TDT4173 - Assignment 5

Håkon Collett Bjørgan - haakoncb
Lars Christian Ek Folkestad - lcfolkes
Mathias Dehli Klev - mathiadk

MTIØT

November 16, 2019

Introduction

Our system consists of three main components. They are separated into their own files: *preprocessing.py*, *ocr.py* and *character_detection.py*. *preproccessing.py* handles loading the data, splitting it however you wish, and augmenting the data if would like to do that. *ocr.py* handles the model training and scoring/ plotting of scores of different models. You can also persist your model from *ocr.py* if you do not want to retrain your model all the time (which can be quite time consuming). *character_detection.py* uses a trained model from file to look for letters in images.

Along with some minor helper libraries, the only library we really leverage to get our results is scikit-learn. We fetch all our models and more from this library.

How to run our code:

We assume you have python3 and pip installed. If not, please download them first. In your terminal, cd into our folder and run the following command: *pip install -r requirements.txt*. This will install all dependencies. Next run *ocr.py*. It is an executable, so the command *./ocr.py* should do the trick. This will train a neural network and print the score for you. Next run *character_detection.py* (command: *./character_detection.py*). This will use the model you just trained on the second image, and plot the result for you. For further configuration you will have to open the code in an editor and alter the parameters yourself.

Feature Engineering

In our quest to get the highest possible accuracy for a classifier, we attempted several pre-processing techniques. Data augmentation and dimensionality reduction are the ones we gave the most effort.

Data Augmentation

While it is debatable whether or not data augmentation qualifies as feature engineering, it is definitely part of the pre-processing, so we are including this here.

As briefly mentioned in one of the task's footnotes, data augmentation is the process of artificially increasing your dataset by slightly tweaking your existing data. In our case, we implemented three methods for tweaking the images. One rotates the image a random degrees between 0 and 25, one adds some noise to the image, and one flips the image 180 degrees.

We chose this technique because we found that the models we trained consequently underperformed for certain letters. We believe this was because the dataset is quite imbalanced. For instance, 'a' has 715 entries in the set, while 'q' only has 88 entries. By augmenting the data and setting a fixed limit for how many entries each character should have, we attempt to make up for the imbalance in

the original dataset.

How well our augmentation works is up for debate. We have 80% as a reference, as this was our best result without the augmented data. By augmenting the data before splitting into train and test sets, we were able to reach an overall accuracy of 88%. We did however see that the characters that were most heavily augmented ended up with the highest scores, and as such we believe these results to be invalid. Most likely we ended up having entries in the test set that were very similar to entries found in the training set. When augmenting the training and testing sets separately, we only saw results that were worse than if we used the original dataset. With this being said, it is very possible that data augmentation with other methods than ours could improve a model trained over this dataset.

Dimensionality Reduction

We reduced the dimensionality of our images through using scikit-learns built in principal component analysis (PCA) tool. PCA is essentially a technique for capturing the essence of data in dimension 'x' and representing it in dimension 'n', n¡=x. Here n would be the number of principal components you chose to keep. PCA analyses the data and creates a linear combination of the features to form the principal components. The dimensions (features) with the most variance will affect the principal components the most.

The idea behind dimensionality reduction was that the more dimensions you have, the more data you need to train your model accurately. As is most often the case in real life, and in this task, data is scarce. As such we attempted to reduce the dimensions to reduce the need for data.

Loading the data five times with PCA enabled and five times without PCA enabled, we found that the accuracy of the model increased slightly, 2%, when PCA was enabled.

Noise Removal and Threshold

We attempted both to remove noise from the images, and to represent the images in black and white. The latter was implemented by feeding all pixels to a function that would set the pixel to 0 if the value was less than 127, and 255 if it was greater. We implemented it to see whether or not the model would more easily identify letters if the distinction between the letters and the background was stronger.

We did not find any noticeable difference when using noise-removal, and the performance of our models dropped by about 5% when we used the binary (black and white) representation.

We would have liked to try HOG, but ended up not finding the time to do so.

Character Classification

From having watched videos on YouTube, we were somewhat biased towards a neural net, as the those are often mentioned while explaining image recognition and computer vision techniques. Also we found it unlikely that the data was linearly separable, so we had to use a model which would be able to take this property into account.

The first model we elected to use was an MLP (multilayer perceptron) which is a type of artificial neural network. We use one hidden layer with 850 neurons, in addition to the input and output layer. We landed on 850 after a bit of trial and error in order to tune the parameter. There is no correct answer to how many neurons the hidden layer should provide, so we figured our approach was as good as any. It is worth mentioning that a rule of thumb is that the most optimal number of neurons in a hidden layer is between the number in the input layer and output layer. That was not the case for us.

The MLP model is trained by getting an input vector of data and passing on the values based on activation functions for the neurons and weights in the network. Once the data reaches the output layer, it is checked against the actual value. Based on the results, the weights in the network are updated using a technique called backpropagation. In essence the goal is to minimize the cost function, which is a measure of how well/bad your network performs. As mentioned we chose this model because of its frequent mentions in videos we have seen on the topic, and the possibility to separate non-linear data.

