Neural Nets from Scratch

Daniel Polites, Ruibo Hou, Sihao Cheng, Yang-Hsuan Huang

2024-05-05

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

Setup

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

```
## Warning: package 'tidyverse' was built under R version 4.2.3
```

```
## Warning: package 'ggplot2' was built under R version 4.2.3
```

```
## Warning: package 'tibble' was built under R version 4.2.3
## Warning: package 'tidyr' was built under R version 4.2.3

## Warning: package 'readr' was built under R version 4.2.3

## Warning: package 'purrr' was built under R version 4.2.3

## Warning: package 'dplyr' was built under R version 4.2.3

## Warning: package 'stringr' was built under R version 4.2.3

## Warning: package 'forcats' was built under R version 4.2.3

## Warning: package 'lubridate' was built under R version 4.2.3

## -- Attaching core tidyverse packages --- tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- 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
```

```r
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.2.3
```

```r
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.2.3

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

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.

Model Form

We have an input vector of p variables $X = \{x_1, x_2, \ldots, x_p\}$, 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 $\{1, 2, \ldots, p\}$. $g(\cdot)$ 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:

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

Activation Functions

Sigmoid:

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

ReLU:

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

softmax:

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

^^ used for the output layer of a categorical response network.

Loss Functions

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

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

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)

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.

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.

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 **W**, and those that came from **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 $\hat{y}$ in our objective function

So we have:

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

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

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

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

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

Taking our derivatives:

Beta: Intercept

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

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

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

Beta: Coefficients

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

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

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

$$= \sum_{i=1}^{n} \hat{\epsilon}_i \; g(z_{ik})$$

W: Intercepts

$$\frac{\partial \mathcal{C}}{\partial w_{k0}} = \frac{\partial}{\partial w_{k0}} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \frac{\partial}{\partial w_{k0}} [\beta_k * g(z_{ik})]$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \; \frac{\partial}{\partial w_{k0}} g(z_{ik})$$

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

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

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

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:

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

Code Example

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

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.

```r
## 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)
```

Parameter Setup

We will have 2 hidden-layer neurons and a single quantitative output, so **W** will be 4×2 and **B** will be 3×1:

```r
## 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}}
```

Output

How the NN will calculate the output:

```r
## 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
```

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
```

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

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

Iterate

Employ gradient descent until objective function stops decreasing:

```r
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
```

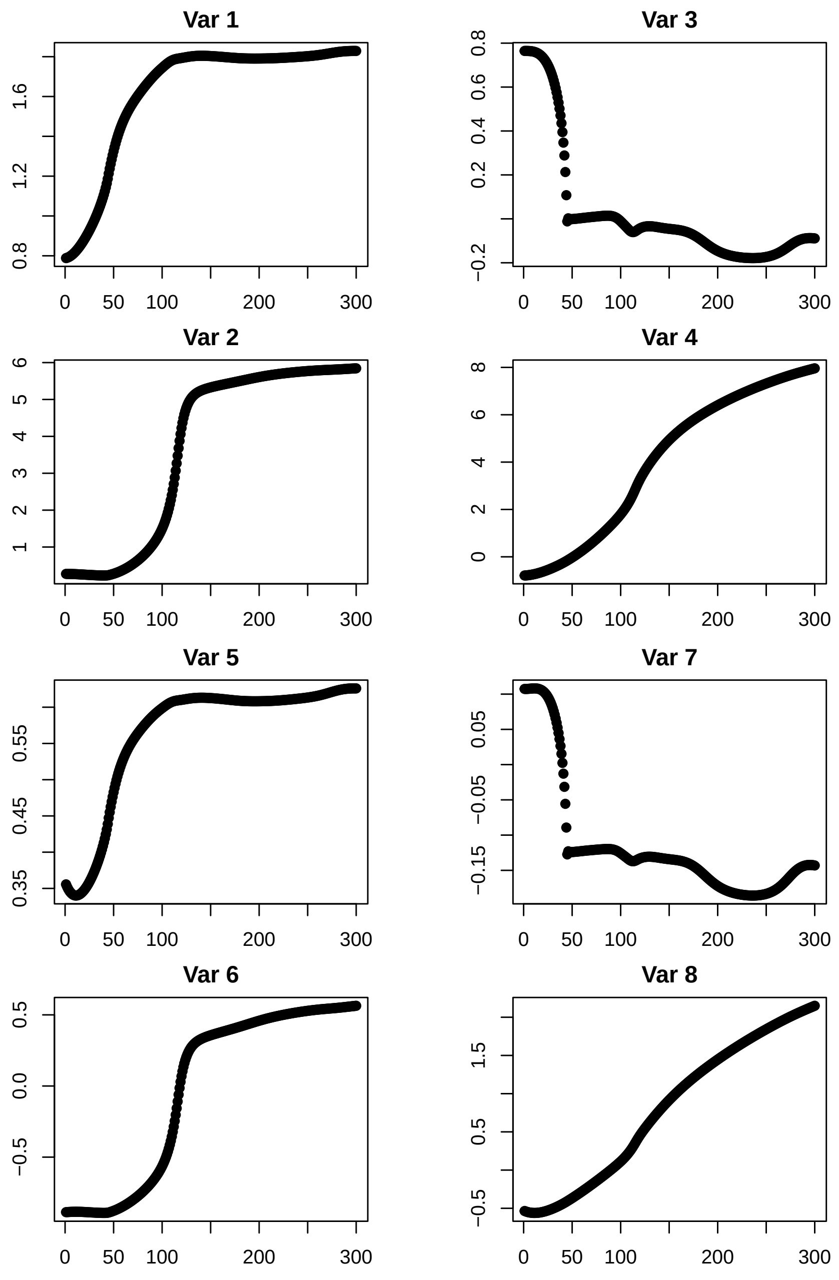
```r
## input B
B
```

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

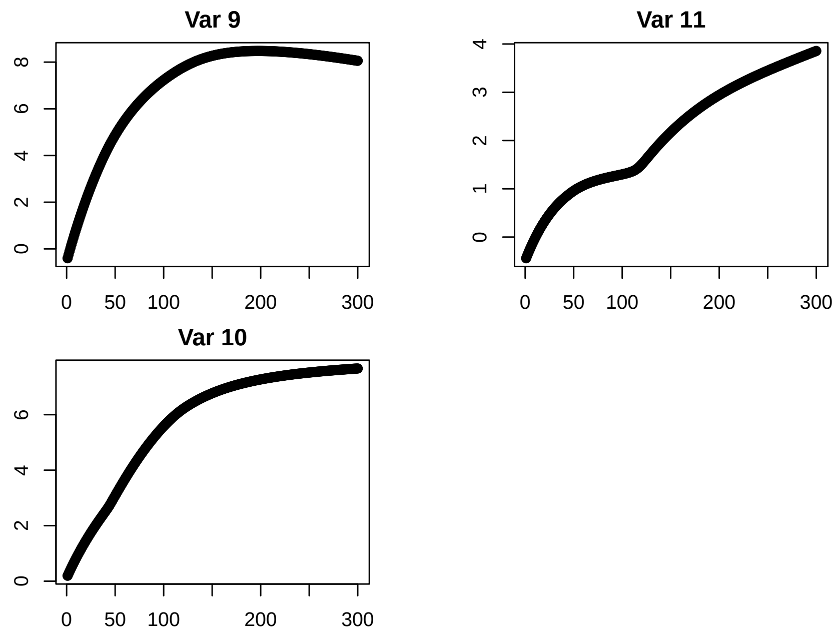
```r
## output B
iteration$B
```

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

```r
## 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))
```

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).

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 $\hat{Y}$:

$$\begin{aligned} \hat{Y} = f(X) &= g(Z) \cdot B + b_2 \\ &= g(X \cdot W + b_1) \cdot B + b_2 \\ &= g\left(\begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{2,1} & \cdots & w_{K,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,p} & w_{2,p} & \cdots & w_{K,p} \end{bmatrix} + \begin{bmatrix} | & | & & | \\ w_{1,0} & w_{2,0} & \cdots & w_{K,0} \\ | & | & & | \end{bmatrix} \right) \cdot \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_K \end{bmatrix} + \begin{bmatrix} | \\ \beta_0 \\ | \end{bmatrix} \\ &= \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} \end{aligned}$$

---

our error matrix:

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

gradients

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

b_2

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

B

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

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

b_1

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

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

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

W

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

$$= ???$$

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.

Linear Model

```
# Import dataset
pixel_numbers <- seq(1, 784)
x_column_names <- paste("pixel", pixel_numbers, sep = "")
y_column_names <- c("Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine
X.train <- read.csv('X_train.csv', header = FALSE, col.names = x_column_names)
X.test <- read.csv('X_test.csv', header = FALSE, col.names = x_column_names)
Y.train <- read.csv('Y_train.csv', header = FALSE, col.names = y_column_names)
Y.test <- read.csv('Y_test.csv', header = FALSE, col.names = y_column_names)

train_set <- cbind(X.train, Y.train)
test_set <- cbind(X.test, Y.test)

# Build logistic regression for each number
lm.fit0 <- glm(Zero ~. -One - Two - Three - Four - Five - Six - Seven - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
lm.fit1 <- glm(One ~. - Zero - Two - Three - Four - Five - Six - Seven - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
lm.fit2 <- glm(Two ~. - Zero - One - Three - Four - Five - Six - Seven - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
lm.fit3 <- glm(Three ~. - Zero - One - Two - Four - Five - Six - Seven - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
lm.fit4 <- glm(Four ~. - Zero - One - Two - Three - Five - Six - Seven - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
lm.fit5 <- glm(Five ~. - Zero - One - Two - Three - Four - Six - Seven - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
lm.fit6 <- glm(Six ~. - Zero - One - Two - Three - Four - Five - Seven - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
lm.fit7 <- glm(Seven ~. - Zero - One - Two - Three - Four - Five - Six - Eight - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```r
lm.fit8 <- glm(Eight ~. - Zero - One - Two - Three - Four - Five - Six - Seven - Nine,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```r
lm.fit9 <- glm(Nine ~. - Zero - One - Two - Three - Four - Five - Six - Seven - Eight,
               data = train_set, family = "binomial")
```

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```r
# Define softmax function
softmax <- function(x) {
  exp_x <- exp(x)
  return(exp_x / rowSums(exp_x))
}

# Function to predict using the model and new data
predict_response <- function(model, newdata = NULL) {
  as.vector(predict(model, type = "response", newdata = newdata))
}

# --- Training set predictions ---
train.preds <- lapply(list(lm.fit0, lm.fit1, lm.fit2, lm.fit3, lm.fit4, lm.fit5, lm.fit6, lm.fit7
train.result.matrix <- do.call(cbind, train.preds)

# Apply softmax and find the predicted classes
softmax_predictions_train <- softmax(train.result.matrix)
predicted_classes_train <- max.col(softmax_predictions_train) - 1
actual_train <- max.col(Y.train) - 1
train_accuracy <- sum(predicted_classes_train == actual_train) / length(actual_train)

# --- Testing set predictions ---
test.preds <- lapply(list(lm.fit0, lm.fit1, lm.fit2, lm.fit3, lm.fit4, lm.fit5, lm.fit6, lm.fit7,
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
```

```
test.result.matrix <- do.call(cbind, test.preds)

# Apply softmax and find the predicted classes
softmax_predictions_test <- softmax(test.result.matrix)
predicted_classes_test <- max.col(softmax_predictions_test) - 1
actual_test <- max.col(Y.test) - 1
test_accuracy <- sum(predicted_classes_test == actual_test) / length(actual_test)

# Print the accuracies
print(paste("Training Set Accuracy:", round(train_accuracy, 3)))
```

```
## [1] "Training Set Accuracy: 0.658"
```

```
print(paste("Testing Set Accuracy:", round(test_accuracy, 3)))
```

```
## [1] "Testing Set Accuracy: 0.639"
```

Binomial Model I

Setup

```
library(tidyverse)
library(keras)
library(glmnet)
library(MASS)
```

```
## Warning: package 'MASS' was built under R version 4.2.3

##
## Attaching package: 'MASS'

## The following object is masked from 'package:dplyr':
##
##     select
```

```
library(logistf)
```

```
## Warning: package 'logistf' was built under R version 4.2.3

## Warning in check_dep_version(): ABI version mismatch:
## lme4 was built with Matrix ABI version 1
## Current Matrix ABI version is 0
## Please re-install lme4 from source or restore original 'Matrix' package
```

```
library(Metrics)
```

```
## Warning: package 'Metrics' was built under R version 4.2.3
```

```
library(prediction)
```

```
## Warning: package 'prediction' was built under R version 4.2.3
```

```
# 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
```

Model I

```
## 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  7  0  0  0  0  0  0  0  0  0
##     1  0 11  0  0  0  0  0  0  0  0
##     2  0  0 11  0  0  0  0  0  1  0
##     3  0  0  0  6  0  0  0  0  1  0
##     4  0  0  0  0  6  0  1  0  3  0
##     5  0  0  1  1  0  1  0  0  5  0
##     6  0  0  0  0  0  0  9  0  0  0
##     7  0  0  0  0  0  0  0 10  0  0
##     8  0  0  0  0  0  0  0  0 19  0
##     9  0  0  0  0  1  0  0  1  3  2
```

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

```
## [1] 0.18
```

Model II

```
## 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()
```

```
# Initialize a list to store the logistic regression models
logistic_models <- list()

# Initialize a list to store the predicted probabilities for each digit
probs_train_list <- list()
probs_test_list <- list()

# Initialize a list to store the binary predictions for each digit
pred_train_list <- list()
pred_test_list <- list()

# Initialize lists to store log loss and classification error for each digit
log_loss_train_list <- list()
classification_error_train_list <- list()
log_loss_test_list <- list()
classification_error_test_list <- list()

# Iterate over digits from 0 to 9
for (digit in 0:9) {
  # Apply the transformation to y_train and y_test for the current digit
  y_train_digit <- ifelse(y_train == digit, 1, 0)
  y_test_digit <- ifelse(y_test == digit, 1, 0)

  # Fit logistic regression model for the current digit
  logistic_model <- glm(y_train_digit ~ ., family = binomial(link = "logit"), data = x
  logistic_models[[as.character(digit)]] <- logistic_model

  # Make predictions on the training set for the current digit
  probs_train <- predict(logistic_model, x_train_2, type = "response")
  probs_train_list[[as.character(digit)]] <- probs_train

  # Convert predicted probabilities to binary predictions (0 or 1)
  pred_train <- ifelse(probs_train > 0.5, 1, 0)
  pred_train_list[[as.character(digit)]] <- pred_train

  # Make predictions on the test set for the current digit
  probs_test <- predict(logistic_model, x_test_2, type = "response")
  probs_test_list[[as.character(digit)]] <- probs_test

  # Convert predicted probabilities to binary predictions (0 or 1)
  pred_test <- ifelse(probs_test > 0.5, 1, 0)
  pred_test_list[[as.character(digit)]] <- pred_test

  # Calculate log loss on the training set for the current digit
  loss_entropy_train <- logLoss(y_train_digit, probs_train)
  cat("Training set log loss for digit", digit, ":", round(loss_entropy_train, 3), "\n")

  # Calculate log loss on the test set for the current digit
  loss_entropy_test <- logLoss(y_test_digit, probs_test)
  cat("Test set log loss for digit", digit, ":", round(loss_entropy_test, 3), "\n")

  # Calculate classification error on the training set for the current digit
  classification_error_train <- 1 - sum(pred_train == y_train_digit) / length(y_train_digit)
  cat("Average number of classification errors on training set for digit", digit, ":", round(clas

  # Calculate classification error on the test set for the current digit
  classification_error_test <- 1 - sum(pred_test == y_test_digit) / length(y_test_digit)
  cat("Average number of classification errors on test set for digit", digit, ":", round(classifi

  # Store log loss and classification error in lists
  log_loss_train_list[[as.character(digit)]] <- loss_entropy_train
  classification_error_train_list[[as.character(digit)]] <- classification_error_train
  log_loss_test_list[[as.character(digit)]] <- loss_entropy_test
  classification_error_test_list[[as.character(digit)]] <- classification_error_test
}
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading

## Training set log loss for digit 0 : 0.255
## Test set log loss for digit 0 : 0.404
## Average number of classification errors on training set for digit 0 : 0.007
## Average number of classification errors on test set for digit 0 : 0.011

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading

## Training set log loss for digit 1 : 0.285
## Test set log loss for digit 1 : 0.371
## Average number of classification errors on training set for digit 1 : 0.008
## Average number of classification errors on test set for digit 1 : 0.01

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading

## Training set log loss for digit 2 : 0.696
## Test set log loss for digit 2 : 0.811
## Average number of classification errors on training set for digit 2 : 0.019
## Average number of classification errors on test set for digit 2 : 0.022

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Training set log loss for digit 3 : 0.914
## Test set log loss for digit 3 : 1.006
## Average number of classification errors on training set for digit 3 : 0.025
## Average number of classification errors on test set for digit 3 : 0.028

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading

## Training set log loss for digit 4 : 0.569
## Test set log loss for digit 4 : 0.681
## Average number of classification errors on training set for digit 4 : 0.016
## Average number of classification errors on test set for digit 4 : 0.019

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading

## Training set log loss for digit 5 : 1.049
## Test set log loss for digit 5 : 1.089
## Average number of classification errors on training set for digit 5 : 0.029
## Average number of classification errors on test set for digit 5 : 0.03

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Training set log loss for digit 6 : 0.571
## Test set log loss for digit 6 : 0.761
## Average number of classification errors on training set for digit 6 : 0.016
## Average number of classification errors on test set for digit 6 : 0.021

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading

## Training set log loss for digit 7 : 0.552
## Test set log loss for digit 7 : 0.728
## Average number of classification errors on training set for digit 7 : 0.015
## Average number of classification errors on test set for digit 7 : 0.02

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading

## Training set log loss for digit 8 : 1.427
## Test set log loss for digit 8 : 1.55
## Average number of classification errors on training set for digit 8 : 0.04
## Average number of classification errors on test set for digit 8 : 0.043

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
## Training set log loss for digit 9 : 1.526
## Test set log loss for digit 9 : 1.507
## Average number of classification errors on training set for digit 9 : 0.042
## Average number of classification errors on test set for digit 9 : 0.042
```

```
# Print the results from lists in a single loop
for (digit in 0:9) {
  cat("Digit", digit, "\n")
  cat(" Training Set Log Loss:", round(log_loss_train_list[[as.character(digit)]], 3), "\n")
  cat(" Training Set Classification Error:", round(classification_error_train_list[[as.character(
  cat(" Test Set Log Loss:", round(log_loss_test_list[[as.character(digit)]], 3), "\n")
  cat(" Test Set Classification Error:", round(classification_error_test_list[[as.character(digit
}
```

