Deep Learning with PyTorch

Print to PDF

Chapter 5: Introduction to Convolutional Neural Networks

第 5 章：卷积神经网络简介 ¶

By Tomas Beuzen 🚀 作者：托马斯·博曾🚀

Chapter Outline 章节大纲 ¶

Chapter Learning Objectives 章节学习目标 ¶

- Describe the terms convolution, kernel/filter, pooling, and flattening
  描述术语“卷积”、“内核/过滤器”、“池化”和“展平”
- Explain how convolutional neural networks (CNNs) work
  解释卷积神经网络 (CNN) 的工作原理
- Calculate the number of parameters in a given CNN architecture
  计算给定 CNN 架构中的参数数量
- Create a CNN in `PyTorch`
  在 `PyTorch` 中创建 CNN
- Discuss the key differences between CNNs and fully connected NNs
  讨论 CNN 和全连接 NN 之间的主要区别

Imports 导入 ¶

```
import numpy as np
import pandas as pd
import torch
from torchsummary import summary
import matplotlib.pyplot as plt
from utils.plotting import *
plt.style.use('ggplot')
plt.rcParams.update({'font.size': 16, 'axes.labelweight': 'bold',
'axes.grid': False})
```

1. Convolutional Neural Networks (CNNs)
1. 卷积神经网络（CNN） ¶

1.1. Motivation 1.1.动机 ¶

Up until now we've been dealing with "fully connected neural networks" meaning that every neuron in a given layer is connected to every neuron in the next layer. This has two key implications:

到目前为止，我们一直在处理"完全连接的神经网络"，这意味着给定层中的每个神经元都连接到下一层中的每个神经元。这有两个关键含义：

1. It results in a LOT of parameters.

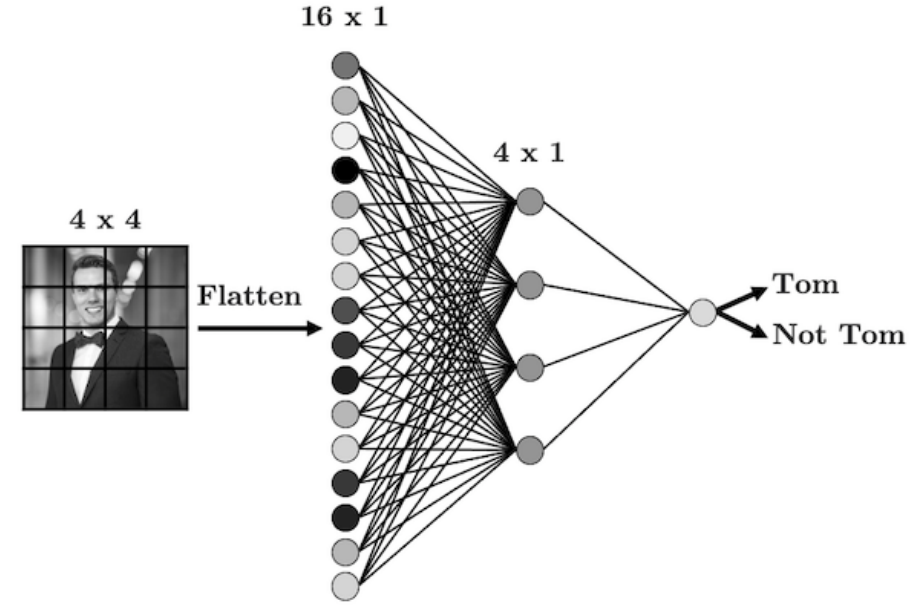
   它会产生很多参数。

2. The order of our features doesn't matter.

   我们的功能的顺序并不重要。

Consider the simple image and fully connected network below:

考虑下面的简单图像和全连接网络：



Every input node is connected to every node in the next layer - is that really necessary? When you look at this image, how do you know that it's me?

每个输入节点都连接到下一层中的每个节点 - 这真的有必要吗？当你看到这张照片时，你怎么知道这是我？

- You notice the structure in the image (there's a face, shoulders, a smile, etc.)

  您注意到图像中的结构（有脸、肩膀、微笑等）

- You notice how different structures are positioned and related (the face is on top of the shoulders, etc.)

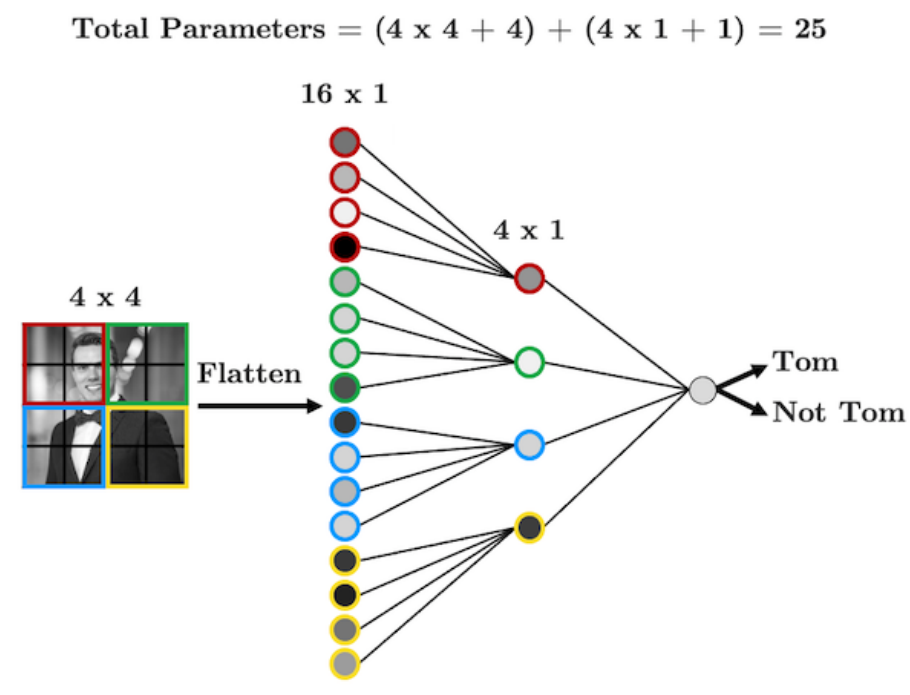
  您会注意到不同结构的定位和关联方式（面部位于肩膀上方等）

- You probably use the shading (colour) to infer things about the image too but we'll talk more about that later.

