Using R's neuralnet to predict breast cancer

Peter Vu

2024-11-29

Introduction

Breast cancer remains a significant health concern, affecting millions of women worldwide. In the United States alone, it is estimated that about 310,720 new cases of invasive breast cancer will be diagnosed in women in 2024 [1]. As the most common cancer in women, excluding skin cancers, breast cancer accounts for approximately 30% of all new female cancer diagnoses [2].

The fight against breast cancer has seen remarkable progress in recent years, with the 5-year relative survival rate for localized breast cancer now at an impressive 99% [1]. This improvement can be attributed to advances in early detection methods, increased awareness, and more effective treatment options. However, the complexity of breast cancer diagnosis and prognosis continues to challenge medical professionals.

Artificial intelligence and machine learning have emerged as powerful tools in revolutionizing breast cancer management. These technologies are enhancing various aspects of breast cancer care, from early detection and diagnostic precision to risk assessment and personalized treatment recommendations [3]. This RMarkdown document will go through the process of training a neural network to classify breast cancer using R's `neuralnet` library on the Wisconsin Breast Cancer Data dataset [4]. This will also serve as a detailed introduction to the `neuralnet` package in R.

Setup

We first import the dataset. The dataset can be found at [5], and information about the dataset can be found [6]. Note that the first row of the dataset seems to be a data row, so when using `read.csv()` to import the dataset, we need to set the `header` flag to `FALSE`.

```r
cancer <- read.csv("https://raw.githubusercontent.com/julien-arino/math-of-data-science/refs/heads/main/
```

we use `head()` to read the first five rows of the dataset and `dim()` to get the dimensions of the dataset:

```r
head(cancer)
```

```
##         V1 V2    V3    V4     V5     V6      V7      V8     V9     V10    V11
## 1   842302  M 17.99 10.38 122.80 1001.0 0.11840 0.27760 0.3001 0.14710 0.2419
## 2   842517  M 20.57 17.77 132.90 1326.0 0.08474 0.07864 0.0869 0.07017 0.1812
## 3 84300903  M 19.69 21.25 130.00 1203.0 0.10960 0.15990 0.1974 0.12790 0.2069
## 4 84348301  M 11.42 20.38  77.58  386.1 0.14250 0.28390 0.2414 0.10520 0.2597
## 5 84358402  M 20.29 14.34 135.10 1297.0 0.10030 0.13280 0.1980 0.10430 0.1809
## 6   843786  M 12.45 15.70  82.57  477.1 0.12780 0.17000 0.1578 0.08089 0.2087
##       V12    V13    V14   V15    V16      V17     V18     V19     V20     V21
## 1 0.07871 1.0950 0.9053 8.589 153.40 0.006399 0.04904 0.05373 0.01587 0.03003
## 2 0.05667 0.5435 0.7339 3.398  74.08 0.005225 0.01308 0.01860 0.01340 0.01389
## 3 0.05999 0.7456 0.7869 4.585  94.03 0.006150 0.04006 0.03832 0.02058 0.02250
## 4 0.09744 0.4956 1.1560 3.445  27.23 0.009110 0.07458 0.05661 0.01867 0.05963
## 5 0.05883 0.7572 0.7813 5.438  94.44 0.011490 0.02461 0.05688 0.01885 0.01756
## 6 0.07613 0.3345 0.8902 2.217  27.19 0.007510 0.03345 0.03672 0.01137 0.02165
##        V22   V23   V24    V25    V26    V27    V28    V29    V30    V31     V32
## 1 0.006193 25.38 17.33 184.60 2019.0 0.1622 0.6656 0.7119 0.2654 0.4601 0.11890
## 2 0.003532 24.99 23.41 158.80 1956.0 0.1238 0.1866 0.2416 0.1860 0.2750 0.08902
## 3 0.004571 23.57 25.53 152.50 1709.0 0.1444 0.4245 0.4504 0.2430 0.3613 0.08758
## 4 0.009208 14.91 26.50  98.87  567.7 0.2098 0.8663 0.6869 0.2575 0.6638 0.17300
## 5 0.005115 22.54 16.67 152.20 1575.0 0.1374 0.2050 0.4000 0.1625 0.2364 0.07678
## 6 0.005082 15.47 23.75 103.40  741.6 0.1791 0.5249 0.5355 0.1741 0.3985 0.12440
```

```r
dim(cancer)
```

```
## [1] 569  32
```

This means the dataset is a 569×32 matrix. In other words, there are 569 entries in the dataset, each of which has 32 attributes. We notice that the columns are unnamed, as the attributes themselves aren't part of the dataset. We look into [6] to learn more about the dataset. Under the *Additional Variable Information* section:

```
1) ID number
2) Diagnosis (M = malignant, B = benign)
3-32)

Ten real-valued features are computed for each cell nucleus:

    a) radius (mean of distances from center to points on the perimeter)
    b) texture (standard deviation of gray-scale values)
    c) perimeter
    d) area
    e) smoothness (local variation in radius lengths)
    f) compactness (perimeter^2 / area - 1.0)
    g) concavity (severity of concave portions of the contour)
    h) concave points (number of concave portions of the contour)
    i) symmetry
    j) fractal dimension ("coastline approximation" - 1)
```

