Solution Lesson07: Neural Networks

Machine Learning using R

Exercise 1: Boston housing data

The Boston housing data set contains information collected by the U.S Census Service concerning housing in the area of Boston Massachusetts.

a) The data set is included in the R package `MASS`. Load the `MASS` package and look at the data by typing `Boston`. Get an overview of the data, what are the dimensions of the data set? You can get more info about the meaning of the variables by calling the help page (**Hint**: `?Boston`).

```
library(MASS)
head(Boston)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12  5.21
##   medv
## 1 24.0
## 2 21.6
## 3 34.7
## 4 33.4
## 5 36.2
## 6 28.7
```

```
str(Boston)
```

```
## 'data.frame':    506 obs. of  14 variables:
##  $ crim   : num  0.00632 0.02731 0.02729 0.03237 0.06905 ...
##  $ zn     : num  18 0 0 0 0 0 12.5 12.5 12.5 12.5 ...
##  $ indus  : num  2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87 7.87 ...
##  $ chas   : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ nox    : num  0.538 0.469 0.469 0.458 0.458 0.458 0.524 0.524 0.524 0.524 ...
##  $ rm     : num  6.58 6.42 7.18 7 7.15 ...
##  $ age    : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
##  $ dis    : num  4.09 4.97 4.97 6.06 6.06 ...
##  $ rad    : int  1 2 2 3 3 3 5 5 5 5 ...
##  $ tax    : num  296 242 242 222 222 222 311 311 311 311 ...
##  $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2 15.2 ...
##  $ black  : num  397 397 393 395 397 ...
##  $ lstat  : num  4.98 9.14 4.03 2.94 5.33 ...
##  $ medv   : num  24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

```
dim(Boston)
```

```
## [1] 506  14
```

There are 506 rows and 14 variables.

b) We want to predict the variable `medv` in the data set using a neural network. Since neural networks usually converge much better with standardized predictors, we want to standardize all predictors in the Boston data (**Hint**: `scale()`). Try to find a way to create a data frame in which all predictors are standardized but the target variable `medv` is still in its original form.

```
### Standardize predictors:
d.bos <- as.data.frame(scale(Boston))
d.bos$medv <- Boston$medv
```

Note: The target variable could also be standardized but normally this is not necessary. If the target variable is standardized then the predictions of the model will be on the standardized scale, too!

c) Fit a single hidden layer neural network to the (scaled) Boston data with only one neuron in the hidden layer and no weight decay. Throughout this exercise, we use `maxit=10000` to make sure the NN can converge (**Hint**: `nnet(..., decay=0, maxit=10000)`). What is the meaning of the `linout` option in the `nnet` function and how do we have to set it in this case?

```
### Fit simple NN:
library(nnet)
nn.bos1 <- nnet(medv ~ ., data=d.bos, size=1, decay=0, linout=TRUE, maxit=10000)
```

```
## # weights:  16
## initial  value 295738.548574
## iter  10 value 31773.042402
## iter  20 value 28594.635305
## iter  30 value 28403.937141
## iter  40 value 28353.571881
## iter  50 value 28350.716623
## iter  60 value 28326.195719
## iter  70 value 28286.853095
## iter  80 value 28201.897369
## iter  90 value 28133.816616
## iter 100 value 28030.169648
## iter 110 value 27770.083310
## iter 120 value 23511.481559
## iter 130 value 22837.107091
## iter 140 value 22350.417858
## iter 150 value 20638.170735
## iter 160 value 19122.605434
## iter 170 value 16916.006118
## iter 180 value 9491.117429
## iter 190 value 8878.649537
## iter 200 value 8633.160311
## iter 210 value 8089.572539
```

```
## iter 220 value 7994.230497
## iter 230 value 7988.998680
## iter 240 value 7924.986952
## iter 250 value 7892.346225
## iter 260 value 7891.692401
## iter 270 value 7891.547719
## iter 270 value 7891.547641
## iter 280 value 7891.526320
## iter 280 value 7891.526284
## final  value 7891.525202
## converged
```

The `linout` option defines whether a linear activation function is used for the output layer. Since the target variable `medv` is numeric, we have to set it to `TRUE`. Otherwise only values in the range of 0-1 would be predicted due to the default logistic activation function.

d) Using only one neuron in the hidden layer might be a too simple structure for the `Boston` data. We now want to let the R package `caret` tune a neural network for us. First, one needs to define a search grid with the possible parameter values. We only want to tune the size of the network and not use any weight decay. Generate a small search grid with the possible sizes 1, 2, 3 and 20. We add the size 20 to see how a much more complex network performs. **Hint**: `expand.grid(..., decay=0)`

```
t.grid <- expand.grid(size=c(1:3, 20), decay=0)
```

e) Use the `train()` function of `caret` to train a single hidden layer neural network along our search grid (This might take approx. 3 minutes). What structure was finally selected? What was the RMSE of this model? How did the most complex NN perform?

