Deep Learning

```r
library(ggplot2)
library(ggforce)
library(ISLR2)
library(glmnet)
library(tidyverse)
theme_set(theme_minimal())
options(scipen= 999)
```

Resources

How to install keras

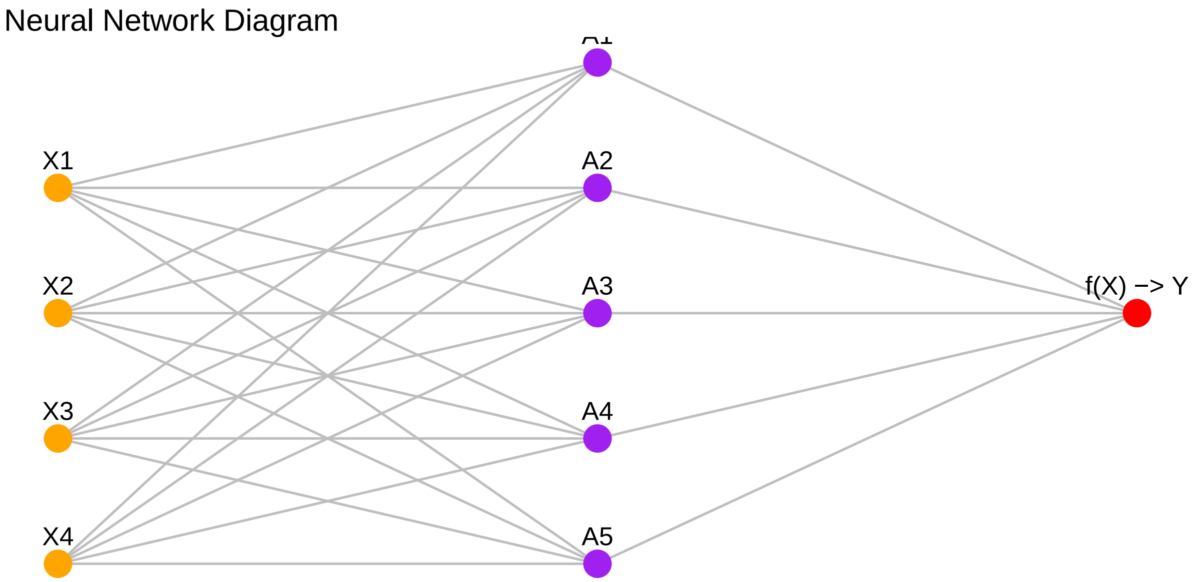
ISLR RBook club

> **Warning**
>
> Following the instructions above did not solve issues for me, so I created a new document on how I installed it (Gemini instructions). I installed python 3.8

Neural Networks.

Single Layer Neural Network (Feed Forward Neural Network)

Neural networks are usually represented by a neural diagram.



In orange we have the input layer, in this example with four variables, and then we have what it is called a hidden layer, with 5 units in there, and finally the output layer.

The hidden layer are transformations of the inputs, the A stands for *activations*

[

$$\begin{aligned} f(X) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j \right) \end{aligned}$$

]

Each of the lines are nonlinear function of a linear combination of the inputs. $$A_k =h_k(X) = g(w_{kj}X_j)$$ are called the activations in the hidden layer. And $g(z)$ is called the *activation function*. These non linear functions can be of different types. A popular ones is ReLU or Rectified Linear Unit. So the activations are like derived features.

The model is fit by minimizing $\sum_{i=1}^{n}(y_i - f(x_i))^2$ for regression.

Imagine we want to identify handwritten digits from 0 to 9. We have images in black and white for the sample digits, each of them in an image of 28x28 pixels, and each pixel get a greyscale from 0 to 255.

Our data has 60k digits for training and 10k for test.

We will have 60k inputs x pixels , and then two hidden layers, one with 256 units and one second hidden layer with 128, then we have 10 outputs (0-9)

Most of the Neural Networks theory is not presented in this document. It can be found in the ISLR book. The subject is too complex for the objective of this document.

When to use Deep Learning

Deep learning or neural networks has very good results when the data has a lot of signal and very little noise, this means that it is difficult to overfit the model, because overfitting is fitting the model on the noise and losing view of the real data trends, the signal. This is true for many things like image recognition, an image usually can be identified by a human into its classes with ease, that means that the image has all the information required to get a classification. Neural networks work very well also when there is some kind of structure in the data, like speech recognition, where there is some order of words to form sentences, or timeline forecasting.

An example where Neural Networks does not work so well is trying to predict if a drug is going to work based on the human genes, because there is a lot of noise in that case because human population gene data has a lot of noise. For those cases, simple models may work better than neural networks.

How to perform Deep Learning in RStudio

There are two ways to fit the Neural Network:

- using `keras`
- using `torch`

Keras requires some installation on RStudio see document Tensorflow Installation Guide.

```
# Step 1: Explicitly tell reticulate which Python to use
Sys.setenv(RETICULATE_PYTHON = "C:/Users/vegap/miniconda3/envs/islr-miniconda/python.exe")

# Step 2: Load the reticulate package
library(reticulate)

# Step 3: Verify reticulate's configuration
# This should now show numpy and tensorflow paths correctly
reticulate::py_config()
```

```
python:         C:/Users/vegap/miniconda3/envs/islr-miniconda/python.exe
libpython:      C:/Users/vegap/miniconda3/envs/islr-miniconda/python38.dll
pythonhome:     C:/Users/vegap/miniconda3/envs/islr-miniconda
version:        3.8.20 (default, Oct  3 2024, 15:19:54) [MSC v.1929 64 bit (AMD64)]
Architecture:   64bit
numpy:          C:/Users/vegap/miniconda3/envs/islr-miniconda/Lib/site-packages/numpy
numpy_version:  1.24.4

NOTE: Python version was forced by RETICULATE_PYTHON
```

```r
# Step 4: Load the R keras and tensorflow packages
library(keras)
library(tensorflow)

# Step 5: Try to import tensorflow directly via reticulate
# This is the line that previously failed
tf <- reticulate::import("tensorflow")

# Step 6: Confirm TensorFlow version and that it's working
print(tf$`__version__`)
```

```
[1] "2.10.0"
```

```r
# Optional: Basic TensorFlow operation to confirm
hello_tensor <- tf$constant("Hello, TensorFlow from R!")
tf$print(hello_tensor)
```

```r
#test it with a random data set:
x <- matrix(rnorm(1000), ncol = 10)  # Sample data for illustration

# Define and compile the neural network model
modnn <- keras_model_sequential() %>%
  layer_dense(units = 50,
    activation = "relu",
    input_shape = list(ncol(x))) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 1)

# Summary of the model
summary(modnn)
```

```
Model: "sequential"
________________________________________________________________________________
 Layer (type)                       Output Shape                      Param #
================================================================================
 dense_1 (Dense)                    (None, 50)                        550
 dropout (Dropout)                  (None, 50)                        0
 dense (Dense)                      (None, 1)                         51
================================================================================
Total params: 601
Trainable params: 601
Non-trainable params: 0
________________________________________________________________________________
```

We are going to use the `Hitters` dataset as we have done in previous chapters. First we fit a linear model:

```
hitters <- na.omit(Hitters)
n<- nrow(hitters)

set.seed(13)
ntest <-trunc(n/3)
testid<- sample(1:n, ntest)
training <- hitters[-testid,]
testing <- hitters[testid,]

#we fit a linear model to the training data and predict the values
lfit <- lm(Salary ~., data= training)
lpred<- predict(lfit, testing)
pred_test <- cbind(testing, lpred)

#calculate the difference between the result and the predictions.
mean(abs(pred_test$lpred - pred_test$Salary ))
```

```
[1] 254.6687
```

Next we fit the lasso using `glmnet`. Since this package does not use formulas, we create `x` and `y` first.

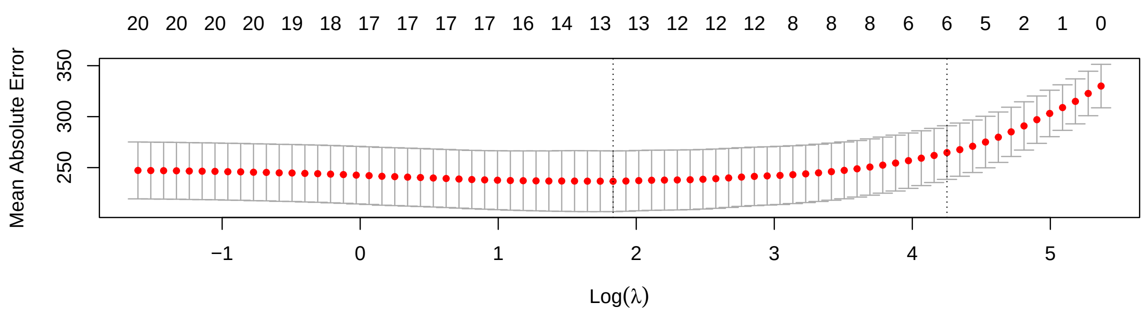
```
x<- scale(model.matrix(Salary ~. -1, data = hitters))
y<- hitters$Salary
```

The first line makes a call to `model.matrix()`, which produces the same matrix that was used by `lm()` (the `-1` omits the intercept). This function automatically converts factors to dummy variables. The `scale()` function standardizes the matrix so each column has mean zero and variance one. We make the predictions using `lambda.min` which is the min error in the cross validation.

```
cvfit <- cv.glmnet(x[-testid,], y[-testid], type.measure= "mae")
cpred <- predict(cvfit, x[testid,], s="lambda.min")
mean(abs(y[testid] - cpred))
```

```
[1] 252.2994
```

```
plot(cvfit)
```



To fit the neural network, we first set up a model structure that describes the network.

```
modnn <- keras_model_sequential() %>%
   layer_dense(units = 50, activation = "relu",
        input_shape = ncol(x)) %>%
   layer_dropout(rate = 0.4) %>%
   layer_dense(units = 1)
```

We have created a vanilla model object called `modnn`, and have added details about the successive layers in a sequential manner, using the function `keras_model_sequential()`. It allows us to specify the layers of a neural network in a readable form.

The object `modnn` has a single hidden layer with 50 hidden units, and a ReLU activation function. It then has a dropout layer, in which a random 40% of the 50 activations from the previous layer are set to zero during each iteration of the stochastic gradient descent algorithm. Finally, the output layer has just one unit with no activation function, indicating that the model provides a single quantitative output.

Next we add details to `modnn` that control the fitting algorithm. Here we have simply followed the examples given in the Keras book. We minimize squared-error loss `mse`. The algorithm tracks the mean absolute error on the training data, and on validation data if it is supplied.

```
modnn %>% compile(loss = "mse",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_absolute_error")
   )
```

In the previous line, the pipe operator passes `modnn` as the first argument to `compile()`. The `compile()` function does not actually change the `R` object `modnn`, but it does communicate these specifications to the corresponding `python` instance of this model that has been created along the way.

Now we fit the model. We supply the training data and two fitting parameters, `epochs` and `batch_size`. Using 32 for the latter means that at each step of SGD, the algorithm randomly selects 32 training observations for the computation of the gradient. An epoch amounts to the number of SGD steps required to process n observations. Since the training set has $n = 176$, an epoch is $176/32 = 5.5$ SGD steps. The `fit()` function has an argument `validation_data`; these data are not used in the fitting, but can be used to track the progress of the model (in this case reporting the mean absolute error). Here we actually supply the test data so we can see the mean absolute error of both the training data and test data as the epochs proceed. To see more options for fitting, use `?fit.keras.engine.training.Model`.

