MACHINE LEARNING

CONVOLUTIONS

Convolutions

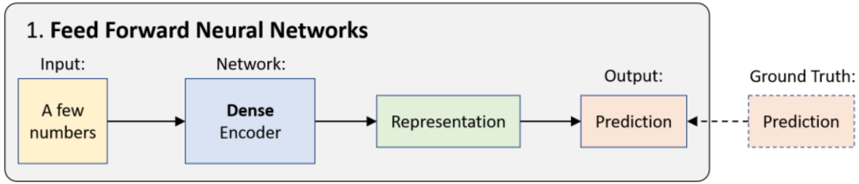
Understanding convolution

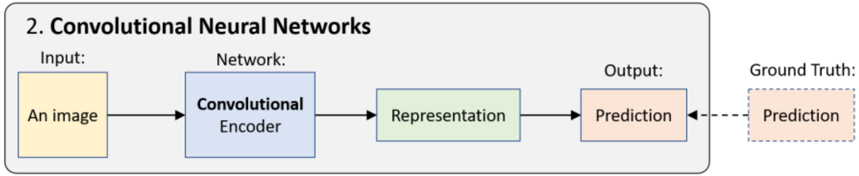
Creating custom nn.Module subclasses

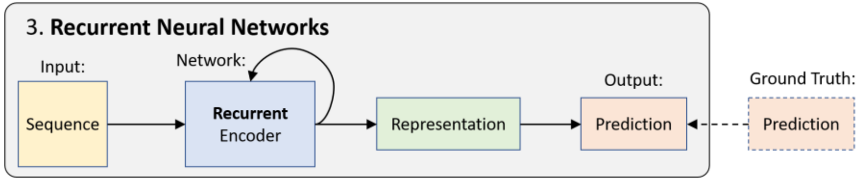
Building a convolutional neural network

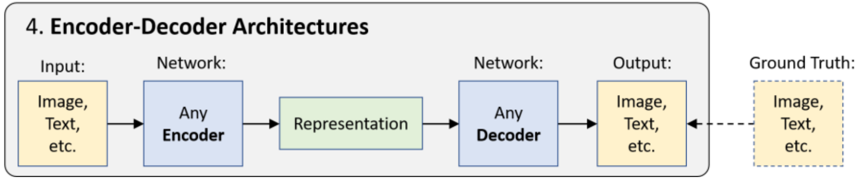
Overview

Supervised Learning

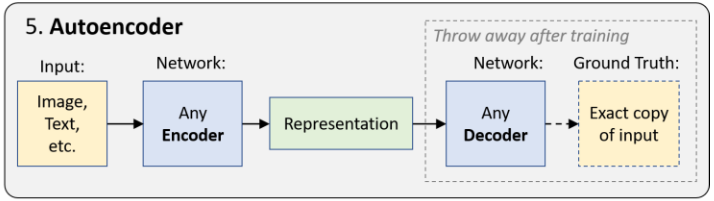


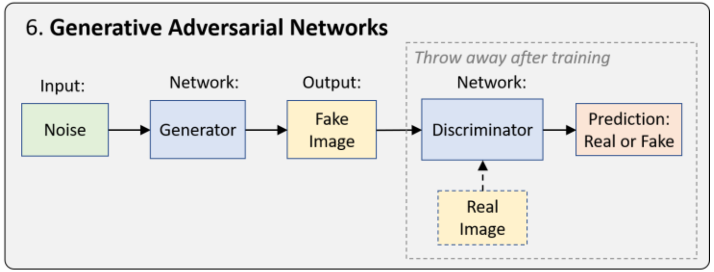




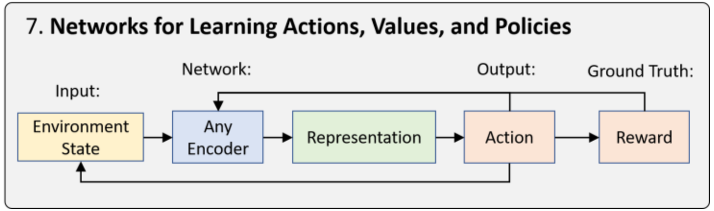


Unsupervised Learning





Reinforcement Learning



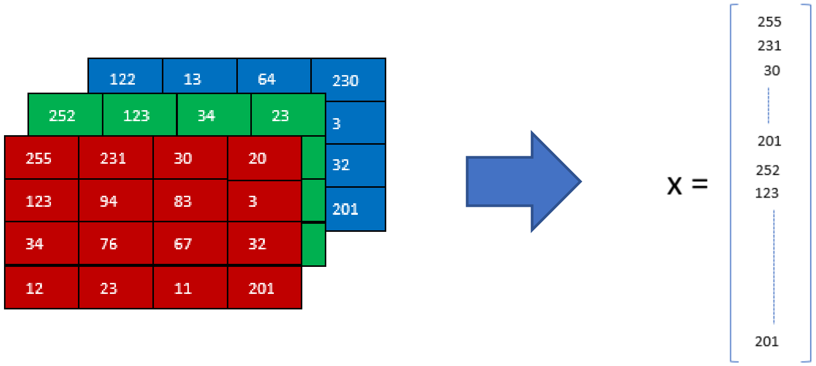
CONVOLUTIONS

- *Understanding convolutions*

Spatial structure of data

With the nn.Linear design, we have lost the spatial structure of the data:

- Consider the effect of the translation of an image of an airplane in the classifier
- Consider the relevance of neighboring pixels when interpreting a pixel as forming part of wing



No matter what the spatial distribution is, the x_train vector will be the same

Convolution

Objective, to create:

- a translation invariant model, that
- retains spatial relationship between pixels (locality)

Convolution, or more precisely, discrete is defined for a 2D image as the scalar product of a weight matrix, the kernel, with every neighborhood in the input.

Universidad Francisco de Vitoria
UFV Madrid

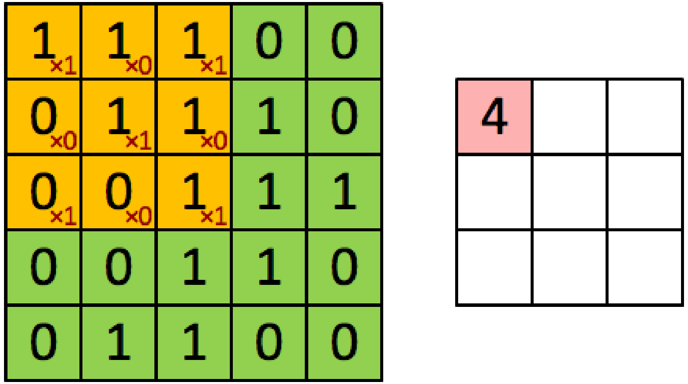
The Kernel

Kernel/Filter K: the element involved in carrying out the convolution operation in the first part of a Convolutional Layer

We select K as a 3x3x1 matrix.

```
Kernel/Filter, K =

1  0  1
0  1  0
1  0  1
```

Convolution



Image

Convolved Feature

$$\begin{bmatrix}1 & 1 & 1\\0 & 1 & 1\\0 & 0 & 1\end{bmatrix} * \begin{bmatrix}1 & 0 & 1\\0 & 1 & 0\\1 & 0 & 1\end{bmatrix} =$$

$$1\cdot1+0\cdot0+0\cdot1+1\cdot0+1\cdot1+0\cdot0+1\cdot1+1\cdot0+1\cdot1=4$$

$$\begin{bmatrix}1 & 1 & 0\\1 & 1 & 1\\0 & 1 & 1\end{bmatrix} * \begin{bmatrix}1 & 0 & 1\\0 & 1 & 0\\1 & 0 & 1\end{bmatrix} =$$

$$1\cdot1+1\cdot0+0\cdot1+1\cdot0+1\cdot1+1\cdot0+0\cdot1+1\cdot0+1\cdot1=3$$

$$\begin{bmatrix}1 & 0 & 0\\1 & 1 & 0\\1 & 1 & 1\end{bmatrix} * \begin{bmatrix}1 & 0 & 1\\0 & 1 & 0\\1 & 0 & 1\end{bmatrix} =$$

$$1\cdot1+1\cdot0+1\cdot1+0\cdot0+1\cdot1+1\cdot0+0\cdot1+0\cdot0+1\cdot1=4$$

So, why is this Convolution useful for our purpose?

