

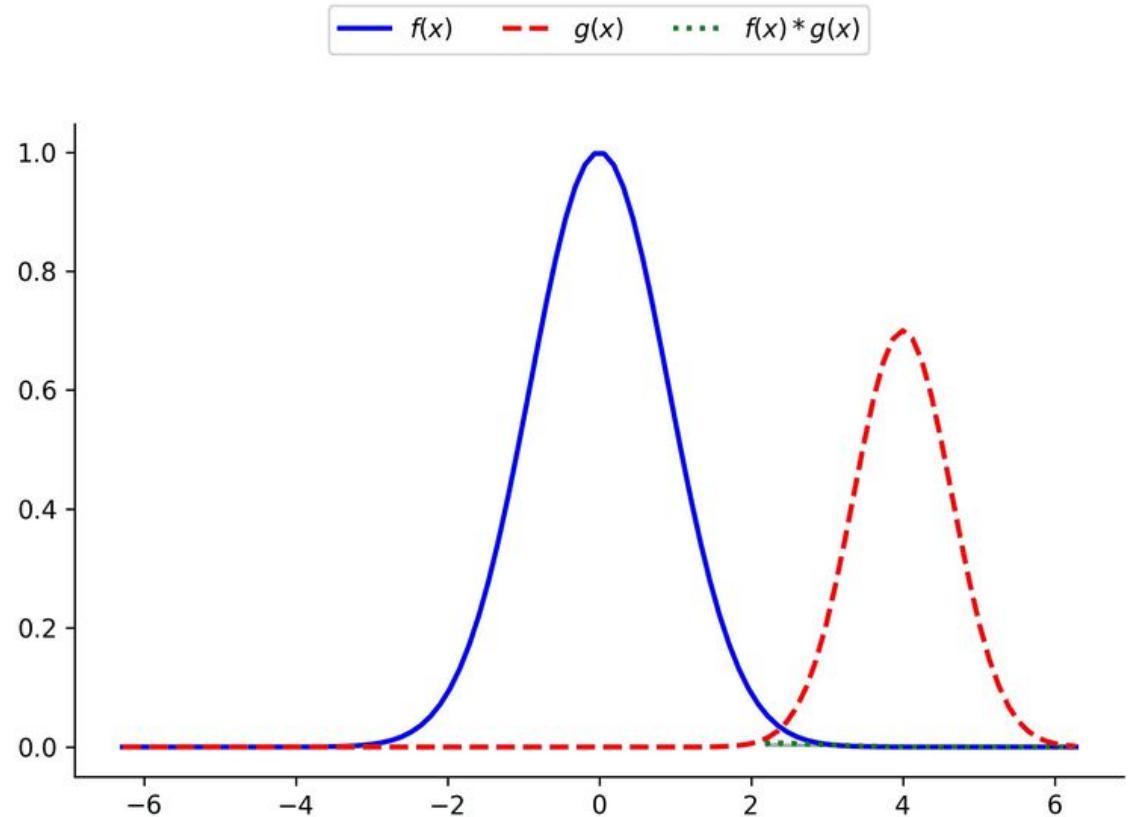
PPGEEC2318

Machine Learning

Convolutions

Ivanovitch Silva

ivanovitch.silva@ufrn.br



Convolutions

Agenda

In this lesson, we will

1

Understand
Arithmetic of
Convolution layers

2

Build
A model for multiclass
classification

3

Understand
The role of the softmax
function

4

Use
Cross-entropy losses

5

Visualize
Filters learned by Conv.
Layers



Original



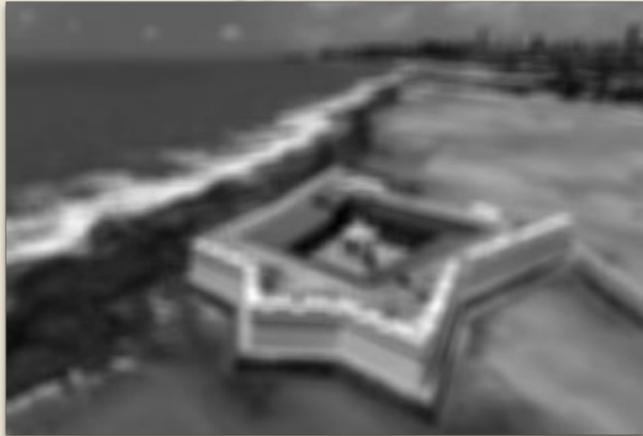
small.blur - convolve



sobel.y - convolve



large.blur - convolve



laplacian - convolve



Deep Learning with PyTorch Step-by-Step

A Beginner's Guide



Chapter 5: Convolutions

Spoilers

- › Jupyter Notebook
- › Convolutions

Pooling

Flattening

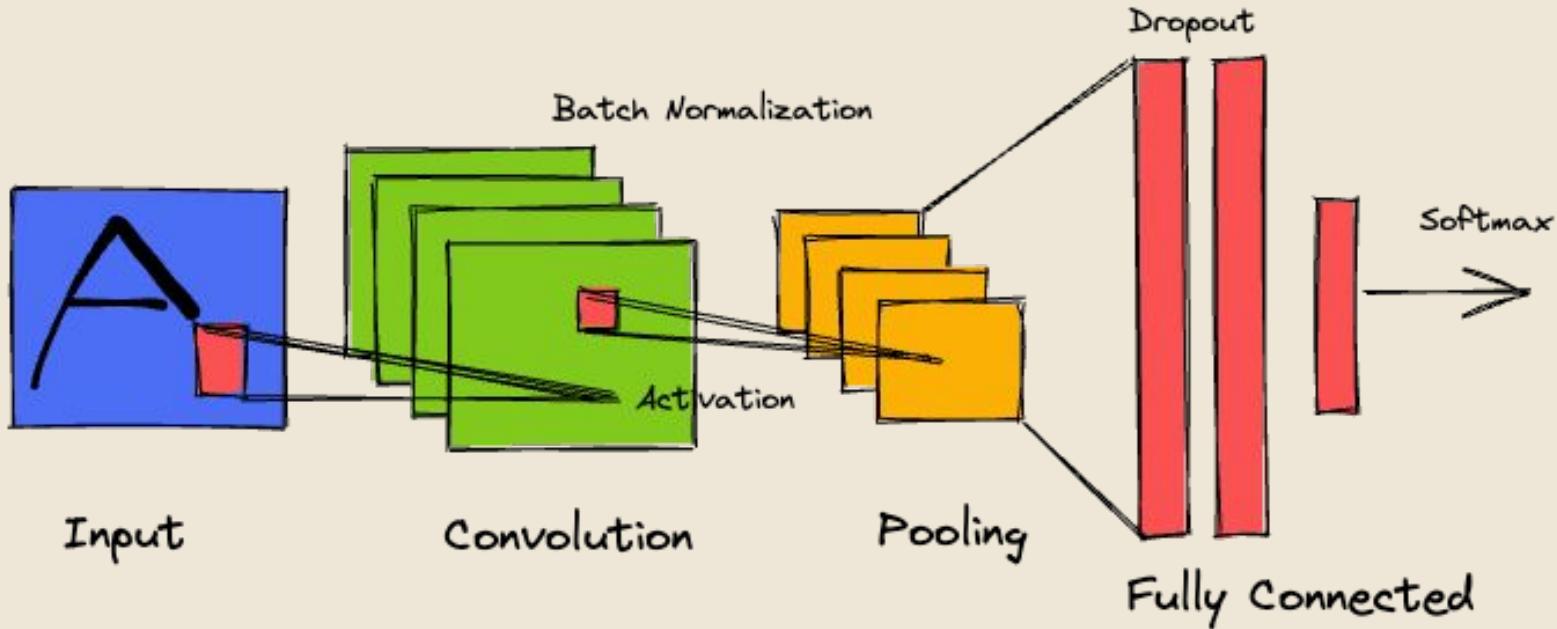
Dimensions

- › Typical Architecture
- › A Multiclass Classification Problem
- › Visualizing Filters and More!

Putting It All Together

Recap

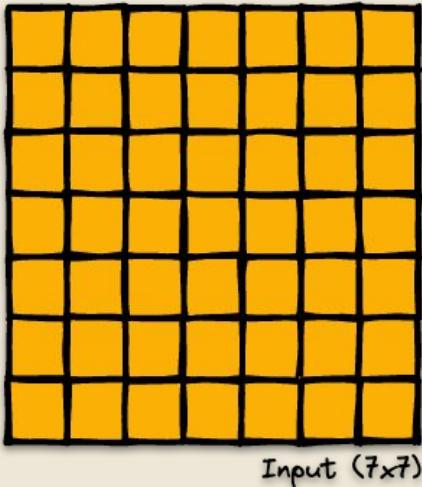
CNN Building Blocks



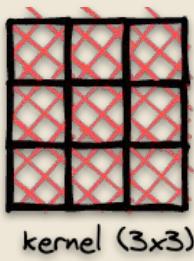
Convolution Explained in Code

```
np.sum(conv)  
8
```

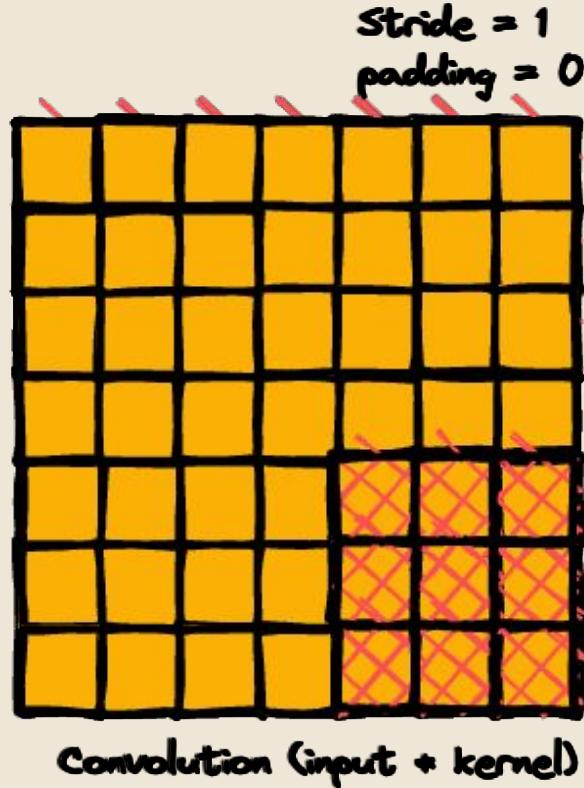
```
import numpy as np  
image = np.array([  
    [1,2,3],  
    [4,5,6],  
    [7,8,9]  
])  
kernel = np.array([  
    [-1,0,1],  
    [-2,0,2],  
    [-1,0,1]  
])  
  
conv = image*kernel  
conv  
array([[ -1,   0,   3],  
       [ -8,   0,  12],  
       [ -7,   0,   9]])
```