We are displaying the output for the first 5 epochs only

```
 #| class-output: "scrollable-output"


history <- modnn %>% fit(
  x[-testid, ], y[-testid], epochs = 600, batch_size = 32,
  validation_data = list(x[testid, ], y[testid]),
  verbose = 0 # This is the key to suppress the full output
)

# Now, display only the first few epochs of the training history
# The history object contains a 'metrics' dataframe
cat("Displaying metrics for the first 5 epochs:\n")
```

```
Displaying metrics for the first 5 epochs:
```

```r
print(head(history$metrics, 5))
```

```
$loss
  [1] 457426.9 457105.0 456786.8 456507.2 456164.6 455883.5 455677.2 455444.8
  [9] 455021.0 454583.6 454438.2 454124.6 453982.8 453647.0 453405.6 453123.2
 [17] 452733.8 452468.0 451999.3 451541.2 451483.3 450967.4 450368.2 450283.4
 [25] 449722.0 449371.9 449302.2 448575.5 448259.0 447704.5 447101.9 446740.8
 [33] 446380.2 445777.5 445092.5 444586.4 444674.1 444272.7 443211.7 442978.8
 [41] 441794.7 441127.4 440759.3 440440.8 440146.9 438796.5 439457.9 439046.3
 [49] 437384.5 435950.5 435702.5 434709.5 434076.0 432933.7 433488.6 432198.3
 [57] 430819.9 430820.3 430093.3 429173.7 428054.1 426866.4 426669.4 425879.1
 [65] 424330.6 424150.8 423821.5 421607.0 420418.5 422200.2 419415.8 417883.1
 [73] 415862.9 417524.0 413220.2 412619.3 411873.5 413343.1 409380.0 409763.8
 [81] 408257.3 408472.3 407041.5 404289.4 404480.3 402384.1 400672.4 399163.9
 [89] 398147.3 395974.5 396741.6 393372.7 393258.2 390437.0 391929.2 387796.5
 [97] 388568.8 386365.1 385637.3 385566.7 383500.4 383859.1 380469.5 379537.9
[105] 377373.3 375897.7 375280.1 373541.8 375571.7 367490.5 369319.9 367397.9
[113] 366212.2 367213.3 364876.5 363669.4 362097.1 358350.0 353643.0 355791.8
[121] 354806.8 351230.6 353866.7 350635.0 347844.0 346897.8 347240.6 346653.0
[129] 344897.7 339681.9 338875.7 341677.5 337162.2 339987.9 333909.7 332672.4
[137] 328426.5 326055.8 326524.3 327401.9 321181.3 319839.5 325381.9 318292.5
[145] 318585.0 316735.9 313082.4 310881.7 310620.8 305907.1 306991.2 308472.6
[153] 296781.7 305072.9 296307.2 297945.4 299010.8 297987.0 291994.1 287971.6
[161] 292664.5 289986.5 286204.0 291253.5 287622.2 286047.1 280703.4 280659.3
[169] 278704.3 278460.1 273736.2 272423.0 275460.2 275921.5 272304.0 268518.8
[177] 270236.6 265567.7 263063.4 263054.7 261574.3 261872.9 256483.6 255904.4
[185] 252006.0 248885.1 255869.1 251631.2 247429.9 250898.2 243882.7 247846.8
[193] 247600.7 240379.9 245681.9 240835.1 238705.3 235482.2 239482.5 235335.5
[201] 232181.4 232782.3 233254.5 230783.9 229222.3 231600.8 219747.0 228739.8
[209] 227040.0 222399.2 225139.0 219685.4 215423.5 219985.0 217396.1 217701.8
[217] 209205.8 216674.8 211767.6 207810.8 213481.5 205091.8 206243.4 208095.2
[225] 207267.9 202387.0 204287.9 199027.5 202789.2 200056.0 199123.6 207745.8
[233] 201853.3 200740.1 195610.1 191162.6 193315.8 191815.8 185667.5 197832.7
[241] 189703.0 197717.5 192905.0 195380.6 192260.6 193230.9 189478.1 188921.8
[249] 188222.3 188412.1 180417.8 178606.5 183496.7 197779.4 184614.1 183261.5
[257] 182323.7 185008.4 183240.2 176364.3 168912.4 178523.7 168674.0 173693.1
[265] 183511.8 181090.8 182740.9 177606.1 176268.5 169765.8 176650.3 181828.6
[273] 169633.5 171456.4 176988.9 168967.1 173769.3 166897.1 165794.1 172766.2
[281] 169383.4 172562.3 165261.4 160498.6 168462.1 162581.8 161980.9 162404.4
[289] 159002.3 169730.1 156200.3 166040.2 161691.9 166615.6 168182.7 159617.9
[297] 161367.1 164163.9 174715.6 173318.4 168016.8 157402.9 158569.7 160344.0
[305] 155677.0 168190.8 167143.7 159047.0 156433.1 157460.4 153673.9 155516.4
```

```
[313] 165089.8 158299.0 157765.3 161909.5 153533.7 151508.6 160147.0 164586.3
[321] 157089.7 151643.8 163139.4 154033.0 156232.1 154587.7 151703.7 152962.0
[329] 155383.0 154476.8 161977.7 159580.4 154266.7 142890.4 144598.7 154256.7
[337] 139739.5 154085.6 147342.9 152130.1 156936.9 157297.8 145576.8 151426.3
[345] 147671.8 154692.4 139056.9 146580.7 150290.1 148221.2 150202.2 155278.6
[353] 156004.4 132648.1 146735.1 149363.9 134468.5 150339.7 148378.8 144492.9
[361] 141805.2 142802.4 140928.3 153909.6 144823.0 145269.2 142226.5 146694.9
[369] 151427.3 140400.8 139108.0 139913.4 140836.8 145547.3 131106.1 131131.2
[377] 144066.6 147556.6 143767.7 144947.7 142830.4 147192.7 136878.0 144853.8
[385] 131318.4 138563.6 145636.4 143009.5 149687.8 139818.0 144708.2 142788.4
[393] 143840.5 142265.0 142676.2 146469.4 139368.8 140640.3 131961.7 137874.8
[401] 134831.3 138737.5 135504.8 131756.3 154292.1 148542.8 132040.3 134158.4
[409] 135395.4 135863.2 145613.4 138735.8 130494.4 141065.3 139816.0 133484.0
[417] 148208.6 143835.0 137097.4 146936.2 140316.0 145018.7 139785.5 134470.5
[425] 135619.6 139591.7 135372.3 141695.0 154011.6 135802.5 141990.6 134946.1
[433] 140660.8 133059.6 128540.2 142021.3 138820.9 131555.6 128468.1 141849.0
[441] 131774.9 134855.7 141965.5 134853.1 130445.2 138264.3 128491.3 124970.6
[449] 127016.2 125905.8 146083.2 132322.9 134512.0 130461.6 131318.8 134718.9
[457] 134520.0 135798.9 135538.2 136850.9 124009.4 118850.3 138992.1 128370.1
[465] 144988.2 127782.1 141435.9 140469.0 137141.4 135922.8 134802.9 132647.4
[473] 136087.8 141258.2 136449.5 129301.4 128254.7 133687.7 131045.2 126298.8
[481] 123529.8 128443.9 135927.8 130404.5 132117.9 123649.4 143044.2 136225.0
[489] 116067.4 118648.7 121727.0 120619.4 130895.7 128498.2 128767.8 123591.1
[497] 128663.5 125254.7 129618.1 118859.7 117110.2 129670.1 127824.9 121319.9
[505] 138875.1 136269.7 127243.3 129464.3 118448.8 126885.9 126038.4 134928.3
[513] 150565.3 126408.5 126602.9 128167.6 136712.4 122245.9 128004.3 127124.9
[521] 123442.5 149960.9 127073.0 126003.9 124322.6 129640.7 124337.5 137769.7
[529] 119271.4 130939.1 116777.6 127613.2 129726.9 126130.3 127990.1 125002.2
[537] 115933.0 119667.2 128412.2 125368.8 119235.5 111757.7 125422.2 143181.5
[545] 129986.5 125154.0 121973.4 117368.8 125002.2 118097.7 125282.7 120835.6
[553] 127143.2 130296.4 123255.0 125486.6 119532.6 121425.2 118278.9 125370.5
[561] 131434.3 120893.2 122862.9 129559.3 126082.5 115081.1 117263.7 124808.3
[569] 126547.5 111573.3 115401.8 111396.2 127996.9 125141.5 117714.1 117516.8
[577] 119762.1 127281.2 116414.5 123037.8 127671.2 114437.2 120135.4 112633.3
[585] 125617.4 120028.6 121268.9 128547.7 122050.0 123716.9 117464.0 123131.8
[593] 117854.6 122328.7 121186.3 132471.3 123069.8 118177.7 117712.6 119237.0

$mean_absolute_error
  [1] 534.0764 533.8375 533.6259 533.4158 533.1595 532.9612 532.8033 532.5934
  [9] 532.3209 532.1465 531.9408 531.7629 531.5975 531.3837 531.1519 531.0219
 [17] 530.7593 530.5785 530.1959 529.8079 529.7175 529.4205 529.0709 529.1391
 [25] 528.6569 528.4461 528.4084 527.8592 527.6586 527.2325 526.8318 526.6155
 [33] 526.2866 525.9366 525.6436 525.2006 525.0632 524.7968 524.2117 523.9957
```

```
 [41] 523.2493 522.8034 522.4621 522.5107 521.9797 521.1496 521.5699 521.2261
 [49] 520.0566 519.2003 518.7061 518.4019 517.8407 517.2855 517.4044 516.9612
 [57] 515.6985 515.5418 515.0053 514.3403 514.0087 513.1183 512.4984 512.1486
 [65] 511.1161 511.1242 510.8182 510.0217 508.4409 509.3468 507.5437 506.3014
 [73] 505.8159 505.8396 503.7534 503.2872 502.6010 502.8631 501.2284 500.9111
 [81] 500.0283 499.4632 498.8720 496.9669 497.3326 495.9336 494.3522 493.5821
 [89] 492.7046 491.9189 491.5581 489.9324 489.2643 487.1202 488.0879 484.6119
 [97] 484.6284 483.3521 483.8942 483.3155 482.2548 481.4937 479.6472 479.3014
[105] 477.5486 476.0214 474.6928 474.0528 475.8712 470.5245 472.0078 469.3866
[113] 468.3789 468.4299 467.2626 465.9980 464.8028 463.2331 461.0399 461.8090
[121] 460.4215 457.8720 457.9518 456.1375 455.5193 453.5052 453.8017 454.6745
[129] 452.9416 449.3977 448.3670 447.3305 446.3047 448.0367 442.6850 441.4679
[137] 439.5703 437.7251 436.4525 438.1873 433.8329 433.4948 435.4602 430.8208
[145] 431.0534 430.9062 427.2038 426.6747 425.1649 424.9293 422.8479 422.5788
[153] 418.0280 419.2631 415.1066 416.5985 417.7910 414.9089 410.0288 405.7748
[161] 408.8218 407.1146 405.2621 405.5876 405.0904 403.4519 400.1825 401.1215
[169] 399.6678 398.1116 394.4839 392.9328 395.9058 391.7146 390.0295 388.8638
[177] 388.5736 387.6448 386.4353 384.0442 382.0762 383.8100 377.0071 375.2611
[185] 375.9980 370.5583 375.8903 374.0382 370.4830 372.9235 366.0027 367.2145
[193] 368.3841 362.9835 365.8818 363.9456 363.1423 362.7086 361.6967 356.2124
[201] 355.9682 356.7356 357.2177 353.3253 354.9816 351.4104 346.7877 350.9695
[209] 350.6838 348.0096 352.6464 343.3067 341.7938 345.4630 343.0324 339.5841
[217] 333.8775 342.7906 335.2445 338.2388 337.1299 335.9468 332.8345 335.6761
[225] 334.4158 329.9001 336.7921 326.9107 330.7974 329.1034 327.7060 333.2936
[233] 326.9941 327.7560 322.5524 318.1271 321.7742 322.2395 316.2229 327.8273
[241] 320.6897 322.7473 325.5179 321.1801 320.4884 323.8885 320.3700 315.9822
[249] 317.5717 317.2804 311.2188 308.9698 316.3600 324.5518 313.6835 316.2126
[257] 309.9879 318.8824 308.4511 309.1116 303.3874 310.6136 301.5703 304.4984
[265] 315.4140 314.7350 310.9785 313.2262 308.3032 306.2030 304.0384 311.6614
[273] 299.9759 301.8875 307.6259 304.5699 307.0880 300.3018 303.9969 306.3234
[281] 305.5483 301.6530 299.2872 295.0905 303.1748 298.0484 292.4889 292.4941
[289] 298.4986 303.6788 292.8255 297.9537 299.5232 303.3006 302.1539 298.2629
[297] 292.4982 293.1672 306.6802 309.4366 298.7049 296.1880 290.8538 294.3400
[305] 291.4016 305.4348 294.3860 292.2815 292.8140 294.6284 290.0042 291.2739
[313] 300.4004 291.5100 297.9059 305.5119 298.1767 288.5288 294.6046 298.1508
[321] 289.8538 287.1805 297.6087 290.7987 295.7070 293.6728 292.3849 288.4983
[329] 288.8514 294.3996 297.4393 301.9040 292.6550 280.8357 285.8022 292.9222
[337] 278.0804 287.2150 289.4912 287.9710 289.3495 290.5872 280.1661 286.4594
[345] 291.0176 291.4508 277.3537 288.7073 287.3499 282.2365 289.0565 289.6556
[353] 292.3675 276.1229 289.1843 288.0238 274.6543 289.4128 289.5840 284.7433
[361] 280.9196 281.3486 278.8032 290.4746 285.0428 286.7723 283.2708 286.9196
[369] 289.2824 282.1638 277.4463 276.5322 287.4185 284.6328 271.5266 276.3943
[377] 285.7389 291.7892 279.6536 287.6373 285.8135 285.2011 280.0472 290.9365
```

```
[385] 274.4446 280.6970 289.1635 281.7610 281.7751 276.9247 283.5603 279.6413
[393] 287.7726 287.1505 287.0679 283.5865 276.2893 287.2442 273.3459 281.4748
[401] 274.1863 277.1289 273.1158 278.9325 286.1258 288.8821 274.9549 277.2747
[409] 278.3159 278.6355 286.1546 283.3498 275.2108 287.2569 284.8154 276.0953
[417] 291.3596 280.9917 284.0516 285.2537 283.7782 285.8843 283.1956 279.2600
[425] 278.2707 286.6525 278.5099 279.6276 287.6781 277.2738 286.6077 282.6704
[433] 281.4072 277.8648 271.1628 284.2177 279.6729 280.1224 279.1689 282.5355
[441] 277.4952 271.8465 285.4475 283.4992 278.3719 277.6978 276.0013 262.2294
[449] 271.4271 269.2167 284.9672 277.0579 279.5096 277.8329 274.2219 279.9214
[457] 276.0135 278.3332 280.1437 278.0380 272.9134 261.8793 282.9756 267.2744
[465] 282.5146 277.5072 289.3059 290.1453 279.7015 279.3384 277.7360 280.9148
[473] 280.8338 284.3850 278.0607 275.3495 276.7013 279.2299 271.9633 275.3681
[481] 262.8353 276.5849 281.2008 276.5038 281.4617 271.7160 282.9564 280.5397
[489] 263.5309 265.8843 269.7831 268.7096 271.8011 269.8916 272.1755 265.6997
[497] 274.0224 263.5212 272.8125 267.1635 261.4513 277.5412 274.8162 264.4865
[505] 279.4555 279.3780 266.9396 273.5645 258.6902 271.3578 276.4441 284.2550
[513] 291.2151 275.8195 271.9153 270.1165 281.1704 272.5388 273.8737 261.9825
[521] 276.0764 282.3683 268.4767 268.3818 261.7008 272.1948 263.0056 277.8087
[529] 267.4514 270.5582 260.1375 271.3654 260.6216 272.2541 265.9941 264.8153
[537] 264.3142 265.3331 273.0323 270.9555 262.3951 256.3882 267.2043 278.4402
[545] 280.5172 276.7117 265.3564 263.7223 266.1082 264.0421 264.5583 261.4802
[553] 273.8542 277.0399 265.9807 267.2198 264.9817 267.0056 260.9408 265.8924
[561] 274.1902 266.0485 263.1012 269.5190 274.6228 258.8459 262.8877 268.4784
[569] 266.0708 256.3053 263.7083 255.2731 262.0892 271.9853 261.1222 262.5635
[577] 266.0603 269.8400 260.3029 267.1953 273.2884 262.9211 261.7530 258.3799
[585] 273.1089 264.5924 262.8964 270.4568 264.2060 266.8306 259.0047 262.4690
[593] 259.2670 269.9737 267.2032 274.8774 262.6718 261.4230 256.8223 261.4754

$val_loss
  [1] 556449.8 556087.8 555759.1 555454.2 555162.4 554863.9 554576.7 554272.5
  [9] 553939.8 553596.8 553291.2 552971.4 552626.9 552286.5 551923.5 551577.6
 [17] 551195.1 550834.4 550429.2 550009.0 549601.8 549217.1 548757.9 548334.1
 [25] 547876.4 547415.0 546935.7 546424.9 545887.8 545353.3 544795.1 544273.9
 [33] 543701.3 543131.6 542510.1 541867.6 541299.1 540679.8 539999.8 539317.2
 [41] 538592.9 537912.2 537187.1 536426.9 535656.8 534892.4 534136.1 533337.2
 [49] 532449.8 531572.6 530674.5 529757.9 528845.0 527880.0 526913.5 525972.8
 [57] 525055.4 524069.2 523041.2 522045.2 520987.9 519861.8 518738.0 517531.5
 [65] 516446.4 515263.8 514093.5 512878.2 511725.9 510634.7 509363.7 508032.8
 [73] 506759.6 505473.4 504164.4 502917.4 501516.5 500119.7 498763.7 497428.3
 [81] 495971.2 494566.0 493038.2 491569.8 490099.1 488610.2 486963.5 485436.2
 [89] 483915.2 482343.3 480801.7 479067.9 477530.6 475846.4 474261.7 472665.1
 [97] 470989.7 469307.7 467694.8 465913.4 464230.9 462553.5 460837.2 458974.2
[105] 457127.4 455306.7 453523.9 451754.2 449953.6 448177.1 446408.4 444587.7
```

