Convolutional Neural Networks

Statistical Learning with Deep Artificial Neural Networks

2022

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. You can download the original dataset from https://www.cs.toronto.edu/~kriz/cifar.html

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

We choose two classes of images from CIFAR10 to create a convolutional neural network to classify these images.

The Keras package includes several datasets, one of them is cifar 10 so we will use `dataset_cifar10()` function to get it.

Load data and show some images

```
# Load Libraries
library(keras)
```

```
# Data Preparation -----------------------------------------------------

# See ?dataset_cifar10 for more info
cifar10 <- dataset_cifar10()
```

```
## Loaded Tensorflow version 2.4.1
```

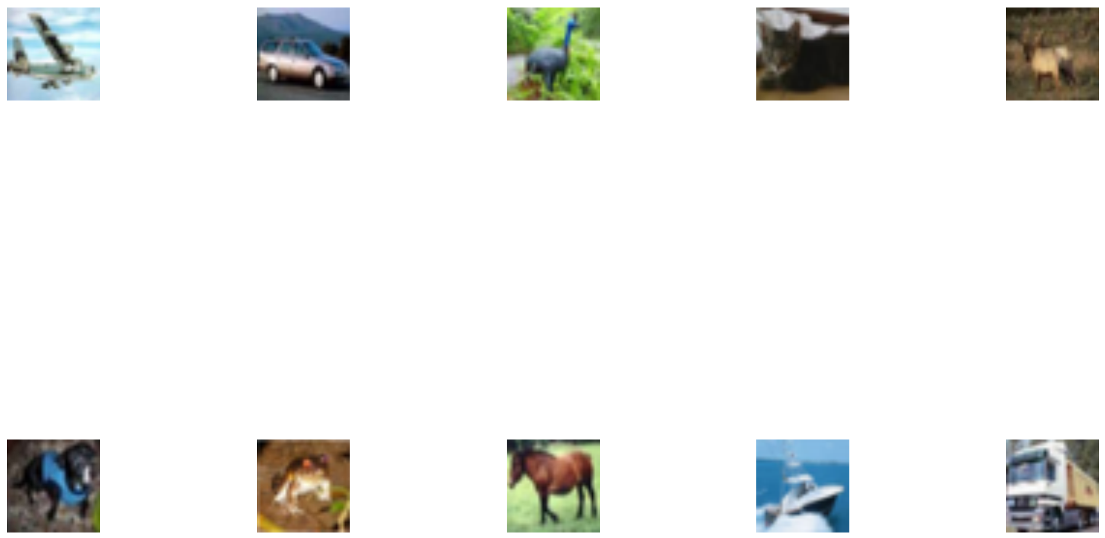
```
# images number by class in train data
table(cifar10$train$y)
```

```
##
##    0    1    2    3    4    5    6    7    8    9
## 5000 5000 5000 5000 5000 5000 5000 5000 5000 5000
```

```
# Show several images
#if (!require("BiocManager", quietly = TRUE))
#   install.packages("BiocManager")

#BiocManager::install("EBImage")
#library(EBImage)
#pictures = c(9802, 5, 7, 10, 4, 28,1, 8, 9, 2)
```

```r
# fig_img  = list()
# for (i in 1:10 ) {
#   fig_mat  = cifar10$train$x[pictures[i], , , ]
#   fig_img[[i]]  = normalize(Image(transpose(fig_mat), dim=c(32,32,3), colormode='Color'))
# }
# fig_img_comb = combine(fig_img[1:10])
# fig_img_obj = tile(fig_img_comb,5)
# plot(fig_img_obj, all=T)
```

```r
pictures = c(9802, 5, 7, 10, 4, 28,1, 8, 9, 2)
op <- par(mfrow = c(2, 5))
for (i in 1:10) {

  plot(as.raster(cifar10$train$x[pictures[i],,,],max = 255L))

}
```



```r
par(op)
```

Selection two classes

```r
# Parameters --------------------------------------------------------------
batch_size <- 32
epochs <- 30
data_augmentation <- TRUE
image.size <- c(32,32,3)


# Select two classes

i<- 4
j <- 7

row.select.train <- cifar10$train$y%in%c(i,j)
# checking
sum(row.select.train)
```

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

```r
y_train <- cifar10$train$y[row.select.train,]
x_train <- cifar10$train$x[row.select.train,,,]

# changing i value by 0 and j value by 1
y_train[y_train==i] <- 0
y_train[y_train==j] <- 1

#checking
dim (x_train)
```

```
## [1] 10000    32    32     3
```

```r
length(y_train)
```

