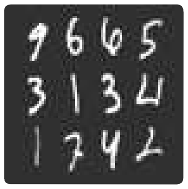
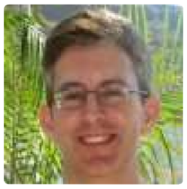




Peter Grenholm

Welcome to deep learning (CNN 99%)

20 voters

last run 2 months ago · Python notebook · 3068 views
using data from Digit Recognizer · Public

Notebook

Convolutional Neural Networks

If you want to apply machine learning to image recognition, convolutional neural networks (CNN) is the way to go. It has been sweeping the board in competitions for the last several years, but perhaps its first big success came in the late 90's when Yann LeCun (http://yann.lecun.com/exdb/lenet/) used it to solve MNIST with 99.5% accuracy. I will show you how it is done in Keras, which is a user-friendly neural network library for python.

Many other notebooks here use a simple fully-connected network (no convolution) to achieve 96-97%, which is a poor result on this dataset. In contrast, what I will show you here is nearly state-of-the-art. In the Kernel (<20 minutes training) we will achieve 99%, but if you train it overnight (or with a GPU) you should reach 99.5. If you then ensemble over several runs, you should get close to the best published accuracy of 99.77% . (Ignore the 100% results on the leaderboard; they were created by learning the test set through repeat submissions)

Here goes:

In [1]:
```
import numpy as np # linear algebra
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
```

If you don't already have Keras (https://keras.io/), you can easily install it through conda or pip. It relies on either tensorflow or theano, so you should have these installed first. Keras is already available here in the kernel and on Amazon deep learning AMI.

In [2]:
```
from keras.utils.np_utils import to_categorical # convert to one-hot-encoding
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D, BatchNormalization
from keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import LearningRateScheduler
```

```
Using TensorFlow backend.
```

In [3]:
```
train_file = "../input/train.csv"
test_file = "../input/test.csv"
```

```
output_file = "submission.csv"
```

Load the data

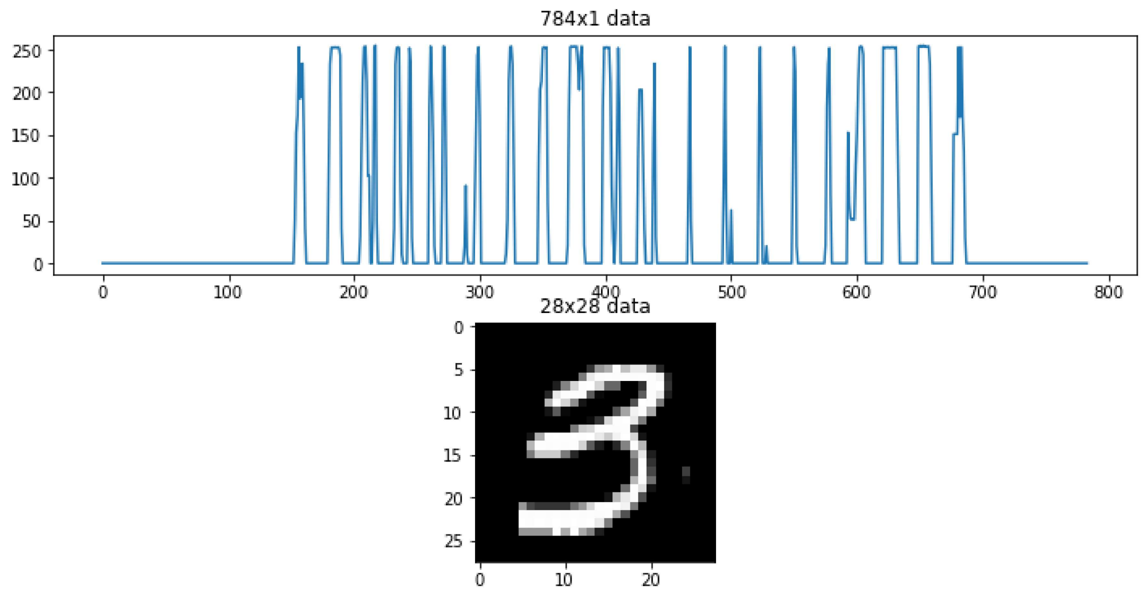
As always, we split the data into a training set and a validation set, so that we can evaluate the performance of our model.

```
In [4]: raw_data = np.loadtxt(train_file, skiprows=1, dtype='int', delimiter=',')
        x_train, x_val, y_train, y_val = train_test_split(
            raw_data[:,1:], raw_data[:,0], test_size=0.1)
```