Input (7x7)



kernel (3x3)

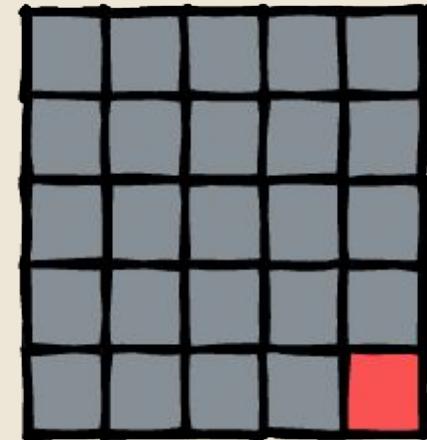


```
import torch.nn as nn

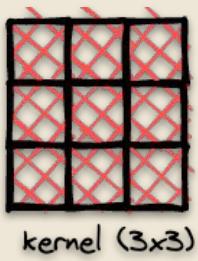
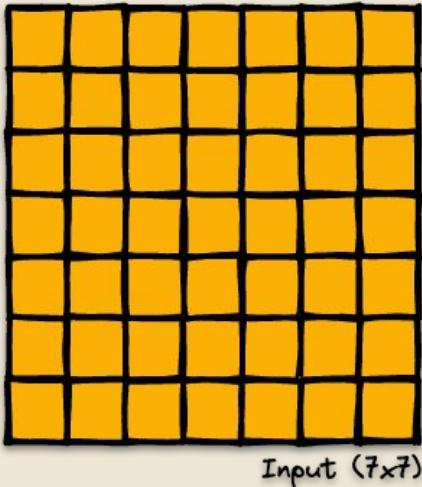
conv = nn.Conv2d(in_channels=1,
                out_channels=1,
                kernel_size=3,
                stride=1, padding=0)

fm = conv(image)
fm.shape
torch.Size([1, 1, 5, 5])
```

```
conv.weight
Parameter containing:
tensor([[[[ 0.1872,  0.1167,  0.1626],
          [-0.2925,  0.1007, -0.0899],
          [-0.0626, -0.0032,  0.2826]]]], requires_grad=True)
conv.bias
Parameter containing:
tensor([-0.0145], requires_grad=True)
```

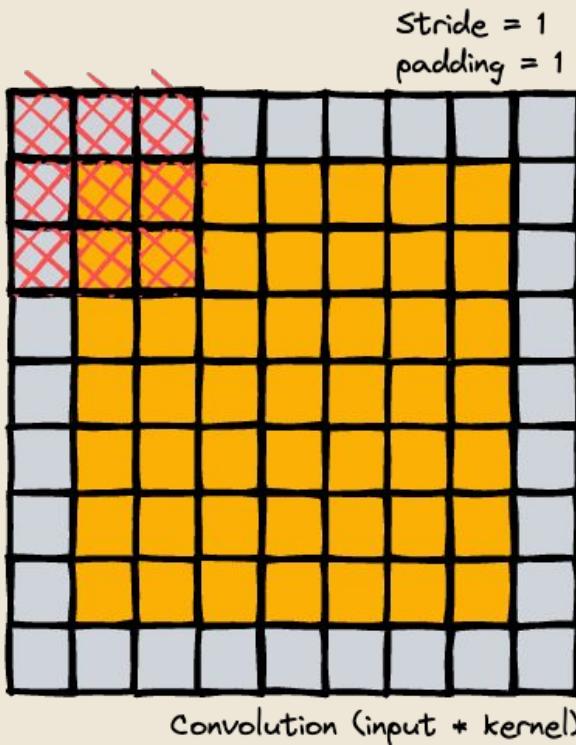


feature map (5x5)

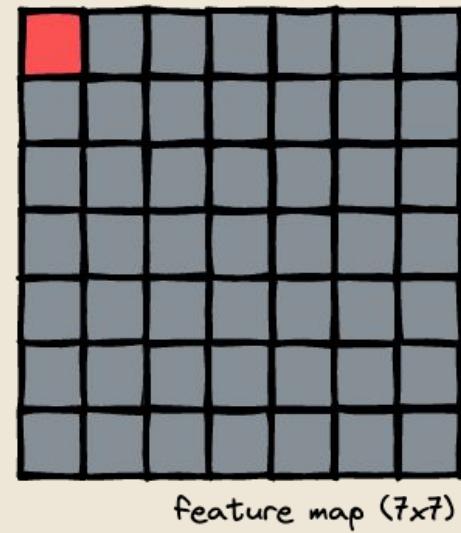


Summary of Convolutions

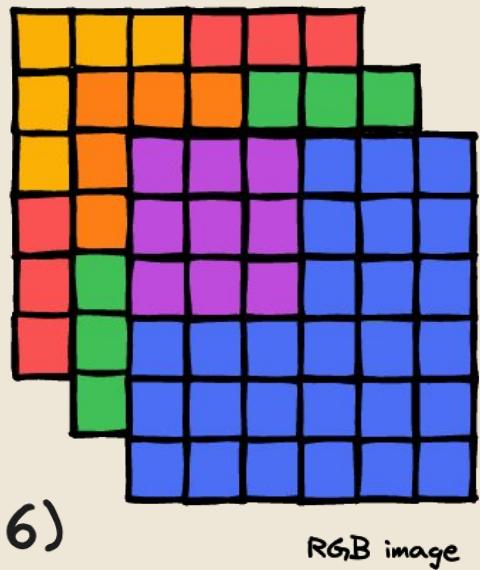
$n \times n$ input
 $k \times k$ kernel
padding p
stride s



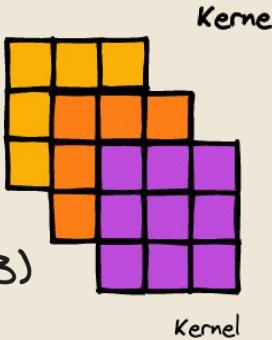
$$\left\lfloor \frac{n + 2p - k}{s} + 1 \right\rfloor$$



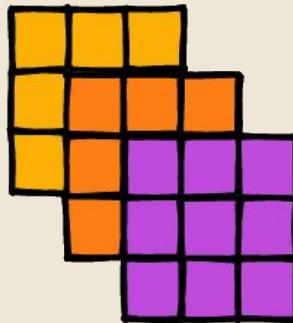
$(1, 3, 6, 6)$



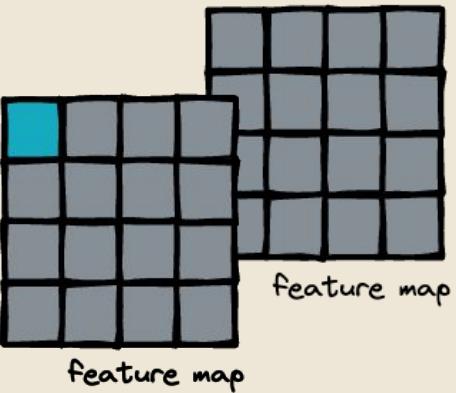
$(2, 3, 3, 3)$



$*$

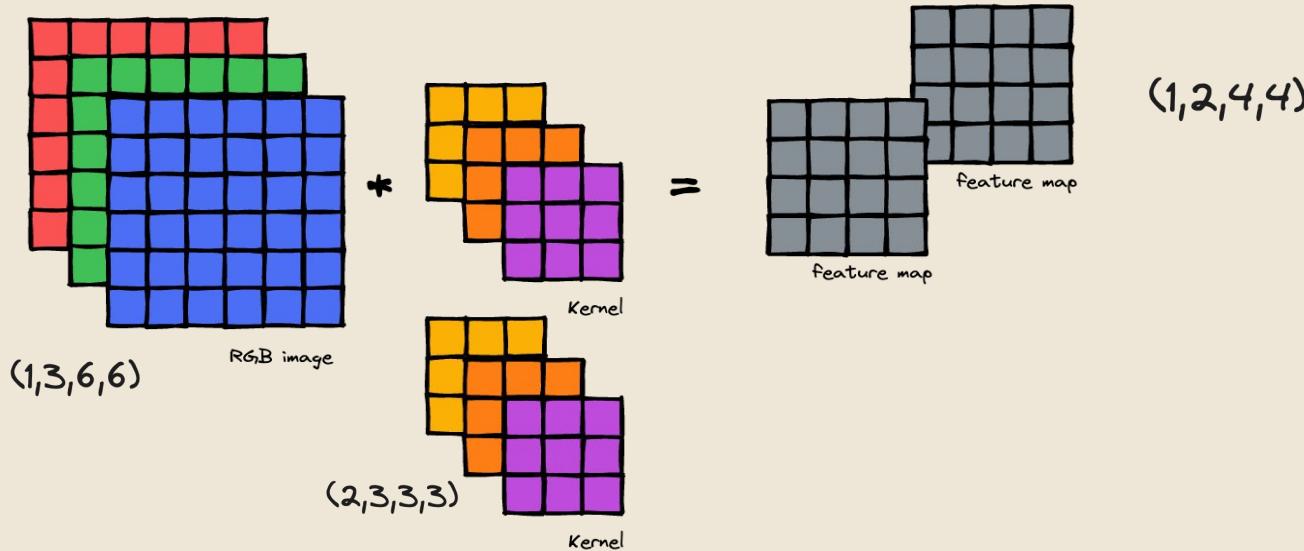


$=$



$(1, 2, 4, 4)$

$$\left\lfloor \frac{n + 2p - k}{s} + 1 \right\rfloor$$



```

import numpy as np

random_array = np.random.rand(1, 3, 6, 6)
image = torch.as_tensor(random_array).float()

conv_multiple = nn.Conv2d(in_channels=3, out_channels=2,
                        kernel_size=3, stride=1, padding=0)

fm = conv_multiple(image)

