FashionMNIST-training

January 2, 2019

```
In [157]: library(keras)  # FashionMNIST dataset
          library(nnet)  # Neural networks
          library(caret)  # Cross Validation - loads nnet directly on trainControl

          library(doMC)   # Parallel cross-validation
```

1 Dataset

First we load the dataset from keras package. *Check legacyLoad.R to see how to load the dataset without using the package.*

```
In [2]: fashion <- dataset_fashion_mnist()
```

```
In [3]: str(fashion)
        attach(fashion) # So we can access test and train directly!
```

```
List of 2
 $ train:List of 2
  ..$ x: int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ y: int [1:60000(1d)] 9 0 0 3 0 2 7 2 5 5 ...
 $ test :List of 2
  ..$ x: int [1:10000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ y: int [1:10000(1d)] 9 2 1 1 6 1 4 6 5 7 ...
```

We get the following structure:

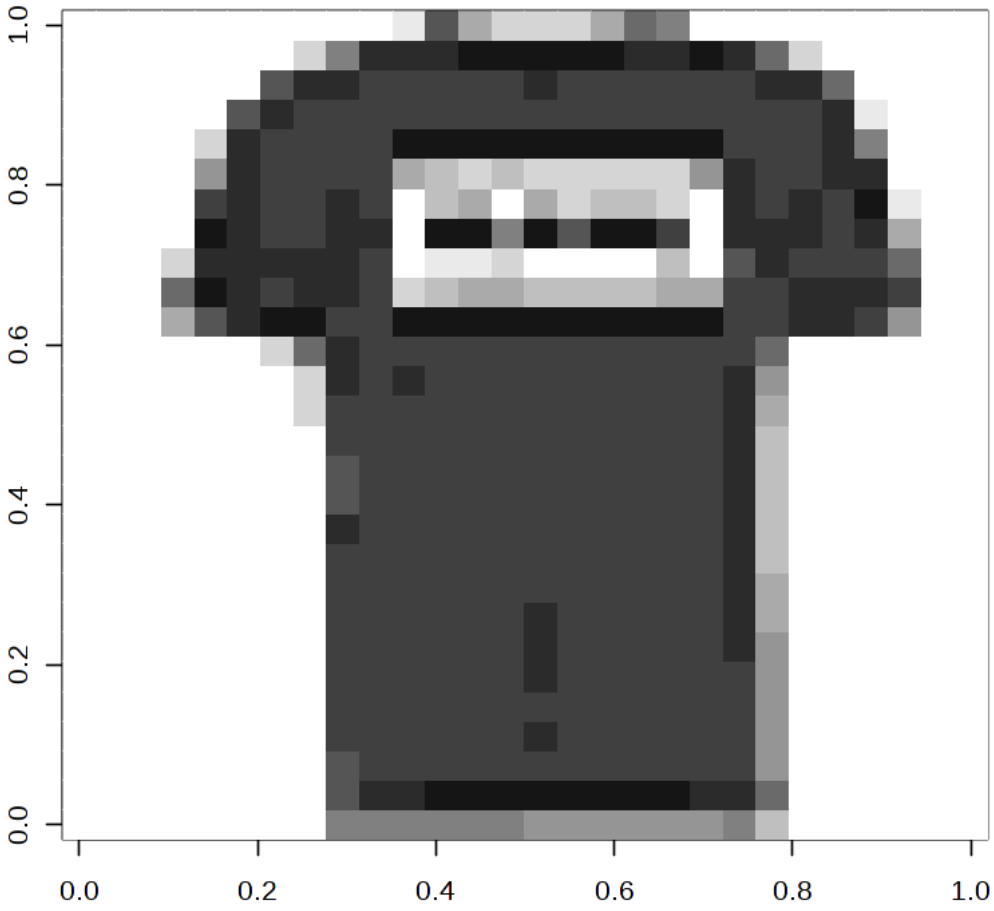
- train: Training dataset
  - x: the predictors, 28x28 pixels image in grayscale.
  - y: the response
- test: Testing datset (with x and y)

