AMATH 482 HW3: Music Genre Identification

Ken Lo

March 10, 2018

Abstract

Built a neural network to classify handwritten digits, using the MNIST dataset. The neural network trained were able to return very accurate results classifying new handwritten digits (also from the MNIST data set).

1 Introduction and Overview

In this assignment, I used the MNIST handwritten digit dataset to train both a 1-layer and 2-layer neural networks, which then are used to classify the testing data. The accuracy of the two neural networks are then compared.

2 Theoretical Background

A Neural Network (NN) are consisted of layer of nodes. Firstly, we have the input layer, which is the training dataset for our neural network. The last layer is the output layer, also known as perceptrons. The middle layer(s) are functions that map the previous layer to the next layer, and eventually the the perceptrons. We can think of these middle layers as matrices A_i, where i denotes the ordering of the layer. Then, in mathematical expression, the neural network can be represented by

$$y = A_n A_{n-1} A_{n-2} \cdots x$$

where y is the perceptron(s), and x is the input.

2.1 Convolutional Layers

Convolutional Layers use a small window that slides across the entire layer and transfer the data into a new node through a given activation function.

2.2 Pooling Layers

Pooling Layers are generally recommended to use alongside a convolutional layer to reduce the number of parameters and computation in the network. The type of pooling used in this assignment is max pooling, which takes the maximum value for all the nodes in a convolutional window.

2.3 Activation Functions

Activation functions are functions that will be differentiated in order to be used in gradient descent algorithms for optimizing the neural network training. A common activation function is ReLU.

2.4 Gradient Descent

Training a neural network in a timely manner requires a gradient descent algorithm for optimization. The Stochastic Gradient Descent Algorithm is an iteration algorithm

$$x_{j+1}(\tau) = x_j - \tau \nabla f_k(x_j)$$

where the gradient is computed using only the kth data point and $f_k(\cdot)$. Thus, instead of computing over all data points, only a single point is randomly selected. Using batches of points instead of a single point can also be used for this algorithm.

3 Algorithm Implementation and Development

I used Python Keras package to build and train the neural networks. The one-layer neural network will consists of only a Convolutional Layer, using Stochastic Gradient Descent as the optimizing function and ReLU as the activation function.

The two-layer neural network will have an addition pooling layer after the previous convolutional layer, using the same optimizing function and ReLU.

Both neural networks are trained and then cross-validated by splitting the training dataset into multiple random train and test set.

4 Computational Results

4.1 1-layer NN

```
Training Results
Epoch 1/10
60000/60000 [==============================] - 24s 396us/step - loss: 0.0283 - acc: 0.822
Epoch 2/10
60000/60000 [==============================] - 23s 388us/step - loss: 0.0138 - acc: 0.910
Epoch 3/10
60000/60000 [==============================] - 23s 390us/step - loss: 0.0127 - acc: 0.917
Epoch 4/10
60000/60000 [==============================] - 23s 389us/step - loss: 0.0121 - acc: 0.921
Epoch 5/10
60000/60000 [==============================] - 24s 402us/step - loss: 0.0116 - acc: 0.924
Epoch 6/10
60000/60000 [==============================] - 28s 464us/step - loss: 0.0111 - acc: 0.927
Epoch 7/10
60000/60000 [==============================] - 29s 483us/step - loss: 0.0107 - acc: 0.931
Epoch 8/10
60000/60000 [==============================] - 34s 571us/step - loss: 0.0102 - acc: 0.934
Epoch 9/10
60000/60000 [==============================] - 35s 581us/step - loss: 0.0097 - acc: 0.938
Epoch 10/10
60000/60000 [==============================] - 32s 533us/step - loss: 0.0092 - acc: 0.941
```

From the above training information, we see that the network gradually improved in accuracy, and finished with a 94% accuracy. Training this network took about 10 minutes.

```
Cross-Validation Scores:
20004/20004 [==============================] - 3s 156us/step
Score: [0.012143852326196311, 0.92061587682463508]
19999/19999 [==============================] - 3s 153us/step
```

```
Score: [0.0099316269636154728, 0.93594679729814168]
19997/19997 [==============================] - 3s 157us/step
Score: [0.0072675501430646923, 0.95549332399859976]
```

Then we randomly splitted the training set into multiple train and test sets. The accuracy ranged from 92% to 95%.

Testing this model on the official test dataset, the accuracy is 94%, which is pretty similar to our cross-validation results.

4.2 2-layer network

```
Training
Epoch 1/10
60000/60000 [==============================] - 32s 530us/step - loss: 0.3480 - acc: 0.896
Epoch 2/10
60000/60000 [==============================] - 29s 490us/step - loss: 0.1537 - acc: 0.956
Epoch 3/10
60000/60000 [==============================] - 29s 488us/step - loss: 0.1016 - acc: 0.971
Epoch 4/10
60000/60000 [==============================] - 30s 502us/step - loss: 0.0808 - acc: 0.976
Epoch 5/10
60000/60000 [==============================] - 30s 501us/step - loss: 0.0701 - acc: 0.979
Epoch 6/10
60000/60000 [==============================] - 31s 511us/step - loss: 0.0626 - acc: 0.981
Epoch 7/10
60000/60000 [==============================] - 30s 506us/step - loss: 0.0571 - acc: 0.983
Epoch 8/10
60000/60000 [==============================] - 30s 506us/step - loss: 0.0528 - acc: 0.984
Epoch 9/10
60000/60000 [==============================] - 28s 466us/step - loss: 0.0490 - acc: 0.985
Epoch 10/10
60000/60000 [==============================] - 28s 467us/step - loss: 0.0459 - acc: 0.986
```