```
set.seed(2321)
library(caret)
tune.caret <- train(medv ~ ., data=d.bos,
                    method='nnet', tuneGrid=t.grid, maxit=10000, trace=FALSE,
                    linout=TRUE, preProcess=c("center","scale"))
```

```
## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo, :
## There were missing values in resampled performance measures.
```

```
# trace=FALSE suppresses progress message
tune.caret
```

```
## Neural Network
##
## 506 samples
##  13 predictor
##
## Pre-processing: centered (13), scaled (13)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
## Resampling results across tuning parameters:
##
##   size  RMSE       Rsquared   MAE
```

```
##    1     5.705678  0.6359772  3.888244
##    2     4.825513  0.7321638  3.231899
##    3     4.641168  0.7490473  3.134749
##   20    11.709935  0.4689973  5.772480
##
## Tuning parameter 'decay' was held constant at a value of 0
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were size = 3 and decay = 0.
```

As can be seen the neural network with size 3 performed best with an RMSE of 4.641 and was, therefore, selected. The model with size = 20 performed much worse, which indicates that it was over-fitting the data due to the model's complexity. In practice it would make sense to extend the search grid above the size 3, since larger sizes might still achieve a better performance.

(**Note**: The displayed warning about "`missing values in resampled performance measures`" is not further problematic in this case. It signals that in some evaluation steps for the most simple NN (size=1) certain performance measures (in this case the R-squared value) could not be computed. The reason is that the neural network did possibly not have the best fit and predicted the same number for all cases of the resampling step. For predictions with no variance the R-squared value cannot be computed which triggers this warning.)

f) We want to compare how a Random Forest is performing with this data set. Fit a Random Forest with the `cforest()` function to the Boston data consisting of 500 trees and an `mtry` value of 5. We want to calculate the (OOB) RMSE of the Random Forest. To this end we need to subtract the OOB predictions from the true `medv` values, and take the root of the mean squared differences (**Hint**: `sqrt(mean((d.bos$medv - ...))^2)`). What is the RMSE of the Random Forest?

```
set.seed(7349)
### Fit a random forest:
library(party)
cfor_ctr <- cforest_unbiased(ntree = 500, mtry = 5)
rf.bos <- cforest(medv ~ ., data=d.bos, controls = cfor_ctr)
### RMSE (OOB):
sqrt(mean((d.bos$medv - predict(rf.bos, OOB=TRUE))^2))
```

```
## [1] 3.853376
```

As we can see, the Random Forests performs seemingly better on the Boston data with an RMSE of 3.853. We could also start tuning for example the mtry parameter of the Random Forest with the `caret` package. However, Random Forests are known to perform pretty well "out-of-the-box" without excessive tuning and we will not further refine this model.

Exercise 2: Image classification with the MNIST dataset

The MNIST dataset contains a collection of 28x28 pixel images of handwritten digits (grayscale). It is a commonly used data set to train and test machine learning methods for visual processing.

a) As described in the lecture, grayscale pictures are often represented by tables with three axes. In R, tables of higher dimensions are represented by "multi-way arrays". We can create a multi-way array with the `array()` command, which requires a vector of numbers to fill the array (`data=`) and a second

vector indicating the dimensions of the array (`dim=`). Create a three-dimensional array including the numbers 1 to 24 with the dimensions `c(2,4,3)`. Look at the created table in R.

```
x <- array(data = 1:24, dim = c(2,4,3))
x
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   19   21   23
## [2,]   18   20   22   24
```

```
dim(x)
```

```
## [1] 2 4 3
```

We can see that the array has three dimensions, R plots the first two dimensions separately along the third dimension. These are like three "slices" of a cube.

b) We can access individual elements in multi-way arrays like for normal tables using the square brackets (`[ ]`). Display only the element of the second row, of the third column of the first slice. What happens if you select one entire slice, e.g. the second one?

```
x[2,3,1]
```

```
## [1] 6
```

```
### Select entire slice:
x[,,2]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
```

When we select the entire second slice we see that the output is the corresponding 2-dimensional table.

c) Read in the the MNIST data containing the 28x28 pixel images. The data is stored in the "**Mnist_training_and_test_data.RData**" file. This is not a .csv file as usual but an .RData file. RData files store saved R-objects. Once downloaded, read in the the data with `load("Mnist_training_and_test_data.RData")`. Now there should be four new objects in your

environment: `test_x.0`, `test_y.0`, `train_x.0` and `train_y.0`. These objects store the pictures and the corresponding labels of the training and test set. Look at the dimensions of the arrays. How many pictures are in the training and test set? Also look at the target variable of the data (`test_y.0` and `train_y.0`).

