Analysis of Heart Failure Data

Adam Goran & Alexander Crompton & Willie Langenberg

2022-03-08

Introduction

In this project we analyse a Heart Failure dataset collected from kaggle. The data consists of 299 rows of patients that have suffered heart failures and 13 columns. In the data we have 12 explanatory variables with medical information about the patients in the form of both continuous and discrete data with information such as the patient's age, whether they are a smoker or a non-smoker and the person's serum creatinine level.

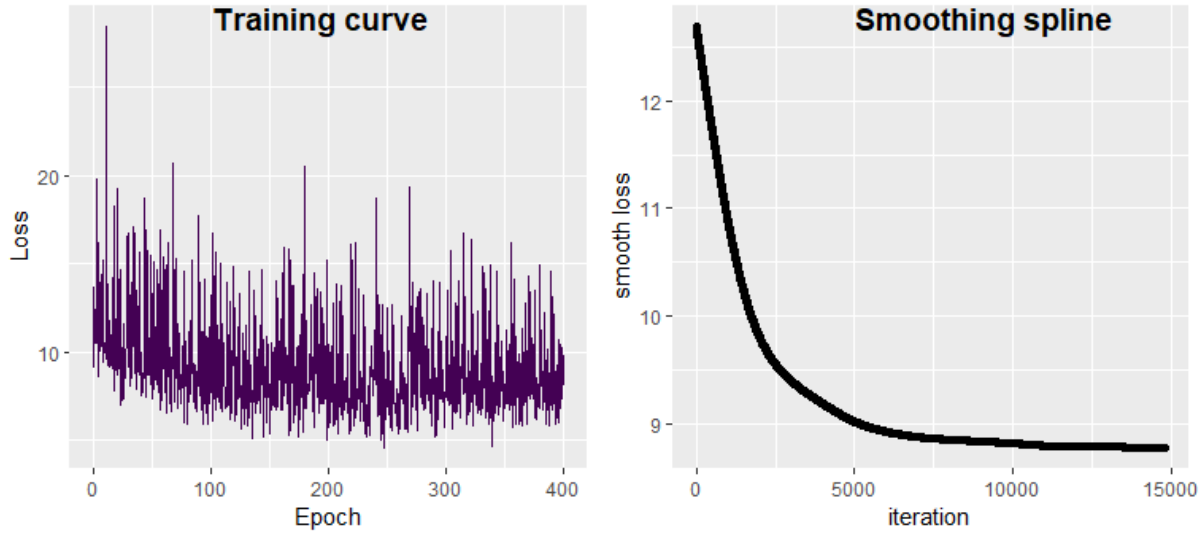
The ambition of our work is to predict the binary outcome variable “death_event”. The variable contains information on whether the patient died during the follow-up period and has 96 observed deaths and 203 cases where the person survived, so the classes are unbalanced. We use Autoencoder and feed-forward neural networks while trialling different activation functions for predictions. Then we use the permutation importance measure, which is a variable importance method for neural networks, to find which explanatory variables may affect the chance of survival for the patients.