```

```

fm.shape
torch.Size([1, 2, 4, 4])

conv_multiple.weight.shape
torch.Size([2, 3, 3, 3])

conv_multiple.bias.shape
torch.Size([2])

```

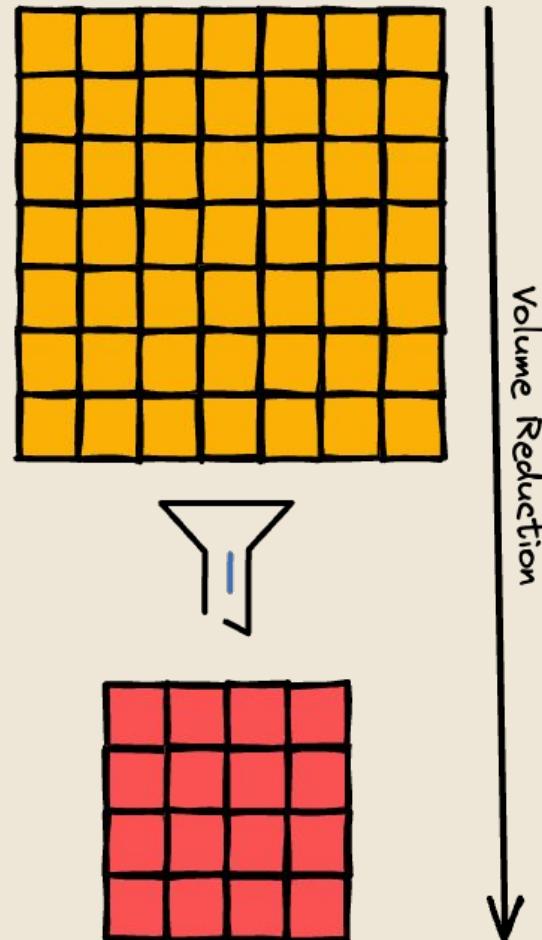
Activation Function

```
fm = np.array([[[[-10, 3, 5, 0],  
                 [4, -3, 3, -3],  
                 [1, 0, 20, 4],  
                 [5, 6, -10, 9]]]])  
  
tensor = torch.tensor(fm).float()  
# Apply ReLU function  
relu = nn.ReLU()  
relu(tensor)  
  
tensor([[[[ 0.,  3.,  5.,  0.],  
          [ 4.,  0.,  3.,  0.],  
          [ 1.,  0.,  20.,  4.],  
          [ 5.,  6.,  0.,  9.]]]])
```

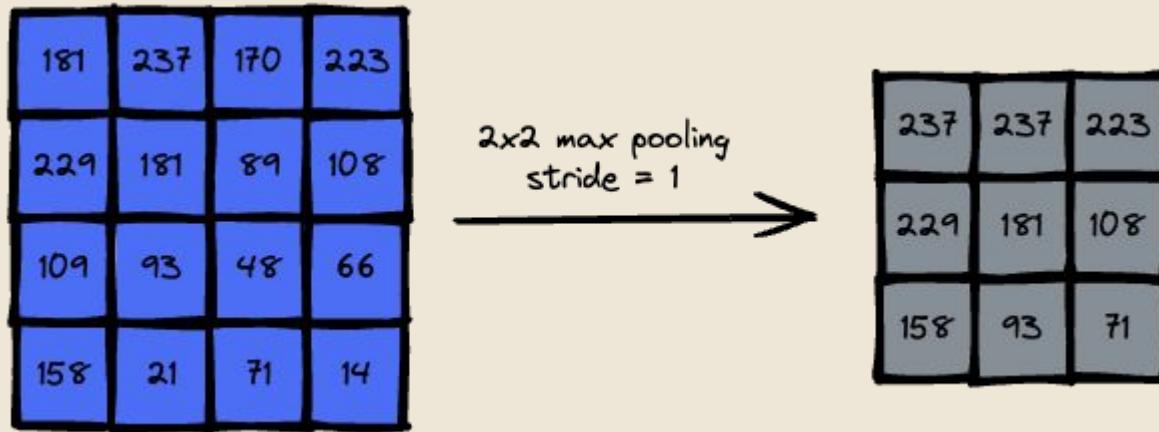
Pooling Layers

Two methods to reduce size of volume in CNN

1. CONV layers, typically with stride $S > 1$
2. Pooling layers
 - a. Kernel (only a function and without weights, $S > 2$)



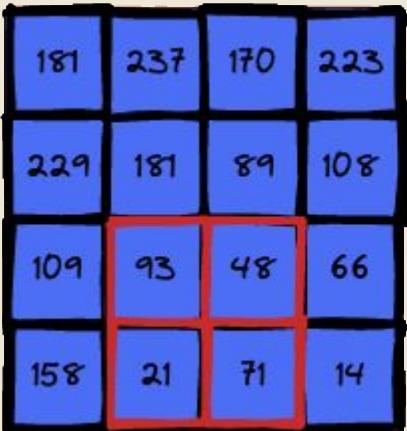
Pooling Layers



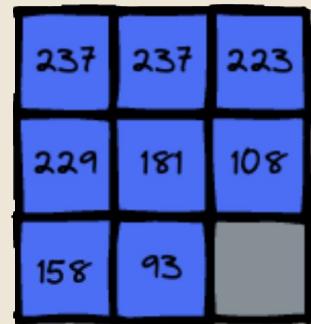
$$\left\lfloor \frac{n + 2p - k}{s} + 1 \right\rfloor$$

Pooling Layers

```
pool_data = np.array([[[[181,237,170,223],  
[229,181,89,108],  
[109,93,48,66],  
[158,21,71,14]]]])  
  
tensor = torch.tensor(pool_data).float()  
tensor.shape  
torch.Size([1, 1, 4, 4])
```



*2x2 max pooling
stride = 1*

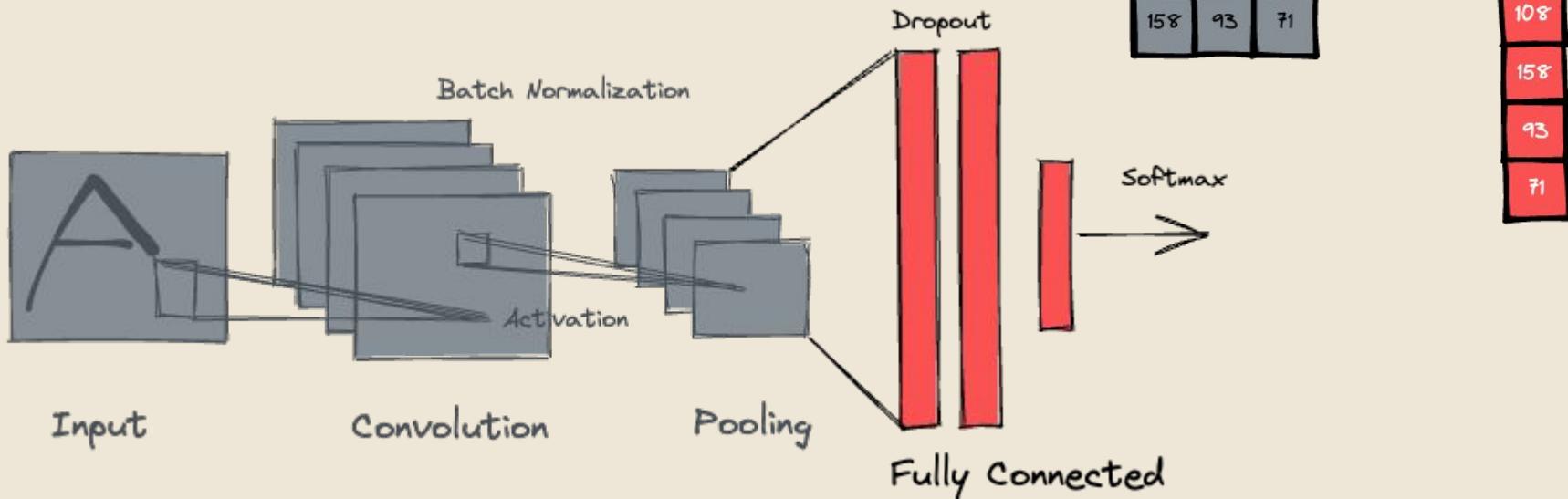


```
maxpool = nn.MaxPool2d(kernel_size=2,stride=1)  
pooled = maxpool(tensor)  
pooled.shape  
torch.Size([1, 1, 3, 3])
```

```
pooled  
tensor([[[[237., 237., 223.],  
[229., 181., 108.],  
[158., 93., 71.]]]])
```

