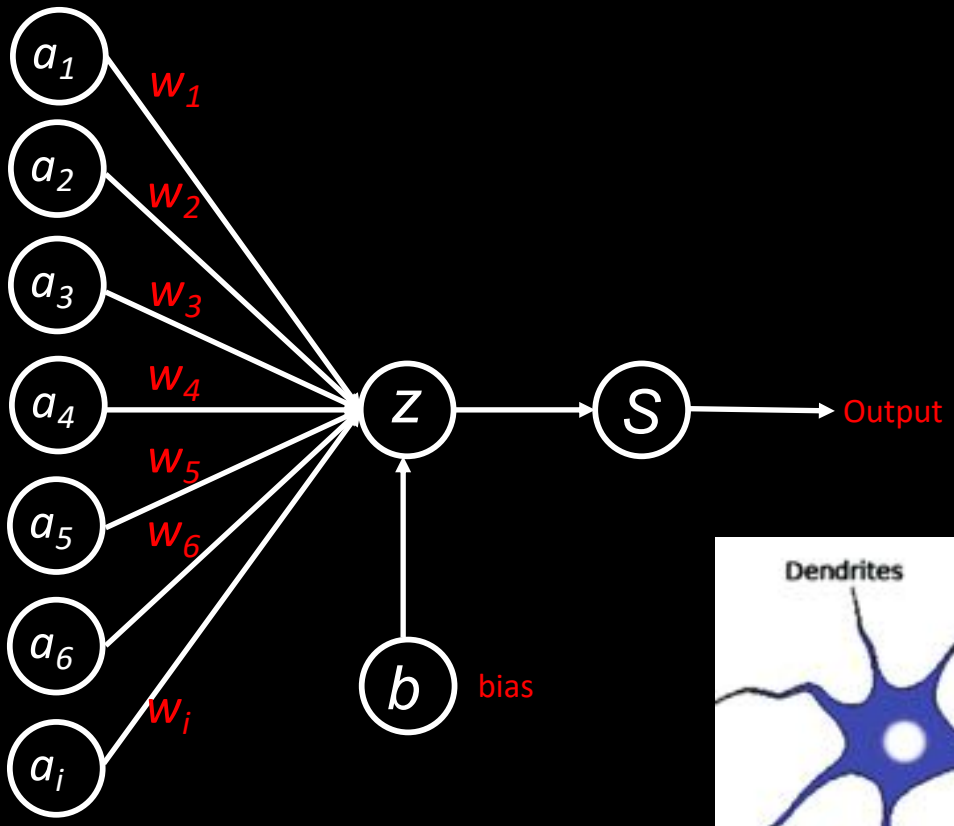
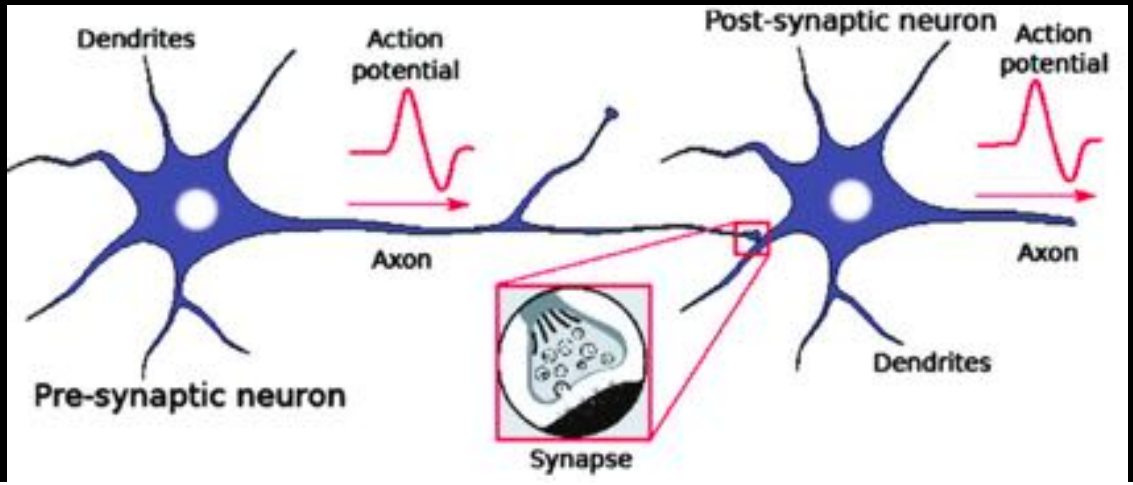
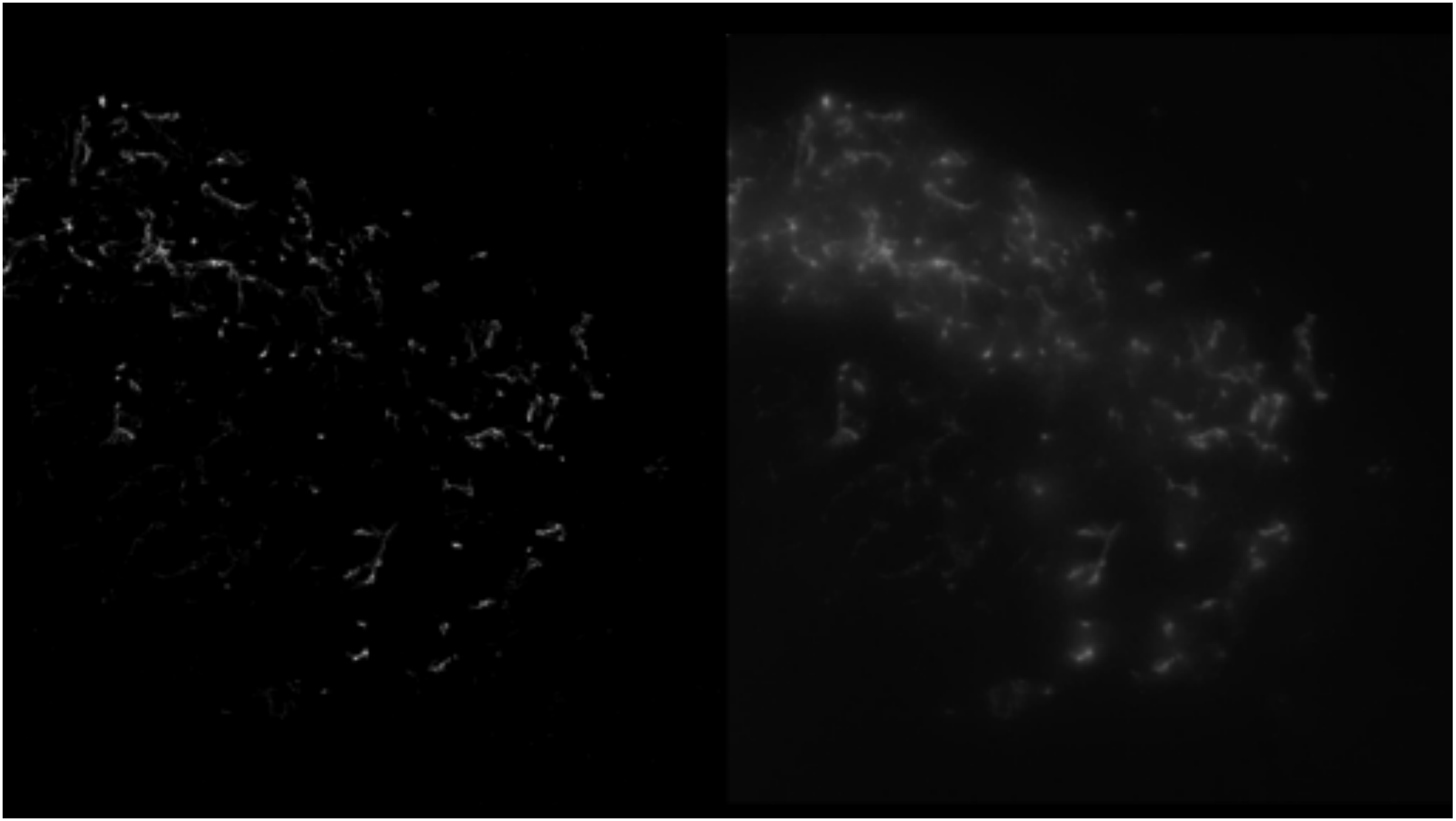
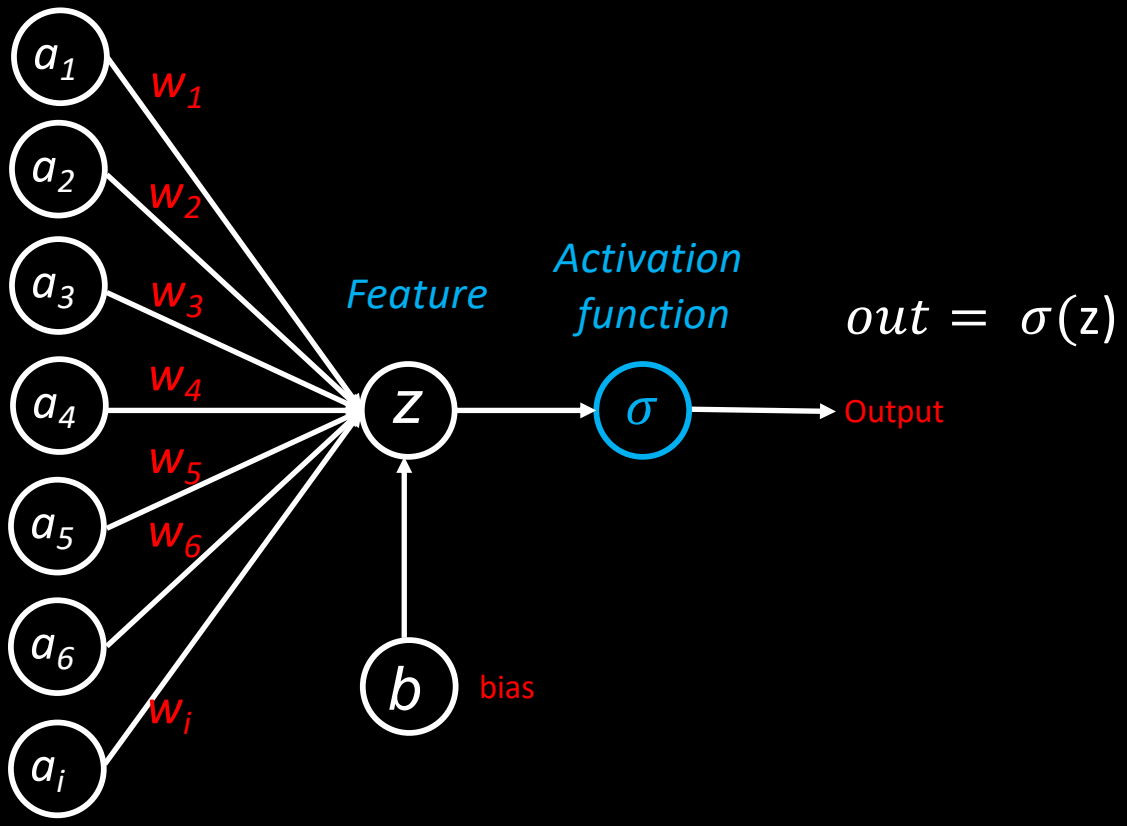
Neurons and Neural Networks

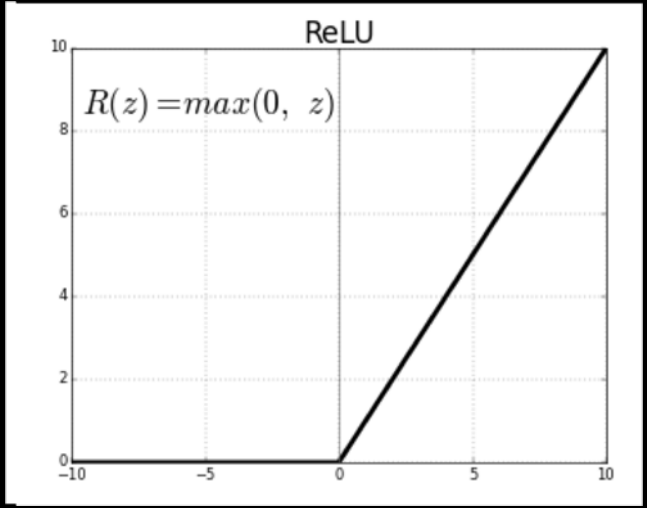

a_1
a_2
a_3
a_4
a_5
a_6
a_i
w_1
w_2
w_3
w_4
w_5
w_6
w_i
z
S
Output
b
bias

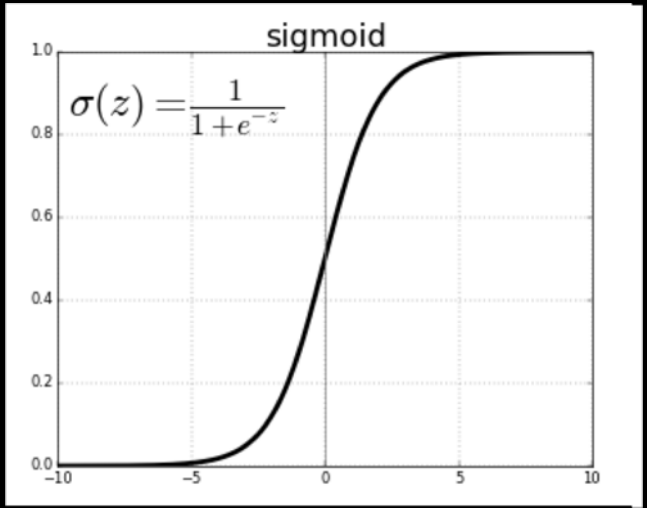

Dendrites
Action potential
Post-synaptic neuron
Action potential
Axon
Axon
Dendrites
Pre-synaptic neuron
Synapse

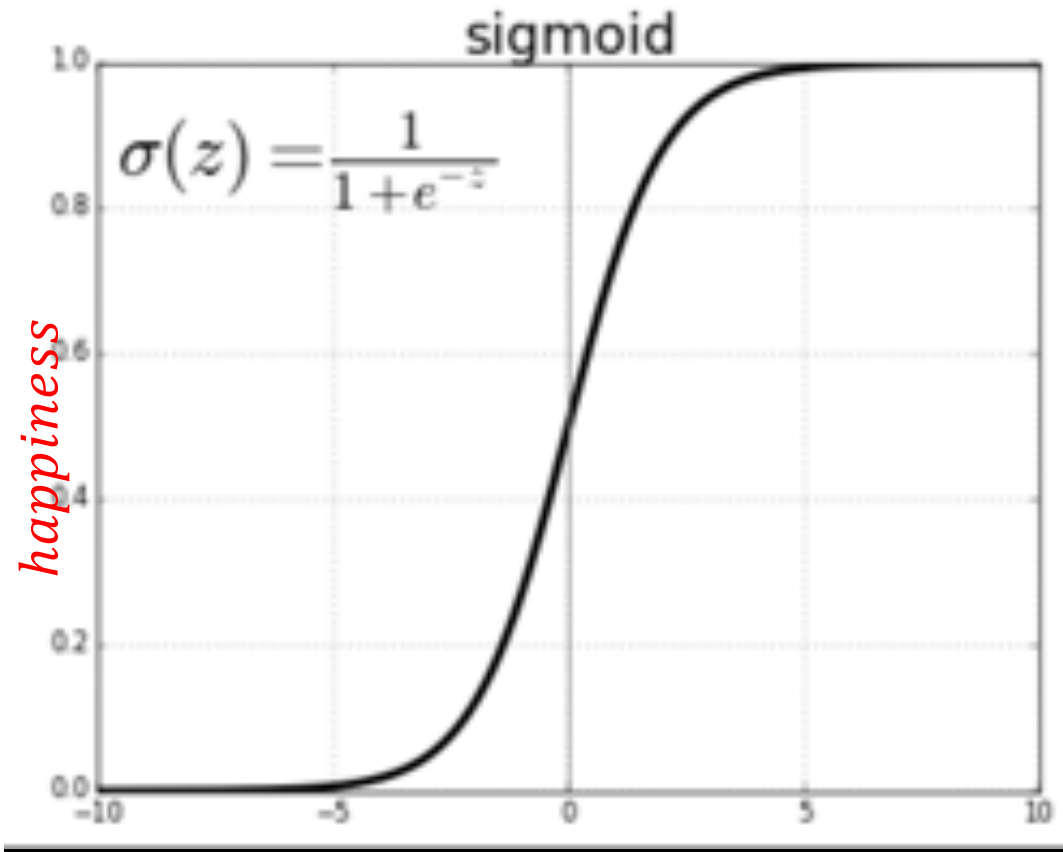


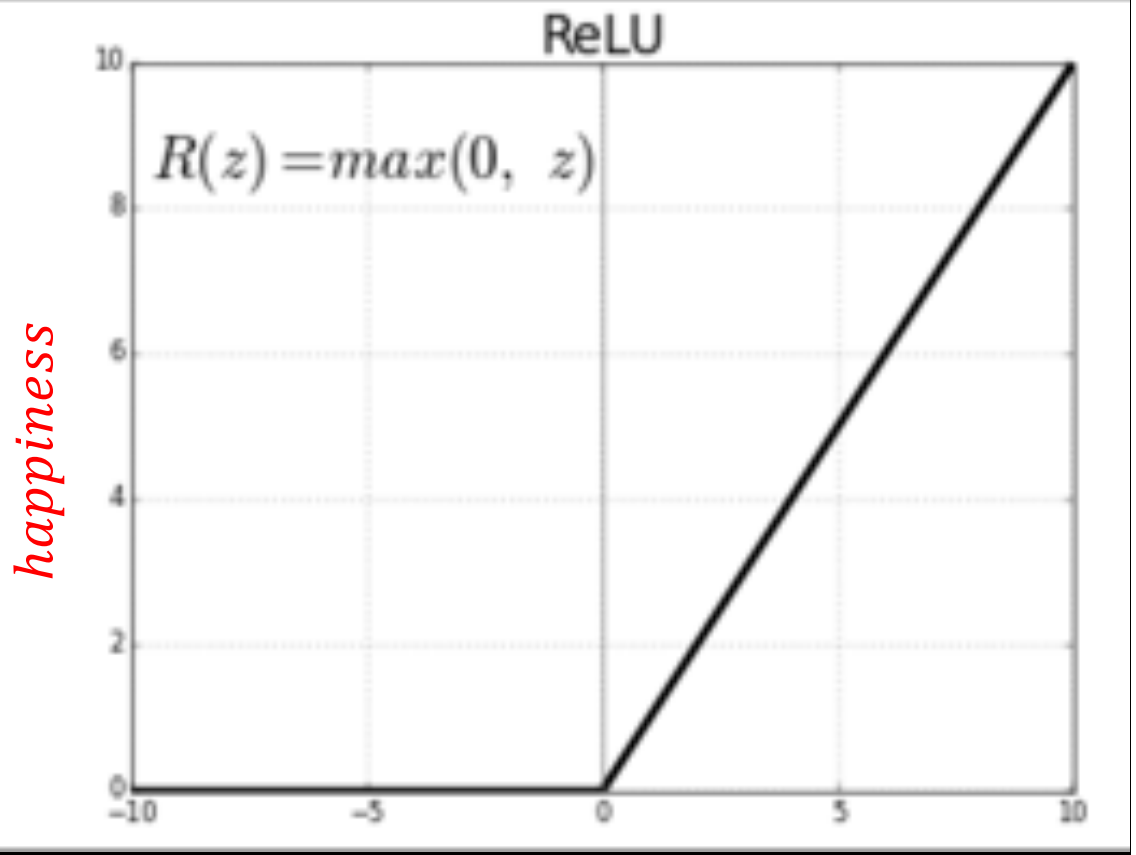


Activation function

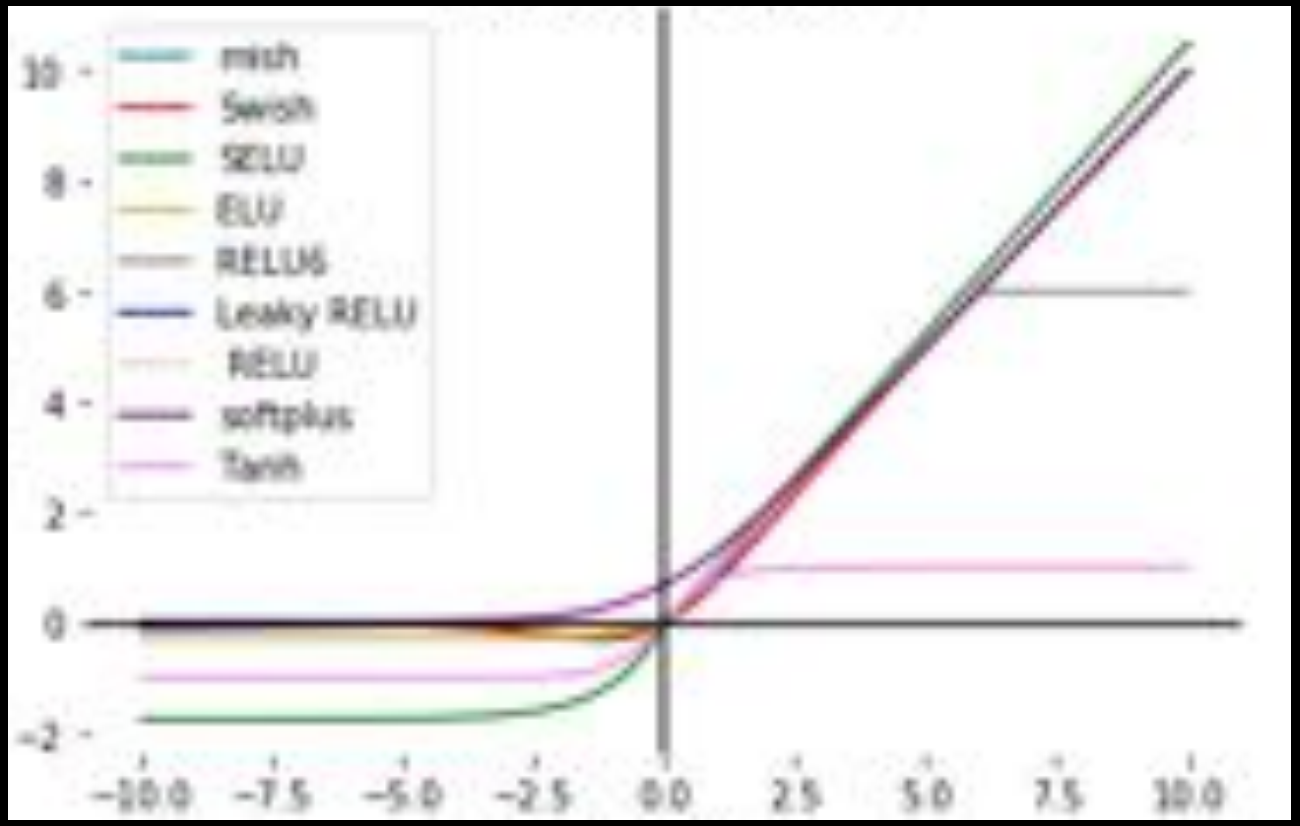


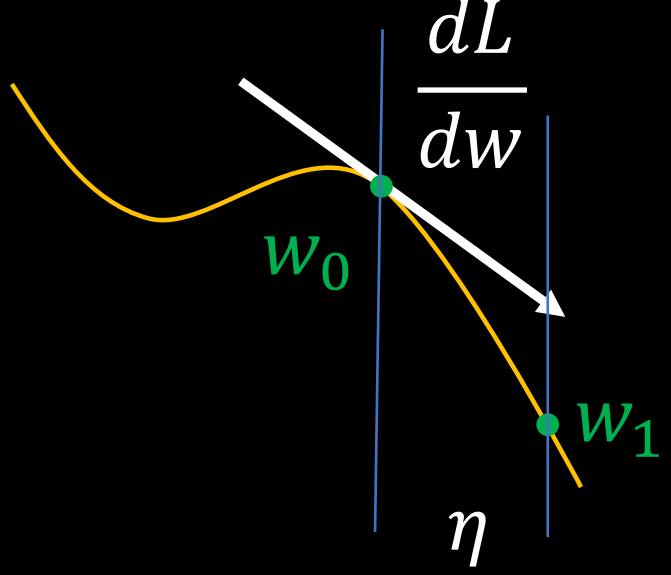


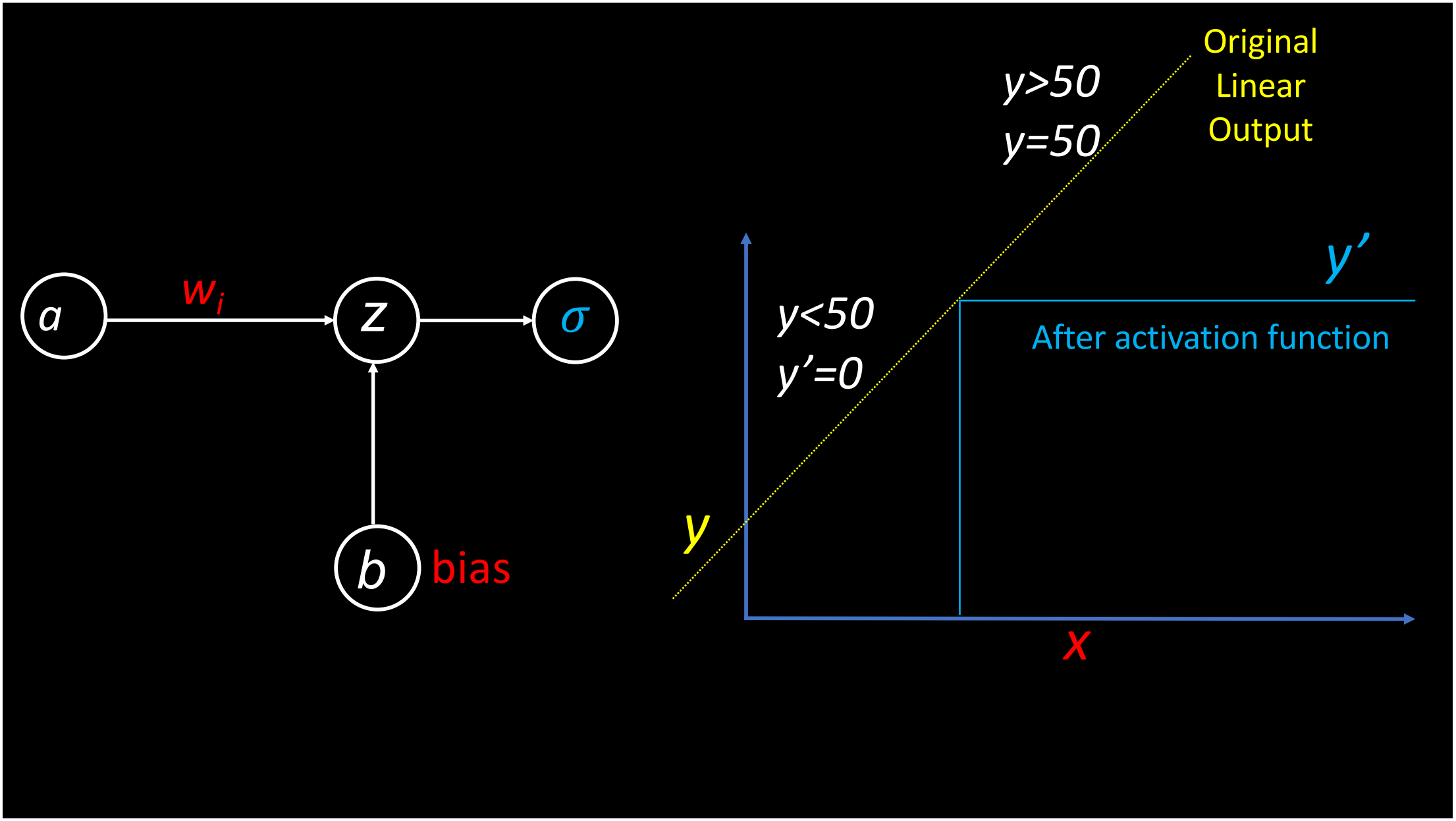

sigmoid
$\sigma(z) = \frac{1}{1+e^{-z}}$
happiness
pizza

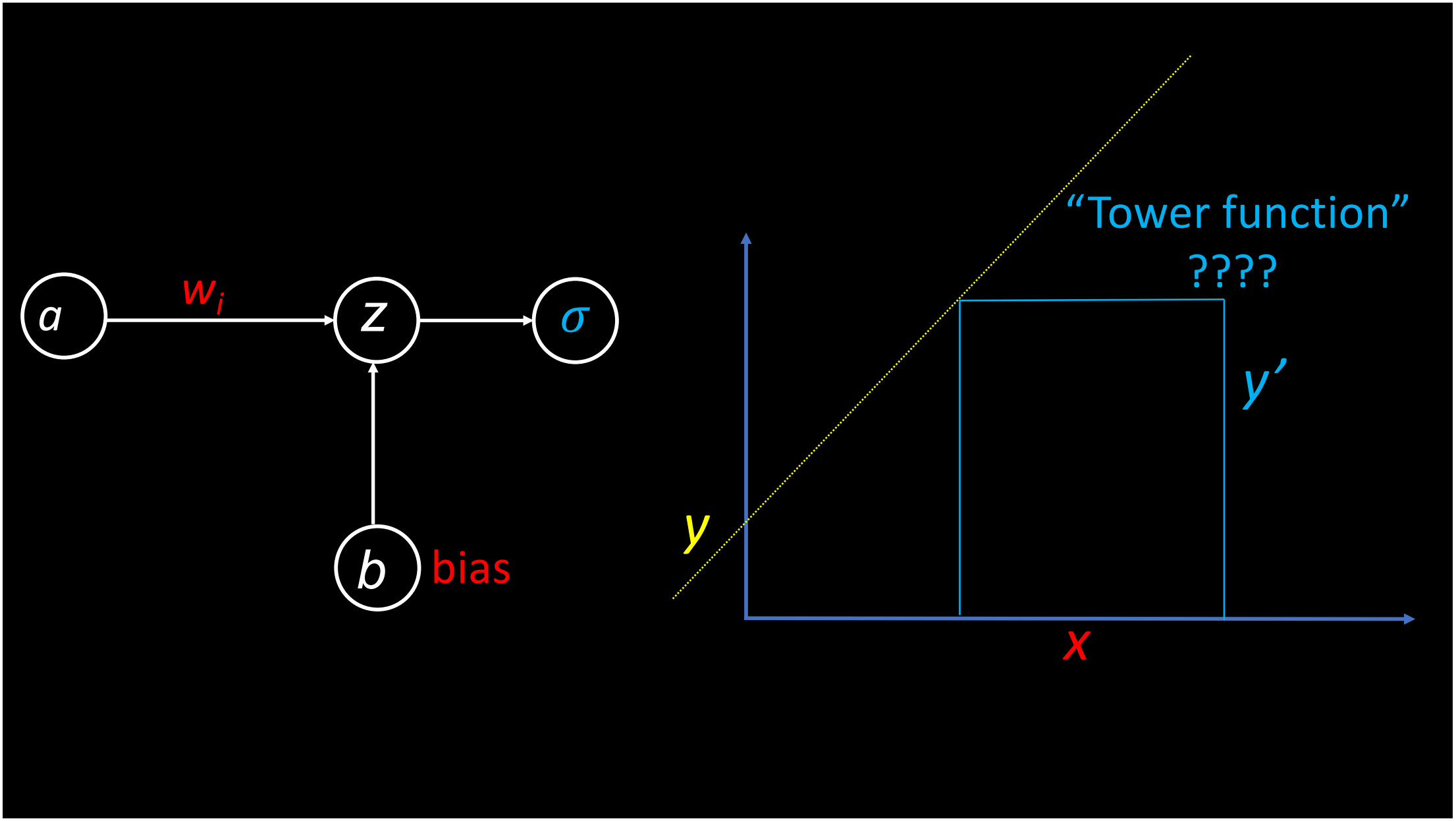

ReLU
$R(z) = max(0,\ z)$
happiness
pizza

Activation Functions






a
w_i
z
σ
b
bias
Original
Linear
Output
y>50
y=50
y'
y<50
y'=0
After activation function
y
X


a
w_i
z
σ
b
bias
"Tower function"
????
y'
y
x

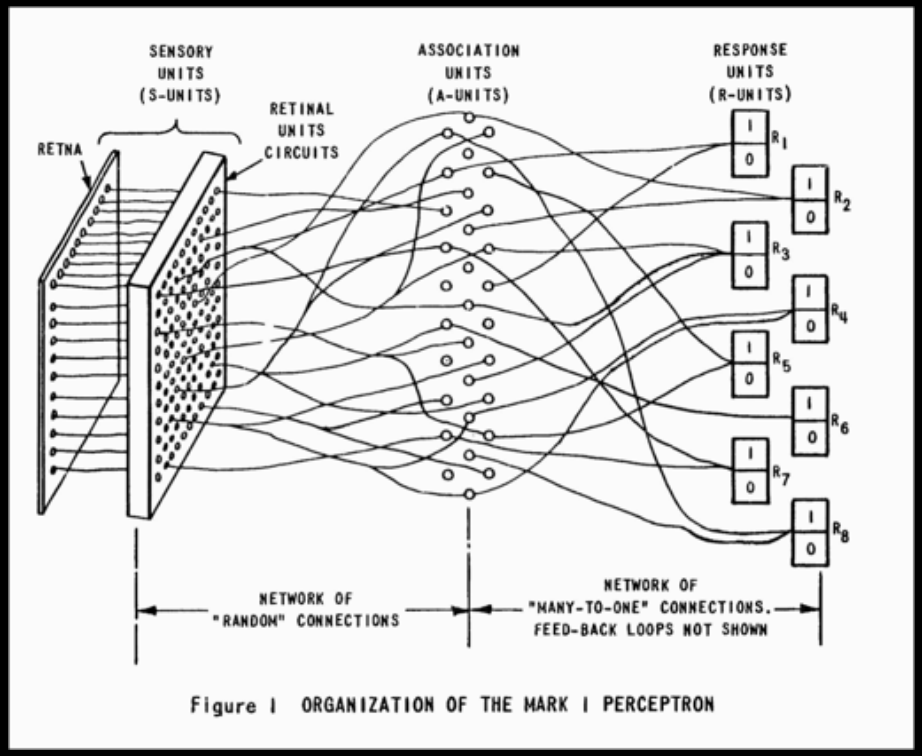
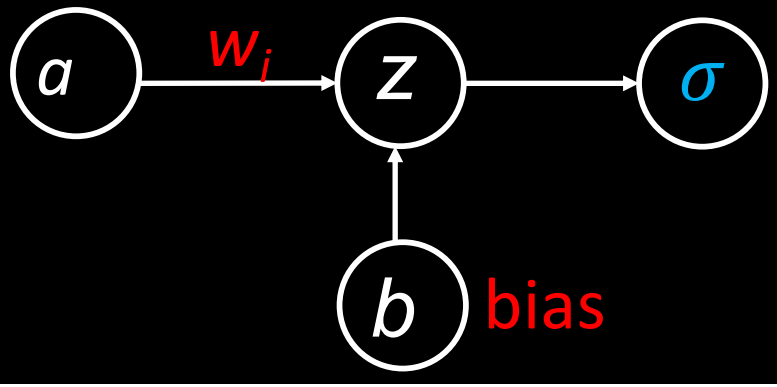