The second model we chose was k-nearest neighbors. The reasoning behind this choice was that a letter is likely written somewhat in the same way several times. Therefore the closest neighbors to an incoming letter should be a good guesstimate to classify it.

Unlike the MLP, k-nn is not trained. Instead it is loaded with the available data. When a new instance is entered for classification, the algorithm goes through the available data, and calculates which are the closest. Based on your configuration, the k-nearest neighbors, will vote, either weighted or uniformly, on what to classify the new instance as.

Both models' performance was measured using 5-fold cross validation. The five in order to make the 80/20 partitioning as required in the task. Cross validation is a way of eliminating the risk of being lucky/unlucky with what data was chosen as training data, and what was chosen as test data. It eliminates this risk by using each fraction, in this case there are five fractions, as the test data. You then can calculate a mean and standard deviation from the results. We used sklearn's built in $cross\_val\_score()$ for this.

In the following graph the accuracy for k-nn is shown as a function of its neighbors. We used uniform weights on the voting. As the graph indicates, the optimal number of neighbors was one, and that had an accuracy of 73,4% +- 1,4%.

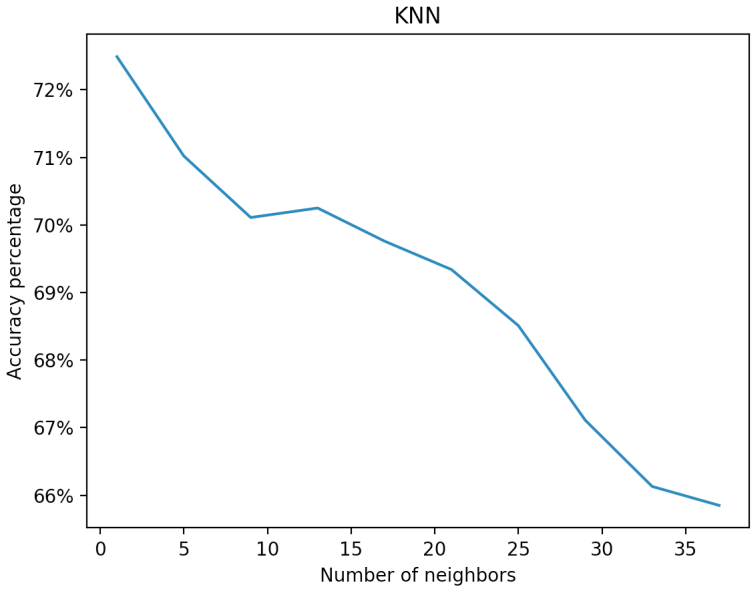

Figure 1: Accuracy as a function of neighbors

Our most accurate model was the MLP. With 850 neurons in the hidden layer, PCA to 40 components and no data augmentation, we achieved an accuracy of 83,5% +-0,76%. Not super great, but not catastrophic either.

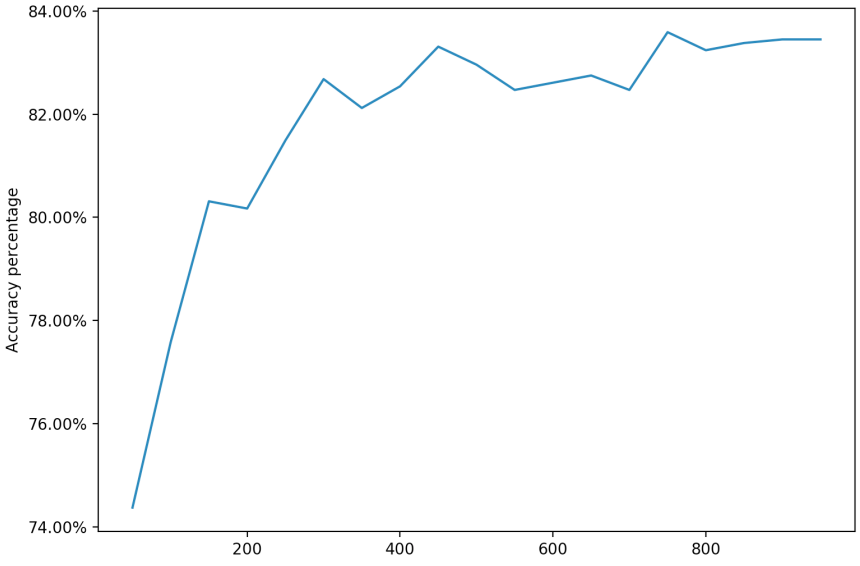

Figure 2: Accuracy as a function of neurons in the hidden layer

Some of the results from our MLP can be seen in 3 and 4. Allthough it is only a small outtake, it gives us some insight into what our model is able to classify. We can see that the clear-cut images are correctly labeled. The more difficult examples are where it fails.

