Neural Nets from Scratch

Daniel Polites

2024-04-02

Contents

1 Intro **5**

2 Single-Layer NN Notes **7**

2.1 Model Form 7

2.2 Activation Functions 8

2.3 Loss Functions 9

2.4 Parameterization 9

2.5 Network Fitting 10

2.6 Gradient Descent 11

2.7 Code Example 15

2.8 Vectorized Calculations 23

3 Digit Model **27**

3.1 Binomial Model 27

3.2 Multinomial Model 31

4 Multi-Layer NN Notes **55**

4.1 Notation Setup 55

4.2 Forward Propagation 59

4.3 Backward Propagation 61

5 Multi-Layer NN Model **65**

5.1 Generate Data 65

5.2 Functions 66

5.3 Test 75
5.4 Next Steps 78

Chapter 1

Intro

This is a write-up of iRisk's Spring 2024 project: Neural Nets from Scratch

---

The organization is:

2. Single-Layer NN Notes
    - intro notation & methodology for single-hidden layer neural network
3. Digit Model
    - GLMs on MNIST handwritten digits as an application intro
4. Multi-Layer NN Notes
    - intro notation & methodology for multi-hidden layer neural network
5. Multi-Layer NN Model
    - implementation of multi-layer neural network in R

1.0.1 Setup

```
knitr::opts_chunk$set(echo = TRUE)
set.seed(50)
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.3.1
```

```
## Warning: package 'readr' was built under R version 4.3.1

## Warning: package 'forcats' was built under R version 4.3.1

## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts
```

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.3.2
```

```
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.3.2

## Loading required package: Matrix

## Warning: package 'Matrix' was built under R version 4.3.2

##
## Attaching package: 'Matrix'
##
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
##
## Loaded glmnet 4.1-8
```

Chapter 2

Single-Layer NN Notes

These are notes for a single-layer neural network, mostly based off of *An Introduction to Statistical Learning.*

This chapter starts by outlaying some concepts and notation, then proceeds with an example of a single-layer neural network implemented 'by-hand'. The notation is quite non-standard and will be refined in later chapters.

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R.* New York: Springer, 2013.

2.1 Model Form

We have an input vector of p variables $X = \{x_1, x_2, \dots, x_p\}$, and an output scalar Y. We want to build a function $f : \mathbb{R}^p \to \mathbb{R}$ to approximate Y.

For a single layer NN, we have an input layer, hidden layer (with K activations), and output layer. Thus, the model's form is:

$$\begin{aligned} f(X) &= \beta_0 + \sum_{k=1}^{K} [\beta_k * h_k(X)] \\ &= \beta_0 + \sum_{k=1}^{K} \left[\beta_k * g\left(w_{k0} + \sum_{j=1}^{p} [w_{kj} * X_j] \right) \right] \end{aligned}$$

we have k indexing our hidden layer neurons, j indexing the weights within each neuron as they relate to each input variable $\{1, 2, ..., p\}$. $g(\cdot)$ is our activation function.

This model form is built in 2 steps:

$h_k(X)$ is known as the activation of the kth neuron of the hidden layer; it is denoted A_k:

$$A_k = h_k(X) = g\left(w_{k0} + \sum_{j=1}^{p}[w_{kj} * X_j]\right)$$

These get fed into the output layer, so that:

$$f(X) = \beta_0 + \sum_{k=1}^{K}(\beta_k * A_k)$$

2.2 Activation Functions

2.2.1 Sigmoid:

$$g(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}$$

2.2.2 ReLU:

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$

2.2.3 softmax:

$$f_s(X) = P(Y = s|X) = \frac{e^{Z_s}}{\sum_{l=1}^{w} e^{Z_l}}$$

$\widehat{\ }$ used for the output layer of a categorical response network.

2.3 Loss Functions

For a quantitative response variable, typical to use a squared-error loss function:

$$\sum_{i=1}^{n} [(y_i - f(x_i))^2]$$

For a qualitative / categorical response variable, typical to use cross-entropy:

$$-\sum_{i=1}^{n} \sum_{m=1}^{w} [y_{im} * \ln(f_m(x_i))]$$

Where w is the number of output categories. The behavior of this function is such that if the correct category is predicted as 1, the loss is 0. Otherwise, higher certainty for the correct category is rewarded for the correct answer, and lower certainty is punished.

The output matrix Y has been transformed using one-hot encoding in this circumstance, that's how there are multiple output dimensions (details).

Recall that y_{im} can only be 1 for the correct category; otherwise it is 0. So for each observation, only adding one number here to the total loss.

(3B1B also shows the sum of squared loss for the probability of each category)

2.4 Parameterization

For a single-layer neural network, we have 2 parameter matrices; one for the weights of the hidden layer, and one for the weights of the output layer. These are denoted **W** and **B**, respectively.

In **W**, each row represents an input (with the first row being the '1' input / the neuron's 'bias'); each column represents a neuron:

$$\mathbf{W} = \begin{bmatrix} w_{1,0} & w_{2,0} & \cdots & w_{K,0} \\ w_{1,1} & w_{2,1} & \cdots & w_{K,1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,p} & w_{2,p} & \cdots & w_{K,p} \end{bmatrix}$$

For **B**, each row is a hidden-layer neuron's activation (& a bias term).

If the output is quantitative, there is only 1 column for the output:

$$\mathbf{B} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

If the output is qualitative, there is one column per output category:

$$\mathbf{B} = \begin{bmatrix} \beta_{1,0} & \beta_{2,0} & \cdots & \beta_{w,0} \\ \beta_{1,1} & \beta_{2,1} & \cdots & \beta_{w,1} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{1,K} & \beta_{2,K} & \cdots & \beta_{w,K} \end{bmatrix}$$

We can combine $\mathbf{W}$ and $\mathbf{B}$ into one parameter vector:

$$\theta = \begin{bmatrix} w_{1,0} \\ w_{2,0} \\ \vdots \\ w_{K,p} \\ \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

Note that $\mathbf{W}$ is a $(p+1) \times K$ dimension matrix, and $\mathbf{B}$ is a $(K+1) \times w$ dimension matrix. So, θ has $(p+1) * K + (K+1) * w$ total parameters.

2.5 Network Fitting

Starting with a quantitative output. Our goal is to find:

$$\arg\min_{\theta} \sum_{i=1}^{n} \mathcal{L}(y_i, f(x_i))$$

We will use a scaled squared-error loss function:

$$\sum_{i=1}^{n} \frac{1}{2} \left[(y_i - f(x_i))^2 \right]$$

The scaling make for easier derivative-taking down the line. Recall that:

$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \left[\beta_k * g \left(w_{k0} + \sum_{j=1}^{p} [w_{kj} * x_{ij}] \right) \right]$$

So, we are trying to find:

$$\arg\min_{\theta} \sum_{i=1}^{n} \frac{1}{2} \left[y_i - \left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}) \right) \right]^2$$

We will denote the summation (our objective function) $\mathcal{C}(\theta)$.

This is nearly impossible to calculate by taking the derivative with respect to every variable and solving for a simultaneous 0; however, we can approximate solutions via gradient descent.

2.6 Gradient Descent

Our goal is to find $\arg\min_\theta \mathcal{C}(\theta)$ with gradient descent:

1. Start with a guess θ^0 for all parameters in θ, and set $t = 0$
2. Iterate until $\mathcal{C}(\theta)$ fails to decrease:
   - $\theta^{t+1} \leftarrow \theta^t - \rho * \nabla \mathcal{C}(\theta)$

ρ is our learning rate: it controls how quickly we respond to the gradient. $\nabla \mathcal{C}(\theta)$ points in the direction of the greatest increase, so we subtract it to move in the direction of the greatest decrease. Our change in parameter values is proportional to both the learning rate and the gradient magnitude.

The last step for us is taking the gradient. In our parameter vector, we have two 'types' of parameters: those that came from $\mathbf{W}$, and those that came from $\mathbf{B}$. These can be split further into those which are intercept terms (—> simpler derivatives) or not.

We will start by manipulating the notation of our objective function to make it easier to work with:

- let $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$
  - so z_{ik} is the ith input of the activation function of the kth hidden-layer neuron
- let $\hat{y}_i = \beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik})$
  - so $\hat{y}_i$ is our ith prediction
- let $\hat{\epsilon}_i = \hat{y}_i - y_i$
  - so $\hat{\epsilon}_i$ is our ith residual
  - note that $\hat{\epsilon}_i = \left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i$

  - (against convention here because this is a negative residual; playing fast & loose w/ notation)

- because $(a-b)^2 = (b-a)^2$, we will flip y and $\widehat{y}$ in our objective function

So we have:

$$\mathcal{C}(\theta) = \sum_{i=1}^{n} \frac{1}{2} \left[y_i - \left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}) \right) \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\widehat{y}_i - y_i \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\widehat{\epsilon}_i \right]^2$$

Taking our derivatives:

2.6.1 Beta: Intercept

$$\frac{\partial \mathcal{C}}{\partial \beta_0} = \frac{\partial}{\partial \beta_0} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]$$

$$= \sum_{i=1}^{n} \widehat{\epsilon}_i$$

2.6.2 Beta: Coefficients

$$\begin{aligned}
\frac{\partial \mathcal{C}}{\partial \beta_k} &= \frac{\partial}{\partial \beta_k} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2 \\
&= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \frac{\partial}{\partial \beta_k} [\beta_k * g(z_{ik})] \\
&= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] g(z_{ik}) \\
&= \sum_{i=1}^{n} \hat{\epsilon}_i \; g(z_{ik})
\end{aligned}$$

2.6.3 W: Intercepts

$$\begin{aligned}
\frac{\partial \mathcal{C}}{\partial w_{k0}} &= \frac{\partial}{\partial w_{k0}} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2 \\
&= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \frac{\partial}{\partial w_{k0}} [\beta_k * g(z_{ik})] \\
&= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \; \frac{\partial}{\partial w_{k0}} g(z_{ik}) \\
&= \sum_{i=1}^{n} \hat{\epsilon}_i \; \beta_k \; g'(z_{ik})
\end{aligned}$$

note that $\frac{\partial}{\partial w_{k0}} z_{ik} = \frac{\partial}{\partial w_{k0}} \left[w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij} \right] = 1$

2.6.4 W: Coefficients

$$\frac{\partial \mathcal{C}}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \frac{\partial}{\partial w_{kj}} [\beta_k * g(z_{ik})]$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \ \frac{\partial}{\partial w_{kj}} g(z_{ik})$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \ g'(z_{ik}) \ \frac{\partial}{\partial w_{kj}} z_{ik}$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \ g'(z_{ik}) \ x_{ij}$$

$$= \sum_{i=1}^{n} \hat{\epsilon}_i \ \beta_k \ g'(z_{ik}) \ x_{ij}$$

note that $\frac{\partial}{\partial w_{kj}} z_{ik} = \frac{\partial}{\partial w_{kj}} \left[w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij} \right] = x_{ij}$

2.6.5 Combining

Given:

$$\theta = \begin{bmatrix} w_{1,0} \\ w_{2,0} \\ \vdots \\ w_{K,p} \\ \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

and

$$\mathcal{C}(\theta) = \sum_{i=1}^{n} \frac{1}{2} \left[\hat{\epsilon}_i\right]^2$$

We have computed:

$$\nabla \mathcal{C}(\theta) = \begin{bmatrix} \frac{\partial \mathcal{C}}{\partial w_{1,0}} \\ \frac{\partial \mathcal{C}}{\partial w_{2,0}} \\ \vdots \\ \frac{\partial \mathcal{C}}{\partial w_{1,1}} \\ \vdots \\ \frac{\partial \mathcal{C}}{\partial w_{K,p}} \\ \frac{\partial \mathcal{C}}{\partial \beta_0} \\ \frac{\partial \mathcal{C}}{\partial \beta_1} \\ \vdots \\ \frac{\partial \mathcal{C}}{\partial \beta_K} \end{bmatrix} = \sum_{i=1}^{n} \begin{bmatrix} \hat{\epsilon}_i \ \beta_1 \ g'(z_{i1}) \\ \hat{\epsilon}_i \ \beta_2 \ g'(z_{i2}) \\ \vdots \\ \hat{\epsilon}_i \ \beta_1 \ g'(z_{i1}) \ x_{i1} \\ \vdots \\ \hat{\epsilon}_i \ \beta_K \ g'(z_{ik}) \ x_{ip} \\ \hat{\epsilon}_i \\ \hat{\epsilon}_i \ g(z_{i1}) \\ \vdots \\ \hat{\epsilon}_i \ g(z_{ik}) \end{bmatrix}$$

2.7 Code Example

A simple example of using a small single-layer neural network to act on simulated data:

2.7.1 Generate Data

For now, having 3 inputs and combining them to create y, with a random error term. Would like to tweak the setup eventually.

