Table of Contents

[Requirements 1](#_Toc86132377)

[References 1](#_Toc86132378)

[Convolutional Neural Networks (CNN’s) 1](#_Toc86132379)

[Common Filters (Kernels) 3](#_Toc86132380)

[Stride 3](#_Toc86132381)

[Padding 4](#_Toc86132382)

[Pool Size 5](#_Toc86132383)

[Applying Multiple Filters 5](#_Toc86132384)

[Increasing Filters in Each Consecutive Layer 5](#_Toc86132385)

[Kernel Size 9](#_Toc86132386)

[Filters 9](#_Toc86132387)

[Strides 9](#_Toc86132388)

[Early Stopping and Saving the Best Model 13](#_Toc86132389)

## Requirements

This was implemented using **tensorflow 2.4** and **keras 2.4.3**.

On Windows I had to run PyCharm as Administrator.

On my Mac I had to install open CV when the CV2 module was missing. The installation required admin status: **sudo pip install opencv-python**

## Introduction

The goal of this document is to provide a light introduction to convolutional neural networks. Tuning them can be a challenge so the focus will be more on the general terminology instead.

## References

<https://www.analyticsvidhya.com/blog/2017/06/architecture-of-convolutional-neural-networks-simplified-demystified/>

<https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148#:~:text=Strides,with%20a%20stride%20of%202>.

## Convolutional Neural Networks (CNN’s)

Convolutional Neural Networks, or CNNs for short, are a type of network designed for image input. They are comprised of models with convolutional layers which extract and transform features into simpler pooled elements. The pooled features are then flattened into arrays which are fed through a neural network for classification.

Figure 1: Feature Pooling and Classification Activities of a Convolutional Neural Network

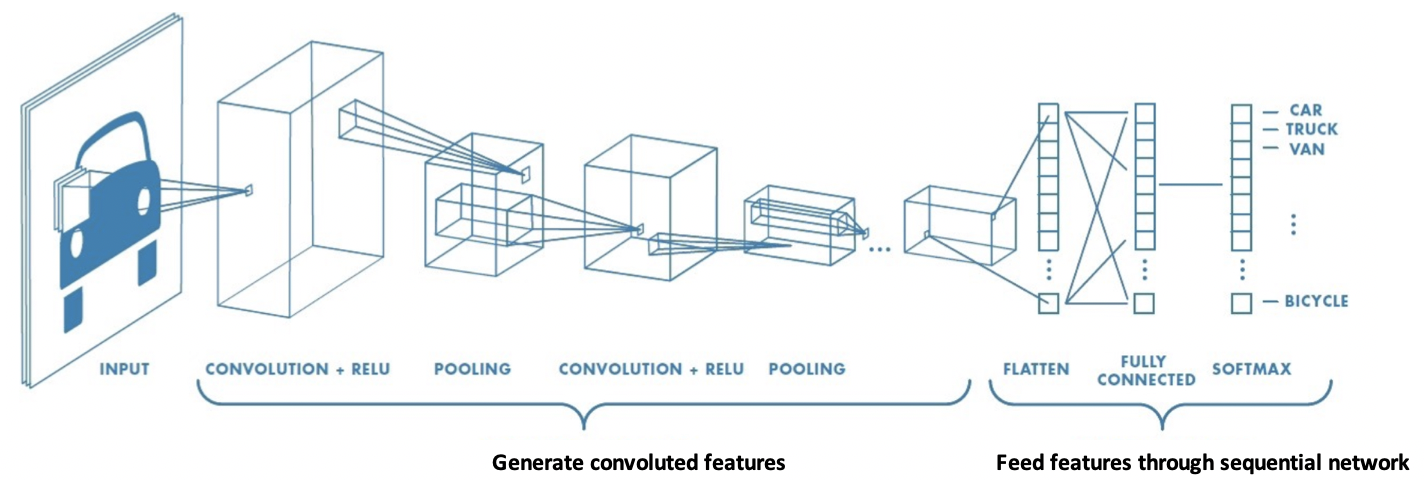
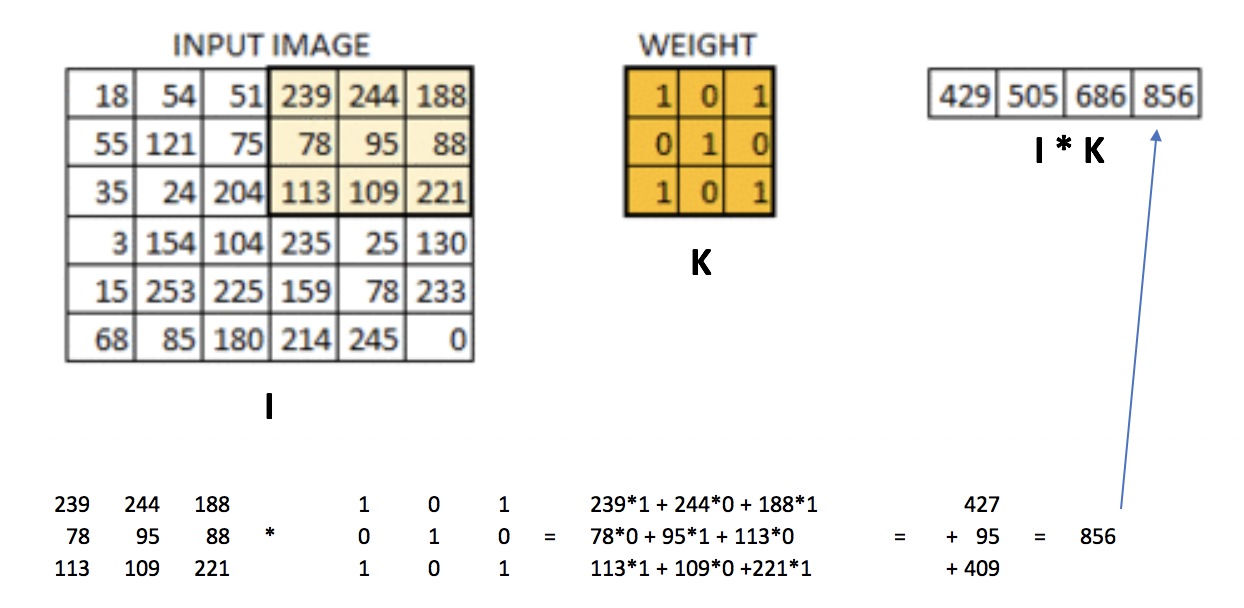
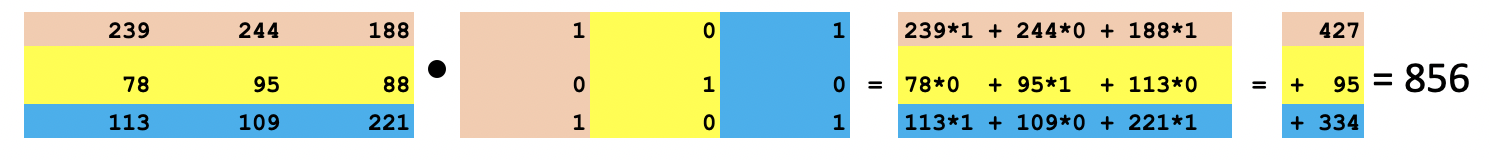


