CIS 8392
Topics in Big Data Analytics

#Convolutional Neural Networks

Yu-Kai Lin

Agenda

- Introduction to convnets
- Training a convnet from scratch on a small dataset
- Using a pretrained convnet

[**Acknowledgements**] The materials in the following slides are based on the source(s) below:

- Deep Learning with R by Francois Chollet and J.J. Allaire
- R interface to Keras

Prerequisites

```
#install.packages("tidyverse")
#install.packages("keras")

library(tidyverse)
library(keras)
#install_keras() # will take a while; you should have done this already
```

Some of the deep neural networks are too large to train on a normal laptop, you may use the ones that I have already trained:

```
model <- load_model_hdf5("some_model_filename.h5") # change the filename here
history <- read_rds("some_history_filename.rds") # change the filename here
```

Download pretrained models (and their histories) here:
https://www.dropbox.com/s/hpdhvzrx9p9etnl/deeplearning-pretrained.zip?dl=0

If you are using the VM, the pretrained model and history files are stored in C:/CIS8392/data/deeplearning-pretrained/

Convolutional neural networks

Also known as CNNs or **convnets**

Convnets are a type of deep neural networks that have been extensively used in computer vision applications.

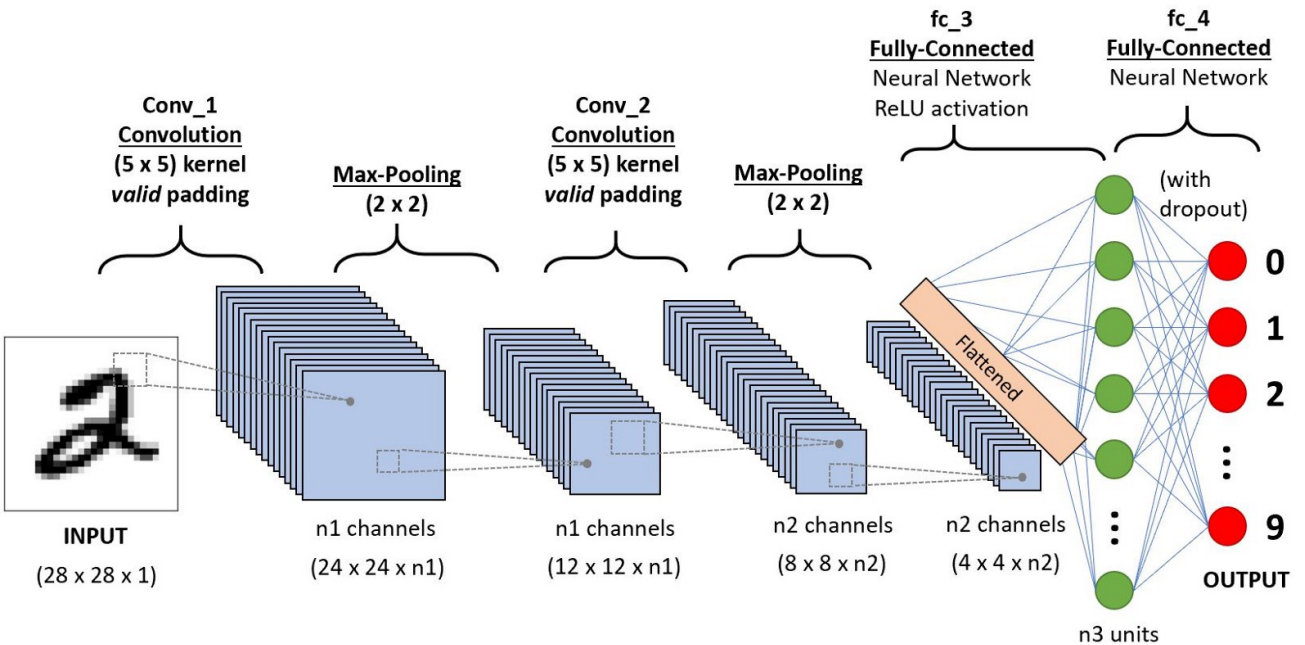
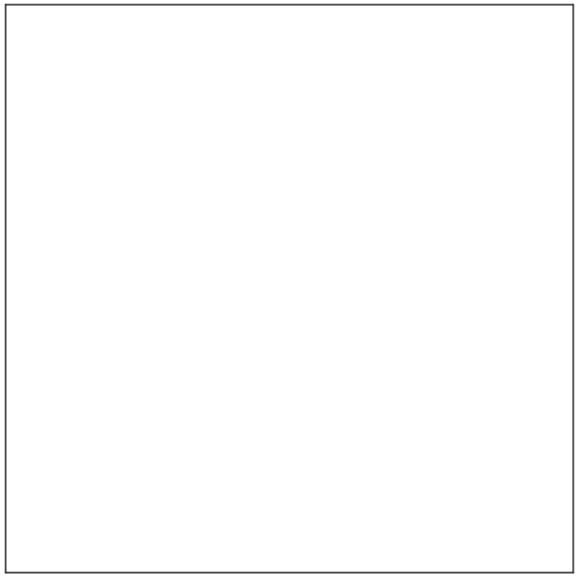
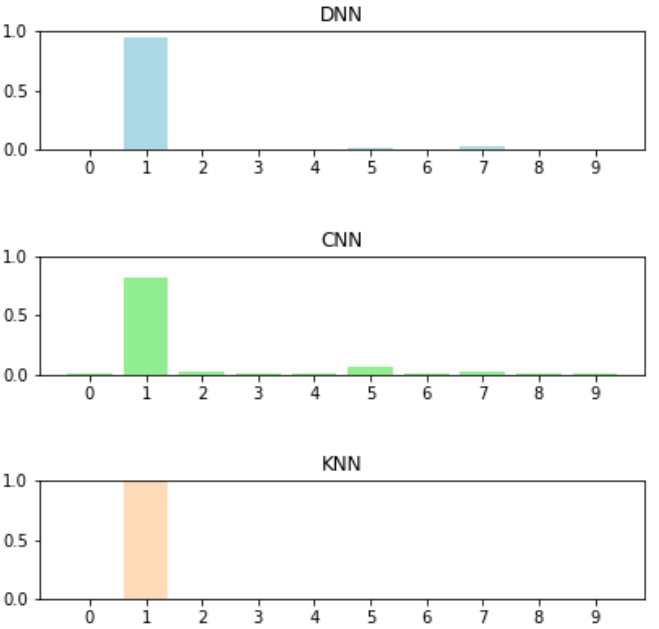


