Part 4 - Computer Vision & CNN

Convolutional neural network (CNN) = a **representation of specified *IMAGE* data in convoluted layers**

- The fundemental difference between a dense layer and a convolutional layer is that **dense layers detect patterns globally** while **convolutional layers detect patterns locally**.
- **The goal of convolutional neural networks is to classify and detect images or specific objects from within the image that can be used to classify the image or parts of it**.
- **CNN plays a main role to perform** the ***image recognition*** and ***object detection/classification*** using deep ***computer vision***.
- We will be using image data as our features and a label for those images as our label or output.

This is our image; the goal of our network will be to determine whether this image is a cat or not.

Dense Layer: A dense layer will **consider the ENTIRE image for all the pixels to generate the output at the exact pattern**.

Convolutional Layer: The convolutional layer **will look at specific parts of the image, which analyzes the highlighted parts (features) and detects patterns in the particular regions.**

Image Data

Image data is usually made up of 3 dimensions.

These 3 dimensions are as follows:

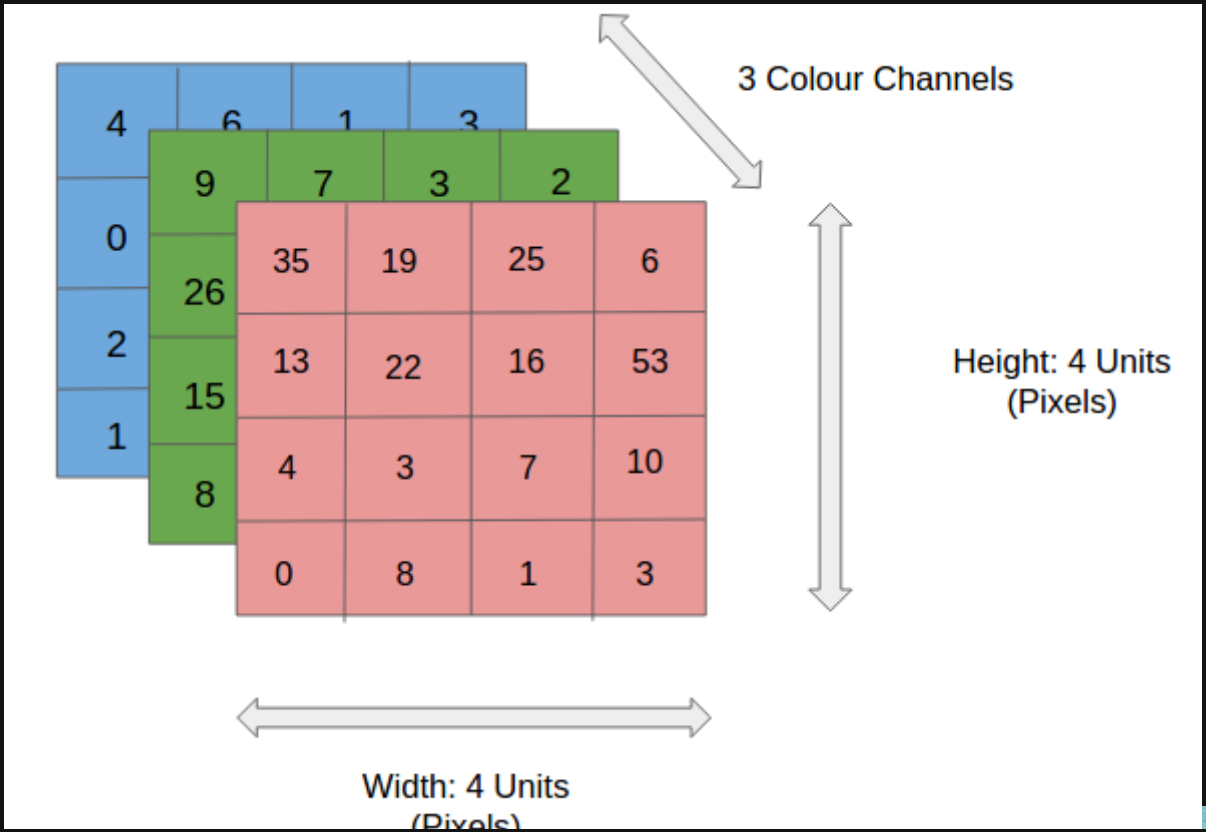
1. Image height
2. Image width
3. **Color channels**

Color Channels

The number of color channels represents the depth of an image & **correlates to the colors** used in it.

For example, an image with **three channels** is likely made up of **rgb (red, green, blue) pixels**. So, for each pixel we have three numeric values in the range 0-255 that define its color. For an image of

color depth 1 we would likely have a greyscale image with one value defining each pixel, again in the range of 0-255.



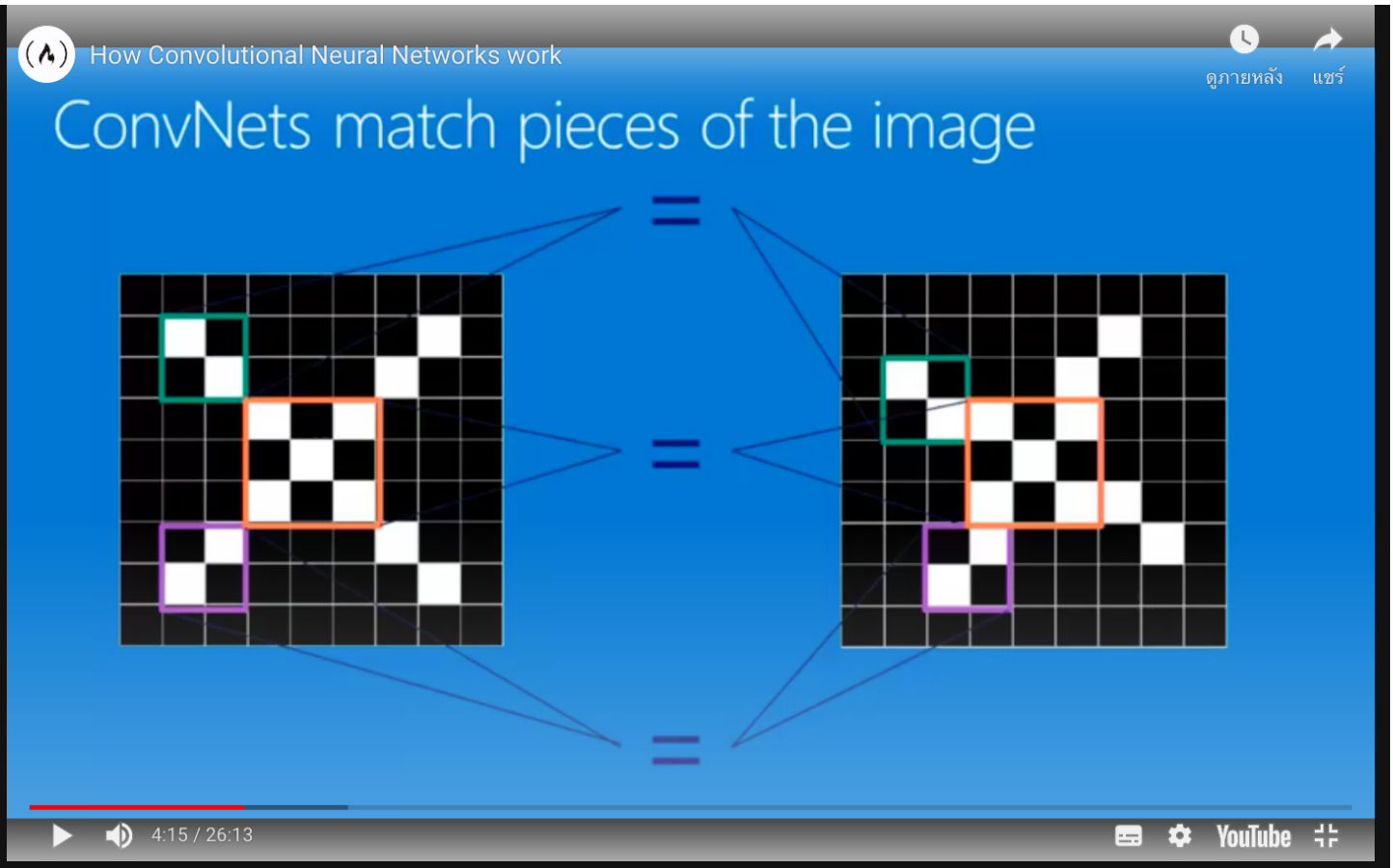
Convolutional Neural Network

Dense Layer: considers the **ENTIRE image** in exact pattern **-> which detects pattern globally** & then recognize pattern in specific area.

- In DNN, when we have a densely connected layer, **each node in that layer sees all the data from the previous layer**.
- This means that this dense layer is **looking at all the information which is only capable of analyzing the data in a global capacity**.

Convolutional Layer: considers the **specific little parts of the image** anywhere in the photo **-> which detects pattern locally** & **then learn these detected local patterns**.

- Our convolutional layer however is **NOT densely connected**, this means it **can detect local patterns using part of the input data to that layer**.

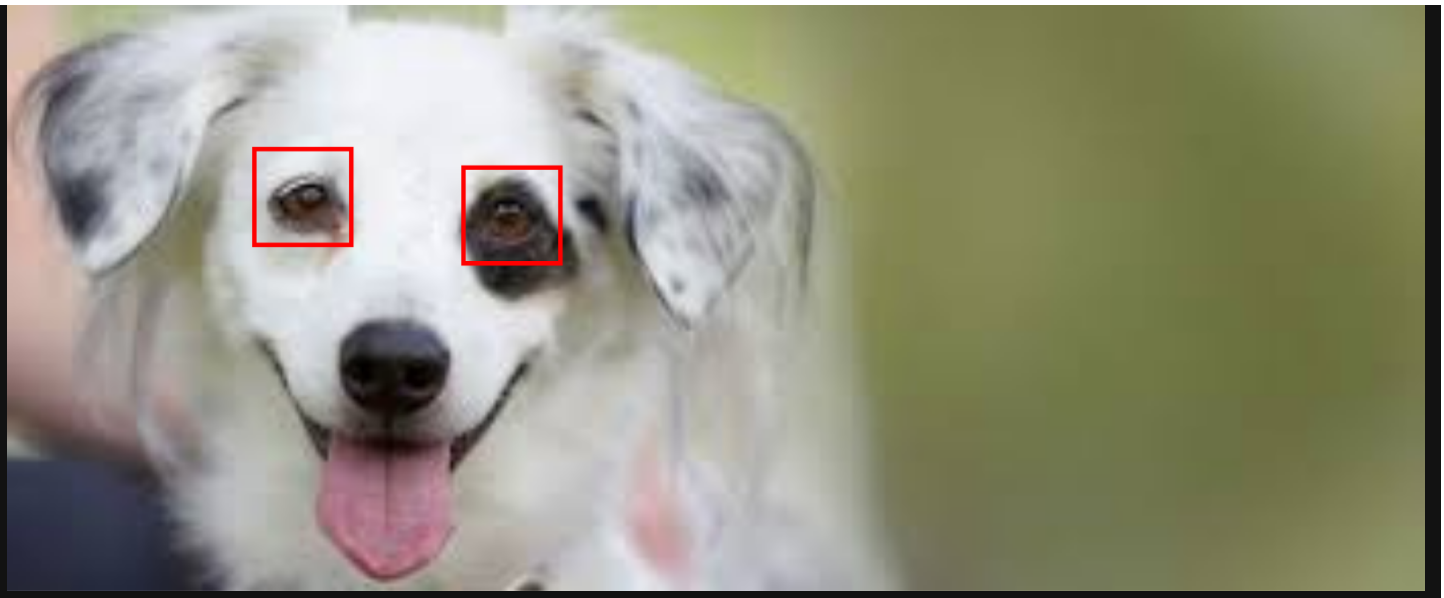


How CNN Works

A **dense neural network** learns patterns that are present in one specific area of an image. This means **if a pattern that the network knows is present in a different area of the image it will have to learn the pattern again in that new area to be able to detect it**.

Let's use an example to better illustrate this.

We'll consider that we have a dense neural network that has learned what an eye looks like from a sample of dog images.



Let's say it's determined that an image is likely to be a dog if an eye is present in the boxed off locations of the image above.

Now let's flip the image.

Since our **densely connected network** has **only recognized patterns globally**, it will look (fix on) where it thinks the eyes should be present. Clearly it does not find them there and therefore would likely determine this image is not a dog. Even though the pattern of the eyes is present, it's just in a different location.

Since **convolutional layers** learn and **detect patterns from different areas of the image**, they don't have problems with the example we just illustrated. They know what an eye looks like and **by analyzing different parts of the image can find where it is present**.

Multiple Convolutional Layers

In our models it is quite common to have more than one convolutional layer. Even the basic example we will use in this guide will be made up of **3 convolutional layers**.

- **These layers work together by increasing complexity and abstraction at each subsequent layer**.
    1. **The first layer** might be responsible for **picking up edges** and short lines, while
    2. **The second layer** will **take as input these lines and start forming shapes** or polygons. Finally,
    3. **The last layer** might **take these shapes and determine which combinations make up a specific image**.

Feature Maps

Feature map = **2D-map + depth (as CNN output)**. That is a 3D tensor with two spatial axes (width and height) and one depth axis.

- Our convolutional layers **take feature maps as their input** and **return a new feature map that represents the prescence of specific filters** from the previous feature map.
- These are what we call **response maps**.

Layer Parameters

The **properties of a convolutional layer** are:

- **Input size**
- **Filters = the pattern of pixels that we are looking for in an image**
    - **The number of filters** in a convolutional layer represents **how many patterns each layer is looking for** and **what the depth of our response map** will be.
- **Sample size of filters**

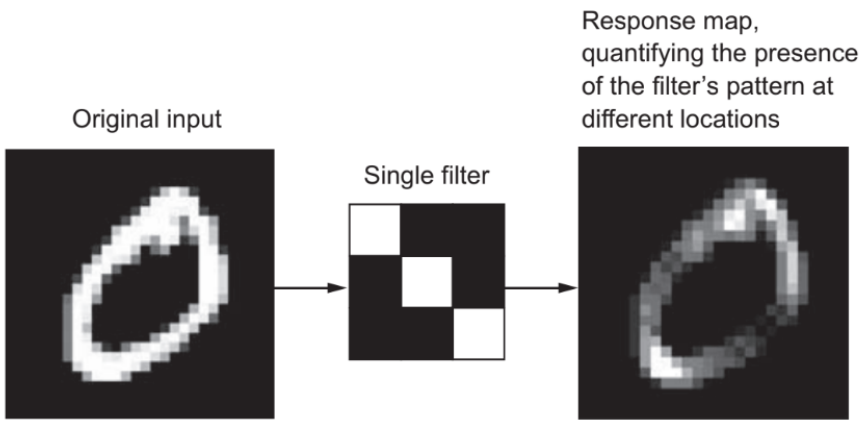


Figure 5.3 The concept of a *response map*: a 2D map of the presence of a pattern at different locations in an input

The process goes as follows:

1. **The sample sized filter** is **calculated by (matrix-wise) dot-product with the input image averaging with #pixel in size of filter.**
    - **Each sample size filter within the layer is shifted (stride) and then calculated via dot-product with the input averaging with #pixel in size of filter all over again to create an output feature map**
2. **Once all of the filters are calculated in the input layer**, we then **start the process all over again for the next layer**.
    - Instead of using pixels, **we are using the calculated numbers (output feature map) to find combinations of features that exist in the image**.
    - **This helps us to find lines & curves,** combinations of lines and curves, etc.

