Lecture_7_DL_for_CV_leveraging_pretrained_models

November 24, 2022

0.0.1 Applying the preprocessing layers to the dataset

[1]:
```
import matplotlib.pyplot as plt
import pathlib
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import image_dataset_from_directory
from tensorflow import get_logger

get_logger().setLevel('ERROR')    # Suppress info and warnings in logger

base_dir = pathlib.Path("cats_vs_dogs_small")

def plot_helper(x_data, y_data, plot_styles, labels, title, x_label, y_label):
    assert len(x_data) == len(y_data)
    assert len(x_data) == len(plot_styles)
    assert len(x_data) == len(labels)

    fig = plt.figure(figsize=(9 / 2.54, 5 / 2.54), dpi=150)
    plt.rc('axes', titlesize=7)  # fontsize of the axes title
    plt.rc('axes', labelsize=6)  # fontsize of the x and y labels
    plt.rc('xtick', labelsize=6)  # fontsize of the tick labels
    plt.rc('ytick', labelsize=6)  # fontsize of the tick labels
    plt.autoscale(enable=True, axis='x', tight=True)

    for i in range(len(x_data)):
        plt.plot(x_data[i], y_data[i], plot_styles[i], label=labels[i],␣
 ↪linewidth=1, markersize=3)
    plt.title(title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.legend(fontsize=5.5)


def get_datasets(base_dir):
    train_dataset = image_dataset_from_directory(
        base_dir / "train",
        image_size=(180, 180),
```

```
        batch_size=32)
    validation_dataset = image_dataset_from_directory(
        base_dir / "validation",
        image_size=(180, 180),
        batch_size=32)
    test_dataset = image_dataset_from_directory(
        base_dir / "test",
        image_size=(180, 180),
        batch_size=32)
    return train_dataset, validation_dataset, test_dataset
```

There are two ways to use the preprocessing layers when implementing data augmentation: - Make the preprocessing layers part of the model (introduced in the previous lecture) - Apply the preprocessing layers to the dataset.

Let's recall the first approach.

[2]:
```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)
```

[3]:
```
def cats_vs_dogs_model_augm_1():
    inputs = keras.Input(shape=(180, 180, 3))
    x = data_augmentation(inputs)     # Preprocessing layers are part of the␣
 ↪model
    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=256, kernel_size=3, activation="relu")(x)
    x = layers.MaxPooling2D(pool_size=2)(x)
    x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
    x = layers.Flatten()(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs=inputs, outputs=outputs,␣
 ↪name="cats_vs_dogs_augm_1")
    model.compile(loss="binary_crossentropy",
                  optimizer="rmsprop",
                  metrics=["accuracy"])
```

```
    return model
```

There are two important points to be aware of in the first approach:

- Data augmentation will run on-device, synchronously with the rest of your layers, and benefit from GPU acceleration.
- When we export our model using `model.save`, the preprocessing layers will be saved along with the rest of the model. If we later deploy this model, it will automatically standardize images (according to the configuration of the layers). We don't have to reimplement that logic server-side.

When we apply the preprocessing layers to the dataset, we use `Dataset.map` to create a dataset that yields batches of augmented images. In this case:

- Data augmentation will happen asynchronously on the CPU. We can overlap the training of our model on the GPU with data preprocessing using `Dataset.prefetch`.
- The preprocessing layers will not be exported with the model when we call `model.save`. We'll need to attach them to our model before saving it or reimplement them server-side.

To configure the datasets for performance, we can use parallel reads and buffered prefetching to yield batches from disk without I/O become blocking.

Configuring the dataset for performance

Make sure that **buffered prefetching** is used, so that data could be yielded from disk without having I/O become blocking. These are two important methods we should use when loading data:

- `Dataset.cache` keeps the images in memory after they're loaded off disk during the first epoch. This will ensure the dataset does not become a bottleneck while training our model. If the dataset is too large to fit into memory, we can also use this method to create a performant on-disk cache.
- `Dataset.prefetch` overlaps data preprocessing and model execution while training.

Prefetching

A general performance tip is to put all the data processing pipeline on the CPU to make sure that the GPU is only used for training a deep neural network model. When the GPU is working on forward / backward propagation on the current batch, we want the CPU to process the next batch of data so that it is immediately ready. As the most expensive part of the computer, we want the GPU to be fully used all the time during training. We call this **consumer / producer overlap**, where the consumer is the GPU and the producer is the CPU.

