05.17.2020

Feature Extraction from MNIST dataset using scikit-learn and scikit-image

Spring 2020 Independent Study – the end semester report

Sriparna Majumdar

Supervisor: Prof. Aaron Brick, Computer Science Department, City College of San Francisco

Background:

I have always been interested in learning how to implement machine learning / deep learning algorithms for biological data analytics and predictions. I started to pursue this interest after I finished my postdoctoral research in experimental neuroscience and data analytics.

A discussion with Aaron in the spring of 2018 got me interested into one of Aaron's long term interests: find better algorithms to extract contextually meaningful texts from natural images. The images may contain scriptures, symbols and writings in any language. The algorithm should be able to extract text strings with high confidence, irrespective of length, language and meaning. Aaron reflected on the possibilities to use Artificial Intelligence techniques to build a high performance optical text recognizing tool that would offer better performance than the existing methods.

Aaron's suggested reading on this topic introduced me to the field of optical character deciphering methods. Optical Character Recognition (OCR[1]) tools can read characters, and strings of words from printed texts with high efficiency, however their performance fall drastically when applied to natural scenes. OCR techniques depend on a segmentation step that correctly separates printed text from background pixels. Success of segmentation is poor in case of natural images because of the variability in the imaging conditions, color noise, blurs etc. There is often far less text and there exists less overall structure in the textual parts in a natural scene, making it harder to separate string of characters from background with high confidence.

Ephstein et al. in their article from 2010 described the challenges of Optical Character Recognition (OCR) of the texts of a natural scene, and discussed their results from implementing a suitable image operator, named Stroke Width Transform (SWT), to extract text efficiently from input images[2]. SWT and similar algorithms use one or more standard filtering methods like Canny edge detection[3], to suppress background, and use logical and flexible geometric reasoning to place similar stroke widths in groups to comprehend bigger components that are likely to be words. In my present study I explored Hough Transform (HT) and its variants to extract features from images[4-6]. HT is similar to SWT in its concept of using flexible geometric reasoning to group points on image plane in regular geometric shapes that are later put together to generate a transformed image more suitable for training and testing an artificial neural network.

In search of suitable machine learning / deep learning tools for extracting texts from natural images, I looked up python's scikit-learn module. I took on the simpler task of extracting individual digits from

photographs of handwritten digits, instead of handwritten texts or texts found in natural images. Using MNIST dataset provided by Yann Lacun[7], I trained and tested a deep (multilayer) hidden network and optimized parameters for the scikit-learn's Multilayer Perceptron Classifier: MLPClassifier[8]. I went on to run the optimized classifier to train and test using Canny filtered or Hough transformed MNIST data. Finally, I combined MNIST original, Canny filtered and Hough transformed features and used MLPClassifier to train and test the artificial network using combined features. The results and future directions are discussed in details below.

Specifications & Requirements:

1. Use MNIST dataset

MNIST dataset of handwritten images comes with 60,000 training and 10,000 test images. They are already reduced to 1bit 28 X 28 pixel greyscale images to facilitate efficient data handing and fitting.

2. Use Python3 :: Scikit-learn

We decided to exclusively use python libraries for the project. Scikit-learn was the library of choice for its ease-of-use interface and generally good quality estimators, that gave fairly high percentage of correct test scores on in-built datasets. A review of most scikit-learn estimators was done using MNIST dataset during the summer of 2018 to validate that fact. Numpy, matplotlib, pillow and scikit-image modules were used for matrix manipulations of the datasets, plotting and extracting features from the datasets.

3. Use Unix/Windows platform

I used hills and Anaconda on my windows laptop for coding and testing. Runs on either platform gave me comparable results, proving scikit-learn to be platform independent.

4. Use supervised Multilayer Perceptron Classifier (MLPClassifier) for modeling an artificial neural network for deep learning

I used exclusively MLPClassifier for classifying the MNIST handwritten digit images into 10 different classes (0 – 9). Parameters of MLPClassifier are fully customizable.

