**Binary and multi-class image classification**

We will learn about object detection models which identify objects in images by drawing a box around them.We will also apply image segmentation models to segment images into meaningful areas.

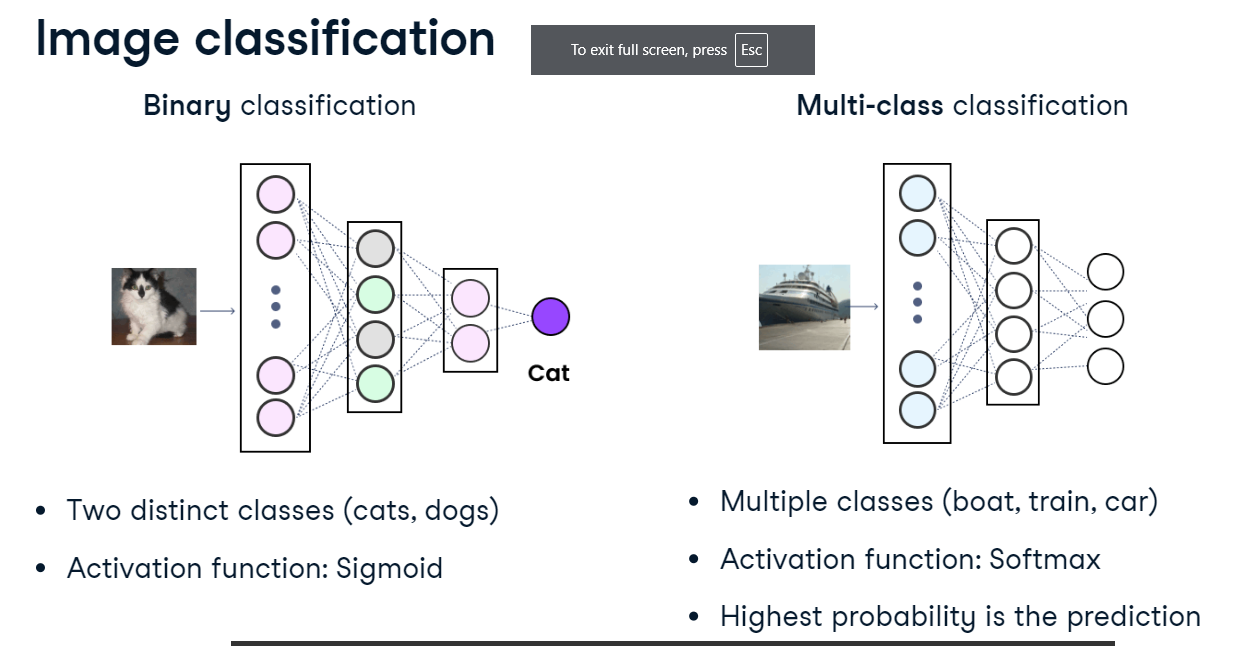
Finally, we will create new images based on learned patterns using image generation models.

Before starting, you should already be familiar with Convolutional Neural networks, including how they work and how to construct them in PyTorch, as well as with PyTorch model training in general, as taught in this prerequisite course.

We will use TorchVision throughout this course. It is a PyTorch image library that provides useful tools, including transformations for image pre-processing,pre-trained CNN models, and labeled image datasets for training and testing.

**Image classification**

Let's begin with image classification, commonly categorized into two types. The first type is a binary classification with two distinct classes, for example, cats and dogs. We use the sigmoid activation function to produce get the probability of either class. The second type is multi-class classification. Here, we deal with more than two classes, for example, boat, train, and car. We use the softmax activation to get the probability of each class. The class with the highest probability is the final prediction.



