CS 1675 Spring 2022 Homework: 10

Assigned April 7, 2022; Due: April 14, 2022

Sameera Boppana

Submission time: April 14, 2022 at 11:00PM EST

Collaborators

Include the names of your collaborators here.

Jeffery Janotka

Overview

This assignment focuses on the architecture of single hidden layer feedforward neural networks. You will practice calculating hidden units and neural network responses for a regression task. You will gain experience understanding the interaction between the hidden unit and output layer parameters. You will fit a regression neural network to data by minimizing the sum of squared errors (SSE), and tune the number of hidden units via a hold-out test set. Lastly, you will use `caret` to manage the resampling and tuning of a neural network for you.

IMPORTANT: code chunks are created for you. Each code chunk has `eval=FALSE` set in the chunk options. You **MUST** change it to be `eval=TRUE` in order for the code chunks to be evaluated when rendering the document.

You are allowed to add as many code chunks as you see fit to answer the questions.

Load packages

This assignment will use packages from the `tidyverse` suite.

```
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──
```

```
## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.1.6     ✓ dplyr   1.0.8
## ✓ tidyr   1.2.0     ✓ stringr 1.4.0
## ✓ readr   2.1.2     ✓ forcats 0.5.1
```

```
## Warning: package 'tidyr' was built under R version 4.0.5
```

```
## Warning: package 'readr' was built under R version 4.0.5
```

```
## Warning: package 'dplyr' was built under R version 4.0.5
```

```
## ── Conflicts ─────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

This assignment also uses the `scale_color_colorblind()` function from the `ggthemes` package. If you do not have `ggthemes` already installed please type `install.packages("ggthemes")` into the `R` console, or use the RStudio package installer GUI. You only need to run that command **ONCE**. Once the package is installed, you do **not** need to run the command again.

Problem 01

In lecture we discussed that neural networks are just matrix multiplications. Hidden units are essentially transformed linear models. The output layer is a linear basis function model with the hidden units acting as the basis functions. In this problem, you will work through the various matrix calculations. Neural networks have already been fit to a data set and the the neural network parameters (weights and biases) are provided to you.

As practice, you will work with a noise-free toy problem. The response, f, is simply equal to $\sin(x)$. You will practice neural network calculations to approximate that functional relationship.

The toy data set is loaded for you in the code chunk below and a glimpse is printed to screen. To distinguish between noisy observations (that we typically work with) the response is named `f` in the data set. The input is named `x` as in previous homework assignments.

```
url_01 <- 'https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/10/hw
_10_prob_01_train.csv'

prob_01_df <- readr::read_csv(url_01, col_names = TRUE)
```

```
## Rows: 25 Columns: 2
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl (2): x, f
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
prob_01_df %>% glimpse()
```

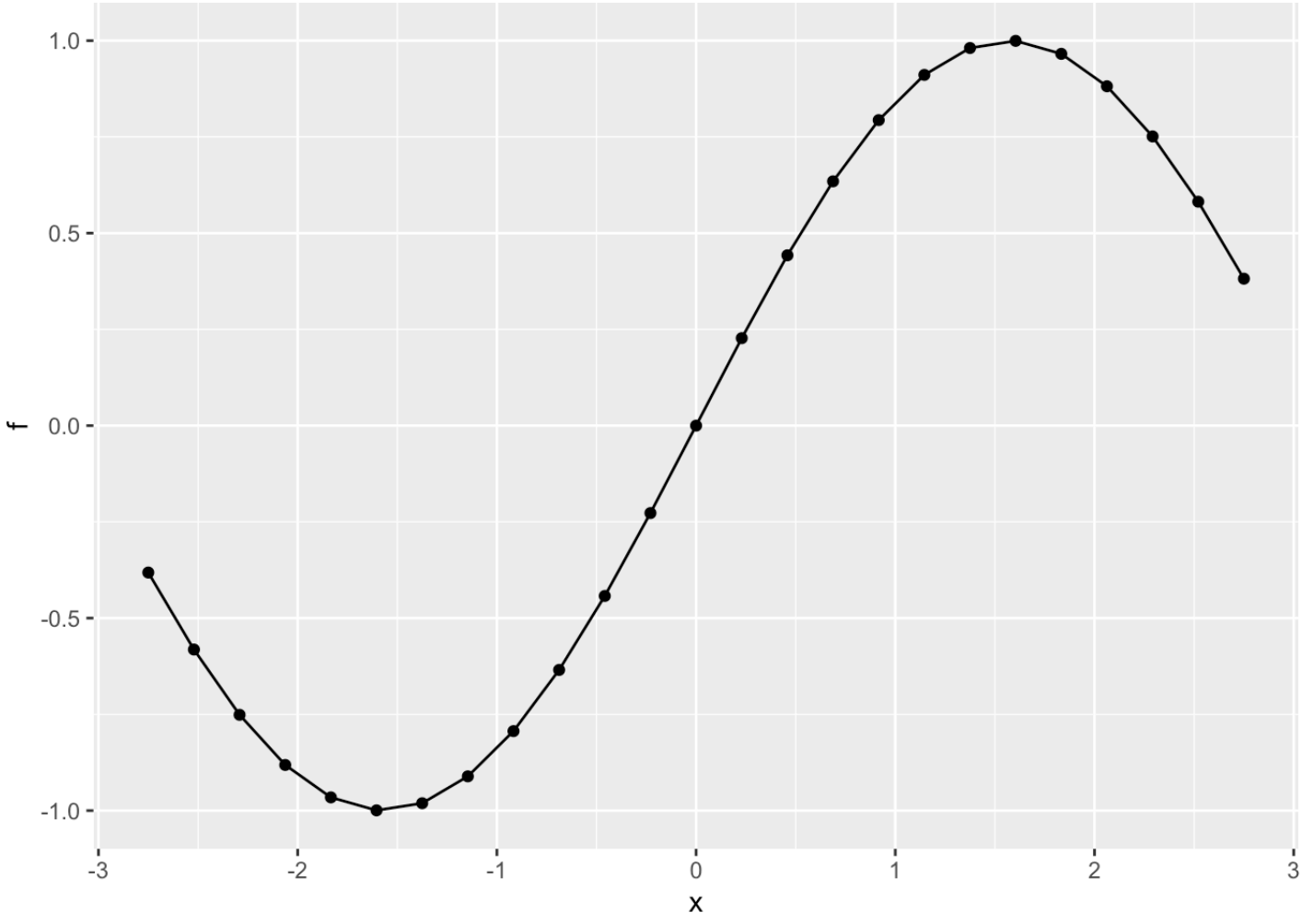
```
## Rows: 25
## Columns: 2
## $ x <dbl> -2.7500000, -2.5208333, -2.2916667, -2.0625000, -1.8333333, -1.60416…
## $ f <dbl> -0.3816610, -0.5816530, -0.7512316, -0.8815298, -0.9657347, -0.99944…
```

1a)

Plot `f` with respect to `x` with `ggplot()`. Since the response is noise-free, use `geom_line()` and `geom_point()` geoms to make "connect the dots".

SOLUTION

```
ggplot(mapping = aes(x = x, f), data = prob_01_df) + geom_point() + geom_line()
```



1b)

A set of neural network parameters are downloaded in the code chunk below. The parameters are stored in a list consisting of two elements. The first element named, `beta_matrix` is a matrix containing the β-parameters associated with the hidden units. The second, `alpha_vector` , is a "regular vector" containing the output layer parameters. The list is printed to screen for you.

```
url_load_dir <- 'https://github.com/jyurko/CS_1675_Spring_2022/blob/main/HW/10'

url_load_01_2a <- paste(paste(url_load_dir, "hw_10_prob_01_params_2a.rds", sep = "/")
,
                        "raw=true",
                        sep="?")

params_01_2a <- readr::read_rds( file = url_load_01_2a )

params_01_2a
```

```
## $beta_matrix
##            [,1]        [,2]
## [1,] 0.05470786 -0.02391862
## [2,] 0.46580905 -1.75883045
##
## $alpha_vector
## [1]  4.724008 -5.210946 -4.132372
```

The `$beta_matrix` is constructed such that each column corresponds to a separate *hidden unit* within the hidden layer. The particular neural network model associated with the `params_01_2a` parameters therefore has 2 hidden units.

Why does the `$beta_matrix` contain 2 rows and why does `$alpha_vector` consist of 3 elements? What does each β parameter correspond to relative to the hidden units?

SOLUTION

What do you think?

Each β parameter corresponds to the intercept of each of the two hidden layer weights/coefficients. The alpha_vector contains three elements because the output layer contains the number of hidden layers + 1 to account for the intercept as well. The beta parameters represent the hidden weights. The beta parameters are learned from the data.

1c)

The hidden units consist of two calculations. The first calculates the "linear predictors" based on the inputs and hidden unit parameters. The second is a non-linear transformation of the "linear predictors". We discussed in lecture that there are many possible non-linear transformation functions to use, but the logistic

function is a popular choice. Assume the inputs are stored in a design matrix $\mathbf{X}$, which includes a column of 1s, and the hidden unit parameters are stored in a matrix $\mathbf{B}$.

Write out the expressions for the linear predictor matrix $\mathbf{A}$ and the non-linear hidden unit values $\mathbf{H}$ assuming a logistic (inverse logit) function is used.

SOLUTION

Add as many equation blocks as you feel are necessary.

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

$$\mathbf{H} = \text{logit}^{-1}(\eta) = \frac{1}{1 + \exp(\eta)} = \frac{\exp(\eta)}{1 + \exp(\eta)}$$

1d)

You will now define a function which calculates the hidden unit "linear predictor" and non-linear values. The function is named `calc_hidden_units()` and it has three input arguments. The first, `X`, is the input design matrix, the second, `B`, is the hidden unit parameter matrix, and the third is the non-linear transformation function. The transformation function argument is named `g` to be consistent with the lecture notation of $g(\cdot)$.

This function is general and therefore allows passing in an arbitrary non-linear transformation function.

Complete the code chunk below. Calculate the linear predictor matrix `A` and the non-linear transformed hidden unit matrix `H`. The results are returned in a list for you.

SOLUTION

```
calc_hidden_units <- function(X, B, g)
{
  ### your code
  A <- X %*% B
  H <- g(X %*% B)

  ### book keeping
  return(list(A = A, H = H))
}
```

1e)

You will calculate the hidden units for the `prob_01_df` dataset and the `params_01_2a` parameters. The code chunk below provides a function which visualizes the hidden units with respect to a single input, `x`. You will use this function to interpret the behavior of the hidden units. `viz_hidden_trend_wrt_x()` accepts three input arguments. The first, `v_mat`, is a matrix of hidden unit values. The second, `x_df`, is a `data.frame` which must contain a column named `x`. The third, `trend_type`, is a character string

containing the “type” of hidden unit values contained in the `v_mat` matrix. The `trend_type` variable is assigned to the legend title. The `trend_type` argument is used to state whether the resulting figure is plotting the hidden unit “linear predictors” or the non-linear hidden unit values.

