Kermit Mitchell III, Zeke Hammonds Start Date: 04/02/2019 | Last Edited: 05/15/2019
Dr. Gary Hu - Computer Vision - Final Project Report

Pokedex - Pokemon Image Classification

Premise

In the video game and media franchise *Pokemon*, humans embark on expansive adventures to collect, train, and battle with Pokemon, fictional animal-like creatures endued with mysterious magical abilities. At the onset of their journey, Pokemon Trainers are bestowed a device called a *Pokedex* to aid them and track their progress in exposure to different kinds of Pokemon. The Pokedex is a gadget with artificial intelligence, containing a comprehensive crowdsourced encyclopedia about all species of Pokemon inhabiting a particular region. Access to the information on the Pokedex is restricted however, until the trainer has been introduced to the Pokemon they want information for. To gain access to the information and log their progress, trainers must use the Pokedex's camera sensors and attempt creature image recognition to identify the Pokemon in question. Upon successful identification, the Pokedex will recite the Pokemon's name and description, and upon failure the Pokedex will simply say "Unable to identify."

While *Pokemon* is a fictional fantasy sci-fi universe, the underlying technologies behind certain aspects of that universe such as a Pokedex are now possible to tangibly implement in our real world, in this case using smartphones, telecommunications, artificial intelligence, machine learning, and computer vision. *For my final project, I will attempt to create a proof-of-concept Pokedex*.

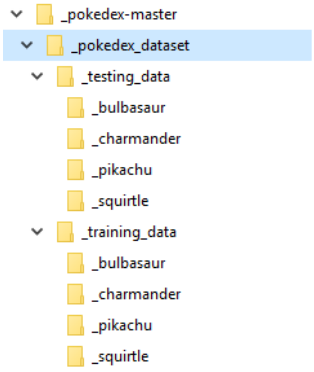
Framing the Problem

As this is a proof-of-concept, I'm only concerned with accurately identifying a tiny subset of the over 800 Pokemon in existence (or 151 for originalists), thus, *I will use 4 categories or 'classes' of images*, namely the iconic Kanto starter Pokemon: *Pikachu*, *Bulbasaur*, *Charmander*, and *Squirtle*. Because I plan on using machine learning solutions for this task, I will need a dataset containing images of these Pokemon and I will need to choose a learning model to train. Initially, I wanted to use a Convolutional Neural Network (CNN) approach, however, those require *thousands* of training images to work properly, and after searching for countless hours for datasets I was unable to find a satisfying one. From there, I concluded that I'd just create my own dataset and publish it on Kaggle later for posterity. Abandoning the CNN, I ultimately locked on using Support Vector Machines (SVM) for the job. SVMs don't require nearly as much data as CNNs do, yet have proven to be fairly accurate in previous similar classification problems. Regarding my tech stack, *I will use OpenCV Python for image processing and scikit-learn for machine learning*. I previous tried OpenCV JavaScript, however, the Python edition was much

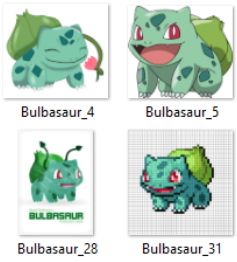
more reliable, required less dependencies, and had overall better integration with scikit-learn and other modules.

```
# Essential Imports
import cv2 as cv # Image processing like HoG
import numpy as np # Fast matrix manipulation
from matplotlib import pyplot as plt # Graphing data
from sklearn import svm as SVM # machine learning models
from joblib import dump, load # saving and writing PCA/SVM
import os # file system access
import sys # for debugging and terminal print length
#np.set_printoptions(threshold=sys.maxsize) | toggle for full values in terminal
```

Data Collection and Preprocessing



First, *I downloaded 1200 images* from Bing Images, Google Images, and Kaggle; *300 for each class of Pokemon*. I organized my dataset as one folder with two sub folders (one for training data and one for testing data). Within each sub dataset, I have a folder for each class of Pokemon. I partitioned the dataset such that *80% is training data and 20% is testing data*, basically, 240 training images per class and 60 testing images per class.