Results

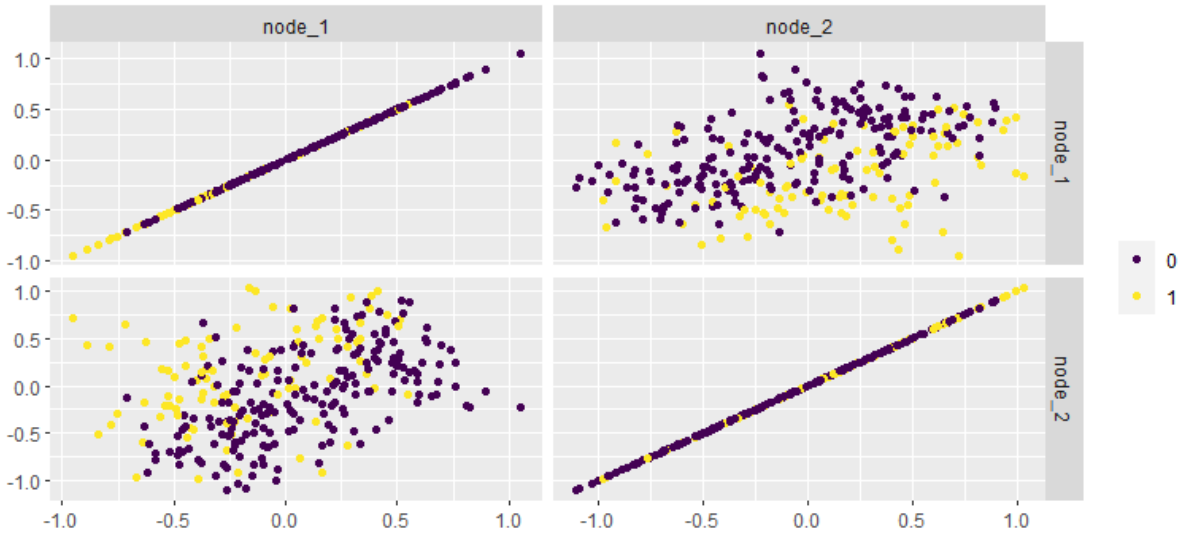
Autoencoder

To try the method and see how well the data separates in two dimensions we apply autoencoders using all variables except the response. The results might give us a hint of how well the feed forward network will perform. The first issue we run into is the choice of objective function to use to train the model. Some variables are binary, but more variables are continuous so we cannot use cross-entropy. A more sensible choice is the squared error loss in this case.

For training we use the “Adam” optimizer with learning rate 0.0001 and otherwise default parameters suggested by the book (0.9 and 0.999 for the specific momentum parameters). Our network has the form 12-10-6-2-6-10-12, so it is symmetric with two nodes in the middle layer. This way, we do not shrink the number of dimensions too fast and allow for much nonlinearity. The hyperbolic tangent activation function is used as activation function in all layers except the final reconstruction layer, where linear activation is used. In this context, ReLU is not a very good choice and tanh is usually preferred over the sigmoid activation. We have a batch size of 8 and 400 epochs. The network is implemented in **R** using the package **ANN2** containing the function **autoencoder.** To evaluate the training process, we show the training curve below.



For this implementation, the training curve fluctuates a lot. This is because the plot tries to show the loss for each batch during a single epoch, instead of averaging the loss over all batches. To get a more clear visualization of the training curve we apply smoothing splines, which is shown to the right. Note the change of the x-axis, one iteration is one run using a single batch, so we get 400*299/8≅15000 iterations. It seems like the training error has settled and there is no improvement in further training the network. Hence, we use this model to construct the compression plot below, i.e. a plot of the values in the middle layer.



In the plot above, we can see how the values from the two nodes in the middle layer appear in one and two dimensions. The points are coloured according to the response variable, where 1 denotes the event of decease. Even though the separation of the two classes might not have been what we hoped for, there are still areas in the plots where deaths are more likely. In particular, we can see that observations with large values of the first node more often correspond to people who survived.

One interesting application of the autoencoder is that we can look at the reconstruction of each point individually and detect observations with very high reconstruction errors. Using this method we looked at the observation with the highest reconstruction error, which was a

