Neural Networks

Alfonso R. Reyes

2019-09-18

Contents

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Chapter 1

Neural Network from scratch: Examples

1.1 Load all the the neural network functions in R

```
source("./nn_from_scratch.R")
```

1.2 Train on the Iris dataset

```
head(iris)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
```

```
train_test_split <- train_test_from_df(df = iris, predict_col_index = 5, train_ratio = 0.7)
training_data <- train_test_split[[1]]
testing_data <- train_test_split[[2]]

in_n <- length(training_data[[1]][[1]])
out_n <- length(training_data[[1]][[-1]])

# [4, 40, 3]
trained_net <- neuralnetwork(
    c(in_n, 40, out_n),
    training_data=training_data,
    epochs=30,
    mini_batch_size=10,
    lr=0.5,
    C='ce',
    verbose=TRUE,
    validation_data=testing_data
```

```
)
#> Epoch:  1  complete[1] 0.4
#> Epoch:  2  complete[1] 0.6
#> Epoch:  3  complete[1] 0.422
#> Epoch:  4  complete[1] 0.4
#> Epoch:  5  complete[1] 0.644
#> Epoch:  6  complete[1] 0.8
#> Epoch:  7  complete[1] 0.6
#> Epoch:  8  complete[1] 0.889
#> Epoch:  9  complete[1] 0.756
#> Epoch:  10  complete[1] 0.6
#> Epoch:  11  complete[1] 0.956
#> Epoch:  12  complete[1] 0.6
#> Epoch:  13  complete[1] 0.6
#> Epoch:  14  complete[1] 0.6
#> Epoch:  15  complete[1] 0.644
#> Epoch:  16  complete[1] 0.822
#> Epoch:  17  complete[1] 0.844
#> Epoch:  18  complete[1] 0.956
#> Epoch:  19  complete[1] 0.644
#> Epoch:  20  complete[1] 0.711
#> Epoch:  21  complete[1] 0.689
#> Epoch:  22  complete[1] 0.933
#> Epoch:  23  complete[1] 0.889
#> Epoch:  24  complete[1] 0.956
#> Epoch:  25  complete[1] 0.933
#> Epoch:  26  complete[1] 0.844
#> Epoch:  27  complete[1] 0.978
#> Epoch:  28  complete[1] 0.756
#> Epoch:  29  complete[1] 0.889
#> Epoch:  30  complete[1] 0.756
#> Training complete in:  0.573Training complete
```

```
# Trained matricies:
biases <- trained_net[[1]]
weights <- trained_net[[-1]]

# Accuracy (train)
evaluate(training_data, biases, weights)  #0.971
#> [1] 0.771
#>
#>      1  2  3
#>   1 39  0  0
#>   2  0 32 24
#>   3  0  0 10
# Accuracy (test)
evaluate(testing_data, biases, weights)  #0.956
#> [1] 0.756
#>
#>      1  2  3
#>   1 11  0  0
#>   2  0 18 11
#>   3  0  0  5
```

1.3 Train on the MNIST digits dataset

```
library(grid)
```

1.3.1 Load the data

```
# Here we have splits for train-test already (may take a minute to download)
# Train
if (!file.exists(file.path(data_raw_dir, "mnist_training.rda"))) {
  cat("reading from web ...\n")
  mnist <- read.table('https://iliadl.blob.core.windows.net/nnet/mnist_train.csv',
                      sep=",", header = FALSE)
  } else {
    cat("reading from disk ...\n")
    load(file = file.path(data_raw_dir, "mnist_training.rda"))
}
#> reading from disk ...

mnist_training$V1 <- factor(mnist_training$V1)
training_data <- train_test_from_df(df = mnist_training, predict_col_index = 1,
                                    train_ratio = 1)[[1]]

# Test
if (!file.exists(file.path(data_raw_dir, "mnist_testing.rda"))) {
  cat("reading from web ...\n")
  mnist <- read.table('https://iliadl.blob.core.windows.net/nnet/mnist_test.csv',
                      sep=",", header = FALSE)
  } else {
    cat("reading from disk ...\n")
    load(file = file.path(data_raw_dir, "mnist_testing.rda"))
}
#> reading from disk ...

mnist_testing$V1 <- factor(mnist_testing$V1)
testing_data <- train_test_from_df(df = mnist_testing, predict_col_index = 1,
                                   train_ratio = 1)[[1]]
```

```
if (interactive()) {
  write.csv(mnist_train, file.path(data_raw_dir, "mnist_train.csv"))
  write.csv(mnist_test, file.path(data_raw_dir, "mnist_test.csv"))
}
```

1.3.2 What does the data exactly look like?

```
example_entry <- training_data[[1]]
example_x <- example_entry[[1]]
example_y <- example_entry[[2]]

# Y-vector looks like this:
print(example_y)
```

```
#>  [1] 0 0 0 1 0 0 0 0 0 0
# It corresponds to digit:
print(which.max(example_y)-1)
#> [1] 3

# X-vector has length
print(length(example_x))
#> [1] 784

# We can think of it as a 28x28 matrix where entries are a shade of gray
grid.raster(matrix(example_x, nrow=28, byrow=TRUE))
```

1.3.3 Training

Let's train a neural net with one 100-neuron hidden-layer to predict (given 784 vector of gray intensity) the digit (from 0 to 9)

```
# Input and output neurons
in_n <- length(training_data[[1]][[1]])
out_n <- length(training_data[[1]][[-1]])

# MNIST: 784, 100, 10 (one hidden-layer)
print("THIS WILL TAKE 20-30 MINUTES...")
#> [1] "THIS WILL TAKE 20-30 MINUTES..."
trained_net <- neuralnetwork(sizes = c(in_n, 100, out_n),
                             training_data = training_data,
                             epochs = 3,           # 30
                             mini_batch_size = 2,  # 10
                             lr = 3,
                             C = 'ce',
```

```
                            verbose=  TRUE,
                            validation_data = testing_data)
#> Epoch:  1  complete[1] 0.498
#> Epoch:  2  complete[1] 0.637
#> Epoch:  3  complete[1] 0.623
#> Training complete in:  7.94Training complete
```

