INFSCI 2595 Spring 2020: Homework 06

Assigned April 3, 2020; Due: April 10, 2020

Ying Zhang

Submission time: April 10, 2020 at 11:30PM EST

Collaborators Wenbiao Li, Yen-Ming Chen, Ying Zhang, Qiang Tang

Overview

This assignment is focused on optimization of neural network parameters (weights and biases) for a regression task. The optimization problem is formulated to minimize the sum of squared errors (SSE), rather than to maximize the Gaussian log-likelihood. In this way, the assignment is consistent with the derivations presented in lecture. As discussed in lecture however, minimizing the SSE is analogous to maximizing the Gaussian log-likelihood.

You will first program the SSE objective function and then use `optim()` to fit the neural network using a quasi-Newton optimization algorithm (specifically BFGS). The default setting in `optim()` is to estimate the gradient vector with finite differences. As we discussed, such an approach does not scale to large problems. To overcome this bottle neck, you will then implement the backpropagation algorithm to analytically calculate the gradient vector. In this assignment you will program the backpropagation equations in matrix form, and so no for-loops are required. Neural networks are extensions of the generalized linear model. Therefore, the lectures covering derivations of linear and generalized linear models will be useful for helping to write out the backpropagation equations in matrix form.

After programing the backpropagation equations and fitting the neural network on your own, you will use existing libraries to fit multilayer neural networks and identify the best number of hidden units to use. A large portion of the code required to complete this assignment has been provided to you in the in class demonstration `R` scripts. Simply copying and pasting that code will not work, but the scripts given to you will certainly help complete most of the problems in this assignment.

IMPORTANT: Please pay attention to the `eval` flag within the code chunk options. Code chunks with `eval=FALSE` will **not** be evaluated (executed) when you Knit the document. You **must** change the `eval` flag to be `eval=TRUE`. This was done so that you can Knit (and thus render) the document as you work on the assignment, without worrying about errors crashing the code in questions you have not started. Code chunks which require you to enter all of the required code do not set the `eval` flag. Thus, those specific code chunks use the default option of `eval=TRUE`.

Load packages

The code chunk below loads in the usual `dplyr` and `ggplot2` packages. You will also need several other packages from the `tidyverse` to complete this assignment, just as in previous assignments. The last problems require you to use the `neuralent` and `caret` packages. Those packages are loaded in later in the assignment. If you have not downloaded and installed those packages, please do so before starting the assignment. You may use the RStudio Install Packages GUI to download them.

```
library(dplyr)
library(ggplot2)
```

Load data

This assignment uses a single data set, that is read in for you in the code chunk below. As shown by the `glimpse()` function there are just 2 variables. The variable `x` is a continuous input and the variable `y` is a continuous response.

```
train_data <- readr::read_csv("https://raw.githubusercontent.com/jyurko/INFSCI_2595_Spring_2020/master/
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_double()
## )
```

```
train_data %>% glimpse()
```

```
## Rows: 250
## Columns: 2
## $ x <dbl> -1.4952076, 1.5834295, 0.8732551, -0.0223565, -1.3466422, 1.32522...
## $ y <dbl> 1.28773997, 2.03997239, 2.35159181, 0.55792266, 1.02477609, 2.132...
```

Problem 1

In this problem you will get famaliar with the data and the basics of the neural network architecture.

1a)

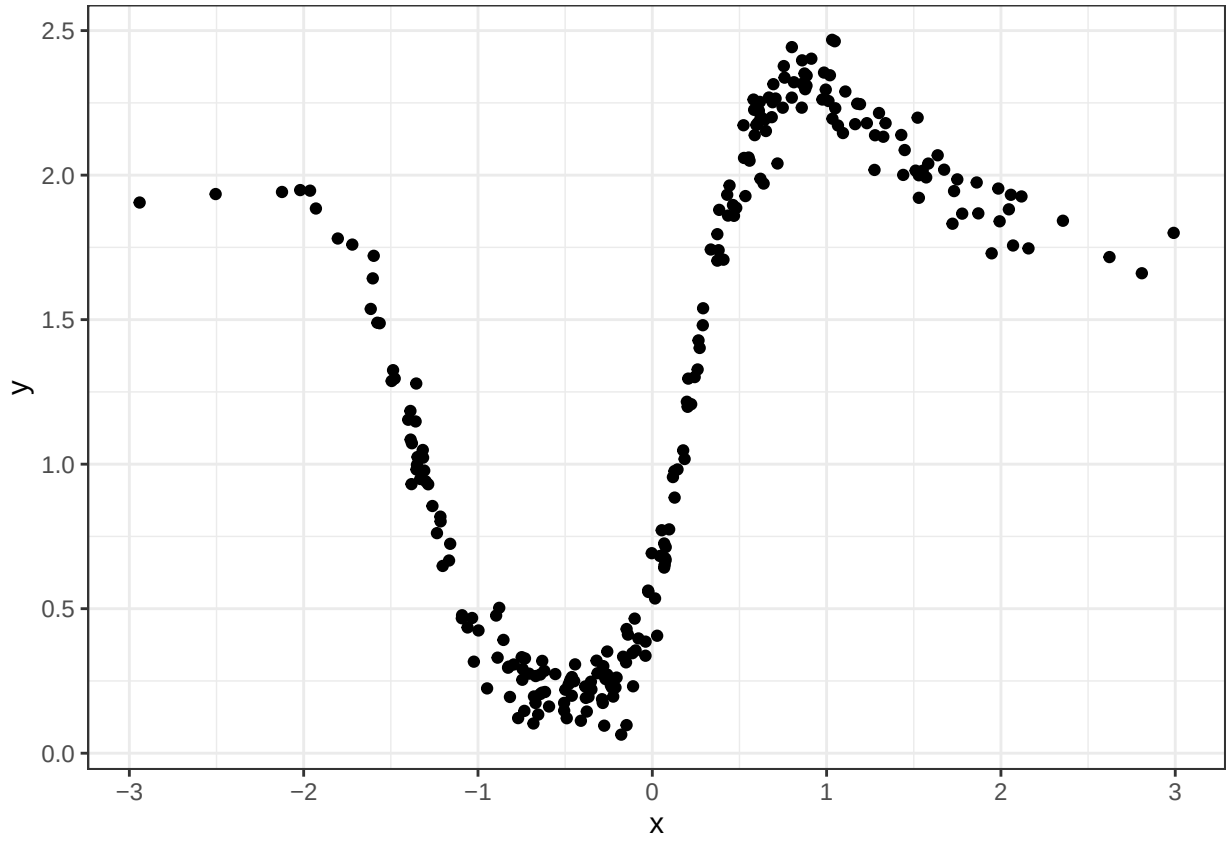
You will ultimately fit several networks of varying degrees of complexity to the data. However, before doing that let's just visualize the data. After making a basic scatter plot, you will use the `geom_smooth()` function to help visualize the non-linear relationship between the input and the response. Smoothing functions have their own set of parameters, and so you will try out the default arguments and manually change that argument.

The `geom_smooth()` function by default will use the locally weighted scatter plot (LOESS) smoother. The degree of "wiggliness" of a LOESS smoother is controlled by the `span` argument. You can think of `span` as the similar to the degrees-of-freedom from the natural spline models we considered earlier in the semester. The lower the `span` value is the more local behavior the smoother will capture (thus small `span` values are similar to high degrees-of-freedom splines).

PROBLEM **You will create three figures. First, create a scatter plot between the response and the input using `ggplot()`. Second, create the same basic scatter plot, but add a smoother with the `geom_smooth()` function. Lastly, create the scatter plot with the smoother, but set the `span` argument within `geom_smooth()` to be 0.2. You should not specify `span` within `aes()`.**

SOLUTION Create the simple scatter plot.

