Find the “largest” digit

Comp551-002, Winter 2018

Richard Santiago
Richard.santiagotorres@mail.mcgill.ca
ID: 260583280

Rony Azrak
rony.azrak@mail.mcgill.ca
ID : 260606812

Mandana Samiei
mandana.samiei@mail.mcgill.ca
ID: 260779555

Kaggle Team Name: RichManRon

I. Introduction

We consider the problem of finding the digit with the maximum area in an image containing two or three digits of different sizes. To be more precise, by area of a digit here we mean the area of the rectangle which encompasses the digit. The provided dataset consists of 50k grayscale images of size (64,64) for training and 10k for validation. Examples of training examples are shown here:

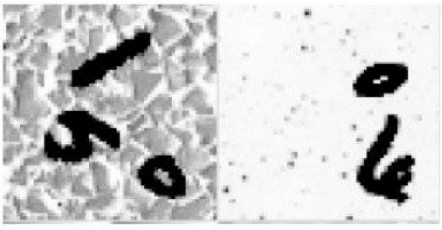

Figure 1- An example of training set images

At the high level our approach consists of reducing an instance of the original problem to a MNIST one via a thorough preprocessing step. Then, we can make use of the extensive available literature and methods to solve the MNIST handwritten classification problem. The methods we tried include Logistic Regression and Linear SVM (as baseline linear learners), a fully connected feed forward neural network trained by backpropagation, and several other methods like k-NN, random forests, kernelized SVM, and CNNs. Among these, the latter gave the best performance with an accuracy of 94%. Moreover, given that we are essentially training our models on data that looks exactly like the MNIST dataset, we also tried to enrich our training data by adding some of the images from MNIST. This did not seem to significantly affect (in either way) the performance of our top classifier, although it did allow us to gain an extra 0.5% improvement that pushed the accuracy closer to 95%. Further details about the preprocessing, supplemental data, algorithms, and results are described in the next sections.

II. Feature Design

Preprocessing is a key factor in computer vision problems, we used many image processing methods. Mainly, our preprocessing consists of the following steps: 1) We first binarize the images, setting the background to zero and all digits to 1 as in MNIST. For this purpose we use a thresholding function. We also tried Otsu thresholding but this did not seem to work well, since for some instances it was not removing the background correctly. 2) We detect the largest digit in an image by looking at the area of the smallest square which encompasses each digit. We use the findContours function from openCV to find the contours, and then use the boundingRect function to fit the tightest square. We remark that we also tried to fit the smallest rectangle (as originally suggested in the statement of the project), however, we found that fitting a square gives better results. One example of this is the image below, where fitting a square selects the correct digit (number 1) while fitting a rectangle does not (it selects 9).

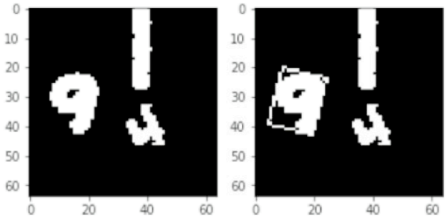

Figure 2- rectangle bounding

Once the largest digit is found, we remove the other digits from the image.

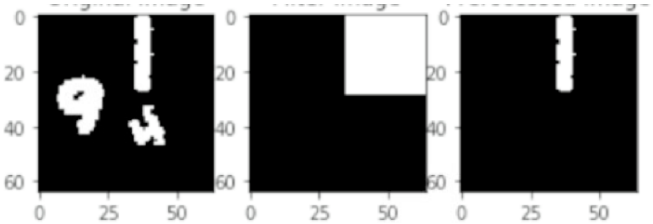

Figure 3- Filtering and square bounding

3) We shift the largest digit to the center. This is done by finding the centroid of the largest component and shifting it to the image's middle point (32,32). 4) We resize the image to (28,28), since this is the size of the images in the MNIST dataset. 5) Finally, we deskew the image using its second order moments. The whole preprocessing step looks as follows:

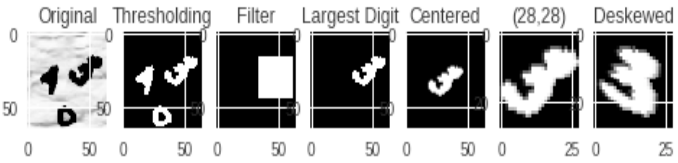

Figure 4- All preprocessing steps in one image

III. ALGORITHMS

We implemented Linear SVM and Logistic Regression as baselines learners by using sklearn libraries.

1) *Linear SVM* is a parametric algorithm, it tries to maximize the margin between support vectors and finds a linear decision boundary that maximizes the margin. SVM assumes data is independently and identically distributed, it takes a penalty parameter C of the error term, that shows how sensitive are we to wrong classifications, also it takes dual which we set it false because number of examples are more than features. A key limitation is that, if the data is not linearly separable, SVM fail at properly classifying data.

2) *Logistic Regression*: LR maximizes posterior probability, it takes C as a parameter. LR improve predictive performance when class-conditional density assumption give a poor approximation to the true distributions.