Figure 2 shows how nine pixels of an image can be convoluted into one element of a pooled layer. Notice how the pooled layer dimensions are much smaller than the original image.

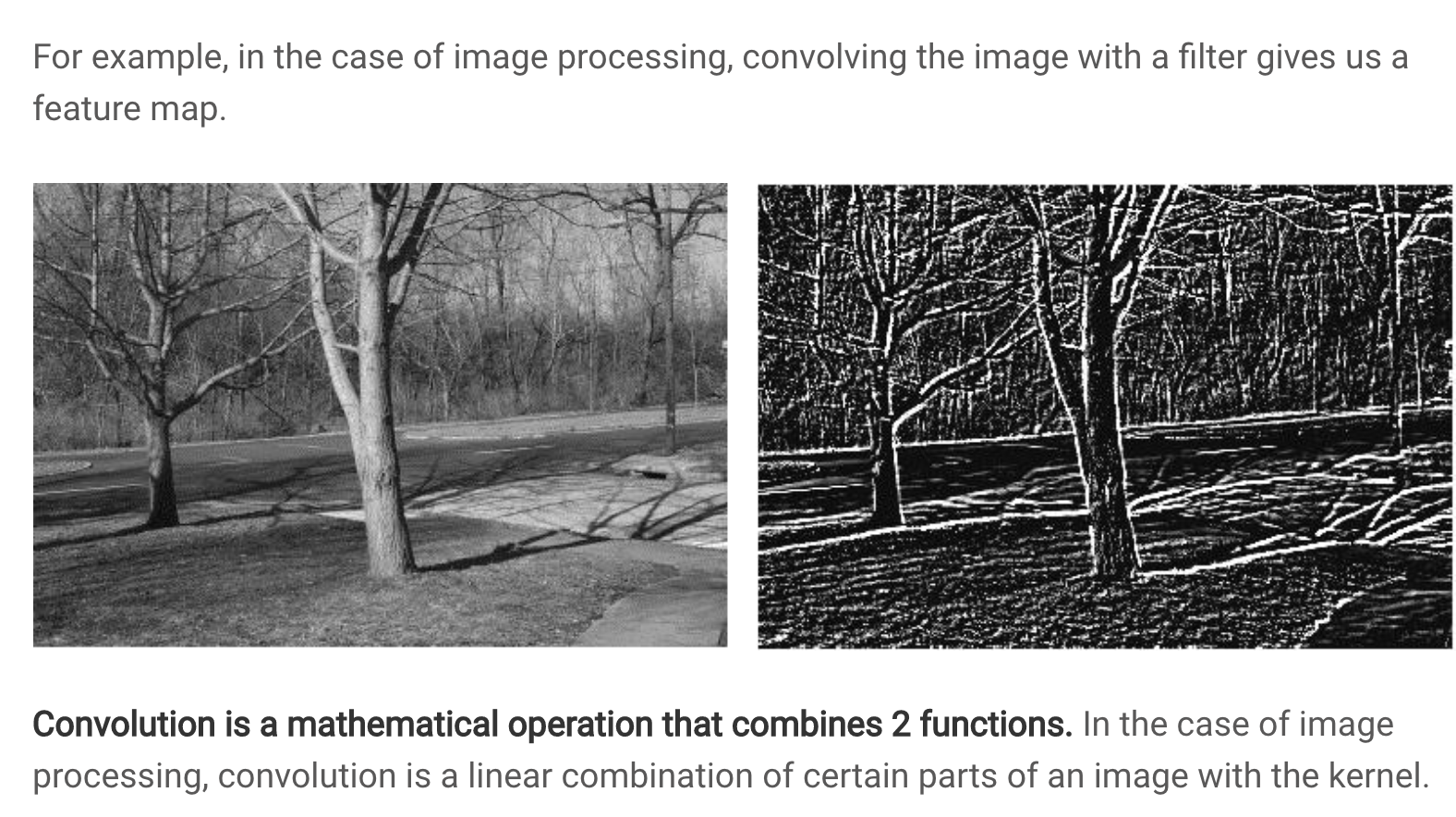
Figure 2: Transforming Nine Elements into One Pooled Element





