# Projekt

*20 nov 2019*

## Comparing trained and pretrained models on Image Classification for multiple classes

### Image classification for multiple classes

In this project, we are focusing on dealing with image classification, which refers to a process in computer vision that can classify an image according to its visual content. Over the last years, it has grown effective due to it using deep learning. Deep learning is a class of machine learning that uses multiple layers to extract higher level features. Some expamples on its usage are Face Recognition on social platforms like Facebook or Product Discoverability which allows consumers to search for similar images or products.

### Pretrained models

Keras has 10 pretrained models, for example VGG19 and ResNet50, which are trained on ImageNet, a large collection of images in 1000 classes, and the pretrained models should be able to classify an image that falls into any of these 1000 classes. An image classifier contains convolutional and classifier layers. The convolutional layers extract features and classifier layers classify them using these features. Pretrained models already know how to extract features, and therefore you don't need to train it from scratch. Therefore, it is a possiblity, but not certain, that they will perform better than models we train from scratch.

We have choosen to work with two of the "simpler" pretrained models, here the VGG19 model and the ResNet50, but you can easily use different models from the Keras library as the procedure is the same. The VGG19 model is a CNN that is trained on more than one million images from ImageNet and has 19 layers deep. The ResNet50 model

### The data set

In this project we're making a guide on image classification in R. We're working with the Fruit data set from Kaggle, found on https://www.kaggle.com/moltean/fruits/data. Here 120 different fruits and vegetables are stored in different classes. The data set has 82213 images of fruit and vegetables. The data set is split into a training set containing 61488 images and a test set containing 20622 images. The images sizes are 100x100 pixels.

In the data set there was a difference in lighting conditions, the background was not uniform and beforehand there has been written a dedicated algorithm which extracts the fruit from the background, so the fruit and vegetables are seen on the images with a white background.

### CASE: Comparing CNN and pretrained models for multiple classes

In this section, we will compare a CNN with two pretrained models, here

#### An example using CNN

#### Load the data

We start by cleaning the global environment.

And then we'll import Keras, which is essential for the anaysis. You have to use "install_keras" if you're installing Keras for the first time.

```
devtools::install_github("rstudio/keras", force = TRUE)
```

```
## Downloading GitHub repo rstudio/keras@master
```

```
##
##
```

```
   checking for file 'C:\Users\Emma\AppData\Local\Temp\RtmpmOUkC9\remotes34ac405d1e05\rstudio-keras-e78
```

```
   checking for file 'C:\Users\Emma\AppData\Local\Temp\RtmpmOUkC9\remotes34ac405d1e05\rstudio-keras-e78
```

```
v  checking for file 'C:\Users\Emma\AppData\Local\Temp\RtmpmOUkC9\remotes34ac405d1e05\rstudio-keras-e78
##
```

```
-  preparing 'keras': (14.9s)
##    checking DESCRIPTION meta-information ...
```

```
   checking DESCRIPTION meta-information ...
```

```
v  checking DESCRIPTION meta-information
##
```

```
-  checking for LF line-endings in source and make files and shell scripts (7.1s)
##
```

```
-  checking for empty or unneeded directories
##
```

```
Removed empty directory
```

```
   Removed empty directory 'keras/man-roxygen'
##
```

```
-  building 'keras_2.2.5.0.tar.gz'
##
```

```
##
```

```
## Installing package into 'C:/Users/Emma/Documents/R/win-library/3.6'
```

```
## (as 'lib' is unspecified)
```

```
library(keras)
#install_keras()
```

And then we'll load a bunch of other packages.

### Preprocessing

And then we can load the data the data from Kaggle. We just loaded it locally, downloading and unpacking it from Kaggle.

Then we need to define a vector of the fruits and vegetables we want to train our model to classify, as we're not going to train it to classify 120, therefore we choose 16 totally random fruits and veetables.

```
FV_list = c("Quince","Tangelo","Cactus fruit","Physalis","Pepino","Orange","Blueberry","Mulberry", "Kaki

length = length(FV_list)
```

Then we're using the function "image_data_generator", which can load and generate batches of the image data. Here we're first rescaling the image data. For most image data, the pixel values are integers with values between 0 and 255, which is why dividing the data with 255. This is performed across all channels, regardless of the actual range of pixel values that are present in the images.

```
train_images_rescale = image_data_generator(rescale = 1/255)

test_images_rescale = image_data_generator(rescale = 1/255)
```

We now want to load our images into the memory and resize them. To do this the 'flow_images_from_directory' function from the Keras package will come in use. It allows us to generate batches of data from images in a directory. We start by setting the 'class_mode' to categorical, due to the data list we created earlier. The same applies for the which "classes", we are refering back to. At last we the set the seed to 31, as it allows us to reproduce the same results later.

```
train_image_array <- flow_images_from_directory(train_images,
                                                train_images_rescale,
                                                target_size = c(64,64),
                                                class_mode = "categorical",
                                                classes = FV_list,
                                                seed = 31)


test_image_array <- flow_images_from_directory(test_images,
                                                test_images_rescale,
                                                target_size = c(64,64),
                                                class_mode = "categorical",
                                                classes = FV_list,
                                                seed = 31)
```

We can now explore how many images we have in each of our list of fruits.

```
cat("Number of images per class:")
```

```
## Number of images per class:
```

```
table(factor(train_image_array$classes))
```

```
##
##   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
```

```
## 490 490 490 492 490 479 462 492 490 490 468 490 450 492 427 490
```

As the table show, there is a nice distrubtion of number of images in every classes. That's good so we dont have a group of classes which is over represented over others classes.