- **Padding = Adding the number of pixels** to your ***input*** data **such that each pixel can be centered by the filter**. --> **Extending the input image**
    - Adds a border to help look at the edges of a photo.
- **Stride = How many rows/cols we will move the filter in each time**.
- **Pooling = Taking specific values from a sample** of an ***output*** feature map **to reduce the size of the feature map.**
    - **Pooling Values: Min, Max, Average**

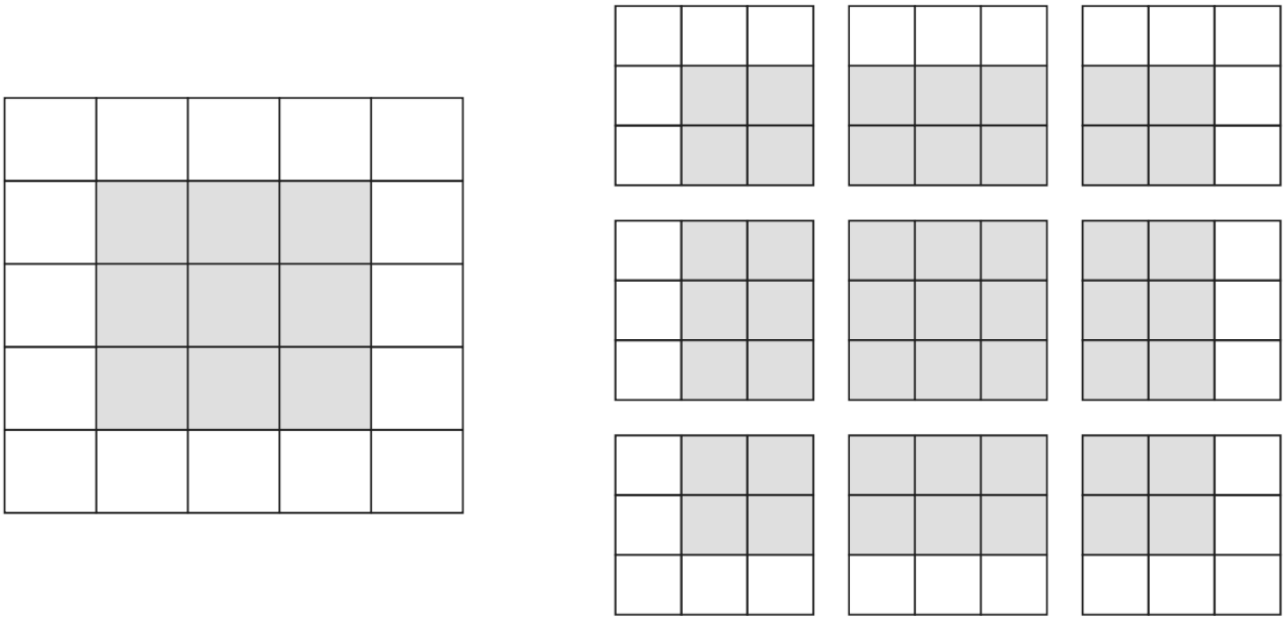
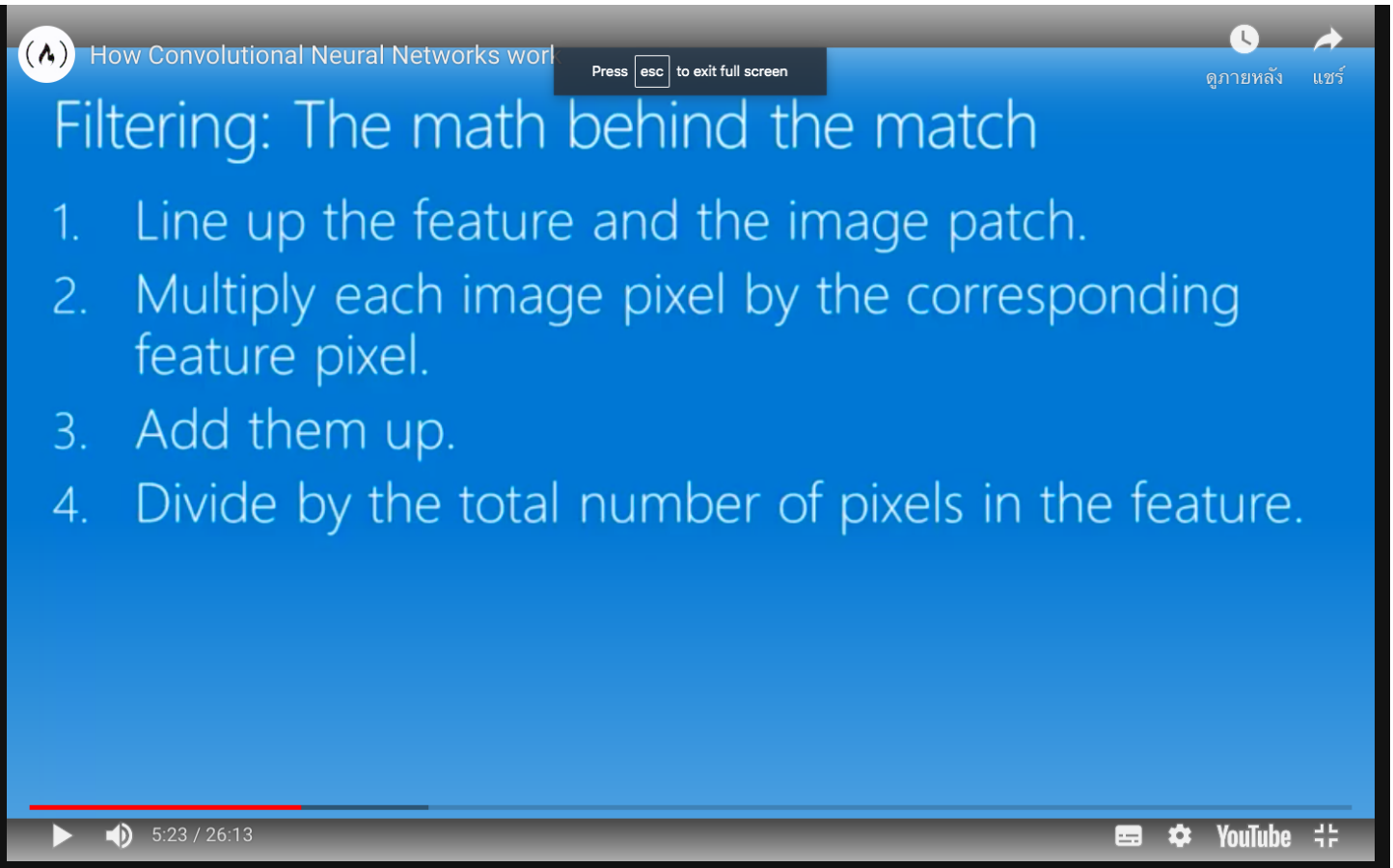


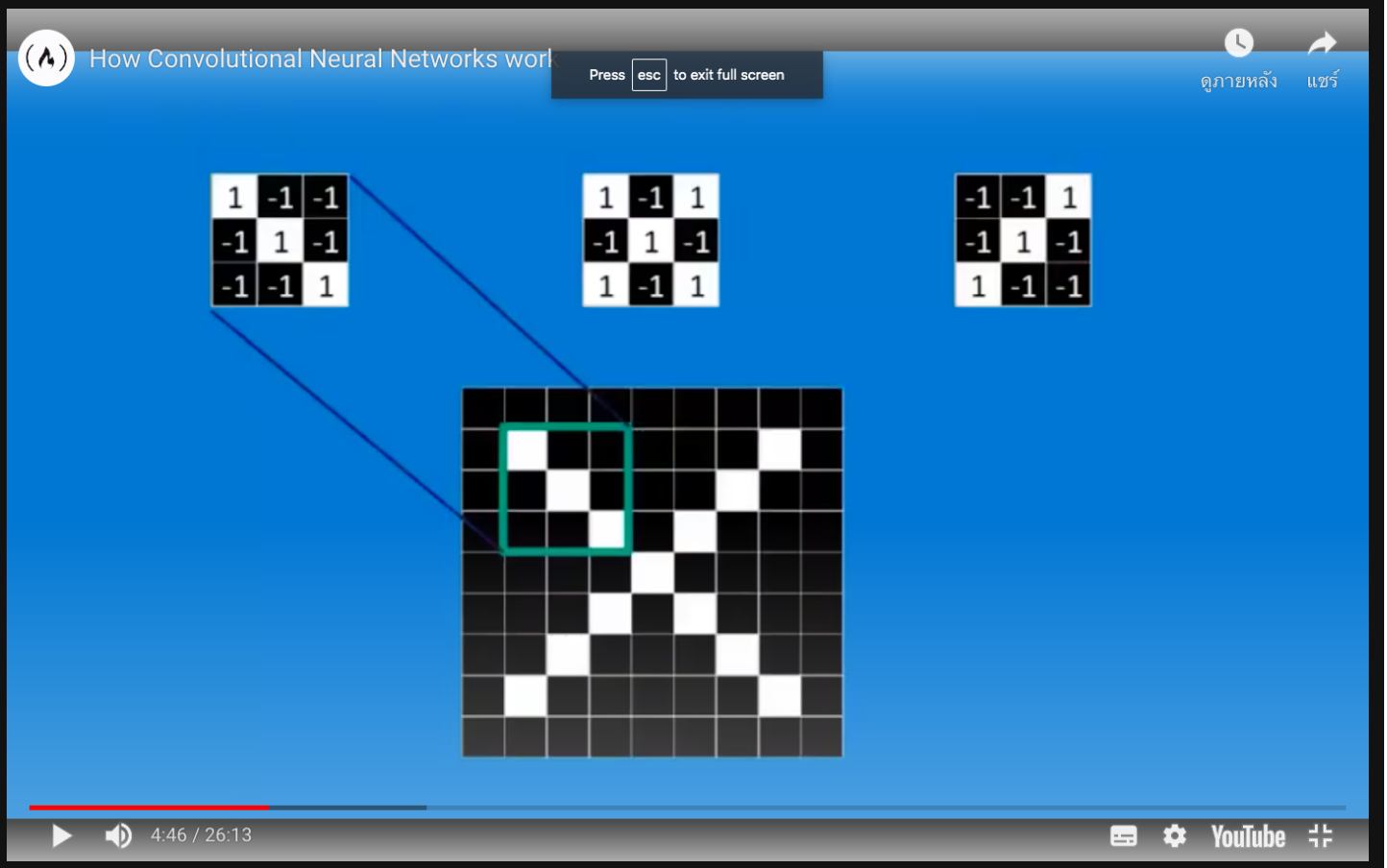
Figure 5.5 Valid locations of 3 × 3 patches in a 5 × 5 input feature map