Figure I ORGANIZATION OF THE MARK I PERCEPTRON

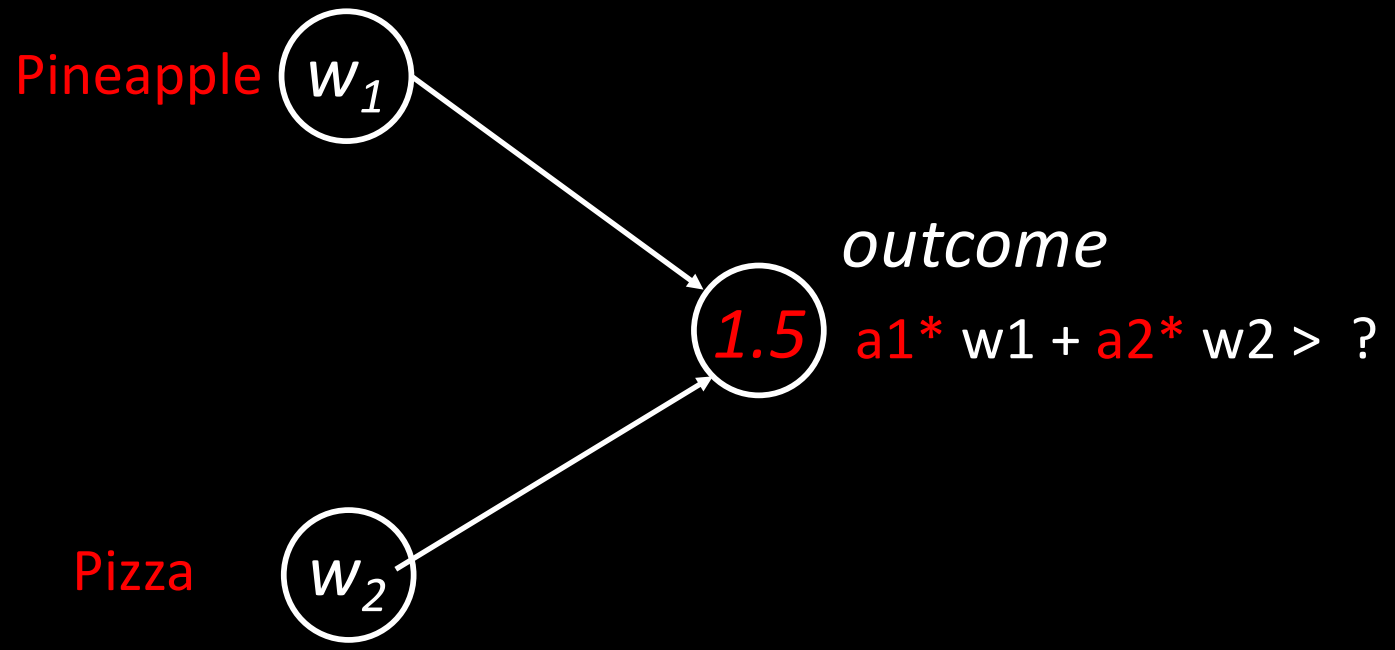


Mark 1 perceptron
1958

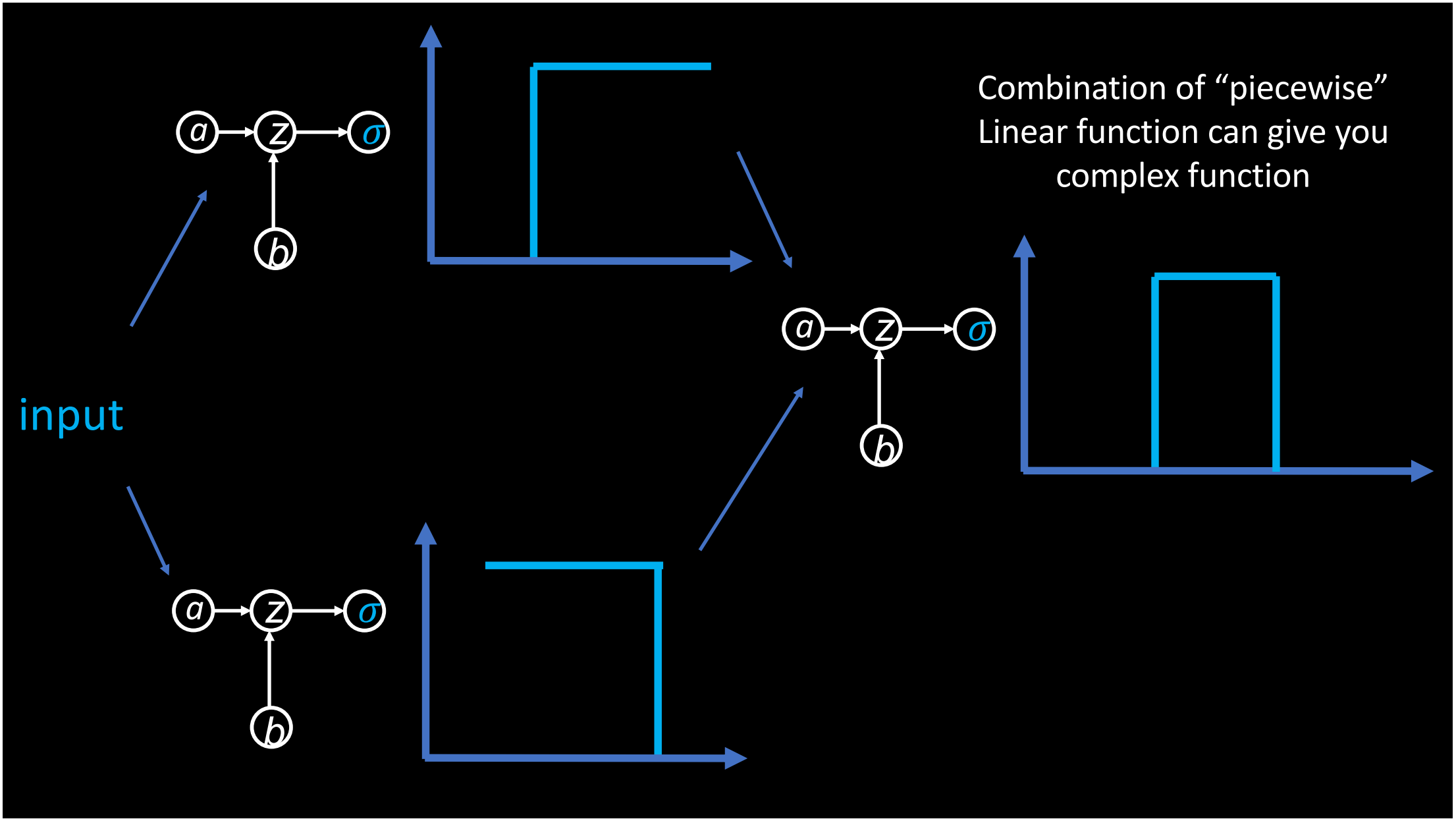
XOR

Pizza Pineapple Happiness

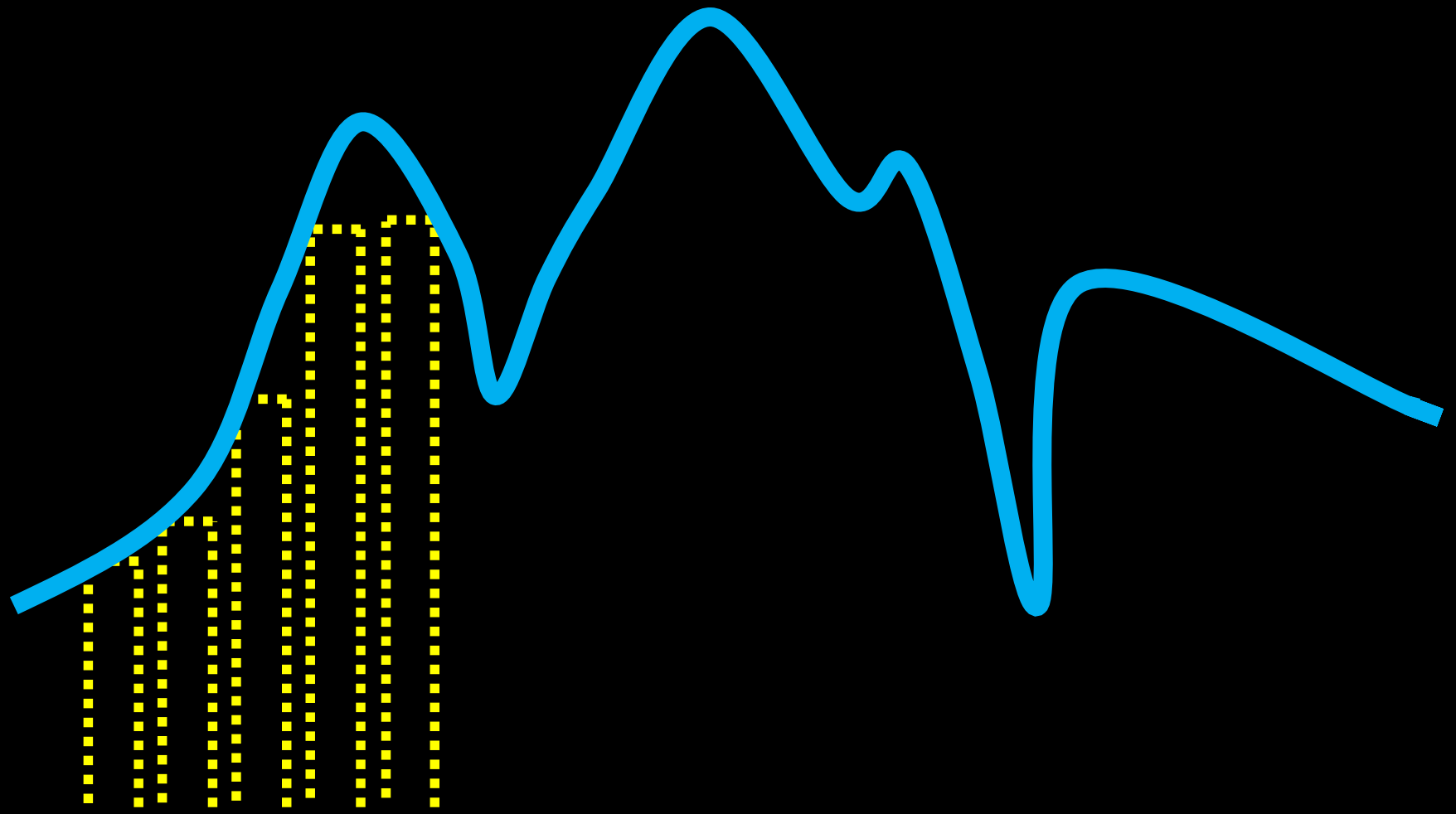
A	B	Out
0	0	0
1	0	1
0	1	1
1	1	0



Linear function can’t mimic complex behavior

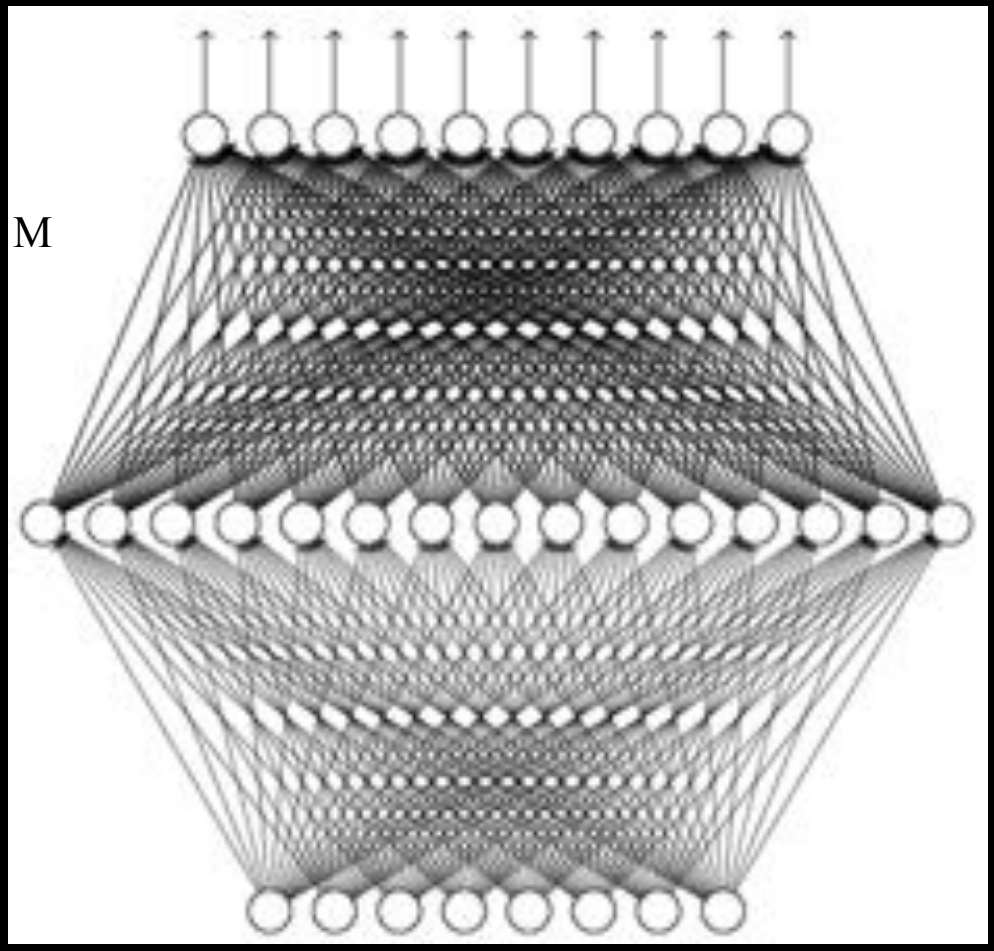

Combination of “piecewise” Linear function can give you complex function
input
a
z
σ
b
a
z
σ
b
a
z
σ
b

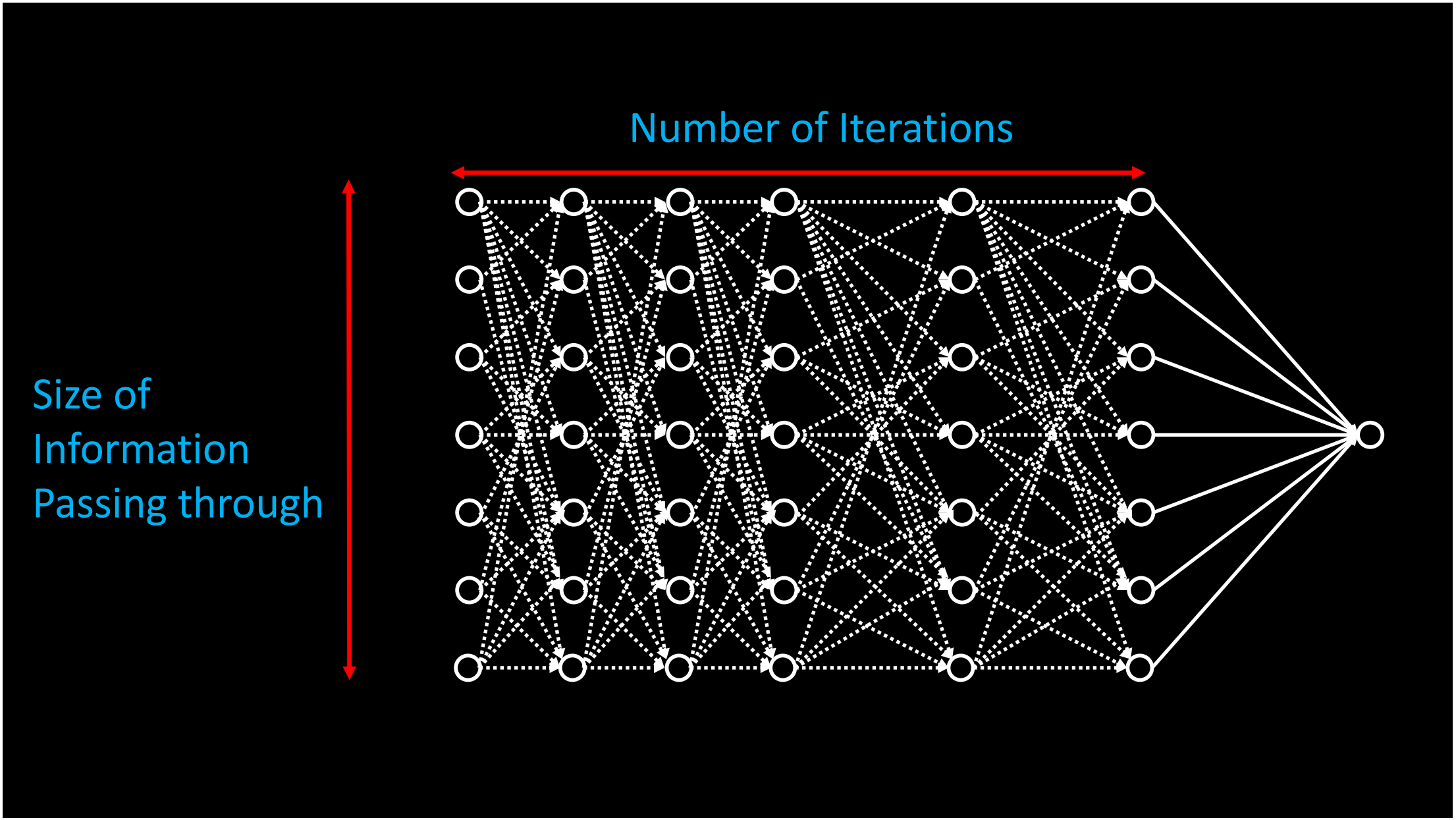
We can approximate any complex function using enough neurons

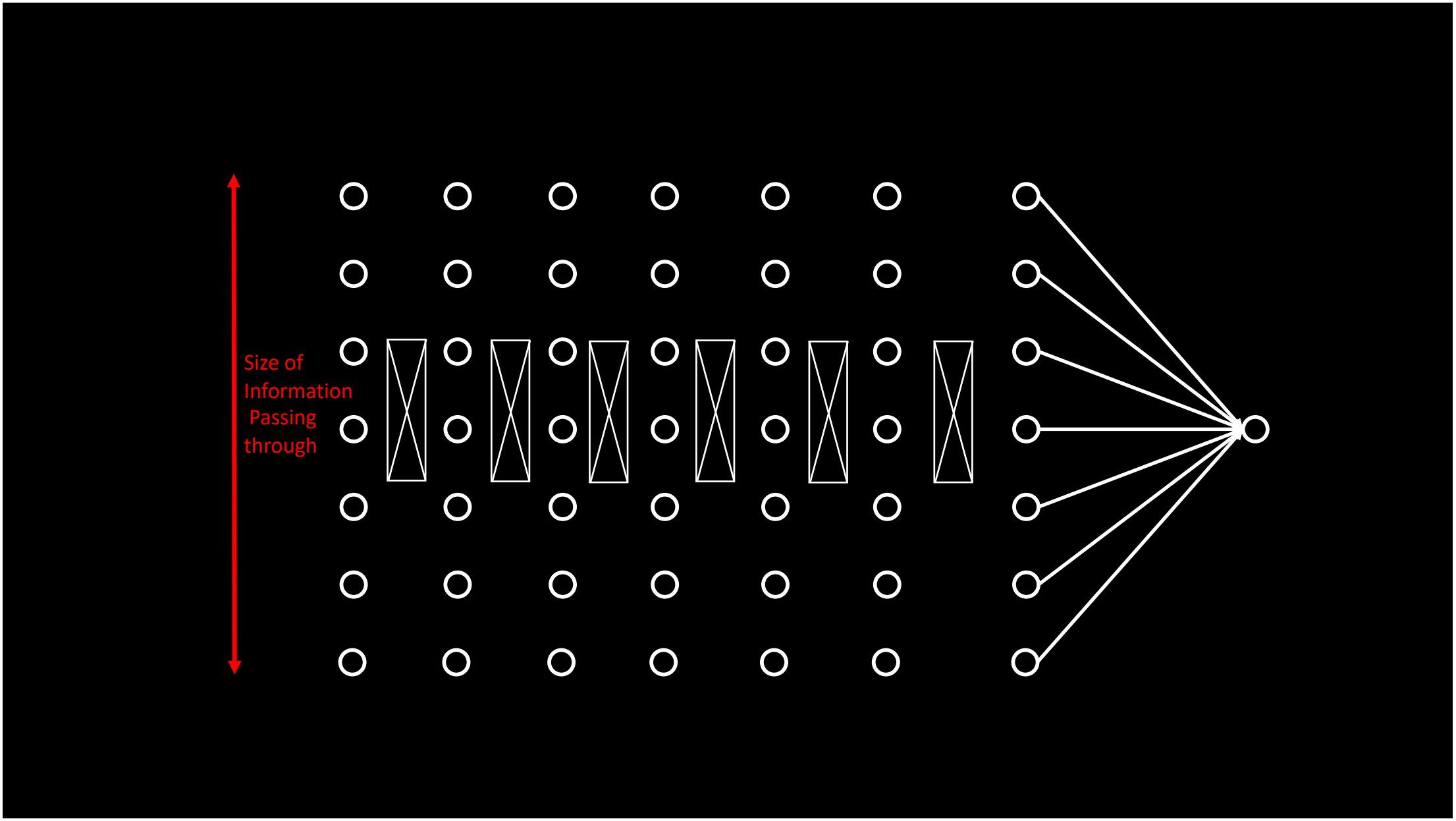


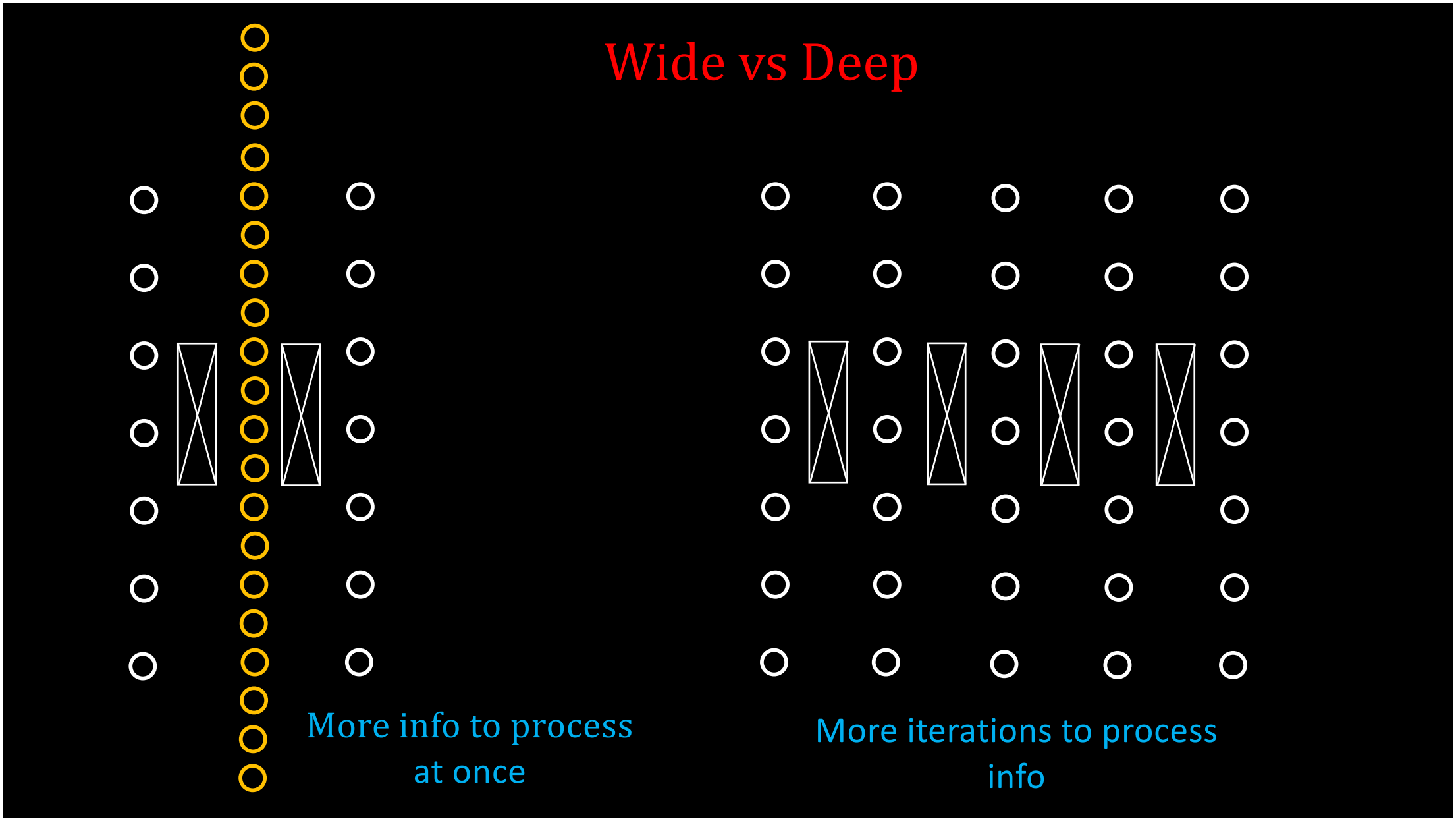
Universality Approximation Theorem

$$f: R^N > R^M$$




Number of Iterations
Size of
Information
Passing through


Size of
Information
Passing
through


Wide vs Deep
More info to process at once
More iterations to process info

Convolutional Neural Networks

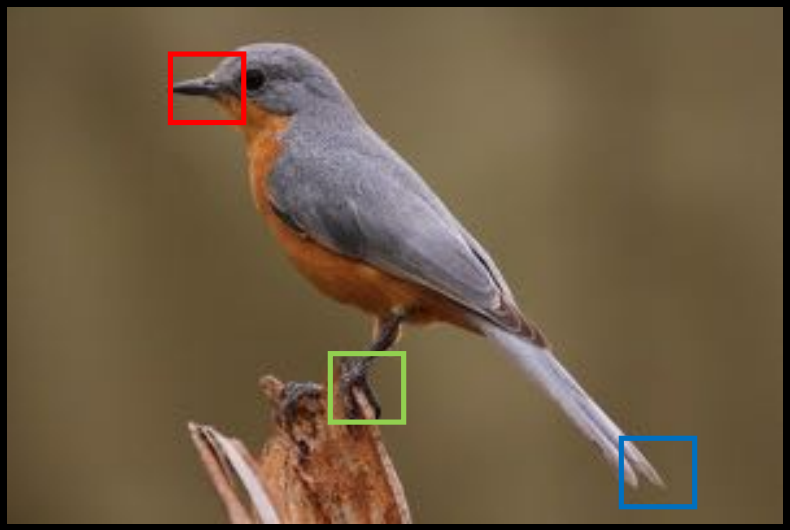
Image to grayscale pixel value

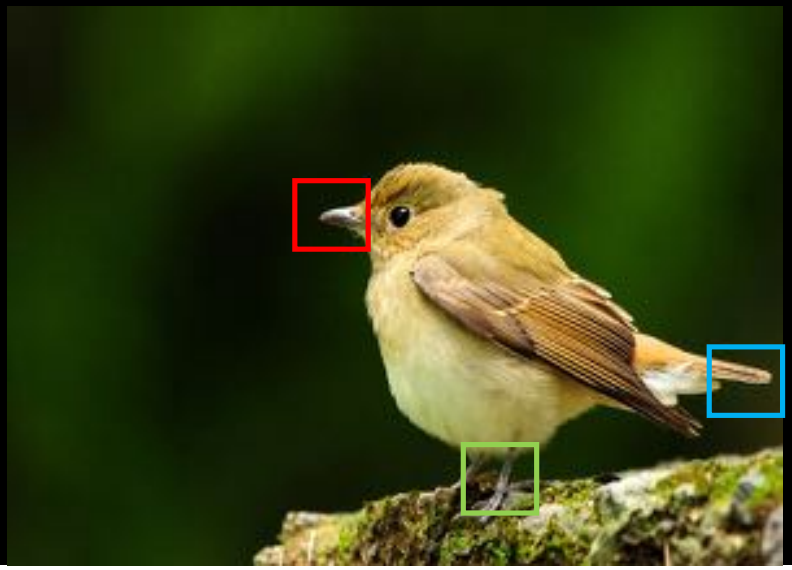
1	0.5	0	0
0.5	0	0	0
0.5	0	0	0
0	0	0.5	1

1
0.5
0
0
0.5
0
0
0
0.5
0
0

numpy.ndarray.flatten

torch.array.view(-1)





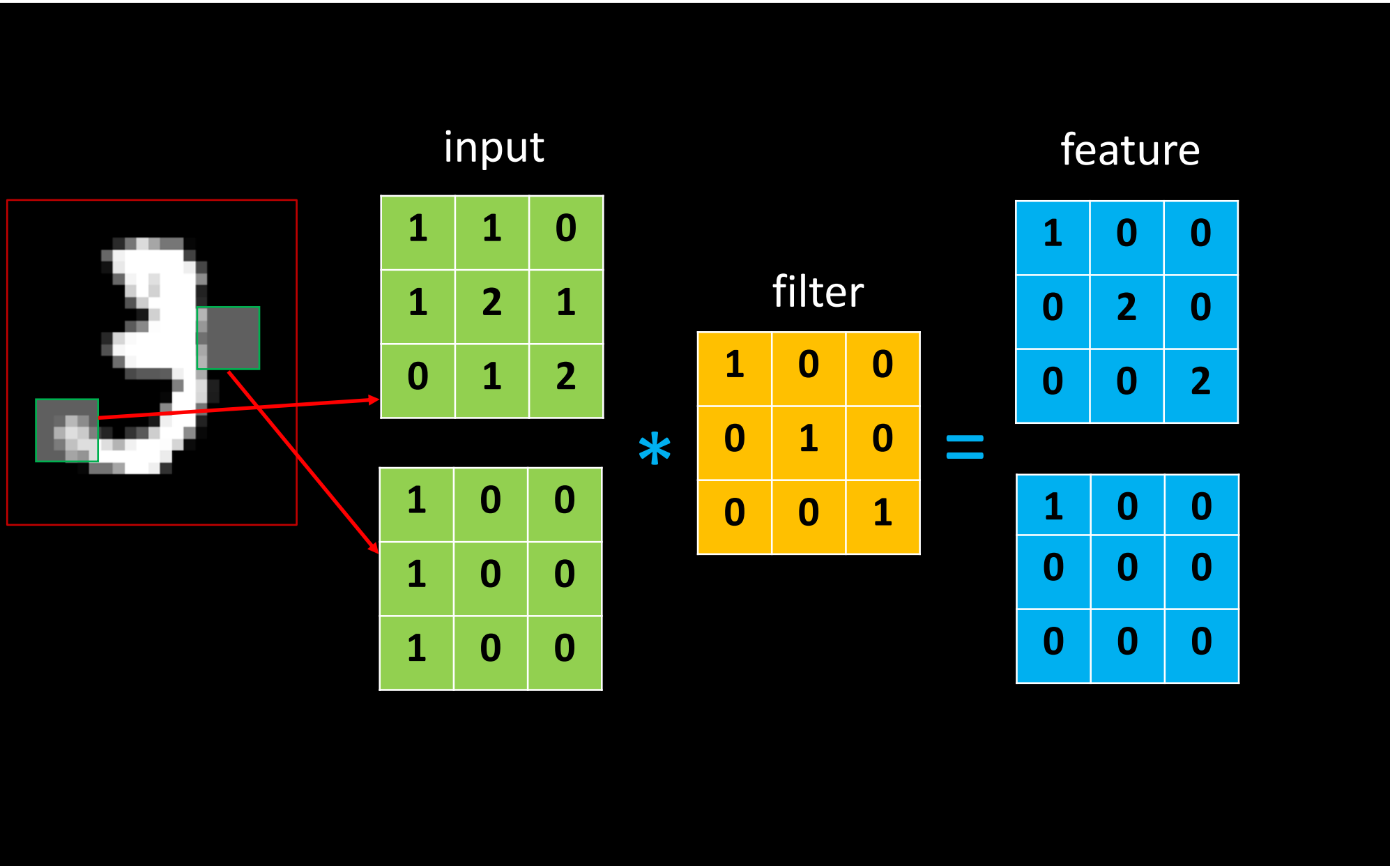
"Convolutional"

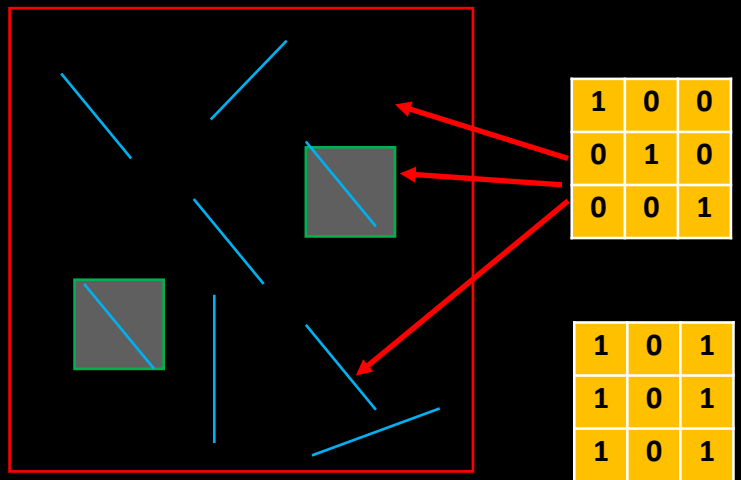
Beak (left, right, middle….)