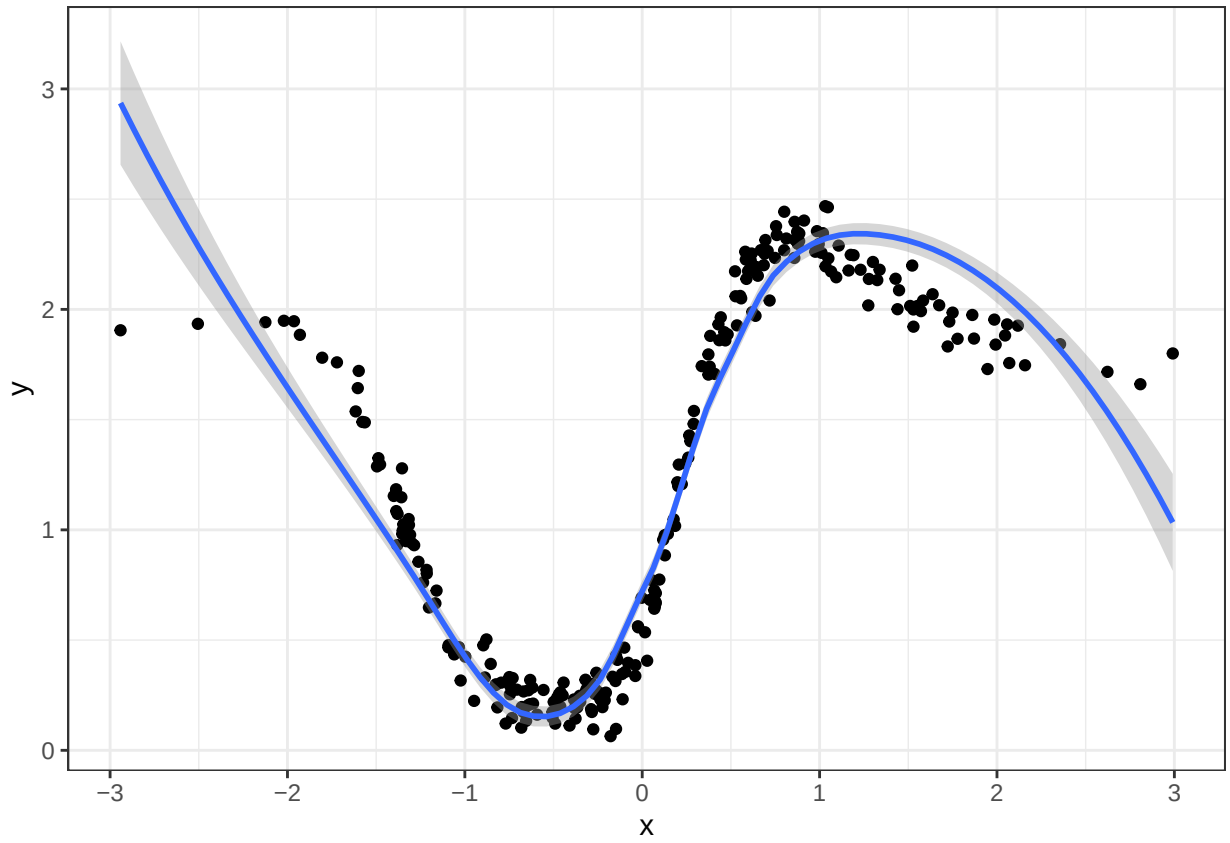
```
train_data%>%
ggplot(mapping = aes(x=x,y=y))+
  geom_point() +
  theme_bw()
```



Next add in the default LOESS smoother.

```
train_data%>%
ggplot(mapping = aes(x=x,y=y))+
  geom_point()+
  geom_smooth(method = 'loess') +
  theme_bw()
```

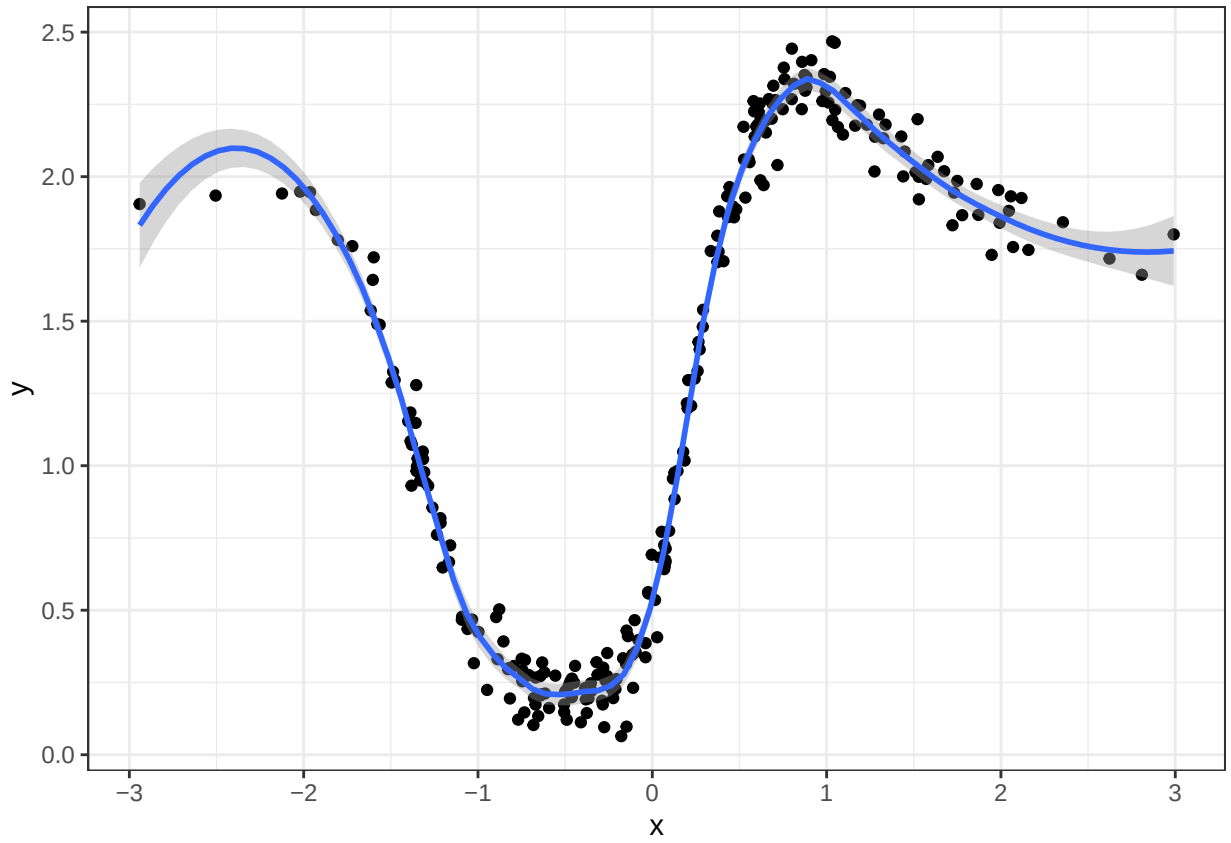
```
## `geom_smooth()` using formula 'y ~ x'
```



And then manually adjust the **span** parameter.

```
train_data%>%
ggplot(mapping = aes(x=x,y=y))+
  geom_point()+
  geom_smooth(method = 'loess', span = 0.2
              ) +
  theme_bw()
```

```
## `geom_smooth()` using formula 'y ~ x'
```



1b)

We need to define the objective function (or analogously the likelihood) between the neural network output and the observed response in order to fit or train a neural network. We therefore need to first go through how a neural network calculates the response given the inputs.

We will use nomenclature consistent with the lecture slides and so assume that there are H hidden units in a single layer neural network. The hidden units will be indexed by the letter k. The hidden unit parameters (the weights and biases) will be denoted as β_{dk}, which corresponds to the d-th input within the k-th hidden unit. The $\beta_{d=0,k}$ parameter is the intercept or bias associated with the k-th hidden unit. The k-th hidden unit's linear predictor for the n-th observation will be denoted as $\eta_{n,k}$. The linear predictor is usually described in neural network literature as a linear combination of the inputs. However, as discussed in lecture, the neural network can be represented as a series of matrix operations, which map the inputs to the response.

Assume that the D inputs (which in this case is $D = 1$) have been stored in a $N \times (D + 1)$ design matrix, $\mathbf{X}$. The intercept column of ones has therefore been included in the design matrix. Also assume that the hidden unit parameters are contained in H separate $(D + 1) \times 1$ column vectors, $\boldsymbol{\beta}_k$. Those column vectors have been concatenated together into the $(D + 1) \times H$ matrix of all hidden unit parameters, $\mathbf{B}$.

