual error function E (equation (28.5)) for the generic second order boundary value problem (28.1). The files *demo-*.c* define concrete instances of the boundary value problem (28.1)—see Section 28.5 for the details—and then call the *Nelder Mead* module to minimize E by tuning the neural network's parameters. With the network thus trained, the solution $u(x, y)$ of the boundary value problem may be evaluated at any desired point (x, y).

The files *plot-with-maple.c* defines a function which applies the trained neural network to produce a sequence of pairs $\big(x_i, y_i u(x_i, y_i)\big)$, $i = 0, 1, \ldots, n$, 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.

Note: This chapter's *plot-with-maple.c* and *plot-with-matlab.c* which are available at the book's website, are quite different from the previous chapter's files of the same name which were made for plotting solution curves of ODEs. The ones in this chapter plot solutions of PDEs as surfaces in 3D.

Here is the transcript of a session on executing the compiled *demo-square.c*:

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

We see that when *demo-square* 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 which is used to generate the random training points, as described in the previous section.

When *demo-square* is supplied with the requisite arguments, it produces the output shown in Listing 28.1. That corresponds to solving the boundary value problem with five units in the hidden layer, and six training points. The Nelder–Mead algorithm minimizes the residual error E after 8983 function evaluation. The minimum value of E is of the order 10^{-11}, which is pretty 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-square.c*. Figure 28.2 shows the graphs plotted by MAPLE for the solutions of the boundary value problems defined in *demo-square.eps*, *demo-square-with-hole.eps*, and *demo-L-shaped.eps*. Listing 28.2 shows the outputs generated by executing *demo-square-with-hole* and *demo-L-shaped.c*.

Listing 28.1: Output produced by executing *demo-square*.

```
$ ./demo-square 5 6
q = 5, nu = 6
number of training points = 6 of 6
weights before training:
 -0.135   0.013   0.452   0.416   0.136   0.217  -0.358
  0.107  -0.484  -0.257  -0.363   0.304  -0.343  -0.099
 -0.370  -0.391   0.499  -0.282   0.013   0.339
Nelder-Mead: Converged after 8983 function evaluations
Nelder-Mead: Neural network's residual error = 5.35436e-11
weights after training:
 20.108 -23.137   9.365  -7.271  -9.930  24.238  -1.786
 -8.237  16.158  12.689 -18.810  -1.276   3.854 -11.819
  8.867  17.996  -6.666   3.778 -10.032 -20.416
Error versus the pde's exact solution = 0.00154184
```

Listing 28.2: Outputs produced by executing *demo-square-with-hole* and *demo-L-shaped.c*.

```
$ ./demo-square-with-hole 5 12
    q = 5, nu = 12
    number of training points = 9 of 12
    weights before training:
     -0.343  -0.099  -0.370  -0.391   0.499  -0.282   0.013
      0.339   0.113  -0.204   0.138   0.024  -0.006   0.473
     -0.207   0.271   0.027   0.270  -0.100   0.392
    Nelder-Mead: Converged after 12839 function evaluations
    Nelder-Mead: Neural network's residual error = 0.000888615
    weights after training:
     -2.850   0.214  -0.391  -0.687   1.502  -0.103  -0.678
      0.457   4.097   1.023   0.311  -0.489  -1.341   0.851
      1.874  -0.904  -0.089  -1.671   0.054   0.552
    Error versus the pde's exact solution = 0.00759185

$ ./demo-L-shaped 5 12
    q = 5, nu = 12
    number of training points = 11 of 12
    weights before training:
     -0.343  -0.099  -0.370  -0.391   0.499  -0.282   0.013
      0.339   0.113  -0.204   0.138   0.024  -0.006   0.473
     -0.207   0.271   0.027   0.270  -0.100   0.392
    Nelder-Mead: Converged after 12172 function evaluations
    Nelder-Mead: Neural network's residual error = 7.36134e-07
    weights after training:
     -1.200  -0.299   1.243  -1.440   2.030   0.124  -2.435
      1.176   1.800  -0.276   0.601  -0.470   1.655   1.125
     -1.807  -1.794   0.156   0.235   0.274   0.097
    Error versus the pde's exact solution = 0.00610988
```

28.3 ▪ The interface

The contents of the header file *neural-nets-pde.h* is shown in Listing 28.3. It declares a structure **`struct`** `Neural_Net_PDE` that holds the necessary data for defining a boundary value problem for a PDE and a neural network to solve it. Let us begin by examining the structure's details. Line numbers refer to those in Listing 28.3.

