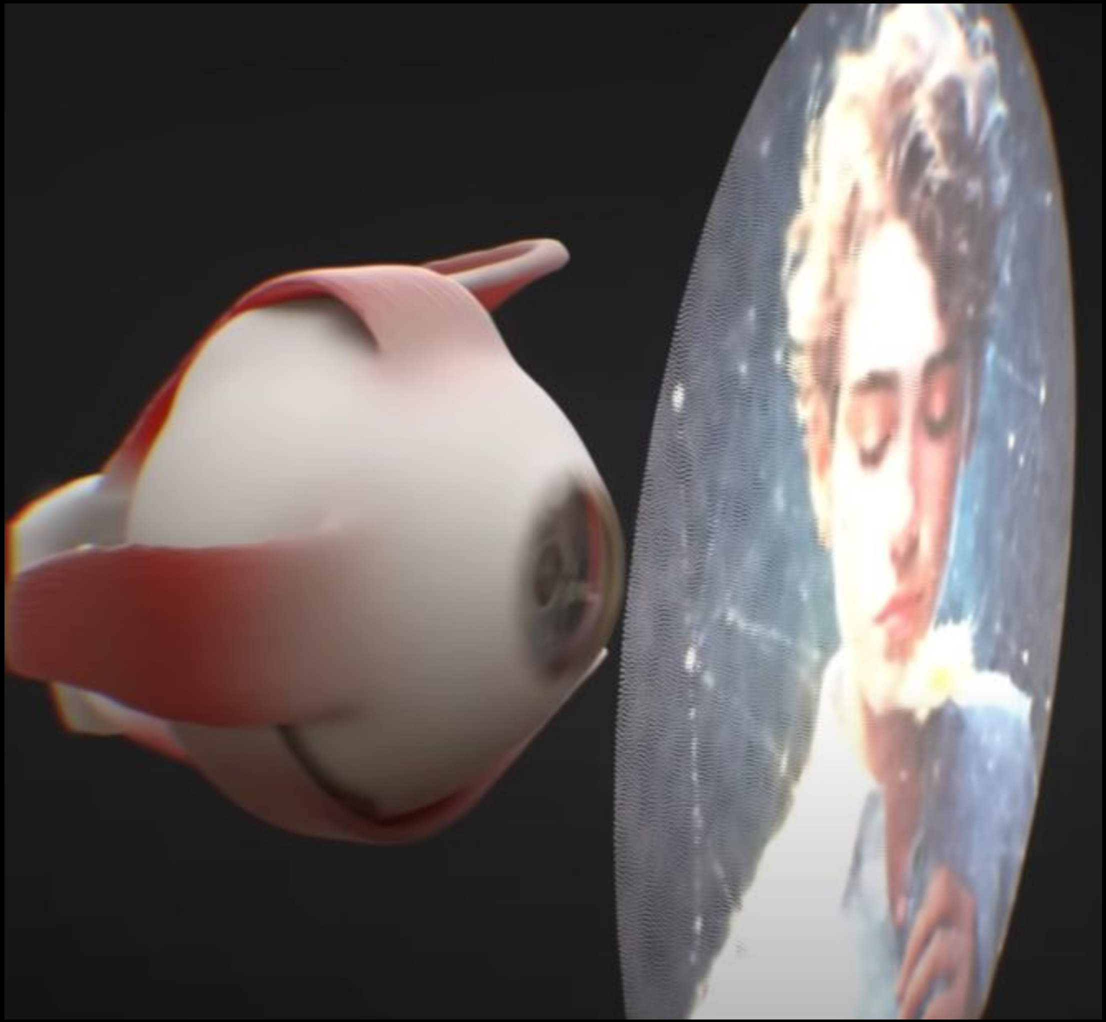
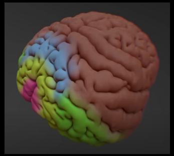
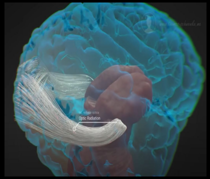
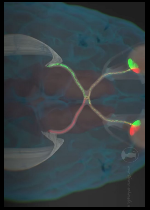
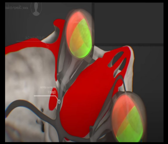
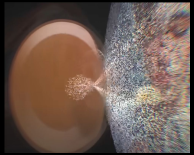
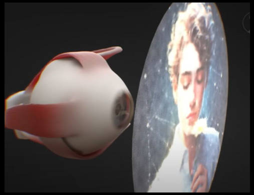
*"... Picture says a thousand words ..."*

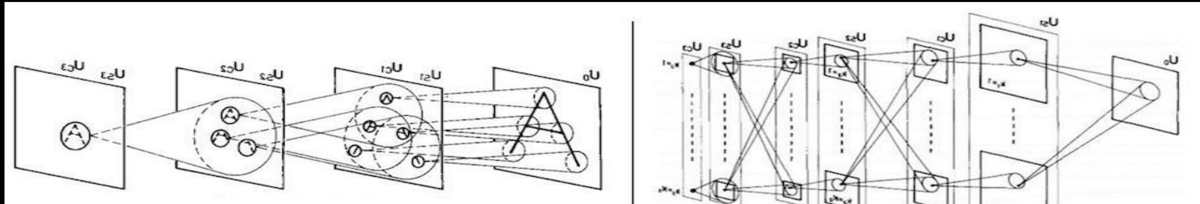
# Convolutional Neural Network

Arghya Pal
FIT5215

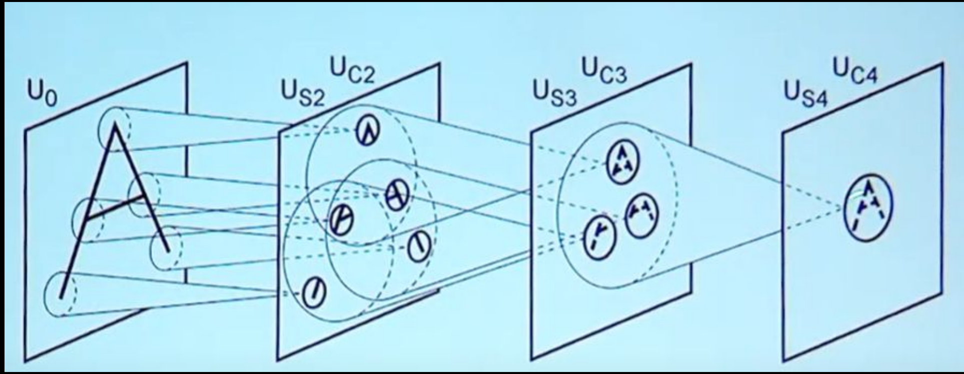


# NOTE



Neocognitron 1982, Fukushima




## Simple Cell (S):

Extract features, filter

## Complex Cell (C):

Shift tolerance, pooling

References

1. https://ml4a.github.io/ml4a/convnets/
2.

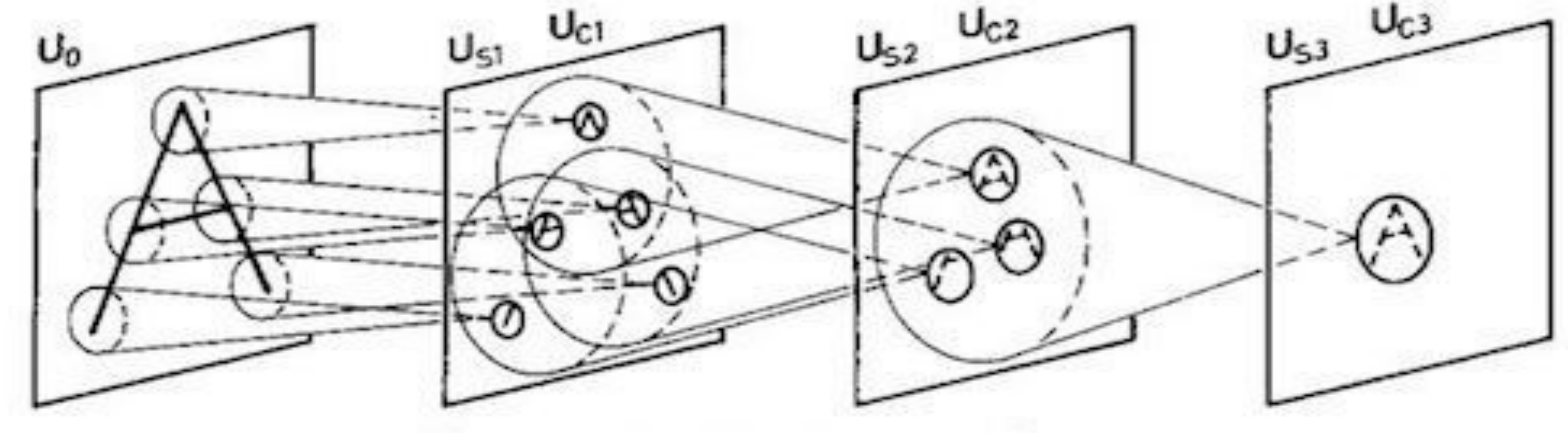



Image Processing Zone

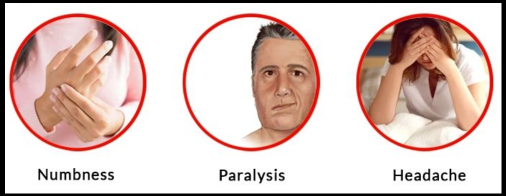
Computer Vision Zone

Deep Learning Zone

*how* to process the pixels.