Feet (left, right, middle….)

Tail (left, right, middle….)

Should share the same parameters!

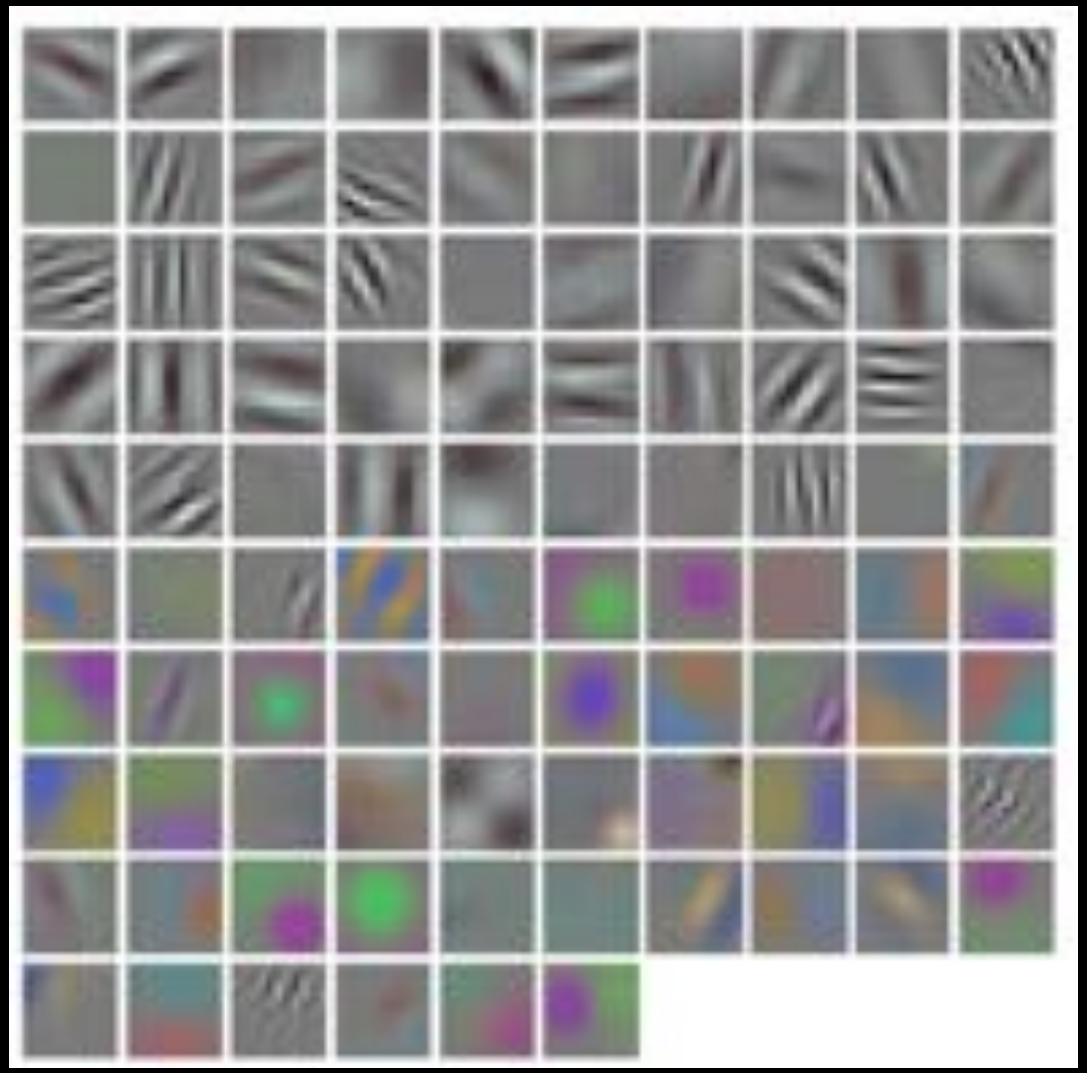

input
1 1 0
1 2 1
0 1 2
1 0 0
1 0 0
1 0 0
*
filter
1 0 0
0 1 0
0 0 1
=
feature
1 0 0
0 2 0
0 0 2
1 0 0
0 0 0
0 0 0

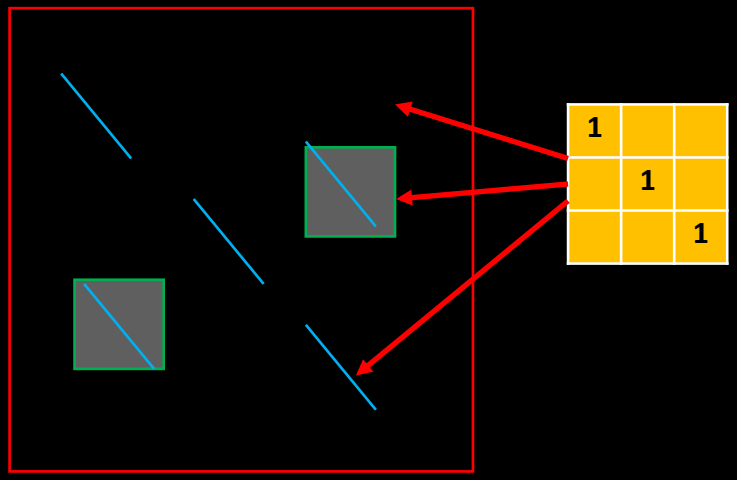


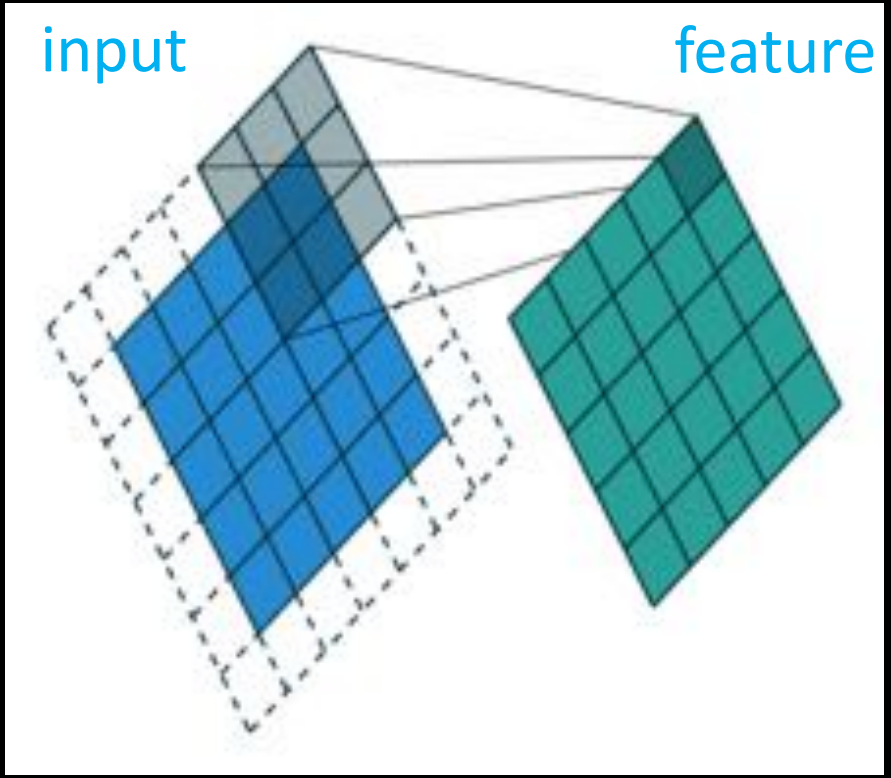
1	0	0
0	1	0
0	0	1

1	0	1
1	0	1
1	0	1

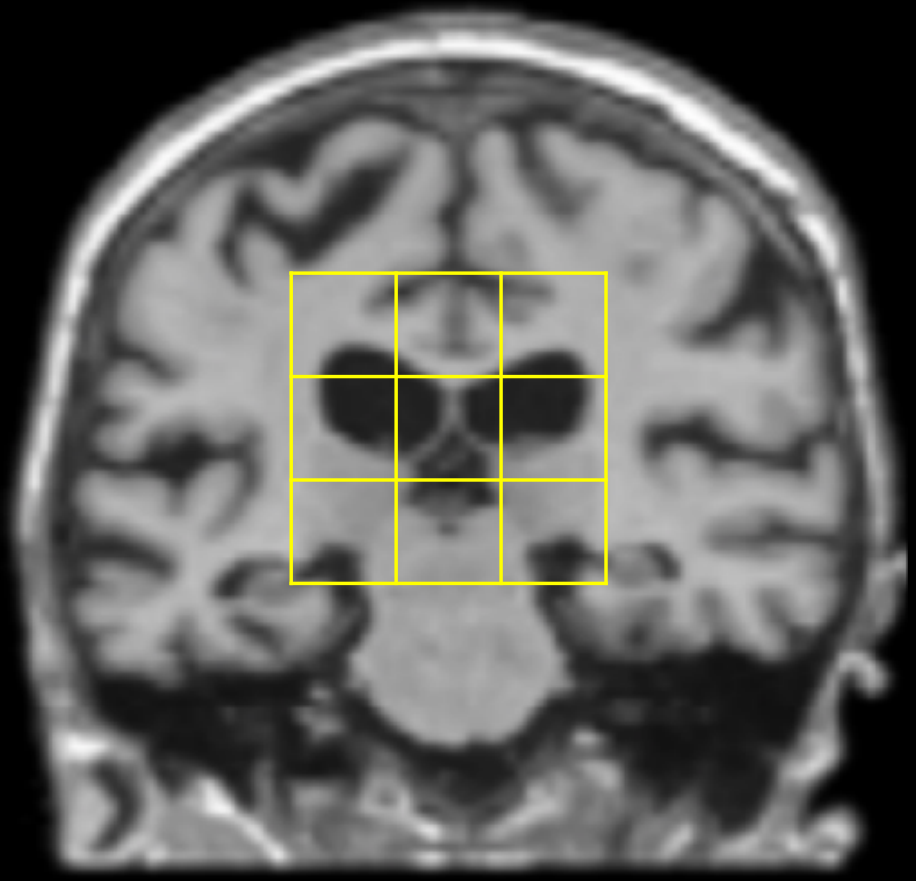
1	1	1
1	1	1
1	1	1



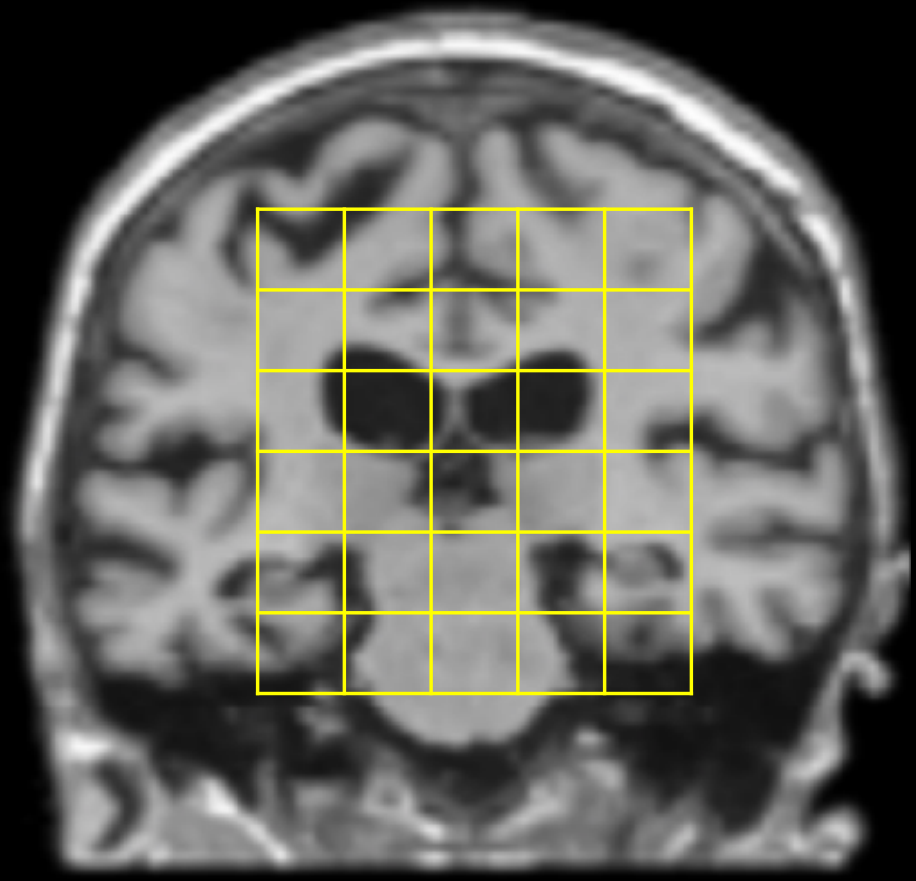

1
1
1


input
feature

Size of convolutional kernels

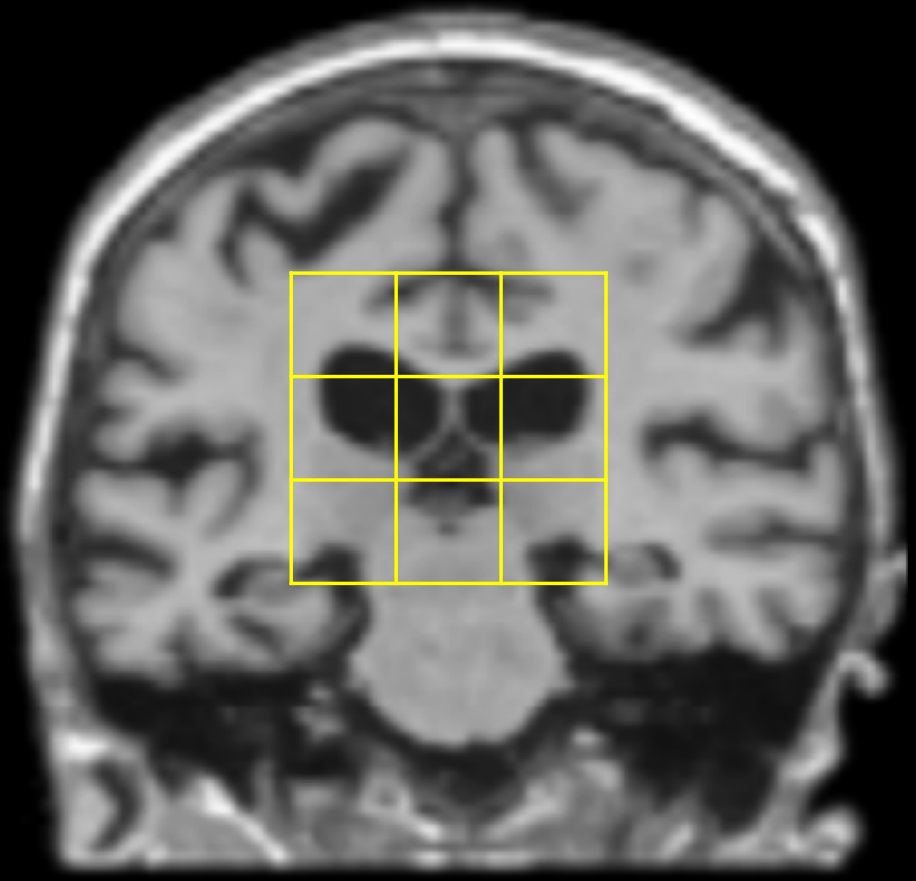


Less info from neighbors

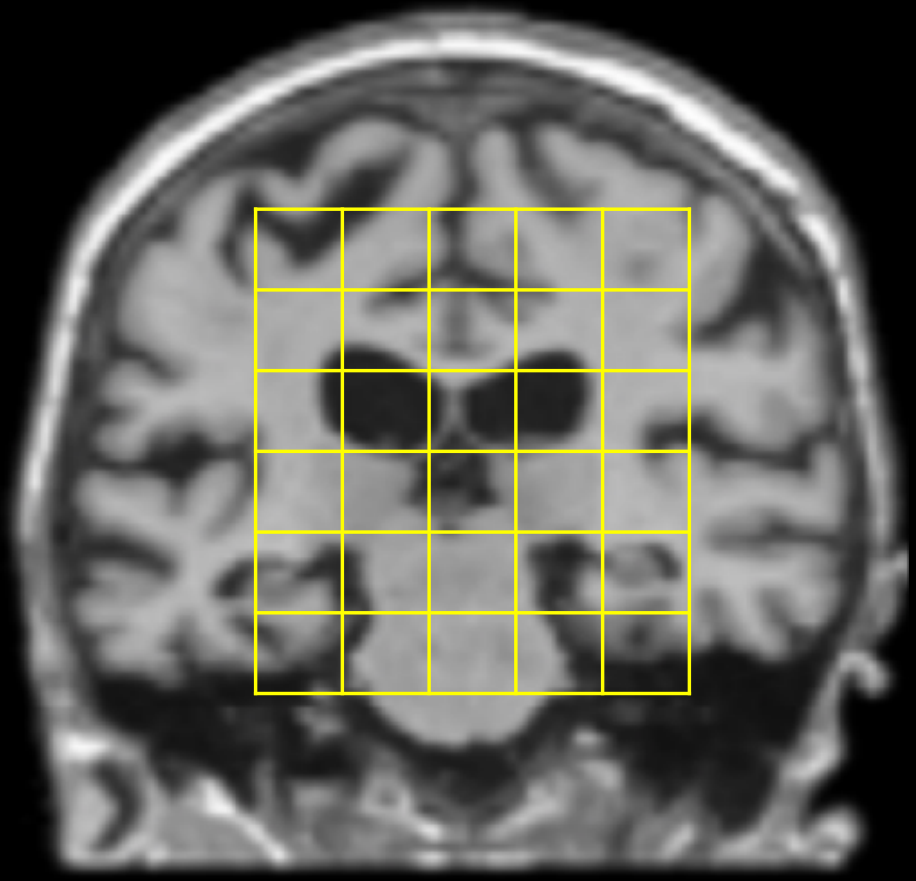


More info

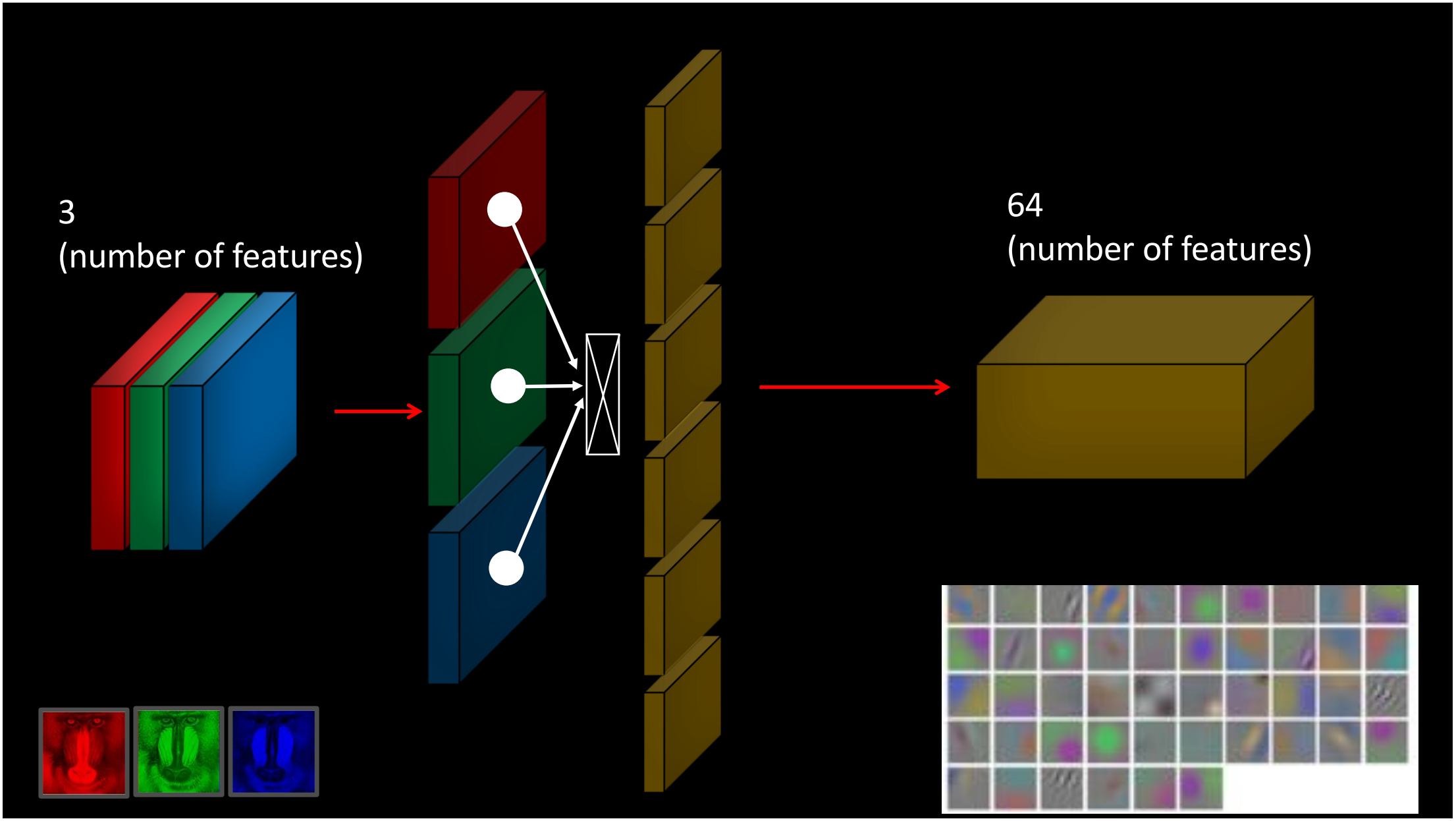
Size of convolutional kernels



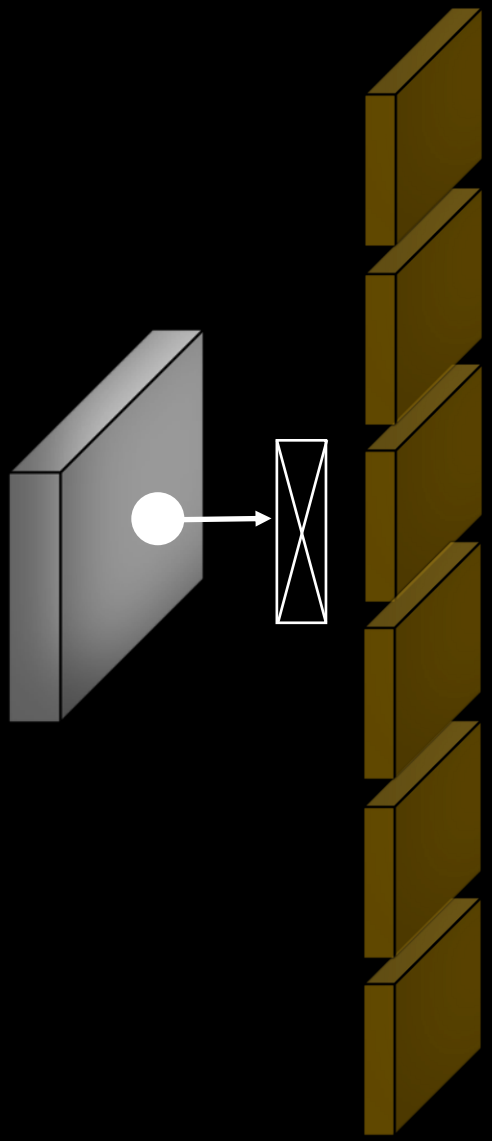
Number of parameters 3^2



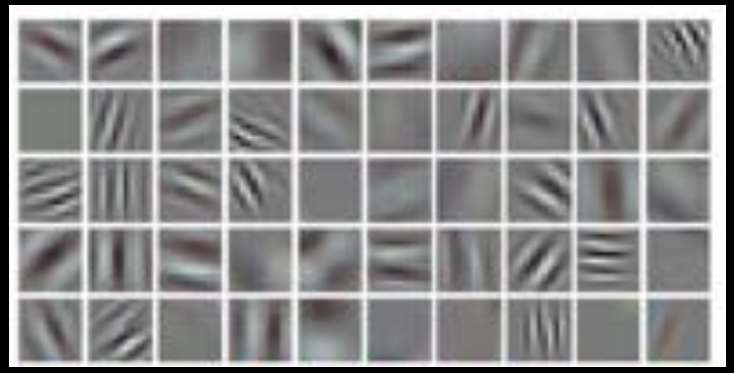
Number of parameters 5^2

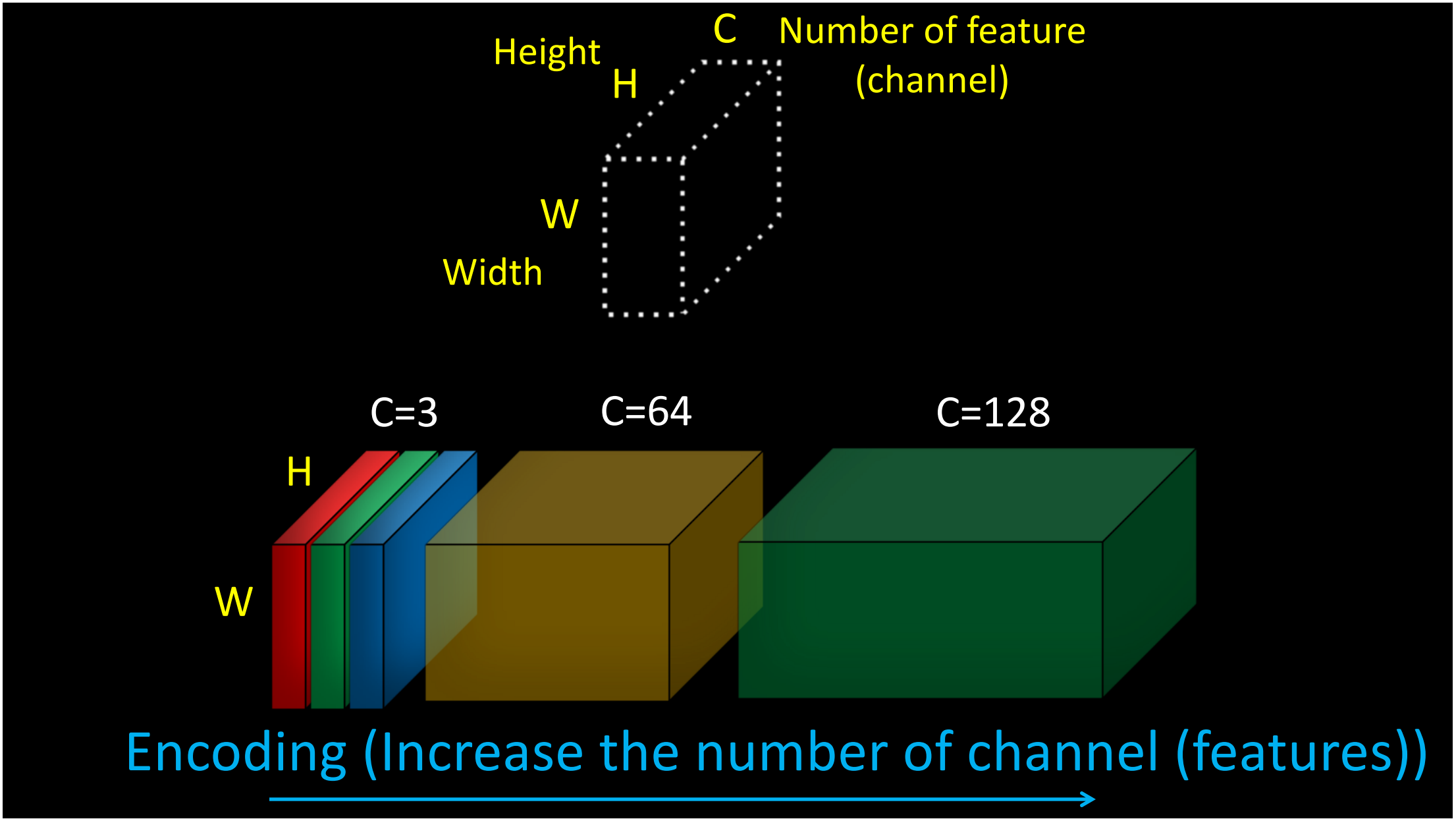

3
(number of features)
64
(number of features)

3
(number of features)

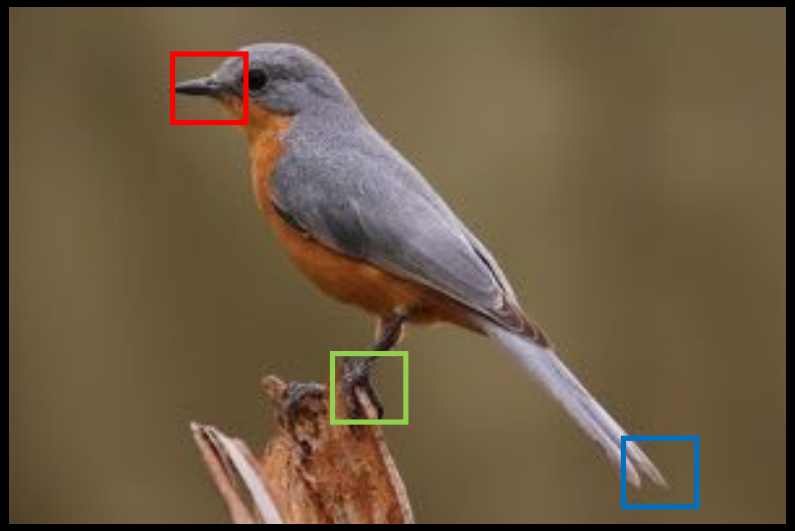


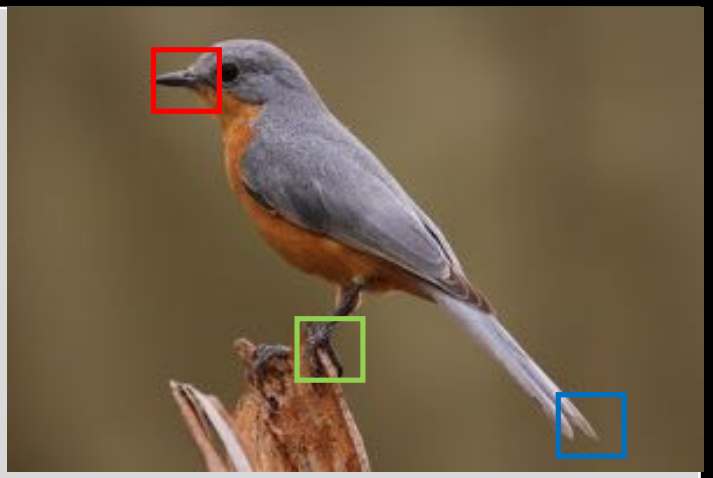
64
(number of features)

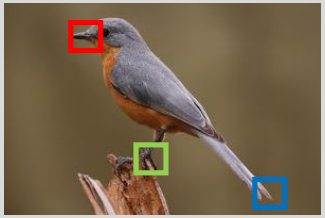



C
Number of feature
(channel)
Height
H
W
Width
C=3
C=64
C=128
H
W
Encoding (Increase the number of channel (features))

