Machine Learning for Image Processing

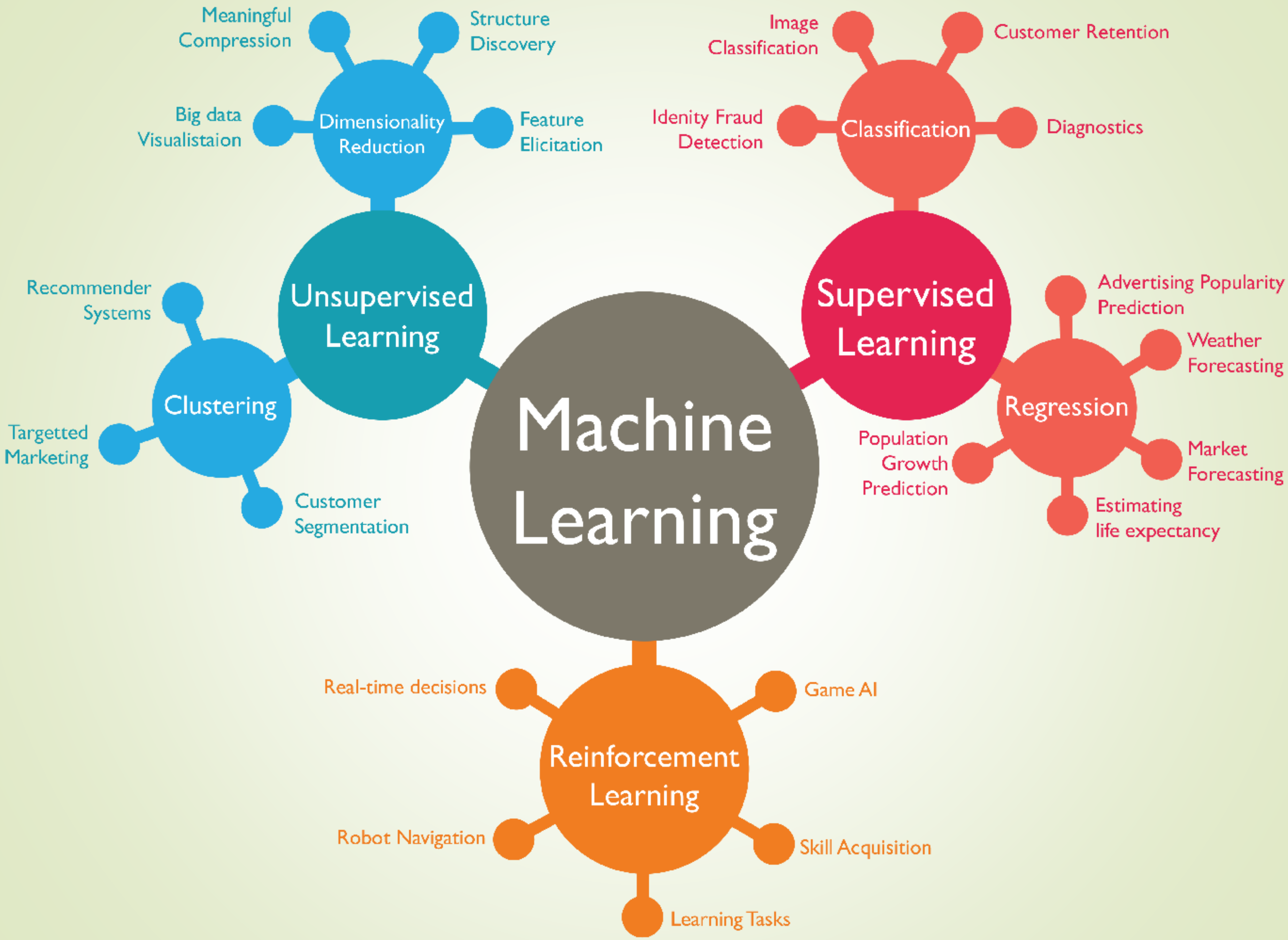
Zhaohan Xiong, Jichao Zhao

(20th June 2018)

Auckland Bioengineering Institute
The University of Auckland, New Zealand

NIH National Institutes of Health
Turning Discovery Into Health

hrc Health Research Council of New Zealand

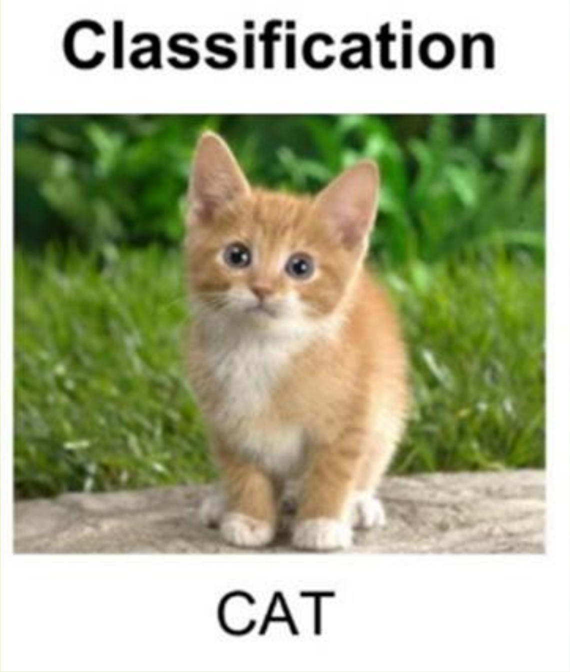


https://www.datasciencecentral.com/profiles/blogs/types-of-machine-learning-algorithms-in-one-picture

Supervised Learning

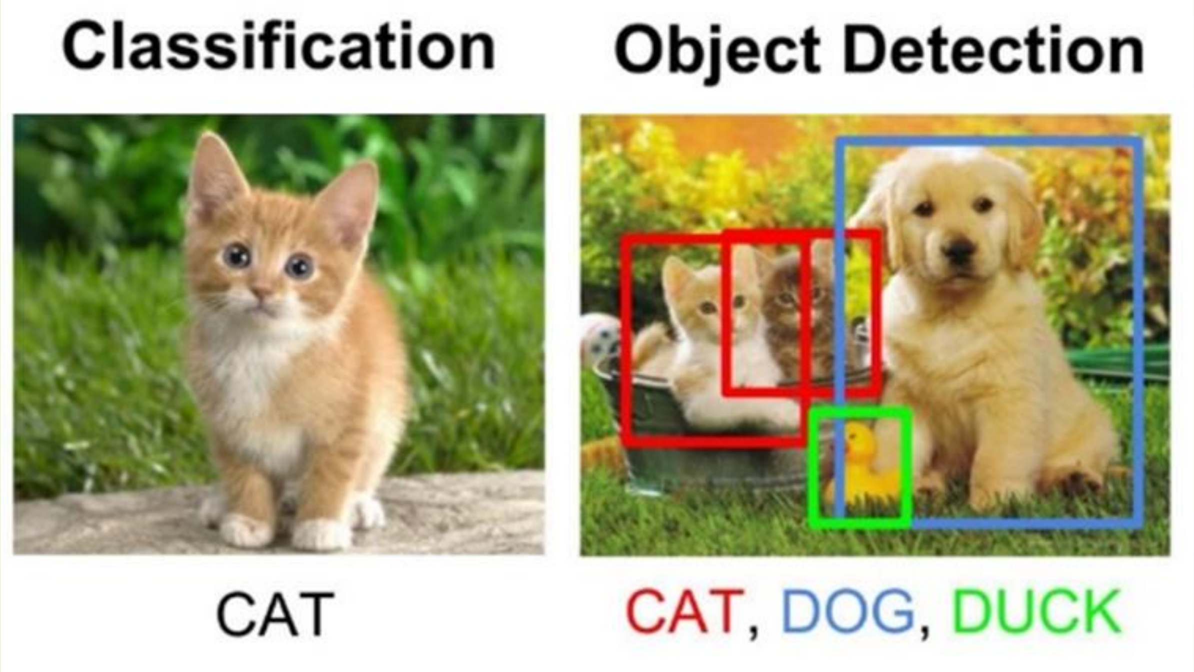
- Given a dataset with known labels.
- “Train” a machine learning model using this dataset.
- Use the trained model to make predictions for new data which we do not know the labels for.

Types of Image Recognition Tasks



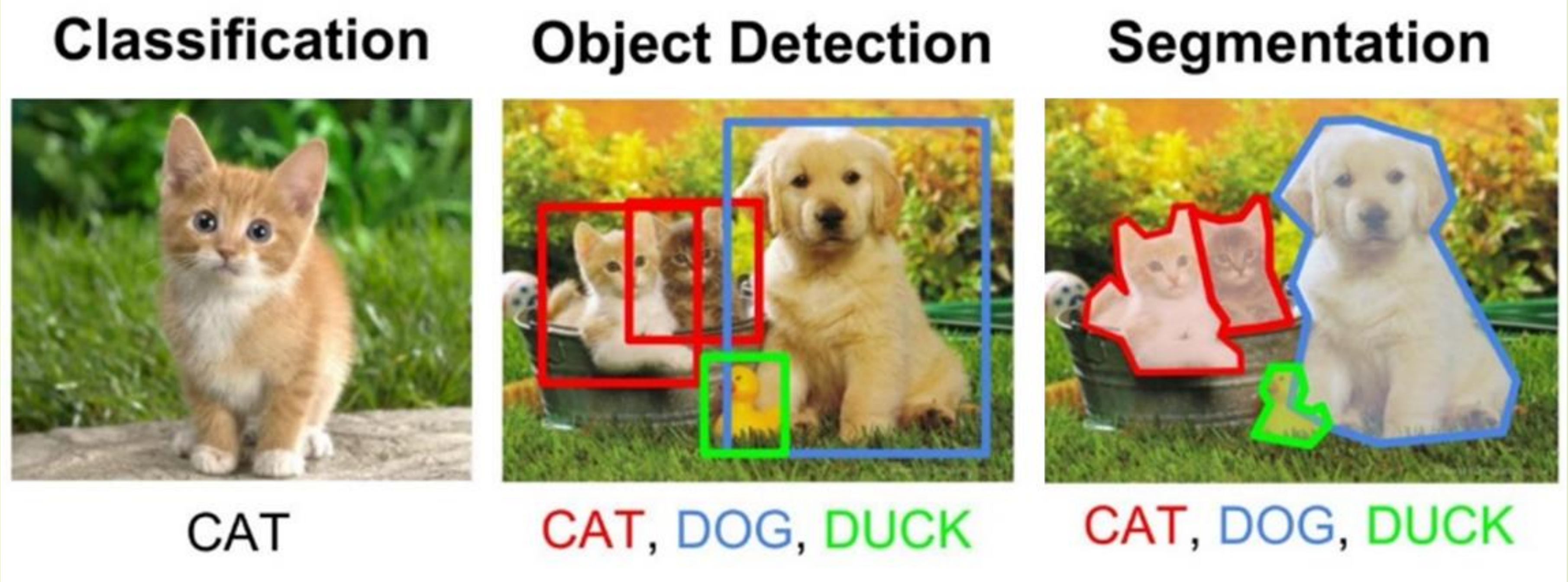
https://medium.com/comet-app/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852

Types of Image Recognition Tasks



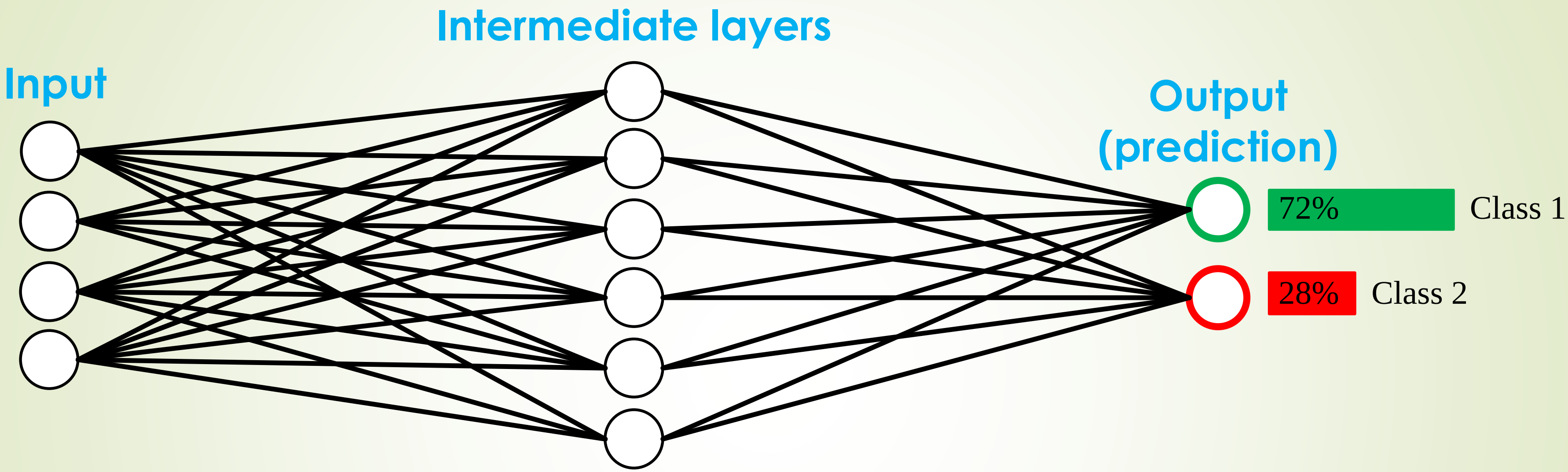
https://medium.com/comet-app/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852