Extract *meaning* from visual data.

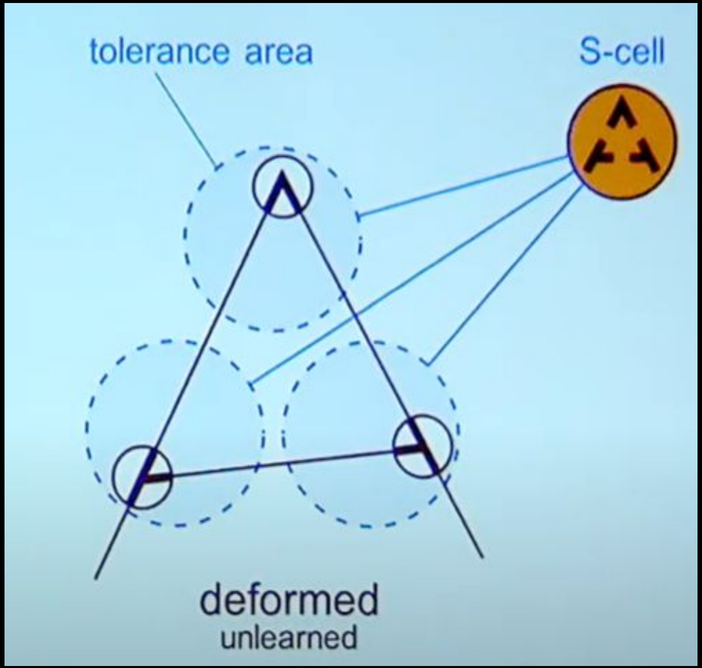
Decision



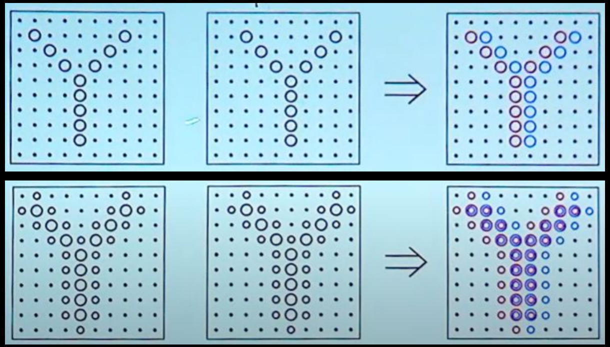
# Neocognitron
## *how* to process the pixels

## Tutorial 3a

# Making of Complex Cell




## A. Shift → Blur



## B. Size → Pooling

NOTE

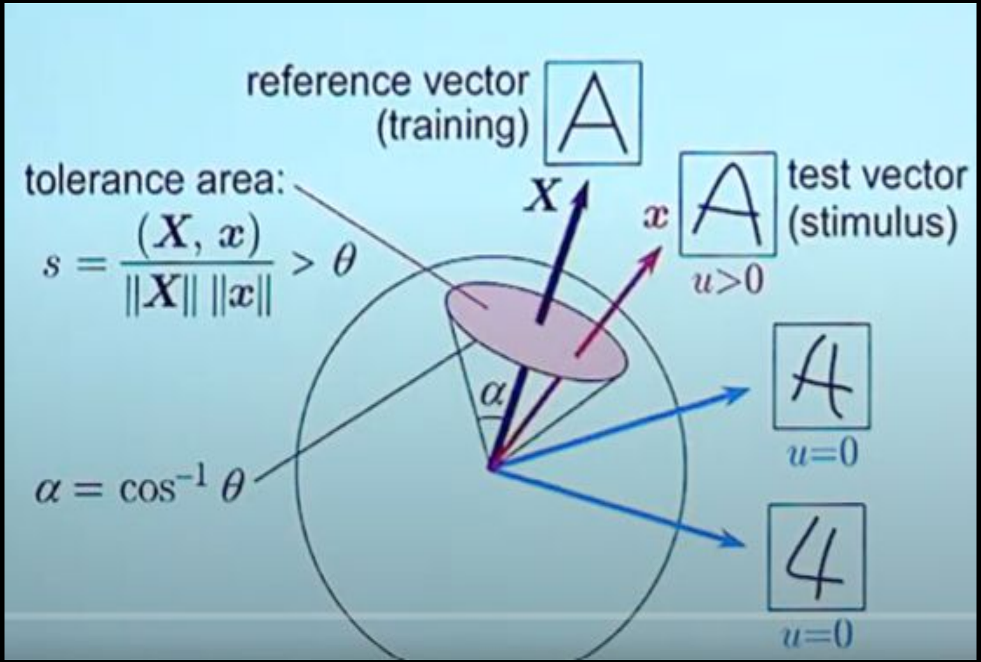

References

1. https://ml4a.github.io/ml4a/convnets/
2.


