Classification with CNN on MNIST Dataset

March 30, 2018

0.1 My changes

I changed the first conv2D from 32 and a kernel-size of 2,2 to 64 and 5,5

I also added an extra convolution layer and pooling layer after the first one.

the last change is a locallConnected2D layer

The original model gets an accuracy of 0.8583 with a loss of .58

Whereas the updated model gets an accuracy of .9863 and a loss of 0.04377

Note this is one of two notebooks, my other one goes over a bit of dependent Haskell and how the grenade library uses dependent types to do nerual networks and the issues I had running it.

Cite From Keras' Official Website: *Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.*

Keras is essentially a set of functions that calling a neural network backend (TensorFlow, CNTK, or Theano) to do related learning jobs. Keras has a good taste of design. After you understand the fundamentals in deep learning, you'll find Keras is very intuitive to use. Besides, Keras also has a very active community and provides complete documentations. **If you have something don't understand in the procss, I highly recommend you to read its documents at first.**

0.2 Installation and Setup

This simplest way to setup Keras is through **pip/pip3** (you can find more details here):

```
>> pip3 install tensorflow Keras
```

After successfully install Keras, we can import related classes into notebook's workspace.

```
In [42]: import keras
         from keras.models import Sequential
         from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
         from keras.layers import *
```

Here we import **sequential** model, which is a model with each elemental unit connected sequentially. This structure is simple but very common in deep learning. Besides, we includes different kinds of layers, which is the fundamental unit in deep learninng. Among them, **Dense** is the linear transformation layer, while **Conv2D** corresponds to convolutional transformation. **MaxPooling2D** is a layer that down-samples data in 2D with *max* operation. **Flatten** is a layer reshape the data to fit for linear transformation. You can find more detail for each layer in documents.

There is also a very good explanation of convolutional neural network (CNN), which is what we would use in this demo.

0.3 Obtain Dataset from Internet

The dataset we are going to deal with is MNIST, a handwriten digits dataset which you have seen before. It's simple, however, we need it in the form that match assumptions of Keras (which can be a hugh amount of week for your own dataset). Here, we directly fetch this popular dataset from Keras' dataset library. You can also find other datasets in it.

```
In [34]: from keras.datasets import mnist

         # download data from internet (only the first time) and split it into training and te
         (imgTrain, labelTrain), (imgTest, labelTest) = mnist.load_data()
```

However, our work haven't done yet. There are two things we need to do: 1. **Conv2D** layers only deal with 3D samples, which is for color images, however MNIST only have two dimensions. So we need to reshape it into fake 3D samples with the length of 3rd dimension being 1. 2. The value of each pixel in the dataset is integers from 0 to 255, which is not good for calculation. Here, we'll map them to float numbers in the range of $[0, 1]$.

One thing you should know is that different backend use different dimension order. So, here I use a conditional process to reshape the data. However, if you are confident to which one your backend use, you can directly reshape your data into corresponding form. In my case, I use **TensorFlow** as the backend. It puts dimension of color channel to the last dimension in *ndarray*.

```
In [35]: from keras import backend as K

         imgRows, imgCols = 28, 28 # size of source image

         # reshape images in training and testing set into fake 3D
         if K.image_data_format() == 'channels_first':
             imgTrain = imgTrain.reshape(imgTrain.shape[0], 1, imgRows, imgCols)
             imgTest  = imgTest.reshape(imgTest.shape[0], 1, imgRows, imgCols)
             smpSize  = (1, imgRows, imgCols)
         else:
             imgTrain = imgTrain.reshape(imgTrain.shape[0], imgRows, imgCols, 1)
             imgTest  = imgTest.reshape(imgTest.shape[0], imgRows, imgCols, 1)
             smpSize  = (imgRows, imgCols, 1)

         # convert pixels to floats and map them into range of [0,1]
         imgTrain = imgTrain.astype('float') / 255
         imgTest  = imgTest.astype('float') / 255

         # show shape and type of our datasets
         print('Training set in shape of ', imgTrain.shape, ' with element type ', type(imgTra
         print('Testing set in shape of  ', imgTest.shape, ' with element type ', type(imgTrai
```

```
Training set in shape of  (60000, 28, 28, 1)  with element type  <class 'float'>
Testing set in shape of   (10000, 28, 28, 1)  with element type  <class 'float'>
```

