Machine Learning

Susan Fox

Activity Goals

This activity will look at several kinds of machine learning that are built into OpenCV or can be a simple add-on to OpenCV. We will have a special focus on face detection, which is a problem that has not been solved without the use of machine learning. After completing this activity, you should be able to:

- Understand how to use OpenCV's machine learning tools
- Understand what Haar features are
- Understand how face detection works

Installing new tools

You are going to need to install `matplotlib`, a library that allows us to view data in many different ways. It can be used to view images as well, and we may use it on Google Colab for the next activity.

Installing libraries often requires us to try several different approaches. I'm going to enumerate them in order of ease here. Try them in this order, and let me know which ones worked (or if none of them did).

1. Installing the library through PyCharm.
    a. Going into Preferences/Settings (it's called Preferences on Mac, Settings on Windows), and select the project interpreter for your current project (presumably the one that has opencv installed in it: verify that first!)
    b. The big panel on the right shows packages that are installed in this version of Python. If you are using Anaconda Python, you can click on the green Anaconda symbol at the bottom of the big panel, then click the + button and type "matplotlib" into the search bar.
    c. Otherwise, just click the + button and type "matplotlib" into the search bar.
    d. Select the item that just has matplotlib as its name, and click "Install Package"
2. Installing the library through Anaconda
    a. Open up the Anaconda Navigator application, and click on "Environments" on the left.
    b. Select the environment where you have OpenCV installed. It will show installed packages on the right.

    c. Change the drop-down menu at the top from “Installed” to “Not Installed”, and then type matplotlib into the search box.
    d. Click the “Install” or “Apply” button at the bottom of the window.
3. If you are **not** using Anaconda Python, and the PyCharm method doesn’t work, then you should try installing matplotlib through pip
    a. Open a Terminal window (Command Prompt in Windows), and navigate to the folder where your python installation is
    b. In the same folder with the python interpreter executable, there should also be an executable called pip
    c. If you have multiple versions of python on your computer, you may have multiple pips as well, so you want to be sure that you call the pip that is associated with the python interpreter you are using!
    d. At the prompt try these:
        i. pip install matplotlib
        ii. sudo pip install matplotlib
        iii. ./pip install matplotlib
        iv. sudo ./pip install matplotlib

Overview of Machine Learning

Machine learning refers to a set of algorithms that process data and seek patterns in data. There are many variations on machine learning, in terms of the general task and the algorithm chosen.

General ML approaches: supervised, unsupervised, reinforcement learning

- Supervised learning: We have a dataset that contains features for many instances of the data, along with the correct output for each instance
- Unsupervised learning: We have a dataset with features for many instances of data, but don’t have a correct output for them
- Reinforcement learning: We have a dataset, and we tell the algorithm “good” or “bad” for its responses, but we don’t tell it what the correct answer is

General ML tasks: classification, regression, clustering

- Classification: given a set of data that comes from two or more categories, identify the category for each instance, and generalize to correctly classify new instances
- Regression: given a set of data that serve as inputs to a function that produces a numeric result, and the correct result for each data instance, fit a function to the data to produce the right result
- Clustering: given a set of data, organize it into K clusters based on similarity (which can be measured in various ways)

Computer vision ML tasks:

- Object detection: given a set of image data, and "ground truth" that marks the location of specific objects in an image, learn how to locate the objects in images
- Image segmentation: given a set of image data, and "ground truth" that divides the pixels of the image into different categories, learn how to segment images
- Tagging/text descriptions: given a set of image data, and text tags or a text description, learn to associate textual terms to an image based on its content

Machine learning process:

1. Select a task and learning algorithm
2. Collect and prepare the data (this can be a big, ugly, complicated process)
3. Divide the data into training, verification, and testing data
4. Select the parameters of the learning algorithm
5. Train the algorithm by presenting the training data
6. Test the model created by the algorithm on the verification data
7. Repeat from any step above while tweaking your choices until the algorithm performs adequately
8. Finally, test the final model on the testing data

In this activity, we will look at:

- Supervised learning for classification, in identifying digits and characters using the K-Nearest-Neighbor (KNN) algorithm
- Supervised learning for object detection, in detecting faces in images, using the Haar Cascade data

Character Classification with KNN

(This is going to use code and examples from the OpenCV Python Tutorials, but I'll try to explain things a bit more thoroughly than the tutorials do.)

