# Binary and multi-class image classification

DEEP LEARNING FOR IMAGES WITH PYTORCH

**Michal Oleszak**
Machine Learning Engineer

datacamp

# What will we learn with PyTorch?

Image Classification


Cat


Dog

# What will we learn with PyTorch?

Image Classification

Cat

Dog

Object Detection

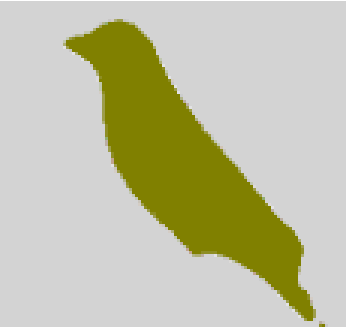
# What will we learn with PyTorch?

Image
Classification

Cat

Dog

Image
Segmention



Object Detection

# What will we learn with PyTorch?

Image Classification

Cat

Dog

Image Segmention

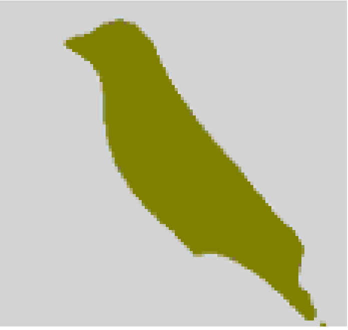
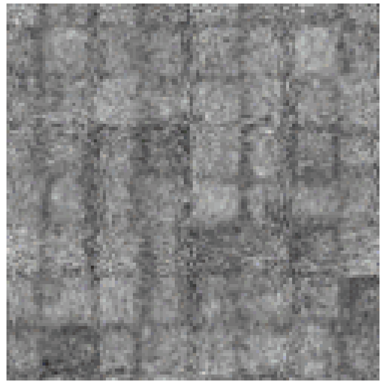


Image Generation



Object Detection

# Prerequisites

- Convolutional Neural Networks
- Model training in PyTorch
- Prerequisite course: **Intermediate Deep Learning with PyTorch**

