Recognizing Digits using Neural Networks

Ayoub Bouhachmoud April 2020

Contents

1 Introduction **2**

2 MNIST Dataset **2**

2.1 Data Description 3

2.2 Data Preparation 3

3 Convolutional Neural Network (CNN) **5**

3.1 Network Architecture 5

3.2 Pooling Layers 7

3.3 Model Summary 7

4 Performance Evaluation **8**

4.1 Dopout value 8

4.2 Testing different optimizers 8

4.3 Precision, Recall, F1 9

4.4 ROC Curve 9

4.5 Confusion Matrix 11

5 Effect of Hyperparameters **12**

5.1 Number of Epochs 12

5.2 Batch size 13

5.3 Learning Rate 14

6 Other Perspectives **14**

6.1 Saving the model 14

6.2 Using our own handwritten digits 14

7 Conclusion **16**

1 Introduction

In recent times, with the increase of Artificial Neural Network, deep learning has brought a dramatic twist in the field of machine learning by making it more artificially intelligent. Deep learning is remarkably used in vast ranges of fields because of its diverse range of applications such as surveillance, health, medicine, sports, robotics, drones, etc. In deep learning, Convolutional Neural Network (CNN) is at the center of spectacular advances that mixes Artificial Neural Network and up to date deep learning strategies. It has been used broadly in pattern recognition, sentence classification, speech recognition, face recognition, text categorization, document analysis, scene, and handwritten digit recognition. The goal of this project is to observe the variation of accuracies of CNN to classify handwritten digits using various numbers of hidden layers and epochs and to make the comparison between performances. For this performance evaluation of CNN, we performed our experiment using Modified National Institute of Standards and Technology (MNIST) dataset.

In handwritten recognition digits, characters are given as input. The model can be recognized by the system. A simple artificial neural network (ANN) has an input layer, an output layer and some hidden layers between the input and output layer. CNN has a very similar architecture as ANN. There are several neurons in each layer in ANN. The weighted sum of all the neurons of a layer becomes the input of a neuron of the next layer adding a biased value. In our CNN all the neurons are not fully connected. Instead, every neuron in the layer is connected to the local receptive field. A cost function generates in order to train the network. It compares the output of the network with the desired output. The signal propagates back to the system, again and again, to update the shared weights and biases in all the receptive fields to minimize the value of cost function which increases the network's performance. The goal of our work is to observe the influence of many parameters of a CNN for handwritten digits. We have applied our Convolutional Neural Network algorithm on Modified National Institute of Standards and Technology (MNIST) dataset using Keras, a Neural Network library written in python. The main purpose of our project is to analyze the variation of outcome results for using a different combination of parameters of Convolutional Neural Network.

2 MNIST Dataset

These package imports are pretty standard — we'll get back to the Keras-specific imports further down.

```
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4
5 # let's keep our keras backend tensorflow quiet
```

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
# for testing on CPU
#os.environ['CUDA_VISIBLE_DEVICES'] = ''

# keras imports for the dataset and building our neural network
from keras.datasets import mnist
from keras.models import Sequential, load_model
from keras.layers.core import Dense, Dropout, Activation
from keras.utils import np_utils
```

2.1 Data Description

Modified National Institute of Standards and Technology (MNIST) is a large set of computer vision dataset which is extensively used for training and testing different systems.

The database contains 60,000 images used for training as well as few of them can be used for cross-validation purposes and 10,000 images used for testing. All the digits are grayscale and positioned in a fixed size where the intensity lies at the center of the image with 28×28 pixels. Since all the images are 28×28 pixels, it forms an array which can be flattened into 28*28=784 dimensional vector. Each component of the vector is a binary value which describes the intensity of the pixel.

We will attempt to identify them using a CNN. The mnist dataset is conveniently provided to us as part of the Keras library, so we can easily load the dataset. Out of the 70,000 images provided in the dataset, 60,000 are given for training and 10,000 are given for testing.

2.2 Data Preparation

Before working with this data, we have processed the images (resizing, normalizing the pixels etc.)

Now we'll load the dataset using this handy function which splits the MNIST data into train and test sets.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Let's inspect a few examples.