```
## Digit 0
##  Training Set Log Loss: 0.255
##  Training Set Classification Error: 0.007
##  Test Set Log Loss: 0.404
##  Test Set Classification Error: 0.011
## Digit 1
##  Training Set Log Loss: 0.285
##  Training Set Classification Error: 0.008
##  Test Set Log Loss: 0.371
##  Test Set Classification Error: 0.01
## Digit 2
##  Training Set Log Loss: 0.696
##  Training Set Classification Error: 0.019
##  Test Set Log Loss: 0.811
##  Test Set Classification Error: 0.022
## Digit 3
##  Training Set Log Loss: 0.914
##  Training Set Classification Error: 0.025
##  Test Set Log Loss: 1.006
##  Test Set Classification Error: 0.028
## Digit 4
##  Training Set Log Loss: 0.569
##  Training Set Classification Error: 0.016
##  Test Set Log Loss: 0.681
##  Test Set Classification Error: 0.019
## Digit 5
##  Training Set Log Loss: 1.049
##  Training Set Classification Error: 0.029
##  Test Set Log Loss: 1.089
##  Test Set Classification Error: 0.03
## Digit 6
##  Training Set Log Loss: 0.571
##  Training Set Classification Error: 0.016
##  Test Set Log Loss: 0.761
##  Test Set Classification Error: 0.021
## Digit 7
##  Training Set Log Loss: 0.552
##  Training Set Classification Error: 0.015
##  Test Set Log Loss: 0.728
##  Test Set Classification Error: 0.02
## Digit 8
##  Training Set Log Loss: 1.427
##  Training Set Classification Error: 0.04
##  Test Set Log Loss: 1.55
##  Test Set Classification Error: 0.043
## Digit 9
##  Training Set Log Loss: 1.526
##  Training Set Classification Error: 0.042
##  Test Set Log Loss: 1.507
##  Test Set Classification Error: 0.042
```

```
# Compute the average log loss & classification errors
average_log_loss_train <- mean(sapply(log_loss_train_list, function(x) x))
cat("Average Training Set Log Loss:", round(average_log_loss_train, 3), "\n")
```

```
## Average Training Set Log Loss: 0.785
```

```
average_classification_error_train <- mean(sapply(classification_error_train_list, func
cat("Average Training Set Classification Errors:", round(average_classification_error_t
```

```
## Average Training Set Classification Errors: 0.022
```

```
average_log_loss_test <- mean(sapply(log_loss_test_list, function(x) x))
cat("Average Test Set Log Loss:", round(average_log_loss_test, 3), "\n")
```

```
## Average Test Set Log Loss: 0.891
```

```
average_classification_error_test <- mean(sapply(classification_error_test_list, functi
cat("Average Test Set Classification Errors:", round(average_classification_error_test
```

```
## Average Test Set Classification Errors: 0.025
```

From the log loss and the average number of classification errors, we can see that the prediction results are quite accurate(training set: loss= 0.785, classification errors= 0.022; test set: loss= 0.891, classification error: 0.025)

Binomial Model II

```
# Download the first 2000 observations of the MNIST dataset
mnist <- read.csv("https://pjreddie.com/media/files/mnist_train.csv", nrows = 2000)
colnames(mnist) <- c("Digit", paste("Pixel", seq(1:784), sep = ""))
save(mnist, file = "mnist_first2000.RData")
```

1 knn

```
# Load the required packages
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.2.3

## Loading required package: lattice

## Warning: package 'lattice' was built under R version 4.2.3

##
## Attaching package: 'caret'

## The following objects are masked from 'package:Metrics':
##
##     precision, recall

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

```
library(class)
```

```
## Warning: package 'class' was built under R version 4.2.3
```

```
# Load the MNIST dataset
load("mnist_first2000.RData")

# Split the dataset into training data (first 1000 rows) and testing data (last 1000 rows)
train_data <- mnist[1:1000,]
test_data <- mnist[1001:2000,]
```

```
# Train a KNN model using 5-fold cross-validation
train_control <- trainControl(method="cv", number=5)
tune_grid <- expand.grid(.k=1:20)
knn_model <- train(as.factor(Digit)~., data=train_data, method="knn", tuneGrid=tune_gri

# Select the value of K with the lowest cross-validation error
best_k <- knn_model$bestTune$k
print(best_k)
```

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

```
# Use the KNN algorithm for classification
knn_fit <- knn(train=train_data[,1:785], test=test_data[,1:785], cl=train_data$Digit, k

# Calculate the prediction error and confusion matrix
error_rate <- mean(knn_fit != test_data$Digit)
confusion_matrix <- table(knn_fit, test_data$Digit)
print(confusion_matrix)
```

```
##
## knn_fit   0   1   2   3   4   5   6   7   8   9
##       0  89   0   0   1   0   1   1   0   2   1
##       1   1 103   4   1   6   0   0   1   1   1
##       2   0   0  86   4   1   0   0   0   0   0
##       3   0   0   0  76   0   6   0   0   1   1
##       4   0   0   1   0  81   0   1   0   1   5
##       5   0   0   0  11   1  76   1   2   3   0
##       6   2   0   0   0   2   3 103   0   3   0
##       7   0   1   5   1   1   0   0 101   0   5
##       8   0   0   3   2   1   1   0   0  70   0
##       9   1   0   0   2  16   2   0   3   4  97
```

```
# Print the prediction classification error
print(paste("Prediction classification error:", error_rate))
```

```
## [1] "Prediction classification error: 0.118"
```

2 logistic regression

```
# Load the glmnet package
library(glmnet)
```

```
# Load the MNIST dataset
load("mnist_first2000.RData")

# Split the data into training and testing sets
train_data <- mnist[1:1000,]
test_data <- mnist[1001:2000,]

# Extract the predictors and response variables
x_train <- train_data[,2:785]
y_train <- train_data[, 1]
x_test <- test_data[,2:785]
y_test <- test_data[, 1]

# Convert non-numeric columns to numeric using model.matrix()
x_train <- model.matrix(~., data=train_data[, -1])
x_test  <- model.matrix(~., data=test_data[, -1])
# Fit a multi-class logistic regression model with Lasso penalty
fit <- cv.glmnet(x_train, y_train, family="multinomial", alpha=1, type.measure="class")

# Select the best tuning parameter
best_lambda <- fit$lambda.min

# Make predictions on the testing data
pred <- predict(fit, newx=x_test, s=best_lambda, type="class")

# Compute the confusion matrix
confusion_matrix <- table(pred, y_test)

# Compute the prediction classification error
error_rate <- mean(pred != y_test)

# Print the confusion matrix and prediction classification error
print(confusion_matrix)
```

```
##     y_test
## pred  0  1  2  3  4  5  6  7  8  9
##    0 84  0  1  0  0  0  2  0  3  4
##    1  0 90  3  1  1  0  2  1  3  0
##    2  0  1 77 12  3  0  4  0  3  0
##    3  1  1  1 64  1  1  0  3  1  1
##    4  1  0  2  0 83  1  1  2  1 12
##    5  4  0  0  9  2 75  2  0  2  1
##    6  0  1  5  2  2  2 93  1  1  0
##    7  0  1  6  2  3  2  0 92  2  5
##    8   3 10   3   5   2   7   2   1 66   2
##    9   0   0   1   3 12   1   0   7   3 85
```

```
print(paste("Prediction classification error:", error_rate))
```

```
## [1] "Prediction classification error: 0.191"
```

Binomial Model III

```
# Import dataset
pixel_numbers <- seq(1, 784)
x_column_names <- paste("pixel", pixel_numbers, sep = "")
y_column_names <- c("Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Ei
X.train <- as.matrix(read.csv('X_train.csv', header = FALSE, col.names = x_column_name
X.test <- as.matrix(read.csv('X_test.csv', header = FALSE, col.names = x_column_names)
Y.train <- as.matrix(read.csv('Y_train.csv', header = FALSE, col.names = y_column_name
Y.test <- as.matrix(read.csv('Y_test.csv', header = FALSE, col.names = y_column_names)

train_set <- cbind(X.train, Y.train)
test_set <- cbind(X.test, Y.test)

# Create logistic function without using glm function
# Define sigmoid activation function
sigmoid <- function(x) {
  1 / (1 + exp(-x))
}

# Using cross-entropy as the loss function
cross_entropy <- function(X, y, theta) {
  m <- length(y)
  predictions <- sigmoid(X %*% theta)
  cost <- -(1 / m) * sum(y * log(predictions) + (1 - y) * log(1 - predictions))
  return(cost)
}

# We are going to use gradient descent to find the optimal theta value
gradient_descent <- function(X, y, theta, alpha, num_iters) {
  m <- length(y)
  cost_history <- numeric(num_iters)

  for (i in 1:num_iters) {
    prediction <- sigmoid(X %*% theta)
    errors <- prediction[, 1] - y
    updates <- alpha * (1 / m) * (t(X) %*% errors)
    theta <- theta - updates
    cost_history[i] <- cross_entropy(X, y, theta)
  }

  list(theta = theta, cost_history = cost_history)
}

# Tune theta using training set
alpha <- 0.01
num_iters <- 1000
initial_beta <- rep(0, ncol(X.train))

# Tune logistic regression beta using
results <- lapply(1:ncol(Y.train), function(i) {
  gradient_descent(X.train, Y.train[, i], theta = initial_beta,
                   alpha = alpha, num_iters = num_iters)
})
```

Multinomial Model I

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")
}
```

```r
# 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)
```

```r
## 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)
}
```

```r
## 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
```

Model

train

```r
## 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)]
```

```r
## 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")
```

```
## 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 12  0  0  0  0  0  0  0  0  0
##       1  0  7  0  0  0  0  0  0  0  0
##       2  0  0  9  0  0  0  0  0  0  0
##       3  0  0  0 15  0  0  0  0  0  0
##       4  0  0  0  0 11  0  0  0  0  0
##       5  0  0  0  1  0  6  0  0  0  0
##       6  0  0  0  0  0  0 14  0  0  0
##       7  0  0  0  0  0  0  0  7  0  0
##       8  0  0  0  0  0  0  0  0 10  0
##       9  0  0  0  0  0  0  0  0  0  8
```

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

```
## [1] 0.01
```

test

```
## pre-process data
x_multi_test <- dplyr::select(x_test_2, all_of(names(x_multi)))
```

```
## get preds
multi_model_test <- predict(init_model,
                            as.matrix(x_multi_test),
                            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)
```

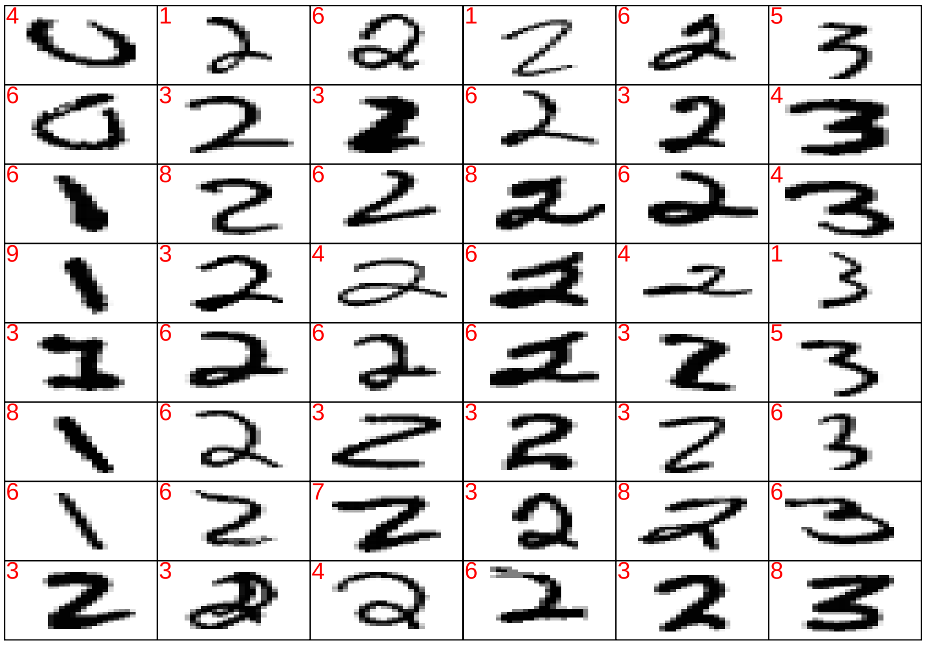
```
##       preds_test
## y_test    0    1    2    3    4    5    6    7    8    9
##      0  911    0    1   17    8    2   37    0    4    0
##      1    0 1065    0    7    3    0   12    0   42    6
##      2   58   25  488  210   41    0  173    4   33    0
##      3   21    9    5  853   34   16   23    6   33   10
##      4   14    4    6    2  882    1   24    1    8   40
##      5   37   22    3  246   81  269   65    9  144   16
##      6   16    4    4    2    6    3  914    0    9    0
##      7   10   41    6   74  182    0    8  603    4  100
##      8   37   60    5  137   53   25   47   11  568   31
##      9   31    6    4   32  296    0    4   73   23  540
```

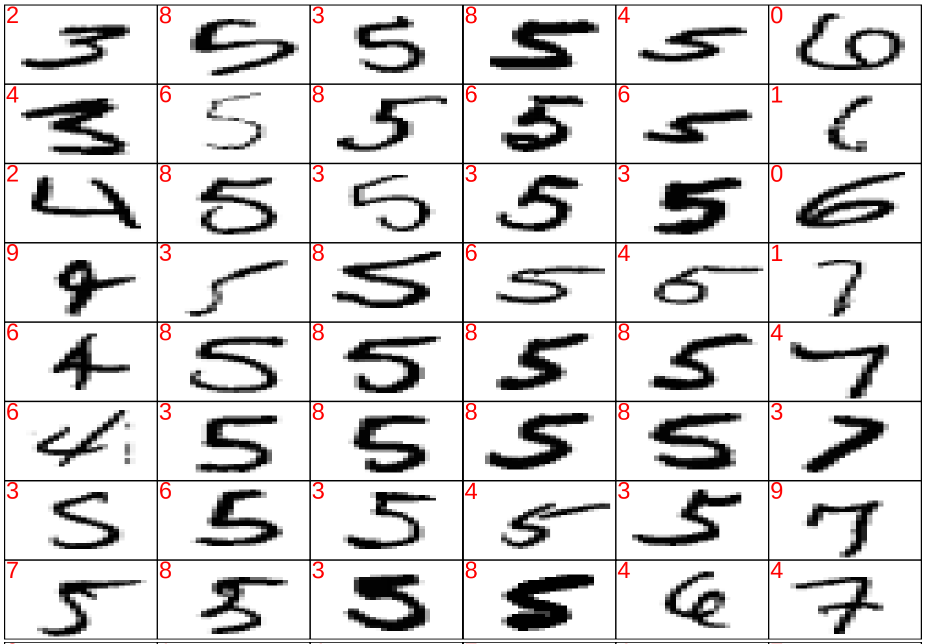
```
## TEST misclassification rate
mean(!(y_test == preds_test))
```

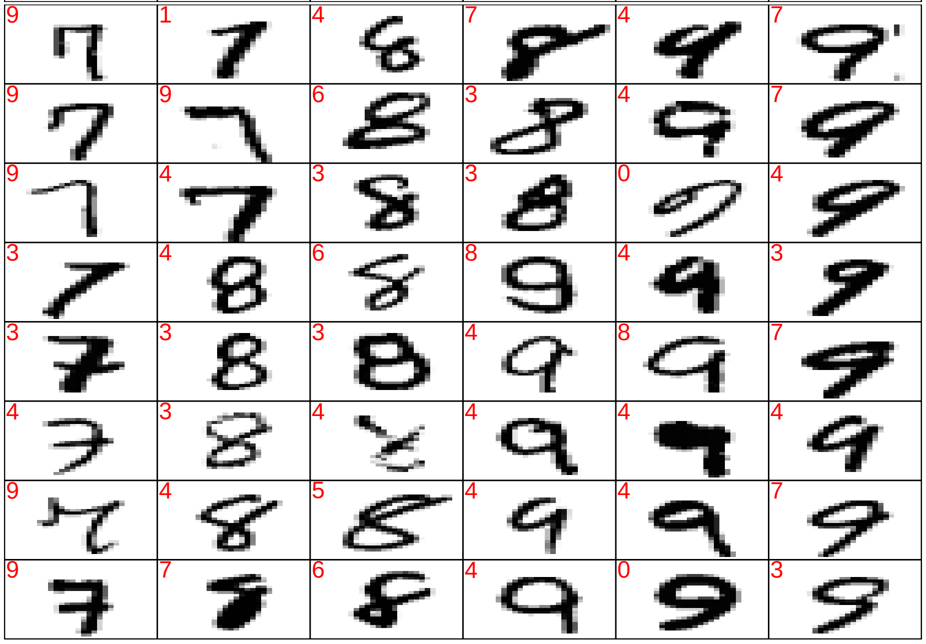
```
## [1] 0.2907
```

```
## 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
```

```r
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])
}
```



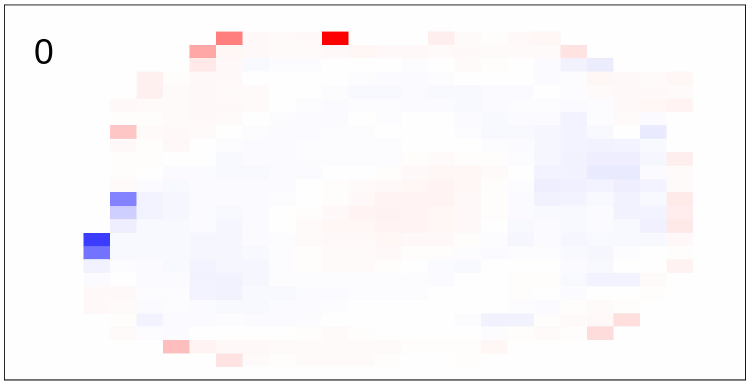


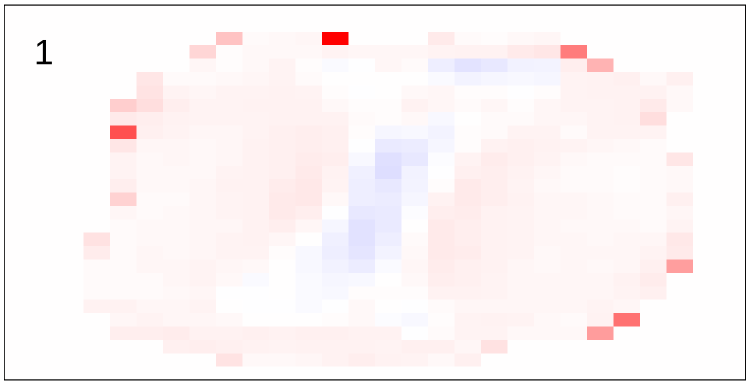


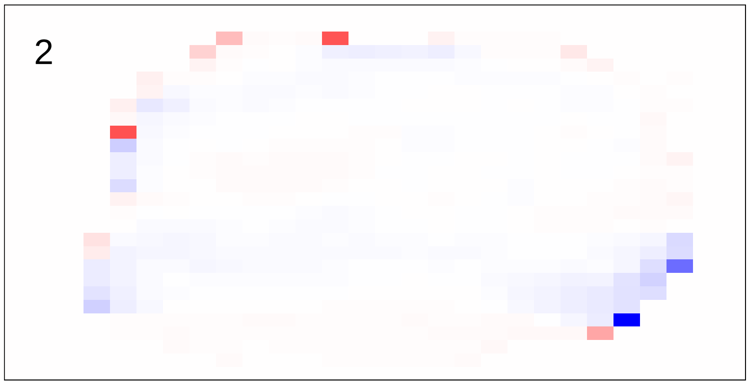
```r
par(mfcol = c(1, 1))
```

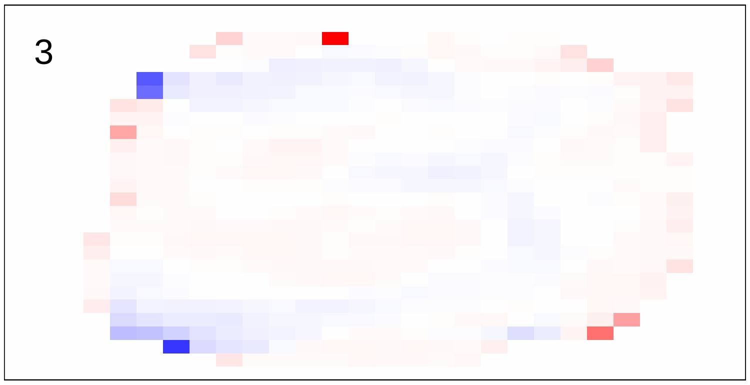
model heatmaps

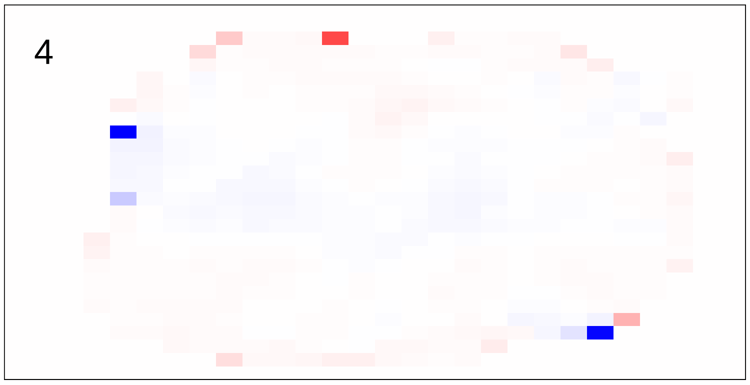
```r
## get coefficients into matrices
model_coef <- coef(init_model, s = init_model$lambda.min) %>%
  lapply(as.matrix) %>%
  lapply(function(x) matrix(x[-1, ], nrow = 28, ncol = 28)) %>%
  ## take sigmoid activation just to help viz
  lapply(function(x) 1 / (1 + exp(-x)) - 0.5)