```
viz_hidden_trend_wrt_x <- function(v_mat, x_df, trend_type)
{
  x_df %>%
    select(all_of(c("x"))) %>%
    tibble::rowid_to_column("obs_id") %>%
    left_join(v_mat %>% as.data.frame() %>%
                tibble::as_tibble() %>%
                purrr::set_names(sprintf("h_%02d", 1:ncol(v_mat))) %>%
                tibble::rowid_to_column("obs_id"),
              by = "obs_id") %>%
    pivot_longer(!c("obs_id", "x")) %>%
    ggplot(mapping = aes(x = x, y = value)) +
    geom_vline(xintercept = 0, color = 'grey50') +
    geom_line(mapping = aes(color = name, group = name),
              size = 1.15) +
    ggthemes::scale_color_colorblind(trend_type) +
    theme_bw() +
    theme(legend.position = "top")
}
```

IMPORTANT: In this problem you will use the **logistic** function as the non-linear (activation) function.

Complete the two code chunks below. You must create the design matrix `X01`. You must call the `calc_hidden_units()` function by passing in the appropriate arguments and storing the result to `nnet_hidden_2a`. Two calls to `viz_hidden_trend()` are started for you. Complete the calls by assigning the correct element from `nnet_hidden_2a`, `A` or `H`, as the first argument to the `viz_hidden_trend()` call.

HINT: The third argument to `viz_hidden_trend_wrt_x()` tells you to whether to assign the “linear predictors” or the non-linear hidden unit values.

HINT: Use the `$` operator to access the elements from the list `nnet_hidden_2a`.

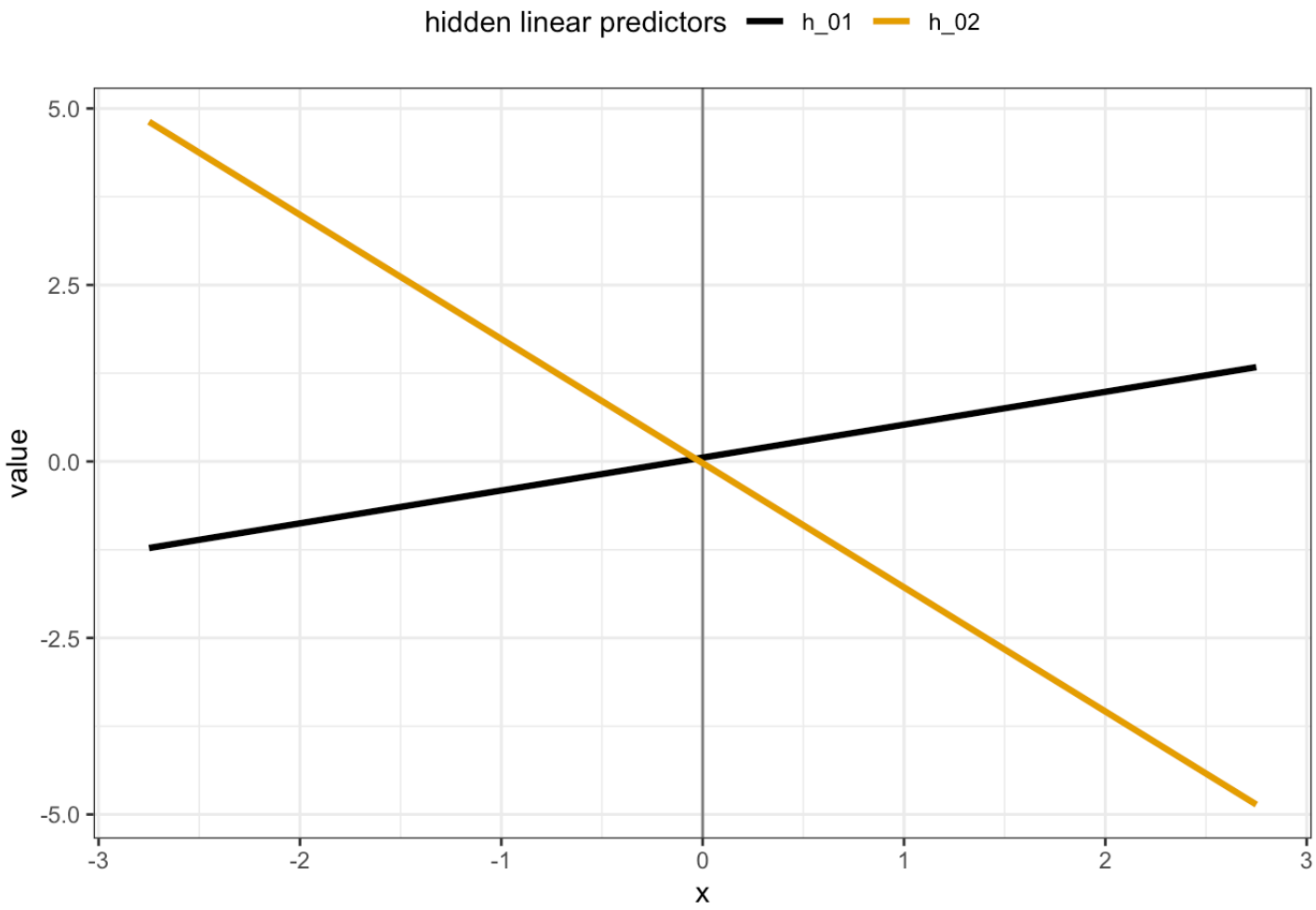
HINT: What function can we use for the logistic function?

SOLUTION

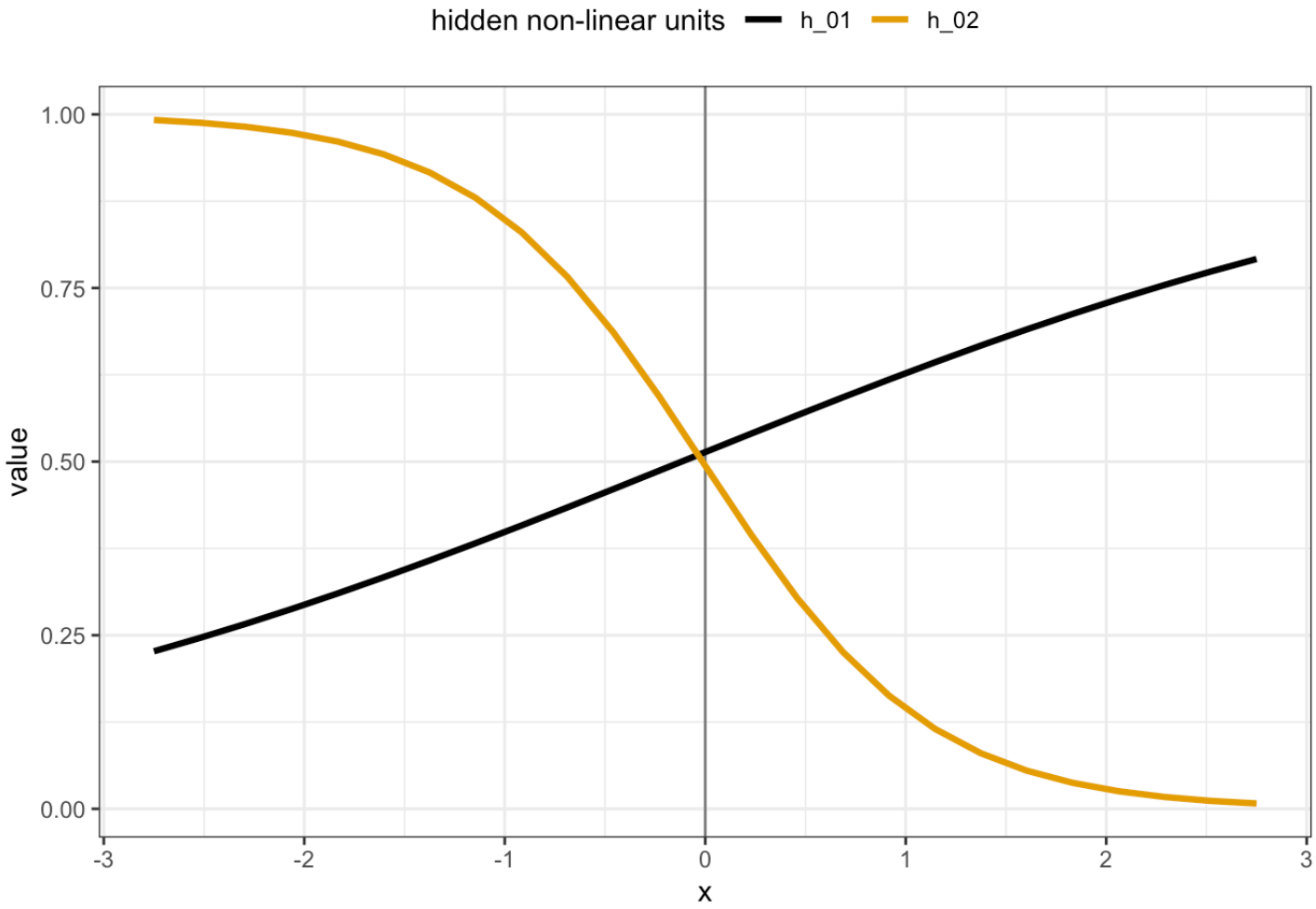
```
X01 <- model.matrix( ~ x, data = prob_01_df)

nnet_hidden_2a <- calc_hidden_units(X01,params_01_2a$beta_matrix, boot::inv.logit )
```

Visualize the hidden unit trends below.

```
viz_hidden_trend_wrt_x(nnet_hidden_2a$A,
                       prob_01_df,
                       "hidden linear predictors")
```




```
viz_hidden_trend_wrt_x(nnet_hidden_2a$H,
                       prob_01_df,
                       "hidden non-linear units")
```



1f)

Describe the trends of the hidden linear predictors relative to the β parameter values displayed in Problem 1b).

SOLUTION

What do you think?

The hidden linear predictors relative to the beta parameter values are very similar to each other.

1g)

The neural network response, f, is calculated as a linear combination of the non-linear hidden unit values in the output layer. Assume the output layer consists of an intercept (bias) α_0 and a column vector of slopes (weights) $\boldsymbol{\alpha}$.

Write out the expression for the response vector $\mathbf{f}$ given the output layer parameters, α_0, $\boldsymbol{\alpha}$, and the matrix of non-linear hidden unit values $\mathbf{H}$.

SOLUTION

Add as many equation blocks as you feel are necessary.

$$\mathbf{f} = \alpha_0 + \phi\alpha$$
$$\mathbf{H} = g(XB) = \phi$$
$$\mathbf{f} = \alpha_0 + \phi\alpha = \alpha_0 + H\alpha$$

1h)

As with the hidden units, you will use a function to calculate the neural network response. The code chunk below defines the `calc_nnet_response()` function. It accepts two arguments. The first, `H`, is the matrix of non-linear hidden unit values and the second, `a`, is the "regular vector" of output layer parameters. Note that `a` contains the intercept (the bias) and the slopes (the weights).

Complete the code chunk below. You must separate the `a` vector into the intercept (bias) and the slopes (weights). Store the intercept as the `a_0` variable and the slopes as the `a_w` regular vector. You must then convert the `a_w` regular vector into a column vector `a_col`. Finally, you must calculate the response, `f`. The response is returned as a vector for you.

SOLUTION

```
calc_nnet_response <- function(H, a)
{

  ### separate the vector into bias and weights
  a0 <- a[1]
  a_w <- a[-1]

  # convert the weights to a column vector
  a_col <- matrix(a_w)

  # calculate the response (the output layer)
  f <- a0 + H %*% a_col

  as.vector(f)
}
```

1i)

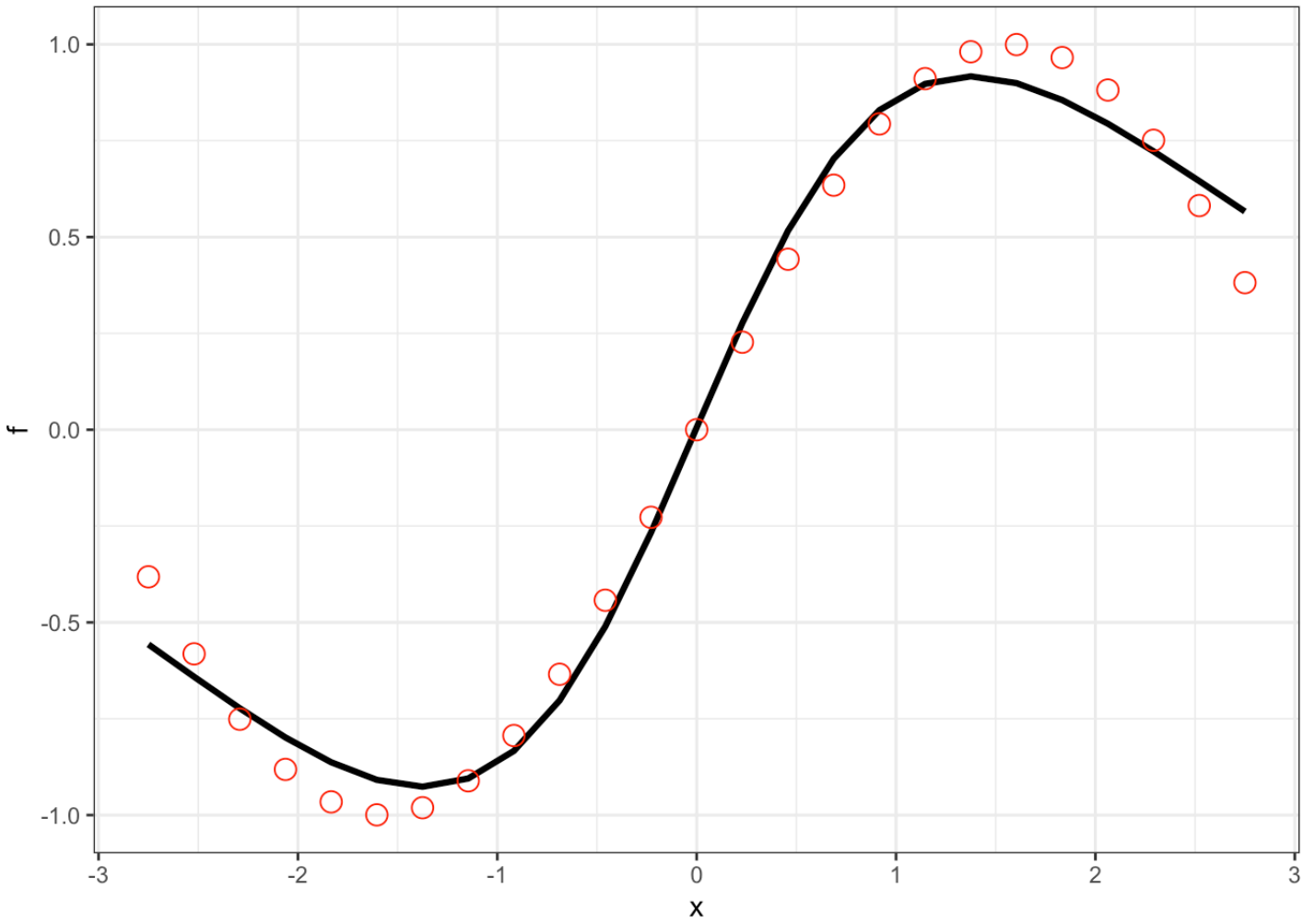
With all the calculations completed, let's now compare the neural network response to the true sine wave.

Complete the code chunk below by assigning the correct the arguments to the `calc_nnet_response()` function. The rest of the code, which generates the figure, is completed for you.

How would you describe the neural network's fit? Which portions are approximated well?

SOLUTION

```
prob_01_df %>%
  mutate(nnet_f = calc_nnet_response(nnet_hidden_2a$H, params_01_2a$alpha_vector )) %
>%
  ggplot(mapping = aes(x = x)) +
  geom_line(mapping = aes(y = nnet_f),
            color = "black", size = 1.15) +
  geom_point(mapping = aes(y = f),
             color = "red", shape = 1, size = 3.5) +
  labs(y = "f") +
  theme_bw()
```



What do you think?

The neural network's fit is pretty close and closely follows the true sine wave. It approximates the sine wave the best in the center portion of the graph and slightly deviates from the true sine wave in the ends of the graph.

Problem 02

You will now repeat your calculations from Problem 1. You will first try out a different set of parameters for a two hidden unit neural network. Then you consider a neural network with 5 hidden units.

2a)

The code chunk below reads in a new set of neural network parameters. The format is consistent with that from Problem 1, in that the hidden unit parameters are contained within the element `$beta_matrix` and the output layer parameters are stored in `$alpha_vector`. The parameters are printed to the screen below.