```
for i in range(9):
  plt.subplot(3,3,i+1)
```

```
  plt.tight_layout()
  plt.imshow(X_train[i], cmap='gray', interpolation='none')
  plt.title("Digit: {}".format(y_train[i]))
  plt.xticks([])
  plt.yticks([])
plt.show()
```

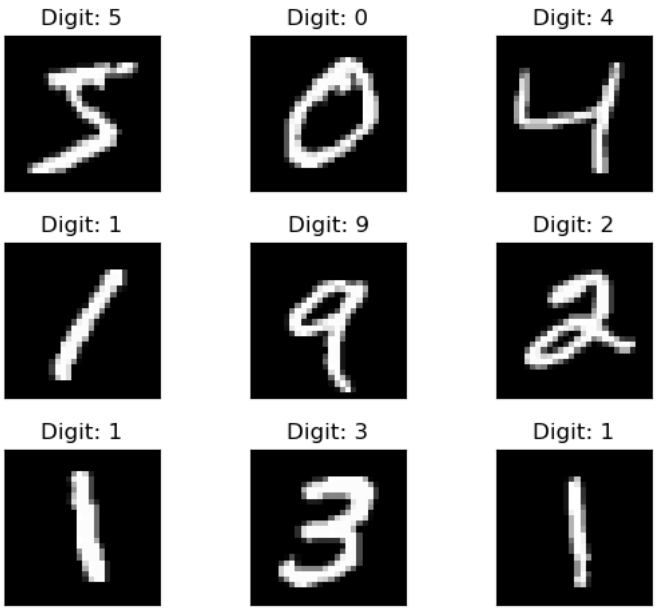

Figure 1: Visualization of MNIST dataset

The MNIST dataset contains only grayscale images. the pixel values range from 0 to 255: the background majority close to 0, and those close to 255 representing the digit.

Normalizing the input data helps to speed up the training. Also, it reduces the chance of getting stuck in local optima, when using stochastic gradient descent (we will be using a variant called SGD) to find the optimal weights for the network.

Let's reshape our inputs to a single vector vector and normalize the pixel values to lie between 0 and 1.

```
# input image dimensions
image_size = x_train.shape[1]
# resize and normalize
x_train = np.reshape(x_train,[-1, image_size, image_size, 1])
x_test = np.reshape(x_test,[-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
```

We will also encode our categories - digits from 0 to 9 - using one-hot encoding. The result is a vector with a length equal to the number of categories. The vector is all zeroes except in the position for the respective category. Thus a '5' will be represented by [0,0,0,0,1,0,0,0,0].

```
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

3 Convolutional Neural Network (CNN)

In this section, we're going solve the MNIST digit classification problem using CNNs.

```
from keras.models import Sequential
from keras.layers import Activation, Dense, Dropout
from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras.utils import to_categorical, plot_model
from keras.datasets import mnist
```

3.1 Network Architecture

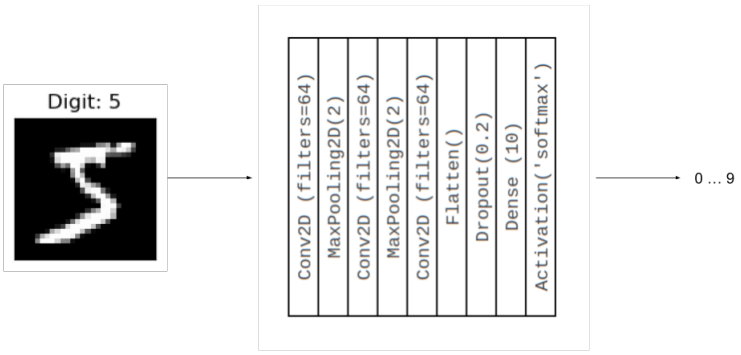

Figure 2: CNN model for MNIST digit classification

The model design process is the most complex factor, having a direct impact on the performance of the model.

Figure 2 shows the CNN model that we'll use for the MNIST digit classification. Instead of having input vector, the input tensor now has new dimensions (height, width, channels) or (image_size, image_size, 1) = (28, 28, 1) for the grayscale MNIST images. Resizing the train and test images will be needed to conform to this input shape requirement.

```
# image is processed as is (square grayscale)
input_shape = (image_size, image_size, 1)
batch_size = 64
kernel_size = 3
pool_size = 2
filters = 64
dropout = 0.3
```

To create the model, we first initialize a sequential model. It creates an empty model object. The first step is to add a convolutional layer which takes the input image:

```
# model is a stack of CNN-ReLU-MaxPooling
model = Sequential()
model.add(Conv2D(filters=filters,
        kernel_size=kernel_size,
        activation='relu',
        input_shape=input_shape))
model.add(MaxPooling2D(pool_size))
model.add(Conv2D(filters=filters,
        kernel_size=kernel_size,
        activation='relu'))
model.add(MaxPooling2D(pool_size))
model.add(Conv2D(filters=filters,
        kernel_size=kernel_size,
        activation='relu'))
model.add(Flatten())
```