```
## create data:
n <- 1000
p <- 3

# initialize Xs
X <- data.frame(X1 = runif(n = n, min = -10, max = 10),
                X2 = rnorm(n = n, mean = 0, sd = 10),
                X3 = rexp(n = n, rate = 1)) %>%
  as.matrix(nrow = n,
            ncol = p)

# get response
Y <- X[, 1] + 10 * sin(X[, 2])^2 + 10 * X[, 3] + rnorm(n = 1000)
```

2.7.2 Parameter Setup

We will have 2 hidden-layer neurons and a single quantitative output, so $\mathbf{W}$ will be 4×2 and $\mathbf{B}$ will be 3×1:

```
## NN properties
K <- 2

## initialize parameter matrices
W <- matrix(data = runif(n = (p + 1) * K, min = -1, max = 1),
            nrow = (p + 1),
            ncol = K)

B <- matrix(data = runif(n = (K + 1), min = -1, max = 1),
            nrow = (K + 1),
            ncol = 1)

## Specify Link Functions & Derivatives:
# identity
# g <- function(x) {x}
# g_prime <- function(x) {1}

# sigmoid
g <- function(x) {1 / (1 + exp(-x))}
g_prime <- function(x) {exp(-x) / (1 + exp(-x))^2}

# ReLU
# g <- function(x) {if (x < 0) {0} else {x}}
# g_prime <- function(x) {if (x < 0) {0} else {1}}
```

2.7.3 Output

How the NN will calculate the output:

```
## create output function
NN_output <- function(X, W, B) {
  cbind(1, g(cbind(1, X) %*% W)) %*% B
}

example <- NN_output(X = X,
                     W = W,
                     B = B)

example[1:5]
```

```
## [1] -0.4570299 -0.8227519 -1.0352693 -0.5013235 -0.7197220
```

2.7.4 Gradient Descent

for now, looping through each observation's gradient then taking the sum — much slower than using matrix/arrays, which will eventually happen:

```
GD_iteration <- function(X, Y, W, B, rho = 1) {

  ## get errors
  errors <- NN_output(X = X, W = W, B = B) - Y

  ## get each obs' gradient
  gradient_array_W <- array(dim = c((p + 1), K, nrow(X)))
  gradient_array_B <- array(dim = c((K + 1), 1, nrow(X)))

  for (i in 1:nrow(X)) {

    ## W
    errors_W <-  matrix(errors[i],
                        nrow = (p + 1),
                        ncol = K)

    B_W <- matrix(B[-1, ],
                  nrow = (p + 1),
                  ncol = K,
                  byrow = TRUE)

    X_W <- matrix(c(1, X[i, ]),
                  nrow = (p + 1),
                  ncol = K,
                  byrow = FALSE)

    g_prime_z_W <- apply(X = c(1, X[i, ]) %*% W,
                         MARGIN = 2,
                         FUN = g_prime) %>%
      matrix(nrow = (p + 1),
             ncol = K,
             byrow = FALSE)

    del_W <- errors_W * B_W * g_prime_z_W * X_W

    gradient_array_W[ , , i] <- del_W

    ## B
```

```
    errors_B <- matrix(errors[i],
                       nrow = (K + 1),
                       ncol = 1)

    g_z_B <- apply(X = c(1, X[i, ]) %*% W,
                   MARGIN = 2,
                   FUN = g) %>%
      c(1, .) %>%
      matrix(nrow = (K + 1),
             ncol = 1)

    del_B <- errors_B * g_z_B

    gradient_array_B[ , , i] <- del_B
  }

  ## get gradients
  del_W_all <- apply(X = gradient_array_W,
                     MARGIN = c(1, 2),
                     FUN = mean)

  del_B_all <- apply(X = gradient_array_B,
                     MARGIN = c(1, 2),
                     FUN = mean)

  ## perform iteration
  W_out <- W - rho * del_W_all
  B_out <- B - rho * del_B_all

  ## return
  return(list(W = W_out,
              B = B_out))
}

## test run
iteration <- GD_iteration(X = X,
                          Y = Y,
                          W = W,
                          B = B,
                          rho = 1 / 100)

## in loss:
sum((NN_output(X = X, W = W, B = B) - Y)^2)
```

```
## [1] 369063.7
```

```
## out loss:
sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)
```

```
## [1] 362779.6
```

2.7.5 Iterate

Employ gradient descent until objective function stops decreasing:

```
threshold <- 1

done_decreasing <- FALSE

iteration <- list()
iterations <- list()

iteration$W <- W
iteration$B <- B

iter <- 1

initial_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)

while ((!done_decreasing) & (iter < 301)) {
  ## get input loss
  in_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)

  ## perform iteration
  iteration <- GD_iteration(X = X,
                            Y = Y,
                            W = iteration$W,
                            B = iteration$B,
                            rho = 1 / 100)

  ## get output loss
  out_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)

  ## evaluate
  if (abs(in_objective - out_objective) < threshold) {
    done_decreasing <- TRUE
  }

  # print(iter)
  # print(out_objective)
```

```
  iterations[[iter]] <- cbind(matrix(iteration$W, nrow = 1),
                              matrix(iteration$B, nrow = 1))

  iter <- iter + 1
}

final_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)

## number of iterations
iter <- iter - 1
iter
```

```
## [1] 300
```

```
## loss improvement ratio
initial_objective
```

```
## [1] 369063.7
```

```
final_objective
```

```
## [1] 99080.37
```

```
final_objective / initial_objective
```

```
## [1] 0.2684641
```

```
## input W
W
```

```
##            [,1]       [,2]
## [1,]  0.7875380  0.3591272
## [2,]  0.2717206 -0.8873001
## [3,]  0.7644715  0.1074179
## [4,] -0.7902263 -0.5297394
```

```
## output W
iteration$W
```

```
##             [,1]       [,2]
## [1,]  1.82875350  0.6259373
## [2,]  5.84398877  0.5638595
## [3,] -0.08899907 -0.1430374
## [4,]  7.95982586  2.1499130
```

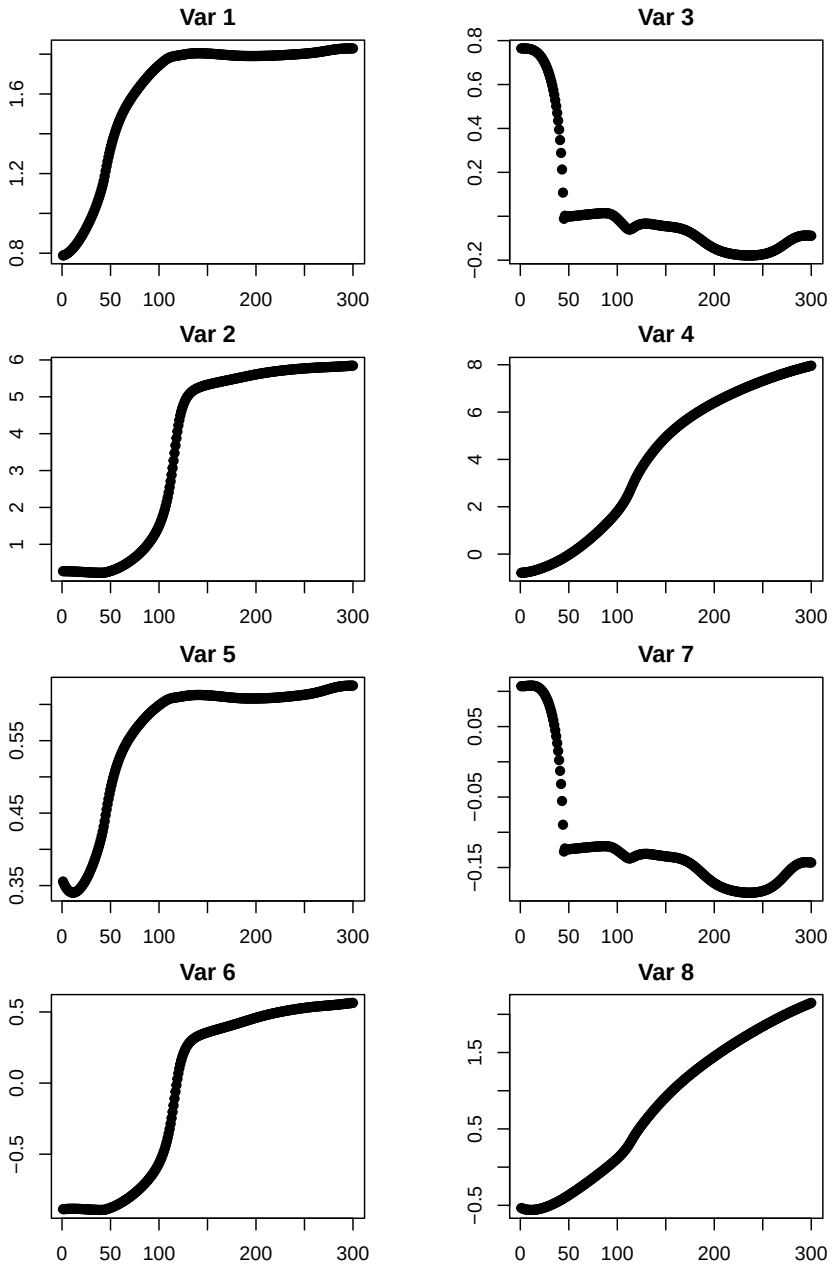
```
## input B
B
```

```
##             [,1]
## [1,] -0.5508596
## [2,]  0.1190147
## [3,] -0.4893450
```

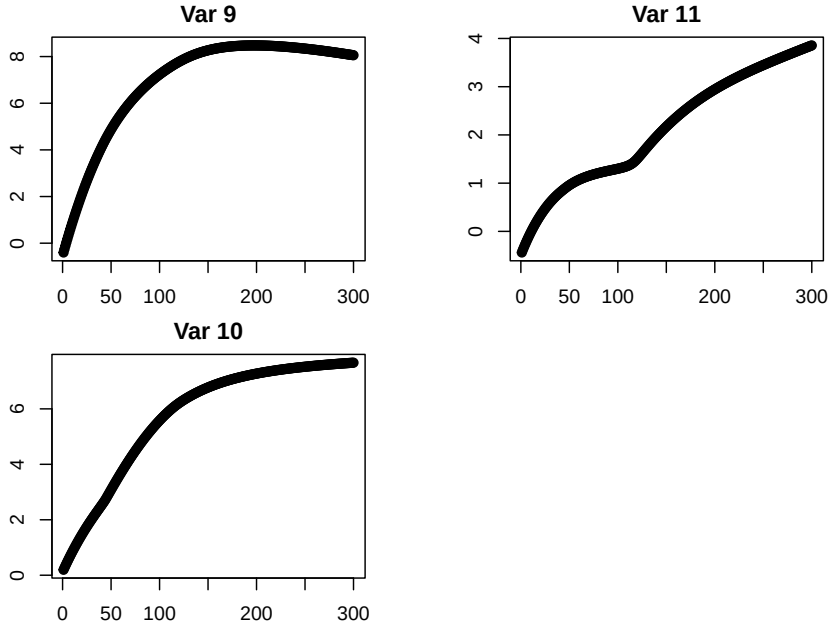
```
## output B
iteration$B
```

