Patrick Steeves
Jingning Li
Yunning Zhu

May 5th, 2017

Digit Recognizer

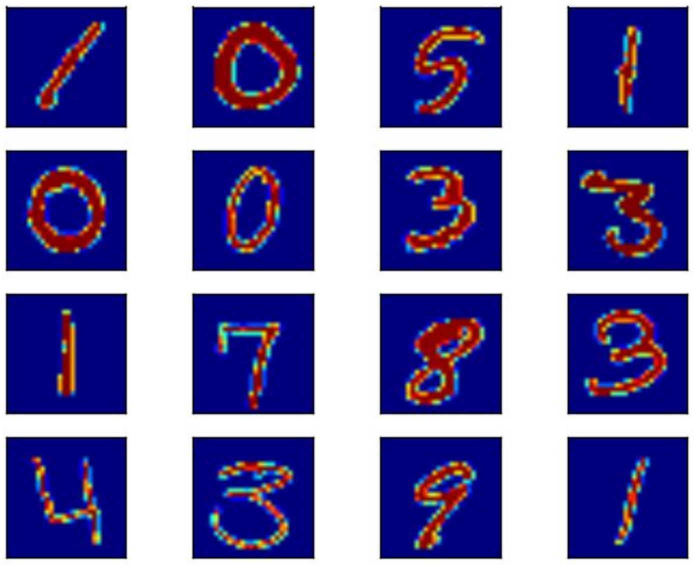
Introduction

Our final project consisted of competing in a live Kaggle contest, Digit Recognizer. The goal of the competition is to correctly classify handwritten digits from the famous MNIST dataset, known as the “hello world” of computer vision. The dataset was released in 1998, and has been used as a benchmark for image classification ever since. In the original paper, the authors tried different models on the dataset and obtained an error rate of 0.7%[1] on the test set using a Boosted LeNet-4 CNN. The best result that we were able to find is 0.21%[2] using a DropConnect CNN.

Data

The MNIST dataset contains 70,000 records of 28x28 grayscale images of numbers. Each record therefore contains 784 dimensions that can take values between 0 and 255. The images have all been centered and normalized in size. Kaggle splits the data into training and test sets of 42,000 and 28,000, respectively, where the training set has image labels and the test set does not.

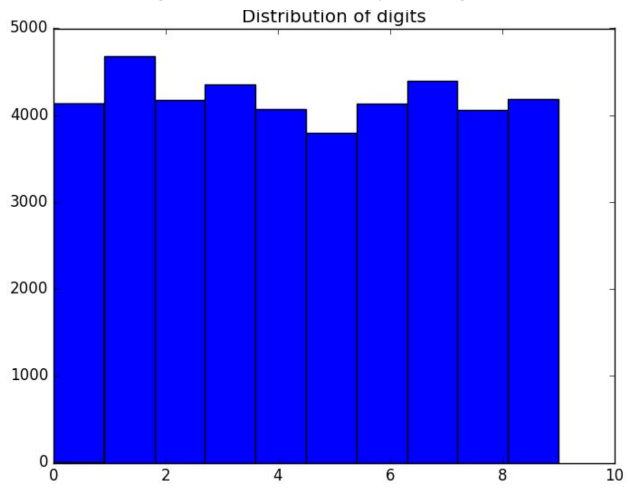
Figure 1: MNIST digits



[1] http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf
[2] http://cs.nyu.edu/~wanli/dropc/

We examined the frequency of the digits (0 through 9) to ensure we did not have to sample the data for our model training. As seen below, the digits are evenly distributed.

Figure 2: Frequency of digits



To preprocess our data, we first normalized all pixel values to be contained between 0 and 1. We then attempted principal components analysis, but found that it did not yield better results. We also tried to augment our training data by shifting, zooming, or distorting images, but this also did not help our training. After the fact, we realized that this was because the images were all centered and size-normalized, therefore augmenting our data is not helpful like it might be if the MNIST digits had not been pre-processed already. Also, we recognized that this would not have helped with the convolutional neural network, which was our best-performing model.