  您可能也使用阴影（颜色）来推断有关图像的信息，但我们稍后会详细讨论。

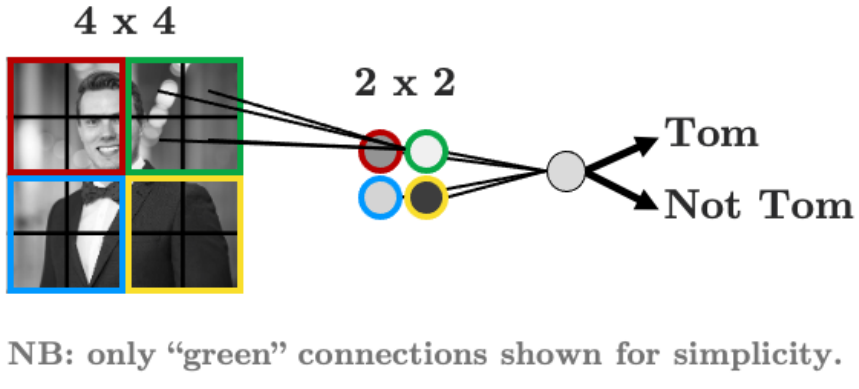
The point here is that **the structure of our data (the pixels) is important**. So maybe, we should have each hidden node only look at a small area of the image, like this:

这里的要点是我们的数据结构（像素）很重要。所以也许我们应该让每个隐藏节点只查看图像的一小部分区域，如下所示：



We have far fewer parameters now because we're acknowledging that pixels that are far apart are probably not all that related and so don't need to be connected. We're seeing that structure is important here, so then why should I need to flatten my image at all? Let's be crazy and not flatten the image, but instead, make our hidden layer a 2D matrix:

我们现在的参数少得多，因为我们承认相距较远的像素可能并不那么相关，因此不需要连接。我们看到结构在这里很重要，那么为什么我需要扁平化我的图像呢？让我们疯狂一下，不要将图像展平，而是将我们的隐藏层设为 2D 矩阵：



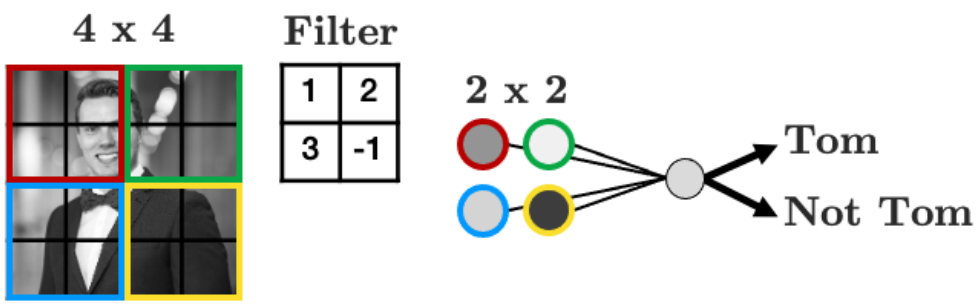
We're almost there! 我们快到了！

As it stands, each group of 2 x 2 pixels has 4 unique weights associated with it (one for each pixel), which are being summed up into a single value in the hidden layer. But we don't need the weights to be different for each group, we're looking for **structure**, we don't care if my face is in the top left or the bottom right, we're just looking for a face!

目前，每组 2 x 2 像素都有 4 个与其关联的唯一权重（每个像素一个），这些权重在隐藏层中被汇总为单个值。但我们不需要每组的权重都不同，我们正在寻找结构，我们不在乎我的脸是在左上角还是右下角，我们只是在寻找一张脸！

Let's summarise the weights into a weight "filter":

让我们将权重总结为权重"过滤器"：



- Let's see how the filter works

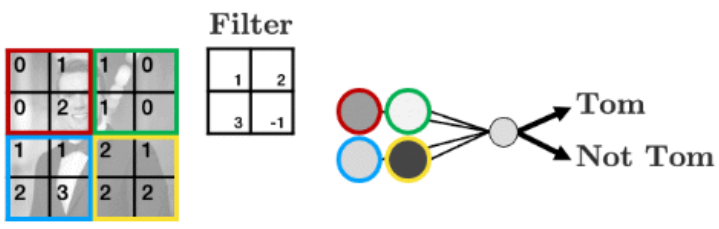
  让我们看看过滤器是如何工作的

- We'll display some arbitrary values for our pixels

  我们将显示像素的一些任意值

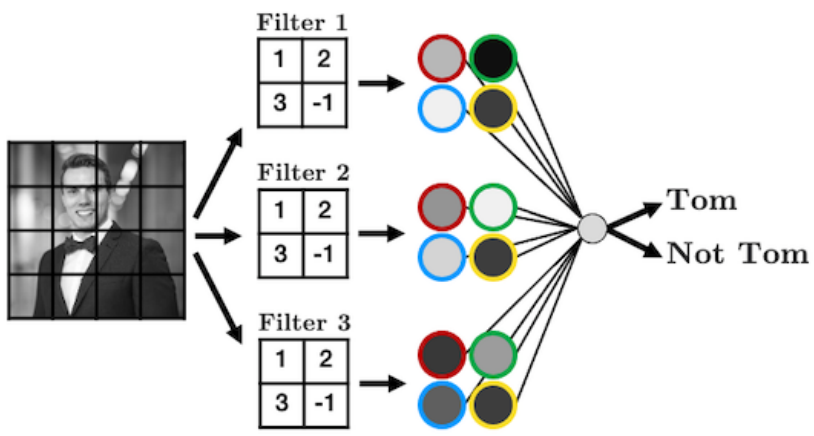
- The filter "convolves" over each group of pixels, multiplies corresponding elements and sums them up to give the values in the output nodes:

  过滤器对每组像素进行"卷积"，将相应的元素相乘并将它们相加以给出输出节点中的值：



As we'll see, we can add as many of these "filters" as we like to make more complex models that can identify more useful things:

正如我们将看到的，我们可以添加任意数量的"过滤器"，以创建更复杂的模型来识别更有用的东西：



We just made a **convolutional neural network** (CNN). Instead of fully-connected hidden nodes, we have 2D filters that we "convolve" over our input data. This has two key advantages:

我们刚刚制作了一个卷积神经网络（CNN）。我们没有完全连接的隐藏节点，而是使用 2D 过滤器对输入数据进行"卷积"。这有两个主要优点：

1. We have less parameters than a fully connected network.

   我们的参数比全连接网络少。

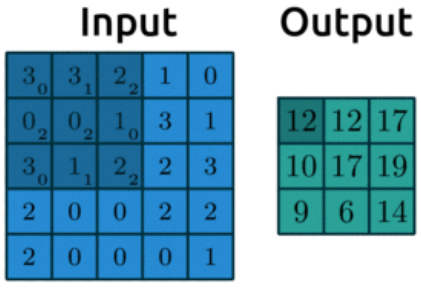
2. We preserve the useful structure of our data.