```
load('Mnist_training_and_test_data.RData')
dim(train_x.0)
```

```
## [1]    28    28 60000
```

```
dim(train_y.0)
```

```
## [1] 60000
```

```
dim(test_x.0)
```

```
## [1]    28    28 10000
```

```
dim(test_y.0)
```

```
## [1] 10000
```

```
### Look at labels (only done for test set):
test_y.0[1:20]
```

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

Each picture has 28x28 pixels. We can see that the arrays are organized so that the third dimension indicates the different pictures, while the first and second dimensions indicate the pixel positions. There are 60000 pictures in the training set and 10000 pictures in the test set. The `test_y.0` and `train_y.0` objects simply list the numbers shown on the individual pictures.

d) In the second image of the training data, what is the grayscale value of the pixel in the top right corner of the image?

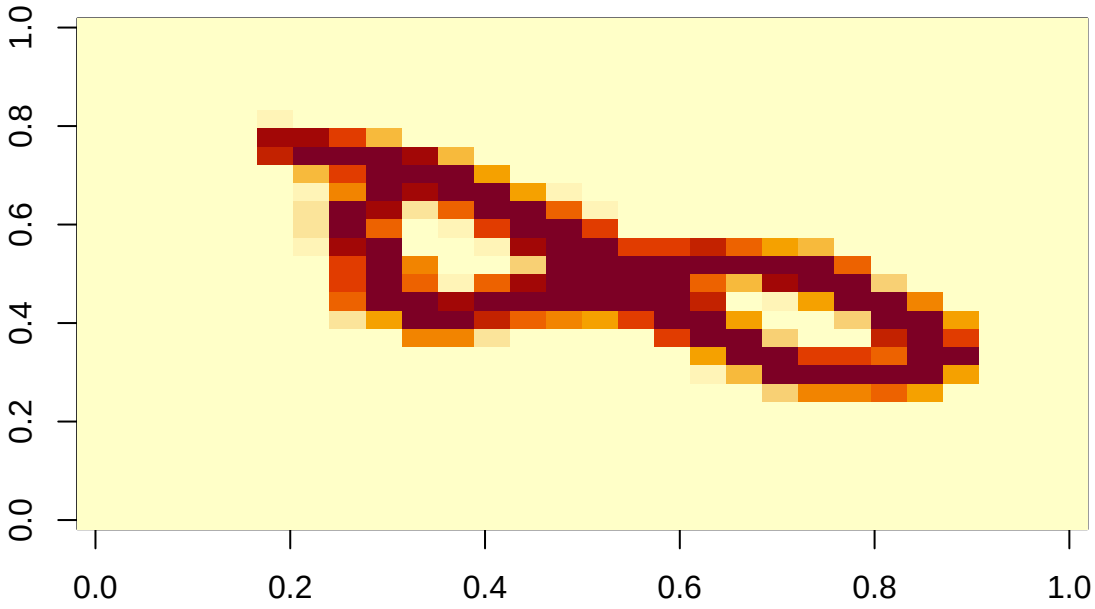
```
train_x.0[1, 28, 2]
```

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

The pixel in the top row on the far right has a value of 0. In the MINST data zero corresponds to white while 255 corresponds to black pixels. So this is a white pixel.

e) Each slice in the `train_x.0` array corresponds to one picture (3rd dimension). We can plot one picture by selecting only one slice in the array and feed it into the `image()` function. Select the 18th picture of `train_x.0` and visualize it with `image()`.

```
pc18 <- train_x.0[,,18]
image(pc18)
```



As we can see the 18th image is a handwritten 8. The `image()` function in its default setting plots the image 90-degree rotated and adds colour to the grayscale image.

f) To directly feed the MNIST pictures to a standard neural network, we need to "flatten" the images, so that each image is one row and the columns represent all the pixels of a picture. Run the code below to flatten the images of the training and test set. The code additionally scales the grayscale values to a range of 0-1 by dividing the values by 255 (better for NN). The target variable is turned into a factor since predicting the written number on a picture is a classification task. Finally, the pixel values are combined with the target variable into data frames. Look at the final data frames.

