Intelligent Handwritten Digit Recognition Using CNN Model

Shuwei Deng
Mengqi Li
Yuxuan Liu
Jiwei Zeng

April 19th, 2018

Abstract

CNN model is convolutional neural networks which are made up of neurons that have learnable weights and biases. It works effectively in areas like images recognition and classification. In this paper, we will introduce you how we build the CNN model to do digit recognition and evaluate it with several techniques.

1. Introduction

Our everyday life involves big data almost everywhere. What we used to analyze was just based on sorted numbers. We are curious about what can we analyze using images. Having these images, can we train a model that is able to automatically recognize the information in the images? Would the model outperform manpower in terms of accuracy and efficiency? With all these kinds of questions, we decided to start our examination about the digital analyzation.

2. Data Collection

The kaggle competition is developed based on MINST Handwritten Digit Recognition for evaluating machine learning models on the handwritten digit classification problem. The dataset has two csv files, “train.csv” and “test.csv”, that was constructed from a number of scanned document dataset available from the National Institute of Standards and Technology. The training data of the dataset contains 42,000 rows of handwritten number inputs that represent images and 785 columns including the “label” column with no null values in figure 2. The test data set, (test.csv), is the same as the training set, (train.csv) except that it does not contain the "label" column. Each image includes 784 pixels in total (28 in height and 28 in width), and below figure 1 is one of the images. Each pixel associated with a pixel-value from 0 to 255 indicates the darkness of the pixel, the higher the value, the darker the pixel.

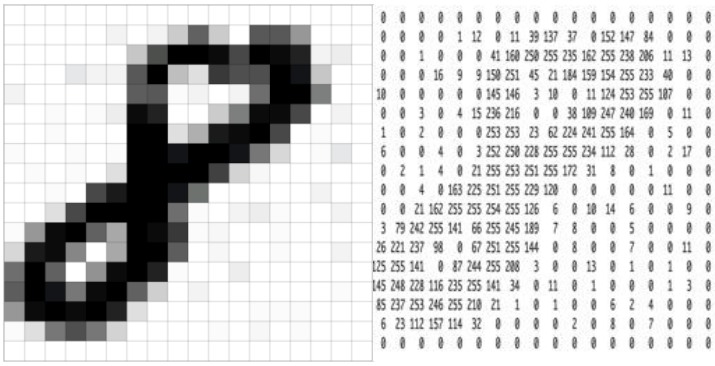

Figure 1 - If you think about it, everything is just numbers.

```
X_train.isnull().any().describe()
```

```
count       784
unique        1
top       False
freq        784
dtype: object
```

```
y_train.isnull().any().describe()
```

```
count         1
unique        1
top       False
freq          1
dtype: object
```

Figure 2 - Check null values in training set

3. Preprocessing Data

Deep learning uses neural nets with a lot of hidden layers that require particular large amount of training data. The most common image data input parameters are the images, the number of channels and the number of levels per input of the data. Building an effective neural network model consists various input processes as well as customized architecture and algorithm of the network in distinct scenarios. We structure the model by performing the following 4 steps:

3.1 Normalize

The pixel-values ranging from 0 to 255. The values have too many variables for the learning algorithm to process, so we make the input parameter have smaller data distribution into a scale of 0 to 1. Moreover, because the inputs are images, the pixel values cannot be negative. This process makes convergence faster while training the

network.

```
# The data right now is in an int64 format,
# so before you feed it into the network you need to convert its type to float32,
# and you also have to rescale the pixel values in range 0 - 1 inclusive.
X_train = X_train.astype('float32')
test = test.astype('float32')
X_train /= 255 # Normalise data to [0, 1] range
test /= 255 # Normalise data to [0, 1] range
```

Figure 3 - Normalizing the data

3.2 Reshape

At first, pandas stock our train and test images in 1D (1*784), we reshaped data to 3D (28*28*1) matrices, since keras needs an extra dimension in the end. If the data is RGB images, there would be 3 channels and it should be reshaped into 3 channels (28*28*3). In this case, the dataset is gray scale image so it only uses one.

```
X_train = X_train.reshape(-1, 28,28, 1)
X_train.shape
```

```
(42000, 28, 28, 1)
```

```
test = test.as_matrix()
test = test.reshape(-1, 28,28, 1)
test.shape
```

```
(28000, 28, 28, 1)
```