It remains to use `names()` to name properly the columns in our dataset, according to the *Variables Table* section:

```r
names(cancer) <- c("ID", "Diagnosis", "radius1", "texture1",
                   "perimeter1", "area1", "smoothness1", "compactness1",
                   "concavity1", "concave_points1", "symmetry1", "fractal_dimension1",
                   "radius2", "texture2", "perimeter2", "area2",
                   "smoothness2", "compactness2", "concavity2", "concave_points2",
                   "symmetry2", "fractal_dimension2", "radius3", "texture3",
                   "perimeter3", "area3", "smoothness3", "compactness3",
                   "concavity3", "concave_points3", "symmetry3", "fractal_dimension3")
head(cancer)
```

```
##         ID Diagnosis radius1 texture1 perimeter1  area1 smoothness1
## 1   842302         M   17.99    10.38     122.80 1001.0     0.11840
## 2   842517         M   20.57    17.77     132.90 1326.0     0.08474
## 3 84300903         M   19.69    21.25     130.00 1203.0     0.10960
## 4 84348301         M   11.42    20.38      77.58  386.1     0.14250
## 5 84358402         M   20.29    14.34     135.10 1297.0     0.10030
## 6   843786         M   12.45    15.70      82.57  477.1     0.12780
##   compactness1 concavity1 concave_points1 symmetry1 fractal_dimension1 radius2
## 1      0.27760     0.3001         0.14710    0.2419            0.07871  1.0950
## 2      0.07864     0.0869         0.07017    0.1812            0.05667  0.5435
## 3      0.15990     0.1974         0.12790    0.2069            0.05999  0.7456
## 4      0.28390     0.2414         0.10520    0.2597            0.09744  0.4956
## 5      0.13280     0.1980         0.10430    0.1809            0.05883  0.7572
## 6      0.17000     0.1578         0.08089    0.2087            0.07613  0.3345
##   texture2 perimeter2  area2 smoothness2 compactness2 concavity2
## 1   0.9053      8.589 153.40    0.006399      0.04904    0.05373
## 2   0.7339      3.398  74.08    0.005225      0.01308    0.01860
## 3   0.7869      4.585  94.03    0.006150      0.04006    0.03832
## 4   1.1560      3.445  27.23    0.009110      0.07458    0.05661
## 5   0.7813      5.438  94.44    0.011490      0.02461    0.05688
## 6   0.8902      2.217  27.19    0.007510      0.03345    0.03672
##   concave_points2 symmetry2 fractal_dimension2 radius3 texture3 perimeter3
## 1         0.01587   0.03003           0.006193   25.38    17.33     184.60
## 2         0.01340   0.01389           0.003532   24.99    23.41     158.80
## 3         0.02058   0.02250           0.004571   23.57    25.53     152.50
## 4         0.01867   0.05963           0.009208   14.91    26.50      98.87
## 5         0.01885   0.01756           0.005115   22.54    16.67     152.20
## 6         0.01137   0.02165           0.005082   15.47    23.75     103.40
##    area3 smoothness3 compactness3 concavity3 concave_points3 symmetry3
## 1 2019.0      0.1622       0.6656     0.7119          0.2654    0.4601
## 2 1956.0      0.1238       0.1866     0.2416          0.1860    0.2750
## 3 1709.0      0.1444       0.4245     0.4504          0.2430    0.3613
## 4  567.7      0.2098       0.8663     0.6869          0.2575    0.6638
## 5 1575.0      0.1374       0.2050     0.4000          0.1625    0.2364
## 6  741.6      0.1791       0.5249     0.5355          0.1741    0.3985
##   fractal_dimension3
## 1            0.11890
## 2            0.08902
## 3            0.08758
## 4            0.17300
## 5            0.07678
## 6            0.12440
```

According to [6], there should be no missing data in the dataset. We use `sum(is.na.data.frame())` to count how many missing values there are just to be sure:

```
sum(is.na.data.frame(cancer))
```

```
## [1] 0
```

This verifies that there are no missing values in the dataset. We aren't really interested in the ID of each entry, so we can truncate that too. It's in the first column, so it's sufficient to remove that and keep every other column.

```r
cancer <- cancer[,2:ncol(cancer)]
```

We now import the `neuralnet` library. More information about the library can be found in [7].

```r
if (!require("neuralnet")) {
  install.packages("neuralnet")
  library("neuralnet")
}
```

```
## Loading required package: neuralnet
```

```
## Warning: package 'neuralnet' was built under R version 4.4.2
```

The goal of our neural network is to, given cell nuclei features, identify whether a tumor is *benign* (is not cancerous) or *malignant* (is cancerous). We will undergo the process of *supervised learning*, where we give the neural network a set of features with a known "result" (benign or malignant) to "correct" the neural network if it classifies something wrongly. After doing this multiple times, the neural network will readjust itself to assign the best *weights* for each feature, or how much each feature would contribute to the desired result.

For the last part of the setup, we want to separate the dataset into *training data* and *validation data*. We will run the training data through the algorithm multiple times, and use the validation data as a test for how accurate our algorithm is. The reason we need a validation dataset is because, after training the neural network multiple times, it may pick up on certain trends in our training dataset that might not accurately represent the relation between cell nuclei features and whether a tumor is benign or malignant. This is called *overfitting*, where the neural network adjusts to the training data very well but is much less accurate when dealing with new data. Therefore, our validation dataset will be a good estimator of how good the neural network truly is, as it's data it was never trained on, nor will it get to learn from.