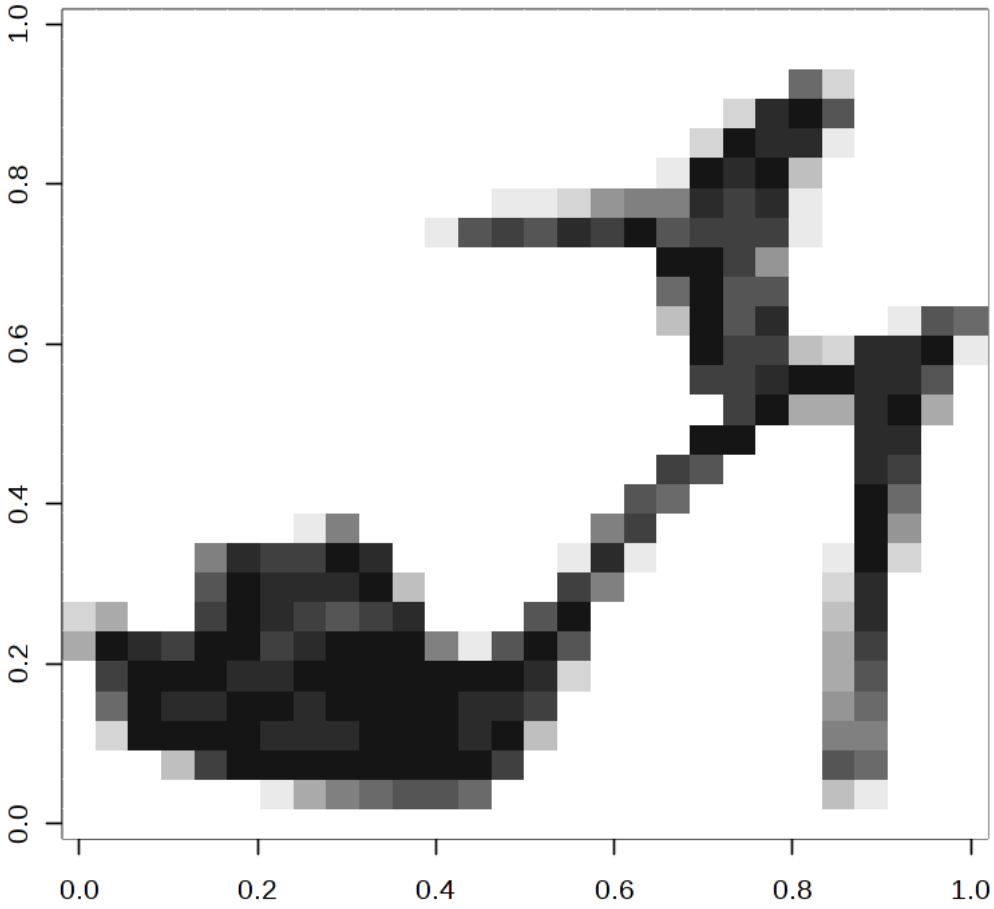
We can see the images with the following function:

1.1 dataset visualization

In [4]:
```
rotate <- function(x) t(apply(x, 2, rev))
show_image <- function(imgarray, col=gray(12:1/12), ...) {
  image(rotate(matrix(imgarray, nrow=28)), col=col, ...)
}

show_image(train$x[2,,])
show_image(train$x[10,,])
```





1.2 Response reencode

Notice that in y we have an integer from 0 to 9 (10 classes). They are in fact the following: - 0: T-shirt/top - 1: Trouser - 2: Pullover - 3: Dress - 4: Coat - 5: Sandal - 6: Shirt - 7: Sneaker - 8: Bag - 9: Ankle boot

We recode the response variable to factor.

```
In [5]: classString <- c("T-shirt/top","Trouser", "Pullover", "Dress", "Coat", "Sandal",
                       "Shirt","Sneaker", "Bag","Ankle boot")

        # y+1 because 0 is the first class and in R we start indexing at 1!
        train$yFactor <- as.factor(classString[train$y+1])
        test$yFactor <- as.factor(classString[test$y+1])
```

For the CNN we use one hot encoding to produce a vector of 10 values per sample, with a one on the class (probablity of belonging to a given class).

```
In [119]: train$yOneHot <- class.ind(train$yFactor)
          test$yOneHot <- class.ind(test$yFactor)
```

```
In [120]: str(train$yOneHot)
          train$y[1:10]
          train$yOneHot[1:10,]
```

```
 num [1:60000, 1:10] 1 0 0 0 0 0 0 0 0 0 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:10] "Ankle boot" "Bag" "Coat" "Dress" ...
```

1. 9 2. 0 3. 0 4. 3 5. 0 6. 2 7. 7 8. 2 9. 5 10. 5

Ankle boot	Bag	Coat	Dress	Pullover	Sandal	Shirt	Sneaker	T-shirt/top	Trouser
1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0

class.ind reorders the classes alfabetically, therefore we need to revert this order to the original provided. We use *match* over the column names to get a vector of the reorder to match the column names to **classString**.

```
In [121]: colnames(train$yOneHot)
          classString
          (m <- match(classString, colnames(test$yOneHot)))
```

1. 'Ankle boot' 2. 'Bag' 3. 'Coat' 4. 'Dress' 5. 'Pullover' 6. 'Sandal' 7. 'Shirt' 8. 'Sneaker' 9. 'T-shirt/top' 10. 'Trouser'