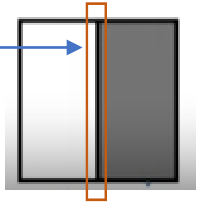
The convolution operators helps the model to define the Edge of the figures

Edge Detection

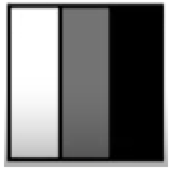
$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

6x6 3x3 4x4

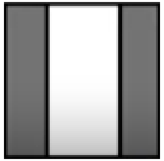
It has found the edge



$*$

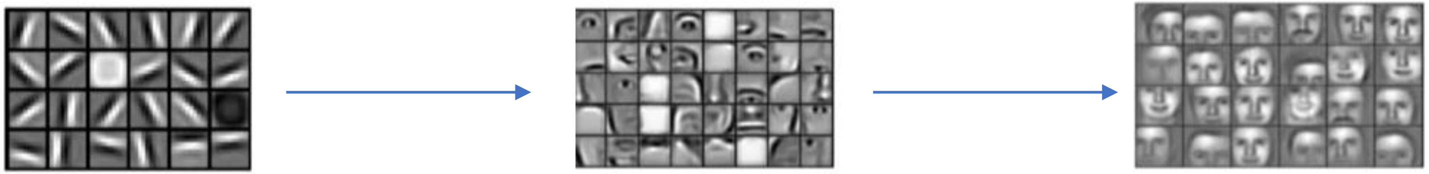


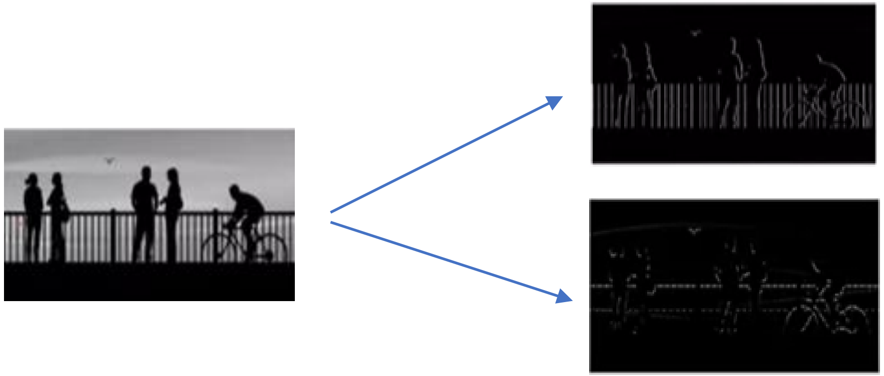
$=$



Edge detection in image recognition

Edge detection is the way used by the living beings to recognize the environments





Vertical Edge

Horizontal Edge

The classic literature of convolutions is plenty of different Kernels for Edge Detection

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

Vertical Edge

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

Horizontal Edge

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

Sobel Filter

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

Scharr Filter

How do we choose the best filter for our convolution (for example cats?

We’ve got a powerful tool now!

Let’s the Machine Learning choose the kernel for us!

$$\begin{bmatrix} 3 & 2 & 10 & 20 & 21 & 10 \\ 1 & 6 & 11 & 35 & 22 & 12 \\ 5 & 8 & 25 & 23 & 25 & 13 \\ 2 & 0 & 56 & 12 & 67 & 14 \\ 4 & 9 & 45 & 11 & 50 & 67 \\ 7 & 10 & 50 & 34 & 23 & 80 \end{bmatrix} * \begin{bmatrix} w_1 \cdot 1 & w_2 \cdot 0 & w_3 \cdot 1 \\ w_4 \cdot 1 & w_5 \cdot 0 & w_6 \cdot 1 \\ w_7 \cdot 1 & w_8 \cdot 0 & w_9 \cdot 1 \end{bmatrix} = \begin{bmatrix} \quad & \quad & \quad \end{bmatrix}$$

Check out the dimensions of the following convolutional operation

$$\begin{bmatrix} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{bmatrix} * \begin{bmatrix} w\cdot 1 & w\cdot 0 & w\cdot 1 \\ w\cdot 1 & w\cdot 0 & w\cdot 1 \\ w\cdot 1 & w\cdot 0 & w\cdot 1 \end{bmatrix} = \begin{bmatrix} y & y & y & y \\ y & y & y & y \\ y & y & y & y \\ y & y & y & y \end{bmatrix}$$

