

Lab 5: Image recognition

Fashion MNIST dataset

Isabel Casas

Objective

After this lab, you should have learnt:

1. To train and predict using MLP for classification
2. Convolutional neural networks, based on [Tutorial](#).

Instructions

Work in the lab computer instead of in Posit Cloud because of possible memory problems.

MLP for classification

Theoretical questions

1. Plot a one layer MLP and write its mathematical expression (on paper).
2. Which are the most common activation functions? Which is the most popular? Plot them.
3. When we say input layer in the context of MLP, what do we refer to? and output layer?
4. What do we mean with backpropagation in the context of ANN?

Classification problem: image recognition

This is an extension of Lecture 6 analysis of the MNIST dataset. The objective is to find the best model to classify number images. That is, to identify handwritten numbers from 0 to 9 well. This is a regression problem because (**your answer**). Use data from Lecture 6.

Exercise 1

Using the package *caret* train the following models to identify numbers in the images.

First, train the following models using the training dataset and answer the questions:

1. Multilayer perceptron with one layer. Which is the best number of nodes in the layer for this problem?
2. Multilayer perceptron with one layer. Which is the best learning rate and number of nodes/lauer for this problem?
3. Multilayer perceptron with 3 layers. which is the best number of nodes in the layer?

Second, using Lecture 5-6 notes, do the following:

1. Plot each of the three MLPs.
2. Predict values using data in your testing sample.
3. Show the confusion matrix of all of them.
4. Compare the three models prediction performance using th overall accuracy? Which model does best?

Convolutional neural networks

Theoretical questions

Research about the following questions:

- What is Deep Learning in the context of machine learning?
- What are convolutional neural networks?

Exercise 2: Install Tensorflow library in R

Deep learning functionality is programmed in the *tensorflow* and *keras* packages written in Python. A version of these packages exists now in R but we have to do the following installation:

- Install tensorflow and keras which are python libraries in R. This worked in the university computers.

```
install.packages(c('remotes'))
remotes::install_github("rstudio/tensorflow", force = TRUE)
library(tensorflow)
tensorflow::install_tensorflow()
library(tensorflow)
tf_config()
install.packages("keras")
```

- Install tensorflow and keras which are python libraries in R. This worked in my MAC but it does not work in the university computer.

```
install.packages(c('reticulate', 'devtools'))
reticulate::install_python()
library(tensorflow)
tensorflow::install_tensorflow()
library(tensorflow)
tf_config()
library(keras)
keras::install_keras()
```

Fashion dataset

- The MNIST fashion dataset consists of fashion product images of 28x28 pixels (see Figure 1).
- The original dataset is available in [Fashion MNIST](#).
- These original files are in IDX format and cannot be opened with basic text editors or functions *read.table()* and *read.csv()* in R. We would need function *file()* and some processing. This is outside the scope of this course, so instead, we are going to use the dataset in package *keras*