Prefetching overlaps the preprocessing and model execution of a training step. The `tensorflow.data` API provides the `Dataset.prefetch` transformation. It can be used to decouple the time when data is produced from the time when data is consumed. In particular, the transformation uses a background thread and an internal buffer to prefetch elements from the input dataset ahead of the time they are requested. The number of elements to prefetch should be equal to (or possibly greater than) the number of batches consumed by a single training step. We could either manually tune this value, or set it to `tf.data.AUTOTUNE`, which will prompt the `tf.data` runtime to tune the value dynamically at runtime.

[4]:
```
from tensorflow import data
AUTOTUNE = data.AUTOTUNE

def prepare(ds, augment=False, cache=False):
    ds = ds.map(lambda x, y: (layers.Rescaling(1./255)(x), y))

    # Keeps the images in memory after they're loaded off disk during the first␣
 ↪epoch
    if cache:
        ds = ds.cache()

    # Use data augmentation only on the training set.
    if augment:
        ds = ds.map(lambda x, y: (data_augmentation(x, training=True), y),
                    num_parallel_calls=AUTOTUNE)

    # Use buffered prefetching on all datasets, which overlaps
    # data preprocessing (CPU) and model execution (GPU) while training
    return ds.prefetch(buffer_size=AUTOTUNE)
```

[5]:
```
train_dataset, validation_dataset, test_dataset = get_datasets(base_dir)
train_dataset = prepare(train_dataset, augment=True, cache=True)
validation_dataset = prepare(validation_dataset)
test_dataset = prepare(test_dataset)
```

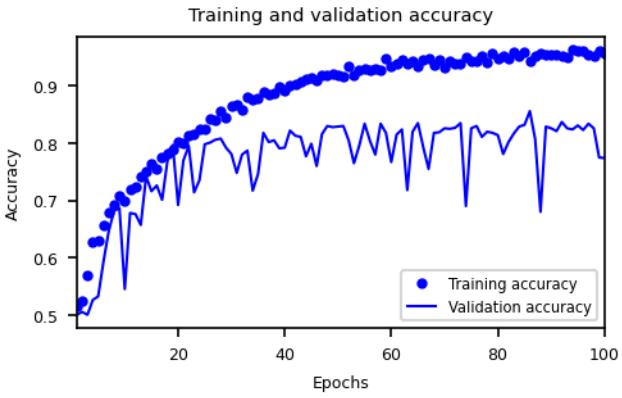
```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 2000 files belonging to 2 classes.
```

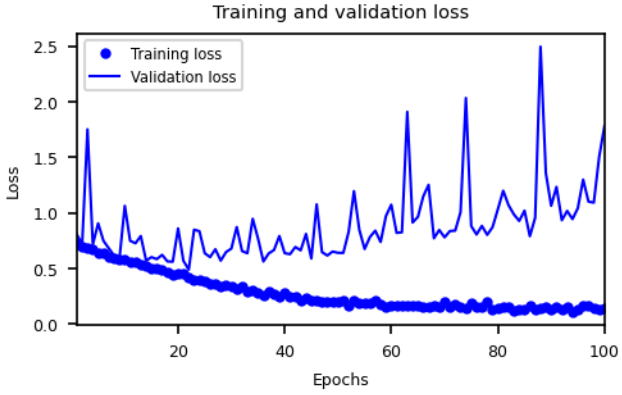
[6]:
```
def cats_vs_dogs_model_augm_2():
    inputs = keras.Input(shape=(180, 180, 3))
    x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
    x = layers.MaxPooling2D(pool_size=2)(x)
    x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
    x = layers.MaxPooling2D(pool_size=2)(x)
    x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
    x = layers.MaxPooling2D(pool_size=2)(x)
    x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
    x = layers.MaxPooling2D(pool_size=2)(x)
    x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
    x = layers.Flatten()(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs=inputs, outputs=outputs,␣
 ↪name="cats_vs_dogs_augm_2")
    model.compile(loss="binary_crossentropy",
                  optimizer="rmsprop",
```

```
                  metrics=["accuracy"])

    return model
```