Image Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Performance of CNN

Notice how fast CNN reaches the answers compared with the other methods





Source: Reddit article - Recognizing Handwritten Digits As They Are Being Drawn

Instantiating a small convnet

A basic convnet looks like a stack of `layer_conv_2d` and `layer_max_pooling_2d` layers.

```
model <- keras_model_sequential() %>%
  layer_conv_2d(input_shape = c(28, 28, 1),  # input image shape
                filters = 32,  # output channel size
                kernel_size = c(3, 3),
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu")
```

Importantly, a convnet takes as input tensors of shape `(image_height, image_width, image_channels)` (not including the batch dimension). In this case, we'll configure the convnet to process inputs of size `(28, 28, 1)`, which is the format of MNIST images. We'll do this by passing the argument `input_shape = c(28, 28, 1)` to the first layer.

Let's display the architecture of the convnet:

```
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d_2 (Conv2D)                   (None, 26, 26, 32)              320
## ________________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)      (None, 13, 13, 32)              0
## ________________________________________________________________________________
## conv2d_1 (Conv2D)                   (None, 11, 11, 64)              18496
## ________________________________________________________________________________
## max_pooling2d (MaxPooling2D)        (None, 5, 5, 64)                0
## ________________________________________________________________________________
## conv2d (Conv2D)                     (None, 3, 3, 64)                36928
## ================================================================================
## Total params: 55,744
## Trainable params: 55,744
## Non-trainable params: 0
## ________________________________________________________________________________
```

- The output of every `layer_conv_2d` and `layer_max_pooling_2d` is a 3D tensor of shape `(height, width, channels)`.
- The width and height dimensions tend to shrink as we go deeper in the network.
- The number of channels is controlled by `filters` (32 or 64), the first argument passed to `layer_conv_2d`.

Adding a classifier on top of the convnet

```
model <- model %>%      # the output of the last conv2d is 3D (3, 3, 64)
  layer_flatten() %>%   # flatten it into 1D (576)
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d_2 (Conv2D)                   (None, 26, 26, 32)              320
## ________________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)      (None, 13, 13, 32)              0
## ________________________________________________________________________________
## conv2d_1 (Conv2D)                   (None, 11, 11, 64)              18496
## ________________________________________________________________________________
## max_pooling2d (MaxPooling2D)        (None, 5, 5, 64)                0
## ________________________________________________________________________________
## conv2d (Conv2D)                     (None, 3, 3, 64)                36928
## ________________________________________________________________________________
## flatten (Flatten)                   (None, 576)                     0
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 64)                      36928
## ________________________________________________________________________________
## dense (Dense)                       (None, 10)                      650
## ================================================================================
## Total params: 93,322
```

Training the convnet on MNIST images

```
# mnist <- dataset_mnist() # get the data from the internet; ~200MB
# write_rds(mnist, "mnist.rds")
mnist <- read_rds("mnist.rds") # assuming that you have saved the data locally
c(c(train_images, train_labels), c(test_images, test_labels)) %<-% mnist

train_images <- array_reshape(train_images, c(60000, 28, 28, 1))
train_images <- train_images / 255

test_images <- array_reshape(test_images, c(10000, 28, 28, 1))
test_images <- test_images / 255

train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)

model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
model %>% fit(
  train_images, train_labels, # For simplicity, we use a small number for
  epochs = 5, batch_size=64   # epochs and batch_size
)
```

Let's evaluate the model on the test data:

```
results <- model %>% evaluate(test_images, test_labels)
results
```

```
## $loss
## [1] 0.03152971
##
## $acc
## [1] 0.9919
```

Even thought the convent is small and we use smaller `epochs` and `batch_size`, the final accuracy turns out higher than the network we built previously (0.981)! **WHY?**

To answer this, let's dive into what `layer_conv_2d` and `layer_max_pooling_2d` do.

The convolution operation

The fundamental difference between a densely connected layer and a convolution layer is this: **dense layers learn global patterns in their input feature space** (for example, for an MNIST digit, patterns involving all pixels), **whereas convolution layers learn local patterns** (that is, patterns in pixels in a small areas).

In the case of images, patterns found in small 2D windows of the inputs. In the previous example, these windows were all 3 × 3 (that is, `kernel_size`).

Image kernels: An image kernel is a small matrix used to apply effects like the ones we might find in Photoshop or Gimp, such as blurring, sharpening, outlining or embossing.

They're also used in machine learning for 'feature extraction', a technique for determining the most important portions of an image. In this context the process is referred to more generally as "*convolution*"

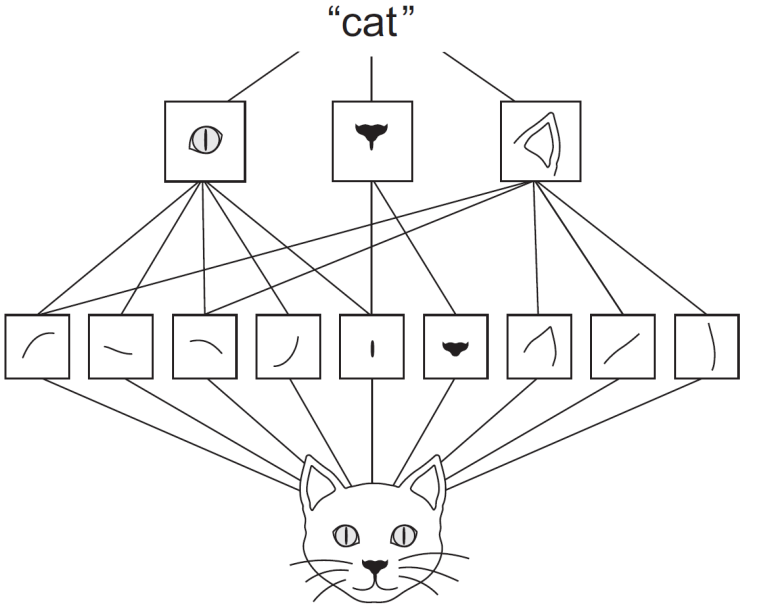
#DEMO

Source: http://setosa.io/ev/image-kernels/

The convolution operation leads to two interesting properties of convnets:

- **The patterns they learn are "translation invariant"**
  - After learning a certain pattern in the upper-right corner of a picture, a convnet can recognize it anywhere: for example, in the lower-left corner.
  - This makes convnets efficient when processing images because the visual world is fundamentally translation invariant
  - Need fewer training samples to learn representations

- **They can learn spatial hierarchies of patterns**
  - A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on.