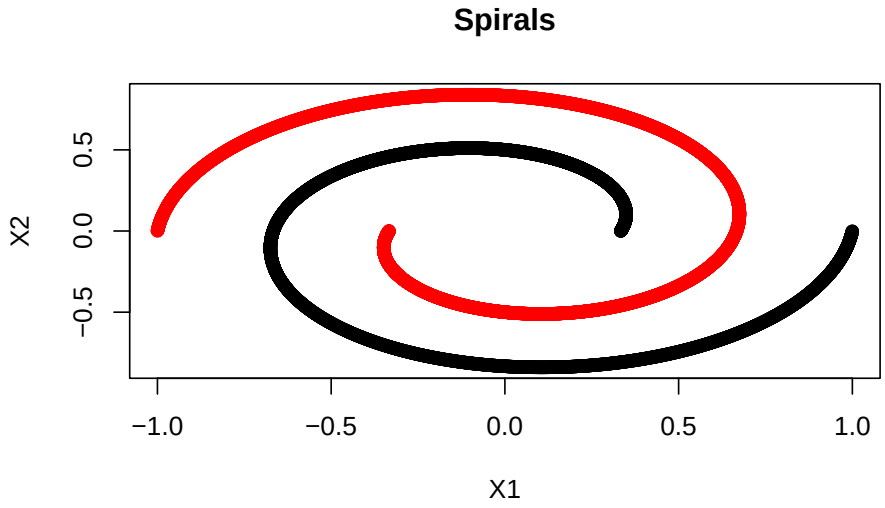
```
# Trained matricies:
biases <- trained_net[[1]]
weights <- trained_net[[-1]]

# CONFUSION TRAIN MATRIX
evaluate(training_data, biases, weights)  #0.98
#> [1] 0.625
#>
#>         1    2    3    4    5    6    7    8    9   10
#>   1  4641    0   22    6   19   86   51   75    9   18
#>   2     2 6237  128   22   11   35   49   81  135    7
#>   3    90  148 4280  136   80  365  746  190  946  133
#>   4   473  285  952 5604   31 2564  263  235 3764  204
#>   5    46   40  271  150 5629  406  905  412  317 4289
#>   6   549    6  166   80   43 1809  442   33  580   69
#>   7    79    3   55   27    7   67 3448    1   16    2
#>   8    39   18   76   90   16   25    8 5202   34  574
#>   9     4    0    6    9    0   26    2    2   36   11
#>   10    0    5    2    7    6   38    4   34   14  642
# CONFUSION TEST MATRIX
evaluate(testing_data, biases, weights)  #0.97
#> [1] 0.623
#>
#>         1    2    3    4    5    6    7    8    9   10
#>   1   764    0    7    2    2   12   15    4    0    2
#>   2     0 1062   25    0    0    2    2   21   11    4
#>   3    13   20  721   11   10   59  107   40  168   18
#>   4    79   46  205  951    5  428   41   37  632   29
#>   5     8    4   35   15  953   76  190   63   54  766
#>   6    95    1   17    8    5  284   62    2   88   10
#>   7    12    1    9    5    2   14  538    0    4    1
#>   8     9    0   11   15    3    5    2  854   12   76
#>   9     0    0    1    3    0    4    0    0    3    0
#>   10    0    1    1    0    2    8    1    7    2  103
```

```
# Test this out with one example
# Do some machine-learning
test_entry <- testing_data[[2]]
test_x <- test_entry[[1]]
test_y <- test_entry[[2]]

# Input
grid.raster(matrix(test_x, nrow=28, byrow=TRUE))

# Output
which.max(feedforward(test_x, biases, weights))-1
#> [1] 2
```

```
# Truth
cat("Truth: ", which.max(test_y)-1)
#> Truth:  2
```

1.4 Example: Visualizations of boundary lines

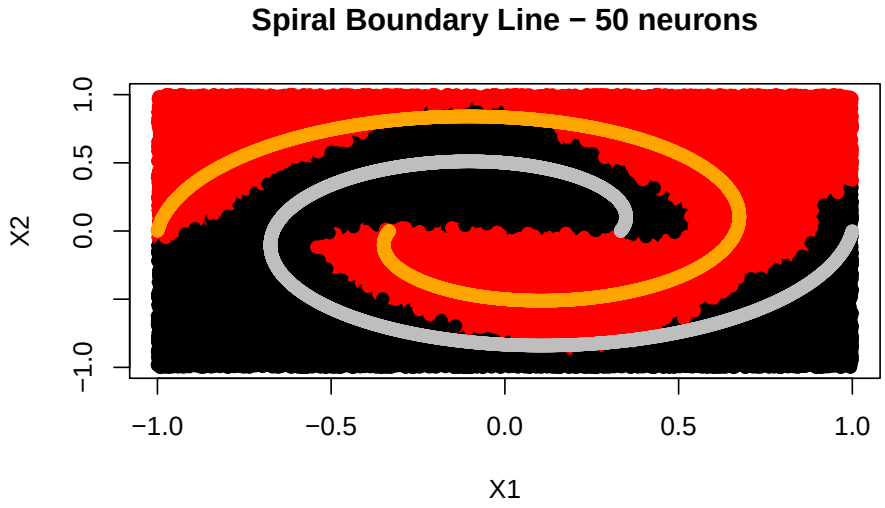
```
# Load some spiral data
library(mlbench)
data_df <- as.data.frame(mlbench.spirals(10000))

# Looks pretty cool
plot(x=data_df[,1], y=data_df[,2], cex = 1, col=data_df[,3],
     main = "Spirals",
     xlab = "X1", ylab = "X2")
```




```
# Let's use all of this data for training
training_data <- train_test_from_df(df = data_df, predict_col_index = 3,
                                    train_ratio = 1, scale_input = TRUE)[[1]]
in_n <- length(training_data[[1]][[1]])
out_n <- length(training_data[[1]][[-1]])
```

```
# Wrap a function that trains a network and plots the classification
plotBoundaryLinePerNeuron <- function(neurons, epochs)
{
  trained_net <- neuralnetwork(c(in_n, neurons, out_n), training_data,
                               epochs, 500, 3, 'ce')
  b <- trained_net[[1]]
  w <- trained_net[[-1]]
  # Evaluate
  confusion <- evaluate(training_data, b, w)
  # Generate new-data
  nd <- as.data.frame(matrix(runif(100000, min=-1, max=1), ncol=2))
  # Get predictions
  nd$res <- as.factor(unlist(get_predictions(as.list(as.data.frame(t(nd))), b, w)))
  # Plot predictions
  plot(x=nd[,1], y=nd[,2], pch=19, col=nd[,3],
       main = paste0("Spiral Boundary Line - ", neurons, " neurons"),
       xlab = "X1", ylab = "X2")
  # Original points
  points(x=data_df[,1], y=data_df[,2], pch=19, col=ifelse(data_df[,3]==2,
                                                          "orange", "grey"))
}
```

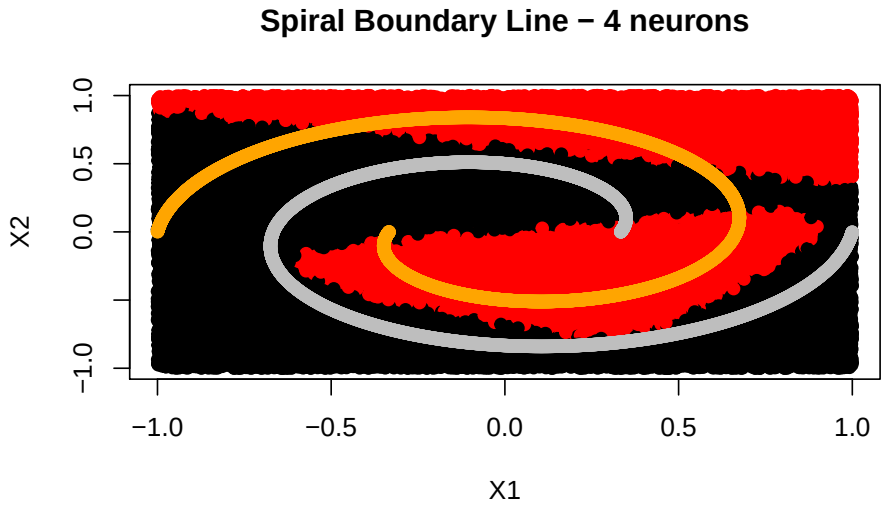
```
 plotBoundaryLinePerNeuron(50, 80)
#> Training complete[1] 1
```




```
# plotBoundaryLinePerNeuron(10, 80)
```

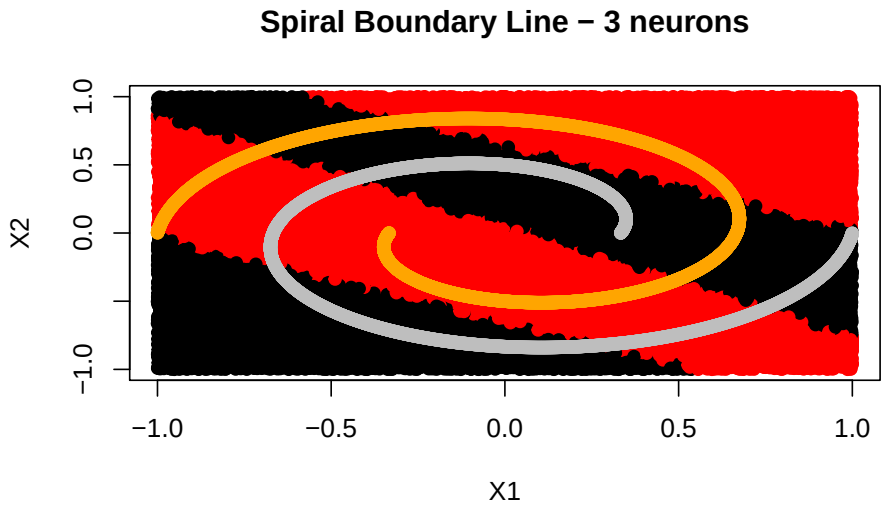
```
# plotBoundaryLinePerNeuron(5, 100)
```

```
plotBoundaryLinePerNeuron(4, 100)
#> Training complete[1] 0.877
```