Each data point consists of 784 values. A fully connected net just treats all these values the same, but a CNN treats it as a 28x28 square. Thes two graphs explain the difference: It's easy to understand why a CNN can get better results.

```
In [5]: fig, ax = plt.subplots(2, 1, figsize=(12,6))
        ax[0].plot(x_train[0])
        ax[0].set_title('784x1 data')
        ax[1].imshow(x_train[0].reshape(28,28), cmap='gray')
        ax[1].set_title('28x28 data')

Out[5]: <matplotlib.text.Text at 0x7f5739ee6978>
```



We now reshape all data this way. Keras wants an extra dimension in the end, for channels. If this had been RGB images, there would have been 3 channels, but as MNIST is gray scale it only uses one.

This notebook is written for the tensorflow channel ordering. If you have Keras installed for Theano backend, you might start seeing some error message soon related to channel ordering. This can easily be solved (https://keras.io/backend/#set_image_dim_ordering).

```
In [6]: x_train = x_train.reshape(-1, 28, 28, 1)
        x_val = x_val.reshape(-1, 28, 28, 1)
```

It would be possible to train the net on the original data, with pixel values 0 to 255. If we use the standard initialization methods for weights, however, data between 0 and 1 should make the net converge faster.

In [7]:
```
x_train = x_train.astype("float32")/255.
x_val = x_val.astype("float32")/255.
```

The labels were given as integers between 0 and 9. We need to convert these to one-hot encoding, i.e. a 10x1 array with one 1 and nine 0:s, with the position of the 1 showing us the value. See the example, with the position of the 1 showing the correct value for the digit in the graph above.

In [8]:
```
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
#example:
print(y_train[0])
```

```
[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
```

Train the model

Keras offers two different ways of defining a network. We will the Sequential API, where you just add on one layer at a time, starting from the input.

The most important part are the convolutional layers Conv2D. Here they have 16-32 filters that use nine weights each to transform a pixel to a weighted average of itself and its eight neighbors. As the same nine weights are used over the whole image, the net will pick up features that are useful everywhere. As it is only nine weights, we can stack many convolutional layers on top of each other without running out of memory/time.

The MaxPooling layers just look at four neighboring pixels and picks the maximal value. This reduces the size of the image by half, and by combining convolutional and pooling layers, the net be able to combine its features to learn more global features of the image. In the end we use the features in two fully-connected (Dense) layers.

Batch Normalization is a technical trick to make training faster. Dropout is a regularization method, where the layer randomly replaces a proportion of its weights to zero for each training sample. This forces the net to learn features in a distributed way, not relying to much on a particular weight, and therefore improves generalization. 'relu' is the activation function x -> max(x,0).

