Lab 8

Hüseyin Anil Gündüz

2022-06-28

Exercise 1

NB: it is impractical to do this exercise on your laptop, even if you have a GPU. You are advised to work on Google Colab.

In this exercise we would like to build a classification system for 120 different breeds of dogs, based on commonplace pictures. The data is available at https://www.kaggle.com/c/dog-breed-identification/data (permanent, requires an account) or https://www.dropbox.com/s/l7b7l5fjwwj6ad2/dog-breed-identification.zip?dl=0 (temporary, no login needed). Download it and unzip it, then put the contents in a folder named “data” in the same directory of this R notebook.

```r
dst = "./dog-breed-identification.zip"

if(!file.exists(dst)) {
  # if you have problems downloading from R, try from a browser
  url = "https://www.dropbox.com/s/l7b7l5fjwwj6ad2/dog-breed-identification.zip?dl=1"
  options(timeout=360)
  download.file(url, dst)
}

if(!dir.exists("data")) {
  unzip(dst, exdir="data")
}
```

This dataset is composed of 10222 pictures in different resolutions and aspect rations. The smallest classes are Briad and Eskimo dog with only 66 images each, whereas the biggest class is the Scottish deerhound with 126 images.

Here are some sample images along with the relative label:

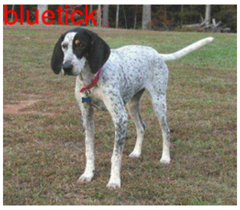
```r
library(magick)
```

```
## Linking to ImageMagick 6.9.10.23
## Enabled features: fontconfig, freetype, fftw, lcms, pango, webp, x11
## Disabled features: cairo, ghostscript, heic, raw, rsvg
```

```
## Using 8 threads
```

```r
data = read.csv("data/labels.csv", stringsAsFactors = FALSE)

par(mfrow = c(3,3), mar=c(1, 0, 0, 0))
for (i in 1:9) {
  img = image_resize(image_read(paste0("data/train/", data$id[i], ".jpg")),
                     geometry = "255x255")
  plot(image_annotate(img, data$breed[i], size = 30, color = "red"))
}
```



This is a challenging problem since there are many classes and only few instances per class. Moreover, the images contain a lot of details that do not help to identify the dogs, but allow a network to easily memorize the data. We will first try to naively approach the problem and directly train a CNN on this dataset. After convincing ourselves that this is not going to work, we will use a pre-trained VGG16 (i.e., trained successfully on some other dataset) and fine-tune it to our data.

But first, we will re-organize the images by reserving a subset of the training images for validation, and creating a directory for every breed of dog.

```r
library(data.table)

data = fread("data/labels.csv")

breeds = unique(data$breed)
if(!dir.exists("data/train_sorted")) {
  dir.create("data/train_sorted")
  dir.create("data/val_sorted")

  for(dog in breeds) {
    dir.create(paste0("data/train_sorted/", dog))
    dir.create(paste0("data/val_sorted/", dog))
    ids = data[breed == dog]$id
    sp = ceiling(length(ids) / 5)
    for (id in ids[1:sp]) {
      file.copy(paste0("data/train/", id, ".jpg"),
                paste0("data/val_sorted/", dog, "/"))
    }
    for (id in ids[(sp+1):length(ids)]) {
     file.copy(paste0("data/train/", id, ".jpg"),
                paste0("data/train_sorted/", dog, "/"))
    }
  }
}
```

Data preparation

As this dataset is fairly small, we can generate more synthetic images by applying random transformations to the images we have. This technique is called *data augmentation*, and it can greatly help in reducing overfitting on small datasets.

Keras provides a function called `image_data_generator` that does this for us; please read its documentation. Every time a new batch of data is requested, the augmentations are randomly applied on-the-fly; this saves a lot of memory, at the price of larger computational resources needed.