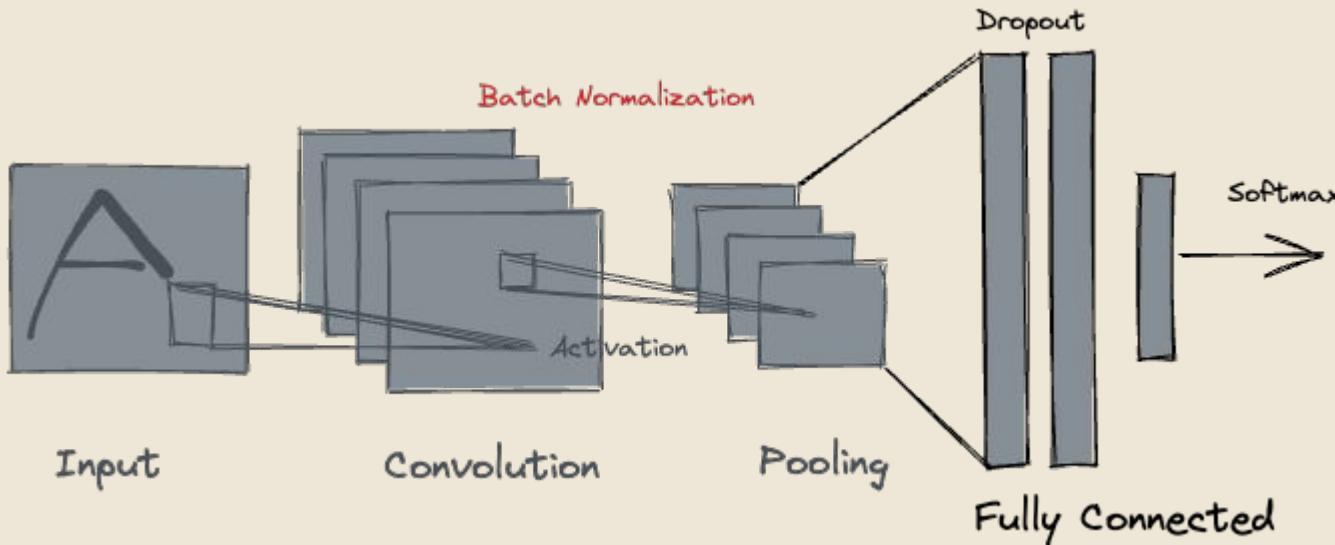
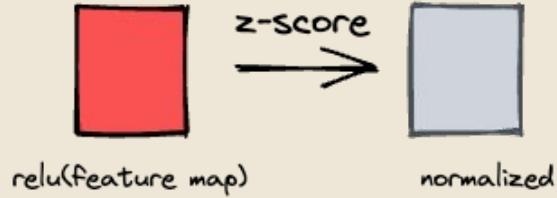
Fully Connected Layer

CONV => RELU => POOL => ... => FC



```
flattened = nn.Flatten()  
flattened(pooled)  
tensor([[237., 237., 223., 229., 181., 108., 158., 93., 71.]])
```

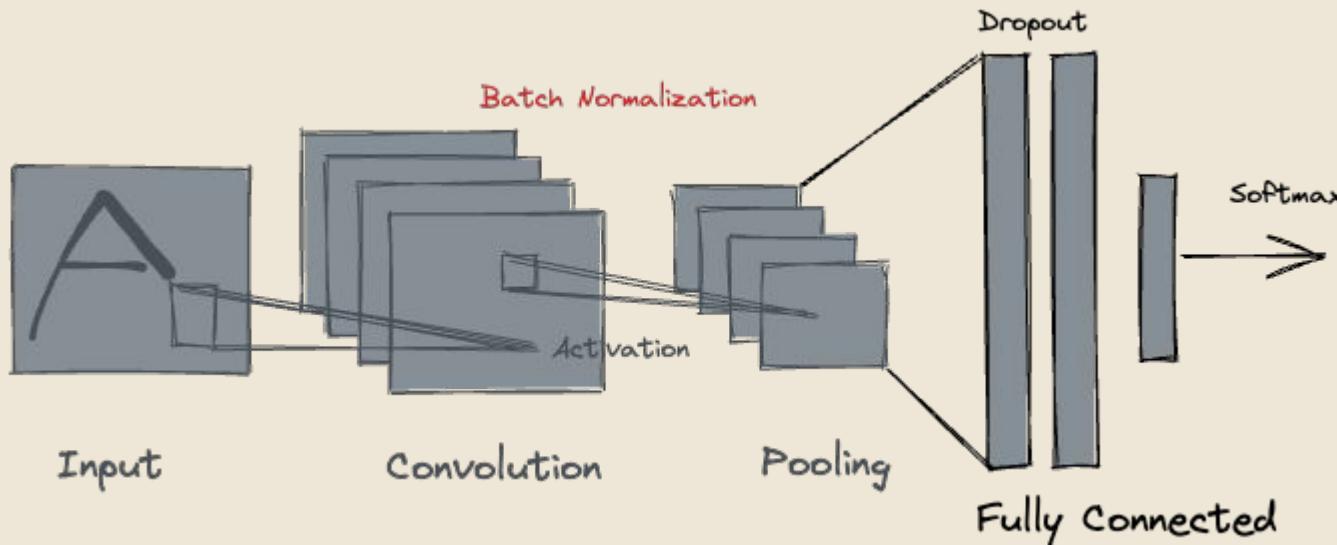
Batch Normalization



Accelerate the training process
Lightweight regularization

```
z = np.random.rand(3,3)
mean, std = np.mean(z), np.std(z)
norm = (z - mean)/(std + 1e-7)
```

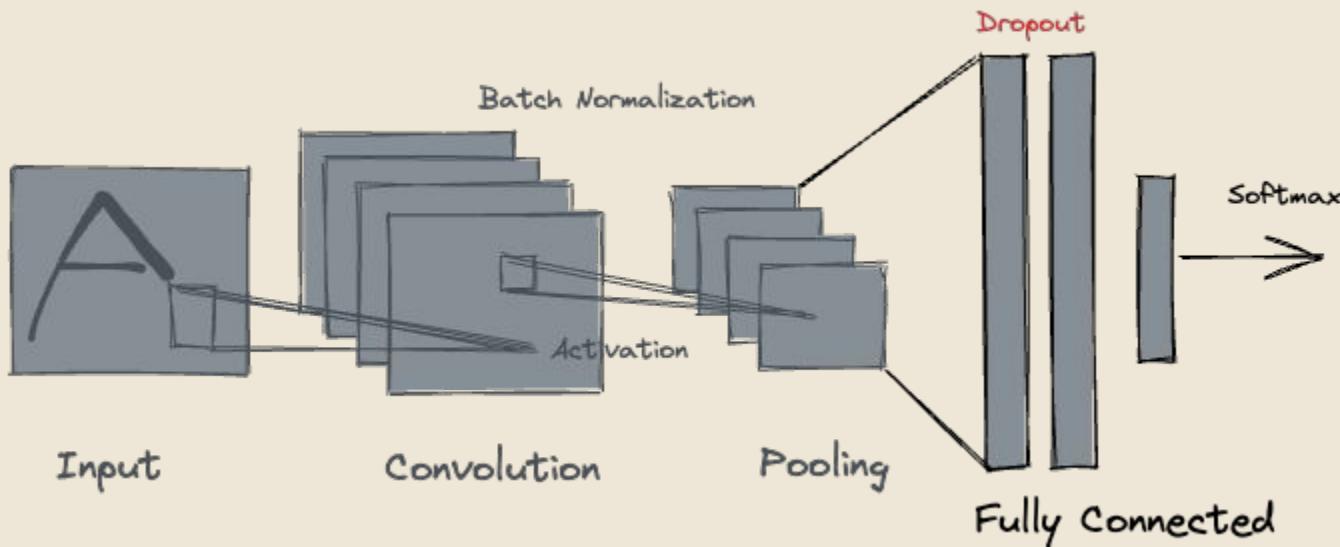
Batch Normalization



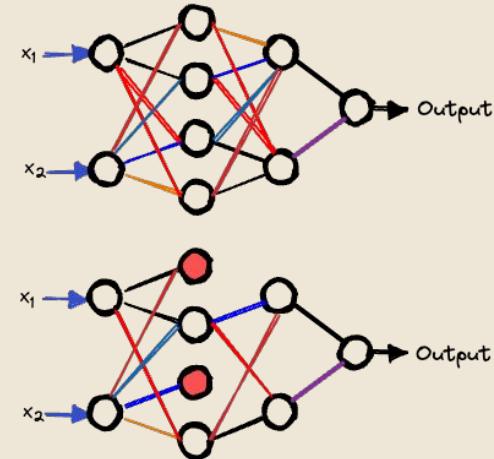
CONV => RELU => **BN** => POOL => ... => FC