**Convolutional Neural Network model**

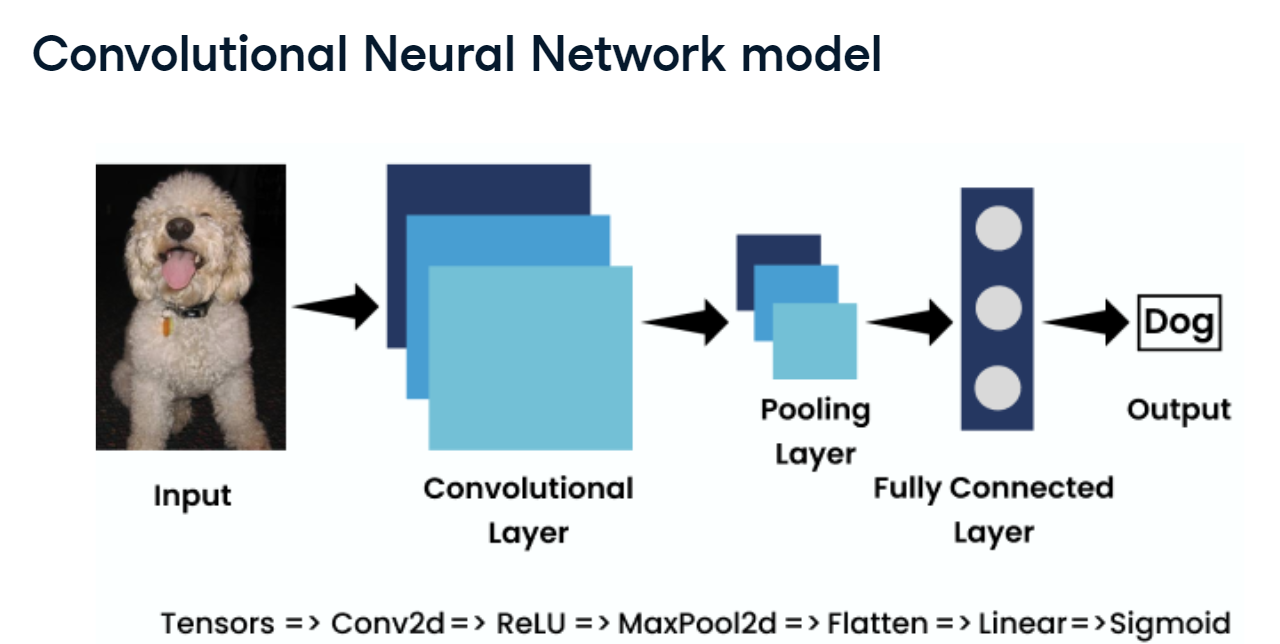
Let's now revisit the CNN model. First, we load a dataset (for example, pet images) and transform it into tensors.

We pass tensors through the convolutional layer, where the network learns image features and generates feature maps. Then, we apply a non-linear activation function, for example, ReLU.

In the pooling layer, we reduce the size of feature maps to decrease the computational workload.

Then, we flatten multi-dimensional tensors into a one-dimensional vector and pass it into the fully connected layer.

Finally, we apply the Sigmoid or Softmax activation function to generate class probabilities.