We now want to randomly perform the following transformations to each image:

- Re-scale the pixel values to be between 0 and 1 (they are now between 0 and 255)
- Flip horizontally
- Rotation of at most 30 degrees,
- Change brightness by at most 50%
- Stretch horizontally and vertically by at most 20%
- Zoom in/out by at most 20%
- Mirror the image to fill possible blanks created by the transformations (parameter: `fill_mode`)

```
library(keras)

train_data_generator = image_data_generator(
  rescale = 1/255,
  horizontal_flip = TRUE,
  rotation_range = 30,
  brightness_range = c(0.5, 1.5),
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  zoom_range = 0.2,
  fill_mode = "reflect"
)
```

```
## Loaded Tensorflow version 2.2.0
```

This generator can be coupled with another utility function called `flow_images_from_directory`. This function automatically loads images from the given directory, assuming that each of the diffent classes is in its own directory (hence the re-organization we previously did). It will also resize the images to a given size, create batches of suitable size, and so on. Again, read the documentation for its complete functionality.

We do not use random augmentation for the validation images except for centering and scaling. Why?

```
data_source_train = flow_images_from_directory(
  directory = "data/train_sorted",
  generator = train_data_generator,
  class_mode = "categorical",
  batch_size = 32,
  target_size = c(224, 224),
  shuffle = TRUE
)

data_source_validation = flow_images_from_directory(
  directory = "data/val_sorted",
  generator = image_data_generator(rescale = 1/255),
  target_size = c(224, 224),
  class_mode = "categorical",
  batch_size = 32,
  shuffle = TRUE
)
```

Here are some examples of how the augmented images look:

```
x = data_source_train[["next"]]()[[1]]
for(i in 1:4) {
  img = x[i,,,]
  r = img[,,1]
  g = img[,,2]
  b = img[,,3]
  col = rgb(r, g, b)
  dim(col) = dim(r)
  grid::grid.newpage()
  grid::grid.raster(col, interpolate=FALSE)
}
```

Define a Network

After preparing the data we define a network architecture. There are a lot of possible architectures; a good start might be a slightly smaller version of the famous VGG16 architecture. It consists of 4 blocks of 2 convolutional layers followed by one max pooling step, then two fully connected layers of size 512 are used, for a total of around 5 million weight parameters.

Global average pooling is used instead of flattening to reduce the number of parameters of the network. It takes the average of every input channel, so that a tensor of shape 14x14x512 results in a vector of 512 elements, each of which is the average of the corresponding 14x14 slice.

```
model = keras_model_sequential() %>%
  # Block 1
  layer_conv_2d(filters=64, kernel_size=c(3,3), padding="same", activation="relu",
                input_shape=c(224, 224, 3)) %>%
  layer_conv_2d(filters=64, kernel_size=c(3,3), padding="same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2), stride=c(2,2)) %>%

  # Block 2
  layer_conv_2d(filters=128, kernel_size=c(3,3), padding="same", activation="relu") %>%
  layer_conv_2d(filters=128, kernel_size=c(3,3), padding="same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2), stride=c(2,2)) %>%

  # Block 3
  layer_conv_2d(filters=256, kernel_size=c(3,3), padding="same", activation="relu") %>%
  layer_conv_2d(filters=256, kernel_size=c(3,3), padding="same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2), stride=c(2,2)) %>%

  # Block 4
  layer_conv_2d(filters=512, kernel_size=c(3,3), padding="same", activation="relu") %>%
  layer_conv_2d(filters=512, kernel_size=c(3,3), padding="same", activation="relu") %>%
  layer_max_pooling_2d(pool_size=c(2,2), stride=c(2,2)) %>%

  # Flatten by global averaging
  layer_global_average_pooling_2d() %>%
  layer_dense(512, activation="relu") %>%
  layer_dense(512, activation="relu") %>%

  # Output for 120 Classes
  layer_dense(120, activation="softmax")
```

```
summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________________
## Layer (type)                         Output Shape                    Param #
## =================================================================================
## conv2d (Conv2D)                      (None, 224, 224, 64)            1792
## _________________________________________________________________________________
## conv2d_1 (Conv2D)                    (None, 224, 224, 64)            36928
## _________________________________________________________________________________
## max_pooling2d (MaxPooling2D)         (None, 112, 112, 64)            0
## _________________________________________________________________________________
## conv2d_2 (Conv2D)                    (None, 112, 112, 128)           73856
## _________________________________________________________________________________
## conv2d_3 (Conv2D)                    (None, 112, 112, 128)           147584
## _________________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)       (None, 56, 56, 128)             0
## _________________________________________________________________________________
## conv2d_4 (Conv2D)                    (None, 56, 56, 256)             295168
## _________________________________________________________________________________
## conv2d_5 (Conv2D)                    (None, 56, 56, 256)             590080
## _________________________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)       (None, 28, 28, 256)             0
## _________________________________________________________________________________
## conv2d_6 (Conv2D)                    (None, 28, 28, 512)             1180160
## _________________________________________________________________________________
## conv2d_7 (Conv2D)                    (None, 28, 28, 512)             2359808
## _________________________________________________________________________________
## max_pooling2d_3 (MaxPooling2D)       (None, 14, 14, 512)             0
## _________________________________________________________________________________
## global_average_pooling2d (GlobalAve (None, 512)                     0
## ________________________________________________________________________________
## dense (Dense)                       (None, 512)                     262656
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 512)                     262656
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 120)                     61560
## ================================================================================
## Total params: 5,272,248
## Trainable params: 5,272,248
## Non-trainable params: 0
## ________________________________________________________________________________
```