Figure 4 - Reshaping the data

3.3 Label One-hot Encoding

One hot encoding is a process that converts categorical variables that most machine learning algorithms do not understand into a binary form. Labels are 10 digits numbers from 0 to 9. Since the algorithms cannot process categorical variables directly, we did one-hot encoding to “binarize” the labels.

```
Y_train = to_categorical(y_train, nClasses) # One-hot encode the labels
# Display the change for category label using one-hot encoding
print('Original label:', y_train[0])
print('After conversion to one-hot:', Y_train[0])
```

```
Original label: 1
After conversion to one-hot: [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Figure 5 - One-hot Encoding before and after

3.4 Splitting Train and Validation Set

We split train and validation set with proportions of 90% and 10%. Since we have a considerable amount of data, 10% for validation, which is 4200, would be enough for validating most of the variance in the data. In addition, each person has a unique writing format. In order to increase the probability of recognizing a new hand-writing the model has never seen before correctly, it is necessary to train the model with substantial data. We did not split test set due to the existence of test.csv in the dataset Kaggle provided.

```
train_X,valid_X,train_label,valid_label = train_test_split(X_train, Y_train, test_size=0.1, random_state=13)
```

```
train_X.shape,valid_X.shape,train_label.shape,valid_label.shape
```

```
((37800, 28, 28, 1), (4200, 28, 28, 1), (37800, 10), (4200, 10))
```

Figure 6 - Splitting train and validation set

4. Build CNN Model

We applied deep convolutional neural networks to image recognition. Convolutional neural networks, usually referred to as CNN or ConvNet, is a kind of deep, feed-forward artificial neural network, which is called multi-layer perceptions(MLPs). The models are called "feed-forward" because information flows right through the model. There are no feed-back connections in which outputs of the model are fed back into itself.

One simple reason why we chose CNN as our final model is that it achieved a higher 99%+ accuracy compared to others simple fully-connected network with around 95% accuracy that we tried. Technically speaking, CNN gets better results since it treats each data point as a 28*28 square instead of treating all 784 values the same.

The following is our CNN model architecture:

In → Conv2D→ LeakyRelu→ [Conv2D→ LeakyRelu → MaxPool2D → Dropout]*3 →

Flatten → Dense → Dropout → Out

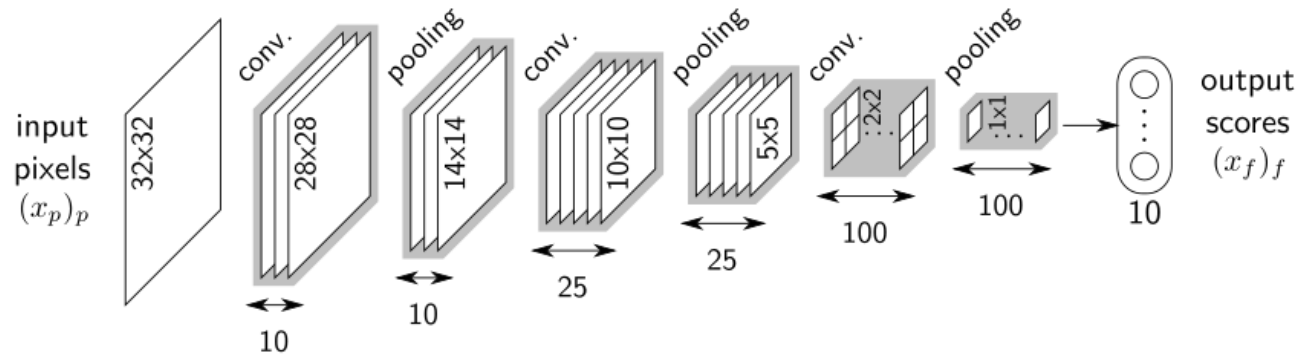

Figure 7 - “How CNN processes” illustration

We used sequential API from Keras, which you add on one layer at a time starting from the input to define the network. We add consecutively Conv+Leaky Relu layers followed by maxpool layer to increase exponentially the number of filters. Each filter transforms a part of the image.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5),activation='linear',input_shape=(28,28,1),padding='same'))
model.add(LeakyReLU(alpha=0.1))
model.add(Conv2D(32, kernel_size=(3, 3),activation='linear',padding='same'))
model.add(LeakyReLU(alpha=0.1))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='linear',padding='same'))
model.add(LeakyReLU(alpha=0.1))
model.add(MaxPooling2D(pool_size=(2, 2),strides=(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(96, (3, 3), activation='linear',padding='same'))
model.add(LeakyReLU(alpha=0.1))
model.add(MaxPooling2D(pool_size=(2, 2),strides=(2,2)))
model.add(Dropout(0.4))

model.add(Flatten())
model.add(Dense(128, activation='linear'))
model.add(LeakyReLU(alpha=0.1))
model.add(Dropout(0.3))
model.add(Dense(nClasses, activation='softmax'))
```

