tds Published in Towards Data Science

Nicolo Cosimo Albanese Follow
Aug 23 · 6 min read · Member-only · Listen

Introduction to Deep Learning with Keras in R

A step-by-step tutorial

View from Monte San Vigilio (Vigiljoch), Trentino-Alto Adige, Italy. Image by author.

Table of contents

Both R and Python are useful and popular tools for Data Science. However, when it comes to Deep Learning, it is most common to find tutorials and guides for Python rather than R.

This post provides a simple Deep Learning example in the R language. It aims at sharing a practical introduction to the subject for R practitioners, using Keras.

2. Environment Setup

In this example, we share code snippets that can be easily copied and pasted on **Google Colab**[1].

Colab allows anyone to create notebooks in Python or R by writing code through the browser, entirely for free.

We can create a new R notebook in Colab through this link[2]. From there, we install Keras as follows:

```
install.packages("keras")
```

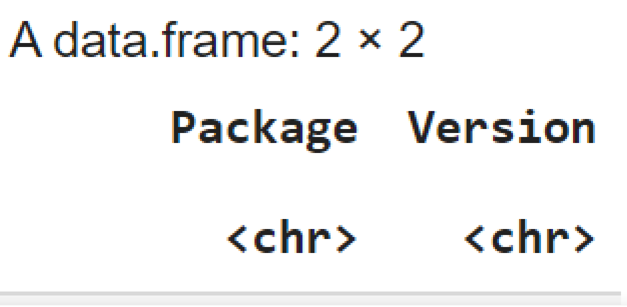
Although we leverage Colab for simplicity, the local installation process is equally straightforward[3]. We can now import the needed libraries:

```
library(tidyverse)
library(keras)
```

Note: `install.packages("keras")` also installs TensorFlow. It is possible to check the available versions of Keras and TensorFlow from the installed packages list:

```
installed.packages()[,c(1,3)] %>%
  as.data.frame() %>%
  filter(Package %in% c("keras", "tensorflow"))
```

[mnist_r] check_packages.r hosted with ❤ by GitHub view raw



```
A data.frame: 2 × 2
        Package  Version
          <chr>    <chr>
```

Image by author.

3. Dataset

Our purpose is to classify images of handwritten digits. For this example,we make use of the MNIST[4] dataset, a classic of the Machine Learning community.

The dataset has the following characteristics:

- 60,000 training images and 10,000 test images.
- Images of size 28 x 28 pixels.
- 10 categories (digits from 0 to 9).
- Grayscale images: pixel values range between 0 (black) and 255 (white).

Neural Networks require data in the shape of **tensors**. Tensors are algebraic objects with an arbitrary number of dimensions (D). For example, we can see vectors as 1D tensors and matrices as 2D tensors.

In the case of images, we need a vector space able to convey:

- Number of images (N)
- Image height (H)
- Image width (W)
- Color channels (C), also known as color depth.

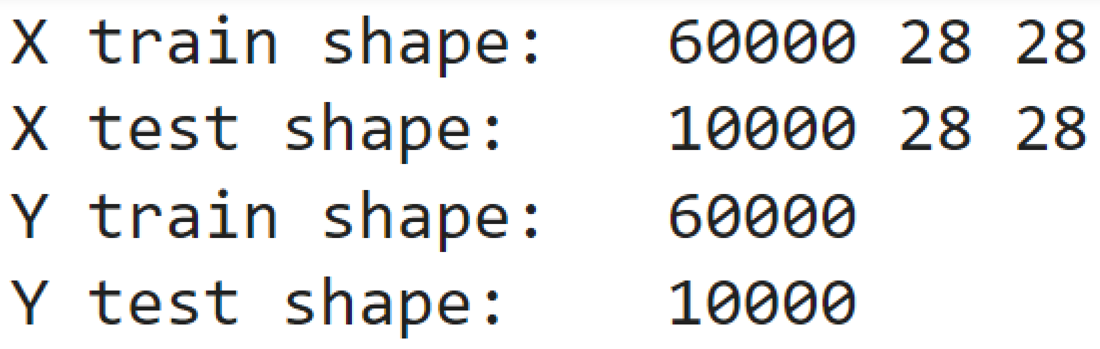
Therefore, in Deep Learning tasks images are generally represented as 4D tensors with shape: N x H x W x C.