[7]:
```
model = cats_vs_dogs_model_augm_2()
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="models/convnet_with_data_augmentation_2.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=100,
    validation_data=validation_dataset,
    callbacks=callbacks,
    verbose=0)
# batch_size is not specified if our data is in the form of datasets, since␣
 ↪they generate batches
```

[8]:
```
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)

plot_helper([epochs, epochs],
            [accuracy, val_accuracy],
            ["bo", "b"],
            ["Training accuracy", "Validation accuracy"],
            "Training and validation accuracy",
            "Epochs",
            "Accuracy")
plot_helper([epochs, epochs],
            [loss, val_loss],
            ["bo", "b"],
            ["Training loss", "Validation loss"],
            "Training and validation loss",
            "Epochs",
            "Loss")
```





[9]:
```
test_model = keras.models.load_model("models/convnet_with_data_augmentation_2.
↪keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
63/63 [==============================] - 1s 13ms/step - loss: 0.5100 - accuracy:
0.7805
Test accuracy: 0.780
```

1 Leveraging a pretrained model

A common and highly effective approach to DL on small image datasets is to **use a pretrained model**. A pretrained model is a model that was previously trained on a large dataset, typically on a large-scale image-classification task. If this original dataset is large enough and general enough, the spatial hierarchy of features learned by the pretrained model can effectively act as a generic model of the visual world, and hence, its features can prove useful for many different computer vision problems, even though these new problems may involve completely different classes than those of the original task. For instance, we might train a model on **ImageNet** (where classes are mostly

animals and everyday objects) and then repurpose this trained model for something as remote as identifying furniture items in images. Such **portability of learned features across different problems** is a key advantage of DL compared to many older, shallow learning approaches, and it makes DL very effective for small-data problems.

In this case, let's consider a large convnet trained on the **ImageNet** dataset (1.4 million labeled images and 1,000 different classes). ImageNet contains many animal classes, including different species of cats and dogs, and therefore we can expect it to perform well on our dogs-versus-cats classification problem.

We'll use the **VGG16** architecture, developed by Karen Simonyan and Andrew Zisserman in 2014 ("Very Deep Convolutional Networks for Large-Scale Image Recognition" https://arxiv.org/abs/1409.1556).

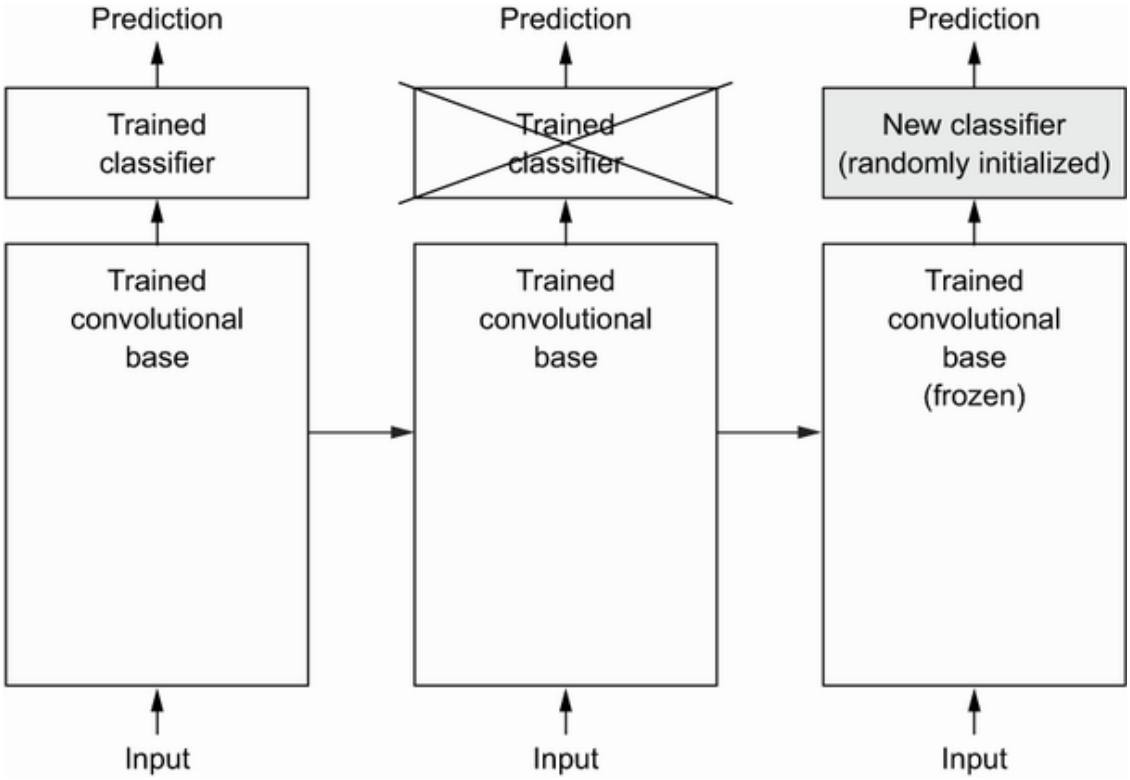
Although it's an older model, far from the current state of the art and somewhat heavier than many other recent models, we chose it because its architecture is similar to what we're already familiar with, and it's easy to understand without introducing any new concepts.