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

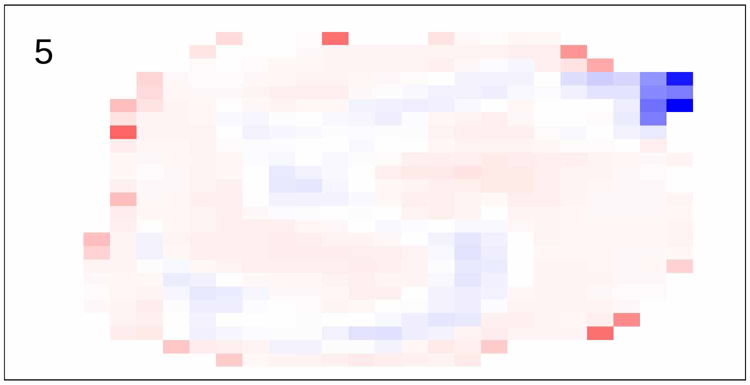


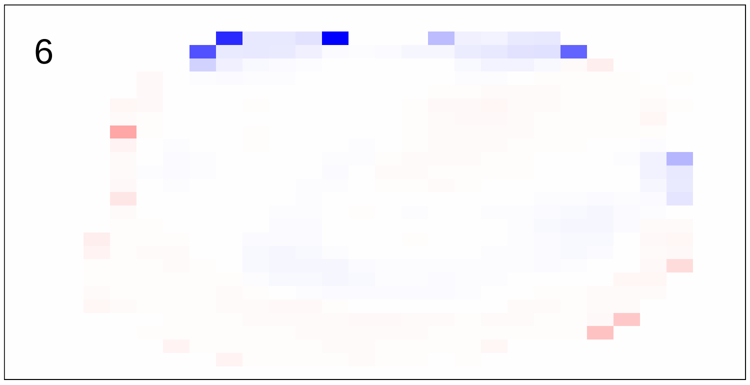

1

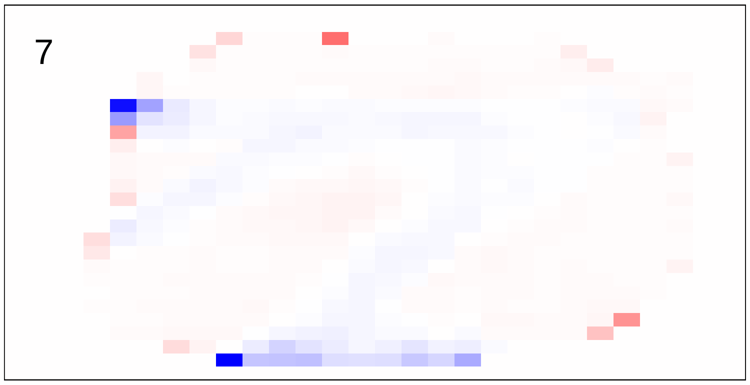

2

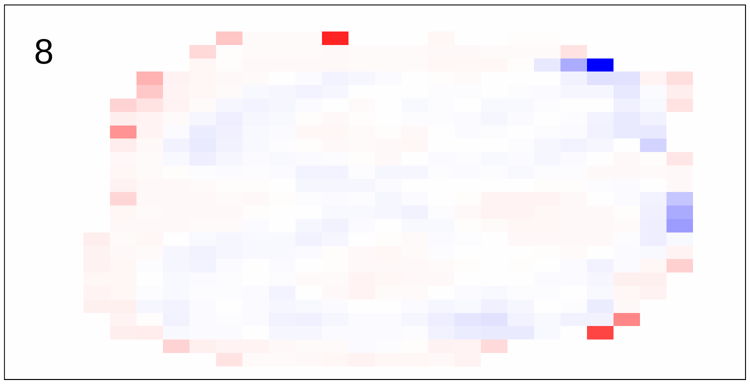

3

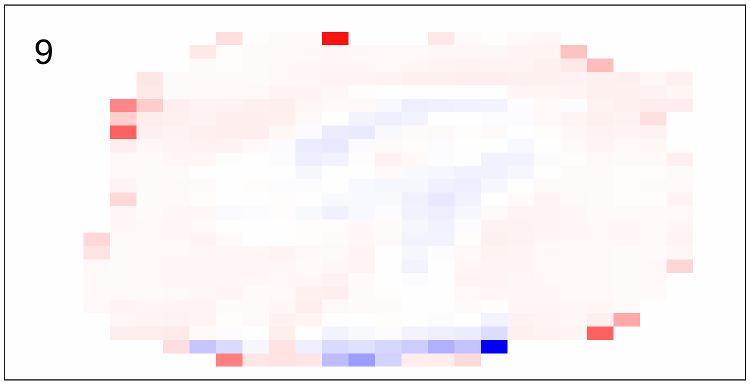

4


5


6


7


8



```
## $`0`
## NULL
##
## $`1`
## NULL
##
## $`2`
## NULL
##
## $`3`
## NULL
##
## $`4`
## NULL
##
## $`5`
## NULL
##
## $`6`
## NULL
##
## $`7`
## NULL
##
## $`8`
## NULL
##
## $`9`
## NULL
```

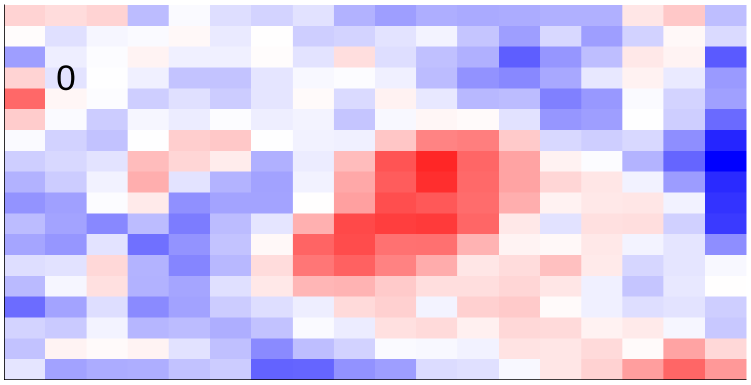
no outside cells model

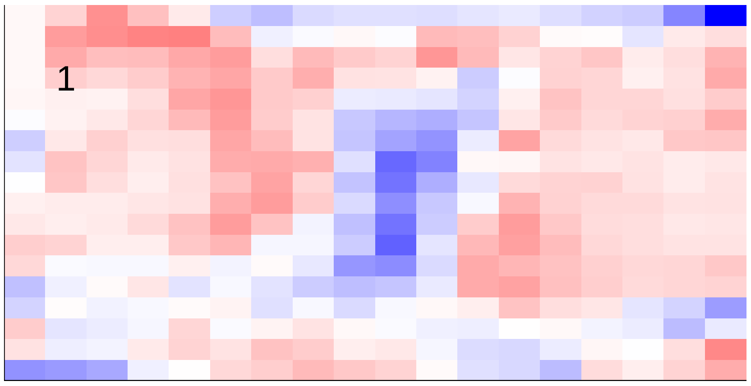
earlier runs of the above sections revealed that for a regularization method that does not perform variable selection, odd importance is given to outermost cell for prediction. Thus, those will be removed:

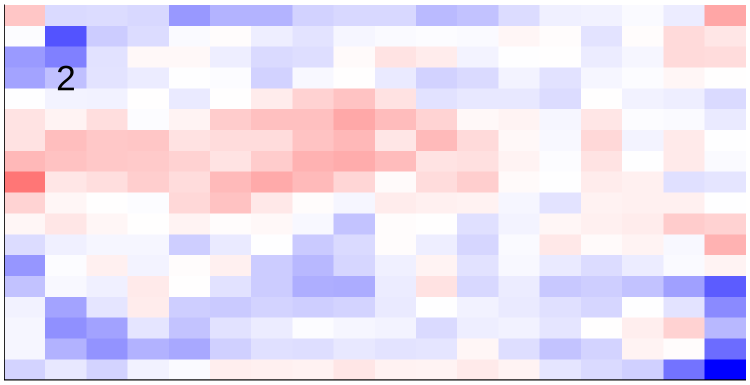
```
## set training data size
# n <- nrow(x_train_2)
n <- 1000

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)]
```

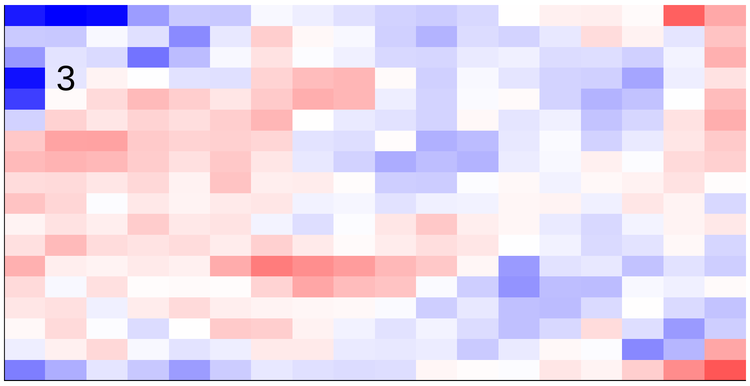
```
## for the sake of the coefficients viz, setting alpha = 0
init_model <- cv.glmnet(x = x_multi %>% as.matrix,
                        y = y_multi %>% factor,
                        family = "multinomial",
                        alpha = 0)

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

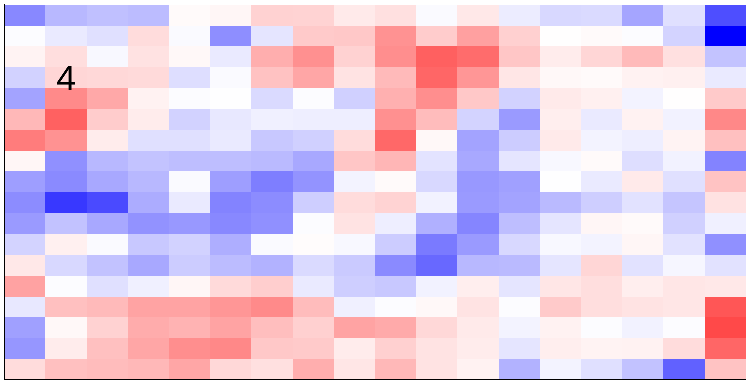
```
## get coefficients into matrices
model_coef <- coef(init_model, s = init_model$lambda.min) %>%
  lapply(as.matrix) %>%
  lapply(function(x) matrix(x[-1, ], nrow = 18, ncol = 18)) %>%
  ## take sigmoid activation just to help viz
  lapply(function(x) 1 / (1 + exp(-x)) - 0.5)

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

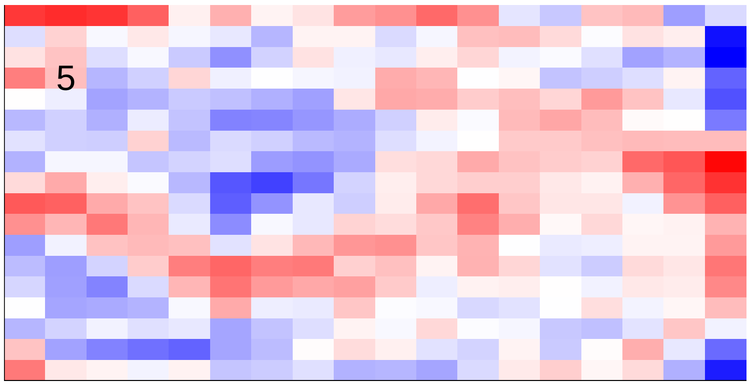


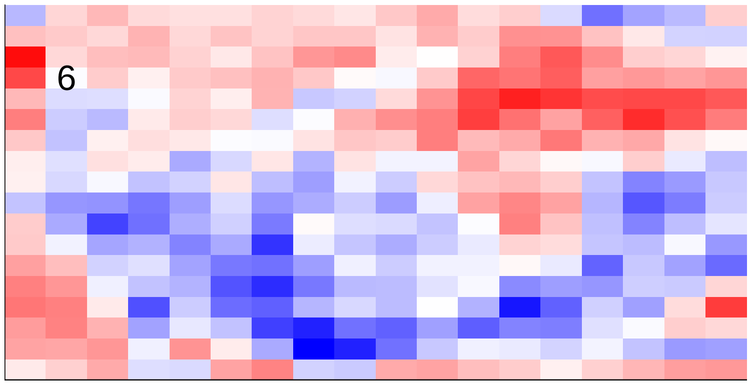


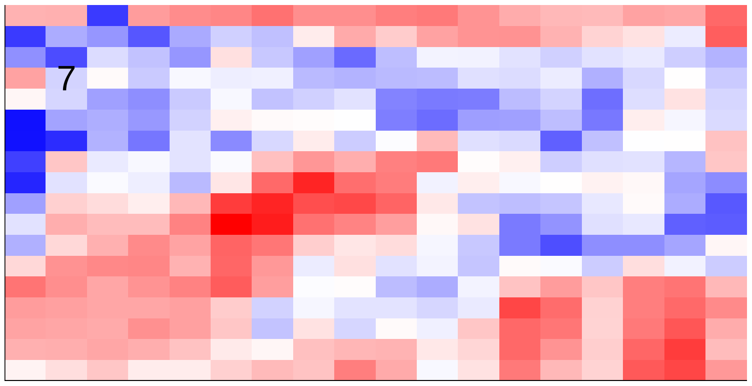

2

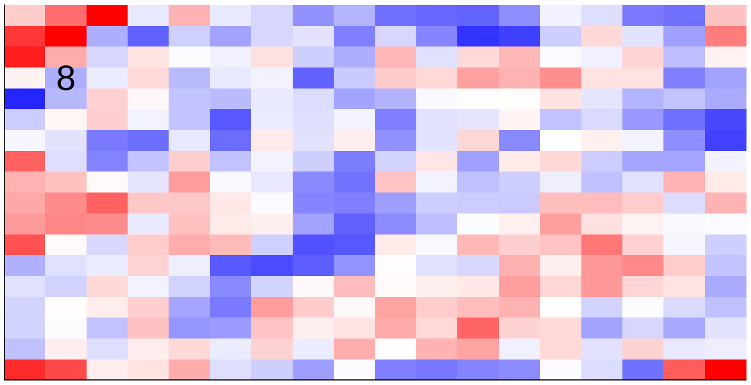

3

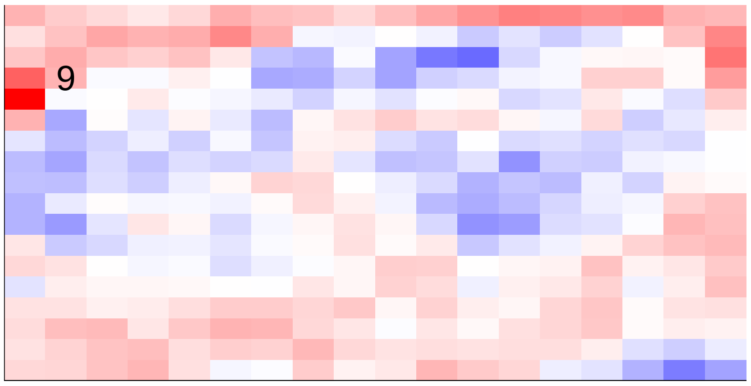

4


5


6


7





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

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

Multinomial Model II

```r
library(tensorflow)
```

```
## Warning: package 'tensorflow' was built under R version 4.2.3
```

```
##
## Attaching package: 'tensorflow'
```

```
## The following object is masked from 'package:caret':
##
##     train
```

```r
# 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")
#}

mnist <- dataset_mnist()

# Access the training and testing sets
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

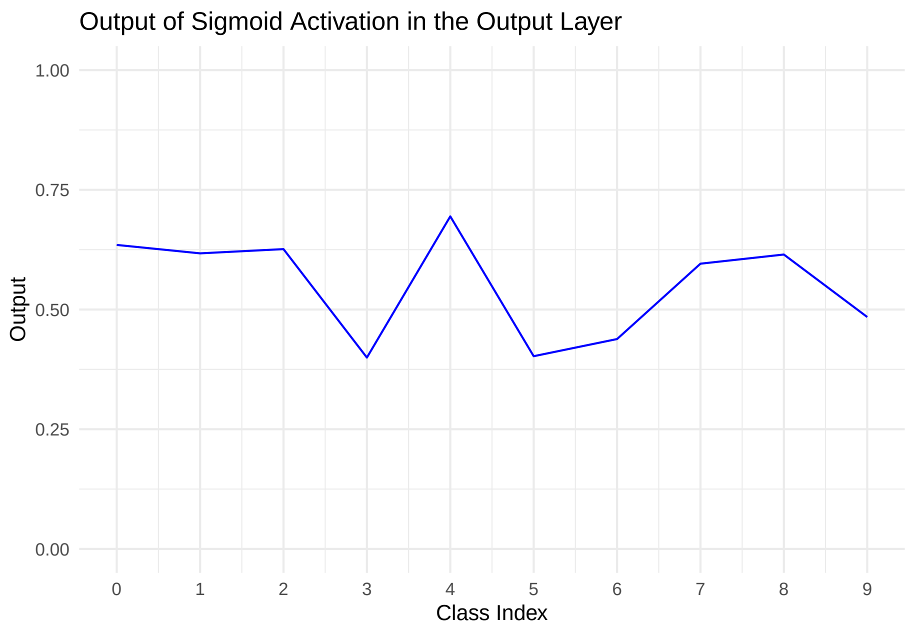
```
# Preprocess the data
train_images <- array_reshape(train_images, c(dim(train_images)[1], 28 * 28)) / 255
test_images <- array_reshape(test_images, c(dim(test_images)[1], 28 * 28)) / 255