Next, we add a "dropout" layer. While neural networks are trained on huge datasets, a problem of overfitting may occur. To avoid this issue, we randomly drop units and their connections during the training process. In this case, we'll drop 30% of the units:

```
# dropout added as regularizer
model.add(Dropout(dropout))

# output layer is 10-dim one-hot vector
model.add(Dense(num_labels))
model.add(Activation('softmax'))
model.summary()
plot_model(model, to_file='cnn-mnist.png', show_shapes=True)

# loss function for one-hot vector
# use of adam optimizer
# accuracy is good metric for classification tasks
```

```
13 model.compile(loss='categorical_crossentropy',
14         optimizer='adam',
15         metrics=['accuracy'])
16
17 # train the network
18 model.fit(x_train, y_train, epochs=15, batch_size=batch_size)
19 loss, acc = model.evaluate(x_test, y_test, batch_size=batch_size)
20 print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

The "softmax" activation is used when we'd like to classify the data into a number of predecided classes.

Since only one result can be correct, we will use categorical crossentropy in the classification problem. As for the optimizer of choice we'll use Adam with default settings

3.2 Pooling Layers

In our NN conception, we have added a MaxPooling2D layer with the argument pool_size=2. MaxPooling2D compresses each feature map. Every patch of size pool_size × pool_size is reduced to one pixel. The value is equal to the maximum pixel value within the patch. MaxPooling2D is shown in the following figure for two patches

The significance of MaxPooling2D is the reduction in feature maps size which translates to increased kernel coverage. For example, after MaxPooling2D(2), the 2 × 2 kernel is now approximately convolving with a 4 × 4 patch. The CNN has learned a new set of feature maps for a different coverage. There are other means of pooling and compression. For example, to achieve a 50% size reduction as MaxPooling2D(2), AveragePooling2D(2) takes the average of a patch instead of finding the maximum. Strided convolution, Conv2D(strides=2,...) will skip every two pixels during convolution and will still have the same 50% size reduction effect. There are subtle differences in the effectiveness of each reduction technique.

In Conv2D and MaxPooling2D, both pool_size and kernel can be non-square. In these cases, both the row and column sizes must be indicated. For example, pool_size=(1, 2) and kernel=(3, 5).

The output of the last MaxPooling2D is a stack of feature maps. The role of Flatten is to convert the stack of feature maps into a vector format that is suitable for either Dropout or Dense layers.

3.3 Model Summary

Using the Keras library provides us with a quick mechanism to double check the model description by calling:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dropout (Dropout)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 10)                5770
_________________________________________________________________
activation (Activation)      (None, 10)                0
=================================================================
Total params: 80,266
Trainable params: 80,266
Non-trainable params: 0
```

Figure 3: Model summary

4 Performance Evaluation

At this point, the model for the MNIST digit classifier is now complete. Performance evaluation will be the next crucial step to determine if the proposed model has come up with a satisfactory solution. Training the model for 15 epochs will be sufficient to obtain comparable performance metrics.

4.1 Dopout value

Note that we are optimizing with a dropout of 0.3%. For the sake of completeness, it could be useful to report the accuracy on the test dataset for different dropout values (Figure 4). We have selected SGD() as the optimizer.

4.2 Testing different optimizers

TensorFlow implements a fast variant of gradient descent known as SGD and many more advanced optimization techniques such as RMSProp and Adam. RMSProp and Adam include the concept of momentum (a velocity component), in addition to the acceleration component that SGD has. This allows faster convergence at the cost of more computation. It can be proven that momentum helps accelerate SGD in the relevant direction and dampens oscillations. The

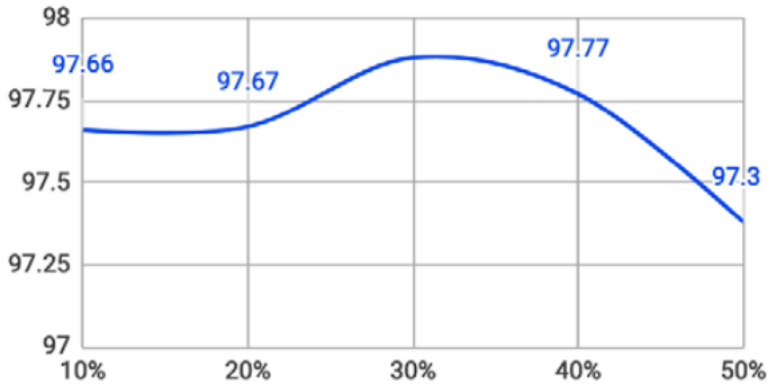

Figure 4: Changes of accuracy for different Dropout values

following table, Table 1, shows the different network configurations and corresponding performance measures.