Pooling





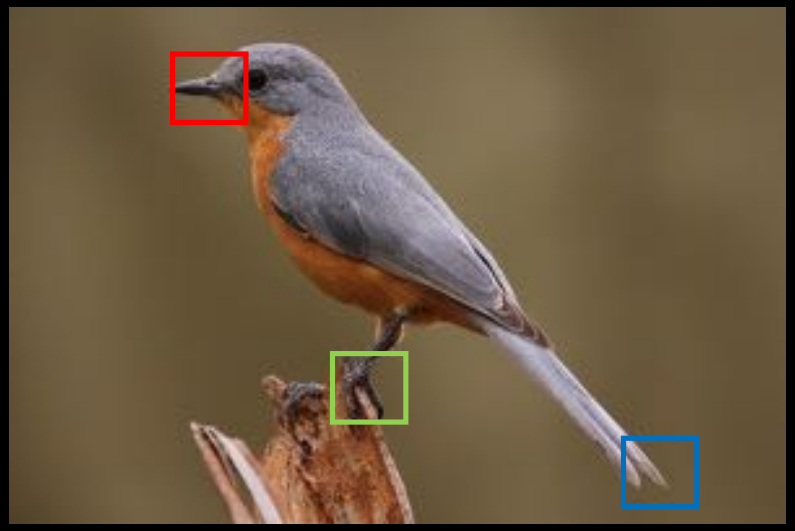


Important features should be independent of scaling

and

Can be in many location

(Max)Pooling



3	1	0			9
4	5	2			
-5	-2	-1			
					4
	7				

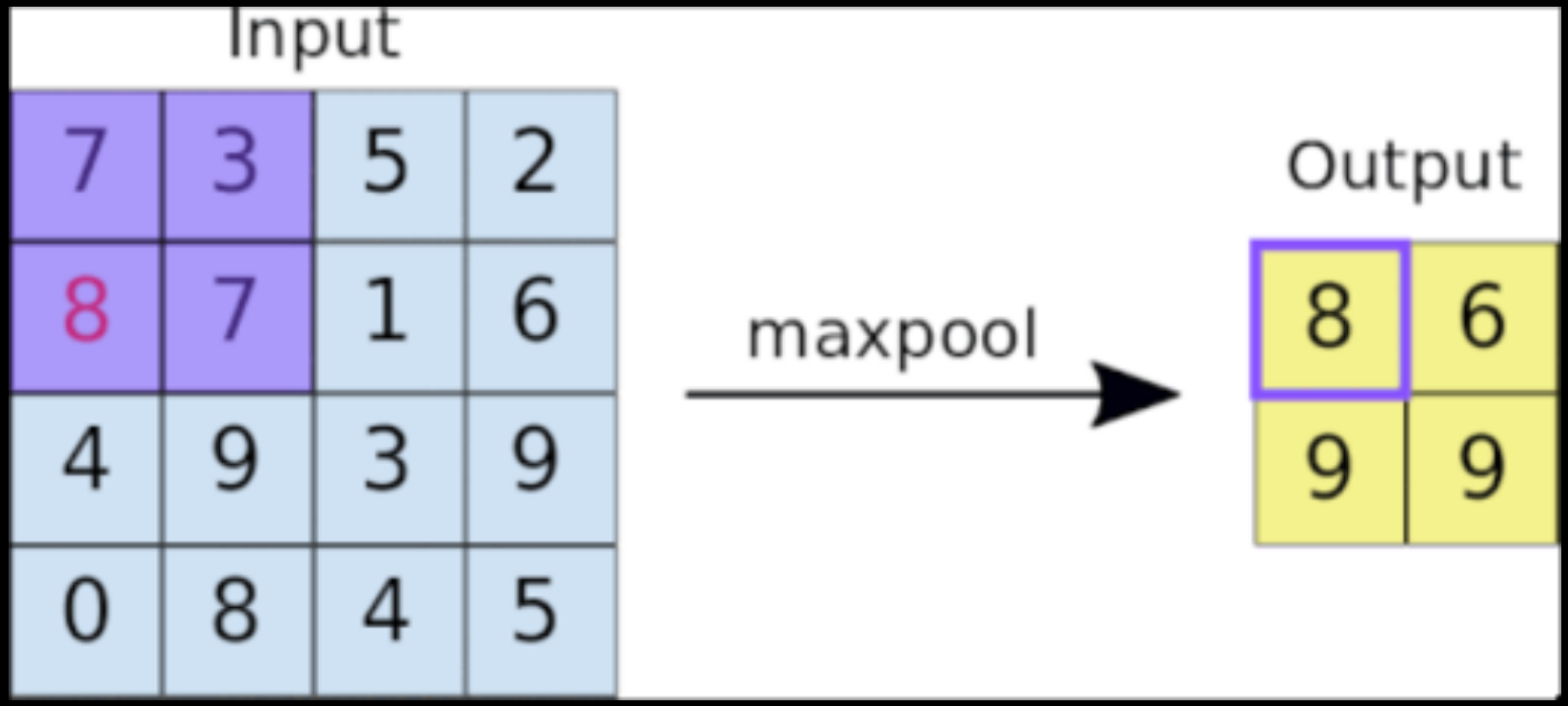
5	9
7	4

Important features should be independent of scaling

and

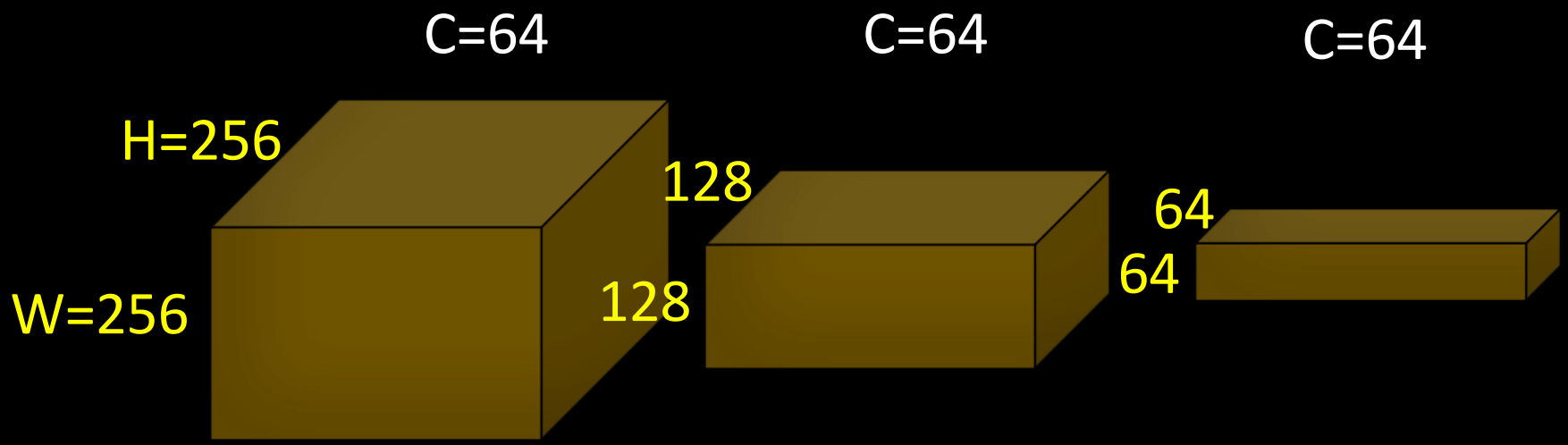
Can be in many location

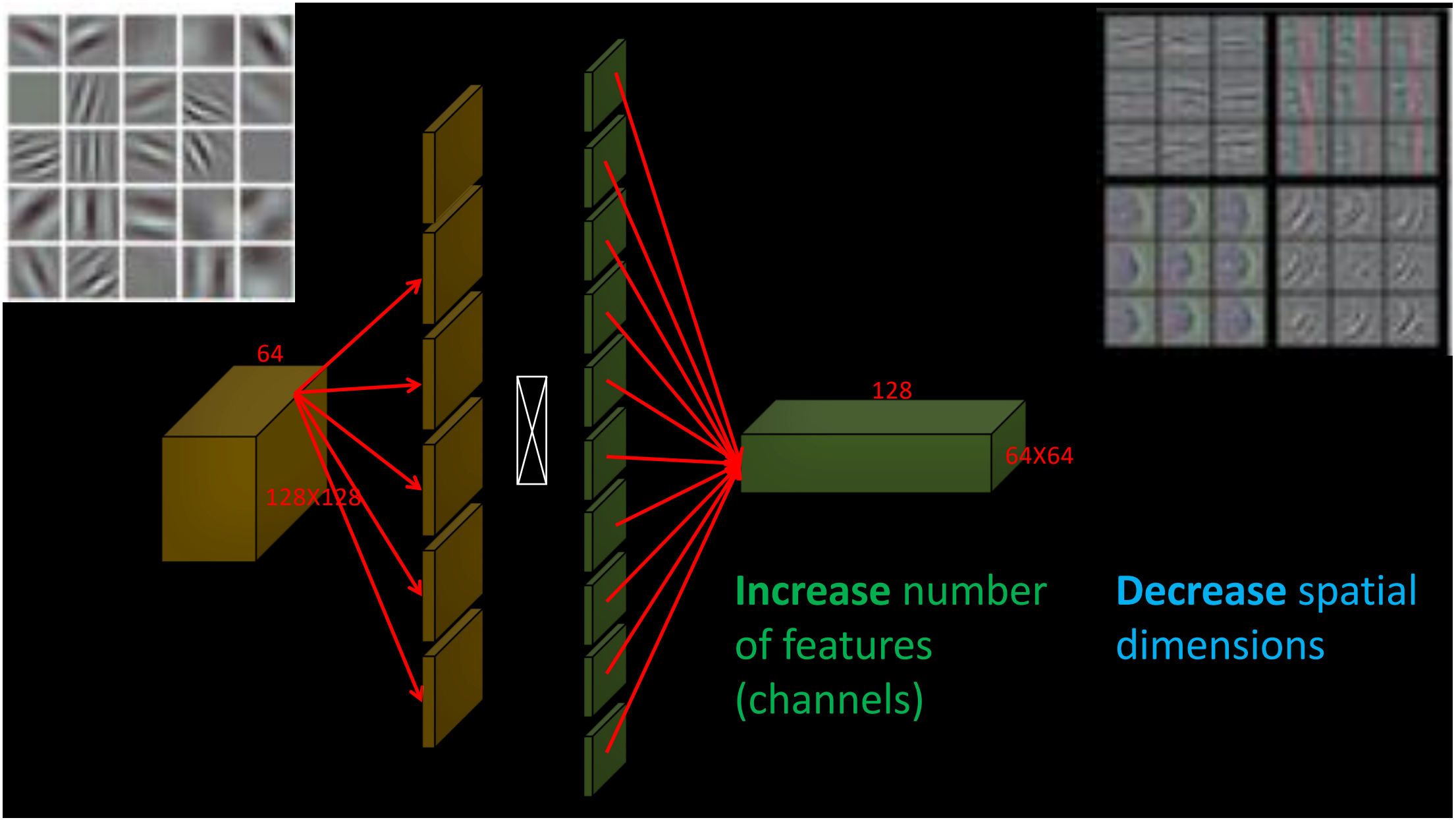
(Max)Pooling

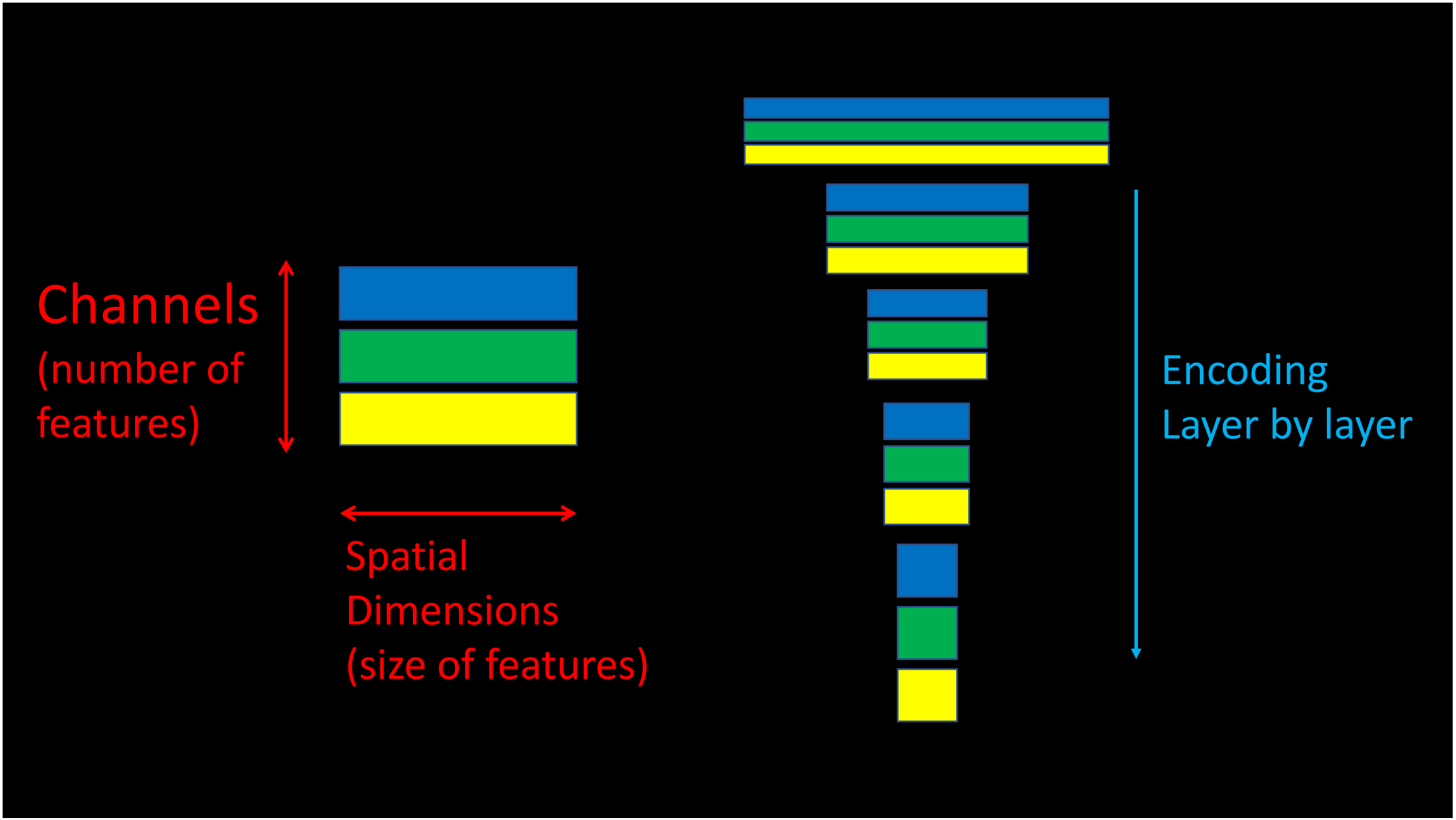


nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

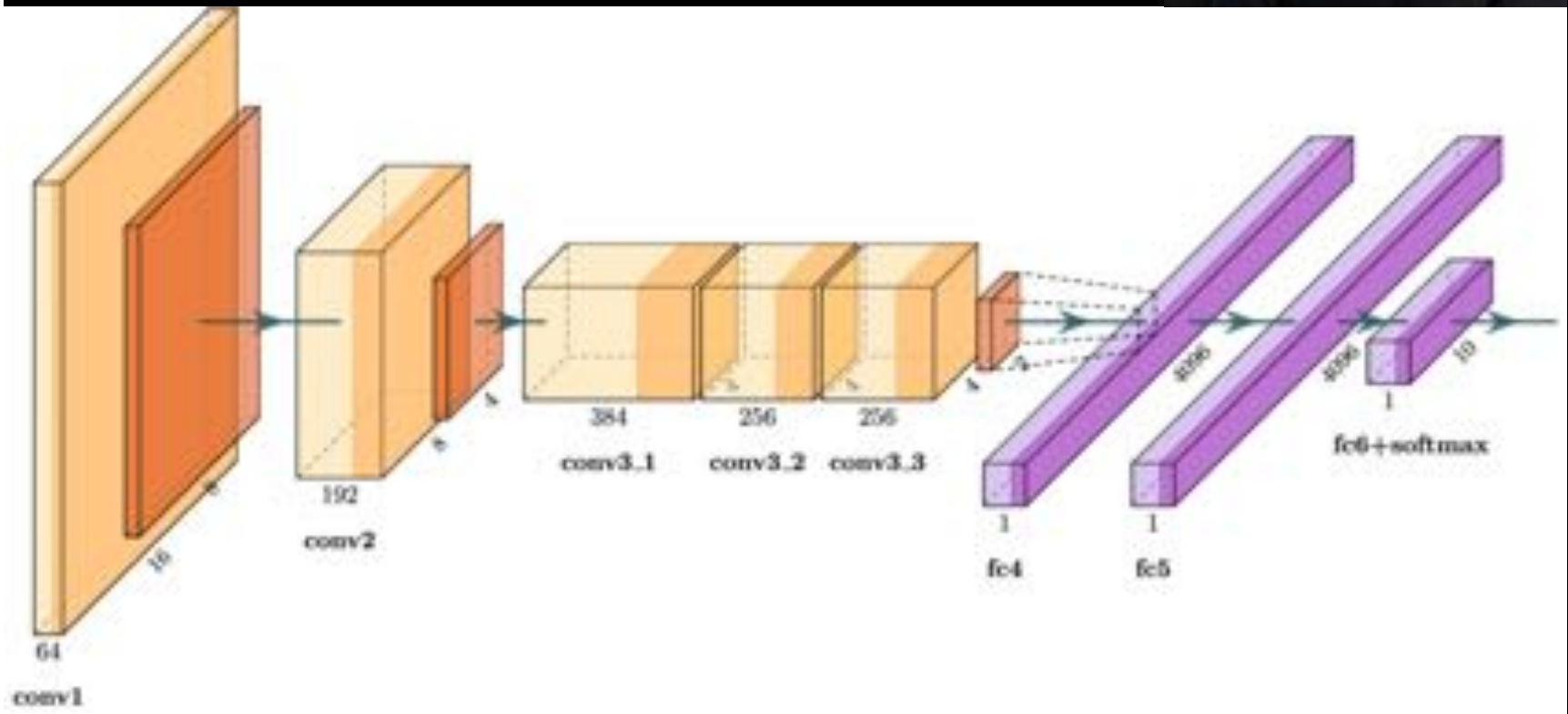
Pooling (decrease the XY dimension)

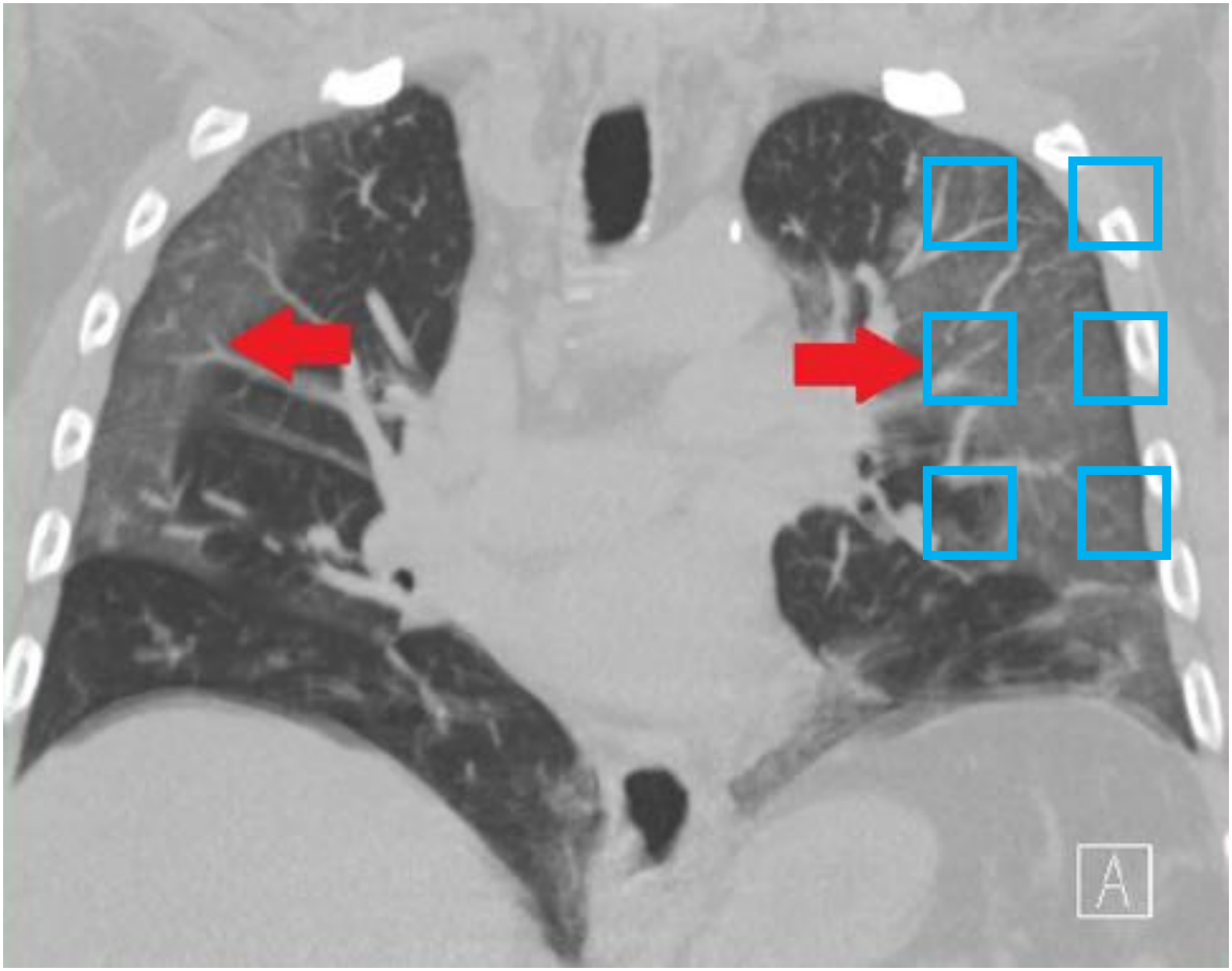



64
128X128
128
64X64
Increase number of features (channels)
Decrease spatial dimensions

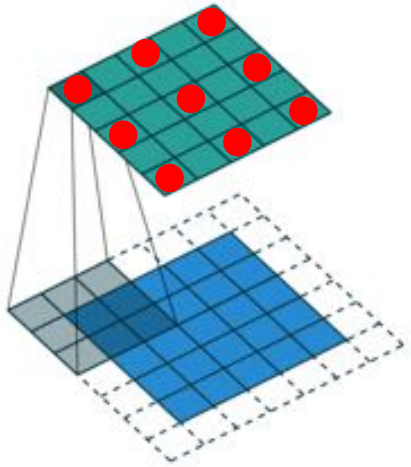

Channels
(number of features)
Spatial Dimensions
(size of features)
Encoding
Layer by layer

Actually an earlier version…
Alexnet

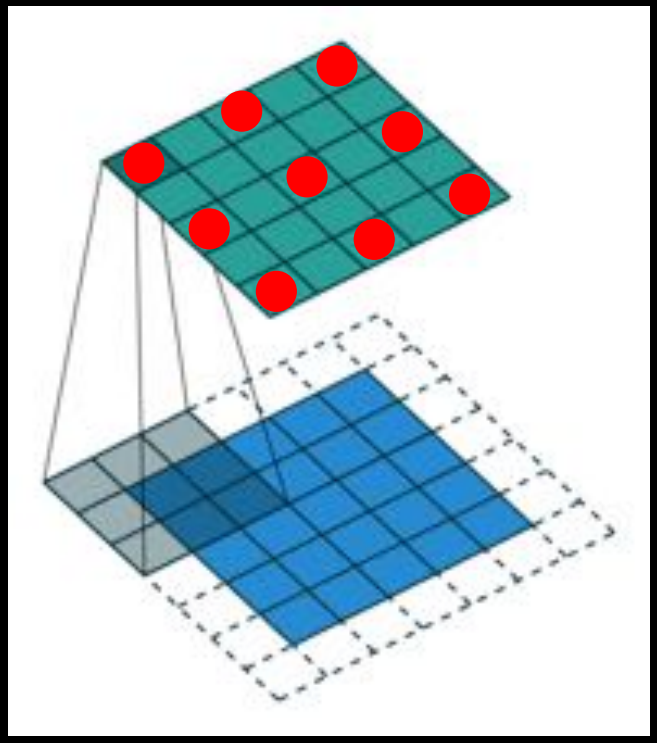




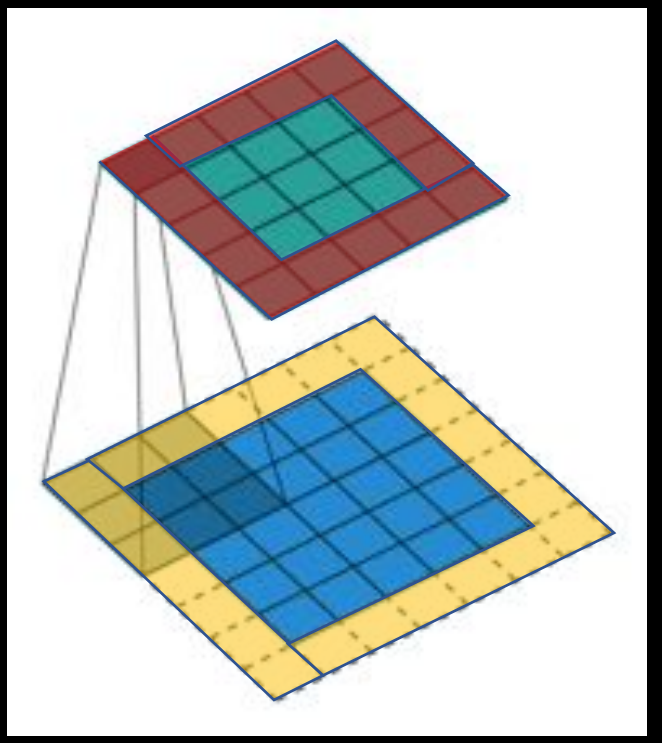
Stride:
We may not need to sample every point..



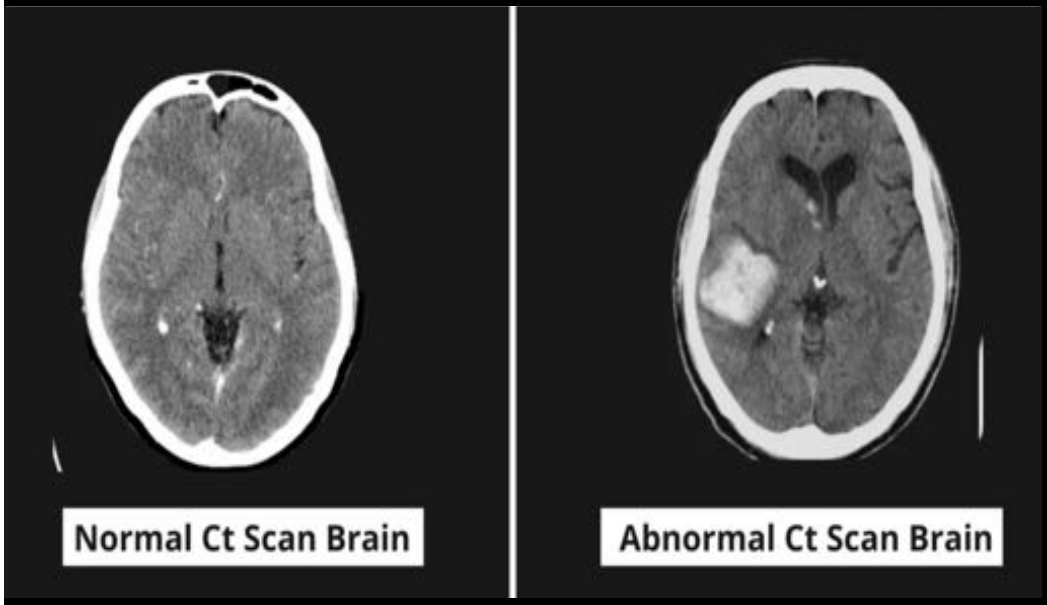
Stride

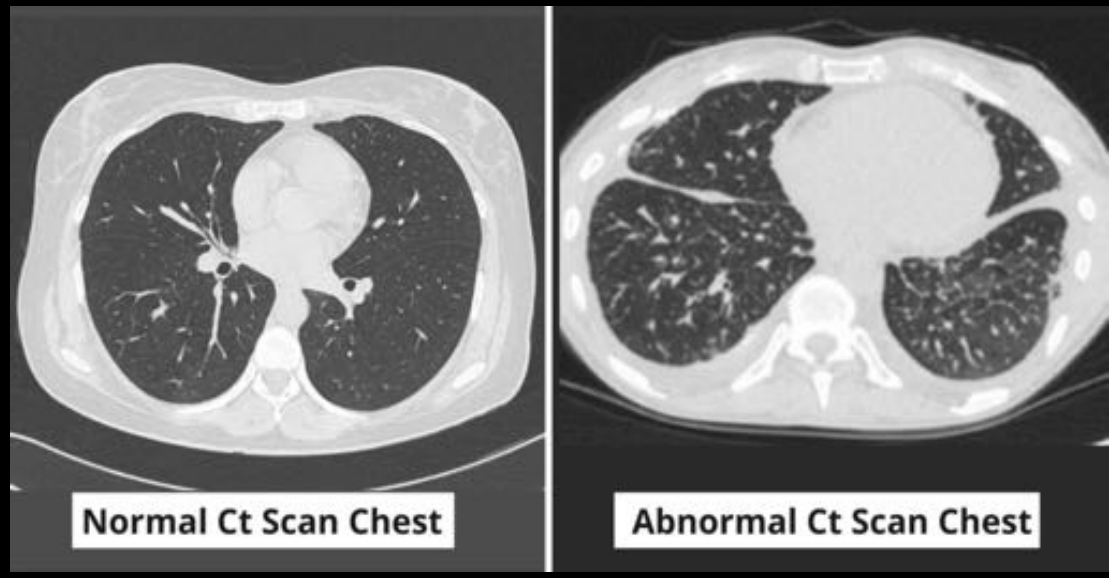


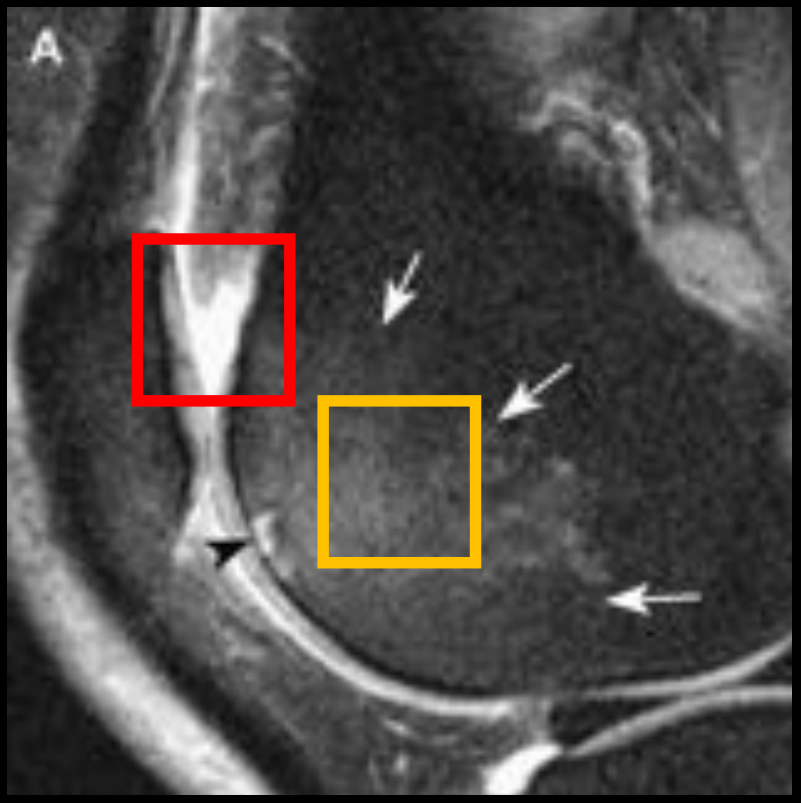
Padding



Large strides: risky for which case?