```
##          [,1]
## [1,] 8.057293
## [2,] 7.665410
## [3,] 3.855817
```

```
## plots
iterations <- do.call(rbind, iterations)

par(mfcol = c(2, 2))
par(mar = c(2, 4.1, 2, 2.1))

for (i in 1:ncol(iterations)) {
  plot(x = 1:iter,
       y = iterations[ , i],
       pch = 19,
       main = paste("Var", i),
       ylab = "",
       xlab = "")
}
```




```
## return to default
par(mfcol = c(1, 1))
```




```
par(mar = c(5.1, 4.1, 4.1, 2.1))
```

2.8 Vectorized Calculations

A wayward attempt at deriving the matrix notation for vectorized operations that result in a simplified $\nabla \mathcal{C}(\theta)$ by avoiding summations, to be replaced by strategic matrix multiplications.

This attempt was abandoned; there's more fertile ground in re-defining some notation and pursuing multi-layer networks (later chapters).

2.8.1 Notation Setup

We have our input matrix X:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix}$$

each row represents an obs (1-n)

each col represents a var (1-p)

---

our Weights matrix W:

$$W = \begin{bmatrix} w_{1,1} & w_{2,1} & \cdots & w_{K,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,p} & w_{2,p} & \cdots & w_{K,p} \end{bmatrix}$$

each col represents a neuron (1-K)

each row represents a var (1-p)

---

our output layer weight matrix B:

$$B = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

each row represents a neuron (1-K)

---

our bias matrices b_1, b_2:

$$b_1 = \begin{bmatrix} | & | & & | \\ w_{1,0} & w_{2,0} & \cdots & w_{K,0} \\ | & | & & | \end{bmatrix}$$

$$b_2 = \begin{bmatrix} | \\ \beta_0 \\ | \end{bmatrix}$$

for b_1, each col has a height of n and represents a neuron (1-K)

for b_2, the col has a height of K

---

our target layer matrix Y:

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

---

also, we have defined: $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$ to get the activation function's input for a given neuron. We can take the neurons in their totality to define Z:

$$Z = X \cdot W + b_1$$

each row represents an obs (1-n)

each col represents a neuron (1-K)

---

our model output is $\hat{Y}$:

$$\begin{aligned} \hat{Y} = f(X) &= g(Z) \cdot B + b_2 \\ &= g(X \cdot W + b_1) \cdot B + b_2 \end{aligned}$$

$$= g\left(\begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{2,1} & \cdots & w_{K,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,p} & w_{2,p} & \cdots & w_{K,p} \end{bmatrix} + \begin{bmatrix} | & | & & | \\ w_{1,0} & w_{2,0} & \cdots & w_{K,0} \\ | & | & & | \end{bmatrix} \right) \cdot \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_K \end{bmatrix} + \begin{bmatrix} | \\ \beta_0 \\ | \end{bmatrix}$$

$$= \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}$$

---

our error matrix:

$$\hat{} = Y - \hat{Y}$$

2.8.2 gradients

We can now vectorize our gradient, $\nabla\mathcal{C}(\theta)$:

2.8.2.1 b_2

$$\begin{aligned}\nabla\mathcal{C}(b_2) &= \sum_{i=1}^{n} \hat{\epsilon}_i \\ &= [\mathbf{1}]^{\mathcal{T}^{\wedge}}\end{aligned}$$

2.8.2.2 B

$$\nabla\mathcal{C}(B) = \sum_{i=1}^{n} \begin{bmatrix} \hat{\epsilon}_i \; g(z_{i1}) \\ \hat{\epsilon}_i \; g(z_{i2}) \\ \vdots \\ \hat{\epsilon}_i \; g(z_{ik}) \end{bmatrix}$$

$$= [g(Z)]^T \cdot^{\wedge}$$

2.8.2.3 b_1

$$\nabla\mathcal{C}(b_1) = \sum_{i=1}^{n} \begin{bmatrix} \hat{\epsilon}_i \; \beta_1 \; g'(z_{i1}) \\ \hat{\epsilon}_i \; \beta_2 \; g'(z_{i2}) \\ \vdots \\ \hat{\epsilon}_i \; \beta_K \; g'(z_{iK}) \end{bmatrix}$$

$$= \left([g'(Z)]^T \cdot^{\wedge}\right) \odot B$$

where $\odot$ is the element-wise multiplication operator

2.8.2.4 W

$$\nabla\mathcal{C}(W) = \sum_{i=1}^{n} \begin{bmatrix} \hat{\epsilon}_i \; \beta_1 \; g'(z_{i1}) \; x_{i1} \\ \hat{\epsilon}_i \; \beta_2 \; g'(z_{i2}) \; x_{i2} \\ \vdots \\ \hat{\epsilon}_i \; \beta_K \; g'(z_{iK}) \; x_{ip} \end{bmatrix}$$

$$= \; ???$$

Chapter 3

Digit Model

In preparation for neural networks, we take a brief chapter to run other models on MNIST hand-written data. First we will run a binomial GLM on each digit and keep the maximum outputted likelihood as the predicted digit, then we will run a multinomial GLM to assess the likelihood of every digit simultaneously.

This chapter can be safely skipped / ignored.

3.1 Binomial Model

3.1.1 Setup

```
# Loads the MNIST dataset, saves as an .RData file if not in WD
if (!(file.exists("mnist_data.RData"))) {

  # ## installs older python version
  # reticulate::install_python("3.10:latest")
  # keras::install_keras(python_version = "3.10")
  # ## re-loads keras
  # library(keras)

  ## get MNIST data
  mnist <- dataset_mnist()
  ## save to WD as .RData
  save(mnist, file = "mnist_data.RData")

} else {
  ## read-in MNIST data
```

```
  load(file = "mnist_data.RData")
}

# Access the training and testing sets
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y

rm(mnist)
```

```
## plot function, from OG data
plot_mnist <- function(plt) {
  ## create image
  image(x = 1:28,
        y = 1:28,
        ## image is oriented incorrectly, this fixes it
        z = t(apply(plt, 2, rev)),
        ## 255:0 puts black on white canvas,
        ## changing to 0:255 puts white on black canvas
        col = gray((255:0)/255),
        axes = FALSE)

  ## create plot border
  rect(xleft = 0.5,
       ybottom = 0.5,
       xright = 28 + 0.5,
       ytop = 28 + 0.5,
       border = "black",
       lwd = 1)
}
```

```
## train data

# initialize matrix
x_train_2 <- matrix(nrow = nrow(x_train),
                    ncol = 28*28)

## likely a faster way to do this in the future
for (i in 1:nrow(x_train)) {
  ## get each layer's matrix image, stretch to 28^2 x 1
  x_train_2[i, ] <- matrix(x_train[i, , ], 1, 28*28)
}

x_train_2 <- x_train_2 %>%
```

```
  as.data.frame()

## test data
x_test_2 <- matrix(nrow = nrow(x_test),
                   ncol = 28*28)

for (i in 1:nrow(x_test)) {
  x_test_2[i, ] <- matrix(x_test[i, , ], 1, 28*28)
}

x_test_2 <- x_test_2 %>%
  as.data.frame()


## re-scale data
x_train_2 <- x_train_2 / 256
x_test_2 <- x_test_2 / 256

## response
# x_test_2$y <- y_test
# x_train_2$y <- y_train
```

3.1.2 Model

```
## for speed
# n <- nrow(x_train_2)
n <- 100
indices <- sample(x = 1:nrow(x_train_2),
                  size = n)

## init data
x_glm <- x_train_2[indices, ]
y_glm <- y_train[indices]
train_pred <- list()

## drop cols with all 0s
x_glm <- x_glm[, (colSums(x_glm) > 0)]
```

```
## 10 model method
for (i in 0:9) {
print(i)

y_glm_i = (y_glm == i)
```

```
init_model <- cv.glmnet(x = x_glm %>% as.matrix,
                        y = y_glm_i,
                        family = binomial,
                        alpha = 1)

train_pred[[i + 1]] <- predict(init_model,
                               x_glm %>% as.matrix,
                               s = init_model$lambda.min,
                               type = "response")
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

```
## format results
predictions <- data.frame(train_pred)
names(predictions) <- c("zero",
                        "one",
                        "two",
                        "three",
                        "four",
                        "five",
                        "six",
                        "seven",
                        "eight",
                        "nine")

#write.csv(predictions, "pred.csv", row.names = FALSE)

## convert to numeric
max_col <- apply(X = predictions,
                 MARGIN = 1,
                 FUN = function(x) names(x)[which.max(x)])

word_to_number <- c("zero" = 0,
                    "one" = 1,
```

```
                    "two" = 2,
                    "three" = 3,
                    "four" = 4,
                    "five" = 5,
                    "six" = 6,
                    "seven" = 7,
                    "eight" = 8,
                    "nine" = 9)

preds <- word_to_number[max_col] %>% as.numeric

## confusion matrix
table(y_glm, preds)
```

```
##       preds
## y_glm  0  1  2  3  4  5  6  7  8  9
##     0 10  0  0  0  0  0  0  0  0  0
##     1  0 12  0  0  0  0  0  0  0  0
##     2  0  0  7  0  0  0  0  0  0  0
##     3  0  0  0  9  0  0  0  0  0  0
##     4  0  0  0  0 10  0  0  0  0  0
##     5  0  0  0  0  0 16  0  0  0  0
##     6  0  0  0  0  0  0 12  0  0  0
##     7  0  0  0  0  0  0  0 11  0  0
##     8  2  0  1  0  0  3  0  0  3  0
##     9  0  0  0  0  2  0  0  0  0  2
```

```
## misclassification rate
mean(!(y_glm == preds))
```

```
## [1] 0.08
```

3.2 Multinomial Model

3.2.1 Setup

```
# Loads the MNIST dataset, saves as an .RData file if not in WD
if (!(file.exists("mnist_data.RData"))) {

  # ## installs older python version
  # reticulate::install_python("3.10:latest")
```

```
  # keras::install_keras(python_version = "3.10")
  # ## re-loads keras
  # library(keras)

  ## get MNIST data
  mnist <- dataset_mnist()
  ## save to WD as .RData
  save(mnist, file = "mnist_data.RData")

} else {
  ## read-in MNIST data
  load(file = "mnist_data.RData")
}

# Access the training and testing sets
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y

rm(mnist)
```

```
## plot function
plot_mnist_array <- function(plt, main_label = NA, color = FALSE, dim_n = 28) {

  ## setup color
  if (color == TRUE) {
      colfunc <- colorRampPalette(c("red", "white", "blue"))

      min_abs <- -max(abs(range(plt)))
      max_abs <- max(abs(range(plt)))

      col <- colfunc(256)
  } else {
    col <- gray((255:0)/255)
    min_abs <- 0
    max_abs <- 255
    }

  ## create image
  image(x = 1:dim_n,
        y = 1:dim_n,
        ## image is oriented incorrectly, this fixes it
        z = t(apply(plt, 2, rev)),
        col = col,
```

```
        zlim = c(min_abs, max_abs),
        axes = FALSE,
        xlab = NA,
        ylab = NA)

  ## create plot border
  rect(xleft = 0.5,
       ybottom = 0.5,
       xright = 28 + 0.5,
       ytop = 28 + 0.5,
       border = "black",
       lwd = 1)

  ## display prediction result
  text(x = 2,
       y = dim_n - 3,
       labels = ifelse(is.na(main_label),
                       "",
                       main_label),
       col = ifelse(color == TRUE,
                    "black",
                    "red"),
       cex = 1.5)
}
```

```
## train data

# initialize matrix
x_train_2 <- matrix(nrow = nrow(x_train),
                    ncol = 28*28)

## likely a faster way to do this in the future
for (i in 1:nrow(x_train)) {
  ## get each layer's matrix image, stretch to 28^2 x 1
  x_train_2[i, ] <- matrix(x_train[i, , ], 1, 28*28)
}

x_train_2 <- x_train_2 %>%
  as.data.frame()

## test data
x_test_2 <- matrix(nrow = nrow(x_test),
                   ncol = 28*28)

for (i in 1:nrow(x_test)) {
```

```
  x_test_2[i, ] <- matrix(x_test[i, , ], 1, 28*28)
}

x_test_2 <- x_test_2 %>%
  as.data.frame()


## re-scale data
x_train_2 <- x_train_2 / 256
x_test_2 <- x_test_2 / 256

## response
# x_test_2$y <- y_test
# x_train_2$y <- y_train
```

