Convolutional Neural Networks

And some practical techniques for training them

Spencer Bertsch
ENGG 192
Winter 2019

In this notebook I explore convolutional networks and use a shallow CNN to classify the MNIST dataset with a ~99% accuracy. The MNIST dataset is a popular dataset used in deep learning becasue it's a comparitively easy problem to solve. Larger, richer datasets with more classes, such as the dataset used to train Alex Net in 2012, are much more difficult to learn and require both more convolutional layers, and a much more robust machine to use for training.

Data Source:

I simply called the mnist.load_data() function to load the MNIST dataset in this notebook. Becasue they're used routinely in image recognition scripts, the MNIST and Iris datasets often come with libraries such as Keras. This makes them easy to work with, and easy to load. The next notebook will use a much larger dataset of color images.

Main Sources:

Keras-model-training-history

Keras Team - Github

Imports

[1]
```
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import matplotlib.pyplot as plt
import time
```

```
/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning:
Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64
== np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Load and shape the dataset

[4]
```
#Load the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# input image dimensions
img_rows, img_cols = 28, 28

data1 = x_train #for visualization
data2 = y_train #for visualization

"""
Our images are greyscale, which means they are two dimensional. We have m
Our neural netowrk, specifically the 'conv2d' layer, is expecting a four
value for each pixel in the image. Becasue our images are greyscale, we h
reshape our data to include an additional (1) which will tell our neural
We can achieve this by using the below if-else statement.
"""

if K.image_data_format() == 'channels_first': #<-- checking the Keras bac
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)


print("Shape of original data: ", data1.shape)
print("Shape of reshaped data: ", x_train.shape)
```

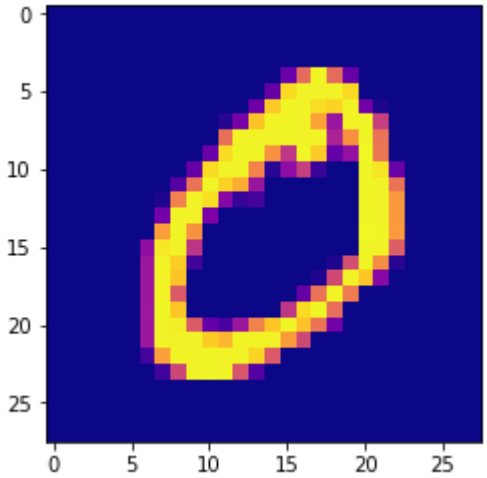
```
Shape of original data:  (60000, 28, 28)
Shape of reshaped data:  (60000, 28, 28, 1)
```

[5]
```
input_shape
```

```
(28, 28, 1)
```

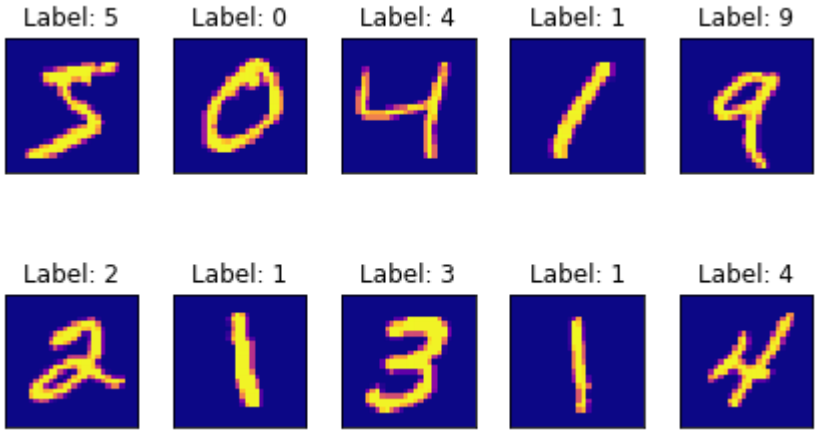
[62]
```
print("LABEL: ", data2[1])
fig = plt.imshow(data1[1, :, :], cmap = plt.cm.plasma);
```

```
LABEL:  0
```



[88]
```
for i in range(10):
    plt.subplot(2,5,1+i)
    plt.title('Label: %d'%data2[i])
    fig = plt.imshow(data1[i, :, :], cmap = plt.cm.plasma);
    fig.axes.get_xaxis().set_visible(False)
    fig.axes.get_yaxis().set_visible(False)

plt.tight_layout()
```



[80]
```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

print('x_train shape:', x_train.shape)
print("We can see that we have",x_train.shape[0], 'train samples')
print("And we have", x_test.shape[0], 'test samples')
print("And each image is", x_test.shape[1], "by", x_test.shape[2] ,'pixel
```

```
x_train shape: (60000, 28, 28, 1)
We can see that we have 60000 train samples
And we have 10000 test samples
And each image is 28 by 28 pixels
Time taken: 0.2001349925994873 seconds
```

Define network parameters

```
[65]  batch_size = 128
      num_classes = 10
      epochs = 12
```

Encode our data

Use the keras.utils to one-hot encode the label vector. Remember that we are simply recognizing handwritten, single digit numbers so we can tell keras.utils that we want an [n x 10] matrix with each row containing only one 1 value and the rest 0 values.