The input and output layers of MLPClassifier come with fixed number of neurons. For MNIST dataset, output layers have 10 neurons, as there are 10 possible classes. Number of input layer neurons (784 for original MNIST) equals number of features in each dataset. The hidden layers contain user-defined number of layers and neurons in each layer.

MLPClassifier is provided with 4 variants of activation functions for the perceptrons[8]: **identity** (f(x) = x), **relu** (f(x) = max(0, x)), **logistic** (f(x) = 1 / (1 + exp(-x)): [0, +1]), **tanh**(f(x) = tanh(x): [-1, +1]). Here values of f(x) are either linearly rectified (identity or relu) or follow a sigmoid nature depending on the values of input x.

MLPClassifier comes with three options for solvers to optimize calculations of loss functions. They are **adam**, **sgd** and **lbfgs**. All of them are either stochastic gradient descent based (sgd, adam) or a variant of stochastic optimizer (lbfgs), but differently defined in their implementations.

Other parameters of MLPClassifier deal with the general performance and efficiency of the classifier and were mostly set to their default values.

Results:

I trained the MLPClassifier using MNIST original dataset containing 60000 images, along with their labels. The default parameters themselves, with just 1 hidden layer containing 100 neurons, gave more than 0.96 of test accuracy. I gradually increased size of hidden layers to up to 9 layers, each containing 50 – 200 neurons. The general idea is to have more neurons in the layers closer to input neurons and fewer neurons in layers closer to output layer. My results indicate that an artificial neural network with 4-6 hidden layers, activation function relu and solver adam gave the best test score. The composition of best performing hidden network was (200, 150, 150, 100, 100, 50), with a test accuracy of 0.98.

I have next used Canny filtered and Hough Transformed MNIST data for training and testing. Canny is one of the most popular and reliable edge detecting algorithms[3]. It efficiently suppresses the background and detects the strong edges in an image by five processing steps: 1. Apply Gaussian filter to remove noise; 2. Find intensity gradient in the image; 3. Apply non-maximum suppression to get rid of spurious edges; 4. Apply double threshold to detect edges; 5. Apply hysteresis to finalize the edges by suppressing all weak edges that are not connected to strong edges. Canny filtering suppressed less important parts of the MNIST handwritten images and highlighted the most important features, thereby improving the training and testing time for MLPClassifier. The test accuracy of 0.92 was achieved at almost twice as fast training and test time, compared to runtime for MNIST original dataset.

Hough transform maps image points on geometric shapes on a transformed plane. In strict Hough Transform, the existence of a point on a projected line ($p = x\text{Cos}\theta + y\text{Sin}\theta$) was counted using a binary counter. In probabilistic Hough Transform (and Fuzzy Hough Transform), points in close proximity to a specific shape are also incorporated with a contribution score $0 < c(i, p) < 1$, where i is a point in image space and p is a point on the Hough shape in transformed space. We used the probabilistic Hough line transform of scikit-image to extract Hough lines from MNIST original and Canny filtered images. In both cases parameters of the Hough Transforms were first optimized for best quality transformed image. Although training and testing using Hough transformed images did not improve run time of MLPClassifier, we still got a fairly high score of 0.95 as test accuracy. This gave us reasons to believe that a combination of Canny and Hough Transformed features along with MNIST original features would give us a perfect test accuracy of 1.0.