### Defining the model

Now it's time to set up the model, that should be able to predict the images.

We are now ready for creating a model. We'll start of with a simple Convolutional Neural Net (CNN) model. It will contain the following layers: 2 Convolutional, 1 Pooling and 1 Dense. The first thing is to use the 'keras_model_sequential' function which allows us to composing a model with diffrent kind of linear layers. In the last layer we use the 'layer_dropout', which is a technique that improves the model with over-fit on neural networks. By using it the classification error in the model will decrease.

```
model <- keras_model_sequential()


model %>%
  layer_conv_2d(filter = 32, kernel_size = c(3,3), padding = "same", input_shape = c(64,64, 3), activati
  layer_conv_2d(filter = 16, kernel_size = c(3,3), padding = "same") %>%
  layer_activation_leaky_relu(0.5) %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dense(100, activation = "relu") %>%
  layer_dropout(0.5) %>%
  layer_dense(length, activation = "softmax")
```

Now it's time to compile the model. Which make sure we can configurate the model for the training. We're choosing the metrics is set to 'accuracy' so we can measure the perfomance of the model.

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

Because we used image_data_generator() and flow_images_from_directory() earlier. We need to use the fit_generator(), when fitting the model on our training data. This function is similiar to the function 'fit()', which require epochs and batch_size. The number of epochs for training indicates, which amount of times the the model will expose itself to the whole training set.

```
history = model %>% fit_generator(train_image_array,
                                  steps_per_epoch = 100,
                                  epochs = 10,
                                  validation_data = test_image_array,
                                  validation_steps = 100
)
```

At last we can find the metrics and how it perform.

```
history$metrics
```

```
## $loss
##  [1] 1.8140220 0.8363954 0.5317935 0.3908751 0.3799773 0.3454767 0.2870984
##  [8] 0.2346096 0.2232554 0.1896768
##
```

```
## $accuracy
##  [1] 0.4722397 0.7053125 0.7953125 0.8531250 0.8615142 0.8725000 0.8940063
##  [8] 0.9088328 0.9103125 0.9290625
##
## $val_loss
##  [1] 2.0340598 1.3245483 0.6871209 0.4187667 0.3732850 0.8835770 0.1750653
##  [8] 0.3168227 0.2679953 0.2338593
##
## $val_accuracy
##  [1] 0.5979317 0.7916014 0.8517706 0.9238483 0.8812284 0.8022563 0.9376371
##  [8] 0.9319962 0.9357568 0.9517393
```

And then the mean values of the accuracy and valuation accuracy.

```
mean(history$metrics$accuracy)
```

```
## [1] 0.8202218
```

```
mean(history$metrics$val_accuracy)
```

```
## [1] 0.8605766
```

As shown above the model performed really well. Where the average accuracy is found to be 81 percent, which is quit high and adequate results. Some reasons to why the result is so high, could depend on pictures format, where the fruit is placed on a white background. So the model doesn't get any noise from the background of the picture. If we put this against the validation accuracy, where is slighty 1 percent higher in validation.

### Pretrained models

In this example, we would use a pretrained model to see how it compares to our own model.

### Setting up your own pretrained model from Keras

1. Choose pretrained model and load it using "application_"
2. Choose if you want to freeze any layers
3. Trainsform model into sequational or functional API models
4. Add layers if wanted
5. Choose optimizer, loss and metrics
6. Run model using "fit_generator"

### An example using VGG19

You have easily access to pretrained model, in R you simply have to write "application_" and then 10 pretrained models will show, amongst other models also the VGG16 model.

We're using a sequential model, which is a linear stack of layers. First we start by loading the pretrained model. Here there's a lot of options for the model. Here we can apply weights to the ImageNet images. Here we also want to remove the default classifier and attach our own classifier, therefore 'include_top' is set to FALSE.

```
vgg193 = keras::application_vgg19(include_top = FALSE,
                                  weights = 'imagenet',
                                  input_shape = c(64, 64 ,3))
```

Then you have the possibility to freeze layers. When a layer is frozen, its weights are frozen as well while training. If your current dataset is similar to the one it was freezed on, then it's good to freeze it, otherwise you can train the bottom layers. Here our model is trained on ImageNet, therefore we freeze it.

```
vgg193 %>% freeze_weights(from = 1, to = 20)
```

And then turn it into a sequential model.

```
model = keras_model_sequential(vgg193)
```