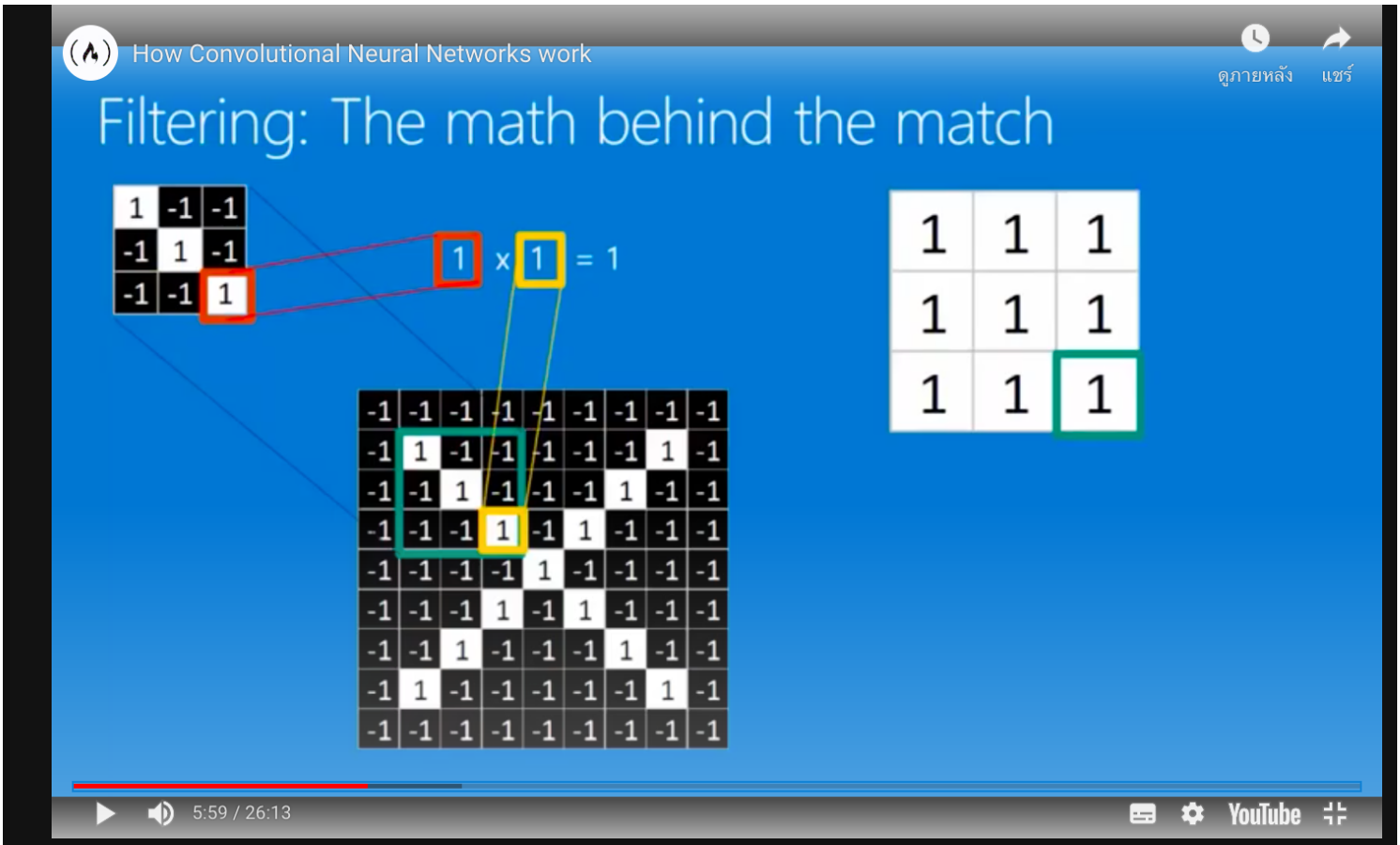
---

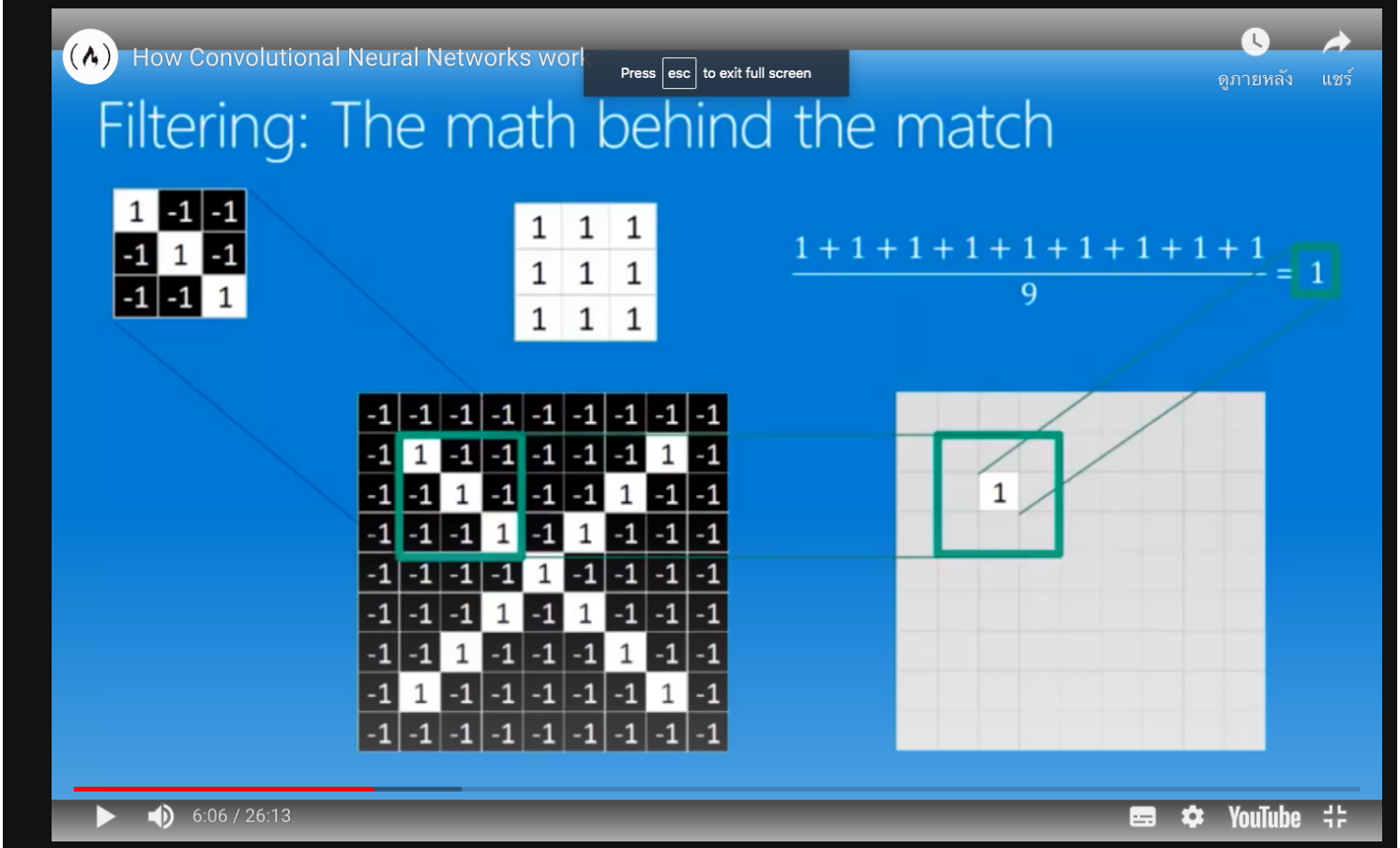
CNN working process : Lecture by Brandon Rohrer

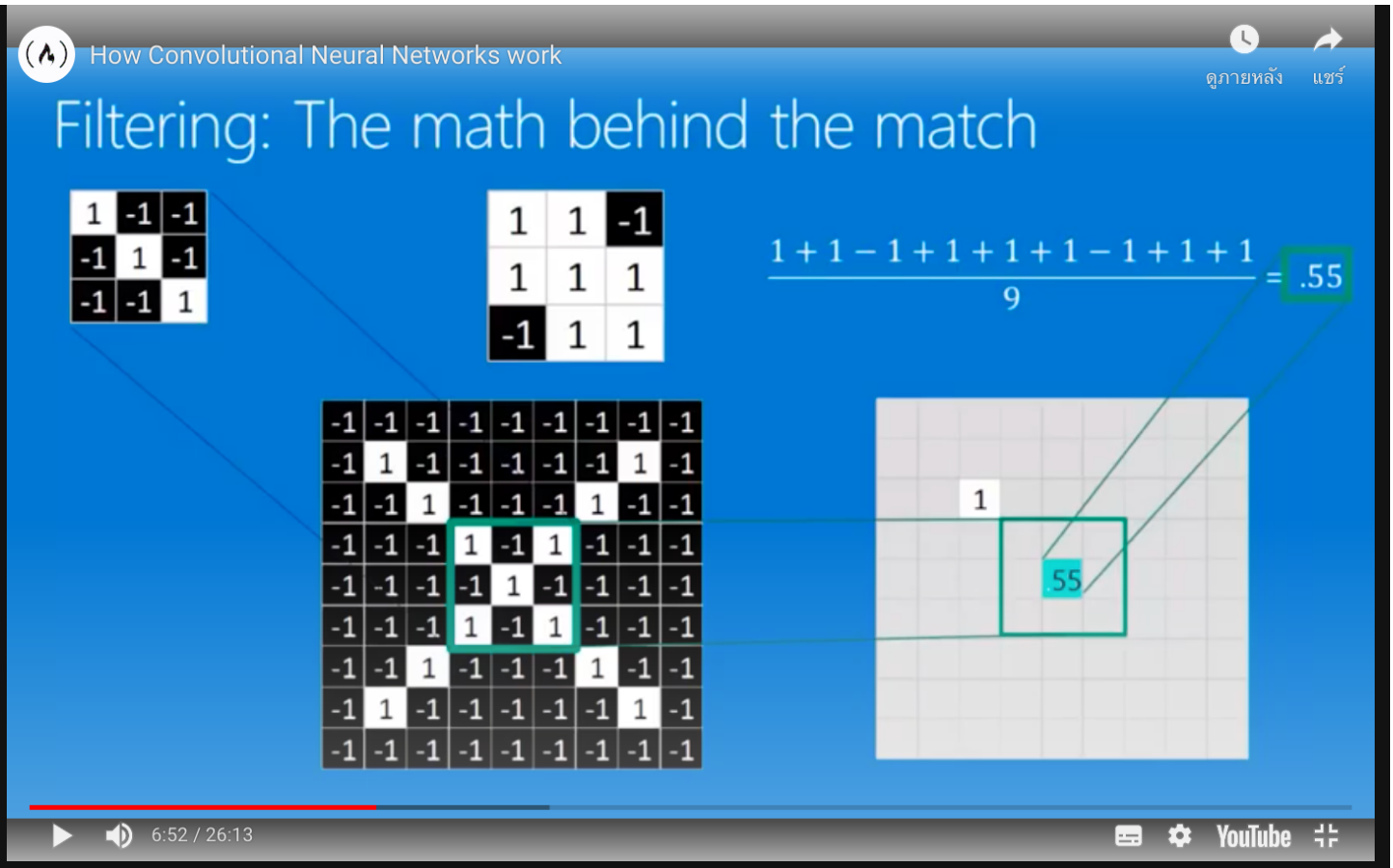
1) Filtering


How Convolutional Neural Networks work
Press esc to exit full screen
ดูภายหลัง
แชร์
Filtering: The math behind the match
1. Line up the feature and the image patch.
2. Multiply each image pixel by the corresponding feature pixel.
3. Add them up.
4. Divide by the total number of pixels in the feature.
5:23 / 26:13
YouTube


How Convolutional Neural Networks work
Press esc to exit full screen
ดูภายหลัง
แชร์
1 -1 -1
-1 1 -1
-1 -1 1
1 -1 1
-1 1 -1
1 -1 1
-1 -1 1
-1 1 -1
1 -1 -1
4:46 / 26:13
YouTube

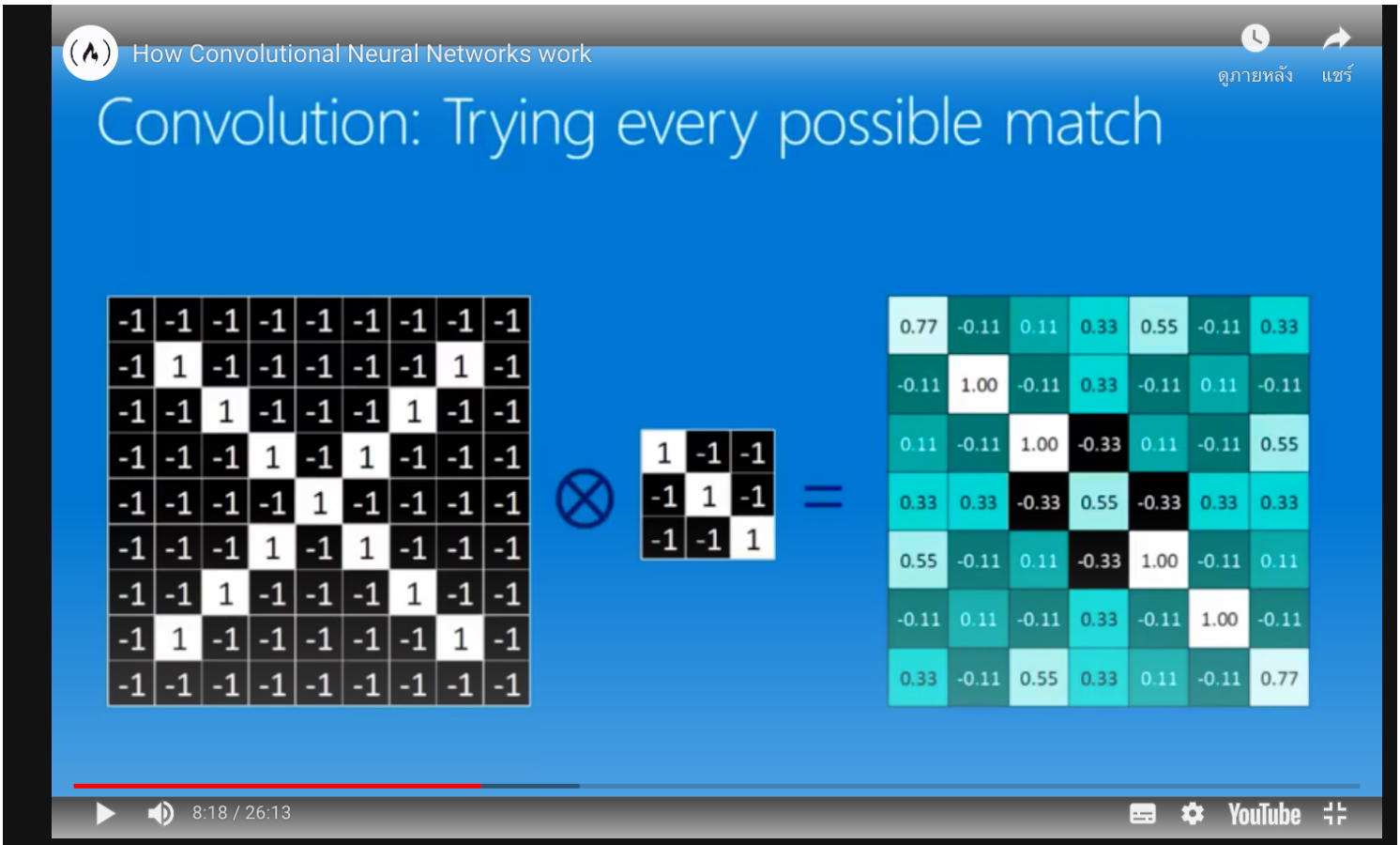

How Convolutional Neural Networks work
ดูภายหลัง
แชร์
Filtering: The math behind the match
1 -1 -1
-1 1 -1
-1 -1 1
1 x 1 = 1
1 1 1
1 1 1
1 1 1
5:59 / 26:13
YouTube

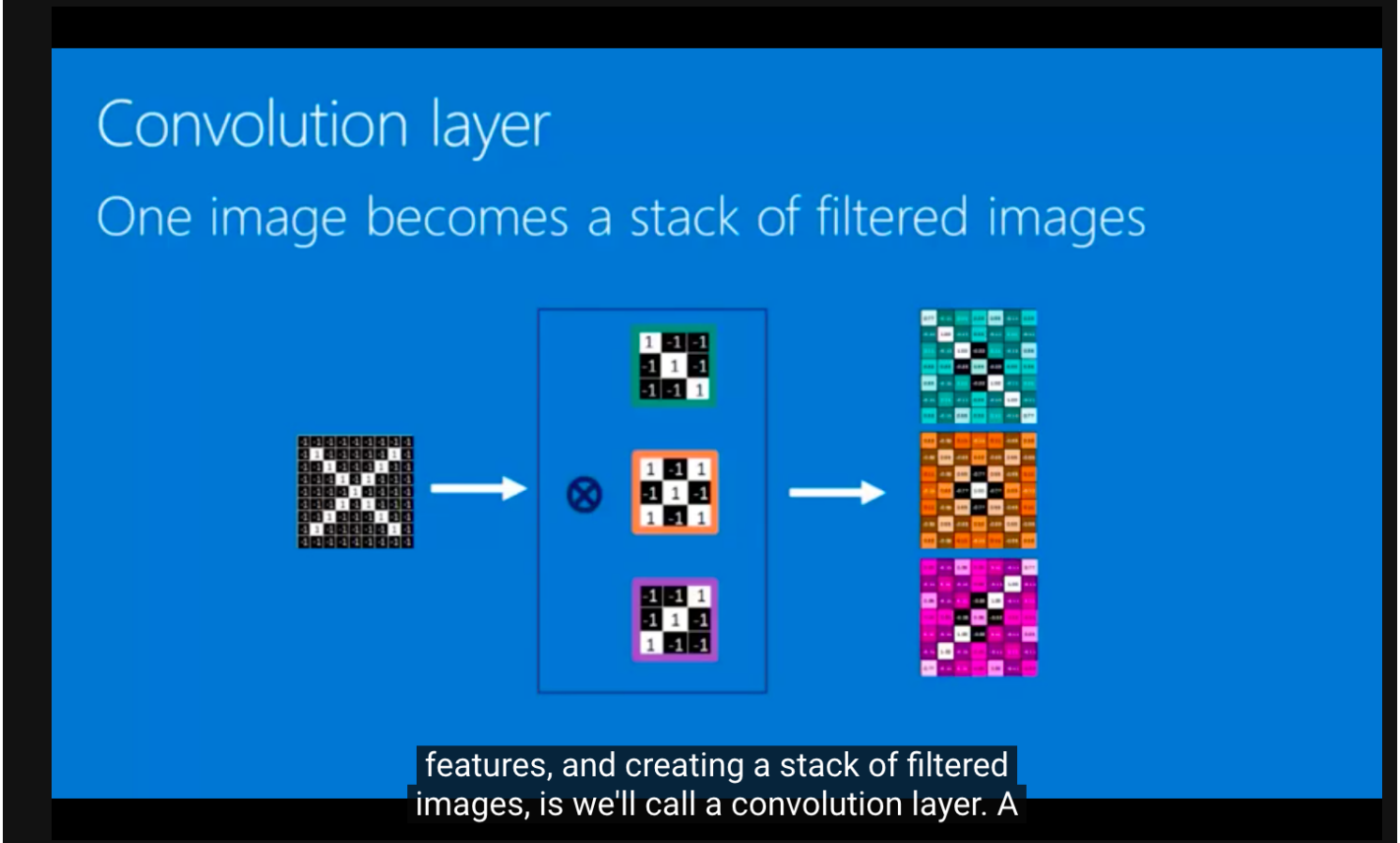

How Convolutional Neural Networks work
Press esc to exit full screen
ดูภายหลัง
แชร์
Filtering: The math behind the match
1 -1 -1
-1 1 -1
-1 -1 1
1 1 1
1 1 1
1 1 1
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
9
= 1
1
6:06 / 26:13
YouTube



1.1) Convolution

- **Convolution = summation of the element products between the filters and filter-sized input image, then divided them by #pixel_of_filter**.
- In depicted speaking, convolution is the filtering process.


