

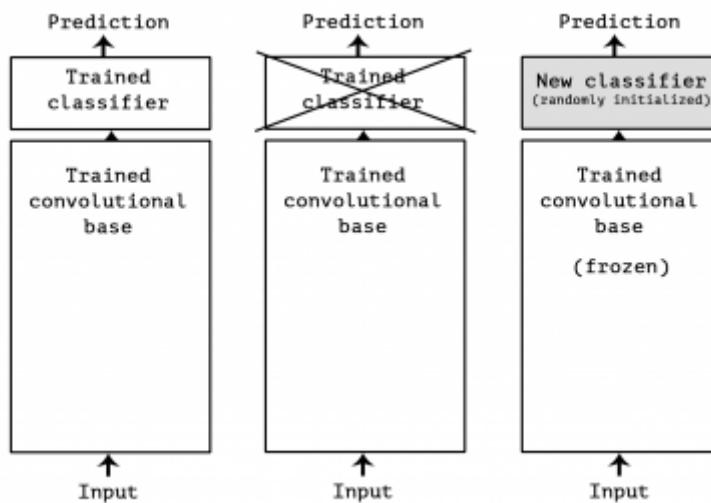
# Pre-trained network (VGG 16)

A common and highly effective approach to deep learning on small image datasets is to leverage a pre-trained network. A pre-trained network is simply a saved network previously trained on a large dataset, typically on a large-scale image classification task. If this original dataset is large enough and general enough, then the spatial feature hierarchy learned by the pre-trained network can effectively act as a generic model of our visual world, and hence its features can prove useful for many different computer vision problems, even though these new problems might involve completely different classes from those of the original task. For instance, one might train a network on ImageNet (where classes are mostly animals and everyday objects) and then re-purpose 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.

During this exercise we will use the VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014, a simple and widely used convnet architecture for ImageNet. Some other pre-trained models are VGG, ResNet, Inception, Inception-ResNet, Xception, DenseNet, MobileNet, EfficientNet. Detailed descriptions can be found on this website: [Pre-trained network architectures](#). Please have a look at the VGG architecture.

## Transfer learning

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” will simply consist 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.



Feature maps of a convnet are presence maps of generic concepts over a picture, which is likely to be useful regardless of the computer vision problem at hand. On the other end, the representations learned by the classifier will necessarily be very specific to the set of classes that the model was trained on – 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 would be 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), while layers higher-up extract more abstract concepts (such as “cat ear” or “dog eye”). So if your new dataset differs a lot from the dataset that the original model was trained on, you 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 our case, since the ImageNet class set did contain multiple dog and cat classes, it is likely that it would be beneficial to reuse the information contained in the densely-connected layers of the original model. However, we will choose not to, in order to cover the more general case where the class set of the new problem does not overlap with the class set of the original model. Let's put this in practice by using the convolutional base of the VGG16 network, trained on ImageNet, to extract interesting features from our cat and dog images, and then training a cat vs. dog classifier on top of these features.

The VGG16 model, among others, comes pre-packaged with Keras. You can import it from the `keras.applications` module. Here's the list of image classification models (all pre-trained on the ImageNet dataset) that are available as part of `keras.applications`:

- Xception
- InceptionV3
- ResNet50
- VGG16
- VGG19
- DenseNet
- MobileNet
- EfficientNet

## VGG-16 model

Import the model from Keras:

```
from tensorflow.keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

We passed three arguments to the constructor:

- `weights`, to specify which weight checkpoint to initialize the model from
- `include_top`, which refers to including or not the densely-connected classifier on top of the network. By default, this densely-connected classifier would correspond to the 1000 classes from ImageNet. Since 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`, the shape of the image tensors that we will feed to the network. This argument is purely optional: if we don't pass it, then the network will be able to process inputs of any size.

**Ex. 1.1** Please analyze the model of VGG-16 (model summary) - after compilation.

**Ex. 1.2** What is the final feature map shape ?

At this point, there are two ways we could proceed to leverage a pre-trained network: **feature extraction** and **fine-tuning**:

- Feature Extraction: a new classifier is trained from scratch on top of the pre-trained model.
- Fine-Tuning: unfreezing a few of the top layers of a frozen model base and training both the newly-added classifier layers and the last layers of the base model.

## Basic feature extraction

Let's start with feature extraction - recording the output of `conv_base` on our data and using these outputs as inputs to a new model. We will start by simply running instances of the previously-introduced `ImageDataGenerator` (lab. 3 and 4) to extract images as Numpy arrays as well as their labels. We will extract features from these images simply by calling the `predict` method of the `conv_base` model. Import the required module, create the required directory, and preliminarily set up the generator.

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

base_dir = '/cats_and_dogs_small'

train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20
```