3.2.2 Model

3.2.2.1 train

```
## set training data size
# n <- nrow(x_train_2)
n <- 100

indices <- sample(x = 1:nrow(x_train_2),
                  size = n)

## init data
x_multi <- x_train_2[indices, ]
y_multi <- y_train[indices]

## drop cols with all 0s
#x_multi <- x_multi[, (colSums(x_multi) > 0)]
```

```
## for the sake of the coefficients viz, setting alpha = 0
init_model <- cv.glmnet(x = x_multi %>% as.matrix,
                        y = y_multi %>% factor,
                        family = "multinomial",
                        alpha = 0)
```

```
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
```

```
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
```

```r
multi_model <- predict(init_model,
                       x_multi %>% as.matrix,
                       s = init_model$lambda.min,
                       type = "response")
```

```r
## format results
preds_init <- multi_model[, , 1] %>%
  as.data.frame()

preds <- apply(X = preds_init,
               MARGIN = 1,
               FUN = function(x) names(which.max(x)) %>% as.numeric)

## TRAIN confusion matrix
table(y_multi, preds)
```

```
##         preds
## y_multi  0  1  2  3  4  5  6  7  8  9
##       0 11  0  0  0  0  0  0  0  1  0
##       1  0 14  0  0  0  0  0  0  0  0
##       2  0  1  4  0  0  0  0  0  0  0
##       3  0  0  0 12  0  0  0  0  0  0
##       4  0  0  0  0 11  0  0  0  0  0
##       5  0  0  0  0  0  4  0  0  0  0
##       6  0  0  0  0  0  0 11  0  0  0
##       7  0  2  0  0  0  0  0  6  0  0
##       8  0  1  0  0  0  0  0  0 13  0
##       9  0  0  0  0  1  0  0  0  0  8
```

```
## TRAIN misclassification rate
mean(!(y_multi == preds))
```

```
## [1] 0.06
```

3.2.2.2 test

```
## pre-process data
x_multi_test <- x_test_2 %>%
  select(all_of(names(x_multi)))

## get preds
multi_model_test <- predict(init_model,
                            x_multi_test %>% as.matrix,
                            s = init_model$lambda.min,
                            type = "response")
```

```
## format results
preds_init_test <- multi_model_test[, , 1] %>%
  as.data.frame()

preds_test <- apply(X = preds_init_test,
                    MARGIN = 1,
                    FUN = function(x) names(which.max(x)) %>% as.numeric)

## TEST confusion matrix
table(y_test, preds_test)
```

```
##       preds_test
```

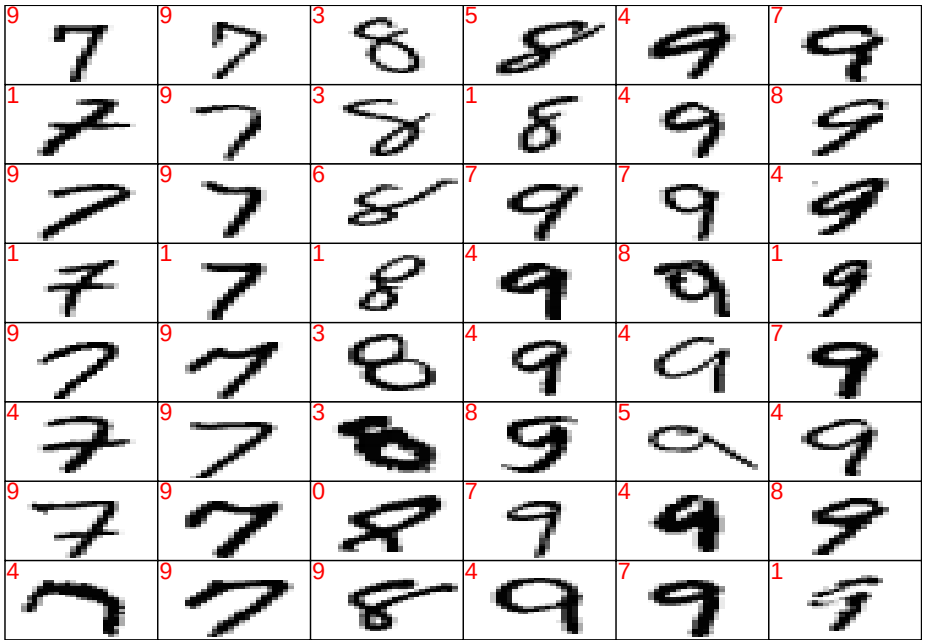
```
## y_test    0    1    2    3    4    5    6    7    8    9
##      0  859    1    2   10    7   16   26    1   55    3
##      1    0 1102    0    5    4    1    4    0   19    0
##      2   62  102  243  226   55   68   93   11  170    2
##      3   12   46   35  693    3    5    9   23  178    6
##      4    7   36    2    0  817   13   39    6   23   39
##      5   78   94    8  119   40  102   48   16  364   23
##      6   37   19    3   28   94    2  760    0   15    0
##      7    7   91    0    2   37    1    1  614   38  237
##      8   19   74    2   41    7   20   16    4  768   23
##      9   19   41    0    8  289    8    0  129  127  388
```

```
## TEST misclassification rate
mean(!(y_test == preds_test))
```

```
## [1] 0.3654
```

```
## sort vectors so outputs are grouped
x_test_sort <- x_test[order(y_test), , ]
y_test_sort <- y_test[order(y_test)]
preds_test_sort <- preds_test[order(y_test)]

## get misclassified obs
wrong <- which(!(y_test_sort == preds_test_sort))

## plot a sample of misclassified obs
plot_wrong <- wrong[sample(x = 1:length(wrong), size = 3*8*6)] %>%
  sort()

## plot params
par(mfcol = c(8, 6))
par(mar = c(0, 0, 0, 0))

for (i in plot_wrong) {
  plot_mnist_array(plt = x_test_sort[i, , ],
                   main_label = preds_test_sort[i])
}
```

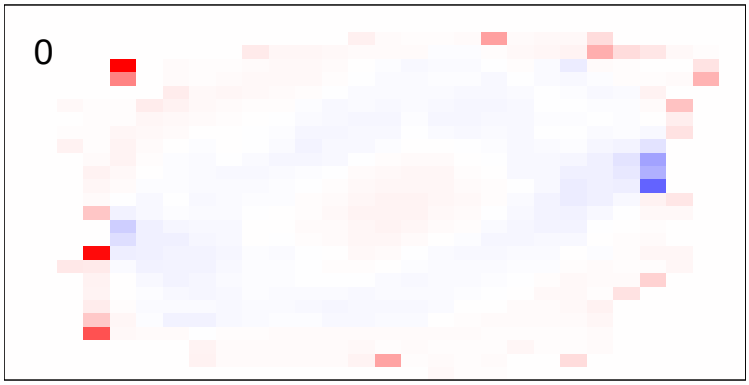

6 1 8 8 8 8
6 8 5 3 0 6
5 1 6 3 0 1
5 3 6 5 4 5
6 0 0 6 4 1
8 3 1 6 5 8
1 4 7 8 7 0
8 1 3 0 8 8
1 8 8 9 0 3
0 9 2 0 2 0
7 8 8 8 0 4
6 8 8 8 9 4
8 3 4 0 8 4
9 8 3 1 3 4
0 8 0 1 0 9
8 3 8 9 1 9

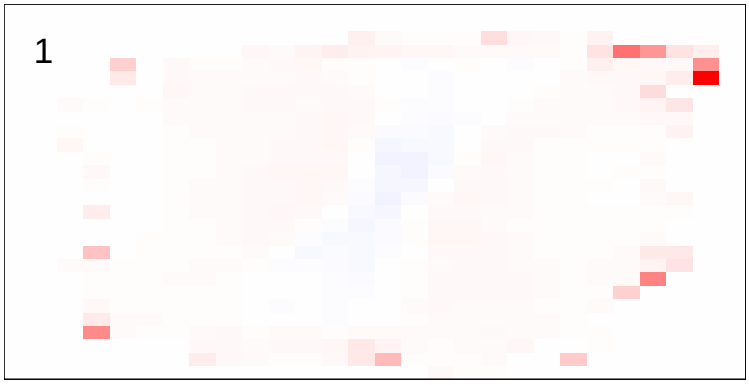


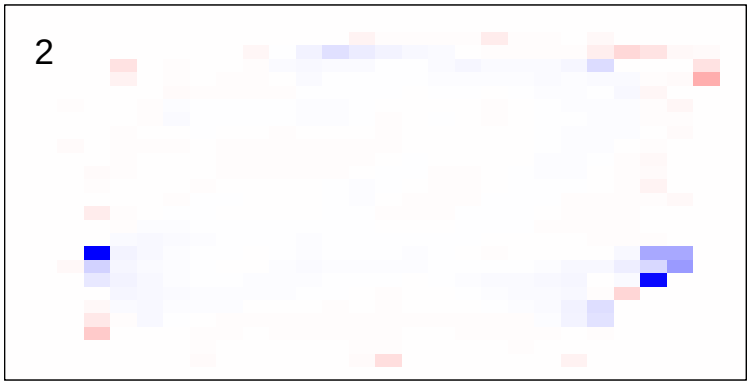

```
par(mfcol = c(1, 1))
```

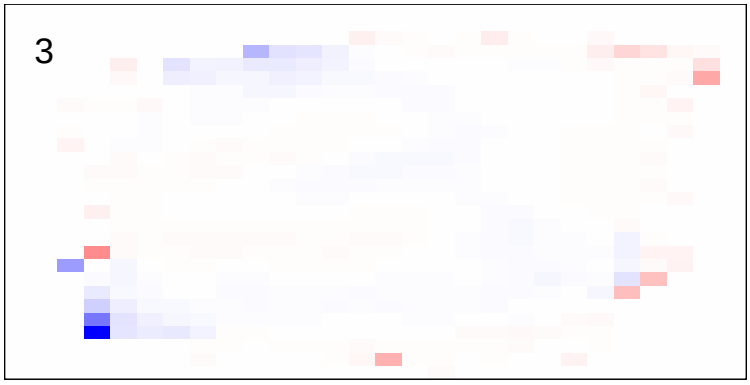
3.2.3 model heatmaps

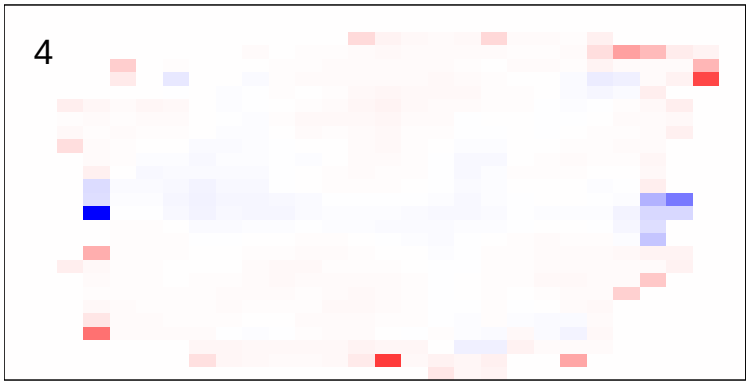
```
## get coefficients into matrices
model_coef <- coef(init_model, s = init_model$lambda.min) %>%
  lapply(as.matrix) %>%
  lapply(function(x) matrix(x[-1, ], nrow = 28, ncol = 28)) %>%
  ## take sigmoid activation just to help viz
  lapply(function(x) 1 / (1 + exp(-x)) - 0.5)

mapply(FUN = plot_mnist_array,
       plt = model_coef,
       main_label = names(model_coef),
       color = TRUE)
```