MONASH University

# Neocognitron: Extract *meaning* from visual data

猫
बिल्ली
بلی
chat
Katze
বিড়াল

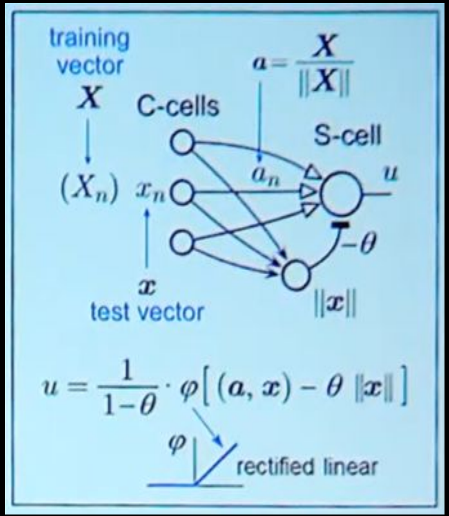


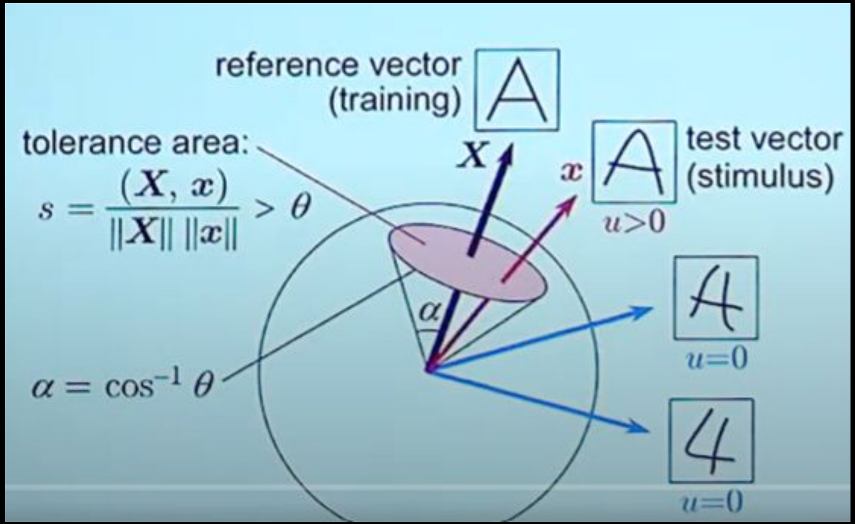

NOTE

References

# Neocognitron: Extract *meaning* from visual data

NOTE

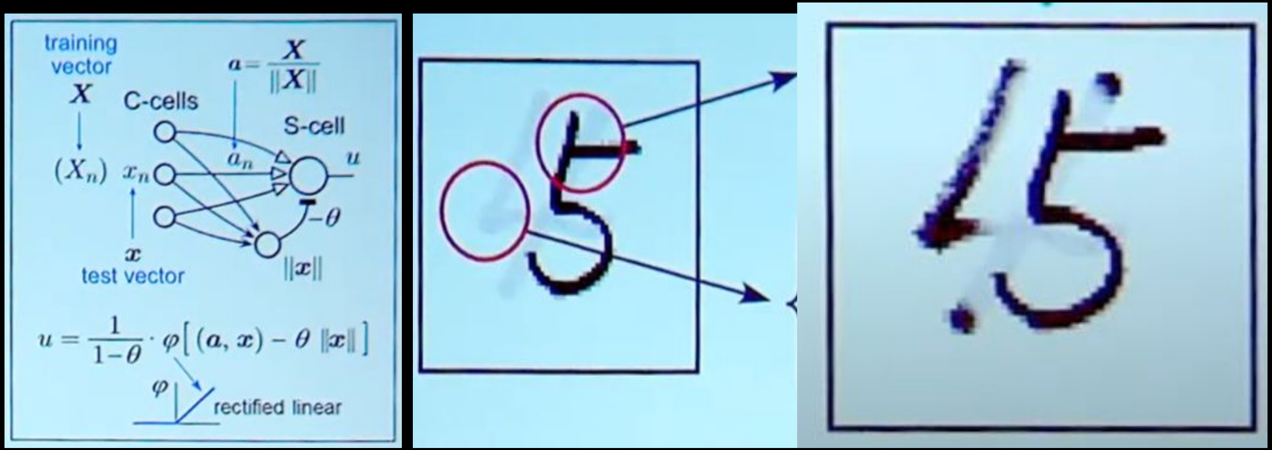



References

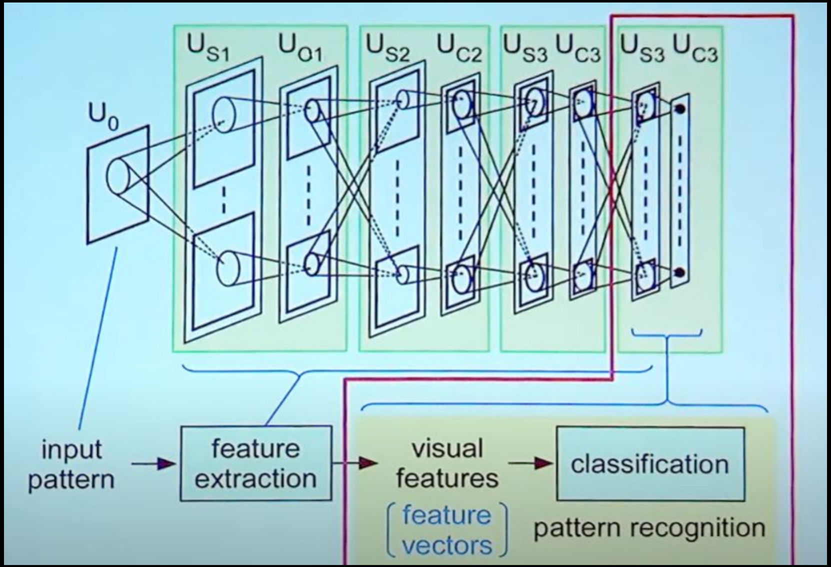
# Neocognitron: Extract *meaning* from visual data

NOTE




References

# Neocognitron: Decision

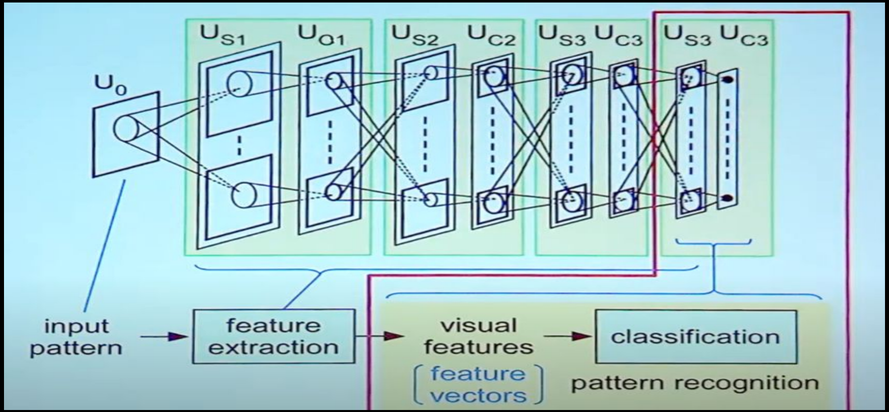


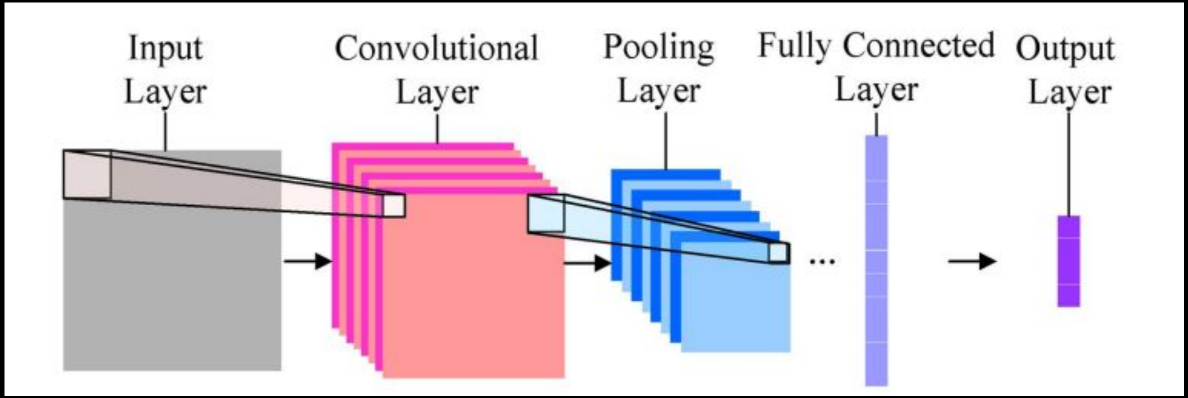

NOTE

References

MONASH University

# From Neocognitron to Convolutional Neural Network





MONASH University

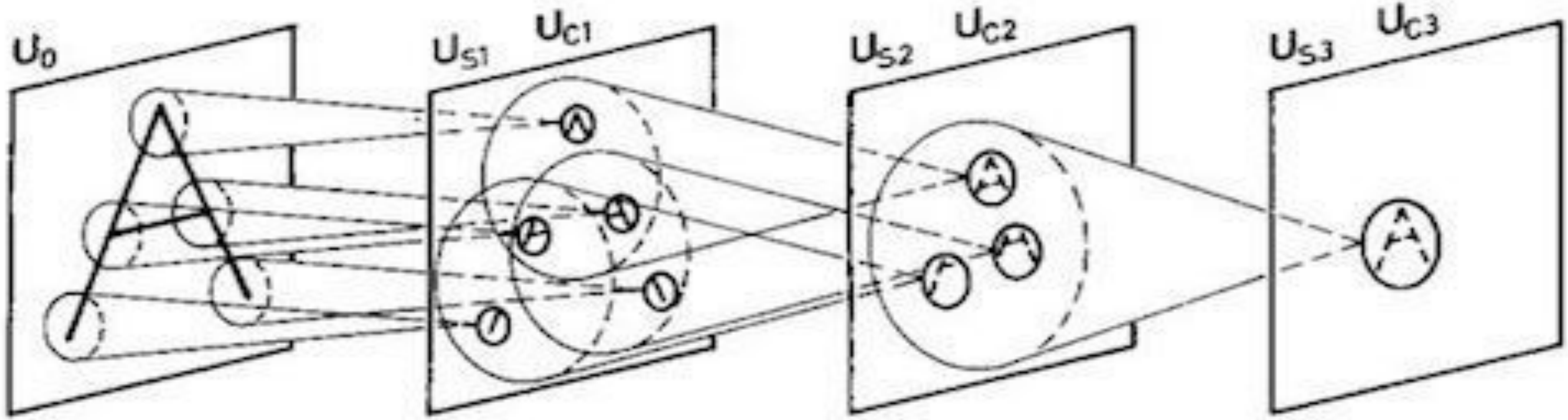



Image Processing Zone

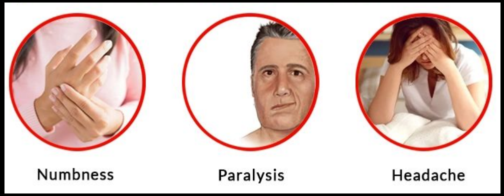
*how* to process the pixels.

Computer Vision Zone

Extract *meaning* from visual data.