Line 4: The member `PDE` of the **`struct`** `Neural_Net_PDE` points to a user-defined function that defines the differential equation to be solved, which is, in effect, the func-

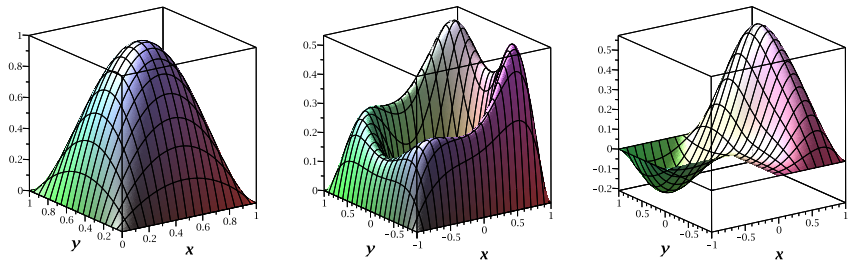

Figure 28.2: The graphs of the solutions $u(x)$ of the boundary value problems defined in *demo-square.eps*, *demo-square-with-hole.eps*, and *demo-L-shaped.eps*, as plotted in MAPLE.

Listing 28.3: The header file *neural-nets-pde.h*.

```
#ifndef H_NEURAL_NET_PDE_H
#define H_NEURAL_NET_PDE_H

struct Neural_Net_PDE {
    double (*PDE)(double x, double y, double u,
            double u_x, double u_y,
            double u_xx, double u_xy, double u_yy);
    void (*phi_func)(struct Neural_Net_PDE *nn, double x, double y);
    double bb_xrange[2];
    double bb_yrange[2];
    int q;                              // number of units in the hidden layer
    int nu;                             // the number of training points
    double **training_points;           // the array of training points
    double (*exact_sol)(double x, double y);
    char *geomview_out;                                 // output file for geomview graphics
    char *maple_out;                                    // output file for maple graphics
    char *matlab_out;                                   // output file for matlab graphics
    int grid[2];                                        // plotting grid size

    // no user modifiable parts beyond this point
    int nweights;                       // 4 × q
    double *weights;                    // the array of u, v, w
    double sigma[3];                    // array to hold σ, σ′, σ″
    double phi[3][3];                   // array to hold φ and its derivatives
    double N[3][3];                     // array to hold N and its derivatives
};

void Neural_Net_init(struct Neural_Net_PDE *nn);
void Neural_Net_end(struct Neural_Net_PDE *nn);
void Neural_Net_eval(struct Neural_Net_PDE *nn, double x, double y);
void Neural_Net_plot_with_maple(struct Neural_Net_PDE *nn, int n, int m,
        char *outfile);
void Neural_Net_plot_with_matlab(struct Neural_Net_PDE *nn, int n, int m,
        char *outfile);
double Neural_Net_residual(double *weights, int nweights,
        void *params);
double Neural_Net_error_vs_exact(struct Neural_Net_PDE *nn, int m, int n);

#endif /* H_NEURAL_NET_PDE_H */
```

tion F in (28.1a). See the description of the file *demo-square.c* for an example.

Line 8: Here is the user-supplied function $\phi(x, y)$ which is positive in the domain Ω, zero on the boundary of the domain, $\partial\Omega$, and negative outside Ω.

Lines 9 and 10: The domain Ω is enclosed in a minimal bounding box $(a, b) \times (c, d)$. The values of a and b are stored in the array `bb_xrange`, and the values of c and d in the array `bb_yrange`.