```
url_load_01_2b <- paste(paste(url_load_dir, "hw_10_prob_01_params_2b.rds", sep = "/")
,
                        "raw=true",
                        sep="?")

params_01_2b <- readr::read_rds( url_load_01_2b )

params_01_2b
```

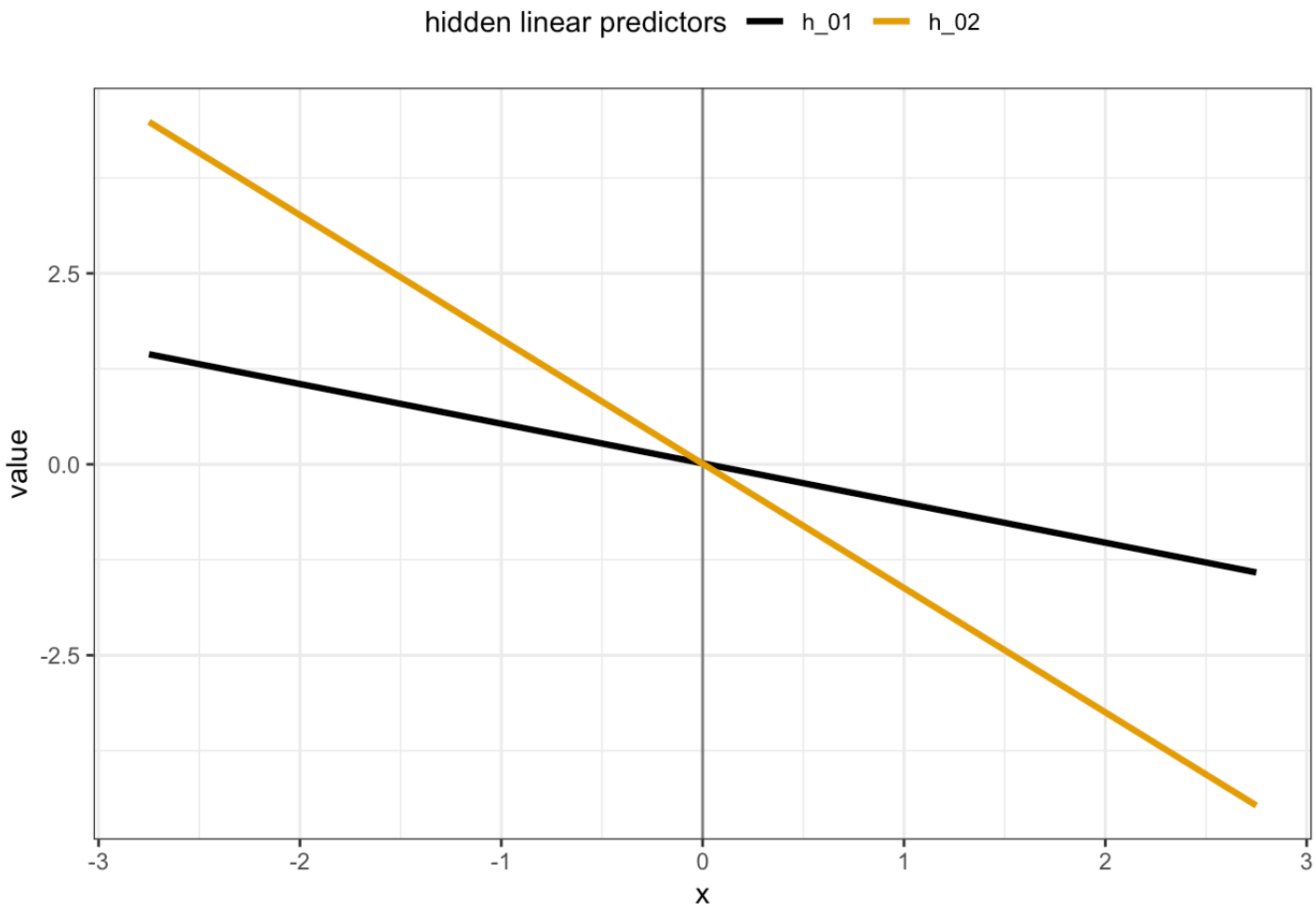
```
## $beta_matrix
##              [,1]         [,2]
## [1,]  0.01294902  0.008012561
## [2,] -0.51935552 -1.627414579
##
## $alpha_vector
## [1] -0.5509242  5.8885173 -4.8101438
```

You will use these new parameters to calculate the hidden unit "linear predictors" and the non-linear hidden unit values for the two hidden unit neural network.

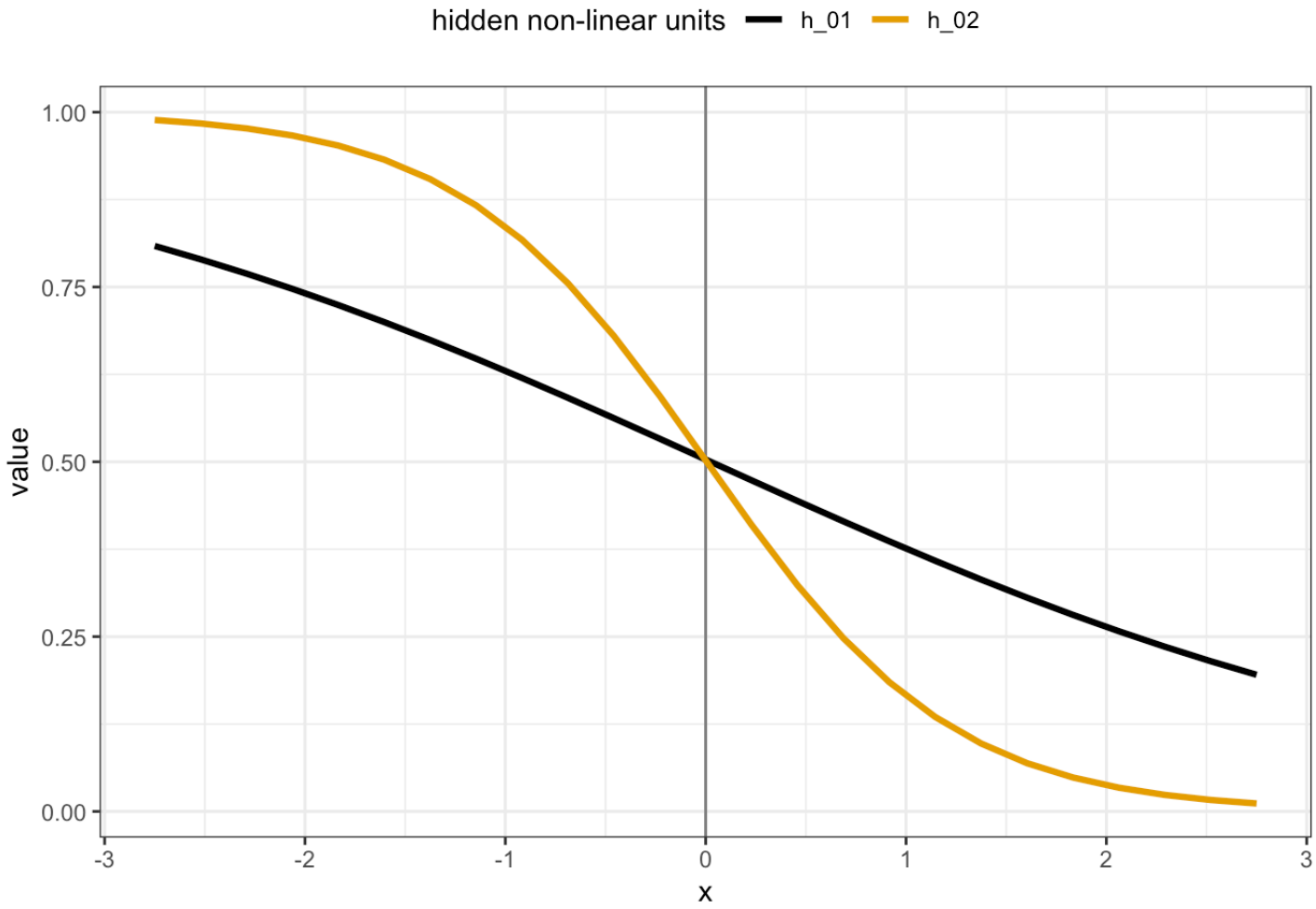
Complete the code chunk below. Call the `calc_hidden_units()` function with the appropriate arguments and assign the result to the `nnet_hidden_2b` object. Then assign the correct arguments to the `viz_hidden_trend()` functions calls. Are the trends of the hidden unit values (linear and non-linear) consistent with the trends from Problem 1? If not, what would be causing the change? Based on the figures generated in the code chunk below, do you think the resulting neural network will be different from that in Problem 1?

SOLUTION

```
nnet_hidden_2b <- calc_hidden_units(X01,params_01_2b$beta_matrix, boot::inv.logit )

viz_hidden_trend_wrt_x(nnet_hidden_2b$A ,
                       prob_01_df,
                       "hidden linear predictors")
```




```
viz_hidden_trend_wrt_x(nnet_hidden_2b$H ,
                       prob_01_df,
                       "hidden non-linear units")
```



What do you think?

The linear predictor values are higher in this figure than in the first one, so the resulting nerual network may be different.

2b)

Let's now compare the responses associated with the two sets of parameters.

Complete the first code chunk below by assigning the correct arguments to the two `calc_nnet_response()` calls. The first call is associated with the parameters from Problem 1, with the result stored to the `nnet_fa` variable. The second call is associated with the new set of parameters, and the result is stored to the `nnet_fb` variable.

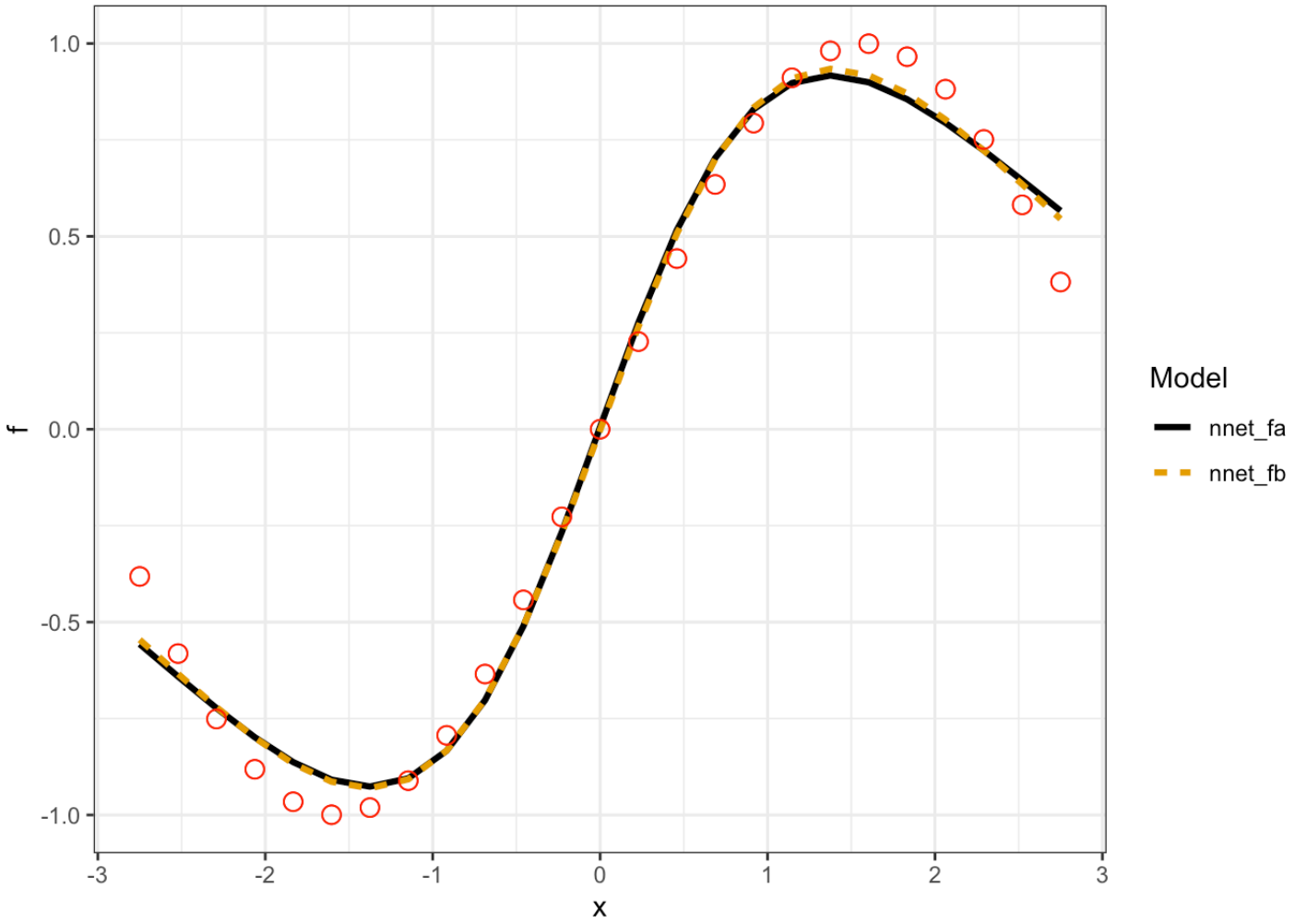
The second code chunk is completed for you. It plots the two responses together with the true sine wave output. Based on the figure below, how do the two different neural network models compare?

SOLUTION

```
results_2hidden <- prob_01_df %>%
  mutate(nnet_fa = calc_nnet_response(nnet_hidden_2a$H, params_01_2a$alpha_vector ),
         nnet_fb = calc_nnet_response( nnet_hidden_2b$H, params_01_2b$alpha_vector))
```

Now visualize the neural network predictions and compare to the true sine wave.