CONV => **BN** => RELU => POOL => ... => FC

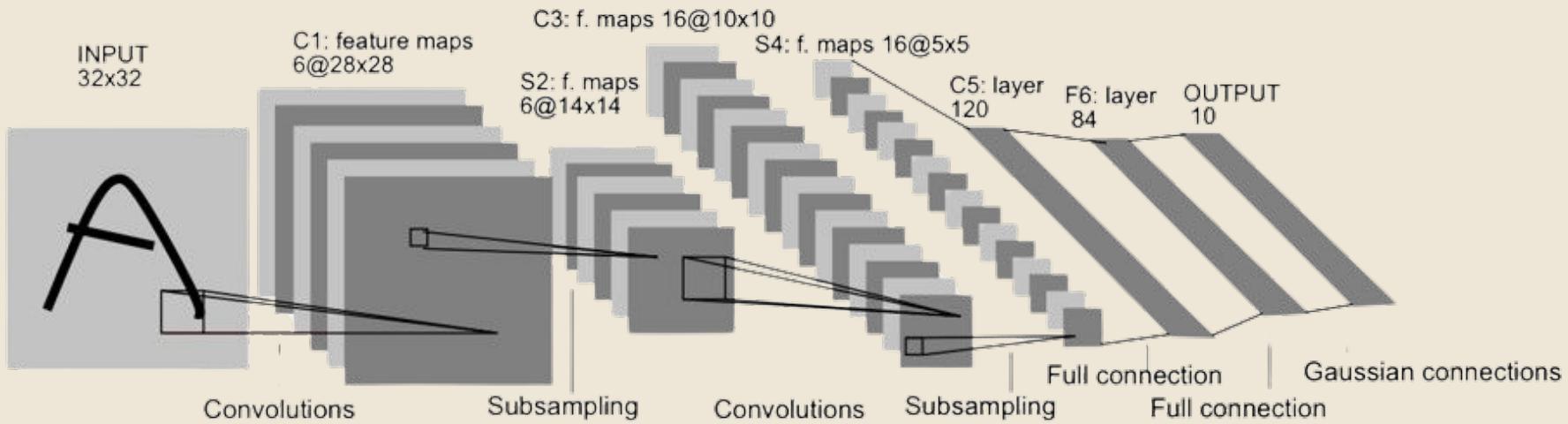
Dropout



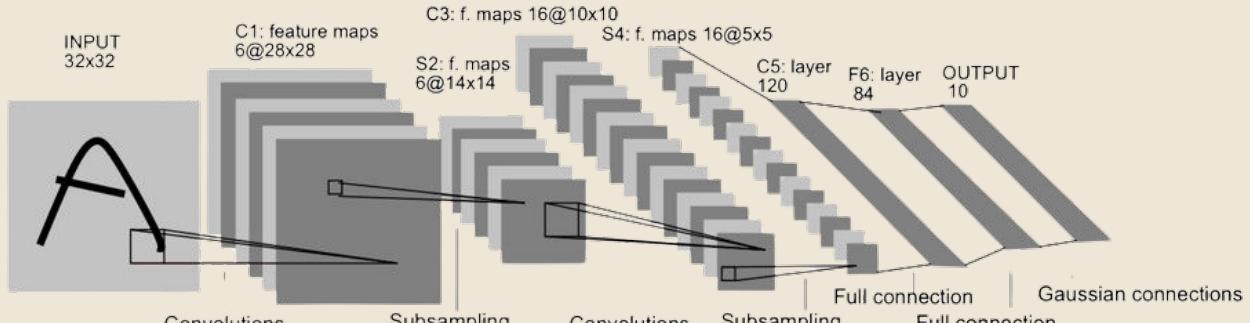
- It is a form of regularization
- Reduces overfitting
- Increases test/validation accuracy (sometimes at expense of training accuracy)
- Randomly disconnects node from current layers to next layer with probability, p



LeNet-5



LeNet-5



<https://ieeexplore.ieee.org/document/726791>

```
lenet = nn.Sequential()

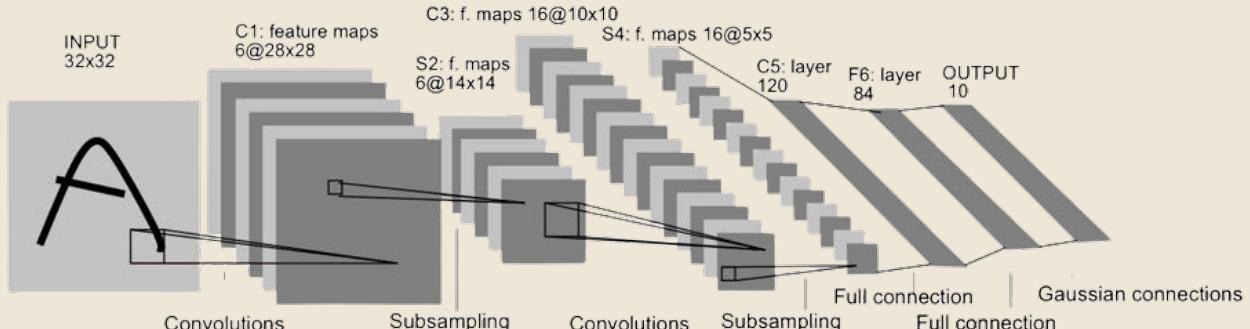
# Featurizer
# Block 1: 1@28x28 -> 6@28x28 -> 6@14x14
lenet.add_module('C1', nn.Conv2d(in_channels=1,
                               out_channels=6, kernel_size=5, padding=2))
lenet.add_module('func1', nn.ReLU())
lenet.add_module('S2', nn.MaxPool2d(kernel_size=2))
```

$$\left\lfloor \frac{n + 2p - k}{s} + 1 \right\rfloor$$

$$C1 = 6 \times (1 \times 5 \times 5) = 150 + 6 \text{ bias} = 156 \text{ parameters}$$



LeNet-5



<https://ieeexplore.ieee.org/document/726791>

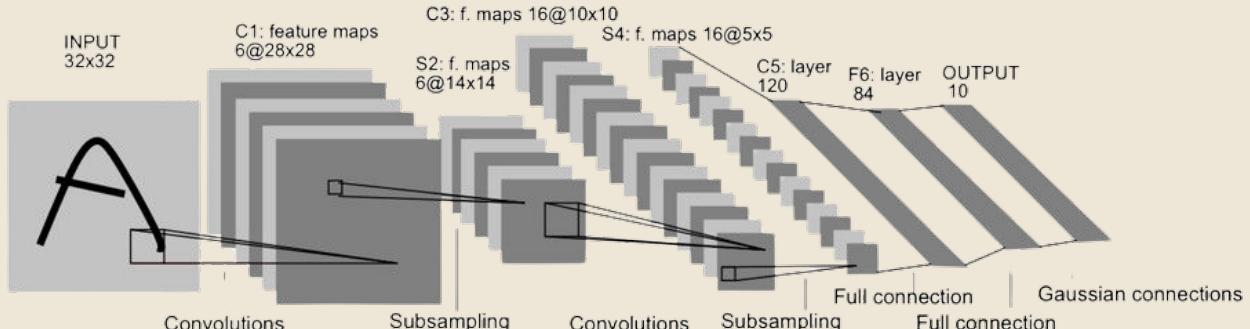
```
# Block 2: 6@14x14 -> 16@10x10 -> 16@5x5
lenet.add_module('C3', nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5))
lenet.add_module('func2', nn.ReLU())
lenet.add_module('S4', nn.MaxPool2d(kernel_size=2))
```

$$\left\lfloor \frac{n + 2p - k}{s} + 1 \right\rfloor$$

$$C3 = 16 \times (6 \times 5 \times 5) = 2400 + 16 \text{ bias} = 2416 \text{ parameters}$$



LeNet-5



<https://ieeexplore.ieee.org/document/726791>

```
# Block 3: 16@5x5 -> 120@1x1
lenet.add_module('C5', nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5))
lenet.add_module('func2', nn.ReLU())
# Flattening
lenet.add_module('flatten', nn.Flatten())
```

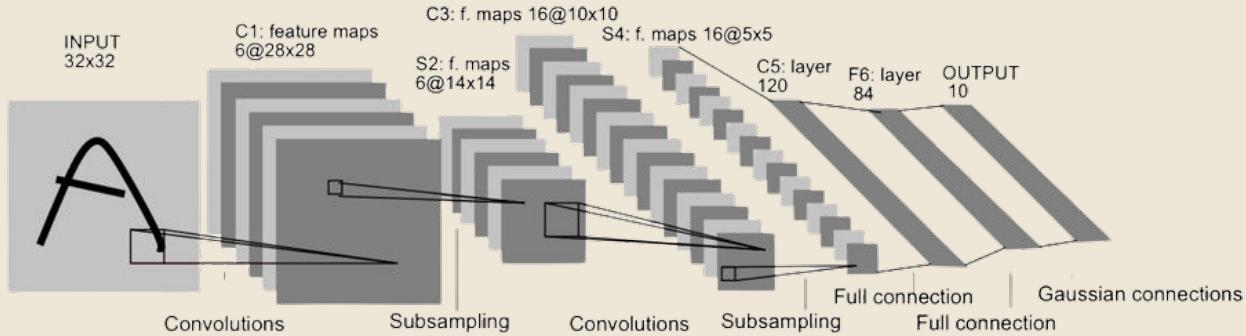
$$\left\lfloor \frac{n + 2p - k}{s} + 1 \right\rfloor$$

$$C5 = 120 \times (16 \times 5 \times 5) = 48.000 + 120 \text{ bias} = 48.120 \text{ parameters}$$



