13 Convolutional neural networks

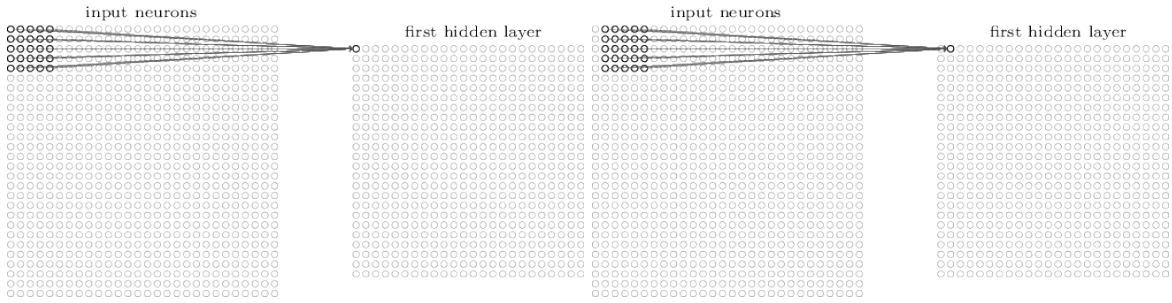
Convolutional neural networks (CNNs) are designed to capture patterns in data that have local (e.g., spatial or temporal) dependencies. They are often used for image processing (2D source) and video processing (3D source).

13.1 CNN basics

In CNNs, one key concept is the **feature map** (also called *kernel* or *filter*). In a 2D CNN, it is often a $k \times k \to 1$ mapping. For example, we can "shrink" a 28×28 image to a 24×24 image by applying a function

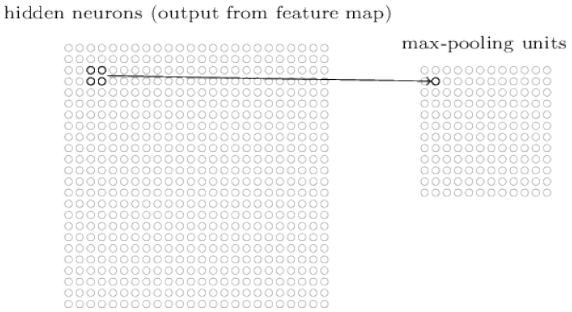
$$f^{(1)}(\{t_{i,j} : 1 \le i, j \le 5\}) = h^{(1)}\left(\sum_{i=1}^{5}\sum_{j=1}^{5} w_{i,j}^{(1)} t_{i,j} + b^{(1)}\right)$$

to every square of 5×5 pixels, as shown in the figures below. Here, $h^{(1)}$ is an activation function. This feature map has 26 parameters. The double summation inside the parentheses is a convolution operation in math; thus the name of the method. The w's are called *shared weights* and the b is called *shared bias*. For every neuron in the resulting 24×24 image, its input, a 5×5 region, is called the **local receptive field** for the hidden neuron. In this feature map, the **stride length** is 1 because we apply the function to a square and then to the next square by moving horizontally or vertically by 1 pixel. If we apply the function to a square and then to the next square by moving 2 pixels, the stride will be 2. For example, we may use a $4 \times 4 \to 1$ feature map with stride 2 to map a 28×28 image to a 13×13 image.



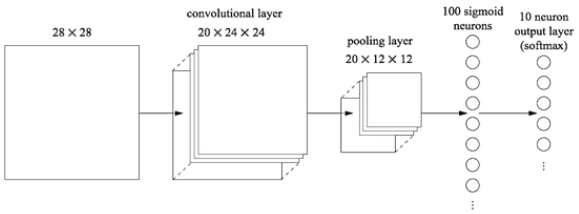
Sometimes the source images are **padded** with zeros to make the result of a feature map having the same resolution as its source image. For example, if we first pad a 28×28 source image with 2 pixels of zeros at all sides to make it a 32×32 image, and then apply a $5 \times 5 \to 1$ feature map with stride 1, we will have a 28×28 result image.