In the case of grayscales images, the color channel is a single number (from 0 to 255) for each sample. Hence, it is possible to either omit the channel axis or leave it equal to one.

Let us import the MNIST dataset from Keras (under the Apache 2.0 License[5]) and verify the shape of the training and test images:

```r
c(c(x_train, y_train), c(x_test, y_test)) %<-% keras::dataset_mnist()

cat("\nX train shape:\t", dim(x_train))
cat("\nX test shape:\t", dim(x_test))
cat("\nY train shape:\t", dim(y_train))
cat("\nY test shape:\t", dim(y_test))
```

[mnist_r] load_dataset.r hosted with ❤ by GitHub view raw



```
X train shape:   60000 28 28
X test shape:    10000 28 28
Y train shape:   60000
Y test shape:    10000
```

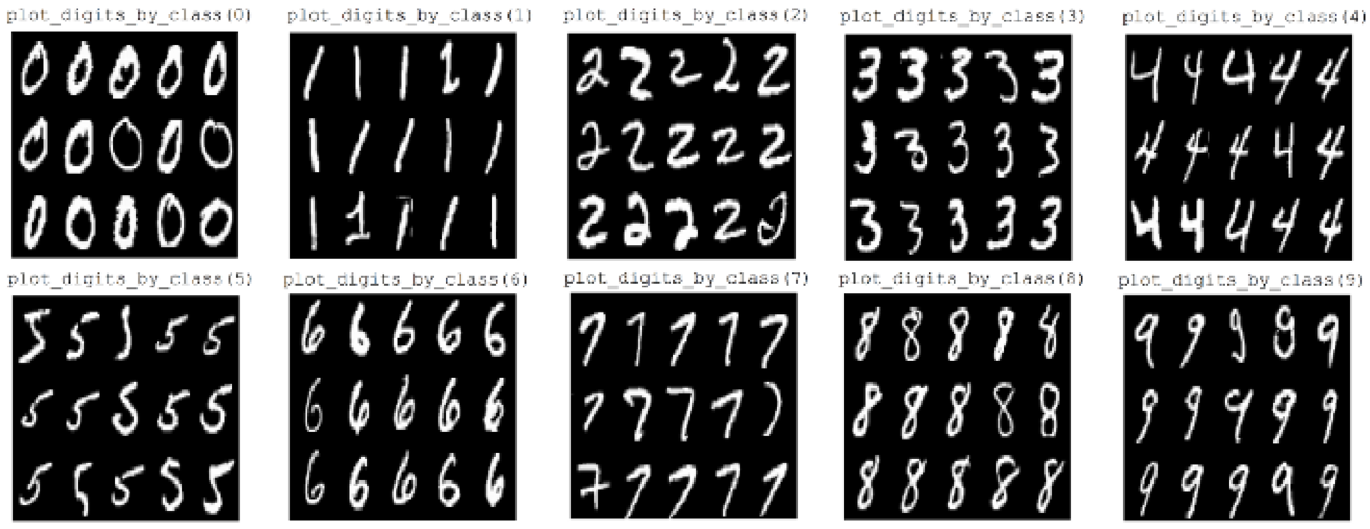
Training and test images (X) as 3D tensors. The color channel axis is omitted (grayscales). Image by author.

The representation of input observations as tensors applies to any data type. For example, tabular data in the form of a csv file with 300 rows (samples) and 8 columns (features) can be seen as 2D tensors of shape 300 x 8.