How Convolutional Neural Networks work
ดูภายหลัง
แชร์
Convolution: Trying every possible match
8:18 / 26:13
YouTube


Convolution layer
One image becomes a stack of filtered images
features, and creating a stack of filtered
images, is we'll call a convolution layer. A

2) Pooling

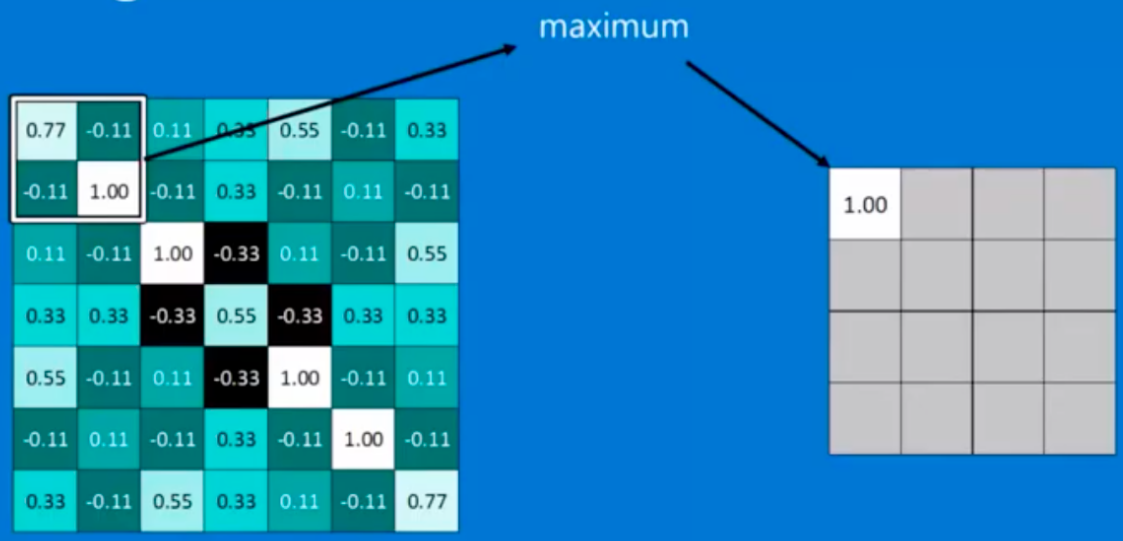
- **Pooling = Extracting values (e.g. max, min, avg) giving smaller image**

Pooling: Shrinking the image stack

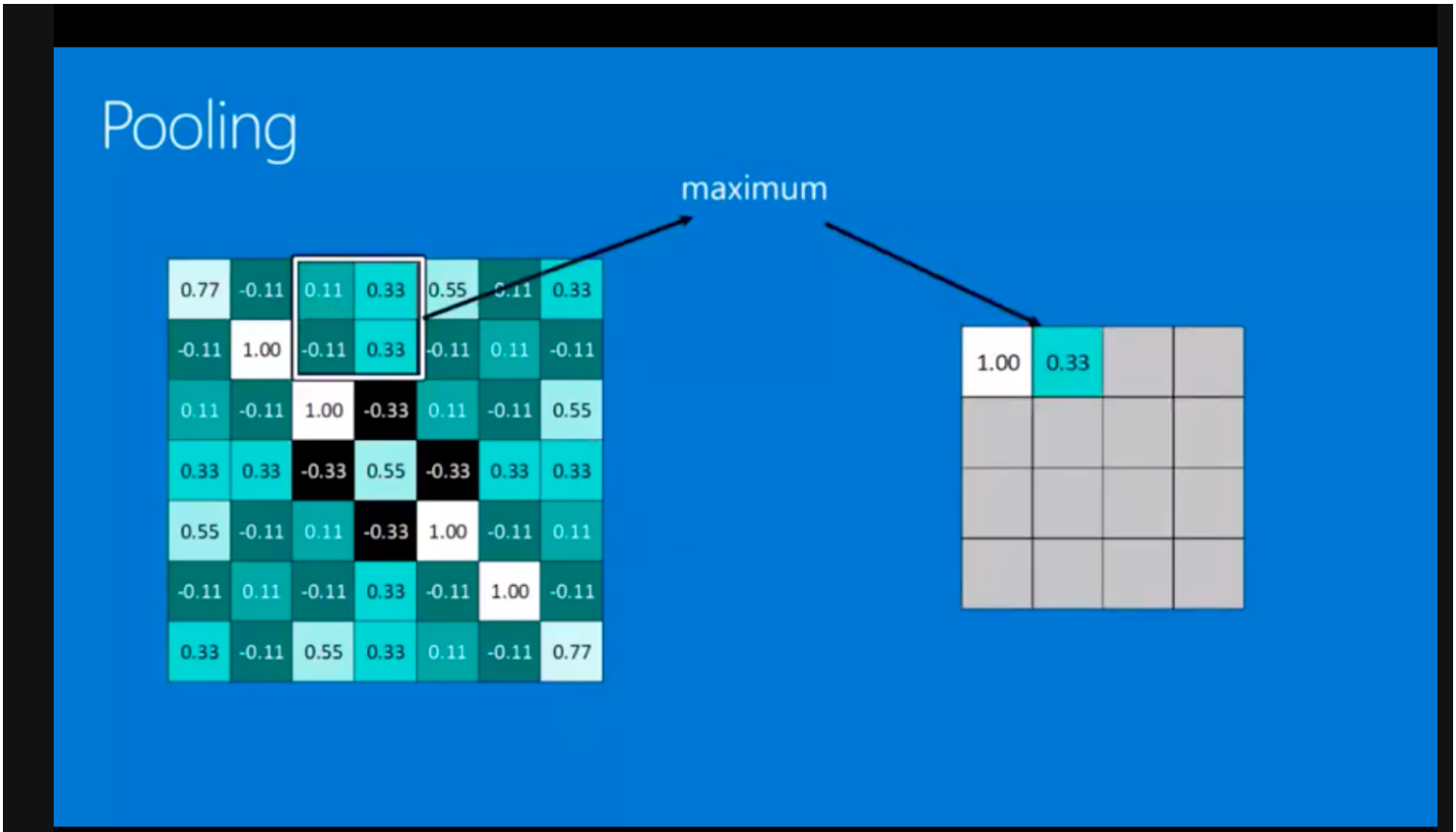
1. Pick a window size (usually 2 or 3).
2. Pick a stride (usually 2).
3. Walk your window across your filtered images.
4. From each window, take the maximum value.

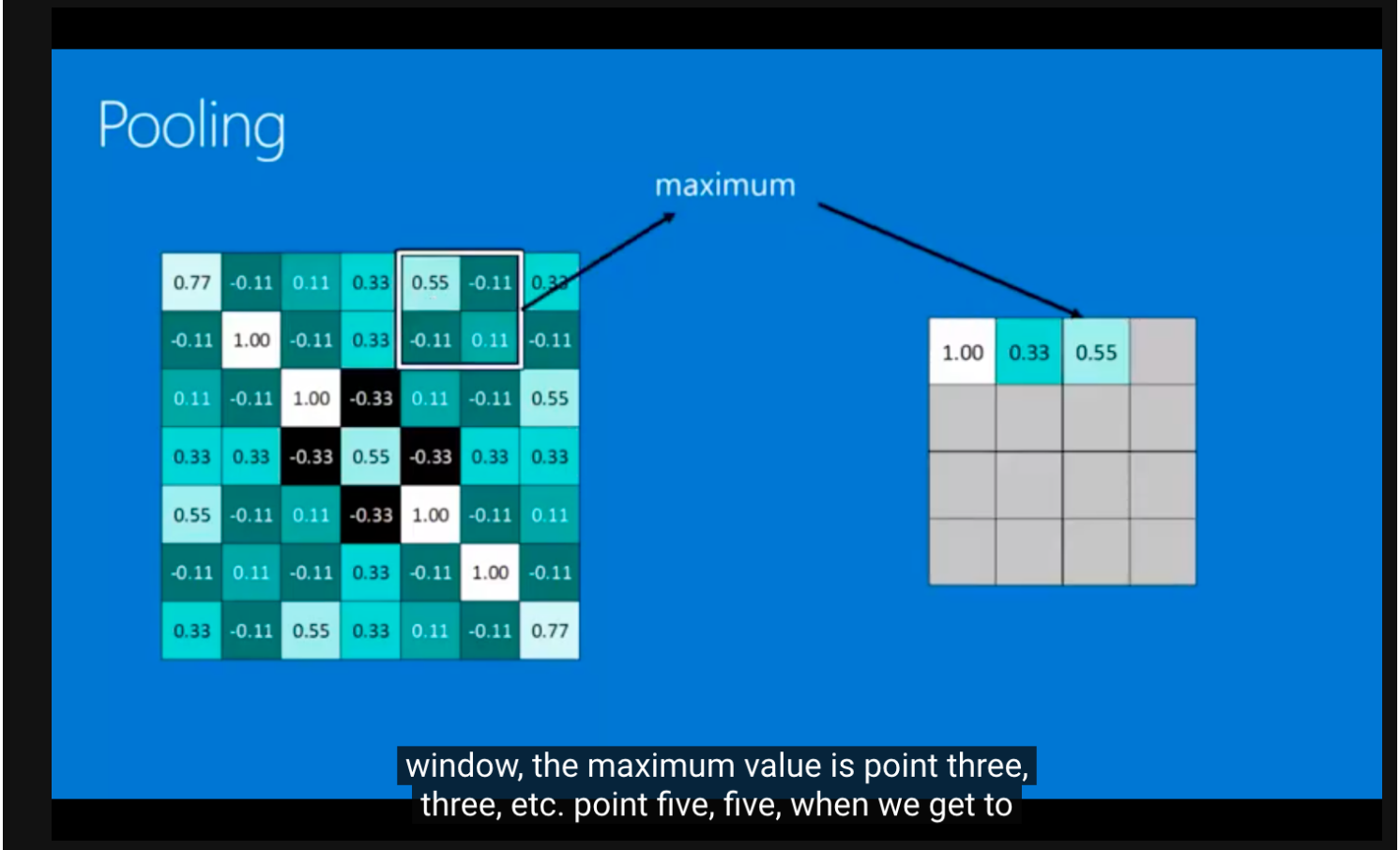
images out as we have filters. So convolution layer is one trick that we have. The next big

Pooling

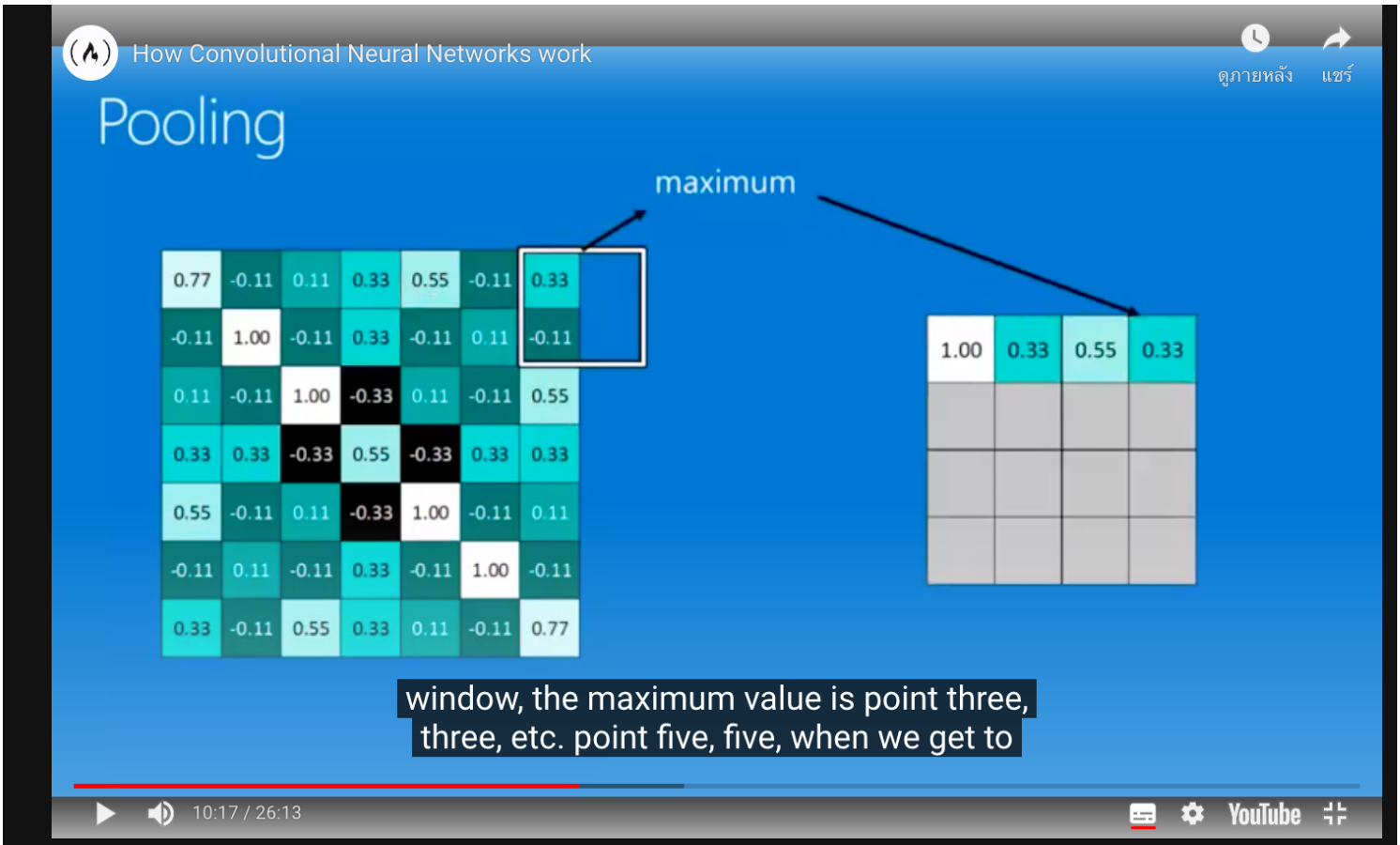


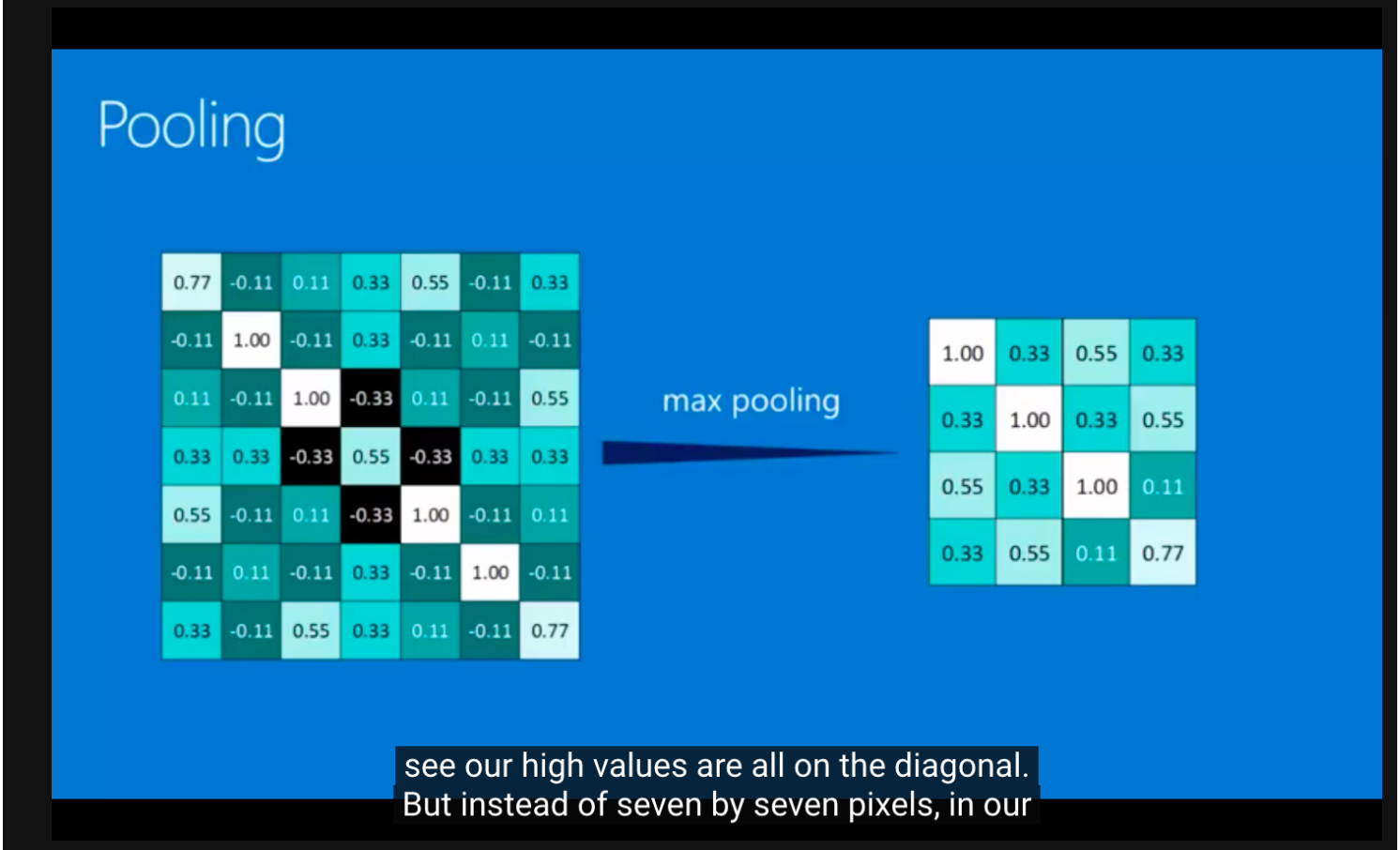
first filtered image. We have our two pixel by two Fix a window, within that pixel, the