```
results_2hidden %>%
  pivot_longer(!c("x", "f")) %>%
  ggplot(mapping = aes(x = x)) +
  geom_line(mapping = aes(y = value,
                          group = name,
                          linetype = name,
                          color = name),
            size = 1.15) +
  geom_point(mapping = aes(y = f),
             color = "red", size = 3, shape = 1) +
  ggthemes::scale_color_colorblind("Model") +
  scale_linetype_discrete("Model") +
  labs(y = "f") +
  theme_bw()
```



What do you think?

The two neural networks are very similar to each other in predicting the true sine wave.

2c)

Based on your results, can you "explain" or interpret the neural network behavior by ONLY examining the slopes (weights) acting on the inputs?

SOLUTION

What do you think?

No, you cannot explain or interpret the neural network behavior by only examining the slopes acting on the inputs. For the two models, they had different slopes on the inputs but produced very similar results, making it difficult to understand what the result will be simply by looking at the slopes.

2d)

Let's now perform the same type of calculations, but on a neural network with 5 hidden units instead of 2. As before, you will compare two sets of parameters for the 5 hidden unit model. The code chunk below reads in the two different sets of neural network parameters. The format is consistent with that from Problem 1, except now there are more parameters since there are more hidden units.

```
url_load_01_5a <- paste(paste(url_load_dir, "hw_10_prob_01_params_5a.rds", sep = "/")
,
                        "raw=true",
                        sep="?")

params_02_5a <- readr::read_rds( url_load_01_5a )

url_load_01_5b <- paste(paste(url_load_dir, "hw_10_prob_01_params_5b.rds", sep = "/")
,
                        "raw=true",
                        sep="?")

params_02_5b <- readr::read_rds( url_load_01_5b )
```

The first set neural network parameters, `params_02_5a` , are displayed for you below to show the difference in structure with the 2 hidden unit neural networks you worked with previously.

```
params_02_5a
```

```
## $beta_matrix
##           [,1]     [,2]       [,3]      [,4]     [,5]
## [1,] -2.907979 0.817151 0.02513943  2.350765 9.723595
## [2,]  1.049041 2.425177 0.69695967 -2.648779 4.050475
##
## $alpha_vector
## [1]  1.1197289 -2.6861415  1.7513428 -0.4276571 -1.3391502 -0.7399066
```

Calculate the hidden unit "linear predictors" and non-linear values for the two different 5 hidden unit models. Then complete the calls to the `viz_hidden_trend_wrt_x()` function to plot the hidden unit trends with respect to the input.

Describe the trends of the non-linear values for the fifth hidden unit, `h_05` , relative to its "linear predictor" values. Discuss the differences between the two models (sets of parameters).

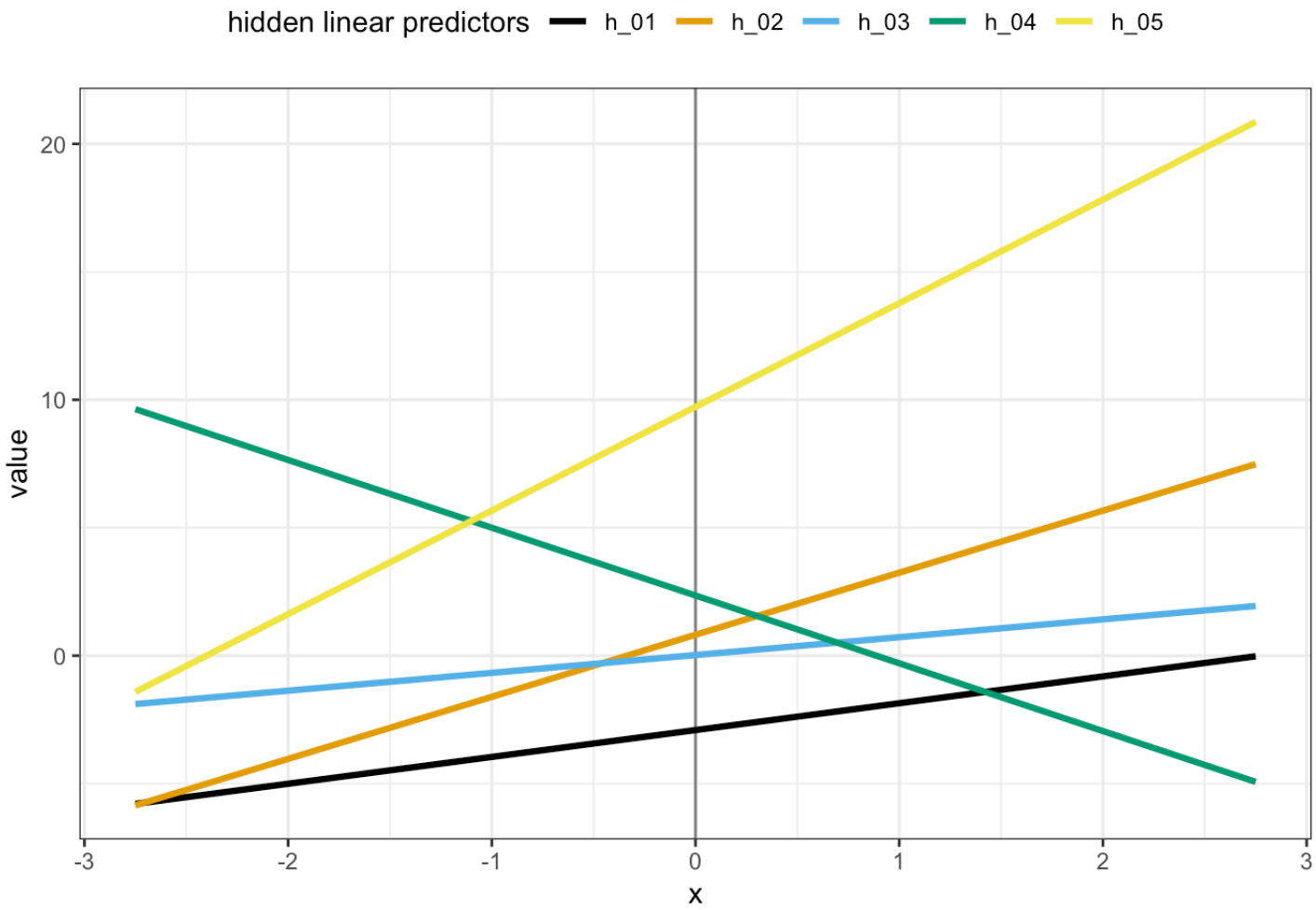
SOLUTION

Calculate the hidden unit "linear predictors" and non-linear hidden unit values.

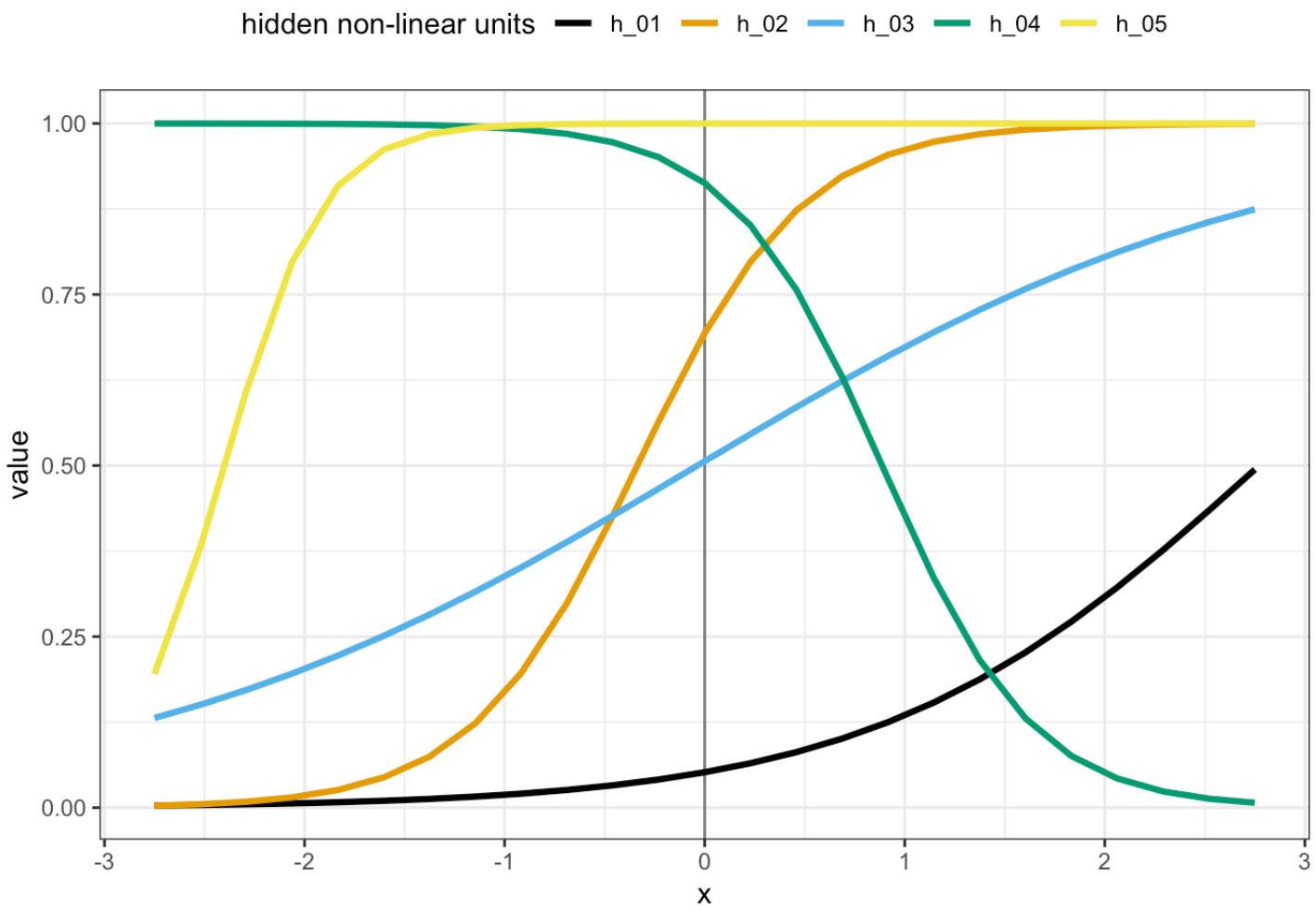
```
nnet_hidden_5a <- calc_hidden_units(X01,params_02_5a$beta_matrix, boot::inv.logit )

nnet_hidden_5b <- calc_hidden_units(X01,params_02_5b$beta_matrix, boot::inv.logit )
```

Visualize the behavior of the hidden units associated with the `params_02_5a` parameters.