The max-pooling operation

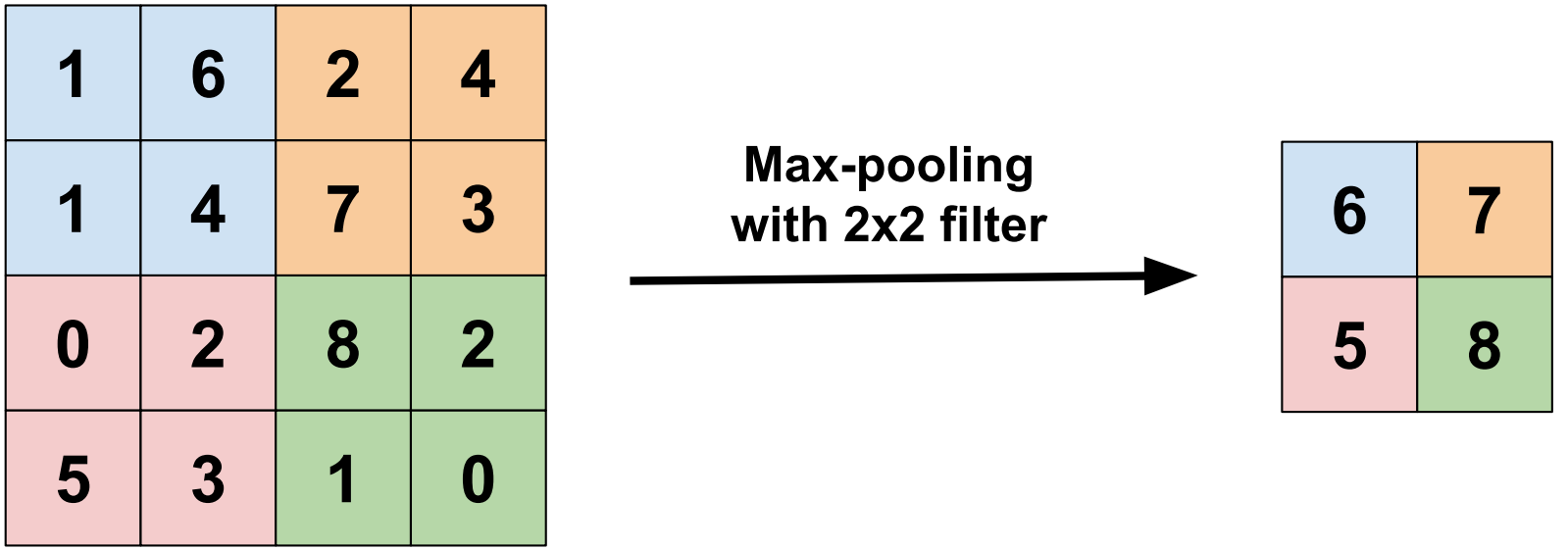
The role of max pooling is to aggressively **"downsample"** the output from the previous layer. In our model, the first `layer_max_pooling_2d` reduces the shape from 26 to 13 and the second one from 11 to 5.

```
summary(model)
```

```
## Model: "sequential_1"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
## _________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)  (None, 13, 13, 32)     0
## _________________________________________________________________________
## conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
## _________________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)  (None, 5, 5, 64)       0
## _________________________________________________________________________
## conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
## _________________________________________________________________________
## flatten_1 (Flatten)          (None, 576)               0
## _________________________________________________________________________
## dense_1 (Dense)              (None, 64)                36928
## _________________________________________________________________________
## dense_2 (Dense)              (None, 10)                650
## =========================================================================
## Total params: 93,322
## Trainable params: 93,322
```

Why downsampling?

The reason to use downsampling is to reduce the number of output coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows.



Recap

What we have learned so far ...

- Use an existing, medium-sized image dataset (MNIST)
- Construct a basic, small convnet
- Discuss why convnets are powerful for computer vision
- Basic operators in convnets

However, the real-world deep learning practice is a bit different:

- You need to construct your own image dataset
- You may not have many images
- You often want to have a larger and more advanced convnet

Let's train a convnet from scratch

The Dogs vs. Cats dataset

- Download from https://www.kaggle.com/c/dogs-vs-cats/data or https://www.dropbox.com/s/dvwkojxvdn86h7p/dogs-vs-cats.zip (faster)
- 837MB (compressed)
- 25,000 images of dogs and cats (12,500 from each class)
- Medium-resolution color JPEGs; images have different sizes

Please download the data and make sure that after decompressing the file in your working directory, you have the following folder structure:

```
YOUR_WORKING_DIRECTORY
+   data
    +   dogs-vs-cats
        +   train
            +   cat.0.jpg
            +   cat.1.jpg
            +   ...
        +   test
            +   1.jpg
            +   2.jpg
            +   ...
```

Create some folders to organize the data:

```
setwd("C:/CIS8392/") # make sure your working directory is correct
original_dataset_dir <- "data/dogs-vs-cats/train" # we will only use the labelled data

base_dir <- "data/cats_and_dogs_small" # to store a sebset of data that we are going to use
dir.create(base_dir)

train_dir <- file.path(base_dir, "train")
dir.create(train_dir)

validation_dir <- file.path(base_dir, "validation")
dir.create(validation_dir)

test_dir <- file.path(base_dir, "test")
dir.create(test_dir)

train_cats_dir <- file.path(train_dir, "cats")
dir.create(train_cats_dir)

train_dogs_dir <- file.path(train_dir, "dogs")
dir.create(train_dogs_dir)

validation_cats_dir <- file.path(validation_dir, "cats")
dir.create(validation_cats_dir)

validation_dogs_dir <- file.path(validation_dir, "dogs")
dir.create(validation_dogs_dir)

test_cats_dir <- file.path(test_dir, "cats")
dir.create(test_cats_dir)

test_dogs_dir <- file.path(test_dir, "dogs")
dir.create(test_dogs_dir)
```