Train the network

After the network is defined, we need to `compile` it to configure it for training, and fit it using the function `fit_generator` (documentation here). This function is used when the dataset is not available in memory (e.g., because it does not fit, or because we are augmenting images on the fly), but we can provide a function that generates the data batch-by-batch, like we are doin now.

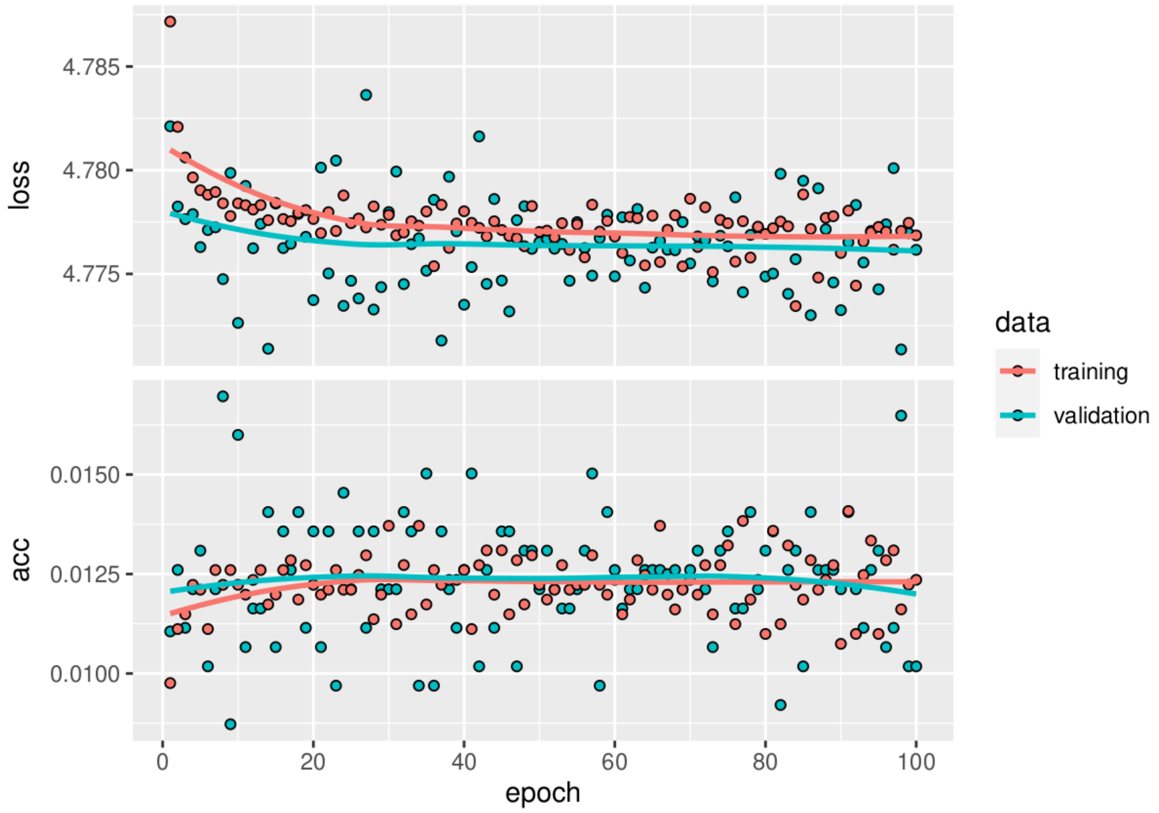
```
compile(
  model,
  optimizer = optimizer_adam(),
  loss = "categorical_crossentropy",
  metrics = "accuracy"
)

steps_per_epoch = floor(8127 / 32)
validation_steps = floor(2095 / 32)

history = fit_generator(
  model,
  generator = data_source_train,
  steps_per_epoch = steps_per_epoch,
  epochs = 100,
  validation_data = data_source_validation,
  validation_steps = validation_steps,
  verbose = 0,
)

plot(history)
```

```
library(png)
img <- readPNG('lab8_1.png')
grid::grid.raster(img)
```