Line 11: q is the number of units in the hidden layer.

Line 12: v is the number of training points.

Line 13: Pointer to a $v \times 2$ array, whose ith row contains the coordinates (x_i, y_i) of the ith training point.

Line 14: 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 17: As seen in Section 28.1, our neural network is characterized by $4q$ parameters (weights) u_i, v_i, w_{1i}, w_{2i}, $i = 1, \ldots, q$, where q is the number units in the hidden layer. The value of `weights` is set to $4q$ in the function `Neural_Net_init()`. It is not intended to be set by the user.

Line 18: `weights` points to an array of length $4q$ which holds the $4q$ parameters u, v, w_1, and w_2, in that order. Thus, for $i = 0, 1, \ldots, q-1$, we have u_i = `weights[i]`, v_i = `weights[q+i]`, w_{1i} = `weights[2*q+i]`. w_{2i} = `weights[3*q+i]`.

Line 19: The array `sigma` will hold the values of $\sigma(x)$, $\sigma'(x)$, $\sigma''(x)$ at varying values of x, just as in the case of the ODEs.

Lines 20 and 21: The calculation of the residual error E calls for the values of $\phi(x, y)$ and its first and second order derivatives ϕ_x, ϕ_y, ϕ_{xx}, ϕ_{xy}, ϕ_{yy}, evaluated at various trial points (x, y). We could store each trial point's calculated values in a one-dimensional array of length six, as in $\left[\phi, \phi_x, \phi_y, \phi_{xx}, \phi_{xy}, \phi_{yy}\right]$, but we don't, since that association of the various of derivatives with the array indices is not quite natural and can obscure the intent of the code. Instead, we store those values in a 3×3 array `phi` according to the more intuitive association

$$\begin{array}{lcl} \phi & \longrightarrow & \texttt{phi[0][0]} \\ \phi_x & \longrightarrow & \texttt{phi[1][0]} \\ \phi_y & \longrightarrow & \texttt{phi[0][1]} \\ \phi_{xx} & \longrightarrow & \texttt{phi[2][0]} \\ \phi_{xy} & \longrightarrow & \texttt{phi[1][1]} \\ \phi_{yy} & \longrightarrow & \texttt{phi[0][2]} \end{array} \tag{28.6}$$

Only six of the array's nine members are used but that's small price to pay for transparency.

The 3×3 array `N` plays a similar role for storing the values $N(x, y)$ of the neural network's output and and its first and second order derivatives.

Lines 24 to end: The function declared here are the public functions exported by our module. The are described in the next section.

Listing 28.4: An outline of the file *neural-nets-pde.c*. Flesh out the parts marked with ▸.

```
1  ▸ supply the necessary header files
2  ▸ static void sigmoid(double x, double *sigma) ...
3  ▸ void Neural_Net_init(struct Neural_Net_PDE *nn)
4  ▸ void Neural_Net_end(struct Neural_Net_PDE *nn) ...
5  ▸ void Neural_Net_eval(struct Neural_Net_PDE *nn,
6              double x, double y) ...
7  ▸ static double residual_at_x_y(struct Neural_Net_PDE *nn,
8              double x, double y) ...
9  ▸ double Neural_Net_residual(double *weights,
10             int nweights, void *params) ...
11 ▸ double Neural_Net_error_vs_exact(struct Neural_Net_PDE *nn,
12             int m, int n) ...
```

28.4 ▪ The implementation

Listing 28.4 presents an outline of the implementation file *neural-nets-pde.c*. It contains a few private functions (marked **static**) and several public function which appear in *neural-nets-pde.h* in Listing 28.3. I will describe the purposes of the various functions in the following subsections.

28.4.1 ▪ The function `sigmoid()`

The function `sigmoid()` is identical to the function of the same name from Chapter 27. It receives a value x and evaluates the sigmoidal function $\sigma(x)$ (equation (27.1) on page 364) and its derivatives $\sigma'(x)$ and $\sigma''(x)$, and stores them in `sigma[0]`, `sigma[1]`, `sigma[2]`, respectively.