A feature map is sometimes followed by a **pooling** operation. For example, the figure below shows a $2 \times 2 \to 1$ "max-pooling" operation, which takes every 2×2 square and extract the maximum of the 4 pixel values. The squares are non-overlapping. Thus a $2 \times 2 \to 1$ "max-pooling" on a 24×24 image results in a 12×12 image. Other pooling operations are "average-pooling" (i.e., the average of the input values) and "l_2-pooling" (i.e., the RMS, the square root of the sum of squares over the input values). A pooling layer does not have any parameters.



In a feature map, the same function is used for all squares. Thus, a feature map captures a certain pattern locally. We often use many feature maps in a layer so that we can capture many different local patterns. For example, in the figure below, the first convolutional layer contains 20 feature maps of $5 \times 5 \to 1$ with stride 1. Thus there are 20 functions, $f^{(1)}, \ldots, f^{(20)}$, for this layer. It is followed by a "max-pooling" layer of $2 \times 2 \to 1$. After that the

20 12×12 images are "flattened" into a vector of 2880 values and fed to a full-connected layer with 100 sigmoid neurons, followed by a full-connected softmax output layer with 10 neurons.



The layers and nodes of the whole model may be represented as

$$28 \times 28 \times 1 \xrightarrow[5\times5\to1]{\text{filter}} 24 \times 24 \times 20 \xrightarrow[2\times2\to1]{\text{max-pool}} 12 \times 12 \times 20 \xrightarrow{\text{flatten}} 2880 \xrightarrow{\text{FC}} 100 \xrightarrow{\text{softmax}} 10$$

Another way to represent this model is "conv5-20, maxpool2, FC-100, softmax-10". This model has 289,630 parameters!

Notes:

- We can visualize a feature map by drawing the shared weights as an image to show what pattern is captured by the feature map. We will show an example below.
- Both feature map and pooling can be done over rectangular regions with different width and height.
- In Keras, the default stride length is 1 both horizontally and vertically. To specify stride lengths, use the argument `strides=` in `layer_conv_2d()`. For example, `strides=2` means stride length 2 both horizontally and vertically; `strides=c(1,2)` means stride length 1 horizontally and 2 vertically.
- In Keras, one can define a customized pooling layer (called a "Lambda layer") with `layer_lambda()`.

13.2 The MNIST example with CNN

MNIST data preparation. Note that the images are reshaped to be a 4D array, as required by the CNN model.

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

dim(x_train) = c(nrow(x_train), 28, 28, 1)
dim(x_test) = c(nrow(x_test), 28, 28, 1)
x_train = x_train / 255  ## raw data has range 0-255
x_test = x_test / 255
y_train10 = to_categorical(y_train, 10)  ## make dummy (one-hot) variables for the outcome
y_test10 = to_categorical(y_test, 10)
```

We now define the CNN model in the section above, using the ReLU activation function.

```
use_session_with_seed(2018)
CNNmodel1 = keras_model_sequential() %>%
  layer_conv_2d(filters = 20, kernel_size = c(5,5), activation = 'relu',
                input_shape = c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 100, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')

summary(CNNmodel1)
```

Number of parameters: (1) between the input and the first convolutional layer, $26 \times 20 = 520$; (2) between the max-pooling layer and the layer with 100 neurons, $2881 \times 100 = 288100$; (3) between the last two layers, $101 \times 10 = 1010$. The total is 289,630. The max-pooling layer and the "flatten" layer do not have any parameters.

In NNDL Chapter 6, the author used this network structure with sigmoid activation, standard SGD with learning rate 0.1, 60 epochs and batch size 10. That model achieves 98.78% test set accuracy. Here we use ReLU activation, Adadelta as the optimizer, 12 epochs and batch size 100. It achieves 98.93% test set accuracy.