# Using a pretrained network Even with the aid of data augmentation, the network overfits badly; this can be explained by the fact that the images are quite diverse in relation to the size of the training set. Data augmentation can only bring you so far, and even with the aid of regularization the task would be difficult.

One popular trick to overcome this difficulty, known as *pre-training*, is to use another CNN that has been trained on a different, larger dataset, for a related task. Most of the weights of this network are then frozen (i.e., will not be updated), and the last few layers (the "head") are replaced with new, freshly re-initialized ones and learned from scratch. After the network has converged, the weights are unfrozen, and fine-tuned again. What is the rationale behind freezing and unfreezing the weights?

In Keras, this is done with the following steps:

1. Download the network architecture without its head.
2. Add a custom head appropriate for the new task and define this as a new Keras model.
3. Freeze the weights of all layers except the new head.
4. Train the network
5. Unfreeze the weights
6. Train the network again.

Do not forget to read the documentation for the functions you do not yet know!

```
# Download Network
vgg_headless = application_vgg16(
  include_top = FALSE,
  weights = "imagenet",
  input_shape = c(224, 224, 3))

# Define the new head
vgg = vgg_headless$output %>%
  layer_global_average_pooling_2d() %>%
  layer_dense(512, activation = "relu") %>%
  layer_dense(512, activation = "relu") %>%
  layer_dense(120, activation = "softmax")
```

```r
# Create the model
vgg_pretrained = keras_model(inputs = vgg_headless$input, outputs = vgg)

# freeze the weights of the first 19 layers
freeze_weights(vgg_pretrained, from = 1, to = 19)

summary(vgg_pretrained)
```

```
## Model: "model"
## _____________________________________________________________________
## Layer (type)                   Output Shape               Param #
## =====================================================================
## input_1 (InputLayer)           [(None, 224, 224, 3)]      0
## _____________________________________________________________________
## block1_conv1 (Conv2D)          (None, 224, 224, 64)       1792
## _____________________________________________________________________
## block1_conv2 (Conv2D)          (None, 224, 224, 64)       36928
## _____________________________________________________________________
## block1_pool (MaxPooling2D)     (None, 112, 112, 64)       0
## _____________________________________________________________________
## block2_conv1 (Conv2D)          (None, 112, 112, 128)      73856
## _____________________________________________________________________
## block2_conv2 (Conv2D)          (None, 112, 112, 128)      147584
## _____________________________________________________________________
## block2_pool (MaxPooling2D)     (None, 56, 56, 128)        0
## _____________________________________________________________________
## block3_conv1 (Conv2D)          (None, 56, 56, 256)        295168
## _____________________________________________________________________
## block3_conv2 (Conv2D)          (None, 56, 56, 256)        590080
## _____________________________________________________________________
## block3_conv3 (Conv2D)          (None, 56, 56, 256)        590080
## _____________________________________________________________________
## block3_pool (MaxPooling2D)     (None, 28, 28, 256)        0
## _____________________________________________________________________
## block4_conv1 (Conv2D)          (None, 28, 28, 512)        1180160
## _____________________________________________________________________
## block4_conv2 (Conv2D)          (None, 28, 28, 512)        2359808
## _____________________________________________________________________
## block4_conv3 (Conv2D)          (None, 28, 28, 512)        2359808
## _____________________________________________________________________
## block4_pool (MaxPooling2D)     (None, 14, 14, 512)        0
## _____________________________________________________________________
## block5_conv1 (Conv2D)          (None, 14, 14, 512)        2359808
## _____________________________________________________________________
## block5_conv2 (Conv2D)          (None, 14, 14, 512)        2359808
## _____________________________________________________________________
## block5_conv3 (Conv2D)          (None, 14, 14, 512)        2359808
## _____________________________________________________________________
## block5_pool (MaxPooling2D)     (None, 7, 7, 512)          0
## _____________________________________________________________________
## global_average_pooling2d_1 (Gl (None, 512)                0
## _____________________________________________________________________
## dense_3 (Dense)                (None, 512)                262656
## _____________________________________________________________________
## dense_4 (Dense)                (None, 512)                262656
## _____________________________________________________________________
## dense_5 (Dense)                (None, 120)                61560
## ========================================================================
## Total params: 15,301,560
## Trainable params: 586,872
## Non-trainable params: 14,714,688
## ________________________________________________________________________
```