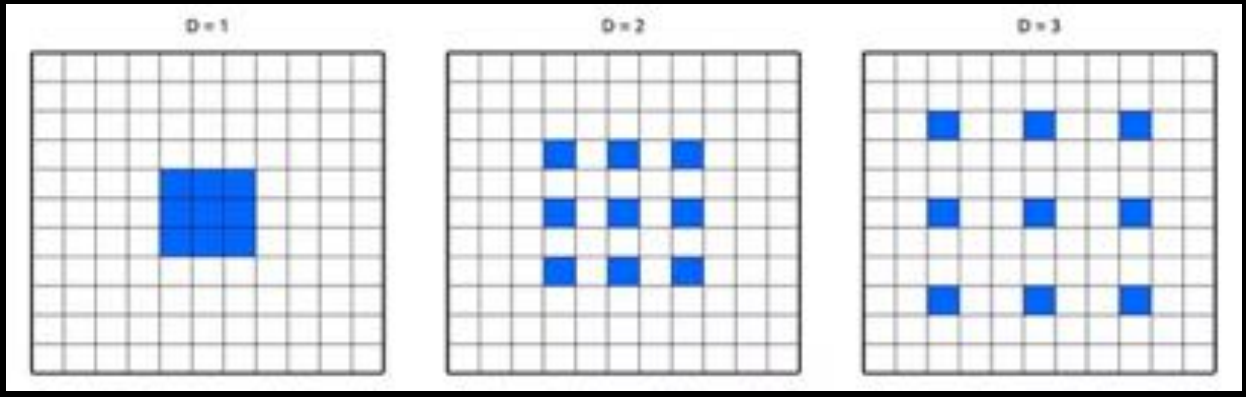




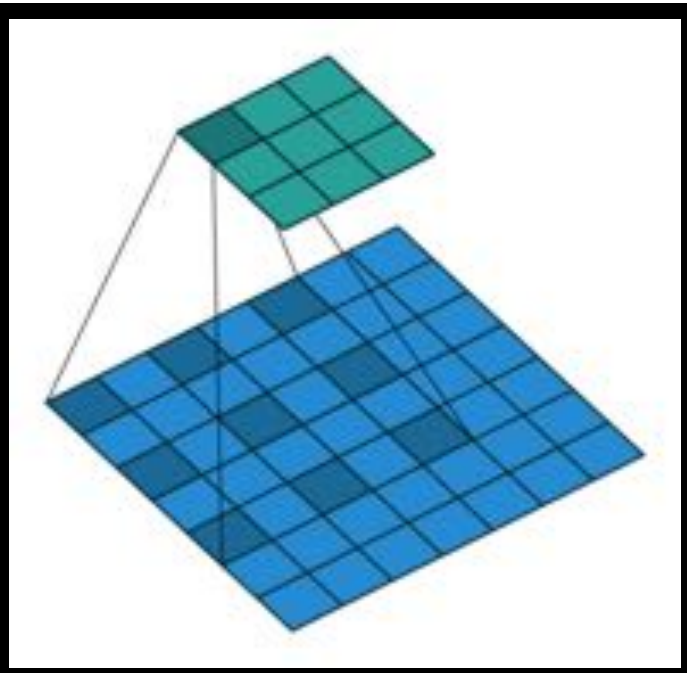


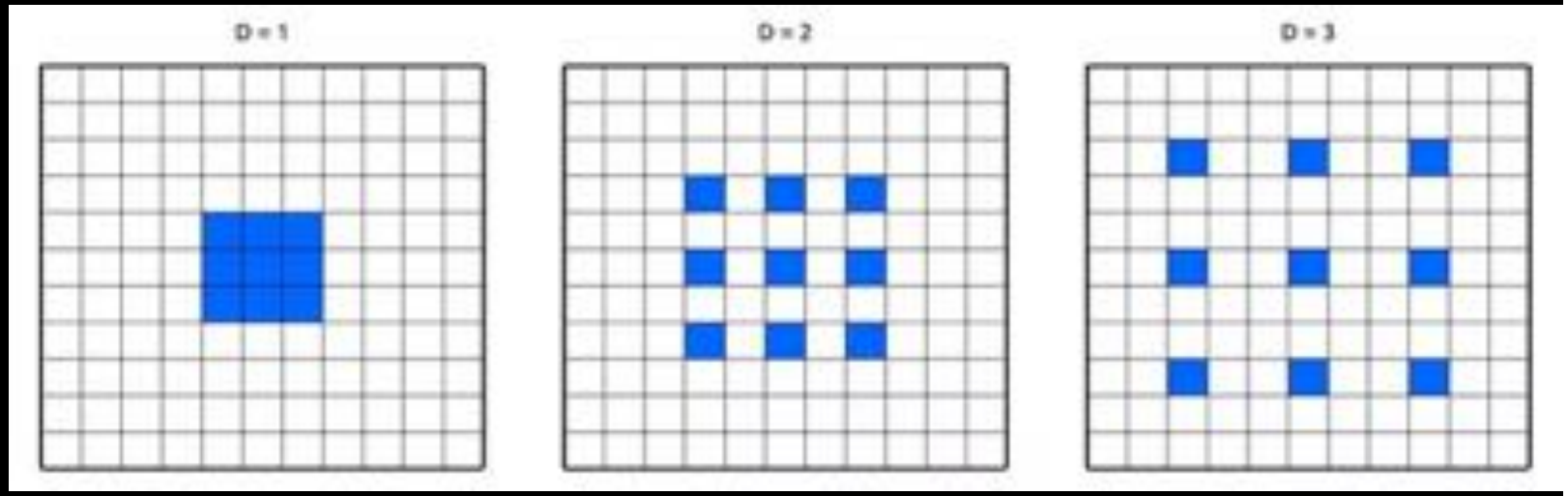
Dilation:

We may want to see further

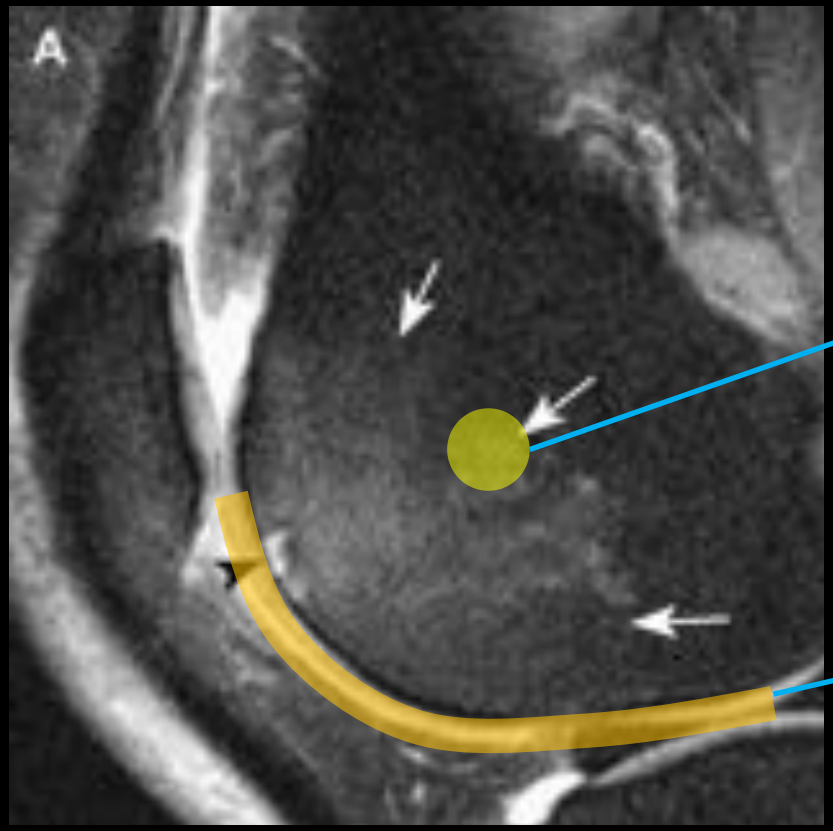


Dilation





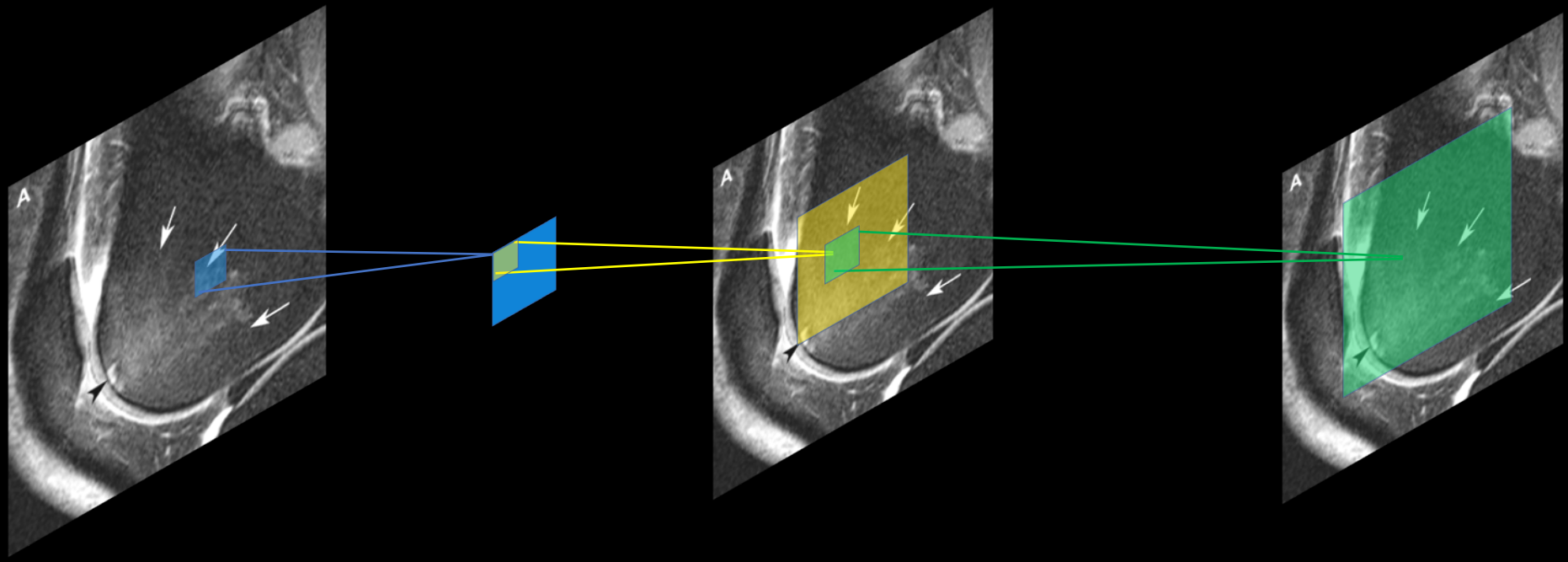
How can the CNN see information afar?

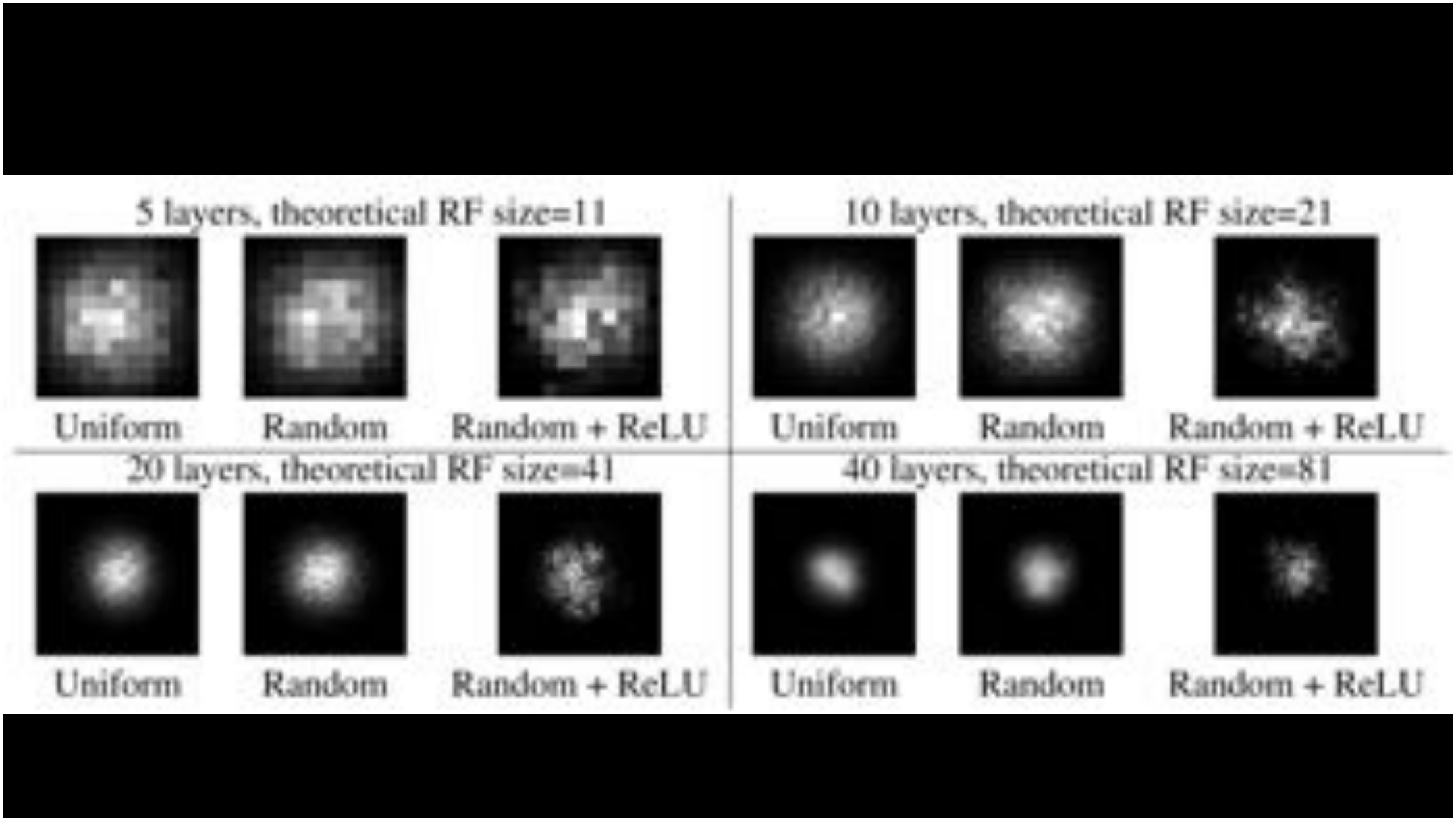


I know it's a lesion

But I also need to see the afar structural to diagnosis sit

Receptive Field

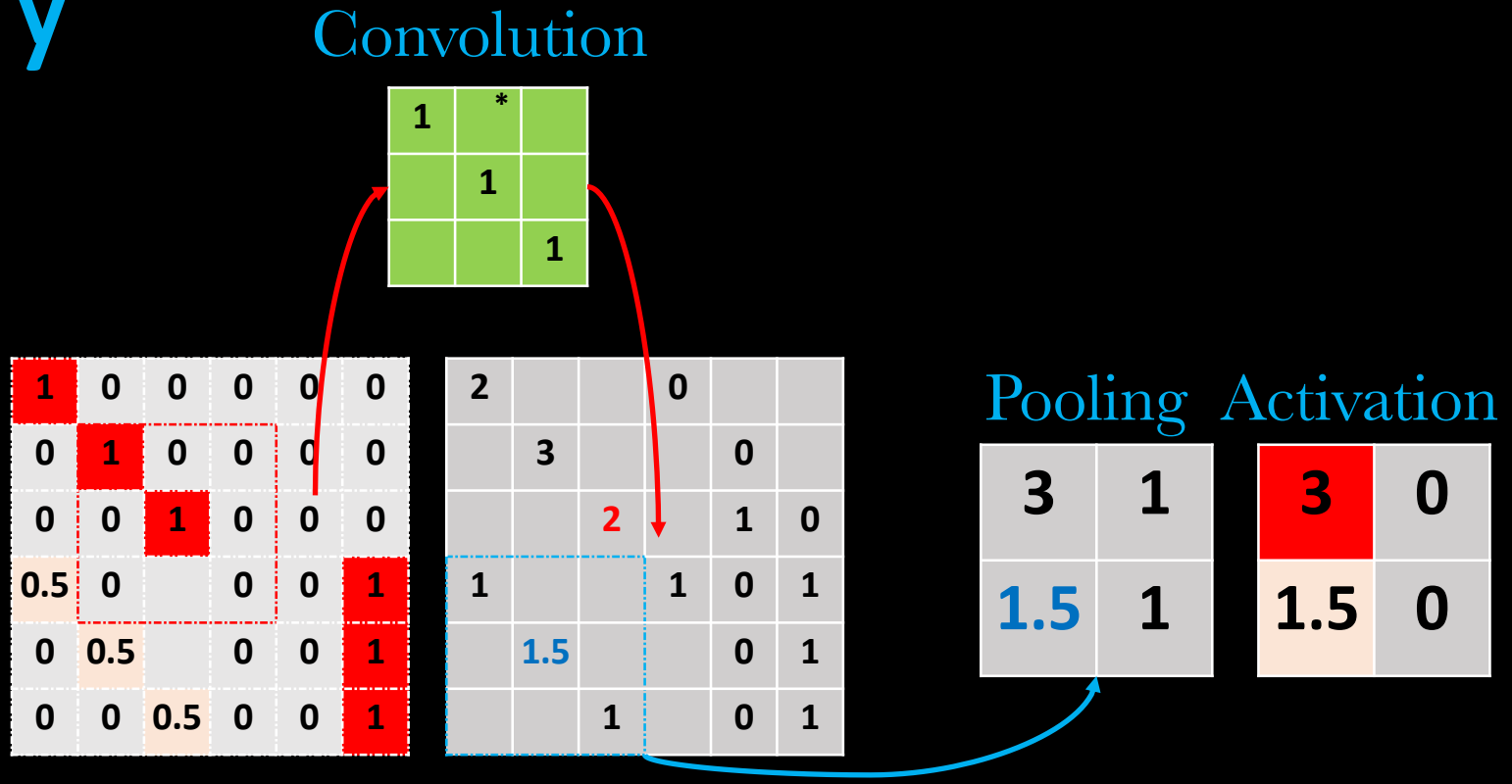


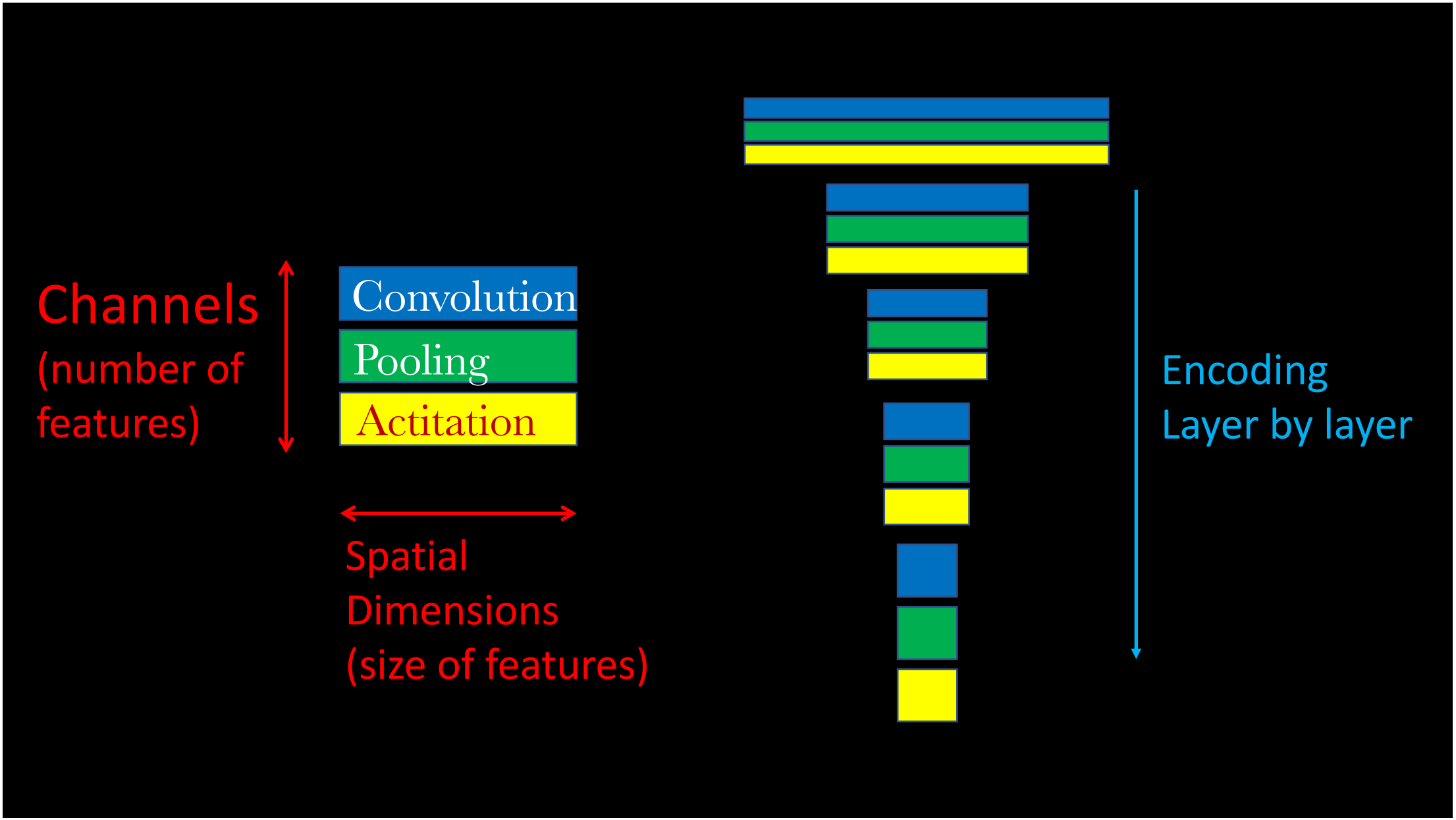

5 layers, theoretical RF size=11
Uniform
Random
Random + ReLU
10 layers, theoretical RF size=21
Uniform
Random
Random + ReLU
20 layers, theoretical RF size=41
Uniform
Random
Random + ReLU
40 layers, theoretical RF size=81
Uniform
Random
Random + ReLU

Summary

Basic "Stack" of CNN layers

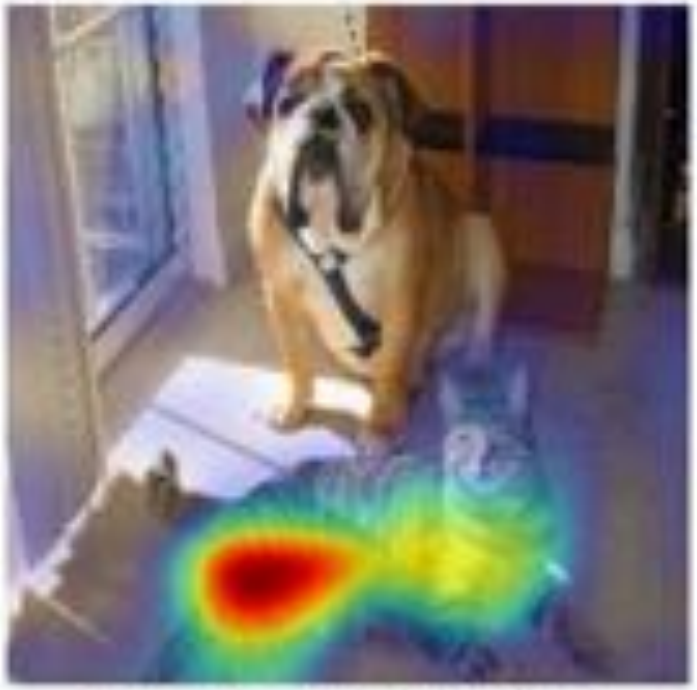
- Convolution
- Pooling
- Actitation



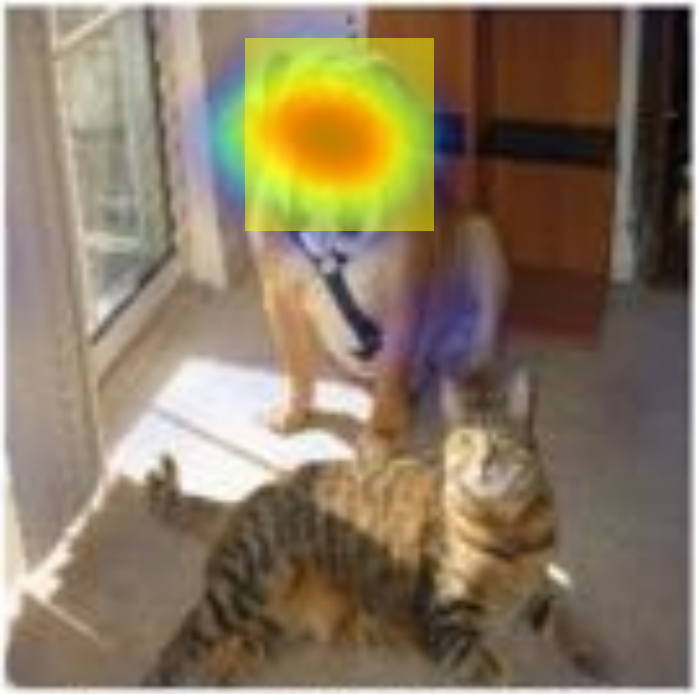

Channels
(number of features)
Convolution
Pooling
Actitation
Spatial Dimensions
(size of features)
Encoding
Layer by layer

Nature Imaging: Often Large Discriminative Parts

(a) Original Image

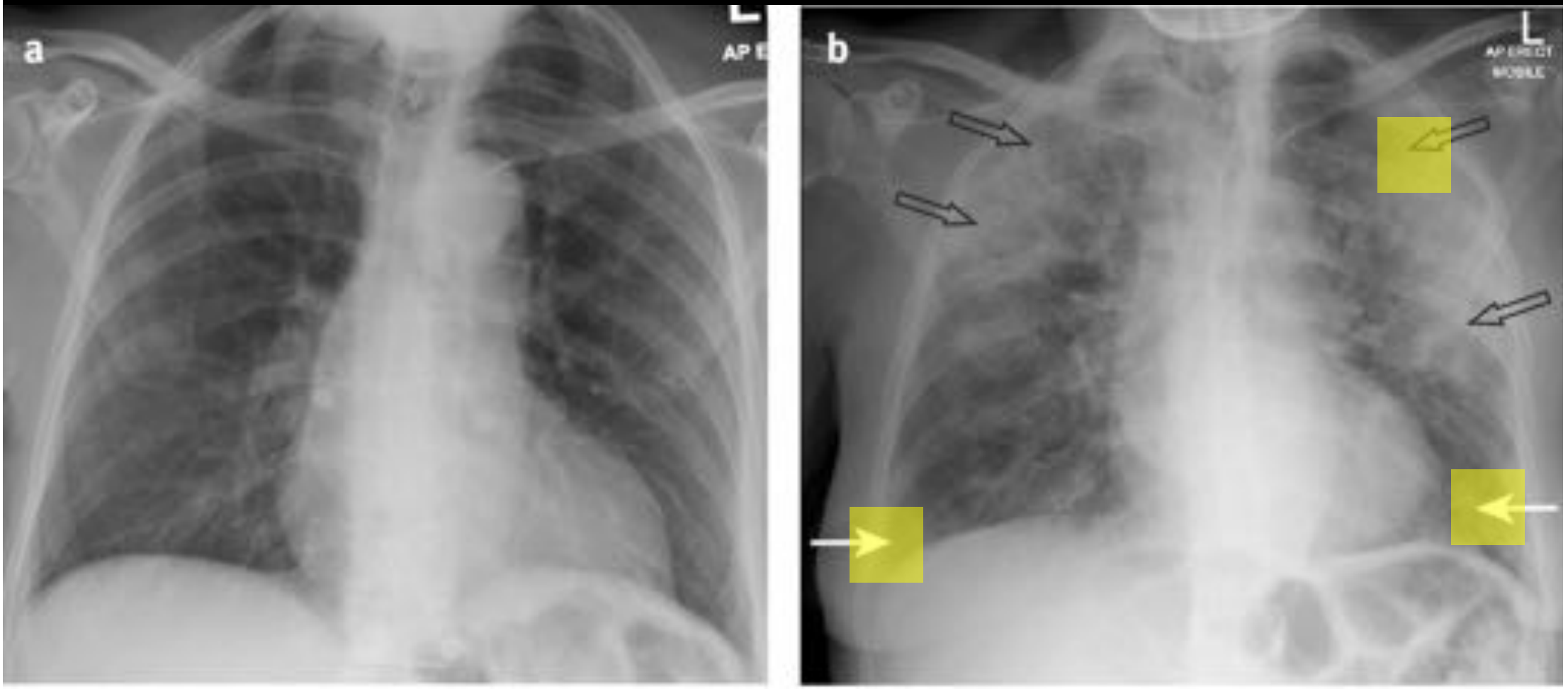


(b) Grad-CAM 'Cat'

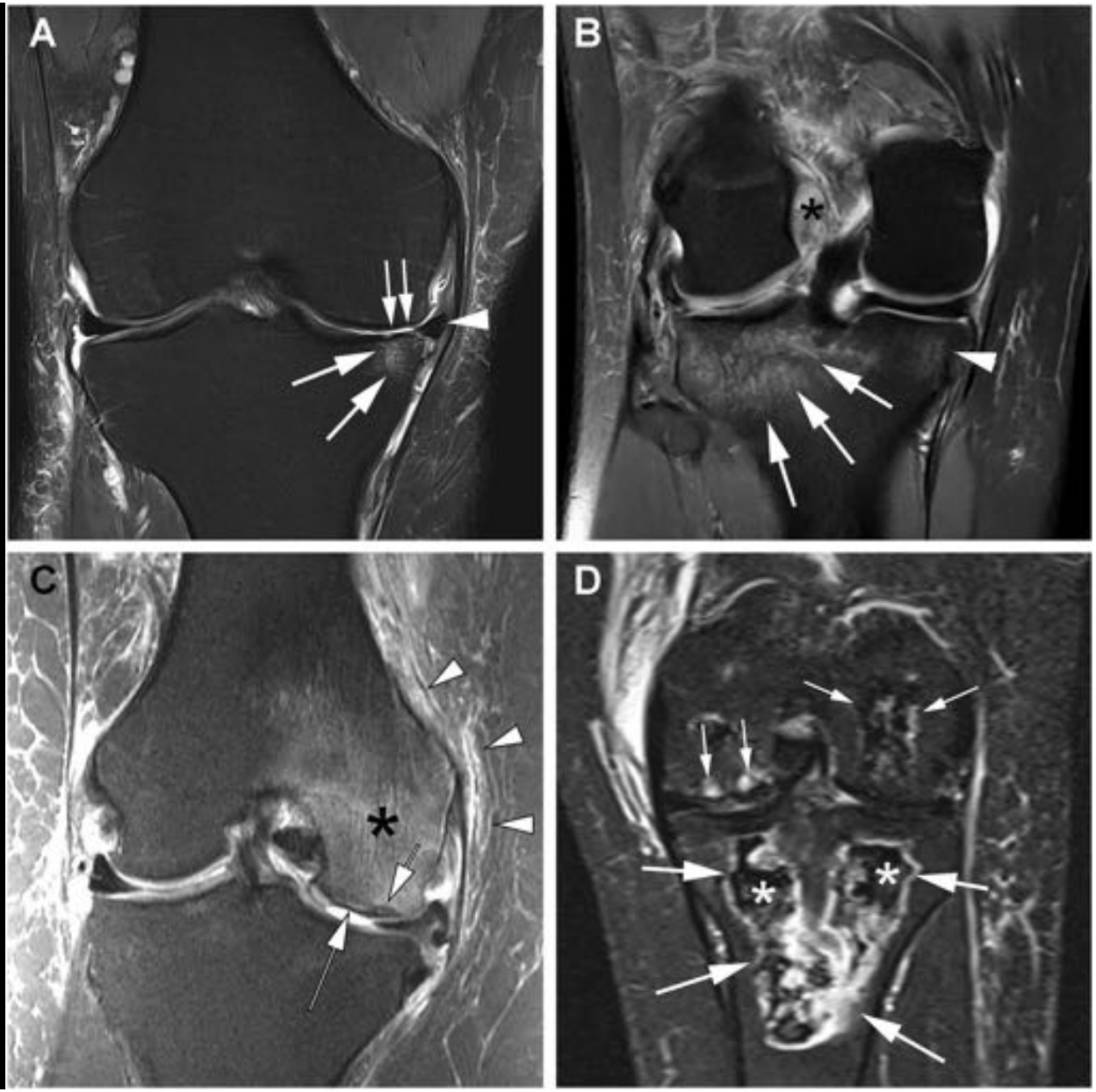


(c) Grad-CAM 'Dog'

Medical Imaging: Fine-grained Discriminative Parts



Medical Imaging:
Fine-grained Discriminative Parts

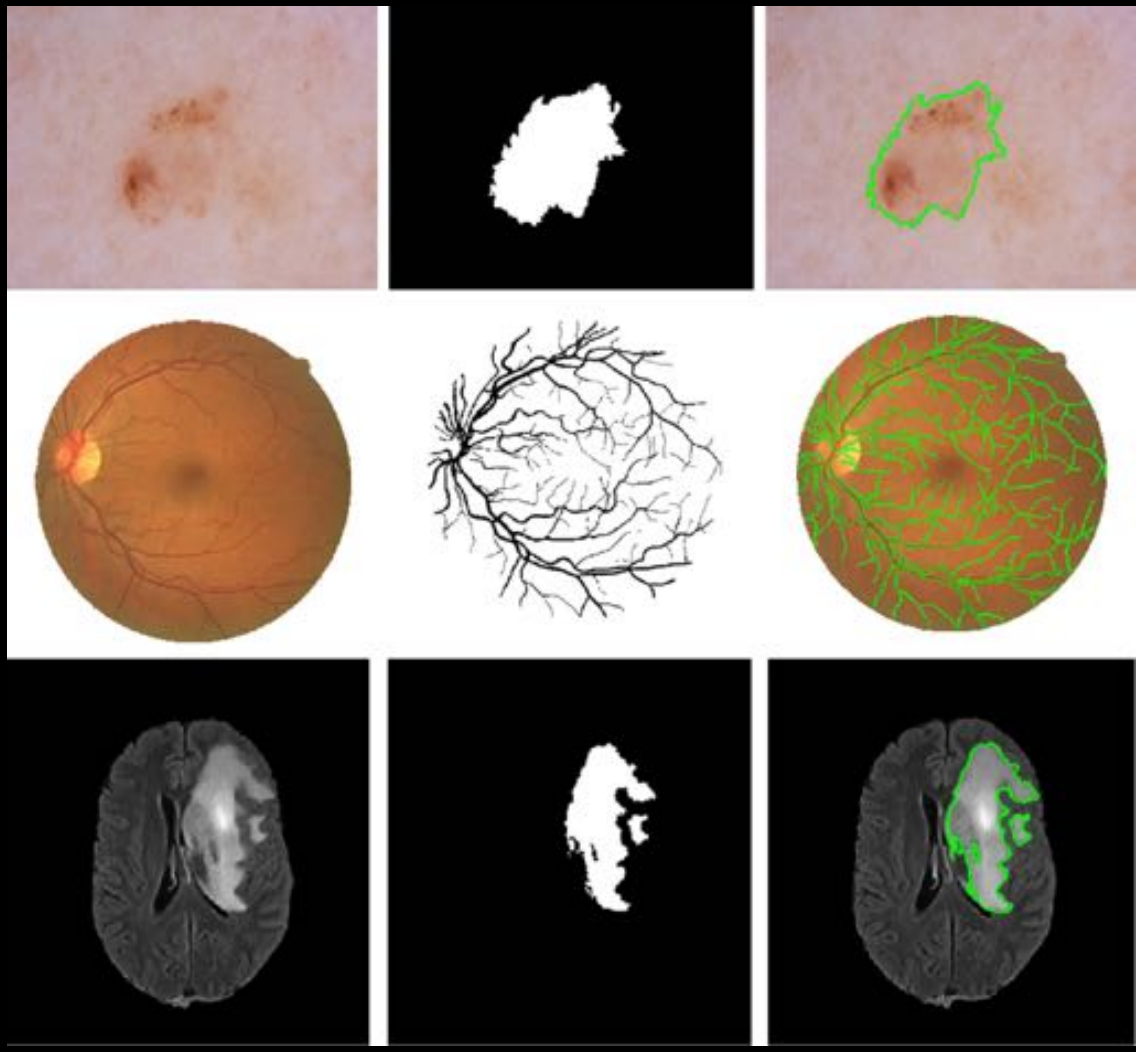


We need features possibly

from different scales

can be high level or low level

can be vastly different between modalities / applications

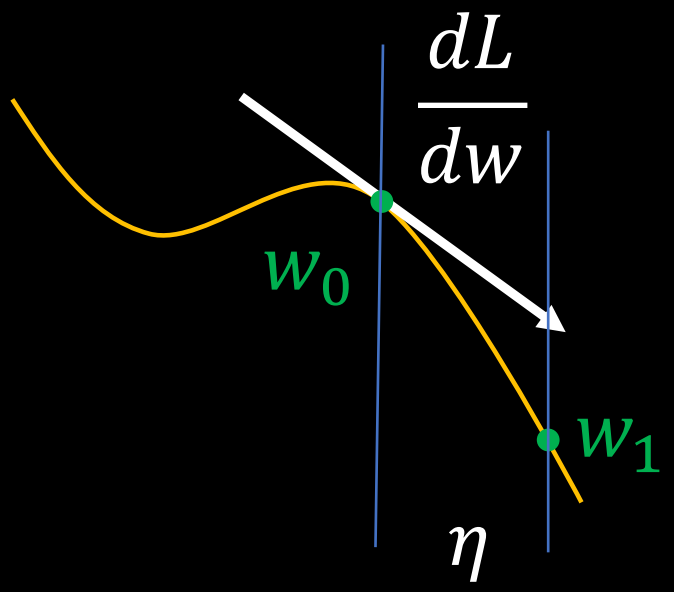


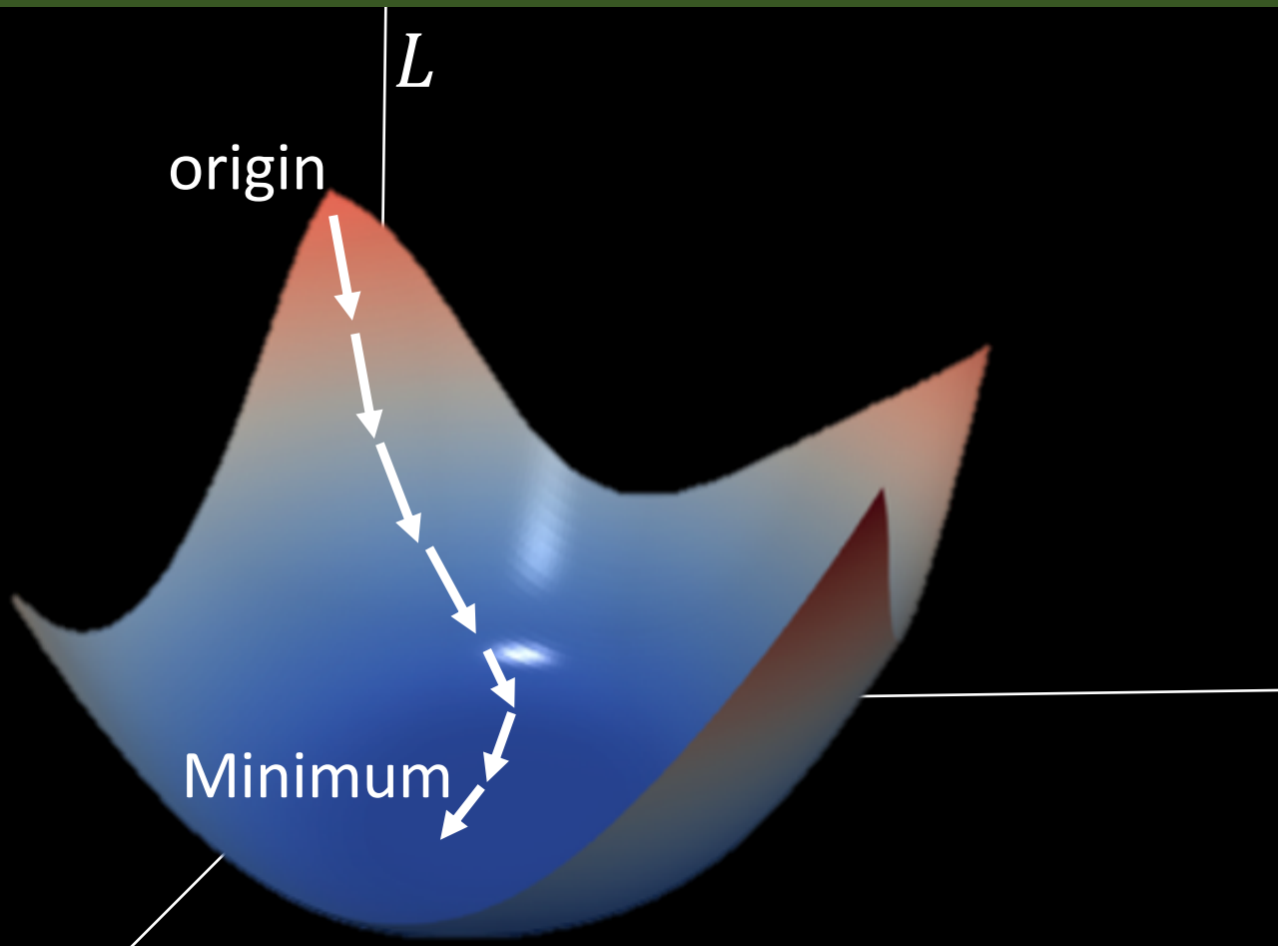
Optimization &
Stochastic Gradient Descend (SGD)