The result shrinks to a n-m+1 dimensión matrix

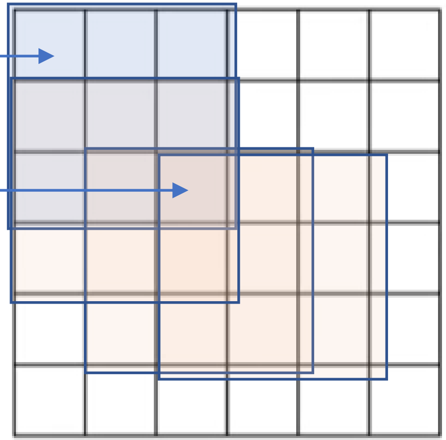
$(6,6)$ $(3,3)$ $(4,4)$

(n,n) (m,m) $(n-m+1, n-m+1)$

Padding

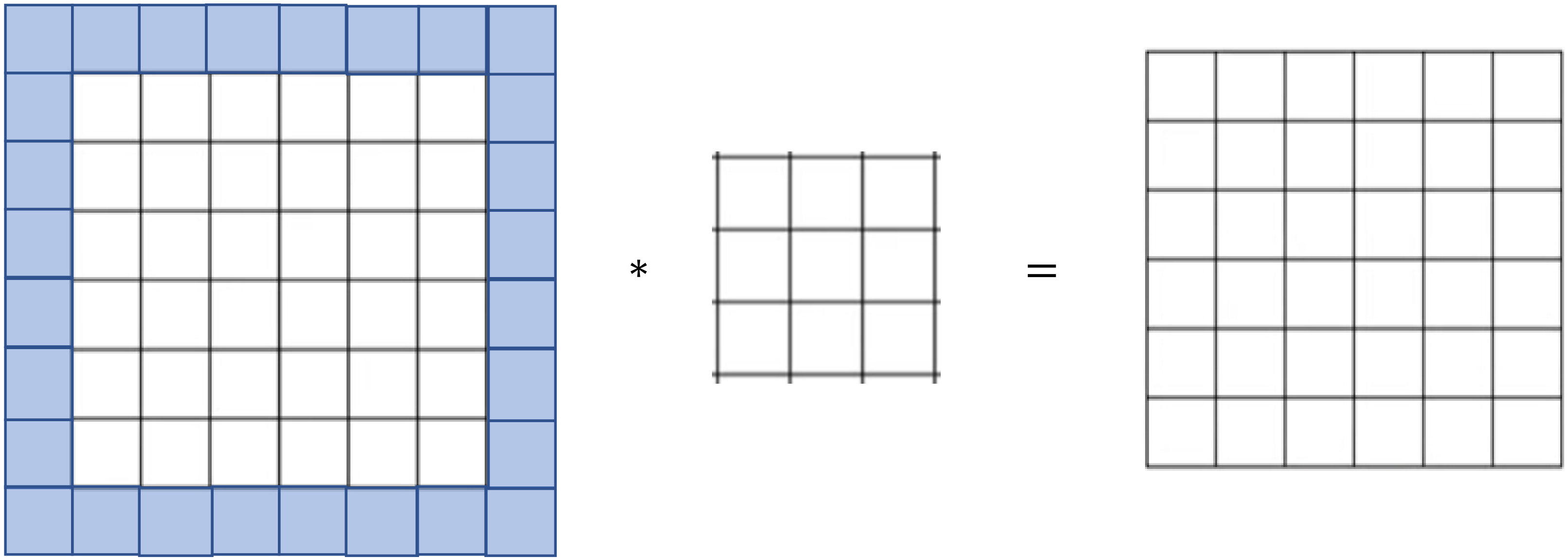
There are several setbacks of the convolution you would like to avoid

- **The result matrix is smaller**
- **The padding takes into account values in the center more than values in the edge**



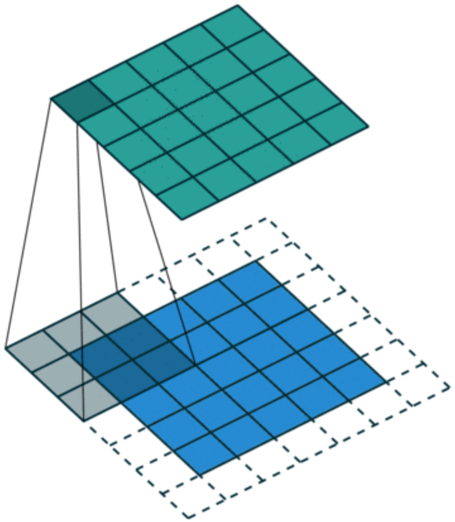
Values in the center of the convoluted matrix are overused

We can augment the dimension of the matrix padding with new columns



(8,8) (3,3) (6,6)

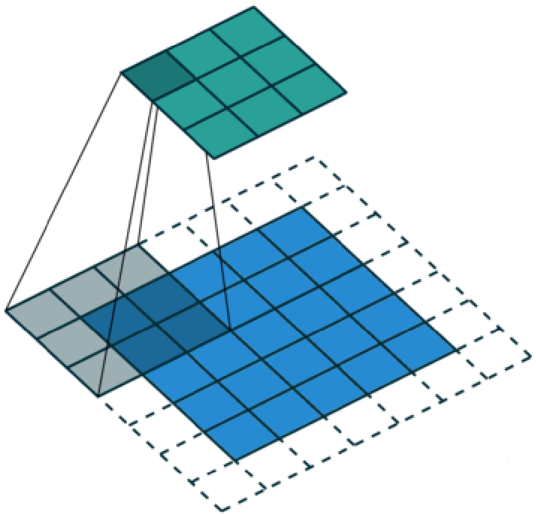
You've managed to keep the dimensions our convoluted matrix



- **Same padding:** When we augment the 5x5x1 image into a 6x6x1 image and then apply the 3x3x1 kernel over it, we find that the convolved matrix turns out to be of dimensions 5x5x1.
- **Valid padding:** if we perform the same operation without padding, the result matrix is smaller

Strided convolution

Instead of moving kernel step-by-step, we can move it further



The chart convolutes a (5,5) matrix padded to (7,7) dimension with a (3,3) and stride=2

Matrix (n, n)

Kernel (f, f)