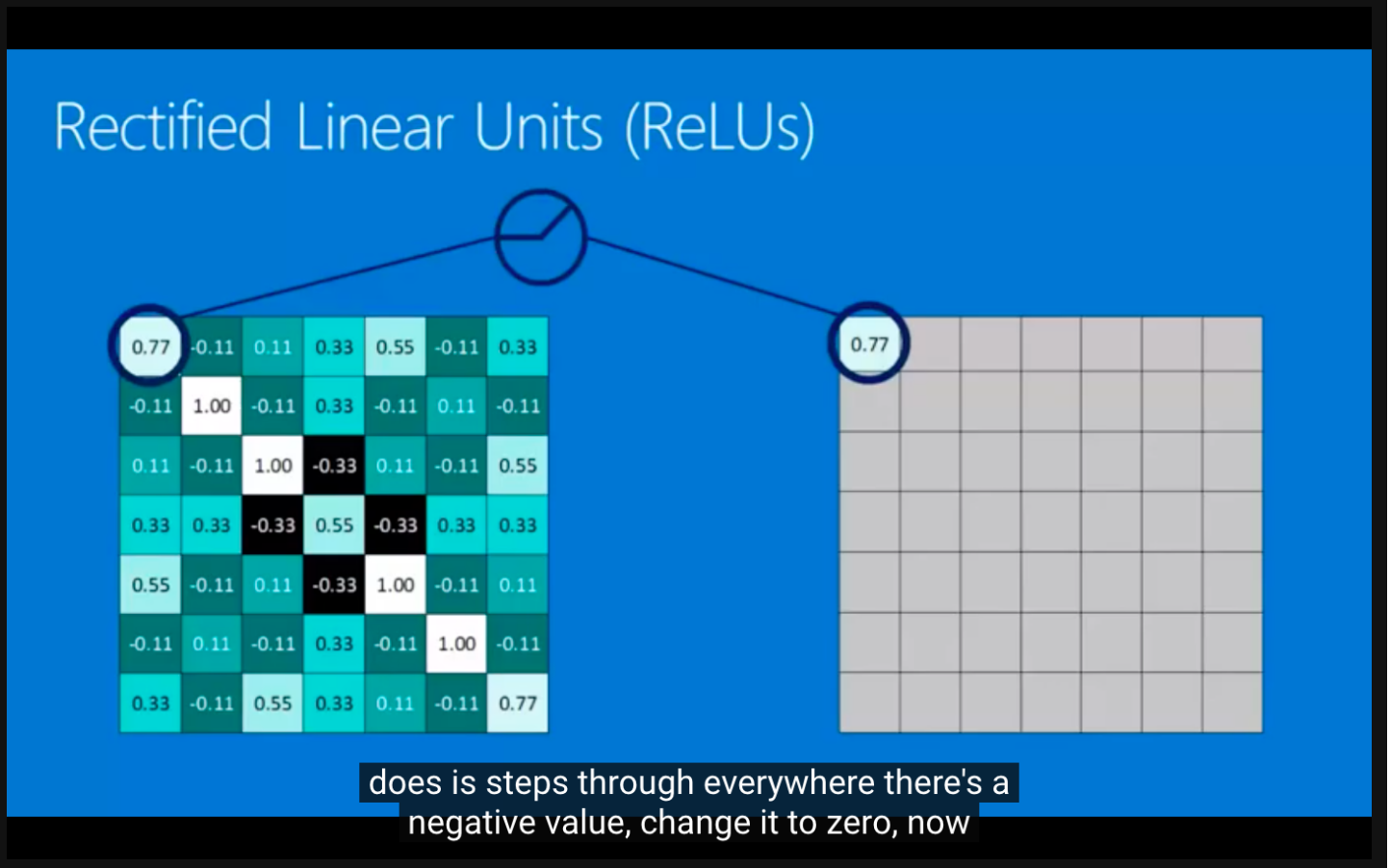
window, the maximum value is point three,
three, etc. point five, five, when we get to

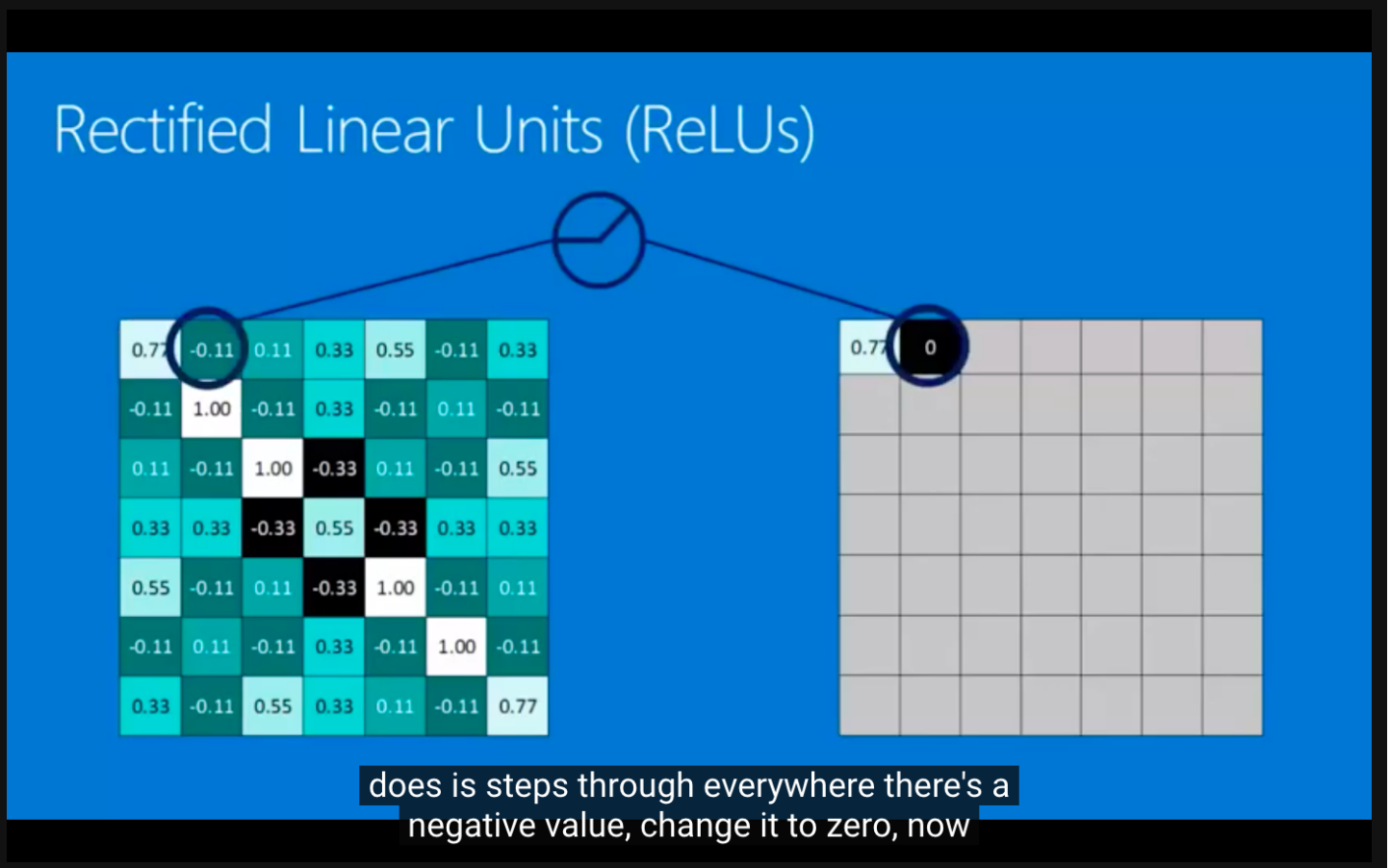

How Convolutional Neural Networks work
ดูภายหลัง
แชร์
Pooling
maximum
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1.00 0.33 0.55 0.33
window, the maximum value is point three,
three, etc. point five, five, when we get to
10:17 / 26:13
YouTube


Pooling
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
max pooling
1.00 0.33 0.55 0.33
0.33 1.00 0.33 0.55
0.55 0.33 1.00 0.11
0.33 0.55 0.11 0.77
see our high values are all on the diagonal.
But instead of seven by seven pixels, in our

3) Rectified Linear Unit (ReLU)

- **ReLU : choose negative values into zero values**





Rectified Linear Units (ReLUs)

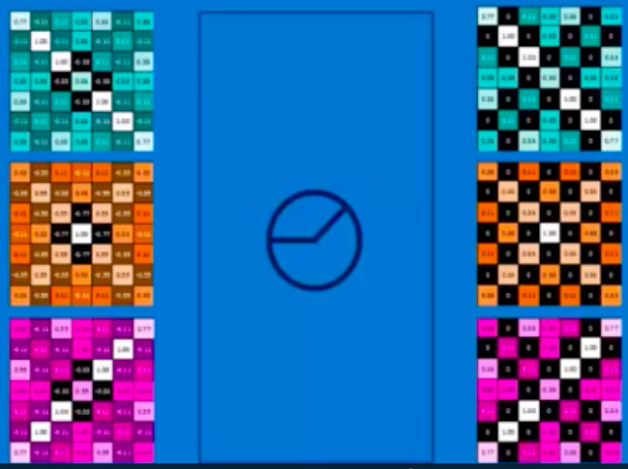
0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	0.77

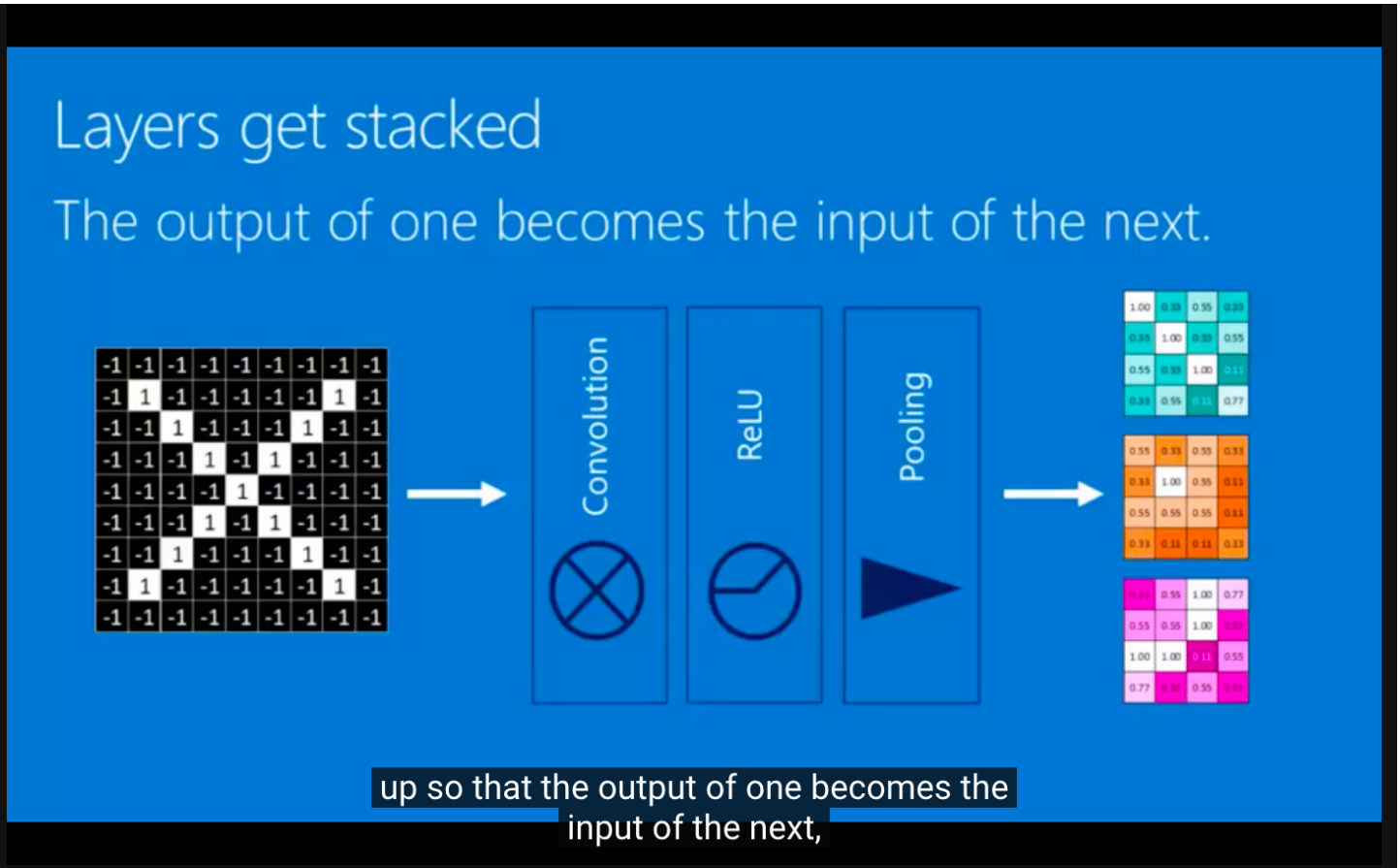
looking image, except there's no negative values, they're just zeros. And we do this

ReLU layer

A stack of images becomes a stack of images with no negative values.