Now we construct a feature extraction function. The size of the first numpy array is: sample size \* 4 \* 4 \* 512, corresponding to the size of the network output, then the label is naturally only a one-dimensional array of sample size. Later, generators are used to package data in batches for easy processing. The generator determines the size of the picture, and then obtains the label of the picture and stores it in the array. The label here is the name of the folder. In this example, there are two folders, so 0 and 1. The processed image is in the form of a data stream, which is equivalent to 20 pictures per batch forming a data stream, and the label is the root.

```
def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count,))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
```

```

    class_mode='binary')
i = 0
for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
    features[i * batch_size : (i + 1) * batch_size] = features_batch
    labels[i * batch_size : (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size >= sample_count:
        # Note that since generators yield data indefinitely in a loop,
        # we must `break` after every image has been seen once.
        break
return features, labels

```

In the later for loop, the processed data stream is input directly into the existing model and output the result. Then the result is stored in the feature array which is defined at the beginning, and the labels are stored in the labels array. In order to avoid infinite loops, a number of restrictions are added to ensure that all data is used without duplication.

Here we create the training set, test set and validation set. Used this method once and keep their labels.

```

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir,
1000)
test_features, test_labels = extract_features(test_dir, 1000)

```

The extracted features are currently of shape (samples, 4, 4, 512). We will feed them to a densely-connected classifier.

**Ex. 1.3** Define our densely-connected classifier. First, flatten the features to (samples, 8192). You can do it using reshape function or by adding a Flatten layer before the first Dense layer. Then use two densely-connected layers with the dropout for regularisation. The first layer will have 256 channels and a ReLU activation function. Dropout set to 0.5. The classification layer has a Sigmoid activation function.

The model summary should look like this:

Layer ( <b>type</b> )	Output Shape	Param #
dense_55 (Dense)	(None, 256)	2097408
dropout (Dropout)	(None, 256)	0
dense_56 (Dense)	(None, 1)	257
<hr/>		
Total params: 2,097,665		
Trainable params: 2,097,665		
Non-trainable params: 0		

**Ex. 1.4** Train the model using the following parameters:

- RMSprop( $\text{lr}=2\text{e}-5$ ) optimizer,
- Binary crossentropy loss function,
- Accuracy metric.

**Ex. 1.5** Fit the model with 30 epochs, batch size 20, and validation data. Training is very fast since we only have to deal with two Dense layers.

Expected outcome:

```
Epoch 30/30
2000/2000 [=====] - 0s - loss: 0.0968 - acc: 0.9680
- val_loss: 0.2428 - val_acc: 0.9030
```

**Ex. 1.6** Plot the loss and accuracy curves during training.

We reach a validation accuracy of about 90%, much better than what we could achieve in the previous section with our small model trained from scratch. However, the overfitting problem is still there...

## Pre-trained part as an extended layer

The second technique is much slower and more expensive, but allows us to leverage data augmentation during training: extending the `conv_base` model and running it end-to-end on the inputs. Note that this technique is in fact so expensive that you should only attempt it if you have access to a GPU: it is absolutely intractable on CPU. **If you cannot run your code on GPU, then the previous technique is the way to go.**

Because models behave just like layers, you can add a model (like our `conv_base`) to a Sequential model just like you would add a layer. So you can do the following:

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

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

**Ex. 2.1** Please analyse the model summary. How many parameters do you have now ? How many parameters are trainable?

Before we compile and train our model, a very important thing to do is to freeze the convolutional base. “Freezing” a layer or set of layers means preventing their weights from getting updated during training. If we don't do this, then the representations that were previously learned by the convolutional base would get modified during training. Since 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, freezing a network is done by setting its `trainable` attribute to `False`:

```
print('This is the number of trainable weights '
      'before freezing the conv base:', len(model.trainable_weights))
```

This is the number of trainable weights before freezing the conv base: 30

```
conv_base.trainable = False

print('This is the number of trainable weights '
      'after freezing the conv base:', len(model.trainable_weights))
```

This is the number of trainable weights after freezing the conv base: 4

With this setup, only the weights from the 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 you ever modify weight trainability after compilation, you should then re-compile the model, or these changes would be ignored.

**Ex. 2.2** Implement and fit the model like in the previous labs (with data augmentation). If the process takes too much time, please instead read saved model from the following repository:[Model](#). Check the summary.

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    # This is the target directory
    train_dir,
    # All images will be resized to 150x150
    target_size=(150, 150),
    batch_size=20,
    # Since we use binary_crossentropy loss, we need binary labels
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
```

```

batch_size=20,
class_mode='binary')

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])

history = model.fit(
    train_generator,
    epochs=30,
    validation_data=validation_generator,
    verbose=2)

```

**Ex. 2.3** Plot the results.

## Fine-tuning

Another widely used technique for model reuse, complementary to feature extraction, is 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 our 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.

Please have a look at the VGG-16 architecture [|VGG-16](#)

We have stated before that it was 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 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 wasn't already trained, then the error signal propagating through the network during training would be too large, and the representations previously learned by the layers being fine-tuned would be destroyed. Thus the steps for fine-tuning a network are as follow:

- 1) Add your custom network on top of an already trained base network.
- 2) Freeze the base network.
- 3) Train the part you added.
- 4) Unfreeze some layers in the base network.
- 5) Jointly train both these layers and the part you added.

We have already completed the first 3 steps when doing feature extraction. Let's proceed with the 4th step: we will unfreeze our conv\_base, and then freeze individual layers inside of it.

**Ex. 3.1** Analyse the conv\_base.summary().

We will fine-tune the last 3 convolutional layers, which means that all layers up until 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. However, we need to consider that:

- Earlier layers in the convolutional base encode more generic, reusable features, while layers higher up encode more specialized features. It is more useful to fine-tune the more specialized features, as these are the ones that need to be repurposed on our new problem. There would be fast-decreasing returns in fine-tuning lower layers.
- The more parameters we are training, the more we are at risk of overfitting. The convolutional base has 15M parameters, so it would be risky to attempt to train it on our small dataset.

Thus, in our situation, it is a good strategy to only fine-tune the top 2 to 3 layers in the convolutional base.

Let's set this up, starting from where we left off in the previous example:

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

Now we can start fine-tuning our network. We will 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 3 layers that we are fine-tuning. Updates that are too large may harm these representations.

Now let's proceed with fine-tuning (like before read the model weights from [model h5](#)):

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit(
    train_generator,
    epochs=100,
    validation_data=validation_generator)
```

**Ex. 3.2** Plot your results using the same plotting code as before. Smooth the curves/plots using following code:

```
def smooth_curve(points, factor=0.8):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
```

```
return smoothed_points

plt.plot(epochs,
         smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs,
         smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs,
         smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs,
         smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

**Ex. 3.3** Evaluate this model on the test data:

```
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')

test_loss, test_acc = model.evaluate(test_generator)
print('test acc:', test_acc)
```

test acc: 0.96799999237

Here we got a test accuracy of 97%. In the original Kaggle competition around this dataset, this would have been one of the top results. However, using modern deep learning techniques, we managed to reach this result using only a very small fraction of the training data available (about 10%). There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!

From:  
<https://mdig.agh.edu.pl/dokuwiki/> - MVG Group



Permanent link:  
[https://mdig.agh.edu.pl/dokuwiki/doku.php?id=teaching:data\\_science:dnn:lab3](https://mdig.agh.edu.pl/dokuwiki/doku.php?id=teaching:data_science:dnn:lab3)

Last update: **2025/11/19 18:07**