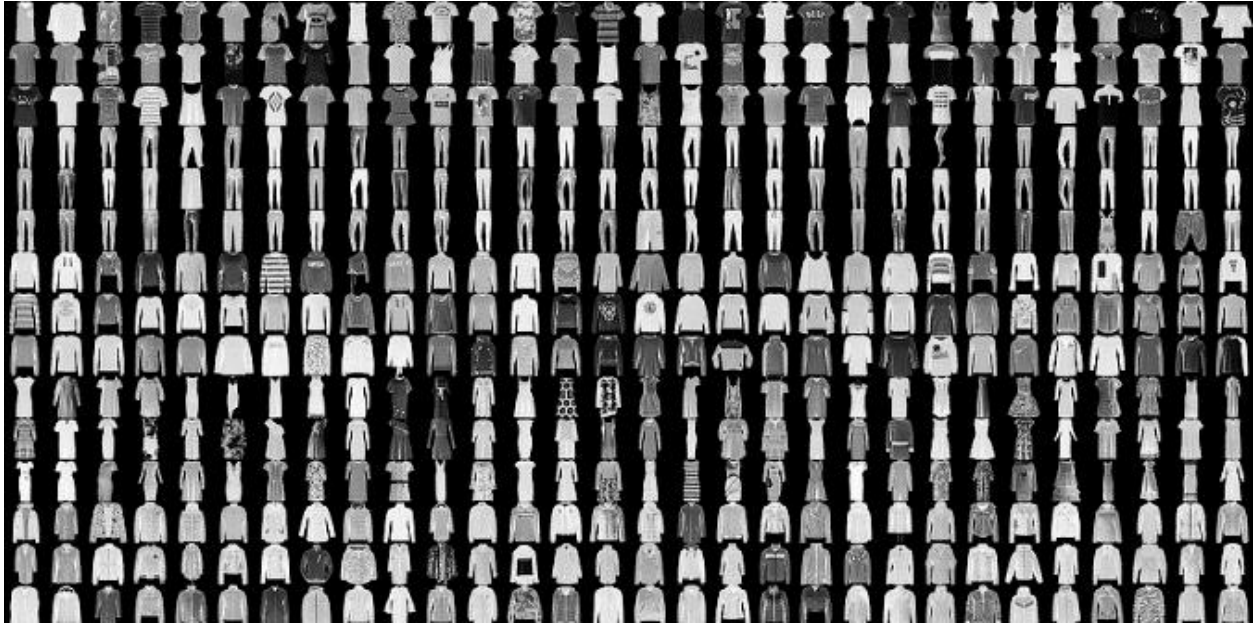


Figure 1: Sample of MNIST Fashion set.

Exercise 3: Obtaining data

With the code below, we load in our environment four different files: two training files with the predictors and labels and two test files.

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.2.3
```

```
fashion_mnist <- dataset_fashion_mnist()
x_train <- fashion_mnist$train$x
y_train <- fashion_mnist$train$y #labels
x_test <- fashion_mnist$test$x
y_test <- fashion_mnist$test$y #labels
#To ensure the machine knows it is categorical
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
label_names = c('T-shirt/top',
                 'Trouser',
                 'Pullover',
                 'Dress',
                 'Coat',
                 'Sandal',
                 'Shirt',
                 'Sneaker',
                 'Bag',
                 'Ankle boot')
```

Using R commands that you already know answer the following questions:

1. How many elements do we have for training? and for testing?
2. What is the class of your data? Hint: Use `class(x_train)`
3. What is the difference between *array* and *matrix*?
4. Look at the first 2 rows of *x_train*, the first 2 rows of *y_train*

```
class(x_train)
```

```
## [1] "array"
```

```
class(y_train)
```

```
## [1] "matrix" "array"
```

```
dim(x_train)
```

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

```
dim(y_train)
```

```
## [1] 60000    10
```

```
x_train[1, 20, 15]
```

```
## [1] 65
```

```
y_train[1:2,]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0    0    0    0    0    0    0    0    1
## [2,]    1    0    0    0    0    0    0    0    0    0
```

Exercise 4: Preprocess the data

We inspect the first image in the training set and we see it has values that range between 0 and 255 (the level of grey).