1. 'T-shirt/top' 2. 'Trouser' 3. 'Pullover' 4. 'Dress' 5. 'Coat' 6. 'Sandal' 7. 'Shirt' 8. 'Sneaker' 9. 'Bag' 10. 'Ankle boot'

1. 9 2. 10 3. 5 4. 4 5. 3 6. 6 7. 7 8. 8 9. 2 10. 1

```
In [122]: train$yOneHot <- train$yOneHot[,m]
          test$yOneHot <- test$yOneHot[,m]
```

Now the order is correct

```
In [124]: colnames(train$yOneHot)
          classString
```

1. 'T-shirt/top' 2. 'Trouser' 3. 'Pullover' 4. 'Dress' 5. 'Coat' 6. 'Sandal' 7. 'Shirt' 8. 'Sneaker' 9. 'Bag' 10. 'Ankle boot'

1. 'T-shirt/top' 2. 'Trouser' 3. 'Pullover' 4. 'Dress' 5. 'Coat' 6. 'Sandal' 7. 'Shirt' 8. 'Sneaker' 9. 'Bag' 10. 'Ankle boot'

1.3 Add missing dimension

Convolutional layers will expect the input to have 4 dimensions: - Sample dimension - Height dimension - Width dimension - Channel dimension

In our case we have only one channel as the image is grayscale. If it's a color image we would have 3 or 4 channels (Red, Green, Blue and Alpha (transparency)). We need to add the missing dimension, however this will not modify the data.

```
In [22]: dim(train$x) <- c(dim(train$x),1)
         dim(test$x) <- c(dim(test$x),1)
```

1.4 Create a dataset for nnet

Now we prepare join the X and the Y in a data.frame.

```
In [6]: nnetData <- data.frame(train$x, class=train$yFactor)
```

2 Training a Neural Network

We can train the model directly as follows, but we will use *caret's trainControl* for CrossValidation.

```
In [159]: model.nnet <- nnet(class ~ ., data=nnetData, size=50, maxit=300,decay=0.5, MaxNWts =

# weights:  39760
```

Specifically a 5 fold cross-validation. We don't go for a 10 fold cross-validation as it will take a lot of time to compute.

```
In [ ]: ## specify 5-CV
        K <- 5
        trc <- trainControl (method="repeatedcv", number=K, repeats=1)
        (decays <- 10^seq(-3,0,by=0.25))
```

We now specify that we want to execute the cross validation in parallel:

```
In [ ]: # Use all cores except one (recommended if you want to use your computer for something
        cores <- min(detectCores()-1, ceiling(length(decays)/2))
        registerDoMC(cores = cores)
```

Beware with the number of cores used, it will impact in the RAM usage. ~10GB per thread with 60K Fashion MNIST samples. **Don't execute this training if you don't have a big machine, just load the model.**

The cross-validation process will take about 30 hours using 7 cores of a Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz and about 80 GB of RAM.

Remember that we're training (number_of_decay_param x number_of_folds) = 14 x 5 = 70 models.

```
In [ ]: ## WARNING: this takes some time
        model.5CV <- train (class ~ ., data=nnetData, method='nnet', maxit = 300, trace = FALSE
                            tuneGrid = expand.grid(.size=50,.decay=decays), trControl=trc, Ma

In [ ]: # Save model
        save(model.5CV, file="nnet.mod")

In [ ]: load("nnet.mod")

In [ ]: model.5CV
```

The best model we got has an accuracy of 84%. Not bad at all for a 10 class classification problem.

3 Convolutional Neural Networks

3.1 Model architecture definition: LeNet

Now we have to define the CNN architecture. In this case we use LeNet, proposed by LeCun et al. (Gradient-based learning applied to document recognition. Proceedings of the IEEE, november 1998).

It is composed by two packs of convolutional-activation(tanh)-pooling layers and two fully connected layers with a softmax layer at the end.

In Keras, as in most of the packages, we define layers as objects and the connections between those objects. In this case we implicitly connect everything using the %>% operator.

```
In [132]: lenet <- keras_model_sequential() %>%
              # First convolutional block
              layer_conv_2d(filters=20, kernel_size=c(5,5), activation="tanh",
                  input_shape=c(28,28,1), padding="same") %>%  # We define here the input size
              layer_max_pooling_2d(pool_size=c(2,2),strides=c(2,2)) %>%
              # Second convolutional block
              layer_conv_2d(filters=50, kernel_size=c(5,5), activation="tanh",
                  input_shape=c(28,28,1), padding="same") %>%
              layer_max_pooling_2d(pool_size=c(2,2),strides=c(2,2)) %>%

              # Flatten the matrix to a vector for the fully connected layers
              layer_flatten() %>%

              # First fully connected block
              layer_dense(units=500, activation="tanh") %>%
              # Second fully connected block
              layer_dense(units=10, activation="softmax")
              # This last layer will produce the final classification (probability of
              # belonging to a class). 10 different units, 10 different classes.
```