# Define the neural network
model2 <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = "sigmoid", input_shape = c(784)) %>%
  layer_dense(units = 64, activation = "sigmoid") %>%
  layer_dense(units = 10, activation = "sigmoid")

# Compile the model
model2 %>% compile(
  optimizer = 'adam',
  loss = 'binary_crossentropy',  # Using binary crossentropy for binary classification
  metrics = c('accuracy')
)

# Display the model summary
summary(model2)
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                     Param #
## ================================================================================
##  dense_2 (Dense)                   (None, 128)                     100480
##  dense_1 (Dense)                   (None, 64)                      8256
##  dense (Dense)                     (None, 10)                      650
## ================================================================================
## Total params: 109386 (427.29 KB)
## Trainable params: 109386 (427.29 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

```
# Assuming 'sample_image' is the input image
sample_image <- test_images[1, , drop = FALSE]

# Reshape the sample image back to its original 28x28 shape
sample_image_reshaped <- array_reshape(sample_image, c(28, 28))

# Get the output of the first hidden layer
layer1_output <- predict(model2, sample_image)
```

```
## 1/1 - 0s - 162ms/epoch - 162ms/step
```

```
layer1_output <- as.vector(layer1_output)

# Plot the result of the sigmoid activation for all 128 units in a line chart
ggplot() +
  geom_line(aes(x = seq_along(layer1_output)-1, y = layer1_output),
            color = "blue") +
  labs(title = "Output of Sigmoid Activation in the Output Layer",
       x = "Class Index",
       y = "Output") +
  theme_minimal() +
  coord_cartesian(ylim = c(0, 1)) +  # Set y-axis limit
  scale_x_continuous(breaks = seq(0, 9, by = 1), limits = c(0, 9))  # Set x-axis breaks
```



$$Sigmoid\ \ Function: f(x) = \frac{1}{1+e^{-x}}$$

```
# Preprocess the data
train_images <- array_reshape(train_images, c(dim(train_images)[1], 28 * 28)) / 255
test_images <- array_reshape(test_images, c(dim(test_images)[1], 28 * 28)) / 255

# Define the neural network
model1 <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = "sigmoid", input_shape = c(28 * 28)) %>%
  layer_dense(units = 10, activation = "softmax")

# Compile the model
model1 %>% compile(
  optimizer = 'Adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy')
)

# Train the model
model1 %>% fit(train_images, train_labels, epochs = 10, batch_size = 64, validation_spl
```

```
## Epoch 1/10
## 750/750 - 2s - loss: 2.2416 - accuracy: 0.2238 - val_loss: 2.1481 - val_accuracy: 0
## Epoch 2/10
## 750/750 - 2s - loss: 2.0098 - accuracy: 0.5084 - val_loss: 1.8288 - val_accuracy: 0.5776 - 2s/
## Epoch 3/10
## 750/750 - 1s - loss: 1.6283 - accuracy: 0.6430 - val_loss: 1.4100 - val_accuracy: 0.7107 - 1s/
## Epoch 4/10
## 750/750 - 2s - loss: 1.2563 - accuracy: 0.7270 - val_loss: 1.0888 - val_accuracy: 0.7563 - 2s/
## Epoch 5/10
## 750/750 - 2s - loss: 0.9948 - accuracy: 0.7777 - val_loss: 0.8710 - val_accuracy: 0.8181 - 2s/
## Epoch 6/10
## 750/750 - 2s - loss: 0.8223 - accuracy: 0.8097 - val_loss: 0.7264 - val_accuracy: 0.8406 - 2s/
## Epoch 7/10
## 750/750 - 2s - loss: 0.7046 - accuracy: 0.8316 - val_loss: 0.6286 - val_accuracy: 0.8532 - 2s/
## Epoch 8/10
## 750/750 - 2s - loss: 0.6222 - accuracy: 0.8464 - val_loss: 0.5576 - val_accuracy: 0.8677 - 2s/
## Epoch 9/10
## 750/750 - 2s - loss: 0.5620 - accuracy: 0.8583 - val_loss: 0.5067 - val_accuracy: 0.8727 - 2s/
## Epoch 10/10
## 750/750 - 2s - loss: 0.5168 - accuracy: 0.8669 - val_loss: 0.4679 - val_accuracy: 0.8820 - 2s/
```

```r
# Evaluate the model on test data
test_loss_and_accuracy <- model1 %>% evaluate(test_images, test_labels)
```

```
## 313/313 - 0s - loss: 0.4779 - accuracy: 0.8762 - 252ms/epoch - 806us/step
```

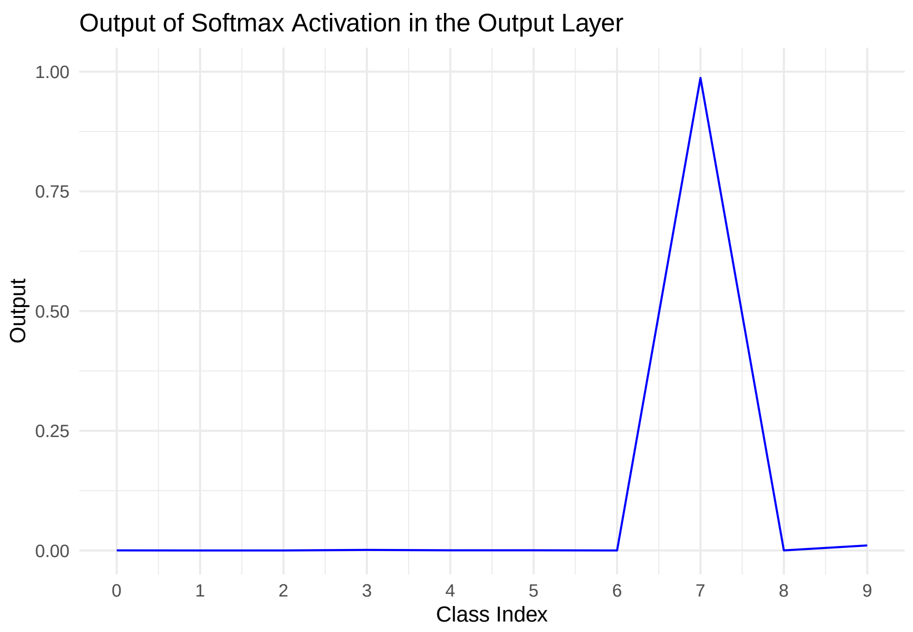
```r
cat("Test loss and accuracy:", test_loss_and_accuracy, "\n")
```

```
## Test loss and accuracy: 0.4779022 0.8762
```

```r
# Sample input image for visualization
sample_image <- test_images[1, , drop = FALSE]
# Reshape the sample image back to its original 28x28 shape
sample_image_reshaped <- array_reshape(sample_image, c(28, 28))

# Predictions from the output layer (softmax activation)
output_layer_output <- predict(model1, sample_image)
```

```
## 1/1 - 0s - 30ms/epoch - 30ms/step
```

```r
output_layer_output <- as.vector(output_layer_output)

# Plot the result of the sigmoid activation in a line chart
ggplot() +
  geom_line(aes(x = seq_along(output_layer_output)-1, y = output_layer_output),
            color = "blue") +
  labs(title = "Output of Softmax Activation in the Output Layer",
       x = "Class Index",
       y = "Output") +
  theme_minimal() +
  coord_cartesian(ylim = c(0, 1)) +  # Set y-axis limit
  scale_x_continuous(breaks = seq(0, 9, by = 1), limits = c(0, 9))  # Set x-axis break
```



$$Softmax\ \ Function : f(x_i) = \frac{e^{-x_i}}{\sum_{i=1}^{N} e^{-x_i}}$$

Neural Network I

```
# Preprocess the data
train_images <- array_reshape(train_images, c(dim(train_images)[1], 28 * 28)) / 255
test_images <- array_reshape(test_images, c(dim(test_images)[1], 28 * 28)) / 255

# Define the neural network
model <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = "sigmoid", input_shape = c(28 * 28)) %>%
  layer_dense(units = 10, activation = "softmax")
```

```
# Compile the model
model %>% compile(
  optimizer = 'Adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy')
)

# Train the model
model %>% fit(train_images, train_labels, epochs = 10, batch_size = 64, validation_split = 0.2)
```

```
## Epoch 1/10
## 750/750 - 2s - loss: 2.3098 - accuracy: 0.1055 - val_loss: 2.3059 - val_accuracy: 0.1081 - 2s/
## Epoch 2/10
## 750/750 - 2s - loss: 2.3069 - accuracy: 0.1052 - val_loss: 2.3140 - val_accuracy: 0.1060 - 2s/
## Epoch 3/10
## 750/750 - 2s - loss: 2.3065 - accuracy: 0.1068 - val_loss: 2.3130 - val_accuracy: 0.1060 - 2s/
## Epoch 4/10
## 750/750 - 2s - loss: 2.3064 - accuracy: 0.1056 - val_loss: 2.3071 - val_accuracy: 0.0997 - 2s/
## Epoch 5/10
## 750/750 - 2s - loss: 2.3066 - accuracy: 0.1055 - val_loss: 2.3080 - val_accuracy: 0.1060 - 2s/
## Epoch 6/10
## 750/750 - 1s - loss: 2.3064 - accuracy: 0.1051 - val_loss: 2.3095 - val_accuracy: 0.1081 - 1s/
## Epoch 7/10
## 750/750 - 2s - loss: 2.3066 - accuracy: 0.1045 - val_loss: 2.3076 - val_accuracy: 0.0956 - 2s/
## Epoch 8/10
## 750/750 - 1s - loss: 2.3046 - accuracy: 0.1078 - val_loss: 2.3116 - val_accuracy: 0.1035 - 1s/
## Epoch 9/10
## 750/750 - 2s - loss: 2.3054 - accuracy: 0.1086 - val_loss: 2.3069 - val_accuracy: 0.1060 - 2s/
## Epoch 10/10
## 750/750 - 1s - loss: 2.3058 - accuracy: 0.1089 - val_loss: 2.3102 - val_accuracy: 0.1035 - 1s/
```

```
# Evaluate the model on test data
test_loss_and_accuracy <- model %>% evaluate(test_images, test_labels)
```

```
## 313/313 - 0s - loss: 2.3078 - accuracy: 0.1010 - 245ms/epoch - 783us/step
```

```
cat("Test loss and accuracy:", test_loss_and_accuracy, "\n")
```

```
## Test loss and accuracy: 2.307767 0.101
```

Reshapes the 28x28 images into flat vectors of size 784 Normalizes the pixel values to the range [0, 1] by dividing by 255

Creates a sequential model using keras_model_sequential() Adds a dense hidden layer with 128 neurons and a sigmoid activation function. Adds the output layer with 10 neurons (one for each digit) and a softmax activation function.

Compiles the model with the Adam optimizer, sparse categorical crossentropy loss function, and accuracy as the evaluation metric.

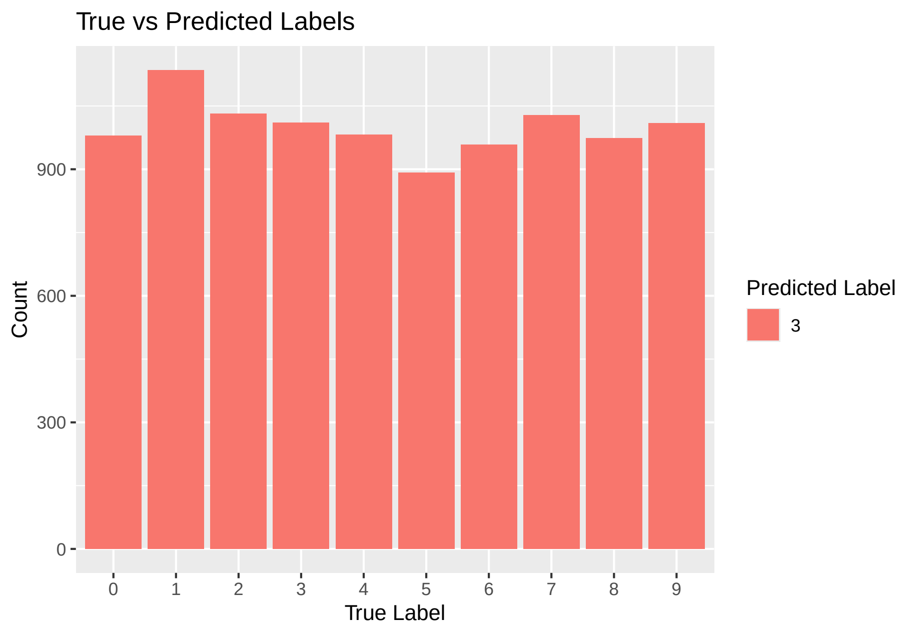
Trains the model on the training data for 5 epochs with a batch size of 64 and a validation split of 20%

Evaluates the trained model on the test data and prints the test accuracy

```
# Make predictions using the trained model
predictions <- model %>% predict(test_images)
```

```
## 313/313 - 0s - 236ms/epoch - 754us/step
```

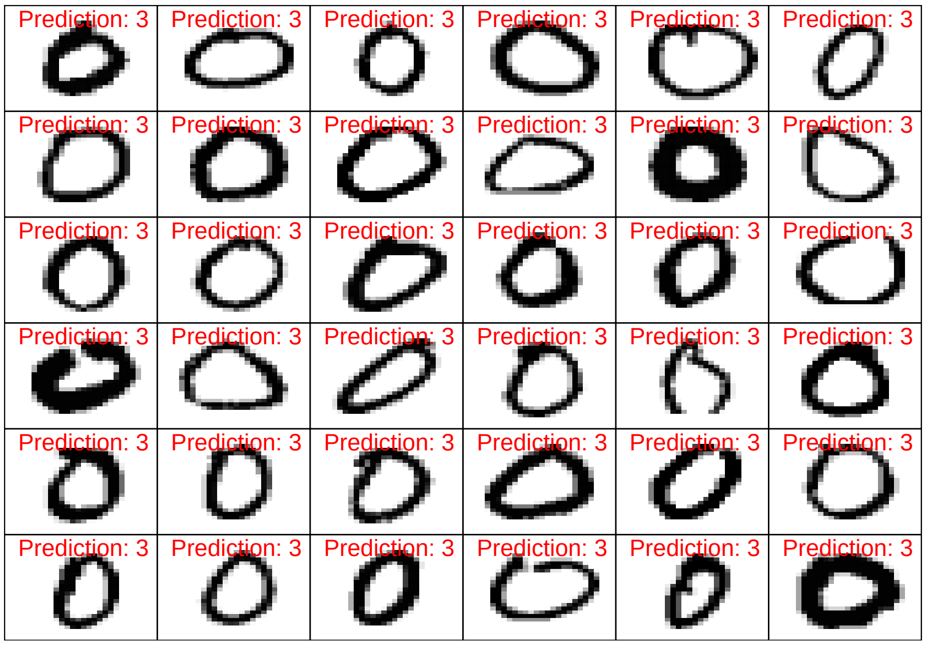
```
# Extract the class with the maximum probability for each sample
predicted_labels <- apply(predictions, 1, function(x) which.max(x) - 1)

# Create a data frame for plotting
plot_data <- data.frame(
  TrueLabel = as.factor(test_labels),
  PredictedLabel = as.factor(predicted_labels)
)

# Plot the results using ggplot2
ggplot(plot_data, aes(x = TrueLabel, fill = PredictedLabel)) +
  geom_bar(position = "dodge") +
  labs(title = "True vs Predicted Labels",
       x = "True Label",
       y = "Count",
       fill = "Predicted Label")
```



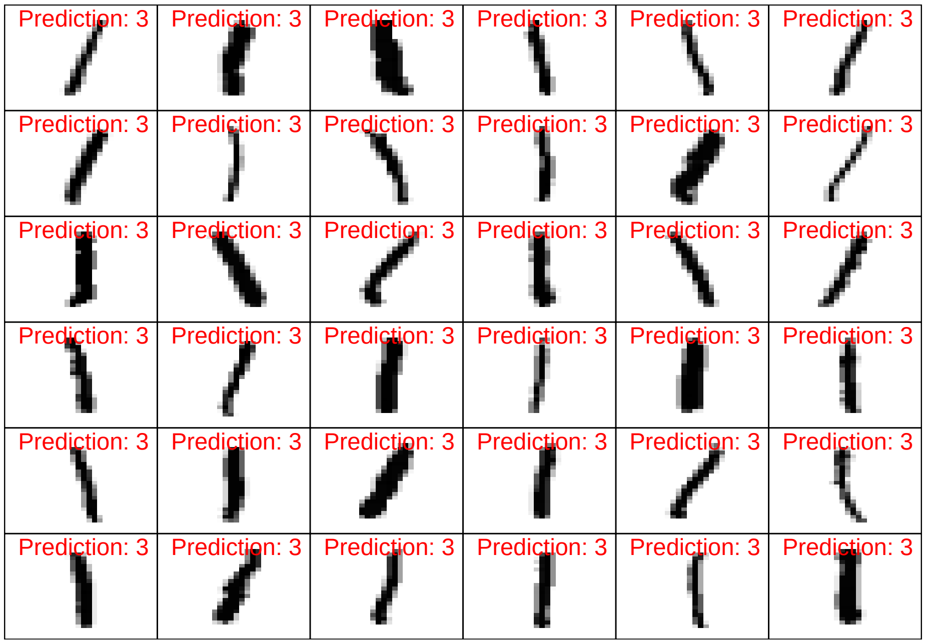
```r
# Define the updated plot_mnist function
plot_mnist <- function(plt, predictions) {
  ## create image
  image(x = 1:28,
        y = 1:28,
        ## image is oriented incorrectly, this fixes it
        z = t(apply(plt, 1, rev)),
        ## 255:0 puts black on white canvas,
        ## changing to 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)

  ## display prediction result
  text(x = 15, y = 25, labels = paste("Prediction:", predictions), col = "red", cex = 1.5)
}

# Display the first 36 predictions for each digit
```

```r
par(mfcol = c(6, 6))
par(mar = c(0, 0, 0, 0))

for (digit in 0:9) {
  cat("Digit", digit, " Predictions:\n")
  digit_indices <- which(test_labels == digit)[1:36]

  for (i in digit_indices) {
    # Reshape the flattened image vector into a 28x28 matrix
    img <- matrix(test_images[i, ] * 255, nrow = 28, ncol = 28)
    plot_mnist(img, predicted_labels[i])
  }
}
```