```
CNNmodel1 %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  #optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

CNNhistory1 = CNNmodel1 %>% fit(
  x_train, y_train10,
  epochs = 12, batch_size = 100, verbose = 1,
  validation_data = list(x_test, y_test10)
)
```

```
y_pred = CNNmodel1 %>% predict_classes(x_test)
table(y_test, y_pred)
```

The validation set accuracy rates for the 12 epochs are:

```
CNNhistory1$metrics$val_acc
```

```
 [1] 0.9784 0.9794 0.9854 0.9853 0.9871 0.9886 0.9901 0.9889 0.9902 0.9896 0.9912 0.9893
```

ReLU vs. sigmoid: When the two `relu` activations are replaced by `sigmoid`, while everything else is kept the same, the model is slower to run (28s vs. 17s per epoch on my office computer), and the model probably requires more epochs to converge. The validation set accuracy rates for the first 12 epochs are:

```
 [1] 0.9034 0.9324 0.9530 0.9591 0.9687 0.9749 0.9764 0.9783 0.9783 0.9812 0.9808 0.9831
```

To save the fitted model and the "history" information:

```
save_model_hdf5(CNNmodel1, "MNIST-CNNmodel1.hdf5")
save(CNNhistory1, file="MNIST-CNNmodel1-history.RData")
```

To load the information:

```
CNNmodel1 = load_model_hdf5("MNIST-CNNmodel1.hdf5", compile=F)
load("MNIST-CNNmodel1-history.RData")
```

We can get the weights and biases for the whole model.

```
wts = CNNmodel1 %>% get_weights
str(wts)
```

```
List of 6
 $ : num [1:5, 1:5, 1, 1:20] -0.037 -0.3167 0.0554 0.4254 0.3592 ...
 $ : num [1:20(1d)] 0.000104 -0.115432 -0.013928 -0.002437 -0.000795 ...
 $ : num [1:2880, 1:100] -0.00703 0.02599 -0.01004 0.03746 -0.01579 ...
 $ : num [1:100(1d)] -4.45e-02 9.46e-05 -1.16e-02 -8.91e-03 -2.87e-02 ...
 $ : num [1:100, 1:10] -0.298 -0.3979 0.0733 -0.3382 0.2603 ...
 $ : num [1:10(1d)] -0.00894 0.04108 -0.04704 -0.03245 -0.04505 ...
```

These are parameters for the layers 2, 5, 6 in this Keras model, as shown from `summary(CNNmodel1)`.

```
wts1 = CNNmodel1 %>% get_layer(index=2) %>% get_weights
str(wts1)
wts2 = CNNmodel1 %>% get_layer(index=5) %>% get_weights
str(wts2)
wts3 = CNNmodel1 %>% get_layer(index=6) %>% get_weights
str(wts3)
```

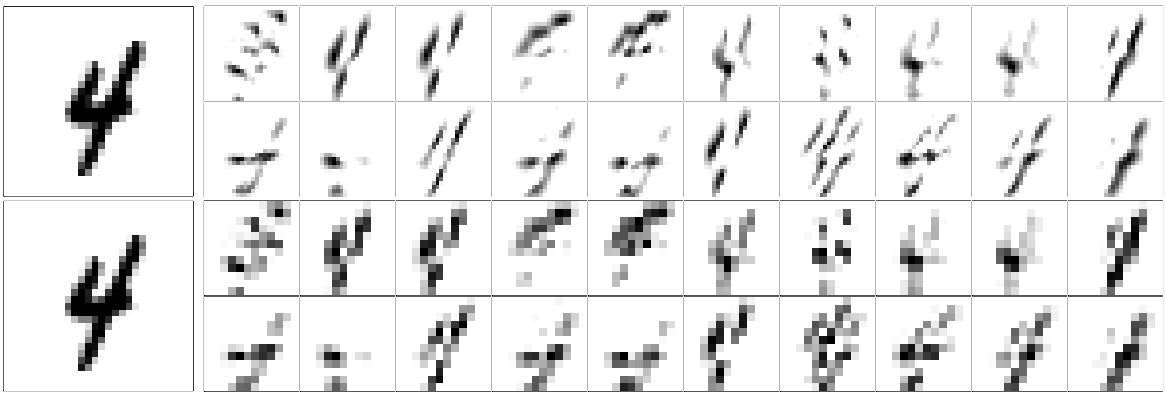
Sanity check: Apply the weights to an observation to see if the results are as expected.

