Deep Learning

Convolutional Neural Networks

Hands-on

POLITÉCNICO DE LEIRIA

Slides by Carlos Grilo & Rolando Miragaia

CNN from scratch to recognize handwritten digits

- We will first use the MINST dataset to create, train and test a CNN from scratch to recognize handwritten digits

- Create a new notebook named

  04_CNN_01_MNIST.ipynb

- Add the following cell code to mount your Google Drive on Colab:

```
from google.colab import drive
drive.mount('/content/drive')
```

Instantiating a small CNN

We will start by creating a CNN to be trained also with the MNIST dataset

```
from keras import layers
from keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28
, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

This is just the feature extraction section

Network's structure

- The network is composed of the following layers:
  - A convolutional layer with 32 channels (filters/feature maps) with 3x3 filters and with the ReLU activation function
  - A maxpooling layer with a 2x2 pooling window
  - A convolutional layer with 64 channels (filters/feature maps) with 3x3 filters and with the ReLU activation function
  - A maxpooling layer with a 2x2 pooling window
  - A convolutional layer with 64 channels (filters/feature maps) with 3x3 filters and with the ReLU activation function

Network's structure

```
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)   (None, 13, 13, 32)     0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)   (None, 5, 5, 64)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

Exercise: compute these values

Network's structure

```
Layer (type)                     Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)        320
```

- `(None, 26, 26, 32)` means that the layer's output is a volume where each feature map has shape 26x26 and there are 32 channels ("None" -> the batch size is unknown)
- Feature maps are 26x26 because no padding is being used (default) and strides = 1 (default)
- See https://keras.io/api/layers/convolution_layers/convolution2d/

Network’s structure

```
 Layer (type)                     Output Shape              Param #
================================================================
conv2d_1 (Conv2D)                 (None, 26, 26, 32)        320
```

- The number of parameters (weights) in the first layer is 320:
  - There are 32 x 3x3 filters -> 288 parameters
  - We must add 32 bias values -> 320 parameters

Network’s structure

```
 Layer (type)                    Output Shape              Param #
================================================================
conv2d_1 (Conv2D)               (None, 26, 26, 32)        320
________________________________________________________________
maxpooling2d_1 (MaxPooling2D)   (None, 13, 13, 32)        0
```

- `(None, 13, 13, 32)` because padding is not being used (default) and strides=2 (by default, strides = pooling window dimensions)
- Max pooling layers don’t have associated parameters
- See https://keras.io/api/layers/pooling_layers/max_pooling2d/

Instantiating a small CNN

We now instantiate the classification section (dense network)

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Network’s structure

```
>>> model.summary()
Layer (type)                    Output Shape          Param #
=================================================================
conv2d_1 (Conv2D)               (None, 26, 26, 32)    320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)   (None, 13, 13, 32)    0
_________________________________________________________________
conv2d_2 (Conv2D)               (None, 11, 11, 64)    18496
_________________________________________________________________
maxpooling2d_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
Non-trainable params: 0
```

The 3x3x64 output of the last convolutional layer is flattened into a one-dimensional array of 576 elements

Flatten layers don’t have associated parameters

Training the network

```
from keras.datasets import mnist
from keras.utils import to_categorical
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['
accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Assessing the network's performance

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
test_acc

0.9922999739646912
```

```
model.save('/content/drive/MyDrive/models/04_CNN_01_MNIST.h5')
```

Predicting

```
import tensorflow as tf

digit = train_images[2]
print(digit.shape)
digit = digit[:, :, 0]
print(digit.shape)
image = tf.expand_dims(digit, 0)
print(image.shape)
result = model.predict(image)
print("Result: ", result.round())

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

We will compute the output of the model for the 3rd image in the training dataset

Output: (28, 28, 1)

This means that each image has in fact three dimensions: each pixel is an array of one element. Something like this:

$$\begin{array}{cc} 28 & \begin{bmatrix} [0] & [1] & \cdots & [0] \\ \vdots & & & \\ [0] & [0] & \cdots & [0] \end{bmatrix} \\ & 28 \end{array}$$

Predicting

```
import tensorflow as tf

digit = train_images[2]
print(digit.shape)
digit = digit[:, :, 0]
print(digit.shape)
image = tf.expand_dims(digit, 0)
print(image.shape)
result = model.predict(image)
print("Result: ", result.round())

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

However, Keras models expect to receive an array (batch) of images

In this case, it expects to receive a numpy array with shape (number_of_images, 28, 28)

That is, we can compute the output for more than one image at once

Predicting

```
import tensorflow as tf

digit = train_images[2]
print(digit.shape)
digit = digit[:, :, 0]
print(digit.shape)
image = tf.expand_dims(digit, 0)
print(image.shape)
result = model.predict(image)
print("Result: ", result.round())

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

So, we first remove the third dimension of the images, reshaping it from shape (28, 28, 1) to (28, 28)

Output: (28, 28)

Predicting

```
import tensorflow as tf

digit = train_images[2]
print(digit.shape)
digit = digit[:, :, 0]
print(digit.shape)
image = tf.expand_dims(digit, 0)
print(image.shape)
result = model.predict(image)
print("Result: ", result.round())

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

Now we had a first dimension (dimension 0) with size 1

Output: (1, 28, 28,)

We now have a numpy array that is able to save one 28x28 image (in our case, we will compute the output of the model for just one image)

Predicting

```python
import tensorflow as tf

digit = train_images[2]
print(digit.shape)
digit = digit[:, :, 0]
print(digit.shape)
image = tf.expand_dims(digit, 0)
print(image.shape)
result = model.predict(image)
print("Result: ", result.round())

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

$\longrightarrow$

Our image is a 4 digit

$\downarrow$

```
Result: [[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

Functional API

The model that we have built can be built using the Keras Functional API

```
from tensorflow import keras
from keras import layers

inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
x = layers.Dense(64, activation="relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Network's structure

```
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)    (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

Exercise: compute these values

Exercise

- Build a model with the following architecture (without and with the Keras Functional API) and train it:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)          (None, 28, 28, 1)    0
conv2d_9 (Conv2D)             (None, 26, 26, 32)   320
max_pooling2d_6 (MaxPooling2D)    (None, 13, 13, 32)   0
conv2d_10 (Conv2D)            (None, 11, 11, 64)   18496
max_pooling2d_7 (MaxPooling2D)    (None, 5, 5, 64)     0
conv2d_11 (Conv2D)            (None, 3, 3, 128)    73856
flatten_3 (Flatten)           (None, 1152)         0
dense_5 (Dense)               (None, 10)           11530
=================================================================
```

The dogs vs. cats example

- We will use the dogs vs. cats dataset from Kaggle (www.kaggle.com/c/dogs-vs-cats/)
- We need a Kaggle account to download the dataset
- We will use a smaller version of the dataset:
  - 2000 images for the training set, 1000 dogs and 1000 cats images
  - 1000 images for the validation set, 500 dogs and 500 cats images
  - 1000 images for the test set, 500 dogs and 500 cats images

Train, validation and test sets

- Train set: used to train the network
- Validation set: used to “test” the model during the training process
- Test set: used to test the model after the training process

Train, validation and test sets

- Why using both the validation and test sets?
  - Developing a model always involves tuning its configuration, for example, choosing the number of layers or the size of the layers
  - We do this tuning by using as a feedback signal the performance of the model on the validation data
  - As a result, tuning the configuration of the model based on its performance on the validation set can result in overfitting to the validation set, even though our model is never directly trained on it
  - So, we need the validation test and the test set

Using the dataset on Google Drive

- Copy your `cats_and_dogs_small.zip` file to your Google Drive directory
- Open the zip file with Zip Extractor and follow the instructions
- Create the 04_CNN_02_CatsAndDogs_01_cnn_from_scratch.ipynb notebook
- Add the following cell code to mount your Google Drive on Colab:

```
from google.colab import drive
drive.mount('/content/drive')
```

Showing directories' size

```
import os, shutil
train_dir = '/content/drive/MyDrive/cats_and_dogs_small/train'
validation_dir = '/content/drive/MyDrive/cats_and_dogs_small/validation'
test_dir = '/content/drive/MyDrive/cats_and_dogs_small/test'
train_cats_dir = '/content/drive/MyDrive/cats_and_dogs_small/train/cats'
train_dogs_dir = '/content/drive/MyDrive/cats_and_dogs_small/train/dogs'
val_cats_dir = '/content/drive/MyDrive/cats_and_dogs_small/validation/cats'
val_dogs_dir = '/content/drive/MyDrive/cats_and_dogs_small/validation/dogs'
test_cats_dir = '/content/drive/MyDrive/cats_and_dogs_small/test/cats'
test_dogs_dir = '/content/drive/MyDrive/cats_and_dogs_small/test/dogs'
print('total training cat images:', len(os.listdir(train_cats_dir)))
print('total training dog images:', len(os.listdir(train_dogs_dir)))
print('total validation cat images:', len(os.listdir(val_cats_dir)))
print('total validation dog images:', len(os.listdir(val_dogs_dir)))
print('total testing cat images:', len(os.listdir(test_cats_dir)))
print('total testing dog images:', len(os.listdir(test_dogs_dir)))
```

Output:

```
total training cat images: 1000
total training dog images: 1000
total validation cat images: 500
total validation dog images: 500
total testing cat images: 500
total testing dog images: 500
```

Preprocessing the data

```
from keras.utils import image_dataset_from_directory

IMG_SIZE = 150

train_dataset = image_dataset_from_directory(
  train_dir,
  image_size=(IMG_SIZE,  IMG_SIZE),
  batch_size=32)

validation_dataset = image_dataset_from_directory(
  validation_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)

test_dataset = image_dataset_from_directory(
  test_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)
```

See next slide

Preprocessing the data

```
…

train_dataset = image_dataset_from_directory(
  train_dir,
  image_size=(IMG_SIZE,  IMG_SIZE),
  batch_size=32)

…
```

The `image_dataset_from_directory` function returns a Dataset object that:

1. Reads the picture files
2. Decodes the JPEG content to RGB grids of pixels
3. Converts these into floating-point tensors
4. Resizes them to a shared size (we will use 150 × 150)
5. Packs them into batches (we will use batches of 32 images)

These operations are undertaken during the training process

The shape of each batch

```
for data_batch, labels_batch in train_dataset:
  print('data batch shape:', data_batch.shape)
  print('labels batch shape:', labels_batch.shape)
  break
```

```
data batch shape: (32, 150, 150, 3)
labels batch shape: (32,)
```

Viewing the first 5 images of the first batch

```
import matplotlib.pyplot as plt

for data_batch, _ in train_dataset.take(1):
  for i in range(5):
    plt.imshow(data_batch[i].numpy().astype("uint8"))
    plt.show()
```

Takes the first n batches of the dataset

In this case, just the first

Creating the neural network

```
from tensorflow import keras
from keras import layers
from keras import models
inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = layers.Rescaling(1./255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Flatten()(x)
x = layers.Dense(512, activation="relu")(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

$\longrightarrow$ Rescales images' pixels by 1/255

… and, then

```
>> model.summary()
```

Compiling the neural network

```
import tensorflow as tf
model.compile(
        loss='binary_crossentropy',
        optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),
        metrics=['acc'])
```

Because we have a binary classification problem

Training the model

```
history = model.fit(
  train_dataset,
  epochs=30,
  validation_data=validation_dataset)
```

Given that the training process takes several minutes, we will instead load an already trained model, see next slide

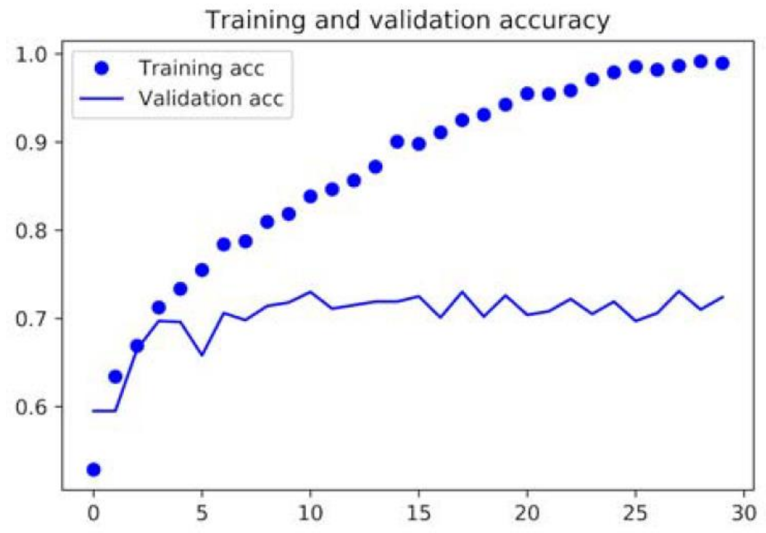
Loading and testing the model

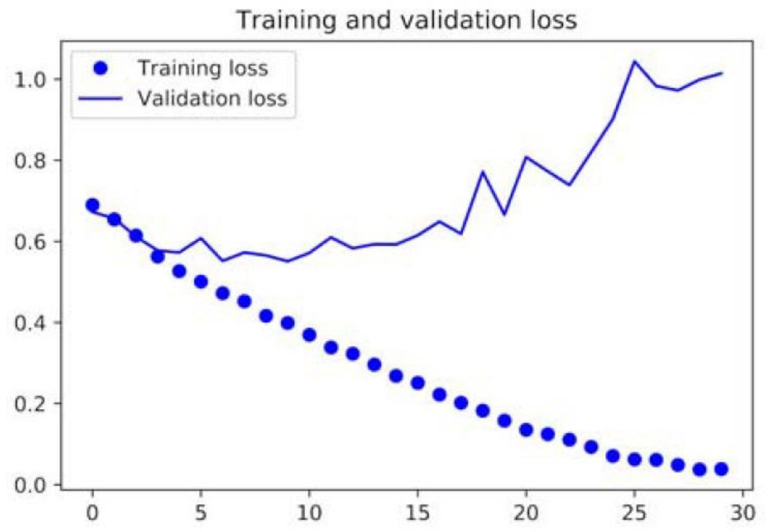
```
from tensorflow import keras
model = keras.models.load_model('/content/drive/MyDrive/models/04_CNN_02_CatsAndD
ogs_01_cnn_from_scratch.h5')
--------------------------
val_loss, val_acc = model.evaluate(validation_dataset)
print('val_acc:', val_acc)

32/32 [==============================] - 3s 65ms/step - loss: 0.6997 - acc: 0.7180
val_acc: 0.7179999947547913
```

Far from the result obtained with the training set

Displaying curves of loss and accuracy

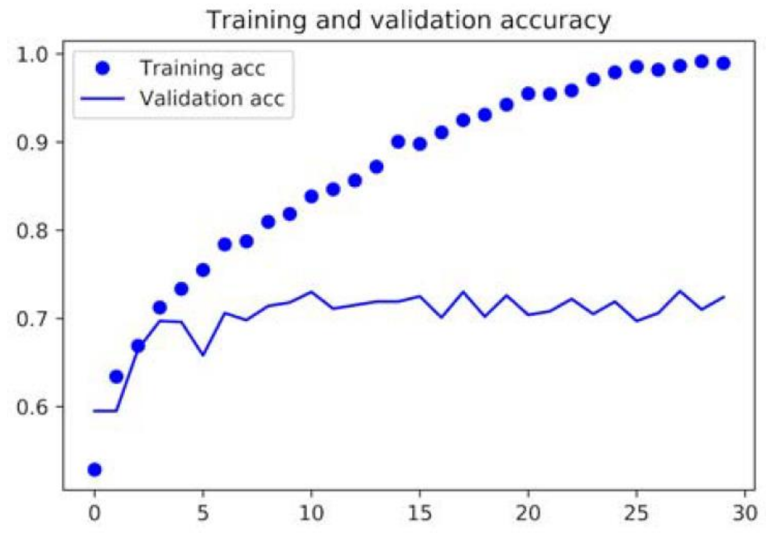
```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

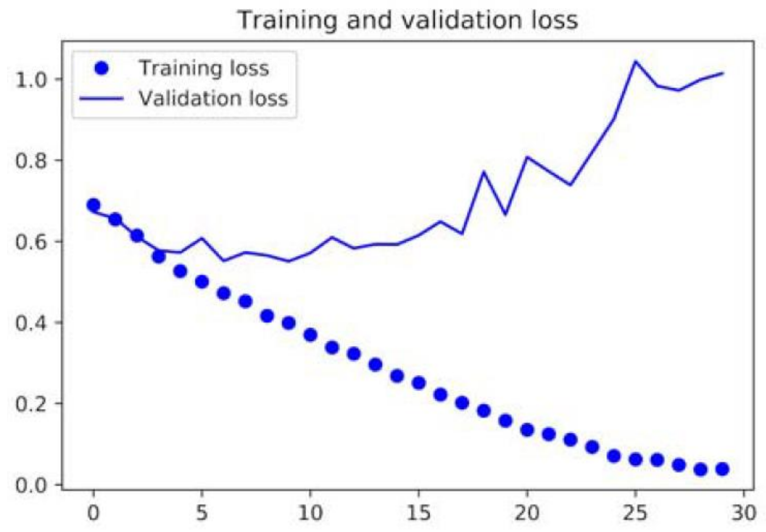




Overfitting

- These plots are characteristic of overfitting
- The training accuracy increases over time, until it reaches nearly 100%, whereas the validation accuracy stalls at 70–72%
- The validation loss reaches its minimum after only 5 epochs and then stalls, whereas the training loss keeps decreasing linearly until it reaches nearly 0





Computing the model output for one image