There are two ways to use a pretrained model: **feature extraction** and **fine-tuning**. We'll cover both of them. Let's start with feature extraction.

1.1 Feature extraction with a pretrained model

Feature extraction consists of using the representations learned by a previously trained model to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.

As we saw previously, convnets used for image classification comprise two parts: they start with a series of pooling and convolution layers, and they end with a densely connected classifier. The first part is called the **convolutional base** of the model. In the case of convnets, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output (see figure below).



Why only reuse the convolutional base? Could we reuse the densely connected classifier as well? In general, doing so should be avoided. The reason is that the **representations learned by the convolutional base are likely to be more generic and, therefore, more reusable**: the feature maps of a convnet are presence maps of generic concepts over a picture, which are likely to be useful regardless of the computer vision problem at hand. But the representations learned by the classifier will necessarily be specific to the set of classes on which the model was trained — they will only contain information about the presence probability of this or that class in the entire picture. Additionally, representations found in densely connected layers no longer contain any information about where objects are located in the input image; these layers get rid of the notion of space, whereas the object location is still described by convolutional feature maps. For problems where object location matters, densely connected features are largely useless.

Note that the level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model. Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), whereas layers that are higher up extract more-abstract concepts (such as "cat ear" or "dog eye"). So if our new dataset differs a lot from the dataset on which the original model was trained, we may be better off using only the first few layers of the model to do feature extraction, rather than using the entire convolutional base.

In this case, because the ImageNet class set contains multiple dog and cat classes, it's likely to be beneficial to reuse the information contained in the densely connected layers of the original model. But we'll choose not to, in order to cover the more general case where the class set of the new problem doesn't overlap the class set of the original model. Let's put this into practice by using the convolutional base of the VGG16 network, trained on ImageNet, to extract interesting features from cat and dog images, and then train a dogs-versus-cats classifier on top of these features.

The VGG16 model, among others, comes prepackaged with Keras. We can import it from the `keras.applications` module. Many other image-classification models (all pretrained on the ImageNet dataset) are available as part of `keras.applications`: - Xception - ResNet - MobileNet - EfficientNet - DenseNet

Let's instantiate the VGG16 model.

Instantiating the VGG16 convolutional base

[10]:
```
conv_base = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False,
    input_shape=(180, 180, 3))
```

We pass three arguments to the constructor:

- `weights` specifies the weight checkpoint from which to initialize the model. It can be `None` (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded.
- `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.
- `input_shape` is the shape of the image tensors that we'll feed to the network. This argument is purely optional: if we don't pass it, the network will be able to process inputs of any size. Here we pass it so that we can visualize (in the following summary) how the size of the feature maps shrinks with each new convolution and pooling layer.

Here's the detail of the architecture of the VGG16 convolutional base. It's similar to the simple convnets we're already familiar with:

[11]:
```
conv_base.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 180, 180, 3)]     0

 block1_conv1 (Conv2D)       (None, 180, 180, 64)      1792

 block1_conv2 (Conv2D)       (None, 180, 180, 64)      36928

 block1_pool (MaxPooling2D)  (None, 90, 90, 64)        0

 block2_conv1 (Conv2D)       (None, 90, 90, 128)       73856

 block2_conv2 (Conv2D)       (None, 90, 90, 128)       147584

 block2_pool (MaxPooling2D)  (None, 45, 45, 128)       0
```

```
 block3_conv1 (Conv2D)       (None, 45, 45, 256)       295168

 block3_conv2 (Conv2D)       (None, 45, 45, 256)       590080

 block3_conv3 (Conv2D)       (None, 45, 45, 256)       590080

 block3_pool (MaxPooling2D)  (None, 22, 22, 256)       0

 block4_conv1 (Conv2D)       (None, 22, 22, 512)       1180160

 block4_conv2 (Conv2D)       (None, 22, 22, 512)       2359808

 block4_conv3 (Conv2D)       (None, 22, 22, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 11, 11, 512)       0

 block5_conv1 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_conv2 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_conv3 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 5, 5, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

The final feature map has shape `(5, 5, 512)`. That's the feature map on top of which we'll stick a densely connected classifier.

At this point, there are two ways we could proceed: - Run the convolutional base over our dataset, record its output to a NumPy array on disk, and then use this data as input to a standalone densely connected classifier. This solution is fast and cheap to run, because it only requires running the convolutional base once for every input image, and the convolutional base is by far the most expensive part of the pipeline. But for the same reason, this technique won't allow us to use data augmentation. - Extend the model we have (`conv_base`) by adding `Dense` layers on top, and run the whole thing from end to end on the input data. This will allow us to use data augmentation, because every input image goes through the convolutional base every time it's seen by the model. But for the same reason, this technique is far more expensive than the first.

We'll cover both techniques. Let's walk through the code required to set up the first one: recording the output of `conv_base` on our data and using these outputs as inputs to a new model.

Fast feature extraction without data augmentation We'll start by extracting features as NumPy arrays by calling the `predict()` method of the `conv_base` model on our training, validation, and testing datasets.

Let's iterate over our datasets to extract the VGG16 features.

[12]:
```
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, verbose=0)
        all_features.append(features)
        all_labels.append(labels)
    return np.concatenate(all_features), np.concatenate(all_labels)

train_dataset, validation_dataset, test_dataset = get_datasets(base_dir)
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)
```

```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 2000 files belonging to 2 classes.
```

Importantly, `predict()` only expects images, not labels, but our current dataset yields batches that contain both images and their labels. Moreover, the VGG16 model expects inputs that are preprocessed with the function **`keras.applications.vgg16.preprocess_input`**, which scales pixel values to an appropriate range.

The extracted features are currently of shape `(samples, 5, 5, 512)`:

[13]:
```
train_features.shape
```

[13]:
```
(2000, 5, 5, 512)
```

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.

Defining and training the densely connected classifier

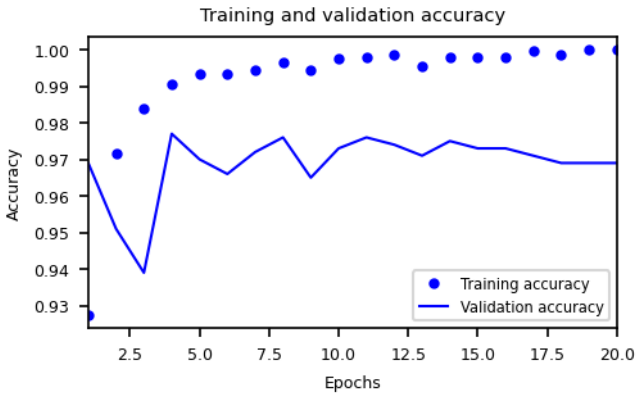
[14]:
```
def feat_extract_model():
    inputs = keras.Input(shape=(5, 5, 512))
    x = layers.Flatten()(inputs)
    x = layers.Dense(256)(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs, outputs)
    model.compile(loss="binary_crossentropy",
                  optimizer="rmsprop",
                  metrics=["accuracy"])
    return model
```

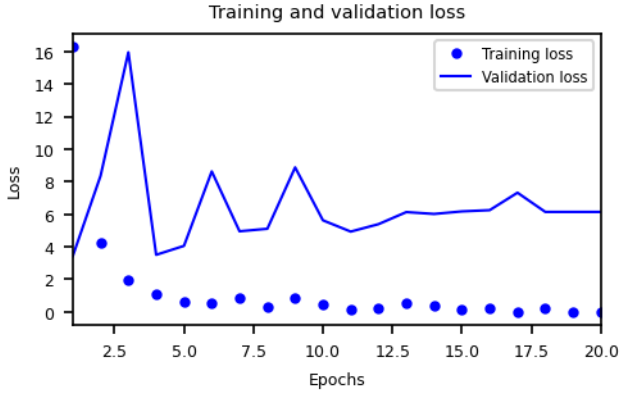
```
callbacks = [
    keras.callbacks.ModelCheckpoint(
      filepath="models/feature_extraction.keras",
      save_best_only=True,
      monitor="val_loss")
]
model = feat_extract_model()
history = model.fit(
    train_features,
    train_labels,
    epochs=20,
    validation_data=(val_features, val_labels),
    callbacks=callbacks,
    verbose=0)
```

Training is very fast because we only have to deal with two `Dense` layers — an epoch takes less than one second even on CPU.

Let's look at the loss and accuracy curves during training.

[15]:
```
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)