Types of Image Recognition Tasks



https://medium.com/comet-app/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852

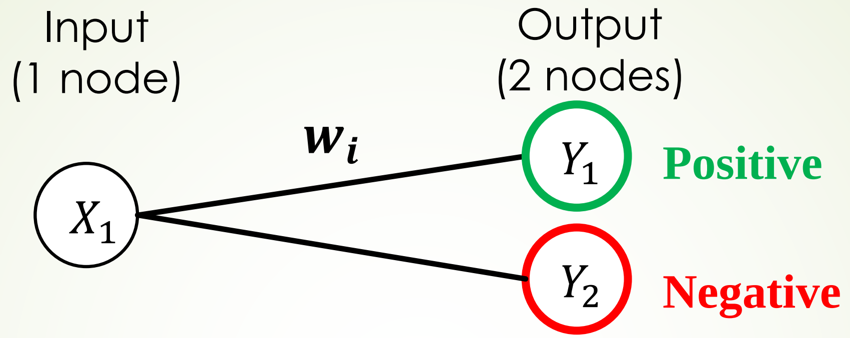
Introducing Neural Networks



- Given some **input** data,
- **Transform** the data through **intermediate** layers (number can be $N \geq 0$ layers) (we can also have as many nodes as we want for the intermediate layers),
- So the **output** becomes the probability of being in each class

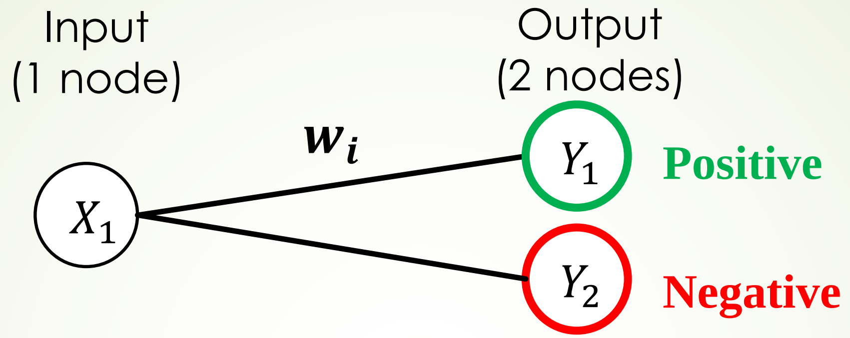
How is the “**Transformation**” Actually Done?

Starting with a (very) simple neural network:



How is the "**Transformation**" Actually Done?

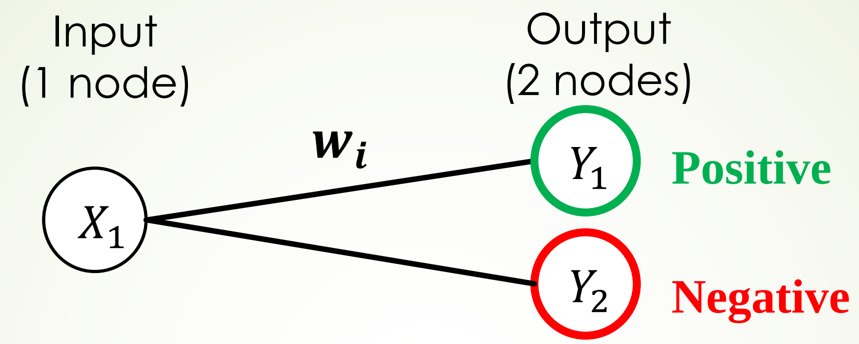
Starting with a (very) simple neural network:



TASK: given a single number at the input (X_1), predict if its **positive** or **negative** (at the output, Y).

How is the "**Transformation**" Actually Done?

Starting with a (very) simple neural network:

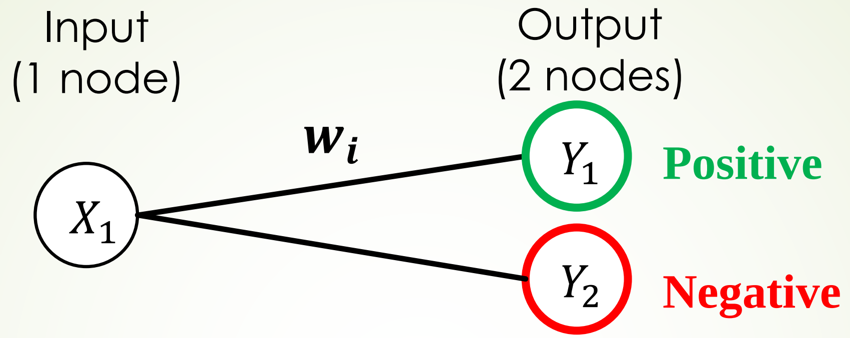


TASK: given a single number at the input (*X_1*), predict if its **positive** or **negative** (at the output, *Y*).

Ideally:
If $x \geq 0$, we would want $[Y_1 \quad Y_2] = [1 \quad 0]$ (or any $Y_1 > Y_2$)
If $x < 0$, we would want $[Y_1 \quad Y_2] = [0 \quad 1]$ (or any $Y_1 < Y_2$)

How is the "**Transformation**" Actually Done?

Starting with a (very) simple neural network:



TASK: given a single number at the input (X_1), predict if its **positive** or **negative** (at the output, Y).

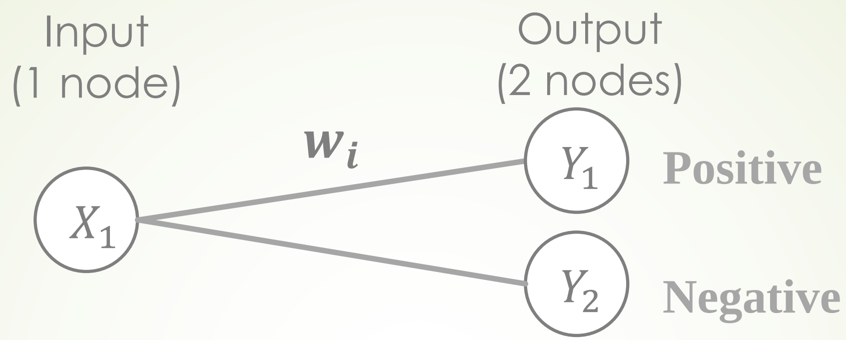
Ideally:
If $x \geq 0$, we would want $[Y_1 \quad Y_2] = [1 \quad 0]$ (or any $Y_1 > Y_2$)
If $x < 0$, we would want $[Y_1 \quad Y_2] = [0 \quad 1]$ (or any $Y_1 < Y_2$)

To transform the input (1 node) to the output (2 nodes), we can perform a matrix multiplication:

$$[X_1][w_1 \quad w_2] = [Y_1 \quad Y_2]$$

How is the "**Transformation**" Actually Done?

Starting with a (very) simple neural network:



TASK: given a single number at the input (X_1), predict if its **positive** or **negative** (at the output, Y).

Ideally:
If $x \geq 0$, we would want $[Y_1 \quad Y_2] = [1 \quad 0]$ (or any $Y_1 > Y_2$)
If $x < 0$, we would want $[Y_1 \quad Y_2] = [0 \quad 1]$ (or any $Y_1 < Y_2$)

To transform the input (1 node) to the output (2 nodes), we can perform a matrix multiplication:

$$[X_1][\boldsymbol{w_1} \quad \boldsymbol{w_2}] = [Y_1 \quad Y_2]$$

*How do we choose which **w**'s get our desired **Y**'s ?*

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [Y_1 \quad Y_2]$$

- Set up an optimization problem to solve the values of ***w***'s.

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [Y_1 \quad Y_2]$$

- Set up an optimization problem to solve the values of ***w***’s.
- Train the neural network with lots of ***X***’s and ***Y***’s so it can incrementally update ***w***’s with gradient descent each iteration.

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [Y_1 \quad Y_2]$$

- Set up an optimization problem to solve the values of ***w***'s.
- Train the neural network with lots of ***X***'s and ***Y***'s so it can incrementally update ***w***'s with gradient descent each iteration.

Sample Training Set:

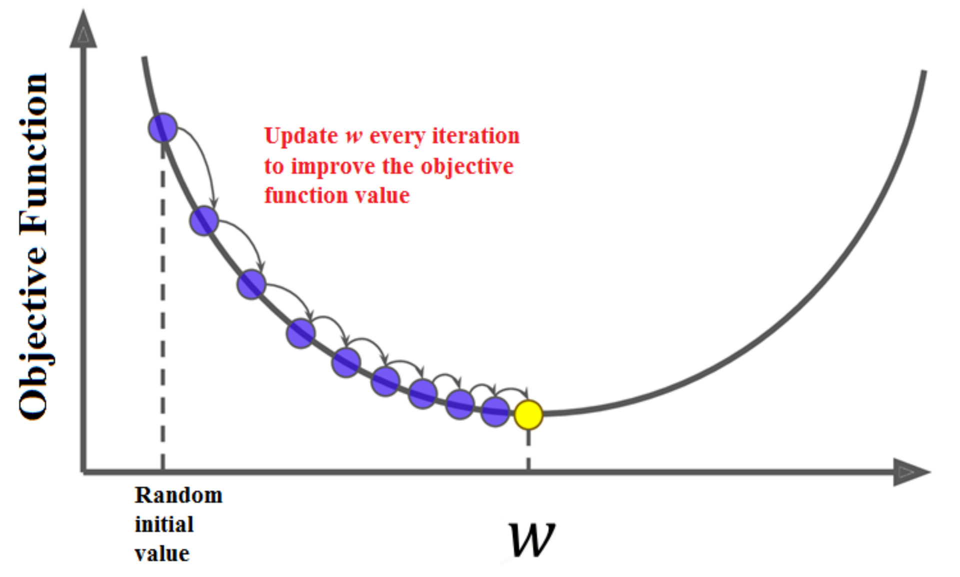
X:	$[3]$	$[-2]$	$[1]$	$[5]$	$[-1]$	$[-3]$	$[-2]$	$[2]$	$[1]$
Y:	$[1 \quad 0]$	$[0 \quad 1]$	$[1 \quad 0]$	$[1 \quad 0]$	$[0 \quad 1]$	$[0 \quad 1]$	$[0 \quad 1]$	$[1 \quad 0]$	$[1 \quad 0]$

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [Y_1 \quad Y_2]$$

- Set up an optimization problem to solve the values of ***w***'s.
- Train the neural network with lots of ***X***'s and ***Y***'s so it can incrementally update ***w***'s with gradient descent each iteration.