It doesn't seem possible to draw fewer than 4 curves that split the classes, hence below 4 neurons we should start to see performance degrade - because the network doesn't enough params to fit the data. This can be a useful to exercise when trying to figure out how many neurons ones needs over-fitting

```
plotBoundaryLinePerNeuron(3, 120)
#> Training complete[1] 0.786
```




```
# plotBoundaryLinePerNeuron(2, 200)
```

```
# plotBoundaryLinePerNeuron(1, 120)
```

```
# plotBoundaryLinePerNeuron(c(40,40,40), 200)
```

1.5 What is a neural network?

A neural network can be thought of as a series of logistic regressions stacked on top of each other. This means we could say that a logistic regression is a neural-network (with sigmoid activations) with no hidden-layer.

This hidden-layer lets a neural-network generate non-linearities and leads to the Universal approximation theorem, which states that a network with just one hidden layer can approximate any linear or non-linear function. The number of hidden-layers can go into the hundreds.

It can be useful to think of a neural-network as a combination of two things: 1) many logistic regressions stacked on top of each other that are 'feature-generators' and 2) one read-out-layer which is just a softmax regression. The recent successes in deep-learning can arguable be attributed to the 'feature-generators'. For example; previously with computer vision, we had to painfully state that we wanted to find triangles, circles, colours, and in what combination (similar to how economists decide which interaction-terms they need in a linear regression). Now, the hidden-layers are basically an optimisation to decide which features (which 'interaction-terms') to extract. A lot of deep-learning (transfer learning) is actually done by generating features using a trained-model with the head (read-out layer) cut-off, and then training a logistic regression (or boosted decision-trees) using those features as inputs.

The hidden-layer also means that our loss function is not convex in parameters and we can't roll down a smooth-hill to get to the bottom. Instead of using Gradient Descent (which we did for the case of a logistic-regression) we will use Stochastic Gradient Descent (SGD), which basically shuffles the observations (random/stochastic) and updates the gradient after each mini-batch (generally much less than total number of observations) has been propagated through the network. There are many alternatives to SGD that Sebastian Ruder does a great job of summarising here. I think this is a fascinating topic to go through, but outside the scope of this blog-post. Briefly, however, the vast majority of the optimisation methods are first-order (including SGD, Adam, RMSprop, and Adagrad) because calculating the second-order is too computionally difficult. However, some of these first-order methods have a fixed learning-rate (SGD) and some have an adaptive learning-rate (Adam), which means that the 'amount' we update our weights by becomes a function of the loss - we may make big jumps in the beginning but then take smaller steps as we get closer to the target.

It should be clear, however that minimising the loss on training data is not the main goal - in theory we want to minimise the loss on 'unseen'/test data; hence all the opimisation methods proxy for that under the assumption that a low lost on training data will generalise to 'new' data from the same distribution. This means we may prefer a neural-network with a higher training-loss; because it has a lower validation-loss (on data it hasn't been trained on) - we would typically say that the network has 'overfit' in this case. There have been some recent papers that claim that adaptive optimisation methods do not generalise as well as SGD because they find very sharp minima points.

Previously we only had to back-propagate the gradient one layer, now we also have to back-propagate it through all the hidden-layers. Explaining the back-propagation algorithm is beyond the scope of this post, however it is crucial to understand. Many good resources exist online to help.

We can now create a neural-network from scratch in R using four functions.

First, we initialise our weights:

```
neuralnetwork <- function(sizes, training_data, epochs,
  mini_batch_size, lr, C, verbose=FALSE,
  validation_data=training_data)
```

Since we now have a complex combination of parameters we can't just initialise them to be 1 or 0, like before - the network may get stuck. To help, we use the gaussian distribution (however, just like with the opimisation, there are many other methods):

```
    biases <- lapply(seq_along(listb), function(idx){
    r <- listb[[idx]]
    matrix(rnorm(n=r), nrow=r, ncol=1)
    })

    weights <- lapply(seq_along(listb), function(idx){
    c <- listw[[idx]]
    r <- listb[[idx]]
    matrix(rnorm(n=r*c), nrow=r, ncol=c)
    })
```

Second, we use stochastic gradient descent as our optimisation method:

```
SGD <- function(training_data, epochs, mini_batch_size, lr, C, sizes,
                num_layers, biases, weights,
                 verbose=FALSE, validation_data)
 {
   # Every epoch
   for (j in 1:epochs){
     # Stochastic mini-batch (shuffle data)
     training_data <- sample(training_data)
     # Partition set into mini-batches
     mini_batches <- split(training_data,
                           ceiling(seq_along(training_data)/mini_batch_size))
     # Feed forward (and back) all mini-batches
     for (k in 1:length(mini_batches)) {
       # Update biases and weights
       res <- update_mini_batch(mini_batches[[k]], lr, C, sizes, num_layers, biases, weights)
       biases <- res[[1]]
       weights <- res[[-1]]
     }
   }
   # Return trained biases and weights
```

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

Third, as part of the SGD method, we update the weights after each mini-batch has been forward and backwards-propagated:

```
update_mini_batch <- function(mini_batch, lr, C, sizes, num_layers, biases, weights)
{
  nmb <- length(mini_batch)
  listw <- sizes[1:length(sizes)-1]
  listb <-  sizes[-1]

  # Initialise updates with zero vectors (for EACH mini-batch)
  nabla_b <- lapply(seq_along(listb), function(idx){
    r <- listb[[idx]]
    matrix(0, nrow=r, ncol=1)
  })
  nabla_w <- lapply(seq_along(listb), function(idx){
    c <- listw[[idx]]
    r <- listb[[idx]]
    matrix(0, nrow=r, ncol=c)
  })

  # Go through mini_batch
  for (i in 1:nmb){
    x <- mini_batch[[i]][[1]]
    y <- mini_batch[[i]][[-1]]
    # Back propogation will return delta
    # Backprop for each observation in mini-batch
    delta_nablas <- backprop(x, y, C, sizes, num_layers, biases, weights)
    delta_nabla_b <- delta_nablas[[1]]
    delta_nabla_w <- delta_nablas[[-1]]
    # Add on deltas to nabla
    nabla_b <- lapply(seq_along(biases),function(j)
      unlist(nabla_b[[j]])+unlist(delta_nabla_b[[j]]))
    nabla_w <- lapply(seq_along(weights),function(j)
      unlist(nabla_w[[j]])+unlist(delta_nabla_w[[j]]))
  }
  # After mini-batch has finished update biases and weights:
  # i.e. weights = weights - (learning-rate/numbr in batch)*nabla_weights
  # Opposite direction of gradient
  weights <- lapply(seq_along(weights), function(j)
    unlist(weights[[j]])-(lr/nmb)*unlist(nabla_w[[j]]))
  biases <- lapply(seq_along(biases), function(j)
    unlist(biases[[j]])-(lr/nmb)*unlist(nabla_b[[j]]))
  # Return
  list(biases, weights)
}
```

Fourth, the algorithm we use to calculate the deltas is the back-propagation algorithm.

