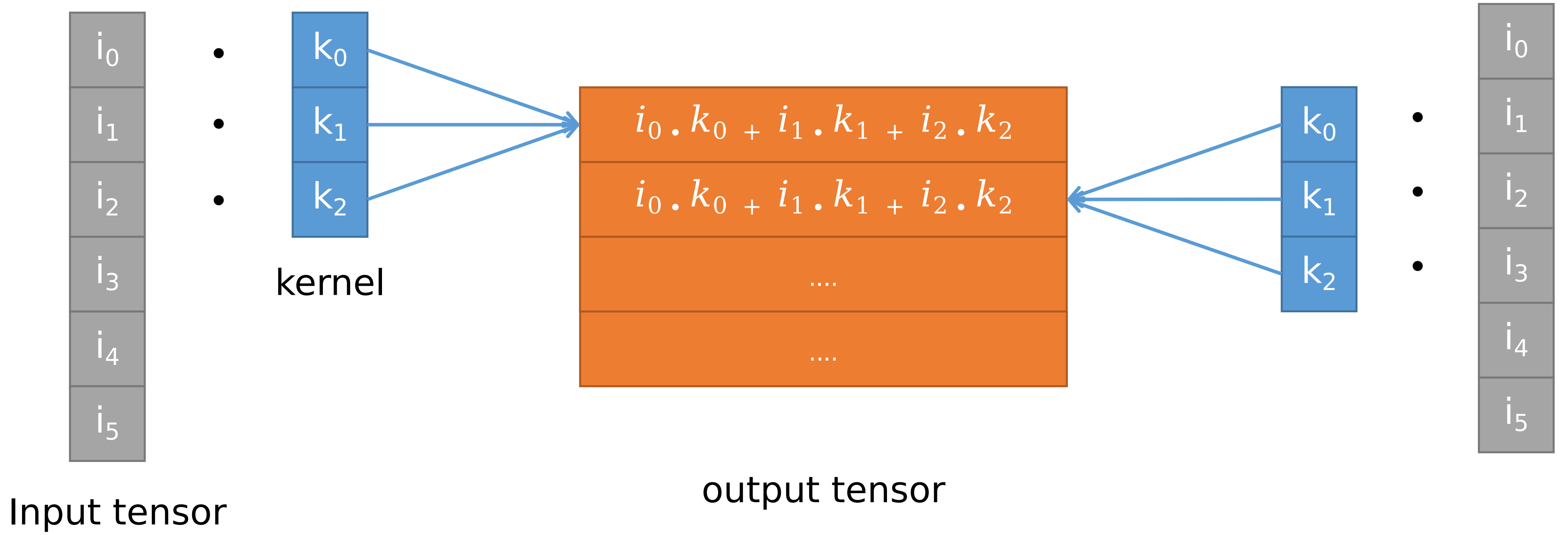
CNN

The convolution operation

The **convolution** operation consists of **sliding** a **kernel** (also called **weights**) across the **dimensions** of a **tensor** and **computing the sum of products** of the corresponding **tensor values** and **weights**

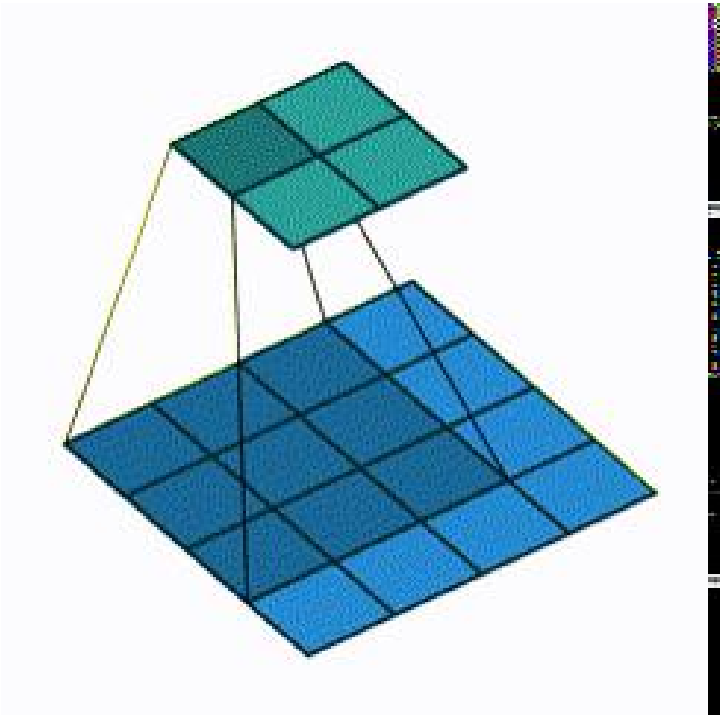


The convolution operation in 2D

In **2D** the convolutional **kernel slides** across the **width** and **height** of the input tensor.