   我们保留数据的有用结构。

1.2. Convolutions and Filters
1.2.卷积和过滤器 ¶

Convolution really just means “to pass over the data”. What are we “passing”? Our filters - which are also called **kernels**. Here’s another gif like the one we saw earlier:

卷积实际上只是意味着“传递数据”。我们“传递”的是什么？我们的过滤器 - 也称为内核。这是另一张 gif，就像我们之前看到的那样：



> Source: modified after theano-pymc.readthedocs.io.
>
> 来源：theano-pymc.readthedocs.io 之后修改。

So how does this help us extract structure from the data? Well let’s see some examples!

那么这如何帮助我们从数据中提取结构呢？好吧，让我们看一些例子！

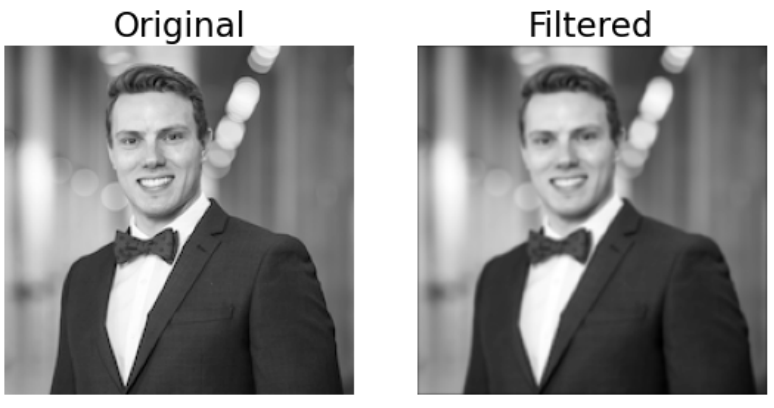
```
image = torch.from_numpy(plt.imread("img/tom_bw.png"))
plt.imshow(image, cmap='gray')
plt.axis('off');
```

We can blur this image by applying a filter with the following weights:

我们可以通过应用具有以下权重的过滤器来模糊该图像：

$$\begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

```
kernel = torch.tensor([[[[ 0.0625,  0.1250,  0.0625],
                         [ 0.1250,  0.2500,  0.1250],
                         [ 0.0625,  0.1250,  0.0625]]]])
plot_conv(image, kernel)
```

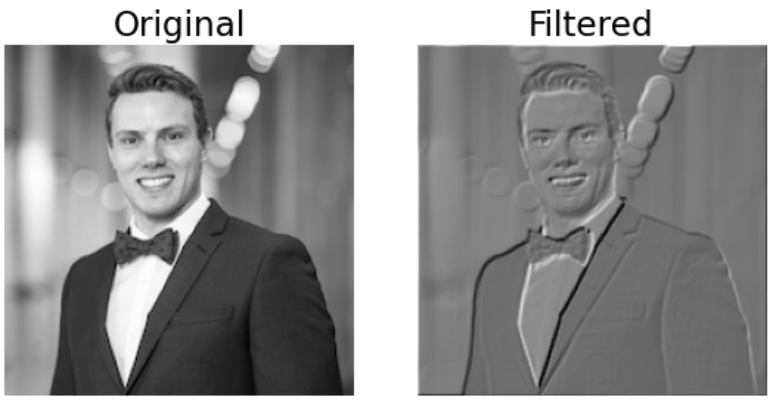


How about this one:

这个怎么样：

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

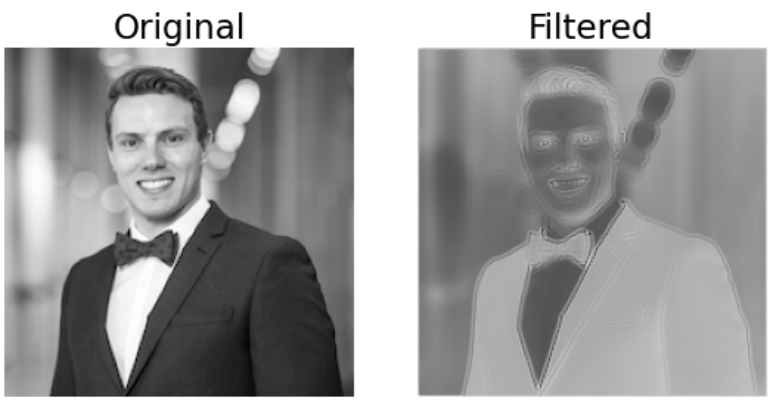
```
kernel = torch.tensor([[[[ -2,  -1,  0],
                         [ -1,   1,  1],
                         [  0,   1,  2]]]])
plot_conv(image, kernel)
```



One more: 多一个：

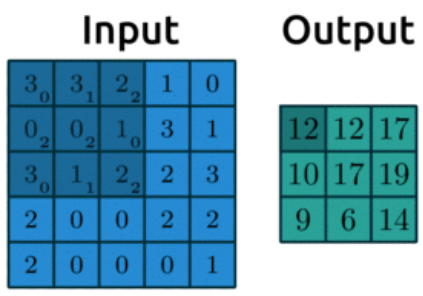
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

```
kernel = torch.tensor([[[[  -1,  -1,   -1],
                         [  -1,   8,   -1],
                         [  -1,  -1,   -1]]]])
plot_conv(image, kernel)
```



Here's a great website where you can play around with other filters. We usually use **odd numbers for filters** so that they are applied symmetrically around our input data. Did you notice in the gif earlier that the output from applying our kernel was smaller than the input? Take a look again:

这是一个很棒的网站，您可以在其中尝试其他过滤器。我们通常使用奇数作为过滤器，以便它们对称地应用在我们的输入数据周围。您是否注意到之前的 gif 中应用内核的输出小于输入？再看一下：



> Source: modified after theano-pymc.readthedocs.io.
>
> 来源：theano-pymc.readthedocs.io 之后修改。

By default, our kernels are only applied where the filter fully fits on top of the input. But we can control this behaviour and the size of our output with:

默认情况下，我们的内核仅适用于过滤器完全适合输入的情况。但我们可以通过以下方式控制这种行为和输出的大小：

- `padding`: "pads" the outside of the input 0's to allow the kernel to reach the boundary pixels

  `padding`： "填充"输入 0 的外部，以允许内核到达边界像素

- `strides`: controls how far the kernel "steps" over pixels.