Modeling

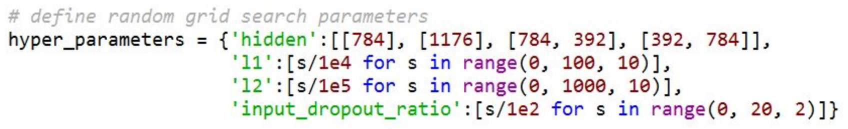
First Attempt: Support Vector Machine

In our first attempt, we used a support vector machine (SVM). SVM isw a popular set of supervised learning methods used for classification, regression and outliers detection. We split the training data from Kaggle into training and validation sets, 50% for each. To train the model, we used the Python package sklearn, which enables SVM model-building. After trying different parameter tunings, the best validation result we were able to get was a 3% misclassification for our validation set. We did not bother submitting our test set to Kaggle because this error rate was so high.

Second Attempt: Feed-forward Neural Net

Next, we decided to use neural networks for the classification task. We first used h2o in Python to build a feed-forward neural net. The most difficult and time consuming part of the model design for this project was tuning model parameters, due to the long time needed to train each model. The h2o package was advantageous because it offers hyperparameter tuning. We let h2o randomly pick values for four parameters out of a given range, as shown in figure 3, and then rank the models based on classification accuracy. The model that scored the best used one hidden layer with 1176 nodes, an input dropout ratio of 0.14, no L1 penalty, and a L2 penalty of 0.002.

Figure 3: hyperparameter tuning



```
# define random grid search parameters
hyper_parameters = {'hidden':[[784], [1176], [784, 392], [392, 784]],
                    'l1':[s/1e4 for s in range(0, 100, 10)],
                    'l2':[s/1e5 for s in range(0, 1000, 10)],
                    'input_dropout_ratio':[s/1e2 for s in range(0, 20, 2)]}
```

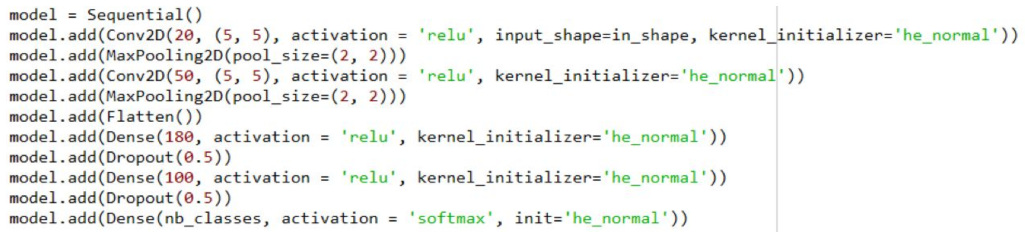
This model did slightly worse than our SVM, reporting a misclassification rate of 3.5%.

Third Attempt: Convolutional Neural Network

In order to improve our accuracy, we did some research and learned that convolutional neural networks are a popular model for image classification. These networks do a better job of processing images because they do not ignore the inherent spatial structure of images by having pixels close together connected to the same hidden nodes, but not with every single pixel in the image, like a simple feed-forward net would. We used an online resource to understand how CNNs work on a basic level.[3] Unfortunately, h2o does not provide a solution to build CNNs, therefore we turned to the keras package in Python for our CNN training. After manually tuning parameters, the best performing model had a first convolution layer of depth 20 and size 5x5, followed by a max pooling layer of size 2x2, followed by another convolution layer of depth 50 and size 5x5, and another identical pooling layer. These were followed by two dense layers of size 180 and 100, where the dropout rate was 0.5 for both. We used an early stopping function provided by keras to limit overfitting. For each layer in the model, we used the ReLU activation function, and for the output layer we used softmax activation. We initialized node weights using a truncated normal distribution. Our code can be seen in figure 4 below. The model reported a misclassification rate of 0.81% on the test set provided by Kaggle. A useful visualization tool for CNNs can be found on this website.[4] The structure of the CNN is not identical to ours, but it provides more insight on how pixel inputs are converted into classification outputs.