28.4.2 ▪ The function `Neural_Net_init()`

The function `Neural_Net_init()` is almost identical to the function of the same name in Chapter 27. The only difference is that it sets the value of `nn→nweights` to $4q$ instead of $3q$ since the current neural network has $4q$ parameters. The `weights` array is initialized to random values in the range $(-0.5, 0.5)$, as before.

28.4.3 ▪ The function `Neural_Net_end()`

This is also identical to the function of the same name in Chapter 27.

28.4.4 ▪ The function `Neural_Net_eval()`

This function evaluate the output $N(x, y)$ of the neural network corresponding to the input (x, y), and its first and second order derivatives, according to (28.2) and (28.3), and places them in the array `N` attached to the structure `nn` according to

$$\begin{array}{lcl} N & \longrightarrow & \texttt{N[0][0]} \\ N_x & \longrightarrow & \texttt{N[1][0]} \\ N_y & \longrightarrow & \texttt{N[0][1]} \\ N_{xx} & \longrightarrow & \texttt{N[2][0]} \\ N_{xy} & \longrightarrow & \texttt{N[1][1]} \\ N_{yy} & \longrightarrow & \texttt{N[0][2]} \end{array}$$

28.4.5 ▪ The function `residual_at_x_y()`

This function calculates and returns the residual $R(x)$ at a given x, according to (28.4). 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`.

28.4.6 ▪ The function `Neural_Net_residual()`

This function evaluates the residual error E according to (28.5). It is different from the function of the same name in Chapter 27 only in minor (and self-evident) details. As before, the function's prototype exactly matches 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, w_1, and w_2 which define the neural network, the first argument of `Neural_Net_residual()` is an array of length $4q$ that holds the values of u, v, w_1, and w_2.

28.4.7 ▪ The function `Neural_Net_error_vs_exact()`

The purpose of this function is to evaluate the now trained neural network on an $(m+1)\times(n+1)$ grid superimposed on the domain Ω and calculate and return the largest discrepancy

$$\max_{\substack{0\le i\le m\\ 0\le j\le n}} \left|\phi(x_i, y_j)N(x_i, y_j) - u_{\text{exact}}(x_i, y_i)\right|$$

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.

The grid is formed by dividing the domain's bounding box into m equal subintervals in the x direction and n equal subintervals in the y direction. Since Ω is generally a strict subset of the bounding box, some of the grid points may fall outside Ω. The maximization noted above is performed only on those grid points that fall *within* Ω. A grid point is inside Ω if the ϕ function is positive at that point. The complete implementation of `Neural_Net_error_vs_exact()` is given in Listing 28.5.

28.5 ▪ The file *demo-square.c*

The file *demo-square.c* provides a demonstration of this module. The unknown is a function $u(x, y)$ on the square $\Omega = (0, 1)\times(0, 1)$, and the boundary value problem is[100]

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} = f(x, y) \qquad (x, y) \in \Omega, \tag{28.7a}$$

$$u = 0 \qquad \text{in } \partial\Omega, \tag{28.7b}$$

where $f(x)$ is selected as

$$f(x, y) = 32(x^2 + y^2 - x - y) \tag{28.7c}$$

so that the exact solution of the problem is

$$u(x) = 16x(1 - x)y(1 - y). \tag{28.8}$$

[100] The PDE (28.7a) is called *Poisson's equation*, and the equations (28.7a) and (28.7b) together constitute *Poisson's boundary value problem* on the two-dimensional domain Ω.