```
### Turn into 2d matrices (each row one picture):
train_x <- array(as.numeric(aperm(train_x.0, perm = c(3, 1, 2))), dim = c(60000, 784))
test_x <- array(as.numeric(aperm(test_x.0, perm = c(3, 1, 2))), dim = c(10000, 784))
### Scale to 0-1 range:
train_x <- train_x/255
test_x <- test_x/255
### Turn number labels to factor:
train_y <- as.factor(train_y.0)
test_y <- as.factor(test_y.0)
### Combine to dataframe:
mnist.dtra <- data.frame("y"=train_y, train_x)
mnist.dte <- data.frame("y"=test_y, test_x)
### Look at dimensions:
dim(mnist.dtra)
```

```
## [1] 60000   785
```

```
dim(mnist.dte)
```

```
## [1] 10000   785
```

We see that the training data is now a normal data frame with the dimensions 60000 x 785. This corresponds to 60000 pictures, each with 28 x 28 = 784 pixels. Each column stores the value of one pixel and the first

column stores the target variable (1 + 784 = 785 columns).

g) Now the data is ready to be fed to a classifier, e.g. a neural network. We want to train a single hidden layer NN on the training data and check its performance on the test data. Since training a neural network with `nnet()` of size=20 on the MNIST data takes approximately 20 hours, you can download an already fitted NN stored in the "NN_mnist.RData" file. The file contains a fitted NN (`nne.mnist` object). It was created with the command:

```
nne.mnist <- nnet(y~., mnist.dtra, size=20, decay=0, maxit=10000, MaxNWts = 16000)
```

Predict the written numbers in the test set with the neural network and create a confusion matrix showing how well it performed (see slides). Calculate the accuracy of the predictions (accuracy = 1 - errorrate).

```
load('NN_mnist.RData')
### Confusion matrix:
conf.nn <- table(test_y, predict(nne.mnist, newdata = mnist.dte, type='class'))
conf.nn
```

```
##
## test_y    0    1    2    3    4    5    6    7    8    9
##      0  936    1    6    2    4    6   11    1    6    7
##      1    3 1099    5    2    2    2    4    3   13    2
##      2    8    2  927   22   16    9    7   17   21    3
##      3    5    8   21  894    3   39    3   16   12    9
##      4    2    4    2    6  908    6   14   12    5   23
##      5    5   18    5   35   11  767   15    5    8   23
##      6    9    5   11    0   10   23  888    2    9    1
##      7    3    6   22   17   10    3    1  942    2   22
##      8   12    1   14   25    8   21   18    7  859    9
##      9    4    5    2   11   44   14    1   18    5  905
```

```
### Accuracy (one minus missclassification rate):
cc <- conf.nn
diag(cc) <- 0
1 - sum(cc)/nrow(mnist.dte)
```

```
## [1] 0.9125
```

The neural network perfoms okay but not too great, reaching an accuracy of 0.912. For a better performance the network would have to be tuned properly. Normally, for picture recognition a convolutional neural network is used, which can easily reach an accuracy of 99.7% on the MNIST data set.

i) **Extra**: Compare how a Random Forest performs in the MNIST classification task. For a change we will use the classic `randomForest()` function from the R package with the same name, because it runs faster than `cforest()`. Fitting a Random Forest to the MNIST training data will be much faster than fitting a NN, in this case it will take approx. 10 minutes. Fit a Random Forest to the MNIST training data consisting (only) of 50 trees and check its performance using the test set. **Hint**: `library(randomForest)`; `randomForest(..., ntree=50)`. (The `mtry` argument can be left away, it will automatically be set to a rule-of-thumb value).

```
library(randomForest)
set.seed(3874)
rf.minst <- randomForest(y~., mnist.dtra, ntree=50)
```

```r
### Look at performance in test set:
conf.rf <- table(test_y, predict(rf.minst, newdata = mnist.dte))
conf.rf
```

```
##
## test_y    0    1    2    3    4    5    6    7    8    9
##      0  970    0    2    0    0    2    3    1    2    0
##      1    0 1122    3    3    0    2    3    0    1    1
##      2    6    0 1000    4    1    0    3   12    6    0
##      3    1    0   10  969    0    8    0   10    9    3
##      4    1    1    1    0  946    0    7    0    4   22
##      5    4    2    1   15    4  849    6    2    6    3
##      6    8    4    0    0    3    3  936    0    4    0
##      7    1    4   19    2    3    0    0  984    2   13
##      8    5    0    3    6    4    6    5    4  931   10
##      9    5    5    3    7   12    8    1    5    9  954
```

```r
### Accuracy:
cc <- conf.rf
diag(cc) <- 0
1 - sum(cc)/nrow(mnist.dte)
```

```
## [1] 0.9661
```

As we can see the Random Forest performs quite well on the MNIST data, reaching an accuracy of 0.966.