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

```r
table(y_train)
```

```
## y_train
##    0    1
## 5000 5000
```

```r
row.select.test <- cifar10$test$y%in%c(i,j)
# checking
sum(row.select.test)
```

```
## [1] 2000
```

```r
y_test <- cifar10$test$y[row.select.test,]
x_test <- cifar10$test$x[row.select.test,,,]
```

```
# changing i value by 0 and j value by 1
y_test[y_test==i] <- 0
y_test[y_test==j] <- 1

#checking
dim (x_test)
```

```
## [1] 2000   32   32    3
```

```
length(y_test)
```

```
## [1] 2000
```

```
table(y_test)
```

```
## y_test
##    0    1
## 1000 1000
```

```
# Feature scale RGB values in test and train inputs
x_train <- x_train/255
x_test <- x_test/255



# Show some images
p <- 5

y_test[p]
```

```
## [1] 0
```

```
plot(as.raster(x_test[p,,,]))
```

CNN model

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                padding="same",
                input_shape = image.size) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu",
                padding="same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

```
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                      Output Shape                    Param #
## ================================================================================
## conv2d_2 (Conv2D)                   (None, 32, 32, 32)              896
## ________________________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)      (None, 16, 16, 32)              0
## ________________________________________________________________________________
## conv2d_1 (Conv2D)                   (None, 16, 16, 64)              18496
## ________________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)      (None, 8, 8, 64)                0
## ________________________________________________________________________________
## conv2d (Conv2D)                     (None, 6, 6, 128)               73856
## ________________________________________________________________________________
## max_pooling2d (MaxPooling2D)        (None, 3, 3, 128)               0
## ________________________________________________________________________________
## flatten (Flatten)                   (None, 1152)                    0
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 512)                     590336
## ________________________________________________________________________________
## dense (Dense)                       (None, 1)                       513
## ================================================================================
## Total params: 684,097
## Trainable params: 684,097
## Non-trainable params: 0
## ________________________________________________________________________________
```

Keras includes a number of image processing helper tools. In particular, it includes the *image_data_generator()* function, which can automatically turn image files on disk into batches of pre-processed tensors.

```r
# image_data_generator Generate batches of image data with real-time data augmentation. The data will b
datagen <- image_data_generator()

#Fit image data generator internal statistics to some sample data.
datagen %>% fit_image_data_generator(x_train)

#flow_images_from_data Generates batches of of augmented/normalized data and labels.
train_generator <- flow_images_from_data(x_train, y_train,
                                          datagen,
                                          batch_size = batch_size)
# Element generate
batch <- generator_next(train_generator)

str(batch)
```

```
## List of 2
##  $ : num [1:32, 1:32, 1:32, 1:3] 0.522 0.302 0.655 0.357 0.753 ...
##  $ : num [1:32(1d)] 1 1 0 0 1 1 1 1 0 0 ...
```

Let's look at the output of one of these generators: it yields batches of 32 × 32 RGB images (shape (32, 32, 32, 3)) and binary labels (shape (32)). There are 32 samples in each batch (the batch size). Note that the generator yields these batches indefinitely: it loops endlessly over the images in the data.

```r
model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("accuracy")
)


history <-model %>% fit(
    train_generator,
    steps_per_epoch = as.integer(10000/batch_size),
    epochs = epochs,
    validation_data = list(x_test, y_test)
  )
```

Let's fit the model to the data using the generator. You have to use the `fit` function (it version before was `fit_generator`. It expects as its first argument a generator that will yield batches of inputs and targets indefinitely, like this one does. Because the data is being generated endlessly, the generator needs to know how many samples to draw from the generator before declaring an epoch over.