```
import tensorflow as tf
import matplotlib.pyplot as plt
from keras.preprocessing import image

img = tf.keras.preprocessing.image.load_img(
     '/content/drive/MyDrive/cats_and_dogs_small/train/cats/cat.1.jpg', target_size=(150, 150), interpolation='bilinear')
#img = tf.keras.preprocessing.image.load_img(
     '/content/drive/MyDrive/cats_and_dogs_small/train/dogs/dog.1.jpg', target_size=(150, 150), interpolation='bilinear')

plt.imshow(img)
plt.show()

img_array = tf.keras.preprocessing.image.img_to_array(img)

img_array = tf.expand_dims(img_array, 0)
print(img_array.shape)

result = model.predict(img_array)
print("Result: ", result.round())
```

"reshapes" the image so that it is put in an array. The size of the array will become dimension 0 (the first dimension).

image_array will become an array with shape (1, 150, 150, 3) -> an array with one image with shape (150, 150, 3).

Mitigating overfitting

- We will see two techniques to mitigate overfitting
  - Data augmentation: generating more training data from existing training samples, by augmenting the samples via a number of random transformations that yield believable-looking images
  - Dropout: consists of randomly dropping out (setting to zero) a number of output features of the layer to which it is applied during training
- Other techniques: reduce model's size and L1 and L2 regularization (dropout is a form of regularization)

Data augmentation

- Create a new notebook named

  04_CNN_02_CatsAndDogs_02_cnn_from_scratch_with_data_aumengtation.ipynb

- Add the following cell code to mount your Google Drive on Colab:

```
from google.colab import drive
drive.mount('/content/drive')
```

Data augmentation

```
import os, shutil
train_dir = '/content/drive/MyDrive/cats_and_dogs_small/train'
validation_dir = '/content/drive/MyDrive/cats_and_dogs_small/validation'
test_dir = '/content/drive/MyDrive/cats_and_dogs_small/test'


from keras.utils import image_dataset_from_directory

IMG_SIZE = 150

train_dataset = image_dataset_from_directory(
  train_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)

validation_dataset = image_dataset_from_directory(
  validation_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)

test_dataset = image_dataset_from_directory(
  test_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)
```

Copy this code from the previous notebook

Data augmentation

- In Keras, data augmentation can be done by adding a number of data augmentation layers at the start of our model

```
from tensorflow import keras
from keras import layers

data_augmentation = keras.Sequential(
  [
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.2),
  ]
)
```

Applies horizontal flipping to a random 50% of the images that go through it

Rotates the input images by a random value in the range [–10%, +10%] (these are fractions of a full circle—in degrees, the range would be [–36 degrees, +36 degrees])

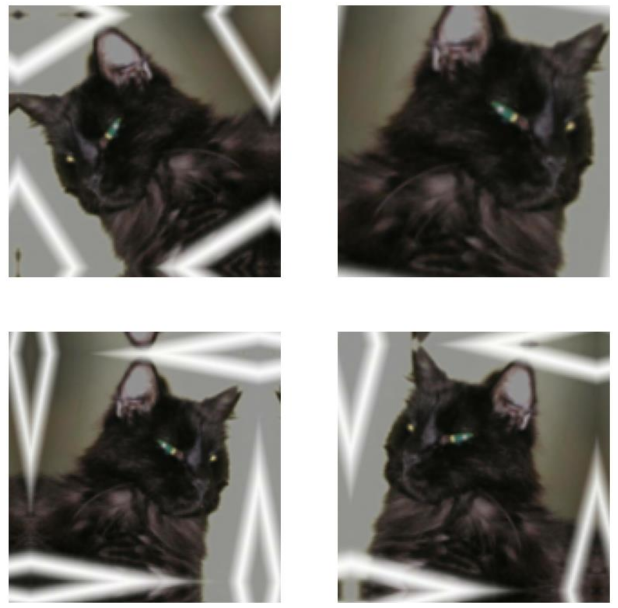
Zooms in or out of the image by a random factor in the range [-20%, +20%]

- These are just a few of the layers available (for more, see the Keras documentation)

Data augmentation

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, _ in train_dataset.take(1):
  for i in range(4):
    augmented_images = data_augmentation(images)
    ax = plt.subplot(2, 2, i + 1)
    plt.imshow(augmented_images[0].numpy().astype("uint8"))
    plt.axis("off")
```

Plotting some images generated by the data augmentation layers



Building the model with data augmentation and dropout

```
from tensorflow import keras
from keras import layers
from keras import models

inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
x = layers.Dense(512, activation="relu")(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

We add the data augmentation layers as the first layers of the model

We also add a dropout layer. In this case, it will be applied to the output of the Flatten layer

Compiling and training the model

```
import tensorflow as tf
model.compile(
        loss='binary_crossentropy',
        optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),
        metrics=['acc'])

-------------------------------------------

history = model.fit(
  train_dataset,
  epochs=30,
  validation_data=validation_dataset)
```

- Once again, we will load a previously trained model because the training process takes some time (a lot, in fact). See next slide

Loading and testing the model

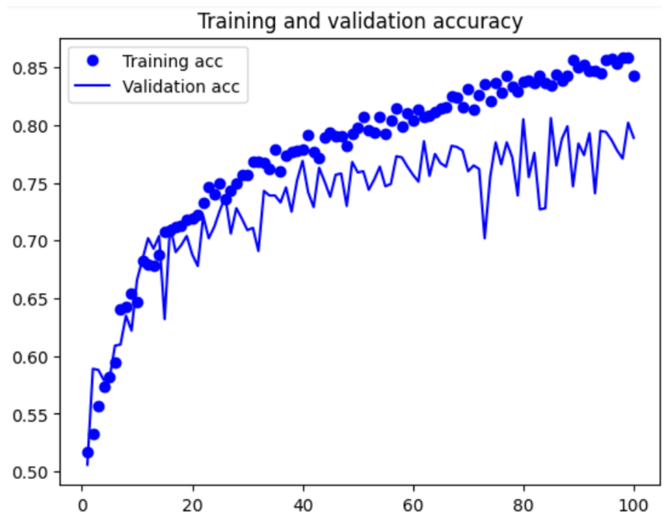
```
from tensorflow import keras
model = keras.models.load_model('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_02_cnn_from_scratch_with_data_aumengtation.h5')
---------------------------
val_loss, val_acc = model.evaluate(validation_dataset)
print('val_acc:', val_acc)
```

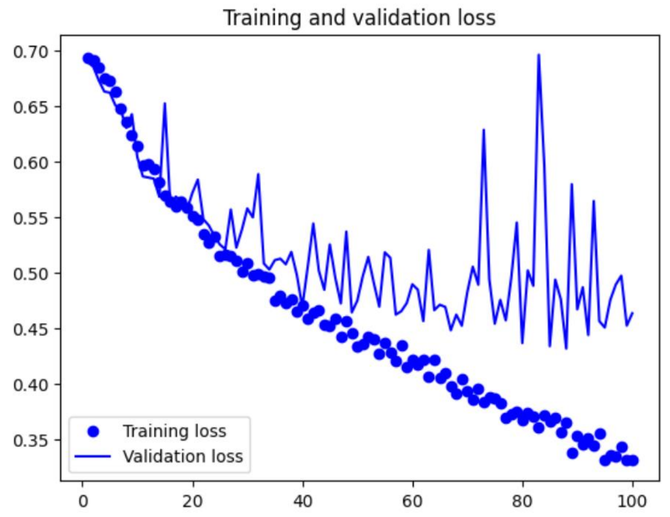
```
32/32 [==============================] - 2s 47ms/step - loss: 0.5392 - acc: 0.7760
val_acc: 0.7760000228881836
```

The result has improved considerably regarding our original model

Displaying curves of loss and accuracy

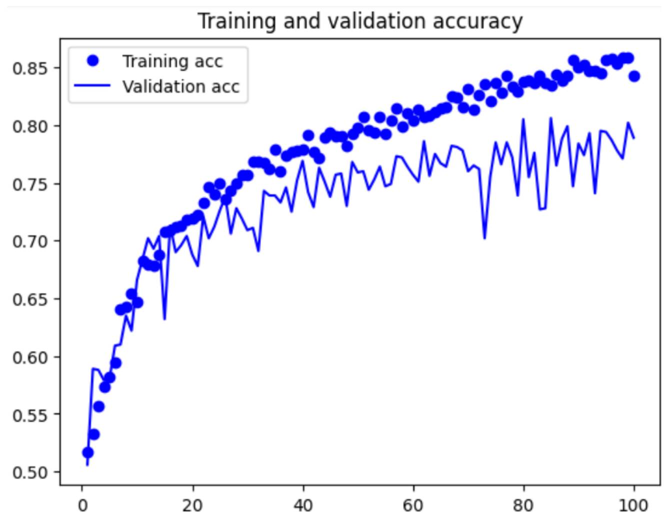
```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