```
viz_hidden_trend_wrt_x(nnet_hidden_5a$A ,
                       prob_01_df,
                       "hidden linear predictors")
```

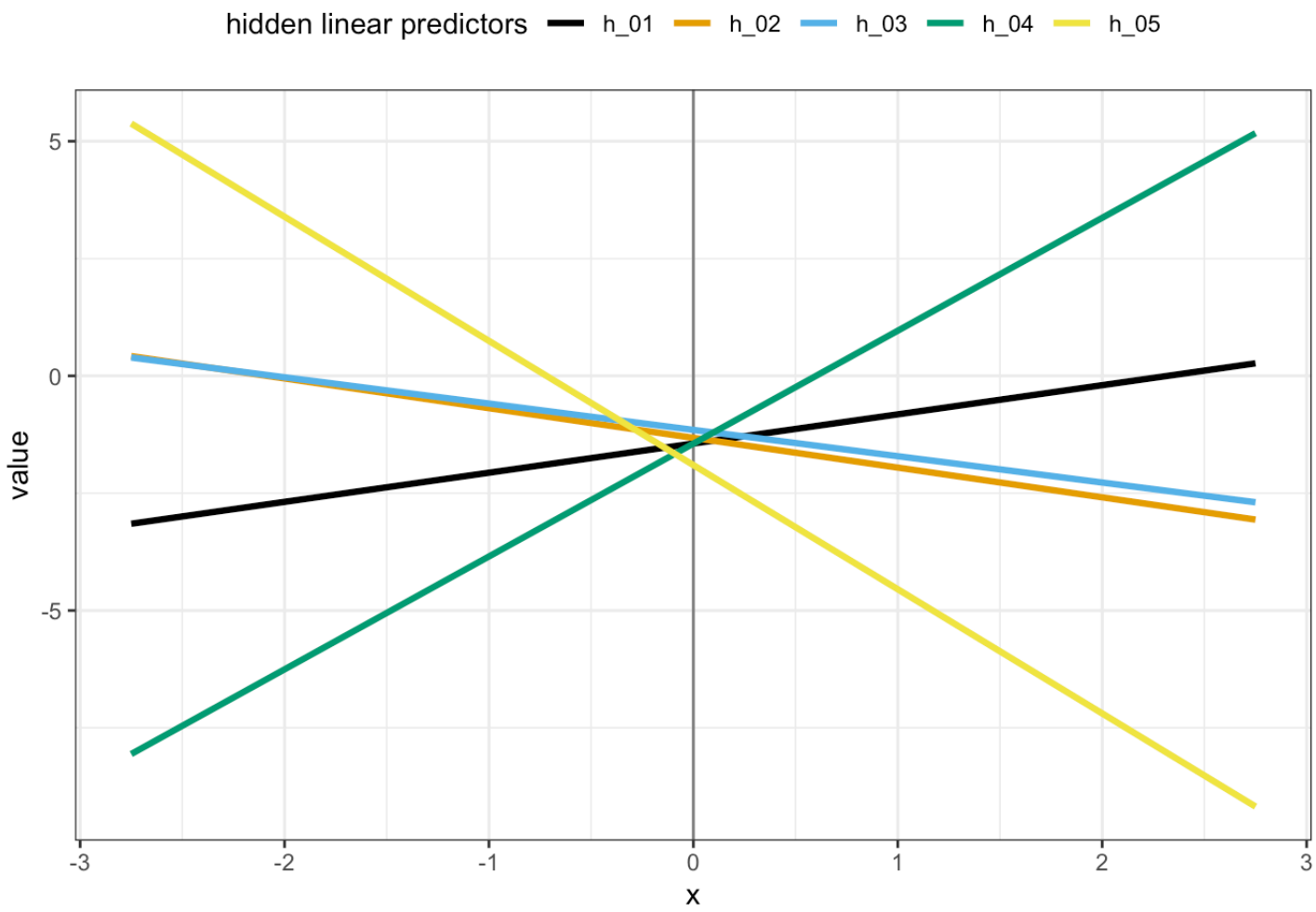



```
viz_hidden_trend_wrt_x( nnet_hidden_5a$H,
                       prob_01_df,
                       "hidden non-linear units")
```

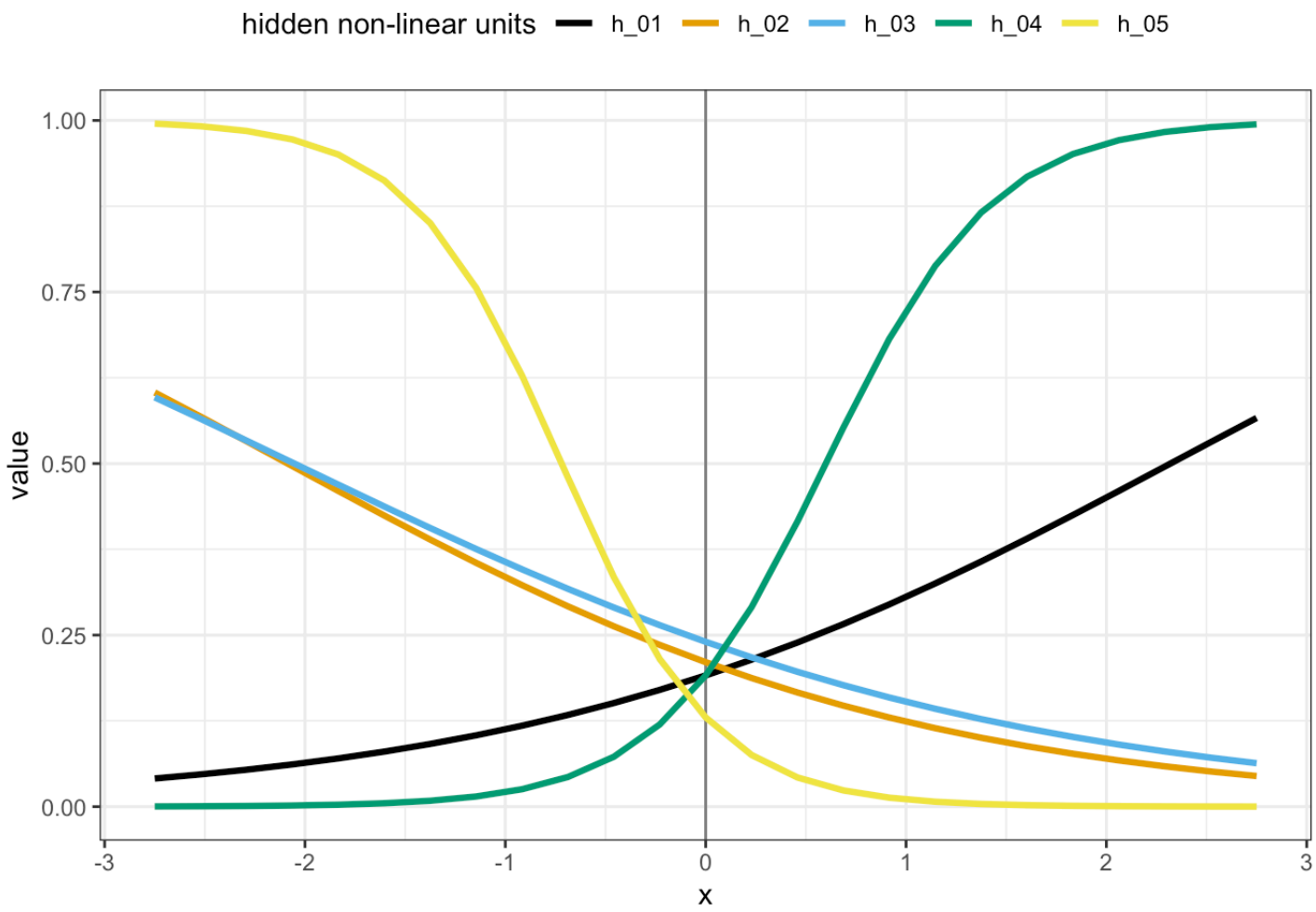


Visualize the behavior of the hidden units associated with the `params_02_5b` parameters.

```
viz_hidden_trend_wrt_x(nnet_hidden_5b$A ,
                       prob_01_df,
                       "hidden linear predictors")
```




```
viz_hidden_trend_wrt_x( nnet_hidden_5b$H,
                        prob_01_df,
                        "hidden non-linear units")
```



What do you think?

The non-linear values compared to the linear values for the fifth hidden unit (h_05) is a negative linear relationship between the value and x. In the non-linear figure, the fifth hidden unit is a negative S-curve starting at 1 and decreasing to 0.

2e)

Let's now calculate the neural network response for both with 5 hidden units.

Complete the first code chunk below by assigning the arguments correctly to the two `calc_nnet_response()` function calls. The first call is intended for the model associated with `params_02_5a` while the second call is intended for the model associated with `params_02_5b`.

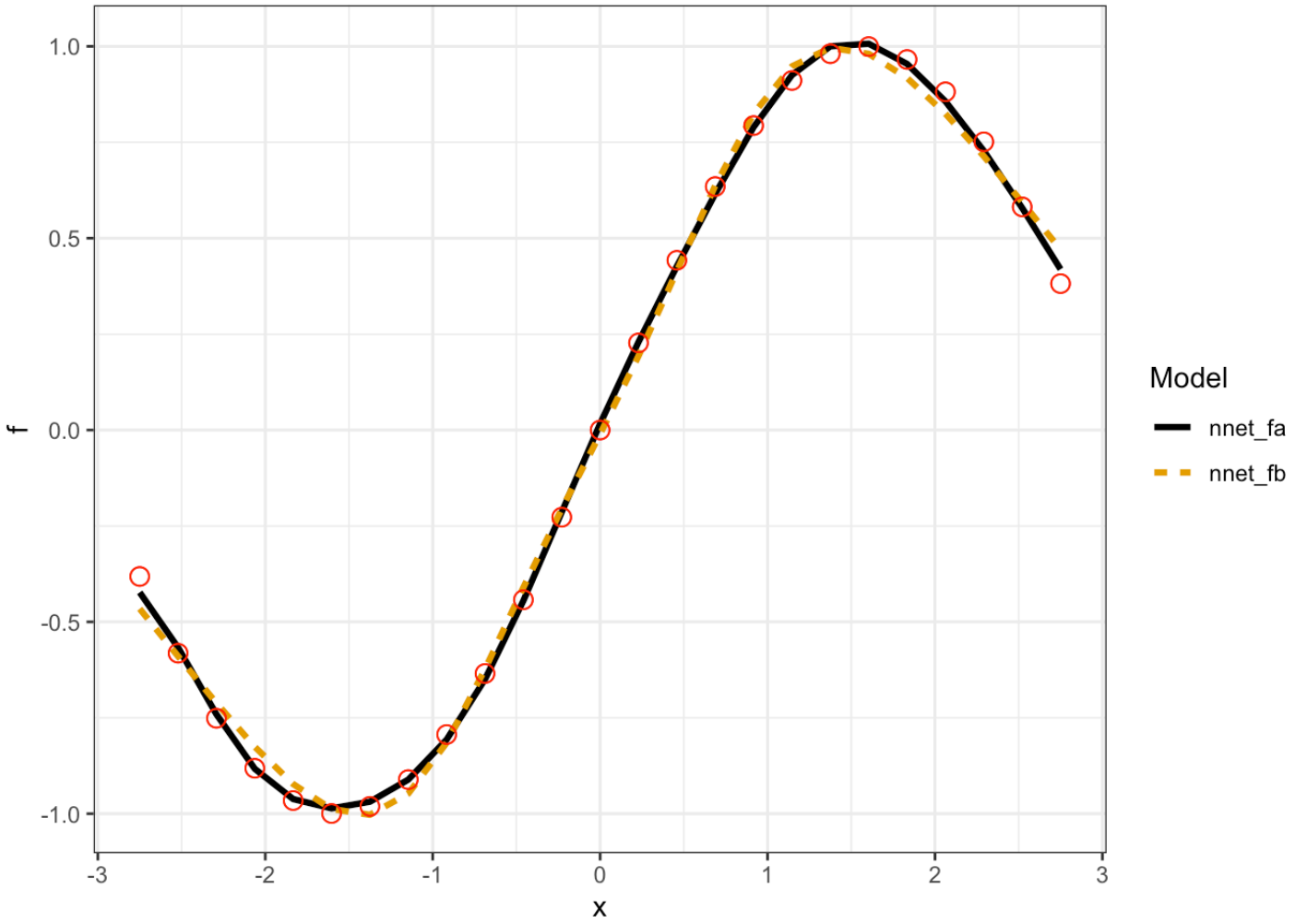
The second code chunk is completed for you. How do the two models compare to each other and to the true sine wave?

SOLUTION

```
results_5hidden <- prob_01_df %>%
  mutate(nnet_fa = calc_nnet_response(nnet_hidden_5a$H, params_02_5a$alpha_vector ),
         nnet_fb = calc_nnet_response(nnet_hidden_5b$H, params_02_5b$alpha_vector ))
```

Visualize the two neural network model predictions and compare with the true sine wave.