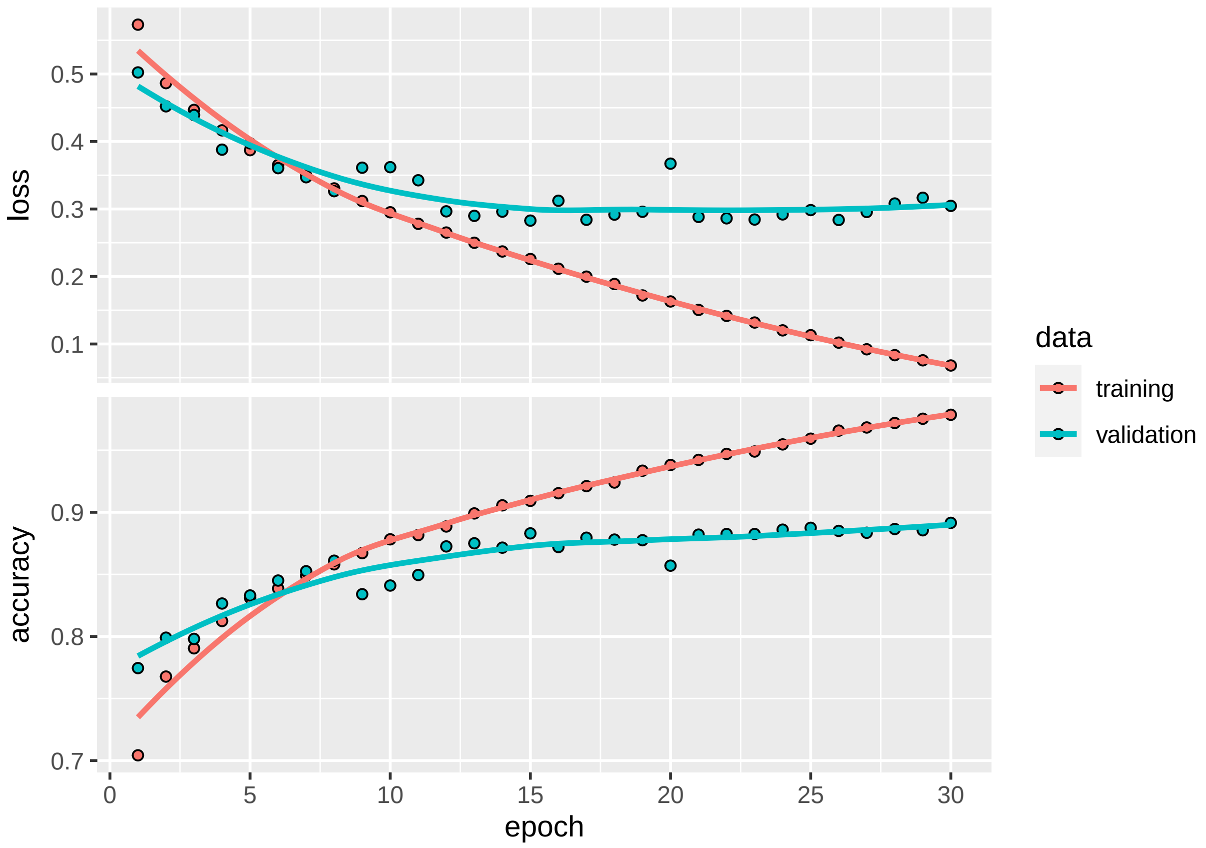
This is the role of the `steps_per_epoch` argument: after having drawn steps_per_epoch batches from the generator—that is, after having run for steps_per_epoch gradient descent steps—the fitting process will go to the next epoch. In this case, batches are 32-samples large, so it will take 312 batches until you see your target of 10,000 samples. When using `fit` function, you can pass a validation_data argument, much as with the fit function. It's important to note that this argument is allowed to be a data generator, but it could also be a list of arrays. If you pass a generator as validation_data, then this generator is expected to yield batches of validation data endlessly; thus you should also specify the validation_steps argument, which tells the process how many batches to draw from the validation generator for evaluation. In this example, a generator has not been used in the validation data.

Save the model

```
model %>% save_model_hdf5(paste0("cifar10_",i,"_",j,".h5"))
```

```
plot(history)
```

```
## ‘geom_smooth()‘ using formula ’y ~ x’
```



These plots are characteristic of overfitting. The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy stalls at 71–75%. The validation loss reaches its minimum after only five epochs and then stalls, whereas the training loss keeps decreasing linearly until it reaches nearly 0. Because you have relatively few training samples (2,000), overfitting will be your number-one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We're now going to introduce a new one, specific to computer vision and used almost universally when processing images with deep-learning models: data augmentation.

Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering you unable to train a model that can generalize to new data. Given infinite data, your model would be exposed to every possible aspect of the data distribution at hand: you would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by augmenting the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data and generalize better. In Keras, this can be done by configuring a number of random transformations to be performed on the images read by an image_data_generator. Let's get started with an example.

These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over this code:

- rotation_range is a value in degrees (0–180), a range within which to randomly rotate pictures.
- width_shift and height_shift are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.

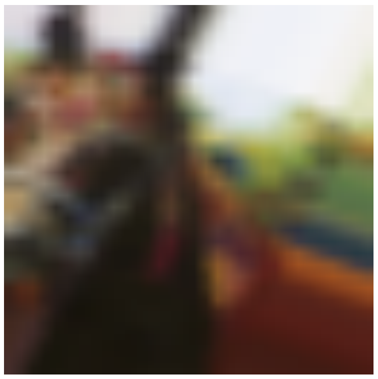
- shear_range is for randomly applying shearing transformations.
- zoom_range is for randomly zooming inside pictures.
- horizontal_flip is for randomly flipping half the images horizontally—relevant when there are no assumptions of horizontal asymmetry (for example, real-world pictures).
- fill_mode is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

```
# image_data_generator Generate batches of image data with real-time data augmentation. The data will b
datagen <- image_data_generator(
#  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest",
)
```

```
#Fit image data generator internal statistics to some sample data.
datagen %>% fit_image_data_generator(x_train)

#flow_images_from_data Generates batches of of augmented/normalized data and labels.
augmentation_generator <- flow_images_from_data(x_train, y_train,
                                          datagen,
                                          batch_size = batch_size)




# Element generate
batch <- generator_next(augmentation_generator)

str(batch)
```

```
## List of 2
##  $ : num [1:32, 1:32, 1:32, 1:3] 0.539 0.452 0.882 0 0.823 ...
##  $ : num [1:32(1d)] 1 0 1 0 1 1 1 1 0 0 ...
```

```
op <- par(mfrow = c(2, 2), pty = "s", mar = c(1, 0, 1, 0))
for (i in 1:4) {
 plot(as.raster(batch[[1]][i,,,]))
}
```



```
par(op)
```