What is K-Nearest-Neighbors?

KNN is a machine learning algorithm that works best with numeric data and is used for classification tasks. KNN treats the input data as points in space. Suppose a dataset contains instances, each of which are an array of N numbers. Then each instance is a vector (or a point) in N-dimensional space.

Given this view of the data, we can calculate how close in N-dimensional space each point is to each other. To compute the distance between two points in two dimensions, we can use the Pythagorean Theorem. That can be extended to work in N-dimensions as well. Other distance measures also exist.

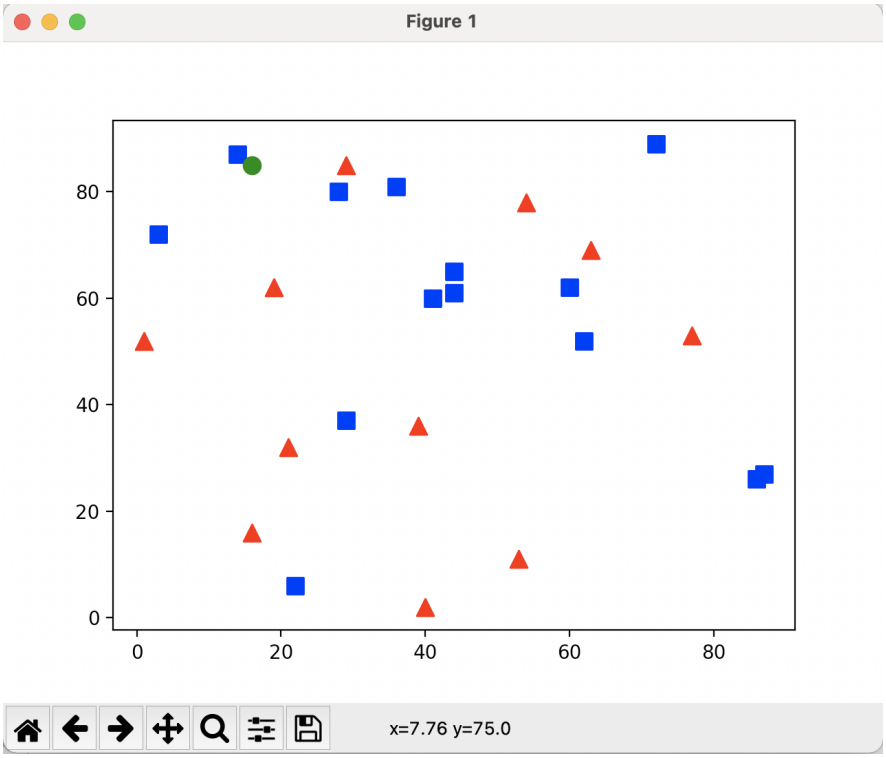
The KNN algorithm classifies a new instance by computing the K instances in the dataset that are closest to the new instance: the K nearest neighbors in the N-dimensional space. It then classifies the new instance based on the majority category of the neighbors.

The code below, adapted from the OpenCV tutorial, generates random data, and assigns half of them to one category and the other half to a second category. It then generates a new random value. It plots this data as a scatterplot (note: you must close the plot window for the program to go on).

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

# This generates 25 pairs of random numbers. Each (x, y) pair is an integer
# between 0 and 99. Once created, this is converted to an array of
# 32-bit floats because KNearest expects data in that form
trainData = np.random.randint(0, 100, (25, 2)).astype(np.float32)

# For each data point, randomly assign it to category 0 or category 1
# Convert to 32-bit floats for the same reason
responses = np.random.randint(0, 2, 25).astype(np.float32)

# Plot the training data (scatterplot, x = horizontal, y = vertical)
# Plot category 0 as red triangles
red = trainData[responses == 0]     # All data assigned category 0
```

```
plt.scatter(red[:, 0], red[:, 1], 80, 'r', '^')
# Plot category 1 as blue squares
blue = trainData[responses==1]
plt.scatter(blue[:, 0], blue[:, 1], 80, 'b', 's')

# Generate and plot a new point
newPoint = np.random.randint(0, 100, (1, 2)).astype(np.float32)
plt.scatter(newPoint[:, 0], newPoint[:, 1], 80, 'g', 'o')
plt.show()
```

This code should produce a scatterplot that looks something like this (your random numbers will be different).



Next, we want to set up the KNN algorithm, and then train it on our dataset. Note that the second input to the `train` method tells it whether the data is organized in rows or columns.

