A Classification model of recognizing letters in images: *Application on notMNIST dataset*

Yuheng HUANG*

November 2018

Abstract

This paper introduces a classification model to recognize letters of all sorts of shapes in images. Dataset for this work is notMNIST, which contains over 600 thousand images of resolution $28 \times 28 \times 1$. One third of images are randomly chosen to the subsample of training set, valid set and test set. To extract the information from images, "imageio" module of python was applied and thus all the images were transformed to 28×28 matrices and then the pixel values were normalized to the range from -0.5 to 0.5. The sample sizes of training set, valid set and test set are 200 thousand, 20 thousand and 10 thousand respectively. Then a filter to trim blank images and similar images was applied to valid and test sets, around 10% of the sample in the two sets being removed. Cleaned data are saved in feather format by functions in "feather" and "pandas" modules. The convolutional neural network was contructed using pytorch module and consists of two convolutional 2D layers, two maxpool 2D layers, two linear layers with "relu" activation function, and lastly a dropout layer. Overall accuracy of the network on test set is around 94%, while there still seems to be room to increase the accuracy rate.

*Ph.D candidate, Graduate School of Economics, University of Tokyo

1 Overview

Image classification has been a classical problem in the field of machine learning and lots of literature has contributed to this field. Along with the development of the researchk, all sorts of datasets are also constructed for the specific purpose of training classification model. As one of those datasets, MNIST with a large amount of handwritten digit is one of the commonly used datasets for training models of image classification. There are also many derivatives of this dataset, and notMNIST is one of these. Since using the raw data might lead to some unpredictable problems, data cleaning must be done before the training of model. The result of the baseline model, though not shown in the paper, was around 80% of accuracy rate, with one layer and "relu" activation function, and the original API, tensorflow graph and session, consumed too much memory and eventually got low accuracy rate if memories ran out. Even if I have to admit tensorflow is the fastest API among the three I have used for the research (the other two are Keras and Pytorch), the manipulation of memory and other settings are more difficult and thus not user-friendly. Switching to convolutional neural network and introduction of learning rate decaying, dropout layer and regularization increased the accuracy rate to 94% at the end of the paper.

2 Dataset

2.1 notMNIST

There are 10 letters from "A" to "J" in the notMNIST dataset with around 60 thousand images for each class.

All the images are monochrome, thus the depth will be one. The raw data are not cleaned, and thus there are many blank or repete images in the dataset. In the compressed file, each folders named by letters contain the corresponding images. See *Figure 1* for the examples. Some images that do not belong to any classes also show up in the data, while I don't have a good solution to this problem but have to leave them as noise.

2.2 Data import and export

Image data are imported to 28×28 numpy matrices by "imageio" module.

Figure 1: Examples from the raw data

All the pixel values are subtracting 127.5 and divided 255 such that their values are in the interval of $[-0.5, 0.5]$. The amount of images for each class is around 60 thousand. Then one third of the data are randomly picked up from each class, along with the corresponding labels denoted by the name of the folder, to three subsets of training set, valid set and test set. (Vanhoucke, 2016)

Next, I bind all the image and label data from different classes and sets together and shuffle the first column of 3D image data and label data respectively. The construction of dataset(s) is now almost complete.

Lastly, The image data are reshaped to a 2d array of $((n \times 28), 28)$ where n is the sample size and converted to pandas dataframe format with label data. All of the data are saved in 6 feather file and prepared for next step.

In the following steps, those feather files can be rapidly imported again and converted to the specific numpy ndarrays by several simple transform.

2.3 Data cleaning

In order to clean the data for a better performance of the model, I write two filters, one is for blank images and the other is for similar images.

The first filter computes the summation of all pixel values in 28×28 matrix for each image. When the summation is close enough to 392.0 (or -392.0), which is equal to $0.5 \times 28 \times 28$, the image is considered completely black or white and will be excluded from the dataset. This filter deleted around 1% of the sample in valid set and test set.

The other filter works in a more complicated rule. For each image in valid set or test set, the algorithm has to search for the whole training set and match similar image. Once there is a similar image in training set, this image will be deleted from valid or test set. The general idea is to flatten the matrix of each image and compute the euclidean distance between two vectors of two images.

Considering the scale of the data, examing all the data one by one will result in a large amount of computation. The original algorithm that loops over all the data

costs more than 2 hours. Limiting the search zone in training set by the specific label helps accelerate the program a bit but it is not literally fast enough. Since there are two loops in the algorithm, it is feasible to vectorise one of both. I choose to vectorise the inner loop which runs at most 20 thousand times. This reduces two thirds of the computation time and the fast record is about 20 minutes.

Alternatively, using image HASH is a good choice because one image only has one identical scaler HASH value. This approach saves the time of computing euclidean distance (or distance defined by other norm). However, it is too late to apply this method at this step, since it requires extracting the HASH information of images directly. It would be better to do this along with extracting the pixel values.

2.4 Other lessons

As a relatively low-level API, tensorflow does not provide with convenient function to import image or other forms of data, whereas Pytorch and Keras have their only way of importing data by one or two function. Even beginners could import data by those high-level APIs with proper functions and parameters within. In Keras, numpy arrays can even be used as input directly without transformation. These conveniences really make life a better one, while I think it is not pointless in doing those steps by hand. It deepens the understanding of the whole process and might be a good experience for incidences in the future (e.g. debugging).