```
testsample = x_test[7000,,,, drop=F]  ## take an observation
dim(testsample)  ## (1, 28, 28, 1)
CNNmodel1 %>% predict_classes(testsample)
CNNmodel1 %>% predict_proba(testsample)
```

We can see what the intermediate images look like after the convolutional layer and after the max-pooling layer.

```
testsample = x_test[7000,,,, drop=F]  ## take an observation

layer2_CNNmodel1 = keras_model(inputs = CNNmodel1$input,
                               outputs = get_layer(CNNmodel1, index=2)$output)
testsample_layer2 = predict(layer2_CNNmodel1, testsample)
dim(testsample_layer2)  ## (1, 24, 24, 20)

layer3_CNNmodel1 = keras_model(inputs = CNNmodel1$input,
                               outputs = get_layer(CNNmodel1, index=3)$output)
testsample_layer3 = predict(layer3_CNNmodel1, testsample)
dim(testsample_layer3)  ## (1, 12, 12, 20)

im1 = t(drop(testsample))[,28:1]    ## draw the original 28 x 28 image
image(1:28, 1:28, im1, col=gray(1-(0:255)/255), xaxt='n', yaxt='n')

for(i in 1:20) {   ## draw the 24 x 24 images
  aa = testsample_layer2[1,,,i]
  tmpim1 = t(aa)[,24:1]
  image(1:24, 1:24, tmpim1, col=gray(1-(0:255)/255), xaxt='n', yaxt='n')
}

for(i in 1:20) {   ## draw the 12 x 12 images
  aa = testsample_layer3[1,,,i]
  tmpim1 = t(aa)[,12:1]
  image(1:12, 1:12, tmpim1, col=gray(1-(0:255)/255), xaxt='n', yaxt='n')
}
```

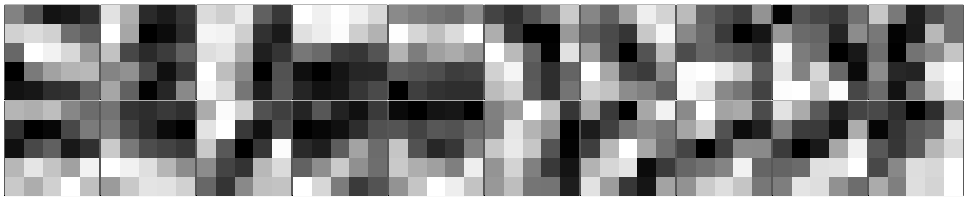
Below the original input image is on the left, the 20 images (24×24) after the first convolutional layer are on the top right, and the 20 images (12×12) after the max-pooling layer are on the bottom right.



We can visualize a feature map by drawing its shared weights as an image to see what pattern is captured by the feature map. The weights need to be transformed to $[0, 1]$ before drawing. We use the sigmoid function for this purpose, which transforms negative weights to $(0, 0.5)$ and positive weights to $(0.5, 1)$. As a result, negative weights are shown in light gray and positive weights are in dark. The higher magnitude a weight is, the lighter or darker.