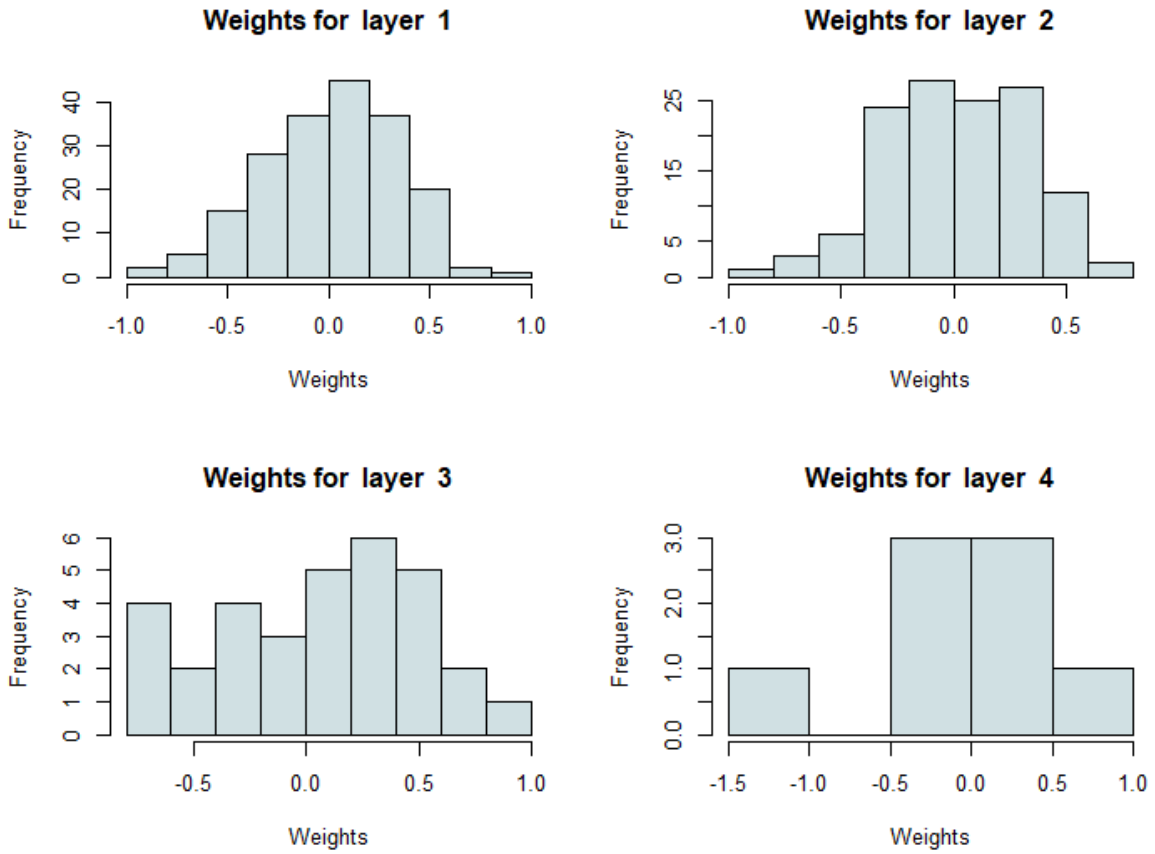
person with a platelets value of 850000. The mean for this variable across all observations is 263358 and the standard deviation is 97804.24, so this person is indeed an outlier. Other than this, it is difficult to evaluate how well the reconstruction is for the experiment and how much information is lost in the shrinking process. All variables are standardized in the training process, so a squared loss of around 9 with 12 variables might be decent.