In this example we use the cross-entropy loss function, which produces the following gradient:

```
cost_delta <- function(method, z, a, y) {
  if (method=='ce'){return (a-y)}
```

```
}
```

Also, to be consistent with our logistic regression example we use the sigmoid activation for the hidden layers and for the read-out layer:

```
# Calculate activation function
sigmoid <- function(z){1.0/(1.0+exp(-z))}
# Partial derivative of activation function
sigmoid_prime <- function(z){sigmoid(z)*(1-sigmoid(z))}
```

As mentioned previously, usually the softmax activation is used for the read-out layer. For the hidden layers, ReLU is more common, which is just the max function (negative weights get flattened to 0). The activation function for the hidden layers can be imagined as a race to carry a baton/flame (gradient) without it dying. The sigmoid function flattens out at 0 and at 1, resulting in a flat gradient which is equivalent to the flame dying out (we have lost our signal). The ReLU function helps preserve this gradient.

The back-propagation function is defined as:

```
backprop <- function(x, y, C, sizes, num_layers, biases, weights)
```

Check out the notebook for the full code — however the principle remains the same: we have a forward-pass where we generate our prediction by propagating the weights through all the layers of the network. We then plug this into the cost gradient and update the weights through all of our layers.

This concludes the creation of a neural network (with as many hidden layers as you desire). It can be a good exercise to replace the hidden-layer activation with ReLU and read-out to be softmax, and also add L1 and L2 regularization. Running this on the iris dataset in the notebook (which contains 4 explanatory variables with 3 possible outomes), with just one hidden-layer containing 40 neurons we get an accuracy of 96% after 30 rounds/epochs of training.

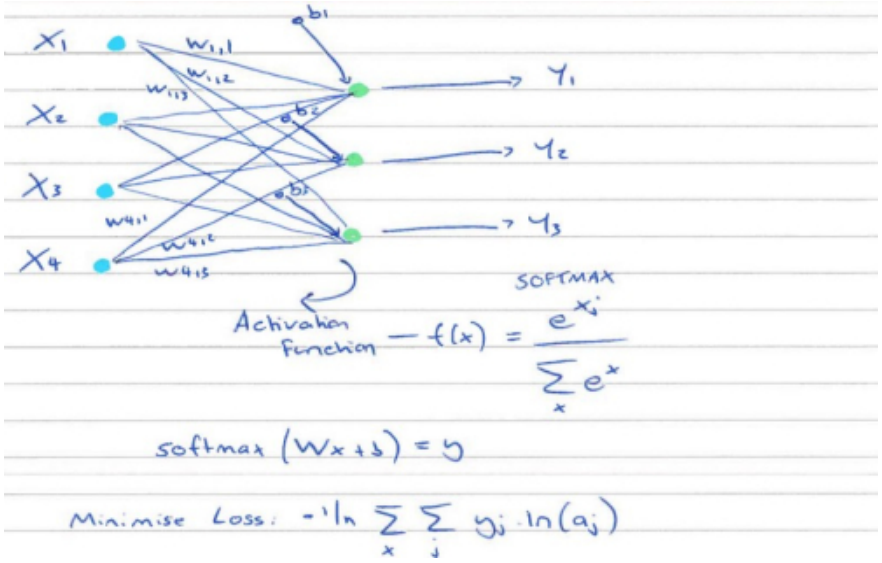
The notebook also runs a 100-neuron handwriting-recognition example to predict the digit corresponding to a 28x28 pixel image.

Chapter 2

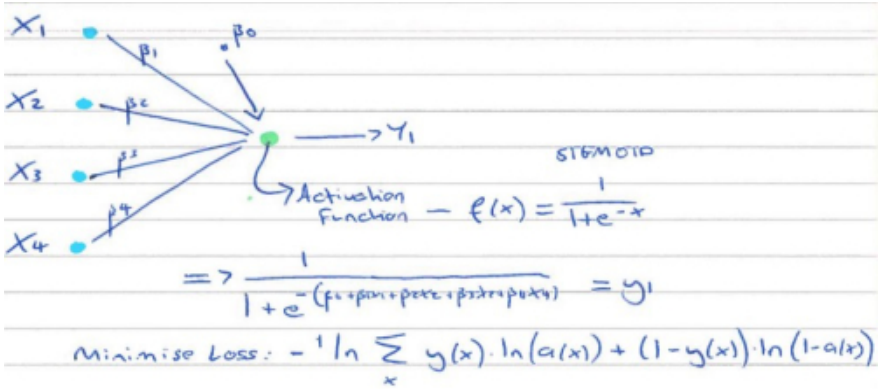
Logistic Regression

https://github.com/ilkarman/DemoNeuralNet/blob/master/02_LogisticRegression.ipynb

```
knitr::include_graphics(file.path(assets_dir, "softmax_regression.jpg"))
```




```
knitr::include_graphics(file.path(assets_dir, "logistic_regression.jpg"))
```



2.1 Generate the data

```
# Reproduce results
set.seed(1234567)

# Two possible outcomes -> binomial
data_df <- as.data.frame(iris)
idx <- data_df$Species %in% c("virginica", "versicolor")
data_df <- data_df[idx,]
y <- ifelse(data_df$Species=="virginica", 1, 0)

# For faster convergence let's rescale X
# So that we can plot this consider only 2 variables
X <- data_df[c(1,3)]
X <- as.matrix(X/max(X))

# Resulting data-set
head(X)
#>    Sepal.Length Petal.Length
#> 51        0.886        0.595
#> 52        0.810        0.570
#> 53        0.873        0.620
#> 54        0.696        0.506
#> 55        0.823        0.582
#> 56        0.722        0.570
head(y)
#> [1] 0 0 0 0 0 0
```

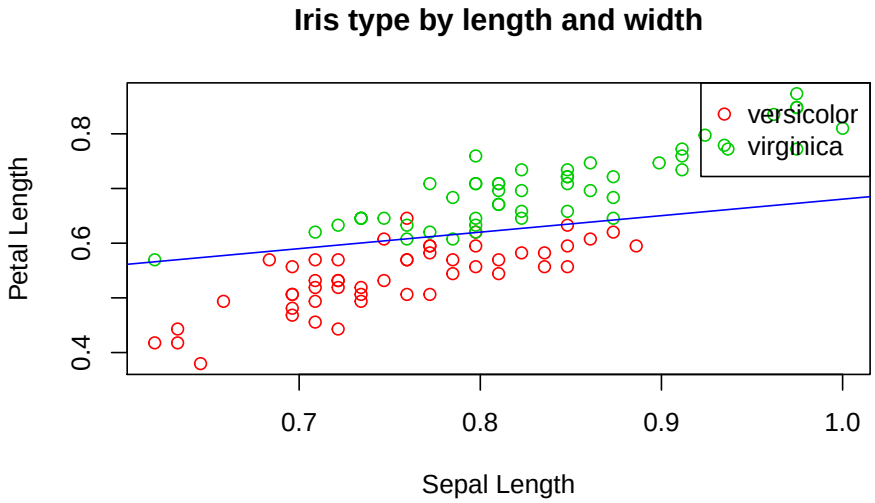
2.2 1. Fit a binomial logit in R

A logistic regression is a linear regression that outputs a number bounded between 0 and 1. This means it is useful for classification problems, where we want to predict the probability of something happening. A binomial logistic regression is used when there are just two-classes, to extend beyound two-classes we would typically use a multi-nomial logistic regression (softmax).