Pretend that we only have a small image dataset:

- train: 1,000 cats and 1,000 dogs
- validation: 500 cats and 500 dogs
- test: 500 cats and 500 dogs

```
fnames <- paste0("cat.", 1:1000, ".jpg")
file.copy(file.path(original_dataset_dir, fnames), file.path(train_cats_dir))

fnames <- paste0("cat.", 1001:1500, ".jpg")
file.copy(file.path(original_dataset_dir, fnames), file.path(validation_cats_dir))

fnames <- paste0("cat.", 1501:2000, ".jpg")
file.copy(file.path(original_dataset_dir, fnames), file.path(test_cats_dir))

fnames <- paste0("dog.", 1:1000, ".jpg")
file.copy(file.path(original_dataset_dir, fnames), file.path(train_dogs_dir))

fnames <- paste0("dog.", 1001:1500, ".jpg")
file.copy(file.path(original_dataset_dir, fnames), file.path(validation_dogs_dir))

fnames <- paste0("dog.", 1501:2000, ".jpg")
file.copy(file.path(original_dataset_dir, fnames), file.path(test_dogs_dir))
```

As a sanity check, let's count how many pictures are in each training split (train/validation/test):

```
cat("total training cat images:", length(list.files(train_cats_dir)), "\n")
```

```
## total training cat images: 1000
```

```
cat("total training dog images:", length(list.files(train_dogs_dir)), "\n")
```

```
## total training dog images: 1000
```

```
cat("total validation cat images:", length(list.files(validation_cats_dir)), "\n")
```

```
## total validation cat images: 500
```

```
cat("total validation dog images:", length(list.files(validation_dogs_dir)), "\n")
```

```
## total validation dog images: 500
```

```
cat("total test cat images:", length(list.files(test_cats_dir)), "\n")
```

```
## total test cat images: 500
```

```
cat("total test dog images:", length(list.files(test_dogs_dir)), "\n")
```

```
## total test dog images: 500
```

Building your network

We built a small convnet for MNIST in the previous example, so we should be familiar with such convnets. We'll reuse the same general structure: the convnet will be a stack of alternated `layer_conv_2d` (with `relu` activation) and `layer_max_pooling_2d` stages.

But we'll make our network larger since this is a more complex task:

```
library(keras)
model_v1 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  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_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")

model_v1 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(),
  metrics = c("acc")
)
```

```
summary(model_v1)
```

```
## Model: "sequential_1"
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## conv2d_6 (Conv2D)            (None, 148, 148, 32)      896
## _________________________________________________________________
## max_pooling2d_5 (MaxPooling2D)  (None, 74, 74, 32)     0
## _________________________________________________________________
## conv2d_5 (Conv2D)            (None, 72, 72, 64)        18496
## _________________________________________________________________
## max_pooling2d_4 (MaxPooling2D)  (None, 36, 36, 64)     0
## _________________________________________________________________
## conv2d_4 (Conv2D)            (None, 34, 34, 128)       73856
## _________________________________________________________________
## max_pooling2d_3 (MaxPooling2D)  (None, 17, 17, 128)    0
## _________________________________________________________________
## conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
## _________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)  (None, 7, 7, 128)      0
## _________________________________________________________________
## flatten_1 (Flatten)          (None, 6272)              0
## _________________________________________________________________
## dense_3 (Dense)              (None, 512)               3211776
## _________________________________________________________________
## dense_2 (Dense)              (None, 1)                 513
## =================================================================
```

Data preprocessing

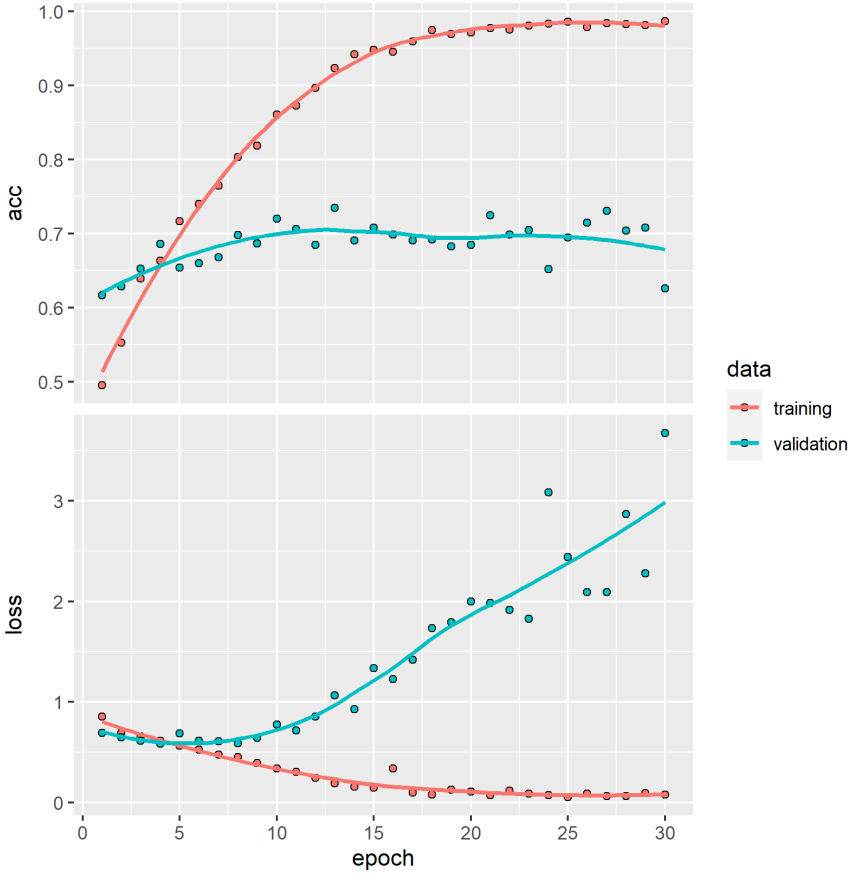
1. Read the image files.
2. Decode the JPEG content to RGB grids of pixels.
3. Convert these into floating-point tensors.
4. Rescale the pixel values (between 0 and 255) to the [0, 1] interval (neural networks prefer to deal with small input values).

