Handwritten Digit Recognition with CNNs

Ekrem Ozturk

2012400006

December 8, 2017

Abstract

We are at the century of data. Agents (cell phones, computers, surveillence cameras) around us are constantly gathering information. In 2020, collected data estimated to be approximately 40000 exabytes. Data is very important for training artificial agents. As we learn through out experiencing, they learn through data. Ironically their infinite life is very short for the market to learn slowly by experiencing. Fortunately, developments in the technology and increase in the size of collected data allows us to interpret huge data, teach our models to classify etc. Handwritten digit recognition is de facto in data science and in this project I compare different models and obtain up to 99.57 % prediction accuracy.

Introduction

Neural networks is a concept that exists for quite some time. However, it was not this popular until now. Incapability in solving XOR problem and then in recognition problems overshaded neural networks. However, pioneers like Geoffrey Hinton does not stopped researching on neural networks. He introduced back-propagation and multilayers and now we call this area as deep learning. Since 2012 neural networks takes lots of credits in learning models. In this project, I am building a convolutional neural network and training it to recognize handwritten digits. Since I am new to computer vision, I will try different models and functions and interpret the results.

MNIST Dataset

MNIST released in 1999, as a part of NIST datasets. It is very famous and also used in benchmarking different algorithms. Since I am in learning period, I have choosen MNIST.

The dataset is available in Yann Lecun's website. It is also easily downloadable within a command in any python interpreter. I have used the second method. There are 55000 images in the training set, and 10000 in the test set. They are 28x28 pixels and vectorized to 1x784 in dataset. So the training set is 55000x784 and test is 10000x784 matrix. From now on I will call training images and labes as x-train, y-train, and the test as x-test and y-test.

Figure 1: Handwritten digits in MNIST

Firstly, I had to reshape them to use. I have created 4 dimensional 2 tensors for x-train and x-test, (55000, 28, 28, 1) and (10000, 28, 28, 1) . They contain the four dimension that I need: Image id, row pixels, column pixels and number of channels. Since the images were black&white, I have needed only 1 channel.

Secondly, I had to turn the labels into interpretible data. Because I am using mathematical functions, and if these functions take 7 as superior to 5, the model cannot learn. In order to prevent this, I have encoded labels to categories(i.e 55000x10 and 10000x10 matrices). The data is well prepared and is ready for the model.

Implementation of the Model

I have started very simple and then increased complexity of the model. Adding more layers or neurons, augmenting the data, changing optimizer and loss functions are several things that I have tried to get better results. There are lots of loss functions and optimizers, but I have tried the ones that seems legitimate to me for this dataset.

Loss functions are calculating the cost of predicted values compared to real ones. So, smaller loss means less error. I have compared 2 functions: Mean squared error and categorical crossentropy.

Optimizer functions are adjusting weights after each batch. It is important because backpropagation depends on optimizer. I have compared 3 optimizer functions: Adagrad, rmsprop and adam.

Data augmentation is important in computer vision. It has 2 main advantages: First, with augmentation, one can have more data on hand to train their network. Secondly, augmentation helps the algorithm to learn better by spoiling images. Incoming data after the training might be unhealthy, i.e bad handwrittten digit in our case. I have discovered another reason that I will explain in evaluation. My model crops, zooms, rotates the image in certain ratios.

The idea of CNNs are coming from our visual system. We always use adjacent pixels to recognize. For example when we look to a person's face, we process the pixels near the eye to recognize it, not the mouth. Edge, orientation, motion, color

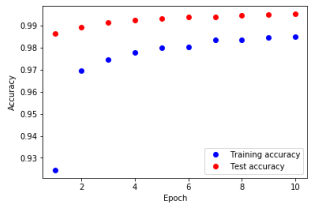
are features of an object that provides saliency and helps us to extract and attend.

Convolution, pooling and flattening are fundamental layers of a CNN. Convolutional layer uses feature detectors to extract feature maps. These maps help us to 'where to look at'. Pooling layer samples the feature maps to process them easily. I have used 2x2 max pooling layer for which takes the max value in pixels of 4. That reduces the size of the data by 4 without spoiling it. Flattening layer just vectorizes the pixels to use them as input, because we have to fully connect to neurons. I have built several models consisting different number of convolution and pooling layers.

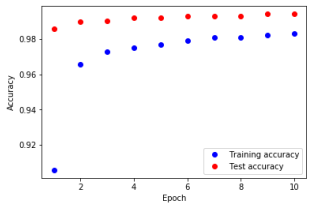
There is input layer of neurons for training and output layer of 10 neurons representing each digit. Output is possibility of the image being that number. I have used rectifier unit function on activation of neurons on all layers except output, which I have used 'softmax'.

Evaluation

Loss functions I have compared mean squared error and categorical crossentropy loss functions. I have intuitively thought that crossentropy would score better, because output is a possibility and it is good function for showing inefficiency of model's predictions.