PROBLEM Write the expression which calculates all H linear predictors for all N observations in matrix notation. Denote the matrix of linear predictors as A. What is the dimensionality of the A matrix?

SOLUTION

$$\mathbf{A} = \mathbf{XB}$$

The dimensionality of the **A** matrix is N X H.

1c)

The flexibility of the neural network comes from transforming the linear predictors through non-linear transformation functions. These transformations, which are usually referred to as activation functions, allow the neural network to adapt a basis to the data.

For generality, denote the transformation function as $g(\cdot)$. If we wanted to, we could apply different transformations to each hidden unit in the model. However, within this assignment we will apply the same transformation function to all hidden units. Thus, the matrix of all hidden unit transformed responses, **H**, can be calcualted by passing the matrix of linear predictors, **A**, through the transformation function:

$$\mathbf{H} = g(\mathbf{A})$$

For a continuous response, the neural network output, f, is a linear model of hidden unit transformed responses. The output layer consists of its own set of parameters, which are distinct from the hidden layer. The output layer intercept (bias) is denoted as α_0 and the output layer weights are stored in the $H \times 1$ column vector $\boldsymbol{\alpha}$.

PROBLEM Write the expression for the neural network response in matrix notation using the hidden unit transformed response matrix H and the output layer parameters, α_0 and $\boldsymbol{\alpha}$. Denote the response as the vector f. What is the dimensionality of the f vector?

SOLUTION

$$\mathbf{f} = \alpha_0 + \mathbf{H}\boldsymbol{\alpha}$$

The dimensionality of the **f** vector is N X 1.

1d)

We will minimze the SSE to fit the neural network in a non-probabilistic setting. The SSE is calculated relative to the observed response y and the neural network output f.

PROBLEM Write the expression for the SSE using the observed response y and the neural network output f. You may write the SSE in either summation or matrix/vector notation. If you use summation notation use the subscript n to denote a single observation. If you use matrix/vector notation, denote the response vector as y.

SOLUTION

$$SSE = \sum_{n=1}^{N}((y_n - f_n)^2)$$

1e)

You will program the expressions from the previous problems into the `my_neuralnet_sse()` function. The function will be setup in the same way as the log-posterior functions from previous assignments. It will consist of two input arguments, a vector of parameters to learn and a list of required information. Before defining the function, you will create the list of information, which is started for you in the code chunk below. Notice that the structure is similar to the lists of information created for linear models. However, two

pieces of information not associated with linear models are required for the neural network. The variable `$num_hidden` specifies the number of hidden units and the variable `$transform_hidden` stores the non-linear transformation function to apply to each hidden unit.

You will start out with a small neural network consisting of just 2 hidden units. Although there are many possible non-linear transformation functions we could use, you will use the logistic function since we have worked with that function throughout the semester. The `boot::inv.logit()` function is assigned to the `$transform_hidden` variable for you. Note that the `()` are not included in the assignment because we are assigning the function object to the variable.

PROBLEM Complete the list of required information by completing the code chunk below. You must create the design matrix based on the single input `x`. Assign the design matrix to the `$design_matrix` variable in the list. Assign the observed responses to the `$yobs` variable in the list. Set the number of hidden units to be 2.

After specifying the `info_two_units` list, calculate the total number of parameters in the single hidden layer neural network with 2 hidden units and assign the result to the `info_two_units$num_params` variable.

SOLUTION The code chunk is started for you below.

```
### design matrix
Xmat <- model.matrix( ~ x, data = train_data)

info_two_units <- list(
  yobs = train_data$y,
  design_matrix = Xmat,
  num_hidden = 2,
  transform_hidden = boot::inv.logit
)

info_two_units$num_params <- info_two_units$num_hidden * ncol(info_two_units$design_matrix) +
  info_two_units$num_hidden + 1
```

The total number of hidden units you calculated in the above code chunk are printed to the screen below.

```
info_two_units$num_params
```

```
## [1] 7
```

1f)

You will now define the SSE objective in the `my_neuralnet_sse()` function below. As described previously, the function consists of two input arguments. The first argument, `theta`, contains all of the unknown parameters to learn. The vector is organized with all hidden unit parameters listed before the output layer parameters. The first part of the `my_neuralnet_sse()` function is completed for you. The hidden unit parameters are extracted from the `theta` vector and organized into the `Bmat` matrix with dimensions consistent with the **B** described in Problem 1b). The output layer parameters are organized into the scalar `a0` for the output layer intercept (bias) and the weights are stored in the "regular" vector `aw`.

You must complete the function by performing the necessary matrix math calculations, transformations, and calculation of the SSE. The comments in the function describe what you must complete in each line.

After completing the function, test that it works using two separate guesses for the unknown parameters. First set all parameters to a value of 0, then set all parameters to a value of -1.25. If your function is specified correctly the SSE should be 544.8725 for the guess of all 0's and it should be 2627.587 for the guess -1.25 for all parameters.

PROBLEM Complete the `my_neuralnet_sse()` function below and test it's operation with the two guesses specified in the problem statement.

SOLUTION The `my_neuralnet_sse()` function is started for you in the code chunk below.

```
my_neuralnet_sse <- function(theta, my_info)
{
  # extract the hidden unit parameters
  X <- my_info$design_matrix
  length_beta <- ncol(X)
  total_num_betas <- my_info$num_hidden * length_beta
  beta_vec <- theta[1:total_num_betas]

  # extract the output layer parameters
  a_all <- theta[(total_num_betas + 1):length(theta)]

  # reorganize the beta parameters into a matrix
  Bmat <- matrix(beta_vec, nrow = length_beta, byrow = FALSE)

  # reorganize the output layer parameters by extracting
  # the output layer intercept (the bias)
  a0 <- a_all[1] # the first element in a_all is the bias
  aw <- a_all[-1] # select all EXCEPT the first element

  # calculate the linear predictors associated with
  # each hidden unit
  A <- X %*% Bmat

  # pass through the non-linear transformation function
  H <- my_info$transform_hidden(A)

  # calculate the response (the output layer)
  f <- as.vector(a0 + H %*% as.matrix(aw))

  # calculate the SSE
  sum((my_info$yobs - f)^2)

}
```

Test out your `my_neuralnet_sse()` function with values of 0 for all parameters.

```
my_neuralnet_sse(rep(0, info_two_units$num_params), info_two_units)
```

```
## [1] 544.8725
```

Test out your `my_neuralnet_sse()` function with values of -1.25 for all parameters.

```
my_neuralnet_sse(rep(-1.25, info_two_units$num_params), info_two_units)
```

```
## [1] 2627.587
```

1g)

With the objective function completed, it's now time to fit the simple neural network with 2 hidden units. You will use the `optim()` function to perform the optimization, just as you have done in the previous assignments. Since we are fitting in a non-probabilistic setting, you will work with `optim()` itself, rather than within the `my_laplace()` wrapper as in previous assignments.

You will fit two neural networks from two different starting guess values. The first starting guess will be a vector of 0's, and the second guess will be -1.25 for all parameters. Complete the two code chunks below by specifying the initial guesses correctly and completing the remaining input arguments to the `optim()` call. You must set the `gr` argument to so that `optim()` uses finite differences to estimate the gradient vector. Pass in the `info_two_units` list of required information to both `optim()` calls. Specify the `method` argument to be `"BFGS"` to use the quasi-Newton BFGS algorithm. Set the `hessian` argument to be TRUE which forces the Hessian matrix to be estimated at the end and returned. The maximum number of iterations is set for you in both `optim()` calls already.

PROBLEM Complete both code chunks below in order to fit the two hidden unit neural network with two different starting guesses. Follow the instructions in the problem statement to specify all the arguments to the `optim()` calls.

After fitting, print out the identified optimal parameters contained in the `$par` field of the `optim()` results for both cases. Are the identified optimal parameter values the same between the two starting guesses? Why would the results not be the same?