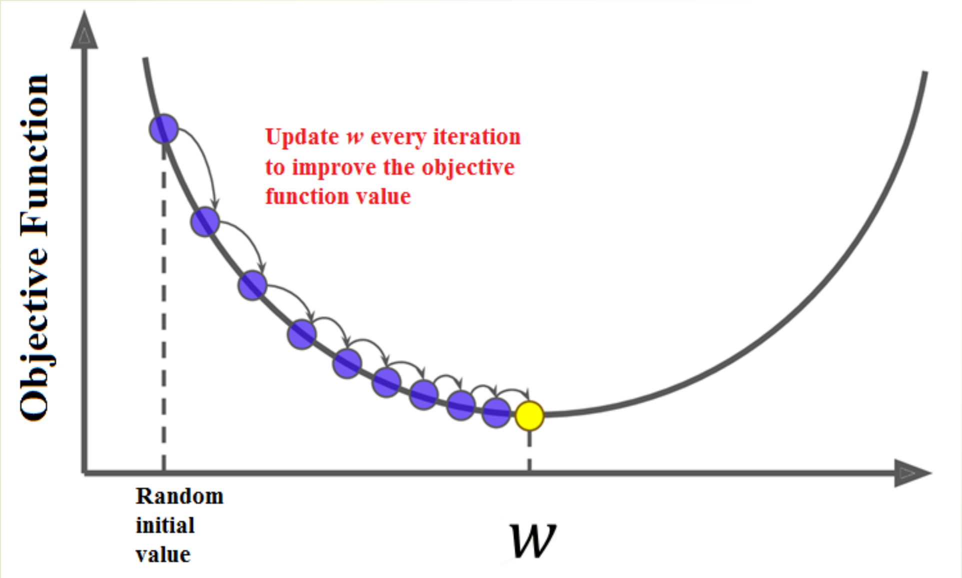


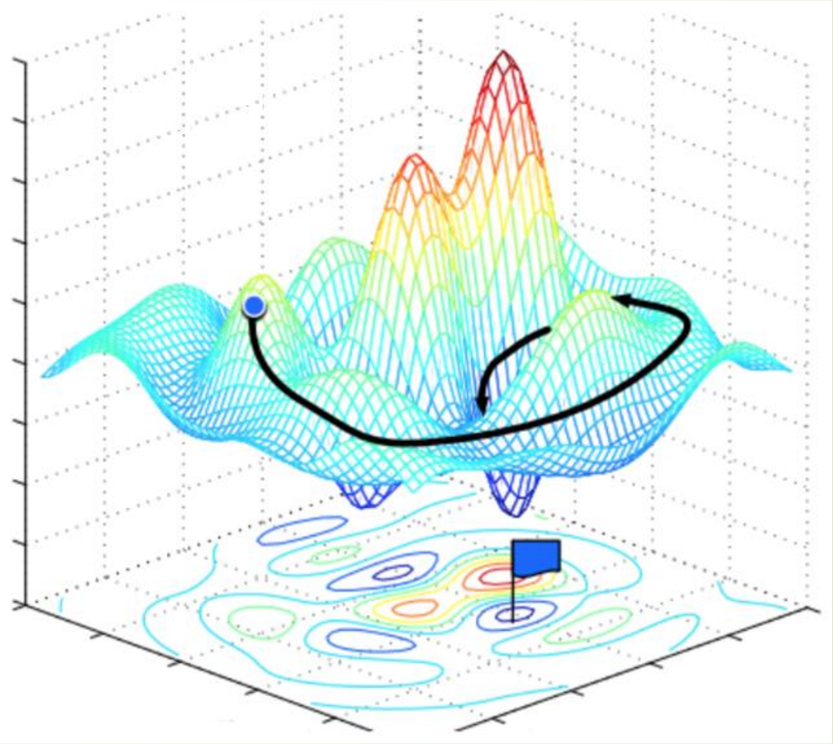
Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [Y_1 \quad Y_2]$$

- Set up an optimization problem to solve the values of ***w***'s.
- Train the neural network with lots of ***X***'s and ***Y***'s so it can incrementally update ***w***'s with gradient descent each iteration.





Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [Y_1 \quad Y_2]$$

- Set up an optimization problem to solve the values of ***w***'s.
- Train the neural network with lots of ***X***'s and ***Y***'s so it can incrementally update ***w***'s with gradient descent each iteration.

This simple problem has many solutions for ***w***, e.g. $[\boldsymbol{w_1} \quad \boldsymbol{w_2}] = [\mathbf{0.6} \quad \mathbf{0.4}]$(or any $w_1 > w_2$).

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [\boldsymbol{Y_1} \quad \boldsymbol{Y_2}]$$

- Set up an optimization problem to solve the values of $\boldsymbol{w}$'s.
- Train the neural network with lots of $\boldsymbol{X}$'s and $\boldsymbol{Y}$'s so it can incrementally update $\boldsymbol{w}$'s with gradient descent each iteration.

This simple problem has many solutions for $\boldsymbol{w}$, e.g. $[\boldsymbol{w_1} \quad \boldsymbol{w_2}] = [\mathbf{0.6} \quad \mathbf{0.4}]$(or any $w_1 > w_2$).

If we input $\boldsymbol{X} = \mathbf{1}$ and $\boldsymbol{X} = -\mathbf{1}$:

$[1][\mathbf{0.6} \quad \mathbf{0.4}] = [\mathbf{0.6} \quad \mathbf{0.4}]$ ($Y_1 > Y_2$, higher value for **positive**)

$[-1][\mathbf{0.6} \quad \mathbf{0.4}] = [-\mathbf{0.6} \quad -\mathbf{0.4}]$ ($Y_1 < Y_2$, higher value for **negative**)

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [\boldsymbol{Y_1} \quad \boldsymbol{Y_2}]$$

- Set up an optimization problem to solve the values of $\boldsymbol{w}$'s.
- Train the neural network with lots of $\boldsymbol{X}$'s and $\boldsymbol{Y}$'s so it can incrementally update $\boldsymbol{w}$'s with gradient descent each iteration.

This simple problem has many solutions for $\boldsymbol{w}$, e.g. $[w_1 \quad w_2] = [0.6 \quad 0.4]$(or any $w_1 > w_2$).

If we input $\boldsymbol{X} = \mathbf{1}$:
$[1][\mathbf{0.6} \quad \mathbf{0.4}] = [\mathbf{0.6} \quad \mathbf{0.4}]$

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [\boldsymbol{Y_1} \quad \boldsymbol{Y_2}]$$

- Set up an optimization problem to solve the values of ***w***'s.
- Train the neural network with lots of ***X***'s and ***Y***'s so it can incrementally update ***w***'s with gradient descent each iteration.

This simple problem has many solutions for ***w***, e.g. $[w_1 \quad w_2] = [0.6 \quad 0.4]$(or any $w_1 > w_2$).

If we input $\boldsymbol{X} = \mathbf{1}$:

$$[1][\mathbf{0.6} \quad \mathbf{0.4}] = [\mathbf{0.6} \quad \mathbf{0.4}]$$

normalize: $\left[\frac{e^{0.6}}{e^{0.6}+e^{0.4}} \quad \frac{e^{0.4}}{e^{0.6}+e^{0.4}}\right] = [\mathbf{0.550} \quad \mathbf{0.450}]$

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [\boldsymbol{Y_1} \quad \boldsymbol{Y_2}]$$

- Set up an optimization problem to solve the values of ***w***'s.
- Train the neural network with lots of ***X***'s and ***Y***'s so it can incrementally update ***w***'s with gradient descent each iteration.

This simple problem has many solutions for ***w***, e.g. $[w_1 \quad w_2] = [0.6 \quad 0.4]$(or any $w_1 > w_2$).

If we input $\boldsymbol{X} = \mathbf{1}$:

$[1][\mathbf{0.6} \quad \mathbf{0.4}] = [\mathbf{0.6} \quad \mathbf{0.4}]$

normalize: $\left[\frac{e^{0.6}}{e^{0.6}+e^{0.4}} \quad \frac{e^{0.4}}{e^{0.6}+e^{0.4}}\right] = [\mathbf{0.550} \quad \mathbf{0.450}]$ ***(Softmax)***

Solving an Optimization Problem

Equation from last slide:

$$[X_1][w_1 \quad w_2] = [\boldsymbol{Y_1} \quad \boldsymbol{Y_2}]$$

- Set up an optimization problem to solve the values of $\boldsymbol{w}$'s.
- Train the neural network with lots of $\boldsymbol{X}$'s and $\boldsymbol{Y}$'s so it can incrementally update $\boldsymbol{w}$'s with gradient descent each iteration.

This simple problem has many solutions for $\boldsymbol{w}$, e.g. $[w_1 \quad w_2] = [0.6 \quad 0.4]$(or any $w_1 > w_2$).

If we input $\boldsymbol{X} = \mathbf{1}$ and $\boldsymbol{X} = -\mathbf{1}$:

$[1][\mathbf{0.6} \quad \mathbf{0.4}] = [\mathbf{0.6} \quad \mathbf{0.4}]$

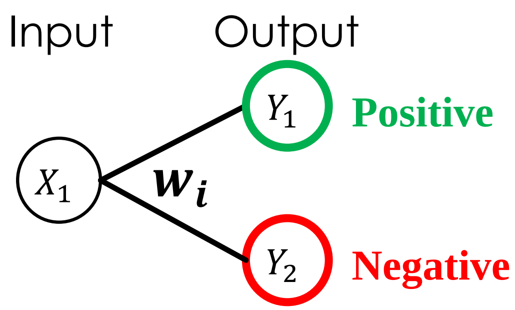
normalize: $\left[\frac{e^{0.6}}{e^{0.6}+e^{0.4}} \quad \frac{e^{0.4}}{e^{0.6}+e^{0.4}}\right] = [\mathbf{0.550} \quad \mathbf{0.450}]$

$[-1][\mathbf{0.6} \quad \mathbf{0.4}] = [-\mathbf{0.6} \quad -\mathbf{0.4}]$

normalize: $\left[\frac{e^{-0.6}}{e^{-0.6}+e^{-0.4}} \quad \frac{e^{-0.4}}{e^{-0.6}+e^{-0.4}}\right] = [\mathbf{0.450} \quad \mathbf{0.550}]$

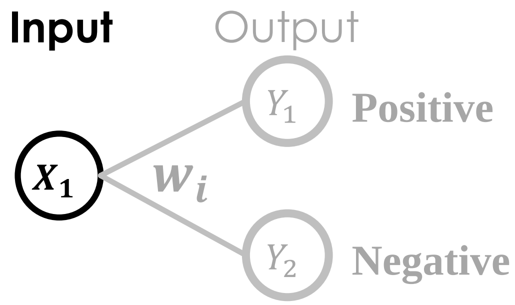
Implementing Previous Example

```
# import the packages we will be needing
import tflearn
```



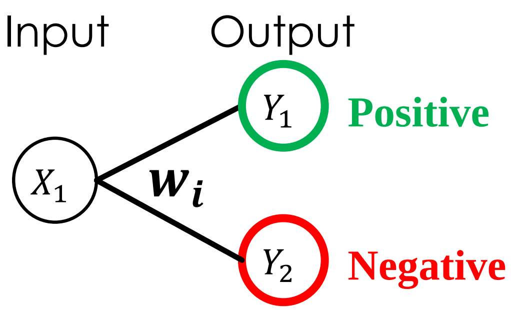
Implementing Previous Example

```
# import the packages we will be needing
import tflearn

# define the input layer (1 node)
X_i = tflearn.input_data(shape=[None, 1])
```



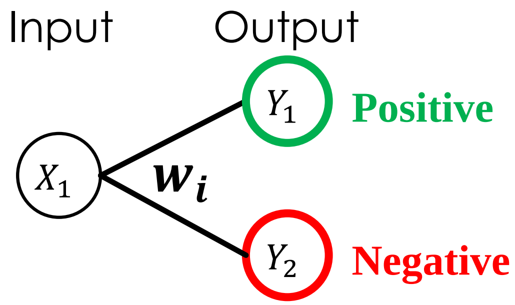
Implementing Previous Example

```
# import the packages we will be needing
import tflearn

# define the input layer (1 node)
X_i = tflearn.input_data(shape=[None, 1])

# define the output layer (2 nodes), softmax normalizes values between 0/1
Y_j = tflearn.fully_connected(X_i, n_units=2, activation='softmax')
```



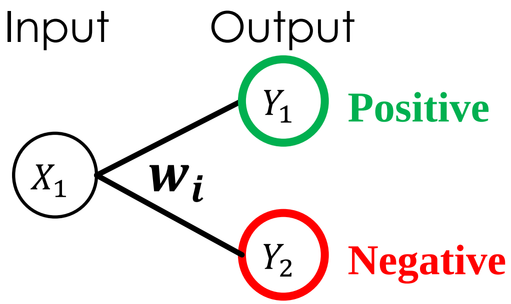
Implementing Previous Example

```
# import the packages we will be needing
import tflearn

# define the input layer (1 node)
X_i = tflearn.input_data(shape=[None, 1])

# define the output layer (2 nodes), softmax normalizes values between 0/1
Y_j = tflearn.fully_connected(X_i, n_units=2, activation='softmax')
```



We don't need to define w_i

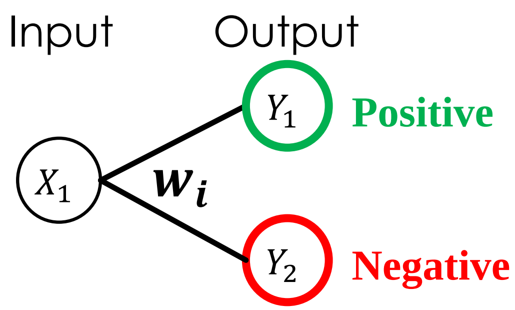
Implementing Previous Example



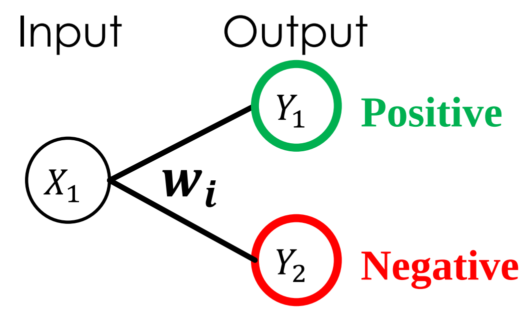

```
# import the packages we will be needing
import tflearn

# define the input layer (1 node)
X_i = tflearn.input_data(shape=[None, 1])

# define the output layer (2 nodes), softmax normalizes values between 0/1
Y_j = tflearn.fully_connected(X_i, n_units=2, activation='softmax')

# set up optimization problem (sgd = stochastic gradient descent)
optimization_problem = tflearn.regression(Y_j, optimizer="sgd")

# initialize variables w_i with random values to provide a starting point for optimization
model = tflearn.DNN(optimization_problem)
```