```
# Create KNN object and train on data
knn = cv.ml.KNearest_create()
knn.train(trainData, cv.ml.ROW_SAMPLE, responses)
```

Finally, we can classify our new point by asking it to find the K nearest data points. This prints the category that is the majority, and then the individual categories of the nearest data, and then their distances from the new point.

```
# Report category of new point (based on k=3)
ret, result, neighbors, dist = knn.findNearest(newPoint, 3)
print("result:", result)
print("neighbors:", neighbors)
print("distances:", dist)
```

Try changing the number of nearest neighbors to look for (keep it odd).

Milestone 1: Playing with artificial data

Part 1: The code above randomly assigns points to category 0 or category 1. Suppose we want to experiment with randomly-created data, but we want something a bit less arbitrary.

Change the part of the code that defines responses so that it does the following:

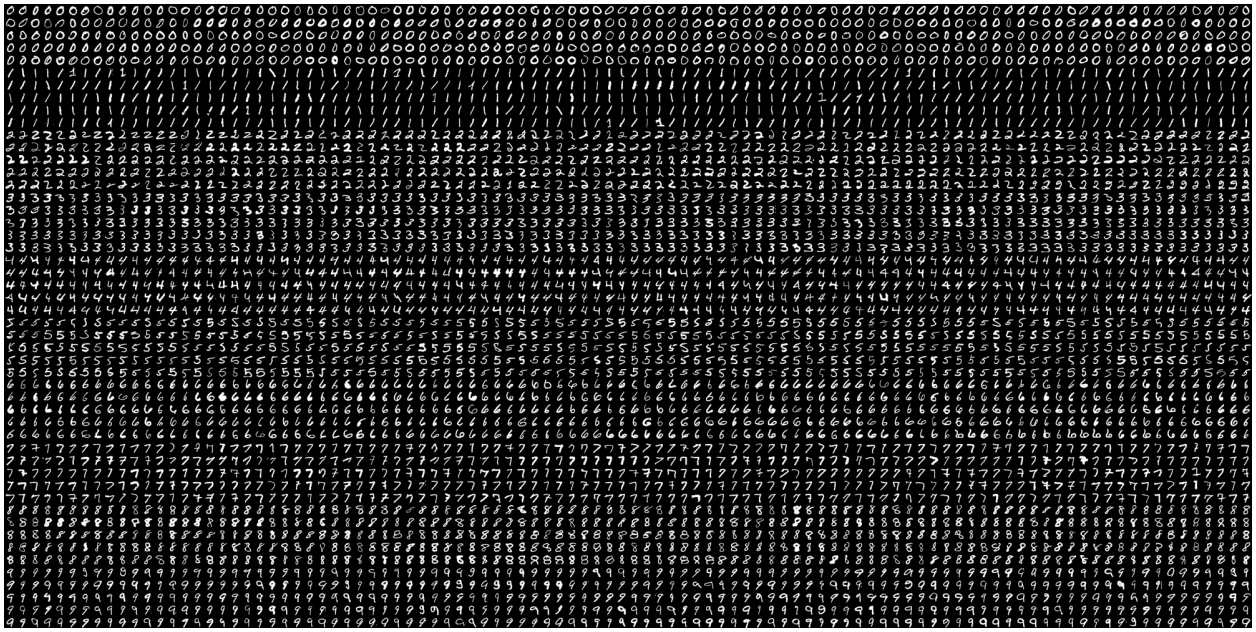
1. Assign a new variable to hold the result of summing the (x, y) values for each of the 25 points (`sum` is an array method in Numpy). For instance, if one of the pairs was (51, 33) then the resulting value should be 84.
2. Use the Numpy command `logical_and` to build a boolean array that is true where the corresponding value from the previous step is between 75 and 125, and false otherwise.
3. Use the `astype` method to convert the boolean array to 32-bit floats.

Part 2: In the code above, we generate just one test point. The `findNearest` method can also take an array of points, similar in shape to the training data, and will report all their results at once. Use the `randint` function to generate 10 random points, and pass the array to `findNearest`. Print the results just as before, and examine how they differ from a single point.

Digit classification

We are going to work with a very simple, small dataset of hand-written digits. The data is stored, in this case, in one image: `digits.png`, which I will provide to you. This data consists of 5000 images of digits, each just 20 by 20 pixels in size. There are 500 images for each digit.

The image is 2000 pixels wide and 1000 pixels tall. Each digit has 5 20x20 rows of images, with 100 images per row. Below is a version of the `digits.png` image:



This step involves some **data wrangling**, which is often a feature of working with machine learning. We have to process the big image and break it up into the small images, attach the right label (0 through 9) to each image, and then convert the images so that they are one-dimensional arrays (400-long, rather than 20 by 20) of floating point numbers. Then they will be suitable for training the `KNearest` algorithm.

Step 1: Break up the big picture

The code below reads in the digits image, converts it to grayscale, and then breaks it up into 20x20 chunks. The `vsplit` and `hsplit` functions from Numpy divide an array into the given number of sub-arrays: `vsplit` splits up the rows of the array, and `hsplit` splits up the columns. This code combines them using a list comprehension, producing a list of arrays of 20x20 grayscale images. The final step converts that list back into a Numpy array, with 50 rows, 100 columns, and each value within that is a small image.