Deep Learning Zone

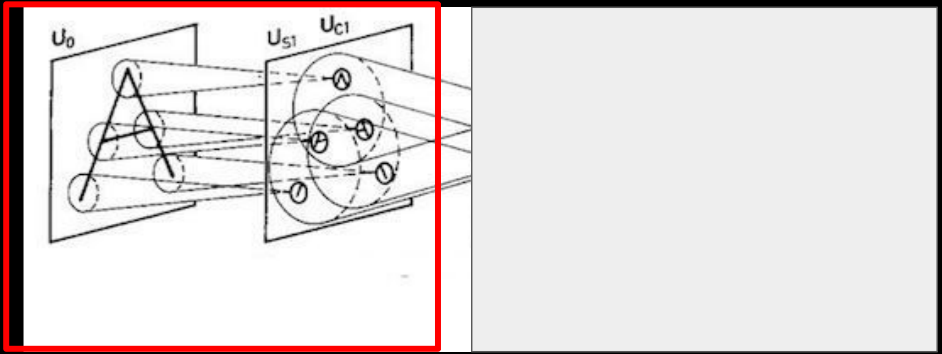
Decision

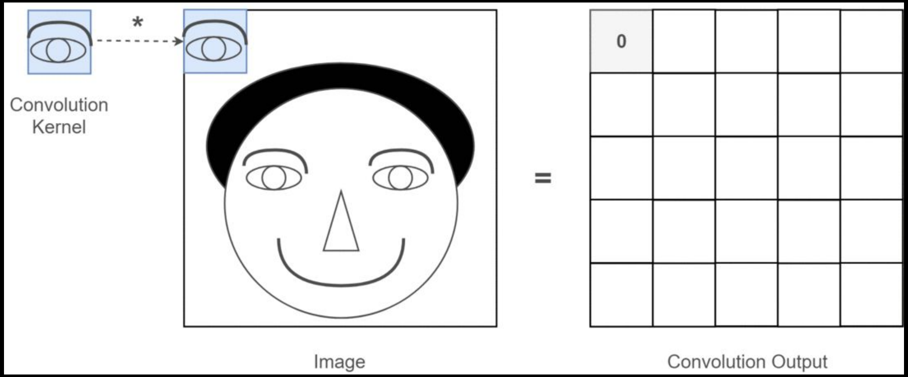


# Convolutional Neural Network: *how* to process the pixels

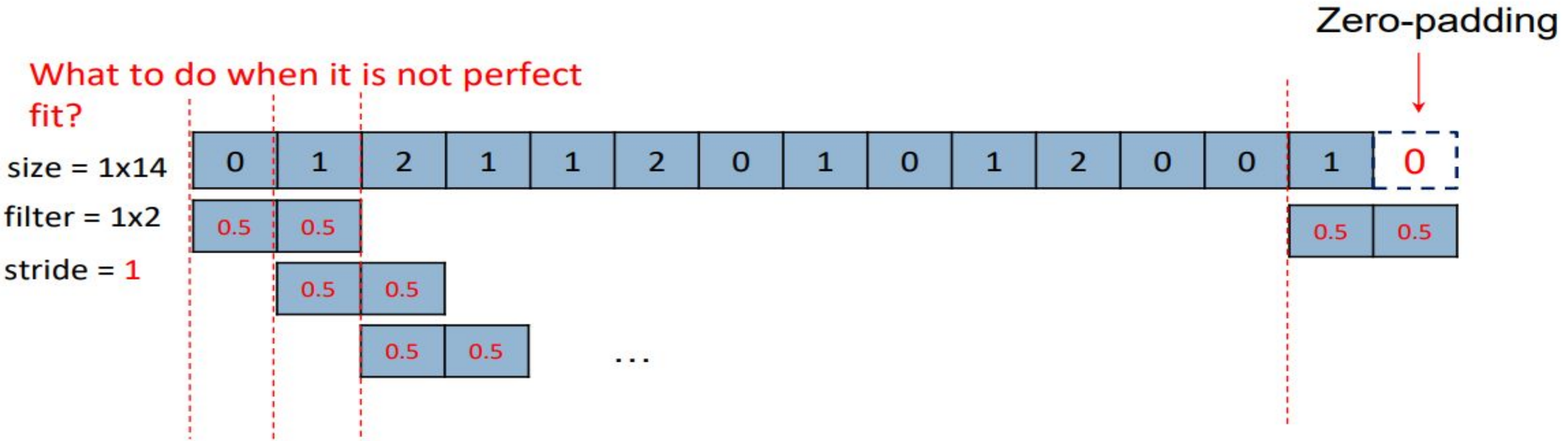
## Tutorial 3a

# Instead of S and C cells we have Convolutional Operation




NOTE


Zero-padding
What to do when it is not perfect fit?
size = 1x14
0
1
2
1
1
2
0
1
0
1
2
0
0
1
0
filter = 1x2
0.5
0.5
stride = 1
0.5
0.5
0.5
0.5
...
0.5
0.5

# Convolution layer with zero padding

**$x$ (input tensor (7,7))** **strides** = (2,2)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | -2 | -1 | 5 | 3 | 2 | 1 | 0 |
| 0 | 1 | 3 | -1 | 4 | 3 | 3 | 1 | 0 |
| 0 | 1 | -2 | 1 | 6 | 3 | 3 | 2 | 0 |
| 0 | 2 | -2 | 2 | 5 | 2 | 1 | 0 | 0 |
| 0 | 0 | 3 | -2 | 5 | 4 | 1 | 2 | 0 |
| 0 | 1 | 2 | -3 | 1 | 1 | 2 | -1 | 0 |
| 0 | 1 | -2 | -1 | 1 | 2 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

$H_i = 7$

$W_i$=7 $p = 1$ **zero padding**

**kernel size** =(3,3)
**$W$ (filter or kernel)**

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

$f_h = 3$

$f_w = 3$

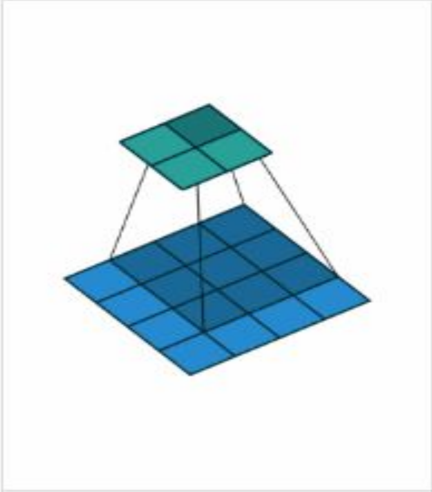
**feature map**

| | | | |
|---|---|---|---|
| 3 | 8 | 19 | 7 |
| 3 | 16 | 30 | 10 |
| 6 | 11 | 22 | 5 |
| 2 | -2 | 8 | 3 |

$H_o = 4$

$W_o = 4$

- The **sliding window** moves from **left** to **right**, **top** to **bottom** with strides.
- We **convolve** the **filter** and the **sliding windows** to work out the **neurons** on the **feature map**.



# Convolution layer with zero padding

***x*** **(input tensor (7,7))** **strides** = (2,2)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | -2 | -1 | 5 | 3 | 2 | 1 | 0 |
| 0 | 1 | 3 | -1 | 4 | 3 | 3 | 1 | 0 |
| 0 | 1 | -2 | 1 | 6 | 3 | 3 | 2 | 0 |
| 0 | 2 | -2 | 2 | 5 | 2 | 1 | 0 | 0 |
| 0 | 0 | 3 | -2 | 5 | 4 | 1 | 2 | 0 |
| 0 | 1 | 2 | -3 | 1 | 1 | 2 | -1 | 0 |
| 0 | 1 | -2 | -1 | 1 | 2 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

$H_i = 7$

$W_i$=7

$p = 1$ **zero padding**

**kernel size** =(3,3)

***W*** **(filter or kernel)**

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

$f_h = 3$

$f_w = 3$

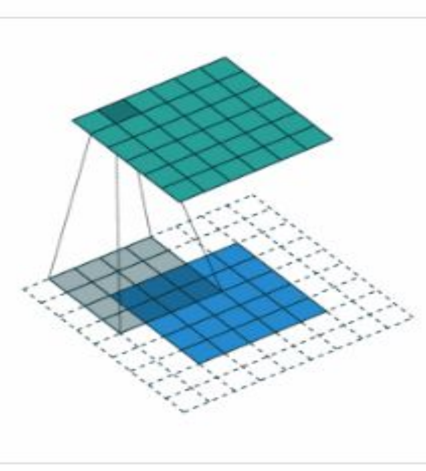
**feature map**

| | | | |
|---|---|---|---|
| 3 | 8 | 19 | 7 |
| 3 | 15 | 30 | 10 |
| 6 | 11 | 22 | 5 |
| 2 | -2 | 9 | 3 |

$H_o = 4$

$W_o = 4$

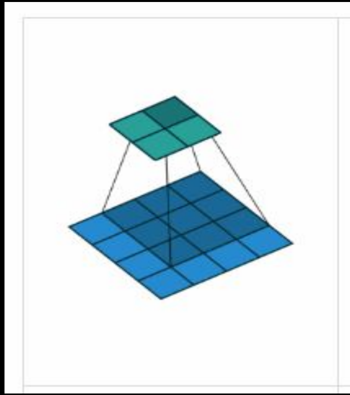
- The **sliding window** moves from **left** to **right**, **top** to **bottom** with strides.
- We **convolve** the **filter** and the **sliding windows** to work out the **neurons** on the **feature map**.



# Padding

"Valid": $n \times n \ast f \times f \rightarrow n-f+1 \times n-f+1$

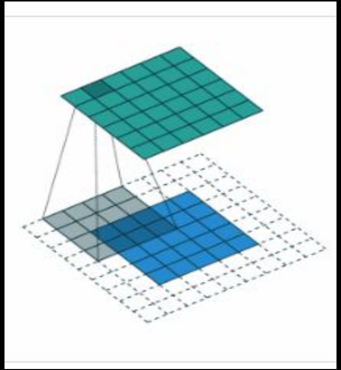
no padding



"Same": Pad so that output size is the same as the input size.