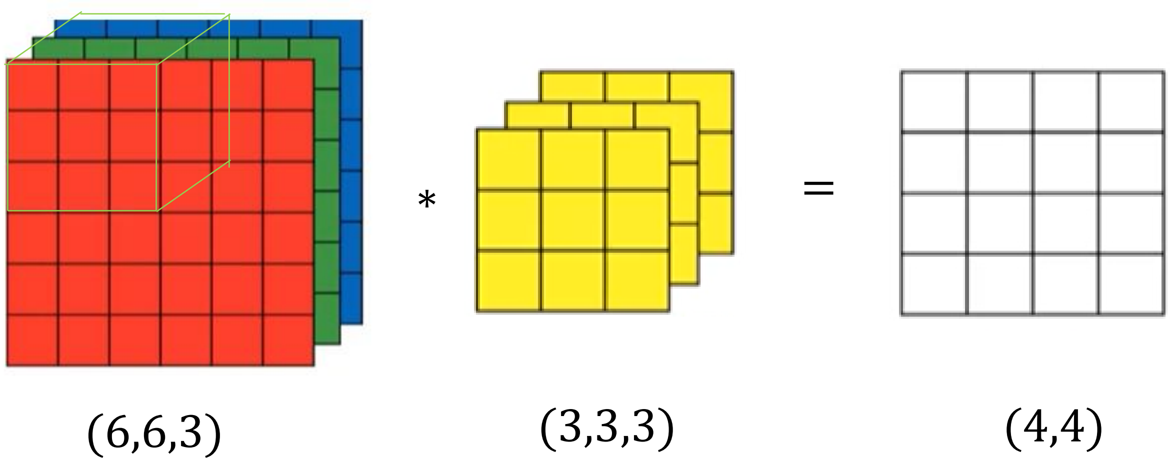
Padding p

Stride s

$$\left(\frac{n+2p-f}{s}+1, \frac{n+2p-f}{s}+1\right)$$

Convolution over 3-dimensional volumes

RGB pictures are represented by (n,m,3) matrices

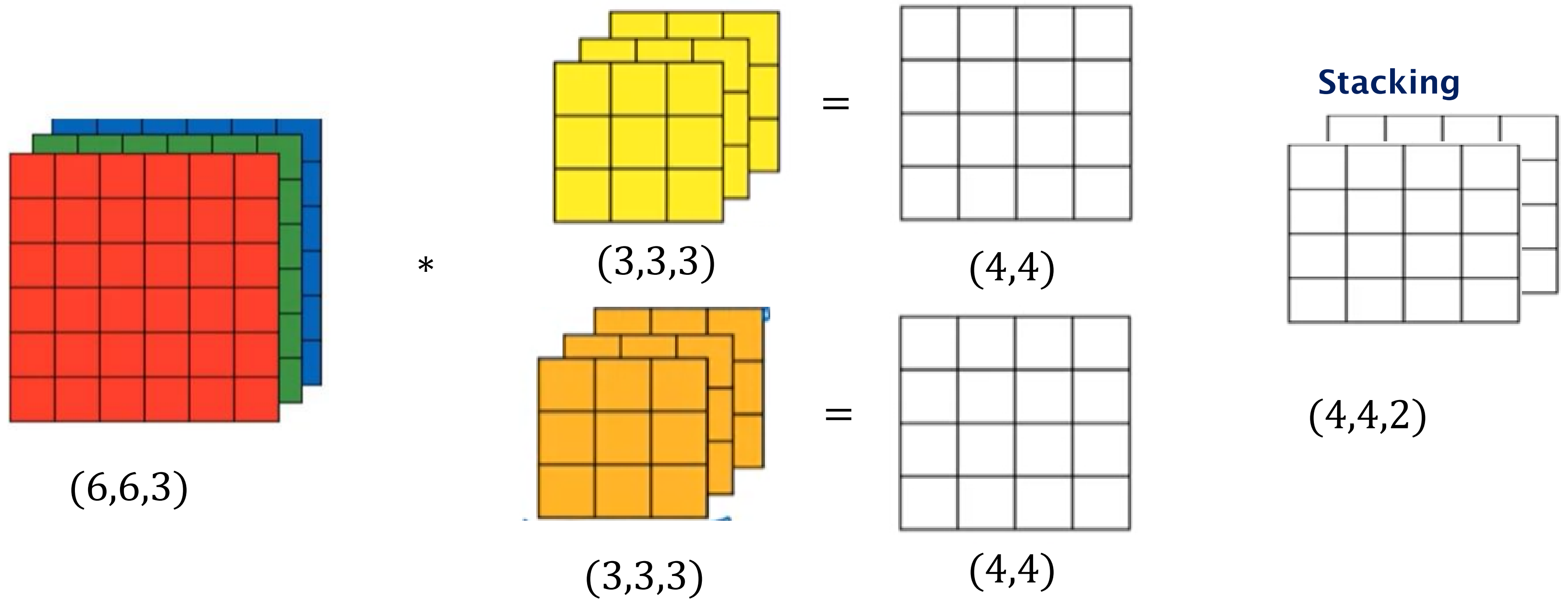


What happened to the third dimensión?

The first value of the convoluted Matrix will be the the sum of the first 9+9+9 values

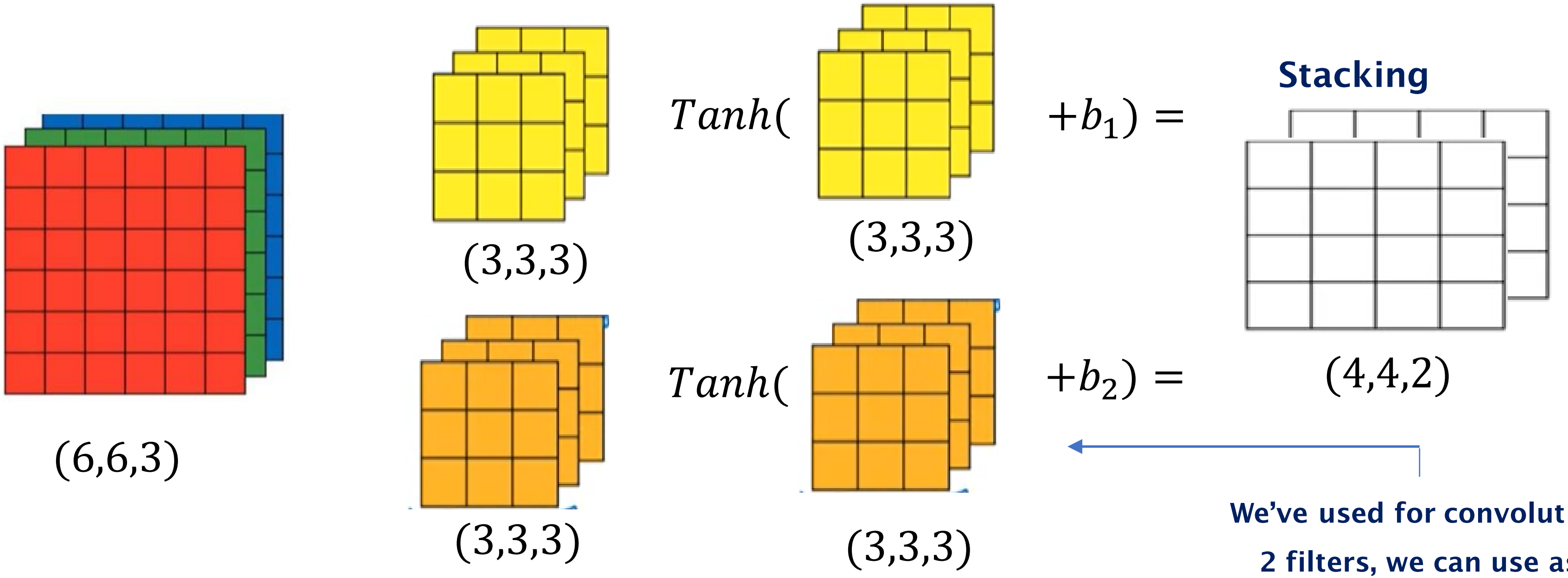
Multiple filters

We can apply as many filters as we want



Creating a Layer of a Convolutional Neural Network

Now we understand how Convolution Works, let's build a Convolution Neural Network



We've used for convolution 2 filters, we can use as many filters as we want

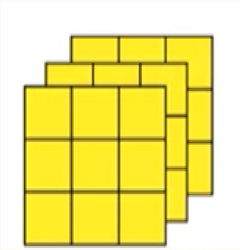
$w^{[1]}$ ← **weights of level 1**

Parameters

Meausuring the size of your Convolutional Neural Network (CNN)

Your team decided to apply a Convolutional Neural Network with 10 filters, each filter will have a 3x3 matrix per color.

How many parameters will your CNN have?



(3,3,3)

$$3 \times 3 \times 3 = 27 + bias = 28$$

$$28\ x\ 10\ filters = 280\ parameters$$

Downsampling

Small kernel size: how do we aggregate the learned micro-structures to “find” relationships at a larger scale?

Strategy in convolutional nets: stacking convolutions layers and, at the same time, downsampling the image between successive convolutions.

Strategies to downsample:

- Average pooling
- Max pooling
- Strided convolution (calculate only every n-th pixel)

Convolutional Neural Network Example

Let's check the notation we're going to use

$$f^{[l]} = filter\ size$$

$$p^{[l]} = padding\ size$$

$$s^{[l]} = stride\ size$$

$$n_c^{[l]} = depth\ of\ filter$$

$$Input: (n_h^{[l-1]}, n_w^{[l-1]}, n_c^{[l-1]})$$

$$Output: (n_h^{[l]}, n_w^{[l]}, n_c^{[l]})$$

$$n_h^{[l]} = [\frac{n_h^{[l-1]} + 2p^{[l]} - n_c^{[l]}}{s^{[l]}} + 1]$$

l= $number\ of\ level, usually\ 3$

Convolutional Neural Network Example

Universidad Francisco de Vitoria
UFV Madrid

(39,39,3)

$$n_h^{[0]} = n_w^{[0]} = 39$$

$$n_c^{[0]} = 3$$

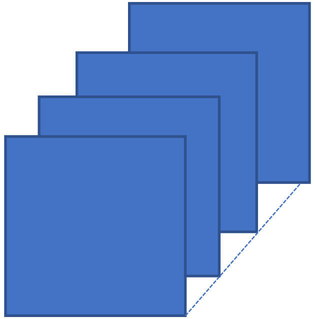
$f_1^{[1]} = 3$

$s^{[1]} = 1$

$p^{[1]} = 0$

$n_c^{[1]} = 3$

10 filters



(37,37,10)

