From linear to non-linear models

```
set.seed(1234)
```

Note: this Rmarkdown document uses some heavy-duty external packages (particularly the `torch` package for neural networks) that may be difficult to install properly. I encourage you to read through the document and study it, but it is not necessary to run the document for yourself (unless you want to do so).

Loading the data

For this demonstration we will use the Boston dataset from the `MASS` package. This dataset contains the median prices of houses in Boston areas (the `medv` column) as well as 13 different predictors. To keep things manageable, we will use only the following 4 predictors:

- `crim`: the crime rate in that neighborhood;
- `zn`: the amount of land available for large residential properties;
- `nox`: the level of nitric oxide (air pollution);
- `dis`: the weighted distance to 5 employment centers in Boston.

The dataset contains data for 506 neighborhoods. We split it into a training dataset of 406 observations (approx. 80%) and a test dataset with the remaining 100 observations (20%). The idea is that we will use the training dataset to build our model, and the test dataset to get an idea of the model's performance on entirely unknown data.

```
library(MASS)

little_boston <- Boston[c("crim", "zn", "nox", "dis", "medv")]

# Split into train/test data
test_idx <- sample(nrow(little_boston), 100)
little_boston_train <- little_boston[-test_idx,]
little_boston_test <- little_boston[test_idx,]

n_predictors <- ncol(little_boston_train)
n_obs <- nrow(little_boston_train)

nrow(little_boston_train)
```

```
## [1] 406
```

```
nrow(little_boston_test)
```

```
## [1] 100
```

Building a linear model, with R

It is easy to build a linear model with R, something that we have done many times in this course. We get a model where all predictors are significant. Chances are that you can improve this model a bit, by e.g. transforming some of the variables, but we will not do so here.

```
m <- lm(medv ~ ., data = little_boston_train)
summary(m)
```

```
##
## Call:
## lm(formula = medv ~ ., data = little_boston_train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -14.624  -4.808  -1.591   2.446  29.001
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  50.81961    3.81315  13.327  < 2e-16 ***
## crim         -0.30567    0.04863  -6.286 8.50e-10 ***
## zn            0.15235    0.02183   6.980 1.23e-11 ***
## nox         -37.23945    5.29275  -7.036 8.61e-12 ***
## dis          -2.14007    0.32824  -6.520 2.12e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.586 on 401 degrees of freedom
## Multiple R-squared:  0.3202, Adjusted R-squared:  0.3134
## F-statistic: 47.22 on 4 and 401 DF,  p-value: < 2.2e-16
```

Building a linear model, by hand

We can build the same model by hand, using the tools that we developed in the section on nonlinear modeling.

We will use matrix algebra for this, so let's convert our dataset in a matrix. We actually build two matrices: the `X`-matrix contains our predictors and has everything other than the last column (the `medv` column) and the `y`-matrix is the last column.

```
X_train <- as.matrix(little_boston_train[-n_predictors])
y_train <- as.matrix(little_boston_train[n_predictors])
```

A convenience function that we will need a few times:

```
prepend_ones <- function(mat) {
  ones <- rep(1, nrow(mat))
  return(cbind(ones, mat))
}

prepend_ones(matrix(c(5, 6, 7, 8), nrow = 2))
```

```
##      ones
## [1,]    1 5 7
## [2,]    1 6 8
```

With that, we can write down our linear model and our objective function.

```
linear_model <- function(X, thetas) {
  y_hat <- prepend_ones(X) %*% thetas
  return(y_hat)
}
```

```
Jtheta <- function(thetas, FUN, X, y) {
  y_hat <- FUN(X, thetas)
  y_resid <- y - y_hat
  J <- sum(y_resid^2)
  return(J)
}
```

Let's check if this model is indeed the same model as what R uses. If we give it the regression coefficients that R obtained earlier, we should get the exact same data as in the regression summary. As a check, we compute the residual standard error of the model.

```
SS <- Jtheta(coef(m), linear_model, X_train, y_train)
df <- n_obs - n_predictors
sqrt(SS/df)
```

```
## [1] 7.586127
```

Now we go one step further. We use `optim` to find the regression coefficients, and we verify that they agree with the values that R found. This is definitely not the most efficient way of finding the regression coefficients, but it is pretty cool that we can get the same results as R with just a few lines of code.

```
thetas_init <- rep(0, n_predictors)
linear_fit <- optim(thetas_init, Jtheta, FUN = linear_model, X = X_train, y = y_train, method = "L-BFGS
linear_fit
```

```
## $par
## [1]  50.8195838  -0.3056648   0.1523487 -37.2395367  -2.1400685
##
## $value
## [1] 23077.28
##
## $counts
## function gradient
##       55       55
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

```
linear_fit$par - coef(m)
```

```
##   (Intercept)          crim            zn           nox           dis
## -2.456964e-05  3.185893e-06  1.004713e-06 -8.345091e-05  1.672181e-06
```

Last, we evaluate the squared error of the model on the test dataset. This is the first time that we use the test dataset, so we can be sure that this provides an unbiased assessment of the model's performance. We will compute this number with the squared error of other models that we will build later on. Note that when comparing two models, lower is better: a model with lower squared error makes predictions that are closer to the data.