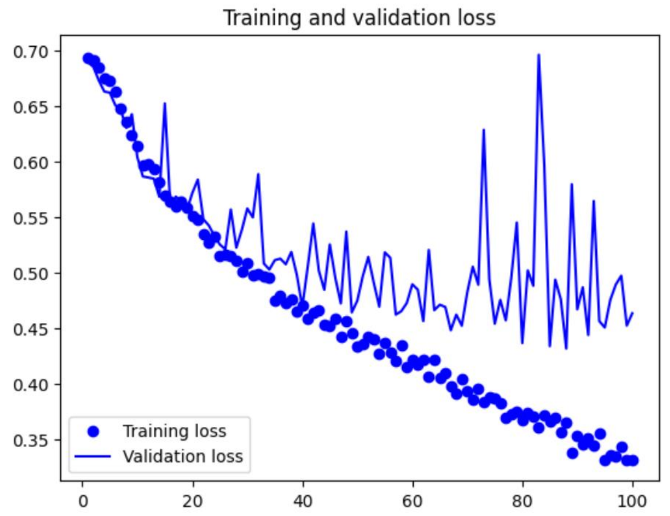




Displaying curves of loss and accuracy

- Thanks to data augmentation and dropout, we start overfitting much later, around epochs 60–70 (compared to epoch 5 for the original model)
- The validation accuracy ends up consistently in the 75–80% range, a big improvement over our first try





Transfer learning

- Often, our dataset is too small, which turns difficult to build a CNN with a good performance
- In these situations, we can use already existing networks that were previously trained on a large dataset, typically on a large-scale image-classification task
- If this original dataset is large enough and general enough, the spatial hierarchy of features learned by the pretrained network can act as a generic model of the visual world and its features can prove useful for many different computer vision problems

Transfer learning

- For instance, we might train a network on the ImageNet dataset (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

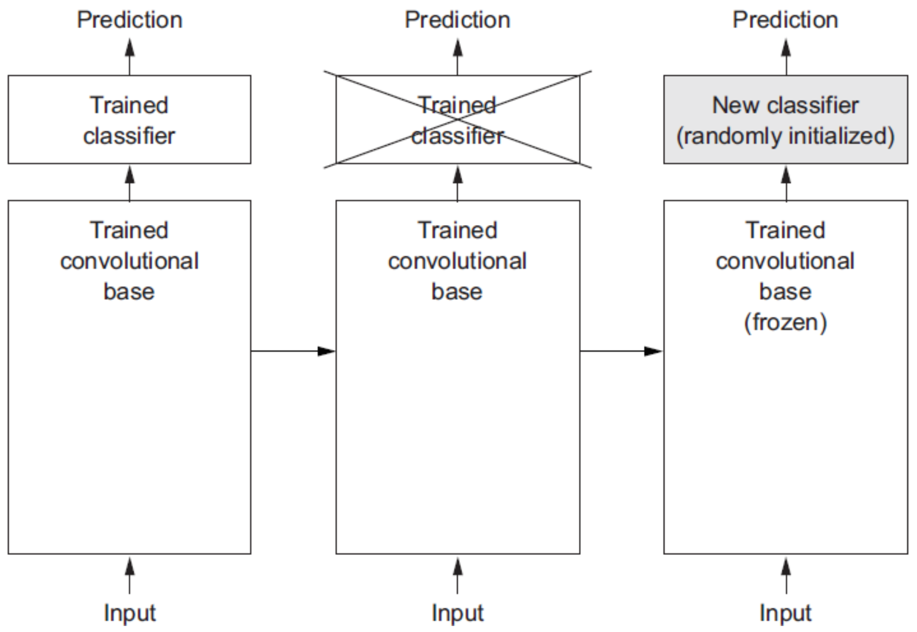
Transfer learning

- There are two ways of using a pretrained network:
  - Feature extraction
  - Fine-tuning

Transfer learning – feature extraction

- Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples

- These features are then run through a new classifier, which is trained from scratch



Prediction
Trained classifier
Trained convolutional base
Input
Prediction
~~Trained classifier~~
Trained convolutional base
Input
Prediction
New classifier (randomly initialized)
Trained convolutional base (frozen)
Input

We may just retrain the weights of the existing classification block

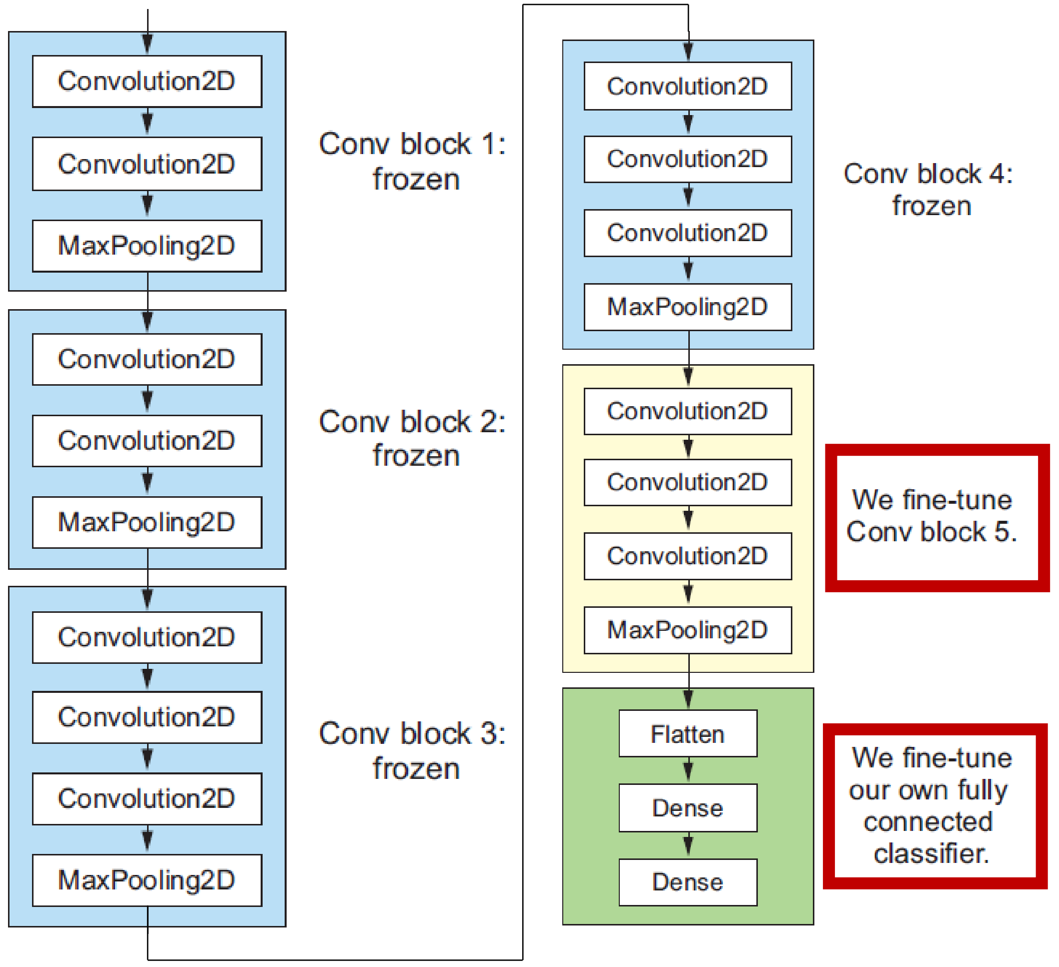
However, often, we need to change the architecture of this block (for example, change the number of nodes of the last layer)

Transfer learning – 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

Transfer learning – fine tuning



Example: fine-tuning the last convolutional block of the VGG16 network

TL – feature extraction without data augmentation

We will first use the “feature extraction” approach without using data augmentation

TL – feature extraction without data augmentation

- Create a new notebook named

  04_CNN_02_CatsAndDogs_03_TL_load_features_no_data_augmentation.ipynb

- Add the following cell code to mount your Google Drive on Colab:

```
from google.colab import drive
drive.mount('/content/drive')
```

TL – feature extraction without data augmentation

```
import os, shutil
train_dir = '/content/drive/MyDrive/cats_and_dogs_small/train'
validation_dir = '/content/drive/MyDrive/cats_and_dogs_small/validation'
test_dir = '/content/drive/MyDrive/cats_and_dogs_small/test'

from keras.utils import image_dataset_from_directory

IMG_SIZE = 150

train_dataset = image_dataset_from_directory(
  train_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)

validation_dataset = image_dataset_from_directory(
  validation_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)

test_dataset = image_dataset_from_directory(
  test_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)
```