[3] https://medium.com/@ageitgey/machine-learning-is-fun-part-3-deep-learning-and-convolutional-neural-networks-f40359318721
[4] http://scs.ryerson.ca/~aharley/vis/conv/flat.html

Figure 4: Code for keras CNN

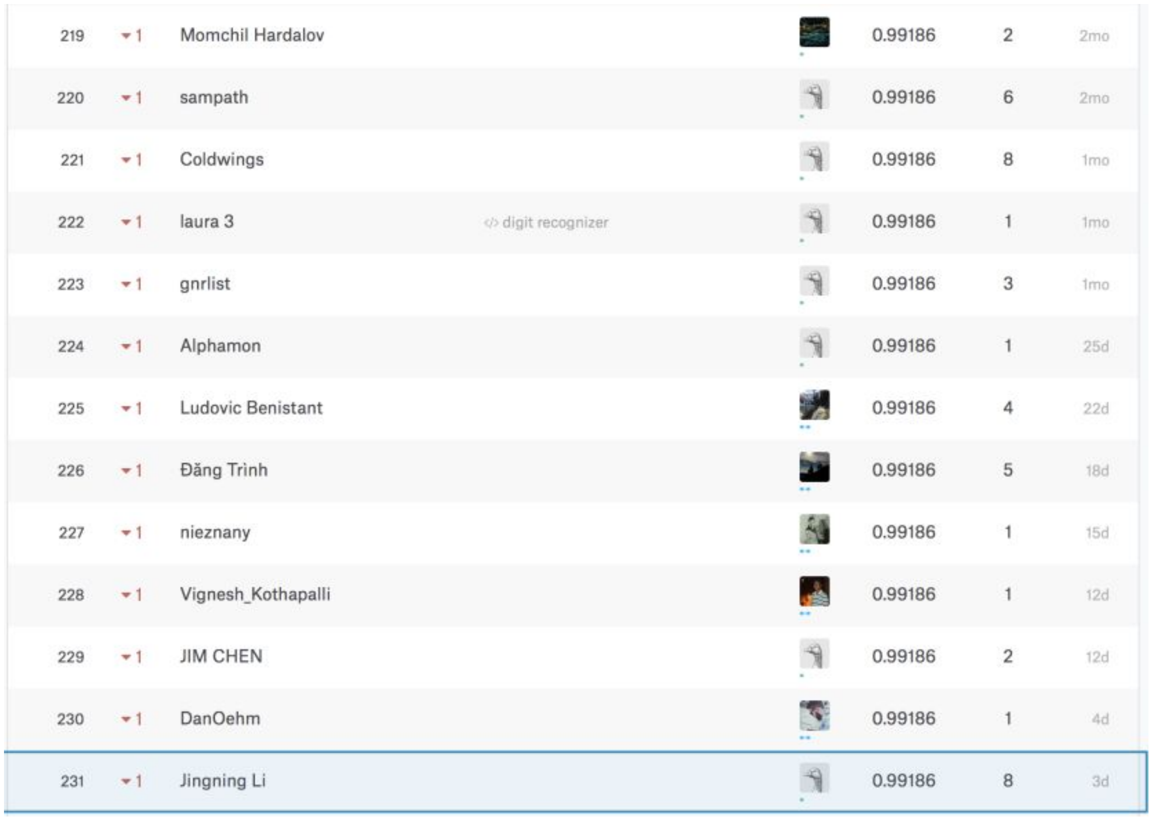


```
model = Sequential()
model.add(Conv2D(20, (5, 5), activation = 'relu', input_shape=in_shape, kernel_initializer='he_normal'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(50, (5, 5), activation = 'relu', kernel_initializer='he_normal'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(180, activation = 'relu', kernel_initializer='he_normal'))
model.add(Dropout(0.5))
model.add(Dense(100, activation = 'relu', kernel_initializer='he_normal'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes, activation = 'softmax', init='he_normal'))
```

Discussion

Out of a feed-forward neural net, a support vector machine, and a convolutional neural network, we found that the best-performing model by far was the convolutional neural network, which we built using a package named keras. Our accuracy for the Kaggle test dataset is around 99.19%, and we tied for 219 out of 1592 teams in the competition, as shown below. One thing to keep in mind is that many teams achieved a score of 1.000, which is clearly cheating and can be easily done since this dataset is publicly available online.