$n+2p-f+1 \times n+2p-f+1$

$\cancel{n}+2p-f+1 = \cancel{n} \Rightarrow p = \frac{f-1}{2}$



Andrew Ng

# Padding

```
m = nn.CircularPad2d(2)
input = torch.arange(9,
dtype=torch.float).reshape(1, 1, 3, 3)
input
tensor([[[[0., 1., 2.],
          [3., 4., 5.],
          [6., 7., 8.]]]])
m(input)
tensor([[[[4., 5., 3., 4., 5., 3., 4.],
          [7., 8., 6., 7., 8., 6., 7.],
          [1., 2., 0., 1., 2., 0., 1.],
          [4., 5., 3., 4., 5., 3., 4.],
          [7., 8., 6., 7., 8., 6., 7.],
          [1., 2., 0., 1., 2., 0., 1.],
          [4., 5., 3., 4., 5., 3., 4.]]]])
# using different paddings for different sides
m = nn.CircularPad2d((1, 1, 2, 0))
m(input)
tensor([[[[5., 3., 4., 5., 3.],
          [8., 6., 7., 8., 6.],
          [2., 0., 1., 2., 0.],
          [5., 3., 4., 5., 3.],
          [8., 6., 7., 8., 6.]]]])
```

```
m = nn.ConstantPad2d(2, 3.5)
input = torch.randn(1, 2, 2)
input
tensor([[[ 1.6585,  0.4320],
         [-0.8701, -0.4649]]])
m(input)
tensor([[[ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  1.6585,  0.4320,  3.5000,  3.5000],
         [ 3.5000,  3.5000, -0.8701, -0.4649,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000]]])
# using different paddings for different sides
m = nn.ConstantPad2d((3, 0, 2, 1), 3.5)
m(input)
tensor([[[ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  1.6585,  0.4320],
         [ 3.5000,  3.5000,  3.5000, -0.8701, -0.4649],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000]]])
```

padding_left, padding_right, Padding_top, padding_bottom

# Padding

```
m = nn.ReflectionPad2d(2)
input = torch.arange(9,
dtype=torch.float).reshape(1, 1, 3, 3)
input
tensor([[[[0., 1., 2.],
          [3., 4., 5.],
          [6., 7., 8.]]]])
m(input)
tensor([[[[8., 7., 6., 7., 8., 7., 6.],
          [5., 4., 3., 4., 5., 4., 3.],
          [2., 1., 0., 1., 2., 1., 0.],
          [5., 4., 3., 4., 5., 4., 3.],
          [8., 7., 6., 7., 8., 7., 6.],
          [5., 4., 3., 4., 5., 4., 3.],
          [2., 1., 0., 1., 2., 1., 0.]]]])
# using different paddings for different
sides
m = nn.ReflectionPad2d((1, 1, 2, 0))
m(input)
tensor([[[[7., 6., 7., 8., 7.],
          [4., 3., 4., 5., 4.],
          [1., 0., 1., 2., 1.],
          [4., 3., 4., 5., 4.],
          [7., 6., 7., 8., 7.]]]])
```

```
m = nn.ReplicationPad2d(2)
input = torch.arange(9,
dtype=torch.float).reshape(1, 1, 3, 3)
input
tensor([[[[0., 1., 2.],
          [3., 4., 5.],
          [6., 7., 8.]]]])
m(input)
tensor([[[[0., 0., 0., 1., 2., 2., 2.],
          [0., 0., 0., 1., 2., 2., 2.],
          [0., 0., 0., 1., 2., 2., 2.],
          [3., 3., 3., 4., 5., 5., 5.],
          [6., 6., 6., 7., 8., 8., 8.],
          [6., 6., 6., 7., 8., 8., 8.],
          [6., 6., 6., 7., 8., 8., 8.]]]])
# using different paddings for different
sides
m = nn.ReplicationPad2d((1, 1, 2, 0))
m(input)
tensor([[[[0., 0., 1., 2., 2.],
          [0., 0., 1., 2., 2.],
          [0., 0., 1., 2., 2.],
          [3., 3., 4., 5., 5.],
          [6., 6., 7., 8., 8.]]]])
```

padding_left, padding_right, Padding_top, padding_bottom

# Convolution 2D without zero padding

$H_i = 7$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | -2 | -1 | 5 | 3 | 2 | 1 | 1 |
| 1 | 3 | -1 | 4 | 3 | 3 | 1 | -1 |
| 1 | -2 | 1 | 6 | 3 | 3 | 2 | 2 |
| 2 | -2 | 2 | 5 | 2 | 1 | 0 | -1 |
| 0 | 3 | -2 | 5 | 4 | 1 | 2 | 1 |
| 1 | 2 | -3 | 1 | 1 | 2 | -1 | 2 |
| 1 | -2 | -1 | 1 | 2 | 1 | 1 | 3 |

$x$ = tensor(7,8) $W_i = 8$

strides s = $(s_w, s_h)$ = (2,2)

kernel size =(3,3)

$f_h = 3$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

$f_w = 3$

feature map

$H_o = 3$

| | | |
|---|---|---|
| 1 | 23 | 21 |
| 3 | 26 | 17 |
| -1 | 8 | 13 |

$W_o = 3$

- $W_i, H_i$: The width and height of the **input** image
- $W_o, H_o$: The width and height of the **output** image (feature map)

$$W_o = \left\lfloor \frac{W_i + 2p - f_w}{s_w} \right\rfloor + 1 \text{ and } H_o = \left\lfloor \frac{H_i + 2p - f_h}{s_h} \right\rfloor + 1$$

- Our case: $W_o = \left\lfloor \frac{8+0-3}{2} \right\rfloor + 1 = 3$ and $H_o = \left\lfloor \frac{7+0-3}{2} \right\rfloor + 1 = 3$

# Convolution 2D with zero padding

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | -2 | -1 | 5 | 3 | 2 | 1 | 0 |
| 0 | 1 | 3 | -1 | 4 | 3 | 3 | 1 | 0 |
| 0 | 1 | -2 | 1 | 6 | 3 | 3 | 2 | 0 |
| 0 | 2 | -2 | 2 | 5 | 2 | 1 | 0 | 0 |
| 0 | 0 | 3 | -2 | 5 | 4 | 1 | 2 | 0 |
| 0 | 1 | 2 | -3 | 1 | 1 | 2 | -1 | 0 |
| 0 | 1 | -2 | -1 | 1 | 2 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

padding = 1

padding = 1

$x$ = tensor(7,7)
strides s = $(s_w, s_h)$ = (2,2)
zero-padding $p = 1$

kernel size =(3,3)

$f_h = 3$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

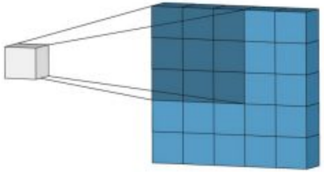
$f_w = 3$

feature

| | | | |
|---|---|---|---|
| 3 | 8 | 20 | 7 |
| 3 | 16 | 30 | 10 |
| 6 | 13 | 22 | 7 |
| 2 | -2 | 7 | 3 |

$H_o = 4$

$W_o = 4$



- $W_i, H_i$: The width and height of the **input** image
- $W_o, H_o$: The width and height of the **output** image (feature map)

$$W_o = \left\lfloor \frac{W_i + 2p - f_w}{s_w} \right\rfloor + 1 \text{ and } H_o = \left\lfloor \frac{H_i + 2p - f_h}{s_h} \right\rfloor + 1$$

- Our case: $W_o = \left\lfloor \frac{7+2\times1-3}{2} \right\rfloor + 1 = 4$ and $H_o = \left\lfloor \frac{7+2\times1-3}{2} \right\rfloor + 1 = 4$

(Source: https://towardsdatascience.com/)