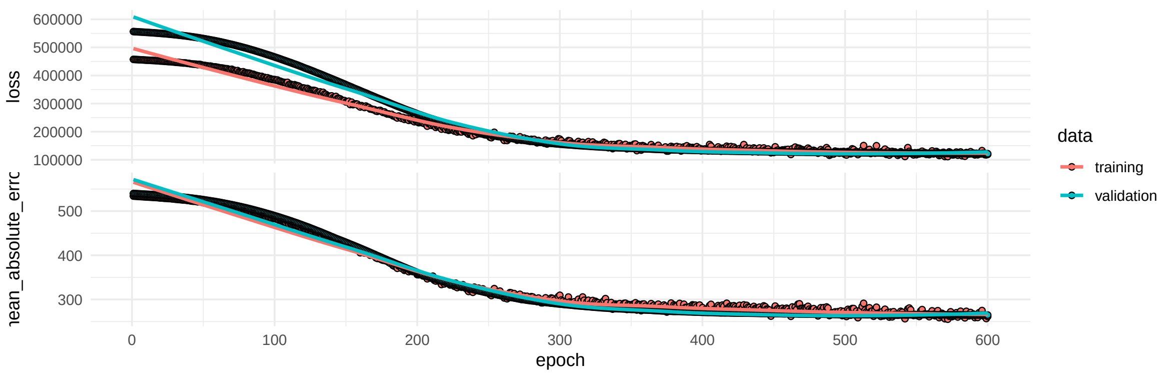
```
[113] 442591.0 440885.1 439070.5 437182.8 435173.2 433157.4 431203.6 429106.7
[121] 426991.2 425157.4 423230.6 421373.8 419318.6 417324.4 415390.3 413351.4
[129] 411248.1 409186.4 407245.0 405303.0 403274.6 401092.0 399015.2 396912.6
[137] 394719.7 392770.6 390673.4 388569.6 386550.1 384435.2 382276.3 380147.9
[145] 377957.4 375940.9 373855.7 371659.6 369563.4 367323.7 365061.7 362988.3
[153] 360676.0 358713.6 356371.2 354230.7 352120.5 350138.0 348004.3 345894.0
[161] 343582.2 341538.5 339480.8 337397.2 335273.4 333035.8 330781.4 328713.1
[169] 326558.2 324512.5 322269.6 320083.0 318070.2 316028.0 313933.2 311961.8
[177] 309859.5 307641.3 305573.3 303346.2 301247.2 299196.3 297186.7 295021.6
[185] 292987.5 290767.8 288967.7 286969.8 284956.5 283020.9 281007.0 279012.8
[193] 276991.0 274946.3 273253.4 271387.0 269447.1 267620.8 265630.4 263543.5
[201] 261632.2 259831.1 258134.6 256336.0 254621.5 252906.6 250941.1 249402.1
[209] 247672.0 246019.2 244505.4 242860.5 241284.5 239614.5 237969.2 236416.2
[217] 234563.9 233086.7 231487.4 229800.6 228130.2 226561.1 224875.3 223559.4
[225] 221955.9 220419.7 219070.4 217648.8 216205.6 214918.2 213415.5 212023.4
[233] 210605.5 209580.6 208447.5 207255.0 205922.8 204654.5 203306.5 202089.1
[241] 200770.2 199512.0 198434.1 197434.2 196321.0 195052.9 193994.0 192912.8
[249] 192074.4 190984.7 190145.9 189060.4 188063.8 187204.7 185956.8 184888.9
[257] 183846.7 183002.6 182210.1 181307.6 180446.5 179503.2 178775.2 177678.8
[265] 176856.8 176080.7 175390.3 174506.7 173870.3 172884.0 172018.5 171385.1
[273] 170676.6 170058.8 169356.9 168554.8 167824.3 167284.7 166630.8 166031.7
[281] 165527.0 165158.5 164645.1 163929.4 163423.1 162932.4 162398.9 161987.4
[289] 161325.3 160755.2 160128.9 159577.1 158972.6 158465.1 158106.5 157595.4
[297] 157128.0 156521.6 156156.2 155696.4 155373.8 154975.3 154484.9 154111.2
[305] 153764.2 153394.5 153130.9 152756.8 152310.7 151956.6 151457.6 151061.5
[313] 150733.4 150367.8 150049.1 149807.0 149397.4 149056.0 148771.9 148515.7
[321] 148146.0 147724.8 147401.2 147067.7 146788.3 146584.3 146219.0 145908.0
[329] 145538.4 145306.4 145080.1 144943.1 144656.6 144380.8 144050.4 143852.5
[337] 143515.8 143350.2 143202.3 143013.1 142729.8 142697.1 142437.8 142305.4
[345] 142092.8 141848.9 141586.3 141225.3 140933.1 140733.2 140624.5 140662.4
[353] 140441.5 140079.5 139969.2 139723.8 139490.6 139230.7 139135.5 138835.3
[361] 138787.4 138592.3 138403.1 138203.5 137872.9 137628.3 137452.0 137266.5
[369] 137069.1 136943.4 136661.3 136473.7 136360.7 136228.3 136041.5 135900.2
[377] 135681.0 135609.4 135350.8 135209.2 135159.9 135084.7 135054.1 135041.0
[385] 134801.3 134659.3 134530.5 134346.9 134214.0 134067.7 134018.0 133923.2
[393] 133684.4 133526.8 133423.4 133389.2 133267.5 133104.5 132915.7 132769.7
[401] 132649.2 132552.8 132428.1 132277.3 132247.0 132303.1 132126.6 132002.3
[409] 131814.6 131623.8 131566.4 131341.5 131148.2 131222.5 131136.2 131047.1
[417] 131061.6 130998.2 130845.2 130717.0 130714.4 130590.5 130492.7 130408.2
[425] 130282.7 130214.6 130188.2 130042.3 130098.2 130013.6 129980.0 129879.5
[433] 129764.9 129678.1 129599.3 129478.6 129390.1 129325.0 129182.8 129102.0
[441] 128928.8 128869.8 128869.9 128745.4 128783.0 128609.6 128490.6 128438.6
[449] 128338.8 128310.8 128267.2 128258.2 128234.6 128090.5 128007.7 127894.3
```

```
[457] 127755.4 127680.9 127780.1 127645.5 127512.2 127315.2 127349.6 127237.4
[465] 127344.6 127220.6 127115.0 127006.4 127059.0 127011.1 126956.8 126820.5
[473] 126657.5 126606.3 126619.6 126553.9 126543.7 126435.5 126335.7 126180.1
[481] 126152.6 126163.6 126223.4 126250.1 126234.2 126029.3 126118.8 126017.7
[489] 125940.4 125798.0 125684.7 125562.6 125454.4 125460.1 125323.6 125194.1
[497] 125150.3 125090.4 125151.9 125136.5 125129.9 125093.0 125019.5 124887.3
[505] 124940.0 124932.1 124910.7 124871.7 124791.8 124682.5 124730.3 124857.1
[513] 124803.4 124775.3 124754.3 124648.6 124721.3 124746.4 124650.8 124557.3
[521] 124484.2 124543.5 124409.7 124302.0 124135.4 123969.2 123843.7 123801.9
[529] 123783.7 123723.2 123562.8 123561.7 123557.0 123565.7 123720.6 123573.0
[537] 123678.7 123583.9 123603.6 123627.9 123475.9 123369.4 123242.7 123372.3
[545] 123375.4 123387.3 123383.6 123339.5 123290.1 123218.4 123258.8 123179.3
[553] 123083.5 123222.2 123282.3 123253.7 123191.5 123171.3 123180.4 123119.3
[561] 123189.0 123092.7 123057.8 122963.8 122917.5 122724.8 122596.0 122643.3
[569] 122598.8 122554.6 122388.5 122246.9 122299.2 122282.7 122177.6 122197.3
[577] 122215.0 122136.4 122143.4 122079.0 122024.6 121947.7 121939.8 121829.3
[585] 121848.9 121830.5 121769.4 121589.3 121638.8 121642.1 121575.6 121643.8
[593] 121633.8 121522.4 121467.1 121592.4 121546.7 121534.2 121480.1 121558.7

$val_mean_absolute_error
  [1] 539.9470 539.7190 539.5236 539.3342 539.1504 538.9667 538.7886 538.6054
  [9] 538.4133 538.2114 538.0250 537.8312 537.6299 537.4260 537.2171 537.0076
 [17] 536.7813 536.5668 536.3242 536.0822 535.8461 535.6136 535.3456 535.1091
 [25] 534.8479 534.5803 534.2993 534.0182 533.7239 533.4286 533.1196 532.8234
 [33] 532.5106 532.1818 531.8400 531.5023 531.1913 530.8494 530.4863 530.1086
 [41] 529.7189 529.3444 528.9497 528.5450 528.1354 527.7224 527.3056 526.8705
 [49] 526.4058 525.9293 525.4493 524.9558 524.4636 523.9366 523.4224 522.9127
 [57] 522.4162 521.8827 521.3314 520.7880 520.2180 519.6064 518.9950 518.3515
 [65] 517.7565 517.1214 516.4908 515.8301 515.1998 514.5920 513.9047 513.1866
 [73] 512.4788 511.7621 511.0631 510.3825 509.6124 508.8633 508.1231 507.3925
 [81] 506.6036 505.8234 504.9840 504.1678 503.3584 502.5331 501.6274 500.7715
 [89] 499.9102 499.0311 498.1322 497.1871 496.3152 495.3817 494.4769 493.5664
 [97] 492.6012 491.6602 490.7300 489.7258 488.7710 487.7834 486.7777 485.7166
[105] 484.6590 483.5938 482.5581 481.5107 480.4618 479.4105 478.3305 477.2569
[113] 476.0786 475.0540 473.9636 472.8244 471.5891 470.4011 469.2293 467.9845
[121] 466.7312 465.5981 464.4235 463.2542 461.9846 460.7548 459.5362 458.2672
[129] 456.9466 455.6409 454.3973 453.1612 451.8569 450.4851 449.1383 447.7900
[137] 446.3901 445.0991 443.7114 442.3546 440.9802 439.6046 438.2090 437.0017
[145] 435.7421 434.5586 433.3191 432.0218 430.7587 429.4349 428.0874 426.7949
[153] 425.4155 424.2183 422.8062 421.4704 420.1593 418.8857 417.5527 416.2139
[161] 414.7849 413.4454 412.1006 410.7436 409.3410 407.9026 406.4498 405.0917
[169] 403.6064 402.2572 400.7603 399.2704 397.9233 396.5550 395.1337 393.7345
[177] 392.3019 390.8316 389.4849 388.0273 386.6653 385.2851 383.9187 382.4768
```

```
[185] 381.1171 379.6411 378.4127 377.0693 375.6914 374.3682 373.0026 371.6696
[193] 370.2953 368.8775 367.7049 366.3896 365.0147 363.6983 362.2929 360.7855
[201] 359.4125 358.0920 356.9308 355.7127 354.4934 353.2710 351.8946 350.7837
[209] 349.7308 348.6873 347.6989 346.6665 345.6412 344.5883 343.5709 342.5846
[217] 341.4670 340.5974 339.6722 338.6964 337.7293 336.7741 335.8461 335.1040
[225] 334.2188 333.3197 332.5230 331.6939 330.8321 330.0539 329.1616 328.3044
[233] 327.4462 326.7636 326.0137 325.2471 324.3499 323.5370 322.6748 321.9075
[241] 321.0621 320.2471 319.5459 318.8987 318.1683 317.3547 316.6483 315.9567
[249] 315.4133 314.7224 314.1750 313.4741 312.8131 312.2213 311.4021 310.6936
[257] 309.9707 309.3788 308.8235 308.1890 307.5857 306.9089 306.3778 305.5883
[265] 305.0114 304.4591 303.9716 303.3085 302.8325 302.0769 301.4287 300.9644
[273] 300.4222 299.9517 299.4511 298.8935 298.3680 297.9582 297.5022 297.0980
[281] 296.7522 296.4997 296.1306 295.6426 295.2997 294.9742 294.5987 294.3048
[289] 293.8349 293.4287 292.9626 292.5603 292.1191 291.7492 291.4778 291.0890
[297] 290.7375 290.2724 289.9873 289.6222 289.3673 289.0475 288.6767 288.3842
[305] 288.1110 287.8101 287.5734 287.2533 286.8702 286.5910 286.1841 285.8679
[313] 285.5999 285.3171 285.0579 284.8414 284.5077 284.2299 284.0024 283.7856
[321] 283.4768 283.1378 282.8516 282.5720 282.3397 282.1631 281.8461 281.5656
[329] 281.2287 281.0219 280.8187 280.7033 280.4576 280.2267 279.9584 279.7861
[337] 279.5061 279.3799 279.2950 279.1820 278.9902 278.9936 278.8181 278.7091
[345] 278.5522 278.3900 278.2055 277.9212 277.7184 277.5529 277.4925 277.5777
[353] 277.4050 277.1100 277.0473 276.8395 276.6613 276.4648 276.4129 276.1289
[361] 276.1094 275.9584 275.8050 275.6649 275.3463 275.1138 274.9456 274.7969
[369] 274.5923 274.4944 274.2475 274.0649 273.9735 273.8692 273.7116 273.5949
[377] 273.3813 273.3102 273.0784 272.9464 272.9448 272.9634 272.9539 272.9683
[385] 272.7238 272.5655 272.4376 272.2708 272.1526 272.0127 271.9943 271.9337
[393] 271.6952 271.5109 271.4381 271.4058 271.2762 271.1691 271.0326 270.9224
[401] 270.8257 270.7755 270.6865 270.5755 270.5742 270.6368 270.4948 270.4041
[409] 270.2370 270.0355 269.9928 269.7867 269.6251 269.7242 269.6566 269.5878
[417] 269.6414 269.6543 269.5203 269.3818 269.3795 269.2853 269.2271 269.1961
[425] 269.1086 269.0837 269.0886 268.9747 269.0554 268.9866 268.9895 268.9607
[433] 268.8954 268.8562 268.8139 268.7382 268.6788 268.6505 268.5357 268.4736
[441] 268.3223 268.3304 268.3260 268.2429 268.3147 268.1378 268.0102 267.9930
[449] 267.9460 267.9500 267.9122 267.9316 267.9399 267.8532 267.7990 267.7125
[457] 267.5865 267.5678 267.6919 267.5832 267.5006 267.3080 267.3685 267.2758
[465] 267.4198 267.2948 267.1583 267.0379 267.1183 267.0644 267.0388 266.9339
[473] 266.7640 266.6998 266.7600 266.7343 266.7364 266.6614 266.5809 266.4905
[481] 266.5025 266.5518 266.6162 266.6579 266.6534 266.4459 266.5578 266.4366
[489] 266.4347 266.2906 266.2181 266.0744 265.9648 265.9904 265.8698 265.7671
[497] 265.7665 265.7391 265.8148 265.8721 265.8568 265.8434 265.7574 265.6187
[505] 265.6631 265.6507 265.6832 265.6015 265.5605 265.4738 265.4760 265.6125
[513] 265.5535 265.5275 265.5101 265.4192 265.5207 265.5650 265.5000 265.4937
[521] 265.4660 265.4786 265.3587 265.2369 265.0716 264.9437 264.8354 264.7720
```

```
[529] 264.8063 264.7317 264.5803 264.6138 264.6480 264.6871 264.9313 264.7922
[537] 265.0094 264.9785 265.0499 265.1317 265.0122 264.9730 264.8896 265.0932
[545] 265.1324 265.2336 265.2838 265.2748 265.2625 265.2232 265.3098 265.2521
[553] 265.1587 265.3680 265.5230 265.5610 265.4940 265.5166 265.6241 265.6012
[561] 265.6837 265.6118 265.5974 265.5117 265.4773 265.3041 265.2153 265.2992
[569] 265.2800 265.2569 265.1122 264.9694 265.0792 265.0840 265.0035 265.0578
[577] 265.0897 265.0441 265.0674 265.0205 264.9562 264.9101 264.9403 264.8134
[585] 264.8647 264.8852 264.8345 264.5805 264.6825 264.6544 264.5813 264.6667
[593] 264.7094 264.5962 264.5455 264.7209 264.6793 264.6786 264.6299 264.7164
```

```
# You might also want to plot the history later
 plot(history)
```



It is worth noting that if you run the `fit()` command a second time in the same R session, then the fitting process will pick up where it left off. Try re-running the `fit()` command, and then the `plot()` command, to see!

Finally, we predict from the final model, and evaluate its performance on the test data. Due to the use of SGD, the results vary slightly with each fit. Unfortunately the `set.seed()` function does not ensure identical results (since the fitting is done in `python`), so your results will differ slightly.

```
npred <- predict(modnn, x[testid, ])
```

```
3/3 - 0s - 69ms/epoch - 23ms/step
```

```
mean(abs(y[testid] - npred))
```

```
[1] 264.7164
```

In this case our results are a bit worse than with the `gmln`

A Multilayer Network on the MNIST Digit Data

The `keras` package comes with a number of example datasets, including the `MNIST` digit data. Our first step is to load the `MNIST` data. The `dataset_mnist()` function is provided for this purpose.

```
mnist <- dataset_mnist() #load the dataset from Keras package
x_train <- mnist$train$x
g_train <- mnist$train$y
x_test <- mnist$test$x
g_test <- mnist$test$y
dim(x_train)
```

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

```
dim(x_test)
```

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

There are 60,000 images in the training data and 10,000 in the test data. The images are 28×28, and stored as a three-dimensional array, so we need to reshape them into a matrix. Also, we need to "one-hot" encode the class label. Luckily `keras` has a lot of built-in functions that do this for us.

```
#create matrix form
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
#change the response to categorical
y_train <- to_categorical(g_train, 10)
y_test <- to_categorical(g_test, 10)
```

Neural networks are somewhat sensitive to the scale of the inputs. For example, ridge and lasso regularization are affected by scaling. Here the inputs are eight-bit grayscale values between 0 and 255, so we rescale to the unit interval. (Eight bits means 2^8, which equals 256. Since the convention is to start at 0, the possible values range from 0 to 255.)

```
#rescale the x values between 0 and 1
x_train <- x_train / 255
x_test <- x_test / 255
```

Now we are ready to fit our neural network.

```
modelnn <- keras_model_sequential()
modelnn %>%
  layer_dense(units = 256, activation = "relu",
       input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")
```

The first layer goes from $28 \times 28 = 784$ input units to a hidden layer of 256 units, which uses the ReLU activation function. This is specified by a call to `layer_dense()`, which takes as input a `modelnn` object, and returns a modified `modelnn` object. This is then piped through `layer_dropout()` to perform dropout regularization. The second hidden layer comes next, with 128 hidden units, followed by a dropout layer. The final layer is the output layer, with activation `"softmax"` (10.13) for the 10-class classification problem, which defines the map from the second hidden layer to class probabilities. Finally, we use `summary()` to summarize the model, and to make sure we got it all right.

```
summary(modelnn)
```

```
Model: "sequential_2"
________________________________________________________________________________
 Layer (type)                       Output Shape                    Param #
================================================================================
 dense_6 (Dense)                    (None, 256)                     200960
 dropout_3 (Dropout)                (None, 256)                     0
 dense_5 (Dense)                    (None, 128)                     32896
 dropout_2 (Dropout)                (None, 128)                     0
 dense_4 (Dense)                    (None, 10)                      1290
================================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
________________________________________________________________________________
```

The parameters for each layer include a bias term, which results in a parameter count of 235,146. For example, the first hidden layer involves $(784 + 1) \times 256 = 200{,}960$ parameters.

Notice that the layer names such as `dropout_1` and `dense_2` have subscripts. These may appear somewhat random; in fact, if you fit the same model again, these will change. They are of no consequence: they vary because the model specification code is run in `python`, and these subscripts are incremented every time `keras_model_sequential()` is called.