Second, I resized all images to 224 x 224 with a batch file extension. This acts as *data normalization* to keep the feature size consistent throughout all images, and is like a form of pooling which helps keep relevant features of images while reducing computational time. I also ran a Python script to rename each Pokemon image as <pokemon_name>_[001-300].

```
import os
path = 'C:/Users/Kermit Mitchell III/Desktop/folder_name'
files = os.listdir(path)
i = 1

for file in files:
    os.rename(os.path.join(path, file), os.path.join(path, 'FilePrefix_'+str(i)+'.jpg'))
    i = i+1
```

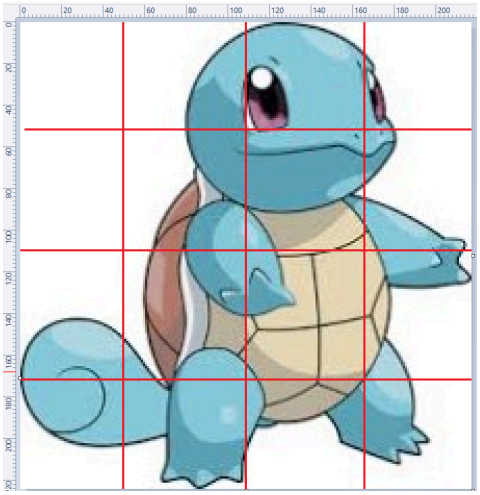
```
# These are our folders with our dataset
trainDataPath = '../_pokedex_dataset/_training_data' # absolute file path to my training data
testDataPath = '../_pokedex_dataset/_testing_data' # absolute file path to my testing data

# A vector of our labels, used below
labelVector = ["pikachu", "bulbasaur", "charmander", "squirtle"]

# Loop through each image in the datasets, and store their filepaths in arrays

'''
for each pokemon in labelVector:
    for each file in the training data/THAT_POKEMON:
        grab the filePath and store in an array
    return an array with all of the filePaths for each image
'''

# Returns an array all dataset image URLs
def labelTheData(dataFolder):
    labeledData = np.array([])
    for file in os.listdir(dataFolder):
        filename = os.fsdecode(file)
        #print(filename)
        labeledData = np.append(labeledData, dataFolder +'/' + filename)
    return labeledData

# Use the labelTheData function for each Pokemon, and for Train/Test
pikachuTrainingData = labelTheData(trainDataPath + '/_' + labelVector[0])
pikachuTestingData = labelTheData(testDataPath + '/_' + labelVector[0])
bulbasaurTrainingData = labelTheData(trainDataPath + '/_' + labelVector[1])
bulbasaurTestingData = labelTheData(testDataPath + '/_' + labelVector[1])
charmanderTrainingData = labelTheData(trainDataPath + '/_' + labelVector[2])
charmanderTestingData = labelTheData(testDataPath + '/_' + labelVector[2])
squirtleTrainingData = labelTheData(trainDataPath + '/_' + labelVector[3])
squirtleTestingData = labelTheData(testDataPath + '/_' + labelVector[3])
```