Regarding how to split the dataset, we want sufficient data for the neural network to train on, but we also want sufficient data to evaluate the neural network effectively. We typically want the training data to be larger than the validation data, so we'll use a 2:1 ratio: two-thirds of the dataset for training and one-third for validation. We can use `sample()` to randomly sample a third of the dataset for the test set, and use the rest for the train set.

```r
set.seed(1) # For reproducibility
test_index <- sample(1:nrow(cancer), trunc(nrow(cancer) / 3))
test_set <- cancer[test_index,]
train_set <- cancer[-test_index,]
```

Training

We use the `neuralnet()` to generate a neural network. We need to provide it a `formula` and `data`, which respectively represent "a symbolic description of the model to be fitted" and "a data frame containing the variables specified in `formula`" [7]. Let's generate our first neural network using default arguments:

- For the formula, we provide `(Diagnosis == "B") + (Diagnosis == "M")~.`, which means we want to output whether the diagnosis is benign or malignant given *every* other feature in the dataset; since we've removed the ID column, everything else in our dataset is a relevant feature. This follows from the `# Multiclass classification` example in [7].
- For the data, we provide our training dataset defined earlier.

```
first_net <- neuralnet((Diagnosis == "B") + (Diagnosis == "M")~., train_set)
```

The `neuralnet()` function itself returns a `nn` class:

```
class(first_net)
```

```
## [1] "nn"
```

We can use `attributes()` to see the full list of attributes of this class:

```
attributes(first_net)
```

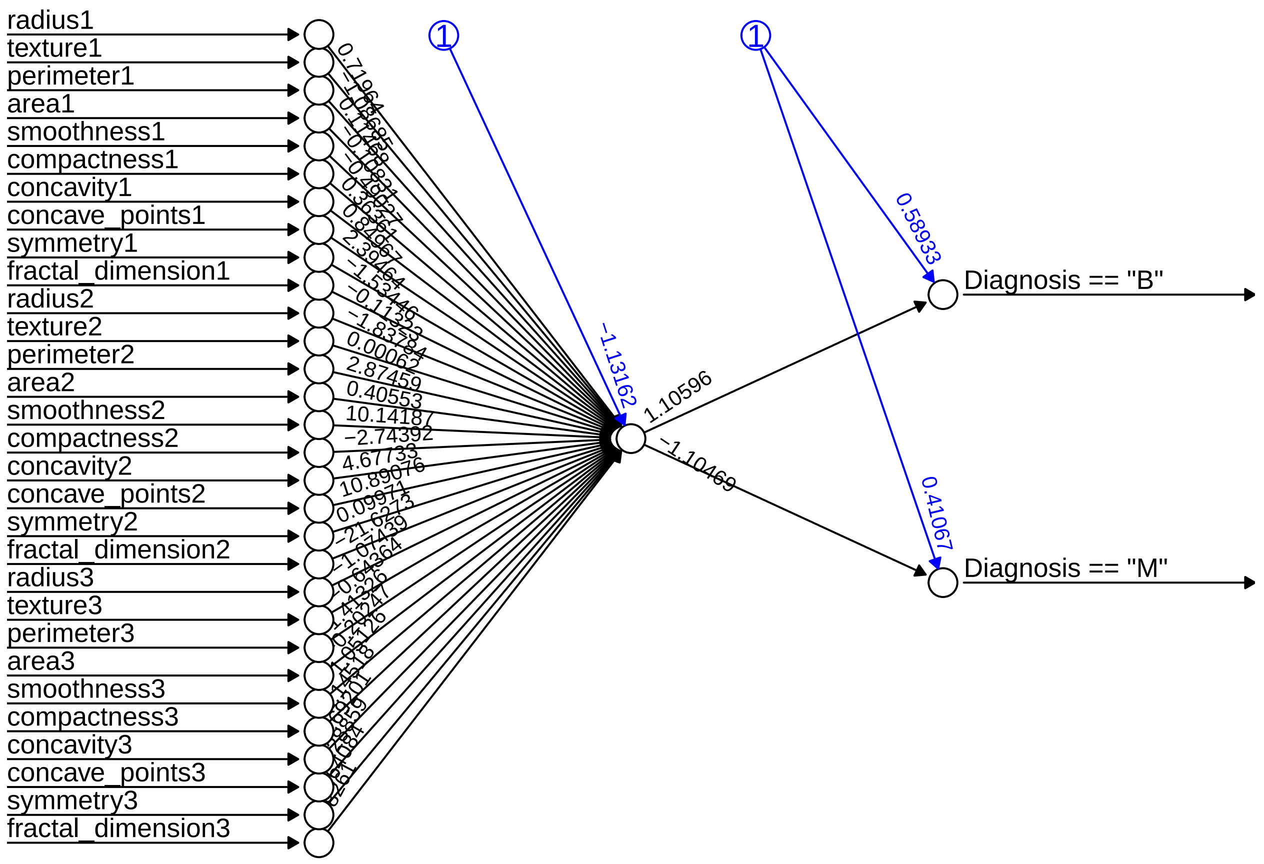
```
## $names
##  [1] "call"                "response"            "covariate"
##  [4] "model.list"          "err.fct"             "act.fct"
##  [7] "linear.output"       "data"                "exclude"
## [10] "net.result"          "weights"             "generalized.weights"
## [13] "startweights"        "result.matrix"
##
## $class
## [1] "nn"
```

According to the `neuralnet` documentation [7], these attributes represent the following:

- `call`: the matched call, which is word-for-word how we called `neuralnet()` with our properties earlier.
- `response`: extracted from the `data argument`, representing the classification of each row, or the desired output.
- `covariate`: the variables extracted from the `data argument`, representing the features of each row, or the input.
- `model.list`: a list containing the covariates and the response variables extracted from the `formula argument`.
- `err.fct`: the error function from the `err.fct argument`, a differentiable function that is used for the calculation of the error. Measures how well a model's predictions match the desired outcomes. Also called a loss function.
- `act.fct`: the activation function from the `act.fct argument`, a differentiable function that is used for smoothing the result of the cross product of the covariate or neurons and the weights. Decides whether a neuron should be activated by calculating the weighted sum of inputs and adding a bias term.
- `linear.output`: the `linear.output argument`. Decides whether `act.fct` should not be applied to the output neurons.
- `data`: the `data argument`.
- `exclude`: a vector or a matrix specifying the weights, that are excluded from the calculation from the `exclude argument`.
- `net.result`: a list containing the overall result of the neural network for every repetition.
- `weights`: a list containing the fitted weights of the neural network for every repetition.
- `generalized.weights`: a list containing the generalized weights of the neural network for every repetition. According to [8], "Generalized weights assist interpretation of NN model with respect to the independent effect of individual input variables. A large variance of generalized weights for a covariate indicates non-linearity of its independent effect. If generalized weights of a covariate are approximately zero, the covariate is considered to have no effect on outcome."
- `startweights`: a list containing the startweights of the neural network for every repetition.
- `result.matrix`: a matrix containing the reached threshold, needed steps, error, AIC and BIC (if computed) and weights for every repetition. Each column represents one repetition.

We can use `plot()` to plot the repetition with the smallest error of this neural network, using some arguments to make it more readable (`rep = "best"` is necessary for the plot to correctly show up when knitting [9]):

```
plot(first_net, rep = "best", radius = 0.1, arrow.length = 0.25, fontsize = 10)
```



The input layer contains 30 neurons, the features of the cell nuclei. Each of them is connected to a hidden node by a synapse, each of which having their own weight. Note that there is a bias/intercept node (the blue “1” node) also connected to the hidden node with its own weight. The hidden node is then connected to both output nodes, which represent respectively whether the Diagnosis is B (benign) and M (malignant). Another bias/intercept node in the output layer is also connected to these output nodes, and each connection has their own weight.

Prediction

Now that we have our first neural network, we can use it to predict data from our test set. The `neuralnet` library provide the `predict()` function for this. We just need to provide an `object`, our neural network, and `newdata`, the test set.

```
first_pred <- predict(first_net, test_set)
```

After saving the prediction results, we put out a table that compare the actual results to the values the prediction gave. We use `table()` for this, with the test set’s Diagnosis column as our first argument, and `apply(first_pred, 1, which.max)` as our second: this function will return either 1 or 2, where 1 is benign and 2 is malignant, depending on whether the input was more likely to be benign or malignant.

```r
table(test_set$Diagnosis, apply(first_pred, 1, which.max))
```

```
##
##       1
##   B 131
##   M  58
```

Only one column? That means the model thought it was more likely for *every* entry in our test set to be benign. The rows represent the actual proportions in the test set, so there are 131 benign and 58 malignant tumors in the test set. This still means the model classified 131 entries correctly, but it's not inspiring to know the model would classify every single entry the same way; it is useless if it can't correctly identify malignant tumors. We will have to build a better neural network without using default parameters.

Refining the model

We now take a deeper dive into the arguments the `neuralnet()` function takes [7]:

- `formula` and `data`: already described.
- `hidden`: a vector of integers specifying the number of hidden neurons (vertices) in each layer. The default value is 1; this explains why the above plot had exactly 1 node in the middle layer. We *will* need to look into changing this to improve our network.
- `threshold`: a numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria. The default value is 0, so the training will stop when the error reaches 1%. Any lower could result in substantially longer training time, so we will keep this for now.
- `stepmax`: the maximum steps for the training of the neural network. Reaching this maximum leads to a stop of the neural network's training process. The default value is `1e+05`, or 100000, which is plenty, so we will keep this the same.
- `rep`: the number of repetitions for the neural network's training. The default value is 1. This essentially means that we can choose to create multiple neural networks and fit them all at the same time, which may be beneficial as there is some randomness in creating a neural network. We can look into modifying this.
- `startweights`: a vector containing starting values for the weights. Set to `NULL` for random initialization. Since the default is already `NULL`, and we would have little sense of what would be good starting weights, we should keep this the same.
- `learningrate.limit`: a vector or a list containing the lowest and highest limit for the learning rate. Used only for RPROP and GRPROP. The default is `NULL`. As this is algorithm-specific, we won't touch this for now.
- `learningrate.factor`: a vector or a list containing the multiplication factors for the upper and lower learning rate. Used only for RPROP and GRPROP. The default is `list(minus = 0.5, plus = 1.2)`. As this is algorithm-specific, we won't touch this for now.
- `learningrate`: a numeric value specifying the learning rate used by traditional backpropagation. Used only for traditional backpropagation. The default is `NULL`. As this is algorithm-specific, we won't touch this for now.
- `lifesign`: a string specifying how much the function will print during the calculation of the neural network. 'none', 'minimal' or 'full'. The default is `"none"`, which we should keep so as not to clog the output.
- `lifesign.step`: an integer specifying the stepsize to print the minimal threshold in full lifesign mode. Doesn't matter as we have `lifesign` on `"none"`.
- `algorithm`: a string containing the algorithm type to calculate the neural network. The following types are possible: 'backprop', 'rprop+', 'rprop-', 'sag', or 'slr'. 'backprop' refers to backpropagation, 'rprop+' and 'rprop-' refer to the resilient backpropagation with and without weight backtracking, while 'sag' and 'slr' induce the usage of the modified globally convergent algorithm (grprop). The default

is "rprop+", or resilient backpropagation with weight backtracking. These are way too complicated to go over fully, but we might try different algorithms out to see if the accuracy improve, though the default should suffice.