LeNet-5

61706
params



<https://ieeexplore.ieee.org/document/726791>

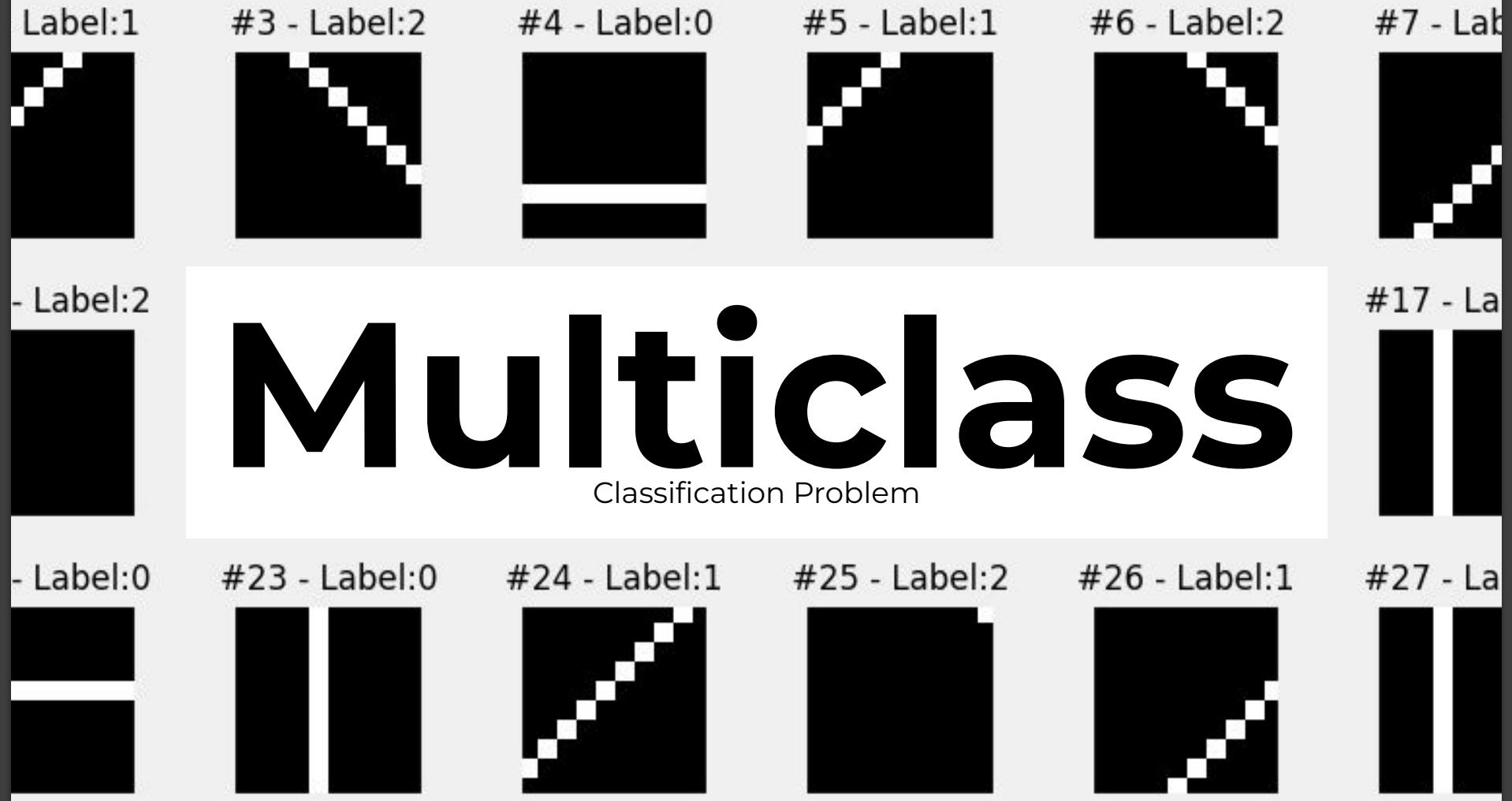
```
# Classification
# Hidden Layer
lenet.add_module('F6', nn.Linear(in_features=120, out_features=84))
lenet.add_module('func3', nn.ReLU())
# Output Layer
lenet.add_module('OUTPUT', nn.Linear(in_features=84, out_features=10))
```

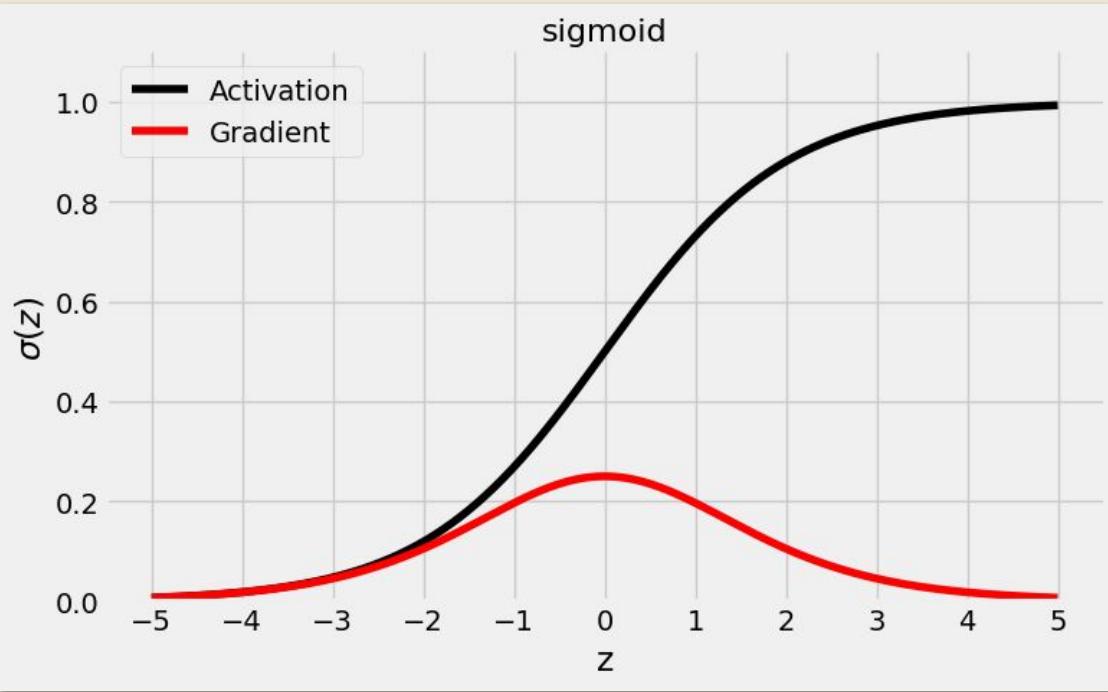
$$\left\lfloor \frac{n + 2p - k}{s} + 1 \right\rfloor$$

$$F6 = 120 \times 84 = 10.080 + 84 \text{ bias} = 10.164 \text{ parameters}$$

$$OUTPUT = 84 \times 10 = 840 + 10 \text{ bias} = 850 \text{ parameters}$$

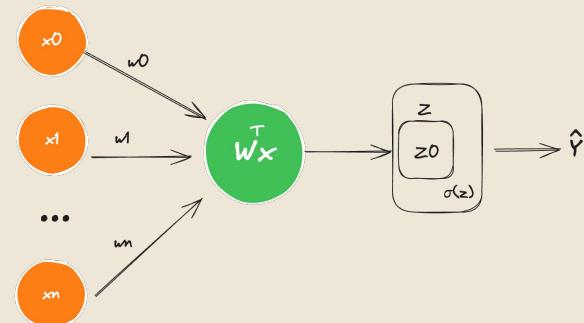






$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

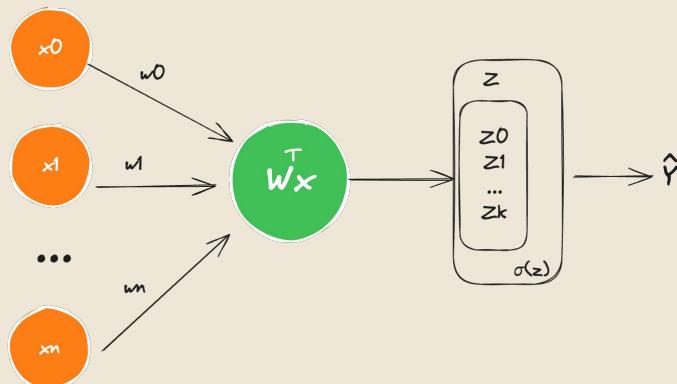
In **binary classification** problems, the model would produce one **logit** , and one logit only, for each data point. It makes sense, as binary classification is about answering a simple question: **"Does a given data point belong to the positive class? "**



```
nn.Softmax(dim=-1)(logits)
tensor([0.7273, 0.1818, 0.0909])
```

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{c=0}^{C-1} e^{z_c}}$$

$$\text{softmax}(z) = \left[\frac{e^{z_0}}{e^{z_0} + e^{z_1} + e^{z_2}}, \frac{e^{z_1}}{e^{z_0} + e^{z_1} + e^{z_2}}, \frac{e^{z_2}}{e^{z_0} + e^{z_1} + e^{z_2}} \right]$$



```
logits = torch.tensor([ 1.3863, 0.0000, -0.6931])
odds_ratios = torch.exp(logits)
softmaxed = odds_ratios / odds_ratios.sum()
tensor([0.7273, 0.1818, 0.0909])
```

Loss Function for Multiclass Classification

$$\text{BCE}(y) = -\frac{1}{(N_{\text{pos}} + N_{\text{neg}})} \left[\sum_{i=1}^{N_{\text{pos}}} \log(\text{P}(y_i = 1)) + \sum_{i=1}^{N_{\text{neg}}} \log(1 - \text{P}(y_i = 1)) \right]$$

For a binary classification

Cross-entropy measures the discrepancy between the true class distribution (p) and the predicted distribution (q).

$$H(p, q) = - \sum_{i=0}^{C-1} p_i \log(q_i) \quad q_i = \frac{e^{z_i}}{\sum_{i=0}^{C-1} e^{z_i}}$$

$$H(p, q) = - \sum_{i=0}^{C-1} p_i \log(q_i) \quad q_i = \frac{e^{z_i}}{\sum_{i=0}^{C-1} e^{z_i}}$$

```

import torch

# Seed definition, create logits and labels
torch.manual_seed(11)
dummy_logits = torch.randn((5, 3))
dummy_labels = torch.tensor([0, 0, 1, 2, 1])

# Softmax function
def softmax(logits):
    exp_logits = torch.exp(logits)
    sum_exp_logits = torch.sum(exp_logits, dim=1, keepdim=True)
    return exp_logits / sum_exp_logits

# Apply Softmax
probabilities = softmax(dummy_logits)

```

```

dummy_logits
tensor([[ 0.7376,  1.9459, -0.6995],
        [-1.3023, -0.5133, -0.2696],
        [ 0.2462,  0.4839,  0.4504],
        [-0.9568,  1.5012, -0.3136],
        [-0.2343, -1.0713,  0.1648]])
```

```

probabilities
tensor([[0.2181, 0.7301, 0.0518],
        [0.1664, 0.3663, 0.4673],
        [0.2861, 0.3629, 0.3510],
        [0.0686, 0.8010, 0.1305],
        [0.3421, 0.1481, 0.5098]])
```

$$H(p, q) = - \sum_{i=0}^{C-1} p_i \log(q_i) \quad q_i = \frac{e^{z_i}}{\sum_{i=0}^{C-1} e^{z_i}}$$

```
# Calculating cross-entropy manually
def cross_entropy_loss(probabilities, labels):
    # Getting the probabilities of the correct classes
    correct_class_probs = probabilities[range(len(labels)), labels]
    # Calculating the cross-entropy loss
    loss = -torch.log(correct_class_probs)
    return torch.mean(loss)
```

```
# Calculating the loss
loss = cross_entropy_loss(probabilities, dummy_labels)

print("\nLoss:")
print(loss.item())

Loss:
1.6552671194076538
```

```
dummy_logits
tensor([[ 0.7376,  1.9459, -0.6995],
        [-1.3023, -0.5133, -0.2696],
        [ 0.2462,  0.4839,  0.4504],
        [-0.9568,  1.5012, -0.3136],
        [-0.2343, -1.0713,  0.1648]])
```

```
probabilities
tensor([[0.2181, 0.7301, 0.0518],
        [0.1664, 0.3663, 0.4673],
        [0.2861, 0.3629, 0.3510],
        [0.0686, 0.8010, 0.1305],
        [0.3421, 0.1481, 0.5098]])
```