3) *Kernelized SVM* : we implemented a kernelized SVM with 3 different kernels, poly, rbf and sigmoid. Kernelized SVM try to map data to different space w.r.t to the kernel, for example in rbf it maps data to gaussian hyperplane and as a result the corresponding decision boundary is non-linear.

4) *KNN*: We also implemented KNN, it is a non-parametric algorithm. KNN looks at the k nearest neighbours and gives a majority of labels as the prediction result. It takes k as a hyperparameter and makes decision based on the entire training data. KNN gives a non-linear decision boundary

5) *Random Forest*: it uses k bootstrap replicates of data to train k different trees, at each node, pick m features at random, determined the best test, 6) *Neural Network*: We implemented a simple Neural Network that runs mini batch gradient descent, with sigmoid as the activation function. The algorithm takes as inputs the batch size which helps us test stochastic gradient descent as well as regular gradient descent. It also takes the learning rate, the number of hidden layers as well as the number of nodes per hidden layer. The computations are done with matrix operations, taken from the class notes, in order to speed up the process. Before training the neural net, preprocessed the images, leaving only the largest digit and centering it. A simple neural net would need good preprocessing to be effective. We also normalized all pixel values. Finally, we run the neural net computations for a constant number of 100 epochs.

7) *Convolutional Neural Network*: Convolutional neural networks ("CNN"), or multi-layer, neural networks where layers applying a convolution in the mathematical sense. A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers. The algorithm can focus on some data features that only depend on a region of the image that is local receptive fields. In CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map. There is different kinds of activation function can be used in CNN which are Relu, LeakyRelu, tanh, softmax, etc. As the number of layers and number of nodes in hidden layer increases , it prones more to overfitting.

IV. METHODOLOGY

We now discuss for each of the three methods presented in the previous section our decisions about training/validation split, regularization strategies, hyper-parameters tuning, and optimization tricks. We postpone the graphs and performance discussion to the Results section.

Our training/validation split always consists of randomly selecting 80% of the data for training and the remaining 20% for validation purposes.

For the case of linear and kernelized SVM we perform a thorough hyper-parameter tuning. We optimize over the regularization parameter C, the type of loss function, and the type of penalty function. Moreover, in the case of kernelized polynomial SVM we optimize over the polynomial degree as well. On the Results section we present a table with the exact range considered for each hyper-parameter, the optimal combination of hyper-parameters chosen, and the corresponding errors. We remark that we solve the primal problem since the number of features (i.e. 28x28=784) is much smaller than the number of samples (50k).

For Logistic Regression we optimize over the regularization parameter C, while for k-NN we did over the number of neighbours.
In the case of random forests we tuned the number of trees in the forest, the maximum depth of the trees., and the number of features to consider when looking for the best split.

For the simple neural network we perform cross-validation to determine the optimal number of hidden layers, the number of nodes inside each hidden layer, the learning rate and the batch size. We present in the next section a detailed table with the range of values we considered, the optimal hyper-parameters found, and the corresponding train and validation errors. We normalize our samples before feeding them to the neural net input layer. In the case of the convolutional neural network we tune the number of epochs, the batch size, the kernel size, and the type of optimizer used. We tried two different architectures that are known for given good results for the MNIST classification problem.

V. Results

A high level view of our results is that the best performance among all methods we tried is given by a CNN with test (i.e. Kaggle) accuracy close to 95%. This was expected, since CNN is the method among the ones we tried that also gives the best performance for the MNIST classification problem. Some of the other methods we implemented like polynomial kernelized SVM, k-NN, or random forests, had a validation accuracy around 90%. Even a linear classifier like linear SVM had an accuracy of 86%, which seems decent for such a simple (i.e. linear) model.
We believe such good performances are highly due to our thorough preprocessing step, where we take an original instance with a noisy background and several digits on it, and transform it into an instance of the MNIST classification problem. Clearly, our preprocessing step is not perfect, and we do not get the accuracy levels that the same methods we discussed here would give for MNIST (99% in the case of CNN). However, it seems that we are not too far either.
We now present several tables that show in more detail our hyper-parameter tuning process for the different algorithms.

A. *Logistic Regression*

TABLE I. LR HYPERPARAMETER TUNING USING GRIDSEARCH

C	Best parameters	Score
[1e-6,1e-5,1e-4,1e-3,1e-2,1e-1,1,10,100,1000]	C=1000	74%

B. *K-Nearest-Neighbor*

TABLE II. K-NN HYPERPARAMETER TUNING USING GRIDSEARCH

n_neighbors	Best parameters	Score
Np.arange(5,100,10)	n_neighbors: 15	89%

C. *Random Forest*

TABLE III. RANDOM FORSEST HYPERPARAMETER TUNING USING GRIDSEARCH

Max depth range	Max features	N_estimators	Best parameters	Score
np.arange(5,100, 5)	np.arange (5, 100, 10)	np.arange(5, 100, 10)	max_depth=25, n_estimator=95 , max_features = 75	90%