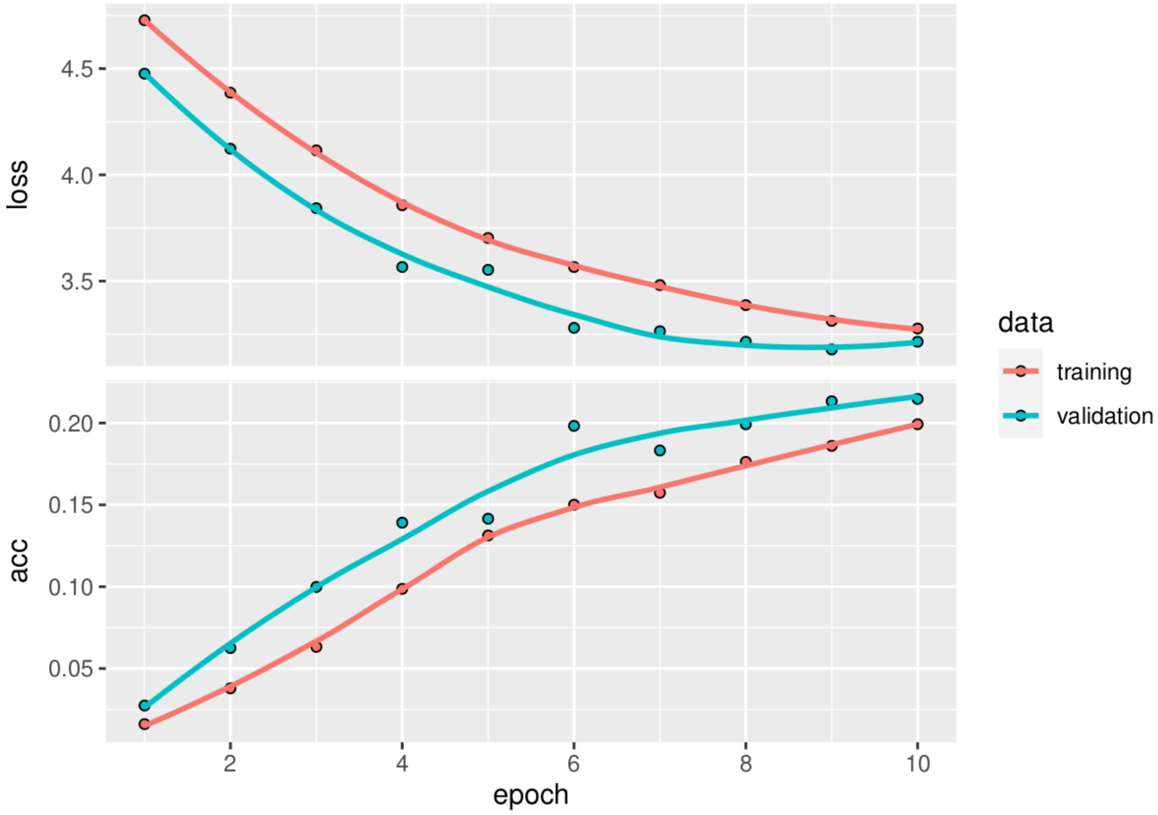
Now the pretrained model can be fine tuned to the new task

```
compile(
  vgg_pretrained,
  optimizer = optimizer_adam(),
  loss = "categorical_crossentropy",
  metrics = "accuracy"
)

history = fit_generator(
  vgg_pretrained,
  generator = data_source_train,
  steps_per_epoch = steps_per_epoch,
  epochs = 10,
  validation_data = data_source_validation,
  validation_steps = validation_steps
)

plot(history)
```

```
library(png)
img <- readPNG('lab8_2.png')
grid::grid.raster(img)
```