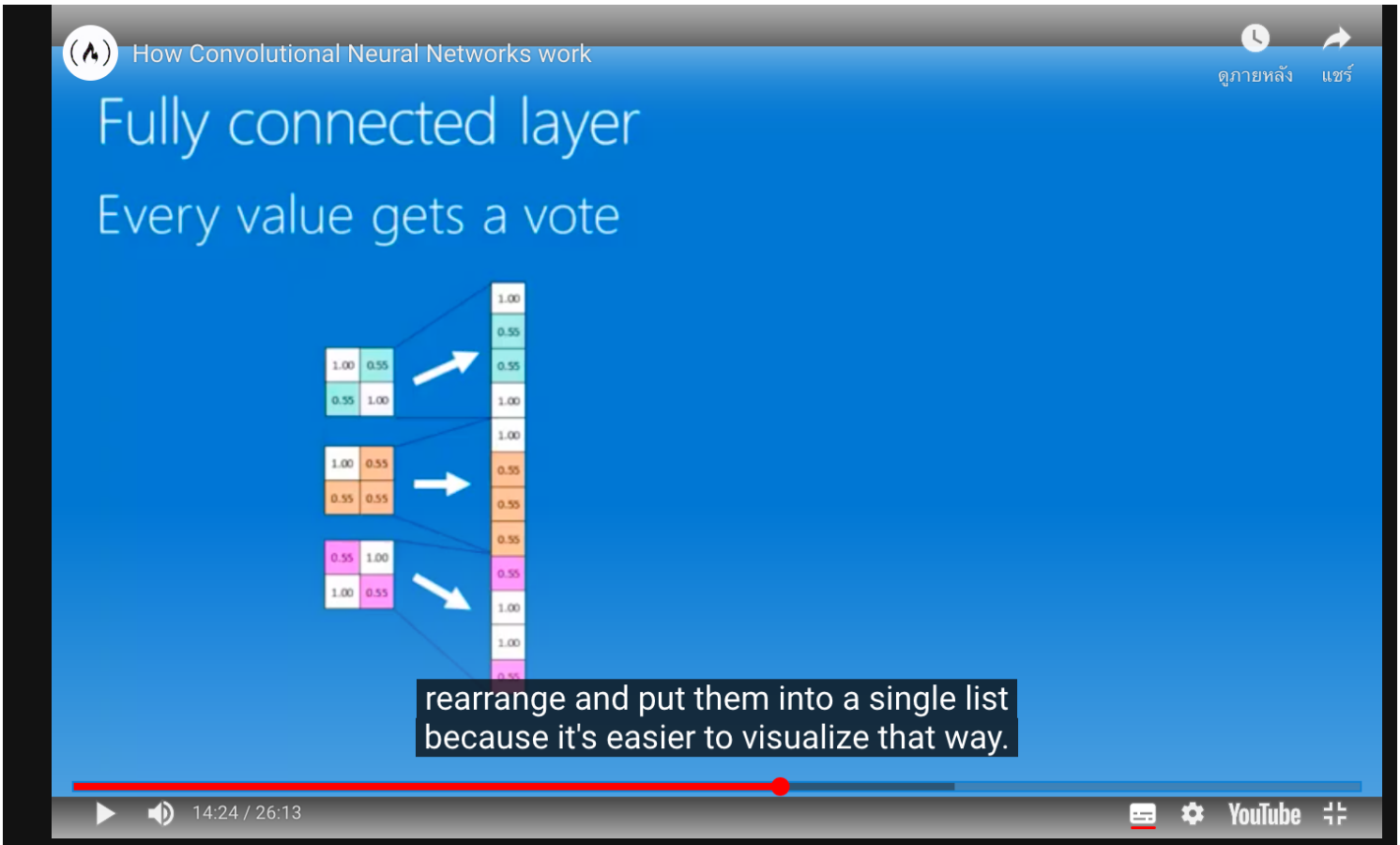


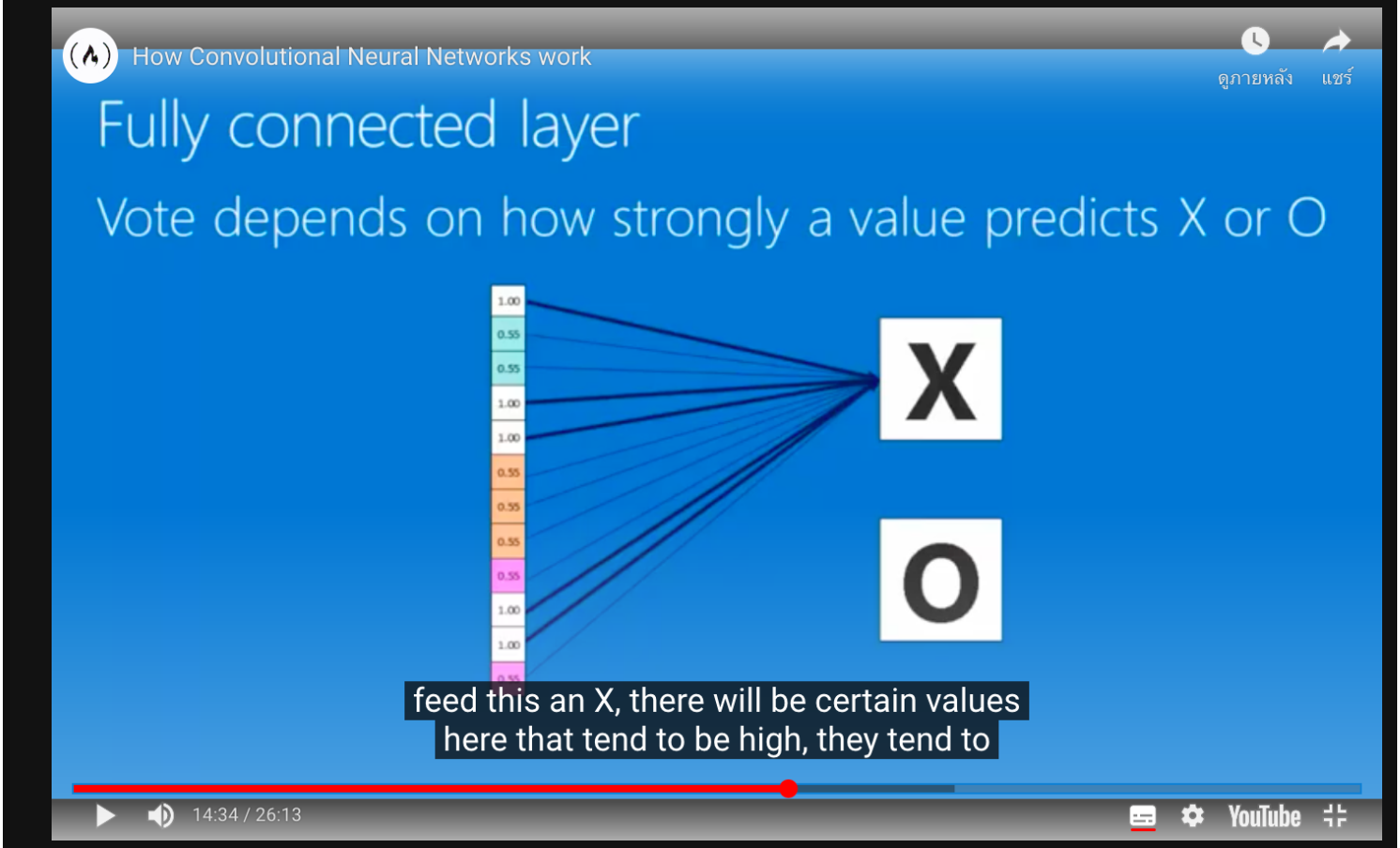
linear unit layer, a stack of images becomes a stack of images with no negative values.