# PyTorch library

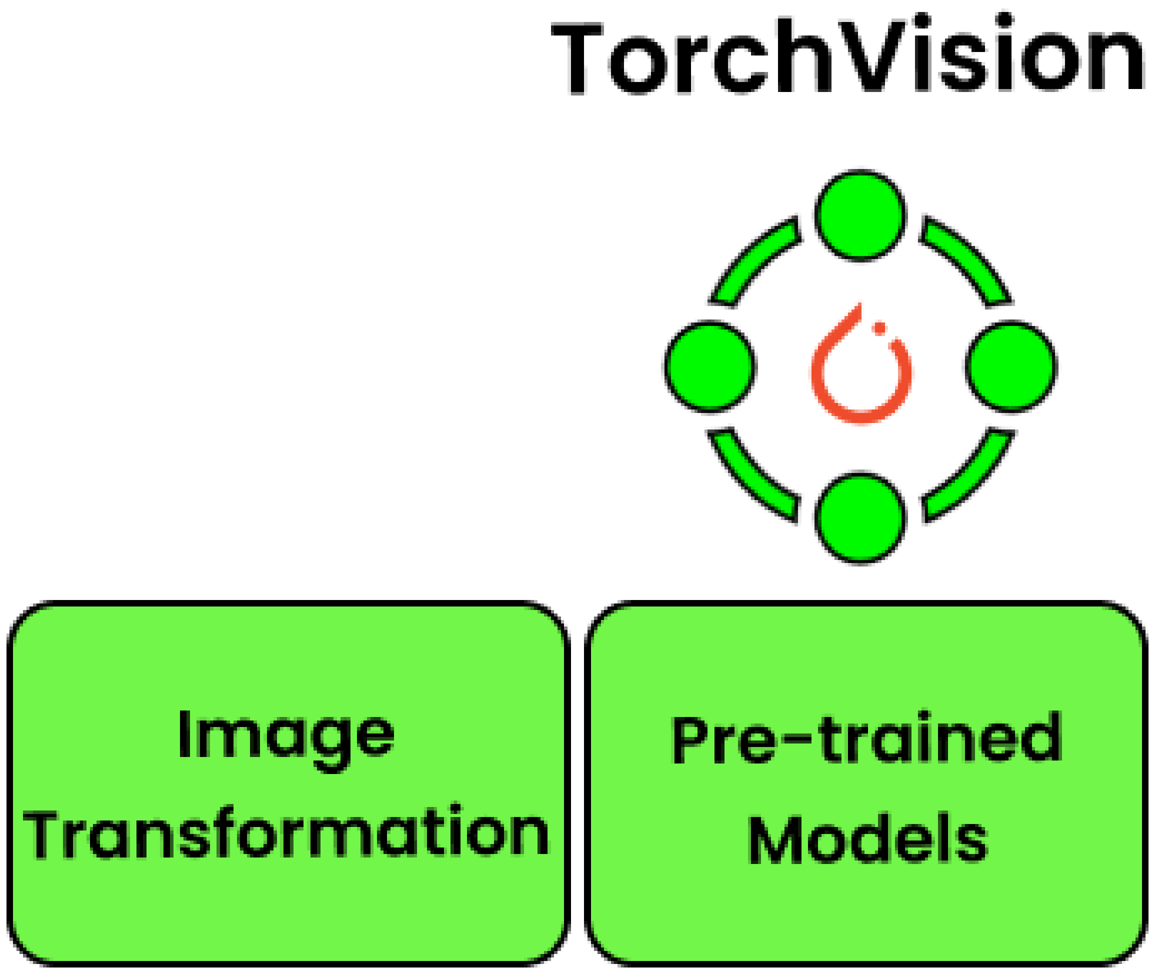
TorchVision

# PyTorch library

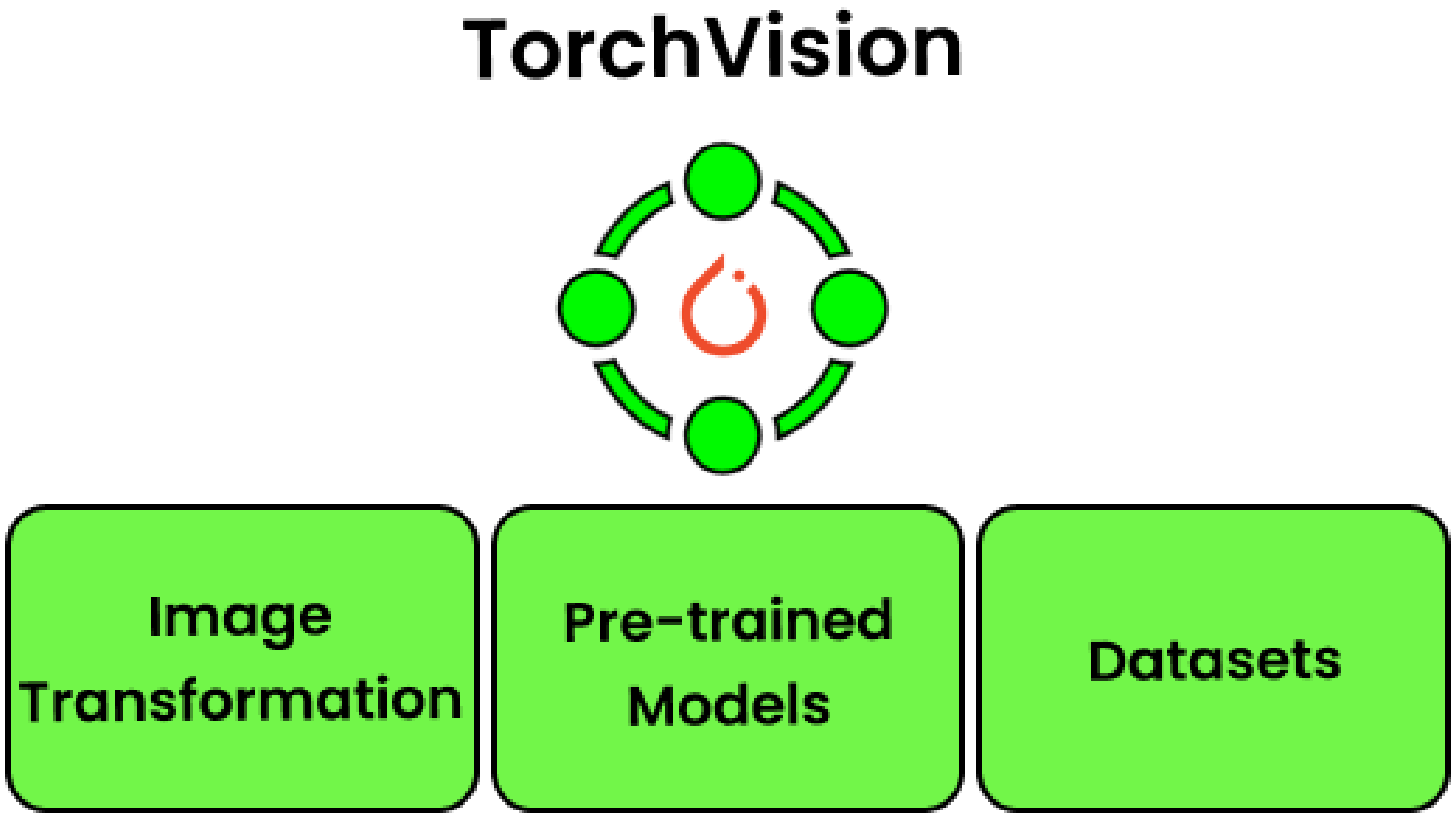
TorchVision

Image Transformation

# PyTorch library

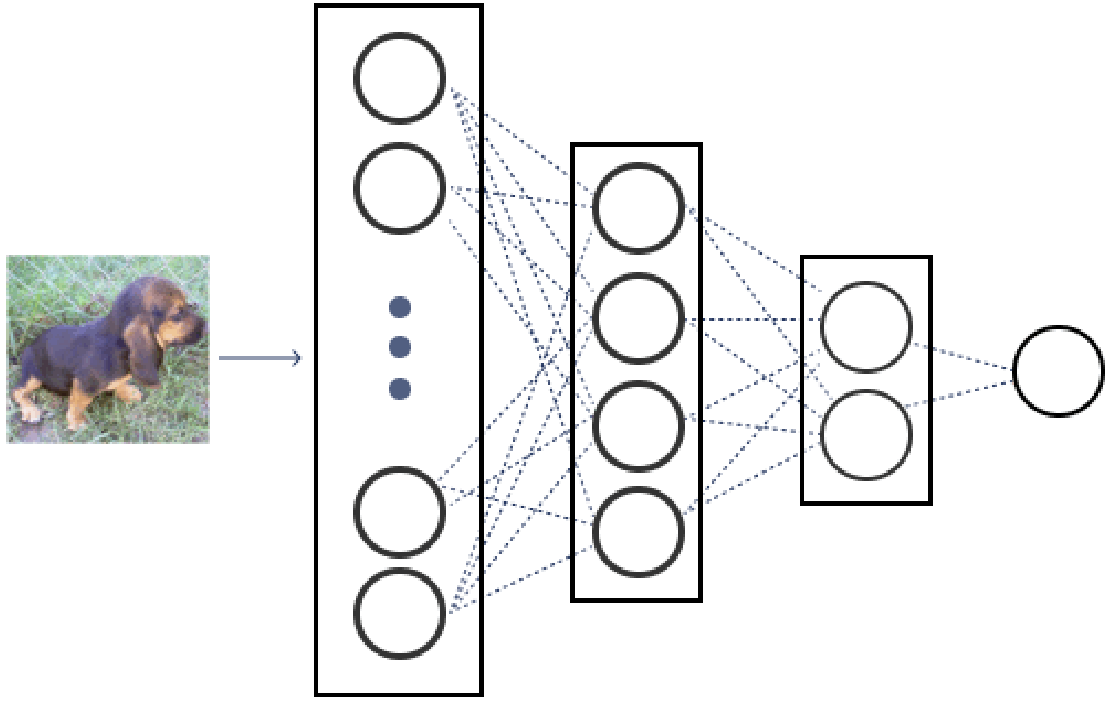


# PyTorch library



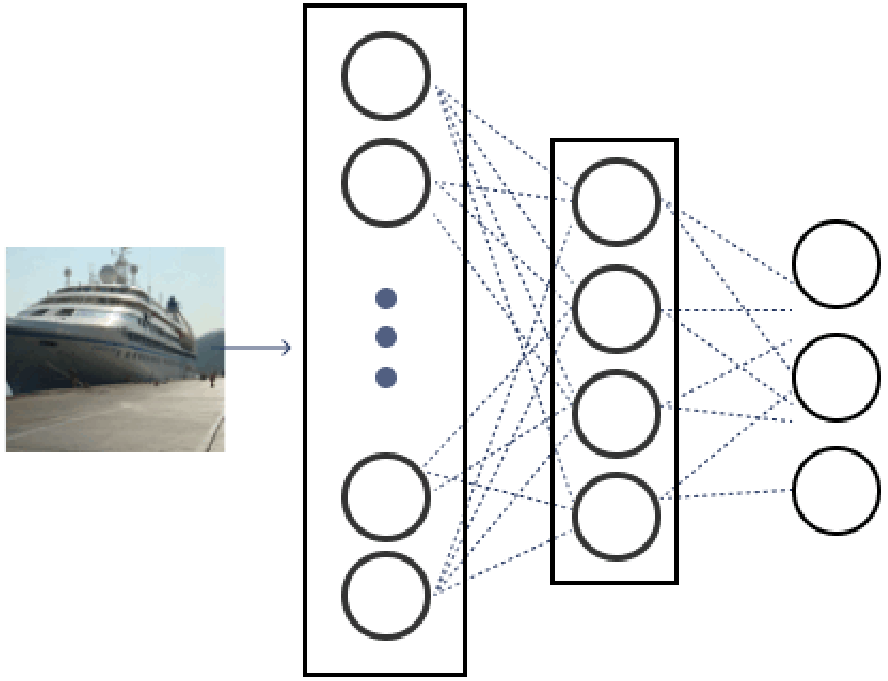