Implementing Previous Example

```
# import the packages we will be needing
import tflearn

# define the input layer (1 node)
X_i = tflearn.input_data(shape=[None, 1])

# define the output layer (2 nodes), softmax normalizes values between 0/1
Y_j = tflearn.fully_connected(X_i, n_units=2, activation='softmax')

# set up optimization problem (sgd = stochastic gradient descent)
optimization_problem = tflearn.regression(Y_j, optimizer="sgd")

# initialize variables w_i with random values to provide a starting point for optimization
model = tflearn.DNN(optimization_problem)

# train the model (assuming we have some data) for 100 iterations, update w every iteration
model.fit(data, label, n_epoch=100)
```

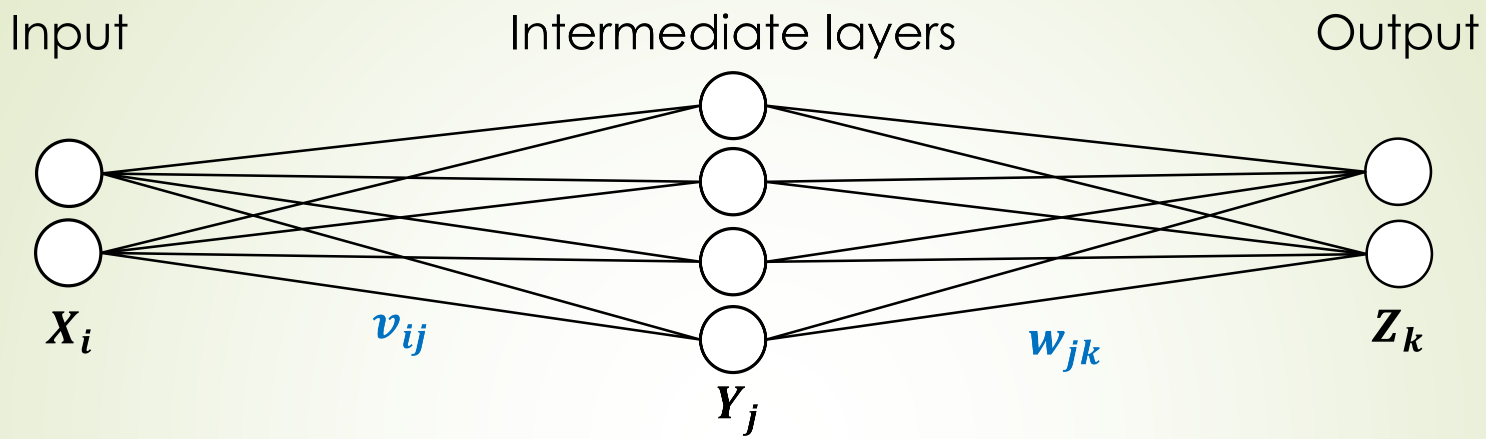


Implementing Previous Example




```
# import the packages we will be needing
import tflearn

# define the input layer (1 node)
X_i = tflearn.input_data(shape=[None, 1])

# define the output layer (2 nodes), softmax normalizes values between 0/1
Y_j = tflearn.fully_connected(X_i, n_units=2, activation='softmax')

# set up optimization problem (sgd = stochastic gradient descent)
optimization_problem = tflearn.regression(Y_j, optimizer="sgd")

# initialize variables w_i with random values to provide a starting point for optimization
model = tflearn.DNN(optimization_problem)

# train the model (assuming we have some data) for 100 iterations, update w every iteration
model.fit(data, label, n_epoch=100)

# make a prediction
print( model.predict([[1]]) )        # should get Y1 > Y2 (output = [Y1, Y2])
print( model.predict([[-1]]) )       # should get Y1 < Y2 (output = [Y1, Y2])
```

More Layers/Nodes



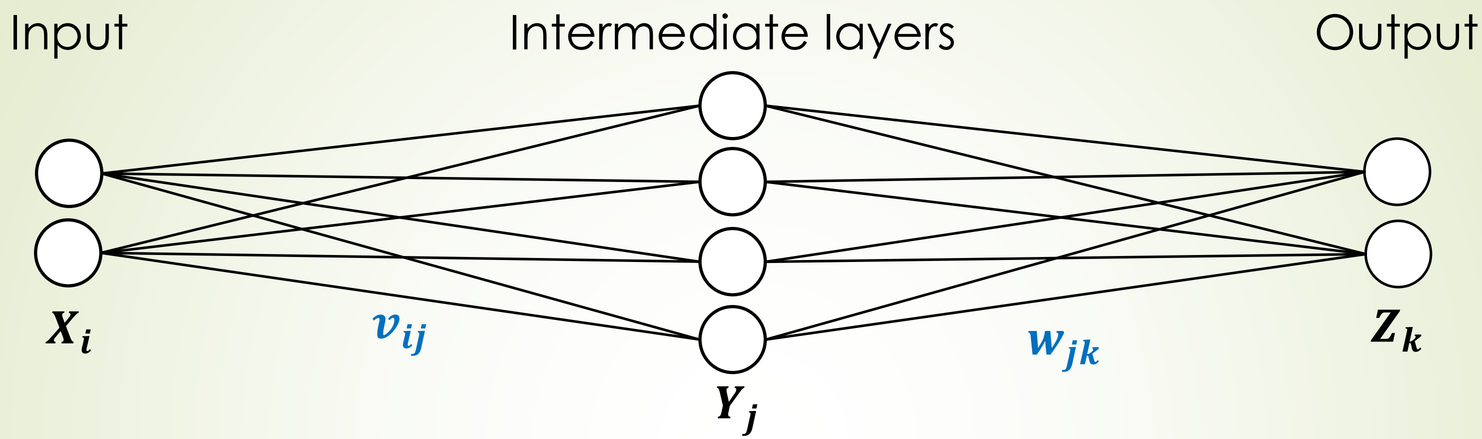
To map $X_i \rightarrow Y_j$:

$$[X_1 \quad X_2]\begin{bmatrix} \boldsymbol{v_{11}} & \boldsymbol{v_{12}} & \boldsymbol{v_{13}} & \boldsymbol{v_{14}} \\ \boldsymbol{v_{21}} & \boldsymbol{v_{22}} & \boldsymbol{v_{23}} & \boldsymbol{v_{24}} \end{bmatrix} = [Y_1 \quad Y_2 \quad Y_3 \quad Y_4]$$

To map $Y_j \rightarrow Z_k$:

$$[Y_1 \quad Y_2 \quad Y_3 \quad Y_4]\begin{bmatrix} \boldsymbol{w_{11}} & \boldsymbol{w_{12}} \\ \boldsymbol{w_{21}} & \boldsymbol{w_{22}} \\ \boldsymbol{w_{31}} & \boldsymbol{w_{32}} \\ \boldsymbol{w_{41}} & \boldsymbol{w_{42}} \end{bmatrix} = [Z_1 \quad Z_2]$$

More Layers/Nodes



To map $X_i \rightarrow Y_j$:

$$[X_1 \quad X_2]\begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \end{bmatrix} = [Y_1 \quad Y_2 \quad Y_3 \quad Y_4]$$

To map $Y_j \rightarrow Z_k$:

$$[Y_1 \quad Y_2 \quad Y_3 \quad Y_4]\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} = [Z_1 \quad Z_2]$$

Transformations are done with matrix multiplications

Convolutional Neural Networks

- Often referred to as “CNNs”.
- More complex model.
- Is able to handle more complex data (such as images).

Convolutional Neural Networks

- Often referred to as “CNNs”.
- More complex model.
- Is able to handle more complex data (such as images).

<u>Main difference with previous example:</u>

Instead of $X_i \rightarrow Y_j$ being a simple matrix multiplication, we transform X_i to Y_j with the **convolution operation**.

Convolution Operation

0	1	0	2	2
1	2	2	1	2
2	2	1	0	0
2	1	2	2	1
0	0	0	1	0

5x5x1 Input

Dot Product

-1	1	0
-1	0	1
0	-1	0

3x3x1 Filter

=

0	-3	2
-1	-4	-4
0	0	-3

3x3x1 Output

Convolution Operation

0	1	0	2	2
1	2	2	1	2
2	2	1	0	0
2	1	2	2	1
0	0	0	1	0

5x5x1 Input

-1	1	0
-1	0	1
0	-1	0

3x3x1 Filter

=

0	-3	2
-1	-4	-4
0	0	-3

3x3x1 Output

Parameters to optimize

Convolution Operation

0	1	0	2	2
1	2	2	1	2
2	2	1	0	0
2	1	2	2	1
0	0	0	1	0

5x5x1 Input

-1	1	0
-1	0	1
0	-1	0

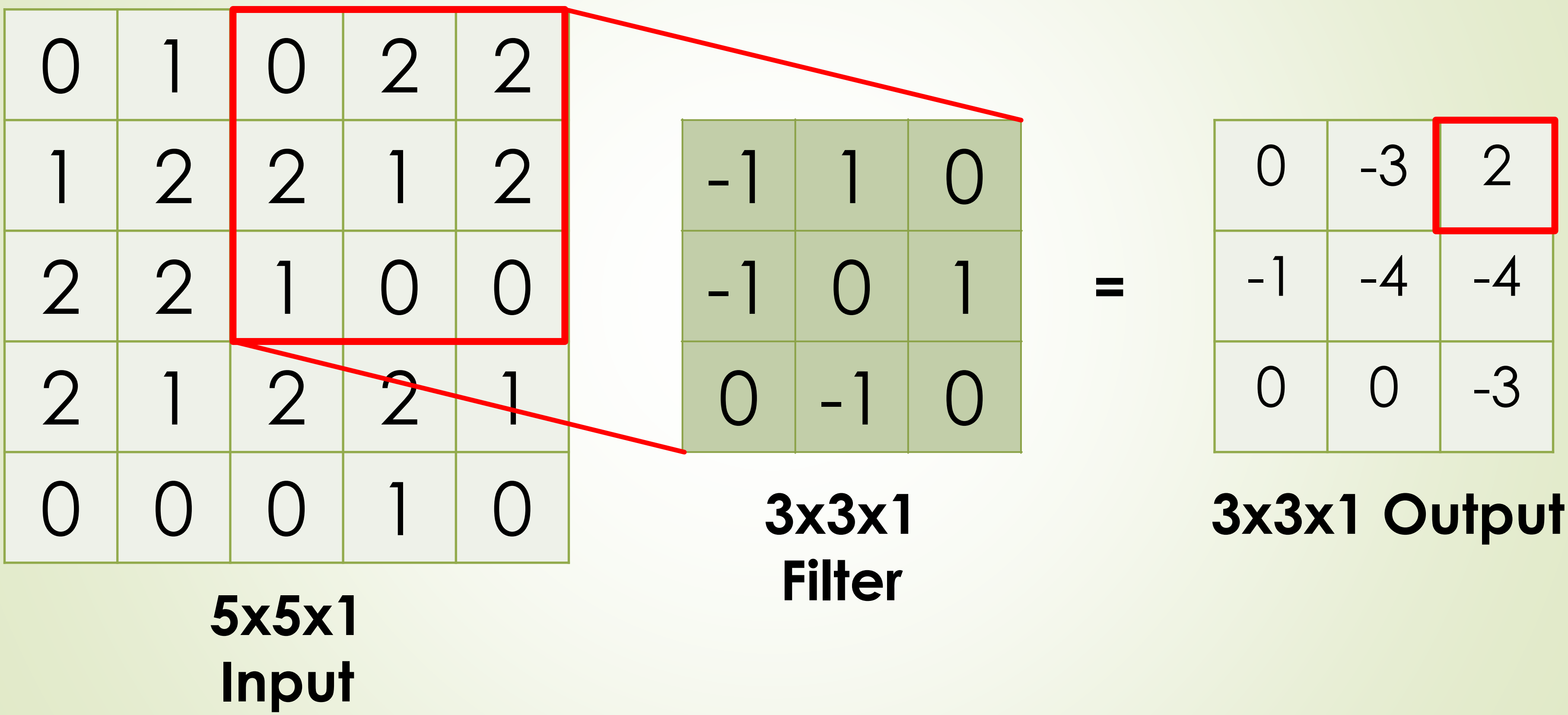
3x3x1 Filter

=

0	-3	2
-1	-4	-4
0	0	-3

3x3x1 Output

Convolution Operation



Convolution Operation

0	1	0	2	2
1	2	2	1	2
2	2	1	0	0
2	1	2	2	1
0	0	0	1	0

5x5x1 Input

-1	1	0
-1	0	1
0	-1	0

3x3x1 Filter

=

0	-3	2
-1	-4	-4
0	0	-3

3x3x1 Output

Convolution Operation

0	1	0	2	2
1	2	2	1	2
2	2	1	0	0
2	1	2	2	1
0	0	0	1	0

5x5x1 Input

-1	1	0
-1	0	1
0	-1	0

3x3x1 Filter

=

0	-3	2
-1	-4	-4
0	0	-3

3x3x1 Output

Convolution Operation

0	1	0	2	2
1	2	2	1	2
2	2	1	0	0
2	1	2	2	1
0	0	0	1	0

5x5x1 Input

-1	1	0
-1	0	1
0	-1	0

3x3x1 Filter

=

0	-3	2
-1	-4	-4
0	0	-3

3x3x1 Output

Convolution Operation

0	1	0	2	2
1	2	2	1	2
2	2	1	0	0
2	1	2	2	1
0	0	0	1	0

5x5x1 Input

-1	1	0
-1	0	1
0	-1	0

3x3x1 Filter

=

0	-3	2
-1	-4	-4
0	0	-3

3x3x1 Output

Loss of pixels

Convolution Operation
(zero-padding)