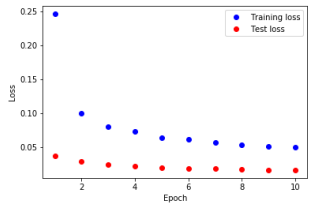


(a) Categorical crossentropy

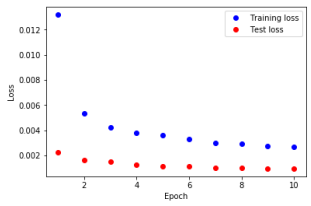


(b) Mean squared error

Figure 2: Accuracies



(a) Categorical crossentropy



(b) Mean squared error

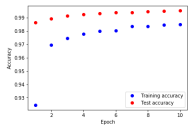
Figure 3: Losses

Function	**Accuracy**	**Loss**
Mean squared error	**0.9941**	**0.0009**
Categorical crossentropy	**0.9952**	**0.0161**

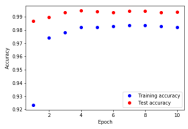
Table 1: Best scores

Optimizers I have compared rmsprop, adagrad and adam optimization algorithms. They are all adaptive gradient descent algorithms and actually they all performed well for this particular problem. Adagrad is my personal favorite but one issue for adagrad is that learning rate continues to shrink to infinitesimal value. Rmsprop and adam solves this issue. In this problem I have trained the model for

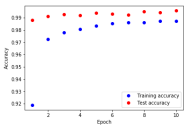
10 epochs because of the low processing power of my pc. Because of that, adagrad's issue never came up.



(a) Adagrad

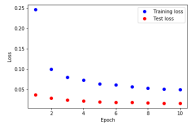


(b) Rmsprop

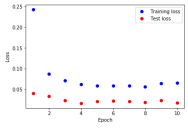


(c) Adam

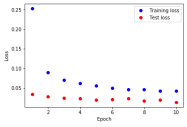
Figure 4: Accuracies



(a) Adagrad



(b) Rmsprop



(c) Adam

Figure 5: Losses

Optimizer	**Accuracy**	**Loss**
Rmsprop	**0.9936**	**0.0187**
Adam	**0.9957**	**0.0133**
Adagrad	**0.9952**	**0.0161**

Table 2: Best scores

Overfitting One issue that concerns all learning procedures is overfitting. We can understand if our model is whether overfitted to data is comparing training and test accuracies. We can see that in all models above, there is no overfitting which means they all well constructed. Dropout is a method to overcome this issue by

dropping a percentage of neurons randomly. There are 3 dropout layers, two with % 20 rate between convolutional and one with % 30 rate after hidden input layer.

One thing that I have discovered was data augmentation's help in overcoming overfitting. I have realized this when I trained the same model with only difference in implementation of augmentation. I think twisted data helps model to overtrain certain neurons, analogous with concept of dropouts. Below is the model trained without dropout and data augmentation. The test accuracy decreases as training converges to 1.

(a) Accuracy (b) Loss

Figure 6: Training of model without dropout layers and data augmentation

Conclusion

Training a neural network model with MNIST dataset is de facto in deep learning. I wanted to try different models and learn, as my models learn, because I am aware of just building and training a model on MNIST provides great results approximately 0.997 and so is rather easy with respect to other datasets. It is very educatory training for enthusiasts like me in the beginning of their journey in artificial intelligence.

Appendix

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: ekrem
"""

# Importing numpy and matplotlib

import numpy as np
import matplotlib.pyplot as plt
import time

current_milli_time = lambda: int(round(time.time()))

# Downloading and scaling the data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
X_train = np.reshape(mnist.train.images, (55000, 28, 28, 1))
y_train = mnist.train.labels
X_test = np.reshape(mnist.test.images, (10000, 28, 28, 1))
y_test = mnist.test.labels

# Importing Keras libraries and packages
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.layers import Dropout

# Building the CNN model
classifier = Sequential()

classifier.add(Conv2D(64, (3, 3), input_shape=(28, 28, 1),
                                  activation='relu'))
classifier.add(Conv2D(64, (3, 3), activation='relu'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))
classifier.add(Dropout(0.2))

classifier.add(Conv2D(64, (3, 3), activation='relu'))
classifier.add(Conv2D(64, (3, 3), activation='relu'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))
classifier.add(Dropout(0.2))
```

```
classifier.add(Flatten())

classifier.add(Dense(units=128, activation='relu'))
classifier.add(Dropout(0.3))
classifier.add(Dense(units=10, activation='softmax'))

classifier.compile(optimizer = 'adadelta',
                   loss = 'categorical_crossentropy',
                   metrics = ['accuracy'])

# Data augmentation and training
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(shear_range=0.2, zoom_range=0
                                   .2, rotation_range=10)

training_set = train_datagen.flow(X_train, y_train, batch_size=
                                   64)

time_before = current_milli_time()

training_log = classifier.fit_generator(training_set,
                                   steps_per_epoch=55000/64, epochs
                                   =10, validation_data=(X_test,
                                   y_test))

time_after = current_milli_time()
m, s = divmod(time_after-time_before, 60)
h, m = divmod(m, 60)

# Acquiring last score

score = classifier.evaluate(X_test, y_test, batch_size=64)

# Acquiring the results from history

result_training_acc = training_log.history['acc']
result_test_acc = training_log.history['val_acc']
result_training_loss = training_log.history['loss']
result_test_loss = training_log.history['val_loss']

# Plotting the results

x = np.arange(1,11,1)
```

```
# Plotting accuracy
plt.plot(x ,result_training_acc, 'bo')
plt.plot(x ,result_test_acc, 'ro')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Training accuracy','Test accuracy'])
plt.show()

# Plotting losses
plt.plot(x ,result_training_loss, 'bo')
plt.plot(x ,result_test_loss, 'ro')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Training loss','Test loss'])
plt.show()

# Saving models and weights
model_json = classifier.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
classifier.save_weights("model_weights.h5")
json_file.close()
```

References

1. https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html
2. http://yann.lecun.com/exdb/mnist/
3. https://www.tensorflow.org
4. http://www.wikizero.org/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3
5. http://ruder.io/optimizing-gradient-descent/
6. https://datascience.stackexchange.com/questions/9302/the-cross-entropy-error-function-in-neural-networks
7. https://www.udemy.com/deeplearning/