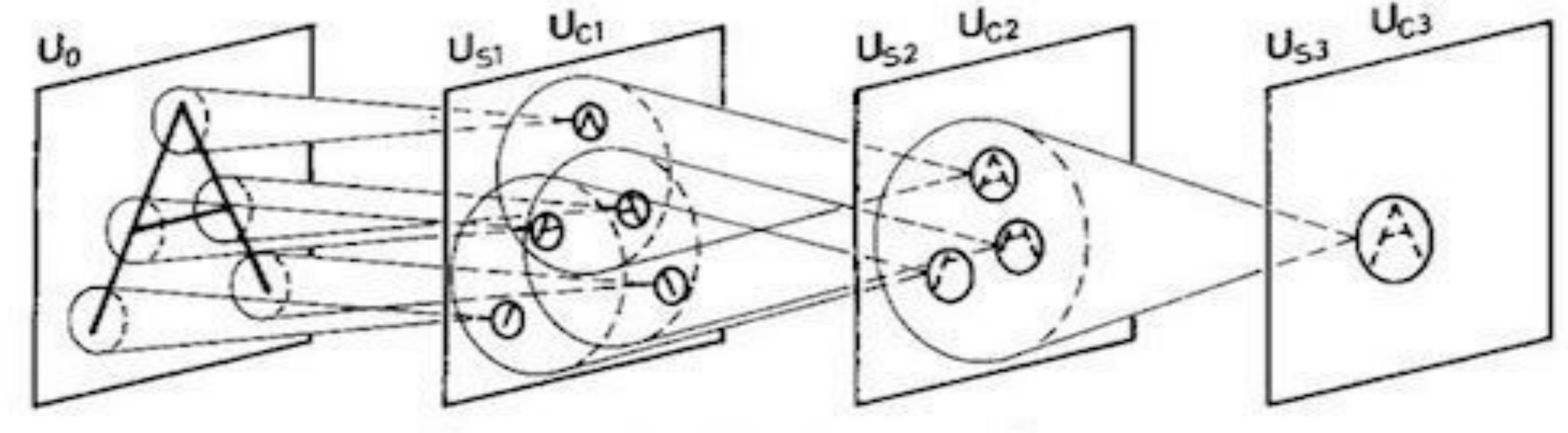



Image Processing Zone

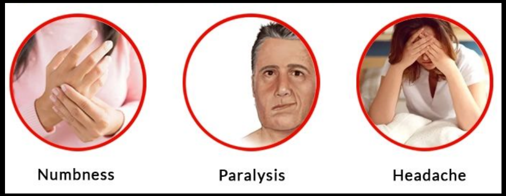
Computer Vision Zone

Deep Learning Zone

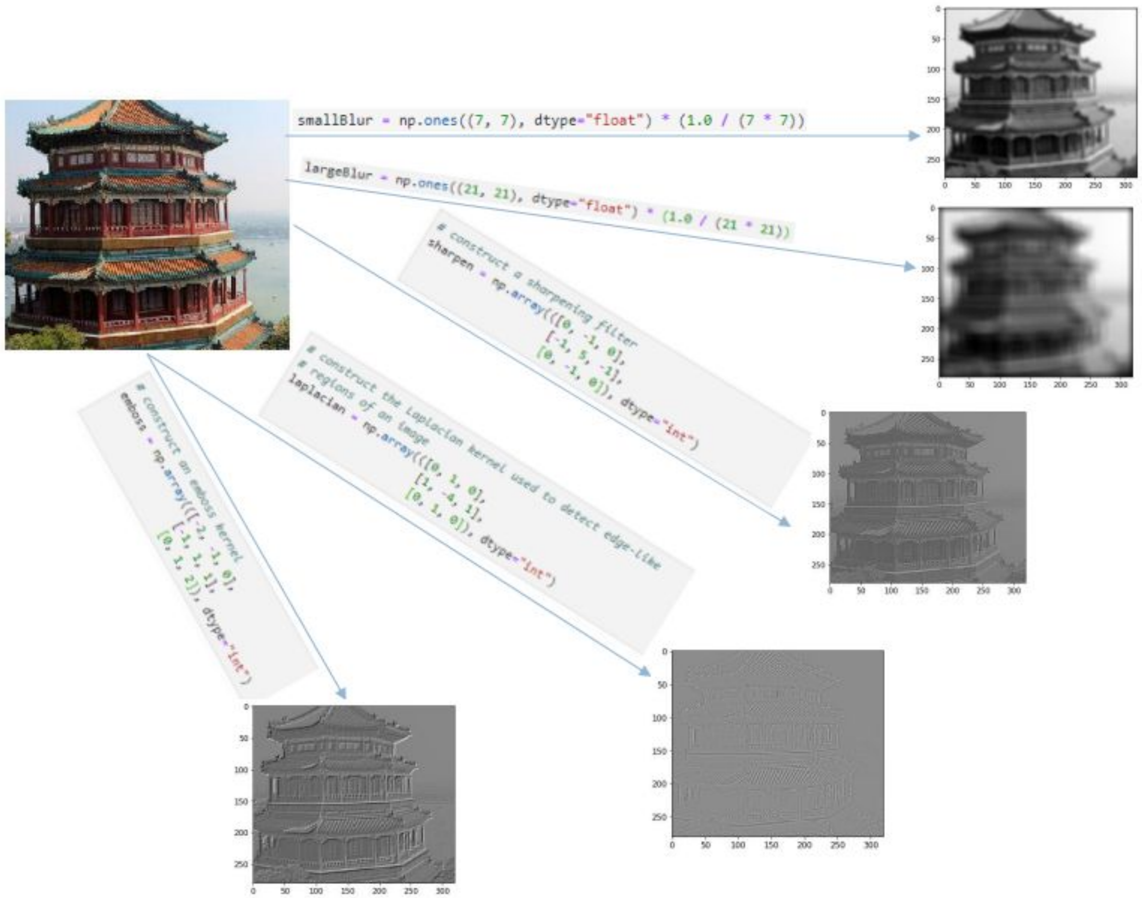
*how* to process the pixels.

Extract *meaning* from visual data.

Decision



# Effects of filters/kernels to images



# Example of pooling with PyTorch

**Max pooling**

```
output = torch.nn.functional.max_pool2d(input = batch_tensor,
                                        kernel_size=(2,2), stride= (2,2))
output = output.numpy()
print(batch[0].shape)
print(output[0].shape)
plot_image(batch[0].transpose((1,2,0)).astype(np.int32), scale=True) # plot the 1st original image
plot_image(output[0].transpose((1,2,0)).astype(np.int32), scale=True) # plot the output for the 1st image
```

```
(3, 280, 320)
(3, 140, 160)
```

**Average pooling**

```
output = torch.nn.functional.avg_pool2d(input = batch_tensor,
                                        kernel_size=(2,2), stride= (2,2))
output = output.numpy()
print(batch[0].shape)
print(output[0].shape)
plot_image(batch[0].transpose((1,2,0)).astype(np.int32), scale=True) # plot the 1st original image
plot_image(output[0].transpose((1,2,0)).astype(np.int32), scale=True) # plot the output for the 1st image
```

```
(3, 280, 320)
(3, 140, 160)
```

