CSE 3521: Artificial Intelligence

Final Project Report: Digit Recognition

Isaac Mattern, Jamie Walters, Noah Lapolt, Sam Gernstetter

30 April 2021

Jeniya Tabassum

Problem definition and data: https://www.kaggle.com/c/digit-recognizer/data

---

Introduction

Our project topic was digit recognition. Our goal was to write two algorithms that would accurately predict the digits (0-9) based on hand-written images uploaded to the algorithms. Our dataset comes from kaggle.com, and is the famous MNIST database. This database includes tens of thousands of images of hand-written digits.

Fig. 1 Digit images

Each image is 28x28, for a total of 784 pixels per image. Each pixel is converted a value that ranges from 0-255. These values encode the darkness or lightness of each pixel, with 0 being white and 255 being the darkest black possible.

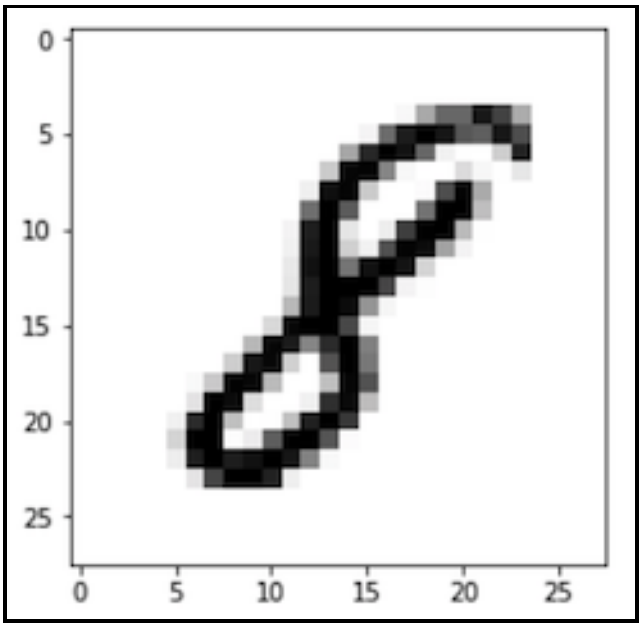

Fig. 2 One digit image, gridded by pixels

The image dataset lives in a csv file, and includes 42,000 training images and 28,000 test images. This was extremely convenient for us, as we did not have to take any extra steps to convert the image to data points. The csv file is neatly organized as follows: each row begins with one column which is the correct identification for that digit. The following 784 columns contain the integer value from indicating the darkness of a pixel. The only difference between the train and test csv files is that the test file does not have a column indicating the correct digit, since this Kaggle problem was a competition.

Algorithm 1: Naive Bayes

To create this algorithm, we started with homework 2 and adapted it to fit the needs of our project. We began by importing our train and test data into a google colab file. Next, we counted the occurrences of each number in the data file, as well as the total number of pixels in the data file. We then created a dictionary of dictionaries to hold the pixels for each digit. Next,

we calculated the probabilities of each digit and made predictions. We then ran the test data through the algorithm and calculated the accuracy. We ended up with an accuracy of 72.675%.

At first, we ran into some issues with the predictions. At the end of our algorithm when we multiplied the probabilities, they all rounded to 0 because the probabilities were so small. This caused (by default) all predictions to go to 1, which is obviously not what they all should be. To fix this problem, we used sums of the logarithms of the probabilities to stop all the probabilities from rounding to 0. Next, instead of checking for dark spaces to identify the digit, we checked for whitespace around the digits. This increased our accuracy to 72.675%. We are happy with this accuracy, since the accuracy may be low with certain datasets since Naive Bayes assumes feature independence.

Algorithm 2: Perceptron / Neural Network

Before attempting a multilayered neural network, we decided to first implement a perceptron algorithm with no middle layer. The input layer was 784 nodes - representative of pixels 0 to 783, and the output layer was 10 nodes - representative of digits 0 to 9 which we wished to correctly predict. Therefore, even without a middle layer, there were 7,840 feature vectors whose weights could be adjusted and fine tuned.

To begin, after loading the data, we normalized the pixel intensities by dividing by 255, so that each intensity was now a decimal value from 0 to 1. We defined two functions: nn and test. The nn function is short for neural network (although, at this point, it is just the perceptron algorithm). Here, we are creating our model and training it. In addition to the test data, the function takes in a parameter called learningRate. We have set the learning rate to 0.01, as this seems to be a common rate. The test function takes in some data to be tested, as well as the

perceptron model. It returns a pandas DataFrame with the predicted digits. Using this function, we were able to test some of the provided training data, as well as the actual test data. With the perceptron algorithm, our training accuracy reaches as high as 80%, while our test data submitted to Kaggle reaches about 66.7%. We were impressed that a single input and output layer was able to achieve results this high, but we wanted to go even further and get higher accuracy. Thus, we implemented a neural network with multiple layers.

For the multiple layer approach, despite testing up to four hidden layers with varying numbers of nodes per each layer, we ended up going with only one hidden layer, as it offered a very high rate of accuracy while not taking eons to run. For the single layer we ended up going with a design that consisted of a 784 node input layer, 98 node hidden layer, and 10 node output layer. This decision was based on the high accuracy that a 98 node hidden layer offered while still being fast enough to do multiple iterations over the entire dataset. We also decided to split each iteration into smaller steps. This may not have been the most optimal decision, but it helped decrease the amount of stress on the RAM and increase the speed the computations were computed. The variable layer represented this division across each iteration of data. The final adjustment we made was the number of iterations that the data ran in order to get the best results without overtraining. After much testing it seemed that four iterations returned the best results with layers that were each eight images. With all of these factors applied to our multiple layer neural network we got an accuracy of 88-91% on the training data and 87-90% on the test data with a runtime of about five minutes to complete the training. While an increase in accuracy was expected from adding hidden layers to the neural network, we did not expect the addition of a single layer to raise it as much as it did. The variance in the results was caused by the random starting weights that the network utilized.

Notes on Accuracy

Unfortunately, because our problem definition and dataset were taken from a Kaggle competition, correct answers are hidden. This means that we were not able to observe the frequencies at which we incorrectly predicted test data.

Work Distribution

Jamie and Sam worked on the Naive Bayes algorithm, while Isaac and Noah worked on the Neural Network algorithm. Isaac organized the original import of data, which Sam modified for the Naive Bayes solution. Noah figured out how to format the submission csv so that we could check our accuracy in Kaggle. Everybody contributed to the final presentation and the final report.

Conclusion

We are happy with the accuracies we were able to achieve. We believe that our implementations are correct, and that our accuracies are simply due to the nature of the problem we were trying to solve and the algorithms we used. We believe that if we used other datasets, we may see other better accuracies. Regardless, we believe our Naive Bayes and Neural Network algorithms are sufficient and yield satisfactory results for our first semester of studying artificial intelligence and machine learning.