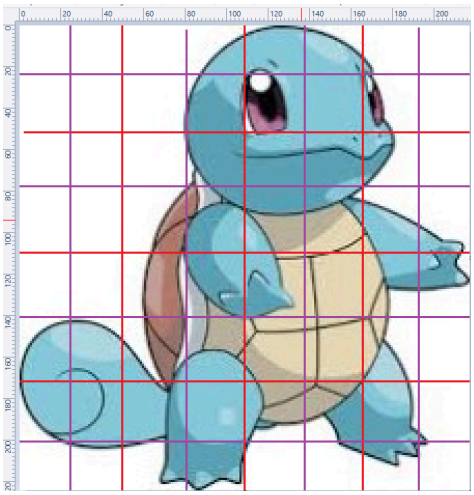
Algorithms and Techniques

Now that I have my properly preprocessed and normalized dataset, I will discuss what algorithms and techniques I am using. Regarding the SVM, *I decided to use RBF kernel* since the data is most likely not going to be linearly separable. Based on studies, instead of using one big SVM to try multiclass classification, better performance can be obtained by using independant SVMs, comparing their outputs, and taking the max likelihood score given a threshold. Therefore, *I will use four binary SVMs, one for each class of Pokemon*.

```
# These are our Support Vector Machines, and will be fed data from our HOG Descriptor
# Toggle to create SVMs from scratch, but then comment out the SVM loads
'''
svm0 = SVM.OneClassSVM(kernel="rbf", gamma=0.5)
svm1 = SVM.OneClassSVM(kernel="rbf", gamma=0.5)
svm2 = SVM.OneClassSVM(kernel="rbf", gamma=0.5)
svm3 = SVM.OneClassSVM(kernel="rbf", gamma=0.5)
'''

svm0 = load('pikachuSVM.joblib')
svm1 = load('bulbasaurSVM.joblib')
svm2 = load('charmanderSVM.joblib')
svm3 = load('squirtleSVM.joblib')
```

In terms of my image feature descriptor, *I am using Histogram of Oriented Gradients (HOG)* for my approach. I chose HOG over scale-invariant feature transform (SIFT) because I (currently) am not trying to track the specific locations of the Pokemon within the images, and therefore aren't using local feature extraction or bag-of-words. HOG is a global descriptor used for classification, and that's exactly what I'm trying to do.





Regarding HOG, I chose a block size of ((224 x 224) / 4 = 56 x 56) to get the images into 16 equal parts (a 4 x 4) shown by the red lines. Then, each block is broken up into 4 equal sub parts called cells ((56 x 56) / 2 = 28 x 28) shown by the purple lines, to make an 8 x 8.

```
# This is our HOG Descriptor, and will be used for feature extraction
IMG_SIZE = 224 # the size of our imcoming images
winSize = (IMG_SIZE, IMG_SIZE) # same size as images
blockSize = (IMG_SIZE // 4, IMG_SIZE // 4) # (224/4)=56
blockStride = (IMG_SIZE // 8, IMG_SIZE // 8) # sliding window processes blocks by cell size
cellSize = (IMG_SIZE // 8, IMG_SIZE // 8) # same as above ^
L2HysThreshold = 0.2
nbins = 9 # angles from 0-180, increments of 20 degrees

hog = cv.HOGDescriptor(winSize, blockSize, blockStride, cellSize, nbins, 1, 4, 0, 0.2, 0)
```

For each cell, the image orientation gradients are computed using partial derivatives and are mapped to a histogram of 9 bins (***nbins)***, with each bin as increments of 20 degrees, and the entire histogram ranging [0-180]. This will produce many histograms per block. There will be 7 block strides in both the horizontal and vertical direction. After computing the histograms, each of them having 9 bins, they are concatenated to
build a single feature vector with (9 * 4) * (7 * 7) = **1764** entries.

After using the HOG, *I used principal component analysis (PCA) to reduce the feature vectors from length* 1764 to **240**. This rapidly sped up training time, as the PCA will only keep the most profound and relevant aspects of the image retained. An HOG_PCA matrix contains the HOG feature vectors for each image of a class, reduced further by the PCA.

```
def HOG_PCA_Matrix(dataset, pokemon):
  originalFeatureSize = computeHOG(dataset[0]).shape[0]
  hogMatrix = np.empty((dataset.shape[0], originalFeatureSize))
  for i in range(0, dataset.shape[0]):
    hist = computeHOG(dataset[i])
    for k in range(0, originalFeatureSize):
      hogMatrix[i][k] = hist[k][0]

  #Run PCA to reduce feature dimensions, and save it for future use
  mean, eigenVectors = cv.PCACompute(hogMatrix, mean=None, maxComponents=240)
  hogMatrix = cv.PCAProject(hogMatrix, mean, eigenVectors)
  PCAMeanOutput = pokemon + 'PCA_Mean.joblib'
  dump(mean, PCAMeanOutput)
  PCAEigenOutput = pokemon + 'PCA_Eigen.joblib'
  dump(eigenVectors, PCAEigenOutput)
  return hogMatrix

# Toggle to create new hogMatrix for each Pokemon
'''
hogMatrix0 = HOG_PCA_Matrix(np.concatenate((pikachuTrainingData, pikachuTestingData), axis=None), labelVector[0])
hogMatrix1 = HOG_PCA_Matrix(np.concatenate((bulbasaurTrainingData, bulbasaurTestingData), axis=None), labelVector[1])
hogMatrix2 = HOG_PCA_Matrix(np.concatenate((charmanderTrainingData, charmanderTestingData), axis=None), labelVector[2])
hogMatrix3 = HOG_PCA_Matrix(np.concatenate((squirtleTrainingData, squirtleTestingData), axis=None), labelVector[3])
'''
```