D. *Kernalized SVM*

TABLE IV. RESULTS OF KERNELIZED SVM

Degree	kernel	C	Best parameters	Score
[3,5,9]	['linear','poly' ,'rbf','sigmoid']	[1e-6,1e-5,1e-4,1e-3,1e-2,1e-1,1,10,100,1000]	kernel='poly', degree=5, C = 1000	86%

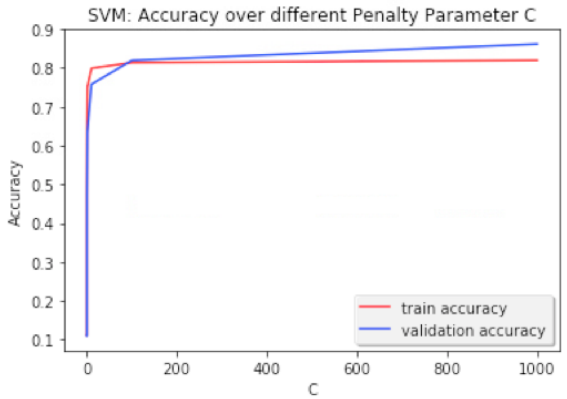

Figure 5- Linear SVM Accuracy over different C

E. Neural Networks (NN)

TABLE V. NEURAL NETWORK HYPERPARAMETER TUNING

Architecture	Batch Size	Number of hidden Layer	Number of Nodes per Hidden Layer	Learning Rate	Score
#1	1 (Stocastic)	3	[3,5,8]	0.005	10.18%
#2	100 (Mini-batch)	6	[2,4,7,3,8,5]	0.1	9.99%
#3	40000 (almost full Batch)	2	[10,5]	0.01	9.45%
#4	32	5	[4,2,8,3,5]	0.01	8.69%
#5	1	6	[2,4,7,3,8,5]	0.005	10.48%

For the simple Neural Network, we found that a smaller learning rate gives us a better score. Also, when we have two architectures with the same set of parameters, but one has more hidden layers, the score seems to increase.
Our neural network implementation from scratch did not seem to be giving good results, and we were having a lot of overflows and numerical errors while running it. We were only able to obtain an accuracy of 10% during the cross validation step. This just matches the performance of a random classifier, so we were expecting something much better than this. We tried with different architectures and activation functions, but our accuracy did not improve for any of the combinations we tried.

F. Convolutional Neural Networks (CNN)

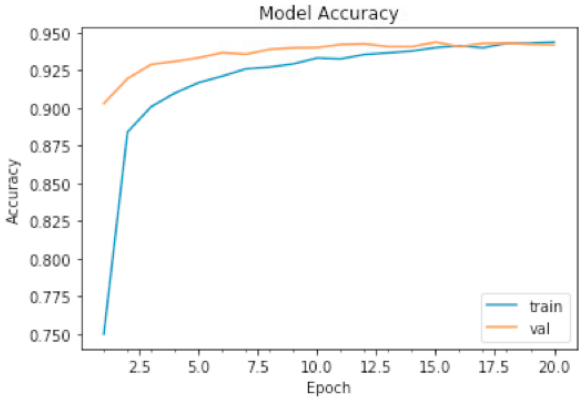

Figure 6- CNN Performance over different epochs

TABLE VI. CONVOLUTIONAL NEURAL NETWORK HYPERPARAMETER TUNING

The best hyperparameter indicated with yellow highlight.

Architecture	preprocessing	Batch Size	Optimizer	Kernel Size	Num of Epochs	Score
#1	Yes	[50,100,150,200]	Adam	(3,3)	20	0.9402
#2	Yes	150	[Adam, Adagrad,Adadelta,SGD]	(3,3)	20	0.9466
#3	Yes	150	Adam	[(3,3),(5,5)]	20	0.9538
#4	Yes	150	Adam	(3,3)	[5,10,15,20]	0.94
#6	No	150	Adam	(3,3)	20	0.50

VI. DISCUSSIONS

The main advantage of our approach is that it reduces the original problem to a very well known and studied problem like the MNIST handwritten classification problem. We then exploit some of the methods that are already known to give good performance for such problem. This allows us to get an accuracy close to 95%, which is not too far from the best accuracy obtained for MNIST (99%).
However, our preprocessing step is not perfect, and there are several components in it that could be further improved. One of such components which seemed very important to us is the deskewing/rotation step for the digits. Although our implementation made things a lot better, we still had several digits that were slightly rotated or deskewed, and this could have caused additional trouble for the classifiers.

VII. STATEMENT OF CONTRIBUTIONS

All students contributed equally to this project. Some had their attention more focused on some specific parts of the project. Rony focused most of his attention on the simple Neural Network and cross validation for it. Richard was in charge of CNN and Mandana did image preprocessing. For all of the areas, each team member did everything from developing the methodology, hyper-parameter tuning, testing to coding the solution.

We hereby state that all the work presented in this report is that of the authors.

VIII. REFERENCES

For the CNN architecture and implementation we use several ideas from the following CNN with Keras:

1. tutorial: http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/
2. https://en.wikipedia.org/wiki/Convolutional_neural_network