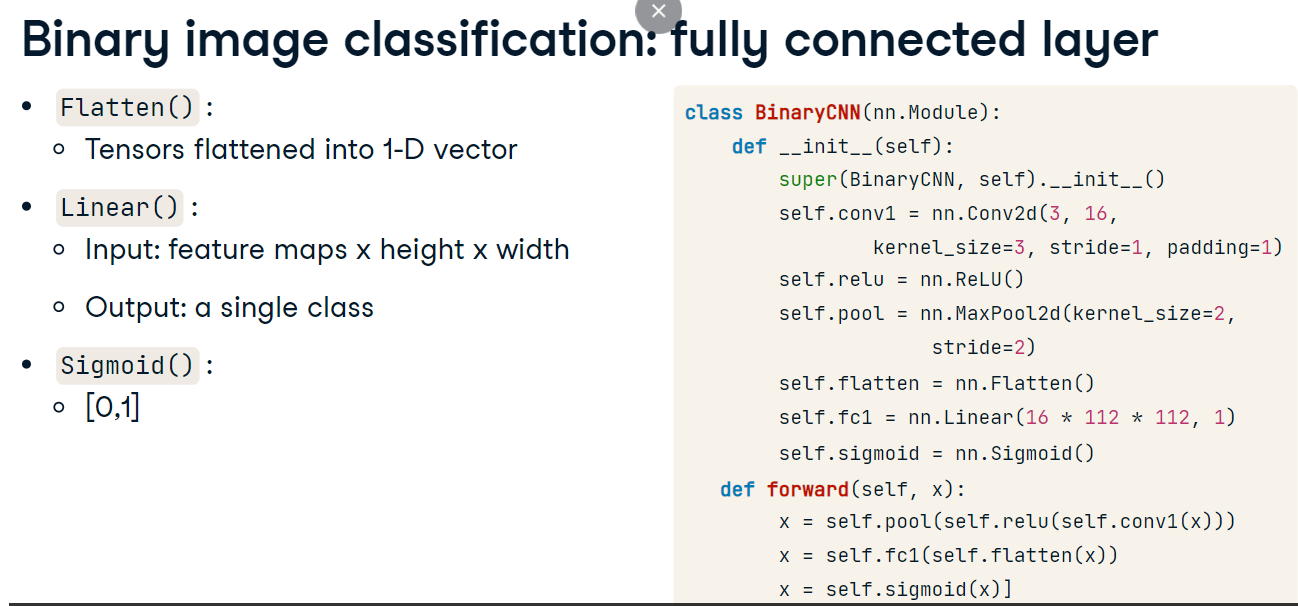
**Datasets: class labels**

Suppose we have pets dataset with separate directories for each class. This is a common format in image classification. We import the datasets and transform modules from torchvision. The training data directory is located in the data train subfolder. To load our dataset into PyTorch we use the ImageFolder class, passing it two arguments: root is the data path, and transform is the transformation to apply to the upon loading, here: conversion to tensors. We assign the dataset to train\_dataset. Now, we can access the class labels from the train dataset using dot-classes. We have two labels, cat and dog. The class to idx attribute maps class labels and their indices. Cat is zero and dog is one.

**Binary image classification: convolutional layer**

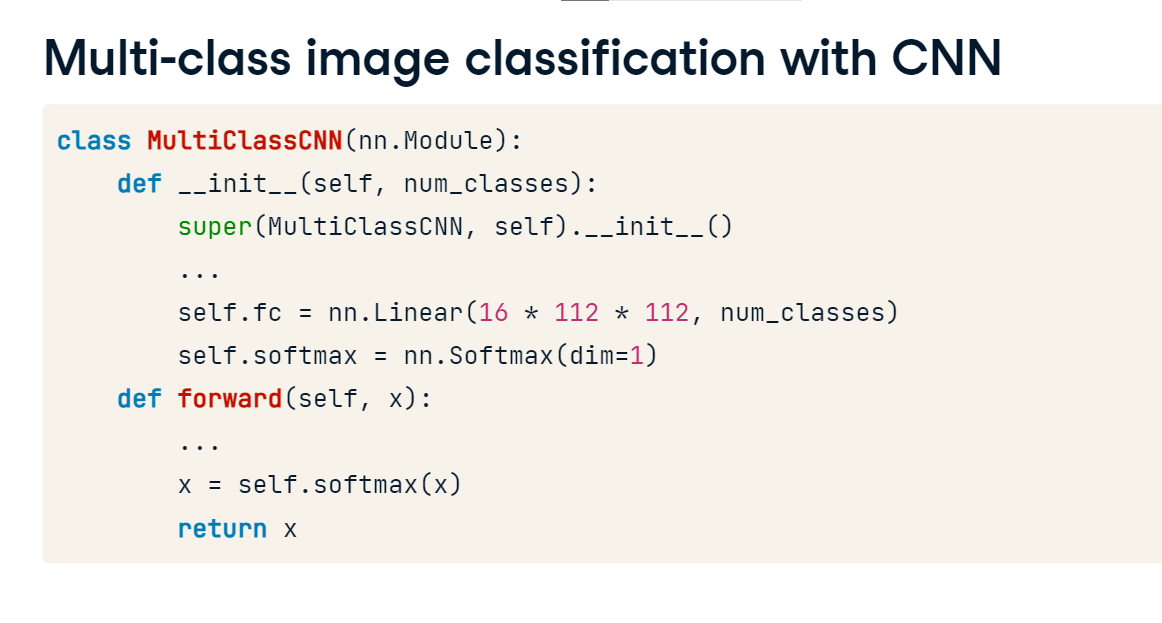
Let's build the binary CNN model. The Conv2d layer has three input RGB channels for red, green, and blue, sixteen output channels, and a three-by-three kernel that moves one stride, or step, at a time. One-pixel padding is added around the image border. We also define the ReLU activation function and the MaxPool2d layer with a two-by-two kernel size and stride of two.

**Binary image classification: fully connected layer**

The flatten layer reshapes tensors into a one-dimensional vector. This vector is passed to the linear layer with input features equal to the number of feature maps times their height and width. The output is a just one value, which we pass to a sigmoid activation. Finally, in the forward method, we pass the input through subsequent layers and return the output.