  `strides`： 控制内核"步进"像素的距离。

Below is an example with:

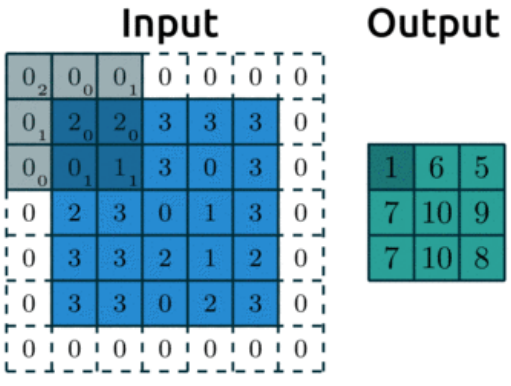
下面是一个示例：

- `padding=1`: we have `1` layer of 0's around our border

  `padding=1`： 我们的边界周围有 `1` 层 0

- `strides=(2,2)`: our kernel moves 2 data points to the right for each row, then moves 2 data points down to the next row

  `strides=(2,2)`： 我们的内核将每行的 2 个数据点向右移动，然后将 2 个数据点向下移动到下一行



> Source: modified after theano-pymc.readthedocs.io.
>
> 来源：theano-pymc.readthedocs.io 之后修改。

We'll look more at these below.

我们将在下面详细介绍这些内容。

2. Cooking up a CNN
2. 构建 CNN ¶

2.1. Ingredient 1: Convolutional Layers
2.1.成分 1：卷积层 ¶

I showed some example kernels above. In CNNs the actual values in **the kernels are the weights your network will learn during training**: your network will learn what structures are important for prediction.

我在上面展示了一些示例内核。在 CNN 中，内核中的实际值是您的网络在训练期间将学习的权重：您的网络将学习哪些结构对于预测很重要。

In PyTorch, convolutional layers are defined as `torch.nn.Conv2d`, there are 5 important arguments we need to know:

在 PyTorch 中，卷积层定义为 `torch.nn.Conv2d` ，我们需要知道 5 个重要参数：

1. `in_channels`: how many features are we passing in. Our features are our colour bands, in greyscale, we have 1 feature, in colour, we have 3 channels.

   `in_channels`：我们传入了多少个特征。我们的特征是我们的色带，在灰度中，我们有 1 个特征，在彩色中，我们有 3 个通道。

2. `out_channels`: how many kernels do we want to use. Analogous to the number of hidden nodes in a hidden layer of a fully connected network.

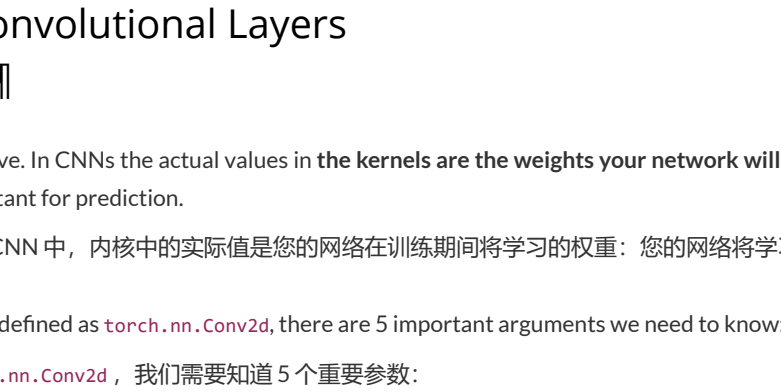
   `out_channels`：我们要使用多少个内核。类似于全连接网络的隐藏层中隐藏节点的数量。

3. `kernel_size`: the size of the kernel. Above we were using 3x3. Common sizes are 3x3, 5x5, 7x7.

   `kernel_size`：内核的大小。上面我们使用的是 3x3。常见尺寸为 3x3、5x5、7x7。

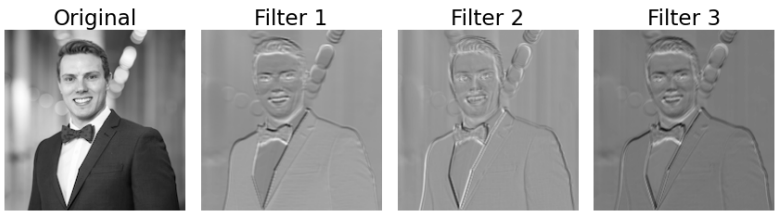
4. `stride`: the "step-size" of the kernel.
5. `padding`: the number of pixels we should pad to the outside of the image so we can get edge pixels.

```
# 1 kernel of (3,3)
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(5, 5))
plot_convs(image, conv_layer)
```



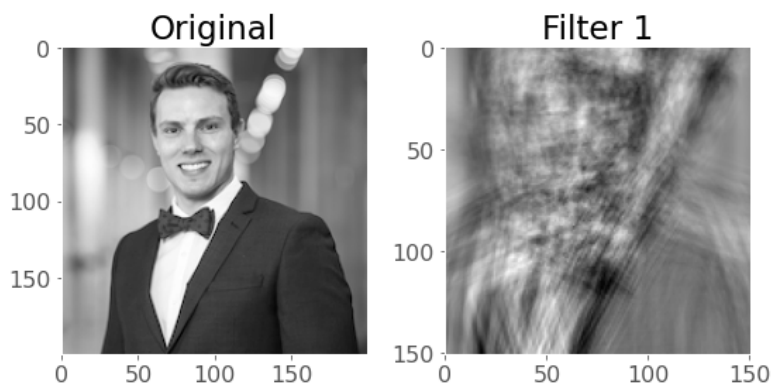

```
# 2 kernels of (3,3)
conv_layer = torch.nn.Conv2d(1, 2, kernel_size=(3, 3))
plot_convs(image, conv_layer)
```


```
# 3 kernels of (5,5)
conv_layer = torch.nn.Conv2d(1, 3, kernel_size=(5, 5))
plot_convs(image, conv_layer)
```