Layers	Optimizer	Regularizer	Train Accuracy %	Test Accuracy %
64-64-64	SGD	Dropout(0.3)	96.4	97.33
64-64-64	RMSprop	Dropout(0.3)	99.47	99.2
64-64-64	Adam	Dropout(0.3)	99.54	99.4
64-64-64	Adam	Dropout(0.2)	99.33	99.2

Table 1: Different CNN network configurations and performance measures

As we can see in the preceding table, Adam is faster than both RMSProp and SDG since we are able to achieve in only 15 epochs an accuracy of 99.54% on training and 99.4% on test. That's a significant improvement on SDG.

4.3 Precision, Recall, F1

Precision-Recall is a useful measure of success of prediction when the classes are very imbalanced. In information retrieval, precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned.

The precision-recall curve (Figure 6) shows the tradeoff between precision and recall for different threshold. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall).

4.4 ROC Curve

In addition, the Area Under the Cover Receiver Operating Characteristic (AUC ROC) curve is represented. This ranges from zero to one, and a higher value indicates a higher-quality model.

	precision	recall	f1-score	support
0	0.99	0.99	0.99	980
1	0.99	1.00	1.00	1135
2	1.00	0.99	0.99	1032
3	1.00	0.99	0.99	1010
4	0.99	0.99	0.99	982
5	0.99	0.99	0.99	892
6	1.00	0.99	1.00	958
7	0.99	0.99	0.99	1028
8	1.00	0.99	1.00	974
9	0.99	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

Figure 5: Precision, recall and F1-score for our model

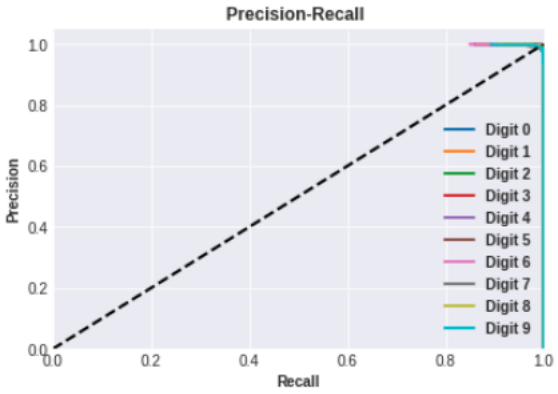

Figure 6: Precision recall plot for our model

This statistic summarizes a AUC ROC curve, which is a graph showing the performance of a classification model at all classification thresholds.

After training the model we can use it to make predictions for test inputs and plot ROC for each of the 10 classes. Before doing that, let's define the metric to evaluate the overall performance across all classes. There are two slightly different metrics, micro and macro averaging.

In "micro averaging", we'd calculate the performance, e.g., precision, from the individual true positives, true negatives, false positives, and false negatives of the k-class model:

$$precision = PRE = \frac{TP}{TP + FP}$$

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

And in macro-averaging, we average the performances of each individual class:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k k}$$

We can see for each class, their ROC and AUC values are slightly different, that gives us a

good indication of how good our model is at classifying individual class.