**Multi-class image classification with CNN**

For the multi-class model, we adjust the final layer output by specifying the number of classes. We also modify the activation function to softmax. We use dim equal one as this dimension stores classes.

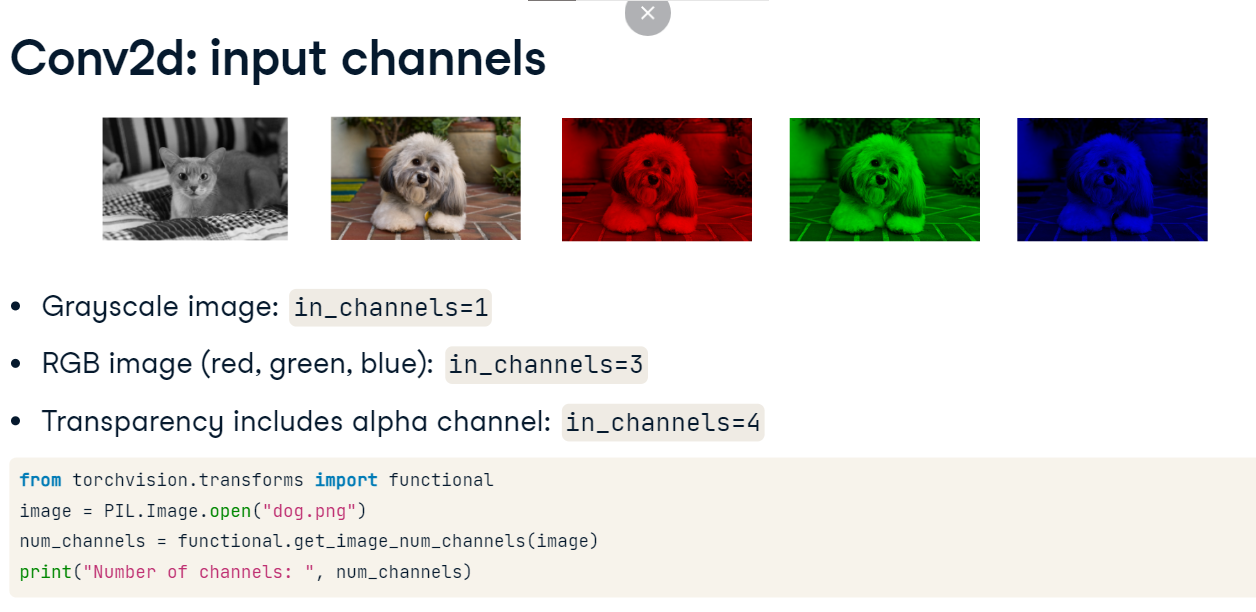


**Convolutional layers for images**

These layers play a key role in our models for object detection and image segmentation. In this video, we’ll review what we know about convolutional layers and apply that to image data. We’ll also see how to access, add, and create blocks with these layers, which are all tools that can be used to adapt an existing model to a specific task. Let’s begin by reviewing their structure and how they work.

