Modern convnet architecture patterns

2021.7.1

A model's architecture is the sum of the choices that went into creating it: which layers to use, how to configure them, in what arrangement to connect them. These choices define the **hypothesis space** of model: the space of possible functions that gradient descent can search over, parameterized by the model's weights. A good hypothesis space encodes *prior knowledge* that we have about the problem. For instance, using convolution layers means that we know in advance that the relevant patterns present in input images are translation-invariant. A good model archetecture is one that **reduce the size of the search space** or otherwise, **makes it easier to converge to a good point of the search space**.

Gradient descent is a pretty stupid search process, so it needs all the help it can get. Model architecture is more an art than a science. The keyword here is **intuitively**.

In the following sections, we'll review a few essential convnet architecture best practices, in particular **residual connection**, **batch normalization**, and **separable convolutions**. We will also demonstrate how to apply them on our cat vs dogs classification problem.

9.3.1 Modularity, hierarchy, and reuse

A universal recipe to make complex system simpler: structure amorphous soup of complexity into **modules**, organize the modules into a **hierarchy**, and start **reusing** the same modules in multiple places as appropriate.

Deep learning model architecture is primarily about making a clever use of modularity, hierarchy, and reuse. You'll notice that all popular convnet architectures are not only structured into layers, they're structured into repeated groups of layers. For instance, VGG16 is structured into repeated "conv-conv-max pooling" blocks.

Most convnets often feature pyramid-like structures (feature heirarchies).

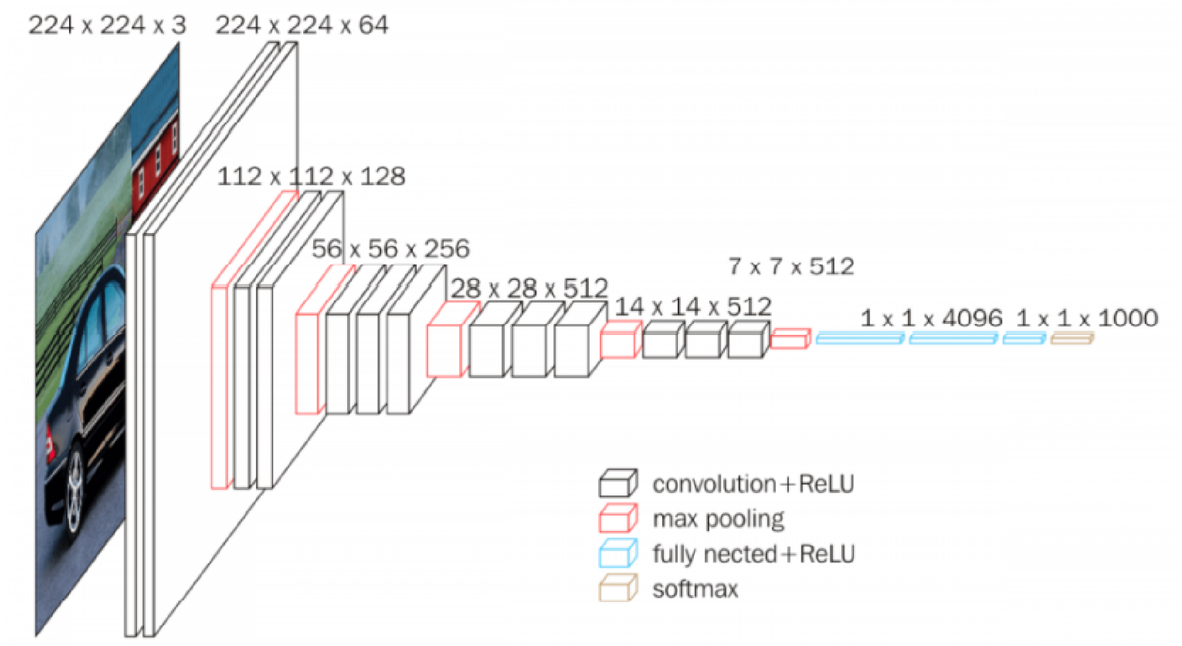


Figure 9.8 The VGG16 architecture: note the repeated layer blocks and the pyramid-like structure of the feature maps.

In general, a deep stack of narrow layers performs better than a shallow stack of large layers. However, there's a limit to how deep we can stack layers: the problem of **vanishing gradients**. This leads to first essential model architecture pattern: residual connections.