<u>For a 2D convolution:</u>

- If the **input** is a **two-dimensional** tensor
  - The **kernel** is a **two-dimensional** tensor
- If the **input** is a **three-dimensional** tensor (like an RGB image [w x h x 3])
  - The **kernel** is a **three-dimensional** tensor with the **last dimension matching** the one of the **input**
- The **input can** be a **four-dimensional** tensor but the **first dimension** must be the **batch** one



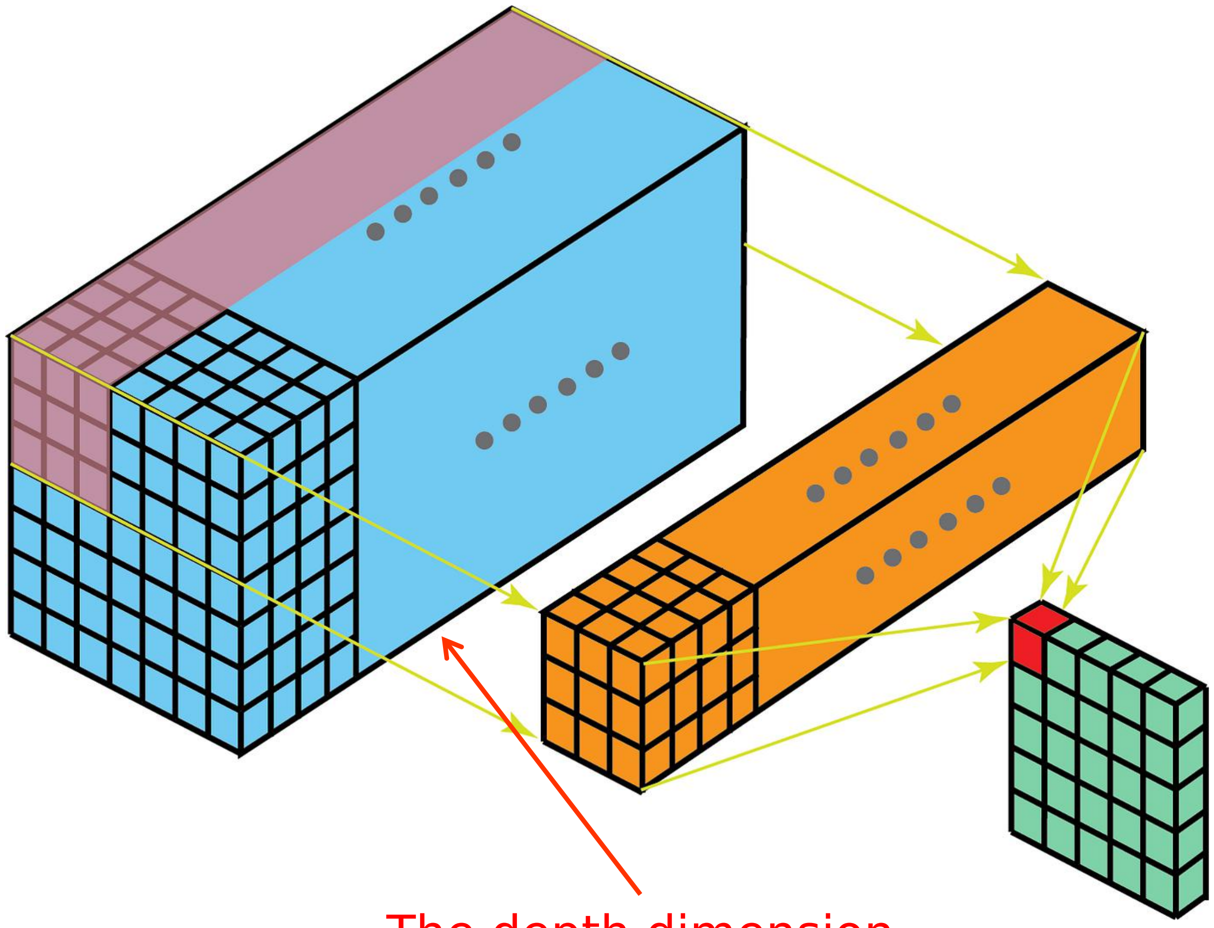
The convolution operation in 2D

In the 2D convolution the **last dimension** of the **input tensor** and **kernel** must **match**