0	0	0	0	0	0	0
0	0	1	0	2	2	0
0	1	2	2	1	2	0
0	2	2	1	0	0	0
0	2	1	2	2	1	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0

7x7x1 Input

-1	1	0
-1	0	1
0	-1	0

3x3x1 Filter

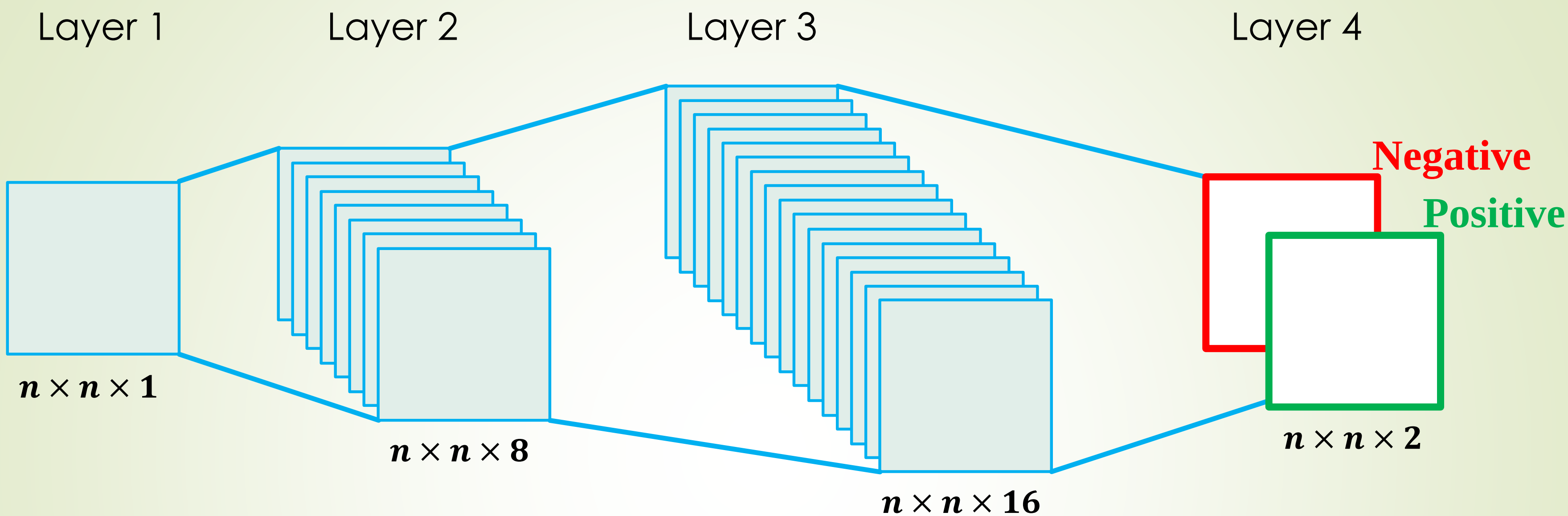
=

0	-2	-1	1	-4
0	0	-3	2	-1
1	-1	-4	-4	0
3	0	0	-3	-2
2	-1	2	0	-2

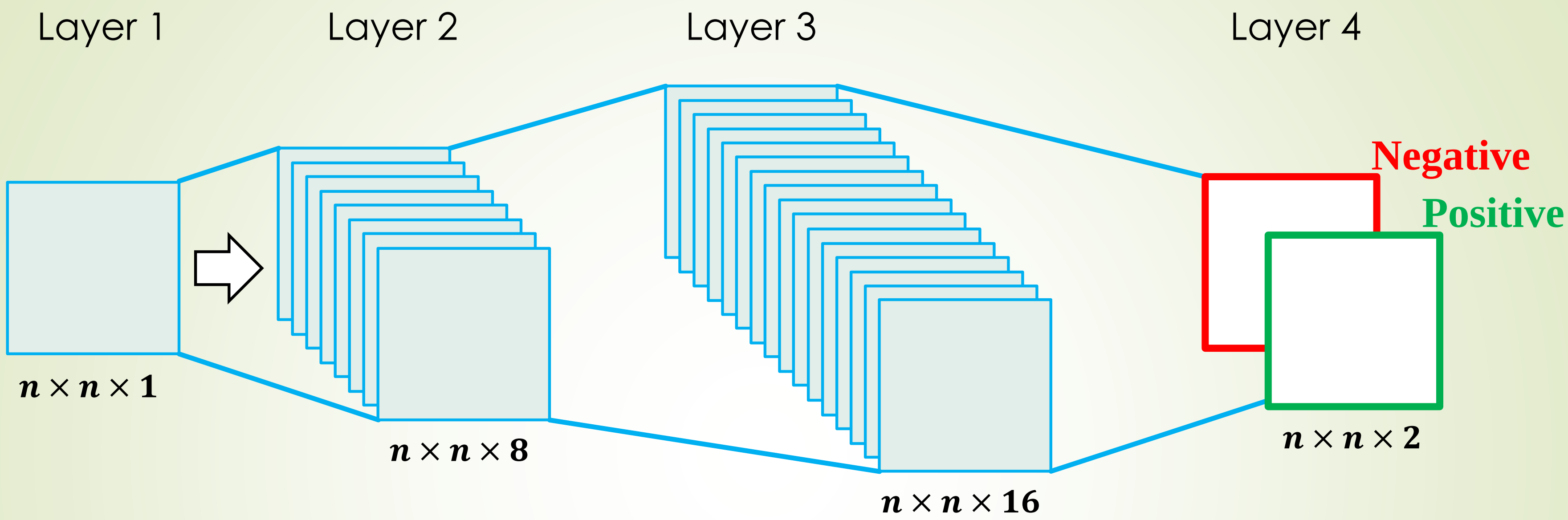
5x5x1 Output

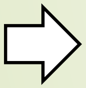
(original dimension maintained)

Convolutional Neural Networks



Convolutional Neural Networks

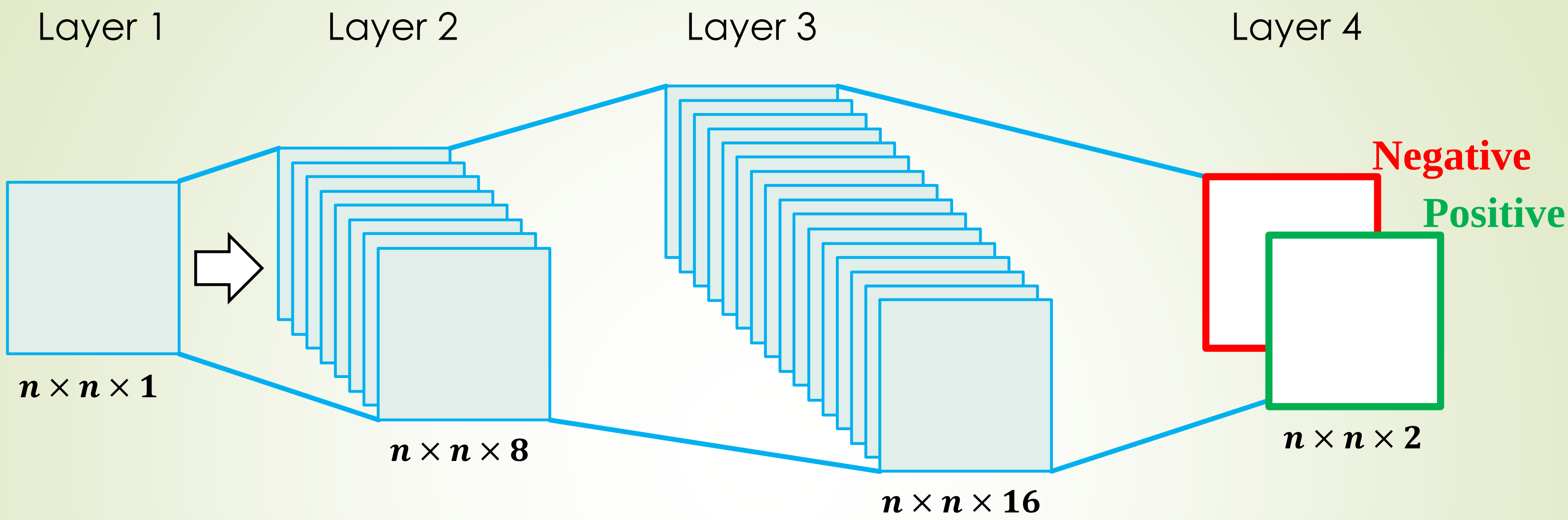


 **Convolution**

Layer 1 → Layer 2:

- Perform the convolution operation **8** times to get **8** *feature maps*.

Convolutional Neural Networks

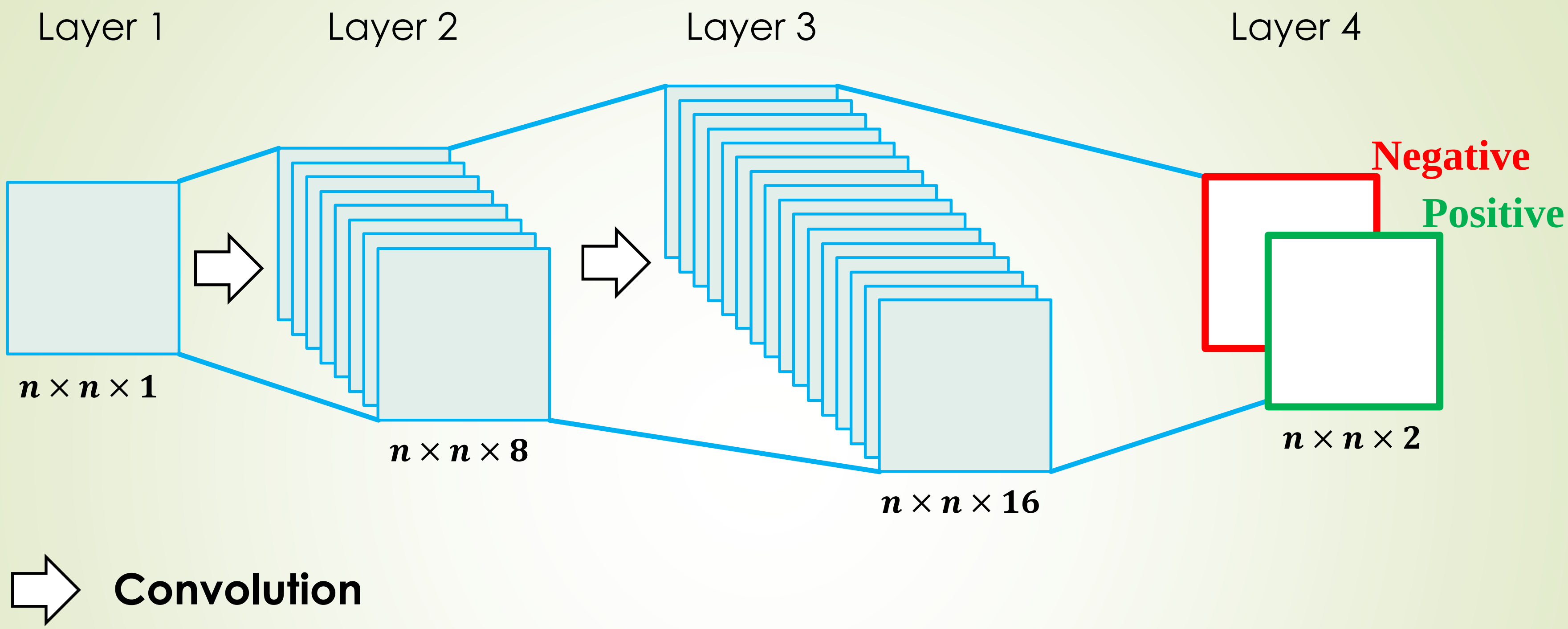


⇨ **Convolution**

Layer 1 → Layer 2:

- Perform the convolution operation **8** times to get **8** feature maps.
- Use **8** "<u>3×3×1</u> filters" to achieve this.