Figure 8 - CNN model we built

The first layer is the convolutional (Conv2D) layer, which acts like a set of learnable filters. We set double 32 filters as the first two Conv2D layers, 64 as the third one, and 96 for the fourth one. Each one of the filters screen a small part of the input image and the kernel filter matrix is used to transform whole image and extract a feature map.

The second layer is the leaky relu layer, which will change the activation into leaky relu function. Compared with Rectified Linear Units (ReLUs), it attempts to fix the problem of “dying unit”. The ReLU activation function is used a lot in neural network architectures and more specifically in convolutional networks, where it has proven to be more effective than the widely used logistic sigmoid function. As of 2017, this activation function is the most popular one for deep neural networks. The ReLU function allows the coefficients to be thresholded at zero. However, during the training, ReLU units may "die". This can happen when large gradient flows through a ReLU neuron: it can cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. Leaky ReLUs attempt to solve this: the function will not be zero but will instead have a small negative slope.

The third layer is another convolutional layer, and the fourth is another leaky relu layer. Next, the fifth layer is the pooling (MaxPool2D) layer, which acts like a downsampling filter to reduce the computational cost and reduce the probability of overfitting. The higher the pooling dimension, the more downsampling is important. We have a max-pooling layer of size 2 x 2 in our model, which will select the maximum pixel intensity value from 2 x 2 region.

The sixth layer is the dropout layer, at which an amount of nodes in the layer will be randomly ignored. This layer improves generalization and reduces the overfitting. This drops randomly a proportion of the network and forces the network to learn features in a distributed way.

We combined the convolution, leaky, pooling and dropout layers which are the third layer through sixth layer, and repeated 3 times with different convolutional filter number to minimize the probability of overfitting, learn more local features, and learn more global features of the image.

Next we have the flatten layer, which convert the final results into a 1D vector. This layer helps us connect all the local features of the previous layers and combine them. Finally, we have a dense layer, which is an artificial neural networks classifier. The Dense layer that has a softmax activation function with 10 units, which is needed for this multi-class classification problem.

5. Compile and train the model

After setting layers of the model, we would compile the model using the Adam optimizer, one of the most popular optimization algorithms. In addition, we specify the loss type as categorical cross entropy used for multi-class classification and specify the metrics as accuracy which enable us to analyze while training model.

```
model.compile(loss=keras.losses.categorical_crossentropy,  # using the cross-entropy loss function
              optimizer=keras.optimizers.Adam(), # using the Adam optimiser
              metrics=['accuracy']) # reporting the accuracy
```

Figure 9 - Compile model

Because the model will be trained in 35 epochs, there may be overfitting in the final model. To avoid the overfitting, the check pointer will be applied. It can save the best model in training based on the accuracy of validate data into external document which can be loaded next time and used to predict the test data.

```
# monitor is defined as val_acc, output coordinating information,
# and show the best val_acc, which is the highest val_acc
check_pointer = ModelCheckpoint(filepath="weights.best.hdf5", monitor='val_acc', verbose=1, save_best_only=True)
```