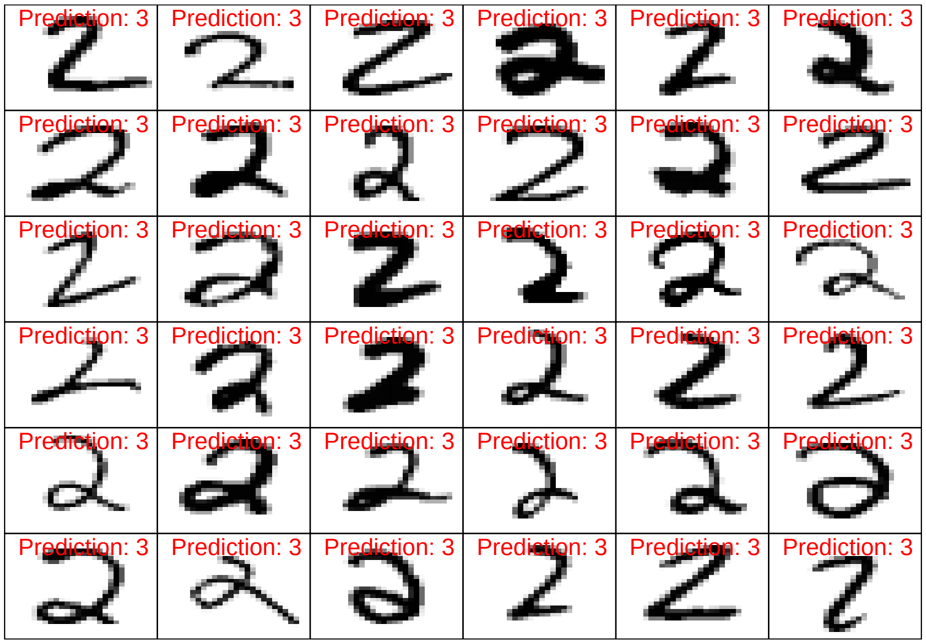
```
## Digit 0  Predictions:
```



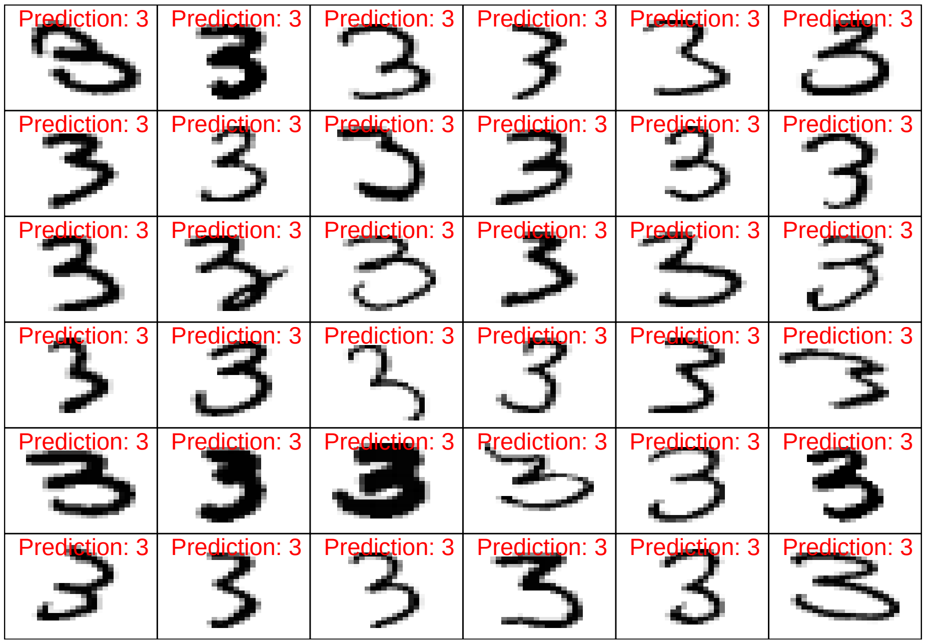
```
## Digit 1  Predictions:
```



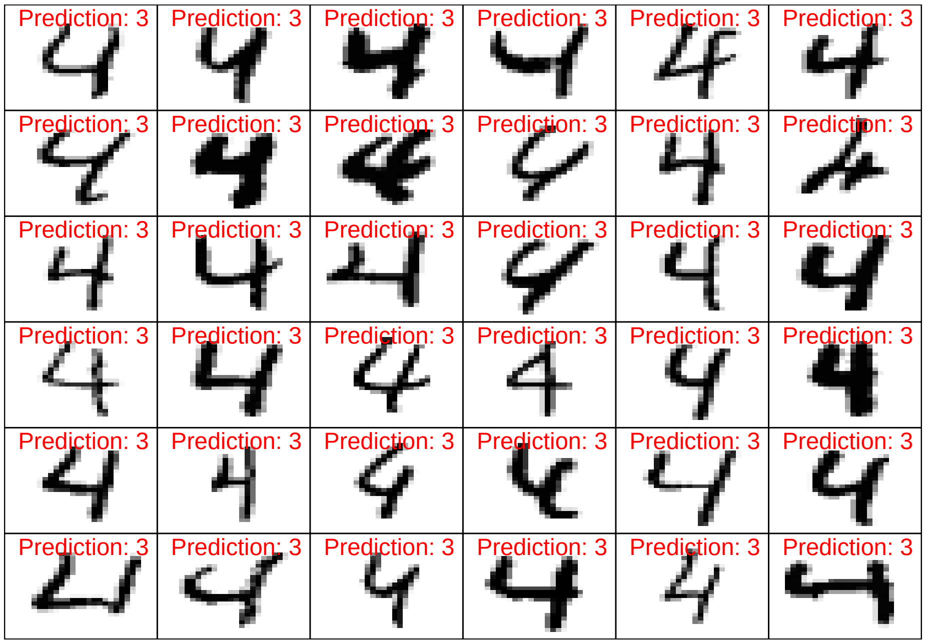
```
## Digit 2  Predictions:
```



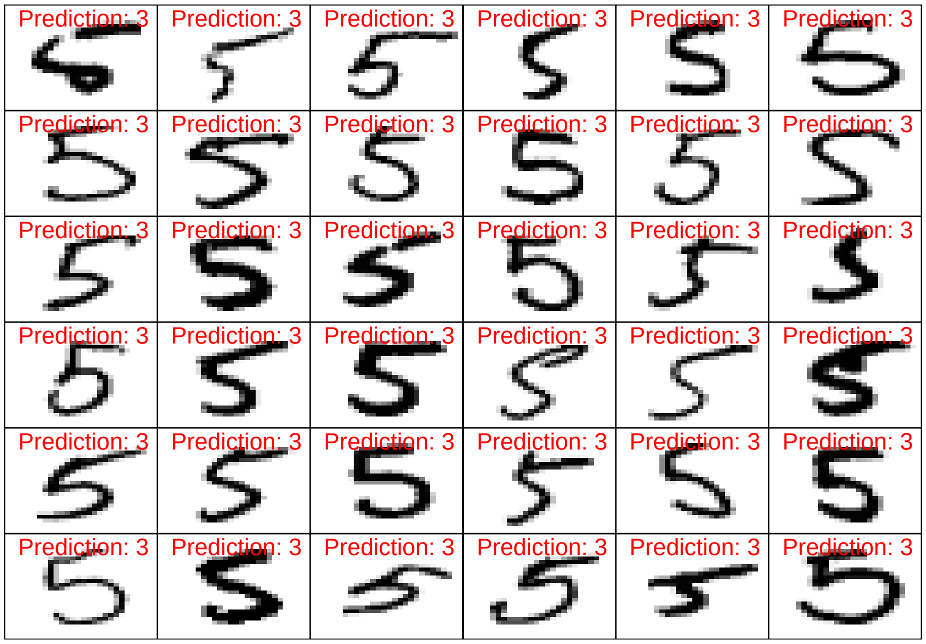
```
## Digit 3  Predictions:
```



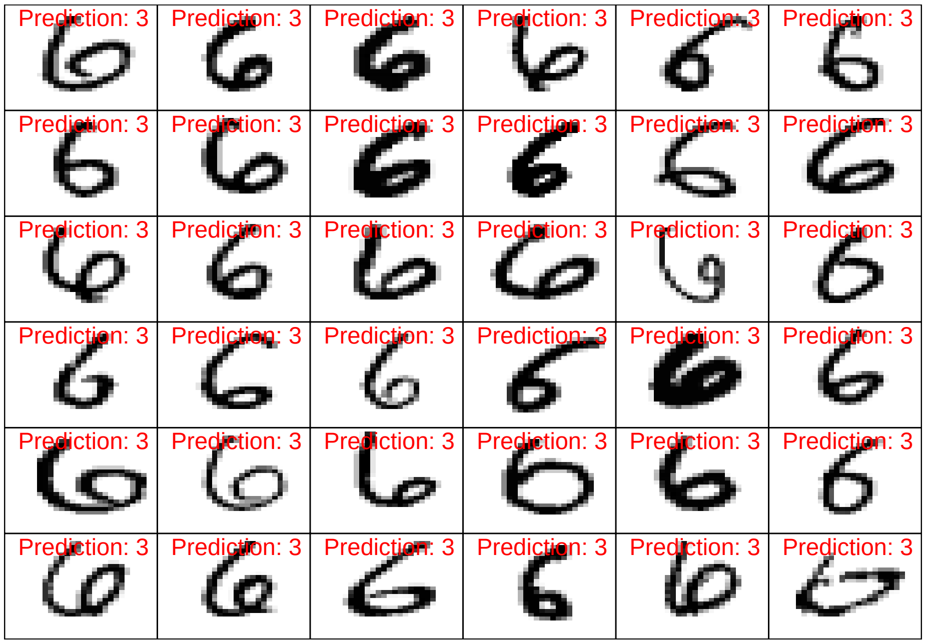
```
## Digit 4  Predictions:
```



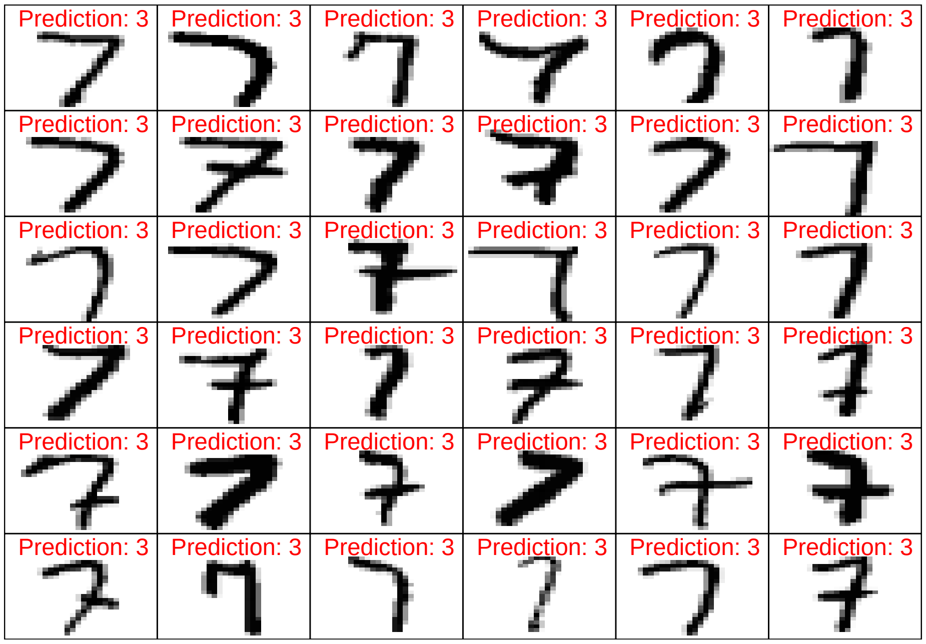
```
## Digit 5  Predictions:
```



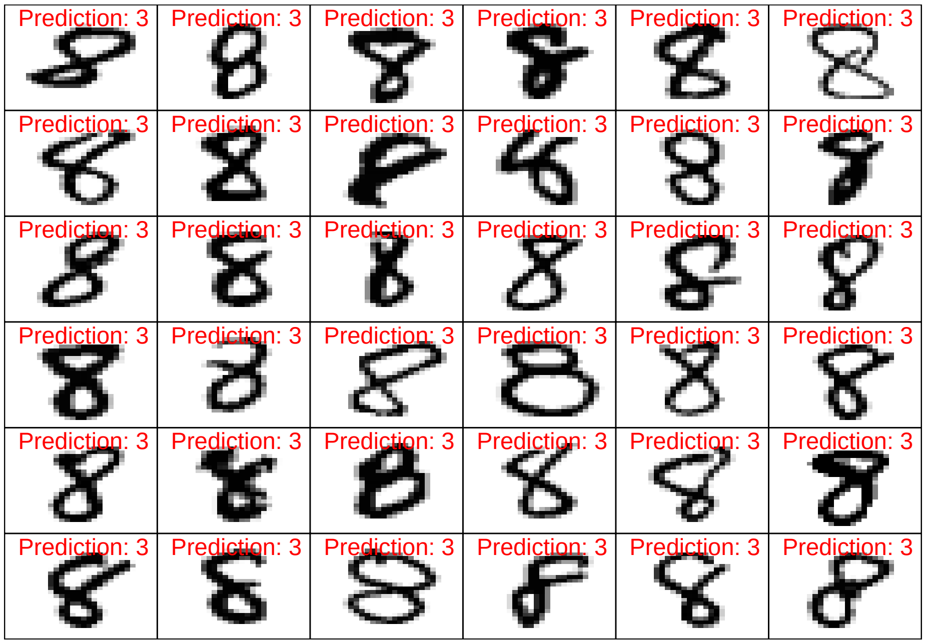
```
## Digit 6  Predictions:
```



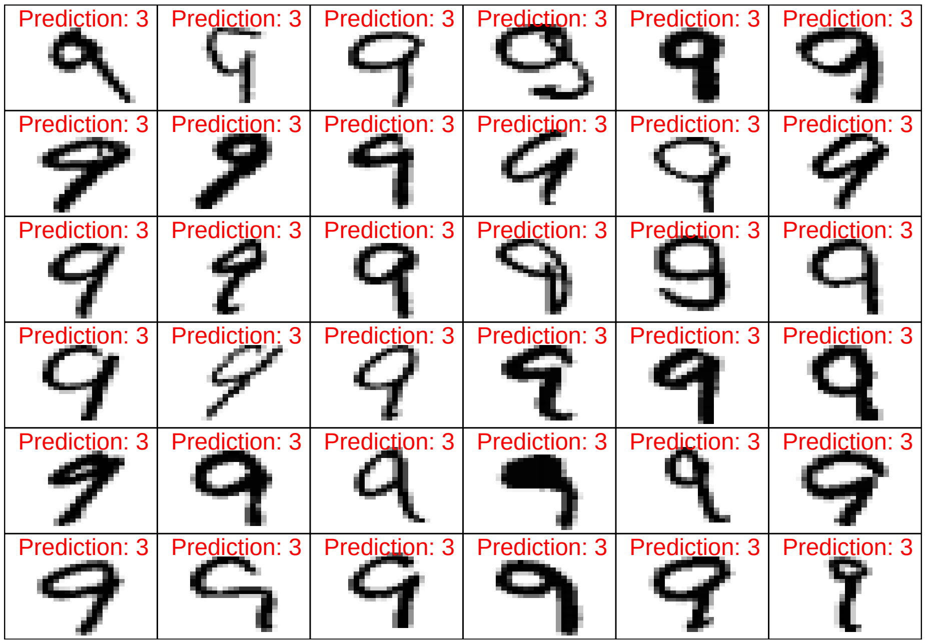
```
## Digit 7  Predictions:
```



```
## Digit 8  Predictions:
```



```
## Digit 9  Predictions:
```