**Conv2d: input channels**

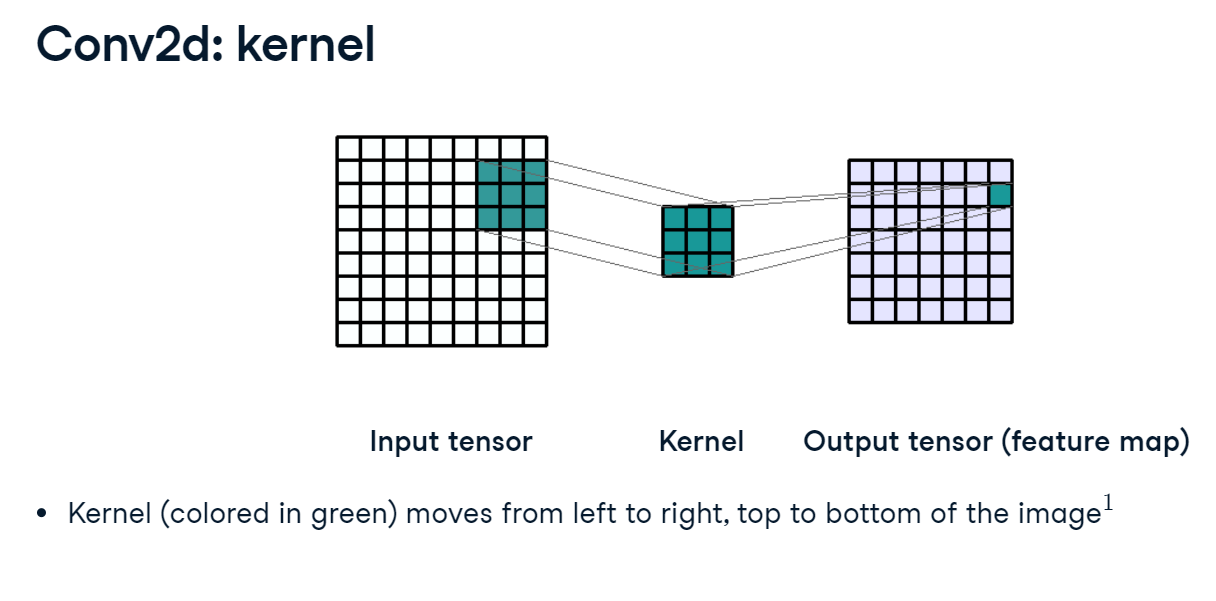
The first parameter in the conv2d layer is the input channels. These channels refer to the image color channels. Grayscale images have one channel, like the cat image on the left. RGB images, like the colored dog image, have three channels: red, green, and blue. Images with transparency will have four channels, due to an additional alpha channel. We can check the number of channels an image has with the functional module from torchvision-dot-transforms. We load an image using the image-dot-open method from the Python library PIL and apply the functional-dot-get image num channels method to the loaded image. The output shows that we have an RGB image with three channels. Knowing this will help us design the right model!



**Conv2d: kernel**

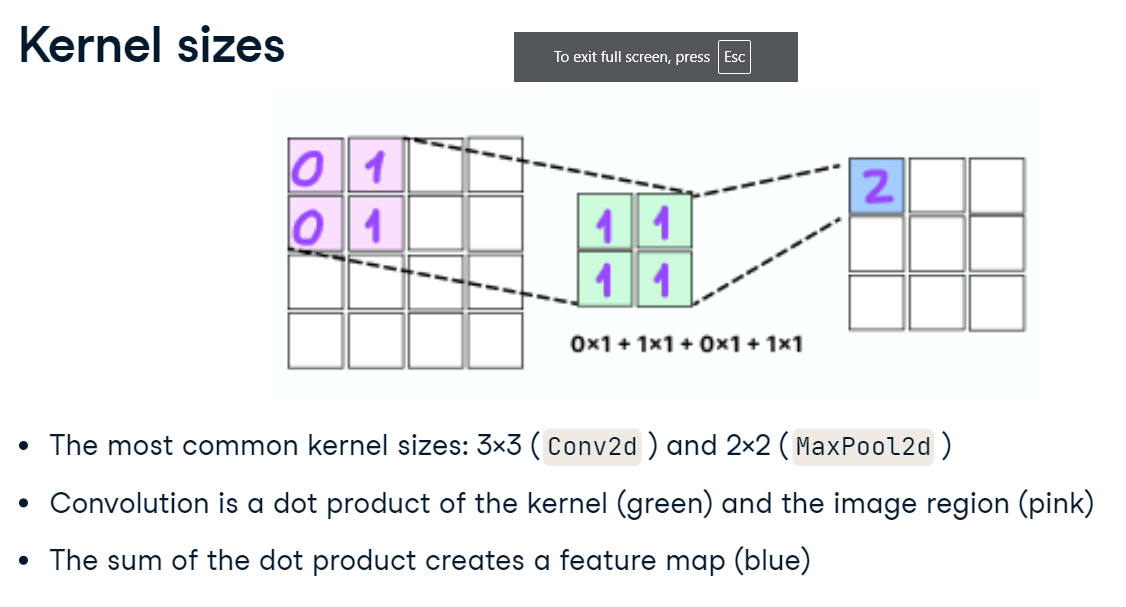
Next we’ll explore the kernel. The convolutional layer learns image patterns by applying a small kernel (shown in green) to the input tensors, or channels, and creating an output tensor with learned features. In the forward pass, this kernel slides from left to right and top to bottom.

1. 1 Thevenot, Axel. 2020. A visual and mathematical explanation of the 2D convolution layer.