In [9]:
```
model = Sequential()

model.add(Conv2D(filters = 16, kernel_size = (3, 3), activation='relu',
                 input_shape = (28, 28, 1)))
model.add(BatchNormalization())
model.add(Conv2D(filters = 16, kernel_size = (3, 3), activation='relu'))
model.add(BatchNormalization())
#model.add(Conv2D(filters = 16, kernel_size = (3, 3), activation='relu'))
#model.add(BatchNormalization())
model.add(MaxPool2D(strides=(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters = 32, kernel_size = (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(filters = 32, kernel_size = (3, 3), activation='relu'))
model.add(BatchNormalization())
#model.add(Conv2D(filters = 32, kernel_size = (3, 3), activation='relu'))
#model.add(BatchNormalization())
model.add(MaxPool2D(strides=(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

Another important method to improve generalization is augmentation. This means generating more training data by randomly perturbing the images. If done in the right way, it can force the net to only learn translation-invariant features. If you train this model over hundreds of epochs, augmentation will definitely improve your performance. Here in the Kernel, we will only look at each image 4-5 times, so the difference is smaller. We use a Keras function for augmentation.

In [10]:
```
datagen = ImageDataGenerator(zoom_range = 0.1,
                            height_shift_range = 0.1,
                            width_shift_range = 0.1,
                            rotation_range = 10)
```

The model needs to be compiled before training can start. As our loss function, we use logloss which is called "categorical_crossentropy" in Keras. Metrics is only used for evaluation. As optimizer, we could have used ordinary stochastic gradient descent (SGD), but Adam is faster.

In [11]:
```
model.compile(loss='categorical_crossentropy', optimizer = Adam(lr=1e-4), metrics=["acc
uracy"])
```

We train once with a smaller learning rate to ensure convergence. We then speed things up, only to reduce the learning rate by 10% every epoch. Keras has a function for this:

In [12]:
```
annealer = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

We will use a very small validation set during training to save time in the kernel.

In [13]:
```
hist = model.fit_generator(datagen.flow(x_train, y_train, batch_size=16),
                           steps_per_epoch=500,
                           epochs=20, #Increase this when not on Kaggle kernel
                           verbose=2,  #1 for ETA, 0 for silent
                           validation_data=(x_val[:400,:], y_val[:400,:]), #For speed
                           callbacks=[annealer])
```

```
Epoch 1/20
55s - loss: 0.8606 - acc: 0.7291 - val_loss: 0.9183 - val_acc: 0.7725
Epoch 2/20
50s - loss: 0.3651 - acc: 0.8872 - val_loss: 0.1092 - val_acc: 0.9625
Epoch 3/20
50s - loss: 0.2826 - acc: 0.9120 - val_loss: 0.1099 - val_acc: 0.9675
Epoch 4/20
48s - loss: 0.2506 - acc: 0.9215 - val_loss: 0.0915 - val_acc: 0.9625
Epoch 5/20
48s - loss: 0.2031 - acc: 0.9399 - val_loss: 0.0548 - val_acc: 0.9825
Epoch 6/20
48s - loss: 0.1780 - acc: 0.9452 - val_loss: 0.0376 - val_acc: 0.9850
Epoch 7/20
49s - loss: 0.1781 - acc: 0.9466 - val_loss: 0.0528 - val_acc: 0.9825
Epoch 8/20
50s - loss: 0.1596 - acc: 0.9540 - val_loss: 0.0242 - val_acc: 0.9950
Epoch 9/20
50s - loss: 0.1308 - acc: 0.9599 - val_loss: 0.0309 - val_acc: 0.9925
Epoch 10/20
51s - loss: 0.1426 - acc: 0.9601 - val_loss: 0.0287 - val_acc: 0.9875
Epoch 11/20
51s - loss: 0.1343 - acc: 0.9622 - val_loss: 0.0227 - val_acc: 0.9900
Epoch 12/20
51s - loss: 0.1190 - acc: 0.9642 - val_loss: 0.0422 - val_acc: 0.9850
Epoch 13/20
52s - loss: 0.1129 - acc: 0.9656 - val_loss: 0.0212 - val_acc: 0.9900
Epoch 14/20
```

```
52s - loss: 0.1022 - acc: 0.9691 - val_loss: 0.0206 - val_acc: 0.9925
Epoch 15/20
52s - loss: 0.1029 - acc: 0.9710 - val_loss: 0.0194 - val_acc: 0.9925
Epoch 16/20
51s - loss: 0.1055 - acc: 0.9714 - val_loss: 0.0157 - val_acc: 0.9950
Epoch 17/20
52s - loss: 0.0941 - acc: 0.9712 - val_loss: 0.0126 - val_acc: 0.9975
Epoch 18/20
53s - loss: 0.0893 - acc: 0.9722 - val_loss: 0.0112 - val_acc: 0.9950
Epoch 19/20
51s - loss: 0.0885 - acc: 0.9720 - val_loss: 0.0091 - val_acc: 1.0000
Epoch 20/20
51s - loss: 0.0845 - acc: 0.9750 - val_loss: 0.0108 - val_acc: 0.9975
```