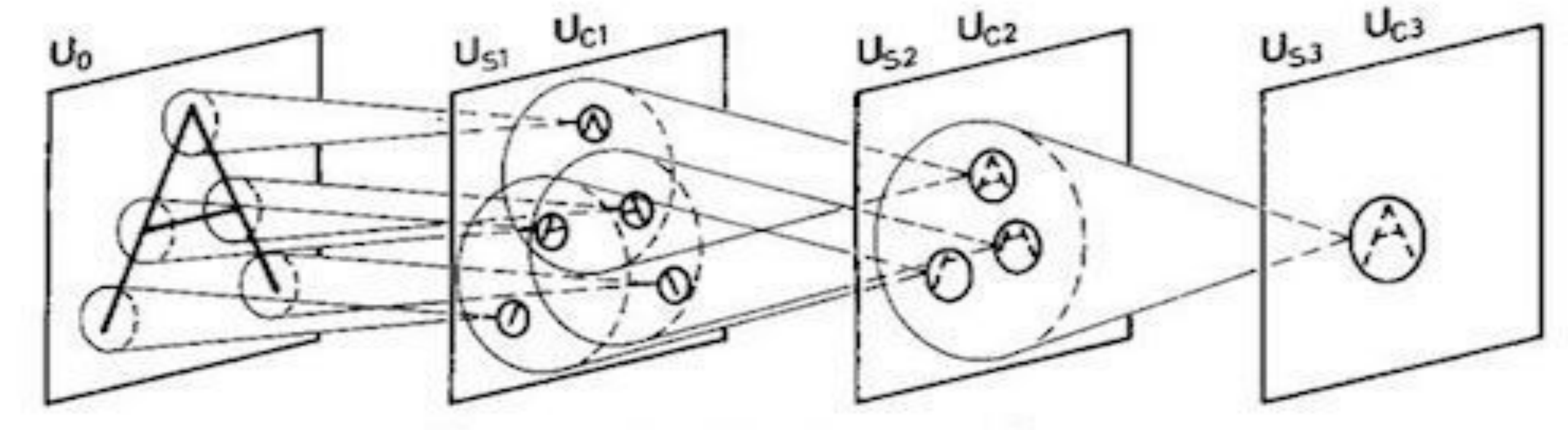



Image Processing Zone

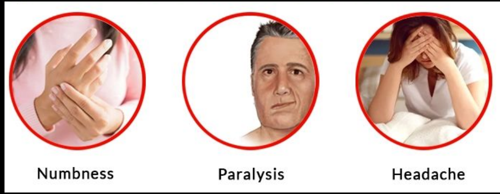
Computer Vision Zone

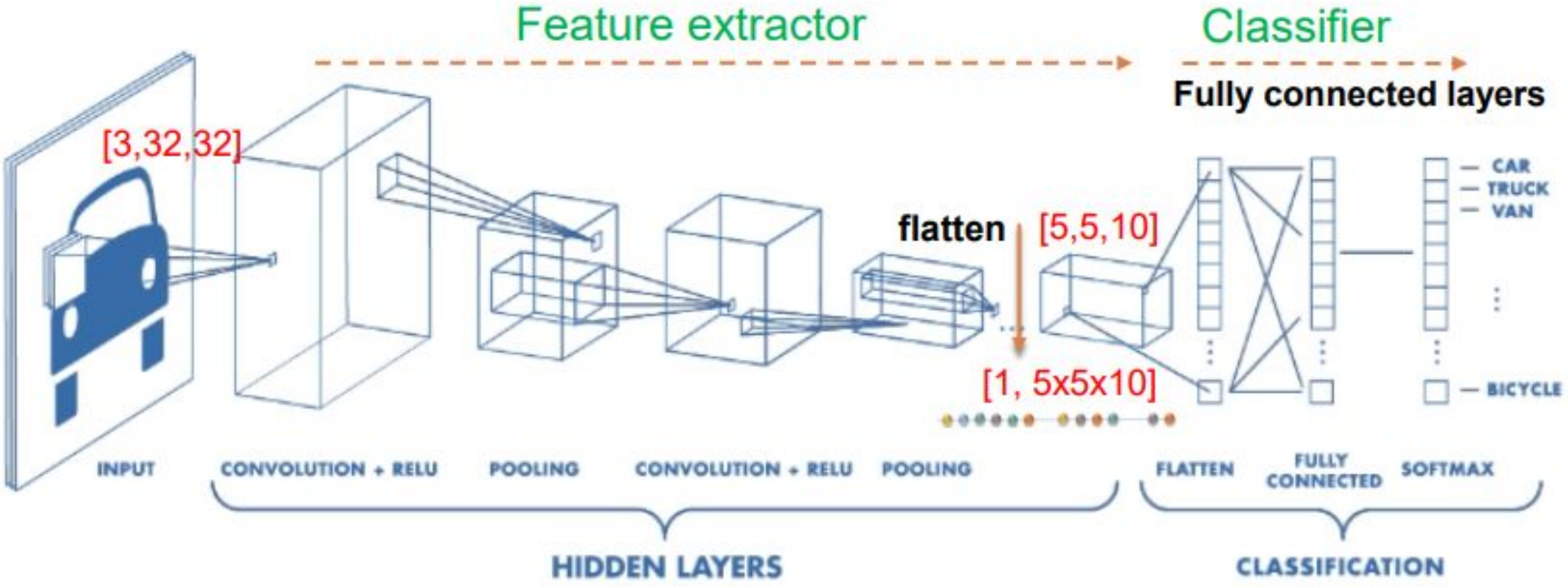
Deep Learning Zone

*how* to process the pixels.

Extract *meaning* from visual data.

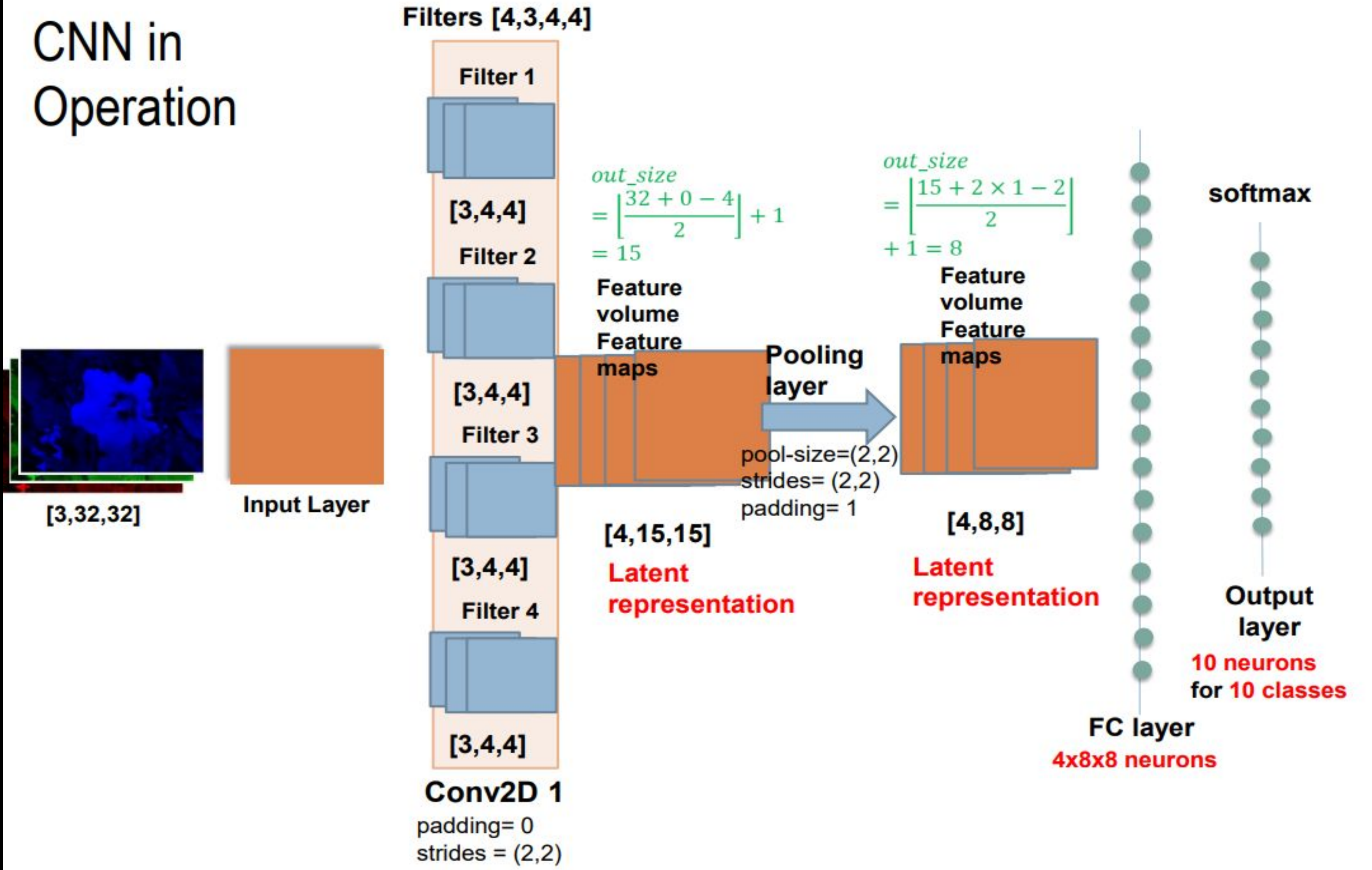
Decision





Feature extractor
Classifier
Fully connected layers
[3,32,32]
flatten
[5,5,10]
[1, 5x5x10]
CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
HIDDEN LAYERS
CLASSIFICATION

# CNN in Operation



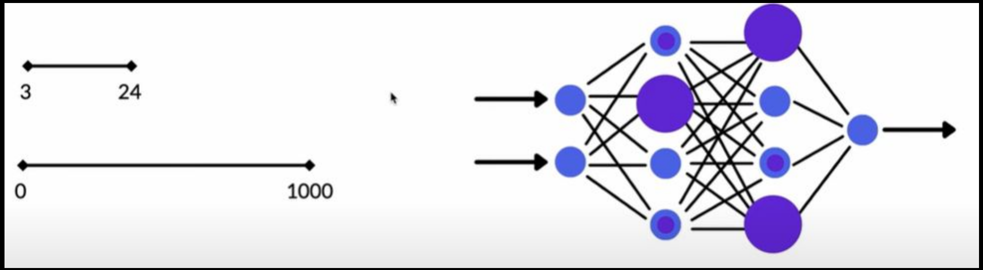
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

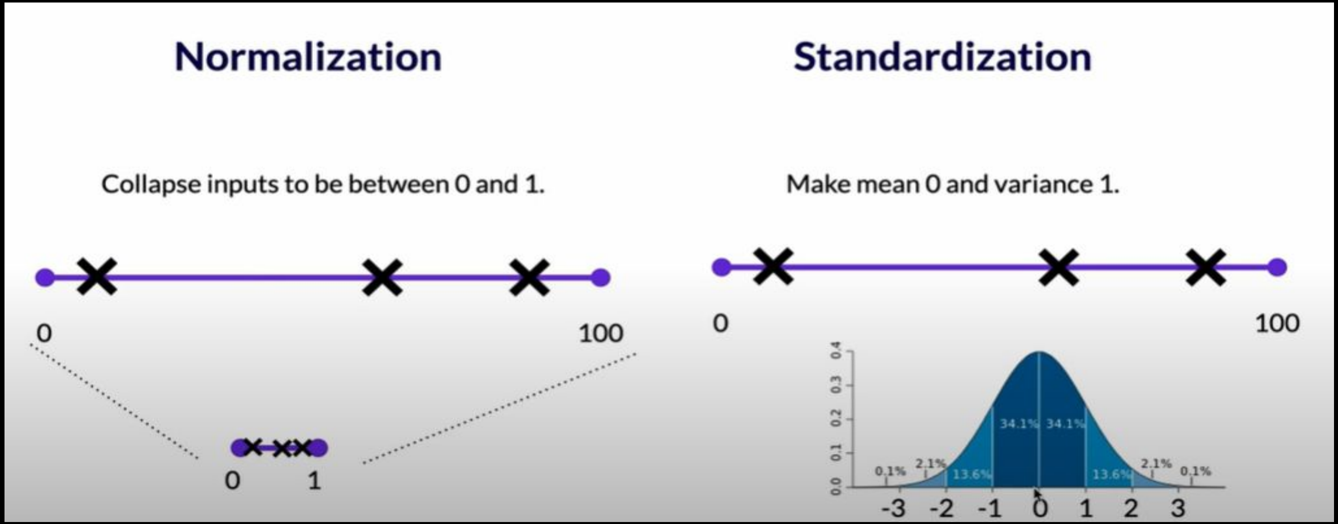
```
torch.nn.Conv2d(
in_channels,
out_channels,
kernel_size,
stride=1,
padding=0,
dilation=1,
groups=1,
bias=True,
padding_mode='zeros',
device=None,
dtype=None
)
```

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 3 \\ 2 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