Next, we add details to the model to specify the fitting algorithm. We fit the model by minimizing the cross-entropy function.

```
modelnn %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(),
    metrics = c("accuracy")
  )
```

Now we are ready to go. The final step is to supply training data, and fit the model.

```
system.time(
  history <- modelnn %>%
#     fit(x_train, y_train, epochs = 30, batch_size = 128,
      fit(x_train, y_train, epochs = 15, batch_size = 128,
        validation_split = 0.2,
      verbose = 0)
)
```

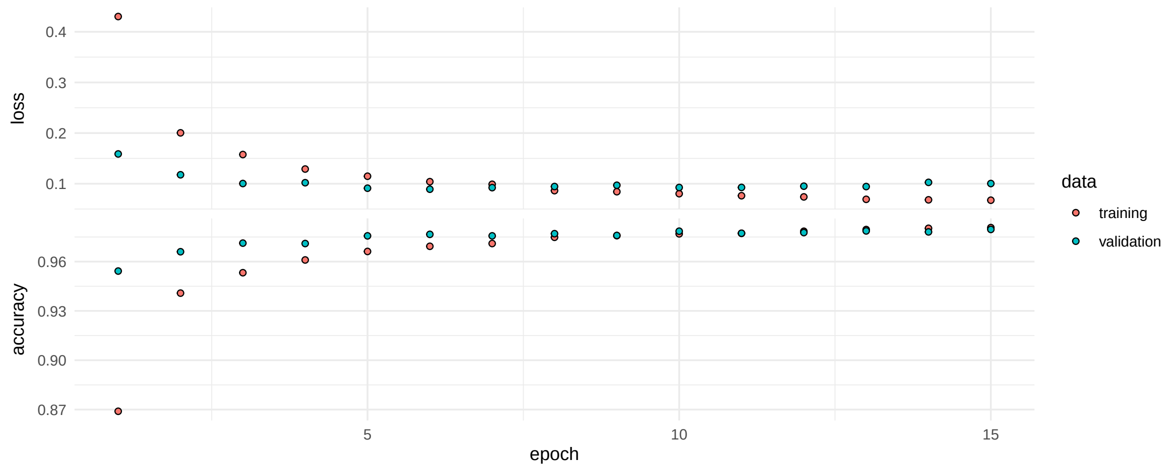
```
   user  system elapsed
  70.42   12.20   17.32
```

```
print(head(history$metrics, 5))
```

```
$loss
 [1] 0.43013632 0.20046210 0.15763985 0.12899542 0.11479534 0.10406683
 [7] 0.09889714 0.08630955 0.08424029 0.08033949 0.07629272 0.07399427
[13] 0.06919461 0.06818173 0.06744172

$accuracy
 [1] 0.8689375 0.9408333 0.9532500 0.9610000 0.9661875 0.9693125 0.9710000
 [8] 0.9747916 0.9757917 0.9768958 0.9772083 0.9784583 0.9795000 0.9802709
[15] 0.9806875

$val_loss
 [1] 0.15877740 0.11766846 0.10042538 0.10200878 0.09115166 0.08916025
 [7] 0.09227723 0.09446624 0.09696231 0.09255157 0.09267700 0.09514039
```

```
[13] 0.09439282 0.10268512 0.10038359

$val_accuracy
 [1] 0.9542500 0.9660000 0.9712500 0.9710000 0.9756666 0.9765834 0.9756666
 [8] 0.9770000 0.9759167 0.9785000 0.9772500 0.9776667 0.9786667 0.9780833
[15] 0.9795833
```

```r
plot(history, smooth = FALSE)
```



We have suppressed the output here, which is a progress report on the fitting of the model, grouped by epoch. This is very useful, since on large datasets fitting can take time. Here we specified a validation split of 20%, so the training is actually performed on 80% of the 60,000 observations in the training set. This is an alternative to actually supplying validation data. See `?fit.keras.engine.training.Model` for all the optional fitting arguments. SGD uses batches of 128 observations in computing the gradient, and doing the arithmetic, we see that an epoch corresponds to 375 gradient steps.

To obtain the test error, we first write a simple function `accuracy()` that compares predicted and true class labels, and then use it to evaluate our predictions.

```r
accuracy <- function(pred, truth)
  mean(drop(as.numeric(pred)) == drop(truth))
```

```r
modelnn %>%
  predict(x_test) %>%
  k_argmax() %>%
  accuracy(g_test)
```

The table also reports LDA and multiclass logistic regression. Although packages such as **glmnet** can handle multiclass logistic regression, they are quite slow on this large dataset. It is much faster and quite easy to fit such a model using the **keras** software.

We can do the model with just an input layer and output layer, and omit the hidden layers, and that will be like fitting a logistic regression:

```
modellr <- keras_model_sequential() %>%
  layer_dense(input_shape = 784, units = 10,
       activation = "softmax")
summary(modellr)
```

```
Model: "sequential_3"
________________________________________________________________________________
 Layer (type)                       Output Shape                     Param #
================================================================================
 dense_7 (Dense)                    (None, 10)                       7850
================================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
________________________________________________________________________________
```

We fit the model just as before.

```
modellr %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(),
    metrics = c("accuracy"))

modellr %>%
  fit(x_train, y_train, epochs = 30,
      batch_size = 128, validation_split = 0.2)
```

```
Epoch 1/30
375/375 - 1s - loss: 0.6660 - accuracy: 0.8329 - val_loss: 0.3591 - val_accuracy: 0.9038 - 93
Epoch 2/30
375/375 - 1s - loss: 0.3520 - accuracy: 0.9031 - val_loss: 0.3079 - val_accuracy: 0.9157 - 53
Epoch 3/30
375/375 - 1s - loss: 0.3155 - accuracy: 0.9130 - val_loss: 0.2893 - val_accuracy: 0.9207 - 51
Epoch 4/30
```

```
375/375 - 1s - loss: 0.2992 - accuracy: 0.9167 - val_loss: 0.2806 - val_accuracy: 0.9218 - 53
Epoch 5/30
375/375 - 1s - loss: 0.2893 - accuracy: 0.9191 - val_loss: 0.2788 - val_accuracy: 0.9228 - 53
Epoch 6/30
375/375 - 1s - loss: 0.2832 - accuracy: 0.9208 - val_loss: 0.2719 - val_accuracy: 0.9257 - 52
Epoch 7/30
375/375 - 1s - loss: 0.2782 - accuracy: 0.9218 - val_loss: 0.2693 - val_accuracy: 0.9265 - 52
Epoch 8/30
375/375 - 1s - loss: 0.2744 - accuracy: 0.9235 - val_loss: 0.2660 - val_accuracy: 0.9279 - 52
Epoch 9/30
375/375 - 1s - loss: 0.2714 - accuracy: 0.9248 - val_loss: 0.2654 - val_accuracy: 0.9285 - 51
Epoch 10/30
375/375 - 1s - loss: 0.2688 - accuracy: 0.9255 - val_loss: 0.2662 - val_accuracy: 0.9287 - 57
Epoch 11/30
375/375 - 1s - loss: 0.2663 - accuracy: 0.9261 - val_loss: 0.2637 - val_accuracy: 0.9298 - 57
Epoch 12/30
375/375 - 1s - loss: 0.2647 - accuracy: 0.9271 - val_loss: 0.2649 - val_accuracy: 0.9280 - 56
Epoch 13/30
375/375 - 1s - loss: 0.2626 - accuracy: 0.9274 - val_loss: 0.2620 - val_accuracy: 0.9303 - 58
Epoch 14/30
375/375 - 1s - loss: 0.2617 - accuracy: 0.9280 - val_loss: 0.2626 - val_accuracy: 0.9298 - 56
Epoch 15/30
375/375 - 1s - loss: 0.2601 - accuracy: 0.9281 - val_loss: 0.2616 - val_accuracy: 0.9304 - 57
Epoch 16/30
375/375 - 1s - loss: 0.2595 - accuracy: 0.9292 - val_loss: 0.2631 - val_accuracy: 0.9283 - 57
Epoch 17/30
375/375 - 1s - loss: 0.2582 - accuracy: 0.9288 - val_loss: 0.2600 - val_accuracy: 0.9302 - 55
Epoch 18/30
375/375 - 1s - loss: 0.2573 - accuracy: 0.9294 - val_loss: 0.2618 - val_accuracy: 0.9301 - 53
Epoch 19/30
375/375 - 1s - loss: 0.2560 - accuracy: 0.9302 - val_loss: 0.2643 - val_accuracy: 0.9294 - 54
Epoch 20/30
375/375 - 1s - loss: 0.2552 - accuracy: 0.9302 - val_loss: 0.2629 - val_accuracy: 0.9295 - 52
Epoch 21/30
375/375 - 1s - loss: 0.2548 - accuracy: 0.9307 - val_loss: 0.2613 - val_accuracy: 0.9310 - 52
Epoch 22/30
375/375 - 1s - loss: 0.2540 - accuracy: 0.9309 - val_loss: 0.2611 - val_accuracy: 0.9310 - 52
Epoch 23/30
375/375 - 1s - loss: 0.2536 - accuracy: 0.9309 - val_loss: 0.2613 - val_accuracy: 0.9313 - 53
Epoch 24/30
375/375 - 1s - loss: 0.2528 - accuracy: 0.9314 - val_loss: 0.2607 - val_accuracy: 0.9317 - 52
Epoch 25/30
```

```
375/375 - 1s - loss: 0.2524 - accuracy: 0.9314 - val_loss: 0.2631 - val_accuracy: 0.9306 - 52
Epoch 26/30
375/375 - 1s - loss: 0.2518 - accuracy: 0.9315 - val_loss: 0.2613 - val_accuracy: 0.9314 - 52
Epoch 27/30
375/375 - 1s - loss: 0.2510 - accuracy: 0.9320 - val_loss: 0.2621 - val_accuracy: 0.9312 - 52
Epoch 28/30
375/375 - 1s - loss: 0.2504 - accuracy: 0.9321 - val_loss: 0.2627 - val_accuracy: 0.9308 - 54
Epoch 29/30
375/375 - 1s - loss: 0.2499 - accuracy: 0.9314 - val_loss: 0.2629 - val_accuracy: 0.9312 - 64
Epoch 30/30
375/375 - 1s - loss: 0.2497 - accuracy: 0.9321 - val_loss: 0.2614 - val_accuracy: 0.9321 - 55
```

```
modellr %>%
  predict(x_test) %>%
  k_argmax() %>%
  accuracy(g_test)
```

```
313/313 - 0s - 197ms/epoch - 630us/step
```

```
[1] 0.9261
```

And we see that the model accuracy is smaller now. This will be a similar result as running `glmn` instead of `keras`.

Convolutional Neural Networks (CNN)

In this section we fit a CNN to the `CIFAR` data, which is available in the `keras` package. It is arranged in a similar fashion as the `MNIST` data.

```
cifar100 <- dataset_cifar100()
names(cifar100)
```

```
[1] "train" "test"
```

```
x_train <- cifar100$train$x
g_train <- cifar100$train$y
x_test <- cifar100$test$x
g_test <- cifar100$test$y
dim(x_train)
```

```
[1] 50000    32    32     3
```

```
range(x_train[1,,, 1])
```

```
[1]  13 255
```

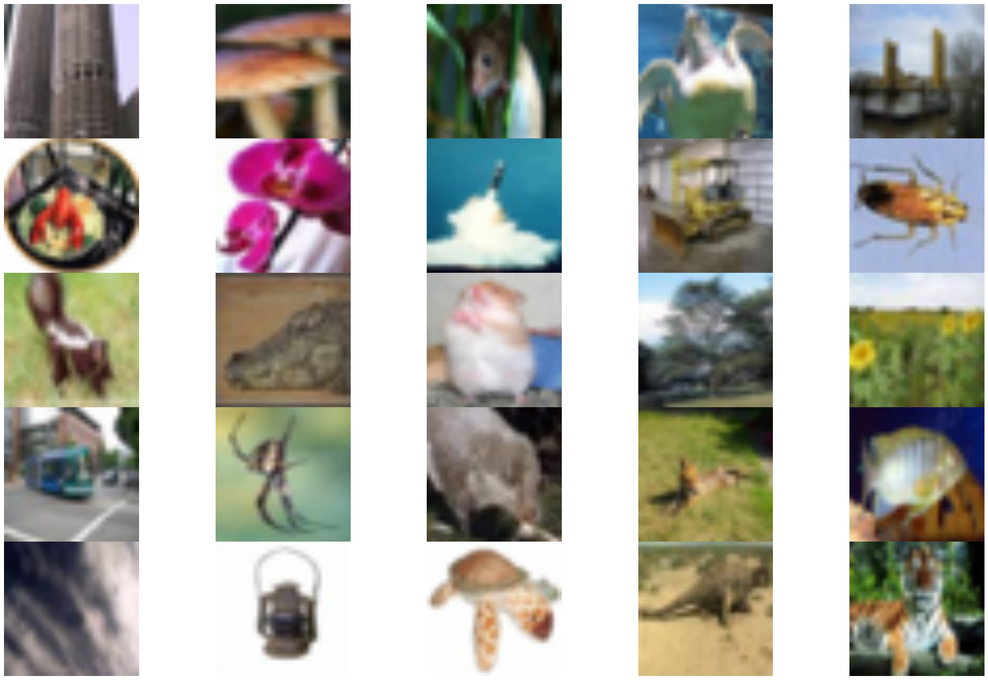
The array of 50,000 training images has four dimensions: each three-color image is represented as a set of three channels, each of which consists of 32×32 eight-bit pixels. We standardize as we did for the digits, but keep the array structure. We one-hot encode the response factors to produce a 100-column binary matrix.

```
x_train <- x_train / 255
x_test <- x_test / 255
y_train <- to_categorical(g_train, 100)
dim(y_train)
```

```
[1] 50000   100
```

Before we start, we look at some of the training images using the `jpeg` package; similar code produced Figure 10.5 on page 411.

```
library(jpeg)
par(mar = c(0, 0, 0, 0), mfrow = c(5, 5))
index <- sample(seq(50000), 25)
for (i in index) plot(as.raster(x_train[i,,, ]))
```



The `as.raster()` function converts the feature map so that it can be plotted as a color image.

Here we specify a moderately-sized CNN for demonstration purposes.

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3),
      padding = "same", activation = "relu",
      input_shape = c(32, 32, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 100, activation = "softmax")
summary(model)
```

```
Model: "sequential_4"
________________________________________________________________________________
 Layer (type)                       Output Shape                     Param #
================================================================================
 conv2d_3 (Conv2D)                  (None, 32, 32, 32)               896
 max_pooling2d_3 (MaxPooling2D)     (None, 16, 16, 32)               0
 conv2d_2 (Conv2D)                  (None, 16, 16, 64)               18496
 max_pooling2d_2 (MaxPooling2D)     (None, 8, 8, 64)                 0
 conv2d_1 (Conv2D)                  (None, 8, 8, 128)                73856
 max_pooling2d_1 (MaxPooling2D)     (None, 4, 4, 128)                0
 conv2d (Conv2D)                    (None, 4, 4, 256)                295168
 max_pooling2d (MaxPooling2D)       (None, 2, 2, 256)                0
 flatten (Flatten)                  (None, 1024)                     0
 dropout_4 (Dropout)                (None, 1024)                     0
 dense_9 (Dense)                    (None, 512)                      524800
 dense_8 (Dense)                    (None, 100)                      51300
================================================================================
Total params: 964,516
Trainable params: 964,516
Non-trainable params: 0
________________________________________________________________________________
```

Notice that we used the `padding = "same"` argument to `layer_conv_2D()`, which ensures that the output channels have the same dimension as the input channels. There are 32 channels in the first hidden layer, in contrast to the three channels in the input layer. We use a 3×3 convolution filter for each channel in all the layers. Each convolution is followed by a max-pooling layer over 2×2 blocks. By studying the summary, we can see that the channels halve in both dimensions after each of these max-pooling operations. After the last of these we have a layer with 256 channels of dimension 2×2. These are then flattened to a dense layer of size 1,024: in other words, each of the 2×2 matrices is turned into a 4-vector, and put side-by-side in one layer. This is followed by a dropout regularization layer, then another dense layer of size 512, which finally reaches the softmax output layer.

Finally, we specify the fitting algorithm, and fit the model.

```r
model %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(),
    metrics = c("accuracy"))
#history <- model %>% fit(x_train, y_train, epochs = 30,#this is better but takes too long
history <- model %>%
  fit(x_train, y_train, epochs = 10,
```

```
    batch_size = 128, validation_split = 0.2,
    verbose =0)
print(head(history$metrics, 5))
```

```
$loss
 [1] 4.132422 3.527766 3.171476 2.926289 2.731802 2.561423 2.401794 2.271233
 [9] 2.147576 2.029118

$accuracy
 [1] 0.063800 0.161025 0.226650 0.271250 0.305175 0.340675 0.376125 0.401250
 [9] 0.427025 0.454625

$val_loss
 [1] 3.720757 3.332056 3.153583 2.849142 2.966730 2.634250 2.534180 2.557547
 [9] 2.377460 2.325225

$val_accuracy
 [1] 0.1267 0.1908 0.2275 0.2866 0.2688 0.3348 0.3625 0.3580 0.3885 0.4000
```

```
model %>%
  predict(x_test) %>%
  k_argmax() %>%
  accuracy(g_test)
```

```
313/313 - 2s - 2s/epoch - 5ms/step
```

```
[1] 0.4099
```

This model takes 10 minutes to run and achieves 46% accuracy on the test data. Although this is not terrible for 100-class data (a random classifier gets 1% accuracy), searching the web we see results around 75%. Typically it takes a lot of architecture carpentry, fiddling with regularization, and time to achieve such results.

Using Pretrained CNN Models

We now show how to use a CNN pretrained on the `imagenet` database to classify natural images. We copied six jpeg images from a digital photo album into the directory `book_images`. (These images are available from the data section of <www.statlearning.com>, the ISL book website. Download **book_images.zip**; when clicked it creates the **book_images** directory.) We first read in the images, and convert them into the array format expected by the `keras` software to match the specifications in `imagenet`. Make sure that your working directory in `R` is set to the folder in which the images are stored.

```r
img_dir <- "book_images"
image_names <- list.files(img_dir)
num_images <- length(image_names)
x <- array(dim = c(num_images, 224, 224, 3))
for (i in 1:num_images) {
  img_path <- paste(img_dir, image_names[i], sep = "/")
  img <- image_load(img_path, target_size = c(224, 224))
  x[i,,, ] <- image_to_array(img)
}
x <- imagenet_preprocess_input(x)
```

We then load the trained network. The model has 50 layers, with a fair bit of complexity.