```
img = cv.imread('digits.png')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)

# Now we split the image to 5000 cells, each 20x20 size
cells = [np.hsplit(row,100) for row in np.vsplit(gray,50)]

# Make it into a Numpy array: its size will be (50,100,20,20)
x = np.array(cells)
```

Step 2: Separate training from testing data

For this example, we will just have training data and testing/validation data (we won't break the dataset into three parts). We're going to assume that, within each digit, the individual pictures are randomized. So we can create training and testing data by using some columns of the x array for training, and the rest for testing. Initially, we will split the data in half (in the milestone you'll experiment with other splits).

Remember that the x array has 100 columns. We will take the first 50 for training, and the second 50 for testing. I am recording how many training images I have per digit with the third line below, so that I can generate the correct categories for each training example later.

```
trainCols = 50
train = x[:, :trainCols]
test = x[:, trainCols:100]
trainPerDigit = 500 / (100 / trainCols)
testPerDigit = 500 / (100 / (100 - trainCols))
```

Step 3: Flatten the data

The next step is to turn this data into a 2d array, with one row for each image. We will convert it to 32-bit floats at the same time.

```
train = train.reshape(-1, 400).astype(np.float32)
test = test.reshape(-1, 400).astype(np.float32)
print(train.shape, test.shape)
```

Step 4: Generate the correct output categories

We know from earlier that each digit has `trainPerDigit` rows in our training data. We want to generate the responses we expect, by repeating the digit for each set of rows the correct number of times (so the array will have 250 0's, followed by 250 1's, etc, in this case). The final piece is to adjust the shape of the array (a common requirement for data wrangling!) so that, instead of being (2500,) it is (2500,1) so that it matches the format that the `KNearest` algorithm will report back to us. The `np.newaxis` option adds a second axis with just one value in it.

```
digitCats = np.arange(10)
train_labels = np.repeat(digitCats,trainPerDigit)[:, np.newaxis]
test_labels = np.repeat(digitCats, testPerDigit)[:, np.newaxis]
```

Milestone 2: Finish the digits program

Complete the KNN training and testing
The steps above end with data that is in the correct format for the KNearest object. Using the earlier program that ran on artificial data as a model, set up a KNearest object, and train it on the training data above. Call findNearest on the testing data above. Add the lines of code below; they compute the percentage of the test examples where KNN gave the correct answer.