```
X_test <- as.matrix(little_boston_test[-n_predictors])
y_test <- as.matrix(little_boston_test[n_predictors])

Jtheta(linear_fit$par, linear_model, X_test, y_test)
```

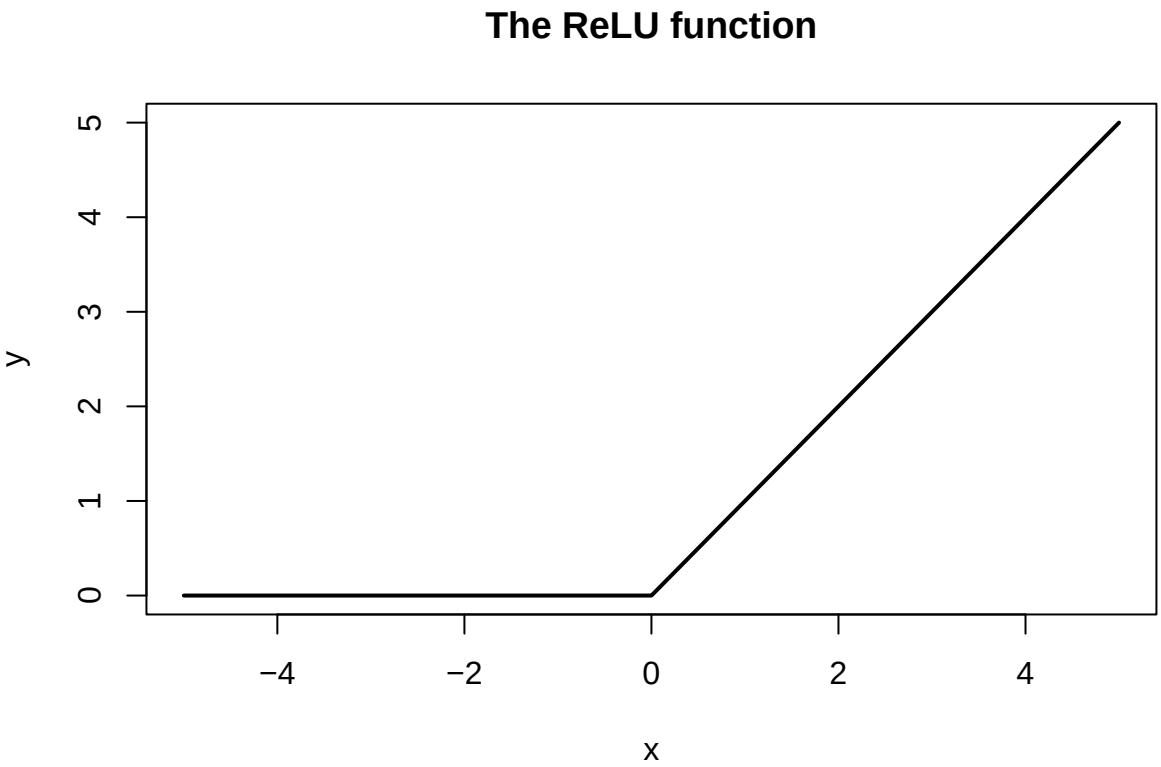
```
## [1] 5445.96
```

Building a nonlinear model

It is unlikely that Boston house prices follow a linear model (and the regression diagnostic plots would confirm that, had we looked at them). We could try to address that defect by transforming the variables and adding higher-order terms to our model. Here, we will not take that approach. Instead, we will take recourse to a bigger model. We will stack two linear models together, separated by a nonlinearity. The nonlinearity that we choose is the so-called "Rectified Linear Unit (ReLU)", defined as $\text{ReLU}(x) = \max(x, 0)$. It is a very simple function, but it will do the trick. It is plotted below:

```
relu <- function(x) {
  pmax(x, 0)
}

x_plot <- seq(-5, 5, length.out = 11)
y_plot <- relu(x_plot)
plot(x_plot, y_plot, type = "l", xlab = "x", ylab = "y", lwd = 2, main = "The ReLU function")
```



The nonlinear model that we use consists of two linear "layers", separated by the ReLU function.

```
nonlinear_model <- function(X, thetas) {
  # Parameters for both models
  thetas_1 <- matrix(thetas[1:15], nrow = 5, ncol = 3, byrow = FALSE)
  thetas_2 <- matrix(thetas[16:19], nrow = 4, ncol = 1)

  # Model 1
  y_1 <- prepend_ones(X) %*% thetas_1

  # Nonlinearity
  y_1 <- relu(y_1)

  # Model 2
```

```
  y_2 <- prepend_ones(y_1) %*% thetas_2

  return(y_2)
}
```

We will initialize the parameter fit with random data. If you start with all 0s, chances are that you will obtain a parameter fit that is not very good.

```
thetas_init <- rnorm(19)
```

Fitting the model. We have to give it some more iterations to converge to an optimum, because the loss landscape is very complex.

