Title: Assignment 1: Create a CNN capable of classifying the MNIST dataset to an accuracy of greater than 99%

Author: Oisin Watkins: 15156176

Date: 05/11/18

Introduction

This Project was outlined in the first week of semester with loose guidelines, meaning that students could take one of multiple directions so long as the end goal of classifying the MNIST testing dataset to an accuracy greater than 99% was achieved honestly. Multiple topologies, optimisers, compilers, loss measurements and metrics can be used to the same end, however this report will only detail one solution in the following sections. This machine will have 2 convolutional layers, followed by Maxpooling players, dropouts, flattening layers and a densely connected neural network layers.

Three separate testing runs were run on this particular machine, which yielded some note worthy results. Without delaying any further, let's inspect the code used to create the machine and the output from its execution.

Code:

Herein is printed the code written to implement the Convolutional Neural Network (CNN)

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

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32')/255
train_labels = to_categorical(train_labels)

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32')/255
test_labels = to_categorical(test_labels)

network = Sequential()
network.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28,
1)))
network.add(Conv2D(64, (3, 3), activation='relu'))
network.add(MaxPooling2D(pool_size=(2, 2)))
network.add(Dropout(0.25))
network.add(Flatten())
network.add(Dense(128, activation='relu'))
network.add(Dropout(0.5))
network.add(Dense(10, activation='softmax'))

network.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

history = network.fit(train_images, train_labels, epochs=10, batch_size=128,
validation_split=0.1)

results = network.evaluate(test_images, test_labels)
print("\n\n")
print(results)
```

The first job with this network, as with any network, is to reshape the input data to suit the coming network design. In this project it makes very little sense to use a 3-D convolutional kernel, hence the input will be reshaped into a series of 28x28 pixel images, 60000 for training and 10000 for testing. The corresponding labels are then categorised and the network initialised.

The network is then initialised as a Sequential model. This means only that every layer after the first layer will inherit its input shape from the output shape of the previous layer. This leaves the input shape of the first layer up to the designer's choice, and naturally here the input shape is set to match that of one 28x28 image. This first layer will have 32 input nodes and a kernel size of 3x3. This should keep computation at the input layer to a reasonably low level. The next layer was added to provide a higher classification accuracy, having 62 nodes in its convolutional topography and the same 3x3 kernel size. The following two layers reduce the complexity of the problem for the coming layers, these layers being a Maxpooling layer, which takes a 2x2 kernel and passes it over the input data and for every 4 values given to the layer only the largest value is fed to the next layer. This next layer is a Dropout layer, which acts as a thresholding function only allowing values above 0.25 through to the Flatten layer to follow. This forces the data to take on a 2-dimensional form, where one axis is 1 and the other is the product of every axis the input had.

Lastly the Dense layers. These classify the filtered and reshaped data from all previous layers. The first and second dense layers, separated by another dropout layer, vary in their functions. The first Dense layer is similar to the convolutional layers previous to it in that it has the box standard Rectified Linear Until activation layer. However, the final dense layer has a softmax layer as its output layer. Given the nature of this network the output of this softmax layer can be interpreted as a probability measure, i.e. the output at every node is interpreted as a confidence score for the input being in that class. This means that there must be the same number of nodes in the output layer as there are classes in the problem. In this example the network is classifying digits from 0-9, therefore there are 10 classes, hence 10 softmax nodes.

Next the learning rules for this network are defined using the compile command. This specifies the weight update rule as "rmsprop", which has a moderately fast and highly accurate convergence, the loss as "categorical crossentropy", which is standard for neural networks, and the metric as "accuracy", meaning the number of correct vs incorrect classifications is determiner of success.

Finally, the network is trained according to the setup of 10 epochs, a batch size of 128 images, and a validation split of 10%. This all means that the network will iterate over 54,000 training images (90% of the sample data) in step sizes of 128 images, updating the weights at the end of each step and running a validation test at the end of each epoch, using the remaining 6,000 images (10% of the sample data) as tests. The information is stored into the variable "history". Once training is complete, the network is tested on the testing data and the results are printed to the screen.

The goal of this project, as defined at the beginning of the report was to achieve an accuracy of greater that 99%. The success or failure of this network is judged by the printed output, which will list the output as > 0.99 if the network succeeded or < 0.99 if the network failed. In the results section will be a quick detailing of some experimental outputs from this script.

Results:

The results given by the test are printed in the order [loss, accuracy]. The accuracy value is the one of interest. Below is listed the outputs from 3 separate runs of the script given in the Code section. Discussions will follow the presentation of the results.

Run1:

[0.033943157916094244, 0.99180000000000001]

Run 2:

[0.039295581829434739, 0.99019999999999997]

Run 3:

[0.037547186534340291, 0.98929999999999996]

The first 2 runs were successful in reaching the target specified in the project outline. The third however was a failure, falling short by less than 0.08%. This is likely due to the uptake in RAM usage by other processes being run on the machine. Given the very long run times (upwards of an hour usually), no more tests were conducted on this network, meaning this data above is all the data conclusions can be drawn from.

Conclusions:

Tests show a success rate of 67%, though it is worth bearing in mind the limitations of running CNN's on standard CPU's. Had it been possible to port this script over to a GPU instead, success rates would doubtless have increased. It is also likely that running on a GPU would increase the average accuracy of test runs, as well as lowering run-times. A GPU's high parallelisation capabilities allow it to handle neural networks with incredible efficiency when compared with other processor types.