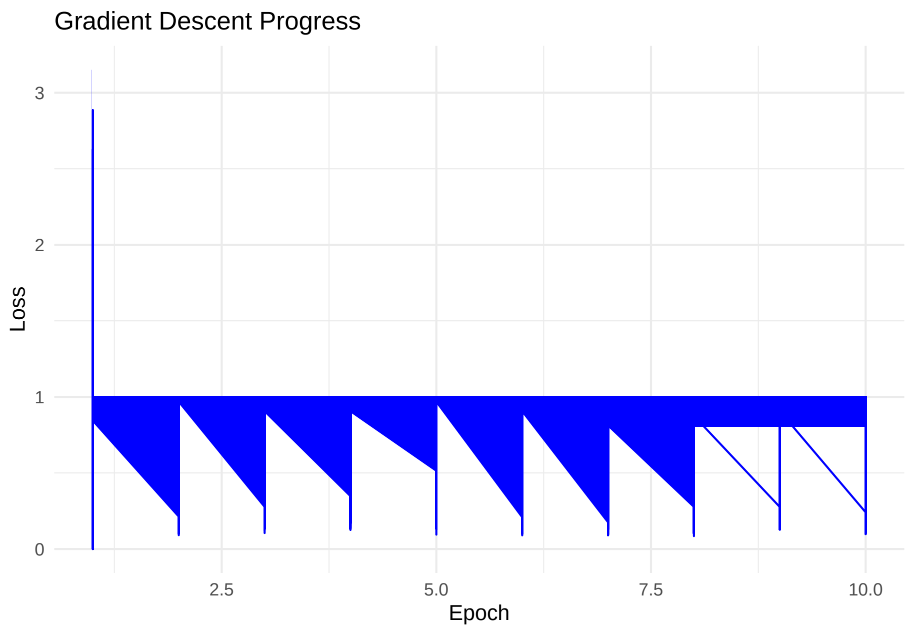
```r
par(mfcol = c(1, 1))
```

Neural Network II

```r
# Function to perform gradient descent
gradient_descent <- function(model, x, y, learning_rate, epochs, batch_size) {
  history <- data.frame(epoch = integer(), loss = numeric())

  for (epoch in 1:epochs) {
    # Shuffle the data for each epoch
    indices <- sample(1:nrow(x))
    x <- x[indices, ]
    y <- y[indices]

    for (i in seq(1, nrow(x), batch_size)) {
      end_index <- min(i + batch_size - 1, nrow(x))  # Ensure the end index is within bounds
      x_batch <- x[i:end_index, ]
      y_batch <- y[i:end_index]

      # Compute gradient and update weights
      gradients <- compute_gradients(model, x_batch, y_batch)
      model <- update_weights(model, gradients, learning_rate)
      # Record loss for plotting
      loss <- model %>% evaluate(x_batch, y_batch, verbose = 0)
      history <- rbind(history, data.frame(epoch = epoch, loss = loss))
    }
  }

  return(list(model = model, history = history))
}


# Function to compute gradients using tf$GradientTape
compute_gradients <- function(model, x, y) {
  with(tf$GradientTape(persistent = TRUE) %as% tape, {
    # Forward pass
    predictions <- model(x)
    loss <- keras$losses$sparse_categorical_crossentropy(y, predictions)
    loss_value <- tf$reduce_mean(loss)

    # Compute gradients
    grads <- tape$gradient(loss_value, model$trainable_variables)
    return(list(loss = loss_value, grads = grads))
  })
}

# Function to update weights using gradients
update_weights <- function(model, gradients, learning_rate) {
  weights <- get_weights(model)
  for (i in seq_along(weights)) {
    weights[[i]] <- weights[[i]] - learning_rate * gradients$grads[[i]]
  }
  set_weights(model, weights)
  return(model)
}
```

```r
# Convert labels to one-hot encoding
train_labels_onehot <- to_categorical(train_labels, 10)
# Flatten train_images and test_images to (60000, 28 * 28)
train_images <- array_reshape(train_images, c(dim(train_images)[1], 28 * 28)) / 255
test_images <- array_reshape(test_images, c(dim(test_images)[1], 28 * 28)) / 255

# Define the neural network
model1 <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = "sigmoid", input_shape = c(28 * 28)) %>%
  layer_dense(units = 10, activation = "softmax")
```

```r
# Set hyperparameters
learning_rate <- 0.01
epochs <- 10
batch_size <- 64

# Compile the model
model1 %>% compile(
  optimizer = 'Adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy')
)

# Perform gradient descent
result <- gradient_descent(model1, train_images, train_labels_onehot, learning_rate, epochs, batc

# Plot the loss during training
ggplot(result$history, aes(x = epoch, y = loss)) +
  geom_line(color = "blue") +
  labs(title = "Gradient Descent Progress", x = "Epoch", y = "Loss") +
  theme_minimal()
```



```r
# Train the model
model1 %>% fit(train_images, train_labels, epochs = 10, batch_size = 64, validation_split = 0.2)
```

```
## Epoch 1/10
## 750/750 - 6s - loss: 2.5244 - accuracy: 0.1046 - val_loss: 2.3108 - val_accuracy: 0
## Epoch 2/10
## 750/750 - 5s - loss: 2.3060 - accuracy: 0.1056 - val_loss: 2.3054 - val_accuracy: 0
## Epoch 3/10
## 750/750 - 5s - loss: 2.3065 - accuracy: 0.1042 - val_loss: 2.3099 - val_accuracy: 0
## Epoch 4/10
## 750/750 - 5s - loss: 2.3063 - accuracy: 0.1051 - val_loss: 2.3060 - val_accuracy: 0
## Epoch 5/10
## 750/750 - 5s - loss: 2.3066 - accuracy: 0.1077 - val_loss: 2.3071 - val_accuracy: 0
## Epoch 6/10
## 750/750 - 5s - loss: 2.3066 - accuracy: 0.1055 - val_loss: 2.3035 - val_accuracy: 0
## Epoch 7/10
## 750/750 - 5s - loss: 2.3076 - accuracy: 0.1023 - val_loss: 2.3083 - val_accuracy: 0
## Epoch 8/10
## 750/750 - 5s - loss: 2.3075 - accuracy: 0.1046 - val_loss: 2.3072 - val_accuracy: 0
## Epoch 9/10
## 750/750 - 5s - loss: 2.3063 - accuracy: 0.1086 - val_loss: 2.3057 - val_accuracy: 0
## Epoch 10/10
## 750/750 - 5s - loss: 2.3062 - accuracy: 0.1084 - val_loss: 2.3108 - val_accuracy: 0
```

```r
# Evaluate the model on test data
test_loss_and_accuracy <- model1 %>% evaluate(test_images, test_labels)
```

```
## 313/313 - 1s - loss: 2.3095 - accuracy: 0.0980 - 578ms/epoch - 2ms/step
```

```r
cat("Test loss and accuracy:", test_loss_and_accuracy, "\n")
```

```
## Test loss and accuracy: 2.309454 0.098
```

```r
library(plotly)
```

```
## Warning: package 'plotly' was built under R version 4.2.3
```

```
##
## Attaching package: 'plotly'
```

```
## The following object is masked from 'package:MASS':
##
##     select
```

```
## The following object is masked from 'package:ggplot2':
##
##     last_plot
## The following object is masked from 'package:stats':
##
##     filter


## The following object is masked from 'package:graphics':
##
##     layout
```

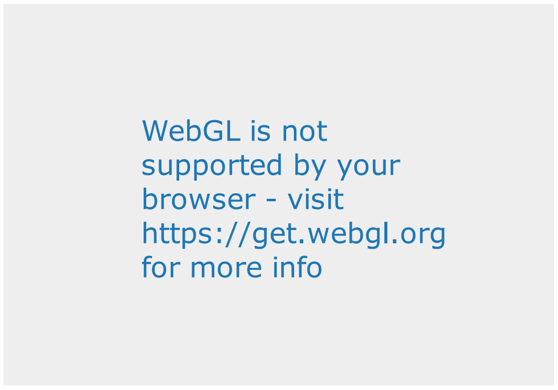
```
# Function to generate background data
generate_bg_data <- function(bg_x, bg_y) {
  z_container <- matrix(0, nrow = length(bg_x), ncol = length(bg_y))

  for (i in seq_along(bg_x)) {
    for (j in seq_along(bg_y)) {
      z_container[i, j] <- z(bg_x[i], bg_y[j])
    }
  }

  return(z_container)
}

# Function to generate max and min data
generate_max_min_data <- function(r) {
  max_num <- floor(r / 2)
  min_num <- floor(-1 * (r / 2))
  return(c(max_num, min_num, max_num, min_num))
}

# Define the z function
z <- function(x, y) {
  return((x^2 + y^2) / 2)
}

# Generate background data
plot_range <- 50
max_min_data <- generate_max_min_data(plot_range)
bg_x <- seq(max_min_data[2], max_min_data[1], length.out = (max_min_data[1] - max_min_data[2] + 1
bg_y <- seq(max_min_data[4], max_min_data[3], length.out = (max_min_data[3] - max_min_data[4] + 1
z_data <- generate_bg_data(bg_x, bg_y)



# Define a custom curved surface function
curved_surface <- function(x, y, W = 1) {
  return(W * (x^2 + y^2) / 2)
```

```r
}

# Generate data for the curved surface
x_1 <- seq(-20, 20, length.out = 500)
y_1 <- seq(-20, 20, length.out = 500)
z_1 <- outer(x_1, y_1, curved_surface)

p <- plot_ly(z = z_1, x = x_1, y = y_1, type = "surface")


# Set hyperparameters
lr <- 0.08
epochs <- 40

# Initialize variables
train_x <- 1
train_y <- 2
w_x <- 0.9
w_y <- 0.9
label_x <- 25
label_y <- 23

# Create 3D plot using plotly
plot <- plot_ly(
  x = rep(bg_x, each = length(bg_y)),
  y = rep(bg_y, times = length(bg_x)),
  z = as.vector(z_data),
  type = "surface",
  colors = colorRamp(c("blue", "red")), opacity = 0.6
) %>%
  layout(scene = list(aspectmode = "cube"))

# Training loop
for (e in 1:epochs) {
  # Predict
  prediction_x <- train_x * w_x
  prediction_y <- train_y * w_y

  # Calculate error
  error_x <- label_x - prediction_x
  error_y <- label_y - prediction_y

  # Display current error
  cat("epoch:", e, ",  error_x:", error_x, ", error_y:", error_y, "\n")
  # Calculate derivatives
  current_z <- z(error_x, error_y)

  # Add points to the 3D plot
  plot <- plot %>% add_trace(x = error_x, y = error_y, z = current_z, type = "scatter3d", mode =

  # Update weights
  if (error_x != 0 || error_y != 0) {
    w_x <- w_x + (error_x * lr) * train_x
    w_y <- w_y + (error_y * lr) * train_y
  } else {
    cat("Error is minimum.\n")
  }
}
```

```
## epoch: 1 ,  error_x: 24.1 , error_y: 21.2
## epoch: 2 ,  error_x: 22.172 , error_y: 14.416
## epoch: 3 ,  error_x: 20.39824 , error_y: 9.80288
## epoch: 4 ,  error_x: 18.76638 , error_y: 6.665958
## epoch: 5 ,  error_x: 17.26507 , error_y: 4.532852
## epoch: 6 ,  error_x: 15.88386 , error_y: 3.082339
## epoch: 7 ,  error_x: 14.61316 , error_y: 2.095991
## epoch: 8 ,  error_x: 13.4441 , error_y: 1.425274
## epoch: 9 ,  error_x: 12.36857 , error_y: 0.9691861
## epoch: 10 ,  error_x: 11.37909 , error_y: 0.6590465
## epoch: 11 ,  error_x: 10.46876 , error_y: 0.4481516
## epoch: 12 ,  error_x: 9.631261 , error_y: 0.3047431
## epoch: 13 ,  error_x: 8.86076 , error_y: 0.2072253
## epoch: 14 ,  error_x: 8.151899 , error_y: 0.1409132
## epoch: 15 ,  error_x: 7.499747 , error_y: 0.09582099
## epoch: 16 ,  error_x: 6.899767 , error_y: 0.06515827
## epoch: 17 ,  error_x: 6.347786 , error_y: 0.04430762
## epoch: 18 ,  error_x: 5.839963 , error_y: 0.03012918
## epoch: 19 ,  error_x: 5.372766 , error_y: 0.02048785
## epoch: 20 ,  error_x: 4.942945 , error_y: 0.01393173
## epoch: 21 ,  error_x: 4.547509 , error_y: 0.00947358
## epoch: 22 ,  error_x: 4.183708 , error_y: 0.006442034
## epoch: 23 ,  error_x: 3.849012 , error_y: 0.004380583
## epoch: 24 ,  error_x: 3.541091 , error_y: 0.002978797
## epoch: 25 ,  error_x: 3.257804 , error_y: 0.002025582
## epoch: 26 ,  error_x: 2.997179 , error_y: 0.001377396
## epoch: 27 ,  error_x: 2.757405 , error_y: 0.000936629
## epoch: 28 ,  error_x: 2.536813 , error_y: 0.0006369077
## epoch: 29 ,  error_x: 2.333868 , error_y: 0.0004330972
## epoch: 30 ,  error_x: 2.147158 , error_y: 0.0002945061
## epoch: 31 ,  error_x: 1.975386 , error_y: 0.0002002642
## epoch: 32 ,  error_x: 1.817355 , error_y: 0.0001361796
## epoch: 33 ,  error_x: 1.671966 , error_y: 9.260215e-05
## epoch: 34 ,  error_x: 1.538209 , error_y: 6.296946e-05
## epoch: 35 ,  error_x: 1.415152 , error_y: 4.281923e-05
## epoch: 36 ,  error_x: 1.30194 , error_y: 2.911708e-05
## epoch: 37 ,  error_x: 1.197785 , error_y: 1.979961e-05
## epoch: 38 ,  error_x: 1.101962 , error_y: 1.346374e-05
## epoch: 39 ,  error_x: 1.013805 , error_y: 9.155341e-06
## epoch: 40 ,  error_x: 0.9327007 , error_y: 6.225632e-06
```

```
# Define the layout with legend position
plot <- plot %>% layout(scene = list(aspectmode = "cube"),
                        legend = list(x = 0.8, y = 0.8))  # Adjust the x and y values

# Combine the two plots into one
combined_plot <- subplot(
  plot,
  p,
  nrows = 1,
  shareX = TRUE,
  shareY = TRUE
)

# Show the combined plot
combined_plot
```

```
## Warning: 'layout' objects don't have these attributes: 'NA'
## Valid attributes include:
## '_deprecated', 'activeshape', 'annotations', 'autosize', 'autotypenumbers', 'calend
```


WebGL is not supported by your browser - visit https://get.webgl.org for more info

Neural Network III

Softmax regression - the final layer of nerual network

1. Data Preprocessing

```r
# Load the data
mnist <- read.csv("https://pjreddie.com/media/files/mnist_train.csv", nrows = 20000)
colnames(mnist) <- c("Digit", paste("Pixel", seq(1:784), sep = ""))

# Normalize the pixel values to range [0, 1]
mnist[, -1] <- mnist[, -1] / 255

# Split the dataset into features (X) and labels (y)
X <- as.matrix(mnist[, -1])
y <- mnist$Digit
```

Explanation: Preprocessing is crucial for machine learning models. Normalizing the pixel values helps in speeding up the convergence during training by ensuring that all features (in this case, pixels) have similar scales. Here, we divide by 255 to scale the pixel values to a [0, 1] range. We also separate the dataset into features (X) and labels (y), which will be used for training the softmax regression model.

2. Initializing Parameters

```
# Initialize weights and biases
num_classes <- length(unique(y))
num_features <- ncol(X)
weights <- matrix(runif(num_features * num_classes) - 0.5, nrow = num_features, ncol =
biases <- runif(num_classes) - 0.5
```

Explanation: Initialization plays a role in the optimization landscape that the training process will navigate. Here, weights and biases are initialized randomly, providing a starting point for optimization. The weight matrix dimensions allow for the calculation of class scores for each of the 10 digits given an input image. Each column in the weight matrix corresponds to a specific class (digit), and biases are used to adjust the output scores independently of the input features.

3. Softmax Function

```
softmax <- function(scores) {
  exp_scores <- exp(scores)
  probs <- sweep(exp_scores, 1, rowSums(exp_scores), '/')
  return(probs)
}
```

Explanation: The softmax function ensures that the output values are interpretable as probabilities, as they are positive and sum up to 1 for each input. This function is essential for multi-class classification tasks like MNIST, where each output corresponds to the model's confidence in each class. Applying softmax allows us to use cross-entropy as a loss function, which measures the difference between the predicted probabilities and the actual distribution of the labels.

4. Cross-Entropy Loss

```
cross_entropy_loss <- function(probs, y) {
  # Convert labels to one-hot encoding
  y_one_hot <- matrix(0, nrow = nrow(probs), ncol = num_classes)
  y_one_hot[cbind(1:nrow(probs), y + 1)] <- 1  # R is 1-indexed

  # Compute the loss
  loss <- -sum(y_one_hot * log(probs)) / nrow(probs)
  return(loss)
}
```

Explanation: Cross-entropy loss is a key component in training classification models, as it provides a measure of how different the predicted probabilities are from the actual labels. In this implementation, we first convert the labels into a one-hot encoded format, where each label is represented as a vector of zeros except for a single one at the index corresponding to the class label. The loss is calculated by taking the negative log of the predicted probabilities for the actual classes and averaging over all examples. This loss function encourages the model to assign high probabilities to the correct classes.

5. Forward Propagation and Class Score Calculation

```
forward_propagation <- function(X, weights, biases) {
  scores <- X %*% weights + matrix(rep(biases, each=nrow(X)), nrow=nrow(X), ncol=ncol(weights), b
  probs <- softmax(scores)
  return(list(scores=scores, probs=probs))
}
```

Explanation: Forward propagation calculates the weighted sum of inputs plus biases for each class, which are the raw class scores. The softmax function is then applied to these scores to derive probabilities. Each probability indicates the likelihood of an input belonging to a particular class. This step is crucial for both training (to compute the loss and gradients) and making predictions.

6. Computing the Gradient

```
compute_gradients <- function(X, y, probs, num_classes) {
  # Convert labels to one-hot encoding
  y_one_hot <- matrix(0, nrow = nrow(probs), ncol = num_classes)
  y_one_hot[cbind(1:nrow(probs), y + 1)] <- 1

  # Compute gradients
  dW <- t(X) %*% (probs - y_one_hot) / nrow(X)
  dB <- colSums(probs - y_one_hot) / nrow(X)

  return(list(dW=dW, dB=dB))
}
```

Explanation: The gradient computation is a critical step in training neural networks. Here, dW represents the gradient of the loss with respect to the weights, and dB represents the gradient with respect to the biases. These gradients indicate how much a change in each parameter would affect the loss. By subtracting a portion of these gradients from the parameters, we can iteratively reduce the loss, making our model more accurate

7. Parameter Update (Gradient Descent)

```r
update_parameters <- function(weights, biases, dW, dB, learning_rate) {
  weights <- weights - learning_rate * dW
  biases <- biases - learning_rate * dB
  return(list(weights=weights, biases=biases))
}
```

Explanation: This function updates the model parameters in the direction that reduces the loss, as indicated by the gradients. The learning rate controls how big a step we take during each update. If the learning rate is too high, we might overshoot the minimum; if it's too low, training might take too long.

8. Model Evaluation

```r
evaluate_accuracy <- function(probs, y) {
  predictions <- max.col(probs) - 1
  accuracy <- sum(predictions == y) / length(y)
  return(accuracy)
}
```

Explanation: After training, evaluating the model's performance on unseen data is crucial. Accuracy measures the proportion of correctly predicted instances. It provides a straightforward metric to gauge how well our model generalizes from the training data to new, unseen data.

9. Model Encapsulation

```r
train_model <- function(X, y, num_classes, learning_rate, epochs, batch_size) {
  num_features <- ncol(X)

  # Initialize weights and biases
  weights <- matrix(runif(num_features * num_classes) - 0.5, nrow = num_features, ncol
  biases <- runif(num_classes) - 0.5

  for (epoch in 1:epochs) {
    # Mini-batch gradient descent
    for (i in seq(1, nrow(X), by=batch_size)) {
      batch_indices <- i:min(i+batch_size-1, nrow(X))
      X_batch <- X[batch_indices, ]
      y_batch <- y[batch_indices]

      # Forward propagation
      forward_result <- forward_propagation(X_batch, weights, biases)
      probs <- forward_result$probs
      # Compute loss (optional here, mainly for monitoring)
      loss <- cross_entropy_loss(probs, y_batch)

      # Compute gradients
      gradients <- compute_gradients(X_batch, y_batch, probs, num_classes)

      # Update parameters
      update_result <- update_parameters(weights, biases, gradients$dW, gradients$dB, learning_ra
      weights <- update_result$weights
      biases <- update_result$biases
    }

    # Optionally evaluate the model on the training set or a validation set
    if (epoch %% 5 == 0) {
      forward_result <- forward_propagation(X, weights, biases)
      accuracy <- evaluate_accuracy(forward_result$probs, y)
      cat("Epoch", epoch, ": Training accuracy =", accuracy, "\n")
    }
  }

  return(list(weights=weights, biases=biases))
}
```

Explanation:

Initialization: We start by initializing the weights and biases with small random values. Training Loop: The outer loop iterates over the number of epochs. Each epoch represents a full pass through the training dataset. Mini-Batch Gradient Descent: Within each epoch, the dataset is divided into mini-batches. This approach helps to stabilize the gradient computation and can lead to faster convergence. Forward Propagation: For each batch, we compute the class probabilities using the current weights and biases. Loss Computation: While not strictly necessary for the update step, calculating the loss allows us to monitor training progress. Gradient Computation: We compute the gradients of the loss with respect to the weights and biases. Parameter Update: Using these gradients, we update the model parameters. Evaluation: Periodically, we evaluate the model's performance on the entire dataset (or a separate validation set) to monitor its accuracy. This function represents a basic framework for training a softmax regression model. In practice, you might enhance this process with additional features like validation checks, early stopping, learning rate schedules, or regularization to improve the model's performance and prevent overfitting.