We tried merging (averaging pixels) and concatenating MNIST original, Canny and Hough transformed images in various combinations, viz. (MNIST, Canny), (MNIST, Hough), (Canny, Hough) and (MNIST, Canny, Hough). Unfortunately none of the cases did any better than 0.98 in test accuracy that was achieved already by using MNIST original data. Here is the summary of all test accuracies for various combinations of original and extracted data. Only best scores are presented:

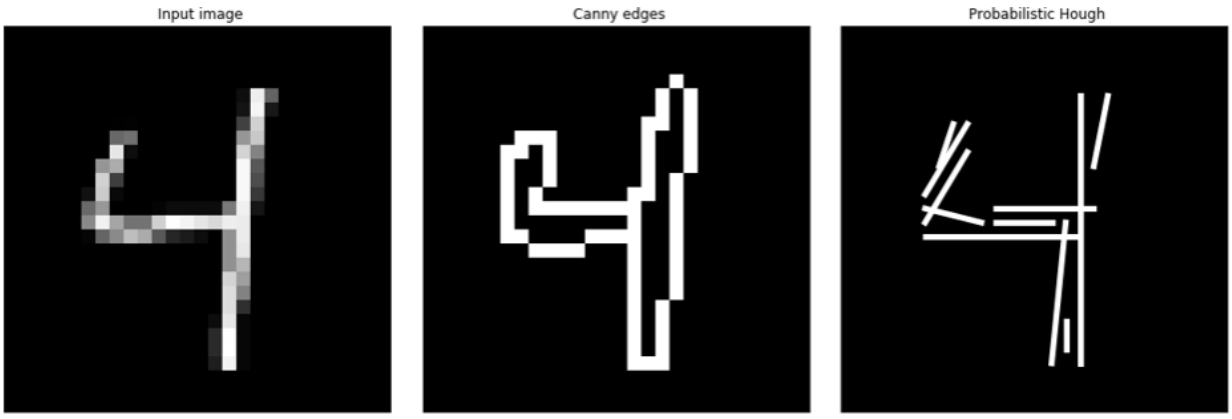
Datasets (MLPClassifier settings: activation: relu, solver: adam, max_iter: 300)	**Test Accuracy** Using MLPClassifier and a deep network of dimension: input layer, (200, 150, 150, 100, 100, 50), output layer
A. MNIST Original	Result of 4 runs: 0.9814/0.9791/0.9795/0.9817
B. Canny Edge Detected	0.9275
C. Hough Transform using MNIST Original data	0.9584
D. Hough Transform on Canny edge detected data	0.9136
Merged Data (averaged pixels): A+B	0.9733
Merged Data (averaged pixels): A+C	0.9750
Merged Data (averaged pixels): B+C	0.9505
Merged Data (averaged pixels): A+D	0.9735
Merged Data (averaged pixels): A+B+C	0.9759
Concatenated features: A+B	0.9739
Concatenated features: A+C	0.9734
Concatenated features: B+C	0.9591
Concatenated features: A+D	0.9762
Concatenated features: A+B+C	0.9753
Concatenated features: B+C+A	0.9746

Below are two examples of MNIST original, Canny filtered and Probabilistic Hough Line Transformed images of handwritten digits:

Label = 4,

Canny edge detector parameter sigma (width of Gaussian) =1

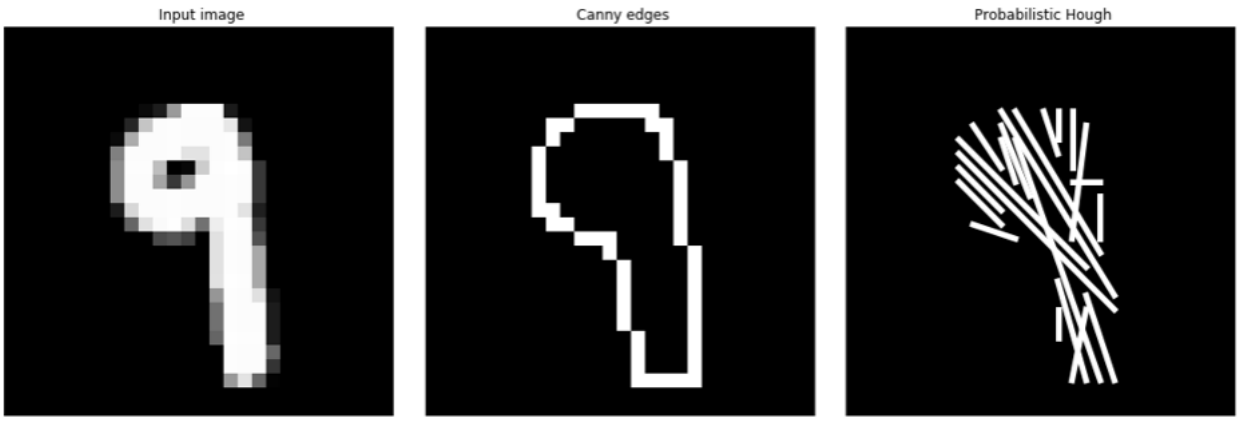
Hough Probabilistic line parameters: threshold=5, line_length=2, line_gap=2