Copy this code from the previous notebook

TL – feature extraction without data augmentation

Let us first load the VGG16 model

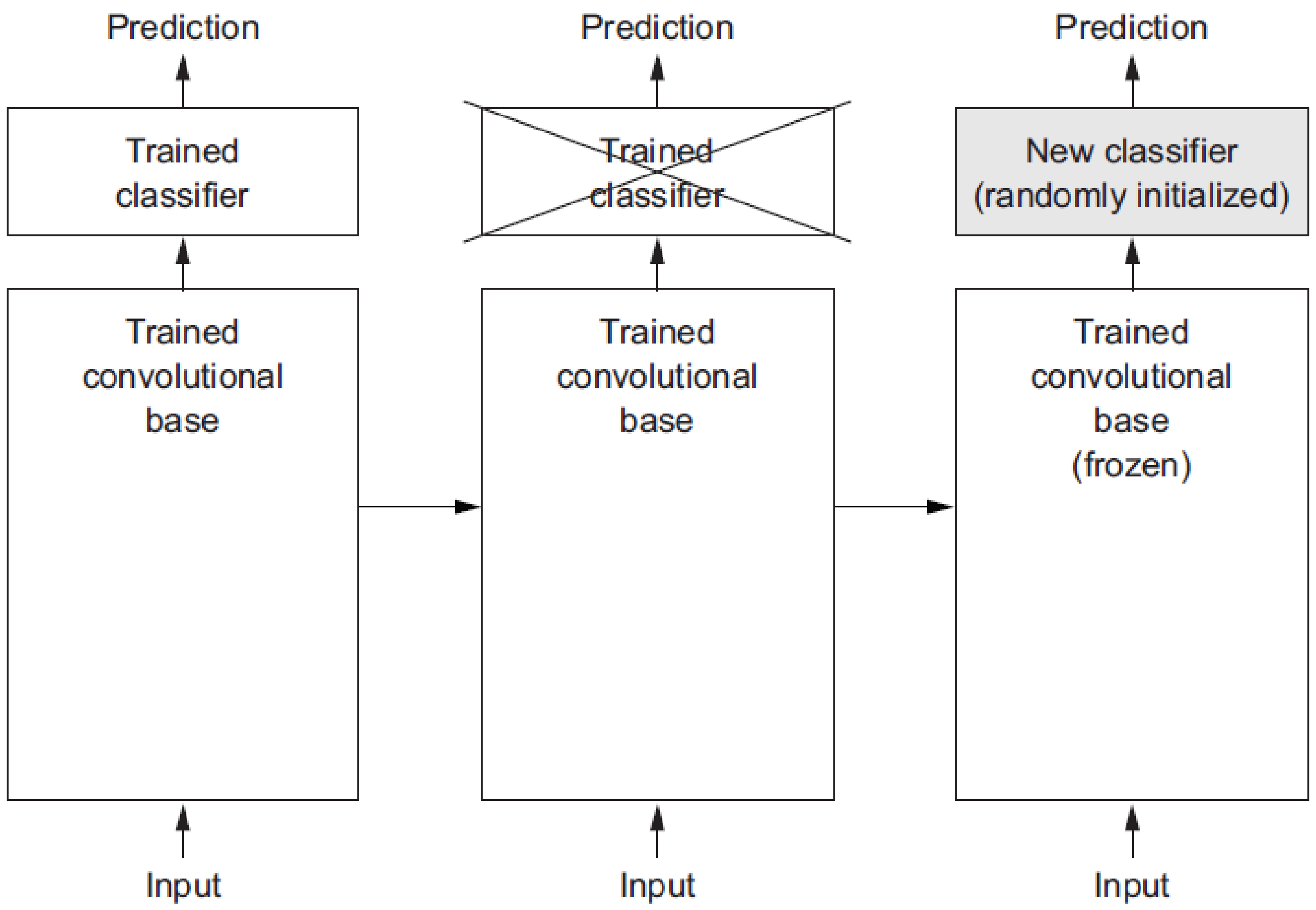
```
from tensorflow.keras.applications.vgg16 import VGG16
conv_base = VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))
```

- `weights`: specifies the weight checkpoint from which to initialize the model
- `include_top`: refers to including (or not) the densely connected classifier on top of the network. By default, this densely connected classifier corresponds to the 1,000 classes from ImageNet. Because we intend to use our own densely connected classifier (with only two classes: cat and dog), we don't need to include it in our example
- `input_shape`: is the shape of the image tensors that we'll feed to the network

TL – feature extraction without data augmentation

- In order to train the classification section of the model, we will need several epochs
- That is, we will need to present the train and validation sets several times to the model
- However, each epoch we will present always the same images (because we are not using data augmentation) and because we are not going to change the feature extraction section of the VGG16 model, the output of this section will be the same for all epochs
- This means that we can compute the output of this section of the model for each of the images in the three datasets (train, validation and test) just once
- This output can then be later used as input for the training process of the classification section of the model

Transfer learning – feature extraction


Prediction
Prediction
Prediction
Trained classifier
~~Trained classifier~~
New classifier (randomly initialized)
Trained convolutional base
Trained convolutional base
Trained convolutional base (frozen)
Input
Input
Input

TL – feature extraction without data augmentation

```
from tensorflow import keras
import numpy as np

def get_features_and_labels(dataset):
  all_features = []
  all_labels = []
  for images, labels in dataset:
    preprocessed_images = keras.applications.vgg16.preprocess_input(images)
    features = conv_base.predict(preprocessed_images)
    all_features.append(features)
    all_labels.append(labels)
  return np.concatenate(all_features), np.concatenate(all_labels)
```

This is the function that we use to compute the output of the feature extraction section for each of the datasets

TL – feature extraction without data augmentation

- Now, we call the `get_features_and_labels` function for each of the 3 datasets

```
train_features, train_labels = get_features_and_labels(train_dataset)
val_features, val_labels = get_features_and_labels(validation_dataset)
test_features, test_labels = get_features_and_labels(test_dataset)
```

Note: Since this operation takes some time, instead of running this code cell, we will load features and labels arrays that we have previously saved... see next slide

TL – feature extraction without data augmentation

- Loading previoulsy saved features and labels arrays:

```
from numpy import load
train_features = load('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_03_train_features.npy')
train_labels = load('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_03_train_labels.npy')
val_features = load('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_03_validation_features.npy')
val_labels = load('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_03_validation_labels.npy')
test_features = load('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_03_test_features.npy')
test_labels = load('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_03_test_labels.npy')
```

TL – feature extraction without data augmentation

```
from tensorflow import keras
from keras import layers

inputs = keras.Input(shape=(4, 4, 512))
x = layers.Flatten()(inputs)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
```

Now, we build a dense network that will play the role of the classification section

- The extracted features are currently of shape (number of samples, 4, 4, 512)

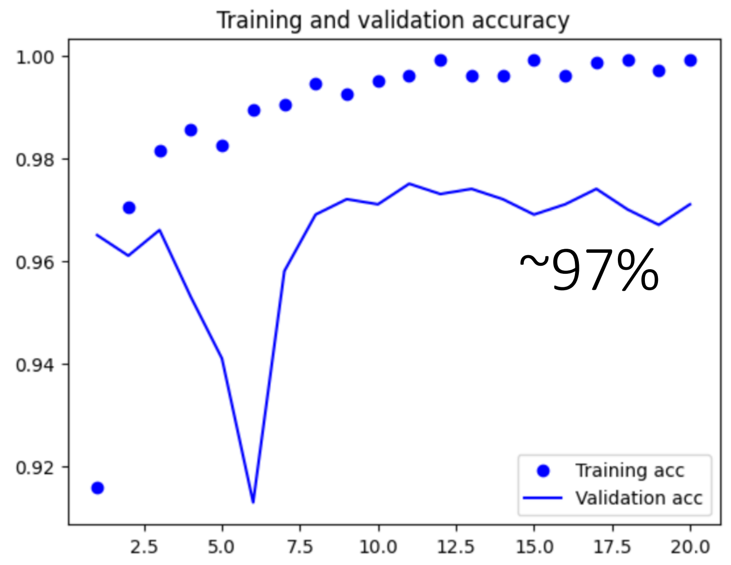
TL – feature extraction without data augmentation

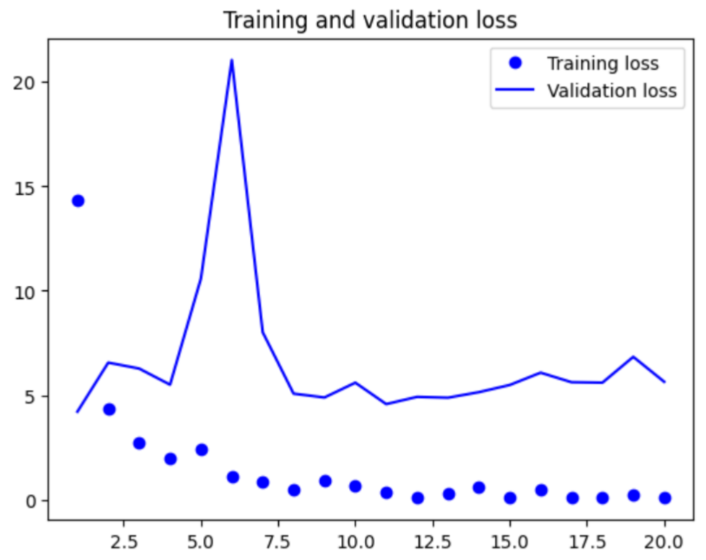
```
model.compile(
  loss='binary_crossentropy',
  optimizer='rmsprop',
  metrics=['accuracy'])

history = model.fit(
  train_features, train_labels,
  epochs=20,
  validation_data=(val_features, val_labels))
```