# Image classification

**Binary** classification



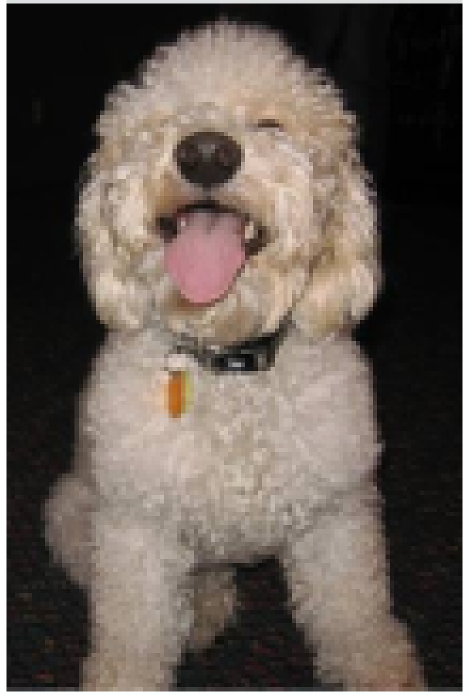
- Two distinct classes (cats, dogs)
- Activation function: Sigmoid

**Multi-class** classification



- Multiple classes (boat, train, car)
- Activation function: Softmax
- Highest probability is the prediction