SOLUTION Fit the neural network with the initial guess of 0's for all parameters.

```
optim_fit_2_a <- optim(rep(0, info_two_units$num_params),
                       my_neuralnet_sse ,
                       gr = NULL,
                       info_two_units,
                       method = "BFGS",
                       hessian = TRUE,
                       control = list(maxit = 5001))
optim_fit_2_a
```

```
## $par
## [1] -3.2675051 12.6930303 -3.2675051 12.6930303  0.5869714  0.7669092  0.7669092
##
## $value
## [1] 36.487
##
## $counts
## function gradient
##       48       24
##
## $convergence
## [1] 0
```

```
##
## $message
## NULL
##
## $hessian
##           [,1]       [,2]      [,3]       [,4]       [,5]       [,6]       [,7]
## [1,] 1.0313132 0.21674056 1.1329694 0.27882720   9.204326   4.296309   4.296247
## [2,] 0.2167406 0.06466815 0.2788272 0.07824525   2.253818   1.446623   1.446592
## [3,] 1.1329694 0.27882720 1.0313132 0.21674056   9.204326   4.296247   4.296309
## [4,] 0.2788272 0.07824525 0.2167406 0.06466815   2.253818   1.446592   1.446623
## [5,] 9.2043258 2.25381826 9.2043258 2.25381826 500.000000 210.136005 210.136005
## [6,] 4.2963089 1.44662330 4.2962466 1.44659204 210.136005 198.134159 198.134159
## [7,] 4.2962466 1.44659204 4.2963089 1.44662330 210.136005 198.134159 198.134159
```

Fit the neural network with the initial guess of -1.25 for all parameters.

```
optim_fit_2_b <- optim(rep(-1.25, info_two_units$num_params),
                       my_neuralnet_sse ,
                       gr = NULL,
                       info_two_units,
                       method = "BFGS",
                       hessian = TRUE,
                       control = list(maxit = 5001))
optim_fit_2_b
```

```
## $par
## [1]   6.0007361 -16.3217346   3.6115880 -24.6329881   2.1266423  -0.8220143
## [7]  -0.7170734
##
## $value
## [1] 36.21362
##
## $counts
## function gradient
##      153       91
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]        [,2]        [,3]        [,4]       [,5]        [,6]
## [1,]  0.86792019  0.32262134  0.14964396  0.03117615  -7.079967  -3.4337554
## [2,]  0.32262134  0.12724260  0.03117615  0.00722321  -2.633954  -1.0238975
## [3,]  0.14964396  0.03117615  0.54687707  0.07227003  -5.002004  -4.7913624
## [4,]  0.03117615  0.00722321  0.07227003  0.01176089  -0.689560  -0.6427715
## [5,] -7.07996685 -2.63395382 -5.00200430 -0.68955998 500.000000 306.0109125
## [6,] -3.43375539 -1.02389753 -4.79136241 -0.64277150 306.010913 297.3979642
## [7,] -0.40152336 -0.06064338 -2.70274958 -0.26836818 273.315303 272.7821026
##              [,7]
## [1,]  -0.40152336
```

```
## [2,]  -0.06064338
## [3,]  -2.70274958
## [4,]  -0.26836818
## [5,] 273.31530341
## [6,] 272.78210255
## [7,] 266.33972150
```

Compare the optimized parameter estimates.

```
optim_fit_2_a$par
```

```
## [1] -3.2675051 12.6930303 -3.2675051 12.6930303  0.5869714  0.7669092  0.7669092
```

```
optim_fit_2_b$par
```

```
## [1]   6.0007361 -16.3217346   3.6115880 -24.6329881   2.1266423  -0.8220143
## [7]  -0.7170734
```

No, they're not the same. Because different initial guesses will cause distinctive SSE, and the optim function is aiming to minimize SSE, so that the β parameters are changing depends on how low the SSE is in each case.

Problem 2

Fitting the neural network with `optim()` is convenient because it allows us to focus on the model and the objective function. However, because `optim()` relies on finite differences to estimate the gradient it does not scale well to large neural networks with many hidden units, and thus many parameters. The backpropagation equations allow us to overcome this limitation. Analytic expressions for the partial derivatives can be written in relatively simple form using the output error, $\delta_n = y_n - f_n$, and the hidden unit error.

In this problem you will write the backpropagation equations to allow efficiently calculating the gradient vector for all parameters. You will use notation consistent with that from lecture where the squared error (or squared residual) of the n-th observation was denoted as R_n.

2a)

Let's start with the partial first derivatives of SSE with respect to the output layer parameters, α_0 and $\boldsymbol{\alpha}$.

PROBLEM Write the expression for the partial first derivative of the n-th squared residual with respect to the α_k parameter in terms of the n-th "delta" and the n-th observation's k-th hidden unit transformed repsonse, $h_{n,k}$.

SOLUTION

$$\frac{\partial R_n}{\partial \alpha_k} = -2\delta_n h_{n,k}$$

2b)

The backprogagation equations discussed in lecure were written in terms of the individual observations and individual parameters. However, when we discussed the gradient vector associated with linear and generalized linear models, we wrote out the general matrix/vector notation for the gradient vector. The first aspect of the matrix/vector notation was to sum over all observations, for each. The "complete" derivatives for all parameters were then "stacked" together to assemble the gradient vector. To be consistent with the earlier lectures on linear and generalized linear models, you will account for all observations in your derivations. Thus, you will be working with "batch mode", where all observations are considered at each optimization iteration.

Let's go through those same steps in the context of the backpropagation equations, starting with the output layer. The "complete" derivative is the summation over all N observations. Specifically, for the k-th output layer parameter:

$$\frac{\partial}{\partial \alpha_k}(SSE) = \sum_{n=1}^{N}\left(\frac{\partial R_n}{\partial \alpha_k}\right)$$

You will write out the above summation in matrix/vector notation, based on your response to Problem 2a). In your response, if you subset the $\mathbf{H}$ matrix by rows, denote a single row as $\mathbf{h}_{n,:}$. However, if you subset the $\mathbf{H}$ matrix by columns, denote a single column as $\mathbf{h}_{:,k}$. Denote the "delta" vector of errors as $\boldsymbol{\delta}$. As an additional hint, the output layer is a linear model relative to the hidden unit transformed responses. Thus, the earlier lectures which stepped through the gradient vector derivations for linear and generalized linear models may be helpful.

PROBLEM Write the summation over all N observations of the partial first derivative of the SSE with respect to the α_k parameter in matrix/vector notation.

SOLUTION

$$\frac{\partial}{\partial \alpha_k}(SSE) = \sum_{n=1}^{N}(\frac{\partial R_n}{\partial \alpha_k}) = \sum_{n=1}^{N}(-2\delta_n h_{n,k}) = -2\boldsymbol{\delta}^T\mathbf{h}_{:,k}$$

2c)

The next step is to "stack" all output layer parameter partial derivatives together into a single vector. This stacking operation can be efficiently written in terms of the matrix of hidden unit transformed responses, $\mathbf{H}$, and the "delta" vector, $\boldsymbol{\delta}$. However, we have to be careful about the dimensionality of this result.

PROBLEM Using your result from Problem 2b), write the vector of "complete" derivatives using and $\boldsymbol{\delta}$. What is the dimensionality of this vector? Are you missing an output layer parameter? If so, which one?

SOLUTION

$$\frac{\partial}{\partial \alpha}(SSE) = -2\mathbf{H}^T\boldsymbol{\delta}$$

The dimensionality of this vector is H X 1. Yes. It misses α_0.

2d)

You should have determined that one of the output layer parameters is missing from your result in Problem 2c). How can you account for that missing parameter in gradient vector? Write the complete gradient with respect to the output layer parameters as $\mathbf{g}_\alpha$.

PROBLEM Account for the missing output layer parameter from Problem 2c) and write out the complete vector of partial derivatives of the SSE with respect to the output layer parameters, $\mathbf{g}_\alpha$.

SOLUTION

$$\mathbf{g}_\alpha = \{\frac{\partial}{\partial \alpha_k}(SSE)\}_{k=0}^{H}$$

2e)

With the output layer parameters accounted for, it's time to move into the hidden layer. The derivatives of the squared residual, R_n, with respect to the hidden unit parameters, β_{dk}, can be written in terms of the "hidden unit error", $\xi_{n,k}$. The hidden unit error backpropagates the output error, δ_n, to each hidden unit.