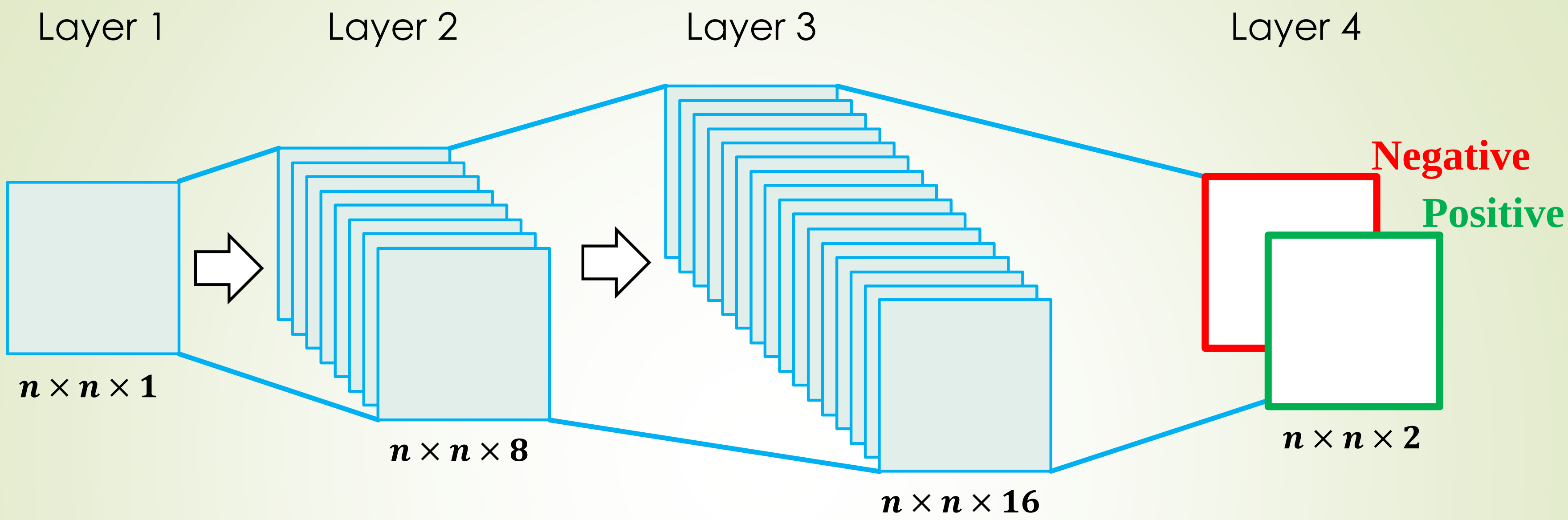
Convolutional Neural Networks



Layer 2 → Layer 3:

- Perform the convolution operation **16** times to get **16** feature maps.

Convolutional Neural Networks

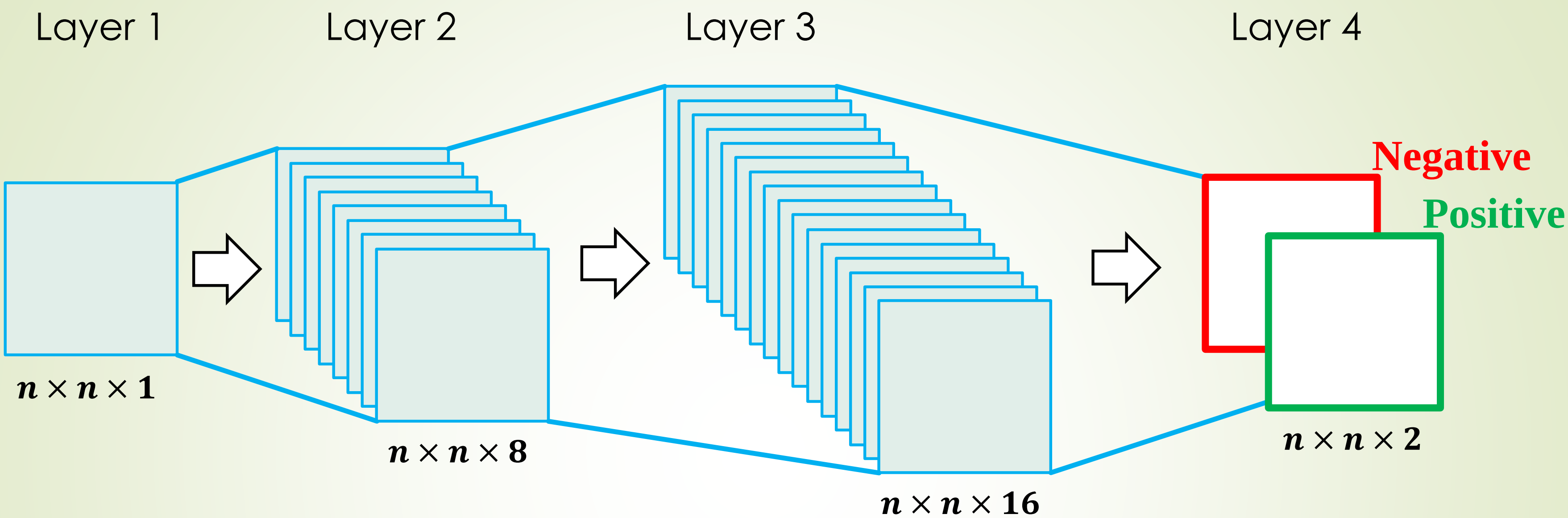


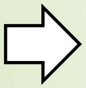
Convolution

Layer 2 → Layer 3:

- Perform the convolution operation **16** times to get **16** feature maps.
- Since the input now has 8 feature maps, use **16** "<u>3×3×8</u> filters" to achieve this.

Convolutional Neural Networks

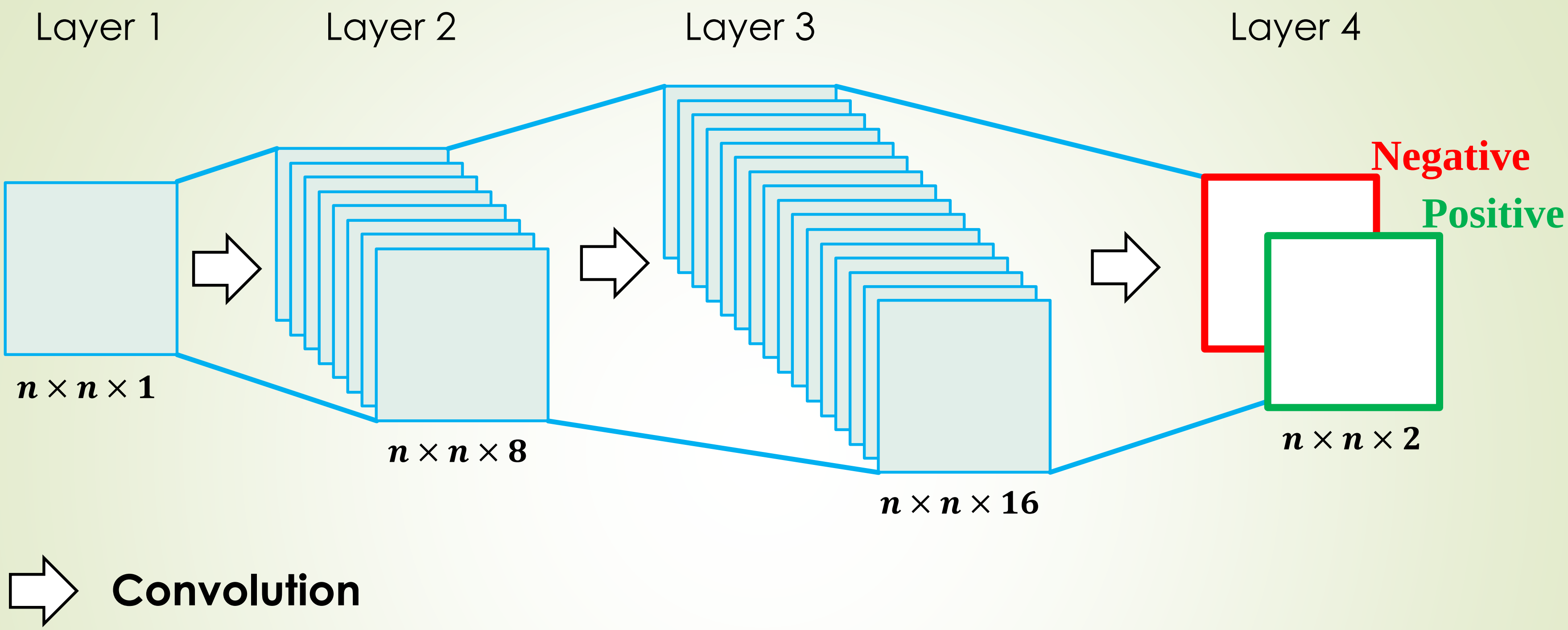



Convolution

Layer 3 → Layer 4:

- Perform the convolution operation **2** times to get **2** feature maps.
- Since the input now has 16 feature maps, use **2** "<u>3×3×16</u> filters" to achieve this.

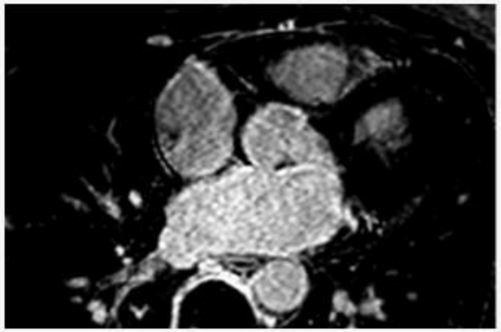
Convolutional Neural Networks



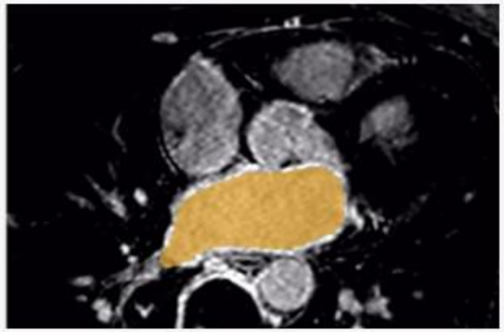
Layer 4:

- The two feature maps represents the probability of each pixel (in n×n pixels) being either positive or negative.

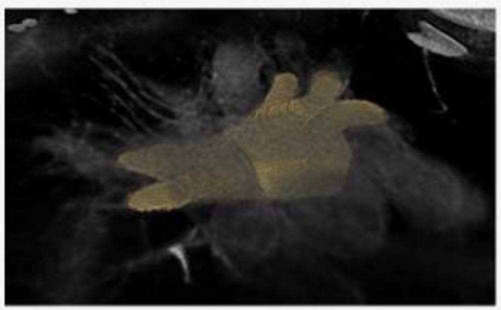
2018 Atrial Segmentation Challenge



Raw LGE-MRI



Left Atrial Cavity



3D Left Atria Superimposed on LGE-MRI

3D Left Atria Visualization

Data Summary

Size	100 Data for Training 25 Data for Testing
Pathology	Atrial Fibrillation
Images	3D Gadolinium-Enhanced Magnetic Resonance Imaging
Labels	3D Binary Masks of the Left Atrial Cavity

Background

Atrial fibrillation (AF) is the most common type of cardiac arrhythmia. The poor performance of current AF treatment is due to a lack of understanding of the structure of the human atria.