```
library(tidyr)
```

```
## Warning: package 'tidyr' was built under R version 4.2.3
```

```
library(ggplot2)
```

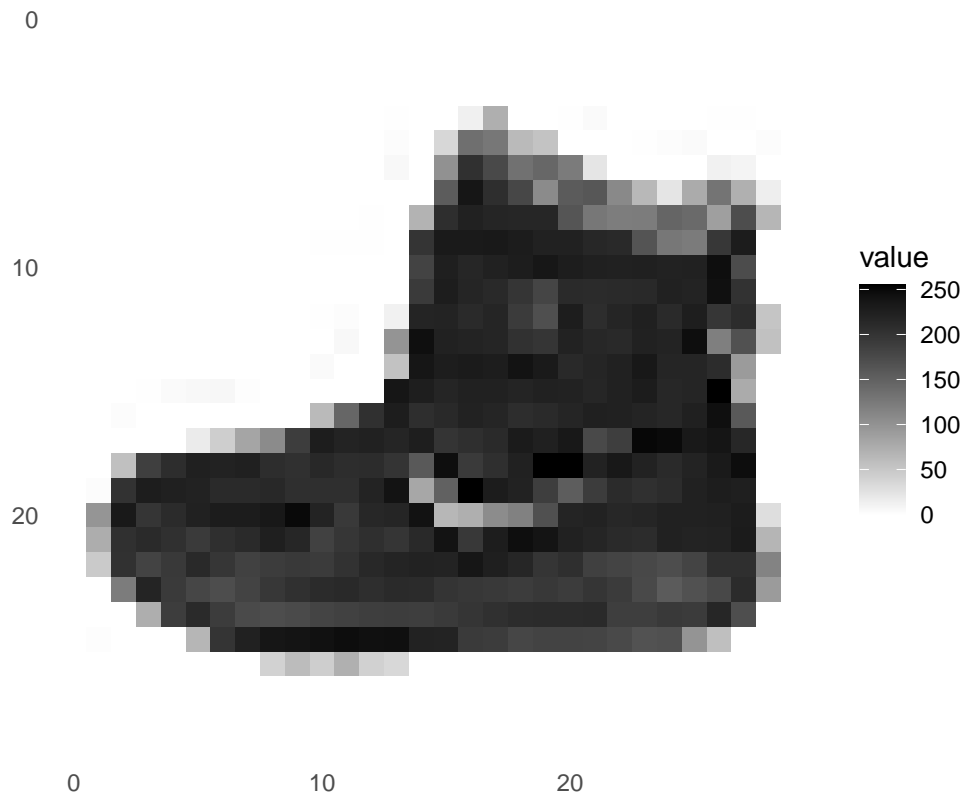
```
## Warning: package 'ggplot2' was built under R version 4.2.3
```

```

image_1 <- as.data.frame(x_train[1, , ])
colnames(image_1) <- seq_len(ncol(image_1))
image_1$y <- seq_len(nrow(image_1))
image_1 <- gather(image_1, "x", "value", -y)
image_1$x <- as.integer(image_1$x)