- Input tensor $\mathbf{H_i}$ **x** $\mathbf{W_i}$ **x** **D**
- Kernel $\mathbf{H_k}$ **x** $\mathbf{W_k}$ **x** **D**

The **output** is:

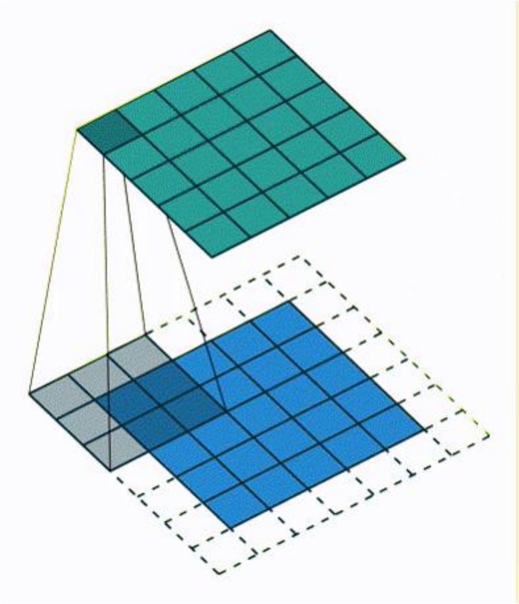
$\mathbf{H_o}$ **x** $\mathbf{W_o}$ **x** **1**

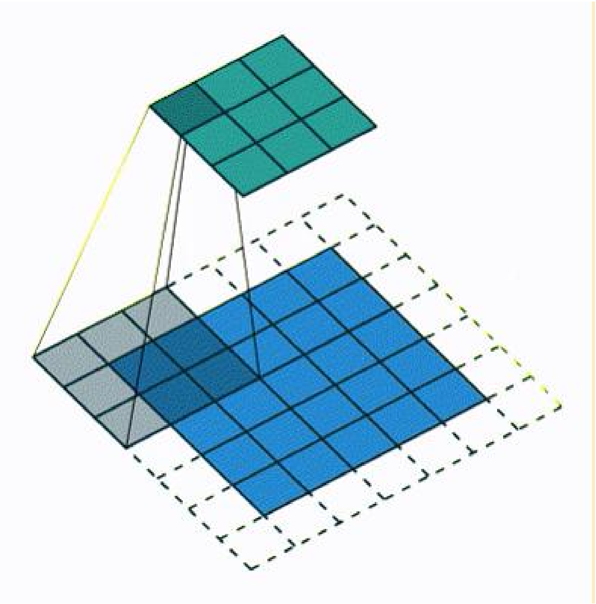


The depth dimension is called "channels"

Convolution: padding and stride

- **Padding** means **adding extra pixels** (zeros, for example) **around** the **input** tensor. This is done to ensure that the convolutional operation can process the entire input, especially at the edges.
- **Stride** refers to the **step size** at which the convolutional **kernel moves** across the input tensor. A **larger stride** results in a **smaller output size**, as the kernel skips more pixels with each step.