```r
model <- application_resnet50(weights = "imagenet")
summary(model)
```

```
Model: "resnet50"
________________________________________________________________________________
 Layer (type)           Output Shape   Param #  Connected to            Trainable
================================================================================
 input_1 (InputLayer)   [(None, 224,    0       []                      Y
                        224, 3)]
 conv1_pad (ZeroPaddi   (None, 230, 2   0       ['input_1[0][0]']       Y
 ng2D)                  30, 3)
 conv1_conv (Conv2D)    (None, 112, 1  9472     ['conv1_pad[0][0]']     Y
                        12, 64)
 conv1_bn (BatchNorma   (None, 112, 1   256     ['conv1_conv[0][0]']    Y
 lization)              12, 64)
 conv1_relu (Activati   (None, 112, 1   0       ['conv1_bn[0][0]']      Y
 on)                    12, 64)
 pool1_pad (ZeroPaddi   (None, 114, 1   0       ['conv1_relu[0][0]']    Y
 ng2D)                  14, 64)
 pool1_pool (MaxPooli   (None, 56, 56   0       ['pool1_pad[0][0]']     Y
 ng2D)                  , 64)
 conv2_block1_1_conv    (None, 56, 56   4160    ['pool1_pool[0][0]']    Y
 (Conv2D)               , 64)
 conv2_block1_1_bn (B   (None, 56, 56   256     ['conv2_block1_1_conv   Y
 atchNormalization)     , 64)                   [0][0]']
 conv2_block1_1_relu    (None, 56, 56   0       ['conv2_block1_1_bn[0   Y
 (Activation)           , 64)                   ][0]']
 conv2_block1_2_conv    (None, 56, 56   36928   ['conv2_block1_1_relu   Y
```

```
(Conv2D)              , 64)                     [0][0]']
conv2_block1_2_bn (B  (None, 56, 56  256        ['conv2_block1_2_conv  Y
atchNormalization)    , 64)                     [0][0]']
conv2_block1_2_relu   (None, 56, 56  0          ['conv2_block1_2_bn[0  Y
(Activation)          , 64)                     ][0]']
conv2_block1_0_conv   (None, 56, 56  16640      ['pool1_pool[0][0]']   Y
(Conv2D)              , 256)
conv2_block1_3_conv   (None, 56, 56  16640      ['conv2_block1_2_relu  Y
(Conv2D)              , 256)                    [0][0]']
conv2_block1_0_bn (B  (None, 56, 56  1024       ['conv2_block1_0_conv  Y
atchNormalization)    , 256)                    [0][0]']
conv2_block1_3_bn (B  (None, 56, 56  1024       ['conv2_block1_3_conv  Y
atchNormalization)    , 256)                    [0][0]']
conv2_block1_add (Ad  (None, 56, 56  0          ['conv2_block1_0_bn[0  Y
d)                    , 256)                    ][0]',
                                                 'conv2_block1_3_bn[0
                                                ][0]']
conv2_block1_out (Ac  (None, 56, 56  0          ['conv2_block1_add[0]  Y
tivation)             , 256)                    [0]']
conv2_block2_1_conv   (None, 56, 56  16448      ['conv2_block1_out[0]  Y
(Conv2D)              , 64)                     [0]']
conv2_block2_1_bn (B  (None, 56, 56  256        ['conv2_block2_1_conv  Y
atchNormalization)    , 64)                     [0][0]']
conv2_block2_1_relu   (None, 56, 56  0          ['conv2_block2_1_bn[0  Y
(Activation)          , 64)                     ][0]']
conv2_block2_2_conv   (None, 56, 56  36928      ['conv2_block2_1_relu  Y
(Conv2D)              , 64)                     [0][0]']
conv2_block2_2_bn (B  (None, 56, 56  256        ['conv2_block2_2_conv  Y
atchNormalization)    , 64)                     [0][0]']
conv2_block2_2_relu   (None, 56, 56  0          ['conv2_block2_2_bn[0  Y
(Activation)          , 64)                     ][0]']
conv2_block2_3_conv   (None, 56, 56  16640      ['conv2_block2_2_relu  Y
(Conv2D)              , 256)                    [0][0]']
conv2_block2_3_bn (B  (None, 56, 56  1024       ['conv2_block2_3_conv  Y
atchNormalization)    , 256)                    [0][0]']
conv2_block2_add (Ad  (None, 56, 56  0          ['conv2_block1_out[0]  Y
d)                    , 256)                    [0]',
                                                 'conv2_block2_3_bn[0
                                                ][0]']
conv2_block2_out (Ac  (None, 56, 56  0          ['conv2_block2_add[0]  Y
tivation)             , 256)                    [0]']
conv2_block3_1_conv   (None, 56, 56  16448      ['conv2_block2_out[0]  Y
(Conv2D)              , 64)                     [0]']
```

```
conv2_block3_1_bn (B  (None, 56, 56  256      ['conv2_block3_1_conv  Y
atchNormalization)    , 64)                   [0][0]']
conv2_block3_1_relu   (None, 56, 56  0        ['conv2_block3_1_bn[0  Y
(Activation)          , 64)                   ][0]']
conv2_block3_2_conv   (None, 56, 56  36928    ['conv2_block3_1_relu  Y
(Conv2D)              , 64)                   [0][0]']
conv2_block3_2_bn (B  (None, 56, 56  256      ['conv2_block3_2_conv  Y
atchNormalization)    , 64)                   [0][0]']
conv2_block3_2_relu   (None, 56, 56  0        ['conv2_block3_2_bn[0  Y
(Activation)          , 64)                   ][0]']
conv2_block3_3_conv   (None, 56, 56  16640    ['conv2_block3_2_relu  Y
(Conv2D)              , 256)                  [0][0]']
conv2_block3_3_bn (B  (None, 56, 56  1024     ['conv2_block3_3_conv  Y
atchNormalization)    , 256)                  [0][0]']
conv2_block3_add (Ad  (None, 56, 56  0        ['conv2_block2_out[0]  Y
d)                    , 256)                  [0]',
                                               'conv2_block3_3_bn[0
                                              ][0]']
conv2_block3_out (Ac  (None, 56, 56  0        ['conv2_block3_add[0]  Y
tivation)             , 256)                  [0]']
conv3_block1_1_conv   (None, 28, 28  32896    ['conv2_block3_out[0]  Y
(Conv2D)              , 128)                  [0]']
conv3_block1_1_bn (B  (None, 28, 28  512      ['conv3_block1_1_conv  Y
atchNormalization)    , 128)                  [0][0]']
conv3_block1_1_relu   (None, 28, 28  0        ['conv3_block1_1_bn[0  Y
(Activation)          , 128)                  ][0]']
conv3_block1_2_conv   (None, 28, 28  147584   ['conv3_block1_1_relu  Y
(Conv2D)              , 128)                  [0][0]']
conv3_block1_2_bn (B  (None, 28, 28  512      ['conv3_block1_2_conv  Y
atchNormalization)    , 128)                  [0][0]']
conv3_block1_2_relu   (None, 28, 28  0        ['conv3_block1_2_bn[0  Y
(Activation)          , 128)                  ][0]']
conv3_block1_0_conv   (None, 28, 28  131584   ['conv2_block3_out[0]  Y
(Conv2D)              , 512)                  [0]']
conv3_block1_3_conv   (None, 28, 28  66048    ['conv3_block1_2_relu  Y
(Conv2D)              , 512)                  [0][0]']
conv3_block1_0_bn (B  (None, 28, 28  2048     ['conv3_block1_0_conv  Y
atchNormalization)    , 512)                  [0][0]']
conv3_block1_3_bn (B  (None, 28, 28  2048     ['conv3_block1_3_conv  Y
atchNormalization)    , 512)                  [0][0]']
conv3_block1_add (Ad  (None, 28, 28  0        ['conv3_block1_0_bn[0  Y
d)                    , 512)                  ][0]',
                                               'conv3_block1_3_bn[0
```

```
                                                       ][0]']
 conv3_block1_out (Ac  (None, 28, 28  0           ['conv3_block1_add[0]  Y
 tivation)             , 512)                      [0]']
 conv3_block2_1_conv   (None, 28, 28  65664       ['conv3_block1_out[0]  Y
 (Conv2D)              , 128)                      [0]']
 conv3_block2_1_bn (B  (None, 28, 28  512         ['conv3_block2_1_conv  Y
 atchNormalization)    , 128)                      [0][0]']
 conv3_block2_1_relu   (None, 28, 28  0           ['conv3_block2_1_bn[0  Y
 (Activation)          , 128)                      ][0]']
 conv3_block2_2_conv   (None, 28, 28  147584      ['conv3_block2_1_relu  Y
 (Conv2D)              , 128)                      [0][0]']
 conv3_block2_2_bn (B  (None, 28, 28  512         ['conv3_block2_2_conv  Y
 atchNormalization)    , 128)                      [0][0]']
 conv3_block2_2_relu   (None, 28, 28  0           ['conv3_block2_2_bn[0  Y
 (Activation)          , 128)                      ][0]']
 conv3_block2_3_conv   (None, 28, 28  66048       ['conv3_block2_2_relu  Y
 (Conv2D)              , 512)                      [0][0]']
 conv3_block2_3_bn (B  (None, 28, 28  2048        ['conv3_block2_3_conv  Y
 atchNormalization)    , 512)                      [0][0]']
 conv3_block2_add (Ad  (None, 28, 28  0           ['conv3_block1_out[0]  Y
 d)                    , 512)                      [0]',
                                                    'conv3_block2_3_bn[0
                                                   ][0]']

 conv3_block2_out (Ac  (None, 28, 28  0           ['conv3_block2_add[0]  Y
 tivation)             , 512)                      [0]']
 conv3_block3_1_conv   (None, 28, 28  65664       ['conv3_block2_out[0]  Y
 (Conv2D)              , 128)                      [0]']
 conv3_block3_1_bn (B  (None, 28, 28  512         ['conv3_block3_1_conv  Y
 atchNormalization)    , 128)                      [0][0]']
 conv3_block3_1_relu   (None, 28, 28  0           ['conv3_block3_1_bn[0  Y
 (Activation)          , 128)                      ][0]']
 conv3_block3_2_conv   (None, 28, 28  147584      ['conv3_block3_1_relu  Y
 (Conv2D)              , 128)                      [0][0]']
 conv3_block3_2_bn (B  (None, 28, 28  512         ['conv3_block3_2_conv  Y
 atchNormalization)    , 128)                      [0][0]']
 conv3_block3_2_relu   (None, 28, 28  0           ['conv3_block3_2_bn[0  Y
 (Activation)          , 128)                      ][0]']
 conv3_block3_3_conv   (None, 28, 28  66048       ['conv3_block3_2_relu  Y
 (Conv2D)              , 512)                      [0][0]']
 conv3_block3_3_bn (B  (None, 28, 28  2048        ['conv3_block3_3_conv  Y
 atchNormalization)    , 512)                      [0][0]']
 conv3_block3_add (Ad  (None, 28, 28  0           ['conv3_block2_out[0]  Y
 d)                    , 512)                      [0]',
```

```
                                                 'conv3_block3_3_bn[0
                                                ][0]']
conv3_block3_out (Ac  (None, 28, 28  0          ['conv3_block3_add[0]  Y
tivation)            , 512)                     [0]']
conv3_block4_1_conv  (None, 28, 28  65664      ['conv3_block3_out[0]  Y
(Conv2D)             , 128)                     [0]']
conv3_block4_1_bn (B  (None, 28, 28  512        ['conv3_block4_1_conv  Y
atchNormalization)   , 128)                     [0][0]']
conv3_block4_1_relu  (None, 28, 28  0          ['conv3_block4_1_bn[0  Y
(Activation)         , 128)                     ][0]']
conv3_block4_2_conv  (None, 28, 28  147584     ['conv3_block4_1_relu  Y
(Conv2D)             , 128)                     [0][0]']
conv3_block4_2_bn (B  (None, 28, 28  512        ['conv3_block4_2_conv  Y
atchNormalization)   , 128)                     [0][0]']
conv3_block4_2_relu  (None, 28, 28  0          ['conv3_block4_2_bn[0  Y
(Activation)         , 128)                     ][0]']
conv3_block4_3_conv  (None, 28, 28  66048      ['conv3_block4_2_relu  Y
(Conv2D)             , 512)                     [0][0]']
conv3_block4_3_bn (B  (None, 28, 28  2048       ['conv3_block4_3_conv  Y
atchNormalization)   , 512)                     [0][0]']
conv3_block4_add (Ad  (None, 28, 28  0          ['conv3_block3_out[0]  Y
d)                   , 512)                     [0]',
                                                 'conv3_block4_3_bn[0
                                                ][0]']
conv3_block4_out (Ac  (None, 28, 28  0          ['conv3_block4_add[0]  Y
tivation)            , 512)                     [0]']
conv4_block1_1_conv  (None, 14, 14  131328     ['conv3_block4_out[0]  Y
(Conv2D)             , 256)                     [0]']
conv4_block1_1_bn (B  (None, 14, 14  1024       ['conv4_block1_1_conv  Y
atchNormalization)   , 256)                     [0][0]']
conv4_block1_1_relu  (None, 14, 14  0          ['conv4_block1_1_bn[0  Y
(Activation)         , 256)                     ][0]']
conv4_block1_2_conv  (None, 14, 14  590080     ['conv4_block1_1_relu  Y
(Conv2D)             , 256)                     [0][0]']
conv4_block1_2_bn (B  (None, 14, 14  1024       ['conv4_block1_2_conv  Y
atchNormalization)   , 256)                     [0][0]']
conv4_block1_2_relu  (None, 14, 14  0          ['conv4_block1_2_bn[0  Y
(Activation)         , 256)                     ][0]']
conv4_block1_0_conv  (None, 14, 14  525312     ['conv3_block4_out[0]  Y
(Conv2D)             , 1024)                    [0]']
conv4_block1_3_conv  (None, 14, 14  263168     ['conv4_block1_2_relu  Y
(Conv2D)             , 1024)                    [0][0]']
conv4_block1_0_bn (B  (None, 14, 14  4096       ['conv4_block1_0_conv  Y
```

```
atchNormalization)    , 1024)                          [0][0]']
conv4_block1_3_bn (B  (None, 14, 14  4096      ['conv4_block1_3_conv  Y
atchNormalization)    , 1024)                          [0][0]']
conv4_block1_add (Ad  (None, 14, 14  0         ['conv4_block1_0_bn[0  Y
d)                    , 1024)                          ][0]',
                                                        'conv4_block1_3_bn[0
                                                       ][0]']
conv4_block1_out (Ac  (None, 14, 14  0         ['conv4_block1_add[0]  Y
tivation)             , 1024)                          [0]']
conv4_block2_1_conv   (None, 14, 14  262400    ['conv4_block1_out[0]  Y
(Conv2D)              , 256)                           [0]']
conv4_block2_1_bn (B  (None, 14, 14  1024      ['conv4_block2_1_conv  Y
atchNormalization)    , 256)                           [0][0]']
conv4_block2_1_relu   (None, 14, 14  0         ['conv4_block2_1_bn[0  Y
(Activation)          , 256)                           ][0]']
conv4_block2_2_conv   (None, 14, 14  590080    ['conv4_block2_1_relu  Y
(Conv2D)              , 256)                           [0][0]']
conv4_block2_2_bn (B  (None, 14, 14  1024      ['conv4_block2_2_conv  Y
atchNormalization)    , 256)                           [0][0]']
conv4_block2_2_relu   (None, 14, 14  0         ['conv4_block2_2_bn[0  Y
(Activation)          , 256)                           ][0]']
conv4_block2_3_conv   (None, 14, 14  263168    ['conv4_block2_2_relu  Y
(Conv2D)              , 1024)                          [0][0]']
conv4_block2_3_bn (B  (None, 14, 14  4096      ['conv4_block2_3_conv  Y
atchNormalization)    , 1024)                          [0][0]']
conv4_block2_add (Ad  (None, 14, 14  0         ['conv4_block1_out[0]  Y
d)                    , 1024)                          [0]',
                                                        'conv4_block2_3_bn[0
                                                       ][0]']
conv4_block2_out (Ac  (None, 14, 14  0         ['conv4_block2_add[0]  Y
tivation)             , 1024)                          [0]']
conv4_block3_1_conv   (None, 14, 14  262400    ['conv4_block2_out[0]  Y
(Conv2D)              , 256)                           [0]']
conv4_block3_1_bn (B  (None, 14, 14  1024      ['conv4_block3_1_conv  Y
atchNormalization)    , 256)                           [0][0]']
conv4_block3_1_relu   (None, 14, 14  0         ['conv4_block3_1_bn[0  Y
(Activation)          , 256)                           ][0]']
conv4_block3_2_conv   (None, 14, 14  590080    ['conv4_block3_1_relu  Y
(Conv2D)              , 256)                           [0][0]']
conv4_block3_2_bn (B  (None, 14, 14  1024      ['conv4_block3_2_conv  Y
atchNormalization)    , 256)                           [0][0]']
conv4_block3_2_relu   (None, 14, 14  0         ['conv4_block3_2_bn[0  Y
(Activation)          , 256)                           ][0]']
```