Listing 28.5: The function `Neural_Net_error_vs_exact()` in the file *neural-nets-pde.c*

```
1  double Neural_Net_error_vs_exact(struct Neural_Net_PDE *nn,
2          int m, int n)
3  {
4      if (nn→exact_sol == NULL) {
5          fprintf(stderr,
6                  "unable to cacluate the error since "
7                  "no exact solution is provided\n");
8          exit(EXIT_FAILURE);
9      }
10
11     double max_err = 0.0;
12     double a = nn→bb_xrange[0];
13     double b = nn→bb_xrange[1];
14     double c = nn→bb_yrange[0];
15     double d = nn→bb_yrange[1];
16
17     for (int i = 0; i ≤ m; i++) {
18         double x = a + (b-a)/m*i;
19         for (int j = 0; j ≤ n; j++) {
20             double y = c + (d-c)/n*j;
21                 nn→phi_func(nn, x, y);
22                 if (nn→phi[0][0] < 0)
23                     continue;
24                 Neural_Net_eval(nn, x, y);
25                 double z = nn→N[0][0] * nn→phi[0][0];
26                 double err = fabs(z - nn→exact_sol(x, y));
27                 if (err > max_err)
28                     max_err = err;
29         }
30     }
31
32     return max_err;
33 }
```

Listing 28.6: A sketch of the file *demo-square.c*. Flesh out the parts marked with ▸.

```
1  ▸ the necessary headers here
2  ▸ static double exact_sol(double x, double y) ...
3  ▸ static double my_pde(double x, double u, double u_x, double u_y,
4              double u_xx, double u_xy, double u_yy) ...
5  ▸ static void my_phi(struct Neural_Net_PDE *nn,
6              double x, double y) ...
7  ▸ static void show_usage(char *progname) ...
8  ▸ int main(int argc, char **argv) ...
```

This is a special case of the boundary value problem (28.1) with

$$F\Big(x, u, u_x, u_y, u_{xx}, u_{xy}, u_{yy}\Big) = u_{xx} + u_{yy} - f(x, y). \tag{28.9}$$

Listing 28.6 provides an outline of the file *demo-square.c*. Here are a few comments on that listing.

Line 2: This function evaluates and returns exact solution of the boundary value problem, given in (28.8). 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.

Listing 28.7: A fragment from the function `main()` in *demo-square.c*, illustrating how random training points are selected in Ω.

```
1     make_matrix(nn.training_points, nu, 2);
2     int i = 0;
3     while (i < nu) {
4         double r = (double)rand() / RAND_MAX;
5         double s = (double)rand() / RAND_MAX;
6         double x = (1-r)*nn.bb_xrange[0] + r*nn.bb_xrange[1];
7         double y = (1-s)*nn.bb_yrange[0] + s*nn.bb_yrange[1];
8         nn.phi_func(&nn, x, y);
9         if (nn.phi[0][0] > 0) {
10            nn.training_points[i][0] = x;
11            nn.training_points[i][1] = y;
12            i++;
13        }
14    }
```

Line 3: The function `my_pde()` implements the function F of equation (28.1a) for the PDE at hand. The F of interest in the current code is that in (28.9).

Line 5: A good choice of ϕ for this demo is $\phi(x, y) = x(1 - x)y(1 - y)$. The function `my_phi()` (the name is immaterial) calculates ϕ and its first and second derivatives at the given (x, y), and stores them in the two-dimensional array `nn→phi` according to the scheme illustrated in (28.6).

Line 7: The function `show_usage()` is identical to the function of the same name in Chapter 27.

Line 8: The function `main()` is different from the one in Chapter 27's Listings 27.4 and 27.5 only in minor ways. A notable difference is in the construction of training points. In the previous chapter we picked a uniformly distributed set of ν points along the ODE's domain, which was the interval (a, b). Although we could construct the analogous uniformly distributed training points by imposing a $\mu \times \mu$ grid on the domain Ω, where $\mu \approx \sqrt{\nu}$, we choose instead, for the sake of variety, to pick the training points *randomly* within the PDE's domain Ω. Listing 28.7 shows one way of doing that. The method is not specific to rectangular domains. It applies to any domain Ω that can fit within a finite bounding box.