```
[66]  # input image dimensions
      img_rows, img_cols = 28, 28

      # convert class vectors to binary class matrices
      y_train = keras.utils.to_categorical(y_train, num_classes)
      y_test = keras.utils.to_categorical(y_test, num_classes)
```

Define the network

We're now ready to define the network with several convolutional, pooling, and dropout layers, ending with a dense, fully connected layer with a softmax activation function which will yeild our final classification.

```
[67]  #Keras models are generally 'sequential' so we can simply stack layers in
      model = Sequential()

      """
      Our images are greyscale, which means they're 2D, so we can simply use Co
      Convolutions with different sizes - 3x3 or 5x5 - would also work, but as
      will begin to lose information for each pass

      We use RELU as the activation function. Ever since "Alex Net" was publish
      of choice for convolutional layers in image recognition nets.
      """


      model.add(Conv2D(32, kernel_size=(4, 4),
                       activation='relu',
                       input_shape=input_shape))
```

```
#Convolutional layer
model.add(Conv2D(64, (3, 3), activation='relu'))
#Max pooling to reduce number of parameter in the model - could have used
model.add(MaxPooling2D(pool_size=(2, 2)))
#Dropout to prevent overfitting
model.add(Dropout(0.25))

"""
Now that we're done with the convolutional layers and pooling layers to r
and train a dense, fully connected layer on the end of the previous pooli
to prevent overfitting, and finally a last output layer with 10 classes r

We use the softmax activation function so that we can simply take the max
"""

#Flatten our model so we can feed the results into a dense neural network
model.add(Flatten())
#Add dense layer to the flattened output of the last max pooling layer wi
model.add(Dense(128, activation='relu'))
#Last dropout layer to prevent overfitting (p = 0.5)
model.add(Dropout(0.5))
#Final dense layer with softmax activation function to predict which of t
model.add(Dense(num_classes, activation='softmax')) #<-- We have 10 class

"""
Note that there are several hyperparameters noted in this network, and in
external network parameters. No explicit hyperparameter optimization was
such as the p value for the dropout layers, the type of pooling and the p
layers, the activatin functions (RELU is probably the best to use here, b
such as the overall topology of the network itself would lead to improved
"""
```

Compile our model

```
[68] #Here we use categorical_crossentropy as the loss function because we have
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

Fit the model to the data

```
[69]  #Measure the time it takes to train our model
tic = time.time()

history = model.fit(x_train, y_train,
          batch_size=batch_size,
```

```
            epochs=epochs,
            verbose=1,
            validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)

toc = time.time()
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 103s 2ms/step - loss: 0.2566
- acc: 0.9210 - val_loss: 0.0530 - val_acc: 0.9812
Epoch 2/12
60000/60000 [==============================] - 76s 1ms/step - loss: 0.0846
- acc: 0.9746 - val_loss: 0.0403 - val_acc: 0.9866
Epoch 3/12
60000/60000 [==============================] - 377s 6ms/step - loss: 0.0653
- acc: 0.9803 - val_loss: 0.0365 - val_acc: 0.9871
Epoch 4/12
60000/60000 [==============================] - 140s 2ms/step - loss: 0.0515
- acc: 0.9842 - val_loss: 0.0296 - val_acc: 0.9905
Epoch 5/12
60000/60000 [==============================] - 103s 2ms/step - loss: 0.0469
- acc: 0.9862 - val_loss: 0.0320 - val_acc: 0.9883
Epoch 6/12
60000/60000 [==============================] - 242s 4ms/step - loss: 0.0423
- acc: 0.9875 - val_loss: 0.0260 - val_acc: 0.9905
Epoch 7/12
60000/60000 [==============================] - 103s 2ms/step - loss: 0.0381
- acc: 0.9888 - val_loss: 0.0294 - val_acc: 0.9898
Epoch 8/12
60000/60000 [==============================] - 443s 7ms/step - loss: 0.0337
- acc: 0.9892 - val_loss: 0.0267 - val_acc: 0.9899
Epoch 9/12
60000/60000 [==============================] - 228s 4ms/step - loss: 0.0316
- acc: 0.9899 - val_loss: 0.0253 - val_acc: 0.9912
Epoch 10/12
60000/60000 [==============================] - 103s 2ms/step - loss: 0.0288
- acc: 0.9913 - val_loss: 0.0247 - val_acc: 0.9916
Epoch 11/12
60000/60000 [==============================] - 102s 2ms/step - loss: 0.0289
```