9.3.2 Residual connections

Backpropagation in a sequential deep learning model is pretty similar to the game of Telephone. We've got a chain of functions, like: $y = f_4(f_3(f_2(f_1(x))))$. It motivates us if your function chain is too deep, this noise starts overwhelming gradient information, and backpropagation stops working. This is called the **vanishing gradients** problem.

The fix is simple: forcing each function in the chain to be non-destructive - to retain a noiseless version of the information contained in the previous input. The easiest way to implement this is called a *residual connection*. Just add the input of a layer or block of layers back to its output. The residual connection acts as an information shortcut around destructive or noisy blocks.

A residual connection in pseudocode

```
# some input tensor
x = ...
# save a pointer to original input. Here it's called 'residual'
residual = x
# A computational block can potentially be destructive or noisy
x = block(x)
# Add the original input to the layer's output: the final output will thus always
    preserve full info about the original input
x = add([x, residual])
```

We'd typically use *padding* = *same* in the convolution layers in target block so as to avoid spatial downsampling due to padding, and you'd use strides in the residual projection to match any downsampling caused by a max pooling layer.

Case where the target block changes the number of output filters

```
from tensorflow import keras
from tensroflow.keras import layers

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation='relu')(inputs)
residual = x

x = layer.Conv2D(64, 3, activation='relu', padding='same')(x)
# the residual only had 32 filters, so using 1*1 Conv2D to project it to the
  correct shape
residual = layers.Conv2D(64, 1)(residual)
x = layers.add(x, residual)
```

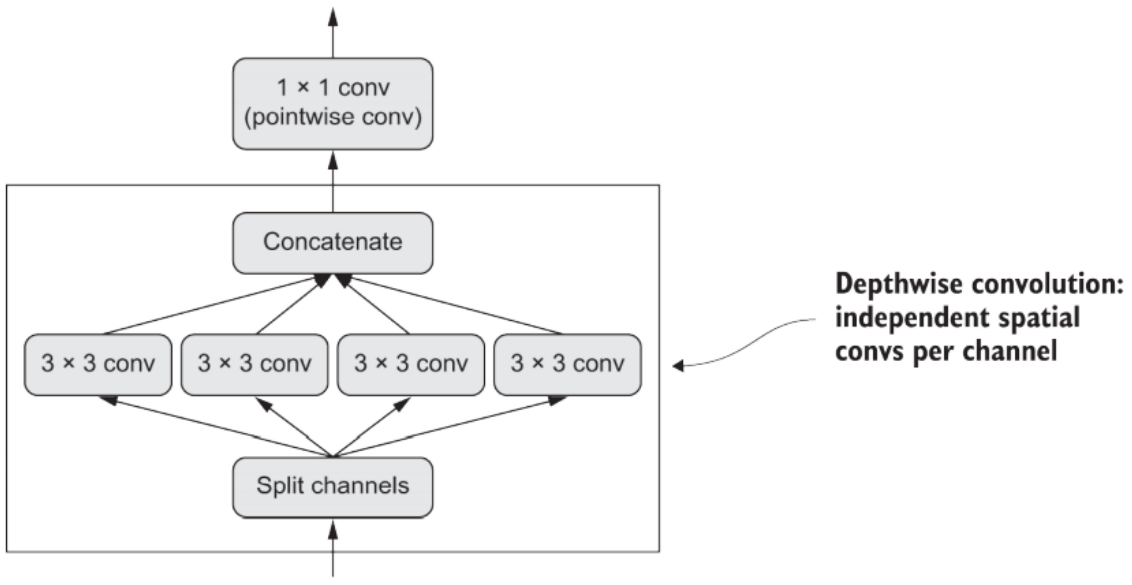
P286 have the solution to handle the target blocks including a max pooling layer.
With residual connections, we can build networks of arbitary depth, without having to worry about vanishing gradients.

9.3.3 Batch normalization

Normalization is a broad category of methods that seek to make different samples seen by a machine-learning model more similar to each other, which helps the model learn and generalize well to new data.

Batch normalization can adaptively normalize data even as the mean and variance change over time during training. During training, it uses the mean and variance of the current batch of data to normalize samples, and during inference, it uses an exponential moving average of the batch-wise mean and variance of the data seen during training. You'll find that this is true of many things in deep learning - **deep learning is not an exact science, but a set of ever-changing, empirically-derived engineering best practices, woven together by unreliable narratives.**

9.3.4 Depthwise separable convolutions


1 × 1 conv
(pointwise conv)
Concatenate
3 × 3 conv
3 × 3 conv
3 × 3 conv
3 × 3 conv
Split channels
Depthwise convolution: independent spatial convs per channel