As you can see, the results are much better now, and would keep improving if we had trained for longer.

Exercise 2

This exercise is about the receptive field of convolutional neural networks. For our purposes, the receptive field of a neuron in layer L contains the features in a preceding layer ℓ that affect the output of said neuron, with $\ell = 0$ being the input to the network. In other words, changing any value in a neuron's receptive field will change the output of that neuron. By going backwards from layer L, convolutions and pooling operations enlarge the receptive field of neurons at layer L, so that the deeper the network, the larger the receptive field of neurons at the end ofthe network.

Let $\mathbf{z}_\ell \in \mathbb{R}^{n_\ell}$ be the output of layer ℓ (and $\mathbf{z}_0$ the input), that is obtained with a one-dimensional convolution or pooling operation from $\mathbf{z}_{\ell-1}$ with a kernel of size k_ℓ and stride s_ℓ. Define r_ℓ to be the size of the receptive field in the ℓ-th layer of a neuron in layer L, i.e. the minimum width of the largest region that contains the elements in $\mathbf{z}_\ell$ that affect a generic element in $\mathbf{z}_L$. Note that this region can contain gaps, i.e. neurons that do not affect the output of the neuron in layer L, if they are in between neurons that do affect it.

Show that $r_{\ell-1}$ can be computed from r_ℓ as follows:

$$r_{\ell-1} = s_\ell \cdot r_\ell + k_\ell - s_\ell$$

You can consider padding to be infinite, or, equivalently, focus on the neurons in the middle of the layer, without analyzing what happens near the borders. Hint: consider the case $k_\ell = 1$ first.

Then solve the recurrence to show that:

$$r_0 = \sum_{\ell=1}^{L} \left((k_\ell - 1) \prod_{i=1}^{\ell-1} s_i \right) + 1$$

with the base case being $r_L = 1$.

Compute the receptive field size of the pre-trained VGG16 architecture we used above, right before the global average pooling layer.

Now suppose to have a dilation of $d_\ell \geq 1$ at every layer. What is the new formula for r_0?

What is the most effective way to increase the size of the receptive field of a neural network?

Solution Start with $k_\ell = 1$. Every neuron in r_ℓ, has a receptive field of 1, and we need to add $s_\ell - 1$ for the gap left between adjacent neurons, resulting in $r_{\ell-1} = r_\ell + (r_\ell - 1)(s_\ell - 1) = r_\ell s_\ell - s_\ell + 1$. If $k_\ell > 1$ and odd, we need to add $(k_\ell - 1)/2$ at both sides of the receptive field, resulting in

$$r_{\ell-1} = r_\ell s_\ell - s_\ell + 1 + 2 \cdot \frac{k_\ell - 1}{2} = r_\ell s_\ell + k_\ell - s_\ell$$

When k_ℓ is even, we need to add $k_\ell/2$ on one side, and $k_\ell/2 - 1$ on the other side; as this adds up to $k_\ell - 1$ again, the result is the same.