Convolving with a filter creates a feature map like the one in Figure 3.

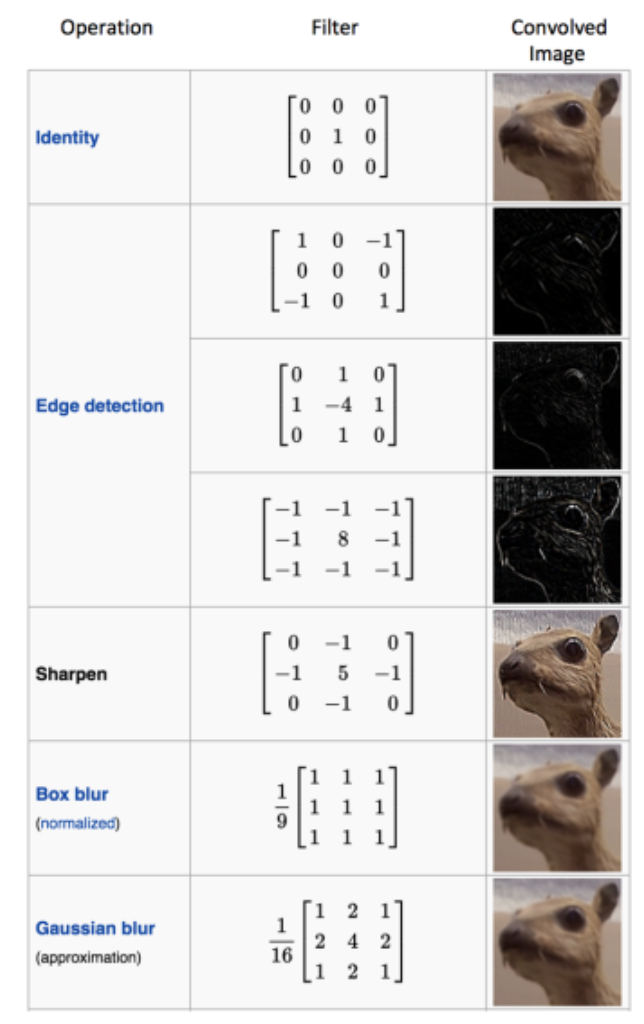
Figure 3: Real Image vs. Convoluted Image



## Common Filters (Kernels)

Not all filters are the same. Figure 4 highlights several filter types.