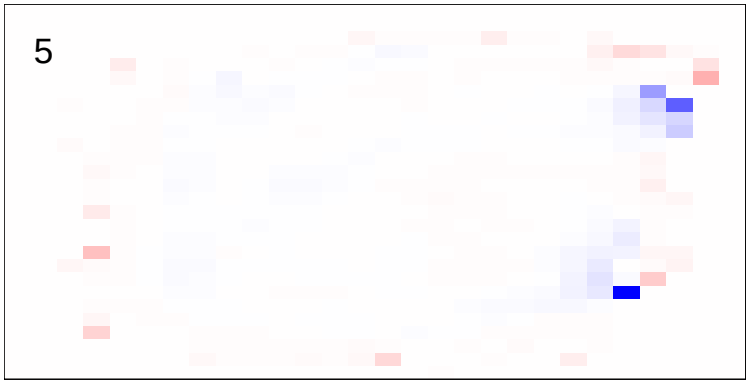

0

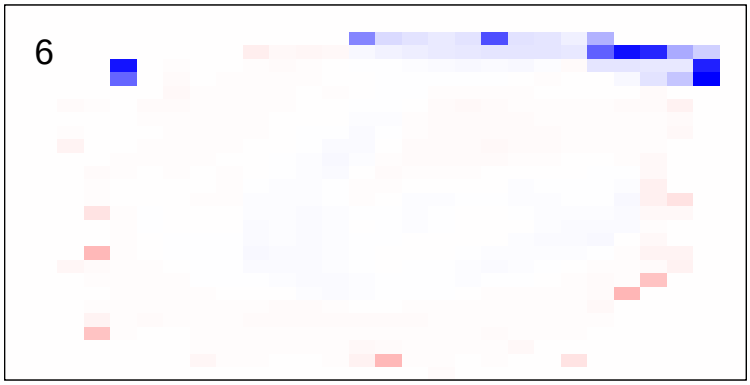

1

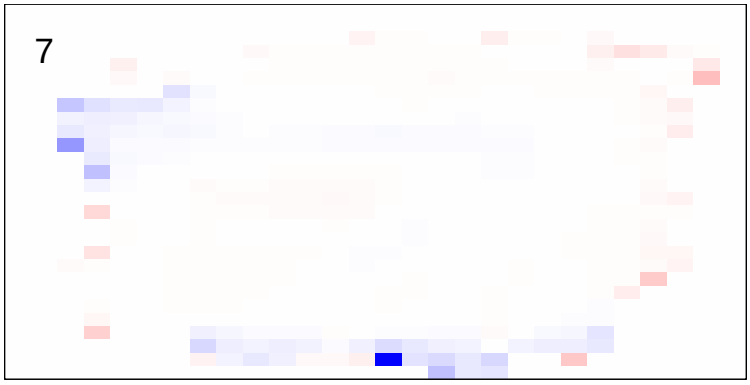

2

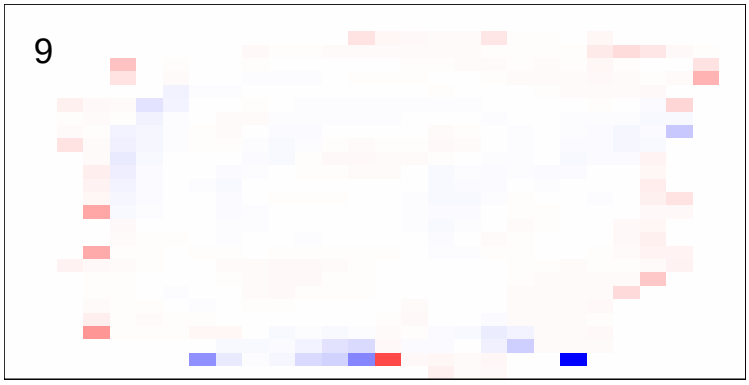

3


4


5


6


7




```
## $`0`
## NULL
##
## $`1`
## NULL
```

```
##
## $`2`
## NULL
##
## $`3`
## NULL
##
## $`4`
## NULL
##
## $`5`
## NULL
##
## $`6`
## NULL
##
## $`7`
## NULL
##
## $`8`
## NULL
##
## $`9`
## NULL
```

3.2.4 no outside cells model

earlier runs of the above sections revealed that for a regularization method that does not perform variable selection, odd importance is given to outermost cell for prediction. Thus, those will be removed:

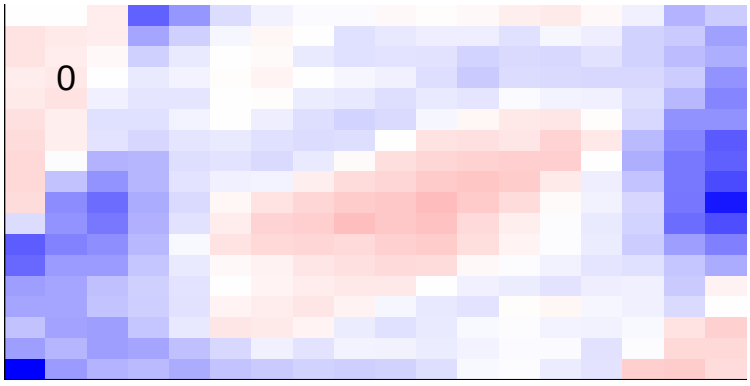
```
## set training data size
# n <- nrow(x_train_2)
n <- 100

indices <- sample(x = 1:nrow(x_train_2),
                  size = n)

## init data
x_multi <- x_train_2[indices, ]
y_multi <- y_train[indices]

## drop outer cells
x_multi <- x_multi[, rep(seq(146, 622, 28), each = 18) + rep(0:17, times = 18)]
```

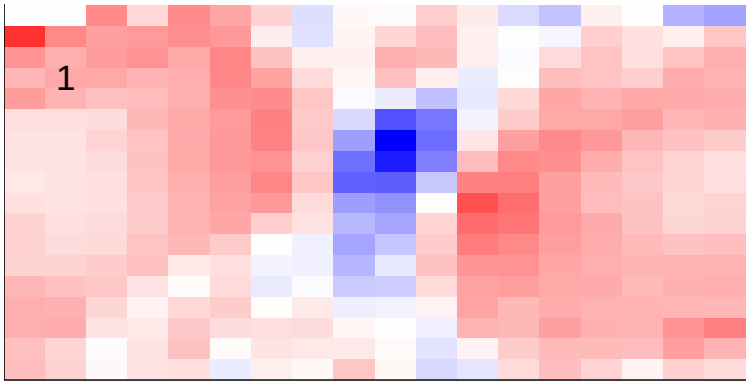
```
## for the sake of the coefficients viz, setting alpha = 0
init_model <- cv.glmnet(x = x_multi %>% as.matrix,
                        y = y_multi %>% factor,
                        family = "multinomial",
                        alpha = 0)
```

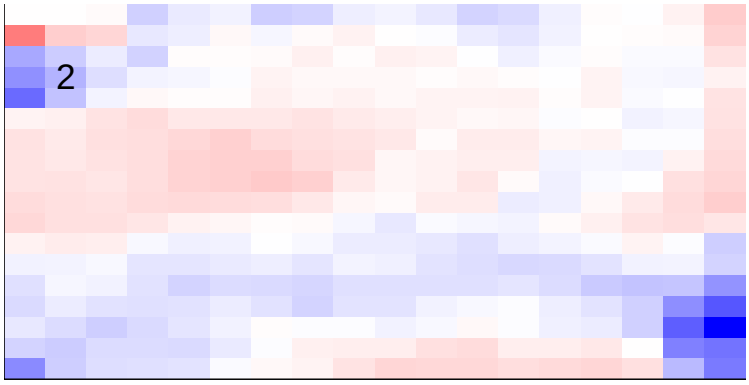
```
## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground

## Warning in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs, : one
## multinomial or binomial class has fewer than 8 observations; dangerous ground
```

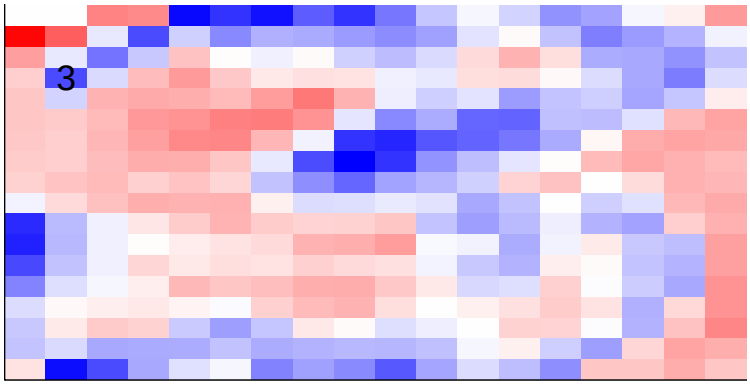
```
multi_model <- predict(init_model,
                       x_multi %>% as.matrix,
                       s = init_model$lambda.min,
                       type = "response")
```

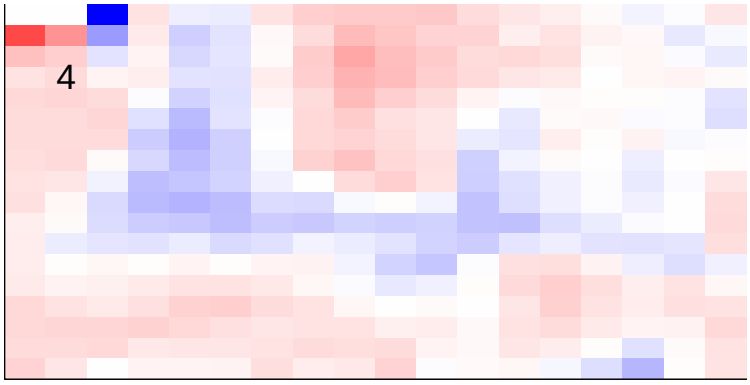
```
## get coefficients into matrices
model_coef <- coef(init_model, s = init_model$lambda.min) %>%
  lapply(as.matrix) %>%
  lapply(function(x) matrix(x[-1, ], nrow = 18, ncol = 18)) %>%
  ## take sigmoid activation just to help viz
  lapply(function(x) 1 / (1 + exp(-x)) - 0.5)

mapply(FUN = plot_mnist_array,
       plt = model_coef,
       main_label = names(model_coef),
       color = TRUE,
       dim_n = 18)
```