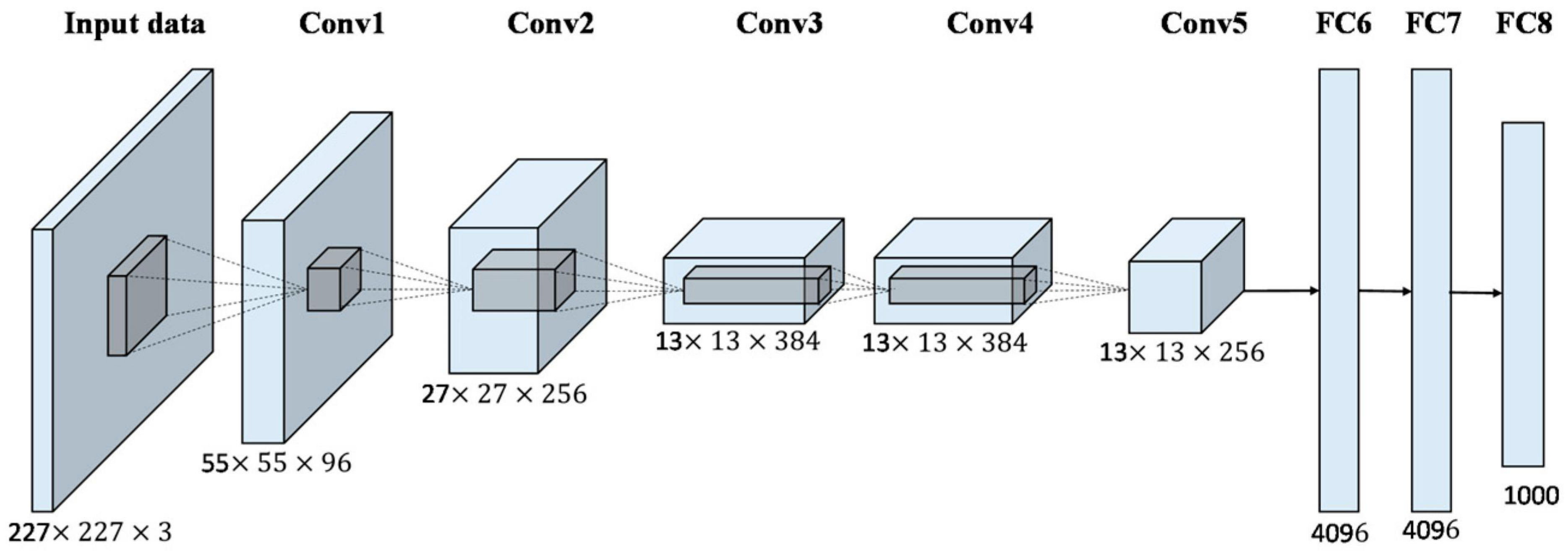


A 2D convolution with N output channels

In this example the **first convolution** is using an **11x11x3 kernel** with **stride=4**, the **input** tensor is **227x227x3** and the **output** is **55x55x96**.

Shouldn’t be 55x55x1?

Yes. There are **96** 11x11x3 **kernels**.



Conv2d layer in PyTorch

The **Conv2d** layer in PyTorch performs **N convolutions** with **N kernels** over the **same input tensor** and **add a bias** to each result, the **output** of the Conv2D layer is a tensor with a **depth of N**

```
import torch

conv = torch.nn.Conv2d(3, 96, (11, 11), stride=4)
```

Input channels output channels kernel size arguments

```
print(conv.weight.shape)
print(conv.bias.shape)
```

```
torch.Size([96, 3, 11, 11])
torch.Size([96])
```

Conv2d layer in PyTorch

The convolutions are carried out in **parallel.**

Since the **machine** needs to **perform** the **same operation** over **multiple inputs** it can exploit the **SIMD** paradigm.

That's why **GPU** are **perfect** for this kind of **operation**.

```
input = torch.rand((32, 3, 227, 227))
output = conv(input)
print(output.shape)
```

```
torch.Size([32, 96, 55, 55])
```

Pooling Layers

You can **reduce** the **height** and **width dimensions** of a three-dimensional **tensor using** a convolution with a **stride or** you can use a **pooling layer**.

A **pooling layer** is a layer that **applies** a **reduction** operation over a convolution **sliding window.**

```
import torch

maxpool = torch.nn.MaxPool2d((2, 2), stride=1)
input = torch.rand(1, 3, 3)
output = maxpool(input)

print(input)
print(output)
```

Activation functions

Activation functions introduce **non-linearity** into the model.

Convolutions are **linear operations** and a **composition** of **linear** operations **is** a **linear** operation.