Consider the iris-dataset where we try to predict whether a flower is “virginica” or “versicolor” by only looking at petal-length and sepal-length. We fit a linear line to ‘best’ split the categories:

```
# Fit model
model <- glm(y ~ X, family=binomial(link='logit'))

# Params
print(coef(model))
#>   (Intercept) XSepal.Length XPetal.Length
#>         -39.8         -31.7         105.2
# Coefficients:
# (Intercept) XSepal.Length XPetal.Length
# -39.83851     -31.73243     105.16992
#summary(model)

# Visualise the decision boundary
intcp <- coef(model)[1]/-(coef(model)[3])
```

```
slope <- coef(model)[2]/-(coef(model)[3])

# Our points
plot(x=X[,1], y=X[,2], cex = 1, col=data_df$Species,
     main = "Iris type by length and width",
     xlab = "Sepal Length", ylab = "Petal Length")
legend(x='topright', legend=unique(data_df$Species),col=unique(data_df$Species), pch=1)
# Decision boundary
abline(intcp , slope, col='blue')
```



The above line has an intercept of -39.84, and coefficient of -31.73 for sepal-length and 105.17 for petal-length. These estimates are obtained by maximising the likelihood.

Because the log function is monotone, maximizing the likelihood is the same as maximizing the log-likelihood (or minimising the negative of the log-likelihood)

$$l_x(\theta) = \log L_x(\theta)$$

For many reasons it is more convenient to use log likelihood rather than likelihood:

$$\log L_x = \sum_{i=1}^{N} y_i \beta^T x_i - \log(1 + e^{\beta^T x_i})$$

```
log_likelihood <- function(X_mat, y, beta_hat)
{
  scores <- X_mat %*% beta_hat
  ll <- (y * scores) - log(1+exp(scores))
  sum(ll)
}
```

The log-likelihood in this example is -11.92.

```
logLik(model)  # Log-likelihood
#> 'log Lik.' -11.9 (df=3)

# Calculate log-likelihood ourself
log_likelihood <- function(X_mat, y, beta_hat)
{
```

```
  scores <- X_mat %*% beta_hat
  # Need to broadcast (y %*% scores)
  ll <- (y * scores) - log(1+exp(scores))
  sum(ll)
}

log_likelihood(cbind(1, X), y, coef(model))  # Match at -11.925
#> [1] -11.9
```

2.3 2. Fit a binomial logit ourselves (GD)

Typically BFGS or other numerical optimisation procedures are used to minimise the cost/max log-likelihood instead of GD, because the parameter space is pretty smaller (compared to neural-networks).

The logistic loss is sometimes called the cross-entropy loss. Let us first examine what would happen if we simply tried to calculate the least-squares loss, like before (for a regression rather than a classification):

$$C = \frac{1}{2n} \sum_x \|(y(x) - a(x)\|^2$$

This loss will not be convex (in parameters) because $a(x)$ (the activation/link function) that transforms our score into a probability is defined as: $\sigma(z) = \frac{1}{1+e^{-z}}$ and thus $a(x) = \frac{1}{1+e^{-\beta^T x_i}}$

We can construct a convex-loss function. This is binary classification and so we can define the loss for both of the classes. When $y = 1$ we want the loss to be large when $\sigma(z)$ is close to zero and for it to be small when it is close to 1. Similarly, when $y = 0$ we want the loss to be large when $\sigma(z)$ is close to one and for it be small when it is close to 0. The following loss function satisifies those conditions:

$$C = -\frac{1}{n} \sum_x y(x) \ln(a(x)) + (1 - y(x)) \ln(1 - a(x))$$

For example, when $y(x) = 1$ our loss-function becomes $C = -\frac{1}{n} \sum_x \ln(a(x))$, which is equal to 0 when $a(x) = 1$, otherwise it becomes very high.

Taking the derivative of this loss-function w.r.t to the parameters, it can be shown:

$$\frac{dC}{d\beta_i} = \frac{1}{n} \sum_x x_i(a(x) - y)$$

Note that the cross-entropy loss is more generally defined as:

$$C = -\frac{1}{n} \sum_x \sum_j y_j \ln(a_j)$$

For binary classifications where $j = 2$, under the condition that the categories are mutually-exclusive $\sum_j a_j = 1$ and that y is one-hot so that $y1 + y2 = 1$, we can re-write it as:

$$C = -\frac{1}{n} \sum_x y_1 \ln(a_1) + (1 - y_1) \ln(1 - a_1)$$

Which is the same equation we first started with.

The process for using GD for a logistic regression is similar to that of a simple linear-regression. Since our loss is convex we can use either gradient-descent or stochastic-gradient descent; for now we will stick with the former.

```
# Calculate activation function (sigmoid for logit)
sigmoid <- function(z){1.0/(1.0+exp(-z))}

logistic_reg <- function(X, y, epochs, lr)
{
  X_mat <- cbind(1, X)
  beta_hat <- matrix(1, nrow=ncol(X_mat))
  for (j in 1:epochs)
  {
    residual <- sigmoid(X_mat %*% beta_hat) - y
    # Update weights with gradient descent
    delta <- t(X_mat) %*% as.matrix(residual, ncol=nrow(X_mat)) *  (1/nrow(X_mat))
    beta_hat <- beta_hat - (lr*delta)
  }
  # Print log-likliehood
  print(log_likelihood(X_mat, y, beta_hat))
  # Return
  beta_hat
}
```

The only major difference is that we apply a sigmoid function to our prediction - to turn it into a probability. Below we can see why: the output is bounded between 0 and 1.

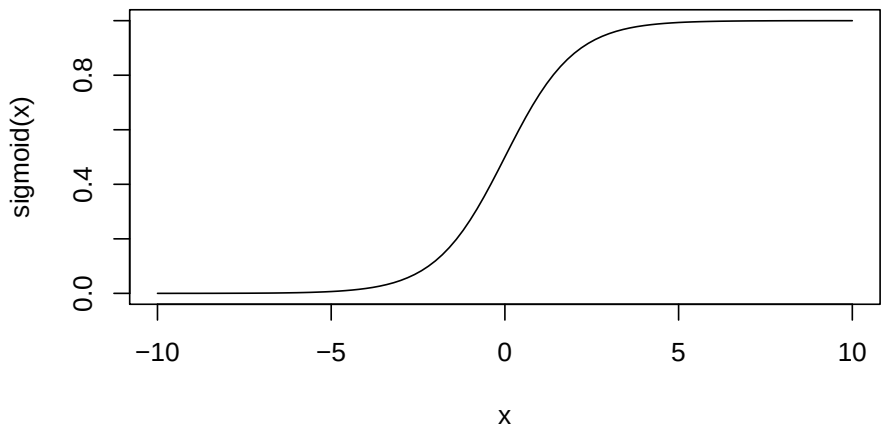
The shape of the sigmoid curve also means that we can increase the speed of convergence by scaling the variables to be closer to 0 - where the gradient is high. Imagine our inputs have a value of 100 - this can create a very high error, however the gradient is nearly flat and thus the update to the coefficients will be tiny.

We run the below to optimise our logistic regression using GD:

```
beta_hat <- logistic_reg(X, y, 300000, 5)
```