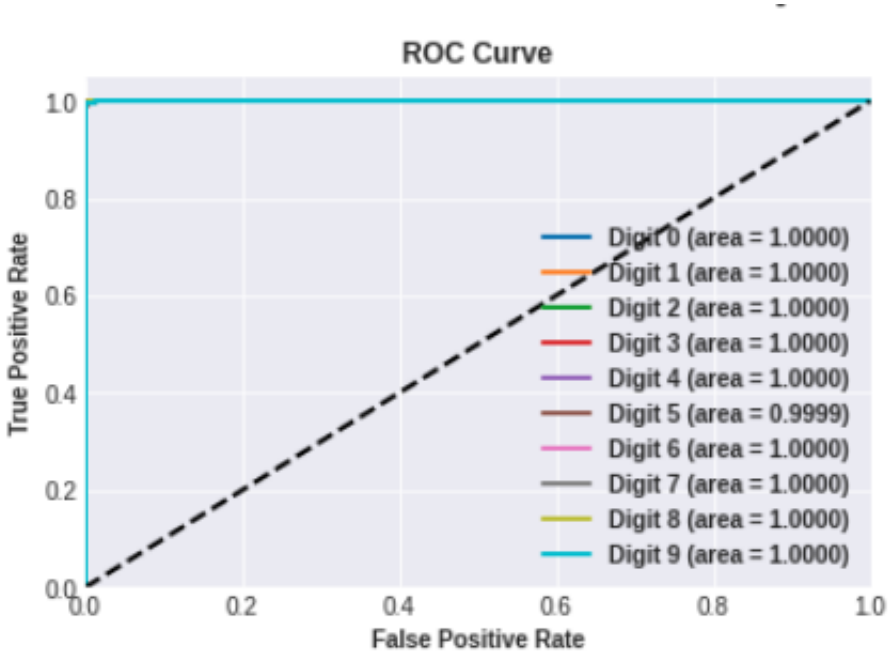

Figure 7: Receiver operating characteristic Curve for our model

4.5 Confusion Matrix

One the great methods to see how well our model can predict or classify is the confusion matrix. A confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. It allows the visualization of the performance of an algorithm. It allows easy identification of confusion between classes e.g. one class is commonly mislabeled as the other. Most performance measures are computed from the confusion matrix

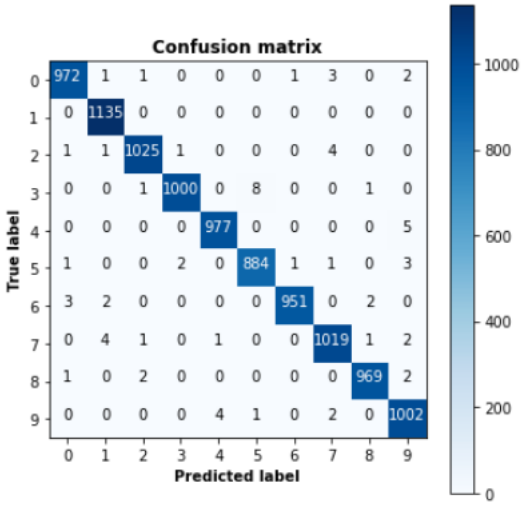

Figure 8: Confusion matrix for our model

As we can see in figure 8, our model is pretty good at classifying which digit is what.

5 Effect of Hyperparameters

We hence forth discuss the tuning problem whose main aim is to set hyperparameters, a parameter whose value is used to control the learning process.

5.1 Number of Epochs

Now that we have a very fast optimizer, let us try to significantly increase the number of epochs up to 50.