We can have a look at some samples by their corresponding label:

```r
plot_digits_by_class <- function(c) {
  # plot first 15 digits of class c

  # accepted labels are between 0 and 9
  if (c > -1 & c < 10) {

    # indexes of the first 15 digits of class c
    idx <- which(y_train == c)[1:15]

    # prepare plotting area
    par(mfcol=c(3, 5))
    par(mar=c(0, 0, 0, 0), xaxs = 'i', yaxs = 'i')

    # plot digits corresponding to indexes
    for (i in idx) {
      img <- x_train[i,,]
      img <- t(apply(img, 2, rev))
      image(1:28, 1:28, img, col = gray((0:255) / 255), xaxt = 'n', yaxt = 'n')
      }
    } else {
      return("Labels are between 0 and 9.")
    }
}
```

[mnist_r] plot_digits.r hosted with ❤ by GitHub view raw



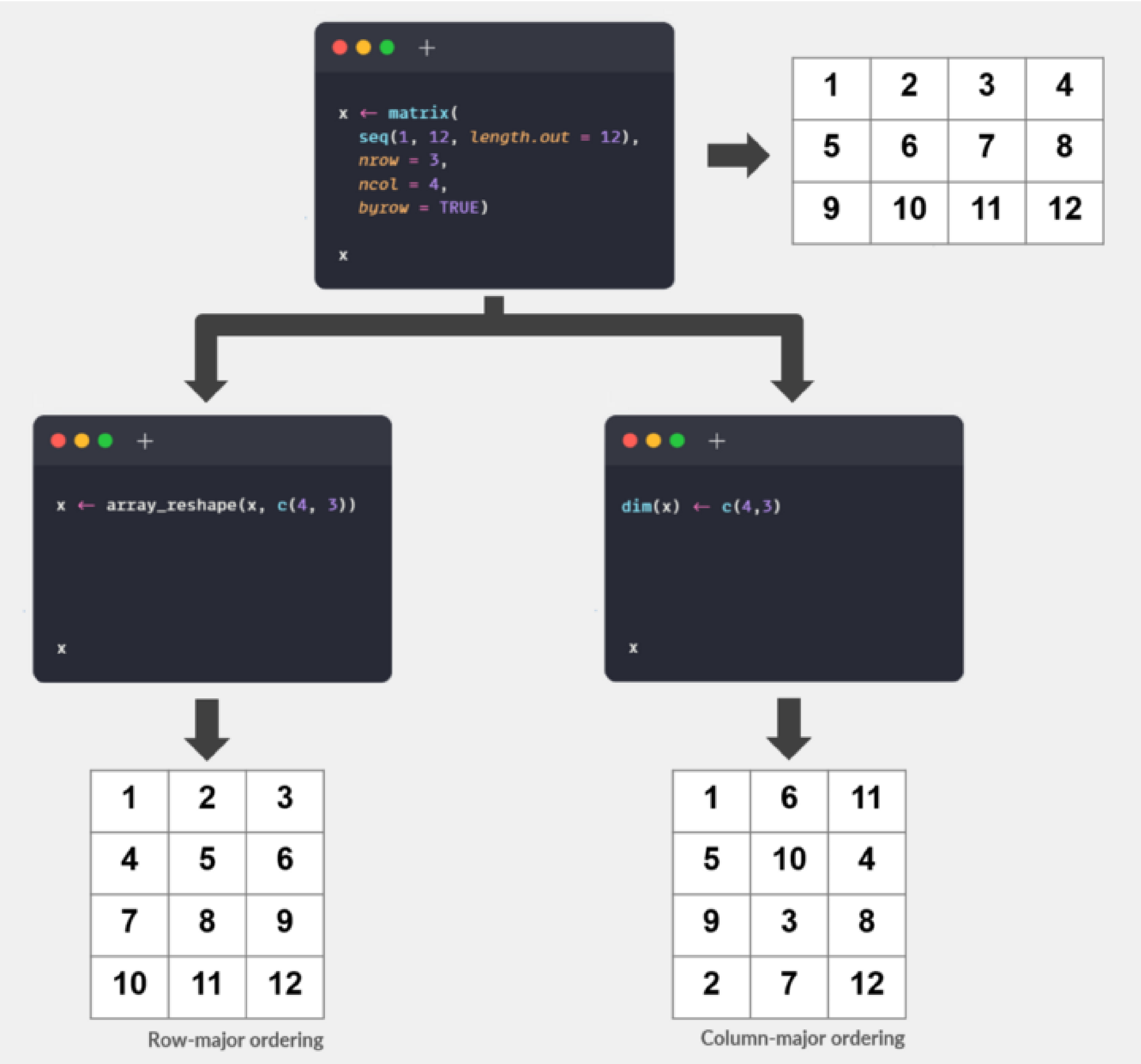
Handwritten digit samples obtained by calling plot_digits_by_class(x) for each category x (from 0 to 9).