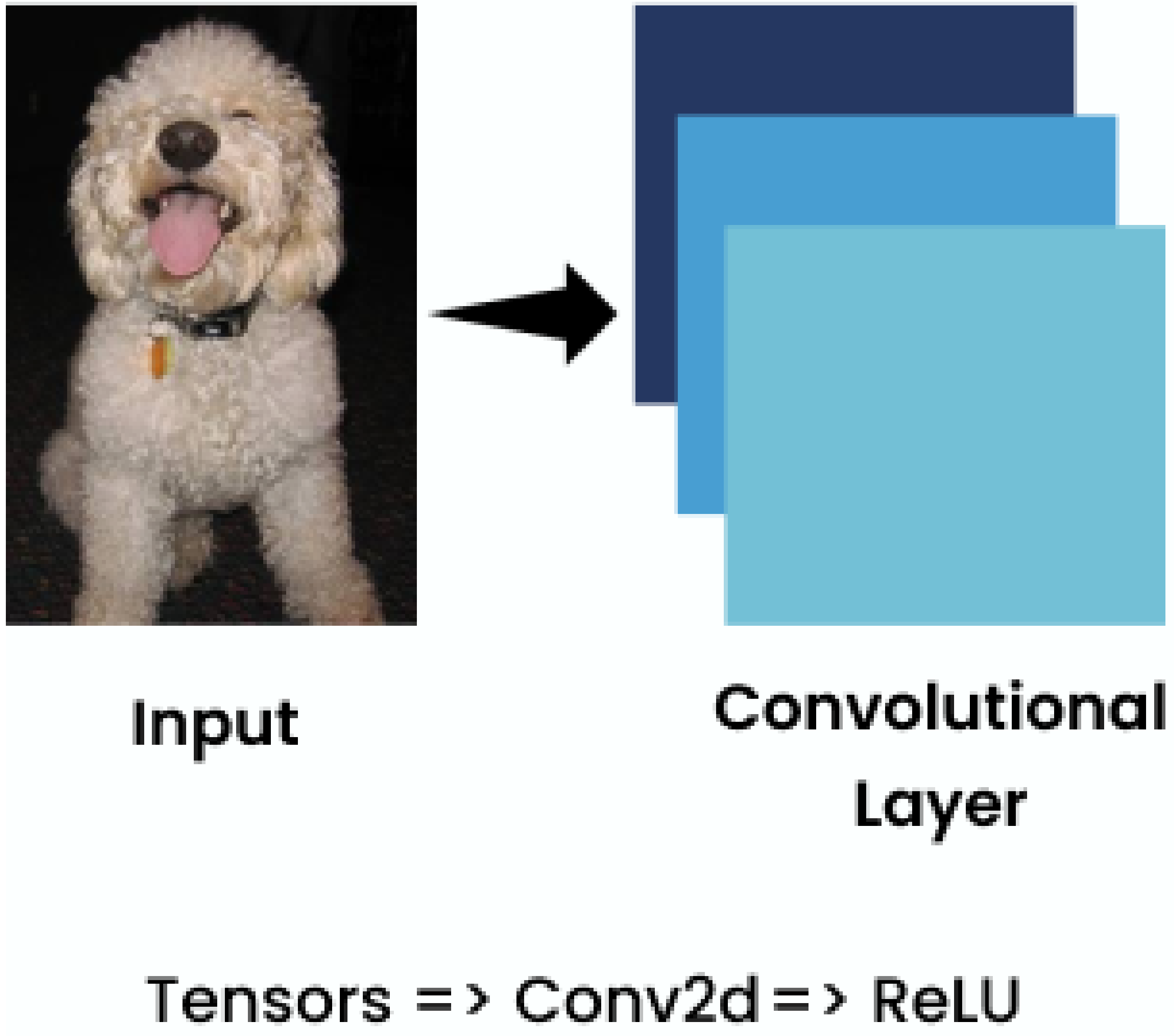
# Convolutional Neural Network model



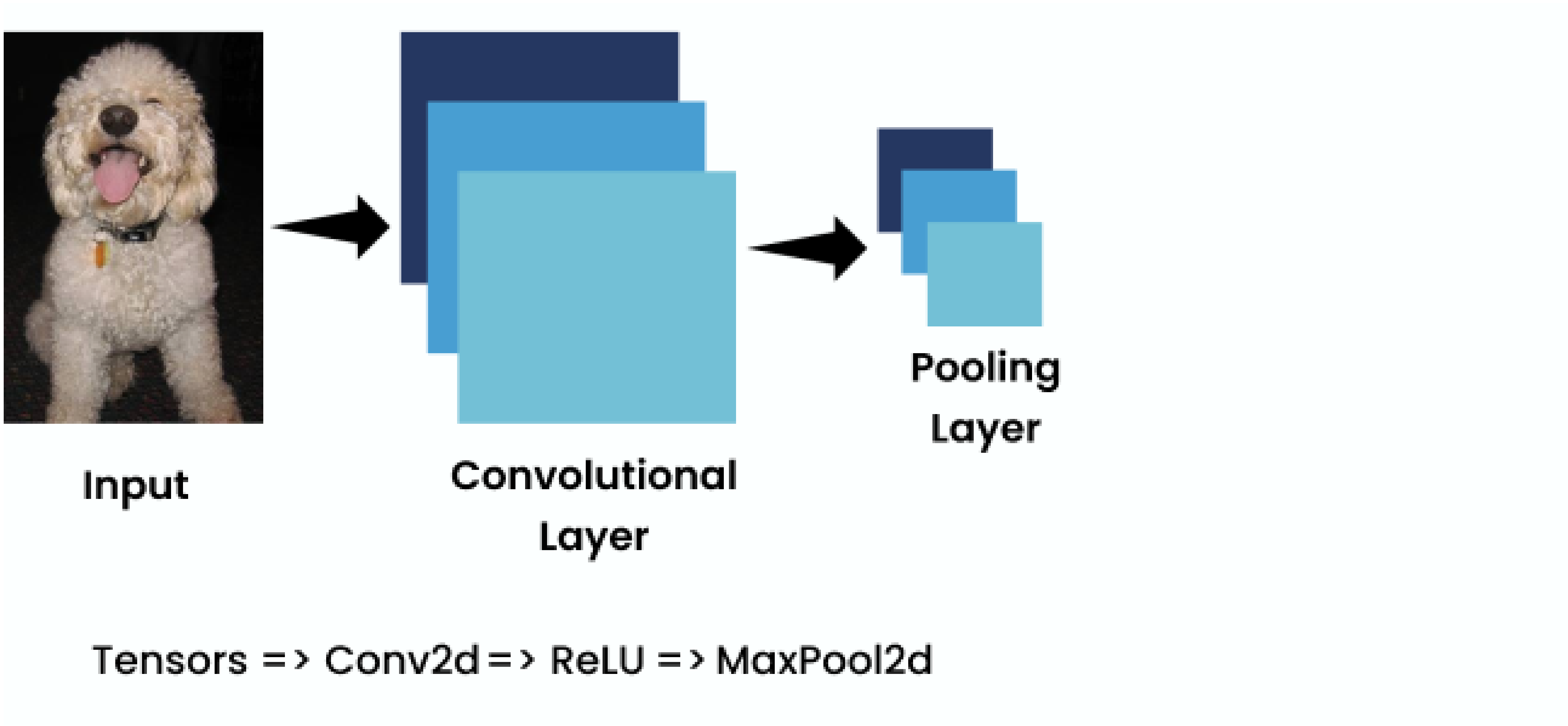
Input

Tensors

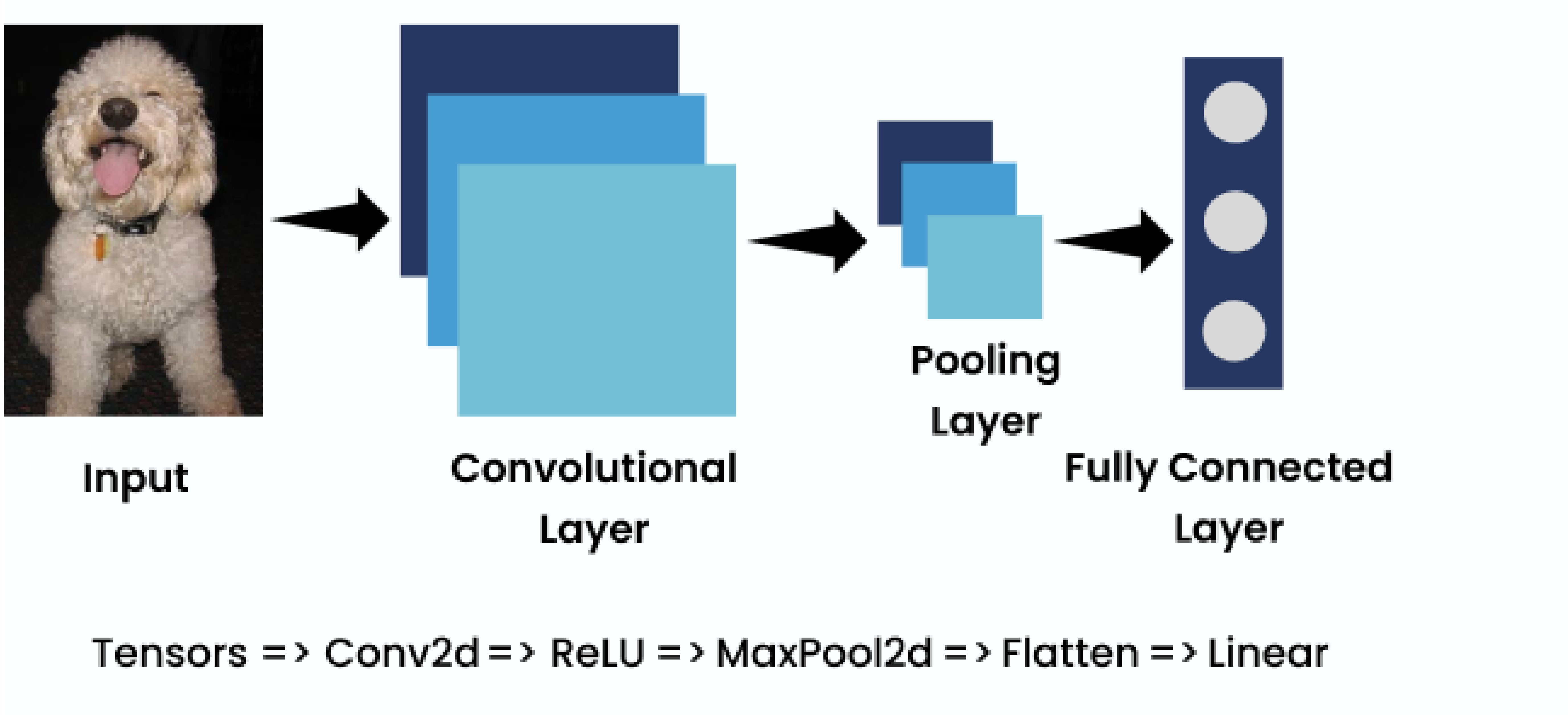
# Convolutional Neural Network model



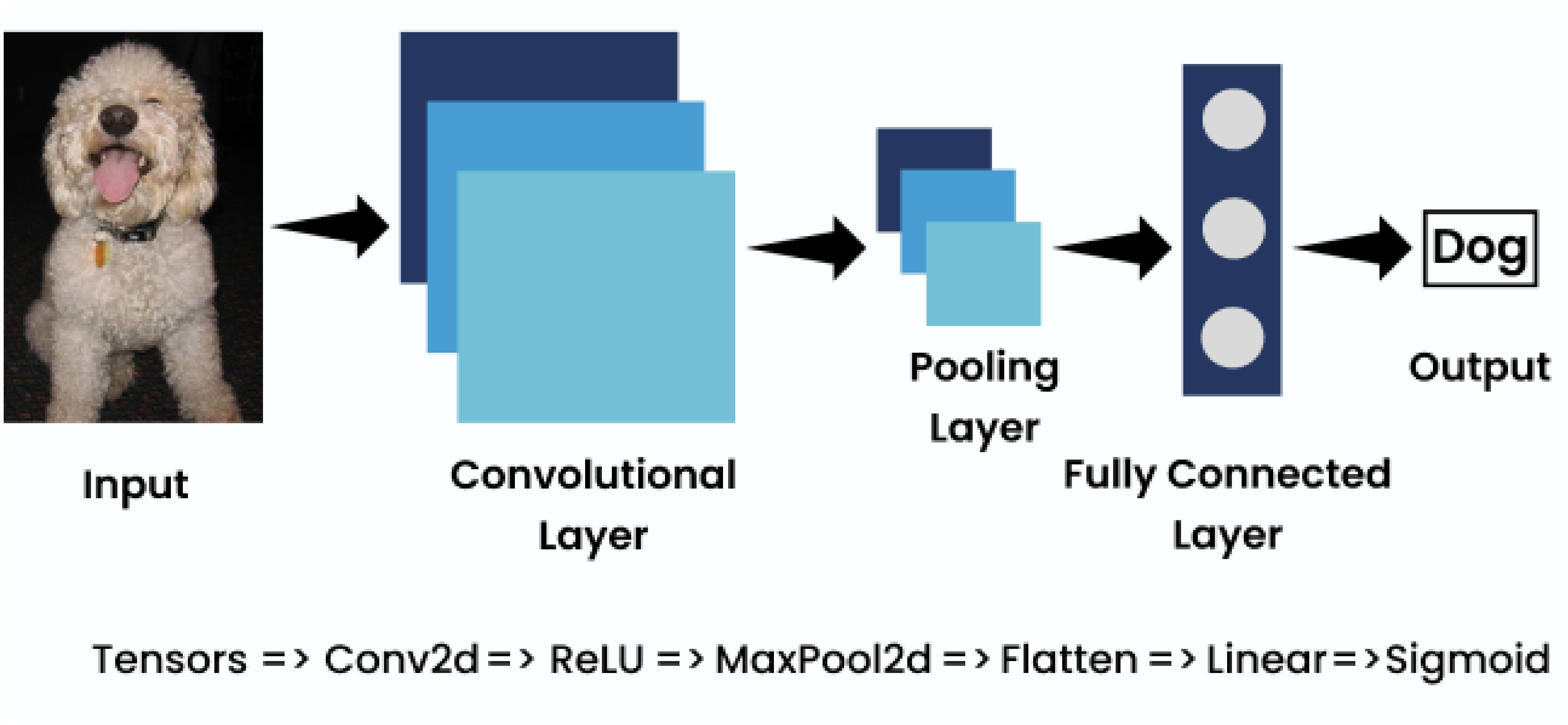
# Convolutional Neural Network model



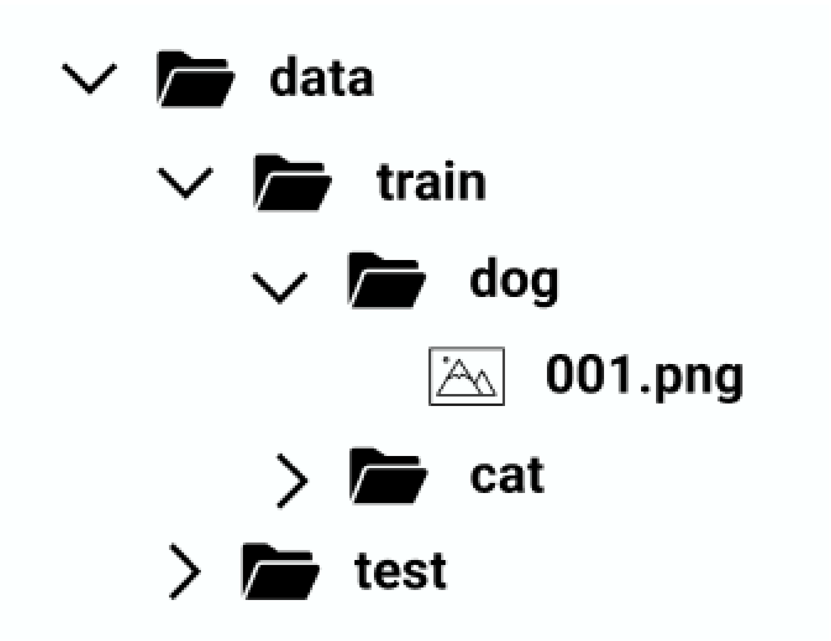
# Convolutional Neural Network model



# Convolutional Neural Network model



# Datasets: class labels




```python
from torchvision import datasets
import torchvision.transforms as transforms

train_dir = '/data/train'
train_dataset = ImageFolder(root=train_dir,
        transform=transforms.ToTensor())
```

```python
classes = train_dataset.classes
print(classes)
```

```
['cat', 'dog']
```

```python
print(train_dataset.class_to_idx)
```

```
{'cat': 0, 'dog': 1}
```

# Binary image classification: convolutional layer

- `Conv2d()` :
  - Input: 3 RGB channels (red, green, blue)
  - Output: 16 channels