- `err.fct`: a differentiable function that is used for the calculation of the error. Alternatively, the strings 'sse' and 'ce' which stand for the sum of squared errors and the cross-entropy can be used. The default is sum of squared errors, which is as it sounds; since our output space isn't complex, it should be sufficient compared to the more complex cross-entropy error function.
- `act.fct`: a differentiable function that is used for smoothing the result of the cross product of the covariate or neurons and the weights. Additionally the strings, 'logistic' and 'tanh' are possible for the logistic function and tangent hyperbolicus. The default is the logistic function, which is also fine as is.
- `linear.output`: logical. If act.fct should not be applied to the output neurons set linear output to TRUE, otherwise to FALSE. The default is `TRUE`, which we should consider changing to `FALSE` to see if it improves accuracy.
- `exclude`: a vector or a matrix specifying the weights, that are excluded from the calculation. If given as a vector, the exact positions of the weights must be known. A matrix with n-rows and 3 columns will exclude n weights, where the first column stands for the layer, the second column for the input neuron and the third column for the output neuron of the weight. The default is `NULL`, which we will keep as there isn't any reason to exclude any weights.
- `constant.weights`:
  a vector specifying the values of the weights that are excluded from the training process and treated as fix. The default is `NULL`, which we will keep as there isn't any reason to fix any weights.
- `likelihood`: logical. If the error function is equal to the negative log-likelihood function, the information criteria AIC and BIC will be calculated. Furthermore the usage of confidence.interval is meaningfull. The default is `FALSE`, but we may turn this on as an alternative model for e.g. evaluating the likelihood of the tumor being malignant.

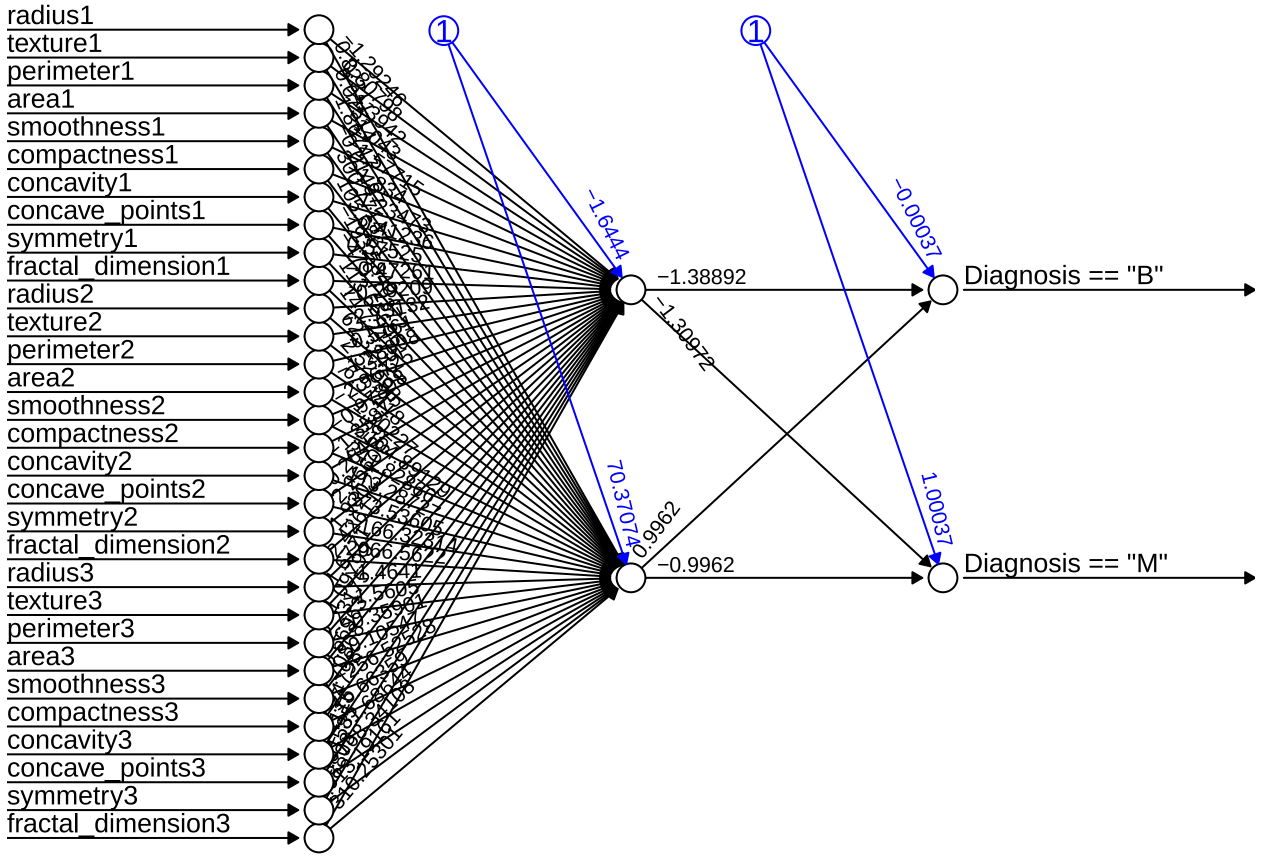
In short, we will want to consider changing `hidden`, `rep`, and `linear.output` for now.