As color values are in the [0, 255] interval, we can **scale** them to be in the [0, 1] interval. Moreover, we can reshape the input by **flattening** images from a 2D 28 x 28 to 1D 784 (28*28) without information loss:

```
x_train_reshaped = array_reshape(x_train, c(60000, 28*28)) / 255
x_test_reshaped = array_reshape(x_test, c(10000, 28*28)) / 255
```

[mnist_r] preproc_x.r hosted with ❤ by GitHub view raw

Note: Keras (as other common libraries) expects to reshape an array by filling new axes in **row-major ordering** (from the C language). This is the behaviour of `array_reshape()`. R practitioners may be more familiar with `dim<-()` to deal with matrices shapes. Nevertheless, `dim<-()` fills new axes in **column-major ordering** (from the Fortran language):



```
x ← matrix(
  seq(1, 12, length.out = 12),
  nrow = 3,
  ncol = 4,
  byrow = TRUE)

x
```

1	2	3	4
5	6	7	8
9	10	11	12

```
x ← array_reshape(x, c(4, 3))

x
```

1	2	3
4	5	6
7	8	9
10	11	12

```
dim(x) ← c(4,3)

x
```

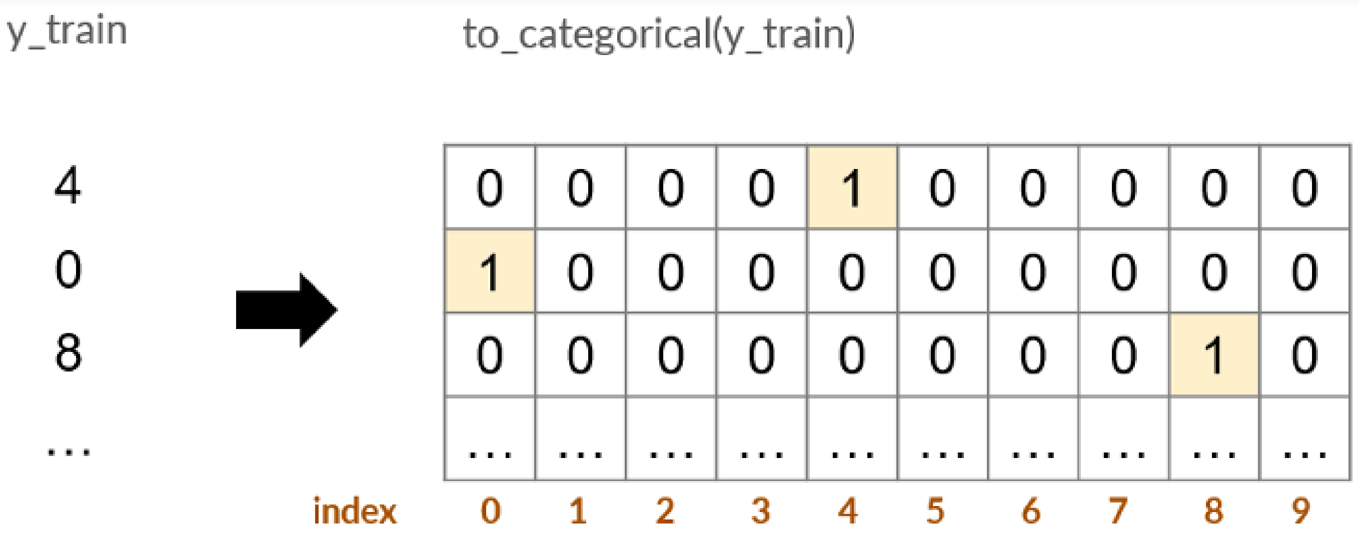
1	6	11
5	10	4
9	3	8
2	7	12

array_reshape vs dim<-(). Image by author.

The labels must be converted from a vector with integers (each integer representing a category) into a matrix with binary values and columns equal to the number of categories:

```
y_train <- to_categorical(y_train)
y_test <- to_categorical(y_test)
```

[mnist_r] preproc_y.r hosted with ❤ by GitHub view raw



index	0	1	2	3	4	5	6	7	8	9
4	0	0	0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0
...	...	...	...	...	...	...	...	...	...	...

Image by author.

5. Building the neural network

5.1 Define the layers

The cornerstone of a neural network is the *layer*. We can imagine the layer as a module that extracts a representation of the input data that is useful to the final goal.