In line 1 of that listing we allocate memory for a $\nu \times 2$ array to store up to ν training points (x_i, y_i). Then within the **`while`**-loop we produce random numbers r and s in the interval $[0, 1]$, and then pick a point (x, y) within Ω's bounding box via

$$x = (1 - r)a + rb, \quad y = (1 - s)c + sd. \qquad \big[\text{bounding box} = (a, b) \times (c, d)\big]$$

If $\phi(x, y) > 0$, then the point lies within Ω. We store the point in the ith slot of the array of training points, and then increment the index i. Otherwise we discard the point. We repeat this until the the desired number, ν, of training points are produced.

28.6 ▪ The file *demo-square-with-hole.c*

This file applies our neural network solver to the Poisson's problem

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} = f(x, y) \qquad (x, y) \in \Omega, \tag{28.10}$$

$$u = 0 \qquad \text{in } \partial\Omega, \tag{28.11}$$

on the domain Ω obtained by removing the disk $x^2 + y^2 \le 1/4$ from the square $(-1, 1) \times (-1, 1)$.

We construct a (u, f) pair by first picking a solution u which satisfies the boundary conditions,

$$u(x, y) = (2 + x)(1 - x^2)(1 - y^2)(x^2 + y^2 - 1/4), \tag{28.12}$$

and then calculating f by substituting (28.12) in (28.10). We get

$$f(x, y) = 2\,x^5 + 4\,x^4 + \frac{\left(64\,y^2 - 49\right)x^3}{2} + \frac{\left(96\,y^2 - 66\right)x^2}{2} + \frac{\left(12\,y^4 - 51\,y^2 + 20\right)x}{2} + 4\,y^4 - 33\,y^2 + 10. \tag{28.13}$$

We feed (28.12) to the neural net solver and then compare the solution it produces against the exact solution (28.12).

For the cutoff function ϕ we pick the natural choice:

$$\phi(x, y) = (1 - x^2)(1 - y^2)(x^2 + y^2 - 1/4).$$

The first and second order derivatives of ϕ can be readily calculated

$$\begin{aligned}
\phi_x &= -2\,x\left(-y^2 + 1\right)\left(x^2 + y^2 - 1/4\right) + 2\left(-x^2 + 1\right)\left(-y^2 + 1\right)x,\\
\phi_y &= -2\left(-x^2 + 1\right)y\left(x^2 + y^2 - 1/4\right) + 2\left(-x^2 + 1\right)\left(-y^2 + 1\right)y,\\
\phi_{xx} &= -2\left(-y^2 + 1\right)\left(x^2 + y^2 - 1/4\right) - 8\,x^2\left(-y^2 + 1\right) + 2\left(-x^2 + 1\right)\left(-y^2 + 1\right),\\
\phi_{xy} &= 4\,xy\left(x^2 + y^2 - 1/4\right) - 4\,x\left(-y^2 + 1\right)y - 4\left(-x^2 + 1\right)yx,\\
\phi_{yy} &= -2\left(-x^2 + 1\right)\left(x^2 + y^2 - 1/4\right) - 8\left(-x^2 + 1\right)y^2 + 2\left(-x^2 + 1\right)\left(-y^2 + 1\right).
\end{aligned}$$

Translating these expressions and (28.13) and keying them in can be tiresome and error-prone. To simplify the process, I calculated those expressions in MAPLE, and had it translate the results to C. I transferred the results to the file *demo-square-with-hole.c* and constructed the functions `my_pde()` and `my_phi()` around them. Listing 28.8 shows the finished product. The code of that listing is available at the book's website.

28.7 ▪ The file *demo-L-shaped.c*

This file applies our neural network solver to the Poisson's problem

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} = f(x, y) \qquad (x, y) \in \Omega, \tag{28.14}$$

$$u = 0 \qquad \text{in } \partial\Omega, \tag{28.15}$$

on the L-shaped domain Ω obtained by removing the lower-lower left quarter from the square $(-1, 1) \times (-1, 1)$.