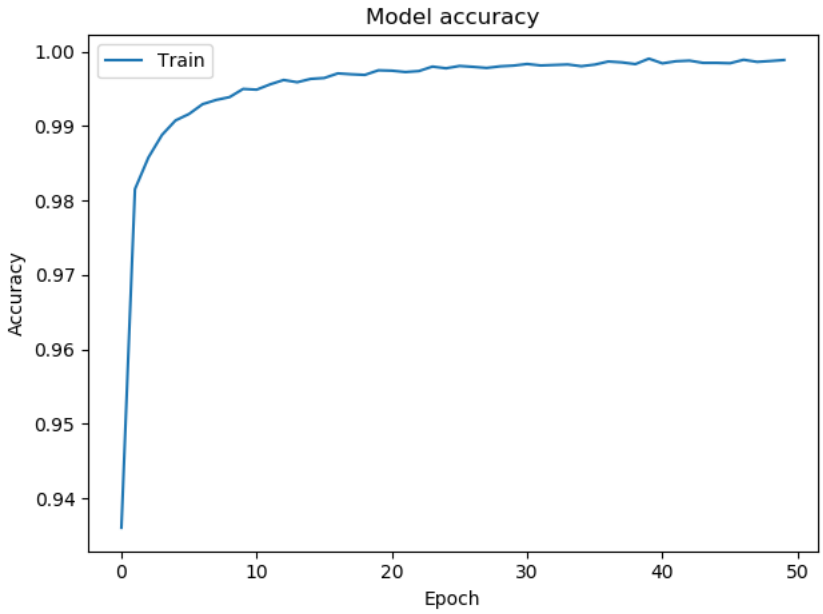

Figure 9: Epoch accuracy with Adam

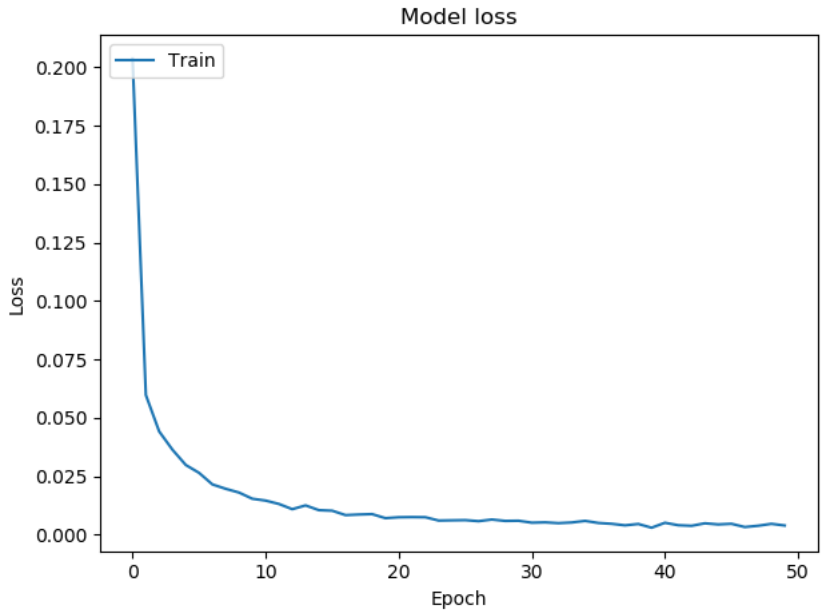

Figure 10: Epoch loss with Adam

It is useful to observe how accuracy increases on training and test sets when the number of epochs increases (Figure 9). It seems then that a number of 15 epochs is a fair choice. Therefore, there is no need to train further after that point.

Having tried an attempt of increasing the number of epochs used for training from 15 to 200. Unfortunately, this choice increases our computation time tenfold, yet gives us no gain. The experiment is unsuccessful, but we have learned that if we spend more time learning, we will not necessarily improve the result. Learning is more about adopting smart techniques and not necessarily about the time spent in computations.

5.2 Batch size

Batch size is the number of training examples utilized in one iteration. Documentation suggests that in general, batch size of 32 is a good starting point. Sizes 64, 128, and 256 are also good to try. Other values (lower or higher) may be fine for some data sets, but the given range is generally the best to start experimenting with. Our next aim was to determining optimal batch size to maximize process capacity for our NN.

Gradient descent tries to minimize the cost function on all the examples provided in the training sets and, at the same time, for all the features provided in input. SGD is a much less expensive variant that considers only BATCH_SIZE examples. So, let us see how it behaves when we change this parameter. As you can see, the best accuracy value is reached for a BATCH_SIZE=64 in our four experiments (Figure 11).

Generally the bigger the batch, the more stable our stochastic gradient descent updates will be. But due to GPU memory limitations, we're going for a batch size of 64 and 10 epochs.

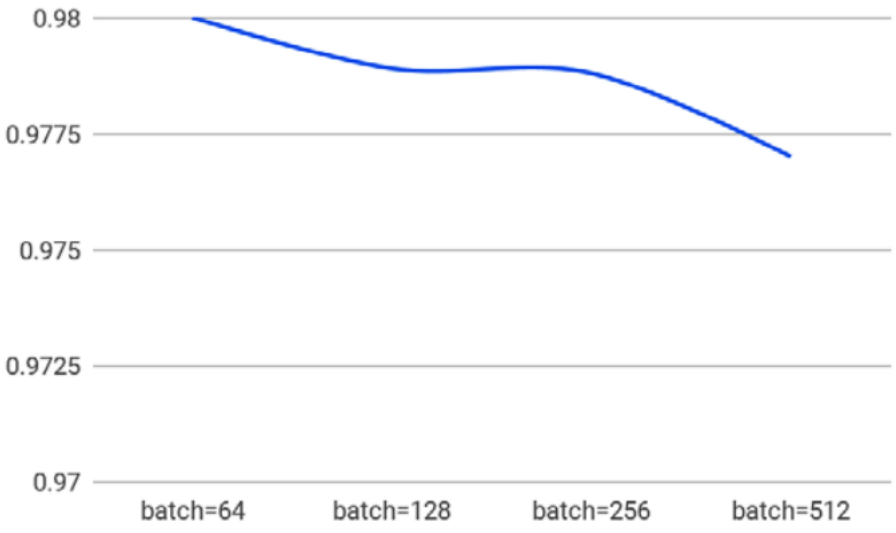

Figure 11: Test Accuracy for different batch values

5.3 Learning Rate

Learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function. It represents the speed at which the network learns.
We can observe that too high learning rate will make the learning jump over minima but a too low learning rate does either take too long to converge or get stuck in an undesirable local minimum.