To solve the recurrence, we can unroll a few steps and try to spot a pattern. For ease of notation, we use $r, r', r'', r''', \ldots$ instead of $r_\ell, r_{\ell-1}, r_{\ell-2}, r_{\ell-3}, \ldots$, similarly for k_ℓ and s_ℓ.

$$\begin{aligned}
r' &= sr + k - s \\
r'' &= s'r' + k' - s' \\
&= s'sr + s'k - s's + k' - s' \\
r''' &= s''r'' + k'' - s'' \\
&= s''s'sr + s''s'k - s''s's + s''k' - s''s' + k'' - s'' \\
&= (r-1)s''s's + (k-1)s''s' + (k'-1)s'' + k'' \\
r'''' &= s'''r''' + k''' - s''' \\
&= (r-1)s'''s''s's + (k-1)s'''s''s' + (k'-1)s'''s'' + (k''-1)s''' + k'''
\end{aligned}$$

The next element would be:

$$
\begin{aligned}
r_{\ell-5} = (r_\ell - 1)&s_{\ell-4}s_{\ell-3}s_{\ell-2}s_{\ell-1}s_\ell \\
&+ (k_\ell - 1)s_{\ell-4}s_{\ell-3}s_{\ell-2}s_{\ell-1} \\
&+ (k_{\ell-1} - 1)s_{\ell-4}s_{\ell-3}s_{\ell-2} \\
&+ (k_{\ell-2} - 1)s_{\ell-4}s_{\ell-3} \\
&+ (k_{\ell-3} - 1)s_{\ell-4} \\
&+ (k_{\ell-4} - 1) \\
&+ 1
\end{aligned}
$$

The pattern should be clear:

$$
r_{\ell-i} = (r_\ell - 1)\prod_{j=1}^{i} s_{\ell-i+j} + \sum_{j=1}^{i}\left((k_{\ell-i+j} - 1)\prod_{k=1}^{j-1} s_{\ell-i+k}\right) + 1
$$

With the convention that $\prod_{i=a}^{b} x_i = 1$ when $a > b$. Now using $i = \ell = L$, and remembering that $r_L = 1$, we find that:

$$
r_0 = \sum_{j=1}^{L}\left((k_j - 1)\prod_{k=1}^{j-1} s_k\right) + 1
$$

The pre-trained VGG16 has five blocks composed by two (first two blocks) or three (last three blocks) convolutions with kernel size 3 and stride 1 followed by a max pooling of kernel size 2 with stride 2. We can easily apply the recursive formula:

```r
strides = c(1,1,2, 1,1,2, 1,1,1,2, 1,1,1,2, 1,1,1,2)
kernels = c(3,3,2, 3,3,2, 3,3,3,2, 3,3,3,2, 3,3,3,2)

Reduce(function(r, l) {
  strides[l] * r + kernels[l] - strides[l]
}, length(strides):1, init = 1)
```

```
## [1] 212
```

Dilation can be seen as further gaps between the elements of every filter, so that a dilation of $d_j \geq 1$ leaves $d_j - 1$ gaps between elements of the filter. For example, a dilation of 2 means that a filter of size 3 will cover 5 elements in total, a dilation of 3 results in 7 elements, and so on. With this insight, we can replace k_j with $k_j d_j - d_j + 1$ in the formula above:

$$
r_0 = \sum_{j=1}^{L}\left((k_j d_j - d_j)\prod_{k=1}^{j} s_k\right) + 1
$$

According to the formula, the best way to increase the size of the receptive field of a network is to increase striding, because strides of successive layers get multiplied together. Note, however, that high strides means that, in practice, there will be large gaps in the receptive field, with many neurons not actually contributing to the neurons in the following layers. For this reason, neural networks since VGG16 have also become much deeper. Moreover, the receptive field is not the only parameter that affects performance, and having a large receptive field seems to be a necessary, but not sufficient condition for well-performing networks.