Now we check the architecture we have defined:

```
In [133]: lenet
```

```
Model
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_7 (Conv2D)            (None, 28, 28, 20)        520
_________________________________________________________________
max_pooling2d_7 (MaxPooling2D)  (None, 14, 14, 20)     0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 14, 14, 50)        25050
_________________________________________________________________
max_pooling2d_8 (MaxPooling2D)  (None, 7, 7, 50)       0
_________________________________________________________________
flatten_4 (Flatten)          (None, 2450)              0
_________________________________________________________________
dense_7 (Dense)              (None, 500)               1225500
_________________________________________________________________
dense_8 (Dense)              (None, 10)                5010
=================================================================
Total params: 1,256,080
Trainable params: 1,256,080
Non-trainable params: 0
_________________________________________________________________
```

Notice that we're adjusting 1 million parameters this time. With the nnet one layer network we were training just 39.760 parameters.

Last thing we have to do is to specify which optimizaton algorithm and metrics we want to use with the compile step.

```
In [134]: sgd <- optimizer_sgd(
                                lr=0.05,
                                decay=0.001,
                                momentum=0.8,
                                clipnorm=1.
          )
          lenet %>% compile(optimizer=sgd,
                        loss='categorical_crossentropy',
                        metrics = "accuracy"
          )
```

3.2 Model training

Now we're going to train the network using CPU (if you're not using tensorflow-gpu). Mind that if you want to use GPUs you need to have the GPU version of the package and the required Nvidia packages (check PlaidML for non-Nvidia GPUs).

```
In [135]: lenet %>% fit(
                    train$x,
                    train$yOneHot,
                    batch_size=50,
                    epochs=10
          )
```

Takes about 3 minuts with 40 cores and 18 GB of RAM. It may take less with GPUs!
And now we save the trained model for convenience:

```
In [136]: lenet %>% save_model_hdf5("lenet-FashionMNIST.h5")
```

3.3 Predicting using the model

Predicting the label for the test set

```
In [137]: pred_prob <- predict(lenet, test$x)
```

```
In [138]: head(pred_prob)
```

3.669661e-07	1.561574e-08	2.946072e-07	1.723674e-07	1.170943e-06	1.056668e-02	3.936590e-06	6.
7.463163e-04	3.227407e-07	9.964588e-01	7.566429e-07	1.791673e-03	2.960686e-07	9.972482e-04	5.
2.184556e-05	9.998372e-01	2.317742e-06	6.979462e-05	3.218573e-05	8.763134e-07	9.698024e-06	1.
2.179830e-06	9.999895e-01	2.741182e-08	6.444282e-06	1.694595e-06	3.628748e-08	1.443464e-07	3.
2.972747e-02	1.438419e-05	6.722302e-03	4.089017e-03	1.156198e-01	1.952142e-07	8.435847e-01	8.
2.682133e-04	9.993576e-01	6.971598e-05	5.819946e-05	2.138864e-04	2.190031e-06	1.985807e-05	1.

For each element we get the probability of that element to be of each class, therefore we search for the value that is maximum in each row and then we create the confusion matrix.

```
In [155]: predClass <- apply(pred_prob,1,which.max)
          predClass <- classString[predClass] # And change the integers by their class tag


          trueClass <- test$yFactor

          # Now we do a confusion matrix and analyze it
          (cMatrix <- table(trueClass,predClass))
```

```
             predClass
trueClass     Ankle boot Bag Coat Dress Pullover Sandal Shirt Sneaker
  Ankle boot         968   1    0     0        0      6     0      25
  Bag                  1 980    3     4        0      2     4       1
  Coat                 0   1  868    20       58      0    51       0
  Dress                1   2   39   903       12      0    16       0
  Pullover             0   1  113     9      803      0    55       0
  Sandal               5   0    0     0        0    983     0      12
  Shirt                0  11   91    29       66      0   655       0
  Sneaker             30   0    0     0        0     20     0     950
  T-shirt/top          0   6    7    18       15      2    53       0
```

```
  Trouser              0   2    4    10        0      0     1       0
```

```
            predClass
trueClass     T-shirt/top Trouser
  Ankle boot            0       0
  Bag                   2       3
  Coat                  1       1
  Dress                17      10
  Pullover             19       0
  Sandal                0       0
  Shirt               145       3
  Sneaker               0       0
  T-shirt/top         897       2
  Trouser               2     981
```

```
In [158]: correctClass <- sum(diag(cMatrix))
          total <- sum(cMatrix)
          (accuracy <- correctClass/total)
```

0.8988

We're getting about a 90% of accuracy that may be improved with further tunning of the network. Notice that, for example, there are 30 ankle boots classified as sneakers, that can have similar shapes. Also there are 113 pullovers are classified as coats.