**Kernel sizes**

A kernel is a convolutional matrix with commonly used sizes of three-by-three for conv2d layers and two-by-two for max pooling layers. Convolutions involve a dot product between the kernel weights (shown in green) and the image pixel values (shown in pink), and the sum of this dot product creates a feature map (shown in blue).

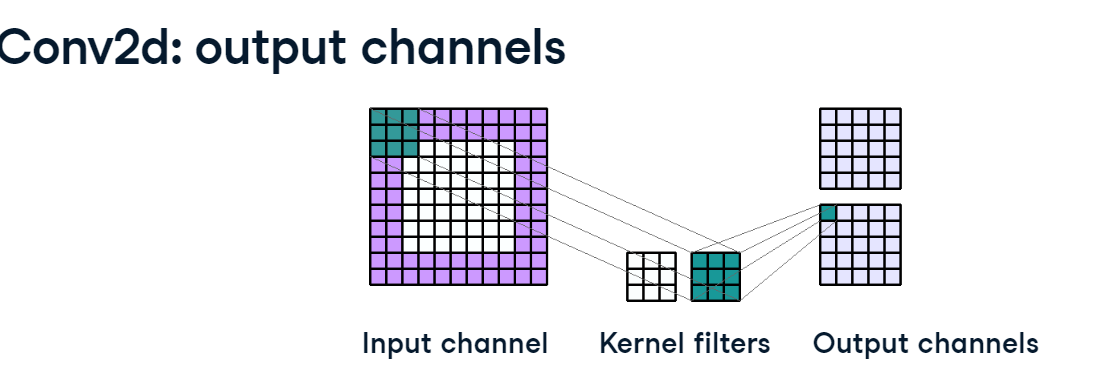


**Kernel is a filter**

This feature map captures essential patterns like edges, boundaries, or shapes. To illustrate, the first filter marks the bird's body. The second filter identifies lines in the building image. These two filters are handcrafted, but in convolutional layers, the filters are trained based on the data.