  - Kernel: 3 x 3 matrix
  - Stride = 1: the kernel moves 1 step
  - Padding = 1: 1 pixel around the border
- `ReLU()` :
  - A non-linear activation function
- `MaxPool2d()` :
  - Kernel: 2×2
  - Stride: 2 steps

```python
class BinaryCNN(nn.Module):
    def __init__(self):
        super(BinaryCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16,
                kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=2,
                             stride=2)




   def forward(self, x):




        return x
```

# Binary image classification: fully connected layer

- `Flatten()`:
  - Tensors flattened into 1-D vector
- `Linear()`:
  - Input: feature maps x height x width
  - Output: a single class
- `Sigmoid()`:
  - [0,1]

```
class BinaryCNN(nn.Module):
    def __init__(self):
        super(BinaryCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16,
                kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=2,
                     stride=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(16 * 112 * 112, 1)
        self.sigmoid = nn.Sigmoid()
   def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.fc1(self.flatten(x))
        x = self.sigmoid(x)]
        return x
```

# Multi-class image classification with CNN

```
class MultiClassCNN(nn.Module):
    def __init__(self, num_classes):
        super(MultiClassCNN, self).__init__()
        ...
        self.fc = nn.Linear(16 * 112 * 112, num_classes)
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        ...
        x = self.softmax(x)
        return x
```

# Let's practice!

DEEP LEARNING FOR IMAGES WITH PYTORCH

# Convolutional layers for images

DEEP LEARNING FOR IMAGES WITH PYTORCH

**Michal Oleszak**
Machine Learning Engineer

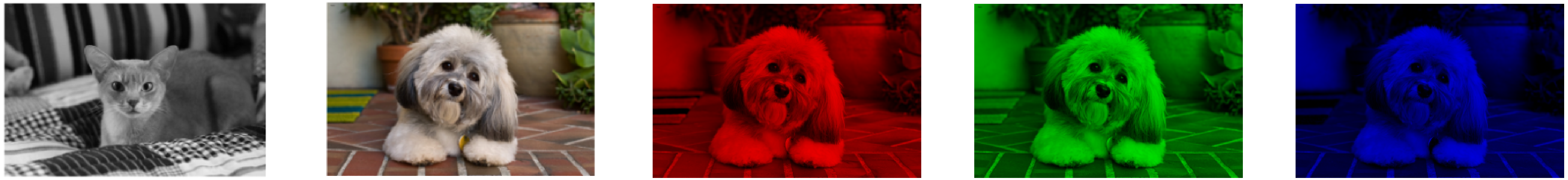
datacamp

# Convolutional layers for images

- Apply convolutional layers to image data
- Access and add convolutional layers
- Create convolutional blocks