ggplot(image_1, aes(x = x, y = y, fill = value)) +
  geom_tile() +
  scale_fill_gradient(low = "white", high = "black", na.value = NA) +
  scale_y_reverse() +
  theme_minimal() +
  theme(panel.grid = element_blank()) +
  theme(aspect.ratio = 1) +
  xlab("") +
  ylab("")

```



It is a good practice to scale variables in classification problems, that is to scale them to values between 0 and 1 before feeding the data to the neural network model. In fact, data normalisation is important for training NNs because it scales the input data to a range where the model can learn more effectively. Without normalisation, features with larger magnitudes may dominate the learning process, causing convergence issues and slower training.

In our problem, we simply divide every value of the training and testing predictors datasets by 255. It is important that the training and testing set are preprocessed in the same way.

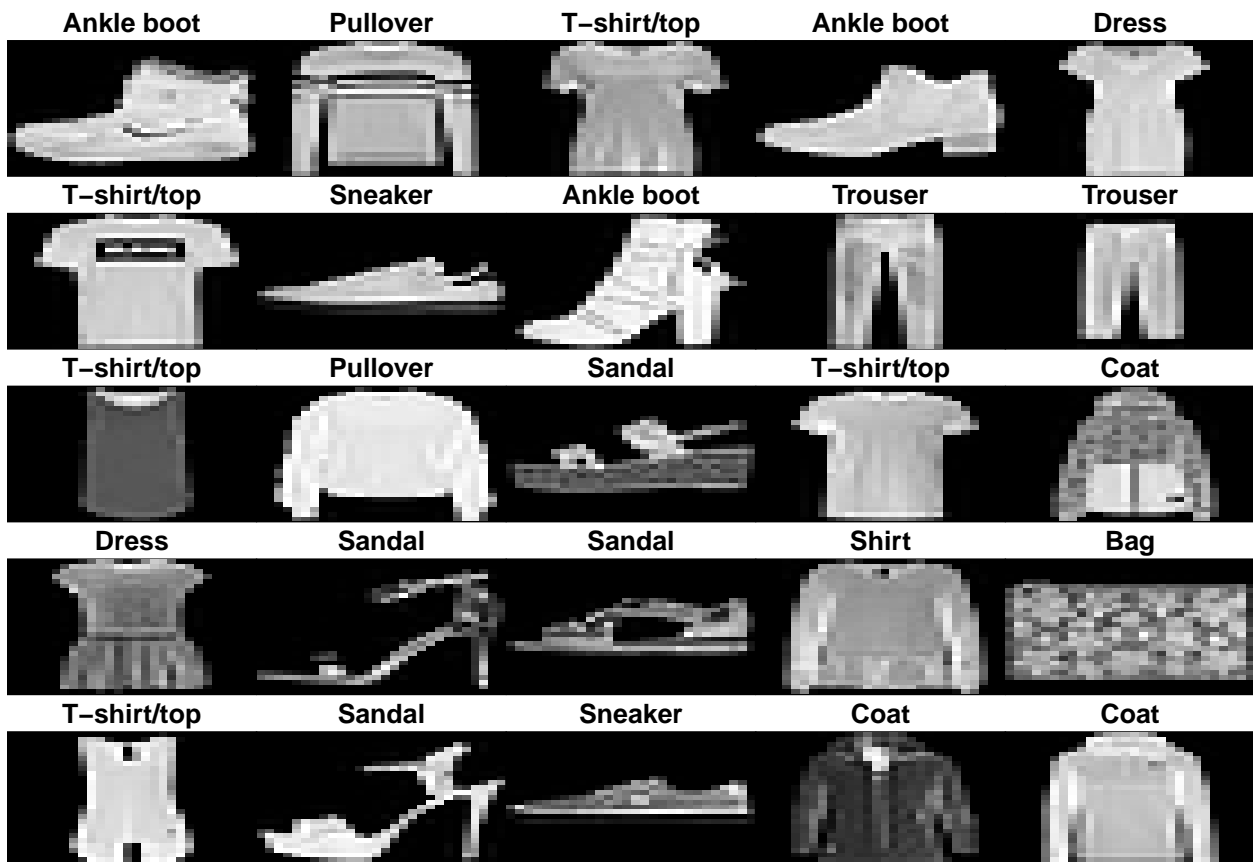
```
x_train <- x_train / 255
x_test  <- x_test  / 255
```

- Look at the first 2 rows of x_train and the first 10 values of y_train . Are they different than before?
- Why don't we divide y_train and y_test by 255?

Displaying the data

Display the first 25 images from the training set and display the class name on top of each image. Verify that the data is in the correct format and we're ready to build and train the network.

```
par(mfcol=c(5,5))
par(mar=c(0, 0, 1.5, 0), xaxs='i', yaxs='i')
for (i in 1:25) {
  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', main = paste(label_names[which.max(y_train[i,])]))
}
```



Exercise 5

Display the last 25 images of the training set.

```
# You code here
```

Create the blueprint

Most of deep learning consists of chaining together simple layers. Most layers, like `layer_dense`, have parameters that are learned during training. We are going to create a simple model with the following architecture:

$$\text{Input} \rightarrow \text{Flatten} \rightarrow \text{FC} \rightarrow \text{FC}$$