plot_helper([epochs, epochs],
            [accuracy, val_accuracy],
            ["bo", "b"],
            ["Training accuracy", "Validation accuracy"],
            "Training and validation accuracy",
            "Epochs",
            "Accuracy")
plot_helper([epochs, epochs],
            [loss, val_loss],
            ["bo", "b"],
            ["Training loss", "Validation loss"],
            "Training and validation loss",
            "Epochs",
            "Loss")
```





[16]:
```
test_model = keras.models.load_model("models/feature_extraction.keras",)
test_loss, test_acc = test_model.evaluate(test_features, test_labels)
print(f"Test accuracy: {test_acc:.3f}")
```

```
63/63 [==============================] - 0s 2ms/step - loss: 4.9850 - accuracy:
0.9670
Test accuracy: 0.967
```

We reach a test accuracy of 96.7% - much better than what we achieved with a small model trained from scratch. This is a bit of an unfair comparison, because ImageNet contains many dog and cat instances, which means that our pretrained model already has the exact knowledge required for the task at hand. This won't always be the case when we use pretrained features.

However, the plots also indicate that we're overfitting almost from the start - despite using dropout with a fairly large rate. That's because this technique doesn't use data augmentation, which is essential for preventing overfitting with small image datasets.

Feature extraction together with data augmentation Now let's review the second technique for feature extraction, which is much slower and more expensive, but which allows us to use data

augmentation during training: creating a model that chains the `conv_base` with a new dense classifier, and training it end to end on the inputs.

In order to do this, we will first **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.

In Keras, we freeze a layer or model by setting its `trainable` attribute to `False`.

Instantiating and freezing the VGG16 convolutional base

[17]:
```
conv_base  = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False)
conv_base.trainable = False
```

Setting `trainable` to `False` empties the list of trainable weights of the layer or model.

Printing the list of trainable weights before and after freezing

[18]:
```
conv_base.trainable = True
print("This is the number of trainable weights "
      "before freezing the conv base:", len(conv_base.trainable_weights))
```

```
This is the number of trainable weights before freezing the conv base: 26
```

[19]:
```
conv_base.trainable = False
print("This is the number of trainable weights "
      "after freezing the conv base:", len(conv_base.trainable_weights))
```

```
This is the number of trainable weights after freezing the conv base: 0
```

Now we can create a new model that chains together 1. A data augmentation stage 2. Our frozen convolutional base 3. A dense classifier

Adding a data augmentation stage and a classifier to the convolutional base

[20]:
```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)


def prepare_pretrain(ds, augment=False, cache=False):
    if cache:
        ds = ds.cache()
    if augment:
```

```
        ds = ds.map(lambda x, y: (data_augmentation(x, training=True), y),
                    num_parallel_calls=AUTOTUNE)
    return ds.prefetch(buffer_size=AUTOTUNE)

train_dataset, validation_dataset, test_dataset = get_datasets(base_dir)
train_dataset = prepare_pretrain(train_dataset, augment=True, cache=True)
validation_dataset = prepare_pretrain(validation_dataset)
test_dataset = prepare_pretrain(test_dataset)

def feat_extract_with_da_model():
    inputs = keras.Input(shape=(180, 180, 3))
    x = keras.applications.vgg16.preprocess_input(inputs)
    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)
    model.compile(loss="binary_crossentropy",
                  optimizer="rmsprop",
                  metrics=["accuracy"])
    return model
```