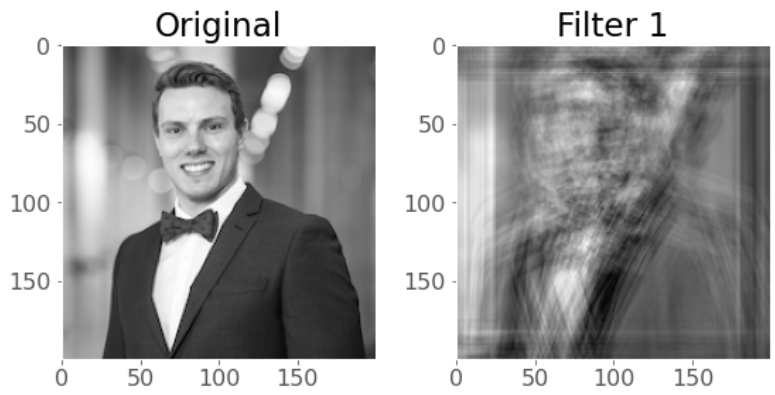
If we use a kernel with no padding, our output image will be smaller as we noted earlier. Let's demonstrate that by using a larger kernel now:

```
# 1 kernel of (51,51)
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(50, 50))
plot_convs(image, conv_layer, axis=True)
```



As we saw, we can add `padding` to the outside of the image to avoid this:

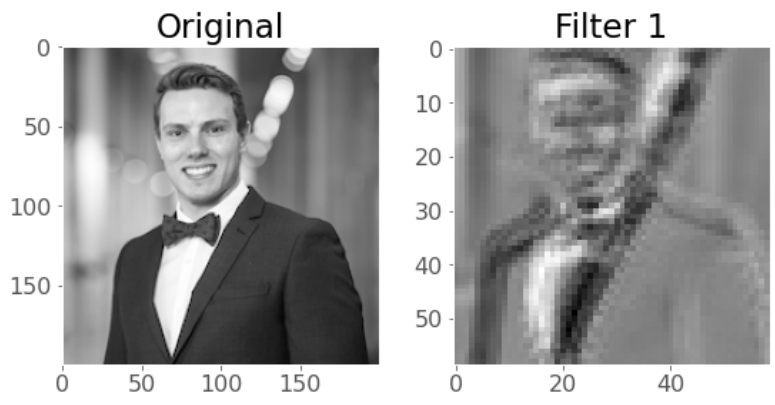
```
# 1 kernel of (51,51) with padding
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(51, 51), padding=25)
plot_convs(image, conv_layer, axis=True)
```



> Setting `padding = kernel_size // 2` will always result in an output the same shape as the input. Think about why this is...

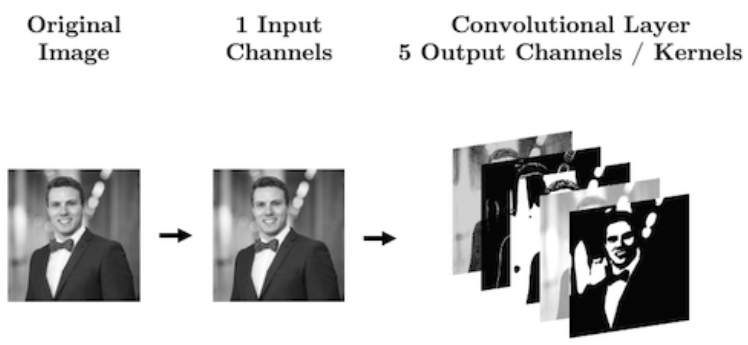
Finally, we also saw before how `strides` influence the size of the output:

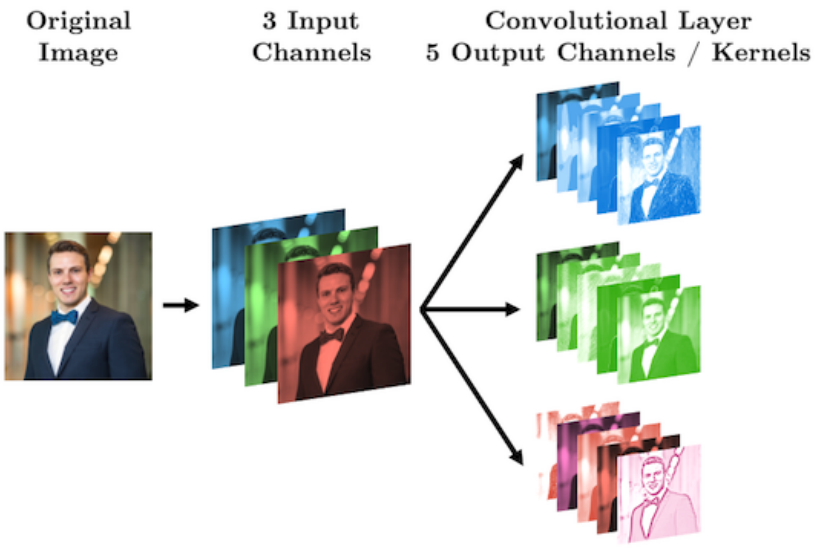
```
# 1 kernel of (25,25) with stride of 3
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(25, 25), stride=3)
plot_convs(image, conv_layer, axis=True)
```



With CNN we are no longer flattening our data, so what are our "features"? Our features are called "channels" in CNN-lingo -> they are like the colour channels in an image:

- A grayscale image has 1 feature/channel
- A coloured image has 3 features/channel





What's important with CNNs is that the **size of our input data does not impact how many parameters we have in our convolutonal layers**. For example, your kernels don't care how big your image is (i.e., 28 x 28 or 256 x 256), all that matters is:

1. How many features ("channels") you have: `in_channels`
2. How many filters you use in each layer: `out_channels`
3. How big the filters are: `kernel_size`

Let's see some diagrams:

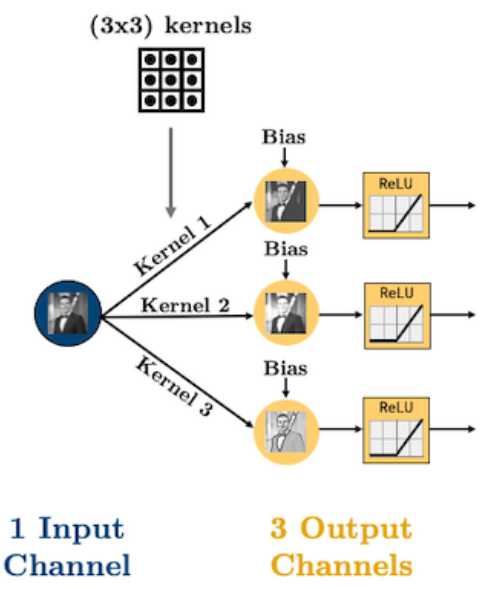
```
nn.Conv2d(
        in_channels=1,
        out_channels=3,
        kernel_size=(3,3)
)
```

Total number of parameters:

- 1 input channel
- 3 output channels
- (3 x 3) filters

1 x 3 x (3 x 3)
= 27 parameters

+ 3 biases
= 30 parameters



For coloured images (3 channels):

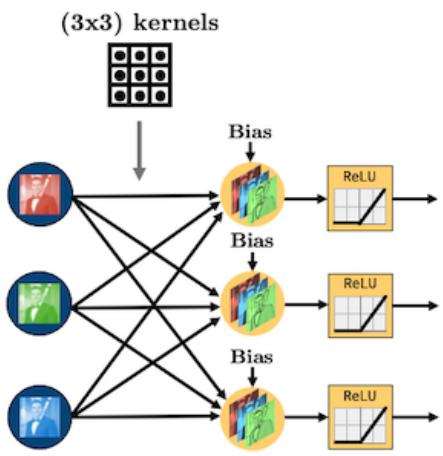
```
nn.Conv2d(
        in_channels=3,
        out_channels=3,
        kernel_size=(3,3)
)
```

Total number of parameters:

- 3 input channels
- 3 output channels
- (3 x 3) filters

3 x 3 x (3 x 3)
= 81 parameters

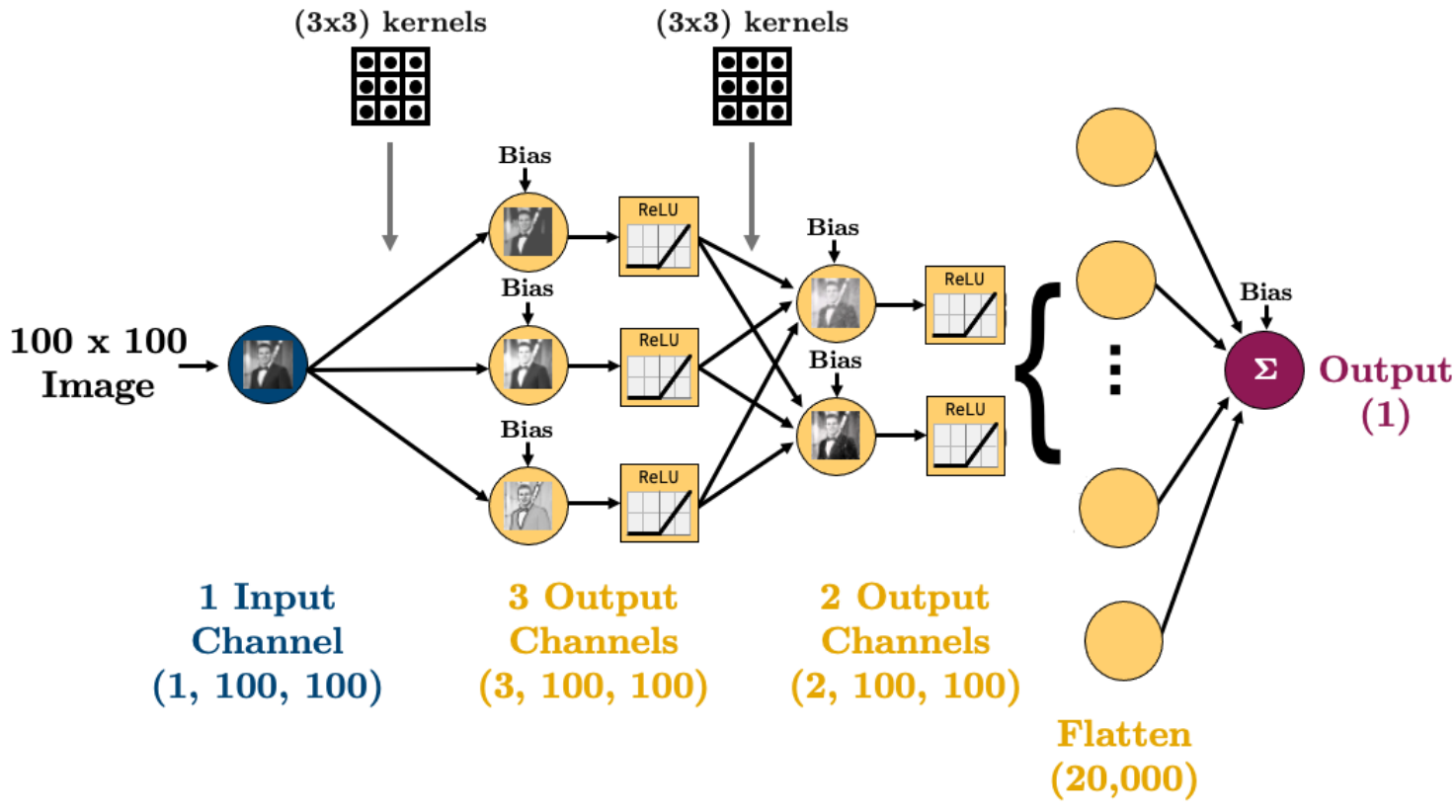
+ 3 biases
= 84 parameters



2.2. Ingredient 2: Flattening

With our brand new, shiny convolutional layers, we're basically just passing images through the network - cool!