Figure 5: Ranking in Kaggle Leaderboard

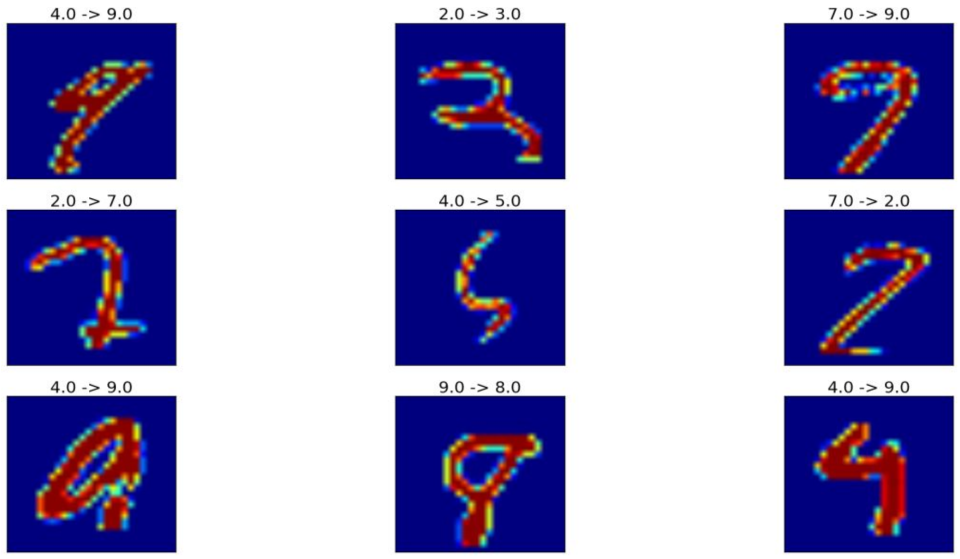


219	▾1	Momchil Hardalov		0.99186	2	2mo
220	▾1	sampath		0.99186	6	2mo
221	▾1	Coldwings		0.99186	8	1mo
222	▾1	laura 3	</> digit recognizer	0.99186	1	1mo
223	▾1	gnrlist		0.99186	3	1mo
224	▾1	Alphamon		0.99186	1	25d
225	▾1	Ludovic Benistant		0.99186	4	22d
226	▾1	Đăng Trịnh		0.99186	5	18d
227	▾1	nieznany		0.99186	1	15d
228	▾1	Vignesh_Kothapalli		0.99186	1	12d
229	▾1	JIM CHEN		0.99186	2	12d
230	▾1	DanOehm		0.99186	1	4d
231	▾1	Jingning Li		0.99186	8	3d

This result is also very similar to the original MNIST paper's best performing model, which we were very pleased with. We suspect that it would be very hard to improve upon this without significantly improving our computing power, or using a powerful GPU.

Of the 0.81% of digits that we misclassified, we found that many of them would also be difficult for a human to correctly classify. Figure 6 shows some misclassification examples of our CNN. The first number in the title of each subplot is our classification, and the second number is the correct label. For example, the digit in the second row and first column is a 7, but it looks very similar to a 2. The 9 in the bottom right could be easily recognized as a 4 instead of a 9. These examples show that our model would have done a lot better with more reasonably-drawn digits that can at least be recognized by a human. However, we still misclassified some digits, such as the 8 in the third row and second column of figure 6, that humans could easily read.

Figure 6: Misclassification Examples



One capability that we think could have greatly helped our model results is the use of hyperparameter tuning for the CNN building. We were only able to do this through h2o when building the feed-forward net, but as previously mentioned, h2o does not provide a way to train CNNs.

Code

Our code can be found in the Python scripts submitted along with this report.