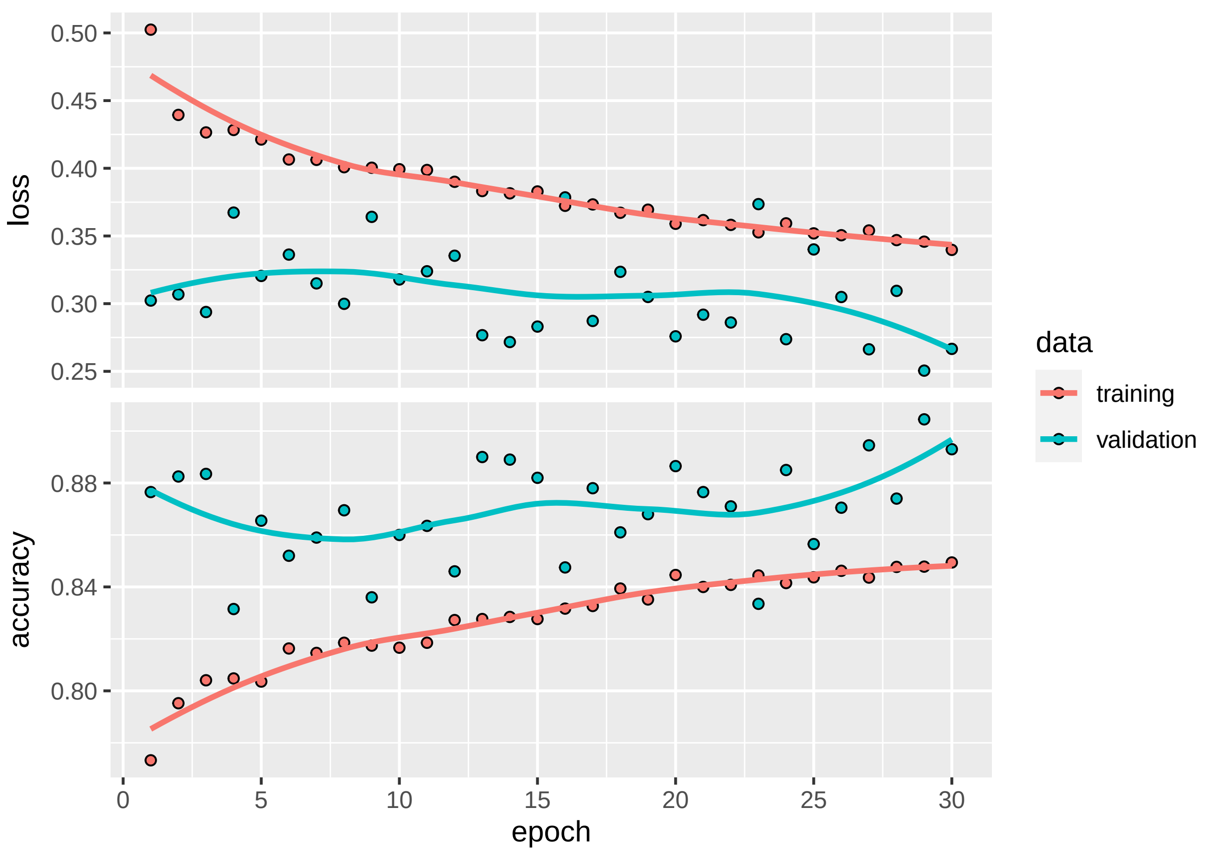
```
model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("accuracy")
)


history <-model %>% fit(
    augmentation_generator,
    steps_per_epoch = as.integer(10000/batch_size),
    epochs = epochs,
    validation_data = list(x_test, y_test)
  )
```

Save the model with data augmentation

```
model %>% save_model_hdf5(paste0("cifar10_",i,"_",j,"_augm.h5"))
```

```
plot(history)
```

```
## ‘geom_smooth()‘ using formula ’y ~ x’
```



Using a pretrained convnet

A common and highly effective approach to deep learning on small image datasets is to use a pretrained network. A pretrained network is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. If this original dataset is large enough and general enough, then the spatial-feature hierarchy learned by the pretrained network can effectively act as a generic model of the visual world, and hence its features can prove useful for many different computer-vision problems, even though these new problems may involve completely different classes than those of the original task. For instance, you might train a network on ImageNet (where classes are mostly animals and everyday objects) and then repurpose this trained network for something as remote as identifying furniture items in images. Such portability of learned features across different problems is a key advantage of deep learning compared to many older, shallow-learning approaches, and it makes deep learning very effective for small-data problems.