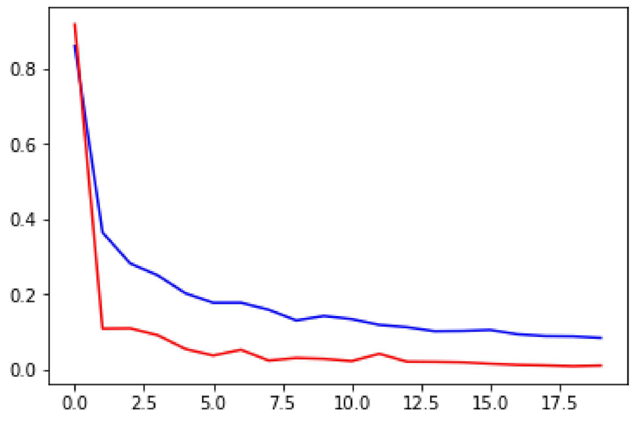
Evaluate

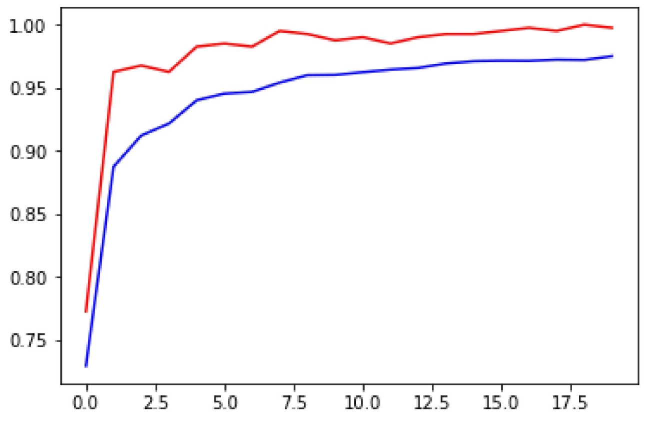
We only used a subset of the validation set during training, to save time. Now let's check performance on the whole validation set.

In [14]:
```
final_loss, final_acc = model.evaluate(x_val, y_val, verbose=0)
print("Final loss: {0:.4f}, final accuracy: {1:.4f}".format(final_loss, final_acc))
```

```
Final loss: 0.0361, final accuracy: 0.9917
```

In [15]:
```
plt.plot(hist.history['loss'], color='b')
plt.plot(hist.history['val_loss'], color='r')
plt.show()
plt.plot(hist.history['acc'], color='b')
plt.plot(hist.history['val_acc'], color='r')
plt.show()
```





In [16]:
```
y_hat = model.predict(x_val)
y_pred = np.argmax(y_hat, axis=1)
y_true = np.argmax(y_val, axis=1)
cm = confusion_matrix(y_true, y_pred)
print(cm)
```

```
[[418   0   0   0   0   0   0   0   0   1]
 [  0 438   0   0   0   0   0   0   0   0]
 [  0   1 422   0   0   0   0   2   2   0]
 [  0   0   0 424   0   2   0   0   3   1]
 [  0   0   0   0 402   0   1   0   1   1]
 [  0   0   0   0   0 365   0   0   0   1]
 [  2   0   1   0   0   0 411   0   1   0]
 [  0   0   3   0   0   0   0 447   0   0]
 [  0   2   1   0   0   0   1   0 455   0]
 [  0   1   0   1   0   0   0   2   4 383]]
```

Not too bad, considering the minimal amount of training so far. In fact, we have only gone through the training data approximately five times. With proper training we should get really good results.