Figure 4: Common Image Filters

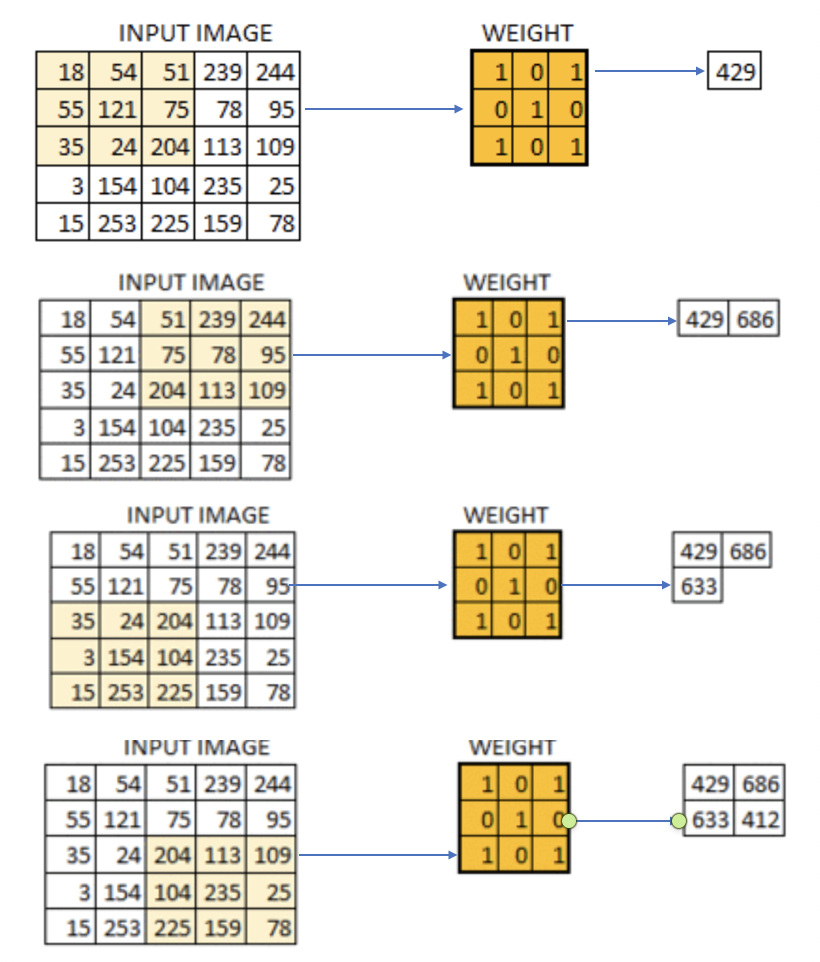


## Stride

Like we saw in Figure 2 the weight matrix was applied to sub sections of the image. In the example given, the sub sections were selected by shifting a window horizontally across the original image one pixel at a time and then down one pixel at a time. This **stride** of 1 pixel x 1 pixel led to a convoluted image that is 4x4 pixels. Lower strides leads to a higher amount of detail in the convoluted image.

A higher stride of 2x2 leads to a smaller convoluted image. Figure 5 illustrates a how a convoluted image is created with a filter by using a 2x2 stride. The first section of the image is selected and is filtered. Then the selection is shifted two pixels across. The new selection is multiplied by the filter. Then the selection is taken from the bottom left of the original image and is multiplied by the filter. Then the selection is taken from the bottom right of the image and this section is multiplied by the filter.

Figure 5: Creating a Convoluted Image with a 2x2 Stride



Exercise 1 (3 marks)

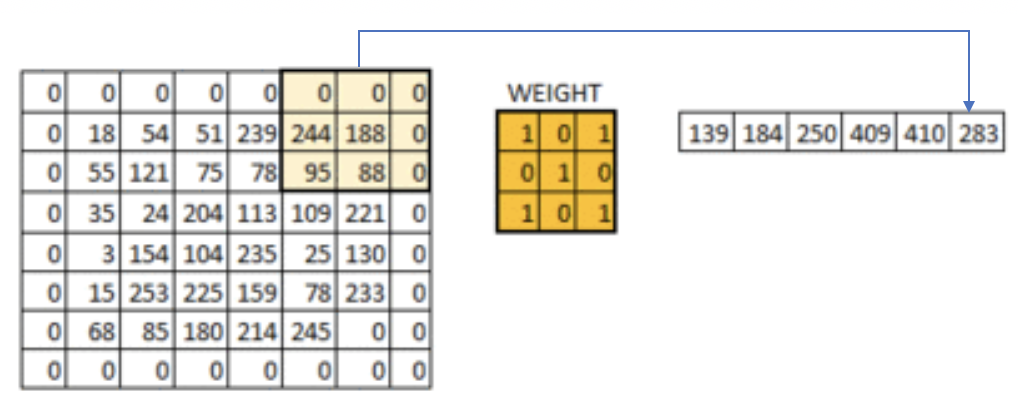
Show the calculation needed to multiply the section of the input image by the filter to generate 686. See Figure 5.

|  |
| --- |
|  |

## Padding

You may have noticed that striding across the image leads to a smaller convoluted image. However, we could pad the border with zeros to retain the original size of the image. Padding also allows us to obtain more information about the borders (see Figure 6).

Figure 6: Padding an Image



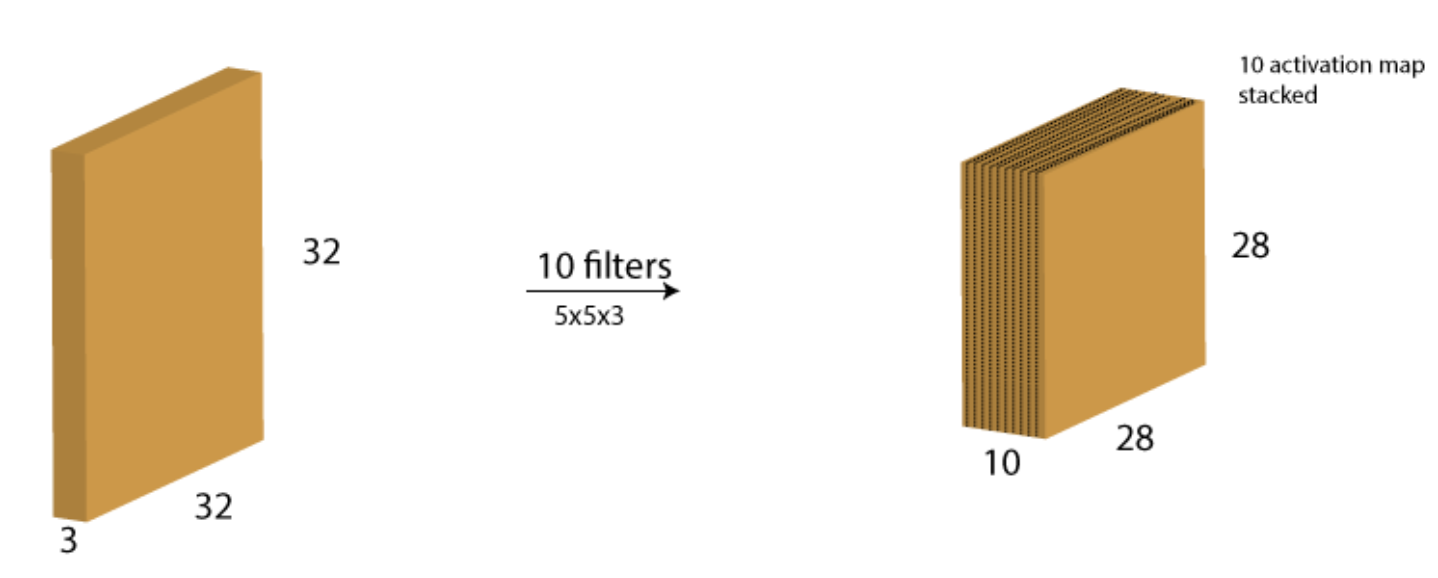
## Pool Size

A pool\_size attribute is used in the network to downsample the image. The calculation used for downsampling is:

output\_shape = (input\_shape - pool\_size + 1) / strides)

## Applying Multiple Filters

If we have a 32x32 image with a 3-byte color layer and we apply a stack of ten 5x5x3 filters **we** end up with a convoluted output of 28x28x3.

****

### Increasing Filters in Each Consecutive Layer

In every layer filters are there to capture patterns. For example in the first layer filters capture patterns like edges, corners, dots etc. In the subsequent layers we combine those patterns to make bigger patterns. Like combine edges to make squares, circle etc.

Now as we move forward in the layers the patterns gets more complex, hence larger combinations of patterns to capture. That's why we increase filter size in the subsequent layers to capture as many combinations as possible.

<https://datascience.stackexchange.com/questions/55545/in-cnn-why-do-we-increase-the-number-of-filters-in-deeper-convolution-layers-fo>

Example 1: Processing and Displaying Images

This first example demonstrates how to prepare the images for modelling.

Initially the images are loaded.

img = cv2.imread(filePath)

The images are then scaled to a size of 96x96 pixels.

img = cv2.resize(img, (IMAGE\_SIZE, IMAGE\_SIZE))

Next, the code eliminates the 3 byte color channel.

img = cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY)

To understand our image data better some print statements are used to display the data attributes and contents.

print("Image Label: " + str(labels[index]))

print("Image Shape: " + str(images[index].shape))

print("Image Data: ")

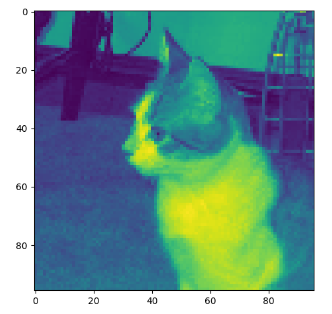
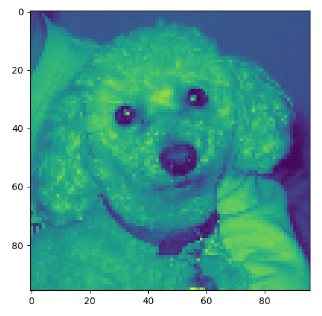
print(images[index])

|  |
| --- |
| Image Label: 0  Image Shape: (96, 96)  Image Data:  [[0.06666667 0.03137255 0.01568627 ... 0.14901961 0.16470588 0.38039216]  [0.02352941 0.05098039 0.03137255 ... 0.18039216 0.21568627 0.37254902]  [0.03921569 0.03137255 0.03529412 ... 0.18823529 0.2 0.32941176]  ...  [0.30196078 0.27843137 0.29019608 ... 0.34901961 0.36470588 0.36470588]  [0.33333333 0.32156863 0.31764706 ... 0.3372549 0.33333333 0.33333333]  [0.34901961 0.31764706 0.32941176 ... 0.33333333 0.33333333 0.35294118]] |

To verify that the images are process properly, two processed test images are displayed on the screen.

plt.imshow(images[index])

plt.show()

|  |
| --- |
| import matplotlib.pyplot as plt  import os  import numpy as np  import cv2  PATH = "/Users/pm/Desktop/DayDocs/data/small\_cats\_and\_dogs/"  # Loads the image file.  def load\_image(fileName, subFolder):  try:  filePath = PATH + subFolder + "/" + fileName  img = cv2.imread(filePath)  return img  except:  return np.NaN  # Extract the actual class from the image file name.  def extract\_label(file\_name):  return 1 if "dog" in file\_name else 0  # Transforms image into scaled and squared image.  def preprocess\_image(img):  IMAGE\_SIZE = 96  try:  # Finds minimum of height and width.  min\_side = min(img.shape[0], img.shape[1])  # Reduce image to square using minimum of width and height.  img = img[:min\_side, :min\_side]  # Reside to 96x96.  img = cv2.resize(img, (IMAGE\_SIZE, IMAGE\_SIZE))  print(img.shape)  # Eliminate three byte color channel.  img = cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY)  print(img.shape)  print("\*\*\*")  # Scale numbers to range between 0 and 1.0  return img / 255.0  except:  return np.NaN  def getImages(dirName):  imageList = []  labelList = []  # List all items in the directory.  image\_files = os.listdir(PATH + dirName + "/")  for i in range(0, len(image\_files)):  image = load\_image(image\_files[i], dirName)  processedImage = preprocess\_image(image)  # Build list of processed images and labels.  if processedImage is not np.NaN:  imageList.append(processedImage)  label = extract\_label(image\_files[i])  labelList.append(label)  return imageList, labelList  # Displays grayscale image and image data to ensure everything is working properly.  def verifyImage(images, labels, index):  print("Image Label: " + str(labels[index]))  print("Image Shape: " + str(images[index].shape))  print("Image Data: ")  print(images[index])  # Display the image.  plt.imshow(images[index])  plt.show()  print("\*\*\*\*\*")  # Add an extra dimension for the signle color channel.  # Changes (2000, 96, 96) to (2000, 96, 96, 1)  # Load transformed and scaled images.  X\_train, y\_train = getImages('train')  X\_test, y\_test = getImages('test')  # Verify first image in training set.  verifyImage(X\_train, y\_train, 0)  # Verify second image in training set.  verifyImage(X\_train, y\_train, 1) |