In this case, let's consider a large convnet trained on the ImageNet dataset (1.4 million labeled images and 1,000 different classes). ImageNet contains many animal classes, including different species of cats and dogs, and you can thus expect to perform well on the cats-versus-dogs classification problem. You'll use the VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014; it's a simple and widely used convnet architecture for ImageNet.8 Although it's an older model, far from the current state of the art and somewhat heavier than many other recent models, we chose it because its architecture is similar to what you're already familiar with and is easy to understand without introducing any new concepts.

```
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = image.size
)

conv_base
```

```
## Model: "vgg16"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                     Param #
## ================================================================================
## input_1 (InputLayer)                [(None, 32, 32, 3)]              0
## ________________________________________________________________________________
## block1_conv1 (Conv2D)               (None, 32, 32, 64)               1792
## ________________________________________________________________________________
## block1_conv2 (Conv2D)               (None, 32, 32, 64)               36928
## ________________________________________________________________________________
## block1_pool (MaxPooling2D)          (None, 16, 16, 64)               0
## ________________________________________________________________________________
## block2_conv1 (Conv2D)               (None, 16, 16, 128)              73856
## ________________________________________________________________________________
## block2_conv2 (Conv2D)               (None, 16, 16, 128)              147584
## ________________________________________________________________________________
## block2_pool (MaxPooling2D)          (None, 8, 8, 128)                0
## ________________________________________________________________________________
## block3_conv1 (Conv2D)               (None, 8, 8, 256)                295168
## ________________________________________________________________________________
## block3_conv2 (Conv2D)               (None, 8, 8, 256)                590080
## ________________________________________________________________________________
## block3_conv3 (Conv2D)               (None, 8, 8, 256)                590080
## ________________________________________________________________________________
## block3_pool (MaxPooling2D)          (None, 4, 4, 256)                0
## ________________________________________________________________________________
## block4_conv1 (Conv2D)               (None, 4, 4, 512)                1180160
## ________________________________________________________________________________
## block4_conv2 (Conv2D)               (None, 4, 4, 512)                2359808
## ________________________________________________________________________________
## block4_conv3 (Conv2D)               (None, 4, 4, 512)                2359808
## ________________________________________________________________________________
## block4_pool (MaxPooling2D)          (None, 2, 2, 512)                0
## ________________________________________________________________________________
## block5_conv1 (Conv2D)               (None, 2, 2, 512)                2359808
## ________________________________________________________________________________
## block5_conv2 (Conv2D)               (None, 2, 2, 512)                2359808
## ________________________________________________________________________________
## block5_conv3 (Conv2D)               (None, 2, 2, 512)                2359808
## ________________________________________________________________________________
## block5_pool (MaxPooling2D)          (None, 1, 1, 512)                0
## ================================================================================
## Total params: 14,714,688
## Trainable params: 14,714,688
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
model <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## vgg16 (Functional)                  (None, 1, 1, 512)               14714688
## ________________________________________________________________________________
## flatten_1 (Flatten)                 (None, 512)                     0
## ________________________________________________________________________________
## dense_3 (Dense)                     (None, 256)                     131328
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 1)                       257
## ================================================================================
## Total params: 14,846,273
## Trainable params: 14,846,273
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
freeze_weights(conv_base)   # Note that in order for these changes to take effect, you must recompile t
```

```
summary(model)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## vgg16 (Functional)                  (None, 1, 1, 512)               14714688
## ________________________________________________________________________________
## flatten_1 (Flatten)                 (None, 512)                     0
## ________________________________________________________________________________
## dense_3 (Dense)                     (None, 256)                     131328
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 1)                       257
## ================================================================================
## Total params: 14,846,273
## Trainable params: 131,585
## Non-trainable params: 14,714,688
## ________________________________________________________________________________
```

```
datagen = image_data_generator(
  # rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)
```

```
#Fit image data generator internal statistics to some sample data.
datagen %>% fit_image_data_generator(x_train)
#flow_images_from_data Generates batches of of augmented/normalized data and labels.
augmentation_generator <- flow_images_from_data(x_train, y_train,
                                                datagen, batch_size = batch_size)
```