The general rule of thumb is, the more hidden layers/nodes there are, the more complex the neural network is. As shown, a neural network with one hidden node can't even give a single malignant prediction, so there's no point going lower. Let's have a neural network with 2 hidden nodes, 1 hidden layer for our second neural network.

```r
second_net <- neuralnet((Diagnosis == "B") + (Diagnosis == "M")~., train_set,
                        hidden = 2)
```

Let's plot this:

```r
plot(second_net, rep = "best", radius = 0.1, arrow.length = 0.25, fontsize = 10)
```



Now we got two nodes in our hidden layer, each with different weights coming into and out of them. Let's use this neural network to predict the test set:

```
second_pred <- predict(second_net, test_set)
second_res <- table(test_set$Diagnosis, apply(second_pred, 1, which.max))
second_res
```

```
##
##       1   2
##   B 127   4
##   M   4  54
```

Immediate progress; our model now is capable of outputting both benign and malignant, and seems to be correct more often than not. To know the exact accuracy, let's quickly define a function that can calculate the accuracy given a table similar to the above.

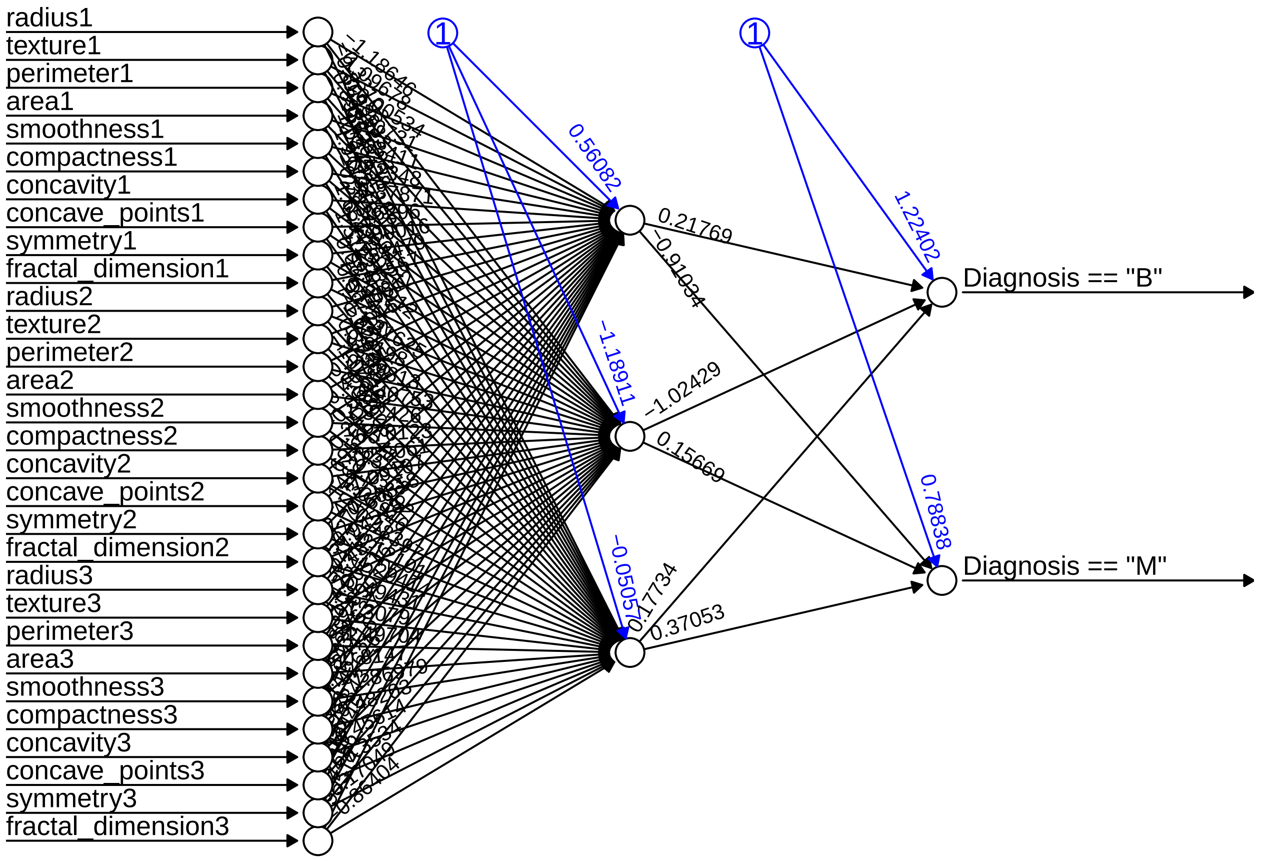
```
acc <- function(table) {
  total <- sum(table)

  # Complete table
  if (ncol(table) == 2) {
    correct <- table["B", 1] + table["M", 2]
  } else {
    # Incomplete table - either only outputted benign or malignant results
    if (colnames(table) == "1") {
      correct <- table["B", 1]
```

```r
    } else {
      correct <- table["M", 2]
    }
  }

  return (correct / total)
}
```

```r
second_acc <- acc(second_res)
```