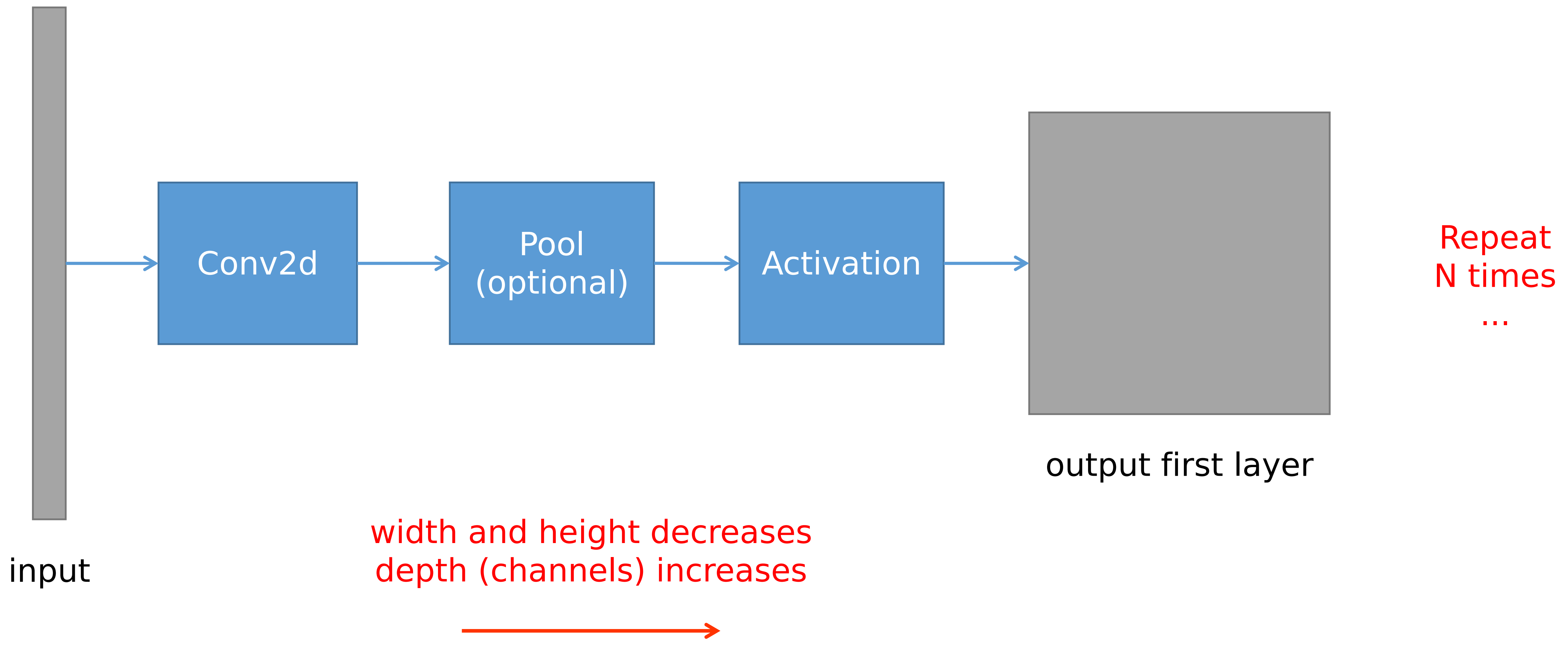
A **composition** of **convolutions** is **equal** to a **single convolution**.

By **following** each **linear** operation **with** a **non-linearity** this **problem** is **solved**

Common activation functions include:

- Sigmoid
  - Squashes values between 0 and 1
- Hyperbolic Tangent (tanh)
  - Squashes values between -1 and 1
- Rectified Linear Unit (ReLU)
  - Replaces negative values with zero

Anatomy of a Convolutional Neural Network



Convolutional Neural Network in PyTorch

```
batch, _ = next(iter(dl))

cnn = torch.nn.Sequential(
    torch.nn.Conv2d(1, 16, (3, 3), stride=1, padding=1),
    torch.nn.MaxPool2d((2, 2), stride=1, padding=0),
    torch.nn.LeakyReLU(),
    torch.nn.Conv2d(16, 32, (3, 3), stride=2, padding=1),
    torch.nn.MaxPool2d((2, 2), stride=2, padding=0),
    torch.nn.LeakyReLU(),
    torch.nn.Conv2d(32, 64, (3, 3), stride=2, padding=0),
    torch.nn.MaxPool2d((2, 2), stride=1, padding=0),
    torch.nn.LeakyReLU(),
)

print(batch.shape)
print(cnn(batch).shape)
```