```
x_train <- array_reshape(x_train, c(nrow(x_train), 28, 28, 1))
x_test  <- array_reshape(x_test,  c(nrow(x_test), 28, 28, 1))

model <- keras_model_sequential()
model %>%
  layer_flatten(input_shape = c(28, 28, 1)) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')
```

1. `layer_flatten`, transforms the format of the images from a 2d-array (of 28x28 pixels), to a 1d-array of $28 * 28 = 784$ pixels. This layer has no parameters to learn; it only reformats the data.
2. `layer_dense` creates a fully connected layer which is similar to the MLP with $\sigma(\sum(w_i x_i) + b_i)$. The first dense layer has 128 nodes (or neurons) in the hidden layer and RELU as the activation function. The second layer is a 10-node layer with a softmax activation function. This returns an array of 10 probability scores that sum to 1 (we have 10 labels so the last fully connected layer much has 10 neurons). Each node contains a score that indicates the probability that the current image belongs to one of the 10 digit classes.
3. The `softmax` activation function in the last layer is used to produce probability distributions over the output classes, making it suitable for multi-class classification problems.

Exercise 6: define the model architecture

In the chunk below, create a new CNN model called `model2` with the following architecture:

$$\text{Input} \rightarrow \text{Conv} \rightarrow \text{Pool} \rightarrow \text{Flatten} \rightarrow \text{FC} \rightarrow \text{FC}$$

So you can reuse the code above for the Flatten and last two FC layers, but you have to use the following functions:

- `layer_conv_2d`. You can see how this function work using `?layer_conv_2d`. We want that the Convolutional layer of `model2` has 32 filters of size 3x3 each, activation function “relu”. You must write `input_shape = c(28,28,1)` because our data is of size 28x28 pixels.
- `layer_max_pooling_2d`. We would like to have a `pool_size = c(2,2)`.

```
# Define model2's architecture
```

Compile the model

Before the model is ready for training, it needs a few more settings. These are added during the model's compilation step:

- **Loss function** which measures how accurate the model is during training. We want to minimize this function to “steer” the model in the right direction. That is to choose the weights with the smaller fit.
- **Optimizer**. This how the model is updated based on the data it sees and its loss function.
- **Metrics**. This measures how well the estimation and prediction is done. We use “accuracy” here because this is a classification example.

```
model %>% compile(  
  optimizer = 'adam',  
  loss = 'categorical_crossentropy',  
  metrics = c('accuracy')  
)
```

Exercise 7

Compile *model2* using “categorical_crossentropy” as the loss function and *optimizer_rmsprop()* as the optimizer.

```
# Compile model2
```

Train the model

Training the neural network model requires the following steps:

- Feed the training data to the model — in this example, the *x_train* and *y_train* arrays.
- The model learns to associate images and labels.

To start training, feed your model into the **fit()** method including the training datasets. This is the part that takes the longest. The more *epochs*, the more iterations. See the plots on the side with the loss and accuracy functions.

```
model %>% fit(x_train, y_train, epochs = 5, verbose = 2)
```

```
## Epoch 1/5  
## 1875/1875 - 5s - loss: 0.4947 - accuracy: 0.8253 - 5s/epoch - 3ms/step  
## Epoch 2/5  
## 1875/1875 - 5s - loss: 0.3721 - accuracy: 0.8642 - 5s/epoch - 3ms/step  
## Epoch 3/5  
## 1875/1875 - 5s - loss: 0.3375 - accuracy: 0.8781 - 5s/epoch - 2ms/step  
## Epoch 4/5  
## 1875/1875 - 4s - loss: 0.3148 - accuracy: 0.8852 - 4s/epoch - 2ms/step  
## Epoch 5/5  
## 1875/1875 - 5s - loss: 0.2945 - accuracy: 0.8913 - 5s/epoch - 2ms/step
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.89 (or 89%) on the training data.

Exercise 8

Train *model2* using 10 epochs and *validation_split= 0.2* (uses 20% of the data for validation and 80% for training. The model is tested at the end of each epoch only using the validation data.

```
# Train model2
```

Questions:

1. Looking at the loss function and accuracy graphics, what do you think is the optimal number of epochs?
2. What is the difference between loss function and accuracy? Do they measure the same thing?
3. Why are the values of the training and validation sets different? Which should be pay attention to to make decisions about the hyperparamters of the model?