Now, the train process, which, in this case, doesn’t take long. So, you can run it yourself in the class

Displaying curves of loss and accuracy

```python
import matplotlib.pyplot as plt
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```





TL – feature extraction without data augmentation

- We reach a validation accuracy of about 97%—much better than we achieved with the small model trained from scratch
- But the plots also indicate that the model is overfitting almost from the start—despite using dropout with a fairly large rate
- This happens because this technique doesn’t use data augmentation, which is essential for preventing overfitting with small image datasets

TL – feature extraction without data augmentation – building and saving the model

- You may have noticed that we have two separated models:
  - The feature extraction section, taken from the VGG16 model
  - The classification section, defined and trained by us
- We will now join these two sections in one single model and save it

TL – feature extraction without data augmentation - building the model

- Because models behave just like layers, we can use them as any other layers to build a new model. In this case, we want to use `conv_base` and `model` to build the full model:

```
from keras import models

inputs = keras.Input(shape=(150, 150, 3))
x = keras.applications.vgg16.preprocess_input(inputs)
x = conv_base(x)
outputs = model(x)
full_model = keras.Model(inputs, outputs)
```

Apply input value scaling

TL – feature extraction without data augmentation - evaluating the model

- Before saving the model, we need to compile it:

```
full_model.compile(
        loss="binary_crossentropy",
        optimizer="rmsprop",
        metrics=["accuracy"])
```

TL – feature extraction without data augmentation - Saving and testing the model

- Now, we save the model

```
full_model.save('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_03_TL_
without_data_augmentation.h5')
```

- We can later load it and test it:

```
from tensorflow import keras

loaded_model =
keras.models.load_model('/content/drive/MyDrive/models/04_CNN_02_CatsAn
dDogs_03_TL_without_data_augmentation.h5')

val_loss, val_acc = loaded_model.evaluate(validation_dataset)
print('val_acc:', val_acc)
```

TL – feature extraction with data augmentation

We will now use the “feature extraction” approach with data augmentation

Note: you need a GPU for this

TL – feature extraction with data augmentation

- Create a new notebook named

04_CNN_02_CatsAndDogs_04_TL_with_data_augmentation_and_finetuning.ipynb

- Add the following cell code to mount your Google Drive on Colab:

```
from google.colab import drive
drive.mount('/content/drive')
```

TL – feature extraction without data augmentation

```
import os, shutil
train_dir = '/content/drive/MyDrive/cats_and_dogs_small/train'
validation_dir = '/content/drive/MyDrive/cats_and_dogs_small/validation'
test_dir = '/content/drive/MyDrive/cats_and_dogs_small/test’

from keras.utils import image_dataset_from_directory

IMG_SIZE = 150

train_dataset = image_dataset_from_directory(
  train_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)

validation_dataset = image_dataset_from_directory(
  validation_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)

test_dataset = image_dataset_from_directory(
  test_dir,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=32)
```

Copy this code from the previous notebook

TL – feature extraction with data augmentation

Let us first load the VGG16 model

```
from keras.applications.vgg16 import VGG16
conv_base = VGG16(weights="imagenet", include_top=False)
conv_base.trainable = False
```

Freezing the feature extraction section of the model

- Setting `trainable` to False empties the list of trainable weights of the layer or model (see next slide for more on this)

TL – feature extraction with data augmentation

- It is very important to freeze the convolutional base
- Freezing a layer or set of layers means preventing their weights from being updated during training
- If we don’t do this, the representations that were previously learned by the convolutional base will be modified during training
- Because the Dense layers on top are randomly initialized, very large weight updates would be propagated through the network, effectively destroying the representations previously learned

TL – feature extraction with data augmentation

Creating the model

```
data_augmentation = keras.Sequential(
  [
     layers.RandomFlip("horizontal"),
     layers.RandomRotation(0.1),
     layers.RandomZoom(0.2),
  ]
)
inputs = keras.Input(shape=(150, 150, 3))
x = data_augmentation(inputs)
x = keras.applications.vgg16.preprocess_input(x)
x = conv_base(x)
x = layers.Flatten()(x)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
```

← Apply input value scaling

TL – feature extraction with data augmentation

Compiling and fitting the model

```
model.compile(
    loss="binary_crossentropy",
    optimizer="rmsprop",
    metrics=["accuracy"])

history = model.fit(
    train_dataset,
    epochs=50,
    validation_data=validation_dataset)
```

Since the training process takes several minutes, we will instead load an already trained model, see next slide

```
model.save('/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_04_TL_with_data
_augmentation.h5')
```

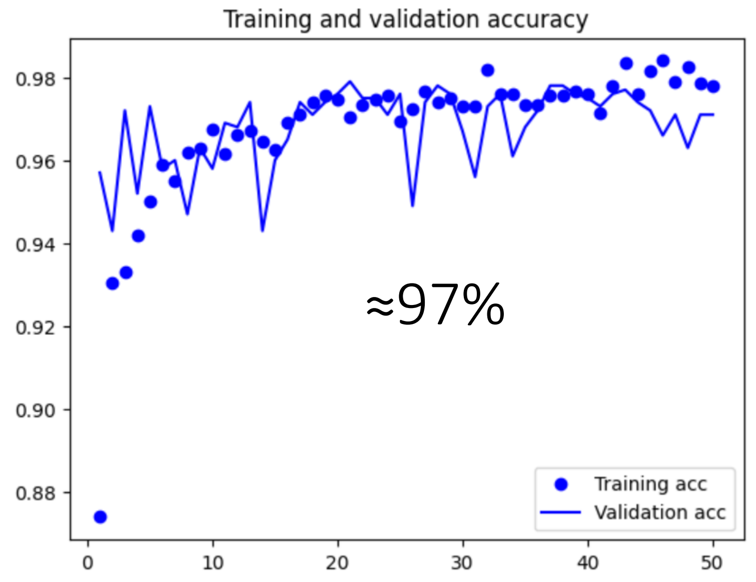
Loading and testing the model

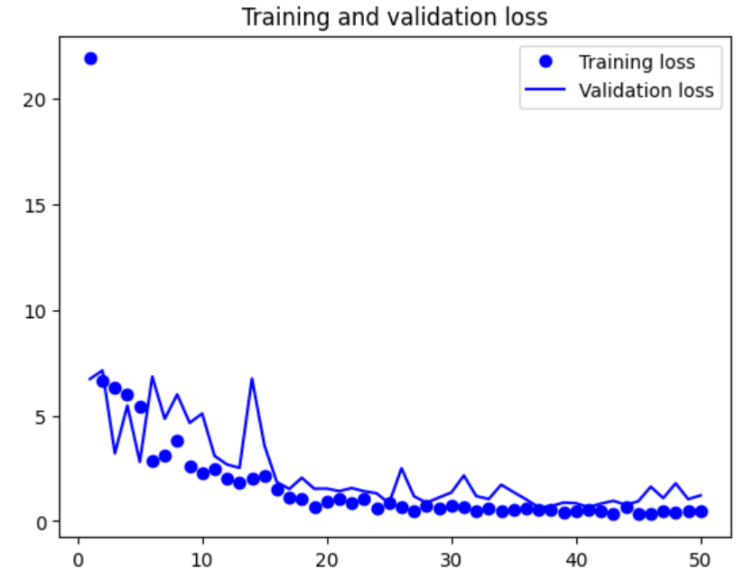
```
from tensorflow import keras
model = keras.models.load_model(
'/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_04_TL_with_data_augmentatio
n.h5')