```
conv4_block3_3_conv   (None, 14, 14  263168  ['conv4_block3_2_relu  Y
(Conv2D)              , 1024)                [0][0]']
conv4_block3_3_bn (B  (None, 14, 14  4096    ['conv4_block3_3_conv  Y
atchNormalization)    , 1024)                [0][0]']
conv4_block3_add (Ad  (None, 14, 14  0       ['conv4_block2_out[0]  Y
d)                    , 1024)                [0]',
                                              'conv4_block3_3_bn[0
                                             ][0]']
conv4_block3_out (Ac  (None, 14, 14  0       ['conv4_block3_add[0]  Y
tivation)             , 1024)                [0]']
conv4_block4_1_conv   (None, 14, 14  262400  ['conv4_block3_out[0]  Y
(Conv2D)              , 256)                 [0]']
conv4_block4_1_bn (B  (None, 14, 14  1024    ['conv4_block4_1_conv  Y
atchNormalization)    , 256)                 [0][0]']
conv4_block4_1_relu   (None, 14, 14  0       ['conv4_block4_1_bn[0  Y
(Activation)          , 256)                 ][0]']
conv4_block4_2_conv   (None, 14, 14  590080  ['conv4_block4_1_relu  Y
(Conv2D)              , 256)                 [0][0]']
conv4_block4_2_bn (B  (None, 14, 14  1024    ['conv4_block4_2_conv  Y
atchNormalization)    , 256)                 [0][0]']
conv4_block4_2_relu   (None, 14, 14  0       ['conv4_block4_2_bn[0  Y
(Activation)          , 256)                 ][0]']
conv4_block4_3_conv   (None, 14, 14  263168  ['conv4_block4_2_relu  Y
(Conv2D)              , 1024)                [0][0]']
conv4_block4_3_bn (B  (None, 14, 14  4096    ['conv4_block4_3_conv  Y
atchNormalization)    , 1024)                [0][0]']
conv4_block4_add (Ad  (None, 14, 14  0       ['conv4_block3_out[0]  Y
d)                    , 1024)                [0]',
                                              'conv4_block4_3_bn[0
                                             ][0]']
conv4_block4_out (Ac  (None, 14, 14  0       ['conv4_block4_add[0]  Y
tivation)             , 1024)                [0]']
conv4_block5_1_conv   (None, 14, 14  262400  ['conv4_block4_out[0]  Y
(Conv2D)              , 256)                 [0]']
conv4_block5_1_bn (B  (None, 14, 14  1024    ['conv4_block5_1_conv  Y
atchNormalization)    , 256)                 [0][0]']
conv4_block5_1_relu   (None, 14, 14  0       ['conv4_block5_1_bn[0  Y
(Activation)          , 256)                 ][0]']
conv4_block5_2_conv   (None, 14, 14  590080  ['conv4_block5_1_relu  Y
(Conv2D)              , 256)                 [0][0]']
conv4_block5_2_bn (B  (None, 14, 14  1024    ['conv4_block5_2_conv  Y
atchNormalization)    , 256)                 [0][0]']
conv4_block5_2_relu   (None, 14, 14  0       ['conv4_block5_2_bn[0  Y
```

```
(Activation)          , 256)                    ][0]']
conv4_block5_3_conv    (None, 14, 14  263168   ['conv4_block5_2_relu  Y
(Conv2D)              , 1024)                   [0][0]']
conv4_block5_3_bn (B   (None, 14, 14  4096     ['conv4_block5_3_conv  Y
atchNormalization)    , 1024)                   [0][0]']
conv4_block5_add (Ad   (None, 14, 14  0        ['conv4_block4_out[0]  Y
d)                    , 1024)                   [0]',
                                                 'conv4_block5_3_bn[0
                                                ][0]']
conv4_block5_out (Ac   (None, 14, 14  0        ['conv4_block5_add[0]  Y
tivation)             , 1024)                   [0]']
conv4_block6_1_conv    (None, 14, 14  262400   ['conv4_block5_out[0]  Y
(Conv2D)              , 256)                    [0]']
conv4_block6_1_bn (B   (None, 14, 14  1024     ['conv4_block6_1_conv  Y
atchNormalization)    , 256)                    [0][0]']
conv4_block6_1_relu    (None, 14, 14  0        ['conv4_block6_1_bn[0  Y
(Activation)          , 256)                    ][0]']
conv4_block6_2_conv    (None, 14, 14  590080   ['conv4_block6_1_relu  Y
(Conv2D)              , 256)                    [0][0]']
conv4_block6_2_bn (B   (None, 14, 14  1024     ['conv4_block6_2_conv  Y
atchNormalization)    , 256)                    [0][0]']
conv4_block6_2_relu    (None, 14, 14  0        ['conv4_block6_2_bn[0  Y
(Activation)          , 256)                    ][0]']
conv4_block6_3_conv    (None, 14, 14  263168   ['conv4_block6_2_relu  Y
(Conv2D)              , 1024)                   [0][0]']
conv4_block6_3_bn (B   (None, 14, 14  4096     ['conv4_block6_3_conv  Y
atchNormalization)    , 1024)                   [0][0]']
conv4_block6_add (Ad   (None, 14, 14  0        ['conv4_block5_out[0]  Y
d)                    , 1024)                   [0]',
                                                 'conv4_block6_3_bn[0
                                                ][0]']
conv4_block6_out (Ac   (None, 14, 14  0        ['conv4_block6_add[0]  Y
tivation)             , 1024)                   [0]']
conv5_block1_1_conv    (None, 7, 7,   524800   ['conv4_block6_out[0]  Y
(Conv2D)              512)                      [0]']
conv5_block1_1_bn (B   (None, 7, 7,   2048     ['conv5_block1_1_conv  Y
atchNormalization)    512)                      [0][0]']
conv5_block1_1_relu    (None, 7, 7,   0        ['conv5_block1_1_bn[0  Y
(Activation)          512)                      ][0]']
conv5_block1_2_conv    (None, 7, 7,   2359808   ['conv5_block1_1_relu  Y
(Conv2D)              512)                      [0][0]']
conv5_block1_2_bn (B   (None, 7, 7,   2048     ['conv5_block1_2_conv  Y
atchNormalization)    512)                      [0][0]']
```

```
conv5_block1_2_relu   (None, 7, 7,   0        ['conv5_block1_2_bn[0  Y
(Activation)          512)                    ][0]']
conv5_block1_0_conv   (None, 7, 7,   2099200  ['conv4_block6_out[0]  Y
(Conv2D)              2048)                   [0]']
conv5_block1_3_conv   (None, 7, 7,   1050624  ['conv5_block1_2_relu  Y
(Conv2D)              2048)                   [0][0]']
conv5_block1_0_bn (B  (None, 7, 7,   8192     ['conv5_block1_0_conv  Y
atchNormalization)    2048)                   [0][0]']
conv5_block1_3_bn (B  (None, 7, 7,   8192     ['conv5_block1_3_conv  Y
atchNormalization)    2048)                   [0][0]']
conv5_block1_add (Ad  (None, 7, 7,   0        ['conv5_block1_0_bn[0  Y
d)                    2048)                   ][0]',
                                               'conv5_block1_3_bn[0
                                              ][0]']
conv5_block1_out (Ac  (None, 7, 7,   0        ['conv5_block1_add[0]  Y
tivation)             2048)                   [0]']
conv5_block2_1_conv   (None, 7, 7,   1049088  ['conv5_block1_out[0]  Y
(Conv2D)              512)                    [0]']
conv5_block2_1_bn (B  (None, 7, 7,   2048     ['conv5_block2_1_conv  Y
atchNormalization)    512)                    [0][0]']
conv5_block2_1_relu   (None, 7, 7,   0        ['conv5_block2_1_bn[0  Y
(Activation)          512)                    ][0]']
conv5_block2_2_conv   (None, 7, 7,   2359808  ['conv5_block2_1_relu  Y
(Conv2D)              512)                    [0][0]']
conv5_block2_2_bn (B  (None, 7, 7,   2048     ['conv5_block2_2_conv  Y
atchNormalization)    512)                    [0][0]']
conv5_block2_2_relu   (None, 7, 7,   0        ['conv5_block2_2_bn[0  Y
(Activation)          512)                    ][0]']
conv5_block2_3_conv   (None, 7, 7,   1050624  ['conv5_block2_2_relu  Y
(Conv2D)              2048)                   [0][0]']
conv5_block2_3_bn (B  (None, 7, 7,   8192     ['conv5_block2_3_conv  Y
atchNormalization)    2048)                   [0][0]']
conv5_block2_add (Ad  (None, 7, 7,   0        ['conv5_block1_out[0]  Y
d)                    2048)                   [0]',
                                               'conv5_block2_3_bn[0
                                              ][0]']
conv5_block2_out (Ac  (None, 7, 7,   0        ['conv5_block2_add[0]  Y
tivation)             2048)                   [0]']
conv5_block3_1_conv   (None, 7, 7,   1049088  ['conv5_block2_out[0]  Y
(Conv2D)              512)                    [0]']
conv5_block3_1_bn (B  (None, 7, 7,   2048     ['conv5_block3_1_conv  Y
atchNormalization)    512)                    [0][0]']
conv5_block3_1_relu   (None, 7, 7,   0        ['conv5_block3_1_bn[0  Y
```

```
 (Activation)         512)                    ][0]']
 conv5_block3_2_conv   (None, 7, 7,   2359808  ['conv5_block3_1_relu  Y
 (Conv2D)             512)                    [0][0]']
 conv5_block3_2_bn (B  (None, 7, 7,   2048     ['conv5_block3_2_conv  Y
 atchNormalization)   512)                    [0][0]']
 conv5_block3_2_relu   (None, 7, 7,   0        ['conv5_block3_2_bn[0  Y
 (Activation)         512)                    ][0]']
 conv5_block3_3_conv   (None, 7, 7,   1050624  ['conv5_block3_2_relu  Y
 (Conv2D)             2048)                   [0][0]']
 conv5_block3_3_bn (B  (None, 7, 7,   8192     ['conv5_block3_3_conv  Y
 atchNormalization)   2048)                   [0][0]']
 conv5_block3_add (Ad  (None, 7, 7,   0        ['conv5_block2_out[0]  Y
 d)                   2048)                   [0]',
                                               'conv5_block3_3_bn[0
                                              ][0]']
 conv5_block3_out (Ac  (None, 7, 7,   0        ['conv5_block3_add[0]  Y
 tivation)            2048)                   [0]']
 avg_pool (GlobalAver  (None, 2048)  0        ['conv5_block3_out[0]  Y
 agePooling2D)                                [0]']
 predictions (Dense)  (None, 1000)   2049000  ['avg_pool[0][0]']     Y
================================================================================
Total params: 25,636,712
Trainable params: 25,583,592
Non-trainable params: 53,120
________________________________________________________________________________
```

Finally, we classify our six images, and return the top three class choices in terms of predicted probability for each.

```r
pred6 <- model %>% predict(x) %>%
  imagenet_decode_predictions(top = 3)
```

```
1/1 - 1s - 915ms/epoch - 915ms/step
```

```r
names(pred6) <- image_names
print(pred6)
```

```
$flamingo.jpg
  class_name class_description       score
1  n02007558          flamingo 0.926349819
2  n02006656         spoonbill 0.071699291
```

```
3  n02002556       white_stork 0.001228213

$hawk.jpg
  class_name class_description     score
1  n03388043          fountain 0.2788655
2  n03532672              hook 0.1785545
3  n03804744              nail 0.1080728

$hawk_cropped.jpeg
  class_name class_description      score
1  n01608432              kite 0.72270948
2  n01622779    great_grey_owl 0.08182576
3  n01532829       house_finch 0.04218859

$huey.jpg
  class_name           class_description      score
1  n02097474             Tibetan_terrier 0.50929701
2  n02098413                       Lhasa 0.42209885
3  n02098105 soft-coated_wheaten_terrier 0.01695857

$kitty.jpg
  class_name    class_description      score
1  n02105641 Old_English_sheepdog 0.83265996
2  n02086240             Shih-Tzu 0.04513895
3  n03223299              doormat 0.03299766

$weaver.jpg
  class_name class_description      score
1  n01843065           jacamar 0.49795479
2  n01818515             macaw 0.22193271
3  n02494079   squirrel_monkey 0.04287853
```

Document Classification

IMDb Document Classification: Now we perform document classification on the `IMDB` dataset, which is available as part of the `keras` package. We limit the dictionary size to the 10,000 most frequently-used words and tokens.

```r
max_features <- 10000
imdb <- keras::dataset_imdb(num_words = max_features)
c(c(x_train, y_train), c(x_test, y_test)) %<-% imdb
```

The third line is a shortcut for unpacking the list of lists. Each element of `x_train` is a vector of numbers between 0 and 9999 (the document), referring to the words found in the dictionary. For example, the first training document is the positive review on page 419. The indices of the first 12 words are given below.

```
x_train[[3]][1:12]
```

```
 [1]   1  14  47   8  30  31   7   4 249 108   7   4
```

To see the words, we create a function, `decode_review()`, that provides a simple interface to the dictionary.

```
word_index <- dataset_imdb_word_index()

decode_review <- function(text, word_index) {
  word <- names(word_index)
  idx <- unlist(word_index, use.names = FALSE)
  word <- c("<PAD>", "<START>", "<UNK>", "<UNUSED>", word)
  idx <- c(0:3, idx + 3)
  words <- word[match(text, idx, 2)]
  paste(words, collapse = " ")
}
decode_review(x_train[[3]][1:12], word_index)
```

```
[1] "<START> this has to be one of the worst films of the"
```

Next we write a function to "one-hot" encode each document in a list of documents, and return a binary matrix in sparse-matrix format.

```
library(Matrix)
one_hot <- function(sequences, dimension) {
  seqlen <- sapply(sequences, length)
  n <- length(seqlen)
  rowind <- rep(1:n, seqlen)
  colind <- unlist(sequences)
  sparseMatrix(i = rowind, j = colind,
      dims = c(n, dimension))
}
```

To construct the sparse matrix, one supplies just the entries that are nonzero. In the last line we call the function `sparseMatrix()` and supply the row indices corresponding to each document and the column indices corresponding to the words in each document, since we omit the values they are taken to be all ones. Words that appear more than once in any given document still get recorded as a one.

```
x_train_1h <- one_hot(x_train, 10000)
x_test_1h <- one_hot(x_test, 10000)
dim(x_train_1h)
```

```
[1] 25000 10000
```

```
nnzero(x_train_1h) / (25000 * 10000)
```

```
[1] 0.01316987
```

Only 1.3% of the entries are nonzero, so this amounts to considerable savings in memory. We create a validation set of size 2,000, leaving 23,000 for training.

```
set.seed(3)
ival <- sample(seq(along = y_train), 2000)
```

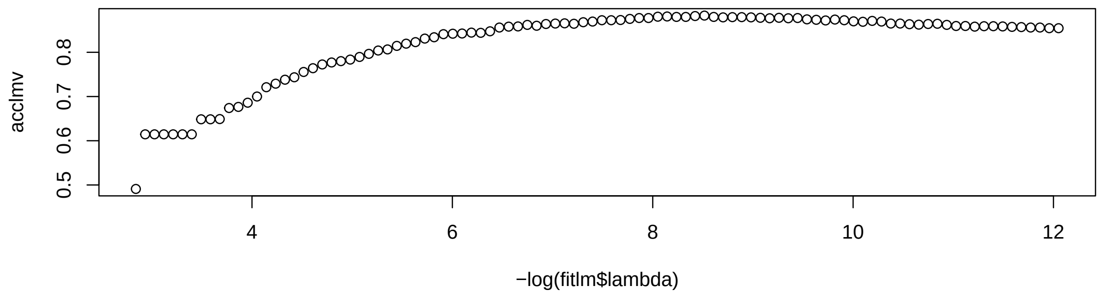
First we fit a lasso logistic regression model using `glmnet()` on the training data, and evaluate its performance on the validation data. Finally, we plot the accuracy, `acclmv`, as a function of the shrinkage parameter, λ. Similar expressions compute the performance on the test data, and were used to produce the left plot in Figure 10.11. The code takes advantage of the sparse-matrix format of `x_train_1h`, and runs in about 5 seconds; in the usual dense format it would take about 5 minutes.

```
library(glmnet)
fitlm <- glmnet(x_train_1h[-ival, ], y_train[-ival],
    family = "binomial", standardize = FALSE)
classlmv <- predict(fitlm, x_train_1h[ival, ]) > 0
acclmv <- apply(classlmv, 2, accuracy,  y_train[ival] > 0)
```

We applied the `accuracy()` function that we wrote in Lab 10.9.2 to every column of the prediction matrix `classlmv`, and since this is a logical matrix of `TRUE/FALSE` values, we supply the second argument `truth` as a logical vector as well.

Before making a plot, we adjust the plotting window.

```r
par(mar = c(4, 4, 4, 4), mfrow = c(1, 1))
plot(-log(fitlm$lambda), acclmv)
```



Next we fit a fully-connected neural network with two hidden layers, each with 16 units and ReLU activation.