```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 2000 files belonging to 2 classes.
```

With this setup, only the weights from two `Dense` layers that we added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector). Note that in order for these changes to take effect, we must first compile the model. If weight trainability is modified after compilation, model should be recompiled, or these changes will be ignored.

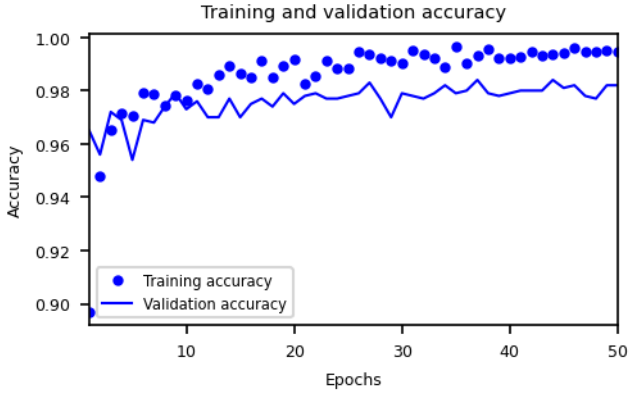
Let's train our model. Because of data augmentation, it will take much longer for the model to start overfitting, so we can train for more epochs, say 50.

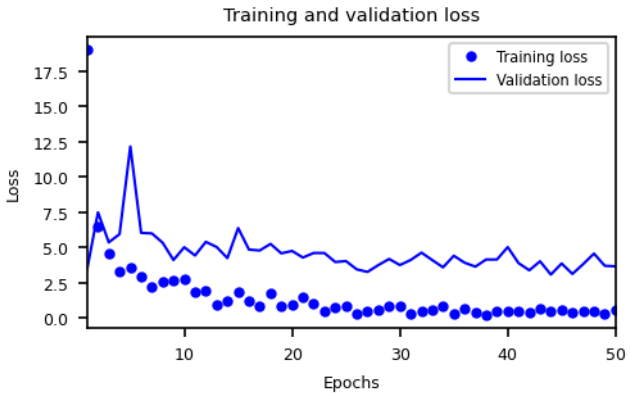
Note that this technique is expensive enough so that it should be run on a GPU, i.e. it's intractable on CPU. If a GPU is not available, then the previous technique is the way to go.

[21]:
```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="models/feature_extraction_with_da.keras",
        save_best_only=True,
        monitor="val_loss")
]
model = feat_extract_with_da_model()
history = model.fit(
    train_dataset,
    epochs=50,
    validation_data=validation_dataset,
```

```
    callbacks=callbacks,
    verbose=0)
```

[22]:
```
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)