Example 2: Convert to 3D Array

Like other neural network examples, we need to ensure our inputs are in vertical array format to enable matrix multiplication. This code transforms our data into a vertical array. Here is the shape and data formatted after the transformation.

|  |  |
| --- | --- |
| **X shape: (2000, 96, 96, 1)** | **y shape: (2000,)** |
| array([[0.27058824, 0.28627451, 0.30588235, ..., 0.33333333, 0.39215686,  0.33333333],  [0.31372549, 0.30588235, 0.30588235, ..., 0.34509804, 0.38039216,  0.40784314],  [0.31372549, 0.3254902 , 0.30980392, ..., 0.36470588, 0.42745098,  0.39215686],  ...,  [0.25490196, 0.25490196, 0.24313725, ..., 0.01960784, 0.01960784,  0.01568627],  [0.28235294, 0.29803922, 0.30588235, ..., 0.01568627, 0.02352941,  0.01176471],  [0.30980392, 0.3254902 , 0.32156863, ..., 0.02352941, 0.02352941,  0.01176471]]),  array([[0.83529412, 0.85882353, 0.8627451 , ..., 0.41960784, 0.45098039,  0.43921569],  [0.79607843, 0.79215686, 0.81176471, ..., 0.41960784, 0.44705882,  0.43921569],  [0.76862745, 0.77647059, 0.78039216, ..., 0.40392157, 0.44313725,  0.4627451 ],  ...,  [0.31764706, 0.3372549 , 0.34509804, ..., 0.24705882, 0.23529412,  0.23137255],  [0.32941176, 0.3372549 , 0.36862745, ..., 0.21568627, 0.24705882,  0.22352941],  [0.34117647, 0.35294118, 0.34117647, ..., 0.22745098, 0.24705882,  0.24313725]]),  ...] | [1,  0,  0,  1,  1,  1,  1,  0,  0,  0,  0,  1,  0,  0,  1,  0,  1,  1,  1,  1,  1,  0,  0,  1,  0,  0 |

This is the code that performs the transformation to a matrix. Add this code to Example 1:

|  |
| --- |
| def convertToArray(x, y):  x = np.array(x)  x = np.expand\_dims(x, -1)  y = np.array(y)  print(x.shape)  print(y.shape)  print("\*\*\*")  return x, y  X\_train, y\_train = convertToArray(X\_train, y\_train) X\_test, y\_test = convertToArray(X\_test, y\_test) |

Example 3: Building and Evaluating the Model

To help understand how the model is built here are some terms.

## Kernel Size

The kernel is our sliding window which performs the convolution. Typical sizes include (1,1), (2,2),(3,3),(4,4),(5,5),(6,6),(7,7). It is rare to have more than a 7x7 kernal size.

## Filters

The number of filters determines the number of ways the kernel data is transformed. The number of filters corresponds to the number of outputs that are generated during the convolution.

## Strides

A stride is the horizontal and vertical step taken when shifting the convolutional window.

The model is decent - this is as good as I could make it.

|  |
| --- |
| Accuracy: 0.755  Confusion Matrix  Predicted 0 1  Actual  0 360 140  1 105 395 |

Here is the code that builds our model. Add this code to Example 2:

|  |
| --- |
| import tensorflow as tf  from tensorflow.keras import Sequential  from tensorflow.keras.layers import Dense  from tensorflow.keras.layers import Conv2D  from tensorflow.keras.layers import MaxPool2D  from tensorflow.keras.layers import Flatten  from tensorflow.keras.layers import Dropout  def create\_model(X\_train, y\_train, X\_test, y\_test):  model = Sequential()  model.add(Conv2D(filters=32, kernel\_size=(3, 3), activation='relu',  kernel\_initializer='he\_uniform',  input\_shape=X\_train.shape[1:]))  # Add a pooling layer to down sample height and widths by half.  # pool\_size is filter size.  model.add(MaxPool2D(pool\_size=(2, 2), strides=(2, 2)))  # It is common to repeat this pattern a few times by doubling the filter  # each time to offset the reduction in size from down sampling.  model.add(Conv2D(filters=64, kernel\_size=(3, 3), activation='relu',  kernel\_initializer='he\_uniform',  input\_shape=X\_train.shape[1:]))  model.add(MaxPool2D(pool\_size=(2, 2), strides=(2, 2)))  model.add(Conv2D(filters=128, kernel\_size=(3, 3), activation='relu',  kernel\_initializer='he\_uniform',  input\_shape=X\_train.shape[1:]))  model.add(MaxPool2D(pool\_size=(2, 2), strides=(2, 2)))  model.add(Conv2D(filters=256, kernel\_size=(3, 3), activation='relu',  kernel\_initializer='he\_uniform',  input\_shape=X\_train.shape[1:]))  model.add(MaxPool2D(pool\_size=(2, 2), strides=(2, 2)))  model.add(Conv2D(filters=512, kernel\_size=(3, 3), activation='relu',  kernel\_initializer='he\_uniform',  input\_shape=X\_train.shape[1:]))  model.add(MaxPool2D(pool\_size=(2, 2), strides=(2, 2)))  # feed data convolved features to a sequential network.  model.add(Flatten())  model.add(Dense(512, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dropout(0.5))  model.add(Dense(256, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dropout(0.5))  # Create output layer which has two classes.  model.add(Dense(units=2, activation='softmax'))  model.compile(optimizer=tf.optimizers.Adam(),  # We are using classification.  loss=tf.losses.SparseCategoricalCrossentropy(),  # Show accuracy.  metrics=['accuracy'])  history = model.fit(X\_train, y\_train, epochs=20, batch\_size=100,  validation\_data=(X\_test, y\_test))  model.save\_weights("model.tf")  return model, history  model, history = create\_model(X\_train, y\_train, X\_test, y\_test)  predictions = model.predict(X\_test)  # Iterates through pairs of predictions and adds most probable option to list.  predictionList = []  for i in range(0, len(predictions)):  prediction = predictions[i]  if(prediction[0] > prediction[1]):  predictionList.append(0)  else:  predictionList.append(1)  predictionArray = np.array(predictionList)  print(predictionList)  import pandas as pd  from sklearn import metrics  # Show confusion matrix and accuracy scores.  confusion\_matrix = pd.crosstab(y\_test, predictionArray,  rownames=['Actual'],  colnames=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, predictionArray))  print("\nConfusion Matrix")  print(confusion\_matrix) |

Exercise (3 marks)

Perform this calculation for one dimension only. Given that the input width is 96, the pooling filter size is 2 and the strides value is 2, prove the convoluted image that is output is approximately half the size of the original.

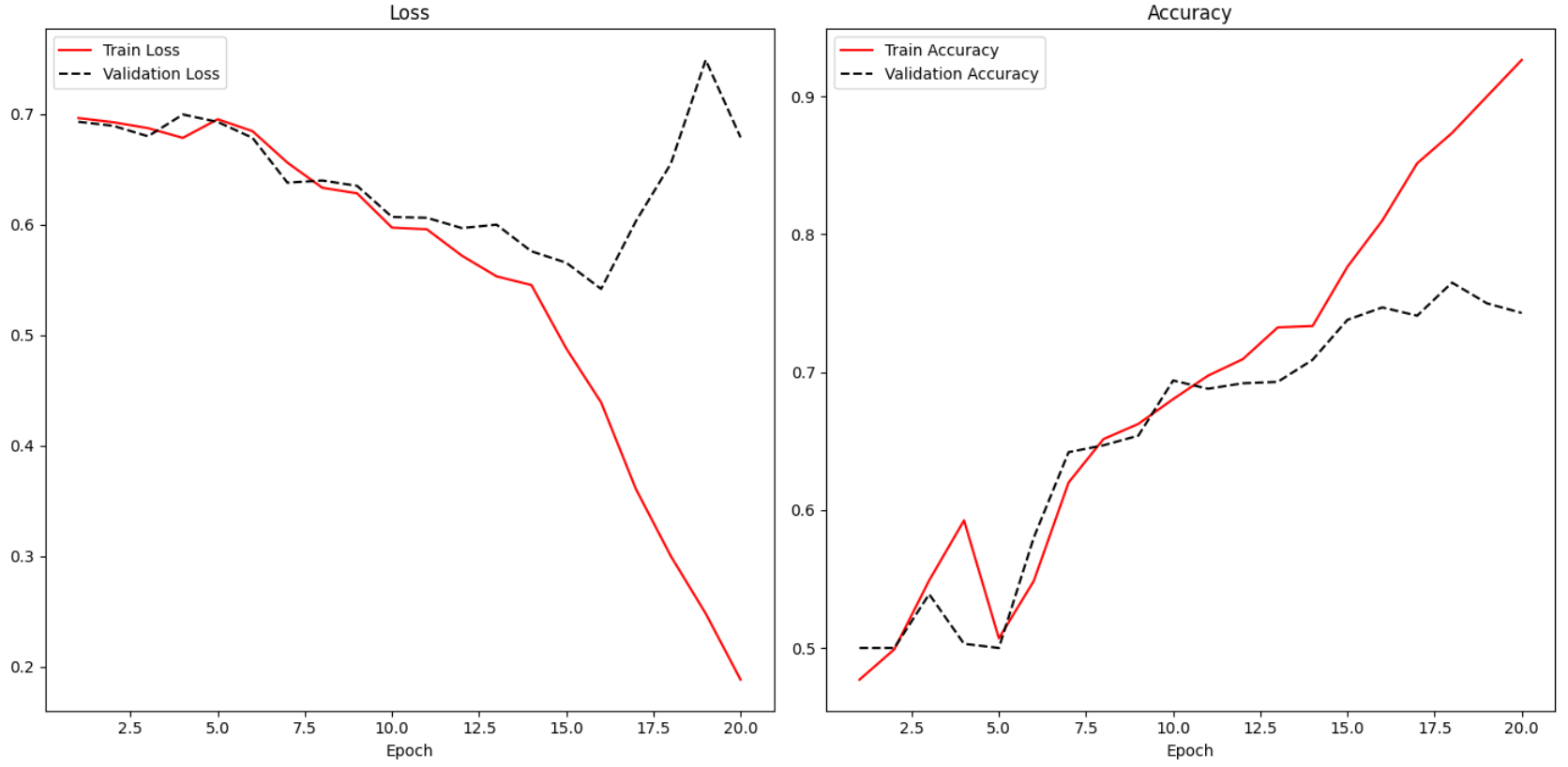
output\_shape = (input\_shape - pool\_size + 1) / strides)