The hidden unit error for the n-th observation and k-th hidden unit is written in terms of the output layer weight, α_k, the derivative of the non-linear transformation, $g'_{n,k}$, and the output error, δ_n:

$$\xi_{n,k} = \alpha_k g'_{n,k} \delta_n$$

PROBLEM Write the expression for the partial first derivative of the n-th squared residual, R_n, with respect to the β_{dk} parameter, in terms of $\xi_{n,k}$ and the input $x_{n,d}$.

SOLUTION

$$\frac{\partial R_n}{\partial \beta_{dk}} = -2\xi_{n,k} x_{n,d}$$

2f)

Writing the gradient of the SSE with respect to the hidden unit parameters in matrix form is a little tricky. Consider that the (n, k) subscript on the hidden unit error, ξ, corresponds to the (n, k) element of the $N \times H$ matrix $\mathbf{E}$. Each column in $\mathbf{E}$ corresponds to the errors associated with each hidden unit. The matrix $\mathbf{E}$ can be written as a series of element-wise products rather than matrix products. The element-wise or element-by-element product is also known as the Hadamard product and is denoted sometimes as an open circle, $\circ$. The $N \times H$ matrix of all hidden unit errors can be written as:

$$\mathbf{E} = \mathbf{A}_W \circ \mathbf{G}_p \circ \boldsymbol{\delta}$$

$\mathbf{A}_W$ is the $N \times H$ matrix of output layer weights where each row corresponds to the transpose of the $\boldsymbol{\alpha}$ vector:

$$\mathbf{A}_W = \begin{bmatrix} \boldsymbol{\alpha}^T \\ \boldsymbol{\alpha}^T \\ \vdots \\ \boldsymbol{\alpha}^T \end{bmatrix}$$

The $\mathbf{G}_p$ matrix is the matrix of non-linear transformation derivatives. Thus, the n-th row and k-th column of $\mathbf{G}_p$ is equal to:

$$\mathbf{G}_p\left(n,k\right) = g'_{n,k}$$

The element-by-element multiplication therefore multiplies each element in $\mathbf{A}_W$ by each element in $\mathbf{G}_p$. The element-by-element multiplication then "broadcasts" the $\boldsymbol{\delta}$ vector to every column in the $(\mathbf{A}_W \circ \mathbf{G}_p)$ matrix. This may sound complex, but the `*` multiplication operator in `R` performs element-by-element multiplication (if you have experience with MATLAB the `.*` operator performs element-by-element multiplication).

Within this assignment, you will be responsible for creating the $\mathbf{G}_p$ matrix, the rest of the $\mathbf{E}$ matrix will be provided to you. We have been working with the logistic function as the non-linear transformation applied to each hidden unit, $g\left(\eta\right) = \text{logit}^{-1}\left(\eta\right)$. Your task will be to write out the expression for the derivative of the non-linear transformation $g'_{n,k}$ and assemble the $\mathbf{G}_p$ matrix.

PROBLEM Write the expression for $g'_{n,k}$ assuming the non-linear transformation function is the logistic function. Then write the expression for the $N \times H$ matrix $\mathbf{G}_p$.

HINT: You should be able to write $\mathbf{G}_p$ in terms of the $\mathbf{H}$ matrix and the element-by-element operator, $\circ$.

SOLUTION

$$g'_{n,k} = \frac{\partial}{\partial \eta_{n,k}}(g(\eta_{n,k})) = \frac{\exp(-\eta_{n,k})}{(1+\exp(-\eta_{n,k}))^2} = h_{n,k}(1-h_{n,k})$$

$$\mathbf{G}_p = \mathbf{H} \circ (1 - \mathbf{H})$$

2g)

With the $\mathbf{E}$ matrix defined, it is now possible to define the "complete" derivative of each hidden unit parameter with matrix notation. You will build from your solution in Problem 2e) to define the $(D+1) \times H$ matrix $\mathbf{J}$. The d-th row and k-th column of the matrix $\mathbf{J}$ contains the partial first derivative of the SSE with respect to each hidden unit parameter:

$$\mathbf{J}\left(d,k\right) = \frac{\partial}{\partial \beta_{dk}}\left(SSE\right)$$

The first column $\mathbf{J}$ contains the derivatives of the SSE with respect to the first hidden unit parameters. The second column contains the derivatives of the SSE with respect to the second hidden unit parameters, and so on. Once the $\mathbf{J}$ matrix is calculated the gradient with respect to the hidden unit parameters can be assembled by stacking each column of $\mathbf{J}$ on top of each other. This stacking operation is usually denoted by the $\text{vec}\left(\cdot\right)$ operation. Thus, the gradient of the SSE with respect to all β-parameters is:

$$\mathbf{g}_\beta = \text{vec}\left(\mathbf{J}\right)$$

The stacking operation can be performed using the `as.vector()` function in `R`. Your task is to therefore define the matrix operation to calculate the $\mathbf{J}$ matrix using the design matrix, $\mathbf{X}$, and the hidden unit error matrix $\mathbf{E}$.

PROBLEM Write the expression to calculate the J matrix in terms of X and E.

HINT: Check the matrix dimensions!

SOLUTION

$$\mathbf{J} = -2\mathbf{X}^T\mathbf{E}$$

Problem 3

You now have all of the expressions necessary to perform the backpropagation method to calculate the gradient vector accounting for all N observations in the training set. It's time to program them into a function to use within `optim()` so you do not have to rely on finite differences!

3a)

The backpropagation equations rely on calculating the derivative of the non-linear transformation function. For this assignment you are using the logistic function as the "activation function". You wrote out the expression for the derivative in matrix notation in Problem 2f). You will now program that expression into the `logistic_deriv()` function. Once you complete the function, it is added to the `info_two_units` list as the `$transform_deriv` variable for you.

PROBLEM Complete the code chunk below by programming the derivative of the logistic function. Assume that the input argument, `H`, is the matrix of all hidden unit responses. You are therefore calculting the $\mathbf{G}_p$ matrix.

HINT: Remember that the element-by-element product, *circ*, is the `*` operator in `R`.

SOLUTION Complete the `logistic_deriv()` function.

```
logistic_deriv <- function(H)
{
  H * (1-H)
}
```

The `logistic_deriv()` function is added to the `info_two_units` list for you.

```
info_two_units$transform_deriv = logistic_deriv
```

3b)

The `backprop_grad_batch()` function is defined in the code chunk below. It has the same input argument structure as `my_neuralnet_see()`, accepting just two input arguments. The vector `theta` of all unknown parameters and the `my_info` list of required information. The first portion of the of the function performs the same book keeping steps to reshape the parameters into the correct format, just as was done in `my_neuralnet_sse()`.

You must complete the remaining steps in the function, which are broken into the forward and backward steps. In the forward step you must calculate the neural network output given the inputs and parameters. In the backward step, you must calculate the derivatives associated with the output layer parameters and then the hidden unit parameters, based on the output error and hidden unit error.

Several portions of the backward step are completed for you. The $\mathbf{E}$ matrix is calculated for you, and the "stacking" operation of the columns of the $\mathbf{J}$ matrix into the $\mathbf{g}_\beta$ vector is performed for you. Lastly, the $\mathbf{g}_\beta$ and $\mathbf{g}_\alpha$ vectors are combined for you in the correct order.

PROBLEM Complete the `backprop_grad_batch()` function in the code chunk below. The comments describe what you need to calculate in each line of code.