```
wts1 = CNNmodel1 %>% get_layer(index=2) %>% get_weights
fmaps = wts1[[1]]
```

```
for(i in 1:20) {
  aa = 1/(1+exp(-fmaps[,,1,i]))  ## sigmoid transformation
  im1 = t(aa)[,5:1]
  image(1:5, 1:5, im1, col=gray(1-(0:255)/255), xaxt='n', yaxt='n')
}
```



For example, the weights for the 4th feature map are:

```
fmaps[,,1,4]
## 1/(1+exp(-fmaps[,,1,4]))
```

```
            [,1]         [,2]       [,3]       [,4]        [,5]
[1,] -0.49833700 -0.444226176 -0.4940618 -0.4414652 -0.36522257
[2,] -0.38966134 -0.435453683 -0.4851711 -0.3311261 -0.23816237
[3,] -0.04484979 -0.003762289  0.1463872  0.1657518  0.06882975
[4,]  0.29662243  0.346830606  0.2765740  0.2178668  0.21510899
[5,]  0.22397776  0.284384161  0.2652722  0.1666256  0.09983233
```

13.3 Multiple input images

Note that in `layer_conv_2d()`, the argument `input_shape=` requires 3 numbers: the first two specify the resolution of the input image(s) and the third is the number of images. For example, a color image contains 3 images (e.g., for the red, green, and blue components).

Another example is the CNN model:

$$28\times28\times1 \xrightarrow[5\times5\to1]{\text{filter}} 24\times24\times20 \xrightarrow[2\times2\to1]{\text{max-pool}} 12\times12\times20 \xrightarrow[5\times5\to1]{\text{filter}} 8\times8\times40 \xrightarrow[2\times2\to1]{\text{max-pool}} 4\times4\times40 \xrightarrow{\text{flatten}} 640 \xrightarrow{\text{FC}} 100 \xrightarrow{\text{softmax}} 10$$

Another way to represent this model is "conv5-20, maxpool2, conv5-40, maxpool2, FC-100, softmax-10". This model has a second convolutional layer with 40 feature maps of $5 \times 5 \to 1$. The input for that layer are 20 12×12 images after the first max-pooling layer. A feature map in the second convolutional layer will be a function like this:

$$f^{(2)}(\{t_{i,j;k} : 1 \le i,j \le 5, 1 \le k \le 20\}) = h^{(2)}\left(\sum_{k=1}^{20}\sum_{i=1}^{5}\sum_{j=1}^{5} w^{(1)}_{i,j;k} t_{i,j;k} + b^{(1)}\right),$$

where $h^{(2)}$ is an activation function. There are $5 \times 5 \times 20 + 1 = 501$ parameters in this feature map. With 40 feature maps, the total number of parameters between the first max-pooling layer and the second convolutional layer is $501 \times 40 = 20040$. This model has 85,670 parameters.

Note that adding the second convolutional layer greatly reduces (not increases) the total number of parameters from 289,630 to 85,670! Despite this, the model with two convolutional layers is more complex and takes longer to run (24s per epoch on my office computer vs. 17s for `CNNmodel1` using 'relu').

13.4 The MNIST example with CNN (cont'd)

The above model can be specified below.

```
use_session_with_seed(2018)
CNNmodel2 = keras_model_sequential() %>%
  layer_conv_2d(filters = 20, kernel_size = c(5,5), activation = 'relu',
                input_shape = c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 40, kernel_size = c(5,5), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 100, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')

summary(CNNmodel2)

CNNmodel2 %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)

CNNhistory2 = CNNmodel2 %>% fit(
  x_train, y_train10,
  epochs = 40, batch_size = 100, verbose = 1,
  validation_data = list(x_test, y_test10)
)
```

```
CNNhistory2$metrics$val_acc
```

```
 [1] 0.9821 0.9895 0.9897 0.9904 0.9904 0.9899 0.9912 0.9902 0.9922 0.9927 0.9921
[12] 0.9925 0.9921 0.9927 0.9927 0.9928 0.9920 0.9925 0.9928 0.9924 0.9922 0.9927
[23] 0.9928 0.9921 0.9926 0.9925 0.9927 0.9929 0.9927 0.9927 0.9926 0.9927 0.9926
[34] 0.9925 0.9925 0.9925 0.9925 0.9925 0.9925 0.9925
```

Again, when the three `relu` activations are replaced by `sigmoid`, the model takes longer to run (41s vs. 24s per epoch) and the validation set accuracy rates are below:

```
 [1] 0.9242 0.9629 0.9735 0.9786 0.9823 0.9839 0.9862 0.9856 0.9866 0.9887 0.9882
[12] 0.9879 0.9889 0.9898 0.9896 0.9898 0.9887 0.9901 0.9896 0.9887 0.9888 0.9899
[23] 0.9900 0.9901 0.9897 0.9897 0.9900 0.9905 0.9904 0.9902 0.9911 0.9904 0.9904
[34] 0.9906 0.9901 0.9899 0.9903 0.9896 0.9904 0.9886
```

To save/load the Keras model.

```
#save_model_hdf5(CNNmodel2, "MNIST-CNNmodel2.hdf5")
#save(CNNhistory2, file="MNIST-CNNmodel2-history.RData")
CNNmodel2 = load_model_hdf5("MNIST-CNNmodel2.hdf5", compile=F)
load("MNIST-CNNmodel2-history.RData")
```

In NNDL Chapter 6, the author used the above network structure with standard SGD with learning rate 0.1, 60 epochs and batch size 10. That model achieved 99.06% test set accuracy. Changing all the activation functions to ReLU improved the performance to 99.23%. (We achieved that above.)

Additional tweaks in NNDL: Expanding the training data improved the performance to 99.37%. Adding another fully connected ReLU layer with 100 neurons before the output layer improved the performance to 99.43%. He finally tried the following model with 1000 neurons in each of the last two hidden layers and a total of 1,672,570 parameters, which achieved 99.60% test set accuracy (with dropout and 40 epochs). He also applied dropout on the softmax layer, which I do not know how to do in Keras.

```
use_session_with_seed(2018)
CNNmodel3 = keras_model_sequential() %>%
  layer_conv_2d(filters = 20, kernel_size = c(5,5), activation = 'relu',
```

```
                input_shape = c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  #layer_dropout(rate = 0.5) %>%
  layer_conv_2d(filters = 40, kernel_size = c(5,5), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  #layer_dropout(rate = 0.5) %>%
  layer_flatten() %>%
  layer_dense(units = 1000, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1000, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(CNNmodel3)
```

```
CNNmodel3 %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)

CNNhistory3 = CNNmodel3 %>% fit(
  x_train, y_train10,
  epochs = 40, batch_size = 100, verbose = 1,
  validation_data = list(x_test, y_test10)
)
```

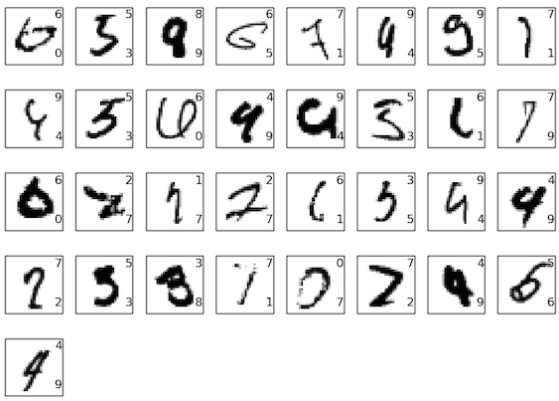
```
CNNhistory3$metrics$val_acc
```

```
 [1] 0.9855 0.9872 0.9900 0.9875 0.9935 0.9908 0.9934 0.9929 0.9929 0.9925 0.9937
[12] 0.9937 0.9941 0.9939 0.9936 0.9941 0.9937 0.9939 0.9927 0.9940 0.9937 0.9923
[23] 0.9938 0.9943 0.9941 0.9936 0.9939 0.9938 0.9938 0.9934 0.9943 0.9941 0.9940
[34] 0.9940 0.9937 0.9944 0.9941 0.9938 0.9940 0.9939
```

To save/load the Keras model.