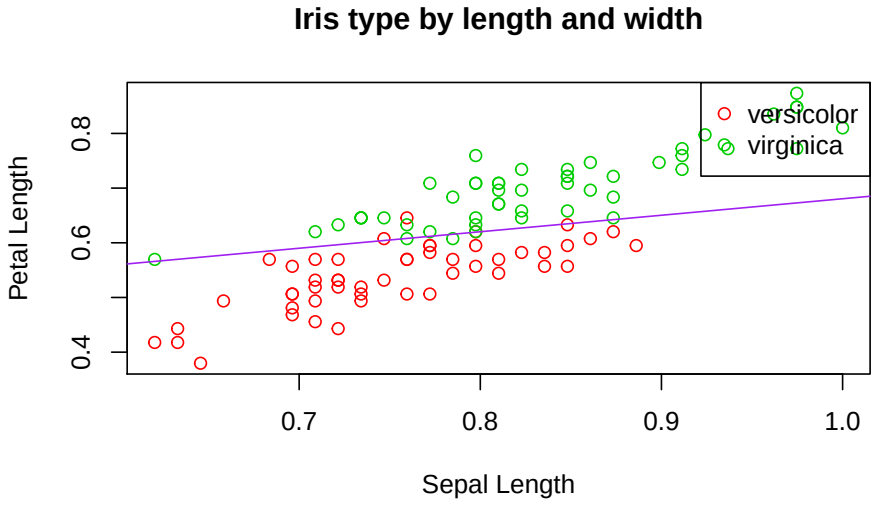
We match the original results with the coefficients: -38.84, -31.73, 105.17

```
# Why did scaling before help with convergence?
# Vanishing gradient
curve(sigmoid, -10, 10)
```




```
# Takes a while to converge with GD!
beta_hat <- logistic_reg(X, y, 300000, 5)
```

```
#> [1] -11.9
print(beta_hat)
#>                [,1]
#>              -39.8
#> Sepal.Length -31.7
#> Petal.Length 105.2

# Intercept    -39.83848
# Sepal.Length -31.73240
# Petal.Length 105.16983

# Visualise the decision boundary
plot(x=X[,1], y=X[,2], cex = 1, col=data_df$Species,
     main = "Iris type by length and width",
     xlab = "Sepal Length", ylab = "Petal Length")
legend(x='topright', legend=unique(data_df$Species),col=unique(data_df$Species), pch=1)

# Visualise the decision boundary
intcp <- beta_hat[1]/-(beta_hat[3])
slope <- beta_hat[2]/-(beta_hat[3])

abline(intcp , slope, col='purple')
```



2.4 What is logistic regression?

A logistic regression is a linear regression for binary classification problems. The two main differences to a standard linear regression are:

We use an 'activation'/link function called the logistic-sigmoid to squash the output to a probability bounded by 0 and 1 Instead of minimising the quadratic loss we minimise the negative log-likelihood of the Bernoulli distribution Everything else remains the same.

We can calcuate our activation function like so:

```
sigmoid <- function(z){1.0/(1.0+exp(-z))}
```

We can create our log-likelihood function in R:

```
log_likelihood <- function(X_mat, y, beta_hat)
{
  scores <- X_mat %*% beta_hat
  ll <- (y * scores) - log(1+exp(scores))
  sum(ll)
}
```

This loss function (the logistic loss or the log-loss) is also called the cross-entropy loss. The cross-entropy loss is basically a measure of 'surprise' and will be the foundation for all the following models, so it is worth examining a bit more.

If we simply constructed the least-squares loss like before, because we now have a non-linear activation function (the sigmoid), the loss will no longer be convex which will make optimisation hard.

$$C = \frac{1}{2n}\sum_x((y(x) - a(x))^2$$

We could construct our own loss function for the two classes. When $y = 1$, we want our loss function to be very high if our prediction is close to 0, and very low when it is close to 1. When $y = 0$, we want our loss function to be very high if our prediction is close to 1, and very low when it is close to 0. This leads us to the following loss function:

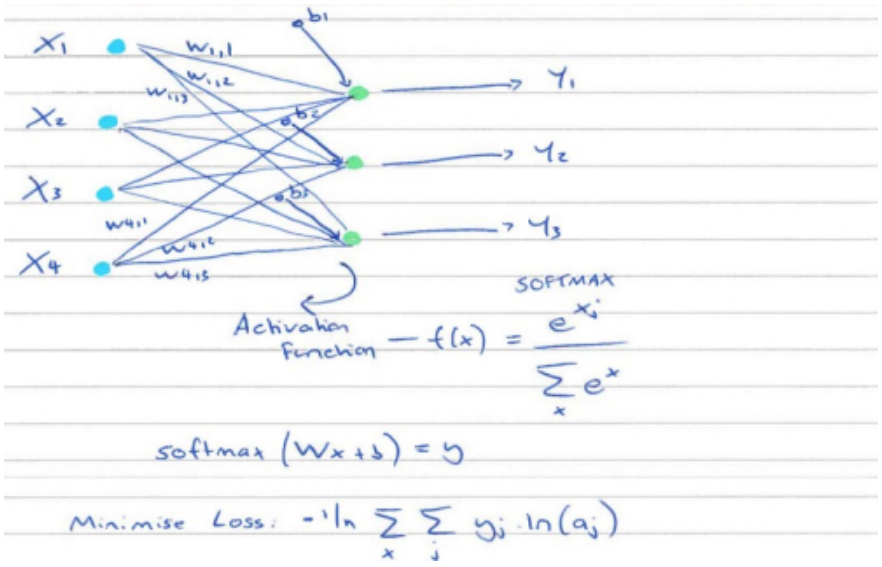
$$C = -\frac{1}{n}\sum_x y(x)\ln(a(x)) + (1 - y(x))\ln(1 - a(x))$$

The delta for this loss function is pretty much the same as the one we had earlier for a linear-regression. The only difference is that we apply our sigmoid function to the prediction. This means that the GD function for a logistic regression will also look very similar:

```
logistic_reg <- function(X, y, epochs, lr)
{
  X_mat <- cbind(1, X)
  beta_hat <- matrix(1, nrow=ncol(X_mat))
  for (j in 1:epochs)
  {
    # For a linear regression this was:
    # 1*(X_mat %*% beta_hat) - y
    residual <- sigmoid(X_mat %*% beta_hat) - y
    # Update weights with gradient descent
    delta <- t(X_mat) %*% as.matrix(residual, ncol=nrow(X_mat))*(1/nrow(X_mat))
    beta_hat <- beta_hat - (lr*delta)
  }
  # Print log-likliehood
  print(log_likelihood(X_mat, y, beta_hat))
  # Return
  beta_hat
}
```

2.5 Softmax Regression

```
knitr::include_graphics(file.path(assets_dir, "softmax_regression.jpg"))
```



A generalisation of the logistic regression is the multinomial logistic regression (also called 'softmax'), which is used when there are more than two classes to predict. I haven't created this example in R, because the neural-network in the next step can reduce to something similar, however for completeness I wanted to highlight the main differences if you wanted to create it.

First, instead of using the sigmoid function to squash our (one) value between θ and 1:

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

We use the softmax function to squash the sum of our n values (for n classes) to 1:

$$\phi(z) = \frac{e^z_j}{\sum_k e^z_k}$$

This means the value supplied for each class can be interpreted as the probability of that class, given the evidence. This also means that when we see the target class and increase the weights to increase the probability of observing it, the probability of the other classes will fall. The implicit assumption is that our classes are mutually exclusive.

Second, we use a more general version of the cross-entropy loss function:

$$C = -\frac{1}{n} \sum_x \sum_j y_j \ln(a_j)$$

To see why, remember that for binary classifications (previous example) we had two classes: $j = 2$, under the condition that the categories are mutually-exclusive $\sum_j a_j = 1$ and that y is one-hot so that $y_1 + y_2 = 1$, we can re-write the general formula as:

$$C = -\frac{1}{n} \sum_x y_1 \ln(a_1) + (1 - y_1) \ln(1 - a_1)$$

Which is the same equation we first started with. However, now we relax the constraint that $j = 2$. It can be shown that the cross-entropy loss here has the same gradient as for the case of the binary/two-class cross-entropy on logistic outputs.

$$\frac{\partial C}{\partial \beta_i} = \frac{1}{n} \sum x_i (a(x) - y)$$

However, although the gradient has the same formula it will be different because the activation here takes on a different value (softmax instead of logistic-sigmoid).

In most deep-learning frameworks you have the choice of 'binary-crossentropy' or 'categorical-crossentropy' loss. Depending on whether your last layer contains sigmoid or softmax activation you would want to choose binary or categorical cross-entropy (respectively). The training of the network should not be affected, since the gradient is the same, however the reported loss (for evaluation) would be wrong if these are mixed up.