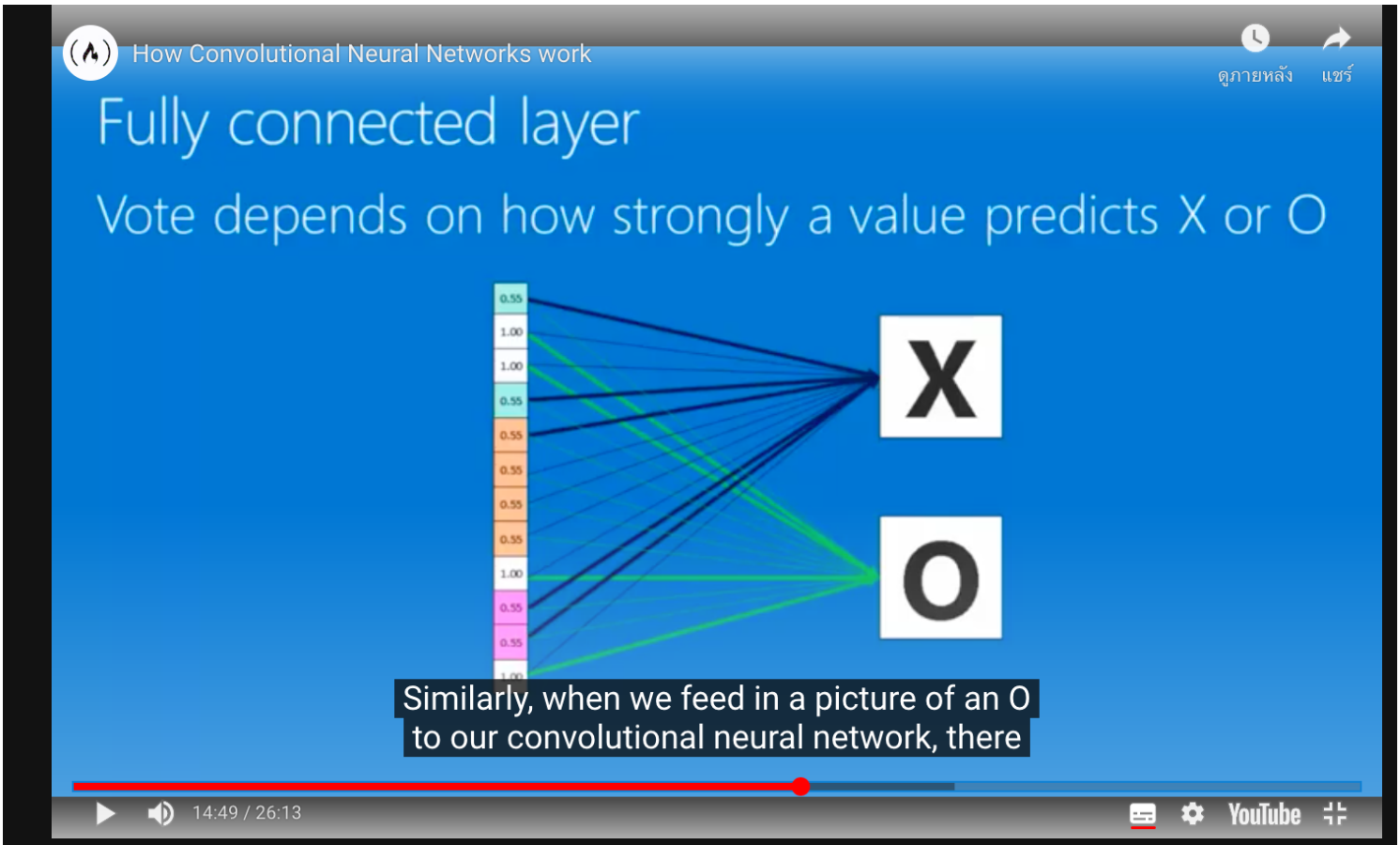


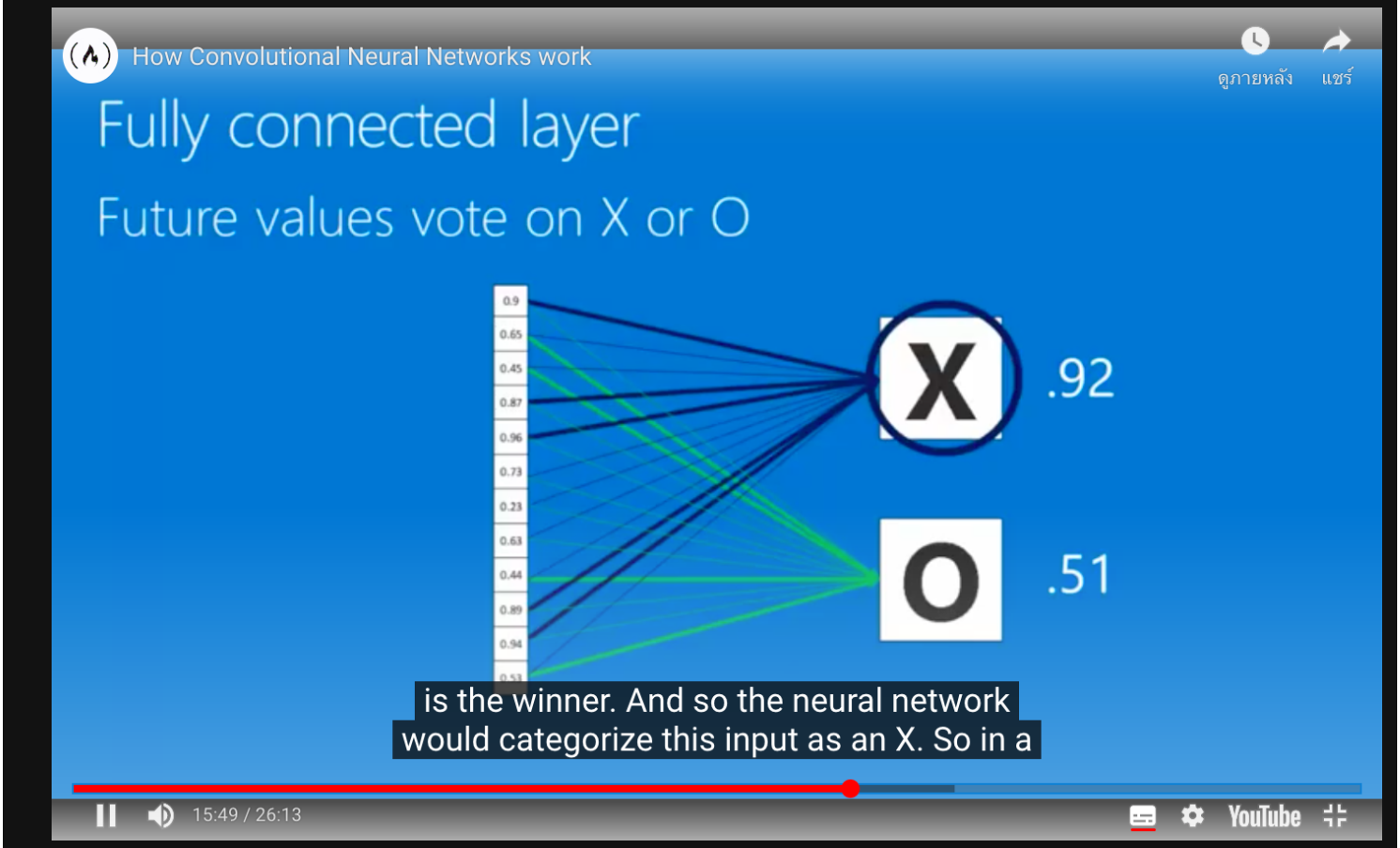
4) Voting : Fully connected Layer

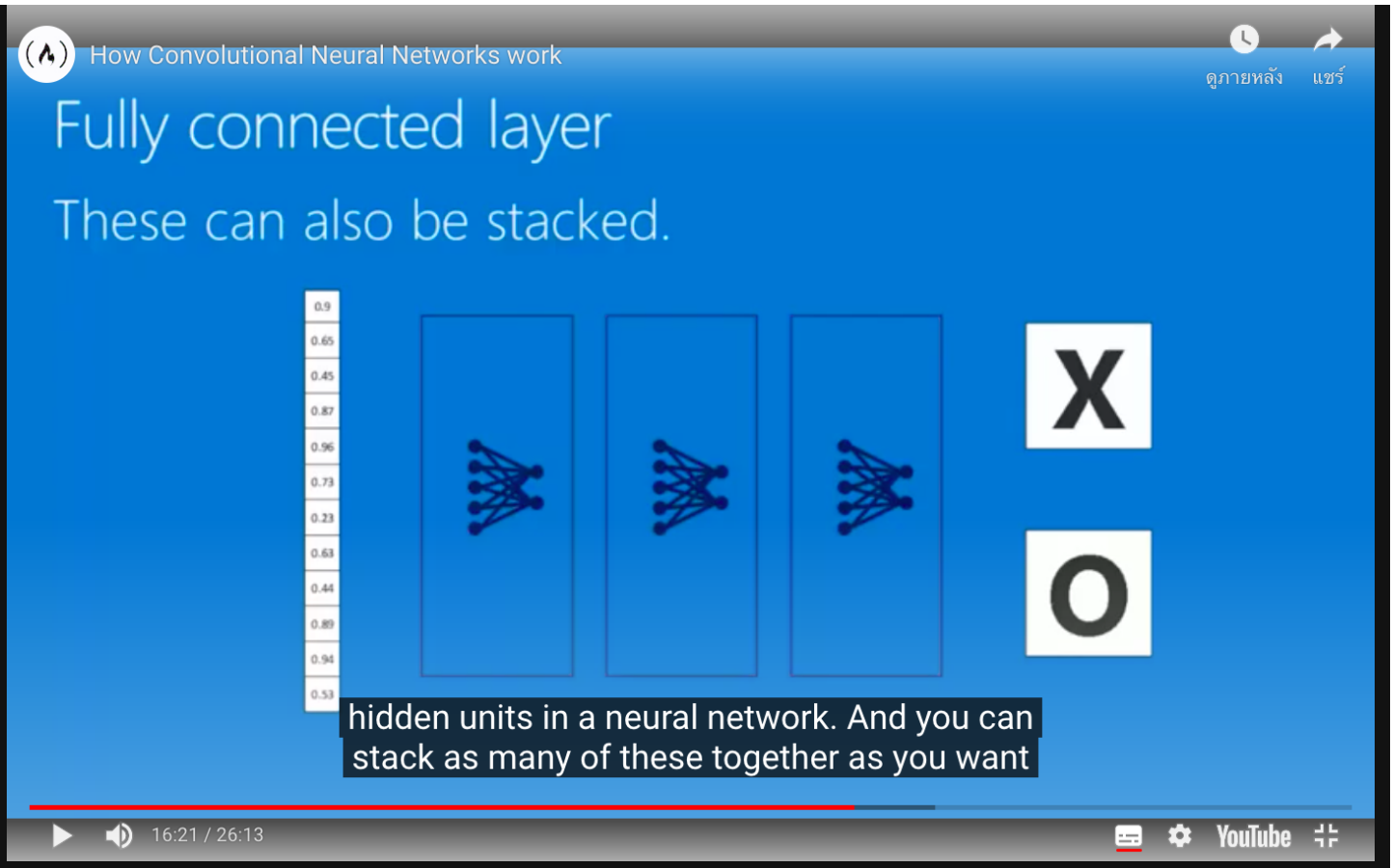
- **Voting : Choose maximum values -> then average them as prediction score**


How Convolutional Neural Networks work
ดูภายหลัง
แชร์
Fully connected layer
Every value gets a vote
rearrange and put them into a single list
because it's easier to visualize that way.
14:24 / 26:13
YouTube

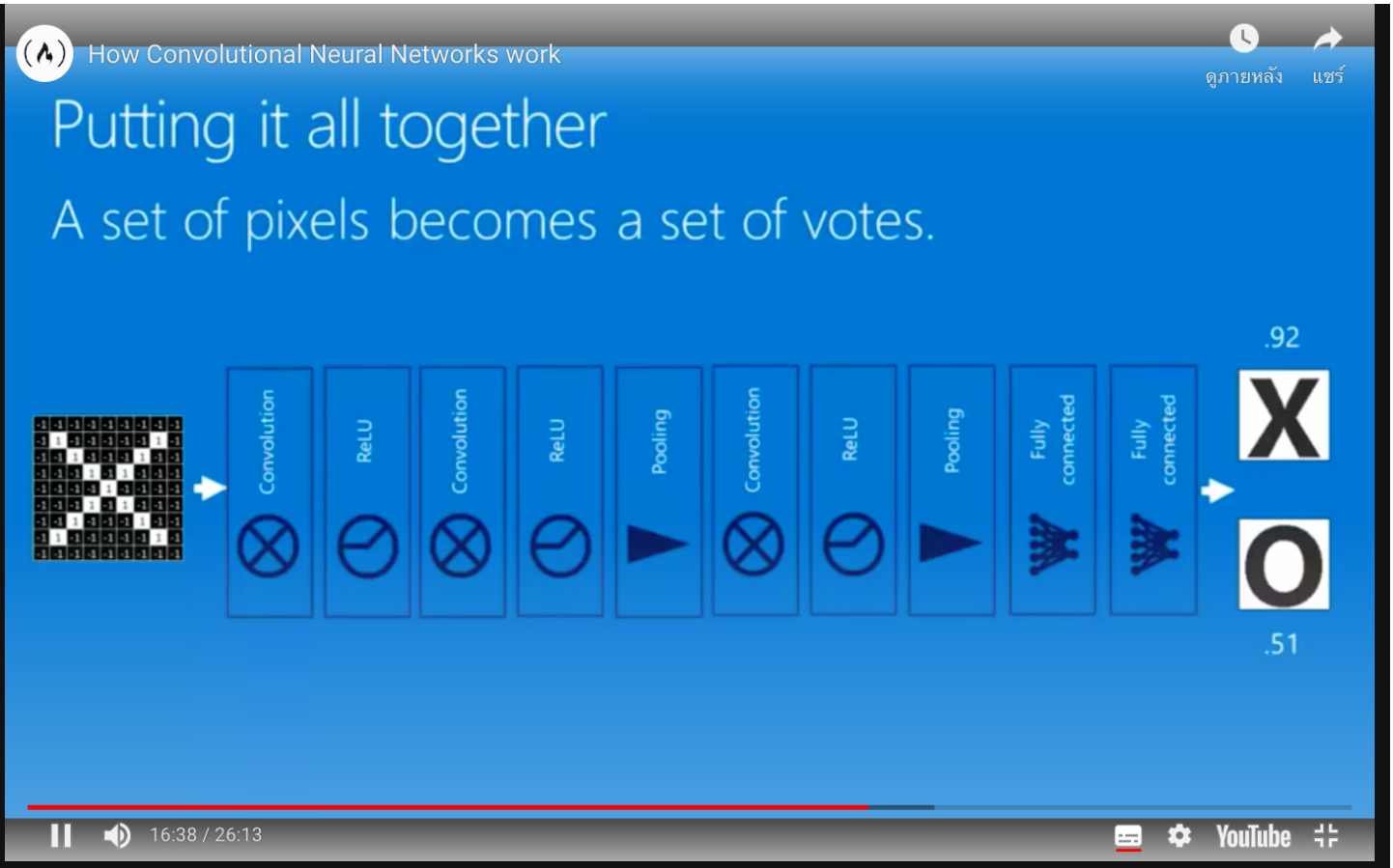

How Convolutional Neural Networks work
ดูภายหลัง
แชร์
Fully connected layer
Vote depends on how strongly a value predicts X or O
X
O
feed this an X, there will be certain values
here that tend to be high, they tend to
14:34 / 26:13
YouTube


How Convolutional Neural Networks work
ดูภายหลัง
แชร์
Fully connected layer
Vote depends on how strongly a value predicts X or O
0.55
1.00
1.00
0.55
0.55
0.55
0.55
0.55
1.00
0.55
0.55
X
O
Similarly, when we feed in a picture of an O
to our convolutional neural network, there
14:49 / 26:13
YouTube


How Convolutional Neural Networks work
ดูภายหลัง
แชร์
Fully connected layer
Future values vote on X or O
0.9
0.65
0.45
0.87
0.96
0.73
0.23
0.63
0.44
0.89
0.94
X
.92
O
.51
is the winner. And so the neural network
would categorize this input as an X. So in a
15:49 / 26:13
YouTube



5) Summary : Putting all layers



Tutorial with Image Dataset

The problem we will consider here is classifying 10 different everyday objects.

The dataset we will use is built into tensorflow and called the **CIFAR Image Dataset.** . It contains 60,000 32x32 color images with 6000 images of each class.

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

```
#  LOAD AND SPLIT DATASET
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_da

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
```

```
                'dog', 'frog', 'horse', 'ship', 'truck']

# Let's look at a one image
IMG_INDEX = 7  # change this to look at other images

plt.imshow(train_images[IMG_INDEX] ,cmap=plt.cm.binary)
plt.xlabel(class_names[train_labels[IMG_INDEX][0]])
plt.show()

```

Build the Convolutional Base

Here we will extract the features from the layer. First, use the **sequential neural network** model.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Layer 1

The input shape of our data will be 32, 32, 3 and we will process 32 filters of size 3x3 over our input data. We will also **apply the activation function `relu` to the output of each convolution operation**.

Layer 2

This layer will perform the max pooling operation using 2x2 samples and a stride of 2.

Other Layers

The next set of layers do very similar things but take as input the feature map from the previous layer. They also increase the frequency of filters from 32 to 64. We can do this as our data shrinks in spatial dimensions as it passed through the layers, meaning we can afford (computationally) to add more depth

```
model.summary()  # let's have a look at our model so far
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

=================================================================
Total params: 56,320
Trainable params: 56,320
Non-trainable params: 0
_________________________________________________________________
```

Adding Dense Layers

Take the convolutional base (extracted features) and add a way to classify them.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

We can see that the **flatten layer changes the shape of our data so that we can feed it to the 64-node dense layer**, followed by the final output layer of 10 neurons (one for each class).

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
```

```
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

Training the model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 [==============================] - 83s 52ms/step - loss: 1.5442 - accu
Epoch 2/10
1563/1563 [==============================] - 74s 48ms/step - loss: 1.1629 - accu
Epoch 3/10
1563/1563 [==============================] - 73s 47ms/step - loss: 0.9866 - accu
Epoch 4/10
1563/1563 [==============================] - 75s 48ms/step - loss: 0.8860 - accu
Epoch 5/10
1563/1563 [==============================] - 74s 48ms/step - loss: 0.8203 - accu
Epoch 6/10
1563/1563 [==============================] - 74s 47ms/step - loss: 0.7604 - accu
Epoch 7/10
1563/1563 [==============================] - 75s 48ms/step - loss: 0.7120 - accu
Epoch 8/10
1563/1563 [==============================] - 75s 48ms/step - loss: 0.6708 - accu
Epoch 9/10
1563/1563 [==============================] - 77s 49ms/step - loss: 0.6287 - accu
```

```
Epoch 10/10
1563/1563 [==============================] - 75s 48ms/step - loss: 0.5961 - accu
```

Evaluating the Model

We can determine how well the model performed by looking at it's performance on the test data set. And run predictions as we did before.

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
print(test_acc)
```

```
313/313 - 4s - loss: 0.8859 - accuracy: 0.7013 - 4s/epoch - 12ms/step
0.7013000249862671
```

```
predictions = model.predict(test_images)
print(predictions[3])
```

```
[ 5.714036   -0.22584799  0.37605742 -0.58427835 -0.5692592  -2.3296711
 -3.6222126  -2.274637    3.8324265  -0.40380895]
```

```
import numpy as np

def pred(n):
  print(class_names[np.argmax(predictions[n])])
  print(test_labels[n])

pred(9)
```

```
automobile
[1]
```

```
#The model should return what it's prediction of the pixelated image is.
#The matplotlib portion of the code block will return a graph of the image in quest

def predpic(n):
  print('The model has guessed: {}'.format(class_names[np.argmax(predictions[n])]))
  plt.figure()
  plt.imshow(test_images[n]) #enter image index
  plt.colorbar()
  plt.grid(False)
  plt.show()

predpic(9)
```

```
The model has guessed: automobile
```

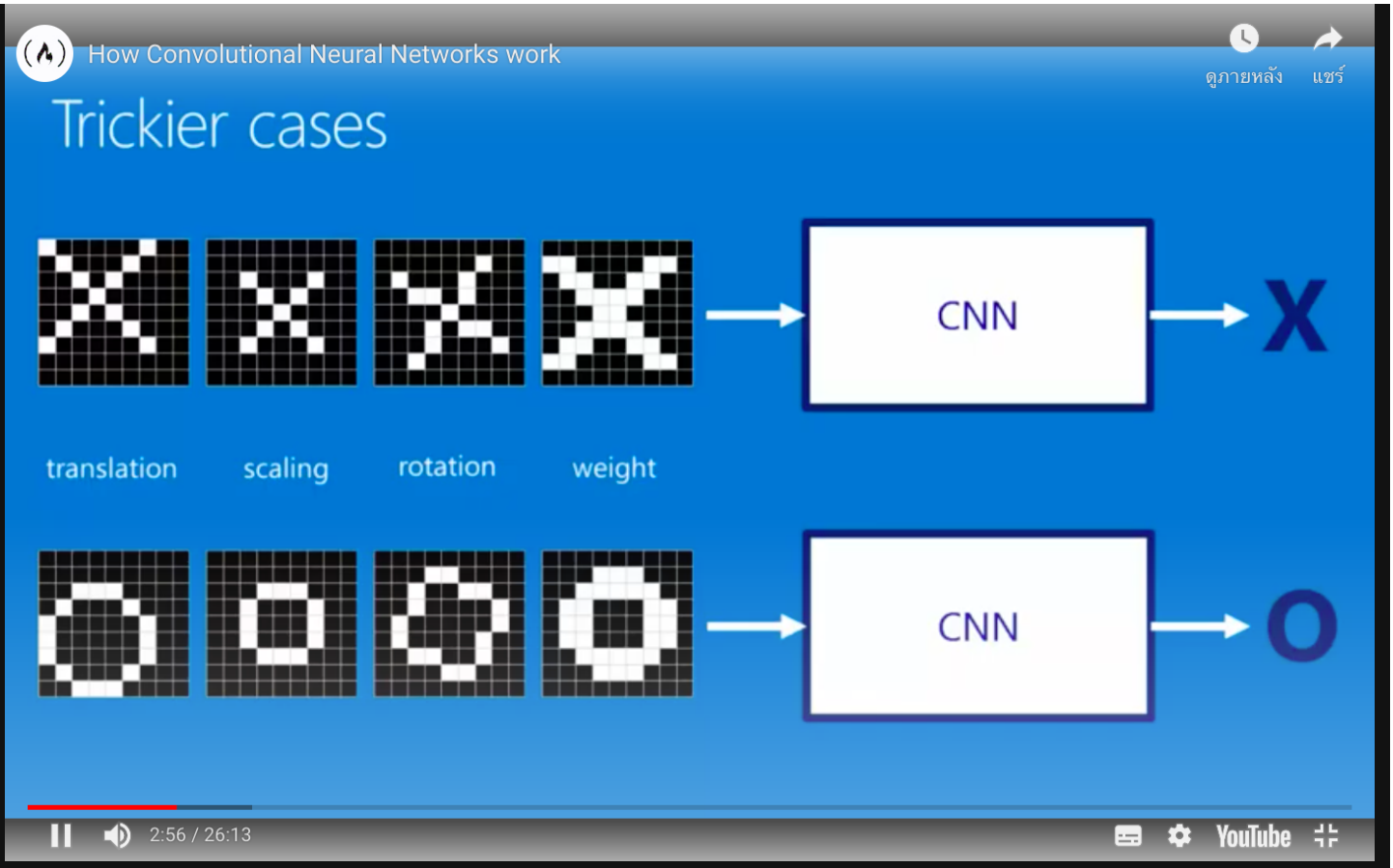
Working with Small Datasets

Our amount of data is too small to train. A few techniques if we're working with small data sets and **need to expand the data.**

Data Augmentation

Data Augmentation: **Performing random transformations on our images so that our model can generalize better**.