Training the Model

With the HOG_PCA matrix for each class of Pokemon, I can now proceed to train the SVMs.

```
# Toggle to train the SVMs from scratch

#The svm training marathon/montage begins here. Get yourself some popcorn:

svm0.fit(hogMatrix0, generateLabels(pikachuTrainingData, labelVector[0]))
svm1.fit(hogMatrix1, generateLabels(bulbasaurTrainingData, labelVector[1]))
svm2.fit(hogMatrix2, generateLabels(charmanderTrainingData, labelVector[2]))
svm3.fit(hogMatrix3, generateLabels(squirtleTrainingData, labelVector[3]))

# Save the SVMs for later use

dump(svm0, 'pikachuSVM.joblib')
dump(svm1, 'bulbasaurSVM.joblib')
dump(svm2, 'charmanderSVM.joblib')
dump(svm3, 'squirtleSVM.joblib')
```

This only takes around 30 seconds at most for me. I thought that this would take hours!

Testing the Model

Now that the SVMs are trained, we need to evaluate each SVM and the aggregate machine learning structure. First, I examined the results of each SVM against the testing data to see the binary classification and the similar scores. Many of the predictions didn't look satisfying…

```
pikachu
1 -1 0.018601720384006426
2 -1 0.0002203076735111109
3 -1 0.0017882041065633292
4 -1 1.947607183527822e-06
5 -1 0.0016442911482302813
6 -1 0.0004403079839847557
7 -1 0.0006383410578745252
8 -1 0.0006655466490838036
9 -1 0.0016765692302611601
10 -1 3.4166143558778117e-06
```