# Tricks



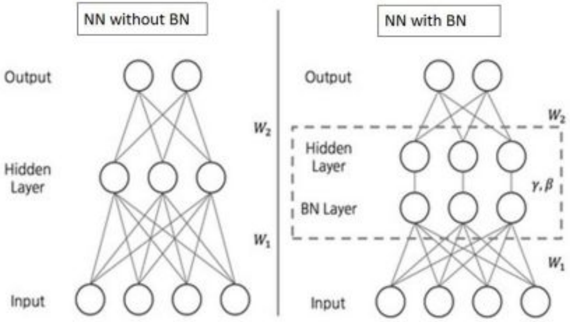

NOTE

References
1. https://www.youtube.com/watch?v=yXOMHOpbon8

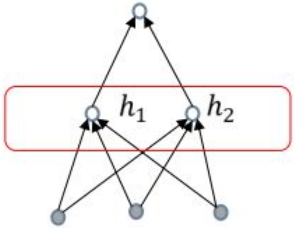
# Batch Normalization

1. Cope with internal covariate shift
2. Reduce gradient vanishing/exploding
3. Reduce overfitting
4. Make training more stable
5. Converge faster
    1. Allow us to train with bigger learning rate

- Let $z = W^k h^k + b^k$ be the mini-batch before activation. We compute the normalized $\hat{z}$ as
    - $\hat{z} = \frac{z-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}$ where $\epsilon$ is a small value such as $1e^{-7}$
    - $\mu_B = \frac{1}{b}\sum_{i=1}^{b} z_i$ is the empirical mean
    - $\sigma_B^2 = \frac{1}{b}\sum_{i=1}^{b}(z_i - \mu_B)^2$ is the empirical variance
- We scale the normalized $\hat{z}$
    - $z_{BN} = \gamma\hat{z} + \beta$ where $\gamma, \beta > 0$ are two learnable parameters (i.e., scale and shift parameters)
- We then apply the activation to obtain the next layer value
    - $h^{k+1} = \sigma(z_{BN})$




(Source: medium.com)



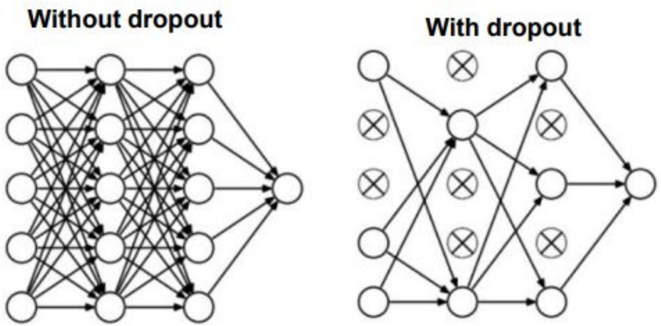

| | |
|---|---|
| 0.1 | 0.2 |
| ... | ... |
| 0.5 | 0.4 |

Mini-batch size

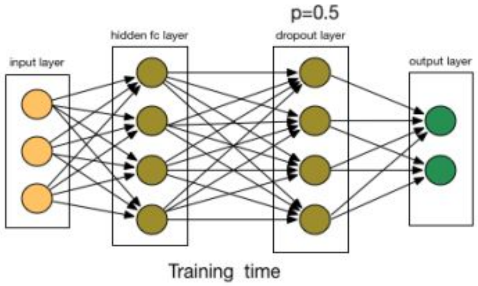
Standardize column-wise

# Dropout

## Reduce Overfitting



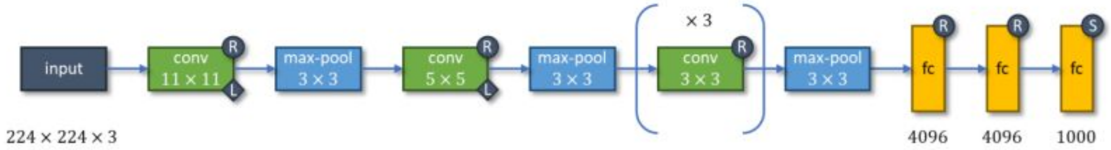

(Source: Analytics Vidhya)



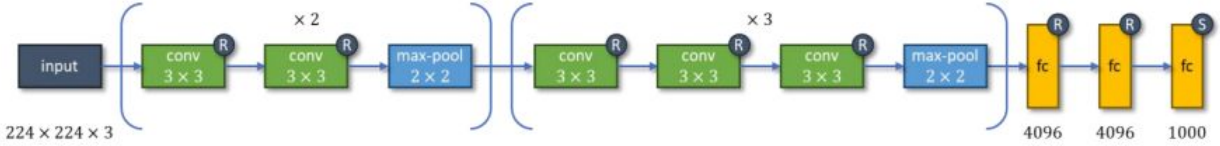

(Source: chatbotslife)

- This is a **cheap technique** to reduce model capacity
  - Reduce overfitting
- In each iteration, at each layer, **randomly choose** some neurons and **drop all connections** from these neurons
  - dropout_rate= 1 – keep_prob

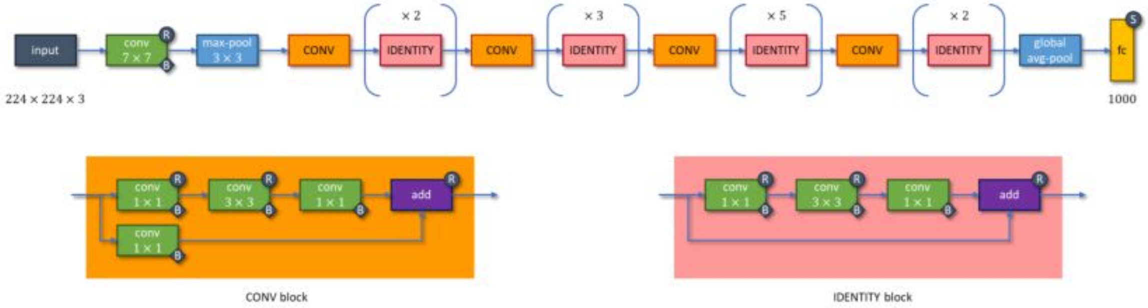
# Network Architectures




**AlexNet**




**VGG16**




**ResNet**

# MiniVGG for Cifar10

## Our Tutorial

| Layer Type | Output Size | Filter Size / Stride |
|---|---|---|
| INPUT IMAGE | $32 \times 32 \times 3$ | |
| CONV | $32 \times 32 \times 32$ | $3 \times 3, K = 32$ |
| ACT | $32 \times 32 \times 32$ | |
| BN | $32 \times 32 \times 32$ | |
| CONV | $32 \times 32 \times 32$ | $3 \times 3, K = 32$ |
| ACT | $32 \times 32 \times 32$ | |
| BN | $32 \times 32 \times 32$ | |
| POOL | $16 \times 16 \times 32$ | $2 \times 2$ |
| DROPOUT | $16 \times 16 \times 32$ | |
| CONV | $16 \times 16 \times 64$ | $3 \times 3, K = 64$ |
| ACT | $16 \times 16 \times 64$ | |
| BN | $16 \times 16 \times 64$ | |
| CONV | $16 \times 16 \times 64$ | $3 \times 3, K = 64$ |
| ACT | $16 \times 16 \times 64$ | |
| BN | $16 \times 16 \times 64$ | |
| POOL | $8 \times 8 \times 64$ | $2 \times 2$ |
| DROPOUT | $8 \times 8 \times 64$ | |
| FC | 512 | |
| ACT | 512 | |
| BN | 512 | |
| DROPOUT | 512 | |
| FC | 10 | |
| SOFTMAX | 10 | |

```python
def create_vgg():
    vgg_model = nn.Sequential(
        #nn.LazyConv2d(32, kernel_size=3, padding=1),
        nn.Conv2d(3, 32, kernel_size=3, padding= 1), #[32,32,32]
        nn.BatchNorm2d(32),
        nn.ReLU(),
        #nn.LazyConv2d(32, kernel_size=3, padding=1),
        nn.Conv2d(32, 32, kernel_size=3, padding=1), #[32,32,32]
        nn.BatchNorm2d(32),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2), #down-sample by two #[32,16,16]
        nn.Dropout(p=0.25),

        #nn.LazyConv2d(64, kernel_size=3, padding=1),
        nn.Conv2d(32, 64 , kernel_size=3, padding=1), #[64,16,16]
        nn.BatchNorm2d(64, momentum=0.1),
        nn.ReLU(),
        #nn.LazyConv2d(64, kernel_size=3, padding=1)
        nn.Conv2d(64, 64, kernel_size=3, padding=1), #[64,16,16]
        nn.BatchNorm2d(64, momentum=0.1),
        nn.ReLU(),
        nn.LazyConv2d(64, kernel_size=3, padding=1),
        nn.BatchNorm2d(64, momentum=0.1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2), #down-sample by two [64,8,8]
        nn.Dropout(p=0.25),

        nn.Flatten(1), #64x8x8
        #nn.Linear(64x8x8, 512)
        nn.LazyLinear(512),
        nn.ReLU(),
        #nn.LazyLinear(10)
        nn.Linear(512, 10),
      )
    return vgg_model
```

60,000 images
10 classes