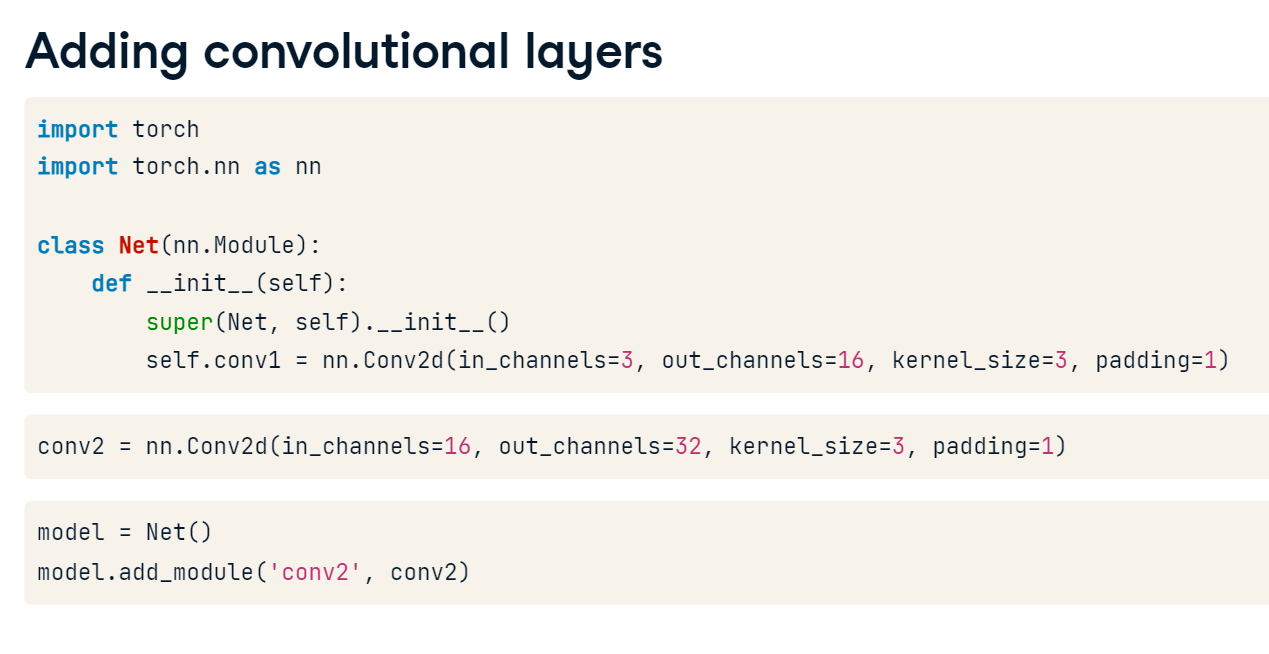
**Conv2d: output channels**

In a convolutional layer, the number of output channels determines how many filters are applied. In this example, there are two output channels with two corresponding filters. Each output channel corresponds to a distinct filter learned during training. Increasing the number of output channels allows the network to learn more complex features. Typically, the number of output channels is a power of two (for example, 16 or 32). It simplifies the process of combining and dividing channels in subsequent layers.



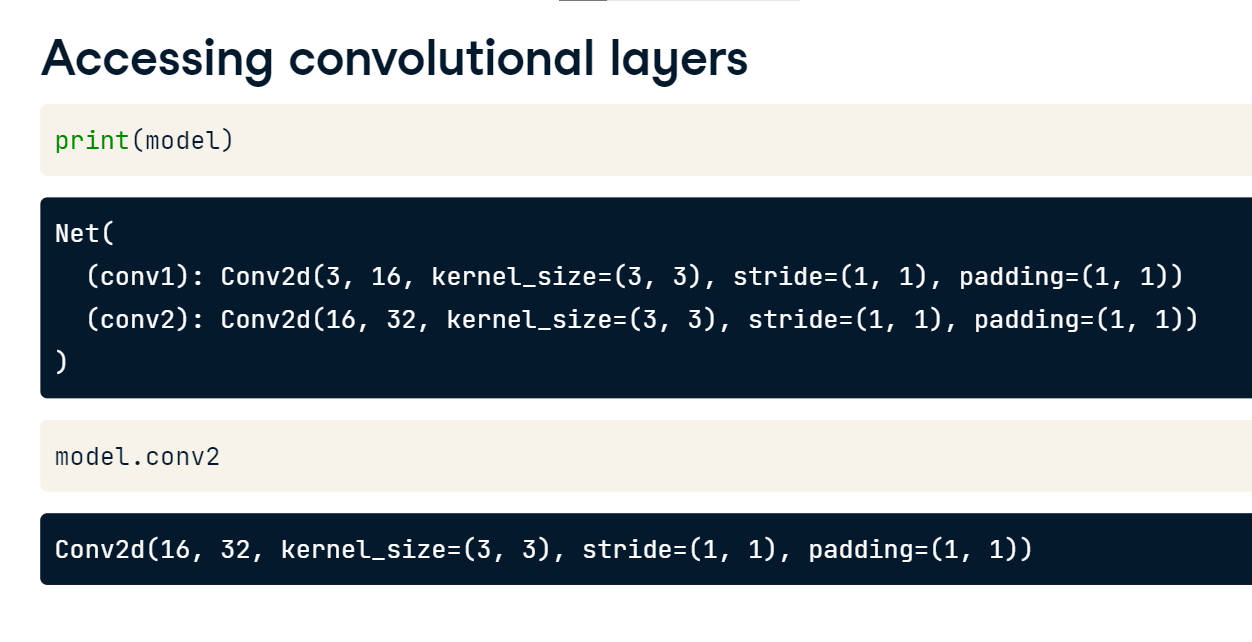
**Adding convolutional layers**

Let's explore how to add more convolutional layers to our model. This comes in handy when the goal is to capture more complex features. We have a model called Net with one layer, conv1. Now, we create an additional layer, conv2. Remember that the in\_channels of conv2 should match the out\_channels of conv1. For conv2, we increase the number of filters to 32. To add the new layer to a model, we first need to instantiate the model. Then, we can incorporate conv2 into the model using the add\_module function with two parameters: the layer's name and the layer.



**Accessing convolutional layers**

Now let's check our model. The model is a two-layer CNN with 3 input channels and 16 output channels in the first convolutional layer, and 16 input channels and 32 output channels in the second convolutional layer. We can also access individual layers, for example, our conv2 layer - this will be handy when we learn about pre-trained models.



**Creating convolutional blocks**

We can also define a sequential block of convolutional layers. This will make our model more flexible to adapt to a different dataset. We use nn-dot-sequential and place the two nn-dot-conv2d layers inside the block along with nn-dot-relu and nn-dot-maxpool2d. The forward method can now pass the input to the conv block instead of separate layers.