Training Neural Networks: optimization

Optimizer

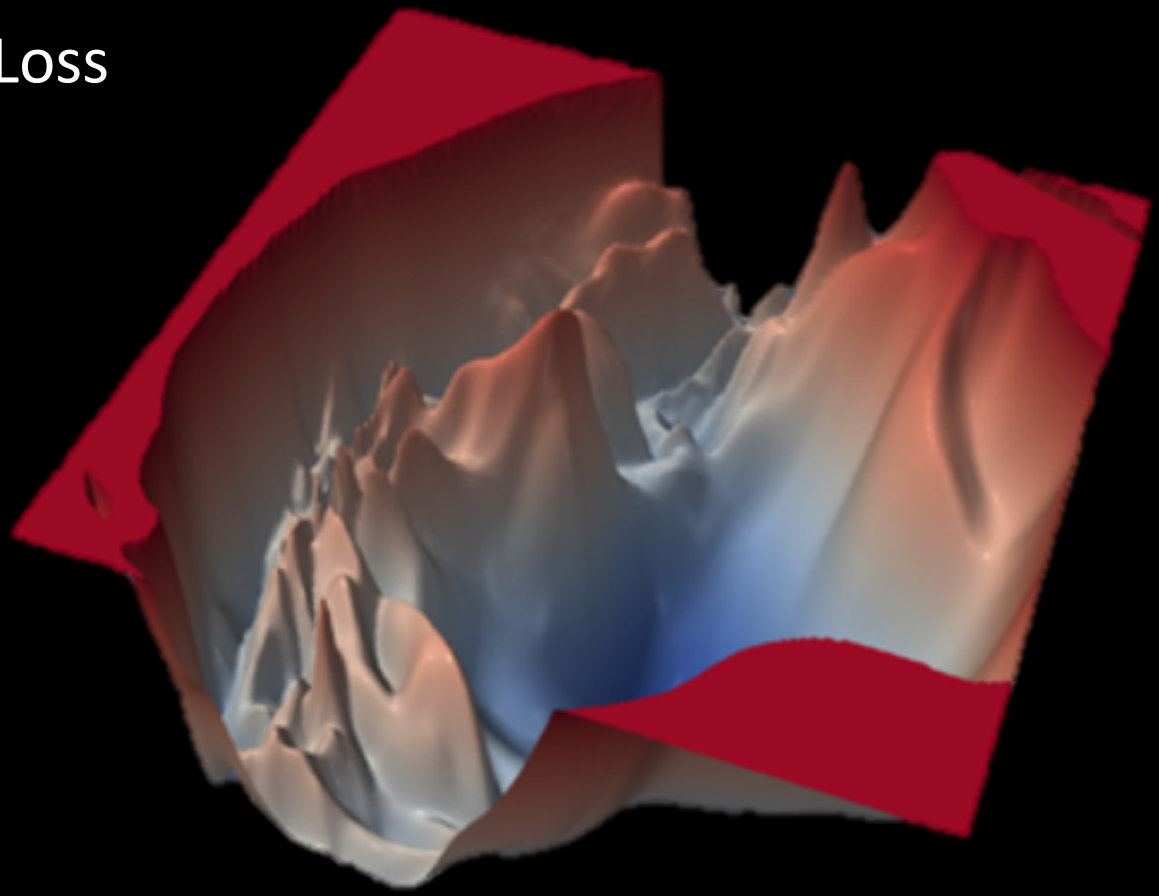
$argmin L(w_i, b)$

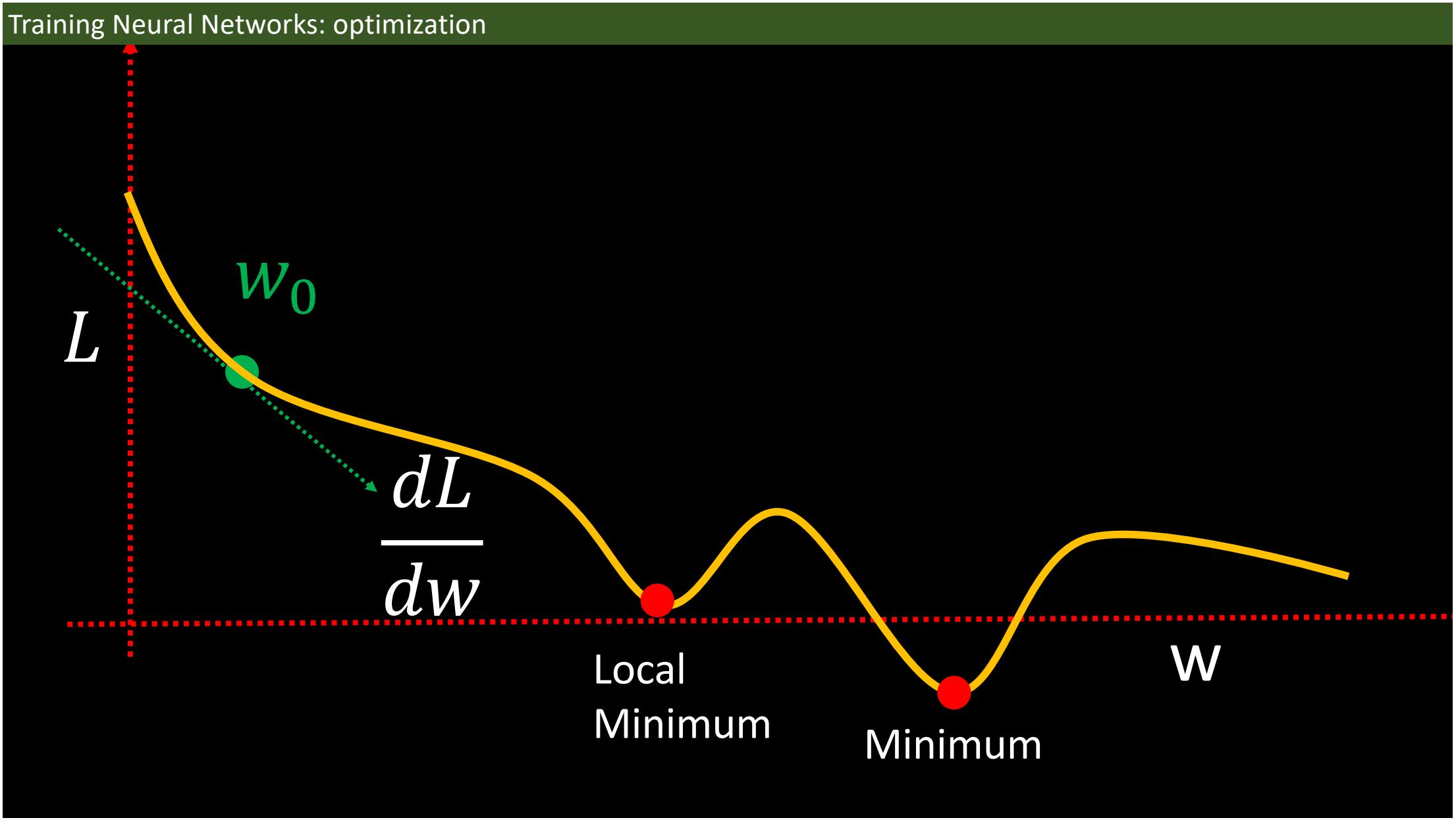




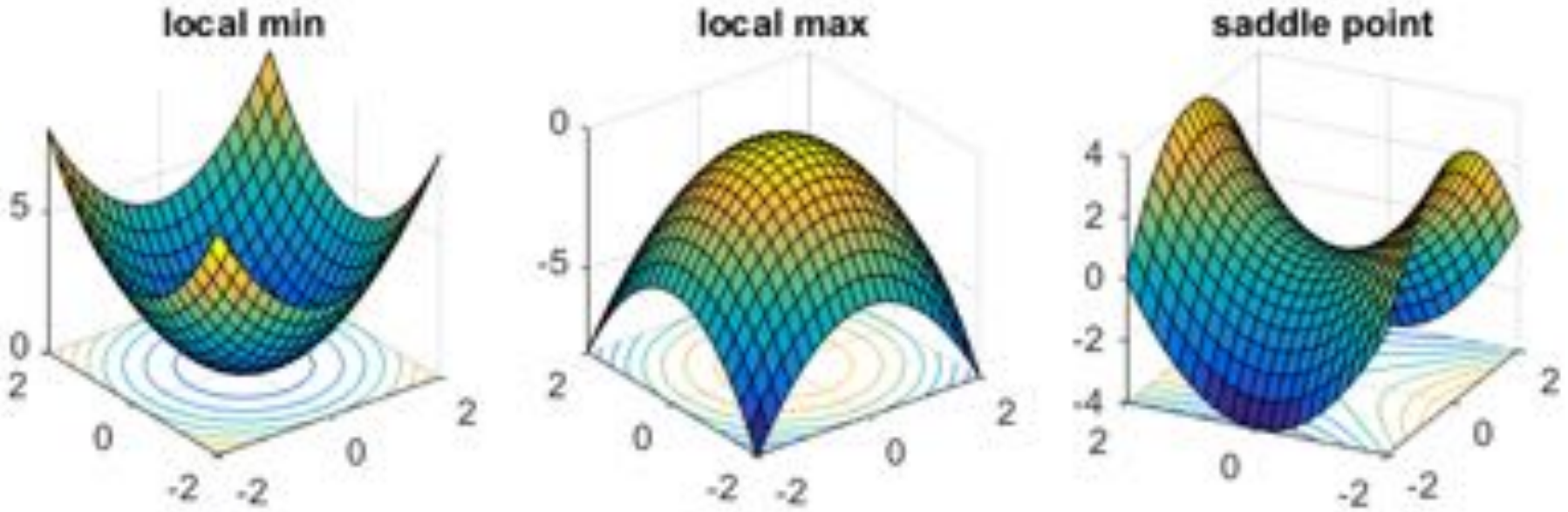
$$w_1 = w_0 - \frac{dL}{dw}\eta$$

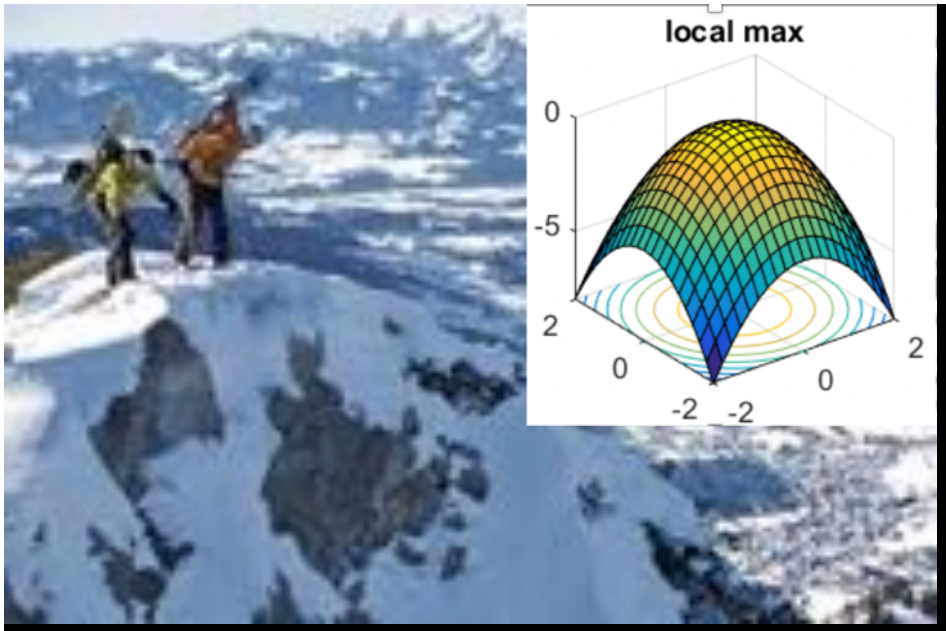
Real Surface of Loss



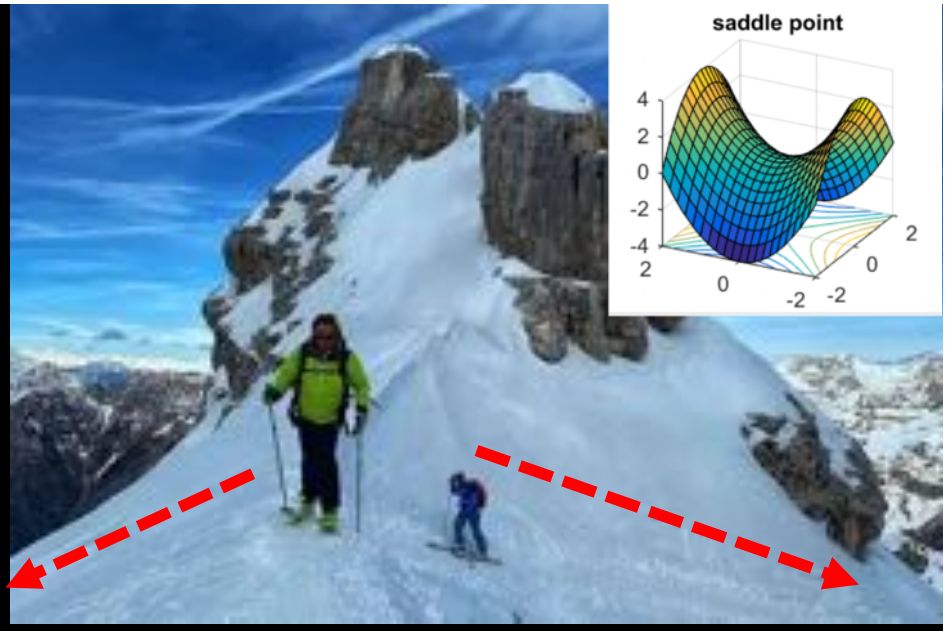

Training Neural Networks: optimization
L
w_0
$\frac{dL}{dw}$
Local
Minimum
Minimum
W

Training Neural Networks: optimization

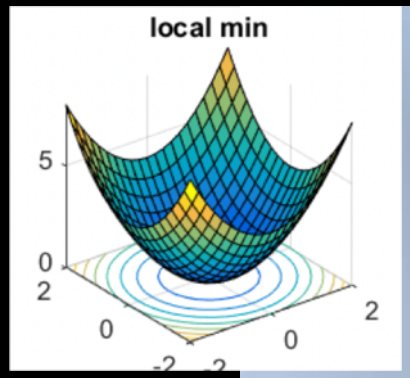




We don't care about this

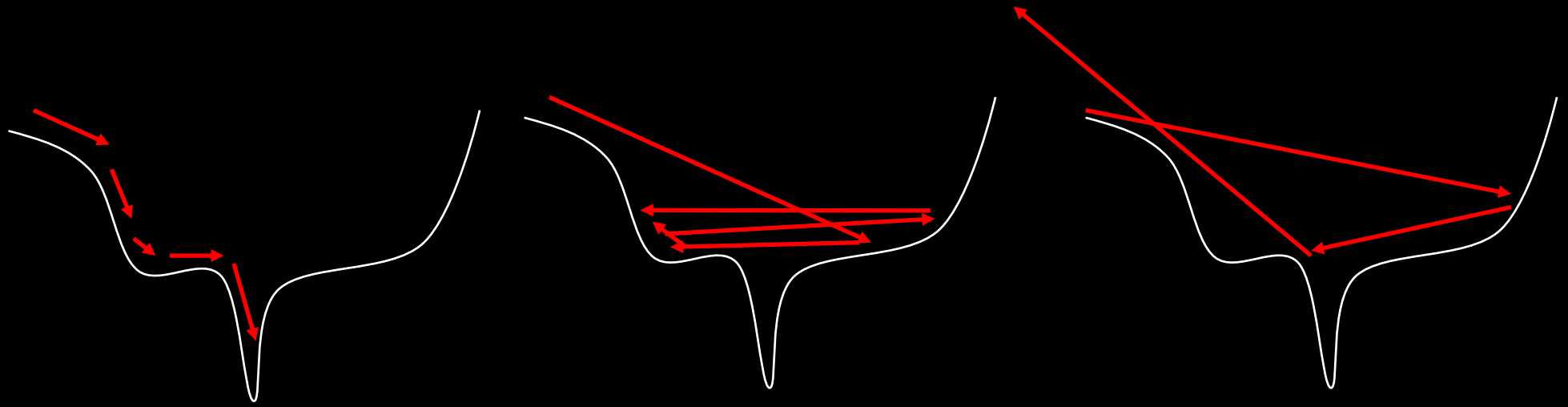


Most common

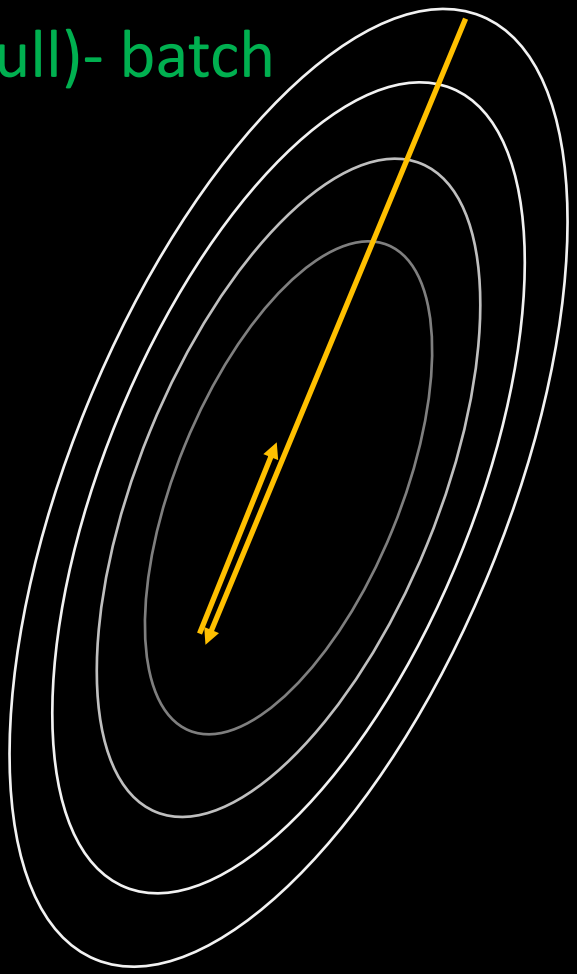


Your model is stuck

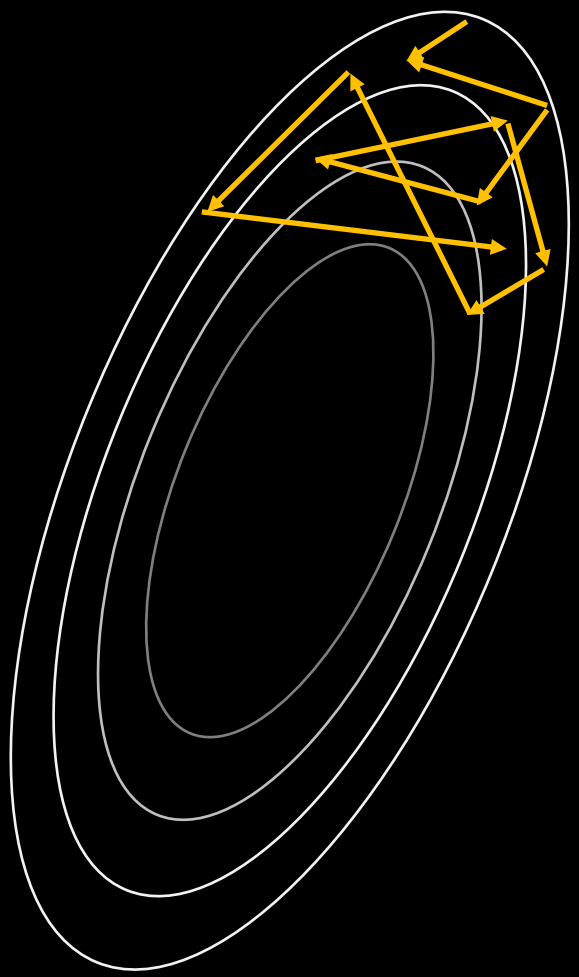
Avoiding stuck in local minimum
1: learning rate



(Full)- batch



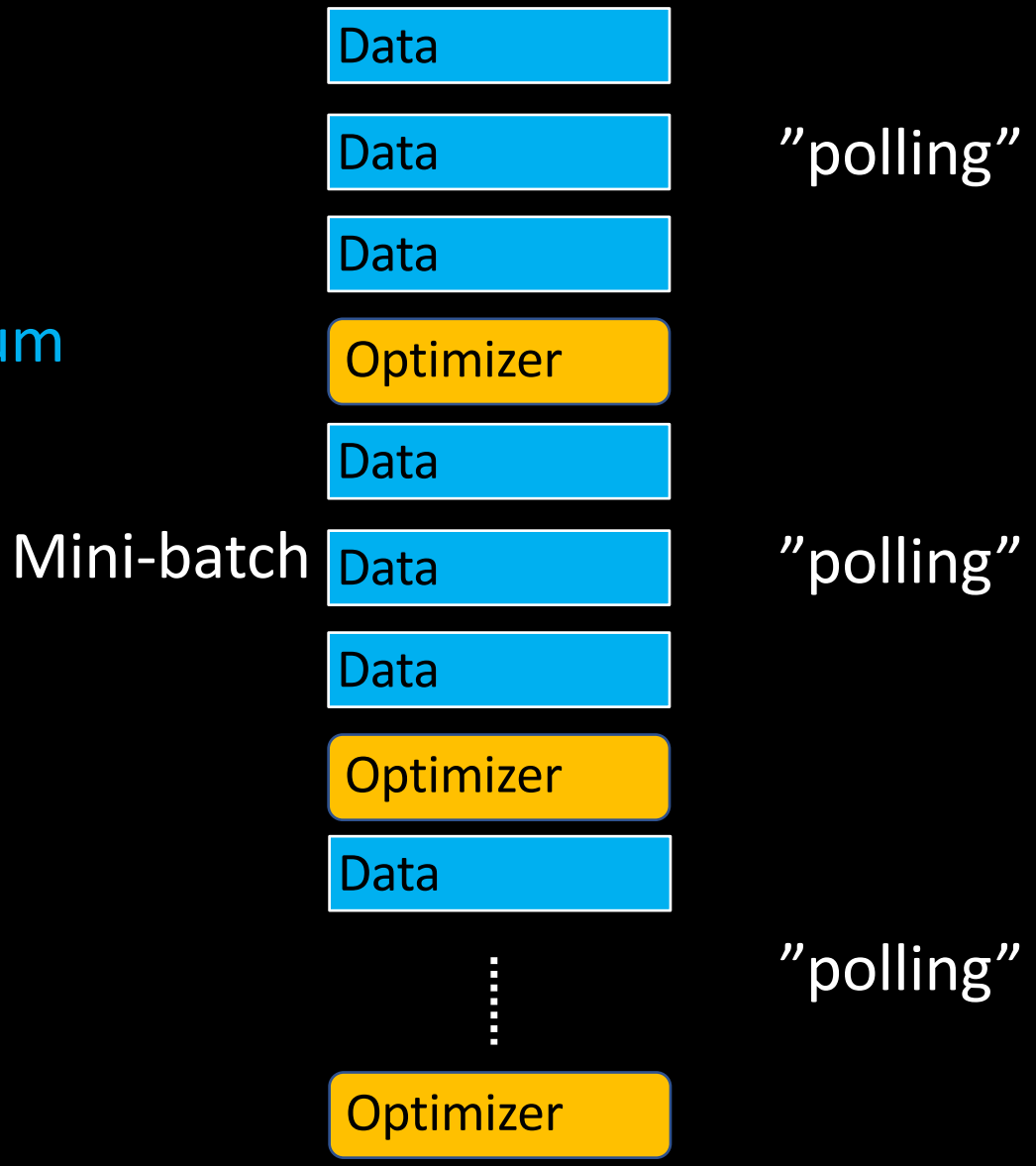
“deterministic”

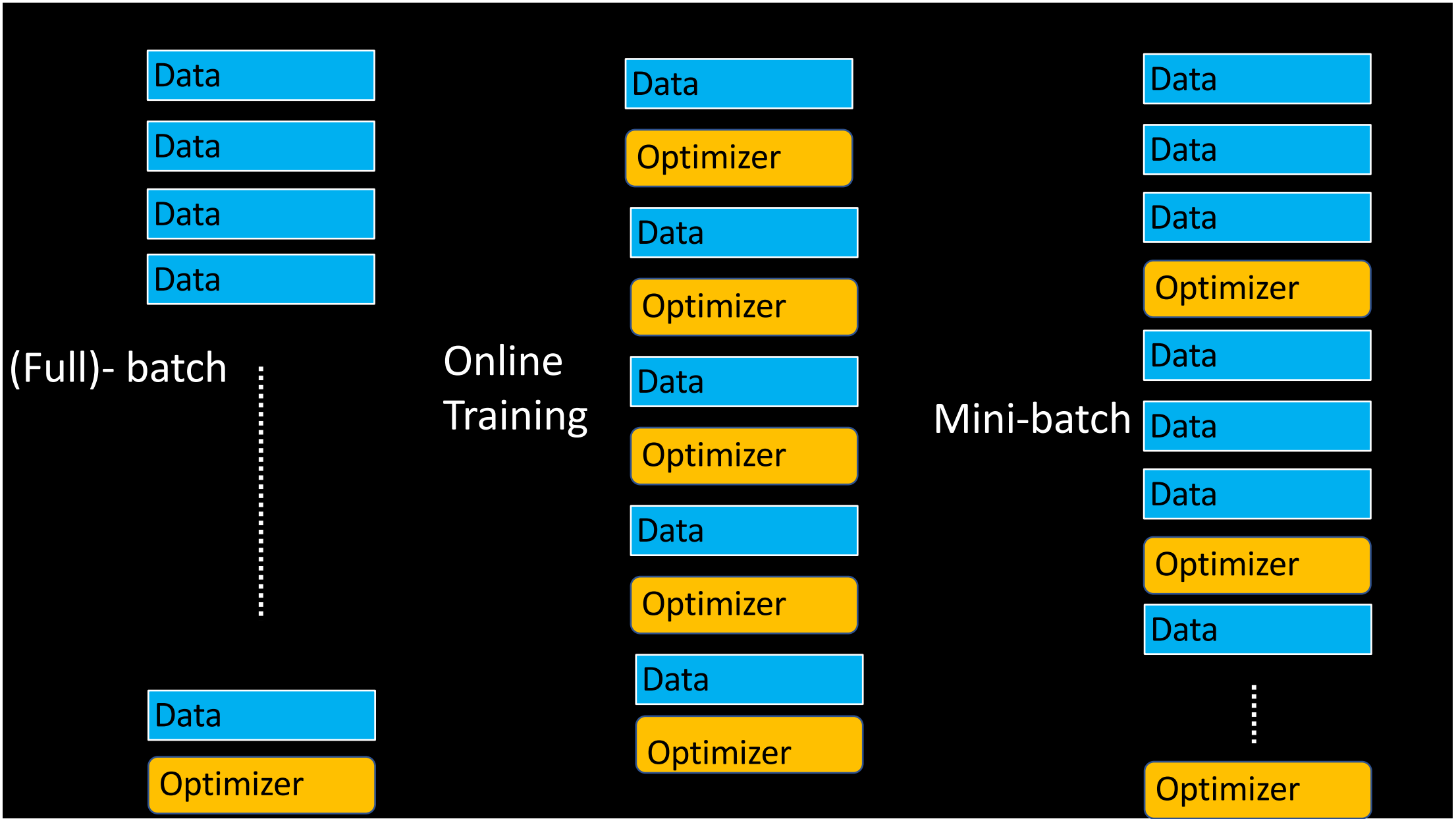


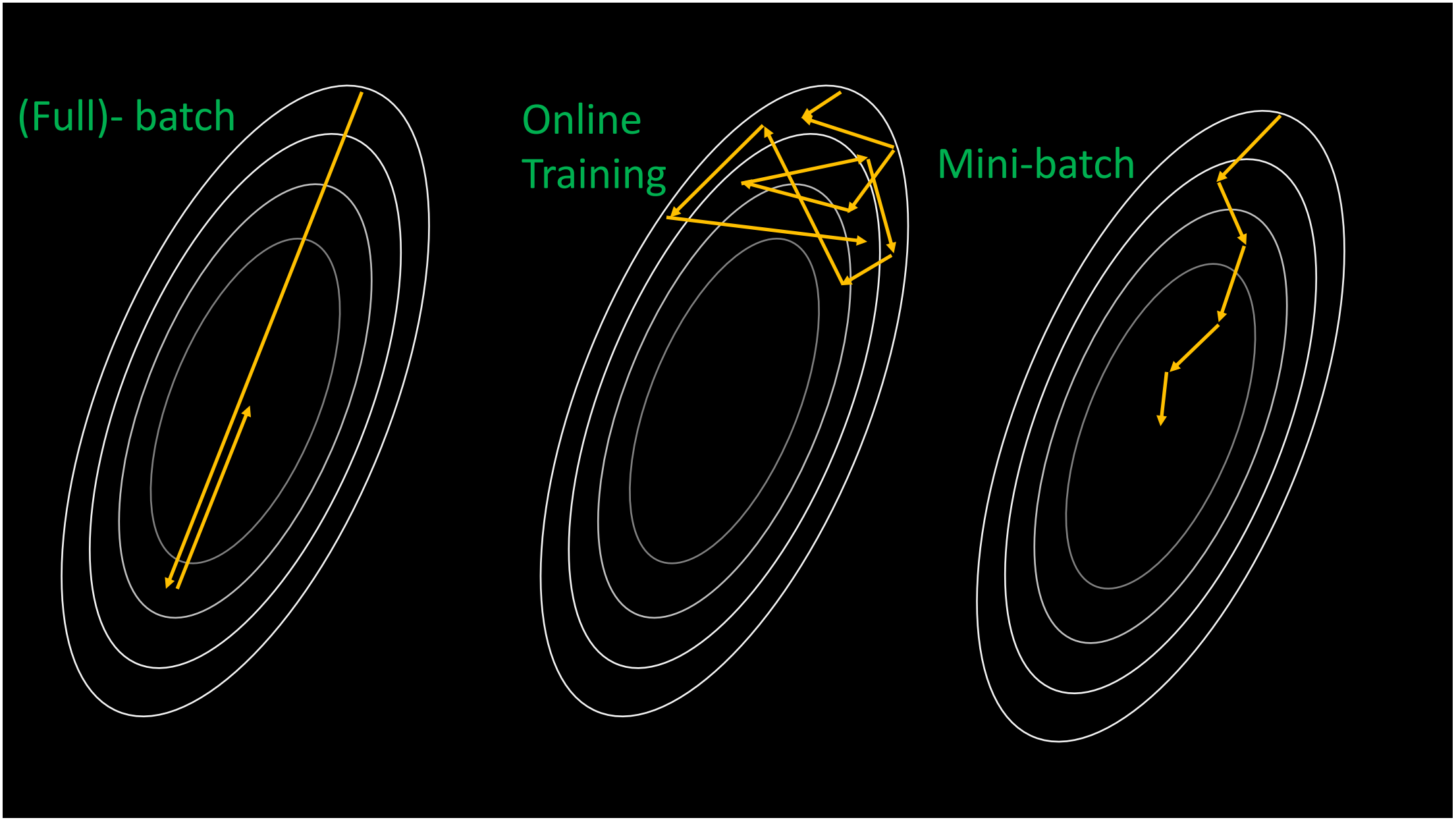
Online (one by one)
Training

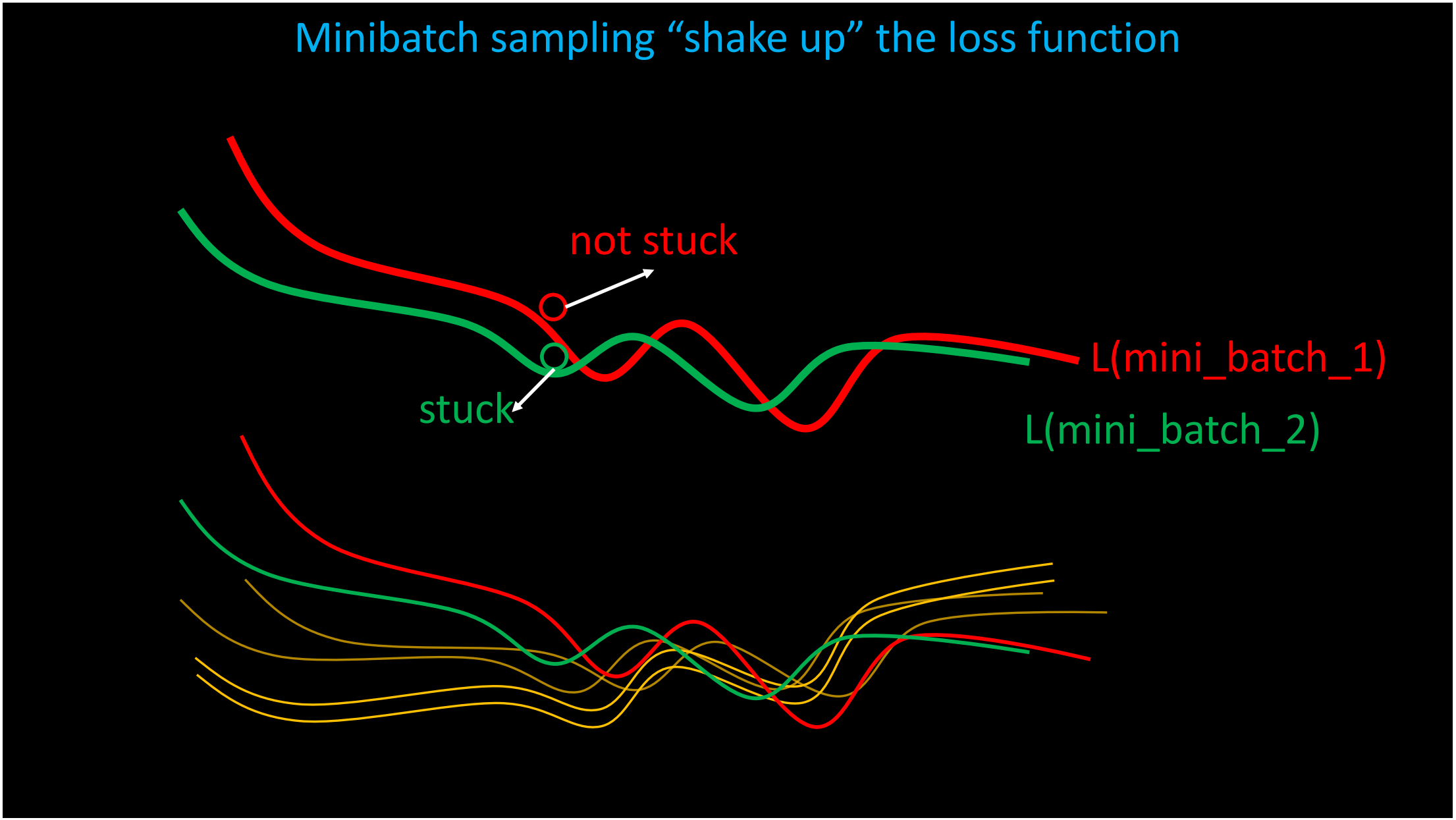
“tend to be crazy”