- Used to adapt models to a specific task

# Conv2d: input channels

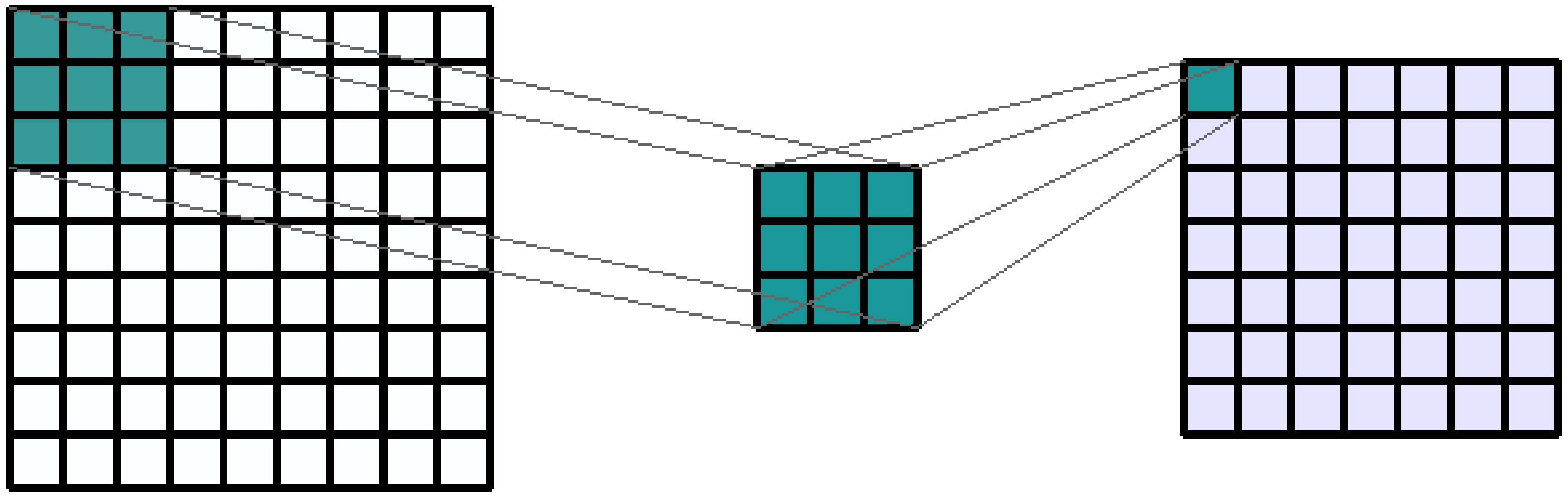


- Grayscale image: `in_channels=1`
- RGB image (red, green, blue): `in_channels=3`
- Transparency includes alpha channel: `in_channels=4`

```
from torchvision.transforms import functional
image = PIL.Image.open("dog.png")
num_channels = functional.get_image_num_channels(image)
print("Number of channels: ", num_channels)
```

```
Number of channels: 3
```

# Conv2d: kernel

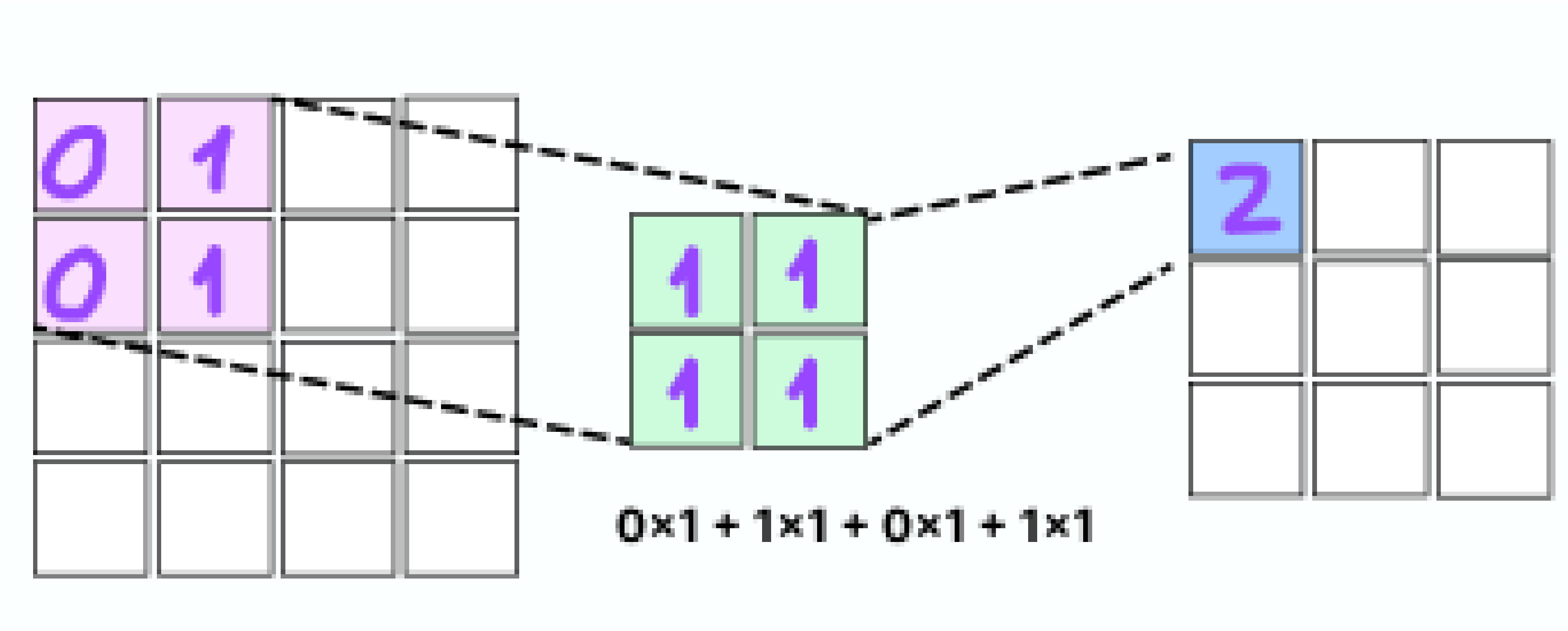


Input tensor Kernel Output tensor (feature map)

- Kernel (colored in green) moves from left to right, top to bottom of the image[1]

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

# Kernel sizes

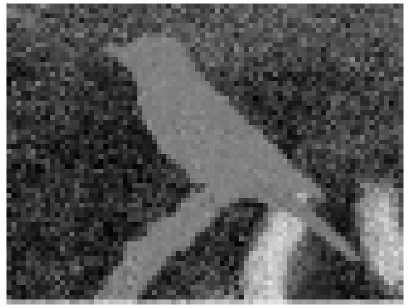



- The most common kernel sizes: 3×3 ( `Conv2d` ) and 2×2 ( `MaxPool2d` )
- Convolution is a dot product of the kernel (green) and the image region (pink)
- The sum of the dot product creates a feature map (blue)