Feed-Forward Neural Network

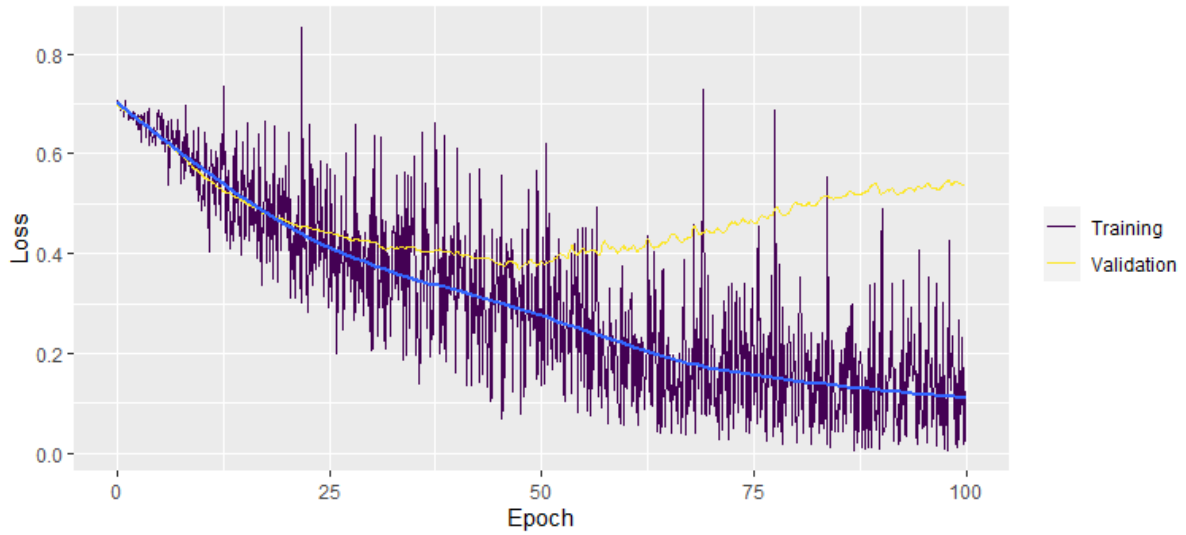
We implement a feed forward neural network to predict death event using the function **neuralnetwork** in the package **ANN2.** Considering this is a classification problem, cross-entropy is used as loss function. Further, we used the Adam optimizer again with the same default settings beta1 = 0.9 and beta2=0.999. We tried various network architectures using both wide networks with few layers and deeper networks with smaller hidden layers. We got similar results suggesting almost any network would work, but finally settled with the deeper network 12-16-8-4-2-1. We thought that the data might not be so complex or nonlinear and therefore we would get away with almost any model architecture with the right hyperparameter tuning. However, since a deeper model is more preferable in general we chose that one. Examine the model summary in the table below, stating the amount of trainable parameters used.

Layer	**Nodes**	**Activation**	**Amount of parameters**
Input	12	-	-
Hidden Layer 1	16	ReLU	208
Hidden Layer 2	8	ReLU	136
Hidden Layer 3	4	ReLU	36
Hidden Layer 4	2	ReLU	10
Output	1	Sigmoid	3
Total	-		**393**

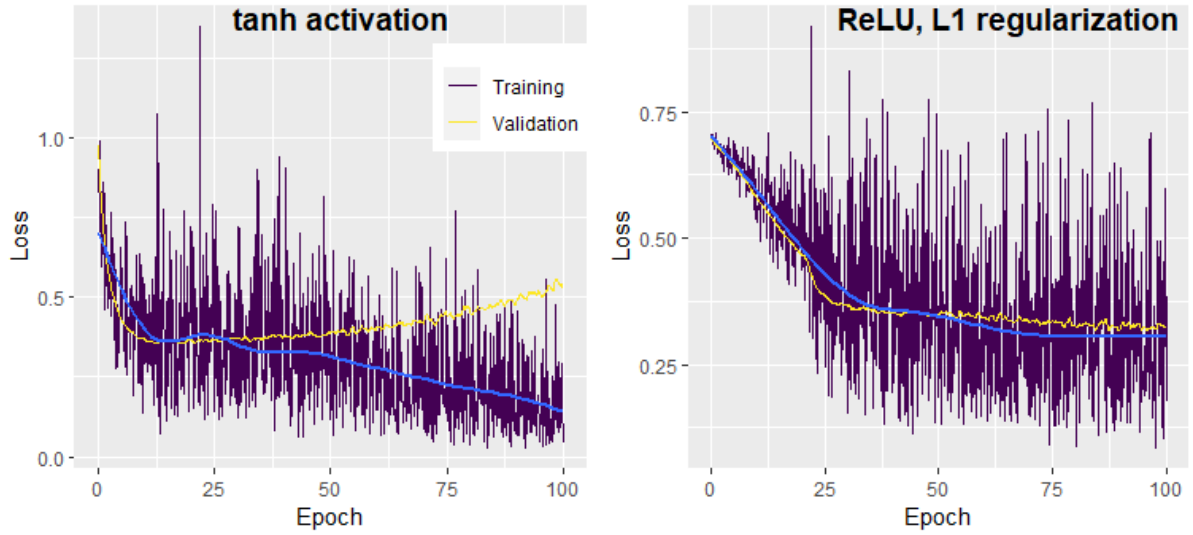
In each hidden layer we used ReLu as the activation function, because of its nice properties with good performance and fast training (due to the simple derivative). We also standardized the inputs, to get faster convergence and training speed. In the output layer we used the sigmoid function, to get probabilities for classifying the response variable. For the training we used a batch size of 10, and 100 epochs. To evaluate the model architecture, we show the distributions of weights in each layer below.