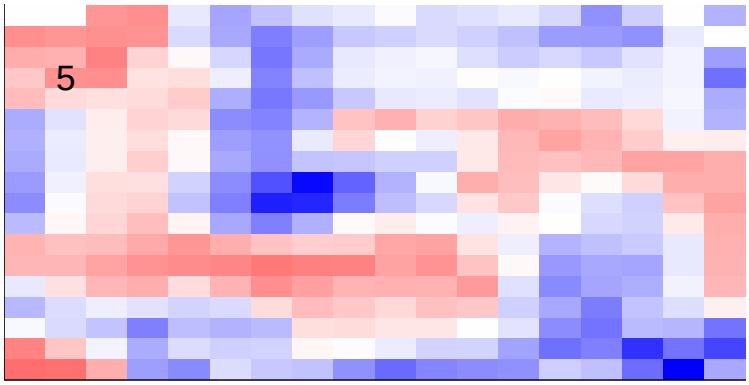


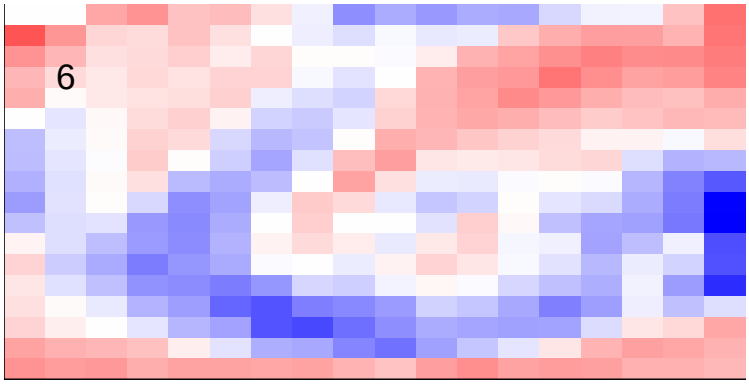

1

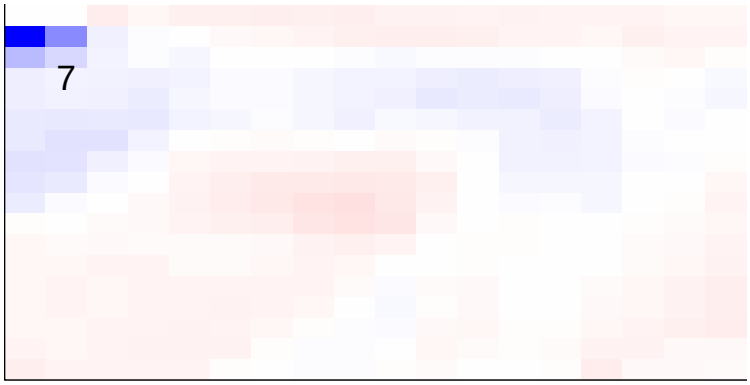

2

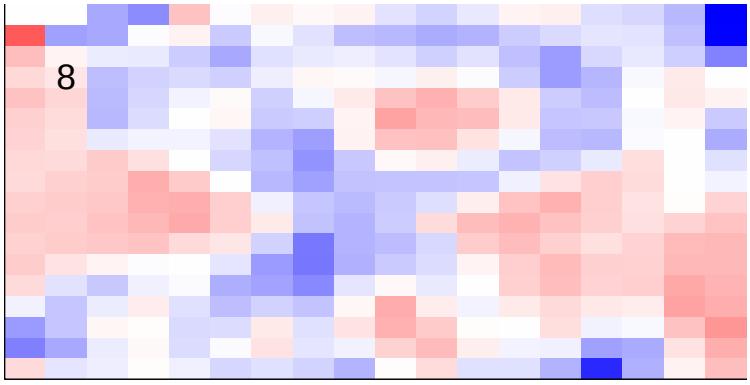

3

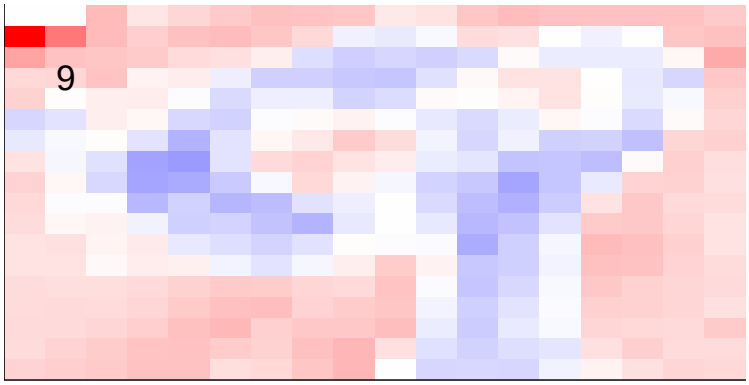

4


5


6


7


8




```
## $`0`
## NULL
##
## $`1`
## NULL
##
## $`2`
## NULL
##
## $`3`
## NULL
##
## $`4`
## NULL
##
## $`5`
## NULL
##
## $`6`
## NULL
##
## $`7`
## NULL
##
## $`8`
```

```
## NULL
##
## $`9`
## NULL
```

Chapter 4

Multi-Layer NN Notes

Similar to the chapter on single-layer NNs, this chapter outlays notation & methodology for a multiple-layer neural network.

---

source: https://arxiv.org/abs/1801.05894

"Deep Learning: An Introduction for Applied Mathematicians" by Catherine F. Higham and Desmond J. Higham, published in 2018

4.1 Notation Setup

4.1.1 Scalars

Layers: 1-L, indexed by l

Number of Neurons in layer l: n_l

Neuron Activations: $a_{\text{neuron num}}^{(\text{layer num})} = a_j^{(l)}$. Vector of activations for a layer is $a^{(l)}$

Activation Function: $g(\cdot)$ is our generic activation function

---

4.1.2 X

We have our input matrix $X \in \mathbb{R}^{\text{vars} \times \text{obs}} = \mathbb{R}^{n_0 \times m}$:

$$X = \overbrace{\left\{ \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n_0,1} & x_{n_0,2} & \cdots & x_{n_0,m} \end{bmatrix} \right.}^{m \text{ obs}} {}^{n_0 \text{ inputs}}$$

The ith observation of X is the ith column of X, referenced as x_i.

4.1.3 W

our Weight matrices $W^{(l)} \in \mathbb{R}^{\text{out} \times \text{in}} = \mathbb{R}^{n_l \times n_{l-1}}$:

$$W^{(l)} = \overbrace{\left\{ \begin{bmatrix} w^{(l)}_{1,1} & w^{(l)}_{1,2} & \cdots & w^{(l)}_{1,n_{l-1}} \\ w^{(l)}_{2,1} & w^{(l)}_{2,2} & \cdots & w^{(l)}_{2,n_{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w^{(l)}_{n_l,1} & w^{(l)}_{n_l,2} & \cdots & w^{(l)}_{n_l,n_{l-1}} \end{bmatrix} \right.}^{n_{l-1} \text{ inputs}} {}^{n_l \text{ outputs}}$$

$W^{(l)}$ is the weight matrix for the lth layer

4.1.4 b

our Bias matrices $b^{(l)} \in \mathbb{R}^{\text{out} \times 1} = \mathbb{R}^{n_l \times 1}$:

$$b^{(l)} = {}^{n_l \text{ outputs}} \left\{ \begin{bmatrix} b^{(l)}_1 \\ b^{(l)}_2 \\ \vdots \\ b^{(l)}_{n_l} \end{bmatrix} \right.$$

$b^{(l)}$ is the bias matrix for the lth layer

4.1.5 Y

our target layer matrix $Y \in \mathbb{R}^{\text{cats} \times \text{obs}} = \mathbb{R}^{n_L \times m}$:

$$Y = \overbrace{\left\{\begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,m} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n_L,1} & y_{n_L,2} & \cdots & y_{n_L,m} \end{bmatrix}\right.}^{m \text{ obs}} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} n_L \text{ categories}$$

Similar to X, the ith observation of Y is the ith column of Y, referenced as y_i.

4.1.6 z

our neuron layer's activation function input $z^{(l)} \in \mathbb{R}^{\text{out} \times 1} = \mathbb{R}^{n_l \times 1}$:

$$z^{(l)} = \left\{\begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{n_l}^{(l)} \end{bmatrix}\right\} n_l \text{ outputs}$$

$z^{(l)}$ is the neuron 'weighted input' matrix for the lth layer
We have that:

$$z^{(l)} = W^{(l)} * a^{(l-1)} + b^{(l)}$$

$$= \overbrace{\left\{\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n_{l-1}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l,1}^{(l)} & w_{n_l,2}^{(l)} & \cdots & w_{n_l,n_{l-1}}^{(l)} \end{bmatrix}\right.}^{n_{l-1} \text{ inputs}} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} n_l \text{ outputs} * \left\{\begin{bmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_{n_l}^{(l-1)} \end{bmatrix}\right\} n_{l-1} \text{ inputs} + \left\{\begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{bmatrix}\right\} n_l \text{ outputs}$$

$$= \left\{\begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{n_l}^{(l)} \end{bmatrix}\right\} n_l \text{ outputs}$$

4.1.7 a

our Neuron Activation $a^{(l)} \in \mathbb{R}^{\text{out} \times 1} = \mathbb{R}^{n_l \times 1}$:

$$a^{(l)} = \ ^{n_l \text{ outputs}} \left\{ \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{n_l}^{(l)} \end{bmatrix} \right.$$

$a^{(l)}$ is the activation matrix for the lth layer

We have that:

$$a^{(l)} = g\left(z^{(l)}\right)$$

$$= g\left(W^{(l)} * a^{(l-1)} + b^{(l)}\right)$$

$$= g\left(\ ^{n_l \text{ outputs}} \left\{ \overbrace{\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n_{l-1}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l,1}^{(l)} & w_{n_l,2}^{(l)} & \cdots & w_{n_l,n_{l-1}}^{(l)} \end{bmatrix}}^{n_{l-1} \text{ inputs}} \right. * \ ^{n_{l-1} \text{ inputs}} \left\{ \begin{bmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_{n_l}^{(l-1)} \end{bmatrix} \right. + \ ^{n_l \text{ outputs}} \left\{ \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{bmatrix} \right. \right)$$

$$= g\left(\ ^{n_l \text{ outputs}} \left\{ \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{n_l}^{(l)} \end{bmatrix} \right. \right)$$

$$= \ ^{n_l \text{ outputs}} \left\{ \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{n_l}^{(l)} \end{bmatrix} \right.$$

4.2 Forward Propagation

4.2.1 Setup

For a single neuron, it's activation is going to be a weighted sum of all the activations of the previous layer, plus a constant, all fed into the activation function. Formally, this is:

$$a_j^{(l)} = g\left(\sum_{i=1}^{n_{l-1}} w_{j,i}^{(l)} * a_i^{(l-1)} + b_j^{(l)}\right)$$

We can put this in matrix form. An entire layer of neurons can be represented by:

$$a^{(l)} = g\left(z^{(l)}\right) = g\left(W^{(l)} * a^{(l-1)} + b^{(l)}\right)$$

as was shown above. We can repeatedly apply this formula to get from X to out predicted $\hat{Y} = a^{(L)}$. We start with the initial layer (layer 0) being set equal to x_i.

Note that we will be forward (& backward) propagating one observation of X at a time by operating on each column separately. However, if desired forward (& backward) propagation can be done on all observations simultaneously. The notation change would involve stretching out $a^{(l)}$, $z^{(l)}$, and $b^{(l)}$ so that they are each m wide:

$$a^{(l)} = g\left(z^{(l)}\right)$$

$$= g\left(W^{(l)} * a^{(l-1)} + b^{(l)}\right)$$

$$= g\Big(\overset{n_l \text{ outputs}}{\Bigg\{} \overbrace{\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n_{l-1}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l,1}^{(l)} & w_{n_l,2}^{(l)} & \cdots & w_{n_l,n_{l-1}}^{(l)} \end{bmatrix}}^{n_{l-1} \text{ inputs}} * \overset{n_{l-1} \text{ inputs}}{\Bigg\{} \overbrace{\begin{bmatrix} a_{1,1}^{(l-1)} & a_{1,2}^{(l-1)} & \cdots & a_{1,m}^{(l-1)} \\ a_{2,1}^{(l-1)} & a_{2,2}^{(l-1)} & \cdots & a_{2,m}^{(l-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_{l-1},1}^{(l-1)} & a_{n_{l-1},2}^{(l-1)} & \cdots & a_{n_{l-1},m}^{(l-1)} \end{bmatrix}}^{m \text{ obs}}$$

$$+ \overset{n_l \text{ outputs}}{\Bigg\{} \overbrace{\begin{bmatrix} - & b_1^{(l)} & - \\ - & b_2^{(l)} & - \\ \vdots & \vdots & \vdots \\ - & b_{n_l}^{(l)} & - \end{bmatrix}}^{m \text{ obs}} \Big)$$

$$= g\left(\overset{n_l \text{ outputs}}{\Bigg\{} \overbrace{\begin{bmatrix} z_{1,1}^{(l)} & z_{1,2}^{(l)} & \cdots & z_{1,m}^{(l)} \\ z_{2,1}^{(l)} & z_{2,2}^{(l)} & \cdots & z_{2,m}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n_l,1}^{(l)} & z_{n_l,2}^{(l)} & \cdots & z_{n_l,m}^{(l)} \end{bmatrix}}^{m \text{ obs}} \right)$$

$$= \overset{n_l \text{ outputs}}{\Bigg\{} \overbrace{\begin{bmatrix} a_{1,1}^{(l)} & a_{1,2}^{(l)} & \cdots & a_{1,m}^{(l)} \\ a_{2,1}^{(l)} & a_{2,2}^{(l)} & \cdots & a_{2,m}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_l,1}^{(l)} & a_{n_l,2}^{(l)} & \cdots & a_{n_l,m}^{(l)} \end{bmatrix}}^{m \text{ obs}}$$

Each column of $a^{(l)}$ and $z^{(l)}$ represent an observation and can hold unique values, while $b^{(l)}$ is merely repeated to be m wide; each row is the same bias value for each neuron.