Label = 9,

Canny edge detector parameter sigma (width of Gaussian) = 2,

Hough Probabilistic line parameters: threshold=5, line_length=2, line_gap=2



Conclusions and future directions:

We experimented on extracting new features and combining many features for MNIST images of handwritten digits, in order to get a perfect test accuracy of 1.0 using MLPClassifier and a deep neural network. Although we did not achieve this goal, I have learned about various data extraction algorithms and how to combine them for better train and test efficiency.

One important take home observation was that both generating Hough Transformed images and training and testing using Hough Transformed images were slow processes. Instead of images, using the coordinates of shapes for Canny filtered or Hough Transformed data may increase efficiency. I will explore possibilities of using shape coordinates to train and test data in scikit-learn. The following are my other personal goals for extending this independent study.

1. I plan to check how to implement different non-linear RELU type activation functions (SELU, ELU) in scikit-learn to improve test score, following the article by Dabal Pedamonti[9].
2. I want to understand and implement fuzzy Hough.
3. I want to compare filters like sobel with Canny and Hough Transform.
4. I want to try pooling methods on our concatenated data to increase efficiency and better score.

References:

1. Wikipedia article on optical character recognition: https://en.wikipedia.org/wiki/Optical_character_recognition

2. Boris Epshtein, Eyal Ofek and Yonatan Wexler. Detecting Text in Natural Scenes with Stroke Width Transform. IEEE, 2010: https://ieeexplore.ieee.org/document/5540041

3. Canny Edge Detectors: https://en.wikipedia.org/wiki/Canny_edge_detector

4. Hough Transform: https://en.wikipedia.org/wiki/Hough_transform

5. Peter E. Hart. How the Hough Transform was invented. IEEE, 2009: https://ieeexplore.ieee.org/document/5230799

6. Joon H. Han, László T. Kóczy, Timothy Poston. Fuzzy Hough Transform. Pattern Recognition Letters, 1994: https://doi.org/10.1016/0167-8655(94)90068-X

7. MNIST Datasets were extracted from this webpage: http://yann.lecun.com/exdb/mnist/

8. Documentation on Scikit-learn's MLPClassifier: https://scikit-learn.org/stable/modules/neural_networks_supervised.html

9. Dabal Pedamonti. Comparison of non-linear activation functions for deep neural networks on MNIST classification task. ArXiv, 2018: https://arxiv.org/pdf/1804.02763.pdf

The codes, reading materials, outputs and log of this study can be found on my CCSF google cloud at this link: https://drive.google.com/open?id=1Dcqb4Jvxlynriw2O0wVbnHp2Du25RGCE

Python codes:

1. mnistData_OrigExtract.py

2. Hough-to-gzip.py

3. MLPClass_OrigExtract.py

4. MLPClass_MergeExtract.py

5. MLPClass_ConcatExtract.py

Jupyter Notebooks for testing on laptop:

Test-Hough-on-MNIST2020.ipynb

test-skimage-piDay2020.ipynb

Hough-MLPClass2020-Copy1.ipynb

Test2020.ipynb