Avoiding stuck in local minimum
2: minibatch sampling

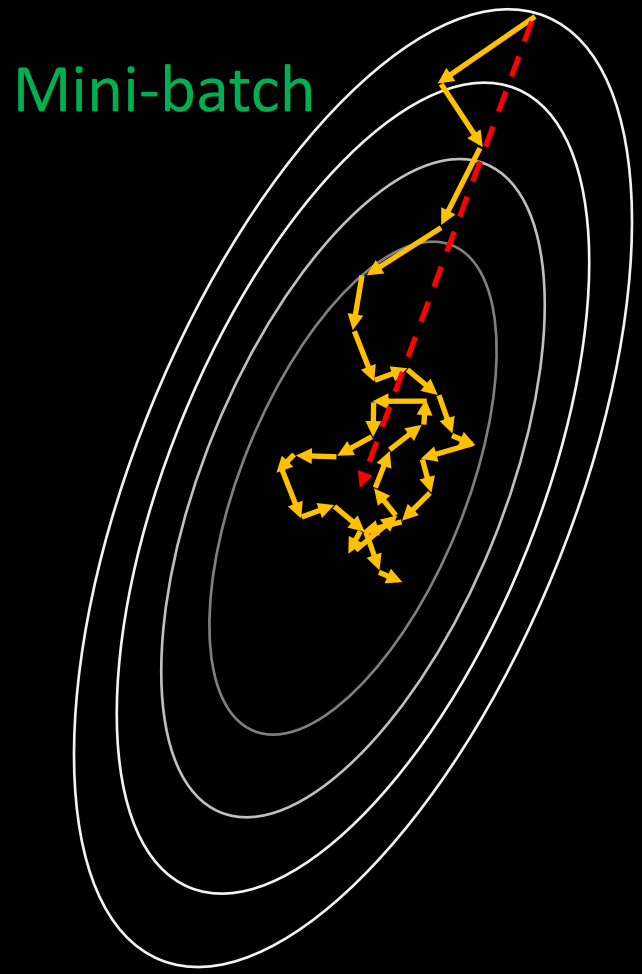


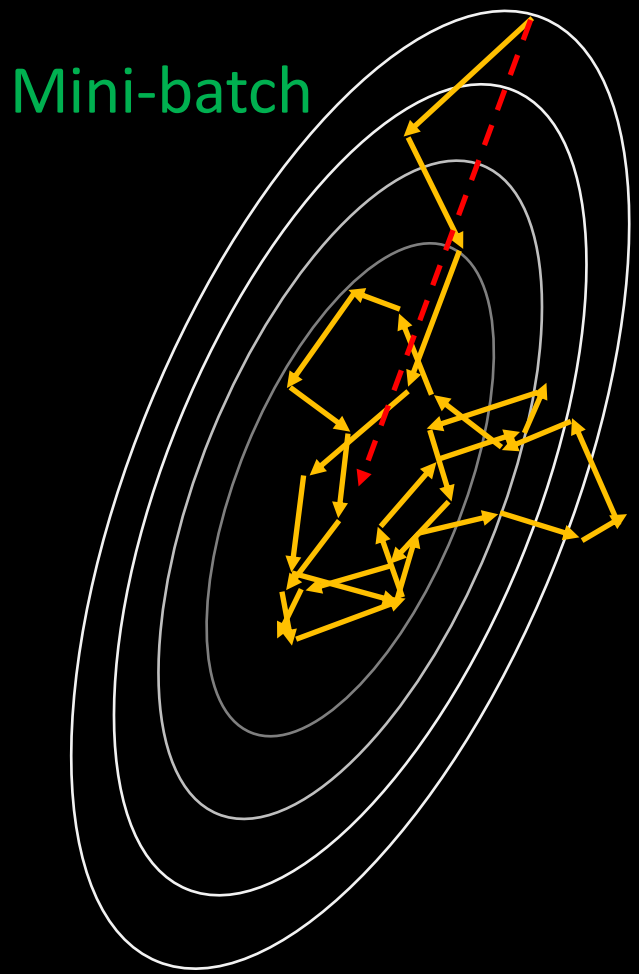

Data
Data
Data
Data
(Full)- batch
Data
Optimizer
Online
Training
Data
Optimizer
Data
Optimizer
Data
Optimizer
Data
Optimizer
Data
Optimizer
Mini-batch
Data
Data
Data
Optimizer
Data
Data
Data
Optimizer
Data
Optimizer


(Full)- batch
Online
Training
Mini-batch

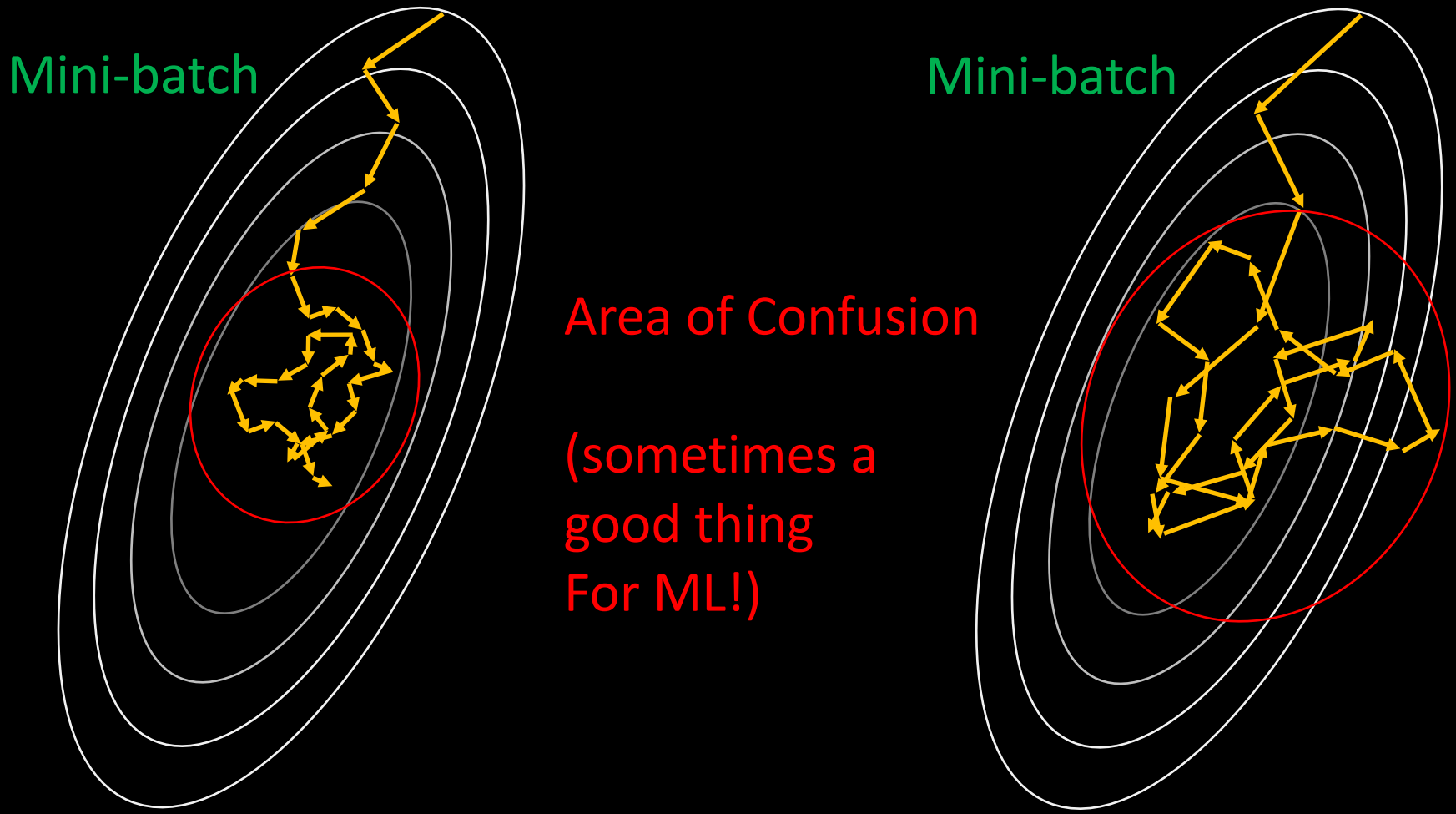

Minibatch sampling “shake up” the loss function
not stuck
stuck
L(mini_batch_1)
L(mini_batch_2)

Training Neural Networks: minibatch





Training Neural Networks: minibatch

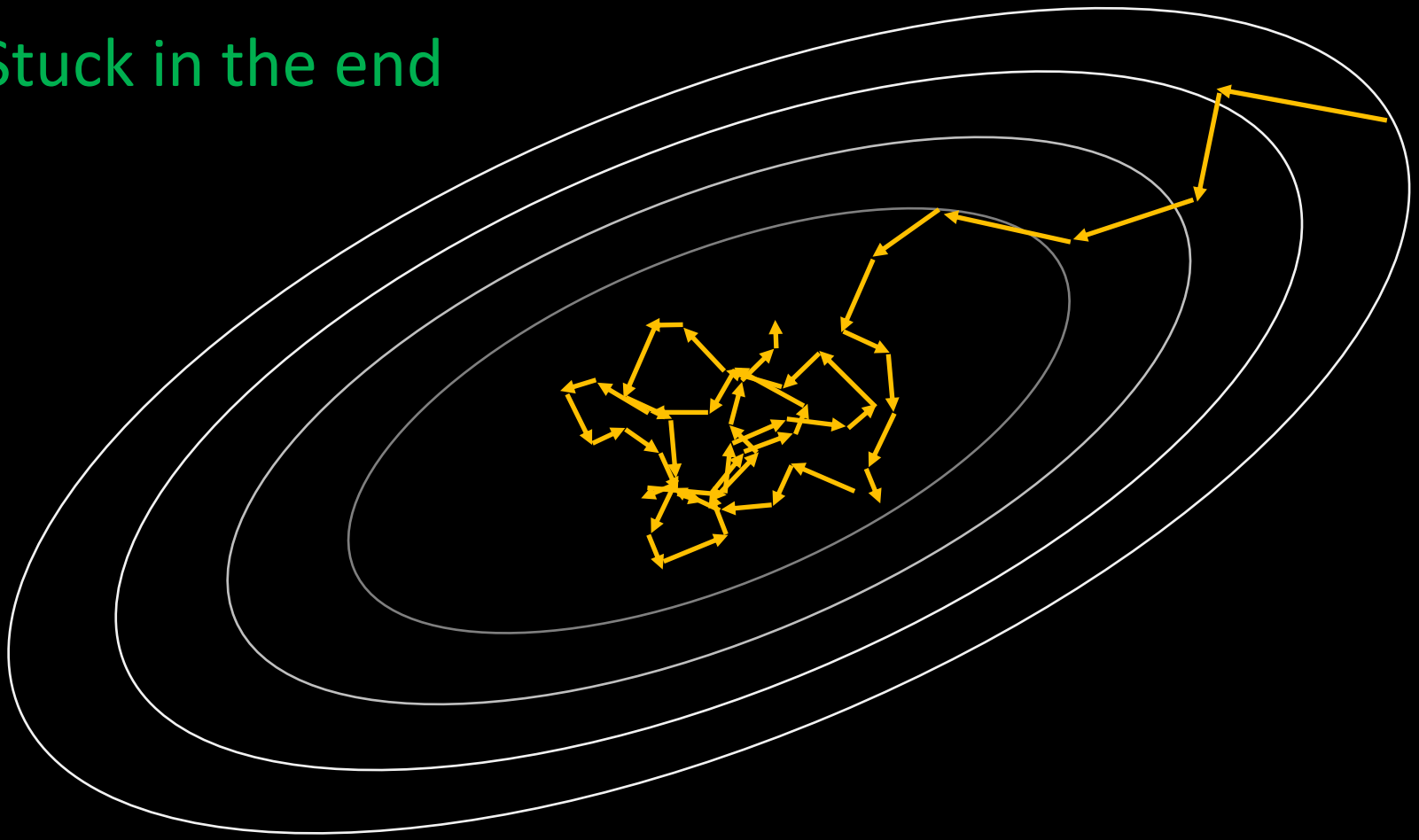


One example of modern deep learning training

	Hardware	Chips	Batch	Optimizer	BN	Accuracy	Time
Goyal et al. [6]	P100	256	8192	Momentum	Local	76.3%	1 hour
Smith et al. [16]	TPU v2	128	8192 → 16384	Momentum	Local	76.1%	30 mins.
Akiba et al. [2]	P100	1024	32768	RMS + Mom.	Local	74.9%	15 mins.
Jia et al. [10]	P40	1024	65536	LARS	Local	76.2%	8.7 mins.
Baseline	TPU v2	4	1024	Momentum	Local	76.3%	8.0 hours
Ours	TPU v2	256	16384	Momentum	Local	75.1%	10 mins.
Ours	TPU v2	256	32768	LARS	Local	76.3%	8.5 mins.
Ours	TPU v3	512	32768	LARS	Local	76.4%	3.3 mins.
Ours	TPU v3	1024	32768	LARS	Distributed	76.3%	**2.2 mins.**

Fast Progress in the Beginning

Fussy / Stuck in the end



Good for big data!

All Data

0	1	2	3	4	5	6	7	8	9

Random Shuffle

8	3	2	7	0	1	6	5	4	9

Sample without replacement…

8	3	2	7	0	1	6	5	4	9

Sample with replacement…

3	2	7	2	1	9	8	1	4	3

There are tones of research go into
how to fix crazy/stuck in the end

(not always matters in practice)

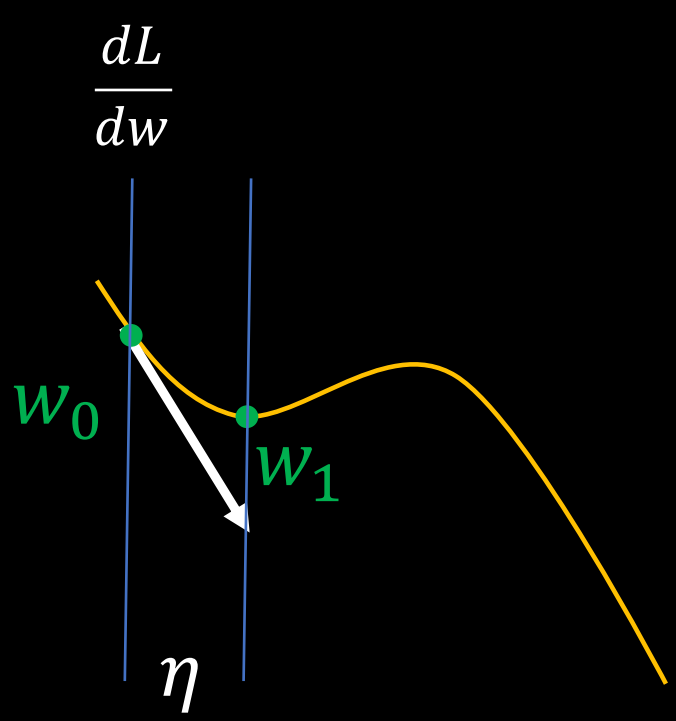
TIPS

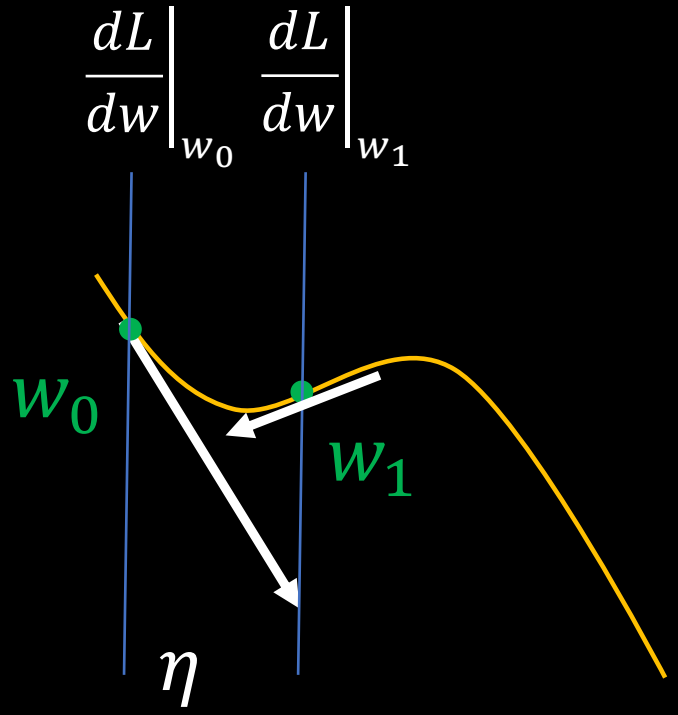
- schemes of gradient calculation

- find the right learning rate to start

- find the good way to change learning rate during training

Momentum





$$w_1 = w_0 - \frac{dL}{dw}\eta$$

$$w_1 = w_0 - \left(\left.\frac{dL}{dw}\right|_0 + \beta \left.\frac{dL}{dw}\right|_{-1}\right)\eta$$

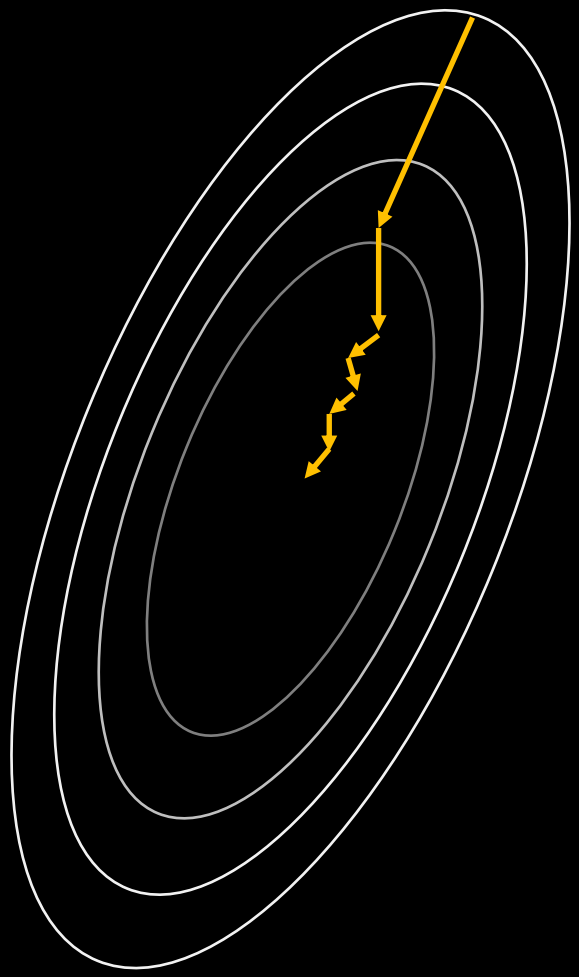
Momentum


gettyimages
Credit: Ashley Jouhar
82214305

Adagrad (adaptive)

$$w_1 = w_0 - \eta \left.\frac{dL}{dw}\right|_0 \epsilon$$

$$\epsilon = \frac{1}{\sqrt{\sum_0^t (\frac{dL}{dw})^2 + \varepsilon}}$$

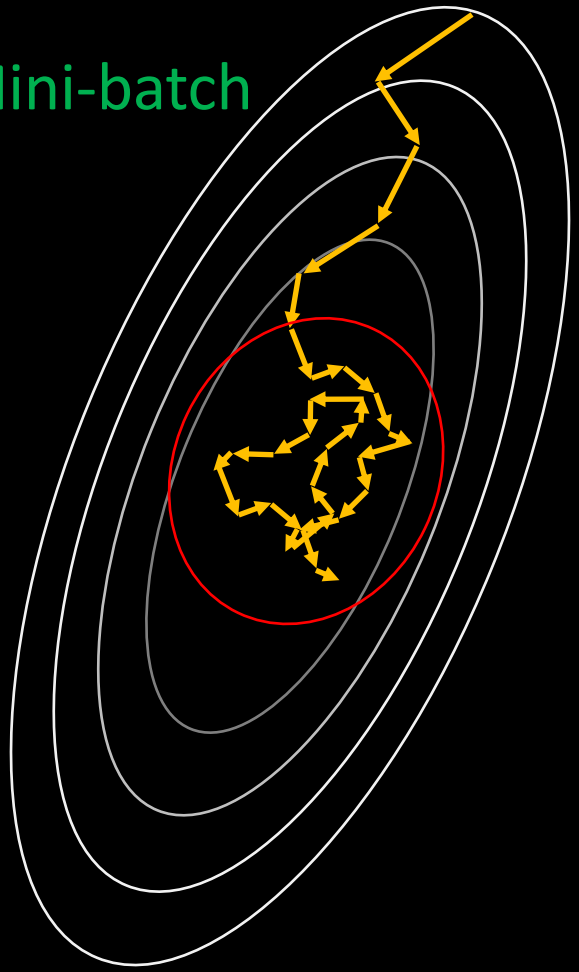


Adams

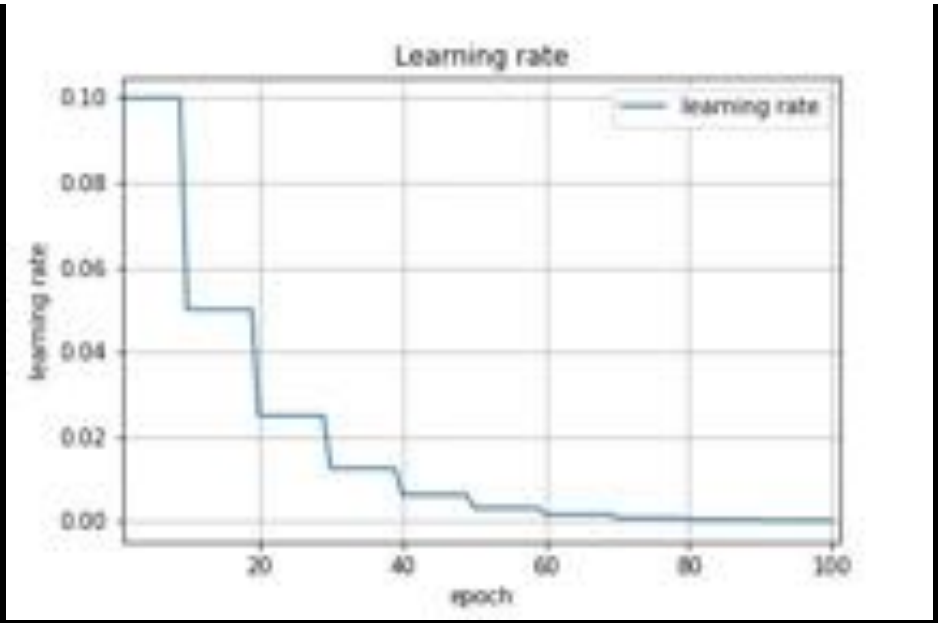
$$w_1 = w_0 - \eta \left(\left.\frac{dL}{dw}\right|_0 + \beta \left.\frac{dL}{dw}\right|_{-1}\right)\epsilon$$

Momentum Ada

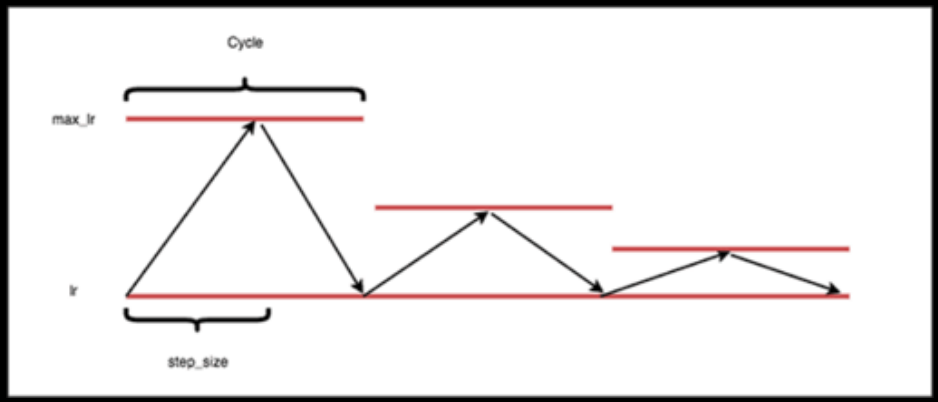
Mini-batch



Learning rate decay



Cyclic learning rate



Control Flow of updating neural network by SGD

Control Flow of SGD

Overall Loop

For epoch = 1…..n_epoch:

Train Loop

for batch = 1 ….. n_batch_training:

do training_step()
do optimize()

Validation Loop

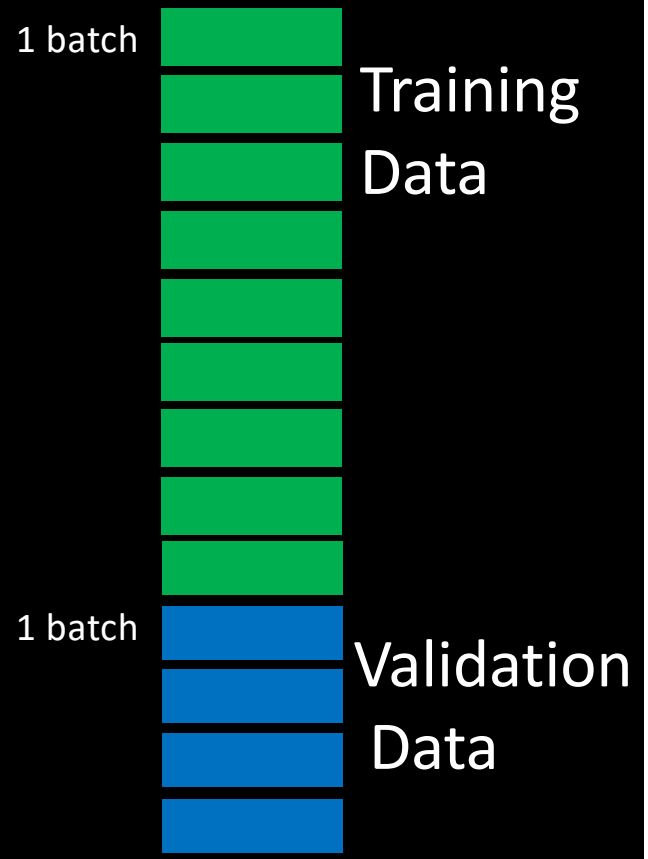
for batch = 1 ….. n_batch_validation:

do validation_step()

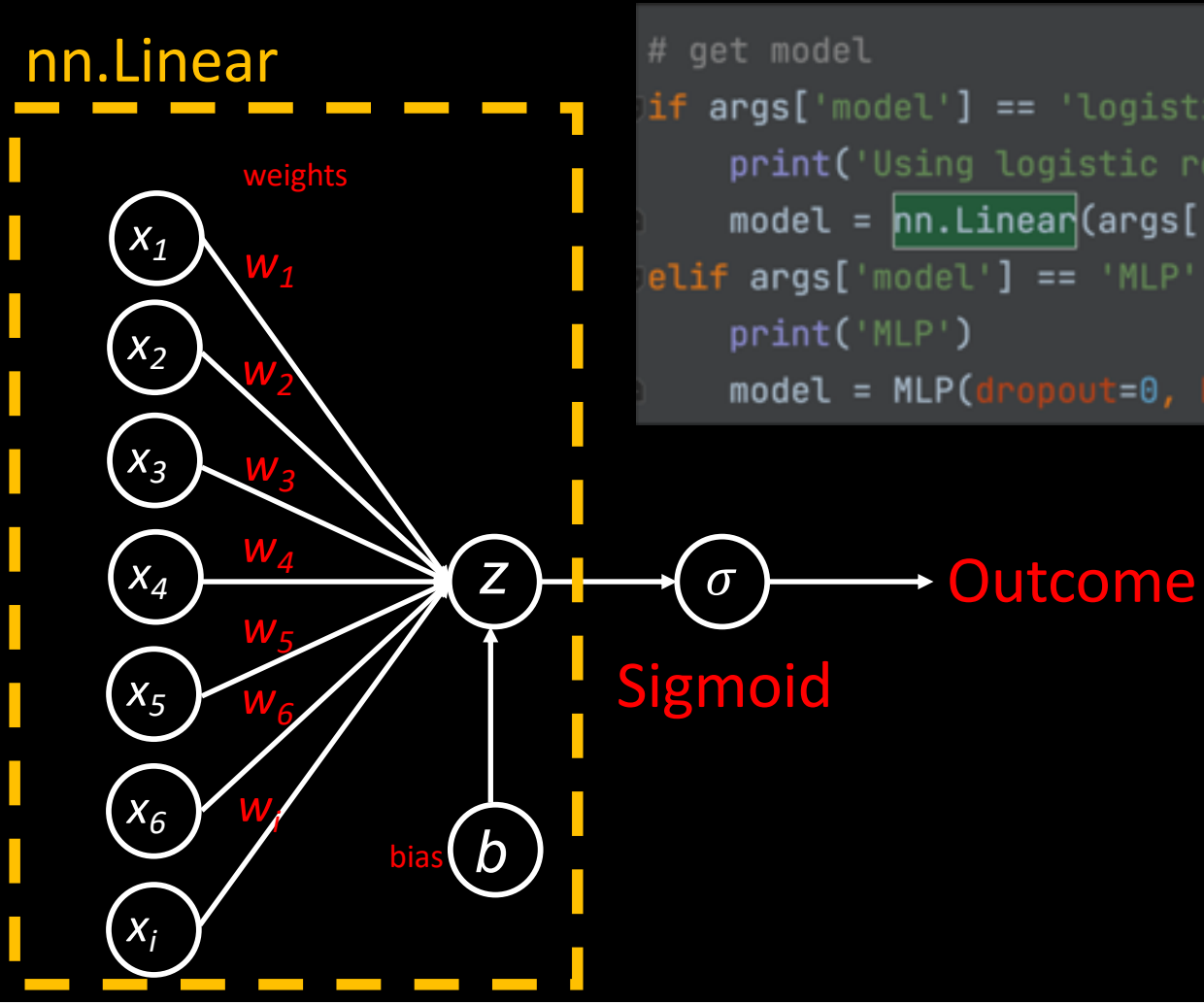
```python
# Medical MNIST dataset (images and labels)
train_loader, validation_loader = get_medical_mnist(args=args)
print('Done with data preparation')
```

```
data folder:  mmnist/
length of all images: 42000
length of all images: 16954
Done with data preparation
```

```
Length of train dataset:
42000
Length of validation dataset:
16954
Length of train dataloader:
2625
Length of validation dataloader:
1060
Why is that?
```