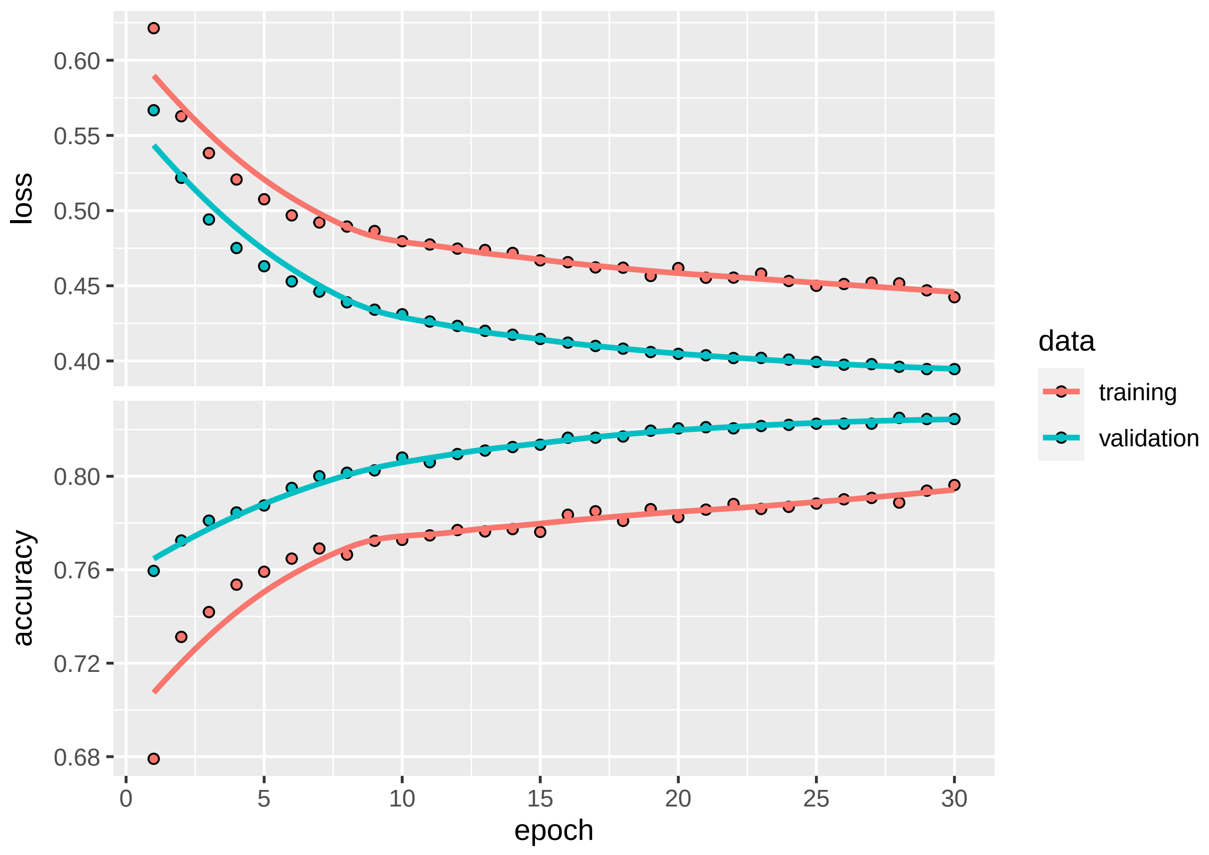
```
model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  metrics = c("accuracy")
)
```

Save the model with transfer learning

```
model %>% save_model_hdf5(paste0("cifar10_",i,"_",j,"_transferlearning.h5"))
```

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Fine-tuning

Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers. This is called fine-tuning because it slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.

```
unfreeze_weights(conv_base,from ="block3_conv1")
summary(model)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                       Output Shape                     Param #
## ================================================================================
## vgg16 (Functional)                 (None, 1, 1, 512)                14714688
## ________________________________________________________________________________
## flatten_1 (Flatten)                (None, 512)                      0
## ________________________________________________________________________________
## dense_3 (Dense)                    (None, 256)                      131328
## ________________________________________________________________________________
## dense_2 (Dense)                    (None, 1)                        257
## ================================================================================
## Total params: 14,846,273
## Trainable params: 14,586,113
## Non-trainable params: 260,160
## ________________________________________________________________________________
```

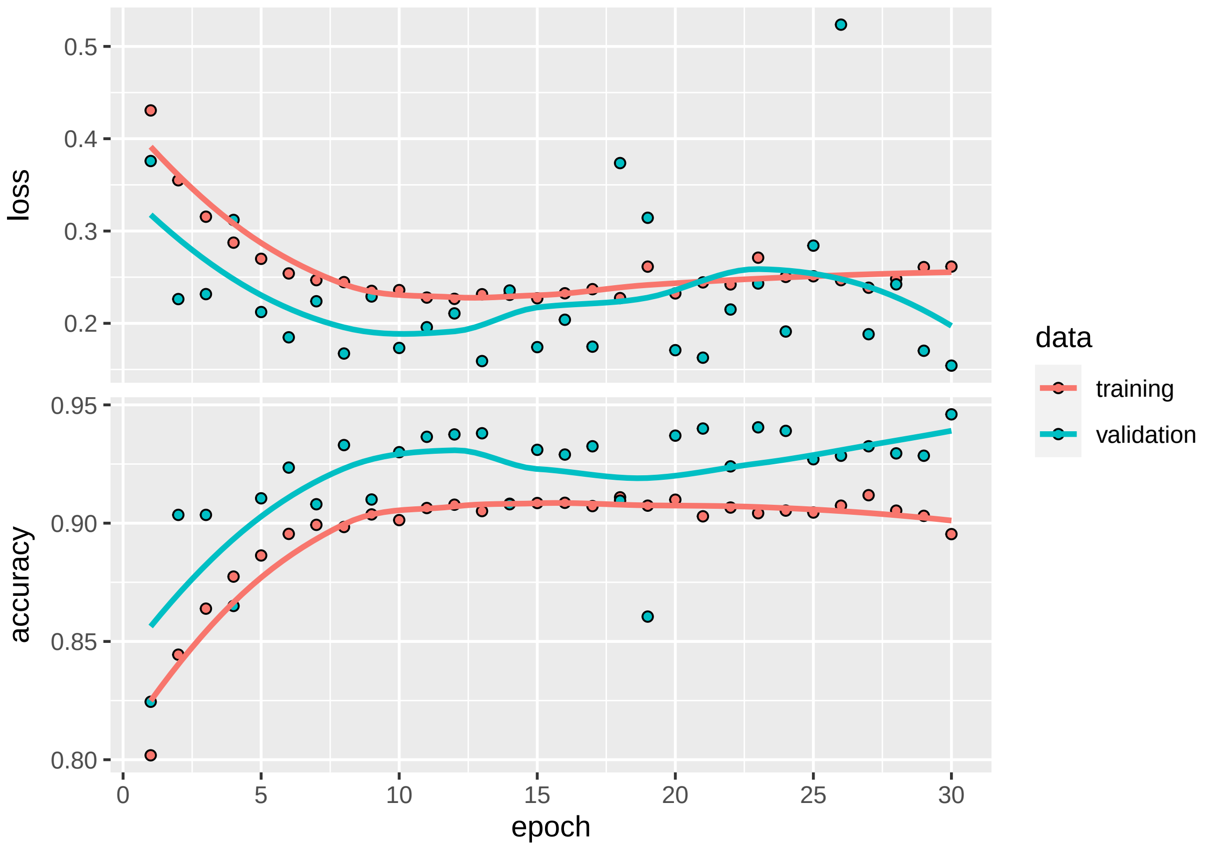
```
model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  metrics = c("accuracy")
)
```