But we're going to eventually want to do some regression or classification. That means that by the end of our network, we are going to need to `torch.nn.Flatten()` our images:



Let's make that simple CNN above in PyTorch:

```
class CNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.main = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1, out_channels=3,
kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),
            torch.nn.Conv2d(in_channels=3, out_channels=2,
kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),
            torch.nn.Flatten(),
            torch.nn.Linear(20000, 1)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

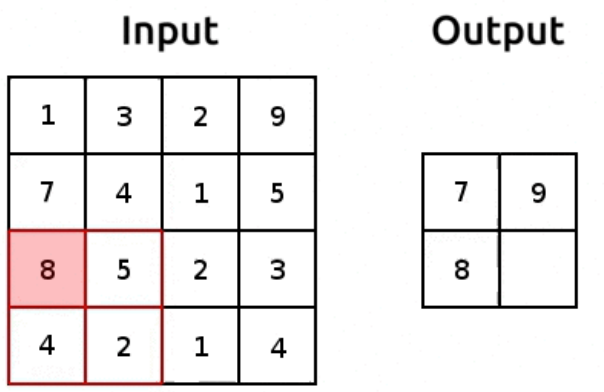
```
model = CNN()
summary(model, (1, 100, 100));
```

```
==========================================================================
======================
Layer (type:depth-idx)                   Output Shape
Param #
==========================================================================
======================
├─Sequential: 1-1                        [-1, 1]                   -
-
|    └─Conv2d: 2-1                       [-1, 3, 100, 100]
30
|    └─ReLU: 2-2                         [-1, 3, 100, 100]         -
-
|    └─Conv2d: 2-3                       [-1, 2, 100, 100]
56
|    └─ReLU: 2-4                         [-1, 2, 100, 100]         -
-
|    └─Flatten: 2-5                      [-1, 20000]               -
-
|    └─Linear: 2-6                       [-1, 1]
20,001
==========================================================================
======================
Total params: 20,087
Trainable params: 20,087
Non-trainable params: 0
Total mult-adds (M): 0.85
==========================================================================
======================
Input size (MB): 0.04
Forward/backward pass size (MB): 0.38
Params size (MB): 0.08
Estimated Total Size (MB): 0.50
==========================================================================
======================
```

Oh man! 20,000 parameters in that last layer, geez. Is there a way we can reduce this somehow? Glad you asked! See you in the next section.

2.3. Ingredient 3: Pooling

Pooling is how we can reduce the number of parameters we get out of a `torch.nn.Flatten()`. It's pretty simple, we just aggregate the data, usually using the maximum or average of a window of pixels. Here's an example of max pooling:



> Source: modified after www.oreilly.com/.

We use “pooling layers” to reduce the shape of our image as it’s passing through the network. So when we eventually `torch.nn.Flatten()`, we’ll have less features in that flattened layer! We can implement pooling with `torch.nn.MaxPool2d()`. Let’s try it out and reduce the number of parameters:

```
class CNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.main = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1, out_channels=3,
kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d((2, 2)),
            torch.nn.Conv2d(in_channels=3, out_channels=2,
kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d((2, 2)),
            torch.nn.Flatten(),
            torch.nn.Linear(1250, 1)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

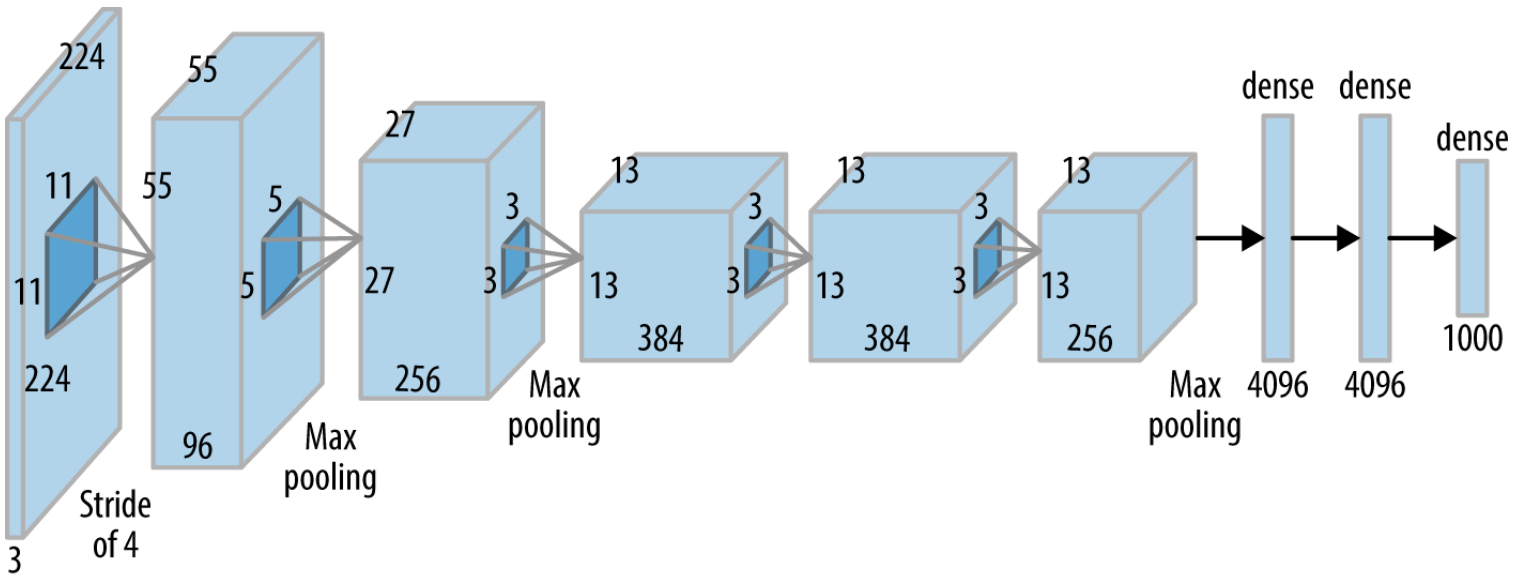
```
model = CNN()
summary(model, (1, 100, 100));
```

```
==========================================================================
======================
Layer (type:depth-idx)                   Output Shape
Param #
==========================================================================
======================
├─Sequential: 1-1                        [-1, 1]                   -
-
|    └─Conv2d: 2-1                       [-1, 3, 100, 100]
30
|    └─ReLU: 2-2                         [-1, 3, 100, 100]         -
-
|    └─MaxPool2d: 2-3                    [-1, 3, 50, 50]           -
-
|    └─Conv2d: 2-4                       [-1, 2, 50, 50]
56
|    └─ReLU: 2-5                         [-1, 2, 50, 50]           -
-
|    └─MaxPool2d: 2-6                    [-1, 2, 25, 25]           -
-
|    └─Flatten: 2-7                      [-1, 1250]                -
-
|    └─Linear: 2-8                       [-1, 1]
1,251
==========================================================================
======================
Total params: 1,337
Trainable params: 1,337
Non-trainable params: 0
Total mult-adds (M): 0.41
==========================================================================
======================
Input size (MB): 0.04
Forward/backward pass size (MB): 0.27
Params size (MB): 0.01
Estimated Total Size (MB): 0.31
==========================================================================
======================
```

We reduced that last layer to 1,251 parameters. Nice job!