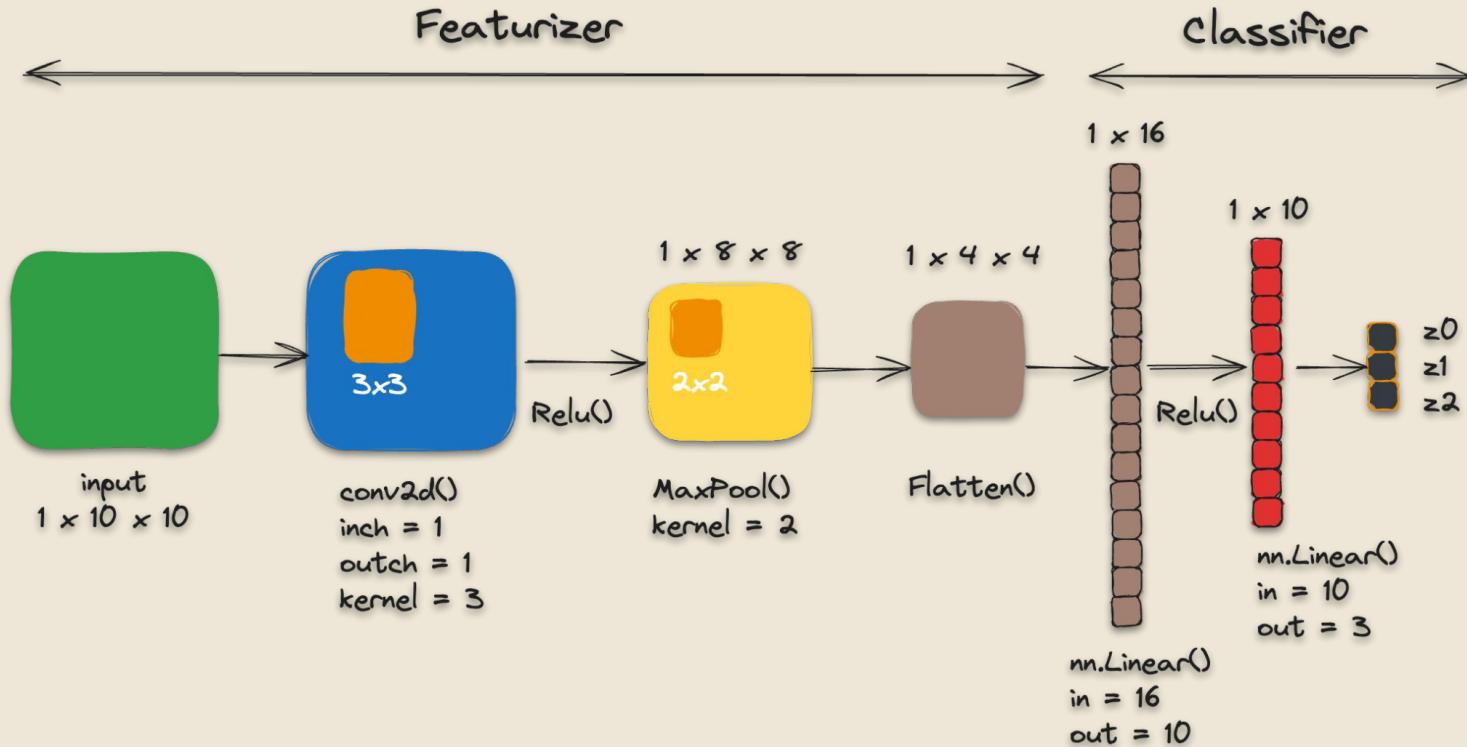
$$H(p, q) = - \sum_{i=0}^{C-1} p_i \log(q_i) \quad q_i = \frac{e^{z_i}}{\sum_{i=0}^{C-1} e^{z_i}}$$

```
torch.manual_seed(11)
dummy_logits = torch.randn(5, 3)
dummy_labels = torch.tensor([0, 0, 1, 2, 1])

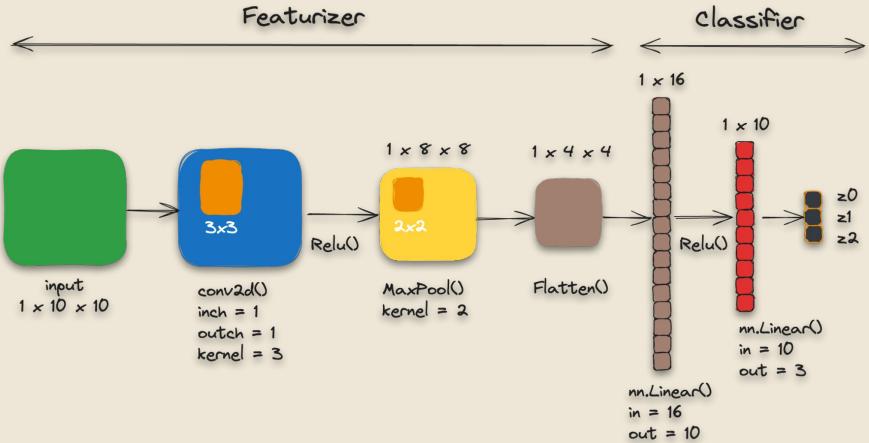
loss_fn = nn.CrossEntropyLoss()
loss_fn(dummy_logits, dummy_labels)
tensor(1.6553)
```

```
dummy_logits
tensor([[ 0.7376,  1.9459, -0.6995],
        [-1.3023, -0.5133, -0.2696],
        [ 0.2462,  0.4839,  0.4504],
        [-0.9568,  1.5012, -0.3136],
        [-0.2343, -1.0713,  0.1648]])
```