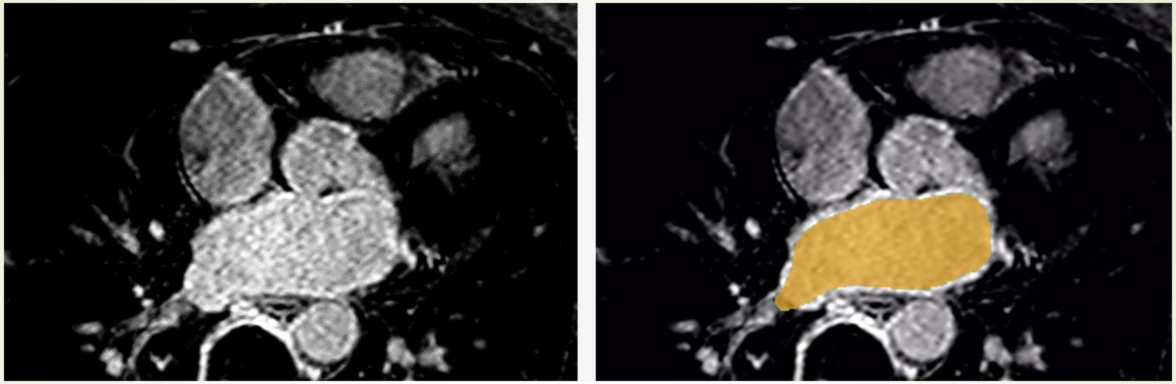
Organizers

Jichao Zhao
Zhaohan Xiong

http://atriaseg2018.cardiacatlas.org/

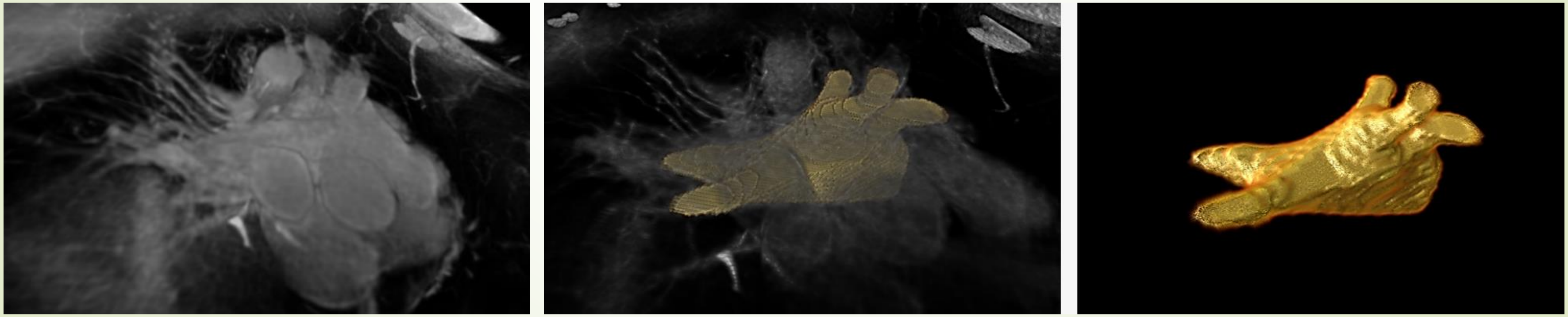
Atria Segmentation

2D



- Given the raw MRI (left), identify the pixels which belong to the atria (right).

3D



- Segmenting each slice of an MRI will result in 3D segmentation.

Atria Segmentation with CNNs

- Make a CNN to predict the mask for given a 3D MRI.
- CNN performs 2D slice-by-slice prediction for every slice of the MRI.
- Train the CNN with 2D MRI slices.

Atria Segmentation with CNNs

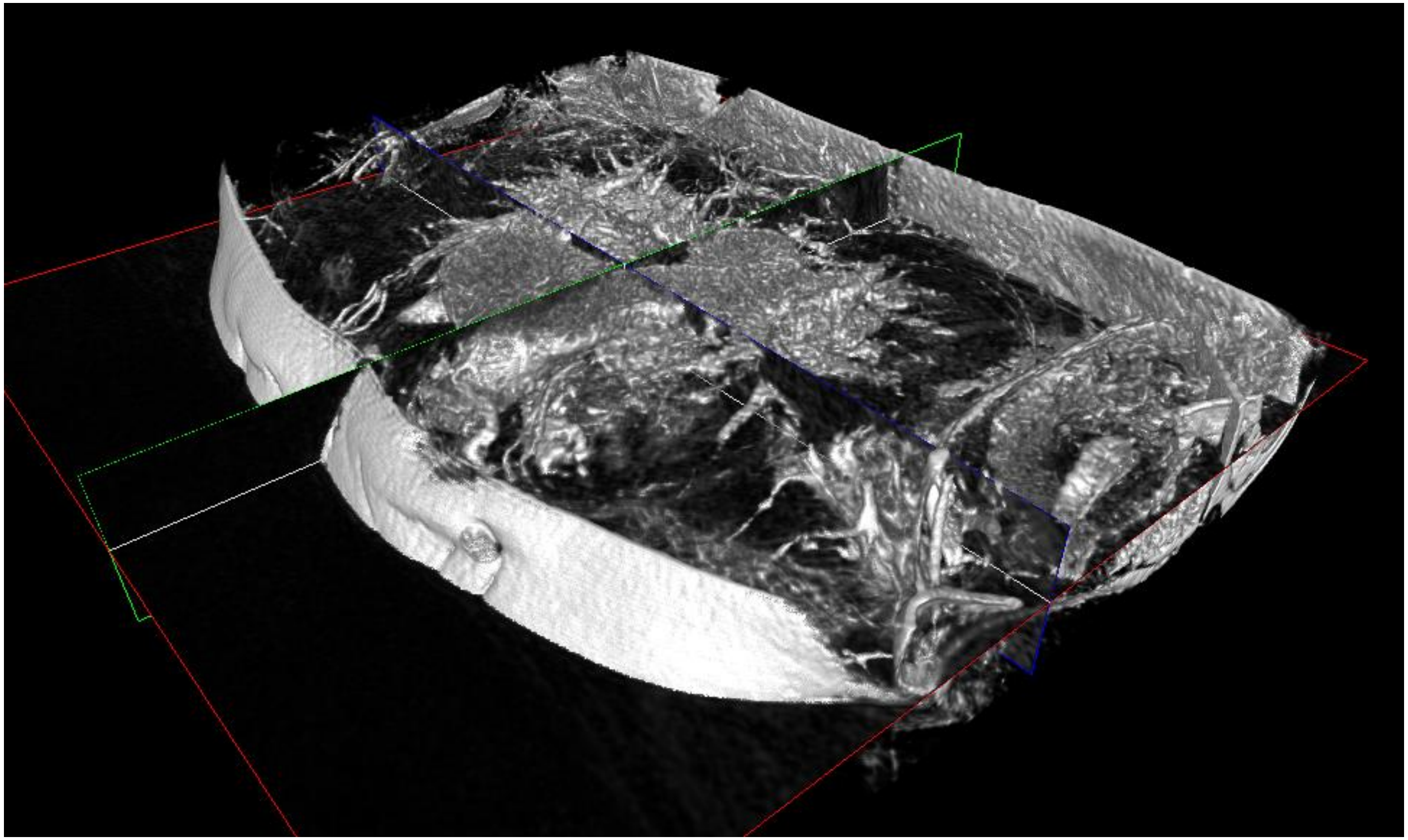
```
# import packages
import tflearn
import numpy as np
import SimpleITK as sitk
```

Atria Segmentation with CNNs

```
# import packages
import tflearn
import numpy as np
import SimpleITK as sitk

# helper function to  load .nrrd files
def load_nrrd(full_path_filename):
        # this function loads .nrrd files into a 3D matrix and outputs it
        # the input is the specified file path
        # the output is the Z x X by Y x Z for Z slices sized X x Y

        data = sitk.ReadImage( full_path_filename )                          # read in image
        data = sitk.Cast( sitk.RescaleIntensity(data), sitk.sitkUInt8 )      # convert to 8 bit (0-255)
        data = sitk.GetArrayFromImage( data )                                # convert to numpy array

        # expand the dimension to n_slices x width x height x 1
        data = np.expand_dims(data,4)

        return(data)
```

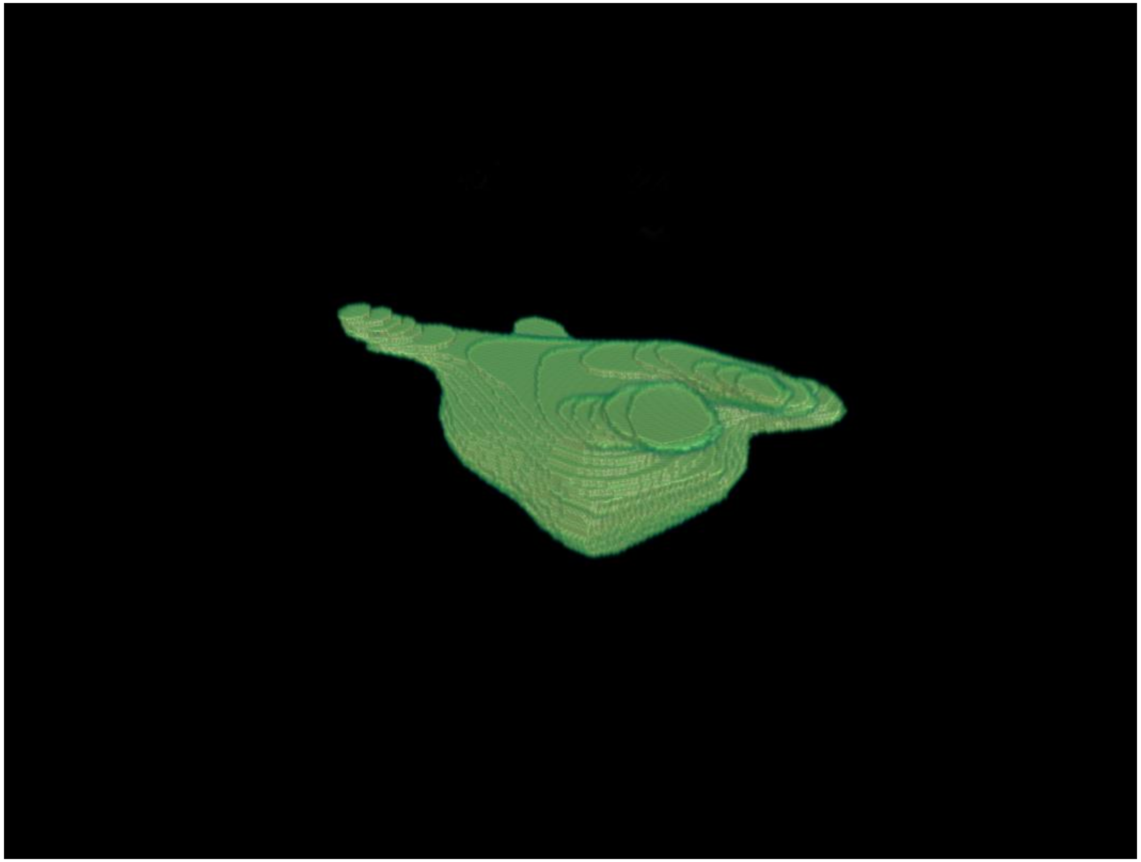
Atria Segmentation with CNNs

```
 # load the data given path to file
image = load_nrrd("lgemri.nrrd")

# the size is 88 x 640 x 640 x 1
print(image.shape)
```



Atria Segmentation with CNNs

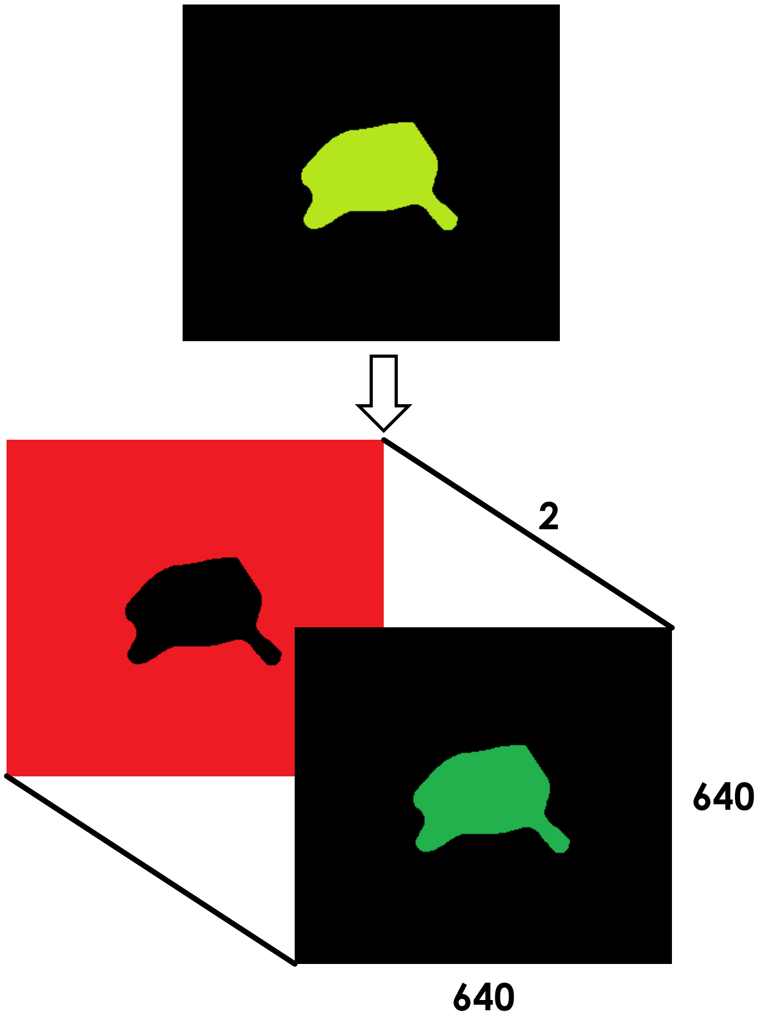
```
# load the data given path to file
image = load_nrrd("lgemri.nrrd")

# the size is 88 x 640 x 640 x 1
print(image.shape)

# load the label, normalize to [0,1]
mask = load_nrrd("mask.nrrd")//255
```