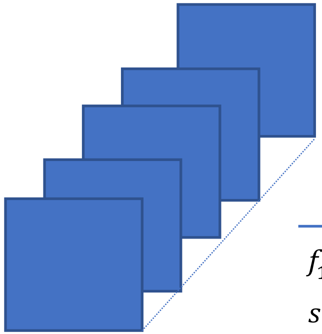
$f_1^{[2]} = 5$

$s^{[2]} = 2$

$p^{[2]} = 0$

$n_c^{[2]} = 10$

20 filters



(17,17,20)

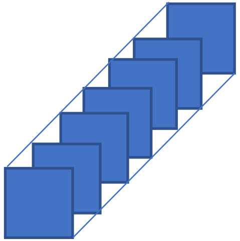
$f_1^{[3]} = 5$

$s^{[3]} = 2$

$p^{[3]} = 0$

$n_c^{[2]} = 20$

40 filters

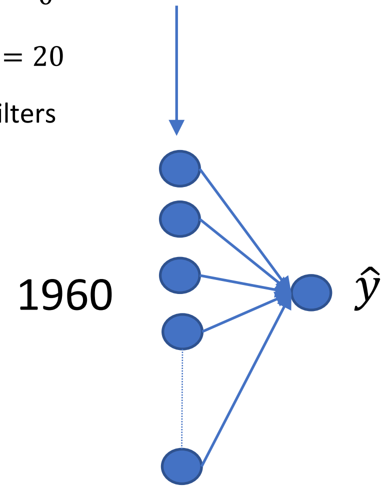


(7,7,40)

$$n_h^{[1]} = \left\lfloor \frac{n_h^{[0]} + 2p^{[1]} - f^{[1]}}{s^{[1]}} + 1 \right\rfloor = \frac{39 + 0 - 3}{1} + 1 = 37$$

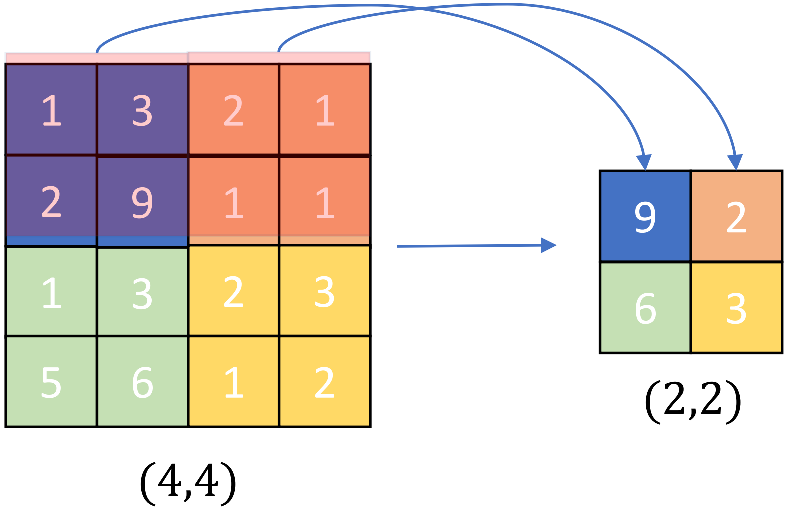
$$n_h^{[2]} = \left\lfloor \frac{n_h^{[1]} + 2p^{[2]} - f^{[2]}}{s^{[2]}} + 1 \right\rfloor = \frac{37 + 0 - 5}{2} + 1 = 17$$

$$n_h^{[3]} = \left\lfloor \frac{n_h^{[2]} + 2p^{[3]} - f^{[3]}}{s^{[3]}} + 1 \right\rfloor = \frac{17 + 0 - 5}{2} + 1 = 7$$



Pooling Layer

Useful tool to speed calculation up a make features more robust

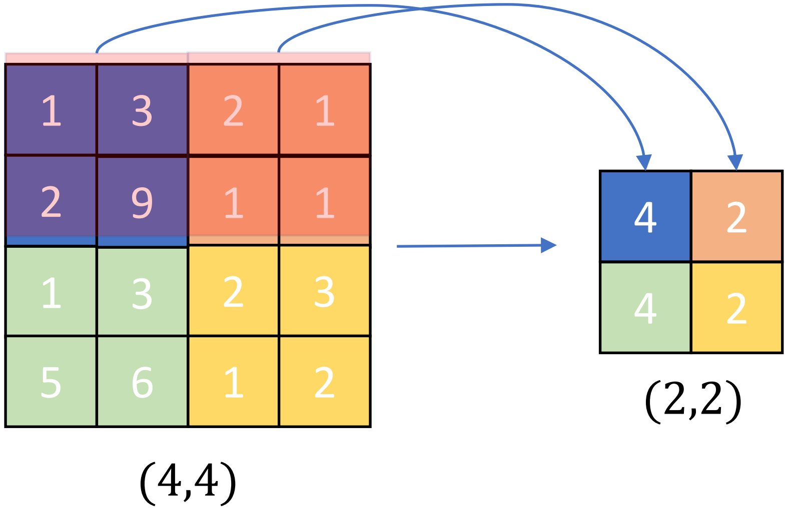


Max Pooling: The máximum value of the área

- **f=2 (filter size)**
- **s=2 (stride=2)**

You've just applied a convolutional filter with kernel size (2,2) and stride 2

Average Pooling



Average Pooling: The mean value of the área

- **f=2 (filter size)**
- **s=2 (stride=2)**

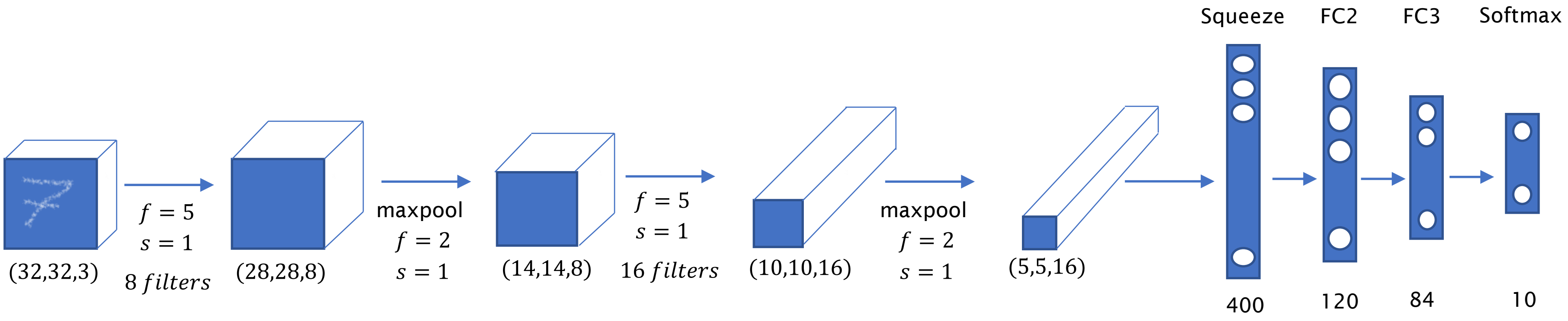
You've just applied a convolutional filter with kernel size (2,2) and stride 2

Some considerations about pooling

- Pooling doesn't involve parameters to learn, it's just a pretty straighforward technique for dimension reduction
- Padding is seldom used with Pooling
- Max pooling is much more used than average pooling

Real Case: LeNet-5

Built by Yann LeCun for digit recognition



Parameters in LeNet-5

	Activation Shape	Activation Size	nº parameters
Input	(32,32,3)	3072	0
Conv2D 1 (f=5,s=1)	(28,28,8)	6072	208
Pool1	(14,14,8)	1568	0
Conv2D (f=5,s=1)	(10,10,16)	1600	416
Pool2	(5,5,16)	400	0
FC2	(120,1)	120	48120
FC3	(84,1)	84	10164
Softmax	(10,1)	10	850

Total: 59758 parameters to optimize

Understading parameters

Think about all parameters shown in the previous spreadsheet

- Figure out how all the numbers are calculated
- Why the softmax has 10 outputs (instead, for example 5)

CONVOLUTIONS

- *Understanding convolutions*
- *Building a convolutional neural network*

Convolutions in PyTorch

The torch.nn module provides convolutions for 1, 2, and 3 dimensions:

- nn.Conv1d for time series,
- nn.Conv2d for images,
- and nn.Conv3d for volumes or videos.