plot_helper([epochs, epochs],
            [accuracy, val_accuracy],
            ["bo", "b"],
            ["Training accuracy", "Validation accuracy"],
            "Training and validation accuracy",
            "Epochs",
            "Accuracy")
plot_helper([epochs, epochs],
            [loss, val_loss],
            ["bo", "b"],
            ["Training loss", "Validation loss"],
            "Training and validation loss",
            "Epochs",
            "Loss")
```





Evaluating the model on the test set

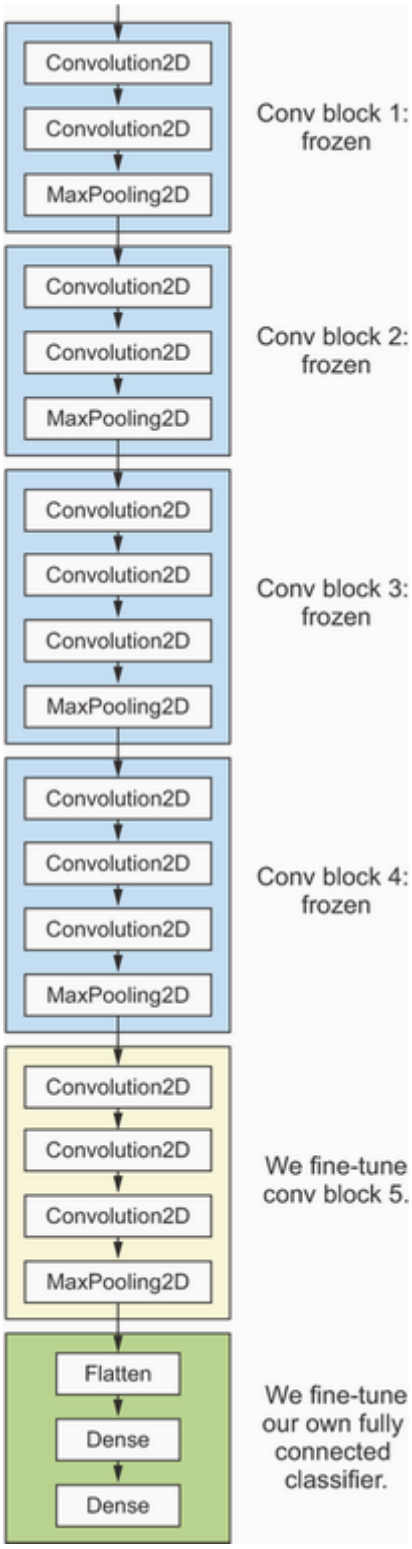
[23]:
```
test_model = keras.models.load_model(
    "models/feature_extraction_with_da.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
63/63 [==============================] - 3s 46ms/step - loss: 5.1397 - accuracy:
0.9785
Test accuracy: 0.979
```

We get a test accuracy of 97.9%. This is similar to the previous test accuracy, which is a bit disappointing given the strong results on the validation data. A model's accuracy always depends on the set of samples you evaluate it on! Some sample sets may be more difficult than others, and strong results on one set won't necessarily fully translate to all other sets.

1.1.1 Fine-tuning a pretrained model

Another widely used technique for model reuse, complementary to feature extraction, is **fine-tuning** (see the figure below). 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.



It's necessary to freeze the convolution base of VGG16 in order to be able to train a randomly initialized classifier on top. For the same reason, it's 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, 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. Thus 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.

We've already completed the first three steps when doing feature extraction. Let's proceed with step 4: we'll unfreeze our `conv_base` and then freeze individual layers inside it.

We'll fine-tune the last three convolutional layers, which means all layers up to `block4_pool` should be frozen, and the layers `block5_conv1`, `block5_conv2`, and `block5_conv3` should be trainable.

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. It's more useful to fine-tune the more specialized features, because these are the ones that need to be repurposed on your new problem. There would be fast-decreasing returns in fine-tuning lower layers. - The more parameters we're training, the more we're at risk of overfitting. The convolutional base has 15 million parameters, so it would be risky to attempt to train it on your small dataset.

Thus, in this situation, it's a good strategy to fine-tune only the top two or three layers in the convolutional base. Let's set this up, starting from where we left off in the previous example.

[24]:
```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
    layer.trainable = False
```

Now we can begin fine-tuning the model. We'll do this with the RMSprop optimizer, using a very low learning rate. The reason for using a low learning rate is that we want to limit the magnitude of the modifications we make to the representations of the three layers we're fine-tuning. Updates that are too large may harm these representations.

[25]:
```
# model = keras.models.load_model("models/feature_extraction_with_da.keras")
model.compile(loss="binary_crossentropy",
              optimizer=keras.optimizers.RMSprop(learning_rate=1e-5),
              metrics=["accuracy"])
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="models/fine_tuning.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks,
    verbose=0)
```

We can finally evaluate this model on the test data:

[26]:
```
model = keras.models.load_model("models/fine_tuning.keras")
test_loss, test_acc = model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
63/63 [==============================] - 3s 46ms/step - loss: 3.9447 - accuracy:
0.9785
Test accuracy: 0.979
```

Here, we get a test accuracy of around 98% (your results may be within one percentage point). In the original Kaggle competition around this dataset, this would have been one of the top results. It's not quite a fair comparison, however, since we used pretrained features that already contained prior knowledge about cats and dogs, which competitors couldn't use at the time.

On the positive side, by leveraging modern DL techniques, we managed to reach this result using only a small fraction of the training data that was available for the competition (about 10%). There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!

1.2 Summary

- Convnets are the best type of machine learning models for computer vision tasks. It's possible to train one from scratch even on a very small dataset, with decent results.
- Convnets work by learning a hierarchy of modular patterns and concepts to represent the visual world.
- On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight overfitting when you're working with image data.
- It's easy to reuse an existing convnet on a new dataset via feature extraction. This is a valuable technique for working with small image datasets.
- As a complement to feature extraction, you can use fine-tuning, which adapts to a new problem some of the representations previously learned by an existing model. This pushes performance a bit further.