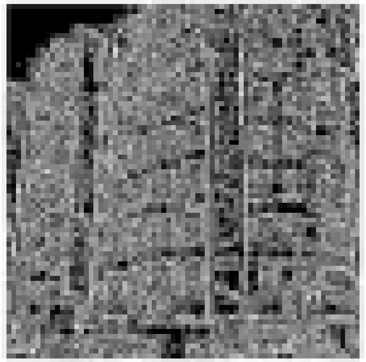
# Kernel is a filter

- Capture image patterns

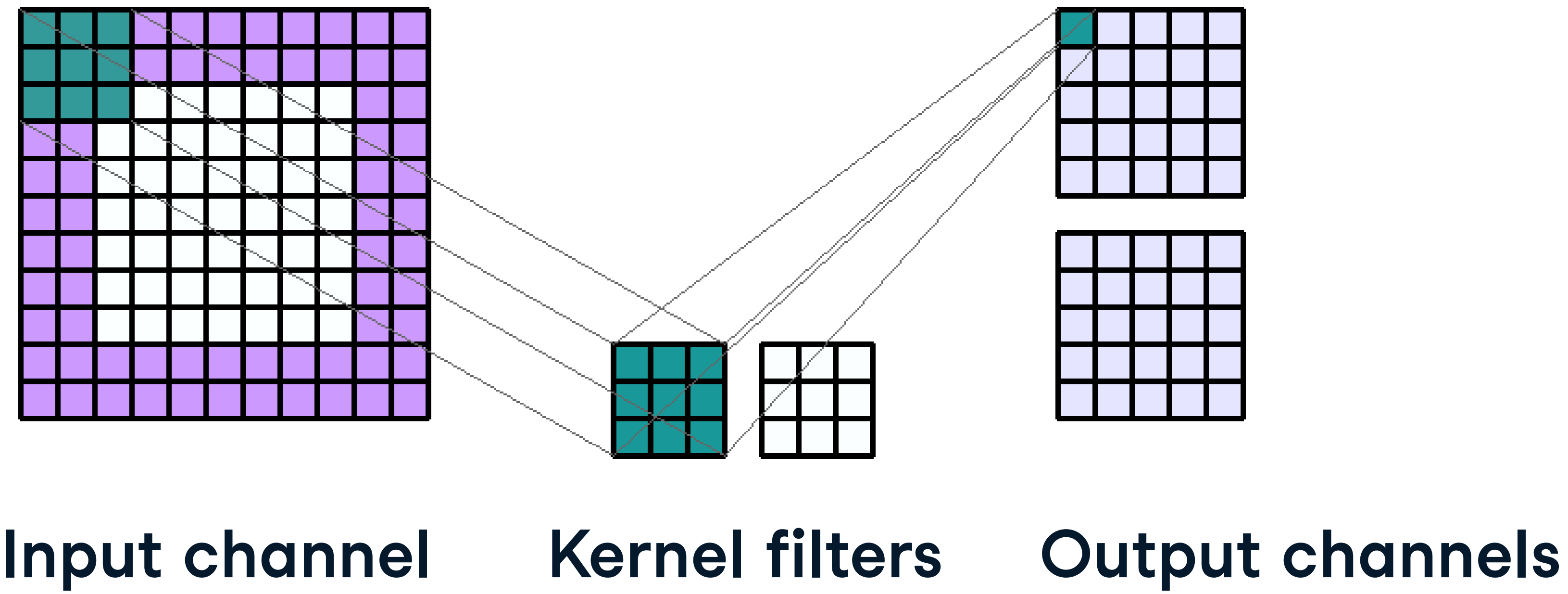
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$



# Conv2d: output channels




- The number of output channels determines how many filters are applied
- Each output channel corresponds to a distinct filter
- A higher number of output channels allows the layer to learn more complex features
- Output channel numbers are commonly chosen as powers of 2 (16, 32, 64, 128)
  - It simplifies the process of combining and dividing channels in subsequent layers

# Adding convolutional layers

```
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
```

```
conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
```

```
model = Net()
model.add_module('conv2', conv2)
```

# Accessing convolutional layers

```
print(model)
```

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
```

```
model.conv2
```

```
Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

# Creating convolutional blocks

- Stacking convolutional layers in a block with `nn.Sequential()`

```
class BinaryImageClassification(nn.Module):
    def __init__(self):
        super(BinaryImageClassification, self).__init__()
        self.conv_block = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
     def forward(self, x):
         x = self.conv_block(x)
```

# Let's practice!

DEEP LEARNING FOR IMAGES WITH PYTORCH

# Working with pre-trained models

DEEP LEARNING FOR IMAGES WITH PYTORCH

**Michal Oleszak**
Machine Learning Engineer

datacamp

# Leveraging pre-trained models

- Training models from scratch:
  - Long process
  - Requires lots of data
- **Pre-trained models** - models already trained on a task
  - Directly reusable on a new task
  - Require adjustment to the new task (transfer learning)
- Steps to leveraging pre-trained models:
  - Saving & loading models locally
  - Downloading `torchvision` models

# Saving a complete PyTorch model

- `torch.save()`
- Model extension: `.pt` or `.pth`
- Save model weights with `.state_dict()`

```
torch.save(model.state_dict(), "BinaryCNN.pth")
```

# Loading PyTorch models

- Instantiate a new model

```
new_model = BinaryCNN()
```

- Load saved parameters

```
new_model.load_state_dict(torch.load('BinaryCNN.pth'))
```

# Downloading torchvision models

```
from torchvision.models import (
    resnet18, ResNet18_Weights
)

weights = ResNet18_Weights.DEFAULT
model = resnet18(weights=weights)
transforms = weights.transforms()
```

- Import `resnet` architecture and weights
- Extract weights
- Instantiate a model passing it weights
- Store required data transforms

# Prepare new input images

```python
from PIL import Image

image = Image.open("cat013.jpg")
image_tensor = transform(image)
image_reshaped = image_tensors.unsqueeze(0)
```

- Load image
- Transform image
- Reshape image

# Generating a new prediction

```
model.eval()

with torch.no_grad():
    pred = model(image_reshaped).squeeze(0)

pred_cls = pred.softmax(0)
cls_id = pred_cls.argmax().item()
cls_name = weights.meta["categories"][cls_id]

print(cls_name)
```

```
Egyptian cat
```

- Evaluation mode for inference
- Disable gradients
- Pass image to model and remove batch dimension
- Apply softmax
- Select the highest-probability class and extract its index
- Map class index to label
- Print class label

# Let's practice

DEEP LEARNING FOR IMAGES WITH PYTORCH