For our CIFAR-10 data, we'll resort to nn.Conv2d.

Need to provide (at a minimum):

- Number of input features: in this case, 3 for the RGB channels
- Number of output features: arbitrary, more output features, higher capacity of the model
- Size of the kernel

Conv2d Module

Please, read the documentation

CONV2D

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of padding applied to the input. It can be either a string {'valid', 'same'} or a tuple of ints giving the amount of implicit padding applied on both sides.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels and producing half the output channels, and both subsequently concatenated.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\frac{\text{out\_channels}}{\text{in\_channels}}$).

Convolutions in PyTorch

Quick reminder of how to load cifar10

```python
label_map = {0: 0, 2: 1, 1: 2}

cifar10 = datasets.CIFAR10(
    data_path,
    train=True,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                             (0.2470, 0.2435, 0.2616))
    ]),
    target_transform=lambda label: label_map[label] if label in label_map else label
)

cifar10_val = datasets.CIFAR10(
    data_path,
    train=False,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                             (0.2470, 0.2435, 0.2616))
    ]),
    target_transform=lambda label: label_map[label] if label in label_map else label
)

birds_aeroplanes_train = [index for index, sample in enumerate(cifar10) if sample[1] in {0, 1}]
birds_aeroplanes_val = [index for index, sample in enumerate(cifar10_val) if
                        sample[1] in {0, 1}]

cifar2 = torch.utils.data.Subset(cifar10, birds_aeroplanes_train)
cifar2_val = torch.utils.data.Subset(cifar10_val, birds_aeroplanes_val)
```

Loading pictures, converting to Tensors

Normalizing

Maping labels

Choosing only airplanes and birds

Building Subsets

Convoluting a sample

```
# Defining de convolution operation
conv = nn.Conv2d(in_channels  = 3,
                 out_channels = 16,
                 kernel_size  = 3,
                 stride       = 1,
                 padding      =0
                 )

# Converting to a 4-dimension Tensor
img_unsqueezed = img.unsqueeze(0)
img_output = conv(img_unsqueezed)

print("Size of tensor previous to the convolution: {0:s}"
      .format(str(img_unsqueezed.size())))
print("Size of tensor after the convolution: {0:s}"
      .format(str(img_output.size())))
```

```
Size of tensor previous to the convolution: torch.Size([1, 3, 32, 32])
```

```
Size of tensor after the convolution: torch.Size([1, 16, 30, 30])
```

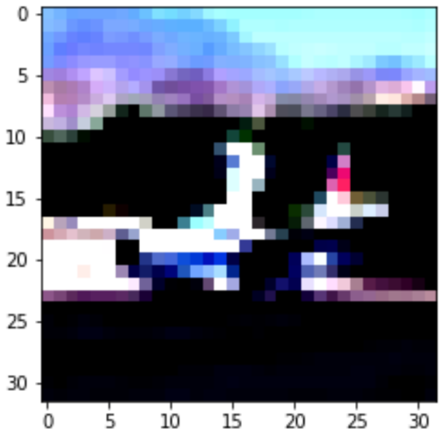
Calculating output dimensions...

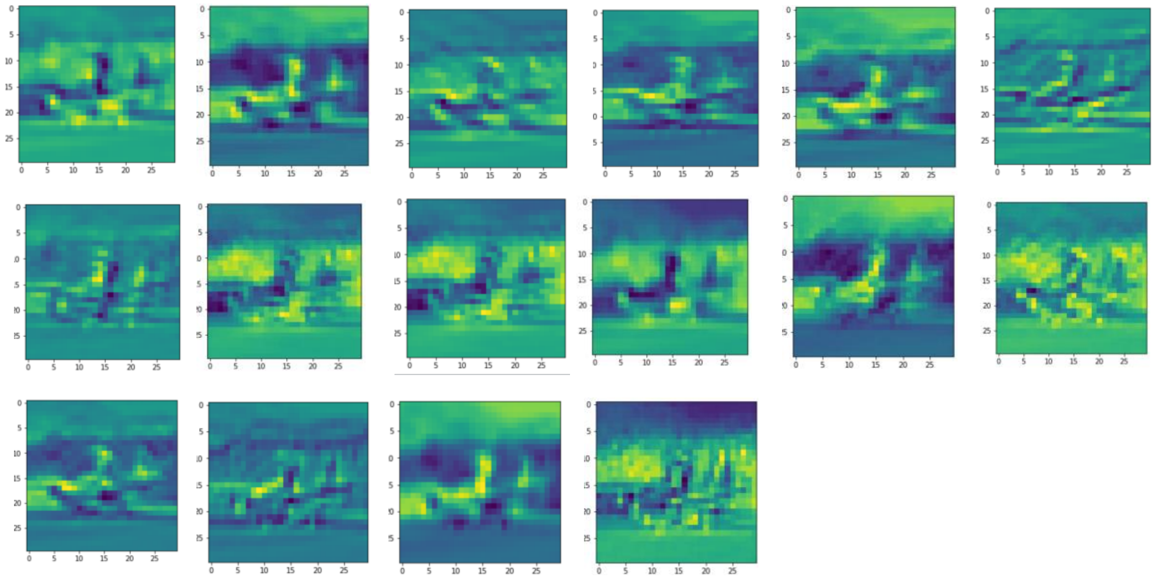
$$n_h = \left(\frac{n+2p-f}{s}+1\right) = \frac{32+0-3}{1}+1 = 30$$

Picture dimensión has been reduced from (32,32) to (30,30)

Convolution output

Convolution generates 16 blurred pictures according to the 16 filters used.





```python
for n in range(0,15):
    plt.imshow(img_output[0,n,:,:].detach())
    plt.show()
```

Dicussion

Universidad Francisco de Vitoria
UFV Madrid

Why are all the outputs different?

Are we using the same kernel?