-----------------------------


val_loss, val_acc = model.evaluate(validation_dataset)
print('val_acc:', val_acc)
```

We would get an accuracy of about 97%

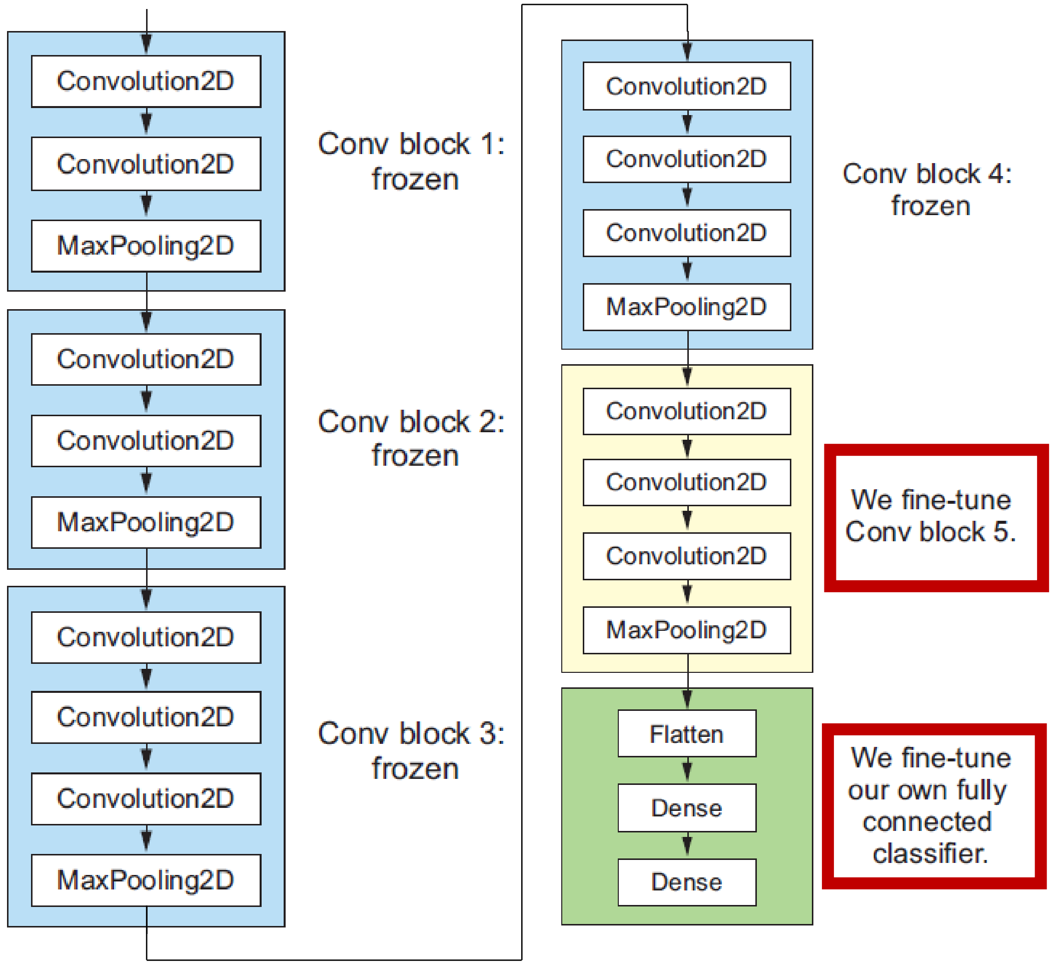
Displaying curves of loss and accuracy

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```





Transfer learning – fine tuning



Example: fine-tuning the last convolutional block of the VGG16 network

Transfer learning – fine tuning

- Why not fine-tune more layers? Why not fine-tune the entire convolutional base? We could. But we need to consider the following:
  - Earlier layers in the convolutional base encode more-generic, reusable features, whereas layers higher up encode more-specialized features
  - So, it’s more useful to fine-tune the more specialized features, because these are the ones that need to be repurposed on our new problem
  - Also, the more parameters we train, the more we are at risk of overfitting

Transfer learning – fine tuning

- Thus, in this situation, it’s a good strategy to fine-tune only the top two or three layers in the convolutional base
- However, it is only possible to fine-tune the top layers of the convolutional base once the classifier on top has already been trained
- If the classifier isn’t already trained, then the error signal propagating through the network during training will be too large, and the representations previously learned by the layers being fine-tuned will be destroyed

Transfer learning – fine tuning

- The steps for fine-tuning a network are as follows:
  1. Add our custom network on top of an already-trained base network
  2. Freeze the base network
  3. Train the part we added
  4. Unfreeze some layers in the base network
  5. Jointly train both these layers and the part we added

Transfer learning – fine tuning

- We have already completed the first three steps when doing feature extraction
- Let’s proceed with step 4: we will unfreeze our `conv_base` and then freeze individual layers inside it
- Freezing all layers except the last 4:

```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
layer.trainable = False
```

Transfer learning – fine tuning

Fine-tuning the model

```
model.compile(
  loss="binary_crossentropy",
  optimizer=keras.optimizers.RMSprop(learning_rate=1e-5),
  metrics=["accuracy"])

history = model.fit(
  train_dataset,
  epochs=30,
  validation_data=validation_dataset)
```

Given that the training process takes several minutes, we will instead load an already trained model, see next slide

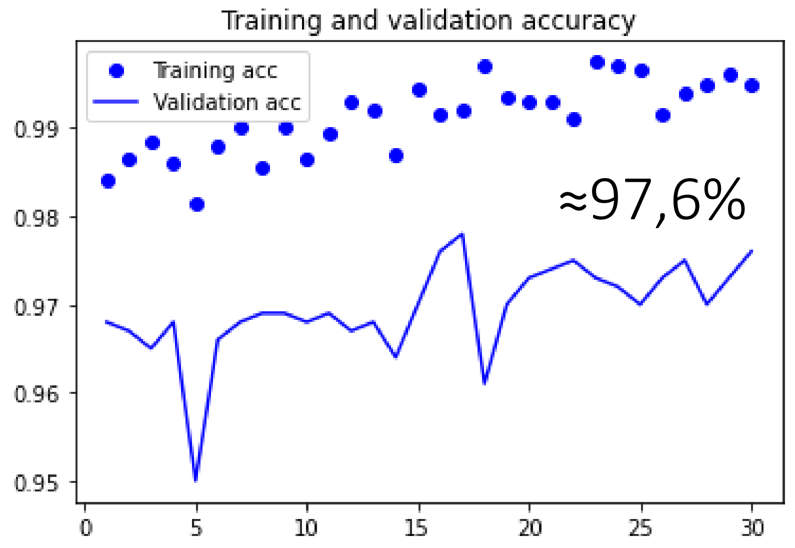
Loading and testing the model

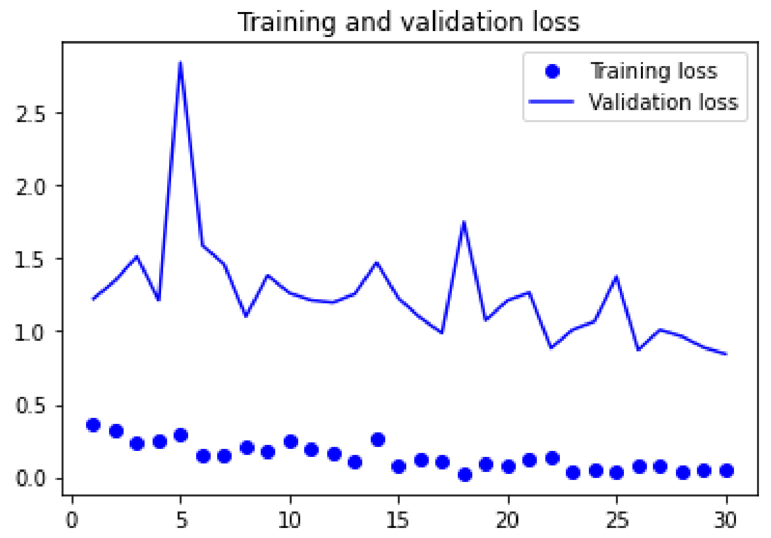
```
from tensorflow import keras
model = keras.models.load_model('/content/drive/MyDrive/models/04_CNN_02_
CatsAndDogs_05_TL_with_data_augmentation_and_fine_tuning.h5')


---------------------------


val_loss, val_acc = model.evaluate(validation_dataset)
print('val_acc:', val_acc)
```

We would get an accuracy of about 97,6%

Displaying curves of loss and accuracy

```python
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```





Saving models during and after training

Saving models during and after training

- So far, in order to train the models in this hands-on, we have used the fit function like this:

```
history = model.fit(
  train_dataset,
  epochs=30,
  validation_data=validation_dataset)
```

- After the training process, the model variable will correspond to the version of the model obtained in the last epoch
- However, the best model can be generated in some epoch other than the last one

Saving models during and after training

- The `tf.keras.callbacks.ModelCheckpoint` callback allows us to continually save the model both *during* and at *the end* of training

- The use of checkpoints allows the user to save the entire model or just the weights of the model at the end of the training process or during it

- This also allows us, for example, to pick-up the training process where we left off in case the training process was interrupted

ModelCheckpoint

```
tf.keras.callbacks.ModelCheckpoint(
    filepath,
    monitor="val_loss",
    verbose=0,
    save_best_only=False,
    save_weights_only=False,
    mode="auto",
    save_freq="epoch",
    options=None,
    initial_value_threshold=None,
    **kwargs
)
```

Class documentation: https://keras.io/api/callbacks/model_checkpoint/

Creating the notebook

- Make a copy of notebook

  04_CNN_02_CatsAndDogs_01_cnn_from_scratch.ipynb,

the first we have created with the cats and dogs dataset (where we built a CNN from scratch without data augmentation)