It may seem a bit daunting, but thankfully Keras has utilities to take care of these steps automatically.

```
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,                  # Target directory
  train_datagen,              # Training data generator
  target_size = c(150, 150),  # Resizes all images to 150 × 150
  batch_size = 20,            # 20 samples in one batch
  class_mode = "binary"       # Because we use binary_crossentropy loss,
                              # we need binary labels.
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
```

Let's fit the model to the data using the generator. We do so using the `fit_generator` function, the equivalent of `fit` for data generators like this one. It expects as its first argument a generator that will yield batches of inputs.

```
# Took 90 minutes on my machine
history_v1 <- model_v1 %>%
  fit_generator(
    train_generator,
    steps_per_epoch = 100,
    epochs = 30,
    validation_data =
      validation_generator,
    validation_steps = 50
)

plot(history_v1)
```



Whenever the training and validation performances are diverging, it is an indicator of **over-fitting**.

This is very common when applying complex model to small datasets so that noises are mis-characterized as signals.

Use pretrained model and history

Download pretrained models (and their histories) here:
https://www.dropbox.com/s/hpdhvzrx9p9etnl/deeplearning-pretrained.zip

If you are using the VM, the pretrained model and history files are stored in `C:/CIS8392/data/deeplearning-pretrained/`

```
model_file = "data/deeplearning-pretrained/cats_vs_dogs_convnet_v1.h5"
history_file = "data/deeplearning-pretrained/cats_vs_dogs_convnet_v1_history.rds"
model_v1 <- load_model_hdf5(model_file)
history_v1 <- read_rds(history_file)
```

Avoiding over-fitting with data augmentation

Given infinite data, our model would be exposed to every possible aspect of the data distribution at hand: we 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.

In other words, we can make up new images from our small image dataset, and add them to our training process.

Setting up a data augmentation configuration via `image_data_generator`

```
datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,      # randomly rotate images up to 40 degrees
  width_shift_range = 0.2,  # randomly shift 20% pictures horizontally
  height_shift_range = 0.2, # randomly shift 20% pictures vertically
  shear_range = 0.2,        # randomly apply shearing transformations
  zoom_range = 0.2,         # randomly zooming inside pictures
  horizontal_flip = TRUE,   # randomly flipping half the images horizontally
  fill_mode = "nearest"     # used for filling in newly created pixels
)
```

Let's look at the augmented images:

```
fnames <- list.files(train_cats_dir, full.names = TRUE)
img_path <- fnames[[3]] # Chooses one image to augment

img <- image_load(img_path, target_size = c(150, 150))
img_array <- image_to_array(img) # Converts the shape back to (150, 150, 3)
img_array <- array_reshape(img_array, c(1, 150, 150, 3))

augmentation_generator <- flow_images_from_data(
  img_array,
  generator = datagen,
  batch_size = 1
)

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

(see next slide)




While data augmentation is helpful, the generated images still heavily intercorrelated.

We can't produce new information, we can only remix existing information. As such, this may not be enough to completely get rid of overfitting.

To further fight overfitting, we'll also add a dropout layer to our model, right before the densely connected classifier.

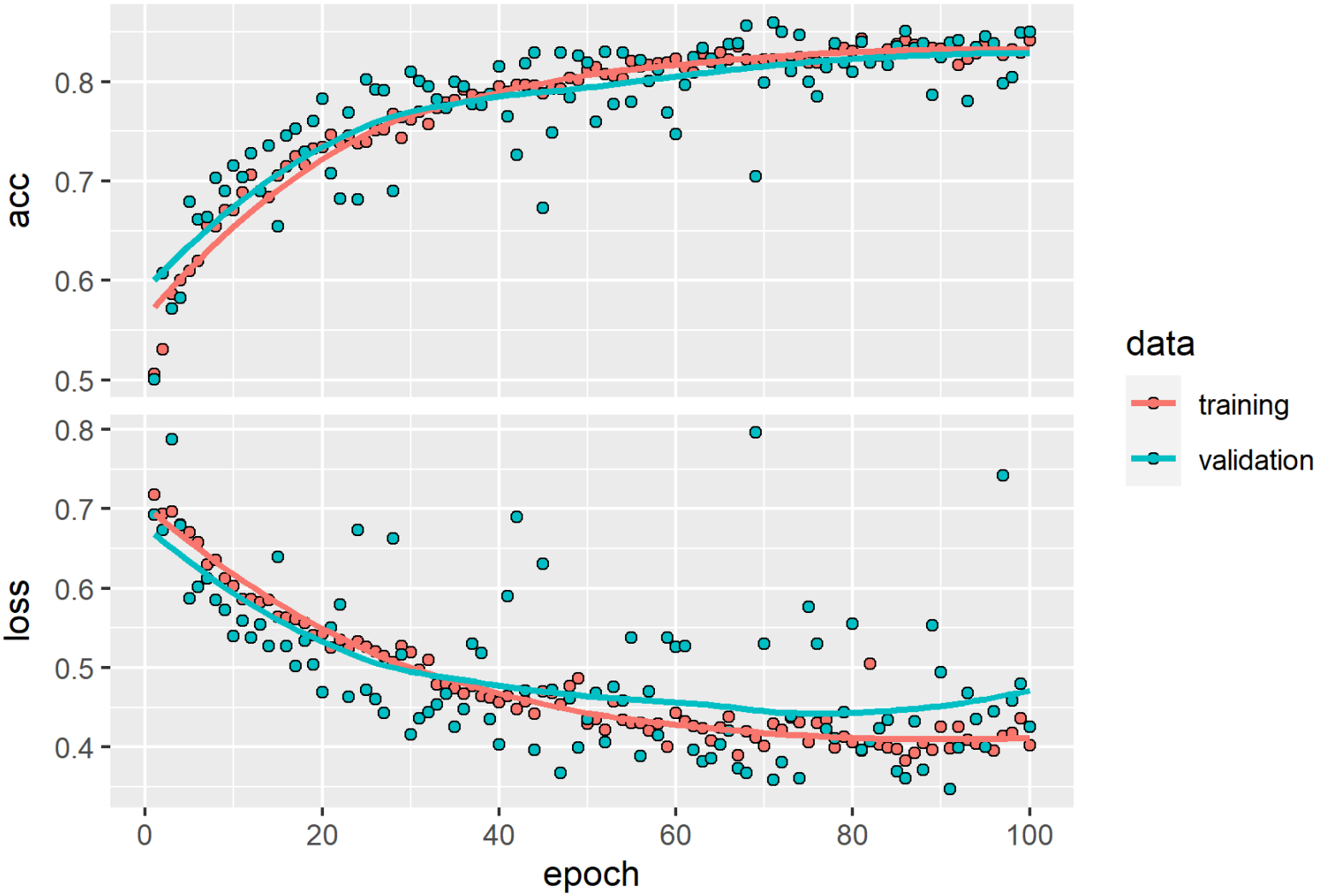
Defining a new convnet that includes dropout:

```
model_v2 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  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_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%  # randomly set 50% of weights to 0
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model_v2 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)
```

Training the convnet using data-augmentation generators:

```
test_datagen <- image_data_generator(rescale = 1/255) # no data augmentation

train_generator <- flow_images_from_directory(
  train_dir,
  datagen, # Our data augmentation configuration defined earlier
  target_size = c(150, 150),
  batch_size = 32,
  class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,   # Note that the validation data shouldn't be augmented!
  target_size = c(150, 150),
  batch_size = 32,
  class_mode = "binary"
)

history_v2 <- model_v2 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 100,
  validation_data = validation_generator,
  validation_steps = 50
)
```

```
plot(history_v2)  # no more overfitting
```

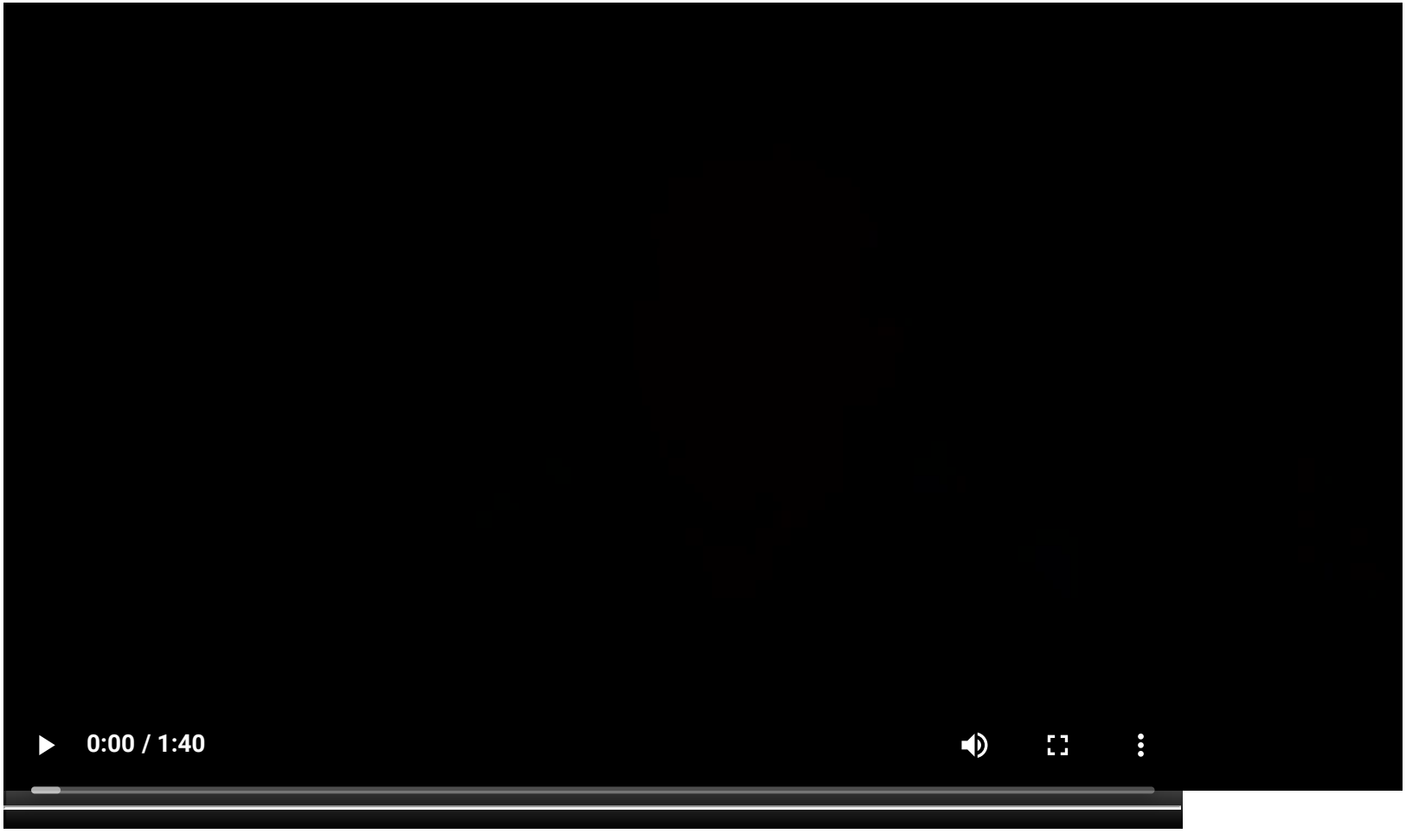


Let's evaluate this model using the test data:

```
test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,    # Note that the test data shouldn't be augmented!
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)

model_v2 %>% evaluate_generator(test_generator, steps = 50)
```

```
##      loss       acc
## 0.6459505 0.8260000
```

NVIDIA's use of data augmentation

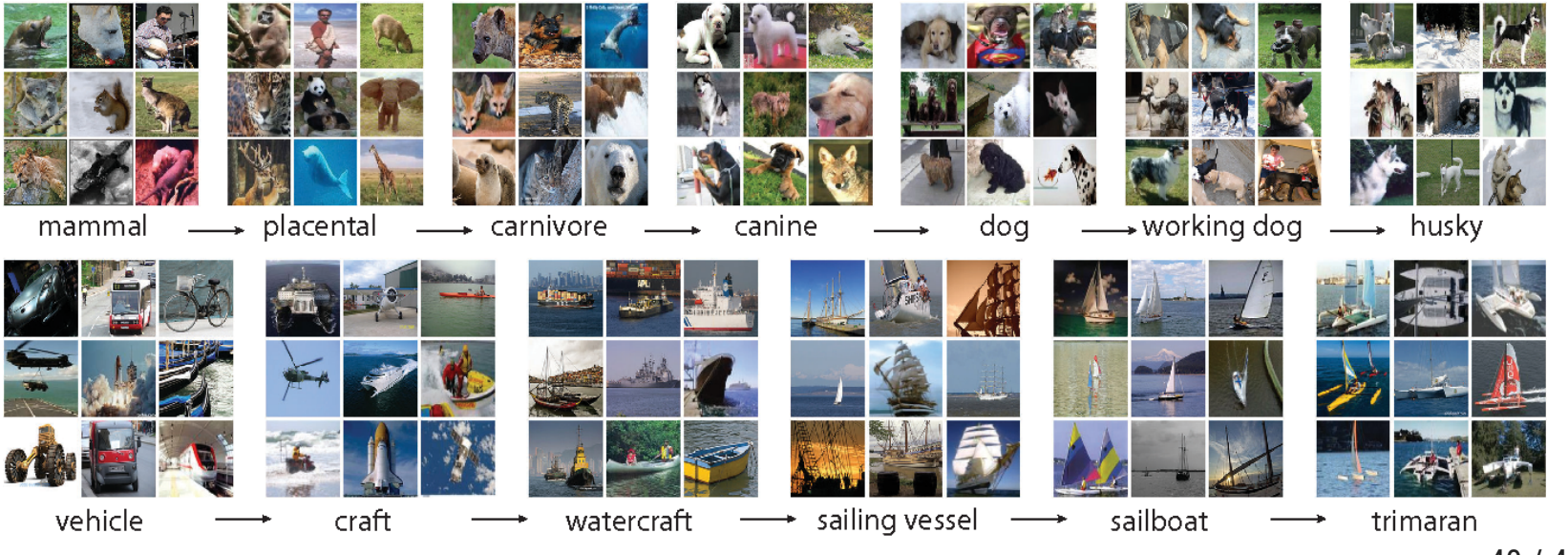


- Video source: https://www.youtube.com/watch?v=nh9oiz3F9ZA
- Discussion: https://blogs.nvidia.com/blog/2020/12/07/neurips-research-limited-data-gan/

Building on a pretrained convnet

A pretrained network is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. This is a common and highly effective approach to deep learning on small image datasets is to use a pretrained network.

When the original dataset is large and general enough, then the spatial hierarchy of features learned by the pretrained network can effectively act as a generic model of the visual world and applied to any specific problems.



Pretrained image classification models in keras

The following image classification models, pre-trained on ImageNet (1.4 million labeled images and 1,000 different classes), are available in keras:

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- InceptionResNetV2
- MobileNet
- MobileNetV2
- DenseNet
- NASNet

Let's instantiate the VGG16 convolutional base:

```
library(keras)

conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150, 150, 3)
)

conv_base
```

The parameter `include_top` indicates whether or not to include the last layer (1,000 classes). We don't want it because our problem is to classify only cats and dogs.

Notice how large is the model!

```
## Model
## Model: "vgg16"
## _________________________________________________________________
## Layer (type)                 Output Shape              Param
## =================================================================
## input_1 (InputLayer)         [(None, 150, 150, 3)]     0
## _________________________________________________________________
## block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
## _________________________________________________________________
## block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
## _________________________________________________________________
## block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
## _________________________________________________________________
## block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
## _________________________________________________________________
## block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
## _________________________________________________________________
## block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
## _________________________________________________________________
## block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
## _________________________________________________________________
## block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
## _________________________________________________________________
## block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
## _________________________________________________________________
## block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
## _________________________________________________________________
## block4_conv1 (Conv2D)        (None, 18, 18, 512)       118016
## _________________________________________________________________
## block4_conv2 (Conv2D)        (None, 18, 18, 512)       235980
## _________________________________________________________________
## block4_conv3 (Conv2D)        (None, 18, 18, 512)       235980
## _________________________________________________________________
## block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0
## _________________________________________________________________
## block5_conv1 (Conv2D)        (None, 9, 9, 512)         235980
## _________________________________________________________________
## block5_conv2 (Conv2D)        (None, 9, 9, 512)         235980
## _________________________________________________________________
## block5_conv3 (Conv2D)        (None, 9, 9, 512)         235980
## _________________________________________________________________
## block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
## =================================================================
## Total params: 14,714,688
```

How to adapt it to our specific classification problem? Two options:

1. Running `conv_base` over our dataset, recording its output to an array on disk, and then using this data as input to a standalone, densely connected classifier similar to those we saw in our first MINST model (without any `layer_conv_2d`).

    - fast and cheap
    - cannot apply data augmentation

2. Extending `conv_base` by adding dense layers on top, and running the whole thing end to end on the input data.

    - can apply data augmentation
    - far more expensive than the first option
    - only do this when you have access to a GPU; it's absolutely intractable on a CPU

We will only demonstrate the first option.

Fast feature extraction without data augmentation

Recall that the first option to adapt a pretrained model is to run `conv_base` over our dataset and collect the output features. So let's write a function for this:

```
datagen <- image_data_generator(rescale = 1/255)
batch_size <- 20
extract_features <- function(directory, sample_count) {
  features <- array(0, dim = c(sample_count, 4, 4, 512))
  labels <- array(0, dim = c(sample_count))
  generator <- flow_images_from_directory(
    directory = directory, generator = datagen,
    target_size = c(150, 150), batch_size = batch_size, class_mode = "binary"
  )
  i <- 0
  while(TRUE) {
    batch <- generator_next(generator)
    inputs_batch <- batch[[1]]; labels_batch <- batch[[2]]
    features_batch <- conv_base %>% predict(inputs_batch)
    index_range <- ((i * batch_size)+1):((i + 1) * batch_size)
    features[index_range,,,] <- features_batch
    labels[index_range] <- labels_batch
    i <- i + 1
    if (i * batch_size >= sample_count) break
  }
  return(list(features = features,labels = labels))
}
train <- extract_features(train_dir, 2000) # will take a while since we are running
validation <- extract_features(validation_dir, 1000) # our images through conv_base
test <- extract_features(test_dir, 1000) # still faster than training such a model
```

The extracted features are currently of shape `(samples, 4, 4, 512)`. This is the last layer of `conv_base`.

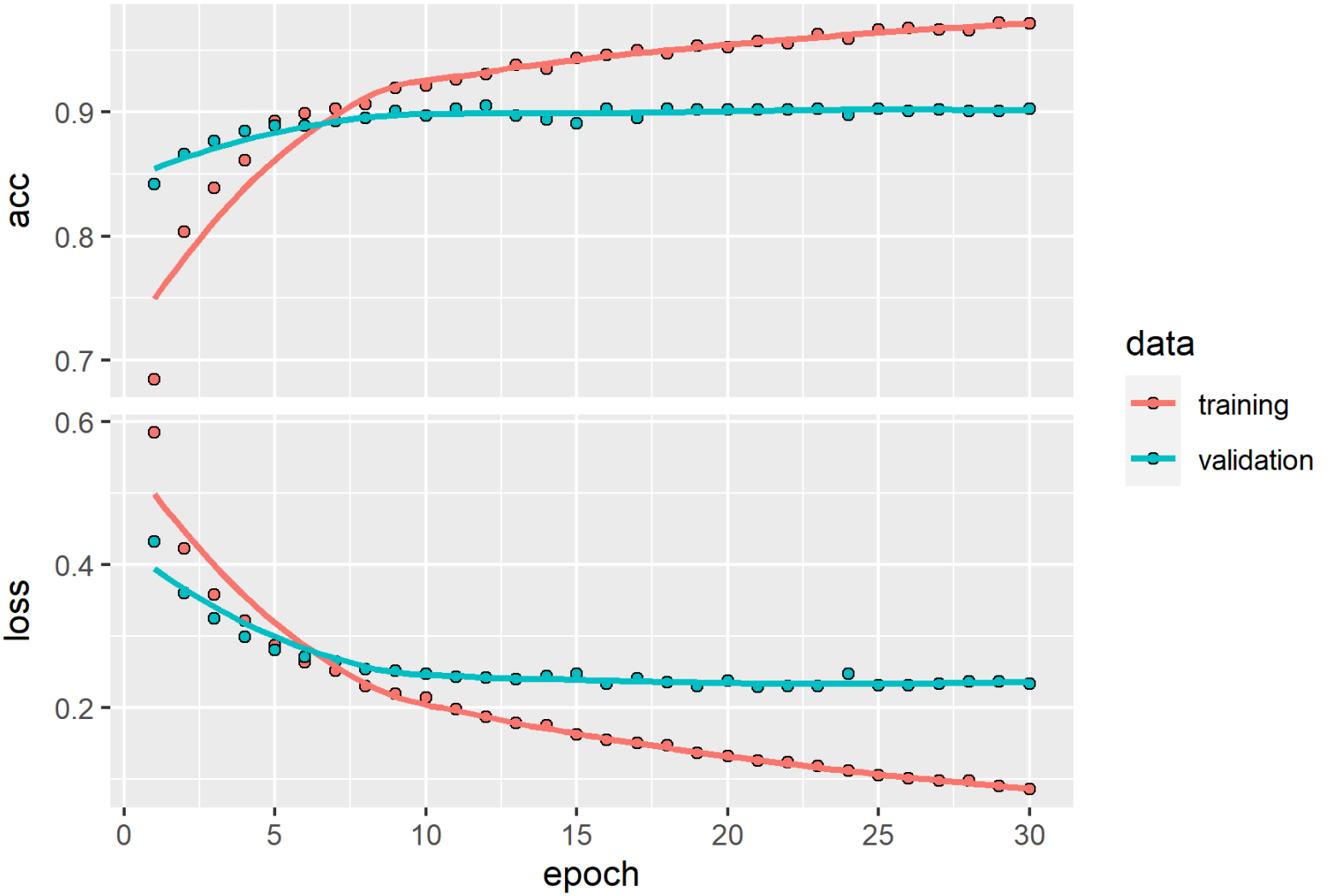
We'll feed them to a densely connected classifier, so first we must flatten them to `(samples, 8192)`. Recall this is exactly like how we reshape MNIST images from a shape of `(samples, 28, 28, 1)` to a shape of `(samples, 784)`:

```
reshape_features <- function(features) {
  array_reshape(features, dim = c(nrow(features), 4 * 4 * 512))
}

train$features <- reshape_features(train$features)
validation$features <- reshape_features(validation$features)
test$features <- reshape_features(test$features)
```

At this point, we can define our densely connected classifier (note the use of dropout for regularization) and train it on the data and labels that we just recorded.

```
model_v3 <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "relu",
              input_shape = 4 * 4 * 512) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid")

model_v3 %>% compile(
  optimizer = optimizer_rmsprop(lr = 2e-5),
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history_v3 <- model_v3 %>% fit(
  train$features, train$labels,
  epochs = 30,
  batch_size = 20,
  validation_data = list(validation$features, validation$labels)
)
```

```
plot(history_v3)
```



We can now finally evaluate this model on the test data:

```
model_v3 %>% evaluate(test$features, test$labels)
```

```
##      loss  accuracy
## 0.2592897 0.8820000
```

The accuracy has improved from 0.826 (data augmentation without using any pretrained model) to 0.882 (using pretrained model without data augmentation)!!

Your turn

Based on the example in the previous slides, try to add one more densely connected layer to our network before the `sigmoid` layer. Can this improve accuracy?

Try to experiment with a different pretrained model and see how the accuracy on test data changes.