3 Models

3.1 Baseline model

The choice for the first model is logit model. Technically, there is only one linear layer with softmax activation function (Wicke and Vanhoucke, 2017).

The API to train the model is tensorflow. This API was introduced in the online course I was attending and is the one I was most familiar with at the beginning. In this API, not only the input data ,neural network and optimizer but also the weights and bias (hyperparameters) are defined in the tensorflow graph. Everything in the graph has to be defined carefully because it contains almost the whole model in a detailed manner. I had a hard time dealing with this API as defining everything properly is a nearly impossible task for a beginner. After having tried three APIs of

Table 1: Accuracy rate of each classes

Class	A	B	C	D	E	F	G	H	I	J
Accuracy Rate	94%	95%	95%	94%	94%	93%	95%	94%	88%	95%

distinct levels, I found tensorflow the fastest one if there is sufficient memory. though it is still a pain to write the whole model by defining everything by oneself.

The accuracy rate of the baseline model is about 80%. This might be a little higher for such a simple model, but if you take the complexity of the sample into account, the result is not that ridiculous. Firstly, the images are all monochrome, thus the depth shall be one. Considering the case of RGB colorful images, the information for one image will triple. And also the letters are simpler than other objects because only shapes matter. Both reasons can explain the high accuracy rate of the baseline model.

3.2 Other trials

I tried many other models, basicall fully connected models, before reaching the destination.

Adding more linear layers with relu activation maintains higher accuracy as expected, while the accuracy rate still can not exceed 85%. Another straightforward way to modity the model is have more neurons on each layer. The process of training the model becomes slower as the number of neurons grows. With all the endeavours above, though the time it takes to train the model is 3 times more than the baseline model, the efficiency gain does not match the cost. Because of the simplicity of the data, it is hard to say a model is a good predictor once the accuracy rate is below 90%. By the information from the online course, the highest accuracy rate based on the specific data is 97%. Given the restuls by far, there is a high probability that I was in the wrong direction, so I have determined to make the model a brand new one instead of making slight changes to the simple baseline model.

3.3 Final version of the model

In this subsession, I used Keras and Pytorch as the APIs to train the model.

In the third section of the online course, Sebastian and Vanhoucke, 2016, there are generally three ways to improve the goodness of fit of the model, (1) drop out layer,

Algorithm 1 Convolutional Neural Network

```
Net(
(conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
(conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
(fc1): Linear(in_features=256, out_features=120, bias=True)
(drop): Dropout(p=0.2)
(fc2): Linear(in_features=120, out_features=84, bias=True)
(fc3): Linear(in_features=84, out_features=10, bias=True) )
```

(2) regularization and (3) learning rate decay. I tested all the three strategies solely and their combinations. The second one adds a penalty term of weights (hyperparameters) to the loss function, thus prevents the model from having some extreme weights because of cases in the training set that are not general. Regularization at the same time increases the burden of computation, too. Given the trade-off between computing time and its accuracy gain, I have decided not to adopt this strategy. Although at present I still have little idea how drop out layer works in the model conceptually, its contribution to the model is undeniable even when accuracy rate is as high as 92%. There are many types of learning rate schedulers in torch.optim module. For simplicity, I choose to have different learning rates in each epoch, not reduce learning rate after single or several iterations.

Keras is the highest-level API among the three. it has almost everything you need in constructing a deep neural network, and those functions and attributes reduce the length of code dramatically. Nevertheless, I have to say it is not suitable for a beginner as I might look like. It took me a large amount of time to calibrate (or adjust) the model because I could not even find where the mistakes were. Some implicit rules of those functions got in my way frequently, too. At last I have to give up since the efficiency of work is really low. I might turn to this API again after I become a veteran of deep learning.

The final version of the model is composed of a convolutional neural network with a drop out layer in the fully connected part, two linear layer with relu activation and a learnign rate decaying scheduler. The overall accuracy rate of this model on test set is 94%, and the accuracy rates for each classes are in *Table 1.*

The detailed structure of the network is in *Algorithm 1*. As we can see from the strcuture, the depth of the input increases from 1 to 6, then to 16 before the fully connected layers. The other parts are general fully connected layers and a dropout layer that forces the model to be more general. The performance of this convolutional neural network is even better than expected. The examples I saw for this structure are usually for images with RGB feature or other data with depth larger than 1. I was afraid that the convolutional neural network wouldn't work out because of the depth of the monochrome images.

4 Conclusion

The flow of the paper starts from extracting image information by "imageio" module, then saves and imports the data by "feather" and "pandas" module, lastly trains a convolutional neural network by "Pytorch". Although eventually the accuracy rate is still 3% lower than the record set by other programmers, it is still high enough to satisfy a python and deep learning beginner. Next time if I have an opportunity to train my deep learning model, I want to challenge other data other than images, and I am also looking forward to learning recurrent neural network some day.

References

Sebastian, Raschka and Vincent Vanhoucke, "tensorflow/3_regularization.ipynb at master_tensorflow," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/ examples/udacity/3_regularization.ipynb 2016. Accessed: 2018-11-18.

Vanhoucke, Vincent, "tensorflow/1_notmnist.ipynb at master_tensorflow," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/ examples/udacity/1_notmnist.ipynb 2016. Accessed: 2018-11-18.

Wicke, Martin and Vincent Vanhoucke, "tensorflow/2_fullyconnected.ipynb at master_tensorflow," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/ examples/udacity/2_fullyconnected.ipynb 2017. Accessed: 2018-11-18.