We can build a neural network by stacking layers sequentially. Keras allows to do it by leveraging `keras_model_sequential`. In this example, we create a network composed of three layers:

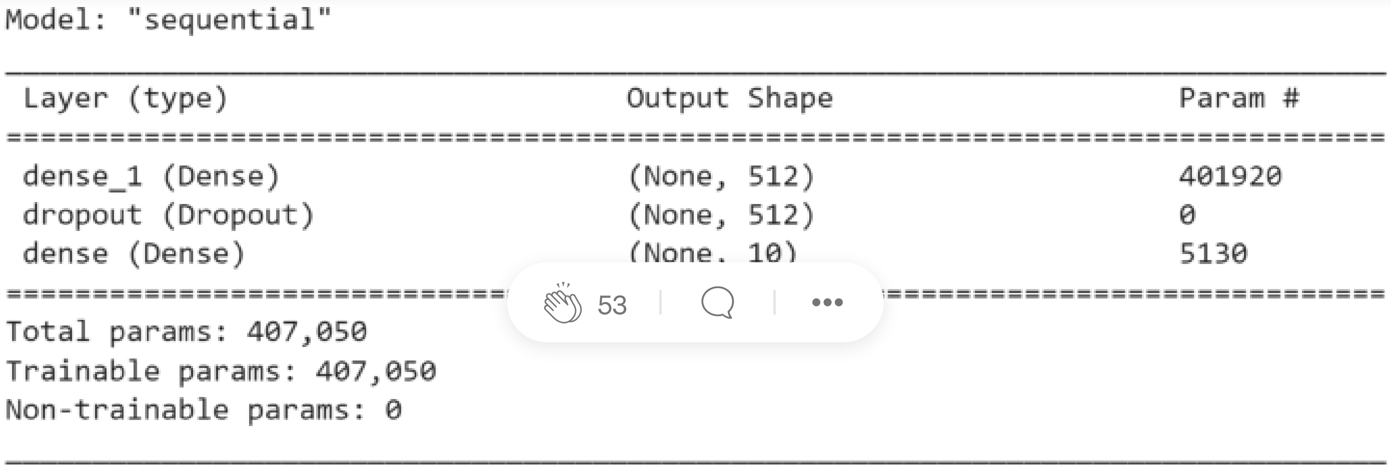
- A fully connected (or dense) layer that produces an output space of 512 units.
- A dropout layer to "*drop out*" randomly 20% of neurons during the training. In brief, this technique aims at improving the generalization capabilities of the model.
- A final dense layer with an output of 10 units and a *softmax* activation function. This layer returns an array of probability scores for each category, each being the probability of the current image to represent a 0, 1, 2, … up to 9:

```
model <- keras_model_sequential(input_shape = c(28 * 28)) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dropout(0.2) %>%
  layer_dense(units = 10, activation = "softmax")
```

[mnist_r] model.r hosted with ❤ by GitHub view raw

We can inspect the model's structure as follows:

```
model %>%
  summary()
```



```
Model: "sequential"
________________________________________________________________________________
 Layer (type)                      Output Shape                     Param #
================================================================================
 dense_1 (Dense)                   (None, 512)                      401920
 dropout (Dropout)                 (None, 512)                      0
 dense (Dense)                     (None, 10)                       5130
================================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
________________________________________________________________________________
```

Image by author.