It could be dangerous with too large weights, because it makes the network sensitive to small changes in the input variables. Moreover, it is also concerning with too small weights, since it essentially could prevent too much information from flowing through the network. In our case we thought the weight distribution looked reasonable. If too many weights would have been close to 0 for a particular layer, there could be redundancy in the number of hidden units or the whole layer. During the training we also tracked the training-/validation loss, seen in the image below. The blue line is the smoothed training loss, whereas the purple line is the loss given by each batch (24 per epoch).



We can see that both losses are decreasing, but after around 50 epochs the validation loss seems to go up again. This suggests that the network is overfitting. To overcome this issue we tried regularization using both L1 and L2. By comparison we found that L1 was most effective, giving the lowest validation loss with a visually distinctive convergence after all epochs. We also tried to train the network with the same hyperparameters but changed the activation function to the hyperbolic tangent function. This unexpectedly manifested in a faster convergence in loss and also a small decrease in the minimal loss. However, since the ReLU is better in general we thought we could achieve the same result with more hyperparameter tuning, and therefore did not change the activation function for the “final” model. See the losses in the image below for reference to the regularization and tanh activation.



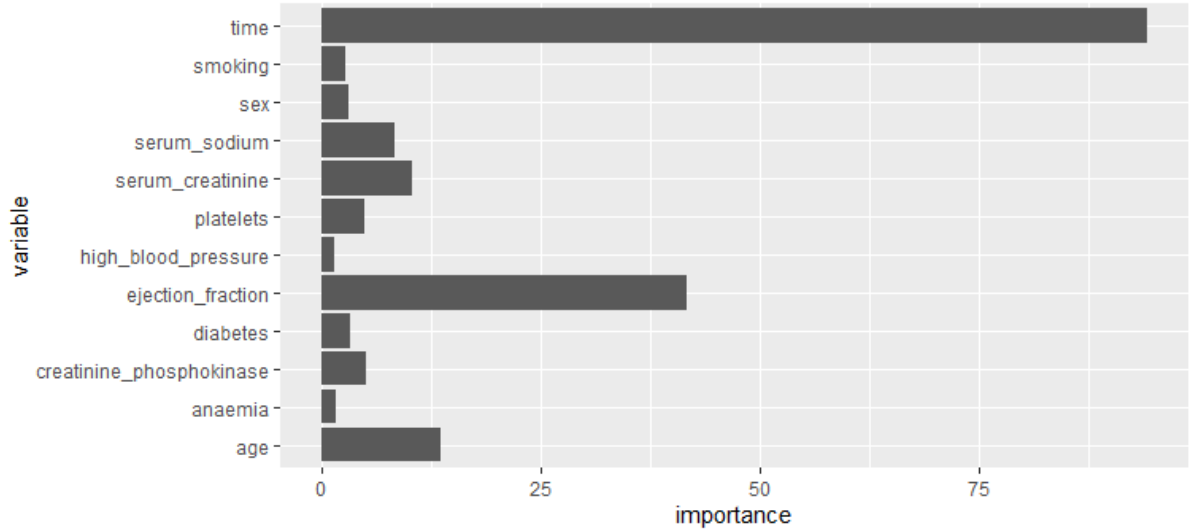
To evaluate the model we created a confusion matrix using all data. It is often best to evaluate the model on unseen data, otherwise we might come to conclusions using biased results, when the training data is included. We reasoned that the total amount of data was too low, hence excluding 20% for testing, and 20% for validation we would only use 64% (~190 observations) for training, which might have been too low compared to the 393 parameters used in the network. Now consider the confusion matrix below.