HINT: The forward step should look familiar...

```r
backprop_grad_batch <- function(theta, my_info)
{
  # extract the hidden unit parameters
  X <- my_info$design_matrix
  length_beta <- ncol(X)
  total_num_betas <- my_info$num_hidden * length_beta
  beta_vec <- theta[1:total_num_betas]

  # extract the output layer parameters
  a_all <- theta[(total_num_betas + 1):length(theta)]

  # reorganize the beta parameters into a matrix
  Bmat <- matrix(beta_vec, nrow = length_beta, byrow = FALSE)

  # reorganize the output layer parameters by extracting
  # the output layer intercept (the bias)
  a0 <- a_all[1] # the first element in a_all is the bias
  aw <- a_all[-1] # select all EXCEPT the first element

  # - # forward step # - #

  # calculate the linear predictors associated with
  # each hidden unit
  A <- X %*% Bmat

  # pass through the non-linear transformation function
  H <- my_info$transform_hidden(A)

  # calculate the response
  f <- as.vector(a0 + H %*% as.matrix(aw))

  # - # backward step # - #

  # calculate the output error - the delta
  d <- my_info$yobs - f

  # calculate the gradient with respect to the output layer
  # parameters (include the output layer intercept - bias)
  da <- -2 * t(cbind(1, H)) %*% d


  # calculate derivative of the transformation function
  Gp <- my_info$transform_deriv(H)

  # reshape output layer weights
  AW <- matrix(rep(aw, nrow(X)), nrow(X), byrow=TRUE)

  # calculate hidden unit error
  E <- AW * Gp * d

  # calculate the J matrix
  J <-  -2 * t(X) %*% E
```

```
  # stack all columns into the gradient vector
  db <- as.vector( J )


  # package all parameters together, `as.vector()`
  # is applied to `da` to modify the data type and
  # make sure some attributes are consistent
  c(db, as.vector(da))
}
```

SOLUTION

3c)

Let's now test that your gradient function is working correctly. Execute the `backprop_grad_batch()` function with a guess of all 0's for all parameters. If your function is correct you should see the following printed to screen:

```
0.0000    0.0000    0.0000    0.0000 -615.7961 -307.8981 -307.8981
```

The `theta` vector is organized such that all hidden unit parameters are listed first. As shown above, all of the derivatives with respect to the β parameters are zero. Why is that?

PROBLEM Execute the `backprop_grad_batch()` function by setting all parameters equal to zero. If your result matches the provided answer, why do you think the derivatives of the SSE with respect to the hidden unit parameters are equal to zero?

HINT: Remember that the `backprop_grad_batch()` function requires the list of required information, just like the `my_neuralnet_sse()` function.

SOLUTION The backpropagation method is applied in the code chunk below.

```
backprop_grad_batch((rep(0, info_two_units$num_params)), info_two_units )
```

```
## [1]    0.0000    0.0000    0.0000    0.0000 -615.7961 -307.8981 -307.8981
```

Mathmetically, initial guesses are all 0, then $\mathbf{A}_W$ is a vector full of 0, so that $\mathbf{E}$ contains 0 and the derivative $\mathbf{J}$ is also a vector of 0. Eventually, β parameters are all 0.

3d)

It's now time to use the analytic expression for the gradient within the optimization algorithm. In Problem 1g) you used `optim()` with `gr` set to for the two initial guesses. You will use the initial guess of all 0's, but this time tell `optim()` to use your `backprop_grad_batch()` function instead of estimating the gradient with finite differences. To do that, you will set the `gr` argument in the `optim()` call to be `backprop_grad_batch`. Note that the `()` are not included because you are assigning the function object to the `gr` variable.

PROBLEM **Complete the code chunk below in order to fit the two hidden unit neural network using the analytic expressions for the gradient. Use the same arguments you used in Problem 1g), except set the `gr` argument equal to `backprop_grad_batch`. Use an initial guess of 0 for all parameters.**

After fitting, print out the identified optimal parameters contained in the `$par` field. How do the values compare to those identified in `optim_fit_2_a` which used finite differences to estimate the gradient?

SOLUTION The optimization is performed in the code chunk below.

```
optim_backprop_2_a <- optim(rep(0, info_two_units$num_params),
                        my_neuralnet_sse ,
                        gr = backprop_grad_batch,
                        info_two_units,
                        method = "BFGS",
                        hessian = TRUE,
                        control = list(maxit = 5000))
optim_backprop_2_a
```

```
## $par
## [1] -3.2675049 12.6930296 -3.2675049 12.6930296  0.5869713  0.7669092  0.7669092
##
## $value
## [1] 36.487
##
## $counts
## function gradient
##       48       24
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##           [,1]       [,2]      [,3]       [,4]       [,5]       [,6]       [,7]
## [1,] 1.0313133 0.21674059 1.1329695 0.27882722   9.204326   4.296309   4.296247
## [2,] 0.2167406 0.06466816 0.2788272 0.07824525   2.253818   1.446623   1.446592
## [3,] 1.1329695 0.27882722 1.0313133 0.21674059   9.204326   4.296247   4.296309
## [4,] 0.2788272 0.07824525 0.2167406 0.06466816   2.253818   1.446592   1.446623
## [5,] 9.2043263 2.25381839 9.2043263 2.25381839 500.000000 210.136005 210.136005
## [6,] 4.2963091 1.44662341 4.2962469 1.44659215 210.136005 198.134159 198.134159
## [7,] 4.2962469 1.44659215 4.2963092 1.44662341 210.136005 198.134159 198.134159
```

What are the optimal parameter estimates, and how do they compare to the previous case based on finite differences?

```
optim_backprop_2_a$par
```

```
## [1] -3.2675049 12.6930296 -3.2675049 12.6930296  0.5869713  0.7669092  0.7669092
```

```
optim_fit_2_a$par
```

```
## [1] -3.2675051 12.6930303 -3.2675051 12.6930303  0.5869714  0.7669092  0.7669092
```

They're the same.

3e)

Let's now fit a more complicated neural network with more hidden units to see the speed benefits of the analytic gradient. You will fit a neural network with 11 hidden units. To do so, you must create a new list of required information. Do you need to create a new design matrix since you are changing the number of hidden units?

A few of the variables are specified for you below. Namely, the function variables are assigned to be the same functions used before.

PROBLEM Create the list of required information to fit a neural network with 12 hidden units. After creating the list, you must calculate the total number of parameters in the model. How many parameters are there compared to the model with just 2 hidden units?

Did you need to change the design matrix in order to change the number of hidden units?

SOLUTION The list of required information is specified below.

```
info_11_units <- list(
  yobs = train_data$y,
  design_matrix = Xmat,
  num_hidden = 11,
  transform_hidden = boot::inv.logit
)

info_11_units$transform_deriv <- logistic_deriv

info_11_units$num_params <- info_11_units$num_hidden * ncol(info_11_units$design_matrix) +
  info_11_units$num_hidden + 1

info_11_units$num_params
```

```
## [1] 34
```

How many parameters are in the model with 11 units?
34
Do we need to define a new design matrix?
No, no matter how many hidden layer it has, the design matrix remains the same while transforming from input to the first hidden layer.

3f)

Before fitting the model with the analytic backpropagation gradient, let's have `optim()` estimate the gradient with finite differences. We will calculate the execution time and then compare this to the case that uses the

backpropagation function. You will notice in the code chunk below that there are two lines of code wrapped around the `optim()` call that assign results from `Sys.time()` to variables. The `Sys.time()` function returns the current date timestamp when the function is called. This is a very simple approach to evaluating the execution time of functions (analogous to using `tic` and `toc` in MATALB). There are more formal approaches to calculating execution times. If you are interested, please see the following R-bloggers post which provides a nice introduction to the `microbenchmark` package.

You will use an initial guess of 0 for all parameters.

PROBLEM Complete the code chunk below which fits the neural network using finite differences. You should therefore use all of the same `optim()` settings as you used in Problem 1g), except pass in the `info_11_units` list, instead of the `info_two_units` list. Set the initial guess to be zero for all parameters.

SOLUTION The 11 hidden unit neural network is fit in the code chunk below. Notice that the execution time is assigned to the `optim_fd_runtime` variable.

```
start_optim_clock <- Sys.time()
optim_fit_11 <- optim(rep(0, info_11_units$num_params),
                      my_neuralnet_sse ,
                      gr = NULL,
                      info_11_units,
                      method = "BFGS",
                      hessian = TRUE,
                     control = list(maxit = 1e4))
end_optim_clock <- Sys.time()

optim_fd_runtime <- end_optim_clock - start_optim_clock
```

With 11 hidden units, the `optim()` call took:

```
optim_fd_runtime
```

```
## Time difference of 9.113161 secs
```

3g)

Now fit the 11 hidden unit neural network using the analytic backpropagation function for the gradient. The execution time is also estimated via the simple `Sys.time()` calls in the code chunk below. Use the same arguments as in Problem 3f), except set the `gr` argument to be `backprop_grad_batch`.

Unfortunately, you cannot directly compare the final parameter estimates between the two cases. With 11 hidden units, the error from the finite differences will cause the optimal parameter estimates to be different even with the same starting guess! Neural network solutions can be that sensitive to changes in the gradient!