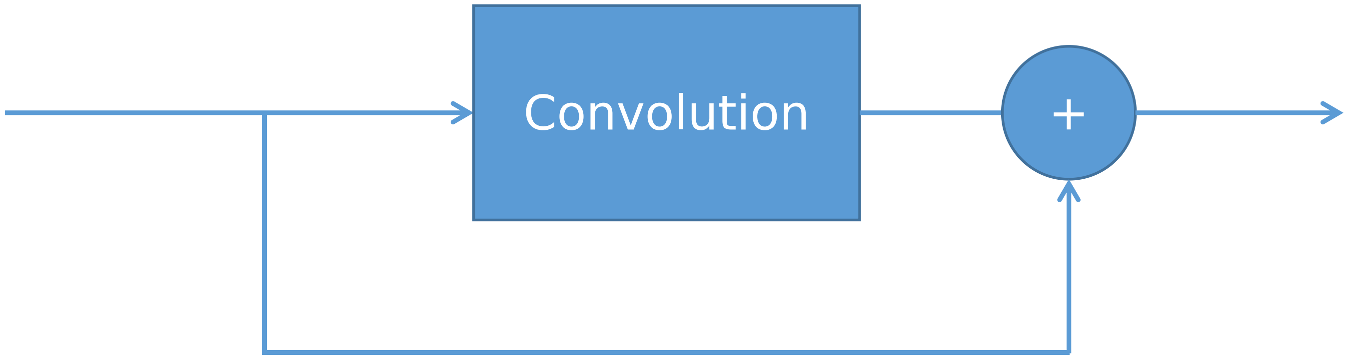
```
torch.Size([8, 1, 28, 28])
torch.Size([8, 64, 2, 2])
```

Residual Networks (ResNet)

ResNet is a deep learning **architecture** designed to **address** the **challenge** of training **very deep** neural networks.

It **introduces** the concept of **residual blocks**, where the **input** to a block is **combined** with its **output** through **skip connections**.

These **skip connection**s allow the **gradient** to **flow** more **directly** through the network during **training**, **mitigating** the **vanishing gradient problem**



How to make a convolution without changing the tensor shape?

Residual Networks (ResNet): half-padding

To make a convolution which preserve the tensor shape you have to use **half-padding**. The **padding** must be **half** of the **kernel size**.

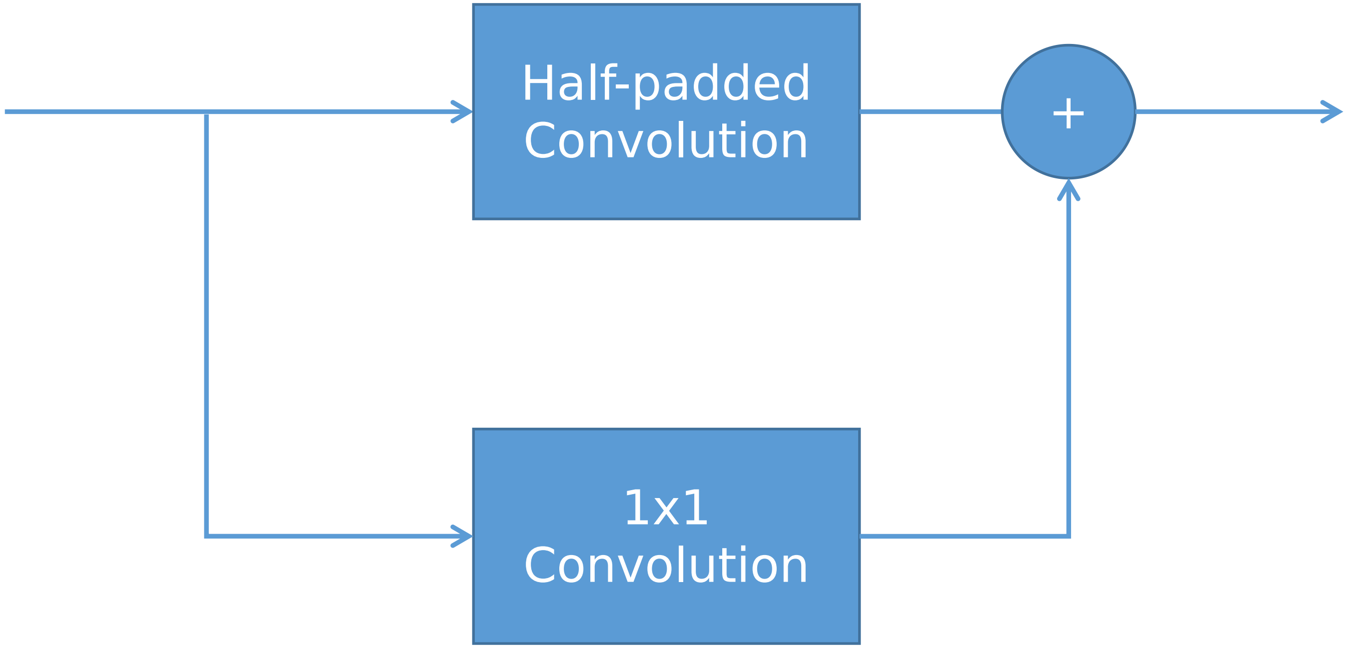
```
import torch

conv1 = torch.nn.Conv2d(1, 16, (3, 3), padding=1)
conv2 = torch.nn.Conv2d(1, 16, (5, 5), padding=2)
conv3 = torch.nn.Conv2d(1, 16, (7, 7), padding=3)

batch, _ = next(iter(dl))

print(batch.shape)
print(conv1(batch).shape)
print(conv2(batch).shape)
print(conv3(batch).shape)
```

```
torch.Size([8, 1, 28, 28])
torch.Size([8, 16, 28, 28])
torch.Size([8, 16, 28, 28])
torch.Size([8, 16, 28, 28])
```