		Actual	
Prediction		0	1
	0	188	22
	1	15	74

From the confusion matrix we can calculate the sensitivity to 92.6% and specificity to 77.1%. The low specificity is probably related to the unbalanced data we have. Depending on the application of the data, one could argue that specificity is more important. It might be worse with false negatives, predicting survival but with the actual outcome of death. We naively tried over-sampling to balance the data, which simply is randomly resampling the data to get equal observations with the outcome of death relative to survival. Under-sampling could also be used, by deleting observations with outcome of survival, but with the same reason as

before this could lead to a too small amount of training data. Over-sampling greatly increased the specificity, and only decreased the sensitivity a few percentage points. However, by doing this we would likely be more prone to overfitting. Although it is important in reality to handle the low specificity, we thought it might be out of scope for this report. A better way could be to modify the loss function in a way that penalizes false negatives more than false positives.

To find which variables are of importance we use Permutation feature importance. The idea behind the feature is to randomly shuffle each predictor one by one and then observe the effect on the loss. The more increase in loss value, the more important the predictor. A potential hazard with the importance measure is that each variable is shuffled alone which means if there are variables with high correlation the measure might fail. However the data has low correlation.



We find that “time”, which is the number of days after they followed up on the patient, is the most important variable by quite some margin. This is most likely due to the fact that many patients that were followed up early had died. Another important variable is “ejection_fraction” which is a medical variable showing the percentage of blood leaving the heart at each contraction. “age” was another important variable which makes sense as age usually is an important factor in fatality data. Otherwise, many variables have very small impact or no impact at all. A drawback with this method is that we do not know in which direction a certain predictor affects the result.