Evaluate accuracy

Next, compare how the model performs on the test dataset:

```
score <- model %>% evaluate(x_test, y_test, verbose = 0)
cat('Test loss of model:', score["loss"], "\n")
```

```
## Test loss of model: 0.3622209
```

```
cat('Test accuracy of model:', score["accuracy"], "\n")
```

```
## Test accuracy of model: 0.8705
```

Exercise 9: Evaluate model2

Question: Which model does best at predicting?

```
# Evaluate model2
```

Make predictions

- We ask the model to make predictions about a training set, using the previously trained -> evaluation
- Or make predictions about a test set, using the previously trained model -> prediction

```
# predicition with train data
estimation <- model %>% predict(x_train)
```

```
## 1875/1875 - 2s - 2s/epoch - 1ms/step
```

```
# prediction test data
predictions <- model %>% predict(x_test)
```

```
## 313/313 - 0s - 351ms/epoch - 1ms/step
```

Here, the model has predicted the label for each image in the testing set. Let's take a look at the first prediction:

```
predictions[1, ]
```

```
## [1] 1.352883e-05 6.253087e-08 6.219402e-08 6.197471e-08 2.084245e-07
## [6] 2.115780e-02 4.149726e-06 2.363384e-01 1.809753e-04 7.423047e-01
```

predictions is an array of 10 numbers (probabilities) for each label. These describe the “confidence” of the model that the image corresponds to each of the 10 different articles of clothing for a particular row. We can see which label has the highest confidence value and which clothing has we predicted for the first image in the test file:

```
index <- which.max(predictions[1, ])
label_names[index]
```

```
## [1] "Ankle boot"
```

The true clothing label is the value 1 in *y_test* of a given row. We see that the first image of the test dataset is “Ankle boot”.

```
dim(y_test)
```

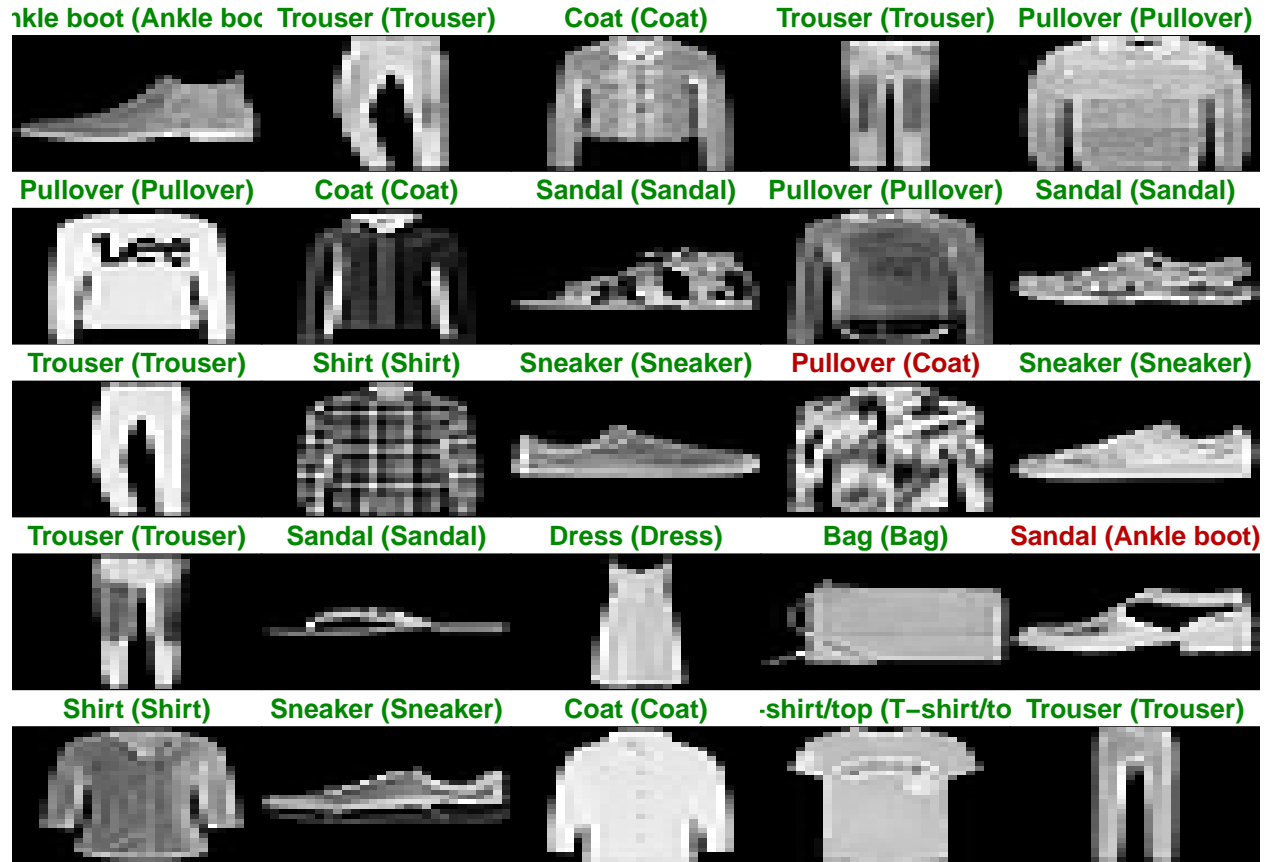
```
## [1] 10000    10
```

```
label_names[which(y_test[1,]==1)]
```

```
## [1] "Ankle boot"
```