5.2 Compile

In the compilation step, we need to define:

- *Loss function*:
  - The loss function must provide a reasonable estimate of the model error.
  - The network tries to minimize this function during training.
- *Optimizer*:
  - It specifies how the weights of the model get updated during training.
  - It makes use of the gradient of the loss function.
- *Metrics*:
  - An array of metrics to monitor during the training procedure.

```r
model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
```

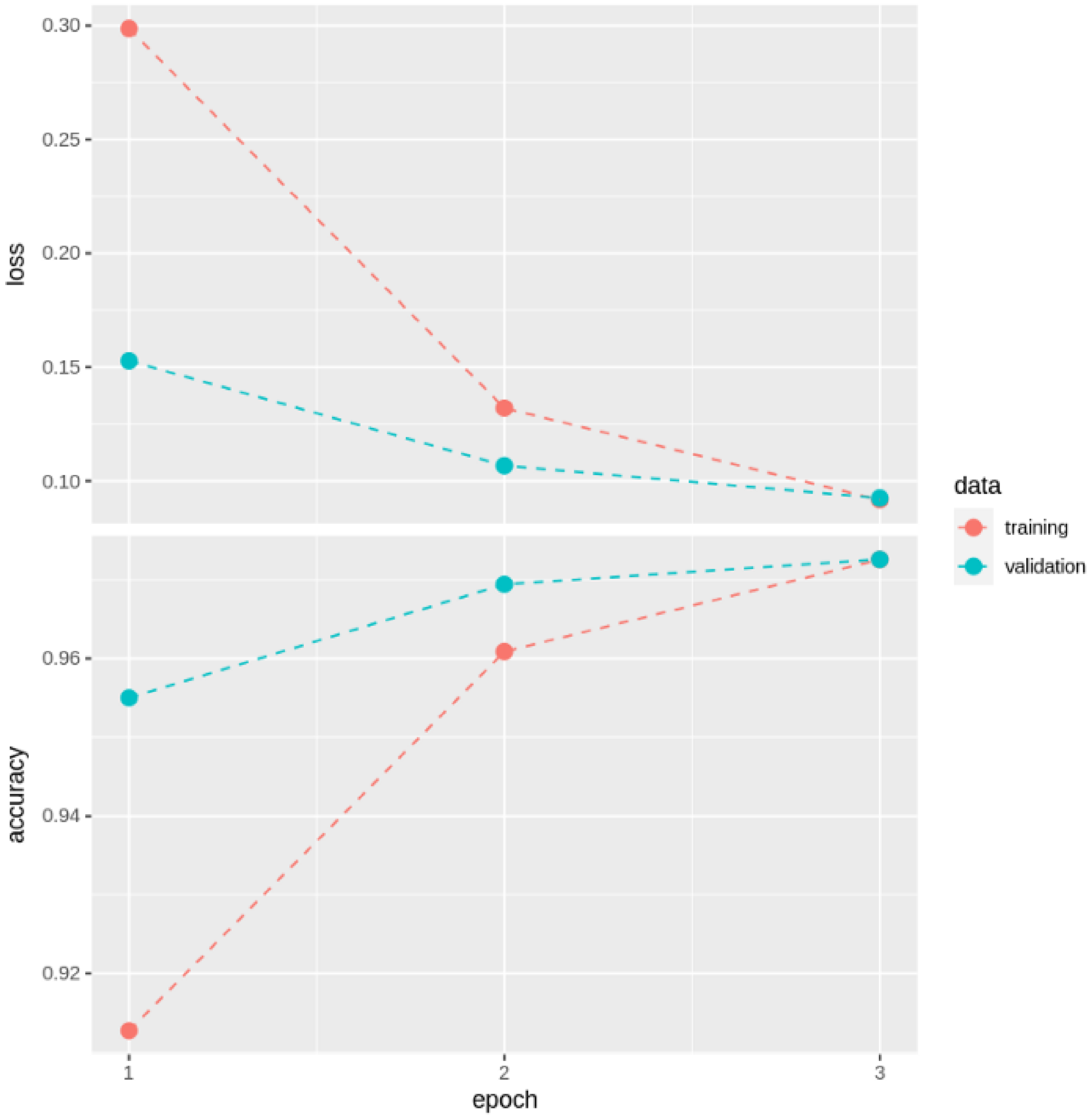
5.3 Fit

Fitting the model means finding a set of parameters that minimizes the loss function during training.

Input data is not processed as a whole. The model iterates over the training data in batches, each of size `batch_size`. An iteration over all the training data is called `epoch`. We must declare the number of epochs when fitting the model. After each epoch, the network updates its weights to minimize the loss.

```r
# fit the model
history <- model %>% fit(
  x_train_reshaped,
  y_train,
  epochs = 3,
  batch_size = 128,
  validation_split = 0.2
  geom_point(size = 3) +
  geom_line(linetype = "dashed")
```

[mnist_r] fit.r hosted with ❤ by GitHub view raw



Metrics during the training procedure. Image by author.

6. Test set performances

After training an estimator, it is a good practice to assess its performances on out-of-sample data. We can measure the accuracy (fraction of handwritten digits correctly classified) on the test set:

```
test_metrics <- model %>%
  evaluate(x_test_reshaped, y_test)

test_metrics["accuracy"] %>% round(., 3)
```