3. The CNN Recipe Book

Here’s a CNN diagram of a famous architecture called AlexNet (we’ll talk more about “famous architectures” next Chapter):



You actually know what all of the above means now! But, deep learning and CNN architecture remains very much an art. Here is my general recipe book (based on experience, common practice, and popular pre-made architectures - more on those next chapter).

Typical ingredients (in order):

- Convolution layer(s): `torch.nn.Conv2d`
- Activation function: `torch.nn.ReLU`, `torch.nn.Sigmoid`, `torch.nn.Softplus`, etc.
- (optional) Batch normalization: `torch.nn.BatchNorm2d` (more on that next Chapter)
- (optional) Pooling: `torch.nn.MaxPool2d`
- (optional) Drop out: `torch.nn.Dropout`
- Flatten: `torch.nn.Flatten`

4. CNN vs Fully Connected NN

As an example of the parameter savings introduced when using CNNs with structured data, let's compare the Bitmoji classifier from last chapter, with an equivalent CNN version.

We'll replace all linear layers with convolutional layers with 3 kernels of size (3, 3) and will assume an image size of 128 x 128:

```
def linear_block(input_size, output_size):
    return torch.nn.Sequential(
        torch.nn.Linear(input_size, output_size),
        torch.nn.ReLU()
    )

def conv_block(input_channels, output_channels):
    return torch.nn.Sequential(
        torch.nn.Conv2d(input_channels, output_channels, (3, 3),
padding=1),
        torch.nn.ReLU()
    )

class NN(torch.nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.main = torch.nn.Sequential(
            linear_block(input_size, 256),
            linear_block(256, 128),
            linear_block(128, 64),
            linear_block(64, 16),
            torch.nn.Linear(16, 1)
        )

    def forward(self, x):
        out = self.main(x)
        return out

class CNN(torch.nn.Module):
    def __init__(self, input_channels):
        super().__init__()
        self.main = torch.nn.Sequential(
            conv_block(input_channels, 3),
            conv_block(3, 3),
            conv_block(3, 3),
            conv_block(3, 3),
            conv_block(3, 3),
            torch.nn.Flatten(),
            torch.nn.Linear(49152, 1)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

```
model = NN(input_size=128 * 128)
summary(model, (128 * 128,));
```

```
==========================================================================
======================
Layer (type:depth-idx)                   Output Shape
Param #
==========================================================================
======================
├─Sequential: 1-1                        [-1, 1]                   -
-
|    └─Sequential: 2-1                   [-1, 256]                 -
-
|    |    └─Linear: 3-1                  [-1, 256]
4,194,560
|    |    └─ReLU: 3-2                    [-1, 256]                 -
-
|    └─Sequential: 2-2                   [-1, 128]                 -
-
|    |    └─Linear: 3-3                  [-1, 128]
32,896
|    |    └─ReLU: 3-4                    [-1, 128]                 -
-
|    └─Sequential: 2-3                   [-1, 64]                  -
-
|    |    └─Linear: 3-5                  [-1, 64]
8,256
|    |    └─ReLU: 3-6                    [-1, 64]                  -
-
|    └─Sequential: 2-4                   [-1, 16]                  -
-
|    |    └─Linear: 3-7                  [-1, 16]
1,040
|    |    └─ReLU: 3-8                    [-1, 16]                  -
-
|    └─Linear: 2-5                       [-1, 1]
17
==========================================================================
======================
Total params: 4,236,769
Trainable params: 4,236,769
Non-trainable params: 0
Total mult-adds (M): 12.71
==========================================================================
======================
Input size (MB): 0.06
Forward/backward pass size (MB): 0.00
Params size (MB): 16.16
Estimated Total Size (MB): 16.23
==========================================================================
======================
```

```
model = CNN(input_channels=1)
summary(model, (1, 128, 128));
```

```
==========================================================================
======================
Layer (type:depth-idx)                   Output Shape
Param #
==========================================================================
======================
├─Sequential: 1-1                        [-1, 1]                   -
-
|    └─Sequential: 2-1                   [-1, 3, 128, 128]         -
-
|    |    └─Conv2d: 3-1                  [-1, 3, 128, 128]
30
|    |    └─ReLU: 3-2                    [-1, 3, 128, 128]         -
-
|    └─Sequential: 2-2                   [-1, 3, 128, 128]         -
-
|    |    └─Conv2d: 3-3                  [-1, 3, 128, 128]
84
|    |    └─ReLU: 3-4                    [-1, 3, 128, 128]         -
-
|    └─Sequential: 2-3                   [-1, 3, 128, 128]         -
-
|    |    └─Conv2d: 3-5                  [-1, 3, 128, 128]
84
|    |    └─ReLU: 3-6                    [-1, 3, 128, 128]         -
-
|    └─Sequential: 2-4                   [-1, 3, 128, 128]         -
-
|    |    └─Conv2d: 3-7                  [-1, 3, 128, 128]
84
|    |    └─ReLU: 3-8                    [-1, 3, 128, 128]         -
-
|    └─Sequential: 2-5                   [-1, 3, 128, 128]         -
-
|    |    └─Conv2d: 3-9                  [-1, 3, 128, 128]
84
|    |    └─ReLU: 3-10                   [-1, 3, 128, 128]         -
-
|    └─Flatten: 2-6                      [-1, 49152]               -
-
|    └─Linear: 2-7                       [-1, 1]
49,153
==========================================================================
======================
Total params: 49,519
Trainable params: 49,519
Non-trainable params: 0
Total mult-adds (M): 5.85
==========================================================================
======================
Input size (MB): 0.06
Forward/backward pass size (MB): 1.88
Params size (MB): 0.19
Estimated Total Size (MB): 2.13
==========================================================================
======================
```

We don’t even have any pooling and our CNN still has a “meager” 49,519 parameters vs 4,236,769 for the fully-connected network. This is a somewhat arbitray comparison but it proves my point.

By Tomas Beuzen