```
results_5hidden %>%
  pivot_longer(!c("x", "f")) %>%
  ggplot(mapping = aes(x = x)) +
  geom_line(mapping = aes(y = value,
                          group = name,
                          linetype = name,
                          color = name),
            size = 1.15) +
  geom_point(mapping = aes(y = f),
             color = "red", size = 3, shape = 1) +
  ggthemes::scale_color_colorblind("Model") +
  scale_linetype_discrete("Model") +
  labs(y = "f") +
  theme_bw()
```

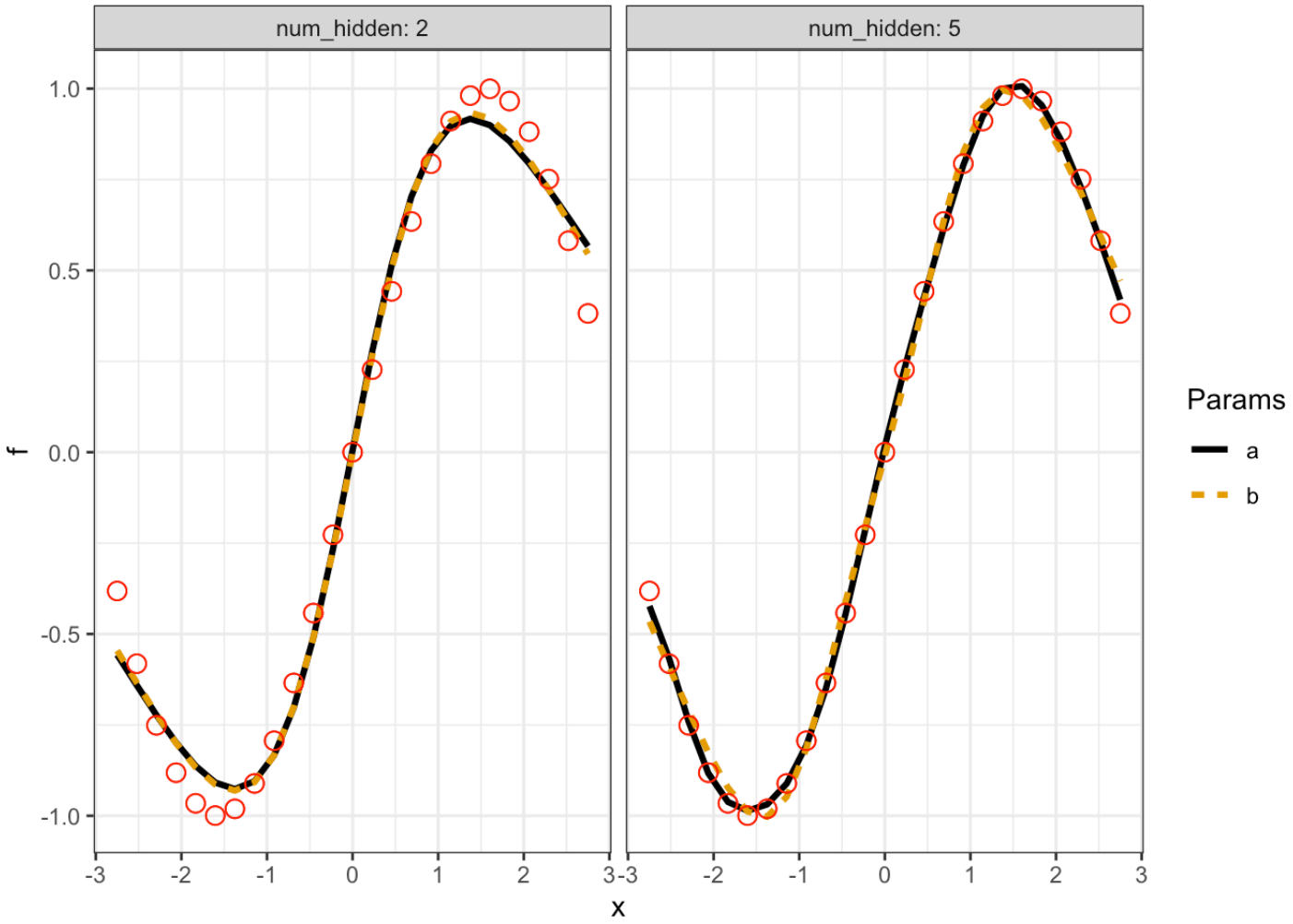


The two models are very similar to the true sine wave.

2f)

The code chunk below is completed for you. The 2 hidden unit and 5 hidden unit model predictions are compared side by side.

```
results_2hidden %>%
  mutate(num_hidden = 2) %>%
  bind_rows(results_5hidden %>%
              mutate(num_hidden = 5)) %>%
  pivot_longer(!c("x", "f", "num_hidden")) %>%
  tidyr::separate(name,
                  c("nnet_word", "fparams"),
                  sep = "_") %>%
  tidyr::separate(fparams,
                  c("fltr", "paramset"),
                  sep = 1) %>%
  ggplot(mapping = aes(x = x)) +
  geom_line(mapping = aes(y = value,
                          group = interaction(paramset,
                                              num_hidden),
                          linetype = paramset,
                          color = paramset),
            size = 1.15) +
  geom_point(mapping = aes(y = f),
             color = "red", size = 3, shape = 1) +
  facet_grid( ~ num_hidden, labeller = "label_both") +
  ggthemes::scale_color_colorblind("Params") +
  scale_linetype_discrete("Params") +
  labs(y = "f") +
  theme_bw()
```



Based on the figure above, which model, the 2 hidden units or 5 hidden units, performs better? What controls complexity within a neural network model and how could you go about "tuning" that complexity?

SOLUTION

What do you think?

The model with the 5 hidden units performs better than the model with 2 hidden units, especially in the ends of the sine wave. The number of hidden units controls the complexity and can be tuned by changing the number of hidden units used in the neural network.

2g)

The toy data within Problem 1 and 2 comes from a simple sine wave:

$$f(x) = \sin(x)$$

Let's see if a linear model, using the correct `sin()` basis can correctly identify that the "slope" acting on `sin(x)` is 1.

Use the `lm()` function to a fit a linear model for the response `f` and the sine of the input, `sin(x)`. Use the `prob_01_df` data set as the `data` argument to the `lm()` call. Assign the result to the `lm_sine_mod` object.

Print the `summary()` of the `lm()` call to the screen. What is the estimate for the slope associated with the `sin(x)`?

SOLUTION

```
lm_sine_mod <- lm(f ~ sin(x), data = prob_01_df)

### print the summary to the screen
lm_sine_mod %>% summary()
```

```
## Warning in summary.lm(.): essentially perfect fit: summary may be unreliable
```

```
##
## Call:
## lm(formula = f ~ sin(x), data = prob_01_df)
##
## Residuals:
##        Min          1Q      Median          3Q         Max
## -1.195e-16 -8.969e-17 -2.337e-17  4.303e-17  5.538e-16
##
## Coefficients:
##              Estimate Std. Error   t value Pr(>|t|)
## (Intercept) 8.882e-17  2.840e-17 3.127e+00  0.00473 **
## sin(x)      1.000e+00  3.842e-17 2.603e+16  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.42e-16 on 23 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:      1
## F-statistic: 6.774e+32 on 1 and 23 DF,  p-value: < 2.2e-16
```

What is the estimate to the slope acting on `sin(x)`?

The slope acting on sin(x) is 1.0

2h)

How many unknown parameters were there in the linear model with the sine wave basis function? How many unknown parameters existed in the neural network with 5 hidden units?

SOLUTION

What do you think?

In the linear model with the sine wave basis function there are 2 unknown parameters, the intercept and slope acting on the sin(x). In the neural network with 5 hidden units there are 16 unknowns parameters. Since there is 1 input and 5 hidden units, there are 5(1+1)+5+1 = 16 hidden units.

2i)

It was really simple to fit the linear model. Why would we want to use a neural network when we can build the exact model in this application using `lm()` ?

SOLUTION

What do you think?

We would want to use a neural network instead of building the exact model using lm() because we are able to be more confident in the results. With the lm() an exact model is produced, which can lead to unreliable results.

Problem 03

In the previous problems, you focused on the predictions of a neural network. You will now work through fitting neural networks, on a slightly more realistic example. The code chunk below reads a data set consisting of 3 continuous inputs, `x1` , `x2` , and `x3` , and a continuous response, `y` . A glimpse of the data set is displayed to the screen for you.

```
url_03_train <- 'https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW
/10/hw_10_prob_03_train.csv'

prob_03_df <- readr::read_csv( url_03_train, col_names = TRUE)
```

```
## Rows: 200 Columns: 4
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl (4): x1, x2, x3, y
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
prob_03_df %>% glimpse()
```

```
## Rows: 200
## Columns: 4
## $ x1 <dbl> -2.54605922, -0.54183583, -0.57203835, -0.95297025, 0.56787408, 2.9…
## $ x2 <dbl> 0.40139263, -0.24831167, -1.45541350, -0.85426629, -2.68426684, -2.…
## $ x3 <dbl> 2.25537905, 2.49124297, 0.78314499, -2.25292859, -0.21615775, 2.779…
## $ y  <dbl> -3.8897131, -3.3908067, -1.6440685, -0.1608670, -1.2154896, -2.8189…
```

3a)

The wide-format data set is converted into a long-format data set for you in the code chunk below. The glimpse displayed to the screen shows that the inputs have been “stacked” or “gathered” together into a column `name` with their values given in the `value` column.

```
prob_03_lf <- prob_03_df %>%
  tibble::rowid_to_column("obs_id") %>%
  pivot_longer(!c("obs_id", "y"))

prob_03_lf %>% glimpse()
```

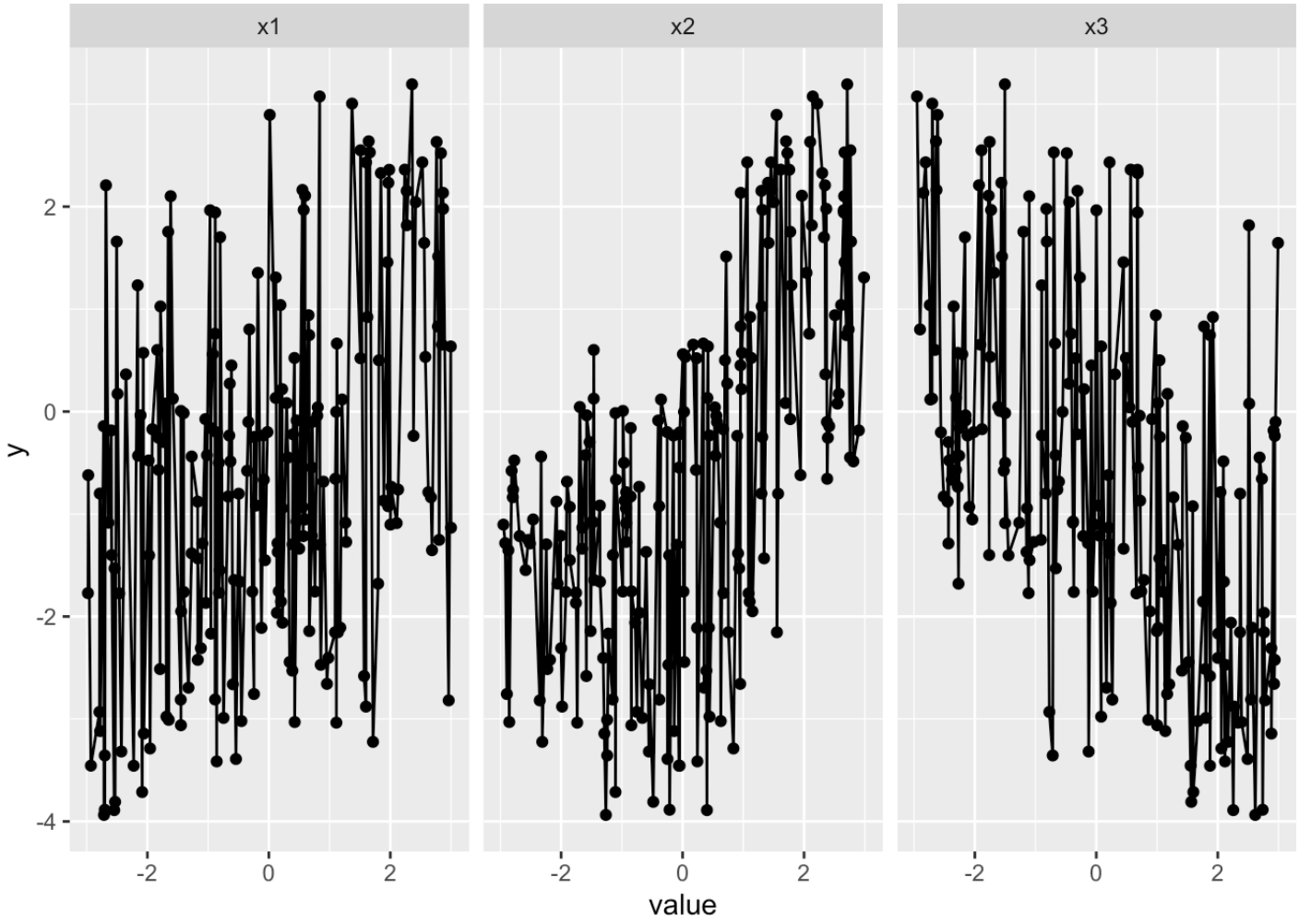
```
## Rows: 600
## Columns: 4
## $ obs_id <int> 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, …
## $ y      <dbl> -3.8897131, -3.8897131, -3.8897131, -3.3908067, -3.3908067, -3.…
## $ name   <chr> "x1", "x2", "x3", "x1", "x2", "x3", "x1", "x2", "x3", "x1", "x2…
## $ value  <dbl> -2.5460592, 0.4013926, 2.2553791, -0.5418358, -0.2483117, 2.491…
```

Plot the noisy response, `y`, with respect to each input using the long-format data set. Using the `geom_point()` geom and create separate facets (subplots) for each input using `facet_wrap()`.

What trends do you see in the scatter plots?

SOLUTION