The accuracy, for this prediction, stands at 95.77%. That's already pretty good, so we can capitalize on that. Let's try making a neural network with three hidden nodes:

```r
third_net <- neuralnet((Diagnosis == "B") + (Diagnosis == "M")~., train_set,
                        hidden = 3)

plot(third_net, rep = "best", radius = 0.1, arrow.length = 0.25, fontsize = 10)
```



```r
third_pred <- predict(third_net, test_set)
third_res <- table(test_set$Diagnosis, apply(third_pred, 1, which.max))
third_acc <- acc(third_res)

third_res
```

```
##
```

```
##         1
##     B 131
##     M  58
```

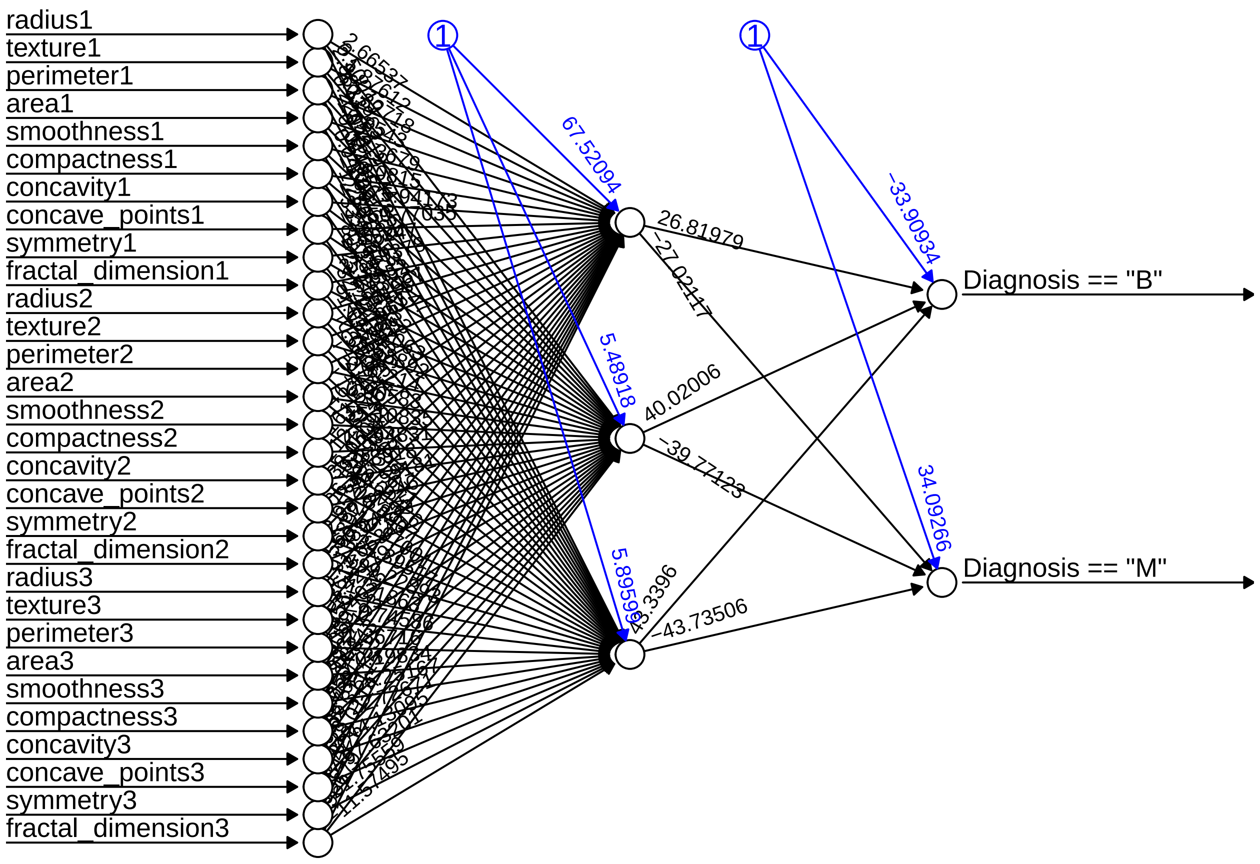
What happened? This is the exact same result as our one hidden node neural network from the earlier section: it tries to classify every entry as benign. The accuracy ends up being 69.31% for this test set, a significant downgrade. This is probably due to bad choices of start weights, leading to overfitting.

Let's try again: still three hidden nodes, but this time have 5 repetitions, and set `linear.output` to `FALSE`.

```
fourth_net <- neuralnet((Diagnosis == "B") + (Diagnosis == "M")~., train_set,
                        hidden = 3, rep = 5, linear.output = FALSE)
```

Since we have 5 repetitions, `rep = "best"` is actually useful to plot what our best network looks like:

```
plot(fourth_net, rep = "best", radius = 0.1, arrow.length = 0.25, fontsize = 10)
```



Unfortunately, `predict()` defaults to using the first repetition for predictions, and we can't specify "use the best repetition", only which one to use. Which means we'll have to put in work defining a function that returns the result table with the best accuracy.

```
table_predict_best <- function(net, data) {
  n <- length(net$net.result)

  # Check first rep by default
  pred <- predict(net, data)
  best_res <- table(data$Diagnosis, apply(pred, 1, which.max))
  best_acc <- acc(best_res)

  # Loop over second rep to however many there are
  if (n > 1) {
    for (i in 2:n) {
      pred <- predict(net, data, rep = i)
      curr_res <- table(data$Diagnosis, apply(pred, 1, which.max))
      curr_acc <- acc(curr_res)

      if (curr_acc > best_acc) {
        best_res <- curr_res
        best_acc <- curr_acc
      }
    }
  }

  return (best_res)
}
```

Now we check the results of the best repetition of this neural network:

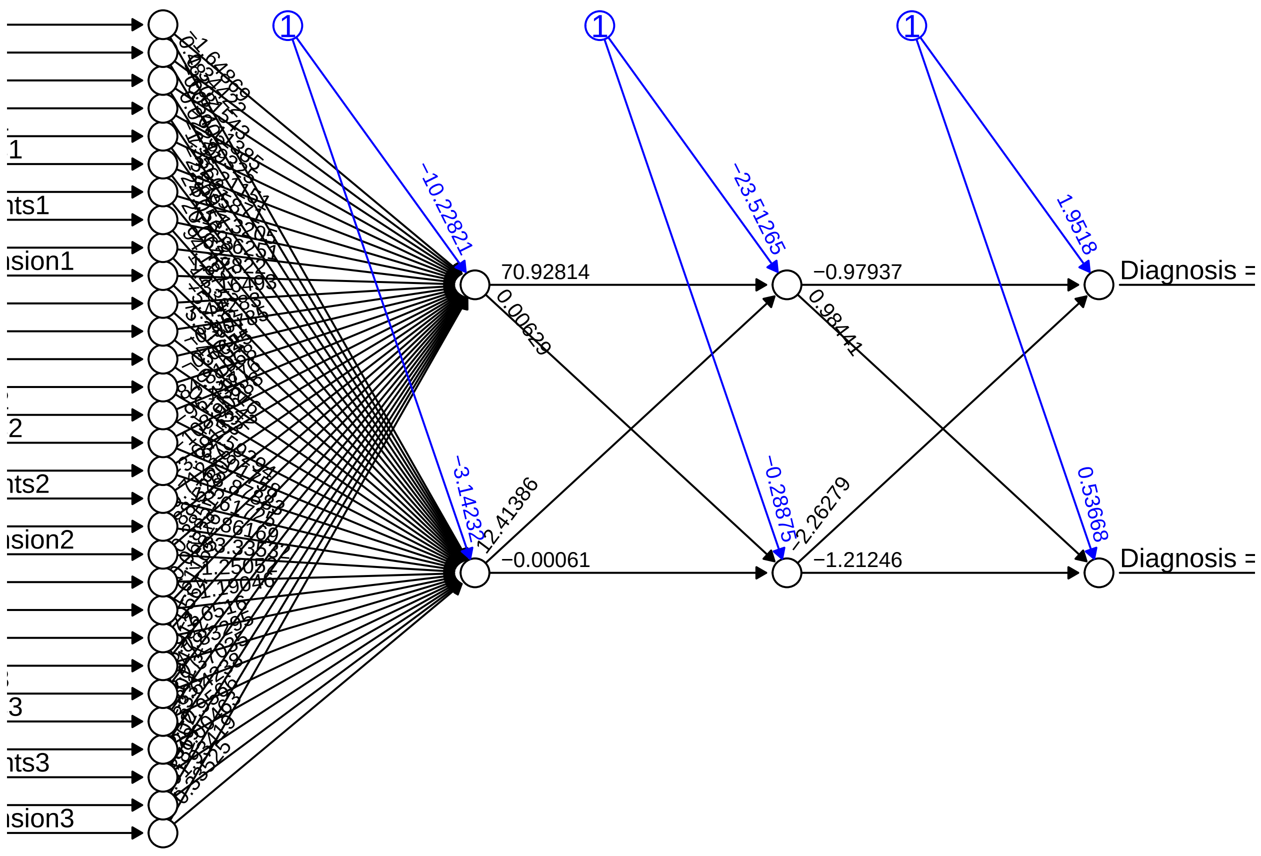
```
fourth_res <- table_predict_best(fourth_net, test_set)
fourth_acc <- acc(fourth_res)

fourth_res
```

```
##
##       1   2
##   B 127   4
##   M   7  51
```

The accuracy for this stands at 94.18%, which is good, though still slightly behind our earlier two hidden nodes network.

Another way we can go about adding hidden nodes is to add more layers. Since two worked so well, let's have two hidden layers of two nodes, with 5 repetitions and linear output:

```
fifth_net <- neuralnet((Diagnosis == "B") + (Diagnosis == "M")~., train_set,
                        hidden = c(2,2), rep = 5)

plot(fifth_net, rep = "best", radius = 0.1, arrow.length = 0.25, fontsize = 10)
```



```r
fifth_res <- table_predict_best(fifth_net, test_set)
fifth_acc <- acc(fifth_res)

fifth_res
```

```
##
##       1   2
##   B 126   5
##   M   1  57
```

An accuracy of 96.83%!

We can stop here for now. The reality is, if we set a different seed at the beginning, all the above results might drastically change. Training a neural network is a very black-box process, and it'll take drastic computing power as well as deeper knowledge about how all these different algorithms and functions work to "consistently" manage a highly accurate neural network.

References

1. https://www.nationalbreastcancer.org/breast-cancer-facts/
2. https://www.cancer.org/cancer/types/breast-cancer/about/how-common-is-breast-cancer.html
3. https://pmc.ncbi.nlm.nih.gov/articles/PMC10772314/
4. https://pages.cs.wisc.edu/~olvi/uwmp/cancer.html
5. https://raw.githubusercontent.com/julien-arino/math-of-data-science/refs/heads/main/CODE/wdbc.csv

6. https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic
7. https://github.com/bips-hb/neuralnet
8. https://atm.amegroups.org/article/view/10492/11983
9. https://stackoverflow.com/questions/43795530/knitr-and-plotting-neural-networks