```
nonlinear_fit <- optim(thetas_init, Jtheta, FUN = nonlinear_model, X = X_train, y = y_train, method = "
                       control = list(maxit = 1000))
nonlinear_fit
```

```
## $par
##  [1]  70.7744933  -1.1925480   0.8814684 -56.9852663  -5.0905821   9.3089156
##  [7]  -2.8663462  -0.4720493  -0.1890297  -2.8642088  35.0049797 -47.9457529
## [13]   1.3349524 -46.6470151  -2.9857599   9.7097560   0.6937652  -3.6877419
## [19]  -0.3533591
##
## $value
## [1] 18830.5
##
## $counts
## function gradient
##      493       96
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The error on the unseen data is a bit lower, so performance is a bit better.

```
Jtheta(nonlinear_fit$par, nonlinear_model, X_test, y_test)
```

```
## [1] 4115.768
```

The nonlinearity is **really necessary**. If you remove it, you get a linear model that is no better than the model on four predictors that we built earlier (compare the performance on unseen data for both models). This is because stacking together linear models without any nonlinear functions in between is mathematically equivalent to just building a single linear model.

```
model_without_nonlinearity <- function(X, thetas) {
  # Parameters for both models
  thetas_1 <- matrix(thetas[1:15], nrow = 5, ncol = 3, byrow = FALSE)
  thetas_2 <- matrix(thetas[16:19], nrow = 4, ncol = 1)

  # Model 1
  y_1 <- prepend_ones(X) %*% thetas_1

  # Nonlinearity
  #y_1 <- relu(y_1)
```

```
  # Model 2
  y_2 <- prepend_ones(y_1) %*% thetas_2

  return(y_2)
}

model_without_nonlinearity_fit <- optim(
  thetas_init, Jtheta, FUN = model_without_nonlinearity,
  X = X_train, y = y_train, method = "BFGS",
  control = list(maxit = 1000))

Jtheta(model_without_nonlinearity_fit$par,
       model_without_nonlinearity, X_test, y_test)
```

```
## [1] 5445.965
```

Building a neural network via a 3-party library

What we have built in the section above is an example of a **neural network** consisting of two linear layers separated by a nonlinear activation function (ReLU). Finding the optimal parameters of such a network via black-box optimization is not very efficient, though, and as the number of parameters grows this will become well-nigh impossible. Moreover, our handcrafted model is not very flexible, and only allows us to build small variations on the same kind of network.

With the explosion of interest in neural networks of the past 10 years, it should be no surprise that there are highly performant libraries that allow us to build industrial-strength neural networks. One such framework is Torch, which comes in a Python version (PyTorch) and an R version (`torch`).

Below we use Torch to build a neural network with 2 layers. Note that training is an iterative process, starting from randomly initialized weights. If you re-run the training process a few times, you will get different results.

```
library(torch)

X_train_t <- torch_tensor(X_train, dtype = torch_float())
y_train_t <- torch_tensor(y_train, dtype = torch_float())

X_test_t <- torch_tensor(X_test, dtype = torch_float())
y_test_t <- torch_tensor(y_test, dtype = torch_float())

model <- nn_sequential(
  # Layer 1
  nn_linear(4, 10),
  nn_relu(),
  # Layer 2
  nn_linear(10, 1),
)

criterion <- nn_mse_loss()
optimizer <- optim_adamw(model$parameters, lr = 0.1)

n_epochs <- 1000
for (epoch in 1:n_epochs) {
  optimizer$zero_grad()
```

```
  y_pred <- model(X_train_t)
  loss <- criterion(y_pred, y_train_t)
  loss$backward()
  optimizer$step()

  if (epoch %% 100 == 0) {
    cat("Epoch: ", epoch, "Loss: ", loss$item(), "\n")
  }
}
```

```
## Epoch:  100 Loss:  69.04303
## Epoch:  200 Loss:  61.73209
## Epoch:  300 Loss:  59.0762
## Epoch:  400 Loss:  54.88323
## Epoch:  500 Loss:  52.26884
## Epoch:  600 Loss:  51.52388
## Epoch:  700 Loss:  51.29399
## Epoch:  800 Loss:  51.95733
## Epoch:  900 Loss:  51.07703
## Epoch:  1000 Loss:  51.17054
```

Last but not least, we evaluate the performance of the network on the unseen test data. We get something that is (usually) a bit lower than with previous models, though not by much. There are two important conclusions to be drawn from this:

1. Fancy, state-of-the-art models such as neural networks are within reach with the tools and techniques that you have learned in this course.
2. Despite this, building a simple model with only a few parameters can be a very powerful thing to do. It is easy and efficient to do so, and simple models are usually straightforward to interpret and to reason about. Compare for example the interpretation that we gave to the coefficients of a linear regression – no such interpretation exists for the weights in a neural network.

Depending on the circumstances, simple models may outshine complex ones.

```
sum((model(X_test_t) - y_test_t)^2)
```

```
## torch_tensor
## 4857.14
## [ CPUFloatType{} ][ grad_fn = <SumBackward0> ]
```