TDT4173 Assignment 5

Optical Character Recognition

Overview

I used Python with scikit-learn, scikit-image and pillow. How to install: See install.md

How to run

`python preprocess.py`	Load data set, preprocess images, split data set, augment training data, write HDF5 file
`python training.py`	Load the HDF5 file, train a classifier, store the classifier to a file, run the classifier on the test set, print an accuracy score
`python detection.py`	Load the trained classifier, run character detection on the images inside the "detection-tests" folder, write images with rectangles and characters overlayed
`python test_classifier.py`	Pick some arbitrary images from the data set, print them to the console and attempt to classify them
`python test_preprocessing.py`	Pick some arbitrary images from the data set, preprocess them and create variations of them (as in data augmentation), store them in the "tmp" folder for you to inspect

Preprocessing

Method 1: Noise reduction and edge detection

Seeing that the images were noisy and some were inverted (white on black vs. black on white), I thought noise reduction and edge detection would be a good idea.

I used a bilateral filter for noise reduction. The good thing about this filter is that while it smoothes out noise, it does not blur edges.

> *A bilateral filter is an edge-preserving and noise reducing filter. It averages pixels based on their spatial closeness and radiometric similarity [0]*

I used a sobel filter for edge detection. This filter combines multiple 3x3 kernels that are convolved with the image. This approximates the gradient of the image intensity.

Result

I quickly found that noise reduction and edge detection wasn't a very good idea. The noise reduction alone seemingly didn't affect the predictive performance at all, while applying sobel edge detection actually hurt predictive performance.

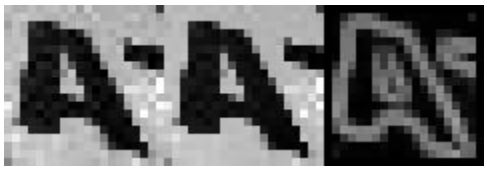

Figure 1: original, noise reduction, noise reduction + sobel filter

Method 2: Contrast stretching, rescale values

Because some images had low contrast, I thought it would be a good idea to kind of normalize those images. Also, I decided to rescale all intensity values from [0, 255] to [0, 1]. This is how the contrast stretching works: Find the 15th and the 85th percentile to obtain the dark and the bright threshold, respectively. Then rescale the intensity values so that these two intensity thresholds become black and white, respectively. This improves the separation between the background and the foreground.

Result

Given that I use a random forest classifier with 30 trees and no data augmentation:

Without contrast stretching	62.7 % accuracy
With contrast stretching	**64.8 % accuracy**

Figure 2: original, contrast stretched

Data augmentation: Rotation, inverting

I tried augmenting the training set by creating combinations of rotated (-10, -5, 5 and 10 degrees) and inverted images. This makes sense because some of the images in the data set are inverted, and orientation is not always perfect.

Results

The data augmentation method significantly improves accuracy. Given that I use preprocessing method 2 (contrast stretching) and a random forest classifier with 30 trees:

Without data augmentation	64.8 % accuracy
With data augmentation	**72.2 % accuracy**

Figure 3: Leftmost image is preprocessed (contrast is stretched). The other images are various combinations of rotation and inversion.

Techniques I want to try

If I had to do more work on this project, I'd try Histogram of Oriented Gradients (HOG) and Scale-invariant feature transform (SIFT). Those are algorithms that can describe local features in images.

Models

When I first looked at the dataset, I initially thought that a support vector machine would work well. But then I started doing lots of data augmentation, which makes the data set too large for the support vector machine.

K nearest neighbours

The K nearest neighbours technique is based on finding the K images that are most similar to the image you are trying to classify. Those the most common class among those K images becomes the predicted class. This model is easy to understand, and it yields good accuracy. Although the training time is low, it takes relatively long to

classify an image with this technique. The reason is that for each classification the algorithm iterates over the entire training set to find the K most similar images.

3 nearest neighbours	78.7 % accuracy
5 nearest neighbours	**79.2 % accuracy**
10 nearest neighbours	78.8 % accuracy

Ensemble methods

Ensemble methods are based on training multiple weak classifiers (decision trees) and combining their predictions into a single prediction. These are good because

- The training is fairly quick (many threads can be run in parallel)
- They can output probabilities for each class
- Classification execution time and accuracy is pretty good
- I tried random forest and extremely randomized trees (both with n_estimators set to 30). I quickly found that the latter was better. I also found that increasing n_estimators improved accuracy.

Random forest, n_estimators=30	72.2 % accuracy
Extremely randomized trees, n_estimators=30	76.6 % accuracy
Extremely randomized trees, n_estimators=150	**80.7 % accuracy**

Additional models

I could try running the data through an artificial neural network. Perhaps a *convolutional* neural network. I've heard that those easily get over 90 % accuracy on this dataset.