Figure 10 - Check pointer

Before training the model, we set the random seed in order to obtain a reproducible result.

```
np.random.seed(12345)

model_train = model.fit(train_X, train_label,
                        batch_size=batch_size,
                        epochs=num_epochs,
                        verbose=1,
                        validation_data=(valid_X, valid_label), callbacks=[check_pointer])
```

Figure 11 - Fit model

6. Model Evaluation

We used validation set to evaluate our model since test.csv did not provide the label and it can not test accuracy of the model. Later, we would conclude the ultimate accuracy score rated by Kaggle competition. As figure 12 showed, our model finally achieved a 99.38% accuracy with validation and only 0.0209 final loss. Although it is not our ultimate accuracy with test, it still indicates our model performs well.

```
final_loss, final_acc = model.evaluate(valid_X, valid_label, verbose=0)
print("Final loss: {0:.4f}, final accuracy: {1:.4f}".format(final_loss, final_acc))
```

```
Final loss: 0.0209, final accuracy: 0.9938
```

Figure 12 - Final loss and accuracy from validation

6.1 Confusion matrix

According to the confusion matrix, our model predicted the numbers mostly correct with a small number of misclassification. It misclassified number “1” as “7” and “2” as “7” as shown in figure 14. It is reasonable because when the handwriting is smooth, human eyes may mis-recognize those numbers and don't even mention the computers.

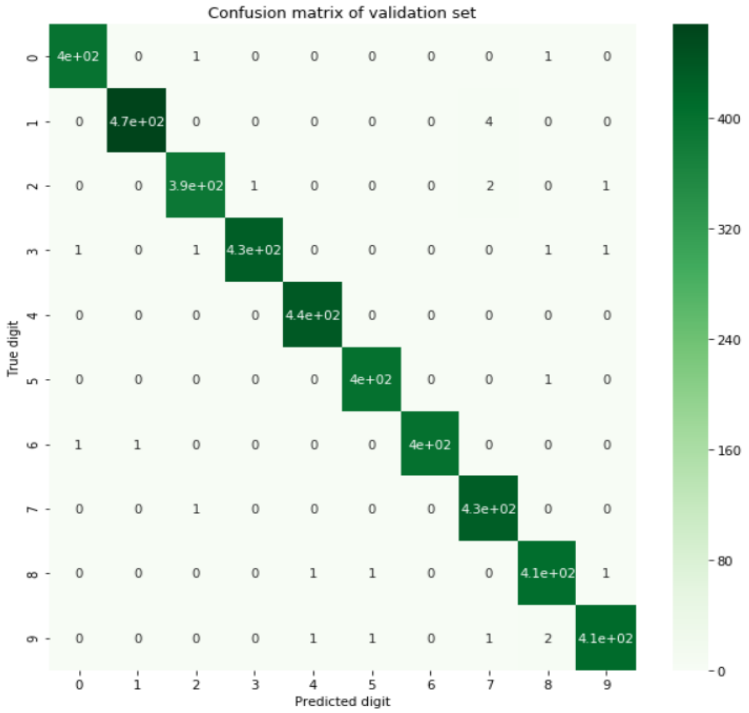

Figure 13 - Confusion Matrix

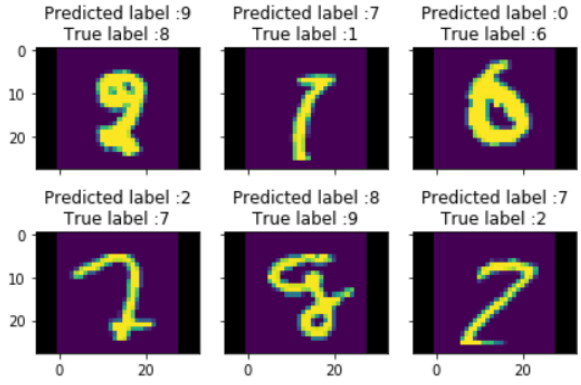

Figure 14 - Misclassification in Validation

6.2 Validation Loss and Accuracy Plot

As shown in figure 15, we can easily find that the best epochs is about 17 that achieved the lowest loss and highest accuracy for both the training and validation set. In addition, we observed that for certain range of epochs, training set outperformed validation set and validation set also outperformed training set for some epochs. It indicates our model is neither overfitting nor underfitting.