The above output shows the training time and accuracies for each epoch while training the 2-layer network, here we can see that the accuracies, as compared to a 1-layer network, improved by almost 4% at best, at around 98%, which is very accurate.

```
Cross Validation
20004/20004 [==============================] - 3s 150us/step
[0.080776027125734562, 0.97660467906418713]
19999/19999 [==============================] - 3s 125us/step
[0.056507525260152416, 0.98329916491354219]
19997/19997 [==============================] - 3s 125us/step
[0.039801153043284916, 0.98739810971645747]
```

The Cross Validation output above shows consistency when using different train and test sets.

The accuracy using the official test dataset on this 2-layer network 98%, consistent with the results in cross-validation as well as during training.

5 Summary and Conclusions

In this assignment we saw how accurate neural networks can be. By adding more layers to the network, accuracies can be improved. Even with only 1-layer, the result is still

extremely well, as compared to using normal machine learning models. However, the tradeoffs to an accuracy result is the training time, here, using only not a large amount of data sets and only minimal number of layers, it still took 10 minutes to train each. Thus, before we consider using a neural network, one should first consider whether it is absolutely necessary. Otherwise the improvement in accuracy as compared to normal ML models may not worth the extra time.

A Python Functions Used

- keras.model.add - add a layer to the neural network
- keras.model.compile - compile the neural network
- keras.model.fit - train the compiled model
- keras.model.evalue - test the trained model on the dataset
- sklearn.model_selection.stratifiedKFold - randomly split the data into multiple train and test sets

B Python Code

```
# coding: utf-8

# In[60]:


import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D, Conv2D
from keras.utils import np_utils
from keras.datasets import mnist
from keras import optimizers


# In[4]:


np.random.seed(482)


# In[5]:


# load data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
print(x_train.shape) # 60000 images, 28x28 pixels per image


# In[7]:
```

```
# plot first sample image of x_train
import matplotlib.pyplot as plt
plt.imshow(x_train[0])
plt.show()


# In[13]:


x_train.dtype


# In[10]:


# normalize the values to [0, 1], as a float
x_train_norm = x_train.astype('float32')
x_test_norm = x_test.astype('float32')
x_train_norm /= 255
x_test_norm /= 255


# In[25]:


# reshape train and test data
x_train_norm_reshaped = x_train_norm.reshape(x_train_norm.shape[0], 28, 28, 1) # samples,
x_test_norm_reshaped = x_test_norm.reshape(x_test_norm.shape[0], 28, 28, 1)
print(x_train_norm_reshaped.shape) # confirm reshape is correct


# In[26]:


print('train label shape', y_train.shape)
print('first 10 labels', y_train[:10])


# In[27]:


# convert train labels to categorical classes
y_train_c = np_utils.to_categorical(y_train, 10)
y_test_c = np_utils.to_categorical(y_test,10)
print('label shape now:', y_train_c.shape)
print('first 10 labels:\n', y_train_c[:10])


# In[82]:


# build NN model(1-layer)
model = Sequential()
```

```
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28,28,1)))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
model.compile(loss='mse', optimizer=optimizers.sgd(lr=0.1), metrics=['accuracy'])


# In[43]:


y_train_c_reshaped = y_train_c.reshape(y_train_c.shape[0],1,1,10)
y_test_c_reshaped = y_test_c.reshape(y_test_c.shape[0],1,1,10)


# In[53]:


model.output_shape


# In[67]:


# fit model
model.fit(x_train_norm_reshaped, y_train_c, batch_size=32, epochs=10, verbose=1)


# In[68]:


# score the model with the test data
score = model.evaluate(x_test_norm_reshaped, y_test_c, verbose=1)
score


# In[83]:


# cross validation
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=3)
print('Scores:')
for traini, testi in skf.split(x_train_norm_reshaped, y_train):
    model.fit(x_train_norm_reshaped[traini], y_train_c[traini], batch_size=32, epochs=8,
    print('Score:', model.evaluate(x_train_norm_reshaped[testi], y_train_c[testi]))


# In[84]:


# build 2-layer NN model
model2 = Sequential()
model2.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28,28,1))) # first layer 2d
model2.add(MaxPooling2D(pool_size=(2,2))) # second later 2d max pooling
```

```
model2.add(Flatten())
model2.add(Dense(10, activation='softmax'))
model2.compile(loss='categorical_crossentropy', optimizer=optimizers.sgd(lr=0.05), metric


# In[74]:


model2.fit(x_train_norm_reshaped, y_train_c, batch_size=32, epochs=10, verbose=1)


# In[75]:


score2 = model2.evaluate(x_test_norm_reshaped, y_test_c, verbose=1)
score2


# In[86]:


# cross validation for 2-layer NN
skf = StratifiedKFold(n_splits=3)
print('Scores:')
for traini, testi in skf.split(x_train_norm_reshaped, y_train):
    model2.fit(x_train_norm_reshaped[traini], y_train_c[traini], batch_size=32, epochs=8,
    print(model2.evaluate(x_train_norm_reshaped[testi], y_train_c[testi]))
```