10. Model Training

```r
# Set hyperparameters
learning_rate <- 0.01
epochs <- 200  # Number of passes through the entire dataset
batch_size <- 100  # Number of training examples used in one iteration
num_classes <- length(unique(y))

# Train the model
model <- train_model(X, y, num_classes, learning_rate, epochs, batch_size)
```

```
## Epoch 5 : Training accuracy = 0.72125
## Epoch 10 : Training accuracy = 0.8064
## Epoch 15 : Training accuracy = 0.83665
## Epoch 20 : Training accuracy = 0.85305
## Epoch 25 : Training accuracy = 0.8639
## Epoch 30 : Training accuracy = 0.8705
## Epoch 35 : Training accuracy = 0.87615
## Epoch 40 : Training accuracy = 0.8798
## Epoch 45 : Training accuracy = 0.88365
## Epoch 50 : Training accuracy = 0.88635
## Epoch 55 : Training accuracy = 0.8892
## Epoch 60 : Training accuracy = 0.8914
## Epoch 65 : Training accuracy = 0.8933
## Epoch 70 : Training accuracy = 0.89555
## Epoch 75 : Training accuracy = 0.89745
## Epoch 80 : Training accuracy = 0.89885
## Epoch 85 : Training accuracy = 0.90045
## Epoch 90 : Training accuracy = 0.9022
## Epoch 95 : Training accuracy = 0.90355
## Epoch 100 : Training accuracy = 0.9044
## Epoch 105 : Training accuracy = 0.9053
## Epoch 110 : Training accuracy = 0.9062
## Epoch 115 : Training accuracy = 0.9074
## Epoch 120 : Training accuracy = 0.909
## Epoch 125 : Training accuracy = 0.90955
## Epoch 130 : Training accuracy = 0.91035
## Epoch 135 : Training accuracy = 0.91095
## Epoch 140 : Training accuracy = 0.91145
## Epoch 145 : Training accuracy = 0.91205
## Epoch 150 : Training accuracy = 0.9126
## Epoch 155 : Training accuracy = 0.9136
## Epoch 160 : Training accuracy = 0.91465
## Epoch 165 : Training accuracy = 0.9151
## Epoch 170 : Training accuracy = 0.9155
## Epoch 175 : Training accuracy = 0.9162
## Epoch 180 : Training accuracy = 0.91685
## Epoch 185 : Training accuracy = 0.91745
## Epoch 190 : Training accuracy = 0.91765
## Epoch 195 : Training accuracy = 0.91805
## Epoch 200 : Training accuracy = 0.91835
```

11. Testing—data Preprocessing

```
# Load the data
mnist_test <- read.csv("https://pjreddie.com/media/files/mnist_train.csv", skip=20000, nrows = 20
colnames(mnist_test) <- c("Digit", paste("Pixel", seq(1:784), sep = ""))

# Normalize the pixel values to range [0, 1]
mnist_test[, -1] <- mnist_test[, -1] / 255

# Split the dataset into features (X) and labels (y)
X_test <- as.matrix(mnist_test[, -1])
y_test <- mnist_test$Digit
```

12. Model Evaluation

```
# Perform forward propagation on the test set to get probabilities
forward_result_test <- forward_propagation(X_test, model$weights, model$biases)
probs_test <- forward_result_test$probs

# Make predictions by selecting the class with the highest probability for each test example
predictions_test <- max.col(probs_test) - 1  # Adjust for R's 1-indexing

# Calculate the accuracy on the test set
accuracy_test <- sum(predictions_test == y_test) / length(y_test)

# Print the test accuracy
cat("Test Accuracy:", accuracy_test, "\n")
```

```
## Test Accuracy: 0.9007
```

Neural Network IV

0. Data Preprocessing

```
# Load the data
mnist <- read.csv("https://pjreddie.com/media/files/mnist_train.csv", nrows = 20000)
colnames(mnist) <- c("Digit", paste("Pixel", seq(1:784), sep = ""))
```

```
# Normalize the pixel values to range [0, 1]
mnist[, -1] <- mnist[, -1] / 255

# Split the dataset into features (X) and labels (y)
X <- as.matrix(mnist[, -1])
y <- mnist$Digit
num_classes <- length(unique(y))
y_one_hot <- matrix(0, nrow = length(y), ncol = num_classes)
y_one_hot[cbind(1:nrow(y_one_hot), y + 1)] <- 1
```

1. Initialize Parameters First, we initialize the weights and biases for a network with three hidden layers. Each layer's weights are initialized randomly to break symmetry, and biases are initialized to zero.

```
initialize_params <- function(input_layer_size, hidden_layer1_size, hidden_layer2_size
  list(
    W1 = matrix(rnorm(input_layer_size * hidden_layer1_size, mean = 0, sd = sqrt(2 / in
    b1 = matrix(0, nrow = 1, ncol = hidden_layer1_size),
    W2 = matrix(rnorm(hidden_layer1_size * hidden_layer2_size, mean = 0, sd = sqrt(2 /
    b2 = matrix(0, nrow = 1, ncol = hidden_layer2_size),
    W3 = matrix(rnorm(hidden_layer2_size * output_layer_size, mean = 0, sd = sqrt(2 / h
    b3 = matrix(0, nrow = 1, ncol = output_layer_size)
  )
}
```

2. Activation Functions Define the RELU and softmax functions for the network's activation:

```
relu <- function(z, alpha=0.01) {
  return(ifelse(z > 0, z, alpha * z))
}

softmax <- function(z) {
  exp(z) / rowSums(exp(z))
}
```

3. Forward Propagation The forward propagation function computes the activation for each layer using the weights, biases, and activation functions:

```
forward_propagation <- function(X, params) {
  Z1 = X %*% params$W1 + matrix(params$b1, nrow = nrow(X), ncol = ncol(params$b1), byro
  A1 = relu(Z1)
  Z2 = A1 %*% params$W2 + matrix(params$b2, nrow = nrow(A1), ncol = ncol(params$b2), byrow = TRUE
  A2 = relu(Z2)
  Z3 = A2 %*% params$W3 + matrix(params$b3, nrow = nrow(A2), ncol = ncol(params$b3), byrow = TRUE
  A3 = softmax(Z3)

  return(list(A1 = A1, A2 = A2, A3 = A3, Z1 = Z1, Z2 = Z2, Z3 = Z3))
}
```

4. Cross-Entropy Loss The cross-entropy loss function evaluates the performance of the model:

```
cross_entropy_loss <- function(y_pred, y_true) {
  -mean(rowSums(y_true * log(y_pred)))
}
```

5. Backpropagation The backpropagation function computes gradients for each parameter in the network by applying the chain rule:

```
backpropagation <- function(X, Y, params, forward_outputs, learning_rate) {
  A1 <- forward_outputs$A1
  A2 <- forward_outputs$A2
  A3 <- forward_outputs$A3
  Z1 <- forward_outputs$Z1
  Z2 <- forward_outputs$Z2


  leaky_relu_derivative <- function(Z, alpha=0.01) {
  dZ <- matrix(alpha, nrow = nrow(Z), ncol = ncol(Z))
  dZ[Z > 0] <- 1
  return(dZ)
}



  dZ3 <- A3 - Y
  dW3 <- t(A2) %*% dZ3 / nrow(X)
  db3 <- colSums(dZ3) / nrow(X)


  dZ2 <- (dZ3 %*% t(params$W3)) * leaky_relu_derivative(Z2)
  dW2 <- t(A1) %*% dZ2 / nrow(X)
  db2 <- colSums(dZ2) / nrow(X)
  dZ1 <- (dZ2 %*% t(params$W2)) * leaky_relu_derivative(Z1)
  dW1 <- t(X) %*% dZ1 / nrow(X)
  db1 <- colSums(dZ1) / nrow(X)

  params$W1 <- params$W1 - learning_rate * dW1
  params$b1 <- params$b1 - learning_rate * db1
  params$W2 <- params$W2 - learning_rate * dW2
  params$b2 <- params$b2 - learning_rate * db2
  params$W3 <- params$W3 - learning_rate * dW3
  params$b3 <- params$b3 - learning_rate * db3

  return(params)
}
```

6. Training the Model Train the model over multiple epochs, adjusting the parameters using the gradients computed by backpropagation:

```
train_model <- function(train_data, train_labels, learning_rate, epochs, hidden_layer_s
  input_layer_size <- ncol(train_data)
  num_classes <- ncol(train_labels)
  params <- initialize_params(input_layer_size, hidden_layer_sizes[1],hidden_layer_size

  for (epoch in 1:epochs) {
    for (i in seq(1, nrow(train_data), by=batch_size)) {
      batch_indices <- i:min(i+batch_size-1, nrow(train_data))
      X_batch <- train_data[batch_indices, ]
      Y_batch <- train_labels[batch_indices, ]

      forward_outputs <- forward_propagation(X_batch, params)
      params <- backpropagation(X_batch, Y_batch, params, forward_outputs, learning_rat
    }

    forward_outputs <- forward_propagation(train_data, params)
    predictions <- max.col(forward_outputs$A3) - 1
    true_labels <- max.col(y_one_hot) - 1
    accuracy <- sum(predictions == true_labels) / length(true_labels)
    cat(sprintf("Epoch %d: Training Accuracy: %.4f\n", epoch, accuracy))
  }

  return(params)
}
```

7. Model Training

```r
learning_rate <- 0.01
epochs <- 200  # Number of passes through the entire dataset
hidden_layer_sizes <- c(512, 256)
batch_size <- 128
# Train the model
model <- train_model(X, y_one_hot, learning_rate, epochs, hidden_layer_sizes, batch_size)
```

```
## Epoch 1: Training Accuracy: 0.7976
## Epoch 2: Training Accuracy: 0.8530
## Epoch 3: Training Accuracy: 0.8771
## Epoch 4: Training Accuracy: 0.8899
## Epoch 5: Training Accuracy: 0.8984
## Epoch 6: Training Accuracy: 0.9044
## Epoch 7: Training Accuracy: 0.9093
## Epoch 8: Training Accuracy: 0.9131
## Epoch 9: Training Accuracy: 0.9156
## Epoch 10: Training Accuracy: 0.9190
## Epoch 11: Training Accuracy: 0.9214
## Epoch 12: Training Accuracy: 0.9234
## Epoch 13: Training Accuracy: 0.9260
## Epoch 14: Training Accuracy: 0.9285
## Epoch 15: Training Accuracy: 0.9306
## Epoch 16: Training Accuracy: 0.9324
## Epoch 17: Training Accuracy: 0.9342
## Epoch 18: Training Accuracy: 0.9356
## Epoch 19: Training Accuracy: 0.9365
## Epoch 20: Training Accuracy: 0.9385
## Epoch 21: Training Accuracy: 0.9398
## Epoch 22: Training Accuracy: 0.9415
## Epoch 23: Training Accuracy: 0.9431
## Epoch 24: Training Accuracy: 0.9440
## Epoch 25: Training Accuracy: 0.9450
## Epoch 26: Training Accuracy: 0.9464
## Epoch 27: Training Accuracy: 0.9475
## Epoch 28: Training Accuracy: 0.9485
## Epoch 29: Training Accuracy: 0.9495
## Epoch 30: Training Accuracy: 0.9507
## Epoch 31: Training Accuracy: 0.9517
## Epoch 32: Training Accuracy: 0.9526
## Epoch 33: Training Accuracy: 0.9535
## Epoch 34: Training Accuracy: 0.9545
## Epoch 35: Training Accuracy: 0.9554
## Epoch 36: Training Accuracy: 0.9567
## Epoch 37: Training Accuracy: 0.9577
## Epoch 38: Training Accuracy: 0.9585
## Epoch 39: Training Accuracy: 0.9595
## Epoch 40: Training Accuracy: 0.9605
## Epoch 41: Training Accuracy: 0.9611
## Epoch 42: Training Accuracy: 0.9620
## Epoch 43: Training Accuracy: 0.9624
## Epoch 44: Training Accuracy: 0.9632
## Epoch 45: Training Accuracy: 0.9638
## Epoch 46: Training Accuracy: 0.9649
## Epoch 47: Training Accuracy: 0.9652
## Epoch 48: Training Accuracy: 0.9656
## Epoch 49: Training Accuracy: 0.9660
## Epoch 50: Training Accuracy: 0.9666
## Epoch 51: Training Accuracy: 0.9672
## Epoch 52: Training Accuracy: 0.9676
## Epoch 53: Training Accuracy: 0.9682
## Epoch 54: Training Accuracy: 0.9687
## Epoch 55: Training Accuracy: 0.9691
## Epoch 56: Training Accuracy: 0.9696
## Epoch 57: Training Accuracy: 0.9705
## Epoch 58: Training Accuracy: 0.9713
## Epoch 59: Training Accuracy: 0.9718
## Epoch 60: Training Accuracy: 0.9726
## Epoch 61: Training Accuracy: 0.9730
## Epoch 62: Training Accuracy: 0.9735
## Epoch 63: Training Accuracy: 0.9739
## Epoch 64: Training Accuracy: 0.9747
## Epoch 65: Training Accuracy: 0.9751
## Epoch 66: Training Accuracy: 0.9756
## Epoch 67: Training Accuracy: 0.9758
## Epoch 68: Training Accuracy: 0.9761
## Epoch 69: Training Accuracy: 0.9765
## Epoch 70: Training Accuracy: 0.9768
## Epoch 71: Training Accuracy: 0.9772
## Epoch 72: Training Accuracy: 0.9780
## Epoch 73: Training Accuracy: 0.9784
## Epoch 74: Training Accuracy: 0.9787
## Epoch 75: Training Accuracy: 0.9790
## Epoch 76: Training Accuracy: 0.9794
## Epoch 77: Training Accuracy: 0.9797
## Epoch 78: Training Accuracy: 0.9799
## Epoch 79: Training Accuracy: 0.9804
## Epoch 80: Training Accuracy: 0.9805
## Epoch 81: Training Accuracy: 0.9809
## Epoch 82: Training Accuracy: 0.9814
## Epoch 83: Training Accuracy: 0.9819
## Epoch 84: Training Accuracy: 0.9822
## Epoch 85: Training Accuracy: 0.9827
## Epoch 86: Training Accuracy: 0.9831
## Epoch 87: Training Accuracy: 0.9836
## Epoch 88: Training Accuracy: 0.9839
## Epoch 89: Training Accuracy: 0.9840
## Epoch 90: Training Accuracy: 0.9842
## Epoch 91: Training Accuracy: 0.9845
## Epoch 92: Training Accuracy: 0.9847
## Epoch 93: Training Accuracy: 0.9850
## Epoch 94: Training Accuracy: 0.9851
## Epoch 95: Training Accuracy: 0.9853
## Epoch 96: Training Accuracy: 0.9856
## Epoch 97: Training Accuracy: 0.9859
## Epoch 98: Training Accuracy: 0.9859
## Epoch 99: Training Accuracy: 0.9860
## Epoch 100: Training Accuracy: 0.9861
## Epoch 101: Training Accuracy: 0.9863
## Epoch 102: Training Accuracy: 0.9865
## Epoch 103: Training Accuracy: 0.9866
## Epoch 104: Training Accuracy: 0.9869
## Epoch 105: Training Accuracy: 0.9870
## Epoch 106: Training Accuracy: 0.9873
## Epoch 107: Training Accuracy: 0.9876
## Epoch 108: Training Accuracy: 0.9879
## Epoch 109: Training Accuracy: 0.9883
## Epoch 110: Training Accuracy: 0.9887
## Epoch 111: Training Accuracy: 0.9889
## Epoch 112: Training Accuracy: 0.9891
## Epoch 113: Training Accuracy: 0.9893
## Epoch 114: Training Accuracy: 0.9897
## Epoch 115: Training Accuracy: 0.9899
## Epoch 116: Training Accuracy: 0.9901
## Epoch 117: Training Accuracy: 0.9903
## Epoch 118: Training Accuracy: 0.9905
## Epoch 119: Training Accuracy: 0.9907
## Epoch 120: Training Accuracy: 0.9909
## Epoch 121: Training Accuracy: 0.9910
## Epoch 122: Training Accuracy: 0.9912
## Epoch 123: Training Accuracy: 0.9913
## Epoch 124: Training Accuracy: 0.9917
## Epoch 125: Training Accuracy: 0.9918
## Epoch 126: Training Accuracy: 0.9919
## Epoch 127: Training Accuracy: 0.9919
## Epoch 128: Training Accuracy: 0.9920
## Epoch 129: Training Accuracy: 0.9922
## Epoch 130: Training Accuracy: 0.9923
## Epoch 131: Training Accuracy: 0.9925
## Epoch 132: Training Accuracy: 0.9927
## Epoch 133: Training Accuracy: 0.9928
## Epoch 134: Training Accuracy: 0.9929
## Epoch 135: Training Accuracy: 0.9932
## Epoch 136: Training Accuracy: 0.9933
## Epoch 137: Training Accuracy: 0.9936
## Epoch 138: Training Accuracy: 0.9939
## Epoch 139: Training Accuracy: 0.9941
## Epoch 140: Training Accuracy: 0.9944
## Epoch 141: Training Accuracy: 0.9944
## Epoch 142: Training Accuracy: 0.9944
## Epoch 143: Training Accuracy: 0.9947
## Epoch 144: Training Accuracy: 0.9949
## Epoch 145: Training Accuracy: 0.9950
## Epoch 146: Training Accuracy: 0.9951
## Epoch 147: Training Accuracy: 0.9952
## Epoch 148: Training Accuracy: 0.9952
## Epoch 149: Training Accuracy: 0.9955
## Epoch 150: Training Accuracy: 0.9957
## Epoch 151: Training Accuracy: 0.9957
## Epoch 152: Training Accuracy: 0.9958
## Epoch 153: Training Accuracy: 0.9960
## Epoch 154: Training Accuracy: 0.9962
## Epoch 155: Training Accuracy: 0.9963
## Epoch 156: Training Accuracy: 0.9965
## Epoch 157: Training Accuracy: 0.9966
## Epoch 158: Training Accuracy: 0.9966
## Epoch 159: Training Accuracy: 0.9967
## Epoch 160: Training Accuracy: 0.9970
## Epoch 161: Training Accuracy: 0.9970
## Epoch 162: Training Accuracy: 0.9970
## Epoch 163: Training Accuracy: 0.9970
## Epoch 164: Training Accuracy: 0.9970
## Epoch 165: Training Accuracy: 0.9970
## Epoch 166: Training Accuracy: 0.9971
## Epoch 167: Training Accuracy: 0.9972
## Epoch 168: Training Accuracy: 0.9972
## Epoch 169: Training Accuracy: 0.9972
## Epoch 170: Training Accuracy: 0.9972
## Epoch 171: Training Accuracy: 0.9972
## Epoch 172: Training Accuracy: 0.9973
## Epoch 173: Training Accuracy: 0.9973
## Epoch 174: Training Accuracy: 0.9974
## Epoch 175: Training Accuracy: 0.9978
## Epoch 176: Training Accuracy: 0.9978
## Epoch 177: Training Accuracy: 0.9978
## Epoch 178: Training Accuracy: 0.9979
## Epoch 179: Training Accuracy: 0.9979
## Epoch 180: Training Accuracy: 0.9980
## Epoch 181: Training Accuracy: 0.9980
## Epoch 182: Training Accuracy: 0.9980
## Epoch 183: Training Accuracy: 0.9980
## Epoch 184: Training Accuracy: 0.9981
## Epoch 185: Training Accuracy: 0.9981
## Epoch 186: Training Accuracy: 0.9981
## Epoch 187: Training Accuracy: 0.9982
## Epoch 188: Training Accuracy: 0.9982
## Epoch 189: Training Accuracy: 0.9984
## Epoch 190: Training Accuracy: 0.9984
## Epoch 191: Training Accuracy: 0.9984
## Epoch 192: Training Accuracy: 0.9985
## Epoch 193: Training Accuracy: 0.9986
## Epoch 194: Training Accuracy: 0.9986
## Epoch 195: Training Accuracy: 0.9986
## Epoch 196: Training Accuracy: 0.9987
## Epoch 197: Training Accuracy: 0.9988
## Epoch 198: Training Accuracy: 0.9988
## Epoch 199: Training Accuracy: 0.9988
## Epoch 200: Training Accuracy: 0.9989
```