CONVOLUTIONS

- *Understanding convolutions*
- *Building a convolutional neural network*

Testing different kernels

Blurring: a kernel with constant weights

```
conv = nn.Conv2d(3, 1, kernel_size=3, padding=1)
with torch.no_grad():
        conv.weight[:] = torch.tensor([[1.0, 1.0, 1.0],
                                       [1.0, 1.0, 1.0],
                                       [1.0, 1.0, 1.0]]) / 9
        conv.bias.zero_()
```

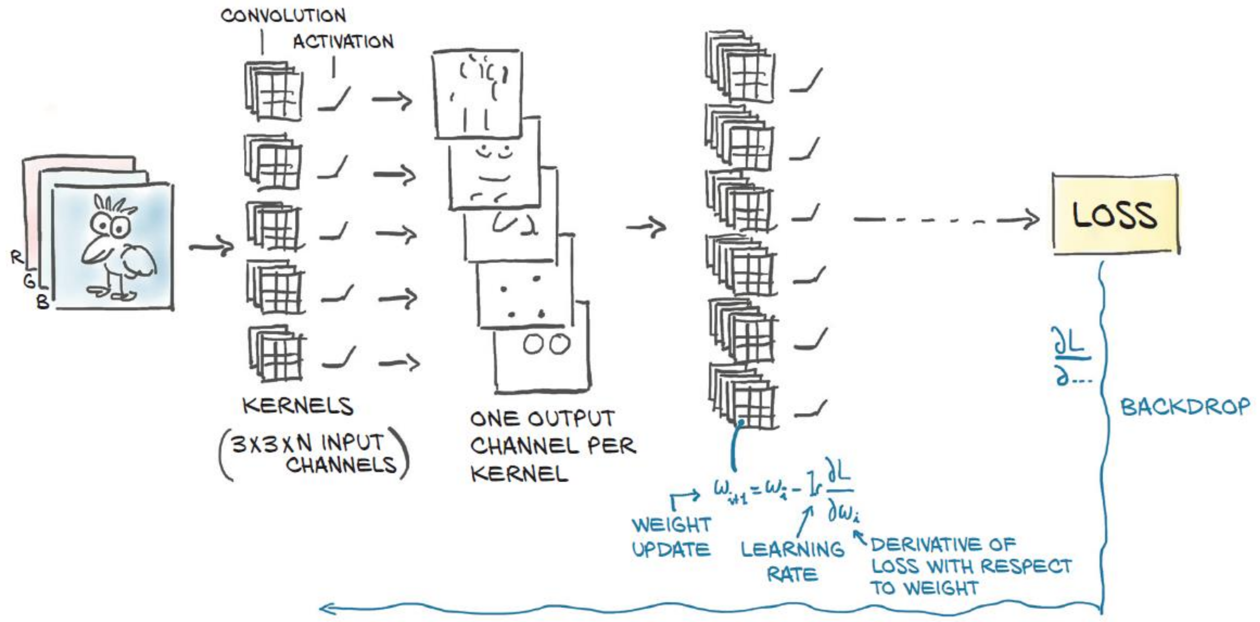
Edge detection: differences between neighboring pixels

```
conv = nn.Conv2d(3, 1, kernel_size=3, padding=1)
with torch.no_grad():
        conv.weight[:] = torch.tensor([[-1.0, 0.0, 1.0],
                                       [-1.0, 0.0, 1.0],
                                       [-1.0, 0.0, 1.0]])
        conv.bias.zero_()
```

Feature generation

The job of a convolutional neural network is to estimate the kernel of a **set of filter banks** in successive layers

- will transform a multichannel image into another multichannel image, where different channels correspond to different features



Idea of downsampling

Output tensors from a convolution layer (usually followed by an activation just like any other linear layer) tend to have high values where certain features corresponding to the estimated kernel are detected (such as vertical lines).

- Through training, kernel weights adjust to detect important features
- These features, where detected, produce high value in some parts of the tensor
- With downsampling, these parts “survive” at the expense of weaker responses

nn.MaxPool2d module

Max pooling provided in Pytorch by the nn.MaxPool2d module

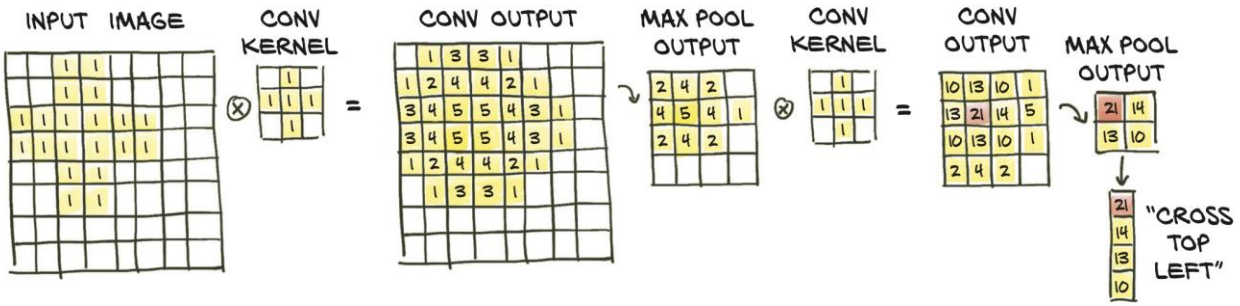
- Takes as input the size of the neighborhood over which to operate the pooling operation.
- Example, to downsample our image by half, use a size of 2

```
pool = nn.MaxPool2d(2)

img_output_pool =   pool(img_unsqueezed)
print("Size of tensor after the pooling: {0:s}"
      .format(str(img_output_pool.size())))
```

```
IPdb [6]: img_unsqueezed.size()
Out  [6]: torch.Size([1, 3, 32, 32])

IPdb [7]: img_output_pool.size()
Out  [7]: torch.Size([1, 3, 16, 16])
```

Combining convolutions and downsampling

Effect of stacking convolutions and downsampling: a large cross is highlighted using two small, cross-shaped kernels and max pooling.



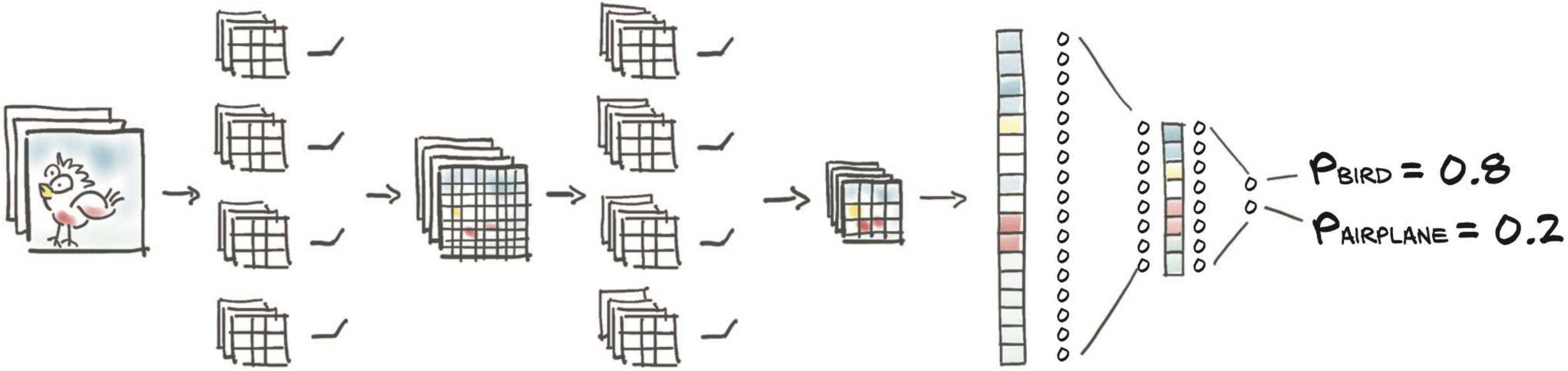
First set of kernels operates on small neighborhoods on first order, low-level features

Second set of kernels effectively operates on wider neighborhoods, producing features that are compositions of the previous features.