[mnist_r] test_acc.r hosted with ❤ by GitHub view raw

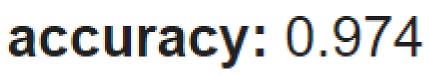
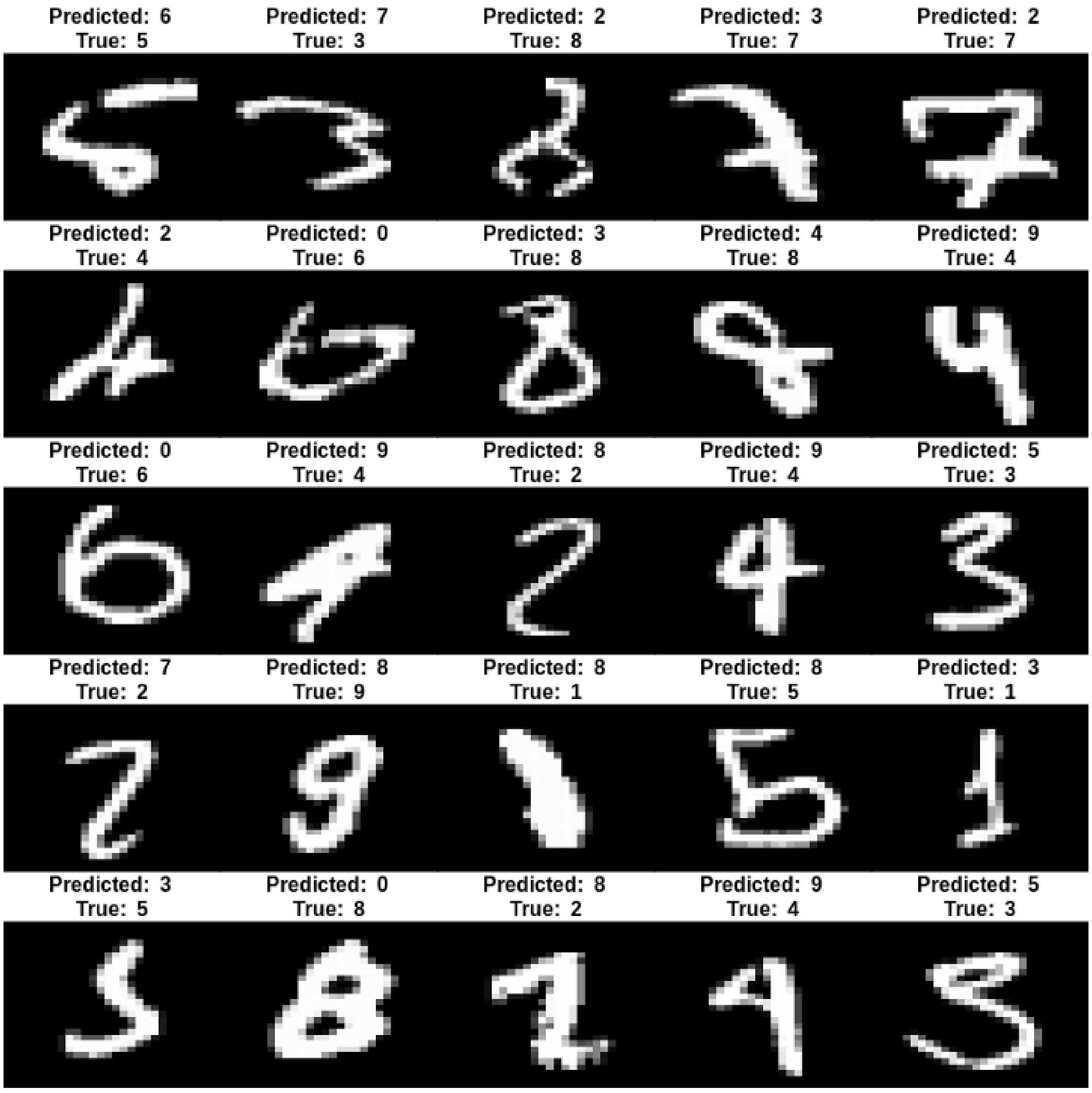

accuracy: 0.974

Image by author.

```
y_hat <- model %>%
  predict(x_test_reshaped) %>%
  k_argmax() %>%
  as.array()

# get test set real labels
y_obs <- y_test %>%
   k_argmax() %>%
   as.array()

# get misclassified samples indexes
misclass_idx = which(y_hat != y_obs)

# plot 25 misclassified digits
par(mfcol=c(5, 5))
par(mar=c(0, 0, 2.5, 0), xaxs = 'i', yaxs = 'i')

for (i in misclass_idx[0:25]) {
      img <- x_test[i,,]
      img <- t(apply(img, 2, rev))
      image(1:28, 1:28, img, col = gray((0:255)/255), xaxt = 'n', yaxt = 'n',
        main = paste("Predicted: ", y_hat[i] , "\nTrue: ", y_obs[i]))
}
```

[mnist_r] plot_misclass.r hosted with ❤ by GitHub view raw



Misclassified digits. Image by author.

7. Conclusions

language, that is generally not as commonly seen in Deep Learning tutorials or guides as Python.

Notably, some the two most popular Deep Learning frameworks, Torch and TensorFlow, support R as well.

It is possible to find more information and examples here:

- *TensorFlow for R*[6]
- *Torch for R*[7]
- Francois Chollet, J.J. Allaire, "*Deep Learning with R*", Manning, 2018[8].

8. References

[1] https://colab.research.google.com

[2] https://colab.research.google.com/notebook#create=true&language=r

[3] https://tensorflow.rstudio.com/install/

[4] https://en.wikipedia.org/wiki/MNIST_database

[5] https://github.com/keras-team/keras/blob/master/LICENSE

[6] https://tensorflow.rstudio.com/

[7] https://torch.mlverse.org/

[8] https://www.manning.com/books/deep-learning-with-r