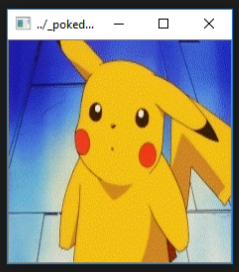
However, when compared to the results from the other binary SVMs, things began to be more accurate. When testing individual images, I have to load that image in, compute the HOG, reduce that feature vector with corresponding PCAs, and then compare that final feature vector against four different SVMs, picking the highest score against a threshold.

```
# Final Test to see which Pokemon should be selected for individual images

testImgURL = pikachuTestingData[0]
#             ^ replace with the actual test image from camera
#bulbasaurTestingData[30] <-- if you wanted to test from testing set instead

testImg = cv.imread(testImgURL)
testImg = cv.resize(testImg, (IMG_SIZE, IMG_SIZE))
cv.imshow(testImgURL, testImg)
cv.waitKey(0)
cv.destroyAllWindows()

# Run this image through HOG and PCA for each Pokemon
testImgHOG = hog.compute(testImg, winSize)
testImgHOG = testImgHOG.transpose()

# Returns the hog-pca matrix based on the input Pokemon
def pokePCA(hogMatrix, pokemon):
  PCA_Mean = load(pokemon + 'PCA_Mean.joblib')
  PCA_Eigen = load(pokemon + 'PCA_Eigen.joblib')
  pca = cv.PCAProject(hogMatrix, PCA_Mean, PCA_Eigen)
  return pca

pikachuTestImgHOG = pokePCA(testImgHOG, labelVector[0])
bulbasaurTestImgHOG = pokePCA(testImgHOG, labelVector[1])
charmanderTestImgHOG = pokePCA(testImgHOG, labelVector[2])
squirtleTestImgHOG = pokePCA(testImgHOG, labelVector[3])
```

```
scoreScaler = 1 # used to scale the scores for readability
predictions = np.empty((4, 2)) # the final predictions of all 4 SVMs
predictions[0][0] = (int)(svm0.predict(pikachuTestImgHOG)[0])
predictions[0][1] = round((svm0.score_samples(pikachuTestImgHOG))[0], 4) * scoreScaler
predictions[1][0] = (int)(svm1.predict(bulbasaurTestImgHOG)[0])
predictions[1][1] = round((svm1.score_samples(bulbasaurTestImgHOG))[0], 4) * scoreScaler
predictions[2][0] = (int)(svm2.predict(charmanderTestImgHOG)[0])
predictions[2][1] = round((svm2.score_samples(charmanderTestImgHOG))[0], 4) * scoreScaler
predictions[3][0] = (int)(svm3.predict(squirtleTestImgHOG)[0])
predictions[3][1] = round((svm3.score_samples(squirtleTestImgHOG))[0], 4) * scoreScaler
print(predictions)

# Calculate the max prediction score, and select the final result
currentMaxIndex = -1
currentMaxValue = sys.float_info.min
for i in range(predictions.shape[0]):
  if(predictions[i][1] > currentMaxValue):
    currentMaxValue = predictions[i][1]
    currentMaxIndex = i

finalAnswer = labelVector[currentMaxIndex]
threshold = 1.0 * (10**-2) # Eliminates scores that are too low
if(currentMaxValue < threshold or currentMaxIndex == -1):
  finalAnswer = "Ditto" # this means the testImg was neither class

print('Final Prediction:', finalAnswer)
cv.imshow(testImgURL, testImg)
cv.waitKey(0)
cv.destroyAllWindows()
```

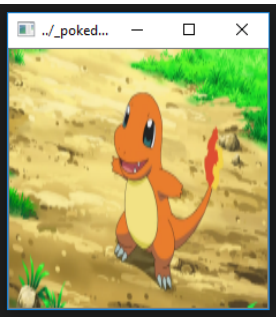
Here are some sample runs:

```
[Running] python -u "c:\Users\Kermit Mitchell III\Desktop\_pokedex
[[-1.      0.0064]
 [-1.      0.0121]
 [-1.      0.009 ]
 [-1.      0.0047]]
Final Prediction: bulbasaur
```



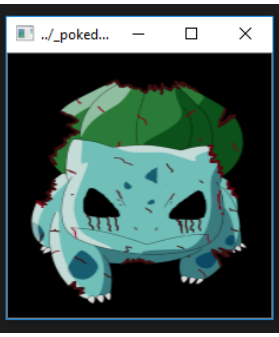

```
[[-1.e+00  0.e+00]
 [-1.e+00  2.e-04]
 [-1.e+00  2.e-04]
 [-1.e+00  1.e-04]]
Final Prediction: Ditto
```


```
[[-1.      0.004 ]
 [-1.      0.0037]
 [-1.      0.4592]
 [-1.      0.0015]]
Final Prediction: charmander
```



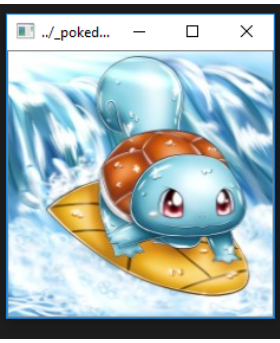

```
[[-1.      0.    ]
 [-1.      0.    ]
 [-1.      0.5951]
 [-1.      0.    ]]
Final Prediction: charmander
```


```
[[-1.00e+00  2.00e-04]
 [-1.00e+00  4.03e-01]
 [-1.00e+00  3.00e-04]
 [-1.00e+00  1.00e-04]]
Final Prediction: bulbasaur
```