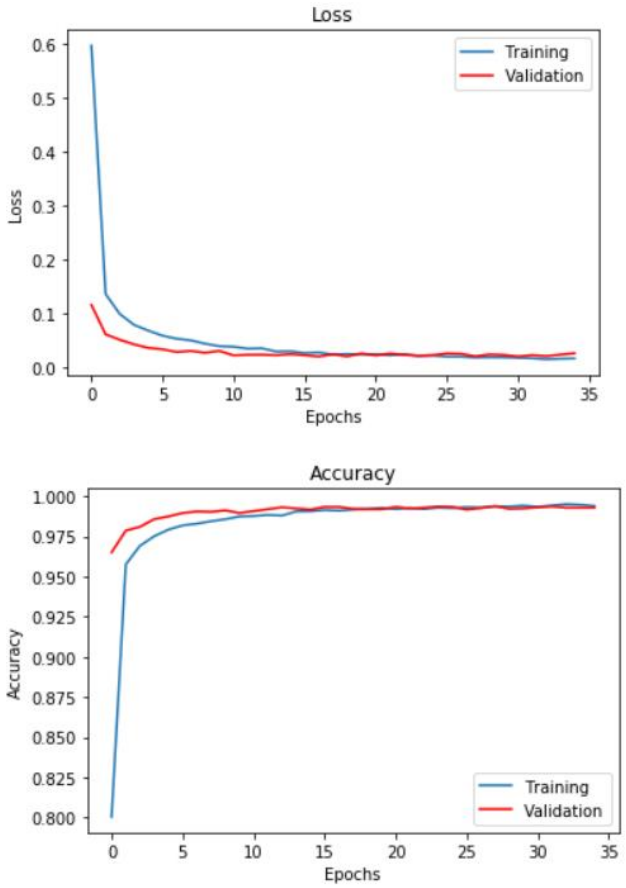

Figure 15 - Validation Loss and Accuracy Plot

7. Application

Our model is useful in many ways. For example, nowadays, according to a new study from Pew Research, 96% of Americans shop online, and the majority of those shoppers prefer shopping online for reasons like saving time and avoiding crowds (Kelly). The increasing daily capacity of categorizing mails in major sorting centers is asking for more accurate and more efficient mail sorting system other than labor. With further development of our model, upon the acceptance of mails, in the future, we will be able to recognize the handwritten labels by scanning instead of manually typing those into the system.

Furthermore, given more specific dataset to train our model, the model will be able to demonstrate more features. For example, photo capture function can be utilized to

avoid typo while using the mobile phones. Moreover, the recognition model will have a lot of more potential in process like detecting the correct suspects with a higher probability. When deposit checks, banks could employ the model to recognize the signature to detect whether the transaction is fraud or not. Additionally, a growing number of people were troubled by poor eyesight while typing on the touch keypads on the smart phones, with the help of the recognizer model, it would be more convenient for them to communicate.

8. Conclusion

According to the findings above, we found out that using CNN model (In → Conv2D→ LeakyRelu→ [Conv2D→ LeakyRelu → MaxPool2D → Dropout]*3 → Flatten → Dense → Dropout → Out) to recognize the patterns, we will get 99.457% of correct classification rate. Under this model, it minimizes the overfitting and underfitting. The model is less time consuming and more accuracy than human assistance. We believe in the future, with further improvements, the model will be implemented in many aspects, such as handwritten labels, zip codes, and etc.

Work Cited

1. Kelly, Leanna. “How Many People Shop Online?”. *CPC Strategy.* May 25, 2017. Web.