Evaluation

Both K nearest neighbours and extremely randomized trees worked well. The latter is best because it yields more accurate probabilities (they are based on 150 estimators) and has better accuracy on the test set. How their performance are measured: predict class for each image in the test set. The percentage of correctly classified images represents the accuracy of the classifier.

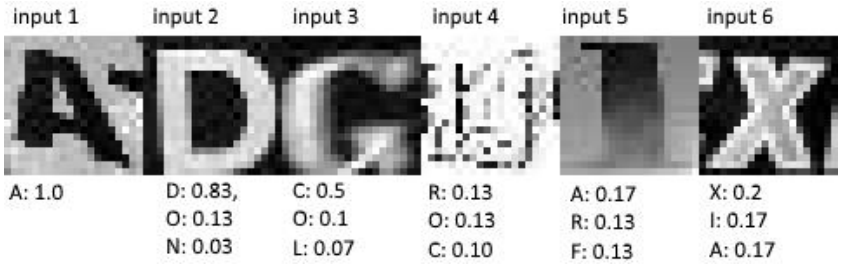

Figure 4: Some samples that were classified correctly and some that were classified incorrectly

Character Detection

The weakest component of my OCR system is the character detection. The classifier assumes that there is a character on the image and the output probabilities sum to 1. The preprocessing also strongly enhances any low-contrast images that may be just noise or weak shadow gradient. These are both problems when using the classifier for character detection/recognition.

In the character detection problem the assumption "there is a character on the image" does not make sense, as many images will not have characters in them. As I see it there are several ways to fix this:

1. Add a class that is "not a character" and train the classifier on that. This requires extra training data. For example one could create a few thousand images of perlin noise with varying roughness.

2. Different kind of preprocessing and feature engineering. For example the amount of contrast and the amount of sharp edges on the image could be useful for checking if there's a character or not.

3. Train n independent classifiers where the output probabilities are independent of each other (i.e. they do not necessarily sum to 1). That way each probability represents for example "what's the probability that the image contains an A" rather than "what's the probability that the character on the image is an A".

I tried approach number 3, and while that worked, I was unable to store and load the classifier, as it would quickly become way too large (0.22 GB with n_estimators=5. With n_estimators set to 150, the stored file would be larger than 6 GB) to handle with cPickle. Sadly, there's a bug in scikit-learn that hinders Windows users from storing tree-based classifiers with joblib, which stores classifiers in a much more efficient way than cPickle. [1] Therefore, I didn't use the n classifiers approach. Hence I expect that the OCR is going to be suboptimal.

Evaluation

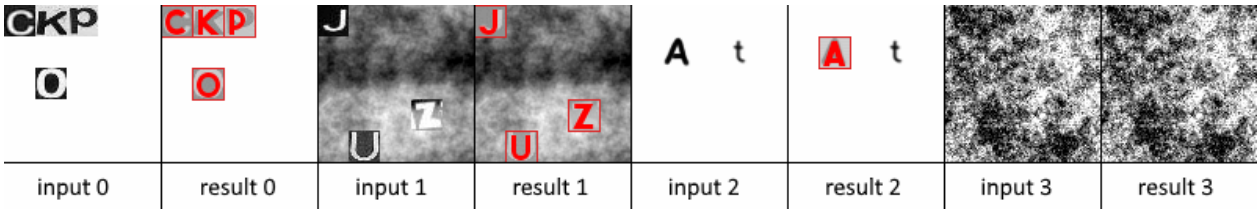

Figure 5: My OCR system seems to be good at classifying images of characters from the data set. Not particularly good on my "hand written" characters (input 2). See that it misses that t. I tried lowering the threshold, and then it would find and classify that t correctly.

Figure 6: I googled "OCR test" and found this image. It looked fairly simple, so I thought I'd give it a try. Since characters don't align perfectly with the 20x20 window and 20x20 stride, my OCR performs badly on this image. I used a small stride and a high threshold. The reasons why it performs so badly is:

- Given the small stride, lots of the images are going to be of partial characters
- The classifier assumes that for each window there is a character and it is in the middle of the image
- The threshold is high, so it misses a lot of the characters

Lessons learned

- Edge detection is not so useful as the only preprocessing
- Data augmentation is a good idea, but makes the training take longer
- Extremely randomized trees is indeed better than random forest
- OCR is hard, and one cannot simply use a classifier that is not created for the purpose

References

0: http://scikit-image.org/docs/dev/auto_examples/plot_denoise.html

1: https://github.com/scikit-learn/scikit-learn/issues/6650