We are sticking with one observation at a time for it's simplicity, and it makes the back-propagation linear algebra easier/cleaner.

4.2.2 Algorithm

For a given observation x_i:

1. set $a^{(0)} = x_i$
2. For each l from 1 up to L:
   - $z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$
   - $a^{(l)} = g\left(z^{(l)}\right)$
   - $D^{(l)} = \text{diag}\left[g'\left(z^{(l)}\right)\right]$
     - this term will be needed later

if Y happens to be categorical, we may choose to apply the softmax function ($\frac{e^{z_i}}{\sum e^{z_j}}$) to $a^{(L)}$. Otherwise, we are done! We have our estimated result $a^{(L)}$.

4.3 Backward Propagation

Recall that we are trying to minimize a cost function via gradient descent by iterating over our parameter vector θ : $\theta^{t+1} \leftarrow \theta^t - \rho * \nabla \mathcal{C}(\theta)$. We will now implement this.

To do so, there is one more useful variable we need to define: $\delta^{(l)}$

4.3.1 Delta

We define $\delta_j^{(l)} := \frac{\partial \mathcal{C}}{\partial z_j^{(l)}}$ for a particular neuron, and its vector form $\delta^{(l)}$ represents the whole layer.

$\delta^{(l)}$ allows us to back-propagate one layer at a time by defining the gradients of the earlier layers from those of the later layers. In particular:

$$\delta^{(l)} = \text{diag}\left[g'\left(z^{(l)}\right)\right] * \left(W^{(l+1)}\right)^T * \delta^{(l+1)}$$

The derivation is in the linked paper, so I won't go over it in full here

---

In short, $z^{(l+1)} = W^{(l+1)} * g\left(z^{(l)}\right) + b^{(l+1)}$; so, $\delta^{(l)}$ is related to $\delta^{(l+1)}$ via the chain rule:

$$\delta^{(l)} = \frac{\partial \mathcal{C}}{\partial z^{(l)}} = \underbrace{\frac{\partial \mathcal{C}}{\partial z^{(l+1)}}}_{\delta^{(l+1)}} * \underbrace{\frac{\partial z^{(l+1)}}{\partial g}}_{\left(W^{(l+1)}\right)^T} * \underbrace{\frac{\partial g}{\partial z^{(l)}}}_{g'\left(z^{(l)}\right)}$$

[eventually, add in a write-up on why the transpose of W is taken. In short, it takes the dot product each neuron's output across the next layer's neurons ($\left(W^{(l+1)}\right)^T$, each row is the input neuron being distributed across the next layer) with the next layer's $\delta^{(l+1)}$]

Note that we scale $\delta^{(l)}$ by $g'\left(z^{(l)}\right)$, which we do by multiplying on the left by:

$$\text{diag}\left[g'\left(z^{(l)}\right)\right] = \begin{bmatrix} g'\left(z_1^{(l)}\right) & & & \\ & g'\left(z_2^{(l)}\right) & & \\ & & \ddots & \\ & & & g'\left(z_{n_l}^{(l)}\right) \end{bmatrix}$$

This has the same effect as element-wise multiplication.

For shorthand, we define $D^{(l)} = \text{diag}\left[g'\left(z^{(l)}\right)\right]$

4.3.2 Gradients

Given $\delta^{(l)}$, it becomes simple to write down our gradients:

$$\delta^{(L)} = D^{(L)} * \frac{\partial \mathcal{C}}{\partial a^{(L)}} \tag{a}$$

$$\delta^{(l)} = D^{(l)} * \left(W^{(l+1)}\right)^T * \delta^{(l+1)} \tag{b}$$

$$\frac{\partial \mathcal{C}}{\partial b^{(b)}} = \delta^{(l)} \tag{c}$$

$$\frac{\partial \mathcal{C}}{\partial W^{(l)}} = \delta^{(l)} * \left(a^{(l-1)}\right)^T \tag{d}$$

The proofs of these are in the linked paper. (could add in a bit with an intuitive explanation. eventually I want to get better vis of the chain rule tho beforehand, because I bet we could get something neat with neuron & derivative visualizations)

(we can also do this with expanded matrix view as above)

For the squared-error loss function $\mathcal{C}(\theta) = \frac{1}{2}(y - a^{(L)})^2$, we would have $\frac{\partial \mathcal{C}}{\partial a^{(L)}} = (a^{(L)} - y)$ [find out what this is for log-loss :) softmax too?]

4.3.3 Algorithm

For a given observation x_i:

1. set $\delta^{(L)} = D^{(l)} * \frac{\partial \mathcal{C}}{\partial a^{(L)}}$

2. For each l from $(L-1)$ down to 1:
   - $\delta^{(l)} = D^{(l)} * \left(W^{(l+1)}\right)^T * \delta^{(l+1)}$
3. For each l from L down to 1:
   - $W^{(l)} \leftarrow W^{(l)} - \rho * \delta^{(l)} * \left(a^{(l-1)}\right)^T$
   - $b^{(l)} \leftarrow W^{(l)} - \rho * \delta^{(l)}$

Chapter 5

Multi-Layer NN Model

This chapter presents the final functional-programming model. Uses functions to define 'neural networks', perform forward propagation, and perform gradient descent. Section at the end details future components that could be added in.

5.1 Generate Data

For now, having 3 inputs and combining them to create y, with a random error term. Would like to tweak the setup eventually.

```
## create data:
m <- 1000
n_1_manual <- 3
n_L_manual <- 1

# initialize Xs
X <- data.frame(X1 = runif(n = m, min = -10, max = 10),
                X2 = rnorm(n = m, mean = 0, sd = 10),
                X3 = rexp(n = m, rate = 1)) %>%
  as.matrix(nrow = m,
            ncol = n_1_manual)

# get response
Y <- X[, 1] + 10 * sin(X[, 2])^2 + 10 * X[, 3] + rnorm(n = 1000)

# fix dims according to NN specs
X <- t(X)
Y <- t(Y)
```

5.2 Functions

5.2.1 Link Functions

```
## Specify Link Functions & Derivatives
get_link <- function(type = "sigmoid") {

  if (type == "identity") {
    # identity
    g <- function(x) {x}

  } else if (type == "sigmoid") {
    # sigmoid
    g <- function(x) {1 / (1 + exp(-x))}

  } else if (type == "relu") {
    # ReLU
    g <- function(x) {x * as.numeric(x > 0)}

  } else (return(NULL))

  return(g)

}

get_link_prime <- function(type = "sigmoid") {

  if (type == "identity") {
    # identity [FIX]
    g_prime <- function(x) {
      ## there's probably a better way to do this
      b <- x / x
      b[is.nan(x/x)] <- 1
      return(b)
    }

  } else if (type == "sigmoid") {
    # sigmoid
    g_prime <- function(x) {exp(-x) / (1 + exp(-x))^2}

  } else if (type == "relu") {
    # ReLU
    g_prime <- function(x) {as.numeric(x > 0)}
```

```
  } else (return(NULL))

  return(g_prime)

}
```

5.2.2 Loss Functions

```
## Specify Loss Functions & Derivatives
get_loss_function <- function(type = "squared_error") {

  if (type == "squared_error") {

    loss <- function(y_hat, y) {sum((y_hat - y)^2)}

  } else if (type == "cross_entropy") {

    loss <- function(y_hat, y) {sum(y * log(y_hat))}

  } else (return(NULL))

  return(loss)

}

get_loss_prime <- function(type = "squared_error") {

  if (type == "squared_error") {

    loss_prime <- function(y_hat, y) {sum(2 * (y_hat - y))}

  } else if (type == "cross_entropy") {

    loss_prime <- function(y_hat, y) {999}

  } else (return(NULL))

  return(loss_prime)

}
```

5.2.3 Misc Helpers

```
## creates a list of n empty lists
create_lists <- function(n) {
  out <- list()

  for (i in 1:n) {
    out[[i]] <- list()
  }

  return(out)
}

## friendlier diag() function
diag_D <- function(x) {

  if (length(x) == 1) {
        out <- x
      } else {
        out <- diag(as.numeric(x))
      }

  return(out)
}

generate_layer_sizes <- function(X,
                                 Y,
                                 hidden_layer_sizes) {

  return(c(nrow(X), hidden_layer_sizes, nrow(Y)))

}
```

```
initialize_NN <- function(layer_sizes,
                          activation_function = "sigmoid",
                          last_activation_function = "identity",
                          lower_bound = 0,
                          upper_bound = 1) {

  n <- layer_sizes

  ## initialize parameter matrices
  W <- list()
  b <- list()
```

```
  ## could vectorize w/ mapply()
  for (l in 2:length(n)) {

    W[[l]] <- matrix(data = runif(n = n[l - 1] * n[l],
                                  min = lower_bound,
                                  max = upper_bound),
                     nrow = n[l],
                     ncol = n[l - 1])

    b[[l]] <- matrix(data = runif(n = n[l],
                                  min = lower_bound,
                                  max = upper_bound),
                     nrow = n[l],
                     ncol = 1)

  }

  ## return
  return(list(W = W,
              b = b,
              activation_function = activation_function,
              last_activation_function = last_activation_function))
}
```

5.2.4 Forward Propagation

```
NN_output <- function(X,
                      NN_obj) {

  L <- length(NN_obj$W)
  ## if X is one obs, input will be a vector so dim will be null
  m <- ifelse(is.null(ncol(X)),
              1,
              ncol(X))

  g <- get_link(NN_obj$activation_function)
  g_last <- get_link(NN_obj$last_activation_function)

  a <- list()

  a[[1]] <- X

  for (l in 2:(L - 1)) {
```

```
    a[[l]] <- g(NN_obj$W[[l]] %*% a[[l - 1]] + matrix(data = rep(x = NN_obj$b[[l]],
                                                                 times = m),
                                                      ncol = m))
  }

  a[[L]] <- g_last(NN_obj$W[[L]] %*% a[[L - 1]] + matrix(data = rep(x = NN_obj$b[[L]],
                                                                    times = m),
                                                         ncol = m))

  return(a[[L]])

}
```