- These transformations can be things like **compressions, rotations, stretches and even color changes**.
- This will allow us **to avoid overfitting** and **create a larger dataset from a smaller one** we can use a technique called the **data augmentation.**
- **In other words, we can take one image and pass different versions (flipped, stretched, rotated etc) of it through the model and augment it multiple times**.




```
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

# creates a data generator object that transforms images
datagen = ImageDataGenerator(
rotation_range=40,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.2,
zoom_range=0.2,
horizontal_flip=True,
fill_mode='nearest')

# pick an image to transform
test_img = test_images[9]
img = image.img_to_array(test_img)  # convert image to numpy arry
img = img.reshape((1,) + img.shape)  # reshape image

i = 0

for batch in datagen.flow(img, save_prefix='test', save_format='jpeg'):  # this loo
    plt.figure(i)
    plot = plt.imshow(image.img_to_array(batch[0]))
```

```
    i += 1
    if i > 4:  # show 4 images
        break

plt.show()
```

Pretrained Models

Here we will use part of an existing model and then "fine tune" it for the use of our own data.

Fine Tuning

Tweaking the final layers in our convolutional base to work better for our specific problem. This involves not touching or retraining the earlier layers in our convolutional base **but only adjusting the final few layers**.

Using a Pretrained Model

In this section we will combine the tecniques we learned above and use a pretrained model and fine tuning to classify images of dogs and cats using a small dataset.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
keras = tf.keras
```

Dataset Example

We will load the `cats_vs_dogs` dataset from the module tensorflow_datatsets.

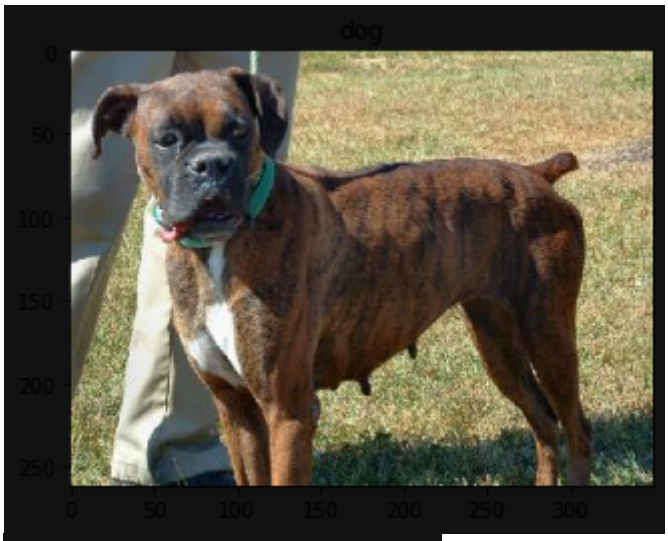
This dataset contains (image, label) pairs where images have different dimensions and 3 color channels.

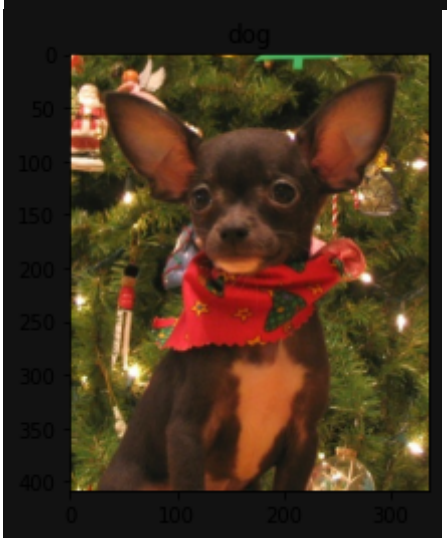
```
import tensorflow_datasets as tfds
tfds.disable_progress_bar()

# split the data manually into 80% training, 10% testing, 10% validation
(raw_train, raw_validation, raw_test), metadata = tfds.load(
    'cats_vs_dogs',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
```

```
    as_supervised=True,
)
```

```
Downloading and preparing dataset 786.68 MiB (download: 786.68 MiB, generated: U
WARNING:absl:1738 images were corrupted and were skipped
Dataset cats_vs_dogs downloaded and prepared to ~/tensorflow_datasets/cats_vs_do
```

```
get_label_name = metadata.features['label'].int2str  # creates a function object th

# display 2 images from the dataset
for image, label in raw_train.take(2):
  plt.figure()
  plt.imshow(image)
  plt.title(get_label_name(label))
```





Data Preprocessing

Since the sizes of our images are all different, we need to convert them all to the same size. We can create a function that will compress the size of the image for us below.

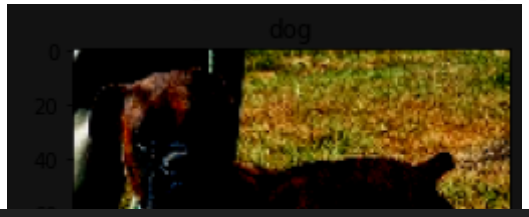
```
IMG_SIZE = 160 # All images will be resized to 160x160

def format_example(image, label):
  """
  returns an image that is reshaped to IMG_SIZE
  """
  image = tf.cast(image, tf.float32) #cast meanst to convert (to a float in this ca
  image = (image/127.5) - 1
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  return image, label
```

Now we can apply this function to all our images using `.map()`.

```
train = raw_train.map(format_example)
validation = raw_validation.map(format_example)
test = raw_test.map(format_example)
```

```
for image, label in train.take(2):
  plt.figure()
  plt.imshow(image)
  plt.title(get_label_name(label))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
```




```
#No clue what this means at this moment
BATCH_SIZE = 32
SHUFFLE_BUFFER_SIZE = 1000

train_batches = train.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
validation_batches = validation.batch(BATCH_SIZE)
test_batches = test.batch(BATCH_SIZE)
```

```
#Looking at the shape of the new image
for img, label in raw_train.take(2):
  print("Original shape:", img.shape)

for img, label in train.take(2):
  print("New shape:", img.shape)
```

```
Original shape: (262, 350, 3)
Original shape: (409, 336, 3)
New shape: (160, 160, 3)
New shape: (160, 160, 3)
```

Picking a Pretrained Model

The model we are going to **use as the convolutional base for our model is the *MobileNet V2*** developed at Google. This model is trained on 1.4 million images and has 1000 different classes.

We want to use this model but only its convolutional base. So, when we load in the model, we'll specify that we don't want to load the top (classification) layer. We'll tell the model what input shape to expect and to use the predetermined weights from imagenet (Googles dataset).

```
IMG_SHAPE = (IMG_SIZE, IMG_SIZE, 3)

# Create the base model from the pre-trained model MobileNet V2
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicatio
```

```
9412608/9406464 [==============================] - 0s 0us/step
9420800/9406464 [==============================] - 0s 0us/step
```

```
base_model.summary()
```

```
Model: "mobilenetv2_1.00_160"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 160, 160, 3  0           []
                                )]

 Conv1 (Conv2D)                 (None, 80, 80, 32)   864         ['input_1[0][0]

 bn_Conv1 (BatchNormalization)  (None, 80, 80, 32)   128         ['Conv1[0][0]']

 Conv1_relu (ReLU)              (None, 80, 80, 32)   0           ['bn_Conv1[0][0

 expanded_conv_depthwise (Depth  (None, 80, 80, 32)  288         ['Conv1_relu[0]
 wiseConv2D)

 expanded_conv_depthwise_BN (Ba  (None, 80, 80, 32)  128         ['expanded_conv_
 tchNormalization)

 expanded_conv_depthwise_relu (  (None, 80, 80, 32)  0           ['expanded_conv_
 ReLU)                                                           ]']

 expanded_conv_project (Conv2D)  (None, 80, 80, 16)  512         ['expanded_conv_
                                                                 [0]']

 expanded_conv_project_BN (Batc  (None, 80, 80, 16)  64          ['expanded_conv_
 hNormalization)

 block_1_expand (Conv2D)        (None, 80, 80, 96)   1536        ['expanded_conv_
                                                                 ]

 block_1_expand_BN (BatchNormal  (None, 80, 80, 96)  384         ['block_1_expan
 ization)

 block_1_expand_relu (ReLU)     (None, 80, 80, 96)   0           ['block_1_expan

 block_1_pad (ZeroPadding2D)    (None, 81, 81, 96)   0           ['block_1_expan

 block_1_depthwise (DepthwiseCo  (None, 40, 40, 96)  864         ['block_1_pad[0
 nv2D)

 block_1_depthwise_BN (BatchNor  (None, 40, 40, 96)  384         ['block_1_depth
 malization)

 block_1_depthwise_relu (ReLU)  (None, 40, 40, 96)   0           ['block_1_depth

 block_1_project (Conv2D)       (None, 40, 40, 24)   2304        ['block_1_depth

 block_1_project_BN (BatchNorma  (None, 40, 40, 24)  96          ['block_1_proje
 lization)
```

```
block_2_expand (Conv2D)        (None, 40, 40, 144)  3456        ['block_1_proje
block_2_expand_BN (BatchNormal  (None, 40, 40, 144)  576        ['block_2_expan
ization)
block_2_expand_relu (ReLU)     (None, 40, 40, 144)  0           ['block_2_expan
```

At this point this base_model will simply output a shape (32, 5, 5, 1280) tensor (from the last row in the summary) that is a feature extraction from our original (1, 160, 160, 3) image. The 32 means that we have 32 layers of differnt filters/features.

```
for image, _ in train_batches.take(1): # He went a little fast will look further in
   pass

feature_batch = base_model(image)
print(feature_batch.shape)
```

```
(32, 5, 5, 1280)
```

Freezing the Base

The freezing term refers to disabling the training property of a layer.

- It simply means **we won't make any changes to the weights of any layers** that are frozen **during training**.
- **This is important as we don't want to change the convolutional base that already has learned weights**.

```
base_model.trainable = False
```

```
base_model.summary()
```

```
Model: "mobilenetv2_1.00_160"
__________________________________________________________________________________
 Layer (type)                Output Shape              Param #     Connected to
==================================================================================
 input_1 (InputLayer)        [(None, 160, 160, 3       0           []
                             )]

 Conv1 (Conv2D)              (None, 80, 80, 32)        864         ['input_1[0][0]

 bn_Conv1 (BatchNormalization)  (None, 80, 80, 32)     128         ['Conv1[0][0]']

 Conv1_relu (ReLU)           (None, 80, 80, 32)        0           ['bn_Conv1[0][0
```

```
expanded_conv_depthwise (Depth  (None, 80, 80, 32)  288         ['Conv1_relu[0]
wiseConv2D)

expanded_conv_depthwise_BN (Ba  (None, 80, 80, 32)  128         ['expanded_conv_
tchNormalization)

expanded_conv_depthwise_relu (  (None, 80, 80, 32)  0           ['expanded_conv_
ReLU)                                                           ]']

expanded_conv_project (Conv2D)  (None, 80, 80, 16)  512         ['expanded_conv_
                                                                [0]']

expanded_conv_project_BN (Batc  (None, 80, 80, 16)  64          ['expanded_conv_
hNormalization)

block_1_expand (Conv2D)        (None, 80, 80, 96)   1536        ['expanded_conv_
                                                                ]

block_1_expand_BN (BatchNormal  (None, 80, 80, 96)  384         ['block_1_expan
ization)

block_1_expand_relu (ReLU)     (None, 80, 80, 96)   0           ['block_1_expan

block_1_pad (ZeroPadding2D)    (None, 81, 81, 96)   0           ['block_1_expan

block_1_depthwise (DepthwiseCo  (None, 40, 40, 96)  864         ['block_1_pad[0
nv2D)

block_1_depthwise_BN (BatchNor  (None, 40, 40, 96)  384         ['block_1_depth
malization)

block_1_depthwise_relu (ReLU)  (None, 40, 40, 96)   0           ['block_1_depth

block_1_project (Conv2D)       (None, 40, 40, 24)   2304        ['block_1_depth

block_1_project_BN (BatchNorma  (None, 40, 40, 24)  96          ['block_1_proje
lization)

block_2_expand (Conv2D)        (None, 40, 40, 144)  3456        ['block_1_proje

block_2_expand_BN (BatchNormal  (None, 40, 40, 144)  576        ['block_2_expan
ization)

block_2_expand_relu (ReLU)     (None, 40, 40, 144)  0           ['block_2_expan
```

Adding our Classifier

Now that we have our base layer setup, we can **add the classifier**. Instead of flattening the feature map of the base layer we will **use a global average pooling layer that will average the entire 5x5 area of each 2D feature map** and return to us a single 1280 element vector per filter.

```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
```

Finally, we will add the prediction layer that will be a single dense neuron. We can do this because we only have two classes to predict for.

```
prediction_layer = keras.layers.Dense(1)  # Flatten into one dense layer
```

Now we will combine these layers together in a model.

```
model = tf.keras.Sequential([
  base_model,
  global_average_layer,
  prediction_layer
])
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_160 (Funct  (None, 5, 5, 1280)       2257984
 ional)

 global_average_pooling2d (G  (None, 1280)             0
 lobalAveragePooling2D)

 dense_2 (Dense)             (None, 1)                 1281

=================================================================
Total params: 2,259,265
Trainable params: 1,281
Non-trainable params: 2,257,984
_________________________________________________________________
```

We have only **1,281 Trainable Parameters** because we only have 1280 connections from the `global_average_pooling2d` layer to the `dense_2` layer and 1 Bias

Training the Model

```
base_learning_rate = 0.0001 #How much are we allowed to modify the network, it's lo
model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=base_learning_rate),
```

```
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/rmsprop.py:130: UserWa
  super(RMSprop, self).__init__(name, **kwargs)
```

```
# We can evaluate the model right now to see how it does before training it on our
initial_epochs = 3
validation_steps=20

loss0,accuracy0 = model.evaluate(validation_batches, steps = validation_steps)

```

```
20/20 [==============================] - 12s 501ms/step - loss: 0.9087 - accurac
```

```
# Now we can train it on our images
history = model.fit(train_batches,
                    epochs=initial_epochs,
                    validation_data=validation_batches)

acc = history.history['accuracy']
print(acc)
```

```
Epoch 1/3
582/582 [==============================] - 334s 565ms/step - loss: 0.2408 - accu
Epoch 2/3
582/582 [==============================] - 328s 560ms/step - loss: 0.0747 - accu
Epoch 3/3
582/582 [==============================] - 324s 554ms/step - loss: 0.0589 - accu
[0.8857066035270691, 0.972864031791687, 0.9779151082038879]
```

```
model.save("dogs_vs_cats.h5")  # we can save the model and reload it at anytime in
new_model = tf.keras.models.load_model('dogs_vs_cats.h5')
#from here we can predict like we did for the previous modules
```

Getting an accuracy of a close to 92 or 93%, is pretty good.

Considering the fact that all we did was using original layer, like base layer that classified up to 1000 different images (which is so general), and applied that just to cats and dogs by adding our dense layer classifier on top.

Object Detection

One of the CNN application is that we can perform object detection and recognition with tensorflow.

https://github.com/tensorflow/models/tree/master/research/object_detection

```
1
```