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

Notation Setup

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

---

X

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

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

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

W

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

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

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

---

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

Y

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

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

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

---

z

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

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

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

We have that:

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

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

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

---

a

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

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

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

We have that:

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

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

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

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

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

Forward Propagation

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:

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

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

$$= g(\begin{array}{c} \\ n_l \text{ outputs} \left\{ \begin{array}{c} \\ \end{array} \right. \end{array} \overbrace{\begin{bmatrix} w^{(l)}_{1,1} & w^{(l)}_{1,2} & \cdots & w^{(l)}_{1,n_{l-1}} \\ w^{(l)}_{2,1} & w^{(l)}_{2,2} & \cdots & w^{(l)}_{2,n_{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w^{(l)}_{n_l,1} & w^{(l)}_{n_l,2} & \cdots & w^{(l)}_{n_l,n_{l-1}} \end{bmatrix}}^{n_{l-1}\text{ inputs}} * \; n_{l-1} \text{ inputs} \left\{ \overbrace{\begin{bmatrix} a^{(l-1)}_{1,1} & a^{(l-1)}_{1,2} & \cdots & a^{(l-1)}_{1,m} \\ a^{(l-1)}_{2,1} & a^{(l-1)}_{2,2} & \cdots & a^{(l-1)}_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a^{(l-1)}_{n_{l-1},1} & a^{(l-1)}_{n_{l-1},2} & \cdots & a^{(l-1)}_{n_{l-1},m} \end{bmatrix}}^{m\text{ obs}} \right.$$

$$+ \; n_l \text{ outputs} \left\{ \overbrace{\begin{bmatrix} - & b^{(l)}_1 & - \\ - & b^{(l)}_2 & - \\ \vdots & \vdots & \vdots \\ - & b^{(l)}_{n_l} & - \end{bmatrix}}^{m\text{ obs}} \right.)$$

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

$$= \; n_l \text{ outputs} \left\{ \overbrace{\begin{bmatrix} a^{(l)}_{1,1} & a^{(l)}_{1,2} & \cdots & a^{(l)}_{1,m} \\ a^{(l)}_{2,1} & a^{(l)}_{2,2} & \cdots & a^{(l)}_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a^{(l)}_{n_l,1} & a^{(l)}_{n_l,2} & \cdots & a^{(l)}_{n_l,m} \end{bmatrix}}^{m\text{ obs}} \right.$$

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.

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 $a^{(L)}$.

Backward Propagation

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

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

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:

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

This has the same effect as element-wise multiplication.

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

Gradients

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

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

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

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

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

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?]

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:
    - $W^{(l)} \leftarrow W^{(l)} - \rho * \delta^{(l)} * \left(a^{(l-1)}\right)^T$
    - $b^{(l)} \leftarrow W^{(l)} - \rho * \delta^{(l)}$

Multi-Layer NN Model

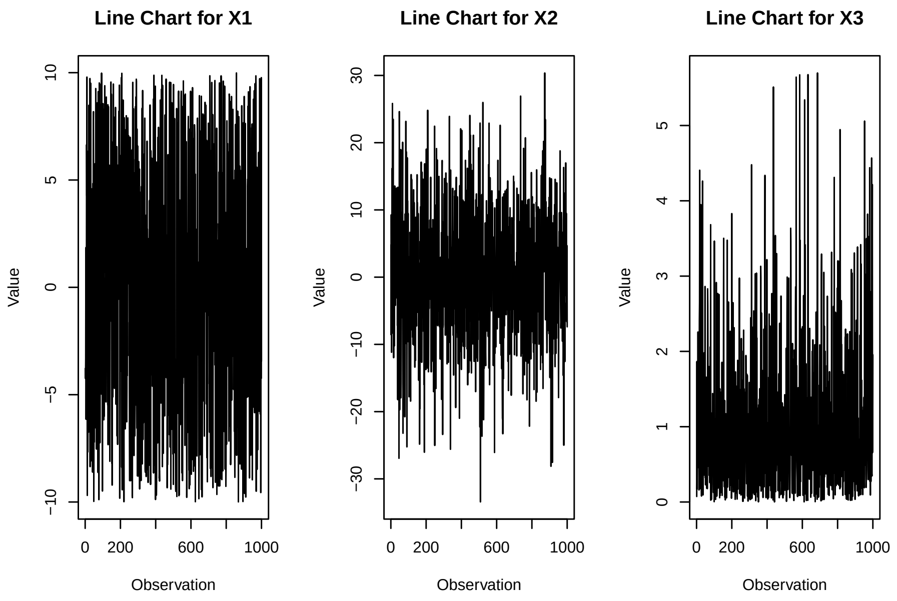
This chapter presents the final functional-programming model. Uses functions to define 'neural networks', perform forward propagation, and perform gradient descent. Section at the end details future components that could be added in.

Generate Data

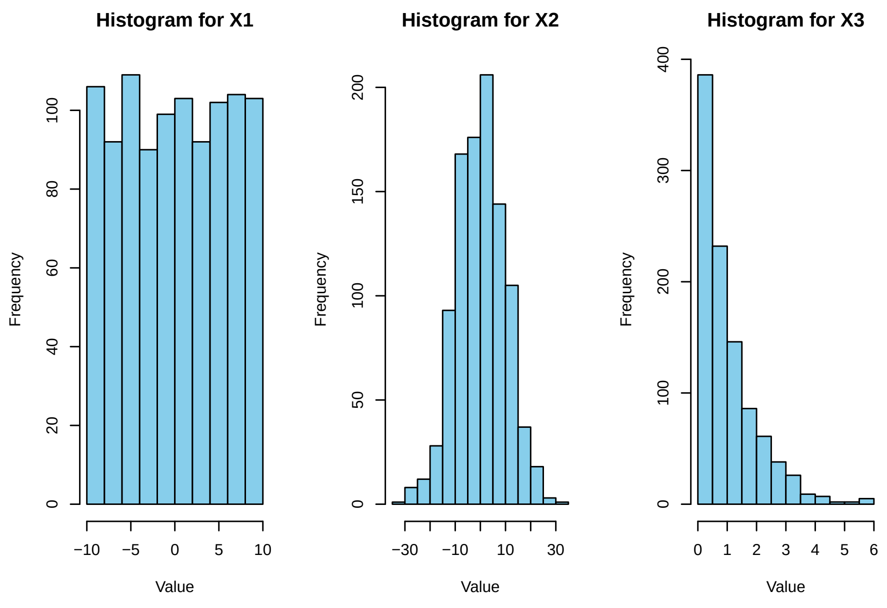
For now, having 3 inputs and combining them to create y, with a random error term. Would like to tweak the setup eventually.

```r
library(tidyverse)

## 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)
```

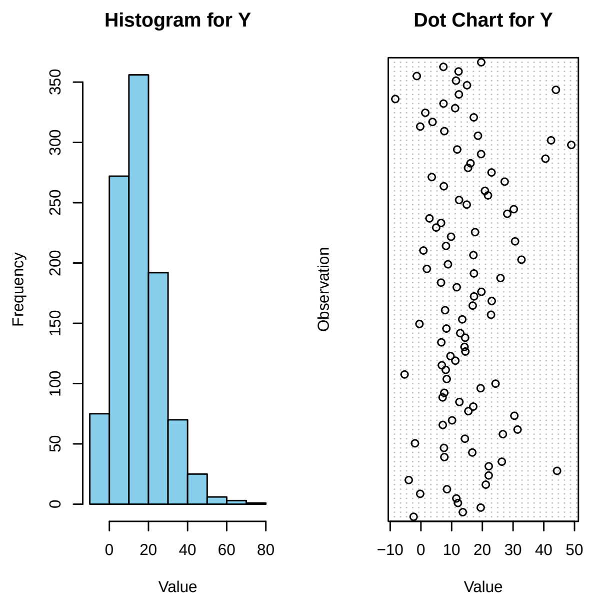
```r
# Create line chart for each variable
par(mfrow = c(1, 3))  # Set up plotting layout
for (i in 1:3) {
  plot(X[i, ], type = "l", main = paste("Line Chart for", rownames(X)[i]),
       xlab = "Observation", ylab = "Value")
}
```



```r
# Create histogram for each variable
par(mfrow = c(1, 3))  # Reset plotting layout
for (i in 1:3) {
  hist(X[i, ], main = paste("Histogram for", rownames(X)[i]),
       xlab = "Value", ylab = "Frequency", col = "skyblue", border = "black")
}
```



```
# Create histogram for Y variable
hist(Y, main = "Histogram for Y", xlab = "Value", ylab = "Frequency", col = "skyblue", border = '

# Select a subset of Y values to display on the dot chart
subset_Y <- Y[seq(1, length(Y), by = 10)]  # Adjust the 'by' value as needed to control the dens

# Create dot chart for Y variable with subset of values
dotchart(subset_Y, main = "Dot Chart for Y", xlab = "Value", ylab = "Observation")
```



Functions

Link Functions

```r
## 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 == "softmax") {
    # softmax
    g <- function(x) {
      exp_x <- exp(x - max(x))  # Subtracting max(x) for numerical stability
      return(exp_x / sum(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) {rep(1, length(x))}

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

  } else if (type == "softmax") {
    # Derivative of softmax
    g_prime <- function(x) {
      s <- get_link("softmax")(x)
      return(s * (1 - s))
    }
  } else if (type == "relu") {
    # ReLU
    g_prime <- function(x) {as.numeric(x > 0)}

  } else (return(NULL))

  return(g_prime)

}
```

Loss Functions

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

  if (type == "squared_error") {
```

```r
    loss <- function(y_hat, y) {sum((y_hat - y)^2)}
  } else if (type == "absolute_error") {
    loss <- function(y_hat, y) {sum(abs(y_hat - y))}
  } else if (type == "binary_cross_entropy") {
    loss <- function(y_hat, y) {-(y * log(y_hat) + (1-y) * log(1 - y_hat))}
  } else if (type == "categorical_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 == "absolute_error") {
    loss_prime <- function(y_hat, y) {sum(sign(y_hat - y))}
  } else if (type == "binary_cross_entropy") {
    loss_prime <- function(y_hat, y) {-((y / y_hat) - ((1 - y) / (1 - y_hat)))}
  } else if (type == "categorical_cross_entropy") {
    loss_prime <- function(y_hat, y) {-sum(y / y_hat)}
  } else (return(NULL))
  return(loss_prime)
}
```

Misc Helpers

```r
## 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)))

}
```

```r
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))
}
```

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]])

}
```

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))
}
```

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)))
}
```

Summary Functions

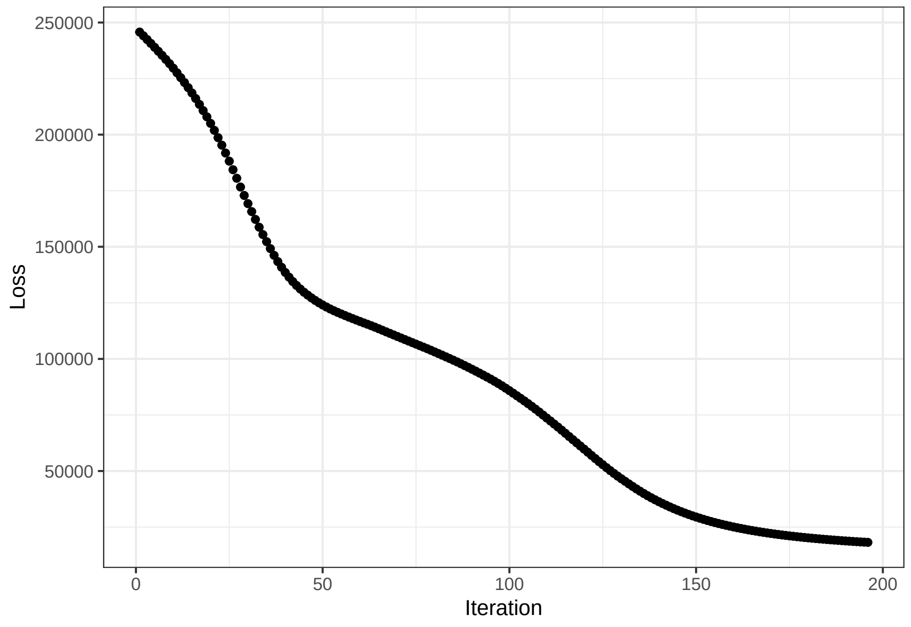
```
GD_plot <- function(GD_obj) {

    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))
}
```

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: 196
##   |
##   |  Initial Objective: 247431.3
##   |  Final Objective: 18190.1
##   |  Ratio: 0.0735
##
## ----------------------------------------
## Initial W:
##
## [[1]]
##            [,1]      [,2]       [,3]
## [1,] 0.02136059 0.9638598 0.32536659
## [2,] 0.78955914 0.7898393 0.09990061
## [3,] 0.86646344 0.9106100 0.84059247
##
## [[2]]
##           [,1]       [,2]      [,3]
## [1,] 0.5498648 0.02462769 0.6427036
##
## ----------------------------------------
## Final W:
##
## [[1]]
##              X1          X2        X3
## [1,] -0.0913354  0.36366438 0.9521866
## [2,]  0.8514598  0.69313868 0.1722769
## [3,]  0.3387587 -0.06274813 2.8887203
##
## [[2]]
##           [,1]       [,2]     [,3]
## [1,] 0.6902068 0.02506727 2.813997
##
## ----------------------------------------
## Initial b:
##
## [[1]]
##           [,1]
## [1,] 0.5031700
## [2,] 0.6528829
## [3,] 0.8822366
##
## [[2]]
##           [,1]
## [1,] 0.6794402
##
## ----------------------------------------
## Final b:
##
## [[1]]
##           [,1]
## [1,] 0.6478762
## [2,] 0.6693501
## [3,] 1.4671085
##
## [[2]]
##          [,1]
## [1,] 1.230873
```

```
GD_plot(GD_NN)
```



Other

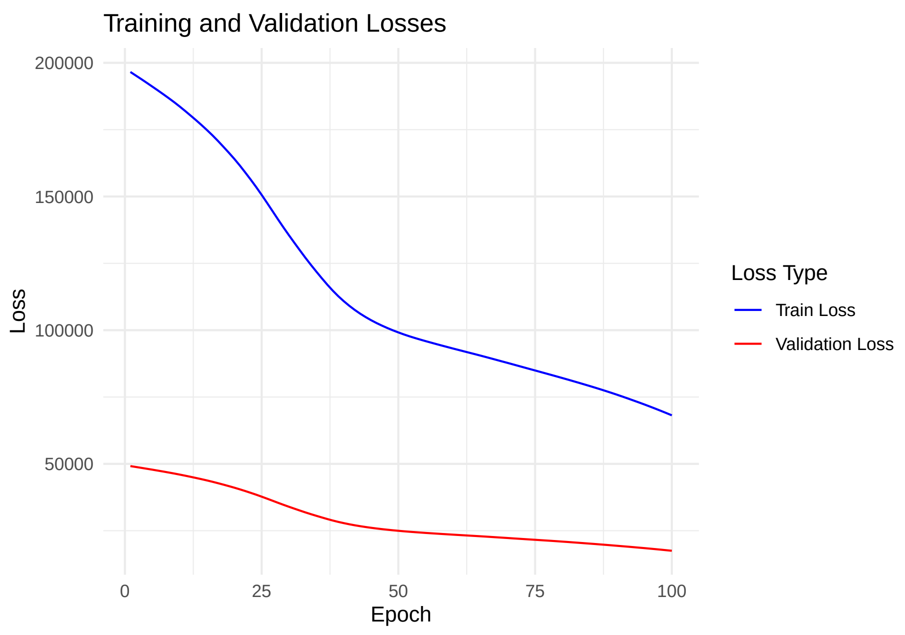
```
## get_layer_size function
get_layer_sizes <- function(NN_obj) {
  n_1 <- ncol(NN_obj$W[[2]])

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

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

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

Cross Validation

```
library(ggplot2)
```

```r
# Number of folds for cross-validation
k <- 5
max_iter <- 100  # Set the maximum number of iterations for gradient descent

# Initialize vectors to store training and validation losses
train_losses <- matrix(NA, nrow = max_iter, ncol = k)
valid_losses <- matrix(NA, nrow = max_iter, ncol = k)

# Perform 5-fold cross-validation
for (fold in 1:k) {

  ## Define fold indices for X and Y separately
  fold_indices_X <- ((fold - 1) * ncol(X) / k + 1):(fold * ncol(X) / k)
  fold_indices_Y <- ((fold - 1) * ncol(Y) / k + 1):(fold * ncol(Y) / k)

  ## Splitting the data into train and validation sets for X and Y
  X_valid_fold <- X[, fold_indices_X]
  Y_valid_fold <- Y[, fold_indices_Y, drop = FALSE]
  X_train_fold <- X[, -fold_indices_X]
  Y_train_fold <- Y[, -fold_indices_Y, drop = FALSE]

  # Perform gradient descent on the training set for this fold
  GD_NN <- GD_perform(X = X_train_fold,
                      Y = Y_train_fold,
                      init_NN_obj = init_NN,
                      rho = 0.001,
                      loss_function = "squared_error",
                      threshold = 100,
                      max_iter = 1000,
                      print_descent = FALSE)

  # Evaluate the model on the validation set for this fold
  objective_function <- function(y, y_hat) {
    return(get_loss_function(type = "squared_error")(y_hat, y))
    }
  for (epoch in 1:max_iter) {
    train_loss <- objective_function(y = Y_train_fold,
                                     y_hat = NN_output(X = X_train_fold,
                                                       NN_obj = GD_NN$intermediate_NN[
    valid_loss <- objective_function(y = Y_valid_fold,
                                     y_hat = NN_output(X = X_valid_fold,
                                                       NN_obj = GD_NN$intermediate_NN[
    train_losses[epoch, fold] <- train_loss
    valid_losses[epoch, fold] <- valid_loss
  }
}

# Plot training and validation losses
epoch <- 1:max_iter
train_loss_mean <- apply(train_losses, 1, mean)
valid_loss_mean <- apply(valid_losses, 1, mean)


df_loss <- data.frame(epoch = epoch,
                      train_loss = train_loss_mean,
                      valid_loss = valid_loss_mean)

ggplot(data = df_loss, aes(x = epoch)) +
  geom_line(aes(y = train_loss, color = "Train Loss")) +
  geom_line(aes(y = valid_loss, color = "Validation Loss")) +
  scale_color_manual(values = c("Train Loss" = "blue", "Validation Loss" = "red")) +
  labs(x = "Epoch", y = "Loss", color = "Loss Type") +
  ggtitle("Training and Validation Losses") +
  theme_minimal()
```



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