```
matches = result == test_labels
correct = np.count_nonzero(matches)
accuracy = correct * 100.0 / result.size
print(accuracy)
```

Experiment!
A major component of working with machine learning is experimenting with how changing the parameters of the problem changes the outcomes. Try the experiments here, and be prepared to report on your findings!

- Dig into the raw results a bit more. Compute the accuracy for each digit. Is KNN equally accurate on all categories, or does it do better on some than on others? Are there patterns to when it is right or wrong?
- Try changing the number of neighbors. For which number of neighbors was the resulting accuracy the best?
- Try changing the number of columns allocated to training versus testing. How does that affect the performance of the algorithm on the test data?

Face Detection with Haar Cascades

Humans are incredibly good at picking faces from our visual field, starting when we are tiny babies. It is much more difficult for a computer to locate faces. Successful computer-based techniques are built on *machine learning*, where the computer is given thousands or tens of thousands of images with faces and images without faces, and it constructs a *classifier* that can distinguish faces from non-faces in an image. While we think about this as a classifier, because it takes a region of an image and reports “yes there is a face” or “no there isn’t,” this

task is actually an example of object detection, and the algorithm has to integrate a search of many different possible locations into its work.

OpenCV provides a trained classifier for faces based on “Haar-like features.” The classifier is called a “cascade” because it learns many different Haar-like features, and needs many of them to vote for the existence of a face to be confident that it has found one. Any one feature might not indicate a face, but each Haar-like feature that is found increases the likelihood of an actual face.

A Haar-like feature is a pattern that is easy to compute on an image, and that bears some similarity to a mathematical idea: a Haar wavelet. The Haar-like feature is in some ways like the kernels we saw earlier. A Haar-like feature checks two side-by-side rectangular regions of an image. It adds the intensity values in each rectangular region, and then it takes the difference between the two. Haar-like features can be many different sizes and orientations (see the picture below) and are applied all over an image, much like the kernels for the filter. At each location the pixels in the gray part are added together, and the pixels in the white part are added together, and then we subtract the white sum from the gray sum.

The cascade learns, for each Haar-like feature, what difference value corresponds to the presence of a face. Any cascade may include hundreds of Haar-like features.

We will not have to train a Haar cascade ourselves. OpenCV provided a set of trained cascades for faces, eyes, and so forth. You should find the zip archive called `haarcascades.zip` in Schoology and download it. Add it to your project just like you did with `SampleImages`. **Note: the names of the haar cascade files may differ from your reading!** We create a classifier, reading its specifications from a file, and then apply it to images. Try the following code sample:

```
import cv2

faceCascade =
cv2.CascadeClassifier("haarcascades/haarcascade_frontalface_alt0.xml")

cap = cv2.VideoCapture(0)

while True:
        ret, frame = cap.read()
```

```
        gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

        face_rects = faceCascade.detectMultiScale(gray, 1.3, 5)

        for (x, y, w, h) in face_rects:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 3)

        cv2.imshow("FD", frame)

        v = cv2.waitKey(20)
        c = chr(v & 0xFF)
        if c == 'q':
            break

cap.release()
cv2.destroyAllWindows()
```

This code should detect any forward-facing faces in the image. Test it with your own web-cam and maybe have a friend or family member join you. You could also change it to work on a still image with multiple faces in it.

Try other cascade files in the folder I've provided. Note that there are two files for detecting cat faces, eyeglasses, license plates, and smiles. Try one or two of these and see how they work either on your video or on still photos. You may have to get some new images for some of these.

Milestone 3: Finding Eyes

Part one: Your reading included an example program for finding eyes within faces. A key part of the program is that you only run the eye-detector on a ROI (region of interest) bounded by the face that was found. Add an eye detector to the program above, created outside of the loop. Then, for each face that is found, create a ROI of that region. Run the eye detector on it, and for every eye that is detected, draw a circle or rectangle to mark it. Note that the eye detector, like the face detector, returns the (x, y) coordinate of the upper left corner and the width and height. Also, **remember that positions in the ROI will start with (0,0) in the corner of the ROI**.

Part two: Get a copy of the `GoogleyEye.png` file, and add it to your project. Modify your previous program so that it draws the eye image at each detected eye.
Suggestions:

- Use `resize` to change the size/shape of the eye image to match the eye rectangle
- Even within the bounds of a detected face, the eye detector may find false eyes (it seems to think I have at least 4 eyes at all times). Try to pick the “best” two eyes detected (the ones you think are most likely to be actual eyes) and draw just those. Beware of times when fewer than two eyes are detected!

Face Detection with Haar Cascades