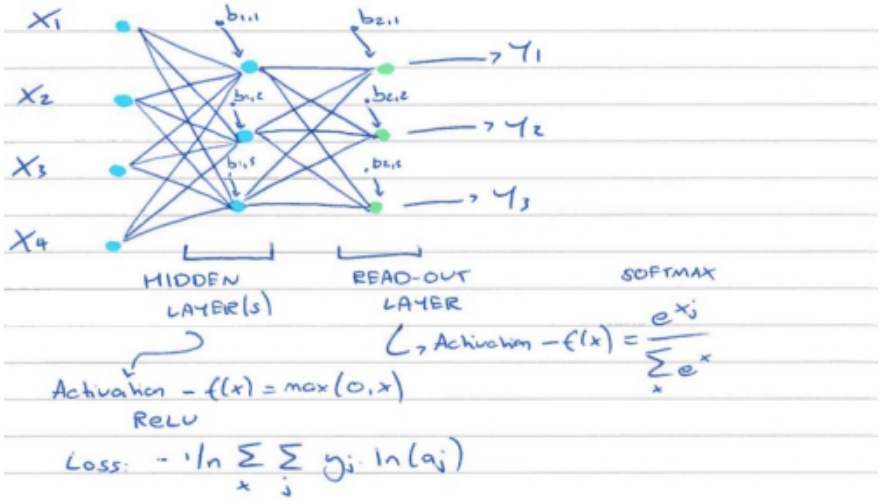
The motivation to go through softmax is that most neural-networks will use a softmax layer as the final/'read-out' layer, with a multinomial/categorical cross-entropy loss instead of using sigmoids with a binary cross-entropy loss — when the categories are mutually exclusive. Although multiple sigmoids for multiple classes can also be used (and will be used in the next example), this is generally only used for the case of non-mutually-exclusive labels (i.e. we can have multiple labels). With a softmax output, since the sum of the outputs is constrained to equal 1, we have the advantage of interpreting the outputs as class probabilities.

Chapter 3

Neural Networks: What is it?

Source: https://github.com/ilkarman/DemoNeuralNet/blob/master/03_NeuralNet.ipynb

```
print(assets_dir)
#> [1] "/home/datascience/repos/machine-learning-rsuite/import/assets"
```




```
# Reproduce results
set.seed(1234567)
```

3.1 Neural Net in R (5 short functions to care about)

In the previous scenarios we used the mean-squared-error to represent our cost-function:

$$C = \frac{1}{2n} \sum_x \|(y(x) - a(x)\|^2$$

For classification problems with neural-networks we will now use the cross-entropy cost:

$$C = -\frac{1}{n} \sum_x y(x) \ln(a(x)) + (1 - y(x)) \ln(1 - a(x))$$

Where $a = \sigma(\sum_i w_i x_i + b) = \sigma(z)$

We can show that:

$$\frac{dC}{dw_i} = \frac{1}{N} \sum_j x_j (\sigma(z) - y)$$

```
cost_delta <- function(method, z, a, y) {if (method=='ce'){return (a-y)}}
```

This means that the bigger the error, the faster our weight will learn.

Our main neural-network functions are:

```
neuralnetwork
SGD
update_mini_batch
backprop
```

The `neuralnetwork` function's main job is to initialise the weight and bias matricies given a list of sizes. For example, if we have 10 variables to predict 4 possible classes and we want a hidden-layer with 20 neurons we would pass: c(10,20,4) to this function. It passes these matricies to the SGD function to commence training.

3.2 The SGD function

The SGD function splits the training-data into random mini-batches and sends them off to the update_mini_batch function, which calculates the deltas for a batch (using backprop) and then updates the weights and bias matricies - so these are held constant within a batch:

```
# After mini-batch has finished update biases and weights:
# Opposite direction of gradient
weights <- lapply(seq_along(weights), function(j)
  unlist(weights[[j]])-(lr/nmb)*unlist(nabla_w[[j]]))
biases <- lapply(seq_along(biases), function(j)
  unlist(biases[[j]])-(lr/nmb)*unlist(nabla_b[[j]]))
```

In other words: $weights = weights - (learningrate/numberinbatch) * nablaweights$

The backprop function applies the backpropogation algorithm to calculate the partial derivatives (to update mini-batch).

The forward step, goes through the network layer-by-layer and calculates the output of the activation function to calculate the delta (cost gradient given the prediction). For example, the activations in layer l are:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

The **backward step** propogates the partial derivative (deltas) across all the neurons so that they get a share proportional to their contribution to the output.

First, we initialise neural network bias and weights matrices

```
neuralnetwork <- function(sizes, training_data, epochs, mini_batch_size, lr, C,
                          verbose=FALSE, validation_data=training_data)
{
  num_layers <- length(sizes)
  listw <- sizes[1:length(sizes)-1] # Skip last (weights from 1st to 2nd-to-last)
  listb <-  sizes[-1]  # Skip first element (biases from 2nd to last)
```

```
  # Initialise with gaussian distribution for biases and weights
  biases <- lapply(seq_along(listb), function(idx){
    r <- listb[[idx]]
    matrix(rnorm(n=r), nrow=r, ncol=1)
  })

  weights <- lapply(seq_along(listb), function(idx){
    c <- listw[[idx]]
    r <- listb[[idx]]
    matrix(rnorm(n=r*c), nrow=r, ncol=c)
  })

  SGD(training_data, epochs, mini_batch_size, lr, C,
      sizes, num_layers, biases, weights, verbose, validation_data)
}
```

3.3 The cost function

Return the derivative of the cost function (quadratic or cross-entropy).

Quadratic cost:

$$C = \frac{1}{2n} \sum_x \|(y(x) - a(x)\|^2$$

Cross-entropy cost:

$$C = -\frac{1}{n} \sum_x y(x) \ln(a(x)) + (1 - y(x)) \ln(1 - a(x))$$

```
cost_delta <- function(method, z, a, y) {if (method=='ce'){return (a-y)}}
```

3.4 Perform stochastic-gradient descent to minimise cost function

```
SGD <- function(training_data, epochs, mini_batch_size, lr, C, sizes, num_layers, biases, weights,
                verbose=FALSE, validation_data)
{
  start.time <- Sys.time()
  # Every epoch
  for (j in 1:epochs){
    # Stochastic mini-batch (shuffle data)
    training_data <- sample(training_data)
    # Partition set into mini-batches
    mini_batches <- split(training_data,
                          ceiling(seq_along(training_data)/mini_batch_size))
    # Feed forward (and back) all mini-batches
    for (k in 1:length(mini_batches)) {
      # Update biases and weights
      res <- update_mini_batch(mini_batches[[k]], lr, C, sizes, num_layers, biases, weights)
```

```
      biases <- res[[1]]
      weights <- res[[-1]]
    }
    # Logging
    if(verbose){if(j %% 1 == 0){
      cat("Epoch: ", j, " complete")
      # Print acc and hide confusion matrix
      confusion <- evaluate(validation_data, biases, weights)
      }}
  }
  time.taken <- Sys.time() - start.time
  if(verbose){cat("Training complete in: ", time.taken)}
  cat("Training complete")
  # Return trained biases and weights
  list(biases, weights)
}
```