```
#save_model_hdf5(CNNmodel3, "MNIST-CNNmodel3.hdf5")
#save(CNNhistory3, file="MNIST-CNNmodel3-history.RData")
CNNmodel3 = load_model_hdf5("MNIST-CNNmodel3.hdf5", compile=F)
load("MNIST-CNNmodel3-history.RData")
```

He then re-trained this model 5 times and applied ensemble on the 5 models (i.e., majority vote, as in random forests), to achieve 99.67%. The 33 misclassified images are below, in which the correct value is in the top right and the predicted value is in the bottom right.



13.5 Example: The lime package (using VGG16)

https://tensorflow.rstudio.com/blog/lime-v0.4-the-kitten-picture-edition.html

The R `lime` package implements the method by Ribeiro, Singh, Guestrin, 2016 to explain the predictions of classifiers.

The demonstration uses the VGG16 model, a winner of the 2014 ILSVRC, which has 16 hidden layers (excluding the pooling and flatten layers). The layers and nodes of the original published model can be represented as

$$
\begin{aligned}
224 \times 224 \times 3 &\xrightarrow[3\times3\to1]{\text{filter}} 224 \times 224 \times 64 \xrightarrow[3\times3\to1]{\text{filter}} 224 \times 224 \times 64 \xrightarrow[2\times2\to1]{\text{max-pool}} 112 \times 112 \times 64 \\
&\xrightarrow[3\times3\to1]{\text{filter}} 112 \times 112 \times 128 \xrightarrow[3\times3\to1]{\text{filter}} 112 \times 112 \times 128 \xrightarrow[2\times2\to1]{\text{max-pool}} 56 \times 56 \times 128 \\
&\xrightarrow[3\times3\to1]{\text{filter}} 56 \times 56 \times 256 \xrightarrow[3\times3\to1]{\text{filter}} 56 \times 56 \times 256 \xrightarrow[3\times3\to1]{\text{filter}} 56 \times 56 \times 256 \xrightarrow[2\times2\to1]{\text{max-pool}} 28 \times 28 \times 256 \\
&\xrightarrow[3\times3\to1]{\text{filter}} 28 \times 28 \times 512 \xrightarrow[3\times3\to1]{\text{filter}} 28 \times 28 \times 512 \xrightarrow[3\times3\to1]{\text{filter}} 28 \times 28 \times 512 \xrightarrow[2\times2\to1]{\text{max-pool}} 14 \times 14 \times 512 \\
&\xrightarrow[3\times3\to1]{\text{filter}} 14 \times 14 \times 512 \xrightarrow[3\times3\to1]{\text{filter}} 14 \times 14 \times 512 \xrightarrow[3\times3\to1]{\text{filter}} 14 \times 14 \times 512 \xrightarrow[2\times2\to1]{\text{max-pool}} 7 \times 7 \times 512 \\
&\xrightarrow{\text{flatten}} 25088 \xrightarrow{\text{FC}} 4096 \xrightarrow{\text{FC}} 4096 \xrightarrow{\text{FC}} 1000 \xrightarrow{\text{softmax}} 1000
\end{aligned}
$$

or as "conv3-64, conv3-64, maxpool2, conv3-128, conv3-128, maxpool2, conv3-256, conv3-256, conv3-256, maxpool2, conv3-512, conv3-512, conv3-512, maxpool2, conv3-512, conv3-512, conv3-512, maxpool2, FC-4096, FC-4096, FC-1000, softmax-1000".

In Keras, the first time you run `application_vgg16()`, it will download the VGG16 model and put it in `$HOME/.keras/models/`. The model file itself has 550 MB! The downloaded model does not have the last hidden layer (FC-1000). The first convolutional layer has $(3 \times 3 \times 3 + 1) \times 64 = 1792$ parameters. This whole model has 138,357,544 parameters!

```
application_vgg16()
```

13.6 Example: prediction of 9-mer peptide binding in cancer immunotherapy

https://tensorflow.rstudio.com/blog/dl-for-cancer-immunotherapy.html

SB: strong binder; WB: weak binder; NB: non-binder