```
prob_03_lf %>% ggplot(mapping = aes(x = value, y = y)) + geom_point() + geom_line() +
facet_wrap(~name)
```



In x1 there appears to be no apparent trend. In x2 there appears to be a positive linear relationship. In x3 there appears to be a negative linear relationship.

3b)

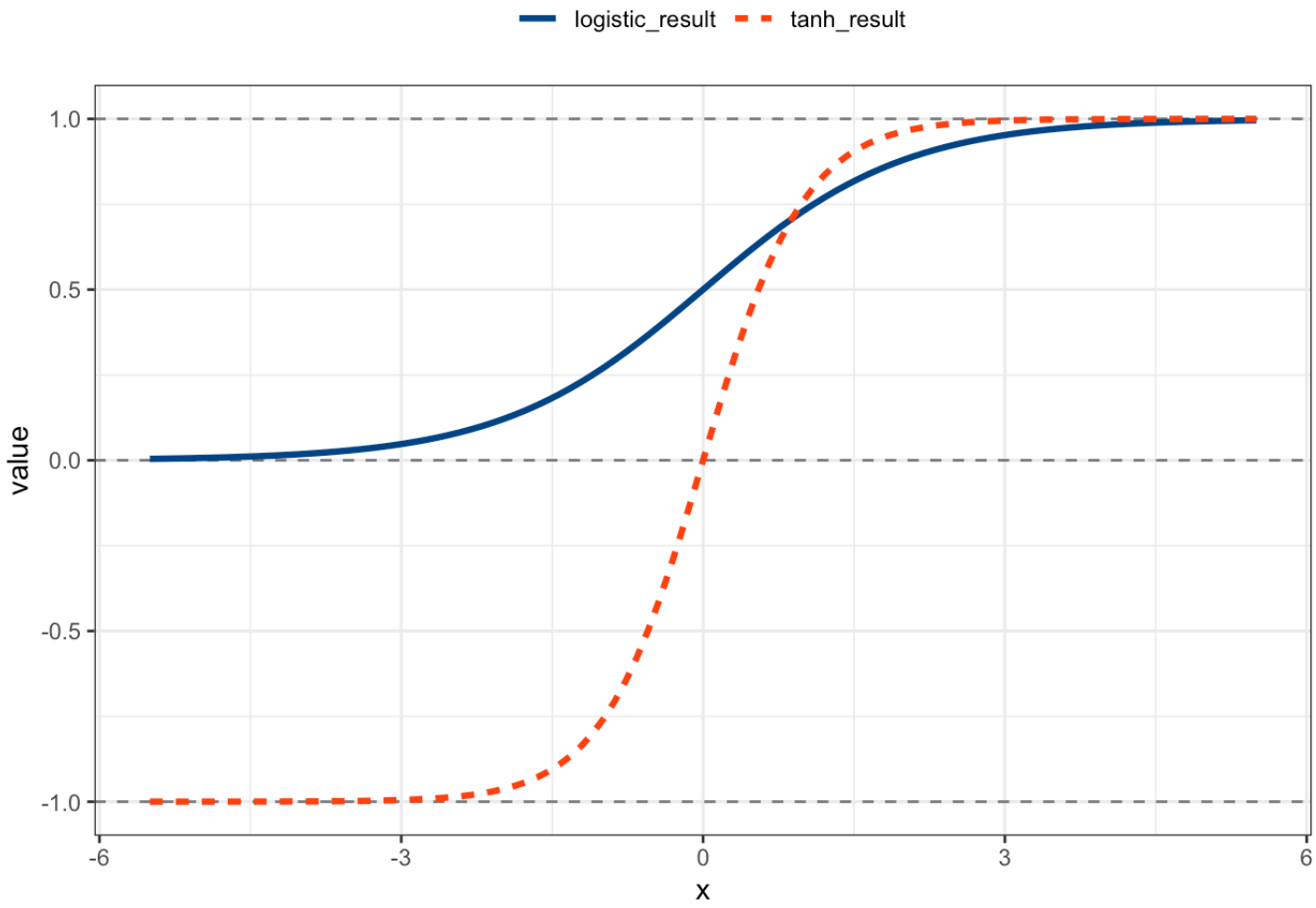
In the previous problems you used a logistic function as the non-linear function associated with each hidden unit. However, there are many different functions that could be used. To get exposure working with a different "activation" function, you will use the hyperbolic tangent function for this problem. Let's first get an idea about how the hyperbolic tangent compares with the logistic function.

Complete the code chunk below by setting `x` to be 101 equally spaced points between -5.5 and 5.5. Calculate the the logistic function of `x` and assign the result to `logistic_result`. Calculate the hyperbolic tangent of `x` and assign the result to `tanh_result`. The result of the code chunk is completed for you. It visualizes the non-linear transformations with respect to the `x` variable.

Is the hyperbolic tangent function similiar to the logistic function? In what ways are the two different?

HINT: The hyperbolic tangent function in `R` is `tanh()`.

```
tibble::tibble(
  x = seq(-5.5, 5.5, length.out = 101)
) %>%
  mutate(logistic_result = boot::inv.logit(x),
         tanh_result = tanh(x) ) %>%
  # rest of the code here is completed for you
  pivot_longer(!c("x")) %>%
  ggplot(mapping = aes(x = x, y = value)) +
  geom_hline(yintercept = c(-1, 0, 1),
             color = 'grey50', linetype = 'dashed') +
  geom_line(mapping = aes(y = value,
                          color = name,
                          linetype = name),
            size = 1.15) +
  ggthemes::scale_color_calc("") +
  scale_linetype_discrete("") +
  theme_bw() +
  theme(legend.position = "top")
```



Both the logistic and hyperbolic tan functions are S-shaped curves. However, the logistic function only goes from 0 - 1, while the hyperbolic tan function goes from -1 - +1.

3c)

You will fit the neural network by minimizing the sum of squared errors (SSE). Thus, we will work with a non-probabilistic setting, even though we derived the linear and generalized linear model fitting with likelihoods and priors. Remember that minimizing the SSE is analogous to maximizing a Gaussian log-likelihood!

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

SOLUTION

Add as many equation blocks as you feel are necessary.

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

3d)

You will now program the error function we wish to minimize in the style of the log-posterior functions from earlier homework assignments. You will name your function `my_neuralnet_sse()`. 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 required information, which is started for you in the code chunk below. Notice that the structure is similar t o the lists of information created for the generalized linear models. However, two pieces of information not associated with GLMs 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 3 hidden units. You will use the hyperbolic tangent as the non-linear transformation function, instead of the logistic function that you worked with in the previous problems. You will therefore need to assign the `tanh()` function correctly to the `$transform_hidden` field in the list. Be careful about the `()` when assigning the function *object*!

You will need to specify the design matrix for your neural network based on the 3 inputs in the `prob_03_df` data set. Think carefully how the design matrix is structured in a neural network.

Complete the list of required information by completing the code chunk below. You must create the design matrix based on the three inputs. 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 3.

After specifying the `info_three_units` list, calculate the total number of parameters in the single hidden layer neural network with 3 hidden units and assign the result to the `info_three_units$num_params` variable.

SOLUTION

The code chunk is started for you below.

```
### design matrix
Xmat_03 <- model.matrix(~x1 +x2 + x3, data = prob_03_df)

info_three_units <- list(
  yobs = prob_03_df$y ,
  design_matrix = Xmat_03,
  num_hidden = 3,
  transform_hidden = tanh
)

D <- ncol(prob_03_df) -1
H <- info_three_units$num_hidden

info_three_units$num_params <- info_three_units$num_hidden * ncol(info_three_units$de
sign_matrix) + info_three_units$num_hidden + 1
```

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

```
info_three_units$num_params
```

```
## [1] 16
```

3e)

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 has several portions completed for you. **You are responsible for determining the number of hidden unit parameters (the betas) for each hidden unit.** You should **not** hard code `length_beta_per_unit`. You must then calculate the total number of hidden unit parameters and assign the result to `total_num_betas`. Again you should **not** hard code this number because later on you will try out more hidden units.

The hidden unit parameters are extracted from the `theta` vector and organized into the `Bmat` matrix with dimensions consistent with the $\mathbf{B}$ described in Problem 01 and 02.

The output layer parameters are extracted for you and assigned to the `a_all` vector. You must reorganize the output layer parameters by separating the bias, `a0` , and output layer weights, `aw` . The bias should be a scalar quantity and the output layer weights should be a “regular vector”.

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 `683.113` for the guess of all 0’s and it should be `2238.39` for the guess -1.25 for all parameters.

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$design_matrix
  length_beta_per_unit <- ncol(X) # how many betas are there???????
  total_num_betas <- my_info$num_hidden * length_beta_per_unit# how many total betas
are there???????

  beta_vec <- theta[1:total_num_betas]

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

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

  # reorganize the output layer parameters by extracting
  # the output layer intercept (the bias)
  a0 <- a_all[1] # output layer bias??????
  aw <- a_all[-1] # output layer weights?????

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

  # pass through the non-linear transformation function
  H <- my_info$transform_hidden(X %*% Bmat)

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

```
theta = rep(0, info_three_units$num_params)
my_neuralnet_sse(theta, info_three_units)
```

```
## [1] 683.113
```

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

```
theta = rep(-1.25, info_three_units$num_params)
my_neuralnet_sse(theta, info_three_units)
```

```
## [1] 2238.39
```

3f)

With the objective function completed, it's now time to fit the simple neural network with 3 hidden units. You will use the `optim()` function to perform the optimization, just as in the previous assignments. Since we are focused on finding the estimates at the moment, 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_three_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 FALSE which forces the Hessian matrix to **NOT** be estimated at the end. We are simply interested in the point estimates at the moment and so we will not be concerned with the curvature of the error surface. The maximum number of iterations is set for you in both `optim()` calls already.

Complete both code chunks below in order to fit the three 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_3_a <- optim( rep(0, info_three_units$num_params),
                        my_neuralnet_sse,
                       gr = NULL ,
                       info_three_units ,
                       method = "BFGS" ,
                       hessian = TRUE ,
                       control = list(maxit = 5001))
```

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

```
optim_fit_3_b <- optim( rep(-1.25, info_three_units$num_params),
                        my_neuralnet_sse,
                       gr = NULL ,
                       info_three_units ,
                       method = "BFGS" ,
                       hessian = TRUE ,
                       control = list(maxit = 5001))
```

Compare the optimized parameter estimates.

```
optim_fit_3_a$par
```

```
##  [1]  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000
## [13] -0.5974253  0.0000000  0.0000000  0.0000000
```

```
optim_fit_3_b$par
```

```
##  [1]  1.00285940 -0.06830407 -1.12774300  0.22106412 -5.33992528 -5.05832466
##  [7] -2.24687029  6.51755096  2.07834809  2.36113376 -2.47127795 -0.98996809
## [13] -0.39993603 -1.82474789 -0.75467292  0.62913367
```

What do you think?

The optimal parameters for the two starting guesses are not the same. The results are not the same because neural networks are multi-modal so depending on where the global and local maximum and minimums are, the optimal parameter identified will be the one that the initial guess “walks” to first based on the gradient.

3g)

Fit the neural network with 3 hidden units again, but this time use 2 randomly generated initial guess values. Use standard normals (mean 0 and standard deviation 1) to generate the initial guesses.

Complete the two code chunks below by generating two random initial guess values. Assign the first random initial guess to `init_guess_03_c` and the second random initial guess to `init_guess_03_d`. Complete the `optim()` calls following the same instructions as the previous question.

Check if the optimized parameter estimates are the same or not.

SOLUTION

Set the random initial guess values.

```
set.seed(412412)
init_guess_03_c <- rnorm(info_three_units$num_params)

set.seed(214214)
init_guess_03_d <-rnorm(info_three_units$num_params)
```

Run the optimization for the first random initial guess.

```
optim_fit_3_c <- optim( init_guess_03_c,
                        my_neuralnet_sse,
                       gr = NULL ,
                       info_three_units ,
                       method = "BFGS" ,
                       hessian = TRUE ,
                       control = list(maxit = 5001))
```

Run the optimization for the first random initial guess.

```
optim_fit_3_d <- optim( init_guess_03_d,
                        my_neuralnet_sse,
                       gr = NULL ,
                       info_three_units ,
                       method = "BFGS" ,
                       hessian = TRUE ,
                       control = list(maxit = 5001))
```

Compare the parameter estimates.

```
optim_fit_3_c$par
```

```
##  [1]  1.00284346 -0.06828951 -1.12772290  0.22105267  5.33799188  5.05664268
##  [7]  2.24615708 -6.51540353 -2.07830059 -2.36106773  2.47124267  0.98994657
## [13] -0.39992332 -1.82476026  0.75468202 -0.62914175
```

```
optim_fit_3_d$par
```

```
##  [1] -5.33897444 -5.05731437 -2.24645105  6.51620757 -1.00287295  0.06830811
##  [7]  1.12774940 -0.22106603  2.07826030  2.36101674 -2.47116336 -0.98992056
## [13] -0.39993820 -0.75467801  1.82474298  0.62913106
```

The optimal parameters are not the same for the two random initial starting guesses.