Save the model with transfer learning

```
model %>% save_model_hdf5(paste0("cifar10_",i,"_",j,"_finetuningtransferlearning.h5"))
```

```
plot(history)
```

```
## ‘geom_smooth()‘ using formula ’y ~ x’
```



Visualizing what convnets learn

Visualizing intermediate activations

```r
# Load DL model
model <- load_model_hdf5("cifar10_4_7.h5")
model
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d_2 (Conv2D)                   (None, 32, 32, 32)              896
## ________________________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)      (None, 16, 16, 32)              0
## ________________________________________________________________________________
## conv2d_1 (Conv2D)                   (None, 16, 16, 64)              18496
## ________________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)      (None, 8, 8, 64)                0
## ________________________________________________________________________________
## conv2d (Conv2D)                     (None, 6, 6, 128)               73856
## ________________________________________________________________________________
## max_pooling2d (MaxPooling2D)        (None, 3, 3, 128)               0
## ________________________________________________________________________________
## flatten (Flatten)                    (None, 1152)                    0
## ________________________________________________________________________________
## dense_1 (Dense)                      (None, 512)                     590336
## ________________________________________________________________________________
## dense (Dense)                        (None, 1)                       513
## ================================================================================
## Total params: 684,097
## Trainable params: 684,097
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
# Show some images
p <- 5

y_test[p]
```

```
## [1] 0
```

```
plot(as.raster(x_test[p,,,]))
```

```
img_tensor <- x_test[p,,,]/255
dim(img_tensor)
```

```
## [1] 32 32  3
```

```
img_tensor <- array_reshape(img_tensor,c(1,image.size))
dim(img_tensor)
```

```
## [1]  1 32 32  3
```

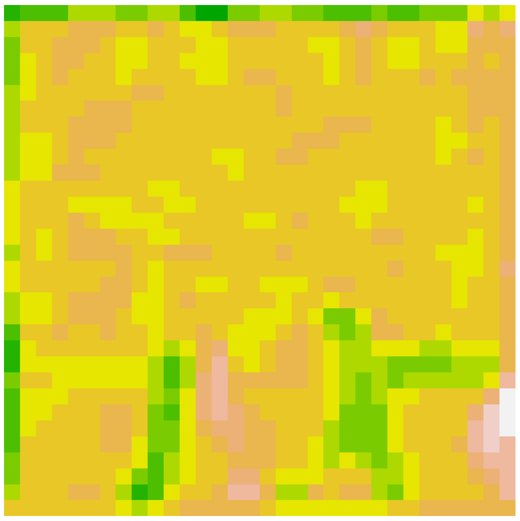
Extracts the outputs of the top eight layers and creates a model that will return these outputs, given the model input.

```
layer_outputs <- lapply(model$layers[1:8], function(layer) layer$output)
activation_model <- keras_model(inputs = model$input, outputs = layer_outputs)
activations <- activation_model %>% predict(img_tensor)

first_layer_activation <- activations[[1]]
dim(first_layer_activation)
```