```r
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu",
      input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model %>% compile(optimizer = "rmsprop",
    loss = "binary_crossentropy", metrics = c("accuracy"))
history <- model %>% fit(x_train_1h[-ival, ], y_train[-ival],
    epochs = 20, batch_size = 512,
    validation_data = list(x_train_1h[ival, ], y_train[ival]),
  verbose = 0)
print(head(history$metrics, 5))
```

```
$loss
 [1] 0.447553039 0.262521893 0.201916799 0.164405093 0.142305508 0.119016998
 [7] 0.104999542 0.090582088 0.076303884 0.066411130 0.056033410 0.047542598
[13] 0.038732834 0.035908751 0.027669312 0.019332321 0.020281816 0.015630113
[19] 0.010610131 0.008690959

$accuracy
 [1] 0.8225217 0.9088261 0.9296522 0.9425652 0.9502174 0.9606956 0.9648696
 [8] 0.9708695 0.9769565 0.9797391 0.9837826 0.9861304 0.9894348 0.9894348
[15] 0.9932609 0.9959565 0.9946957 0.9961739 0.9979131 0.9983478
```

```
$val_loss
 [1] 0.3431589 0.2885661 0.2834007 0.2881699 0.3030240 0.3132887 0.3806255
 [8] 0.3499370 0.3824932 0.4424604 0.4517584 0.4716862 0.4997365 0.5273197
[15] 0.5545352 0.5965419 0.6977289 0.6617905 0.7004055 0.7453282

$val_accuracy
 [1] 0.8760 0.8835 0.8880 0.8855 0.8800 0.8850 0.8610 0.8780 0.8715 0.8760
[11] 0.8730 0.8725 0.8615 0.8635 0.8625 0.8610 0.8445 0.8570 0.8590 0.8570
```

The `history` object has a `metrics` component that records both the training and validation accuracy at each epoch. Figure 10.11 includes test accuracy at each epoch as well. To compute the test accuracy, we rerun the entire sequence above, replacing the last line with

```
history <- model %>% fit(
    x_train_1h[-ival, ], y_train[-ival], epochs = 20,
    batch_size = 512, validation_data = list(x_test_1h, y_test),
      verbose = 0
  )
print(head(history$metrics, 5))
```

```
$loss
 [1] 0.00931599271 0.00703764427 0.00264910678 0.00568766985 0.00411444949
 [6] 0.00119175098 0.00309719611 0.00341222622 0.00055942941 0.00426902343
[11] 0.00032282385 0.00235970668 0.00021256637 0.00304006785 0.00015229669
[16] 0.00011978008 0.00342195784 0.00008774632 0.00247872388 0.00006217395

$accuracy
 [1] 0.9980000 0.9982609 0.9998261 0.9986087 0.9988695 0.9999130 0.9992608
 [8] 0.9989130 0.9999565 0.9986957 1.0000000 0.9991739 1.0000000 0.9992174
[15] 1.0000000 1.0000000 0.9992608 1.0000000 0.9990870 1.0000000

$val_loss
 [1] 0.8638822 0.9131214 0.9530587 0.9942631 1.0319979 1.0799935 1.1114842
 [8] 1.1504575 1.1960199 1.2454590 1.2779648 1.3271402 1.3514003 1.3928775
[15] 1.4196711 1.4763879 1.5038086 1.5324416 1.5653005 1.5881859

$val_accuracy
 [1] 0.84860 0.84700 0.84724 0.84652 0.84496 0.84524 0.84524 0.84388 0.84160
[10] 0.84288 0.84232 0.84112 0.84156 0.84200 0.84204 0.84256 0.84272 0.84224
[19] 0.84220 0.84192
```

Recurrent Neural Networks

Sequential Models for Document Classification. Before we just use the model to give us a response based on the presence or abcense of words, in recurrent neural networks we are actually taking into consideration the sequence of words

Here we fit a simple LSTM RNN for sentiment analysis with the `IMDb` movie-review data. We first calculate the lengths of the documents.

```
wc <- sapply(x_train, length) #count of words
median(wc)
```

```
[1] 178
```

```
sum(wc <= 500) / length(wc) #percentage with less or equal 500 words
```

```
[1] 0.91568
```

We see that over 91% of the documents have fewer than 500 words. Our RNN requires all the document sequences to have the same length. We hence restrict the document lengths to the last $L = 500$ words, and pad the beginning of the shorter ones with blanks.

```
maxlen <- 500
x_train <- pad_sequences(x_train, maxlen = maxlen)
x_test <- pad_sequences(x_test, maxlen = maxlen)
dim(x_train)
```

```
[1] 25000   500
```

```
dim(x_test)
```

```
[1] 25000   500
```

```
x_train[1, 490:500]
```

```
 [1]   16 4472  113  103   32   15   16 5345   19  178   32
```

The last expression shows the last few words in the first document. At this stage, each of the 500 words in the document is represented using an integer corresponding to the location of that word in the 10,000-word dictionary.

The first layer of the RNN is an embedding layer of size 32, which will be learned during training. This layer one-hot encodes each document as a matrix of dimension $500 \times 10,000$, and then maps these $10,000$ dimensions down to 32.

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 32) %>%
  layer_lstm(units = 32) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

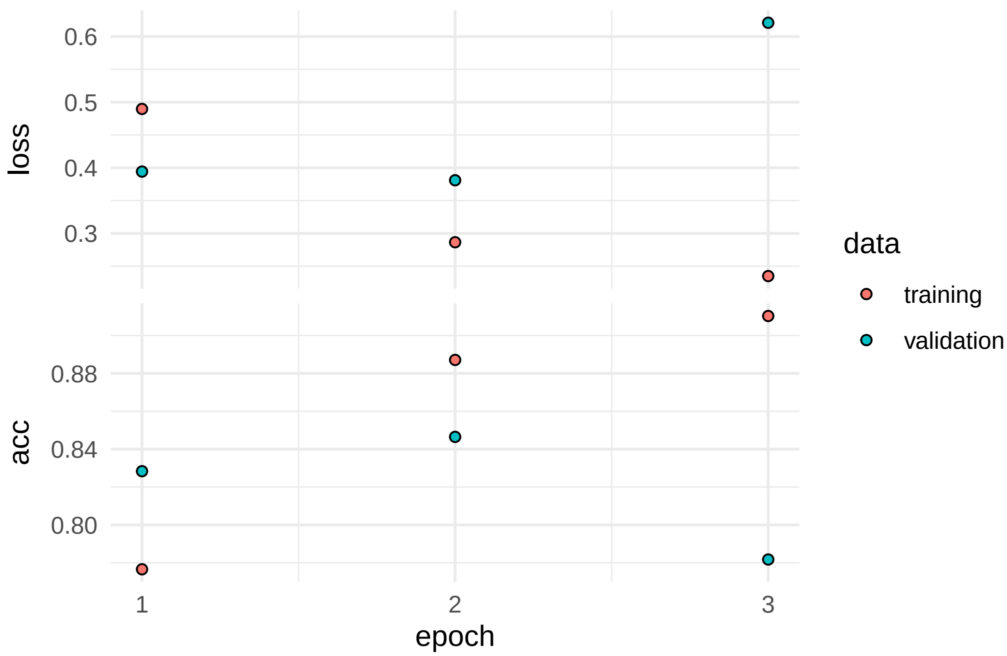
The second layer is an LSTM with 32 units, and the output layer is a single sigmoid for the binary classification task.

The rest is now similar to other networks we have fit. We track the test performance as the network is fit, and see that it attains 87% accuracy.

```
model %>% compile(optimizer = "rmsprop",
    loss = "binary_crossentropy", metrics = c("acc"))
#history <- model %>% fit(x_train, y_train, epochs = 10,
history <- model %>% fit(x_train, y_train, epochs = 3,
    batch_size = 128, validation_data = list(x_test, y_test),
  verbose = 0)
print(head(history$metrics, 5))
```

```
$loss
[1] 0.4896822 0.2864636 0.2349275

$acc
[1] 0.77656 0.88712 0.91032

$val_loss
[1] 0.3941714 0.3809676 0.6210713

$val_acc
[1] 0.82836 0.84648 0.78172
```

```
plot(history)
```




```r
predy <- predict(model, x_test) > 0.5
```

```
782/782 - 23s - 23s/epoch - 29ms/step
```

```r
mean(abs(y_test == as.numeric(predy)))
```

```
[1] 0.78172
```

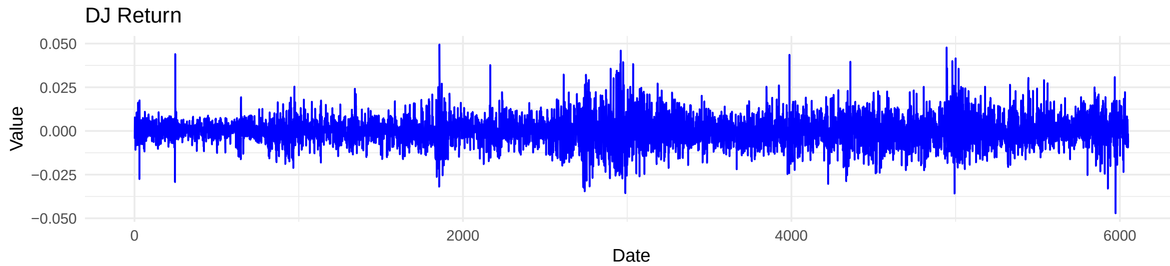
Time series prediction

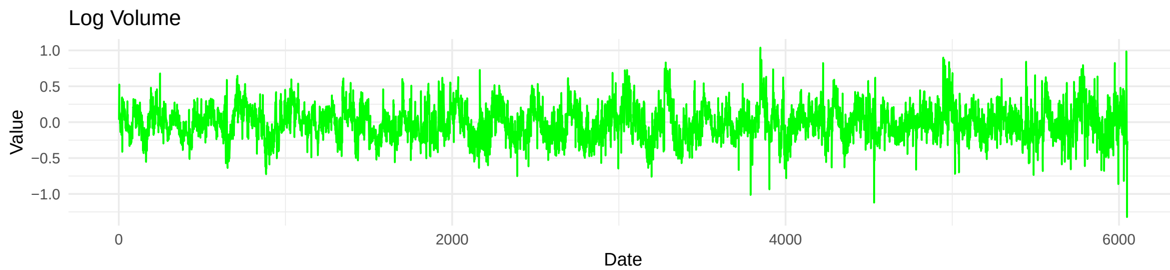
We now show how to fit the models for time series prediction. We first set up the data, and standardize each of the variables. The data has three different time-series: *Log trading volume*. This is the fraction of all outstanding shares that are traded on that day, relative to a 100-day moving average of past turnover, on the log scale. • *Dow Jones return*. This is the difference between the log of the Dow Jones Industrial Index on consecutive trading days. • *Log volatility*. This is based on the absolute values of daily price movements.

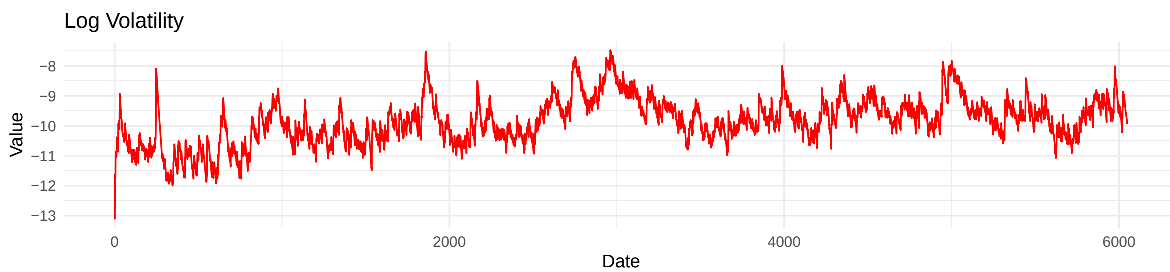
We are interested in predicting trading volume.

```
        date day_of_week DJ_return log_volume log_volatility train
1 1962-12-03         mon -0.004461   0.032573      -13.12740  TRUE
2 1962-12-04        tues  0.007813   0.346202      -11.74930  TRUE
```

```
3 1962-12-05          wed  0.003845   0.525306      -11.66561  TRUE
4 1962-12-06         thur -0.003462   0.210182      -11.62677  TRUE
5 1962-12-07          fri  0.000568   0.044187      -11.72813  TRUE
6 1962-12-10          mon -0.010824   0.133246      -10.87253  TRUE
```

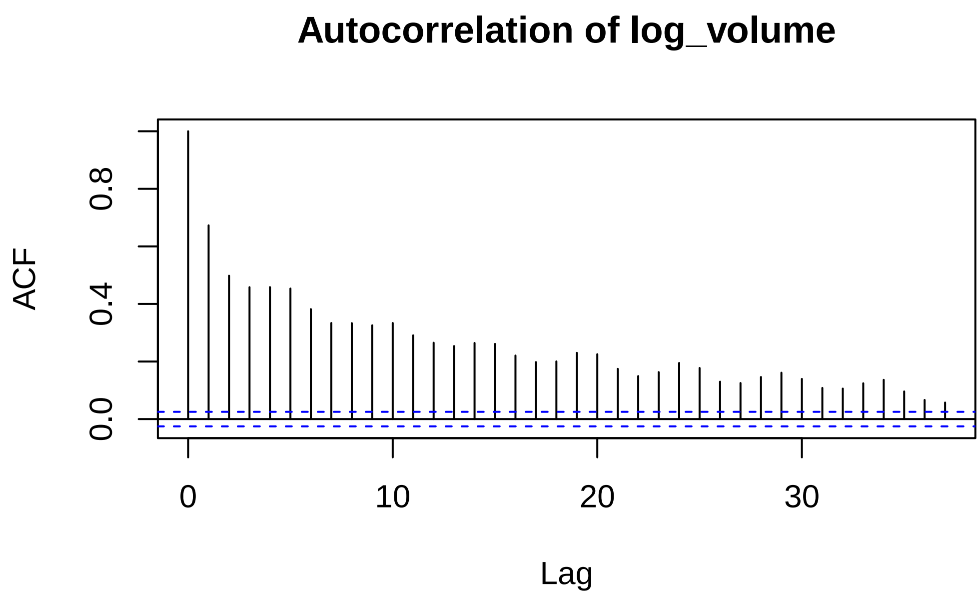






An observation here consists of the measurements (vt, rt, zt) on day t, in this case the values for log_volume, DJ_return and log_volatility. One feature that strikes us immediately is that the dayto-day observations are not independent of each other. The series exhibit auto-correlation — in this case values nearby in time tend to be similar to each other. This distinguishes time series from other data sets we have correlation encountered, in which observations can be assumed to be independent of each other. To be clear, consider pairs of observations (vt, vt−), a lag of days apart. If we take all such pairs in the vt series and compute their correlation coefficient, this gives the autocorrelation at lag .

```r
rt <- NYSE$log_volume  # Replace with your actual time series column

# Calculate and plot the autocorrelation
acf_result <- acf(rt, plot = TRUE, main = "Autocorrelation of log_volume")
```




```
# Alternatively, if you want to customize the plot using ggplot2
acf_values <- acf(rt, plot = FALSE)
acf_data <- with(acf_values, data.frame(lag, acf))
```

We see that nerby values are strongly correlated.

We are going to use the last 5 days to predict the next day's trading volume, this is $L = 5$ First, we are going to create a matrix with the data and standardize the values for our model:

```
xdata <- data.matrix(
    NYSE[, c("DJ_return",
      "log_volume",
      "log_volatility")]
  )
istrain <- NYSE[, "train"]
xdata <- scale(xdata) #this standardize the variables in a dataset
head(xdata)
```

```
       DJ_return log_volume log_volatility
[1,] -0.54977791  0.1750605      -4.356718
[2,]  0.90512515  1.5171653      -2.528849
[3,]  0.43477682  2.2836006      -2.417837
```

```
[4,] -0.43136108  0.9350983      -2.366325
[5,]  0.04633644  0.2247600      -2.500763
[6,] -1.30401843  0.6058680      -1.365915
```

The variable `istrain` contains a `TRUE` for each year that is in the training set, and a `FALSE` for each year in the test set.

The `scale` function in R is used to standardize variables in a dataset. This is particularly useful in machine learning and statistical modeling where different variables may have different scales (units or magnitudes). Here's a breakdown of what scale does: *Centers the Data*: Subtracts the mean of each column (variable) from the values in that column. This shifts the data so that it has a mean of zero. *Scales the Data*: Divides each column by its standard deviation. This rescales the data so that each column has a standard deviation of one.

This normalization ensures that each variable contributes equally to the analysis, avoiding bias towards variables with larger magnitudes. It also helps with the convergence of gradient descent in machine learning algorithms and makes coefficients in regression models more interpretable.

Now we write functions to create lagged versions of the three time series. We start with a function that takes as input a data matrix and a lag L, and returns a lagged version of the matrix. It simply inserts L rows of `NA` at the top, and truncates the bottom.

```r
lagm <- function(x, k = 1) {
  n <- nrow(x)
  pad <- matrix(NA, k, ncol(x))
  rbind(pad, x[1:(n - k), ])
}
```

We now use this function to create a data frame with all the required lags, as well as the response variable.