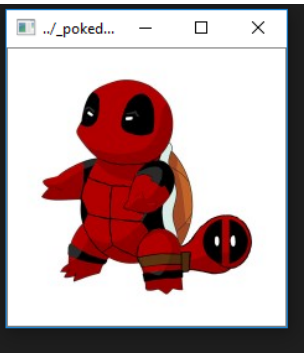



```
[[-1.      0.    ]
 [-1.      0.4691]
 [-1.      0.    ]
 [-1.      0.    ]]
Final Prediction: bulbasaur
```


```
[[-1.000e+00  5.000e-04]
 [-1.000e+00  5.000e-04]
 [-1.000e+00  2.000e-04]
 [-1.000e+00  4.616e-01]]
Final Prediction: squirtle
```




```
[[-1.000e+00  1.000e-04]
 [-1.000e+00  1.000e-04]
 [-1.000e+00  2.000e-04]
 [-1.000e+00  1.181e-01]]
Final Prediction: squirtle
```



Detection Evaluation

```
pikachu
0.3875
bulbasaur
0.625
charmander
0.5416666666666666
squirtle
0.48333333333333334
```

These are the accuracies for each individual SVM:
We could subtract 1 from each and take the absolute value to find the *in sample errors*: [61%, 38%, 46%, 52%]. For in sample, Bulbasaur performed the best while Pikachu performed horrendously.

For the final *out of sample errors*, I created a confusion matrix. The confusion matrix is in this order [Pikachu, Bulbasaur, Charmander, Squirtle, and "Ditto"], with Ditto representing "Unable to identify" or "None of the Above". Here are the results in Precision, Recall, Accuracy, **Error** form:
Pikachu: (0%, 0%, 75%, **25%**);
Bulbasaur: (93.75%, 100%, 98.33%, **1.66%**),
Charmander: (98.33%, 98.33%, 98.75%, **1.25%**),
Squirtle: (100%, 100%, 100%, **0%**),
"Ditto": (0%, 0%, 76.66%, **23.33%**)

```
[[ 0  4  1  0 55]
 [ 0 60  0  0  0]
 [ 0  0 59  0  1]
 [ 0  0  0 60  0]
 [ 0  0  0  0  0]]
```

As we can see from the confusion matrix, there were no errors with classifying Squirtle, and only two errors with classifying Charmander. While the algorithm properly labelled each true Bulbasaur, it also included 4 false positives, mistaking them as Pikachu. Surprisingly, no Pikachu were classified correctly at all, with most being labelled "Ditto". And since all testing data included images of the actual classes, there are no real "Ditto" images.

This image shows the number of support vectors for each SVM:

```
(299, 240)
(297, 240)
(276, 240)
(283, 240)
```

Conclusions and Next Steps

In conclusion, this was a very fun project that incorporated my garnered knowledge of machine learning and computer vision over the semester. Going forward, I'd like to work on creating a more accurate model. I'd start by *image gray-scaling each image as part of the data normalization process*, as it appears that the primary colors of the image greatly influence the ability of the model to make a correct prediction. On that note, there appears to be a significant difference between images containing Pokemon with a white or black background versus images containing a Pokemon with a natural environment background. I'd like to investigate further. Furthermore, ideally, I'd like to *increase my dataset sizes* to have more training points to learn from and more images to test on. I believe I would have to restart from scratch regarding the Pikachu dataset, since it's important to be able to accurately detect everyone's favorite Pokemon! Finally, in the future I can *apply cross validation techniques* to catch errors in the model early on and adjust to learn better without contaminating the model.

Sources

https://medium.com/@muehler.v/machine-learning-with-opencv-and-javascript-part-1-recognizing-handwritten-letters-using-hog-and-88719b70efaa
https://www.researchgate.net/publication/262987479_Efficient_eye_detection_using_HOG-PCA_descriptor
https://www.kaggle.com/manikg/training-svm-classifier-with-hog-features
https://youtu.be/FAr2GmWNbT0?t=176
https://en.wikipedia.org/wiki/Confusion_matrix
https://scikit-learn.org/stable/modules/model_evaluation.html
https://work.caltech.edu/lectures.html
https://docs.opencv.org/master/d1/d73/tutorial_introduction_to_svm.html