Atria Segmentation with CNNs

```
# load the data given path to file
image = load_nrrd("lgemri.nrrd")

# the size is 88 x 640 x 640 x 1
print(image.shape)

# load the label, normalize to [0,1]
mask = load_nrrd("mask.nrrd")//255

# encode the data into 2 layers
# (as shown in the neural network)
label = np.zeros(shape=[88,640,640,2])
label[:,:,:,0] = mask
label[:,:,:,1] = 1 - mask
```

For each slice:

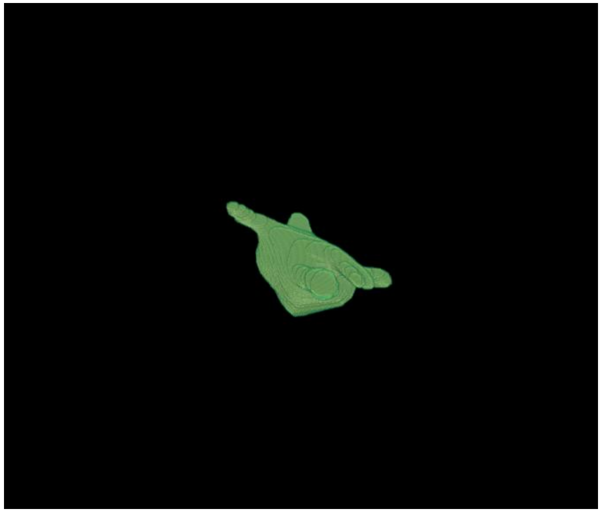


Atria Segmentation with CNNs

```
# load the data given path to file
image = load_nrrd("lgemri.nrrd")

# the size is 88 x 640 x 640 x 1
print(image.shape)

# load the label, normalize to [0,1]
mask = load_nrrd("mask.nrrd")//255

# encode the data into 2 layers
# (as shown in the neural network)
label = np.zeros(shape=[88,640,640,2])
label[:,:,:,0] = mask
label[:,:,:,1] = 1 - mask

# the size is 88 x 640 x 640 x 2
print(label.shape)
```

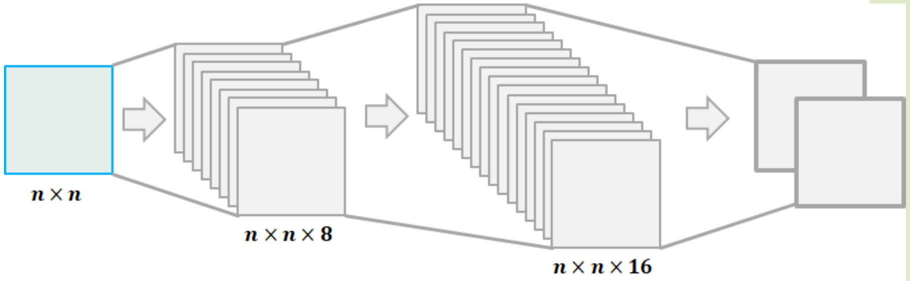
“image” (88x640x640x1)

“label” (88x640x640x2)



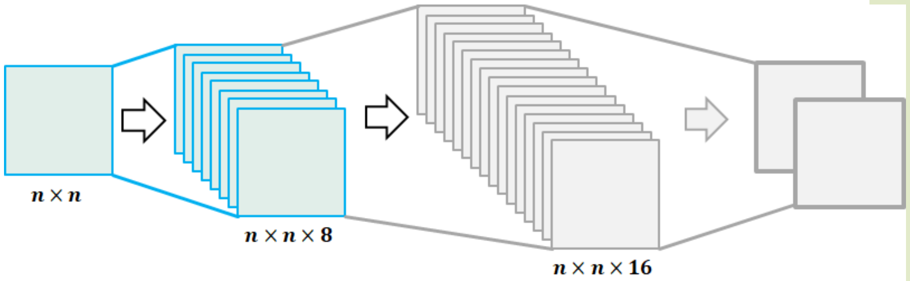
Atria Segmentation with CNNs

```
# # # make the neural network
# 640 x 640 x 1
layer_1 = tflearn.input_data(shape=[None,640,640,1])
```



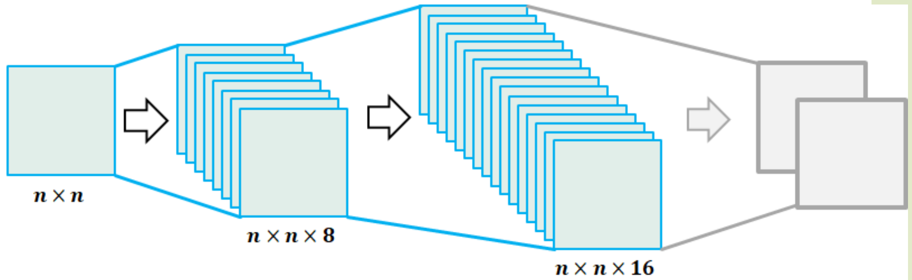
Atria Segmentation with CNNs

```
# # # make the neural network
# 640 x 640 x 1
layer_1 = tflearn.input_data(shape=[None,640,640,1])

# 640 x 640 x 1   --->   640 x 640 x 8
layer_2 = tflearn.conv_2d(layer_1, nb_filter=8, filter_size=3)
```

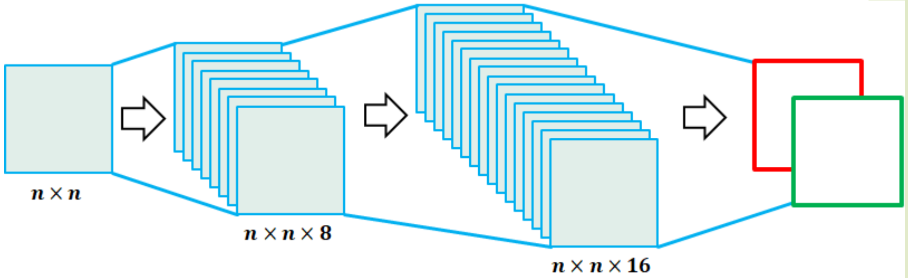


Atria Segmentation with CNNs

```
# # # make the neural network
# 640 x 640 x 1
layer_1 = tflearn.input_data(shape=[None,640,640,1])

# 640 x 640 x 1   --->   640 x 640 x 8
layer_2 = tflearn.conv_2d(layer_1, nb_filter=8, filter_size=3)

# 640 x 640 x 8   --->   640 x 640 x 16
layer_3 = tflearn.conv_2d(layer_2, nb_filter=16, filter_size=3)
```

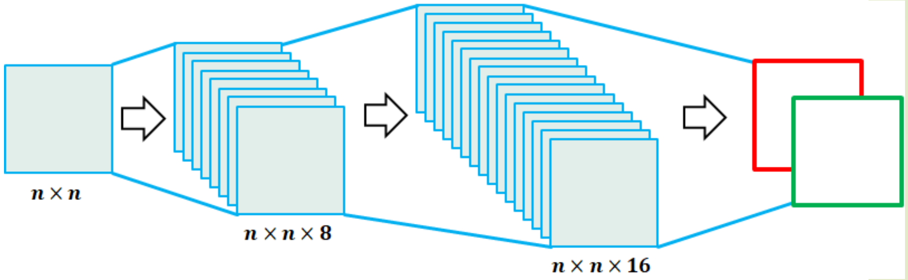


Atria Segmentation with CNNs



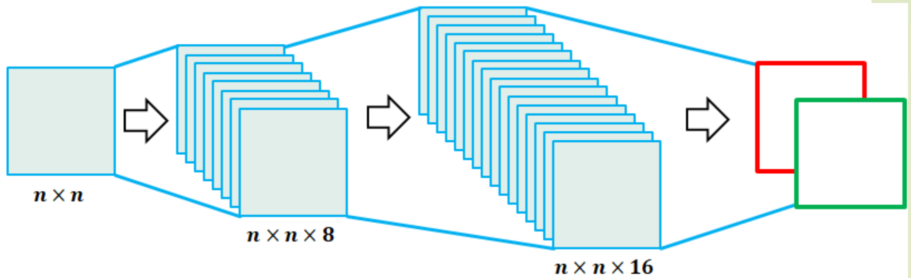

```
# # # make the neural network
# 640 x 640 x 1
layer_1 = tflearn.input_data(shape=[None,640,640,1])

# 640 x 640 x 1   --->   640 x 640 x 8
layer_2 = tflearn.conv_2d(layer_1, nb_filter=8, filter_size=3)

# 640 x 640 x 8   --->   640 x 640 x 16
layer_3 = tflearn.conv_2d(layer_2, nb_filter=16, filter_size=3)

# 640 x 640 x 16   --->   640 x 640 x 2, softmax normalizes values between 0/1
layer_4 = tflearn.conv_2d(layer_3, nb_filter=2, filter_size=3, activation='softmax')
```

Atria Segmentation with CNNs



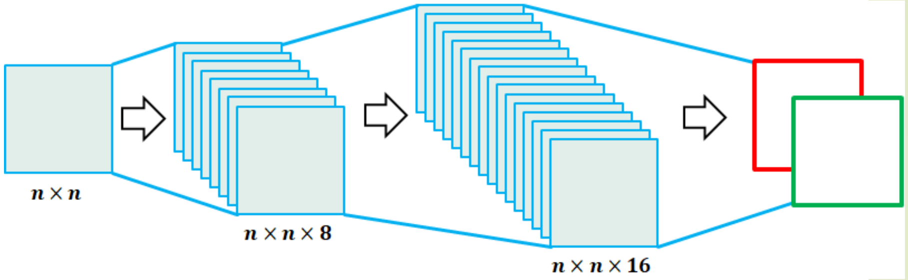

```
# # # make the neural network
# 640 x 640 x 1
layer_1 = tflearn.input_data(shape=[None,640,640,1])

# 640 x 640 x 1   --->   640 x 640 x 8
layer_2 = tflearn.conv_2d(layer_1, nb_filter=8, filter_size=3)

# 640 x 640 x 8   --->   640 x 640 x 16
layer_3 = tflearn.conv_2d(layer_2, nb_filter=16, filter_size=3)

# 640 x 640 x 16   --->   640 x 640 x 2, softmax normalizes values between 0/1
layer_4 = tflearn.conv_2d(layer_3, nb_filter=2, filter_size=3, activation='softmax')

# set up optimization problem (sgd = stochastic gradient descent)
optimization_problem = tflearn.regression(layer_4, optimizer="sgd")

# initialize variables
model = tflearn.DNN(optimization_problem)
```

Atria Segmentation with CNNs




```
# # # make the neural network
# 640 x 640 x 1
layer_1 = tflearn.input_data(shape=[None,640,640,1])

# 640 x 640 x 1   --->   640 x 640 x 8
layer_2 = tflearn.conv_2d(layer_1, nb_filter=8, filter_size=3)

# 640 x 640 x 8   --->   640 x 640 x 16
layer_3 = tflearn.conv_2d(layer_2, nb_filter=16, filter_size=3)

# 640 x 640 x 16   --->   640 x 640 x 2, softmax normalizes values between 0/1
layer_4 = tflearn.conv_2d(layer_3, nb_filter=2, filter_size=3, activation='softmax')

# set up optimization problem (sgd = stochastic gradient descent)
optimization_problem = tflearn.regression(layer_4, optimizer="sgd")

# initialize variables
model = tflearn.DNN(optimization_problem)

# train the model
model.fit(image,  label,  n_epoch=100)
```

Atria Segmentation with CNNs




```
# # # make the neural network
# 640 x 640 x 1
layer_1 = tflearn.input_data(shape=[None,640,640,1])

# 640 x 640 x 1   --->   640 x 640 x 8
layer_2 = tflearn.conv_2d(layer_1, nb_filter=8, filter_size=3)

# 640 x 640 x 8   --->   640 x 640 x 16
layer_3 = tflearn.conv_2d(layer_2, nb_filter=16, filter_size=3)

# 640 x 640 x 16   --->   640 x 640 x 2, softmax normalizes values between 0/1
layer_4 = tflearn.conv_2d(layer_3, nb_filter=2, filter_size=3, activation='softmax')

# set up optimization problem (sgd = stochastic gradient descent)
optimization_problem = tflearn.regression(layer_4, optimizer="sgd")

# initialize variables
model = tflearn.DNN(optimization_problem)

# train the model
model.fit(image,  label,  n_epoch=100)

# make a prediction, repeat this for every slice (0,1,2..... 87)
model.predict([ new_data[0,:,:,:] ])
```