Obtaining probabilities

Procedure:

- Covert the 8-channel 8 × 8 image into a 1D vector (plus batch dimension); and
- Complete our network with a set of fully connected layers



CONVOLUTIONS

- *Understanding convolutions*
- *Building a convolutional neural network*
- *Creating custom nn.Module subclasses*

Subclassing nn.Module

Subclass nn.Module

Define a forward function that takes the inputs to the module and returns the output

- Typically, forward will use other modules—premade like convolutions or customized.
- To include these *submodules*, we define them in the constructor __init___
- Need to call super().__init__()

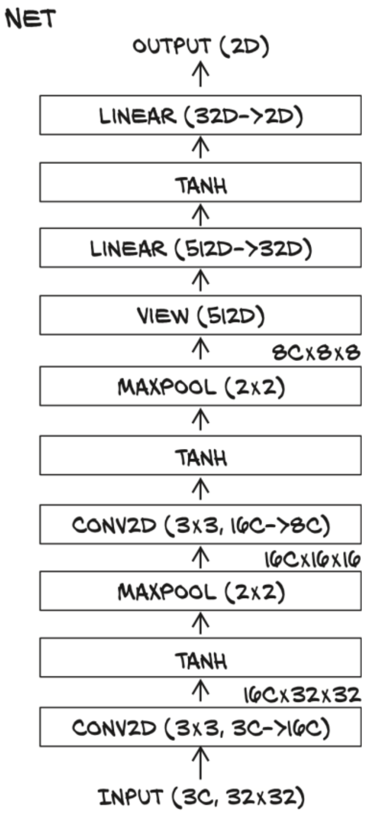
Desired structure

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=(3, 3), padding=1)
        self.act1 = nn.Tanh()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=(3, 3), padding=1)
        self.act2 = nn.Tanh()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.Tanh()
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)  # <1>
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

Missing conversion: we need to modify the output of the second MaxPool2d using view.

We will now see a new, more versatile way of building a neural nets.

Baseline convolutional network architecture



Training

Loading the dataset (the loader with handle batches and shuffling):

```
train_loader = DataLoader(cifar2,
                          batch_size=64,
                          shuffle=True)
val_loader = DataLoader(cifar2_val,
                        batch_size=64,
                        shuffle=False)
```

Activating GPU

```
device = (torch.device('cuda') if torch.cuda.is_available()
          else torch.device('cpu'))
print(f"Training on device {device}.")
```

Training loop

```python
def training_loop(n_epochs: int,
                  optimizer: torch.optim,
                  model: nn.Module,
                  loss_fn: nn.Module,
                  train_loader: DataLoader,
                  device: torch.device):

    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            imgs = imgs.to(device=device)  # <1>
            labels = labels.to(device=device)
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            loss_train += loss.item()

        if epoch == 1 or epoch % 10 == 0:
            print('{} Epoch {}, Training loss {}'.format(
                datetime.datetime.now(), epoch,
                loss_train / len(train_loader)))
```

Loop

Loading images and labels to GPU

Forward Pass

Calculating Loss

Backward Pass

Repetition

Validating

```python
def validate(model: nn.Module,
             train_loader: DataLoader,
             val_loader: DataLoader,
             device: torch.device):
    accdict = {}
    for name, loader in [("train", train_loader), ("val", val_loader)]:
        correct = 0
        total = 0

        with torch.no_grad():
            for imgs, labels in loader:
                imgs = imgs.to(device=device)
                labels = labels.to(device=device)
                outputs = model(imgs)
                _, predicted = torch.max(outputs, dim=1)  # <1>
                total += labels.shape[0]
                correct += int((predicted == labels).sum())

        print("Accuracy {}: {:.2f}".format(name, correct / total))
        accdict[name] = correct / total
    return accdict
```

Checking accuracy from both train and validation datasets

Sending the batch of pictures and labels to device

Predicting

Chossing the max value of prediction

Understading torch.max

torch.max returns the máximum value of the given set of values and “optionally” the index of the maximum value

TORCH.MAX

torch.max(*input*) → Tensor

Returns the maximum value of all elements in the `input` tensor.

WARNING

This function produces deterministic (sub)gradients unlike `max(dim=0)`

Parameters:

input (*Tensor*) – the input tensor.

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.6763,  0.7445, -2.2369]])
>>> torch.max(a)
tensor(0.7445)
```

torch.max(*input*, *dim*, *keepdim=False*, *, *out=None*)

Returns a namedtuple `(values, indices)` where `values` is the maximum value of each row of the `input` tensor in the given dimension `dim`. And `indices` is the index location of each maximum value found (argmax).

Returns a tuple

If `keepdim` is `True`, the output tensors are of the same size as `input` except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see `torch.squeeze()`), resulting in the output tensors having 1 fewer dimension than `input`.

Understading torch.max

The output of the model

```
IPdb [20]: outputs
Out  [20]:
tensor([[ 1.3349, -1.9178],
        [ 1.9040, -1.6632],
        [ 1.5620, -1.0870],
        [ 2.0129, -1.5942],
        [-0.3770,  0.6652],
        [ 0.1299,  0.5198],
        [ 0.5858, -0.4821],
        [-1.7611,  1.8181],
        [ 3.7983, -3.1361],
        [-2.4987,  2.0394],
        [-2.5485,  2.5286],
        [ 3.0376, -2.5142],
        [ 0.2787, -0.1730],
        [-3.0452,  2.9423],
        [ 0.6357, -0.9466],
        [ 2.4407, -2.3764],
        [-3.2843,  3.0517],
        [ 0.2308, -0.1195],
        [ 3.7184, -3.4098],
        [-3.1932,  3.6036],
        [-2.5922,  1.8526],
        [ 0.3395, -0.1361],
        [ 2.0941, -1.5432],
        [-3.3572,  3.3926],
        [-2.3391,  2.0032],
```

The output of torch.max

```
IPdb [25]: torch.max(outputs, dim=1)
Out  [25]:
torch.return_types.max(
values=tensor([1.3349, 1.9040, 1.5620, 2.0129, 0.6652, 0.5198, 0.5858, 1.8181,
        3.7983, 2.0394, 2.5286, 3.0376, 0.2787, 2.9423, 0.6357, 2.4407,
        3.0517, 0.2308, 3.7184, 3.6036, 1.8526, 0.3395, 2.0941, 3.3926,
        2.0032, 2.6029, 0.2059, 1.9168, 1.4595, 3.1268, 4.0068, 3.8308,
        2.3819, 2.9784, 2.3783, 4.2831, 1.9990, 0.6077, 1.4134, 1.8785,
        2.3575, 0.1935, 2.5324, 2.9165, 1.8345, 2.1046, 1.1047, 1.5087,
        1.6084, 1.6869, 2.8519, 0.7044, 1.6892, 3.5769, 1.1302, 0.8916,
        1.9175, 0.6086, 1.4908, 0.5524, 1.8217, 4.0356, 2.7264, 3.0865],
       device='cuda:0'),
indices=tensor([0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0,
        0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1,
        0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1],
       device='cuda:0'))
```

We can compare the position of our máximum value with the ground truth.

Remember! {0:airplane, 1:bird}

```
correct += int((predicted == labels).sum())
```

Saving and loading the model

Saving the model:

```
torch.save(model.state_dict(), data_path + 'birds_vs_airplanes.pt')
```

Loading the model (restores same device it was saved from):

```
loaded_model = Net()
loaded_model.load_state_dict(torch.load(data_path + 'birds_vs_airplanes.pt'))
```

Setting the device

```
loaded_model = Net().to(device=device)
loaded_model.load_state_dict(torch.load(data_path + 'birds_vs_airplanes.pt',
                                        map_location=device))
```