[85]
```
# We can use the 'time' package to measure how long it takes to train the
print('Time taken to train on ', epochs, 'epochs is:', (toc - tic), 'seco
```

```
Time taken to train on  12 epochs is: 0.2001349925994873 seconds
```

Model Evaluation

---

[84]
```
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

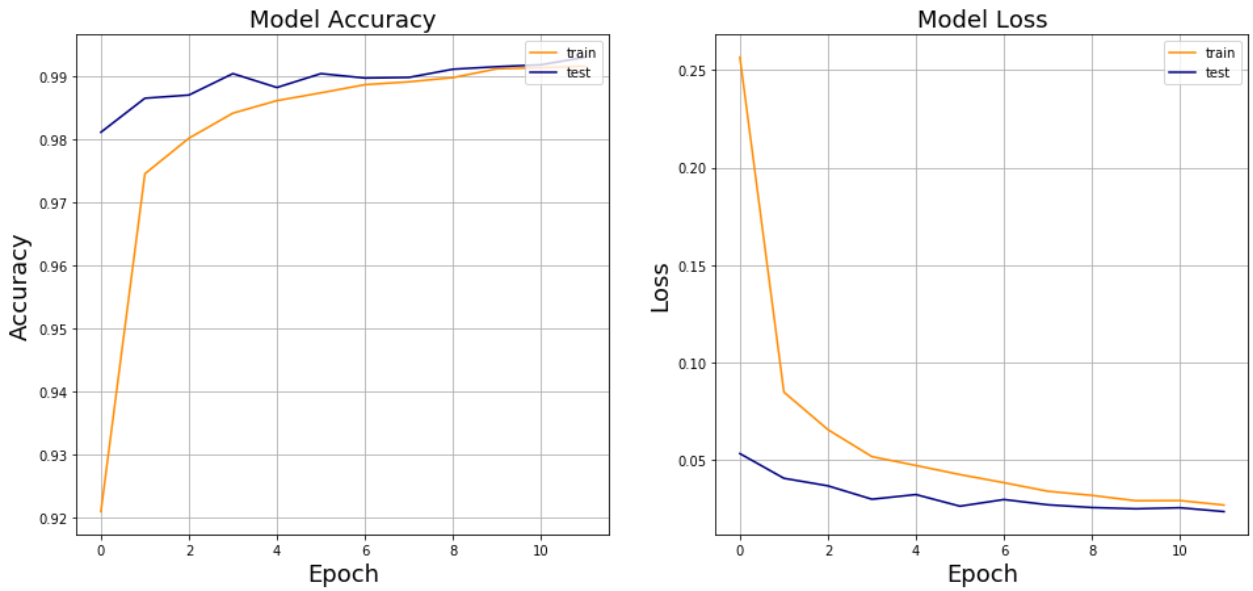
```
Test loss: 0.02323429549929424
Test accuracy: 0.9931
```

[70]
```
#Plot Accuracy
fig = plt.figure(figsize=(16, 7))
ax1 = fig.add_subplot(121)

plt.plot(history.history['acc'], color='darkorange', label = 'train')
plt.plot(history.history['val_acc'], color='navy', label = 'test')
plt.legend(loc='upper right');
plt.xlabel('Epoch', fontsize = 18)
plt.ylabel('Accuracy', fontsize = 18)
plt.title('Model Accuracy', fontsize = 18)
plt.grid()


#Plot Loss
#fig = plt.figure(figsize=(7, 7))
ax1 = fig.add_subplot(122)

plt.plot(history.history['loss'], color='darkorange', label = 'train')
plt.plot(history.history['val_loss'], color='navy', label = 'test')
plt.legend(loc='upper right');
plt.xlabel('Epoch', fontsize = 18)
plt.ylabel('Loss', fontsize = 18)
plt.title('Model Loss', fontsize = 18)
plt.grid()
```



Closing remarks

As we can see above, the accuracy achieved on our test set after 12 epochs is 99.31%. This tells us that our network is doing a good job of learning the different spatial aspects of each of the ten hand written numerals. Subsequently, as expected we can see the training and testing loss

decresing as we march through our 12 epochs, showing that the neurons in the network are needing to update their weights less and less each time.

This dataset is a common one for any introduction to convolutional neural networks prpoject for several reasons - one being that this problem (regognizing hand written digits) is a solved problem. For years now, researchers, professors, and students have been able to achieve very high accuracy on this dataset even when training on a CPU such as the one in my laptop. Another reason this dataset is so popular is that the images are greyscale, not color. That means that each pixel is [1x1] instead of [1x3] for the RGB values in a color pixel. This makes the convolutions less complicated because they can sweep across the image in 2D, and still effectively achieve a very close approximation to f*(x), the target function.

In the next notebook, I will use a dataset of color historology images and a deeper convolutional neural network to classify the images with or without the presence of an IDC, or invasive ductal carcinoma. I will also discuss and explore methods for improving model performance for CNNs when working with limited training data including mirroring, scaling, translation, and illumination.