```r
arframe <- data.frame(log_volume =
    xdata[, "log_volume"],
    L1 = lagm(xdata, 1),
    L2 = lagm(xdata, 2),
    L3 = lagm(xdata, 3),
    L4 = lagm(xdata, 4),
    L5 = lagm(xdata, 5)
 )
head(arframe)
```

```
  log_volume L1.DJ_return L1.log_volume L1.log_volatility L2.DJ_return
1  0.1750605          NA           NA                NA           NA
2  1.5171653 -0.54977791    0.1750605         -4.356718           NA
3  2.2836006  0.90512515    1.5171653         -2.528849   -0.5497779
4  0.9350983  0.43477682    2.2836006         -2.417837    0.9051251
5  0.2247600 -0.43136108    0.9350983         -2.366325    0.4347768
6  0.6058680  0.04633644    0.2247600         -2.500763   -0.4313611
  L2.log_volume L2.log_volatility L3.DJ_return L3.log_volume L3.log_volatility
1            NA                NA           NA            NA                NA
2            NA                NA           NA            NA                NA
3     0.1750605         -4.356718           NA            NA                NA
4     1.5171653         -2.528849   -0.5497779     0.1750605         -4.356718
5     2.2836006         -2.417837    0.9051251     1.5171653         -2.528849
6     0.9350983         -2.366325    0.4347768     2.2836006         -2.417837
  L4.DJ_return L4.log_volume L4.log_volatility L5.DJ_return L5.log_volume
1           NA            NA                NA           NA            NA
2           NA            NA                NA           NA            NA
3           NA            NA                NA           NA            NA
4           NA            NA                NA           NA            NA
5   -0.5497779     0.1750605         -4.356718           NA            NA
6    0.9051251     1.5171653         -2.528849   -0.5497779     0.1750605
  L5.log_volatility
1                NA
2                NA
3                NA
4                NA
5                NA
6         -4.356718
```

If we look at the first five rows of this frame, we will see some missing values in the lagged variables (due to the construction above). We remove these rows, and adjust `istrain` accordingly.

```
arframe <- arframe[-(1:5), ]
istrain <- istrain[-(1:5)]
head(arframe)
```

```
    log_volume L1.DJ_return L1.log_volume L1.log_volatility L2.DJ_return
6   0.60586798  0.046336436    0.22476000         -2.500763 -0.431361080
7  -0.01365982 -1.304018428    0.60586798         -1.365915  0.046336436
8   0.04254846 -0.006293266   -0.01365982         -1.505543 -1.304018428
9  -0.41980156  0.377050100    0.04254846         -1.551386 -0.006293266
```

```
10 -0.55601945 -0.411684210   -0.41980156         -1.597475  0.377050100
11 -0.17673016  0.508742889   -0.55601945         -1.564257 -0.411684210
   L2.log_volume L2.log_volatility L3.DJ_return L3.log_volume L3.log_volatility
6     0.93509830         -2.366325  0.434776822    2.28360065         -2.417837
7     0.22476000         -2.500763 -0.431361080    0.93509830         -2.366325
8     0.60586798         -1.365915  0.046336436    0.22476000         -2.500763
9    -0.01365982         -1.505543 -1.304018428    0.60586798         -1.365915
10    0.04254846         -1.551386 -0.006293266   -0.01365982         -1.505543
11   -0.41980156         -1.597475  0.377050100    0.04254846         -1.551386
   L4.DJ_return L4.log_volume L4.log_volatility L5.DJ_return L5.log_volume
6   0.905125145    1.51716533         -2.528849  -0.54977791     0.1750605
7   0.434776822    2.28360065         -2.417837   0.90512515     1.5171653
8  -0.431361080    0.93509830         -2.366325   0.43477682     2.2836006
9   0.046336436    0.22476000         -2.500763  -0.43136108     0.9350983
10 -1.304018428    0.60586798         -1.365915   0.04633644     0.2247600
11 -0.006293266   -0.01365982         -1.505543  -1.30401843     0.6058680
   L5.log_volatility
6          -4.356718
7          -2.528849
8          -2.417837
9          -2.366325
10         -2.500763
11         -1.365915
```

We now fit the linear AR model to the training data using `lm()`, and predict on the test data.

```
arfit <- lm(log_volume ~ ., data = arframe[istrain, ])
arpred <- predict(arfit, arframe[!istrain, ])
V0 <- var(arframe[!istrain, "log_volume"])
1 - mean((arpred - arframe[!istrain, "log_volume"])^2) / V0
```

```
[1] 0.413223
```

The last two lines compute the R^2 on the test data.

We refit this model, including the factor variable `day_of_week`.

```
arframed <-
    data.frame(day = NYSE[-(1:5), "day_of_week"], arframe)
arfitd <- lm(log_volume ~ ., data = arframed[istrain, ])
arpredd <- predict(arfitd, arframed[!istrain, ])
1 - mean((arpredd - arframe[!istrain, "log_volume"])^2) / V0
```

```
[1] 0.4598616
```

To fit the RNN, we need to reshape these data, since it expects a sequence of $L = 5$ feature vectors $X = {X_\ell}_1^L$ for each observation. These are lagged versions of the time series going back L time points.

```r
n <- nrow(arframe)
xrnn <- data.matrix(arframe[, -1])
xrnn <- array(xrnn, c(n, 3, 5))
xrnn <- xrnn[,, 5:1]
xrnn <- aperm(xrnn, c(1, 3, 2))
dim(xrnn)
```

```
[1] 6046    5    3
```

```r
head(xrnn)
```

```
, , 1

            [,1]         [,2]         [,3]         [,4]         [,5]
[1,] -0.54977791  0.905125145  0.434776822 -0.431361080  0.046336436
[2,]  0.90512515  0.434776822 -0.431361080  0.046336436 -1.304018428
[3,]  0.43477682 -0.431361080  0.046336436 -1.304018428 -0.006293266
[4,] -0.43136108  0.046336436 -1.304018428 -0.006293266  0.377050100
[5,]  0.04633644 -1.304018428 -0.006293266  0.377050100 -0.411684210
[6,] -1.30401843 -0.006293266  0.377050100 -0.411684210  0.508742889

, , 2

          [,1]        [,2]        [,3]        [,4]        [,5]
[1,] 0.1750605  1.51716533  2.28360065  0.93509830  0.22476000
[2,] 1.5171653  2.28360065  0.93509830  0.22476000  0.60586798
[3,] 2.2836006  0.93509830  0.22476000  0.60586798 -0.01365982
[4,] 0.9350983  0.22476000  0.60586798 -0.01365982  0.04254846
[5,] 0.2247600  0.60586798 -0.01365982  0.04254846 -0.41980156
[6,] 0.6058680 -0.01365982  0.04254846 -0.41980156 -0.55601945

, , 3

          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] -4.356718 -2.528849 -2.417837 -2.366325 -2.500763
```

```
[2,] -2.528849 -2.417837 -2.366325 -2.500763 -1.365915
[3,] -2.417837 -2.366325 -2.500763 -1.365915 -1.505543
[4,] -2.366325 -2.500763 -1.365915 -1.505543 -1.551386
[5,] -2.500763 -1.365915 -1.505543 -1.551386 -1.597475
[6,] -1.365915 -1.505543 -1.551386 -1.597475 -1.564257
```

We have done this in four steps. The first simply extracts the $n \times 15$ matrix of lagged versions of the three predictor variables from `arframe`. The second converts this matrix to an $n \times 3 \times 5$ array. We can do this by simply changing the dimension attribute, since the new array is filled column wise. The third step reverses the order of lagged variables, so that index 1 is furthest back in time, and index 5 closest. The final step rearranges the coordinates of the array (like a partial transpose) into the format that the RNN module in `keras` expects.

Now we are ready to proceed with the RNN, which uses 12 hidden units.

```r
model <- keras_model_sequential() %>%
  layer_simple_rnn(units = 12,
      input_shape = list(5, 3),
      dropout = 0.1, recurrent_dropout = 0.1) %>%
  layer_dense(units = 1)
model %>% compile(optimizer = optimizer_rmsprop(),
    loss = "mse")
```

We specify two forms of dropout for the units feeding into the hidden layer. The first is for the input sequence feeding into this layer, and the second is for the previous hidden units feeding into the layer. The output layer has a single unit for the response.

We fit the model in a similar fashion to previous networks. We supply the `fit` function with test data as validation data, so that when we monitor its progress and plot the history function we can see the progress on the test data. Of course we should not use this as a basis for early stopping, since then the test performance would be biased.

```r
history <- model %>% fit(
    xrnn[istrain,, ], arframe[istrain, "log_volume"],
#    batch_size = 64, epochs = 200,
    batch_size = 64, epochs = 75,
    validation_data =
      list(xrnn[!istrain,, ], arframe[!istrain, "log_volume"]),
  verbose = 0
  )
print(head(history$metrics, 5))
```

```
$loss
 [1] 0.8269190 0.5619866 0.5376866 0.5058819 0.5063190 0.5010589 0.4938314
 [8] 0.4902737 0.4852141 0.4851107 0.4859041 0.4832917 0.4839135 0.4762774
[15] 0.4777993 0.4758667 0.4705367 0.4691449 0.4675083 0.4611216 0.4761748
[22] 0.4638441 0.4641424 0.4648597 0.4599679 0.4571057 0.4596542 0.4637221
[29] 0.4624321 0.4643588 0.4556880 0.4649343 0.4688988 0.4673363 0.4653184
[36] 0.4585154 0.4566802 0.4659116 0.4578327 0.4530186 0.4565886 0.4547758
[43] 0.4575248 0.4557547 0.4548748 0.4537613 0.4522369 0.4540089 0.4524329
[50] 0.4531841 0.4453050 0.4511724 0.4545194 0.4579861 0.4561250 0.4474057
[57] 0.4517469 0.4487627 0.4463556 0.4384032 0.4518566 0.4533428 0.4531325
[64] 0.4531381 0.4529711 0.4498225 0.4465671 0.4562317 0.4433665 0.4477214
[71] 0.4428962 0.4530282 0.4498457 0.4507887 0.4477258

$val_loss
 [1] 0.7144673 0.6955919 0.6854635 0.6806739 0.6781889 0.6776626 0.6755297
 [8] 0.6690893 0.6686965 0.6579845 0.6580985 0.6582813 0.6603827 0.6595697
[15] 0.6547570 0.6518834 0.6448734 0.6528118 0.6449838 0.6441239 0.6437474
[22] 0.6381695 0.6409055 0.6339521 0.6430431 0.6345581 0.6349177 0.6387298
[29] 0.6392431 0.6304538 0.6317526 0.6329050 0.6328675 0.6387510 0.6367837
[36] 0.6293262 0.6381230 0.6369861 0.6278794 0.6293920 0.6306012 0.6425406
[43] 0.6327006 0.6286145 0.6453980 0.6284095 0.6319052 0.6234053 0.6302740
[50] 0.6245575 0.6251807 0.6318864 0.6274309 0.6300978 0.6222560 0.6282558
[57] 0.6261851 0.6371560 0.6314958 0.6277037 0.6246527 0.6269179 0.6262379
[64] 0.6331049 0.6302149 0.6230734 0.6241696 0.6262228 0.6252472 0.6219482
[71] 0.6247986 0.6238847 0.6284890 0.6204042 0.6238285
```

```r
kpred <- predict(model, xrnn[!istrain,, ])
```

```
56/56 - 0s - 140ms/epoch - 3ms/step
```

```r
1 - mean((kpred - arframe[!istrain, "log_volume"])^2) / V0
```

```
[1] 0.4081929
```

This model takes about one minute to train.

We could replace the `keras_model_sequential()` command above with the following command:

```r
model <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(5, 3)) %>%
  layer_dense(units = 1)
```

Here, `layer_flatten()` simply takes the input sequence and turns it into a long vector of predictors. This results in a linear AR model.

To fit a nonlinear AR model, we could add in a hidden layer.

However, since we already have the matrix of lagged variables from the AR model that we fit earlier using the `lm()` command, we can actually fit a nonlinear AR model without needing to perform flattening.

We extract the model matrix `x` from `arframed`, which includes the `day_of_week` variable.

```r
x <- model.matrix(log_volume ~ . - 1, data = arframed)
head(x)
```

```
   dayfri daymon daythur daytues daywed L1.DJ_return L1.log_volume
6       0      1       0       0      0  0.046336436    0.22476000
7       0      0       0       1      0 -1.304018428    0.60586798
8       0      0       0       0      1 -0.006293266   -0.01365982
9       0      0       1       0      0  0.377050100    0.04254846
10      1      0       0       0      0 -0.411684210   -0.41980156
11      0      1       0       0      0  0.508742889   -0.55601945
   L1.log_volatility L2.DJ_return L2.log_volume L2.log_volatility L3.DJ_return
6          -2.500763 -0.431361080    0.93509830         -2.366325  0.434776822
7          -1.365915  0.046336436    0.22476000         -2.500763 -0.431361080
8          -1.505543 -1.304018428    0.60586798         -1.365915  0.046336436
9          -1.551386 -0.006293266   -0.01365982         -1.505543 -1.304018428
10         -1.597475  0.377050100    0.04254846         -1.551386 -0.006293266
11         -1.564257 -0.411684210   -0.41980156         -1.597475  0.377050100
   L3.log_volume L3.log_volatility L4.DJ_return L4.log_volume L4.log_volatility
6     2.28360065         -2.417837  0.905125145    1.51716533         -2.528849
7     0.93509830         -2.366325  0.434776822    2.28360065         -2.417837
8     0.22476000         -2.500763 -0.431361080    0.93509830         -2.366325
9     0.60586798         -1.365915  0.046336436    0.22476000         -2.500763
10   -0.01365982         -1.505543 -1.304018428    0.60586798         -1.365915
11    0.04254846         -1.551386 -0.006293266   -0.01365982         -1.505543
   L5.DJ_return L5.log_volume L5.log_volatility
6   -0.54977791     0.1750605         -4.356718
7    0.90512515     1.5171653         -2.528849
8    0.43477682     2.2836006         -2.417837
```

```
9   -0.43136108     0.9350983         -2.366325
10   0.04633644     0.2247600         -2.500763
11  -1.30401843     0.6058680         -1.365915
```

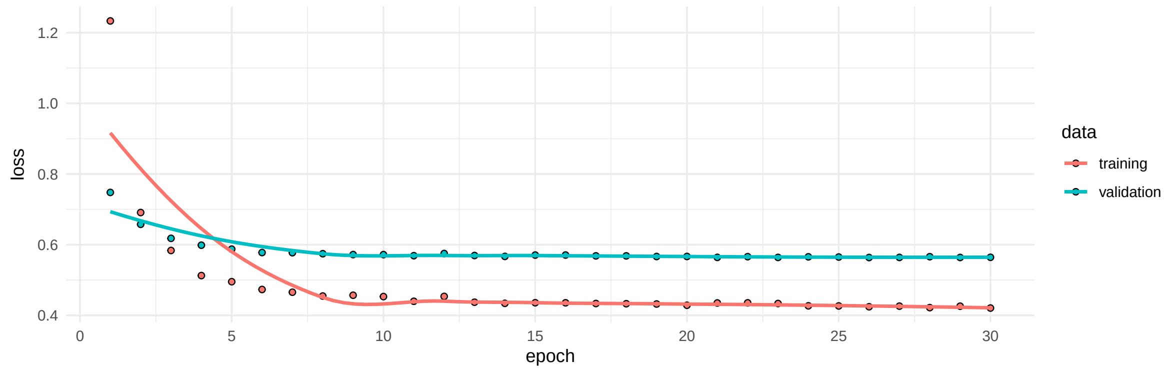
The `-1` in the formula avoids the creation of a column of ones for the intercept. The variable `day_of_week` is a five-level factor (there are five trading days), and the `-1` results in five rather than four dummy variables.

The rest of the steps to fit a nonlinear Auto Regressive model should by now be familiar.

```
arnnd <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = 'relu',
      input_shape = ncol(x)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1)
arnnd %>% compile(loss = "mse",
    optimizer = optimizer_rmsprop())
history <- arnnd %>% fit(
#    x[istrain, ], arframe[istrain, "log_volume"], epochs = 100,
    x[istrain, ], arframe[istrain, "log_volume"], epochs = 30,
    batch_size = 32, validation_data =
      list(x[!istrain, ], arframe[!istrain, "log_volume"]),
  verbose = 0
  )
print(head(history$metrics, 5))
```

```
$loss
 [1] 1.2331693 0.6910301 0.5836641 0.5126253 0.4952895 0.4732618 0.4655321
 [8] 0.4547598 0.4570046 0.4531760 0.4399017 0.4536812 0.4372670 0.4340856
[15] 0.4357720 0.4355728 0.4335757 0.4329647 0.4322048 0.4286356 0.4351058
[22] 0.4354852 0.4335649 0.4270032 0.4266661 0.4243830 0.4261284 0.4219196
[29] 0.4260258 0.4207680

$val_loss
 [1] 0.7479317 0.6576657 0.6179965 0.5986081 0.5874322 0.5777498 0.5773387
 [8] 0.5745054 0.5720753 0.5719293 0.5688995 0.5750841 0.5694293 0.5668634
[15] 0.5706574 0.5707616 0.5684080 0.5683356 0.5665651 0.5667366 0.5641197
[22] 0.5661207 0.5640193 0.5656826 0.5651062 0.5637234 0.5639966 0.5660630
[29] 0.5638607 0.5643280
```

```r
plot(history)
```




```r
npred <- predict(arnnd, x[!istrain, ])
```

```
56/56 - 0s - 64ms/epoch - 1ms/step
```

```r
1 - mean((arframe[!istrain, "log_volume"] - npred)^2) / V0
```

```
[1] 0.4646393
```