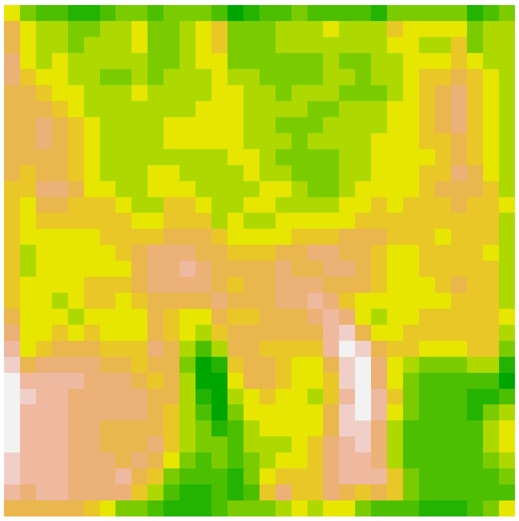
```
## [1]  1 32 32 32
```

Compare with summary of the model

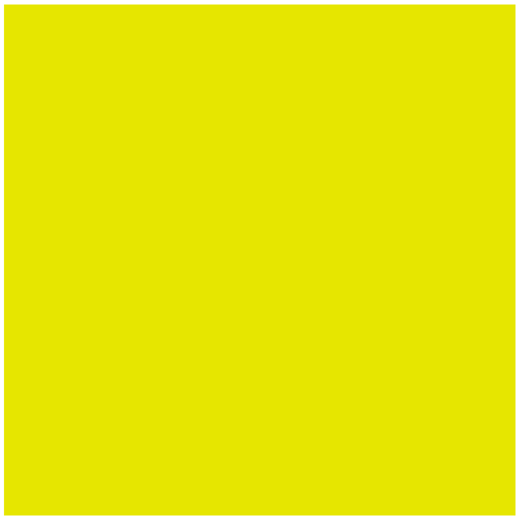
```
plot_channel <- function(channel) {
rotate <- function(x) t(apply(x, 2, rev))
image(rotate(channel), axes = FALSE, asp = 1,
col = terrain.colors(12))
}

plot_channel(first_layer_activation[1,,,1])
```



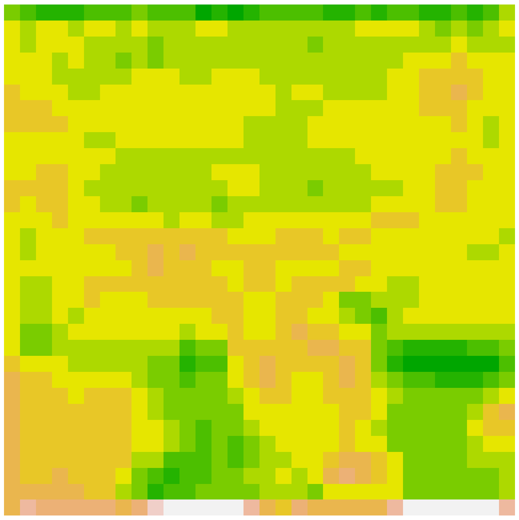
```
plot_channel(first_layer_activation[1,,,5])
```



```
plot_channel(first_layer_activation[1,,,15])
```



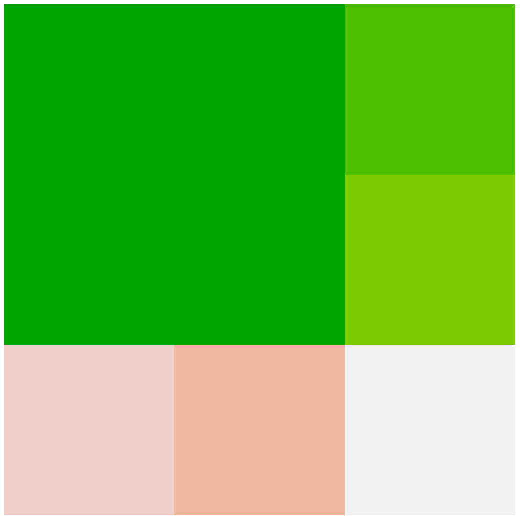
```r
plot_channel(first_layer_activation[1,,,20])
```



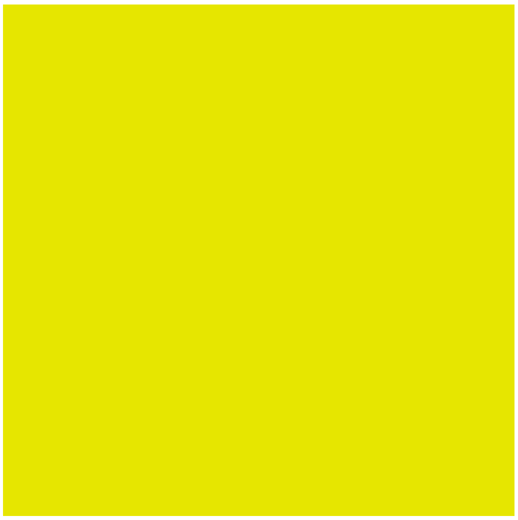
```r
# access other layers
layer_activation <- activations[[6]]
dim(layer_activation)
```

```
## [1]   1   3   3 128
```

```r
plot_channel(layer_activation[1,,,1])
```



```
plot_channel(layer_activation[1,,,50])
```



```r
plot_channel(layer_activation[1,,,110])
```