Show your output here:

|  |
| --- |
|  |

Example 4: Showing the loss and accuracy

It is helpful to visualize the validation loss and accuracy. The loss function begins to rise at epoch 16. This suggests overfitting.



Adding the following code to Example 3 enables the reporting of loss and accuracy for classification algorithms.

|  |
| --- |
| def showLoss(history):  # Get training and test loss histories  training\_loss = history.history['loss']  validation\_loss = history.history['val\_loss']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 1)  # Visualize loss history for training data.  plt.plot(epoch\_count, training\_loss, label='Train Loss', color='red')  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, 'r--', label='Validation Loss',  color='black')  plt.xlabel('Epoch')  plt.legend(loc="best")  plt.title("Loss")  def showAccuracy(history):  # Get training and test loss histories  training\_loss = history.history['accuracy']  validation\_loss = history.history['val\_accuracy']  # Create count of the number of epochs  epoch\_count = range(1, len(training\_loss) + 1)  plt.subplot(1, 2, 2)  # Visualize loss history for training data.  plt.plot(epoch\_count, training\_loss, label='Train Accuracy', color='red')  # View loss on unseen data.  plt.plot(epoch\_count, validation\_loss, 'r--',  label='Validation Accuracy', color='black')  plt.xlabel('Epoch')  plt.legend(loc="best")  plt.title('Accuracy')  plt.subplots(nrows=1, ncols=2, figsize=(14,7))  showLoss(history)  showAccuracy(history)  plt.show() |

## Early Stopping and Saving the Best Model

Early stopping is a technique and saving the best model that is used in machine learning algorithms to detect overfitting. Early stopping allows you to stop training the model when the validation loss increases a specific number of times. You can control the number of times that the loss may increase with the patience level.

Example 5: Enabling Early Stopping and Saving the Best Model

To enable early stopping and save the best model, start with Example 4. Then place this line of code above at the start of the program. Be sure to use the big\_cats\_and\_dogs data set for this test for best results.

|  |
| --- |
| from tensorflow.keras.callbacks import EarlyStopping  from tensorflow.keras.callbacks import ModelCheckpoint |

Next, replace the line of code where the model is fit with this instruction.

|  |
| --- |
| # configure early stopping  es = EarlyStopping(monitor='val\_loss', patience=10)  mc = ModelCheckpoint('best\_model.h5', monitor='val\_accuracy', mode='max', verbose=1,  save\_best\_only=True)  history = model.fit(X\_train, y\_train, epochs=20, batch\_size=100,  validation\_data=(X\_test, y\_test), callbacks=[es,mc]) |

Next, replace this line of code:

predictions = model.predict(X\_test)

With code that loads the binary model before making a prediction:

|  |
| --- |
| # Must match from tensorflow.keras.callbacks import ModelCheckpoint  from tensorflow.keras.models import load\_model  saved\_model = load\_model('best\_model.h5')  predictions = saved\_model.predict(X\_test) |

Exercise 3 (10 marks)

Train your model to check for dogs, cats and octopi. **Be sure to use the small\_cats\_and\_dogs\_and\_octopi data set for this test.**

To enable training with the new files, drop the extracted files from octopusTest into the test folder. Drop the extract files from octopusTrain into the train folder.

After, replace extract\_label() with the following code:

|  |
| --- |
| def extract\_label(file\_name):  if "dog" in file\_name:  return 1  elif "cat" in file\_name:  return 0  else:  return 2 |

The output layer needs to be set to 3 since it is classifying for dogs, cats and octopi. Replace the output layer with this version:

|  |
| --- |
| model.add(Dense(units=3, activation='softmax')) |

Next, fix the code which sets up the confusion matrix. You will want to output 0, 1 or 2 depending on the maximum element in each row of the ***predictions*** list. This is the code that needs to be fixed:

|  |
| --- |
| predictionList = []  # Iterates through pairs of predictions and adds most probable option to list.  for i in range(0, len(predictions)):  prediction = predictions[i]  if(prediction[0] > prediction[1]):  predictionList.append(0)  else:  predictionList.append(1)  predictionArray = np.array(predictionList)  print(predictionList)  import pandas as pd  from sklearn import metrics  # Show confusion matrix and accuracy scores.  confusion\_matrix = pd.crosstab(y\_test, predictionArray,  rownames=['Actual'],  colnames=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, predictionArray))  print("\nConfusion Matrix")  print(confusion\_matrix) |

Show your fixed code here:

|  |
| --- |
|  |

Show your new confusion matrix here:

|  |
| --- |
|  |