PROBLEM Complete the code chunk below to fit the neural network with the analytic gradient function. How does the run time compare to the case that used finite differences?

SOLUTION The 11 hidden unit neural network is fit in the code chunk below using the analytic gradient function. The execution time is assigned to the `backprop_runtime` variable.

```
start_backprop_clock <- Sys.time()
backprop_fit_11 <- optim(rep(0, info_11_units$num_params),
                         my_neuralnet_sse ,
                         gr = backprop_grad_batch,
                         info_11_units,
                         method = "BFGS",
                         hessian = TRUE,
                           control = list(maxit = 1e4))
end_backprop_clock <- Sys.time()

optim_backprop_runtime <- end_backprop_clock - start_backprop_clock
```

With 11 hidden units and the analytic gradient function, the `optim()` call took:

```
optim_backprop_runtime
```

```
## Time difference of 1.187952 secs
```

It comes much faster than the case that used finite differences.

Problem 4

Now that you have gone through and fit a neural network with the backpropagation method on your own, let's practice using existing libraries. Afterall, in the implemention used in Problem 3, we did not try and tune the number of hidden units. We simply fit two different size neural networks. You will practice using the `caret` package to tune a neural network with k-fold cross-validation. The code chunk below loads in the `caret` package. If you have not downloaded and installed `caret` please do so before running the code chunk below.

Note that the code chunk below as the `eval` flag set to FALSE by default.

```
library(caret)
```

4a)

The first step in `caret` is to define the resampling scheme. You will use 5-fold cross-validation to assess the generalization performance of the model for various numbers of hidden units. To do so, set the `method` argument equal to `"cv"`, the `number` argument equal to 5. By default, `caret` simply chooses the model with the absolute lowest cross-validation error, averaged over the hold-out sets.

PROBLEM Complete the code chunk below by specifying to use 5-fold cross-validation within the `trainControl()` function.

```
my_ctrl <- trainControl(method = "cv", number = 5)
```

SOLUTION

4b)

You will now perform 5-fold cross-validation to tune and fit a neural network model with the `neuralnet` package. If you have not downloaded and installed the `neuralnet` package, `caret` will prompt you to do so, or you can download it before running the code chunk below.

The primary function in `caret` is the `train()` function. It is where you specify the formula, the type of model (via the `method` argument), the primary performance metric (via the `metric` argument), as well as a custom tuning grid, if you will pre-process the inputs, and your resampling scheme. You will model the response `y` as a function of the input `x`. You will use the `neuralnet` package, so you must set the `method` argument equal to `"neuralnet"`. Specify the metric to be `"RMSE"`, which instructs `caret` to assess the cross-validation performance based on minimizing the RMSE. Although you do not need to preprocess this single input setting it will be useful to practice doing so. Thus, you must specify the `preProc` argument to be `c("center", "scale")` to instruct `caret` to standardize the input. You must also set the `data` argument to `train_data` in order to tell `caret` what data set to use. Lastly, specify the `trControl` argument to be `my_ctrl`.

With the above arguments specified, `caret` will use 5-fold cross-validation to identify the best neural network based upon a default number of hidden units to try.

PROBLEM Train the neural network using the specifications given the problem statement. How many observations were used to TEST the model in each fold? What was the model with the best cross-validation performance? What is the R-squared associated with that model?

```
set.seed(1101)
fit_nnet_1 <- train(y ~ x ,
                    data = train_data,
                    method = "neuralnet",
                    metric = "RMSE",
                    preProc = c("center", "scale"),
                    trControl = my_ctrl)
```

SOLUTION Display the results to the screen below.

```
fit_nnet_1
```

```
## Neural Network
##
## 250 samples
##   1 predictor
##
## Pre-processing: centered (1), scaled (1)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 199, 201, 201, 199, 200
## Resampling results across tuning parameters:
##
##   layer1  RMSE        Rsquared   MAE
##   1       0.38090027  0.7825077  0.27790691
##   3       0.09422233  0.9875338  0.07267683
##   5       0.08333349  0.9906381  0.06527242
##
```

```
## Tuning parameter 'layer2' was held constant at a value of 0
## Tuning
##  parameter 'layer3' was held constant at a value of 0
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were layer1 = 5, layer2 = 0 and layer3 = 0.
```

Which one is the best?
Based on RMSE, the model with layer1 = 5, layer2 = 0 and lyaer3 = 0 is the best.

For each fold, 51, 49, 49, 51, 50 observations were used to TEST the model. The model with layer1 = 5, layer2 = 0 and lyaer3 = 0 with the best cross-validation performance. The R-squared associated with this model is 0.9906381.

4c)

By default, `caret` tries out just a few hidden units in the first layer and does not use any additional layers. Let's now try out a custom tuning grid to consider more hidden units in the first layer, as well see what happens if we use a second hidden layer. You must use the `expand.grid()` function to create the tuning grid in the correct format for `caret`. Specify the custom grid in the first code chunk below, and then use that grid by setting it equal to the `tuneGrid` argument in the `train()` call in the second code chunk.

PROBLEM Complete the first code chunk by setting the `layer1` variable within the `expand.grid()` equal to a vector of evenly spaced values from 2 to 14 in increments of 3. Thus, you will consider the first layer has between 2 to 14 hidden units. Set the `layer2` variable equal to an evenly spaced vector from 0 to 6 in increments of 2. Thus, you will consider 0 up to 6 hidden units in the second layer. Lastly, set the `layer3` variable also equal to 0. You are therefore considering up to 2 hidden layers, and trying out a total of 20 different neural networks.

Once the grid is specified, train the neural network using 5-fold cross-validation. Use the same arguments as you did in Problem 4b), except set `tuneGrid` equal to `nnet_grid`.

What is the best model identified from cross-validation?

SOLUTION Set the grid of hidden units to use.

```
nnet_grid <- expand.grid(layer1 = seq(2, 14, by = 3),
                         layer2 = seq(0, 6, by = 2),
                         layer3 = 0)
```

Train the model below. Note that this code chunk may take a few minutes to complete, depending on your machine.

```
set.seed(1101)
start_train_clock <- Sys.time()
fit_nnet_cv <- train(y ~ x ,
                     data = train_data,
                     method = "neuralnet",
                     metric = "RMSE",
                     tuneGrid = nnet_grid,
                     preProc = c("center", "scale"),
                     trControl = my_ctrl)
end_train_clock <- Sys.time()
```

```
train_runtime <- end_train_clock - start_train_clock
```

The 5-fold cross-validation of all 20 models completed in:

```
train_runtime
```

```
## Time difference of 51.67304 secs
```

Print out the cross-validation results.

```
fit_nnet_cv
```

```
## Neural Network
##
## 250 samples
##   1 predictor
##
## Pre-processing: centered (1), scaled (1)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 199, 201, 201, 199, 200
## Resampling results across tuning parameters:
##
##   layer1  layer2  RMSE        Rsquared   MAE
##    2      0       0.18785439  0.9236173  0.13788116
##    2      2       0.12427418  0.9767637  0.09236931
##    2      4       0.10965926  0.9827222  0.08564834
##    2      6       0.09176104  0.9877800  0.07091550
##    5      0       0.08357173  0.9906444  0.06615041
##    5      2       0.16603816  0.9314878  0.11747956
##    5      4       0.08317209  0.9907521  0.06629600
##    5      6       0.08481069  0.9902905  0.06709372
##    8      0       0.08349155  0.9905762  0.06535769
##    8      2       0.08353270  0.9906616  0.06595258
##    8      4       0.08247649  0.9908709  0.06498268
##    8      6       0.08353055  0.9906162  0.06628457
##   11      0       0.08421048  0.9903707  0.06669029
##   11      2       0.15691027  0.9345229  0.11075556
##   11      4       0.08197226  0.9910953  0.06442445
##   11      6       0.08244397  0.9908314  0.06532932
##   14      0       0.08193845  0.9909712  0.06497866
##   14      2       0.08314986  0.9907505  0.06592810
##   14      4       0.08435557  0.9903999  0.06681794
##   14      6       0.08356427  0.9904673  0.06606510
##
## Tuning parameter 'layer3' was held constant at a value of 0
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were layer1 = 14, layer2 = 0 and layer3 = 0.
```