6 Other Perspectives

6.1 Saving the model

```
model.summary()
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model identical to the previous one
model = load_model('my_model.h5')
```

6.2 Using our own handwritten digits

It's time for adding our own images to training and test data. For that below function will help us accomplish our goal:

```
# To load images to features and labels
def load_images_to_data(image_label, image_directory, features_data, label_data):
    list_of_files = os.listdir(image_directory)
    for file in list_of_files:
        image_file_name = os.path.join(image_directory, file)
        if ".png" in image_file_name:
            img = Image.open(image_file_name).convert("L")
            img = np.resize(img, (28,28,1))
            im2arr = np.array(img)
            im2arr = im2arr.reshape(1,28,28,1)
            features_data = np.append(features_data, im2arr, axis=0)
            label_data = np.append(label_data, [image_label], axis=0)
    return features_data, label_data
```

Basically, this function takes image label, image directory, features data, labels data as input. It lists all files present in image directory and then checks whether it is png file or not. Then loads the image and convert that to an array which is similar to features data and adds the image array to it. Takes image label and add that to label_data. Once it adds all our images in that folder/directory to current dataset returns them back.

Now let's give our own images directories to load them to existing dataset:

```
# Load your own images to training and test data
X_train, y_train = load_images_to_data('1', 'data/train/1',
                                       X_train, y_train)
X_test, y_test = load_images_to_data('1', 'data/validation/1',
                                       X_test, y_test)
```

To generate our dataset we will be using this awesome website that allows us to do so: https://agenis.shinyapps.io/handwriting/

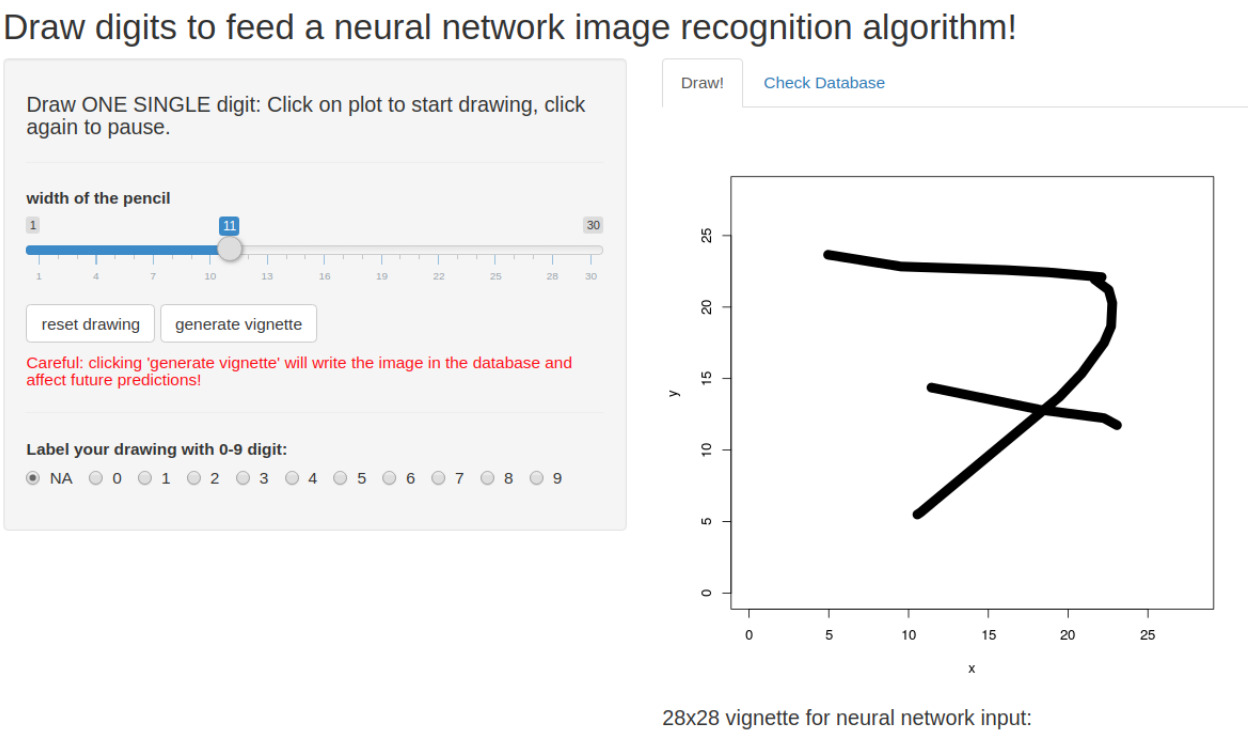

Figure 12: Drawing handwritten digits

7 Conclusion

After tuning the hyperparameters and playing with the model, we conclude the bellow features:

- Optimizer: Adam
- Loss-function: categorical_crossentropy
- Batch_size: 64
- Type of pooling: Maximum Pooling 2D
- Number of dense layers: 1
- Dropout: 0.3
- number of filters: 64

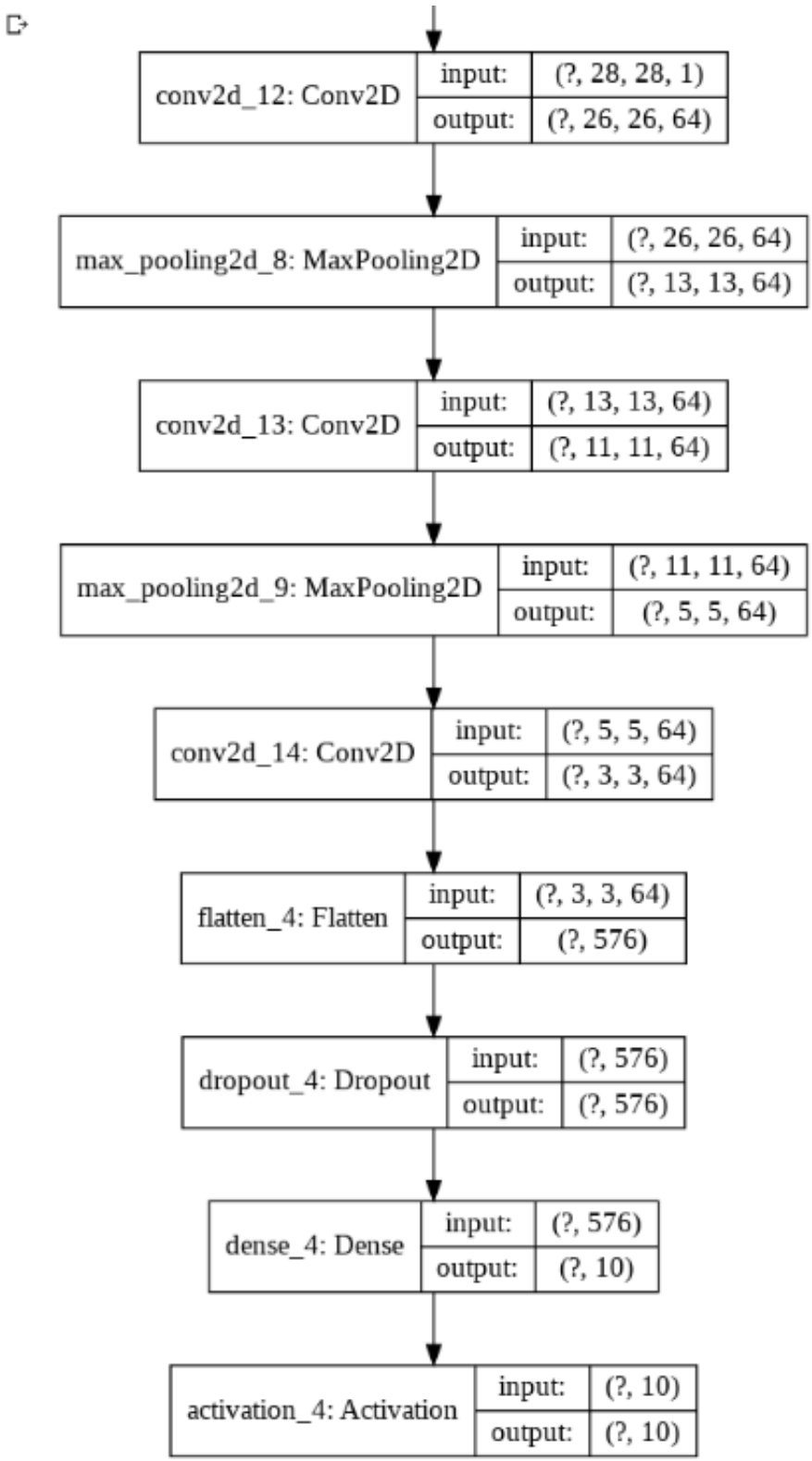

Figure 13: Visualization of our model