I still cant sum the output with the input because the number of channels is different

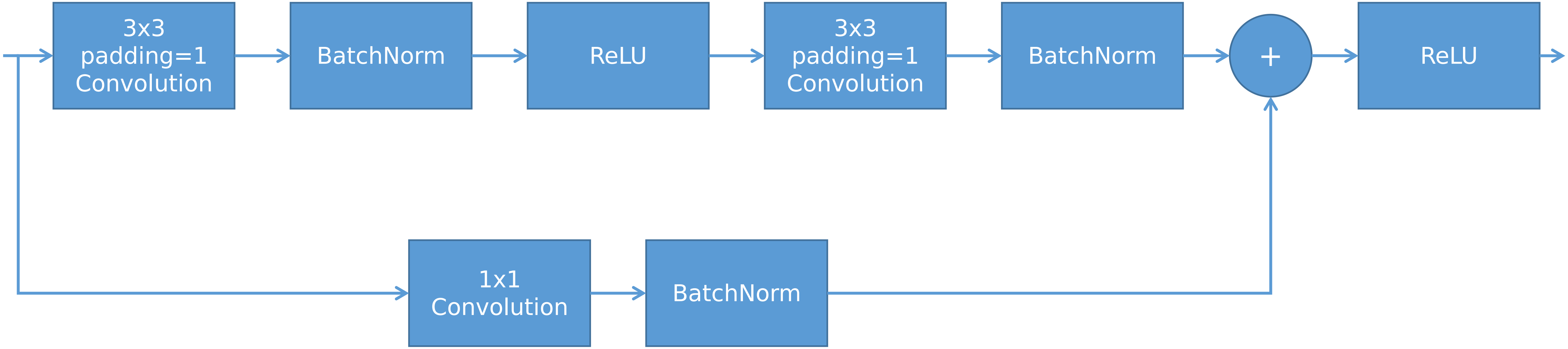
Residual Networks (ResNet): down-scaling

If you want to increment the number of channel in a residual block you have to pad the missing values (for example zero-padding) or add a 1x1 convolution



Residual Networks (ResNet): The “standard” Residual Block

Every network with a skip-connection is technically a ResNet.
Usually a “standard” residual block looks like this.



The BatchNorm2d layer

BatchNorm2d is a technique to **normalize** the **input** of each layer.

It **operates** on **batches** of data and **normalizes** the **input** by subtracting the **mean** and dividing by the **standard deviation**.

This **normalization** helps in **stabilizing** and **accelerating** the **training** process.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

torchvision.datasets

torchvision.datasets is a **module** that provides a **collection** of popular **datasets** for **computer vision** tasks.

It **simplifies** the process of **loading** and **preprocessing** these **datasets**, making them **readily available** for researchers and practitioners working on image-related tasks.