Source code

```
library(ANN2)
library(tidyverse)
library(ggpubr)
library(corrplot)

df<-read.csv('heart_failure_clinical_records_dataset.csv')

## Autoencoder
df_unsup = as.matrix(df[,-13])
# autoencoder model
ae = autoencoder(df_unsup, hidden.layers = c(10,6,2,6,10), val.prop = 0,
                 activ.functions = "tanh", optim.type = "adam", batch.size = 8,
                 n.epochs = 400, random.seed = 19)

# compression plot
compression_plot(ae, df_unsup, colors = factor(df$DEATH_EVENT))

# plot of training curve
plot_1 = plot(ae) + theme(legend.position="none")

# smoothing splines on training losses
losses = ae$Rcpp_ANN$getTrainHistory()
smooth = smooth.spline(x=1:14800, y = losses$train_loss)
plot_2 = tibble(x=smooth$x, y=smooth$y) %>%
  ggplot(aes(x, y)) +
    geom_point() +
    labs(x = "iteration", y = "smooth loss")

ggarrange(plot_1, plot_2, ncol = 2, nrow = 1,
          labels = c("Training curve", "Smoothing spline"), hjust = -1)

# investigate the observation with the highest anomaly score
rec = reconstruct(ae, df_unsup)
m = which.max(recX$anomaly_scores)
df[m, ]
```

```
## Relu Feed Forward

nn = neuralnetwork(X=as.matrix(df[,-13]),
                   y=as.matrix(df[,13]),
                   hidden.layers  = c(16,8,4,2),
```

```
                    loss.type ="log",
                    regression = FALSE,
                    activ.functions = "relu",
                    random.seed = 1999,
                    learn.rates = 0.001,
                    optim.type = "adam",
                    adam.beta1 = 0.9,
                    adam.beta2 = 0.999,
                    n.epochs = 100,
                    batch.size = 10,
                    val.prop = 0.2,
                    verbose = TRUE)

y_seq = round(seq(1, 2400, length.out = 2000), 0)
x_seq = seq(0,100,length.out = 2000)
losses = nn$Rcpp_ANN$getTrainHistory()$train_loss[y_seq]
plot(nn) +
  geom_smooth(aes(x=x_seq, y=losses), se = FALSE)

## Tanh feed forward
nn2 = neuralnetwork(X=as.matrix(df[,-13]),
                    y=as.matrix(df[,13]),
                    hidden.layers  = c(16,8,4,2),
                    loss.type = "log",
                    regression = FALSE,
                    activ.functions = "tanh",
                    random.seed = 1999,
                    learn.rates = 0.001,
                    optim.type = "adam",
                    adam.beta1 = 0.9,
                    adam.beta2 = 0.999,
                    n.epochs = 100,
                    batch.size = 10,
                    val.prop = 0.2,
                    verbose = TRUE)

losses2 = nn2$Rcpp_ANN$getTrainHistory()$train_loss[y_seq]

plot1 = plot(nn2) +
  geom_smooth(aes(x=x_seq, y=losses2), se = FALSE) +
  theme(legend.position = c(0.85,0.8))

## Relu regularized feed forward
nn3 = neuralnetwork(X=as.matrix(df[,-13]),
                    y=as.matrix(df[,13]),
                    hidden.layers  = c(16,8,4,2),
```

```
                    loss.type ="log",
                    regression = FALSE,
                    activ.functions="relu",
                    random.seed = 1999,
                    learn.rates = 0.001,
                    optim.type = "adam",
                    adam.beta1 = 0.9,
                    adam.beta2 = 0.999,
                    n.epochs = 100,
                    batch.size = 10,
                    val.prop = 0.2,
                    verbose = TRUE,
                    L1 = 5)

losses3 = nn3$Rcpp_ANN$getTrainHistory()$train_loss[y_seq]
plot2 = plot(nn3) +
  geom_smooth(aes(x=x_seq, y=losses3), se = FALSE) +
  theme(legend.position = "none")

ggarrange(plot1, plot2, ncol = 2, nrow = 1, hjust = c(-1.1,-0.7),
          labels = c("tanh activation", "ReLU, L1 regularization"))
```

```
# Predict using model "nn3" on whole dataset
pred = predict(nn3, df[, -13])

# Accuracy on whole dataset
mean(pred$predictions == df$DEATH_EVENT)

# Confusion Matrix on whole dataset
conf_matrix <- confusionMatrix(data = as.factor(pred$predictions), reference =
as.factor(df$DEATH_EVENT))
conf_matrix

# Plot distribution of weights
par(mfrow = c(2,2))
for (weight_i in seq(1,4)) {
  weights = nn$Rcpp_ANN$getParams()$weights[[weight_i]]
  hist(c(weights), main = paste("Weights for " , paste('layer ',  weight_i)),
xlab='Weights', col='#d0e0e3')
}

## Permutation importance
# function to evaluate the cross-entropy, given the predicted probabilities probs
loss = function(probs) {
```

```
  -mean(df$DEATH_EVENT*log(probs[,2]) + (1-df$DEATH_EVENT)*log(probs[,1]))
}

# function to shuffle and calculate the loss, given variable index
perm = function(var) {
  out = vector(length = 100)
  for (i in 1:100) {
    d = df[,-13]
    d[,var] = d[sample(nrow(d)),var]
    preds = predict(nn3, newdata = as.matrix(d))$probabilities
    res = loss(preds)
    out[i] = res
  }
  mean(out)
}

# loss on the original data
pred_org = predict(nn3, newdata = as.matrix(df[,-13]))$probabilities
org_loss = loss(pred_org)

# calculating the importance of each variable and rescaling
v1 = (perm(1)/org_loss-1)*100
v2 = (perm(2)/org_loss-1)*100
v3 = (perm(3)/org_loss-1)*100
v4 = (perm(4)/org_loss-1)*100
v5 = (perm(5)/org_loss-1)*100
v6 = (perm(6)/org_loss-1)*100
v7 = (perm(7)/org_loss-1)*100
v8 = (perm(8)/org_loss-1)*100
v9 = (perm(9)/org_loss-1)*100
v10 = (perm(10)/org_loss-1)*100
v11 = (perm(11)/org_loss-1)*100
v12 = (perm(12)/org_loss-1)*100
vals = c(v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12)

# barplot of results
tibble(variable = names(df[,-13]), importance = vals) %>%
  ggplot(aes(variable, importance)) +
    geom_bar(stat="identity") +
    coord_flip()
```