As you can see there are quite a few parameters that could be tweaked (number of layers, number of filters, Dropout parameters, learning rate, augmentation settings). This is often done with trial and error, and there is no easy shortcut.

Getting convergence should not be a problem, unless you use an extremely large learning rate. It's easy, however, to create a net that overfits, with perfect results on the training set and very poor results on the validation data. If this happens, you could try increasing the Dropout parameters, increase augmentation, or perhaps stop training earlier. If you instead wants to increase accuracy, try adding on two more layers, or increase the number of filters.

Submit

To easily get to the top half of the leaderboard, just follow these steps, go to the Kernel's output, and submit "submission.csv"

In [17]:
```
mnist_testset = np.loadtxt(test_file, skiprows=1, dtype='int', delimiter=',')
x_test = mnist_testset.astype("float32")
x_test = x_test.reshape(-1, 28, 28, 1)/255.
```

In [18]:
```
y_hat = model.predict(x_test, batch_size=64)
```

y_hat consists of class probabilities (corresponding to the one-hot encoding of the training labels). I now select the class with highest probability

In [19]:
```
y_pred = np.argmax(y_hat,axis=1)
```

In [20]:
```
with open(output_file, 'w') as f :
    f.write('ImageId,Label\n')
```

Comments (9) All Comments Sort by Hotness

Click here to enter a comment...

Aditya Arora • (243rd in this Competition) • Posted on Latest Version • a month ago • Options • Reply 1

Thanks for sharing this wonderful notebook. What is steps_per_epoch? and How do you come up with this CNN architecture? How do you decide the architecture for a problem?

Peter Grenh... • (418th in this Competition) • Posted on Latest Version • a month ago • Options • Reply 1

Thanks! The parameter steps_per_epoch determines how many times the generator datagen.flow is called in one epoch. This means that the total number of training samples in an epoch is steps_per_epoch*batch_size.

Aditya Arora • (243rd in this Competition) • Posted on Latest Version • a month ago • Options • Reply 0

Thank you, Peter. If I have another Image classification problem, How do I decide the architecture of my CNN(number of layers, optimizer, dropout and all other parameters)?

SevenSteven • (423rd in this Competition) • Posted on Latest Version • a month ago • Options • Reply 1

awesome!

joker1999 • Posted on Version 23 • 3 months ago • Options • Reply 1

Your notebook shows 96% accuracy. How's that 99%?

Peter Grenh... • (418th in this Competition) • Posted on Version 24 • 3 months ago • Options • Reply 1

Hi @joker1999. You need to train it longer to get above 99% - just change 'epochs' or 'steps_per_epoch' before you run it. The version you saw only trained for 2 minutes.

I'll run it again with changed parameters and see how high it can get within the kernel resources. Cheers, p

HighSpirits • (440th in this Competition) • Posted on Latest Version • 19 hours ago • Options • Edit • Reply 0

Great work..thanks for sharing...learnt a lot from it!!

AndreaBattiste.. • Posted on Version 24 • 3 months ago • Options • Reply 0

Hi! Thank you for sharing this notebook, it is very helpful for me as a newbie in data science...

How did you come up with that particular neural network structure? Are there some rules or observations that can help in building such a neural network?

Peter Grenh... • (418th in this Competition) • Posted on Version 24 • 3 months ago • Options • Reply 1

Hi and thanks for the feedback!

Many CNN's have a very similar structure, with a gradual decrease in image size, and increase in the number of filters. See for instance VGG16 which can be created through keras.applications.vgg16, or read https://arxiv.org/abs/1512.00567 by the creators of GoogleNet for a modern discussion. In the end it often becomes a trade-off between computation time and accuracy, where a deeper net with more filters might be more accurate but to slow to train.

The net here is by no means perfect. For instance, you could try increasing the number of filters, and add some BatchNormalization layers to speed up training. I have also not attempted to tune the Dropout parameters.