3.5 Update the bias and weights matricies for each mini-batch

```
update_mini_batch <- function(mini_batch, lr, C, sizes, num_layers, biases, weights)
{
  nmb <- length(mini_batch)
  listw <- sizes[1:length(sizes)-1]
  listb <-  sizes[-1]

  # Initialise updates with zero vectors (for EACH mini-batch)
  nabla_b <- lapply(seq_along(listb), function(idx){
    r <- listb[[idx]]
    matrix(0, nrow=r, ncol=1)
  })
  nabla_w <- lapply(seq_along(listb), function(idx){
    c <- listw[[idx]]
    r <- listb[[idx]]
    matrix(0, nrow=r, ncol=c)
  })

  # Go through mini_batch
  for (i in 1:nmb){
    x <- mini_batch[[i]][[1]]
    y <- mini_batch[[i]][[-1]]
    # Back propogation will return delta
    # Backprop for each obeservation in mini-batch
    delta_nablas <- backprop(x, y, C, sizes, num_layers, biases, weights)
    delta_nabla_b <- delta_nablas[[1]]
    delta_nabla_w <- delta_nablas[[-1]]
    # Add on deltas to nabla
    nabla_b <- lapply(seq_along(biases),function(j)
      unlist(nabla_b[[j]])+unlist(delta_nabla_b[[j]]))
    nabla_w <- lapply(seq_along(weights),function(j)
      unlist(nabla_w[[j]])+unlist(delta_nabla_w[[j]]))
  }
```

```
  # After mini-batch has finished update biases and weights:
  # i.e. weights = weights - (learning-rate/numbr in batch)*nabla_weights
  # Opposite direction of gradient
  weights <- lapply(seq_along(weights), function(j)
    unlist(weights[[j]])-(lr/nmb)*unlist(nabla_w[[j]]))
  biases <- lapply(seq_along(biases), function(j)
    unlist(biases[[j]])-(lr/nmb)*unlist(nabla_b[[j]]))
  # Return
  list(biases, weights)
}
```

3.6 Backpropogation algorithm

3.6.1 calculate partial derivatives using chain-rule (to update mini-batch).

The **forward step**, goes through the network layer-by-layer and calculates the output of the activation function. For example, the activations in layer l are:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

```
backprop <- function(x, y, C, sizes, num_layers, biases, weights)
{
  # Initialise updates with zero vectors
  listw <- sizes[1:length(sizes)-1]
  listb <-  sizes[-1]

  # Initialise updates with zero vectors (for EACH mini-batch)
  nabla_b_backprop <- lapply(seq_along(listb), function(idx){
    r <- listb[[idx]]
    matrix(0, nrow=r, ncol=1)
  })
  nabla_w_backprop <- lapply(seq_along(listb), function(idx){
    c <- listw[[idx]]
    r <- listb[[idx]]
    matrix(0, nrow=r, ncol=c)
  })

  # First:
  # Feed-forward (get predictions)
  activation <- matrix(x, nrow=length(x), ncol=1)
  activations <- list(matrix(x, nrow=length(x), ncol=1))
  # z = f(w.x + b)
  # So need zs to store all z-vectors
  zs <- list()
  for (f in 1:length(biases)){
    b <- biases[[f]]
    w <- weights[[f]]
    w_a <- w%*%activation
    b_broadcast <- matrix(b, nrow=dim(w_a)[1], ncol=dim(w_a)[-1])
    z <- w_a + b
    zs[[f]] <- z
```

```
    activation <- sigmoid(z)
    activations[[f+1]] <- activation  # Activations already contain one element
  }
  # Second:
  # Backwards (update gradient using errors)
  # Last layer
  delta <- cost_delta(method=C, z=zs[[length(zs)]], a=activations[[length(activations)]], y=y)
  nabla_b_backprop[[length(nabla_b_backprop)]] <- delta
  nabla_w_backprop[[length(nabla_w_backprop)]] <- delta %*% t(activations[[length(activations)-1]])
  # Second to second-to-last-layer
  # If no hidden-layer reduces to multinomial logit
  if (num_layers > 2) {
      for (k in 2:(num_layers-1)) {
        sp <- sigmoid_prime(zs[[length(zs)-(k-1)]])
        delta <- (t(weights[[length(weights)-(k-2)]]) %*% delta) * sp
        nabla_b_backprop[[length(nabla_b_backprop)-(k-1)]] <- delta
        testyy <- t(activations[[length(activations)-k]])
        nabla_w_backprop[[length(nabla_w_backprop)-(k-1)]] <- delta %*% testyy
      }
  }
  return_nabla <- list(nabla_b_backprop, nabla_w_backprop)
  return_nabla
}
```

3.6.2 These run a prediction on test-data and evaluate

```
feedforward <- function(a, biases, weights)
{
  for (f in 1:length(biases)){
    a <- matrix(a, nrow=length(a), ncol=1)
    b <- biases[[f]]
    w <- weights[[f]]
    # (py) a = sigmoid(np.dot(w, a) + b)
    # Equivalent of python np.dot(w,a)
    w_a <- w%*%a
    # Need to manually broadcast b to conform to np.dot(w,a)
    b_broadcast <- matrix(b, nrow=dim(w_a)[1], ncol=dim(w_a)[-1])
    a <- sigmoid(w_a + b_broadcast)
  }
  a
}

get_predictions <- function(test_X, biases, weights)
{
  lapply(c(1:length(test_X)), function(i) {
    which.max(feedforward(test_X[[i]], biases, weights))}
  )
}

evaluate <- function(testing_data, biases, weights)
{
  test_X <- lapply(testing_data, function(x) x[[1]])
```

```
  test_y <- lapply(testing_data, function(x) x[[2]])
  pred <- get_predictions(test_X, biases, weights)
  truths <- lapply(test_y, function(x) which.max(x))
  # Accuracy
  correct <- sum(mapply(function(x,y) x==y, pred, truths))
  total <- length(testing_data)
  print(correct/total)
  # Confusion
  res <- as.data.frame(cbind(t(as.data.frame(pred)), t(as.data.frame(truths))))
  colnames(res) <- c("Prediction", "Truth")
  table(as.vector(res$Prediction), as.vector(res$Truth))
}
```

3.6.3 Math helpers

```
# Calculate activation function
sigmoid <- function(z){1.0/(1.0+exp(-z))}

# Partial derivative of activation function
sigmoid_prime <- function(z){sigmoid(z)*(1-sigmoid(z))}
```

3.7 Load the data into a format the net accepts

```
train_test_from_df <- function(df, predict_col_index, train_ratio,
                               shuffle_input = TRUE, scale_input=TRUE)
{
  # Helper functions
  # Function to encode factor column as N-dummies
  dmy <- function(df)
  {
    # Select only factor columns
    factor_columns <- which(sapply(df, is.factor))
    if (length(factor_columns) > 0)
    {
      # Split factors into dummies
      dmy_enc <- model.matrix(~. + 0, data=df[factor_columns],
                              contrasts.arg = lapply(df[factor_columns],
                                                     contrasts, contrasts=FALSE))
      dmy_enc <- as.data.frame(dmy_enc)
      # Attach factors to df
      df <- cbind(df, dmy_enc)
      # Delete original columns
      df[c(factor_columns)] <- NULL
    }
    df
  }

  # Function to standarise inputs to range(0, 1)
  scalemax <- function(df)
  {
```

```
    numeric_columns <- which(sapply(df, is.numeric))
    if (length(numeric_columns)){df[numeric_columns] <- lapply(df[numeric_columns], function(x){
      denom <- ifelse(max(x)==0, 1, max(x))
      x/denom
    })}
    df
  }

  # Function to convert df to list of rows
  listfromdf <- function(df){as.list(as.data.frame(t(df)))}

  # Omit NAs (allow other options later)
  df <- na.omit(df)
  # Get list for X-data
  if (scale_input){
    X_data <- listfromdf(dmy(scalemax(df[-c(predict_col_index)])))
  } else {
    X_data <- listfromdf(dmy(df[-c(predict_col_index)]))
  }
  # Get list for y-data
  y_data <- listfromdf(dmy(df[c(predict_col_index)]))
  # Combine X,y
  all_data <- list()
  for (i in 1:length(X_data)){
    all_data[[i]] <- c(X_data[i], y_data[i])
  }
  # Shuffle before splitting
  if (shuffle_input) {all_data <- sample(all_data)}
  # Split to training and test
  tr_n <- round(length(all_data)*train_ratio)
  # Return (training, testing)
  list(all_data[c(1:tr_n)], all_data[-c(1:tr_n)])
}
```