- Name it 04_CNN_02_CatsAndDogs_05_cnn_from_scratch_checkpoint.ipynb

Saving the best model during training

```
callbacks = [
  keras.callbacks.ModelCheckpoint(
    filepath='/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_05_cnn_from_scratch_CP
best.h5',
    save_best_only=True,
    monitor='val_loss')
]


history = model.fit(
  train_dataset,
  epochs=4,
  validation_data=validation_dataset,
  callbacks=callbacks)
```

- This code defines and uses a callback that saves the model whenever a best model is generated during the training process
- The model is saved using the hdf5 format (also called h5)
- In this case, the loss obtained with the validation set is used as the metric to assess model quality (we could have used accuracy, “val_acc”, instead)

Saving the best model during training

```
callbacks = [
  keras.callbacks.ModelCheckpoint(
    filepath='/content/drive/MyDrive/models/04_CNN_02_CatsAndDogs_05_cnn_from_scratch_CP
best.h5',
    save_best_only=True,
    monitor='val_loss')
]


history = model.fit(
  train_dataset,
  epochs=4,
  validation_data=validation_dataset,
  callbacks=callbacks)
```

- In this case, we are saving the model to the “models” directory in our Google Drive
- Alternatively, we can save it to our Colab machine (just put the name of the file); after the training process, we can load and then save it to our Google drive

Verbose

```
Epoch 1/4
62/63 [============================>.] - ETA: 0s - loss: 0.6927 - acc: 0.5121
Epoch 1: val_loss improved from inf to 0.68860, saving model to models/03_CNN_02_CatsAndDogs_05_cnn_from_scratch_Cpbest.h5
63/63 [==============================] - 10s 121ms/step - loss: 0.6927 - acc: 0.5130 - val_loss: 0.6886 - val_acc: 0.5000
Epoch 2/4
63/63 [==============================] - ETA: 0s - loss: 0.6832 - acc: 0.5505
Epoch 2: val_loss improved from 0.68860 to 0.68489, saving model to models/03_CNN_02_CatsAndDogs_05_cnn_from_scratch_Cpbest.h5
63/63 [==============================] - 6s 90ms/step - loss: 0.6832 - acc: 0.5505 - val_loss: 0.6849 - val_acc: 0.5060
Epoch 3/4
63/63 [==============================] - ETA: 0s - loss: 0.6660 - acc: 0.5995
Epoch 3: val_loss improved from 0.68489 to 0.68285, saving model to models/03_CNN_02_CatsAndDogs_05_cnn_from_scratch_Cpbest.h5
63/63 [==============================] - 7s 111ms/step - loss: 0.6660 - acc: 0.5995 - val_loss: 0.6829 - val_acc: 0.5400
Epoch 4/4
62/63 [============================>.] - ETA: 0s - loss: 0.6434 - acc: 0.6472
Epoch 4: val_loss improved from 0.68285 to 0.64577, saving model to models/03_CNN_02_CatsAndDogs_05_cnn_from_scratch_Cpbest.h5
63/63 [==============================] - 7s 105ms/step - loss: 0.6428 - acc: 0.6480 - val_loss: 0.6458 - val_acc: 0.6350
```

- If we use ***verbose=1*** in the checkpoint model, a message is shown during the training process when the file is updated

Loading the model

```
# Recreates the exact same model, including its weights and the optimizer
loaded_model =
tf.keras.models.load_model('/content/drive/MyDrive/models/04_CNN_02_CatsAn
dDogs_05_cnn_from_scratch_CPbest.h5')

# Show the model architecture
loaded_model.summary()
```

This creates a new model loading the previous trained model "../CPbest.hdf5"

The loaded model includes weights and the state of the optimizer

The state of the optimizer is relevant for continuing the training process

Evaluating the loaded model

```
val_loss, val_acc = loaded_model.evaluate(validation_dataset)
print('val_acc:', val_acc)
```

```
32/32 [==============================] - 2s 45ms/step - loss: 0.6458 - acc: 0.6350
val_acc: 0.6349999904632568
```

Continuing training from a saved point

```
loaded_model.fit(train_dataset, epochs=2, validation_data=validation_dataset)
```

```
Epoch 1/2
63/63 [==============================] - 8s 107ms/step - loss: 0.6171 - acc: 0.6575 - val_loss: 0.6343 - val_acc: 0.6430
Epoch 2/2
63/63 [==============================] - 6s 89ms/step - loss: 0.5914 - acc: 0.6790 - val_loss: 0.6636 - val_acc: 0.6080
<keras.callbacks.History at 0x7efad474ff70>
```

It is possible to continue the training process from a given checkpoint

In this case, the process continues from the previous saved model trained with 4 epochs

2 extra training epochs are performed

Saving model weights (only)

- The use of checkpoints allows users to save the entire model or just the weights of the model at the end of the training process or during it
- Saving only the weights can save disk space
- The weights can be loaded later into models with exactly the same architecture

Defining the callback

```
checkpoint_filepath = '/content/drive/MyDrive/models/cpw/cpsw-{epoch:03d}.ckpt'

model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath=checkpoint_filepath,
        save_weights_only=True,
        verbose=1,
        #monitor="accuracy",
        #mode='max',
        #save_best_only=True
        save_freq="epoch")
```

- The file names depend on the epoch - "cpsw-{epoch:03d}.ckpt"
- Only weights will be saved
- Output messages every time the callback is triggered
- A set of files will be created every epoch

Training the model

```
model.fit(
      train_dataset,
      epochs=4,
      validation_data=validation_dataset,
      callbacks=[model_checkpoint_callback])
```

Training and saving

```
Epoch 1/4
937/938 [============================>.] - ETA: 0s - loss: 0.1753 - accuracy: 0.9460
Epoch 1: saving model to models/cpw/cpsw-001.ckpt
938/938 [==============================] - 51s 54ms/step - loss: 0.1752 - accuracy: 0.9461
Epoch 2/4
937/938 [============================>.] - ETA: 0s - loss: 0.0491 - accuracy: 0.9846
Epoch 2: saving model to models/cpw/cpsw-002.ckpt
938/938 [==============================] - 51s 55ms/step - loss: 0.0491 - accuracy: 0.9846
Epoch 3/4
937/938 [============================>.] - ETA: 0s - loss: 0.0347 - accuracy: 0.9893
Epoch 3: saving model to models/cpw/cpsw-003.ckpt
938/938 [==============================] - 51s 54ms/step - loss: 0.0347 - accuracy: 0.9894
Epoch 4/4
937/938 [============================>.] - ETA: 0s - loss: 0.0262 - accuracy: 0.9919
Epoch 4: saving model to models/cpw/cpsw-004.ckpt
938/938 [==============================] - 51s 54ms/step - loss: 0.0262 - accuracy: 0.9919
<keras.callbacks.History at 0x7f2c646ba610>
```

Files

```
import os
checkpoint_dir = os.path.dirname(checkpoint_filepath)
os.listdir(checkpoint_dir)
```

```
['cpsw-001.ckpt.data-00000-of-00001',
 'cpsw-001.ckpt.index',
 'checkpoint',
 'cpsw-002.ckpt.data-00000-of-00001',
 'cpsw-002.ckpt.index',
 'cpsw-003.ckpt.data-00000-of-00001',
 'cpsw-003.ckpt.index',
 'cpsw-004.ckpt.data-00000-of-00001',
 'cpsw-004.ckpt.index']
```

Each saving action is triggered when an epoch ends

A saving action generates 2 files:
- **data** contains all the weights
- **index** contains the indices of the weights

Our callback includes the epoch number in the file name

Using saved weights

- If we want to load weights that have been saved before and use them in a new notebook/program/session, we need to:
  1. Define a model exactly with the same architecture as the one of the model whose weights we have saved before
  2. Compile the model
  3. Load the weights into the model using function load_weights (see next slide)
  4. Use the model (evaluate, predict, resume the training process)

Using saved weights

```
# We need first to define a model exactly with the same architecture as the one
of the model whose weights we have saved before…

# Then, we need to compile the model…

# Then, we load the weights (for example, the weights of epoch 4)
model.load_weights('/content/drive/MyDrive/models/cpw/cpsw-004.ckpt')

# Then, we use the model (in this case, we evaluate it)
val_loss, val_acc = model.evaluate(validation_dataset)
print('val_acc:', val_acc)
```

Early stopping

- The EarlyStopping callback interrupts training as soon as validation metrics have stopped improving
- It is typically used in combination with ModelCheckpoint, which, as seen in previous slides, lets us continually save the model during training and, optionally, save only the current best model so far

Early stopping

```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="val_loss",
        patience=10),

    keras.callbacks.ModelCheckpoint(
        filepath='models/best_model.h5',
        save_best_only=True,
        monitor='val_loss')
]
```

Interrupts training when validation loss has stopped improving for 10 epochs