Now, we have done the preprocessing for image part. We also need to do one last thing for labels to fit for requirement of learning process. Basically, we are going to convert label of

each image, such as 1, in to one-hot vectors (here is a good blog to explain it in detail), like $[0,1,0,0,0,0,0,0,0,0]$. Becuase this operation is very common in classification, there is a function in Keras to deal with it.

```
In [36]: ncat = 10 # number of categories in our problem

         # convert labels to one-hot vectors
         onehotTrain = keras.utils.to_categorical(labelTrain, ncat)
         onehotTest  = keras.utils.to_categorical(labelTest, ncat)
```

0.4 Build the Model

For sequential model, the building process is essetially add different layers sequentially. **Here, I just use a naive structure. It will not achive best performance, however, enough to show the procedure.** You should try different structures to compare the performance. You can find the standard structure online, but, you also can try different ones as your idea.

```
In [63]: # claim a sequential model
         model = Sequential()

         # add first layer as convolution transformation with 32 filters of size [3, 3]
         # - 'activation' is the non-linear tranform used in neural network, a common one is '
         # - for the first layer, you also need to claim size of input samples
         model.add(Conv2D(64,
                          kernel_size=(5, 5),
                          activation='relu',
                          input_shape=smpSize))

         # add max-pooling layer
         model.add(MaxPooling2D(pool_size=(2, 2)))
         model.add(Conv2D(20,
                          kernel_size=(2, 2),
                          activation='relu',
                          input_shape=smpSize))

         model.add(MaxPooling2D(pool_size=(2, 2)))

         model.add(LocallyConnected2D(32, (3, 3)))
         # add flatten layer to reshape data to fit for linear transform
         model.add(Flatten())

         # linear transofrm with activation of softmax
         model.add(Dense(ncat, activation='softmax'))
```

Now, we need to compile the model. This is the limitation of current packages in deep learning : lack of dynamic features, which is also a hot topic in related community to create extensions to support dynamic feature. However, it still in development, you can try them, but be aware of there should be some problems haven't solved yet.

In the compilation, there are several options you can choose: 1. **loss**, the loss function (or objective function) describe how you evaluate the difference between truth labels with your model's prediction 2. **optimizer**, the mehtod your model will use in optimization, you can use default settings currently 3. **metrics**, is the evaluation method in showing train progress

```
In [64]: # compile the model
         model.compile(loss=keras.losses.categorical_crossentropy,
                       optimizer=keras.optimizers.Adadelta(),
                       metrics=['accuracy'])
```

0.5 Train the model with MNIST dataset

Training in Keras uses function **fit**. Basically, it just provide training and testing set.

```
In [65]: model.fit(imgTrain, onehotTrain, validation_data=(imgTest, onehotTest), batch_size=128

Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 28s 461us/step - loss: 0.2353 - acc: 0.9286 - va
Epoch 2/3
60000/60000 [==============================] - 27s 456us/step - loss: 0.0714 - acc: 0.9783 - va
Epoch 3/3
60000/60000 [==============================] - 27s 457us/step - loss: 0.0538 - acc: 0.9834 - va


Out[65]: <keras.callbacks.History at 0x7f7814939da0>
```

Let see the result of training:

```
In [66]: score = model.evaluate(imgTest, onehotTest, verbose=0)
         print('Test loss     :', score[0])
         print('Test accuracy :', score[1])

Test loss     : 0.04377966066873632
Test accuracy : 0.9863
```

You see, thei model only gets accuracy of 85.8%. It's the time for you to improve it.