Image classification

`Caltech101`(root[, target_type, transform, ...])	Caltech 101 Dataset.
`Caltech256`(root[, transform, ...])	Caltech 256 Dataset.
`CelebA`(root[, split, target_type, ...])	Large-scale CelebFaces Attributes (CelebA)

The MNIST Dataset

The **MNIST dataset** is a **widely used** collection of **handwritten digits.**

It consists of **grayscale images**, each **depicting** a **single digit** (0 through 9).

The **dataset** contains **60,000 training images** and **10,000 testing images,** each of size **28x28** pixels.

```
train_mnist = torchvision.datasets.MNIST(
    "./data",
    train=True,
    download=True,
    transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.1307,), (0.3081,))
    ])
)
```

Exercise 0

Train a **three-layers MLP** to solve the **MNIST classification** problem

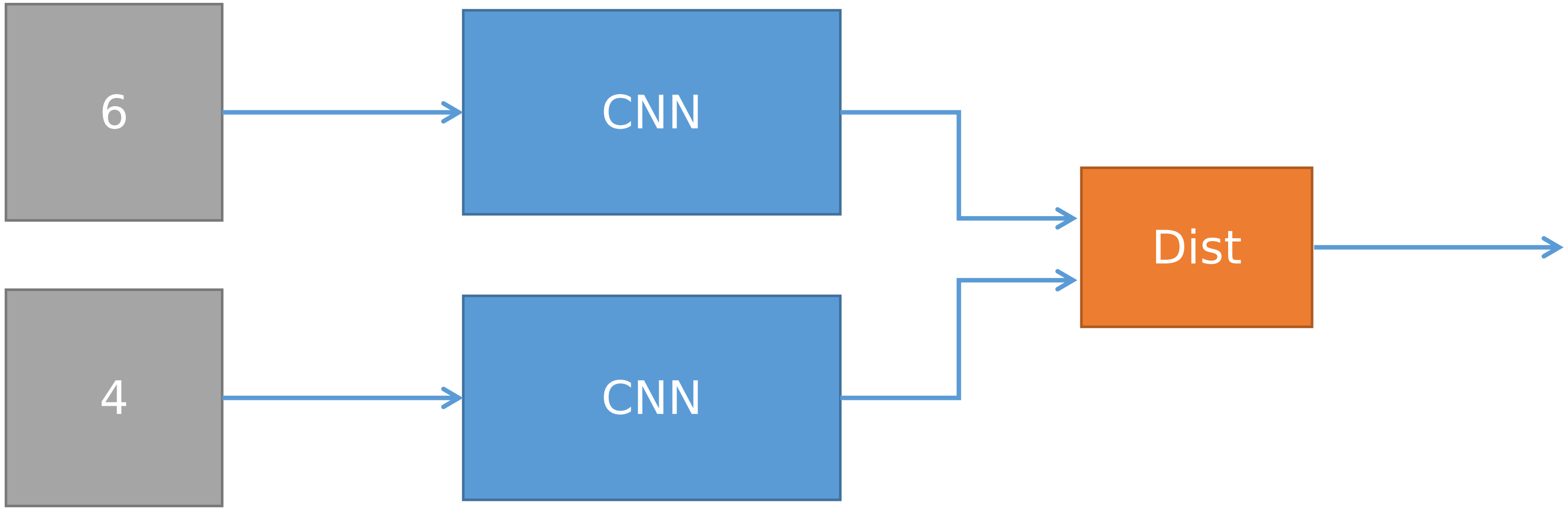
- Each **MNIST** image is **1x28x28**, reshape it to **768**
- The MLP should have **300 hidden neurons**
- The MLP should have **10 output neurons**, one per each class
- Use the **LeakyReLU** activation function
- Use the **CrossEntropyLoss** loss function

Bonus: plot a 2D PCA of the feature vectors

Exercise 1

Train a **CNN** without residual blocks

- The CNN given an **input MNIST** image should **output** a **feature vector**
- Train the network in a way that **euclidean distance** between **feature vectors** of different classes is the **same** as the **difference between the classes**



Train this to be
6 - 2 = 4

Exercise 2

Train a **ResNet**

- The ResNet given an **input MNIST** image should **output** a **image** with the **same shape**
- The **output** image should **depict** the **successive number** w.r.t. the **input** one