5.2.5 Gradient Descent Iteration

```
GD_iter <- function(NN_obj,
                    X,
                    Y,
                    rho = 1,
                    verbose = FALSE,
                    very_verbose = FALSE) {

  L <- length(NN_obj$W)
  ## if X is one obs, input will be a vector so dim will be null
  m <- ifelse(is.null(ncol(X)),
              1,
              ncol(X))

  ## get links
  g <- get_link(NN_obj$activation_function)
  g_prime <- get_link_prime(NN_obj$activation_function)
  g_last <- get_link(NN_obj$last_activation_function)
  g_last_prime <- get_link_prime(NN_obj$last_activation_function)

  z <- create_lists(L)
  a <- create_lists(L)
  D <- create_lists(L)
  delta <- create_lists(L)
  del_W <- create_lists(L)
  del_b <- create_lists(L)

  ## gradient descent
  for (i in 1:m) {
```

```
    ## forward
    a[[1]][[i]] <- X[, i]

    for (l in 2:(L - 1)) {
      z[[l]][[i]] <- NN_obj$W[[l]] %*% a[[l - 1]][[i]] + NN_obj$b[[l]]
      a[[l]][[i]] <- g(z[[l]][[i]])
      D[[l]][[i]] <- diag_D(g_prime(z[[l]][[i]]))

      if (very_verbose == TRUE) {print(paste0("Forward: obs ", i, " - layer ", l))}
    }

    ## last layer
    z[[L]][[i]] <- NN_obj$W[[L]] %*% a[[L - 1]][[i]] + NN_obj$b[[L]]
    a[[L]][[i]] <- g_last(z[[L]][[i]])
    D[[L]][[i]] <- diag_D(g_last_prime(z[[L]][[i]]))

    ## backward
    # eventually fix to match with loss function
    delta[[L]][[i]] <- D[[L]][[i]] %*% (a[[L]][[i]] - Y[, i])

    for (l in (L - 1):2) {
      delta[[l]][[i]] <- D[[l]][[i]] %*% t(NN_obj$W[[l + 1]]) %*% delta[[l + 1]][[i]]
      if (very_verbose == TRUE) {print(paste0("Backward: obs ", i, " - layer ", l))}
    }

    for (l in 2:L) {
      del_W[[l]][[i]] <- delta[[l]][[i]] %*% t(a[[l - 1]][[i]])
      del_b[[l]][[i]] <- delta[[l]][[i]]
      if (very_verbose == TRUE) {print(paste0("del: obs ", i, " - layer ", l))}
    }

    if ((verbose == TRUE) & (i %% 100 == 0)) {print(paste("obs", i, "/", m))}

  }

  ## update parameters

  # get averages
  ## del_W is a list where each element represents a layer
  ## in each layer, there's a list representing the layer's result for that obs
  ## here we collapse the results by taking the sum of our gradients
  del_W_all <- lapply(X = del_W,
                      FUN = Reduce,
                      f = "+") %>%
    lapply(X = .,
```

```
          FUN = function(x) x / m)

  del_b_all <- lapply(X = del_b,
                      FUN = Reduce,
                      f = "+") %>%
    lapply(X = .,
           FUN = function(x) x / m)

  # apply gradient
  W_out <- mapply(FUN = function(A, del_A) {A - rho * del_A},
                  A = NN_obj$W,
                  del_A = del_W_all)

  b_out <- mapply(FUN = function(A, del_A) {A - rho * del_A},
                  A = NN_obj$b,
                  del_A = del_b_all)

  ## return a new NN object
  return(list(W = W_out,
              b = b_out,
              activation_function = NN_obj$activation_function,
              last_activation_function = NN_obj$last_activation_function))
}
```

5.2.6 Perform Gradient Descent

```
GD_perform <- function(X,
                       Y,
                       init_NN_obj,
                       rho = 0.01,
                       loss_function = "squared_error",
                       threshold = 1,
                       max_iter = 100,
                       print_descent = FALSE) {

  ## setup
  done_decreasing <- FALSE

  objective_function <- get_loss_function(type = loss_function)

  iteration_outputs <- list()
  output_objectives <- numeric()
```

```
iteration_input <- init_NN_obj

iter <- 1

initial_objective <- objective_function(y = Y,
                                        y_hat = NN_output(X = X,
                                                          NN_obj = init_NN_obj))

if (print_descent == TRUE) {
  print(paste0("iter: ", 0, "; obj: ", round(initial_objective, 1)))
}

while ((!done_decreasing) & (iter < max_iter)) {

  ## get input loss
  in_objective <- objective_function(y = Y,
                                     y_hat = NN_output(X = X,
                                                       NN_obj = iteration_input))

  ## iterate
  iteration_output <- GD_iter(NN_obj = iteration_input,
                              X = X,
                              Y = Y,
                              rho = rho,
                              verbose = FALSE,
                              very_verbose = FALSE)

  ## outputs
  out_objective <- objective_function(y = Y,
                                      y_hat = NN_output(X = X,
                                                        NN_obj = iteration_output))

  iteration_input <- iteration_output
  iteration_outputs[[iter]] <- iteration_output
  output_objectives[[iter]] <- out_objective

  if (print_descent == TRUE) {
    print(paste0("iter: ", iter, "; obj: ", round(out_objective, 1)))
  }

  iter <- iter + 1

  ## evaluate
  if (abs(in_objective - out_objective) < threshold) {
    done_decreasing <- TRUE
```

```
    }

  }

  return(list(final_NN = iteration_output,
              intermediate_NN = iteration_outputs,
              output_objectives = output_objectives,
              initial_objective = initial_objective,
              params = list(rho = rho,
                            loss_function = loss_function,
                            initial_NN = init_NN_obj)))
}
```

5.2.7 Summary Functions

```
GD_plot <- function(GD_obj) {

    data.frame(x = 1:length(GD_obj$output_objectives),
               y = GD_obj$output_objectives) %>%
    ggplot(aes(x = x,
               y = y)) +
    geom_point() +
    theme_bw() +
    labs(x = "Iteration",
         y = "Loss")

}

GD_summary <- function(GD_obj,
                       print_summary = TRUE) {

  ## num iter
  num_iter <- length(GD_obj$output_objectives)

  ## loss improvement
  initial_objective <- GD_obj$initial_objective %>% round(1)
  final_objective <- last(GD_obj$output_objectives) %>% round(1)
  loss_improvement_ratio <- (final_objective / initial_objective)  %>% round(4)

  if (print_summary == TRUE) {

    ## prints
    cat(paste0("Gradient Descent Summary:", "\n",
```

```
                "  |", "\n",
                "  |  Number of Iterations: ", num_iter, "\n",
                "  |", "\n",
                "  |  Initial Objective: ", initial_objective, "\n",
                "  |  Final Objective: ", final_objective, "\n",
                "  |  Ratio: ", loss_improvement_ratio, "\n", "\n"))

    cat(paste0("----------------------------------------", "\n",
               "Initial W:", "\n", "\n"))
    print(GD_obj$params$initial_NN$W[-1])
    cat(paste0("----------------------------------------", "\n",
               "Final W:", "\n", "\n"))
    print(GD_obj$final_NN$W[-1])

    cat(paste0("----------------------------------------", "\n",
               "Initial b:", "\n", "\n"))
    print(GD_obj$params$initial_NN$b[-1])
    cat(paste0("----------------------------------------", "\n",
               "Final b:", "\n", "\n"))
    print(GD_obj$final_NN$b[-1])

  }

  return(list(num_iter = num_iter,
              initial_objective = initial_objective,
              final_objective = final_objective,
              loss_improvement_ratio = loss_improvement_ratio))
}
```

5.3 Test

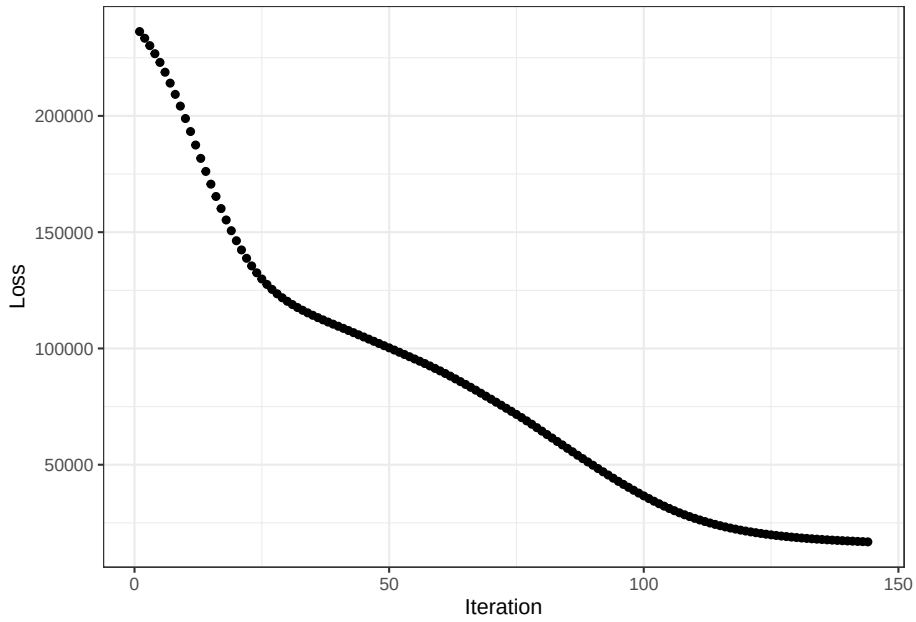
```
## initialize NN
init_NN <- initialize_NN(layer_sizes = generate_layer_sizes(X = X,
                                                            Y = Y,
                                                            hidden_layer_sizes = c(3)),
                         activation_function = "relu",
                         last_activation_function = "identity",
                         lower_bound = 0,
                         upper_bound = 1)

## train NN
GD_NN <- GD_perform(X = X,
```

```
                    Y = Y,
                    init_NN_obj = init_NN,
                    rho = 0.001,
                    loss_function = "squared_error",
                    threshold = 100,
                    max_iter = 1000,
                    print_descent = FALSE)

final_NN <- GD_NN$final_NN

## Summaries
NN_sum <- GD_summary(GD_obj = GD_NN)
```

```
## Gradient Descent Summary:
##   |
##   |  Number of Iterations: 144
##   |
##   |  Initial Objective: 238990.1
##   |  Final Objective: 16834.7
##   |  Ratio: 0.0704
##
## ----------------------------------------
## Initial W:
##
## [[1]]
##           [,1]      [,2]       [,3]
## [1,] 0.3028608 0.5655690 0.95437084
## [2,] 0.5046878 0.5267906 0.02262433
## [3,] 0.2724395 0.2474305 0.40767901
##
## [[2]]
##           [,1]      [,2]      [,3]
## [1,] 0.4370019 0.7598625 0.8453935
##
## ----------------------------------------
## Final W:
##
## [[1]]
##                X1          X2        X3
## [1,] -0.001842719  0.03928530 1.9846660
## [2,]  0.695969115  0.12981398 0.7693617
## [3,]  0.187358905 -0.08034643 2.0087035
##
## [[2]]
##          [,1]     [,2]     [,3]
```

```
## [1,] 1.699798 1.149463 2.324604
##
## ----------------------------------------
## Initial b:
##
## [[1]]
##             [,1]
## [1,] 0.007521215
## [2,] 0.425427183
## [3,] 0.791751480
##
## [[2]]
##           [,1]
## [1,] 0.3945866
##
## ----------------------------------------
## Final b:
##
## [[1]]
##           [,1]
## [1,] 0.2857580
## [2,] 0.6093002
## [3,] 1.2551489
##
## [[2]]
##           [,1]
## [1,] 0.7948542
```

```
GD_plot(GD_NN)
```



5.3.1 Other

```
## get_layer_size function
get_layer_sizes <- function(NN_obj) {
  n_1 <- ncol(NN_obj$W[[2]])

  n_H <- sapply(NN_obj$W[-1],
                nrow)

  return(c(n_1, n_H))
}
```

```
layer_sizes_test <- get_layer_sizes(final_NN)
```

5.4 Next Steps

In the future:

- need some sort of divergence check / pick 'best so far' output

- vis for gradient descent — pick 2 vars and for every combo of those 2, plot the objective function
- vis for gradient descent — show the evolution of the var through gradient descent over iterations
- NN overall vis & perhaps animation
- multi-dimensional output (cat / 1-hot)
- different cost functions (softmax squared-error & cross-entropy)
- 'from scratch' from scratch — mmult and maybe further lol
- get 'best-case' / perfect objective function (if data creation process known)
- stochastic gradient descent, minibatches (what gets passed down to GD_iter from GD_perform)
- regularization methods & CV-validation