As far as we can tell, deep learning is *the* way to go when it comes to image recognition and

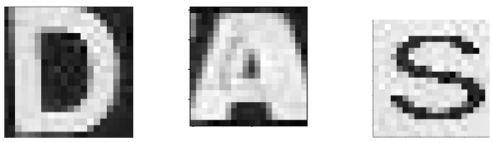

Figure 3: Images correctly classified by our MLP

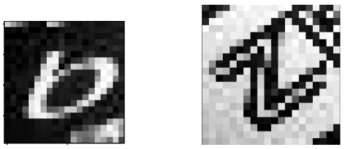

Figure 4: Images incorrectly classified by our MLP. The image on the left is a b, but was classified as n. The one on the right is a z but was classfied as a y.

computer vision in general. We would therefore have liked to try a deeper neural net. But as neither of us really are familiar with it, time is short, and the task specifically recommended not going for deep learning unless we were somewhat experienced with it, we decided that we would stick to the models we were more familiar with.

Character Detection

Detection of characters is performed in the following steps:

- **Sliding window:** A window of size=(20,20) is slided over the input image, with a certain step size. In order to get the best results, we adjusted the step size through trial and error. A smaller step size will generally give more accuracy, but at the cost of performance.
- **Classify window:** Every window is classified with the trained model from the Character Classification assignment. `sklearn's` function `predict_proba()` returns an array with the probabilities our model predicts for each class, i.e. letter. We then choose the one with the highest probability if it is above a certain threshold.
- **Remove duplicates:** After we have collected the windows with a high probability of correctly classifying a letter, we must remove duplicates. This is done by grouping together windows with a euclidean distance below a threshold, i.e. in close proximity. We then choose one window in each of the "clusters", namely the window with the highest classification probability.

Results - detection-1.jpg

As we can see from figure 5, our character detection algorithm works very well with the first test image. All characters are detected and labelled correctly.

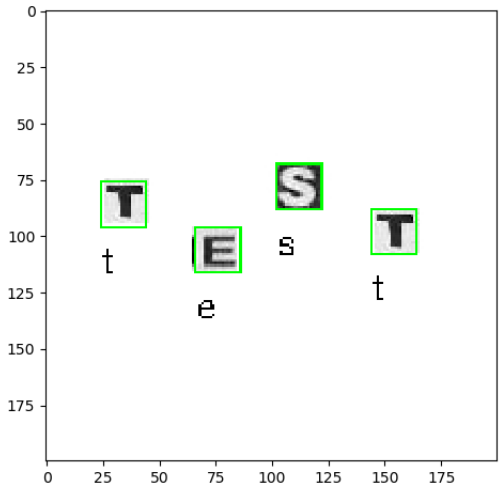

Figure 5: Character detection of `detection-1.jpg`

Results - detection-2.jpg

As seen in figure 6 most of the characters in the second test image were located correctly, however with certain deviations. Moreover, many characters were classified incorrectly. The reason for this is probably twofold. The first one is the methods used to choose the windows that end up being included. If we had more time, we would be able to augment this to choose only windows that are placed directly onto a letter, making the classification more precise. The other reason is insufficient training of the classifier. Obviously the accuracy of the algorithm would be higher with more training.

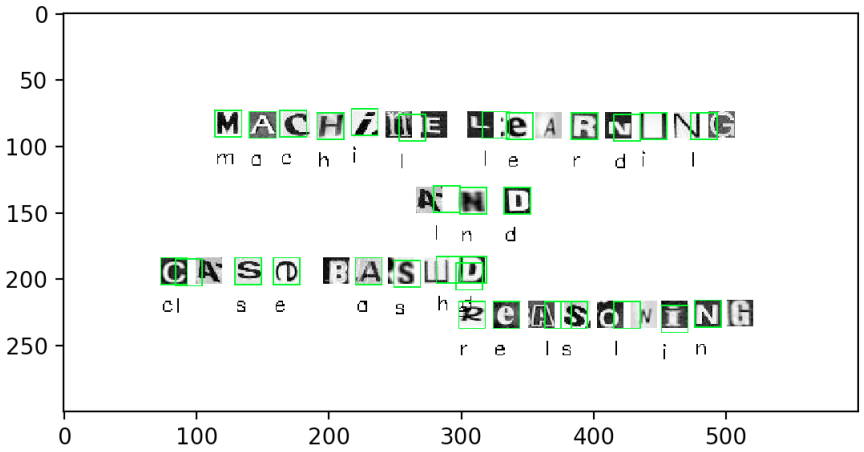

Figure 6: Character detection of `detection-2.jpg`

Conclusion

The weakest parts of our system is the data augmentation and the character detection. Data augmentation is not being used as of now, because we were unable to implement in in a way that contributed positively. The character detection part also has more potential for improvement. With more work, it can be improved on areas like duplicate avoidance and only choosing characters that are centralized in the window.

The strongest part of our OCR-system is the character classification, and especially the choice of model itself. We believe that 83,5% on such a slim data set is quite a good result.

We are happy we chose to make our system in python and using scikit-learn. We saved a lot of time using something we were all familiar with, and, let's be honest, we wouldn't be able to make anything close to sklearn's implementation in terms of efficiency and usability. Some lessons were learned. Firstly, that your model will only get as good as the data you feed it. Secondly that when performing data augmentation on letters, flipping the images might not be the best of ideas. A `p` looks an awful lot like a d when turned upside down.