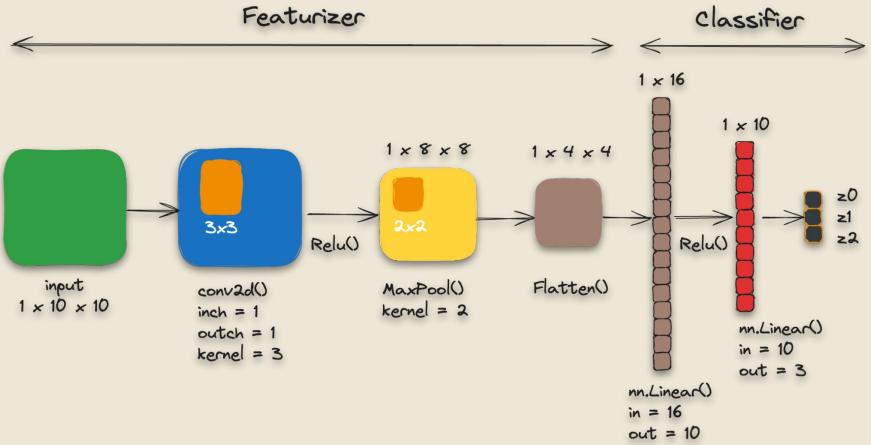
Case Study



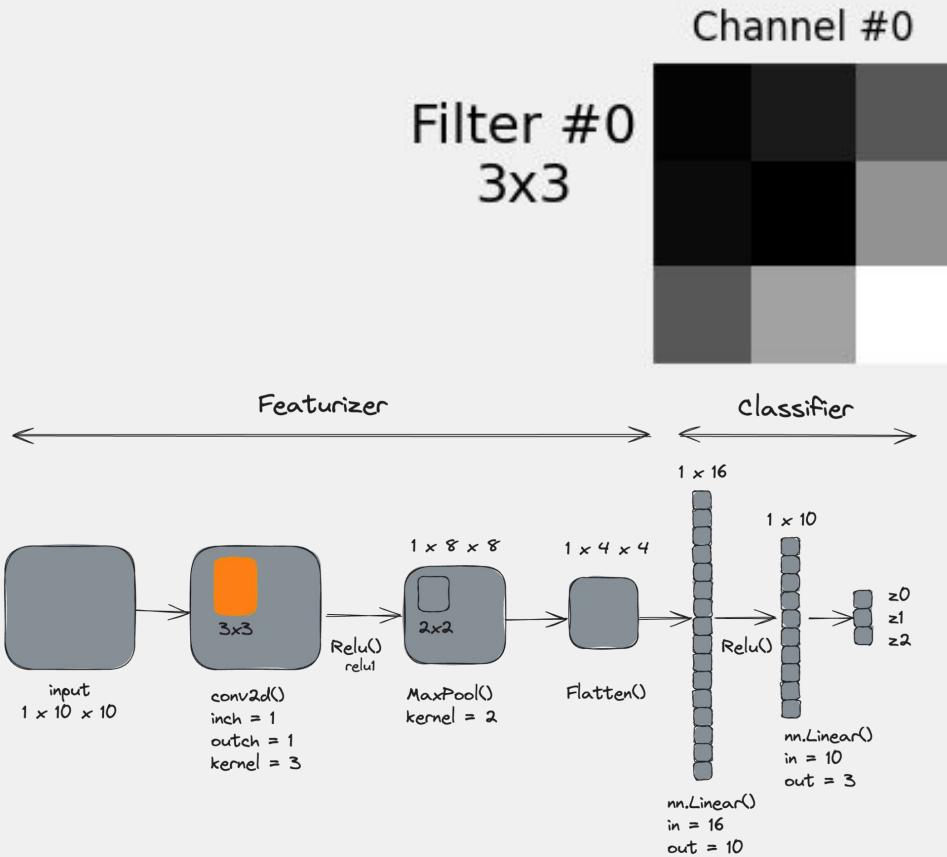
Case Study



Case Study

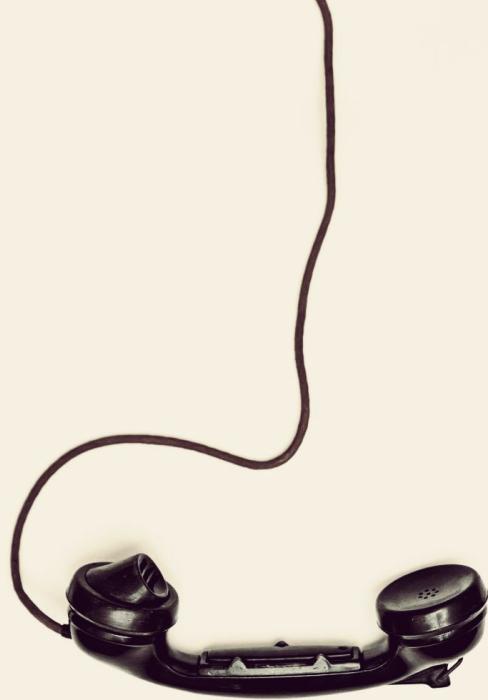


```
# Classification
# Hidden Layer
model_cnn1.add_module('fc1', nn.Linear(in_features=n_channels*4*4, out_features=10))
model_cnn1.add_module('relu2', nn.ReLU())
# Output Layer
model_cnn1.add_module('fc2', nn.Linear(in_features=10, out_features=3))
```



To really understand the effect this filter has on each image, we need to visualize the intermediate values produced by our model.

VISUALIZING FILTERS



A **hook** is simply a way to force a model to execute a function (callback) either after its **forward pass** or after its **backward pass**. Hence, there are forward hooks and backward hooks.

```
# Create a simple linear model with one input and one output
dummy_model = nn.Linear(1, 1)

# Create an empty list to store hook information
dummy_list = []

# Define the hook function
# This function will be called during the forward pass of the network
# The function receives three parameters:
# - layer: the current layer where the hook is registered
# - inputs: the input to the layer
# - outputs: the output from the layer
def dummy_hook(layer, inputs, outputs):
    # Append a tuple containing the layer, the input, and the output to the dummy_list
    dummy_list.append((layer, inputs, outputs))

# Register the hook to the model's layer
# During the model's execution, the hook function will be called
handle = dummy_model.register_forward_hook(dummy_hook)

# Now, whenever the dummy_model layer is executed, the dummy_hook function will be called
# and the layer, input, and output will be added to dummy_list
```

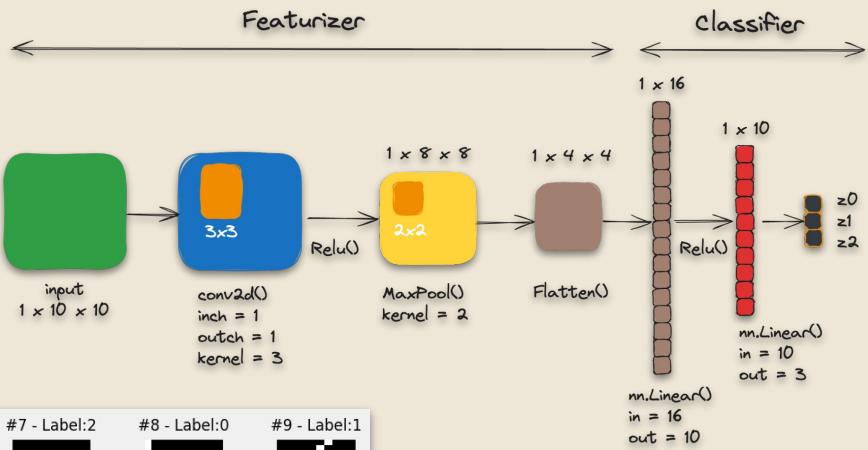
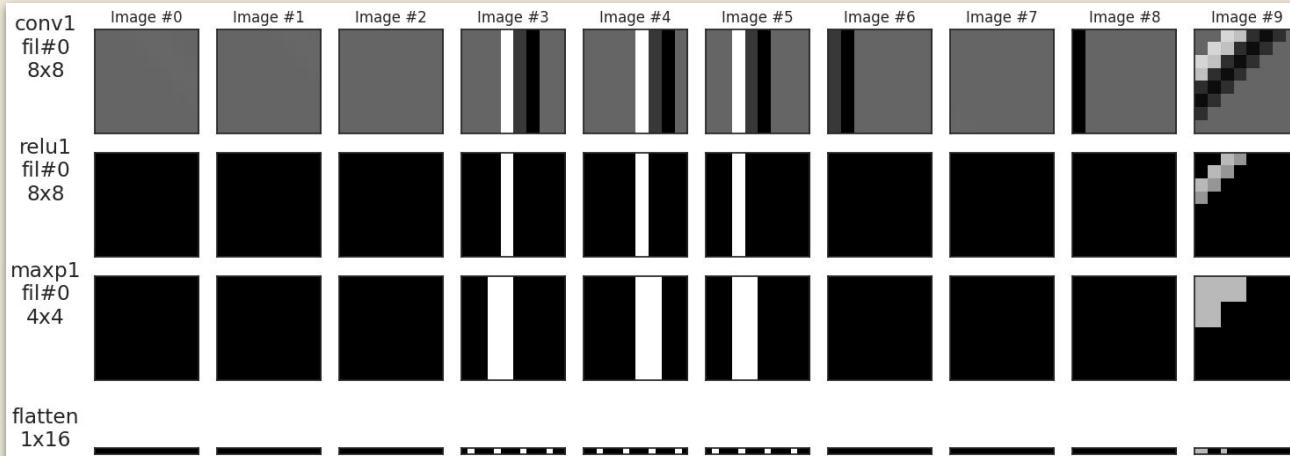
```
# Create a dummy input tensor with a value of 0.3
dummy_x = torch.tensor([0.3])

# Pass the dummy input through the dummy_model
dummy_model(dummy_x)
# Output of the model, which is a tensor with a value and a gradient function
# tensor([-0.2450], grad_fn=<ViewBackward0>)

# Check the dummy_list to see the stored hook information
dummy_list
# The list contains a tuple with:
# - The Linear layer used in the dummy_model
# - The input tensor passed to the layer (dummy_x)
# - The output tensor from the layer with a gradient function
# [(Linear(in_features=1, out_features=1, bias=True),
#   (tensor([0.3000]),),
#   tensor([-0.2450], grad_fn=<ViewBackward0>))]

# Remove the hook from the model
handle.remove()
```

Visualizing Feature Maps

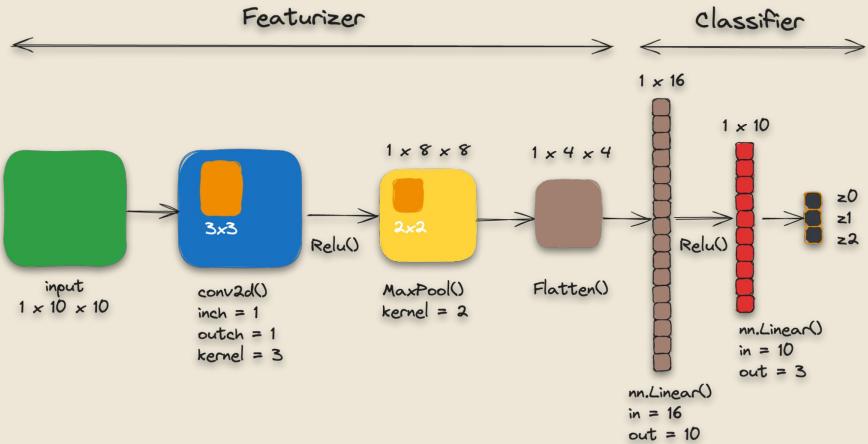


```
images_batch, labels_batch = next(iter(val_loader))
logits = arch_cnn1.predict(images_batch)
fig = plot_images(images_batch.squeeze(),
                   labels_batch.squeeze(),
                   n_plot=10)
```

```
featurizer_layers = ['conv1', 'relu1',
                     'maxp1', 'flatten']

with plt.style.context('seaborn-white'):
    fig = arch_cnn1.visualize_outputs(featurizer_layers)
```

Visualizing Feature Maps



Label: 2 Predicted: 2	Label: 2 Predicted: 2	Label: 2 Predicted: 2	Label: 0 Predicted: 0	Label: 0 Predicted: 0	Label: 0 Predicted: 0	Label: 0 Predicted: 2	Label: 2 Predicted: 2	Label: 0 Predicted: 2	Label: 1 Predicted: 1
--------------------------	--------------------------	--------------------------	--------------------------	--------------------------	--------------------------	--------------------------	--------------------------	--------------------------	--------------------------

#accuracy in this batch

```
arch_cnn1.correct(images_batch,
                   labels_batch)
tensor([[5, 7],
       [3, 3],
       [6, 6]])
```

```
classifier_layers = ['fc1', 'relu2',
                     'fc2']

with plt.style.context('seaborn-white'):
    fig = arch_cnn1.visualize_outputs(classifier_layers)
```

Accuracy

What if I want to compute the accuracy for all mini-batches in a data loader?

```
@staticmethod
def loader_apply(loader, func, reduce='sum'):
    results = [func(x, y) for i, (x, y) in enumerate(loader)]
    results = torch.stack(results, axis=0)

    if reduce == 'sum':
        results = results.sum(axis=0)
    elif reduce == 'mean':
        results = results.float().mean(axis=0)

    return results
```

```
## Loader Apply
Architecture.loader_apply(arch_cnn1.val_loader,
                           arch_cnn1.correct)
tensor([[59, 67],
        [55, 62],
        [71, 71]])
```