Adn then we add layers to out model. Here we add a flatten layer, which flattens a tensor into one dimension. Then we add a two dense layers.

```
model = model %>% layer_flatten() %>% layer_dense(units = 1024, activation = 'relu') %>% layer_dense(uni
```

Let's run a summary of the model.

```
summary(model)
```

```
## Model: "sequential_1"
## _________________________________________________________________________________
## Layer (type)                         Output Shape                      Param #
## =================================================================================
## vgg19 (Model)                        (None, 2, 2, 512)                 20024384
## _________________________________________________________________________________
## flatten_1 (Flatten)                  (None, 2048)                      0
## _________________________________________________________________________________
## dense_2 (Dense)                      (None, 1024)                      2098176
## _________________________________________________________________________________
## dense_3 (Dense)                      (None, 16)                        16400
## =================================================================================
## Total params: 22,138,960
## Trainable params: 4,474,384
## Non-trainable params: 17,664,576
## _________________________________________________________________________________
```

Here we can see which layers we applied to the model. Then we can run the 'compile' function for the optimizer, loss and metrics.

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

Now we run the model.

```
history = model %>% fit_generator(
  train_image_array,
  epochs = 5,
  steps_per_epoch = 25,
  validation_data = test_image_array,
  validation_steps = 100
)
```

And then the metrics.

```
history$metrics
```

```
## $loss
```

```
## [1] 1.16047271 0.13850557 0.10746868 0.08605508 0.01228811
##
## $accuracy
## [1] 0.6725000 0.9600000 0.9662338 0.9750000 0.9975000
##
## $val_loss
## [1] 0.5261897 0.2443847 0.3185520 0.2533683 0.1512502
##
## $val_accuracy
## [1] 0.8664995 0.9470385 0.9109997 0.9269822 0.9517393
```

And the mean value of the validations accuracy and accuracy.

```
mean(history$metrics$accuracy)
```

```
## [1] 0.9142468
```

```
mean(history$metrics$val_accuracy)
```

```
## [1] 0.9206518
```

Here the valuation accuracy is between 0.90 and 0.94, which is quiet high and higher than our own trained model.

## An example using ResNet50

Let's try the exact same thing on a ResNet50 model. We start the same way. Here we add a pooling, here Global Average Pooling.

```
resnet50 = keras::application_resnet50(include_top = FALSE,
                                       weights = 'imagenet',
                                        pooling = 'avg',
                                       input_shape = c(64, 64 ,3))
```

Here we're freezing a few more layers, as this model has a lot more layers than the last model, the VGG19 model.

```
resnet50 = resnet50 %>% freeze_weights(from = 1, to = 176)
```

Again, we run a sequential model.

```
model = keras_model_sequential(resnet50)
```

```
model = model %>% layer_flatten() %>% layer_dense(units = 1024, activation = 'relu') %>% layer_dropout(
```

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

Now we run the model.

```
history = model %>% fit_generator(
  train_image_array,
  epochs = 5,
  steps_per_epoch = 25,
  validation_data = test_image_array,
```

```
  validation_steps = 100
)
```

```
history$metrics
```

```
## $loss
## [1] 0.80956694 0.13641719 0.10648368 0.06374415 0.08589373
##
## $accuracy
## [1] 0.79375 0.95875 0.97250 0.97875 0.98750
##
## $val_loss
## [1] 3.169308 3.176729 3.392551 3.346953 3.313760
##
## $val_accuracy
## [1] 0.06424318 0.06549671 0.06549671 0.06455657 0.06298966
```

```
mean(history$metrics$accuracy)
```

```
## [1] 0.93825
```

```
mean(history$metrics$val_accuracy)
```

```
## [1] 0.06455657
```

Here the valuation accuracy is between 0.90 and 0.94, which is quiet high and higher than our own trained model.

## Why does the pretrained models perform better?

Pretrained models do not necessarily perform better than selftrained model, but they do have an advantage. Pretrained models are a bit like looking at two people, who both have never played football in their life, but one is an athlete and one isn't. As said, none of them know football beforehand, but the athlete does have an advantage and may be better, as he has strength and stamina and is in shape. Pretrained models are already trained

We also tested the ResNet50 model, but it has its problems.

## What have we learned?

We have learned that simple selftrained model like the CNN model we created in the start. Not could outperform the pretrained model. This could occur from the complexity of the model, which may detect more classifications error then the simple model. None the less we actually saw that the pretrained model gets an higher accuracy on around 10 percentage points, then the selftrained CNN model. This is an exiciting topic to dig further in. Due to the pretrained model seems to be more used as a standard procedure. And considering that where DL is being used in industries such as. health, finance and retail, etc. Where the impact of a little increase in the accuracy could make big difference.

## References

[1] Verma, Shiva: A Simple Guide to Using Keras Pretrained Models: https://towardsdatascience.com/step-by-step-guide-to-using-pretrained-models-in-keras-c9097b647b29

[2]