Let's plot several images with their predictions. Correct prediction labels are green and incorrect prediction labels are red.

```
# We need to reshape these datasets to be able to use image()
x_train <- array_reshape(x_train, c(nrow(x_train), 28, 28))
x_test  <- array_reshape(x_test,  c(nrow(x_test),  28, 28))
par(mfcol=c(5,5))
par(mar=c(0, 0, 1.5, 0), xaxs='i', yaxs='i')
for (i in 1:25) {
  img <- x_test[i, , ]
  img <- t(apply(img, 2, rev))
  predicted_label <- which.max(predictions[i, ])
  true_label <- which.max(y_test[i,])
  if (predicted_label == true_label) {
    color <- '#008800'
  } else {
    color <- '#bb0000'
  }
  image(1:28, 1:28, img, col = gray((0:255)/255), xaxt = 'n', yaxt = 'n',
        main = paste0(label_names[predicted_label], " (",
                      label_names[true_label], ")"),
        col.main = color)
}
```



Finally, use the trained model to make a prediction about a single image. In this case the first one.

```
# Grab an image from the test dataset
# take care to keep the batch dimension, as this is expected by the model
img <- x_test[1, , , drop = FALSE]
dim(img)
```

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

Now predict the image:

```
predictions <- model %>% predict(img)
```

```
## 1/1 - 0s - 94ms/epoch - 94ms/step
```

```
predictions
```

```
##           [,1]           [,2]           [,3]           [,4]           [,5]           [,6]
## [1,] 1.352879e-05 6.253087e-08 6.219391e-08 6.19746e-08 2.084241e-07 0.02115778
##           [,7]           [,8]           [,9]          [,10]
## [1,] 4.149726e-06 0.2363384 0.0001809754 0.7423047
```

predict() returns a list of probabilities for each image in the batch of data. For each image, **predict()** assigns probabilities at one of the 10 labels. The predicted image is the one of the highest probability.

```
prediction <- which.max(predictions[1, ])  
prediction
```

```
## [1] 10
```

```
label_names[prediction]
```

```
## [1] "Ankle boot"
```

And, as before, the model predicts a label of 9.

Exercise 10: Predict with model2

Do the same than above and predict image in the 10th position of the test file.

```
# Predict model2
```