Listing 28.8: The functions `my_pde()` and `my_phi` in the file *demo-square-with-hole.c*.

```
static double my_pde(double x, double y, double u,
        double u_x, double u_y,
        double u_xx, double u_xy, double u_yy)
{
    double t1 = x * x;
    double t2 = t1 * t1;
    double t6 = y * y;
    double t16 = t6 * t6;
    double f = 0.2e1 * t2 * x + 0.4e1 * t2
        + (0.64e2 * t6 - 0.49e2) * t1 * x / 0.2e1
        + (0.96e2 * t6 - 0.66e2) * t1 / 0.2e1
        + (0.12e2 * t16 - 0.51e2 * t6 + 0.20e2) * x / 0.2e1
        + 0.4e1 * t16 - 0.33e2 * t6 + 0.10e2;
    return u_xx + u_yy - f;
}

static void my_phi(struct Neural_Net_PDE *nn, double x, double y)
{
    double t1 = x * x;
    double t2 = -t1 + 0.1e1;
    double t4 = y * y;
    double t5 = -t4 + 0.1e1;
    double t6 = t1 + t4 - 0.1e1 / 0.4e1;
    double t7 = t5 * t6;
    double t9 = x * t5;
    double t11 = t2 * t5;
    double t15 = t2 * y;
    double t23 = 0.2e1 * t11;

    nn->phi[0][0] = t2 * t7;
    nn->phi[1][0] = 0.2e1 * (t11 * x - t9 * t6);
    nn->phi[0][1] = 0.2e1 * (t11 * y - t15 * t6);
    nn->phi[2][0] = -0.8e1 * t1 * t5 + t23 - 0.2e1 * t7;
    nn->phi[1][1] = 0.4e1 * (x * y * t6 - t15 * x - t9 * y);
    nn->phi[0][2] = -0.8e1 * t2 * t4 - 0.2e1 * t2 * t6 + t23;
}
```

Constructing a cutoff function ϕ which is positive inside Ω and negative outside can be challenging. Here I will describe the details of a construction that leads to such a function.

Let us begin by constructing a continuous function $q : R^2 \longrightarrow R$ which is negative in the third quadrant and positive elsewhere. It is best to construct such a function in in polar coordinates, r, θ, and then translate the result to the Cartesian coordinates.

We let the angular coordinate θ go from $-\pi$ to π. Thus, the third quadrant corresponds to $\theta \in (-\pi, \pi/2)$. Let $f : (-\pi, \pi) \longrightarrow R$ be a continuous function which is negative on $(-\pi, -\pi/2)$ and positive on $(-\pi/2, \pi)$. Such a function is not difficult to find. For instance, the function $f(\theta) = 1 + \sin\theta + \cos\theta$ has that property. It follows that the function $\tilde{q}(r, \theta) = rf(\theta)$, with $0 \le r < \infty$ and $-\pi \le \theta < \pi$ is negative in the third quadrant and positive elsewhere. An elementary calculation shows that upon changing from polar to Cartesian, the function $\tilde{q}(r, \theta)$ transforms to $q(x, y) = x + y + \sqrt{x^2 + y^2}$. The graph of q is shown in the left diagram in Figure 28.3.

To enforce the zero boundary conditions on the domain Ω, it suffices to multiply $q(x, y)$ by $(1 - x^2)(1 - y^2)$ which is positive in the domain's bounding box and negative elsewhere.

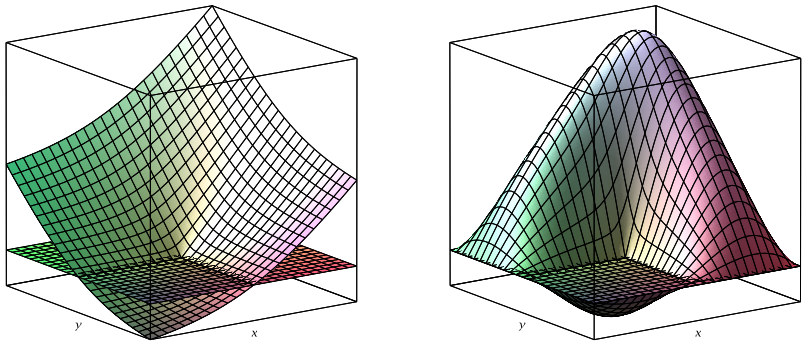

Figure 28.3: On the left is the graph of the function $q(x, y)$, which is negative in the third quadrant and positive elsewhere. On the right is the graph of $\phi(x, y)$, defined on the square $(-1, 1) \times (-1, 1)$. It is positive on the L-shaped domain Ω, zero on its boundary, and negative in the rest of the square.