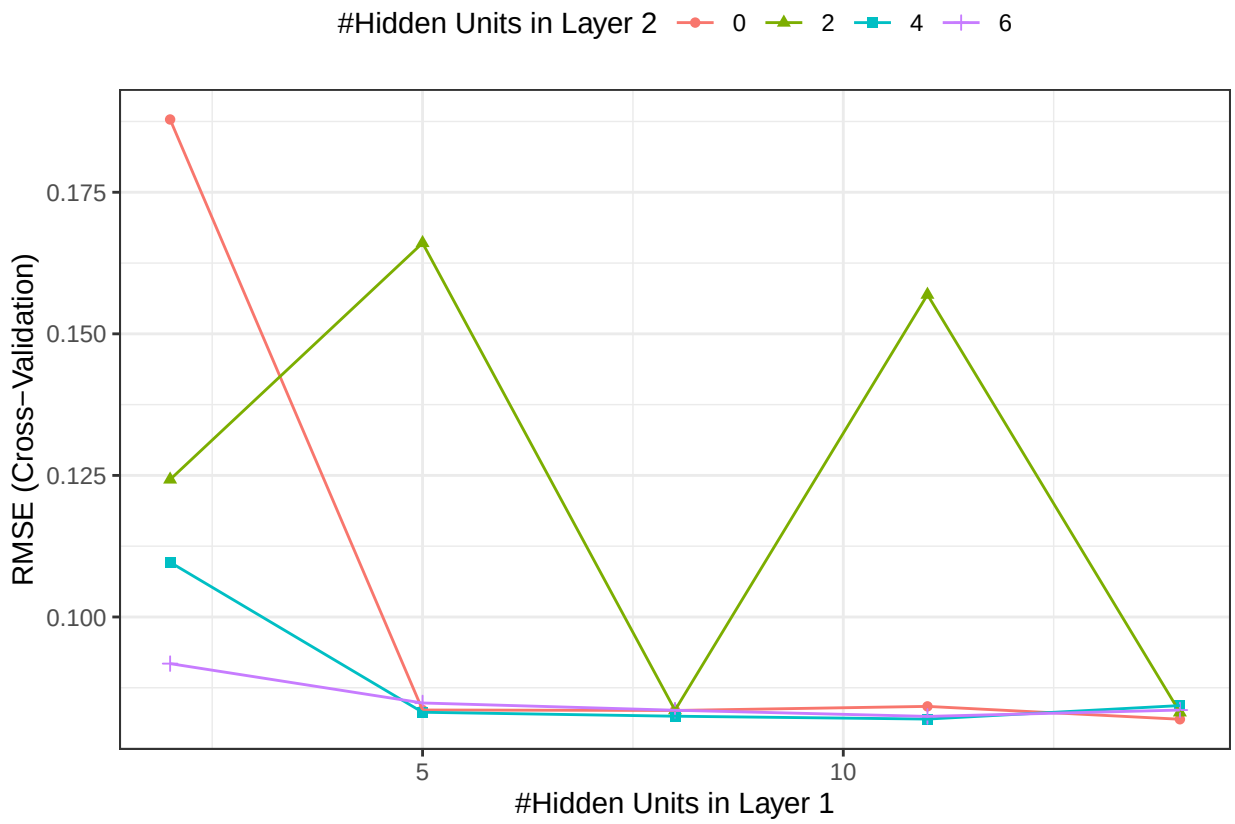
What's the best model?
According to the RMSE, the model that has layer1 = 14, layer2 = 0, layer3 = 0 is the best.

4d)

The two code chunks below are completed for you. The first uses the default plot method in `ggplot` for the `caret` model object to plot the cross-validation RMSE results for each combination of `layer1` and `layer2`. The second code chunk uses the cross-validation results and the estimated standard errors to include the variability of the RMSE estimate averaged over the folds.

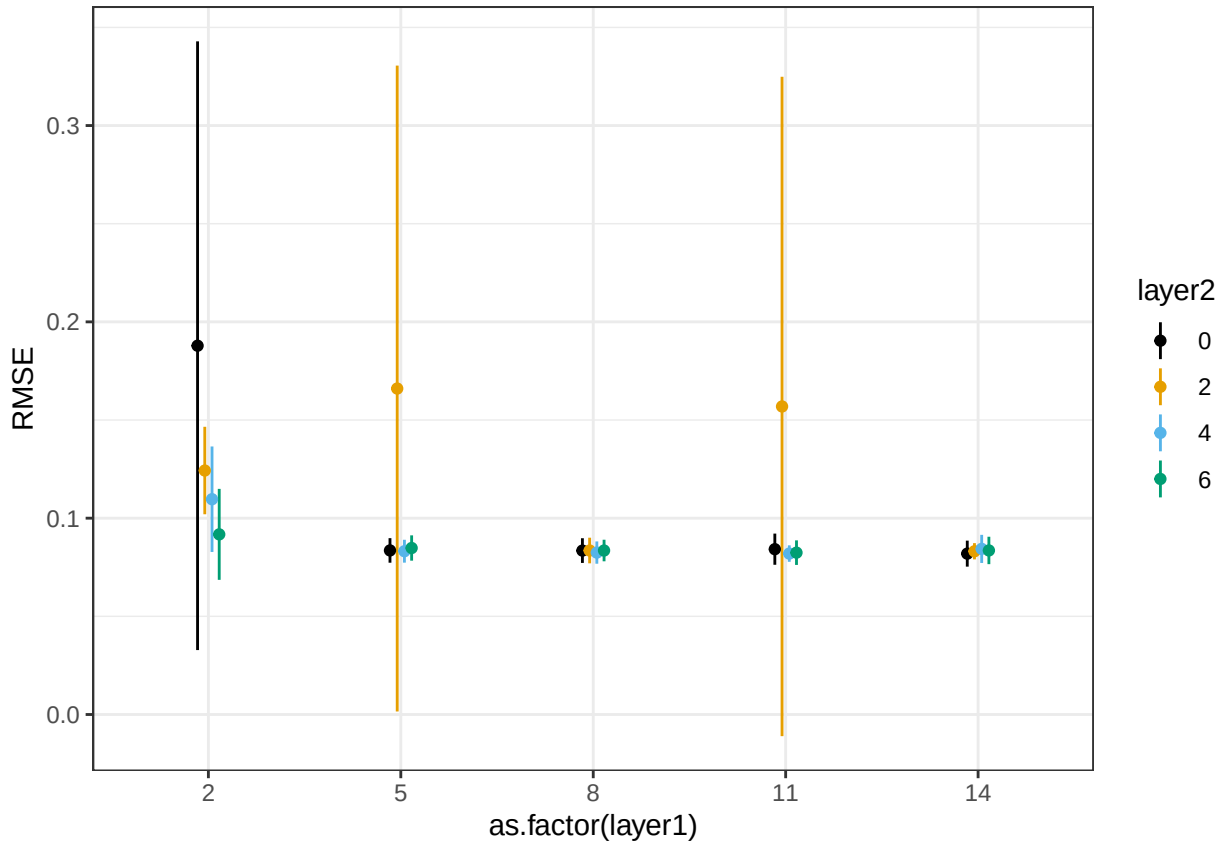
The default plot method.

```
ggplot(fit_nnet_cv) + theme_bw() + theme(legend.position = "top")
```



The custom plot with the standard errors.

```
fit_nnet_cv$results %>% tbl_df() %>%
  ggplot(mapping = aes(x = as.factor(layer1))) +
  geom_linerange(mapping = aes(group = layer2,
                               color = as.factor(layer2),
                               ymin = RMSE - RMSESD,
                               ymax = RMSE + RMSESD),
                 position = position_dodge(0.15)) +
  geom_point(mapping = aes(group = layer2,
                           color = as.factor(layer2),
                           y = RMSE),
             position = position_dodge(0.15)) +
  ggthemes::scale_color_colorblind("layer2") +
  theme_bw()
```



PROBLEM **In class we discussed the "one-standard error rule" for selecting between models with similar cross-validation performance. Based on that concept, which model would you consider to be "good enough"?**

SOLUTION layer1 = 8 or 14, either causes good RMSE results for all possible hidden units in layer2. In addition, layer1 = 5 or 11, if the hidden unit is not 2, then the RMSE results seem also very good.