3h)

The `optim()` results store the objective function value as the `$value` field in the returned list object.

Compare the SSE for the 4 different starting guesses. Which model is better, as viewed by the training set?

SOLUTION

Use as many code chunks as you feel are necessary.

```
optim_fit_3_a$value
```

```
## [1] 611.7296
```

```
optim_fit_3_b$value
```

```
## [1] 33.02024
```

```
optim_fit_3_c$value
```

```
## [1] 33.02024
```

```
optim_fit_3_d$value
```

```
## [1] 33.02024
```

Problem 04

You now have the major pieces in place for fitting neural networks! In this problem, we will fit additional neural networks with more hidden units!

4a)

Let's define a function which will generate a random initial guess for the appropriate number of unknown parameters and then execute the `optim()` call. The `train_1layer_nnet_sse()` function has 4 input arguments. The first argument, `num_hidden`, is the number of hidden units in the hidden layer, the second, `transform_func`, is the non-linear transformation (activation) function, the third `X`, is the design matrix, and the fourth, `y`, the response vector.

Complete the code chunk below which assembles the list of required information and generates the random initial guess, for an arbitrary number of hidden units in the first hidden layer. Do not set the random seed inside the `train_1layer_nnet_sse()` function. We will set the seed before we fit the models.

HINT: If your function below is setup correctly you should be able to replicate the previous results if the **SAME** random seed is used. The second code chunk below resets the random seed for you as a confirmation test.

SOLUTION

```
train_1layer_nnet_sse <- function(num_hidden, transform_func, X, y)
{
  my_info_list <- list(
    yobs = y,
    design_matrix = X,
    num_hidden = num_hidden,
    transform_hidden = transform_func
  )

  my_info_list$num_params <- my_info_list$num_hidden * ncol(my_info_list$design_matri
x) +
    my_info_list$num_hidden + 1# total number of hidden and output layer parameters

  # generate random initial guess
  init_guess <- rnorm(my_info_list$num_params)

  # call optim to fit the neural network
  optim( init_guess,
         my_neuralnet_sse,
         gr =  NULL,
         my_info_list,
         method =  "BFGS",
         hessian =  TRUE,
         control = list(maxit = 10001))
}
```

As a check fit the 3 hidden unit neural network again with the same random seed as used with `init_guess_03_c` . You should get the same parameters as `optim_fit_3_c` .

```
set.seed(412412)
check_optim_fit_3_c <- train_1layer_nnet_sse(3,tanh, Xmat_03, prob_03_df$y )
```

Compare to the previous `optim_fit_3_c` results.

```
check_optim_fit_3_c$value
```

```
## [1] 33.02024
```

```
optim_fit_3_c$value
```

```
## [1] 33.02024
```

4b)

Let's now fit neural networks with 6, 12, and 24 hidden units instead of 3 hidden units. You will use two different initial guesses for each hidden layer size. The random seeds are set for you to make sure the results are reproducible.

Complete the three code chunks below by setting the input arguments to fit 2 pairs of 6, 12, and 24 hidden unit neural networks.

SOLUTION

```
set.seed(412412)
optim_fit_6_a <- train_1layer_nnet_sse(6,tanh, Xmat_03, prob_03_df$y )


set.seed(214214)
optim_fit_6_b <- train_1layer_nnet_sse(6,tanh, Xmat_03, prob_03_df$y )
```

```
set.seed(412412)
optim_fit_12_a <- train_1layer_nnet_sse(12,tanh, Xmat_03, prob_03_df$y )

set.seed(214214)
optim_fit_12_b <- train_1layer_nnet_sse(12,tanh, Xmat_03, prob_03_df$y )
```

Please note that fitting the two 24 hidden unit neural networks may take a few minutes.

```
set.seed(412412)
optim_fit_24_a <- train_1layer_nnet_sse(24,tanh, Xmat_03, prob_03_df$y )

set.seed(214214)
optim_fit_24_b <- train_1layer_nnet_sse(24,tanh, Xmat_03, prob_03_df$y )
```

4c)

Compare the training set SSE across all of the models you trained with random initial guesses (including the 3 hidden unit models).

Which model was considered the best according to the training set?

SOLUTION

Add as many code chunks as you feel are necessary to answer this question.

```
optim_fit_3_c$value
```

```
## [1] 33.02024
```

```
optim_fit_6_a$value
```

```
## [1] 13.68273
```

```
optim_fit_6_b$value
```

```
## [1] 13.259
```

```
optim_fit_12_a$value
```

```
## [1] 9.451085
```

```
optim_fit_12_b$value
```

```
## [1] 9.736034
```

```
optim_fit_24_a$value
```

```
## [1] 2.456799
```

```
optim_fit_24_b$value
```

```
## [1] 2.944205
```

Problem 05

We know that we should **not** compare models strictly based on the training set (unless we were using an information criterion metric). The code chunk below reads in a hold-out test set for you. This hold out test set will be used to compare the performance across the hidden layer sizes that you have fit so far.

```
url_03_test <- 'https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/10/hw_10_prob_03_test.csv'

prob_03_test_df <- readr::read_csv( url_03_test, col_names = TRUE)
```

```
## Rows: 50 Columns: 4
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl (4): x1, x2, x3, y
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
prob_03_test_df %>% glimpse()
```

```
## Rows: 50
## Columns: 4
## $ x1 <dbl> 2.76498634, -2.07813717, -2.37472507, -2.53391425, 2.48377722, -2.9…
## $ x2 <dbl> -1.09621757, 2.54787639, 1.12523665, 0.94841660, -2.62281034, -0.09…
## $ x3 <dbl> 0.6500768, 1.5583043, 1.9817675, -1.1122017, -1.9933706, -2.0585940…
## $ y  <dbl> -1.57257256, 0.35641307, -2.41294276, -0.21927096, -1.32098374, -1.…
```

5a)

You will define a function which makes predictions for you and calculates the Mean Squared Error (MSE) on the test set. The function, `assess_nnet_mse()`, is started for you in the code chunk below. The first argument, `theta`, is a vector of all unknown parameters, the second argument, `num_hidden`, is the number of hidden unit parameters, the third argument, `transform_func`, is the non-linear transformation function, the fourth argument, `X`, is a design matrix, and the fifth argument, `y`, is the response vector.

You have worked with the necessary pieces to complete this function several different ways in this assignment. You are free to decide how best to calculate the MSE for a given set of parameters. The only requirement is that `assess_nnet_mse()` should return a scalar number.

SOLUTION

```
assess_nnet_mse <- function(theta, num_hidden, transform_func, X, y)
{
  length_betas <- ncol(X)
  beta_vec <- theta[1:length_betas]
  total_num_betas <- num_hidden * length_betas

  Bmat <- matrix(beta_vec, nrow = length_betas, byrow = FALSE)

  a_all <- theta[(total_num_betas + 1): length(theta)]
  a0 <- a_all[1]
  aw <- a_all[-1]


  A <- X %*% Bmat
  H <- transform_func(X %*% Bmat)

  f <- as.vector(a0 + H %*% t(matrix(aw)))

  mean((y - f)^2)
}
```

5b)

Before you can calculate the MSE on the hold-out test set, you must create the test design matrix.

Create the appropriate test design matrix associated with all 3 inputs, using the `prob_03_test_df` data set, and assign the result to `Xtest_03` .

SOLUTION

```
Xtest_03 <- model.matrix(~ x1 + x2 + x3, data = prob_03_test_df)
```

5c)

Calculate the MSE for each of the models trained with random initial guess values. You are free to decide how to execute this task.

SOLUTION

Add as many code chunks as you feel are necessary to complete the problem.

```
set.seed(12345)
init_guess_03_test <-rnorm(info_three_units$num_params)

info_three_test_units <- list(
  yobs = prob_03_test_df$y ,
  design_matrix = Xtest_03,
  num_hidden = 3,
  transform_hidden = tanh
)
 info_three_test_units$num_params <- info_three_test_units$num_hidden * ncol(info_thr
ee_test_units$design_matrix) +
   info_three_test_units$num_hidden + 1

assess_nnet_mse(init_guess_03_test, info_three_test_units$num_hidden, info_three_test
_units$transform_hidden, Xtest_03, prob_03_test_df$y)
```

```
## [1] 4.269352
```

```
test_1layer_nnet_mse <- function(num_hidden, transform_func, X, y)
{

  num_params <- num_hidden * ncol(X) + num_hidden + 1# total number of hidden and out
put layer parameters

  # generate random initial guess
  init_guess <- rnorm(num_params)

  assess_nnet_mse(init_guess, num_hidden, transform_func, X, y )
}
```

```
set.seed(412412)
test_three <- test_1layer_nnet_mse(3,tanh, Xtest_03, prob_03_test_df$y )
test_three
```

```
## [1] 4.682671
```

```
test_six <- test_1layer_nnet_mse(6,tanh, Xtest_03, prob_03_test_df$y )
test_six
```

```
## [1] 3.938745
```

```
test_tweleve <- test_1layer_nnet_mse(12,tanh, Xtest_03, prob_03_test_df$y )
test_tweleve
```

```
## [1] 4.902941
```

```
test_24 <- test_1layer_nnet_mse(24,tanh, Xtest_03, prob_03_test_df$y )
test_24
```

```
## [1] 6.118979
```

```
set.seed(124124)
test_three <- test_1layer_nnet_mse(3,tanh, Xtest_03, prob_03_test_df$y )
test_three
```

```
## [1] 4.558016
```

```
test_six <- test_1layer_nnet_mse(6,tanh, Xtest_03, prob_03_test_df$y )
test_six
```

```
## [1] 3.545916
```

```
test_tweleve <- test_1layer_nnet_mse(12,tanh, Xtest_03, prob_03_test_df$y )
test_tweleve
```

```
## [1] 9.916603
```

```
test_24 <- test_1layer_nnet_mse(24,tanh, Xtest_03, prob_03_test_df$y )
test_24
```

```
## [1] 8.05759
```

5d)

Which model is the best as viewed by the hold-out test set performance?

SOLUTION

What do you think?

According to the results of the hold-out test set performance, the model with six hidden units is the best in both models.

Problem 06

You not only fit neural networks from scratch, but you used a hold-out test set to **tune** the number of hidden units! Doing so required you to directly work with the assumptions of the neural network, learning how to make predictions with the matrix operations, calculate the performance metric, and ultimately assess the potential for overfitting as the complexity increases. Understanding the assumptions and concepts are critical when assessing the behavior and performance of a neural network in a practical application when we use existing functions and packages to fit the neural network for us. In this last problem you will practice using `caret` to manage the training, evaluation, and tuning of a neural network. You will use the `nnet` package to fit the neural network. Please download and install `nnet` if you do not have it already. If you do not install it, `caret` will prompt you to install it and so please check the R console if nothing seems to happen when you use the `caret::train()` function.

The code chunk below loads the `caret` package for you. You do not need to load `nnet`, the `caret` package will manage that for you.

```
library(caret)
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

6a)

You must specify the resampling scheme that `caret` will use to train, assess, and tune the model.

Specify the resampling scheme to be 5 fold with 3 repeats. Assign the result of the `trainControl()` function to the `my_ctrl` object. Specify the primary performance metric to be `'RMSE'` and assign that to the `my_metric` object.

SOLUTION

```
my_metric <- 'RMSE'
my_ctrl <- trainControl(method = "repeatedcv" , number = 5, repeats = 3, my_metric)
```

6b)

In a realistic application, it is always best to first fit linear models before we fit neural networks. The linear models (especially regularized models which include interaction features) serve as interpretable baseline models. However, since this assignment is focused on neural networks we will just fit the neural network. You must train, assess, and tune a neural network using the **default** `caret` tuning grid. In the `caret::train()` function you must use the formula interface to specify the inputs are `x1` , `x2` , and `x3` , while the response is `y` . Assign the `method` argument to `'nnet'` and set the `metric` argument to `my_metric` . You must also instruct `caret` to standardize the features by setting the `preProcess` argument equal to `c('center', 'scale')` . Assign the `trControl` argument to the `my_ctrl` object.

Train, assess, and tune the `nnet` neural network with the defined resampling scheme. Assign the result to the `nnet_default` object and print the result to the screen. Which tuning parameter combinations are considered to be the best?

IMPORTANT: include the argument `trace = FALSE` in the `caret::train()` function call. This will make sure the `nnet` package does NOT print the optimization iteration results to the screen.

SOLUTION

```
set.seed(412412)
nnet_default <- caret::train(y ~ x1 + x2 + x3, data = prob_03_test_df, method ='nnet'
, metric= my_metric, preProcess = c('center', 'scale'), trControl = my_ctrl, trace =
FALSE)
```

```
## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo, :
## There were missing values in resampled performance measures.
```

```
nnet_default$bestTune
```

```
##   size decay
## 8    5 1e-04
```

6c)

You will only use the default `caret` tuning grid in this assignment. We could customize it to see if the performance could be improved, but for now the default grid is all we will use.

What do the two tuning parameters in the `nnet` package correspond to?

SOLUTION

What do you think?

The folds and repeats are the two tuning parameters which represent the number of splits in the data set (folds) and the number of times to repeat the splitting into the folds (repeats). So, in this case, we are splitting the data into 5 groups and repeating this grouping 3 times.