Thus we arrive at the following representation of the desired cutoff function:

$$\phi(x, y) = (1 - x^2)(1 - y^2)\Big(x + y + \sqrt{x^2 + y^2}\Big). \tag{28.16}$$

The graph of ϕ is shown in the right diagram in Figure 28.3. It is positive on the L-shaped domain Ω, zero on its boundary, and negative in the rest of Ω's bounding box.

Remark 28.1. It is worth nothing that function ϕ constructed here takes on both positive and negative values *outside the bounding box*. For instance, the points $(3, 4)$ and $(2, 0)$ lie outside of the bounding box, and we have $\phi(3, 4) = 1440 > 0$ while $\phi(2, 0) = -12 < 0$. That said, values taken by ϕ outside the bounding box are immaterial to us since we evaluate ϕ inside the bounding box only, where positive and negative value of $\phi(x, y)$ indicate that (x, y) is inside or outside Ω, respectively.

In Section **??** we will learn how to construct a continuous cutoff function ϕ which is positive in Ω and negative *everywhere* outside Ω. Although that results in a significantly more complex our current ϕ, the method described there may be applied to construct cutoff functions for quite complex geometries where an ad hoc approach that led us to (28.16) would not be practical.

Having constructed the cutoff function ϕ, it's easy to construct other functions which take on zero values on the boundary of Ω. For instance, the function

$$u(x, y) = x\phi(x, y) \tag{28.17}$$

has that property. We plug this into (28.14) and calculate the corresponding f. We feed that f to our neural network solver and then compare the solution it produces against the exact solution (28.17).

The calculation of f and first and second derivatives of ϕ is a messy affair and is best left to a symbolic calculation software. I had Maple do those calculations and translate the results to C code. I transferred the Maple's output to the file *demo-L-shaped.c* and constructed the functions `my_pde()` and `my_phi()` around them. The result is available for downloading from the book's website.

28.8 ▪ Rvachev's method for constructing cutoff functions

TO BE ADDED LATER.

28.9 ▪ *Project Neural networks: PDEs*

Part 28.1. Complete the implementation of the this chapter's PDE solver, and test it with *demo-square-with-hole.c*, *demo-square.c*, *demo-L-shaped.c*. Sample outputs are provided in Listings 28.1 and 28.2. The graphs of the resulting solutions are shown in Figure 28.2.

Part 28.2. Write a program file *exercise-u3.c* to solve the nonlinear boundary value problem

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} - u^3 = f(x, y), \qquad (x, y) \in \Omega, \tag{28.18}$$

$$u = 0 \qquad \text{in } \partial\Omega, \tag{28.19}$$

on the square $\Omega = (0.\pi) \times (0, \pi)$, with

$$f(x, y) = -2\Big(x - \frac{\pi}{4}\Big)\Big(y - \frac{\pi}{3}\Big)\sin x \sin y + 2\Big(y - \frac{\pi}{3}\Big)\cos x \sin y + 2\Big(x - \frac{\pi}{4}\Big)\sin x \cos y - \left[\Big(x - \frac{\pi}{4}\Big)\Big(y - \frac{\pi}{3}\Big)\sin x \sin y\right]^3.$$

This corresponds to the exact solution $u(x, y) = (x - \pi/4)(y - \pi/3)\sin x \sin y$. There are two obvious choices for the cutoff function: $\phi(x, y) = xy(\pi - x)(\pi - y)$ and $\phi(x, y) = \sin x \sin y$. The algorithm seems to convergence faster with the first choice.

See if you can select the command-line arguments q and v so as to produce a solution with an error below 0.05.