Graph of Logistic Regression




```python
# get model
if args['model'] == 'logistic_regression':
    print('Using logistic regression')
    model = nn.Linear(args['img_size'], args['num_classes'])
elif args['model'] == 'MLP':
    print('MLP')
    model = MLP(dropout=0, hidden_1=512, hidden_2=512)
```

```
>>> import torch
>>> import torch.nn as nn

>>> model = nn.Linear(100, 1)
>>> model

Linear(in_features=100, out_features=1, bias=True)
```

torch: basic things.

torch.nn: neural network components

a 100 features to 1 class linear model

Usually, the first dimension means the batch number (how many data were put into NN at once)

The second dimension means how many features (channels) of the data

```
>>> one_case = torch.rand(1, 100)
>>> print(one_case.shape)

torch.Size([1, 100])
```

One data with 100 features

Usually, the first dimension means the batch number (how many data were put into NN at once)

```
>>> one_case = torch.rand(1, 100)
>>> print(one_case.shape)

torch.Size([1, 100])

>>> print(model(one_case).shape)

torch.Size([1, 1])
```

One data with 100 features

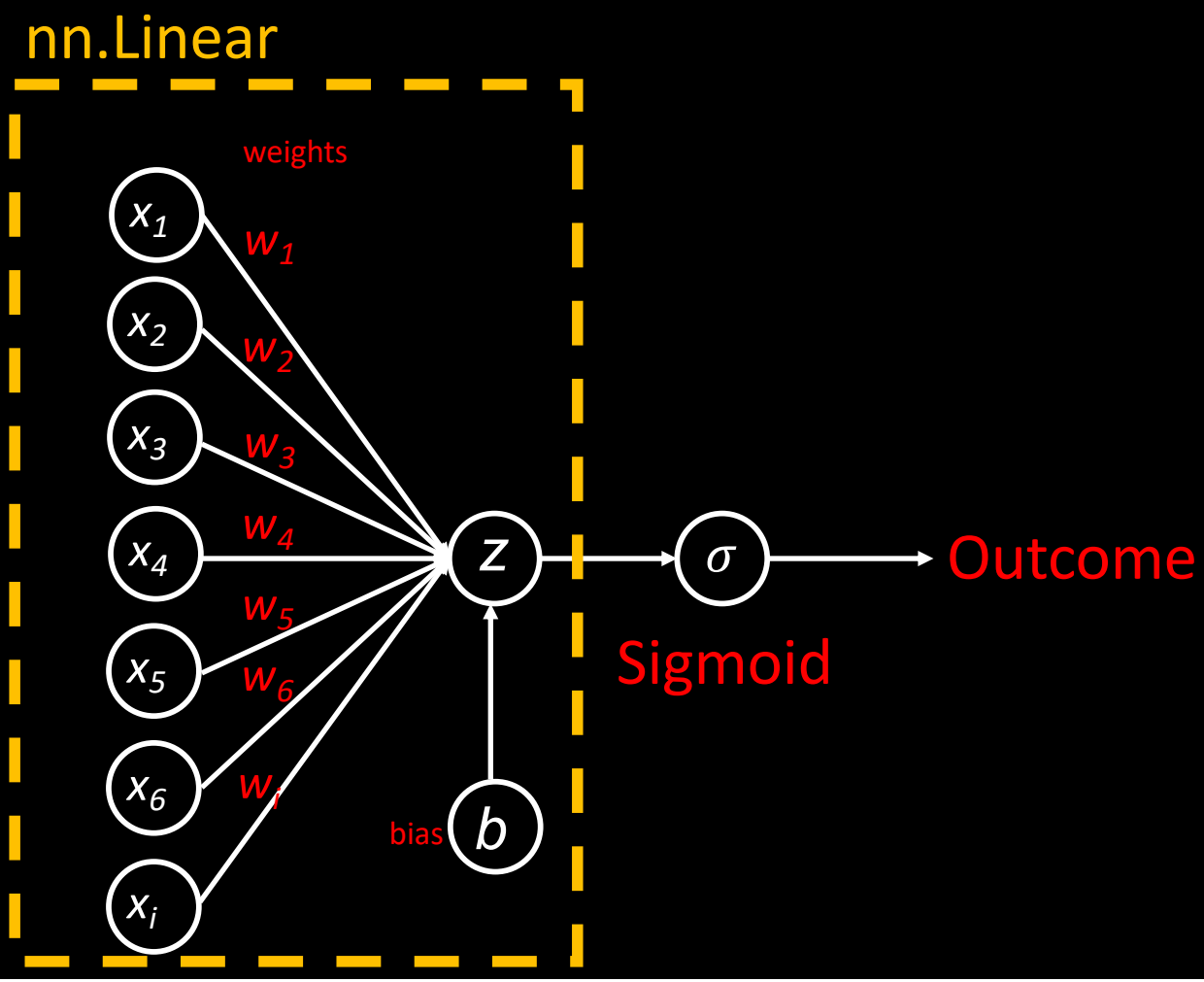
One data create one output

```
>>> three_case = torch.rand(3, 100)
>>> print(three_case.shape)

torch.Size([3, 100])

>>> print(model(three_case).shape)

torch.Size([3, 1])
```

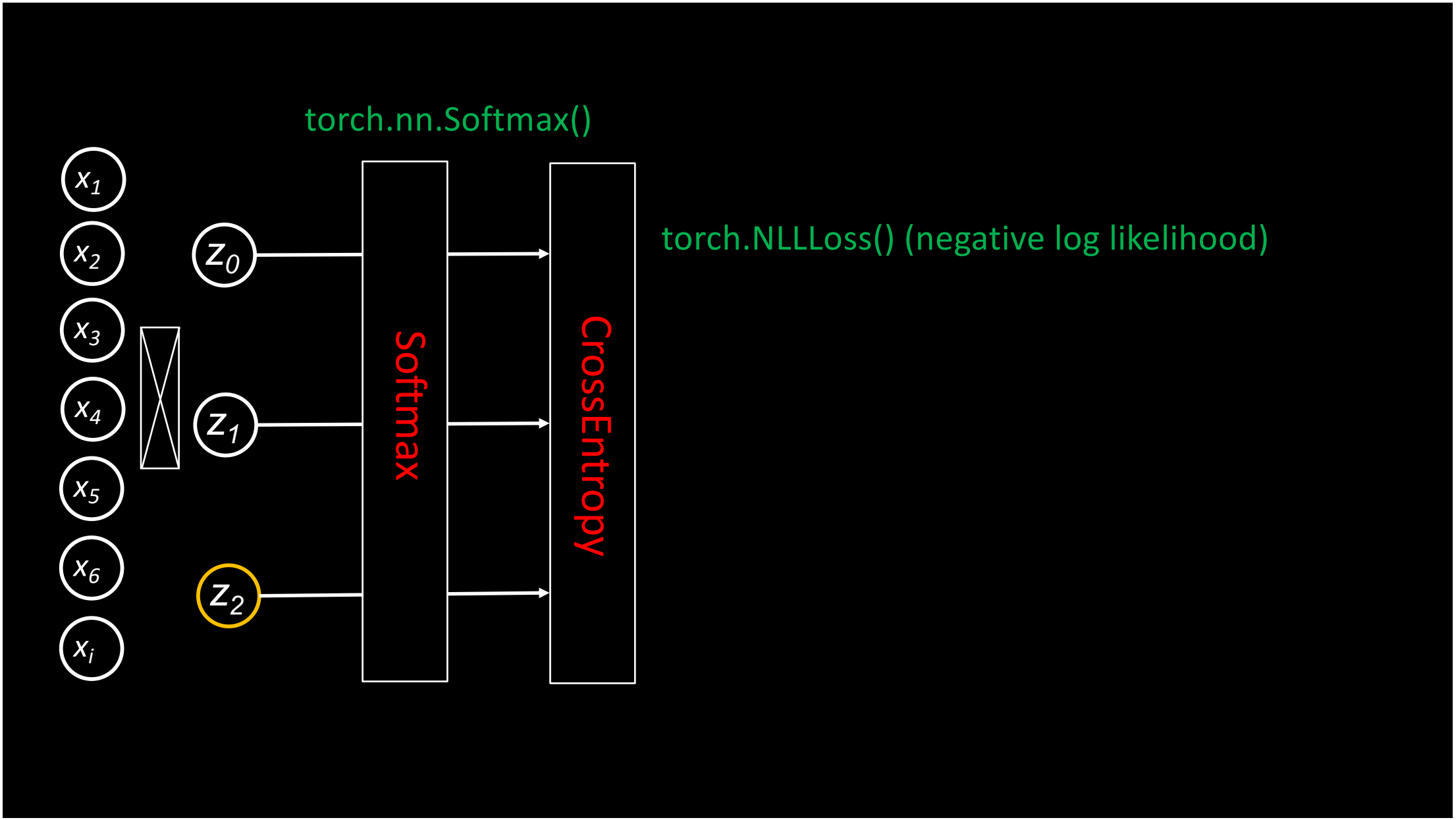
three data with 100 features

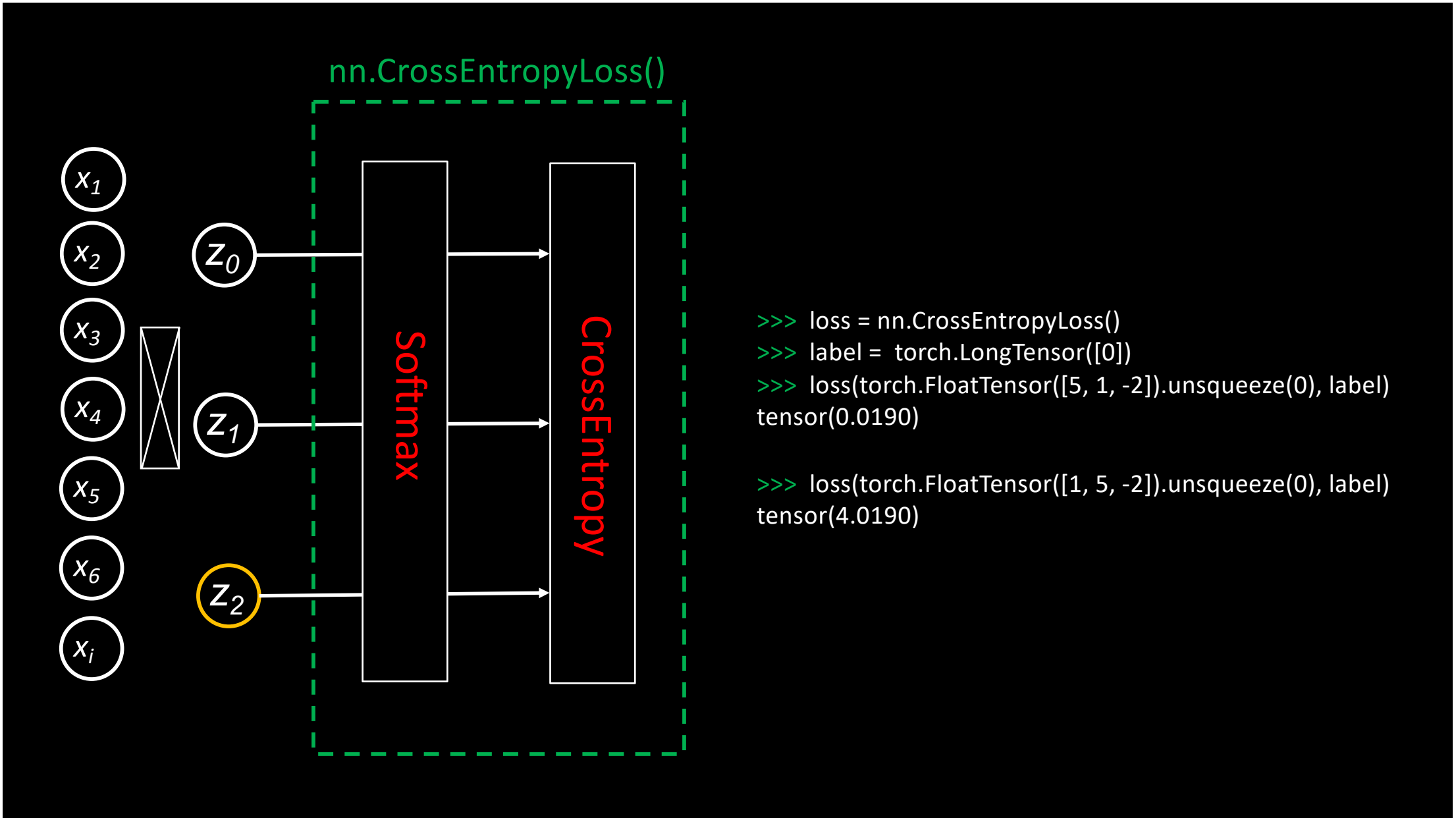
three data create three output

Graph of Logistic Regression

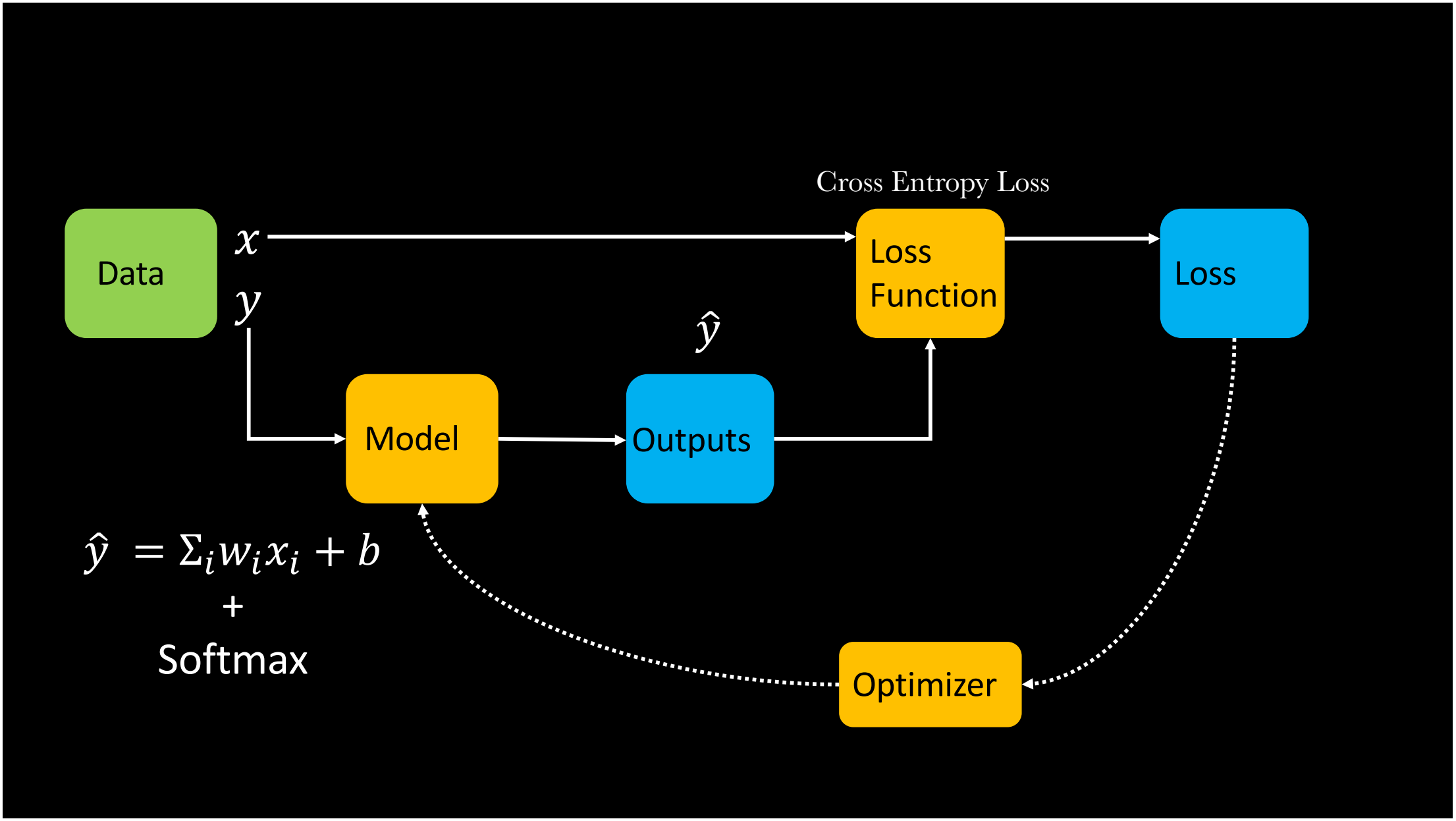


```
# Logistic regression model
model = nn.Linear(args['img_size'], args['num_classes'])
                          784 = 28 X 28        10


model
Linear(in_features=784, out_features=10, bias=True)

model.weight.shape
torch.Size([10, 784])

model.bias.shape
torch.Size([10])

model(torch.rand(3, 28*28)).shape
torch.Size([3, 10])
```


torch.nn.Softmax()
x_1
x_2
x_3
x_4
x_5
x_6
x_i
z_0
z_1
z_2
Softmax
CrossEntropy
torch.NLLLoss() (negative log likelihood)




```
>>> loss = nn.CrossEntropyLoss()
>>> label =  torch.LongTensor([0])
>>> loss(torch.FloatTensor([5, 1, -2]).unsqueeze(0), label)
tensor(0.0190)

>>> loss(torch.FloatTensor([1, 5, -2]).unsqueeze(0), label)
tensor(4.0190)
```


Cross Entropy Loss
Data
x
y
Loss Function
Loss
$\hat{y}$
Model
Outputs
$\hat{y} = \Sigma_i w_i x_i + b$
+
Softmax
Optimizer

```
# Hyper-parameters
args = {'img_size': 28 * 28,

        'num_classes': 10,

        'num_epochs': 50,                How many training steps until finished?

        'batch_size’: 16,                How many cases the model see each time?

        'learning_rate': 0.001}          How fast we are moving in training?
```

```
# MNIST dataset (images and labels)
train_dataset = torchvision.datasets.MNIST(root='../../data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='../../data',
                                          train=False,
                                          transform=transforms.ToTensor())
```

Location

Training or testing

Download if not avalible

```
len(train_dataset)
60000
x = train_dataset.__getitem__(10)
len(x)
2
x[0].shape
torch.Size([1, 28, 28])
x[1]
3
```

```
# Data loader (input pipeline)
train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,        Using training data
                            batch_size=args['batch_size'],
                             shuffle=True)

test_loader =
torch.utils.data.DataLoader(dataset=test_dataset,
                            batch_size=args['batch_size'],
                            shuffle=False)



len(train_loader)
3750
i, x=next(enumerate(train_loader))
x[0].shape
torch.Size([16, 1, 28, 28])
x[1]
tensor([7, 5, 1, 0, 8, 7, 4, 5, 4, 1, 2, 6, 1, 6, 1, 2])
```

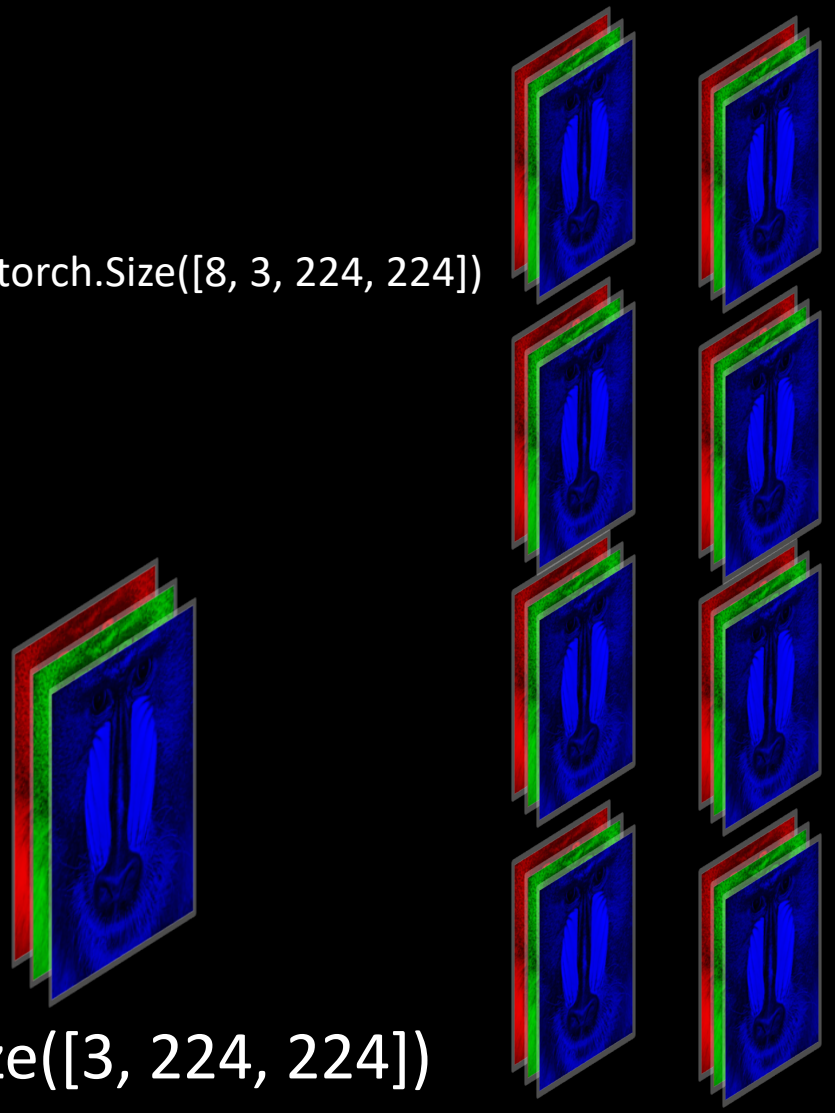
Tensor Shapes

```
>>> torch.from_numpy(np.ones((8, 3, 224, 224)).astype(np.float32)).shape
torch.Size([8, 3, 224, 224])
```

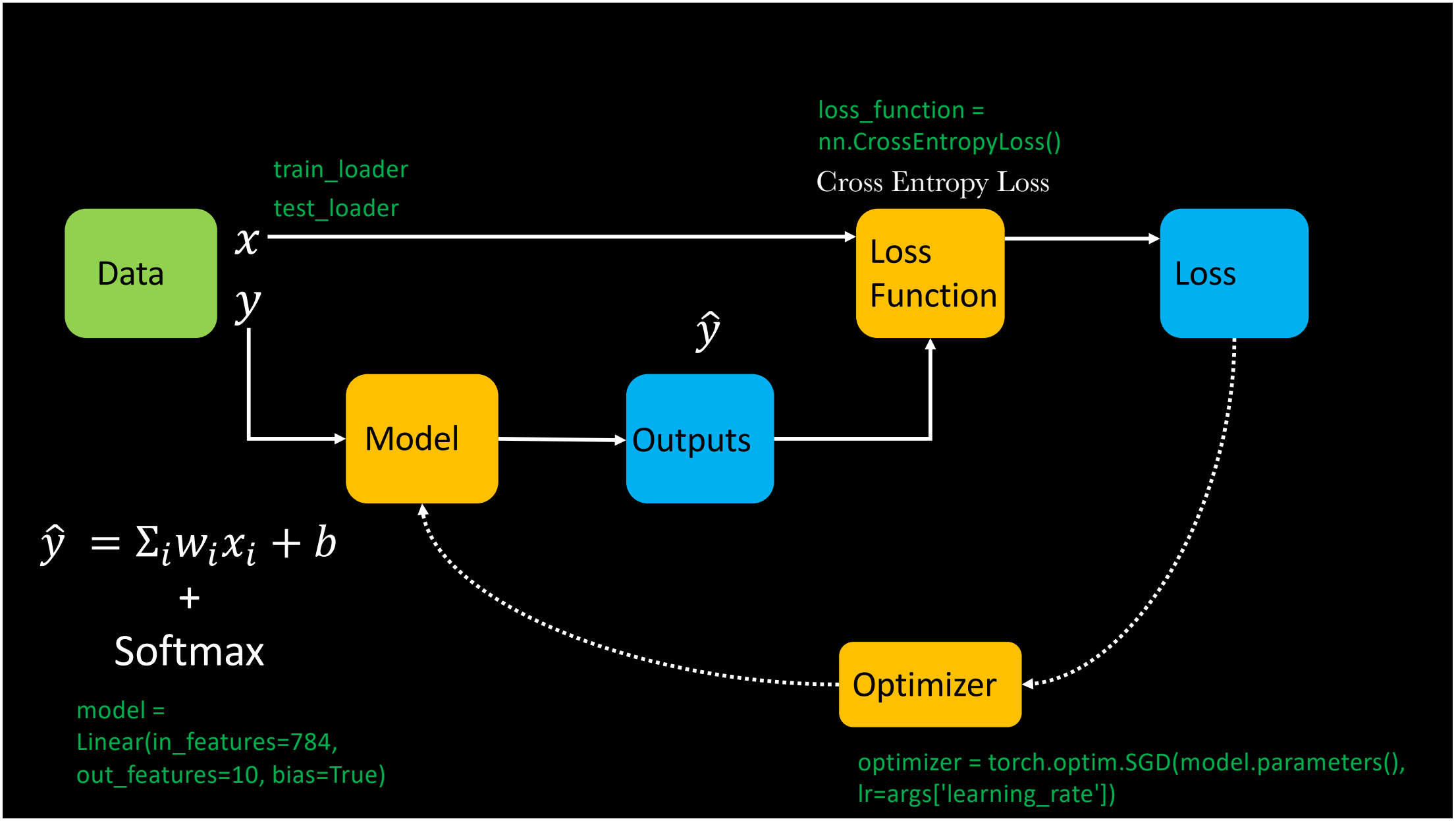
X.shape
torch.Size([8, 3, 224, 224])

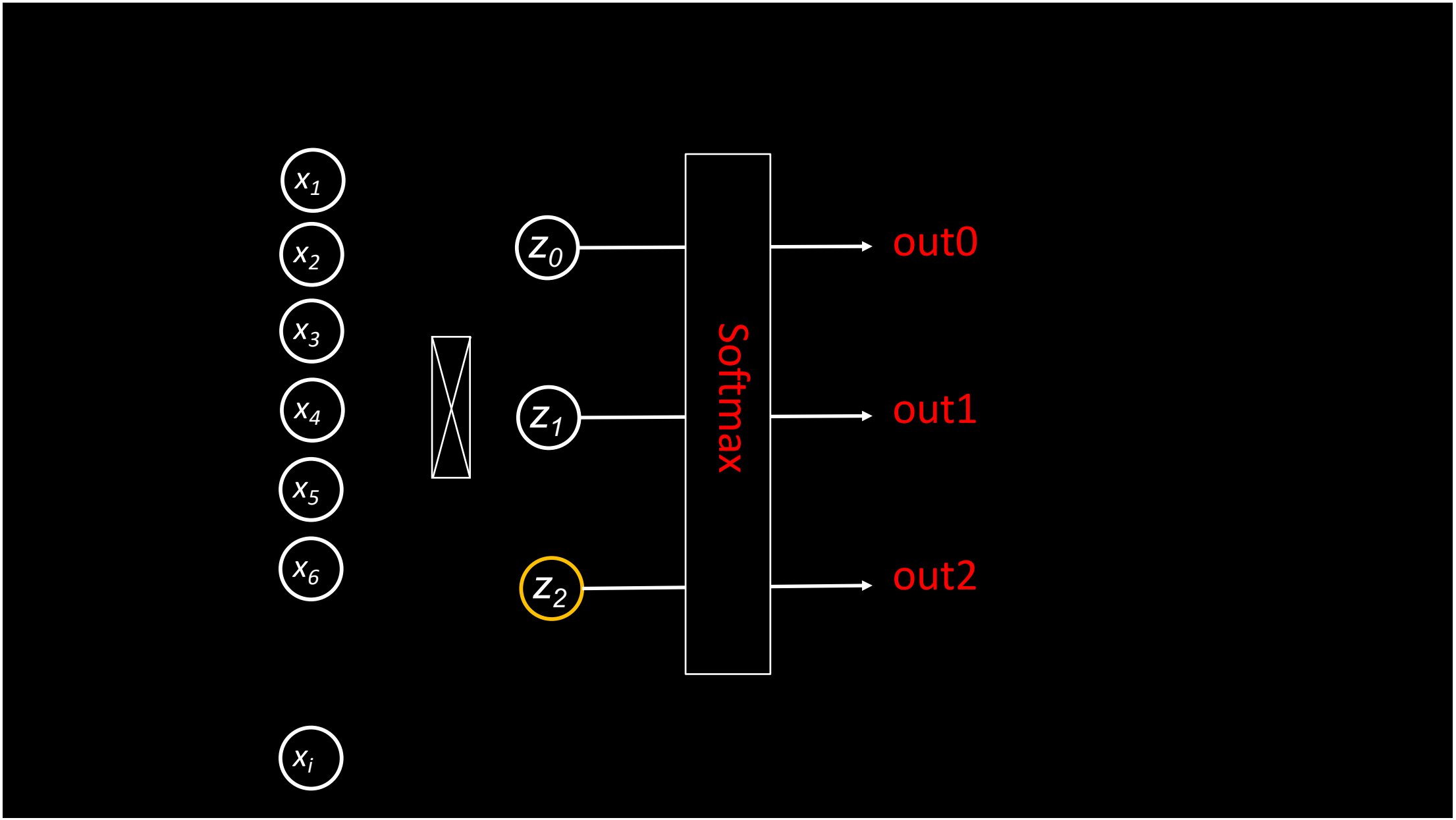
Batch size
Channel #
Height
Width

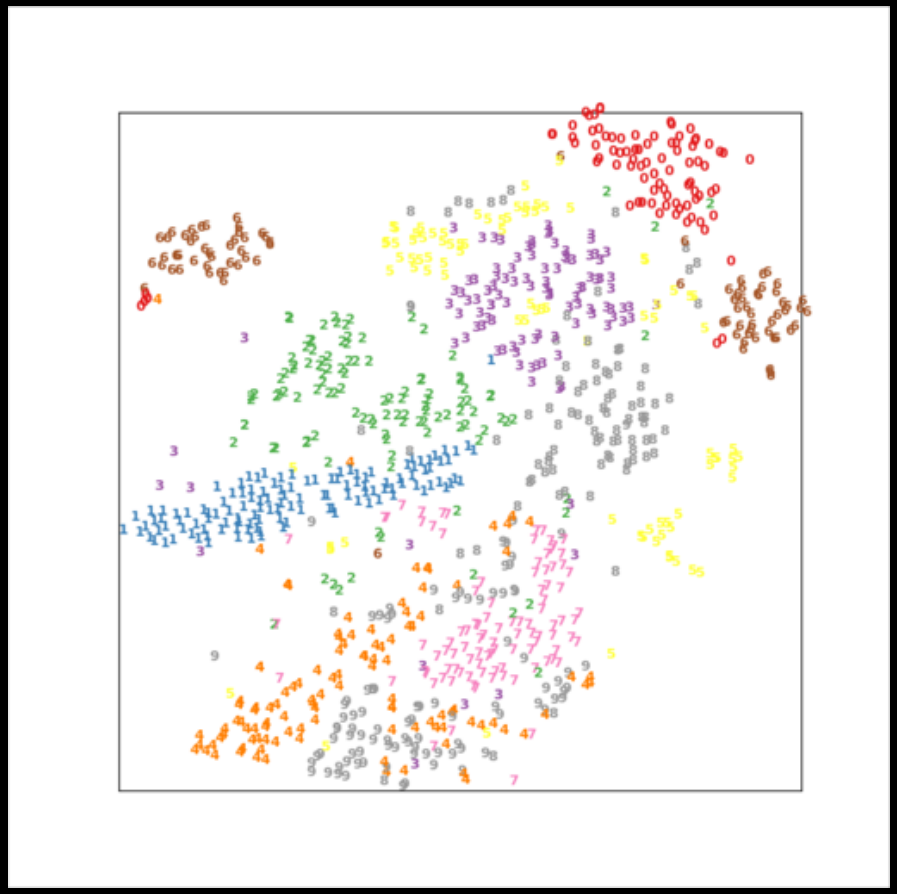
torch.Size([8, 3, 224, 224])



torch.Size([3, 224, 224])


loss_function =
nn.CrossEntropyLoss()
Cross Entropy Loss
train_loader
test_loader
Data
x
y
Loss Function
Loss
ŷ
Model
Outputs
ŷ = Σ_i w_i x_i + b
+
Softmax
Optimizer
model =
Linear(in_features=784,
out_features=10, bias=True)
optimizer = torch.optim.SGD(model.parameters(),
lr=args['learning_rate'])


x_1
x_2
x_3
x_4
x_5
x_6
x_i
z_0
z_1
z_2
Softmax
out0
out1
out2

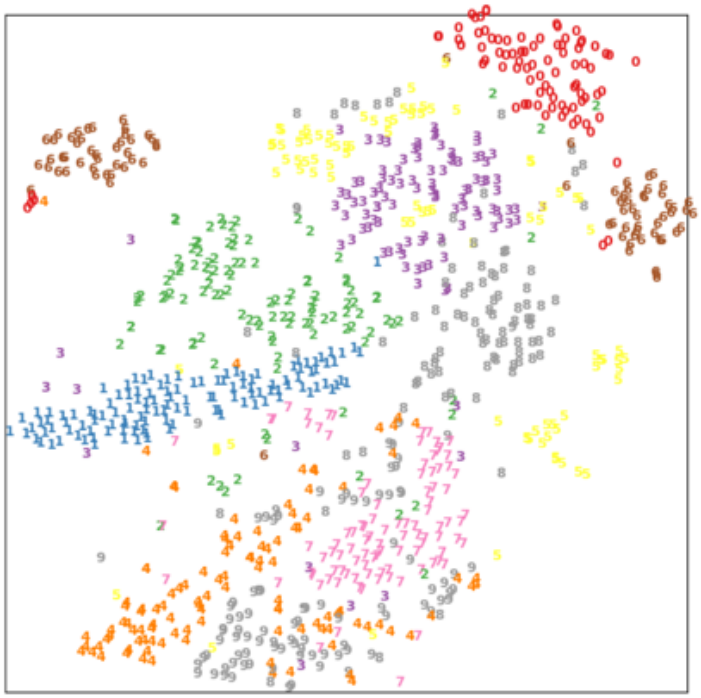


Visualize the results by tsne

784

10

Before logistic regression



After logistic regression