ML101-MNIST

October 14, 2016

1 More realistic handwritten digits - MNIST dataset

The *Hello world* of Machine Learning

- more complex dataset than before
- but still very far from the current state-of-art challanges

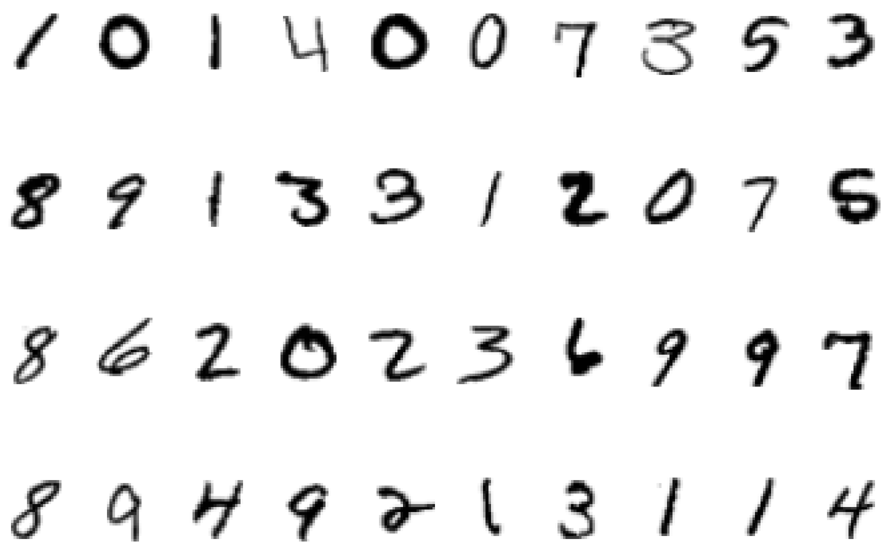
```
In [1]: %matplotlib inline
        from numpy import *
        import matplotlib.pyplot as plt
```

```
In [2]: X=load('train.npz')['arr_0']
        y=load('trainlabels.npz')['arr_0']
        Xt=load('test.npz')['arr_0']
        yt=load('testlabels.npz')['arr_0']
```

```
In [3]: print X.shape, amin(X), amax(X)
        print Xt.shape, amin(Xt), amax(Xt)
```

```
(33600, 784) 0.0 1.0
(8400, 784) 0.0 1.0
```

```
In [4]: for i in range(40):
            plt.subplot(4,10,i+1)
            plt.axis('off')
            plt.imshow(X[i].reshape((28,28)),cmap=plt.cm.gray_r)
        plt.show()
```



```
In [5]: print y[:40]
```

```
[ 1.  0.  1.  4.  0.  0.  7.  3.  5.  3.  8.  9.  1.  3.  3.  1.  2.  0.
  7.  5.  8.  6.  2.  0.  2.  3.  6.  9.  9.  7.  8.  9.  4.  9.  2.  1.
  3.  1.  1.  4.]
```

1.1 Logistic Regression

```
In [8]: from sklearn.linear_model import *
        from sklearn.metrics import *
```

```
In [21]: clf=LogisticRegression()
         %time clf.fit(X,y)
```

```
CPU times: user 34.3 s, sys: 40 ms, total: 34.3 s
Wall time: 34.4 s
```

```
Out[21]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

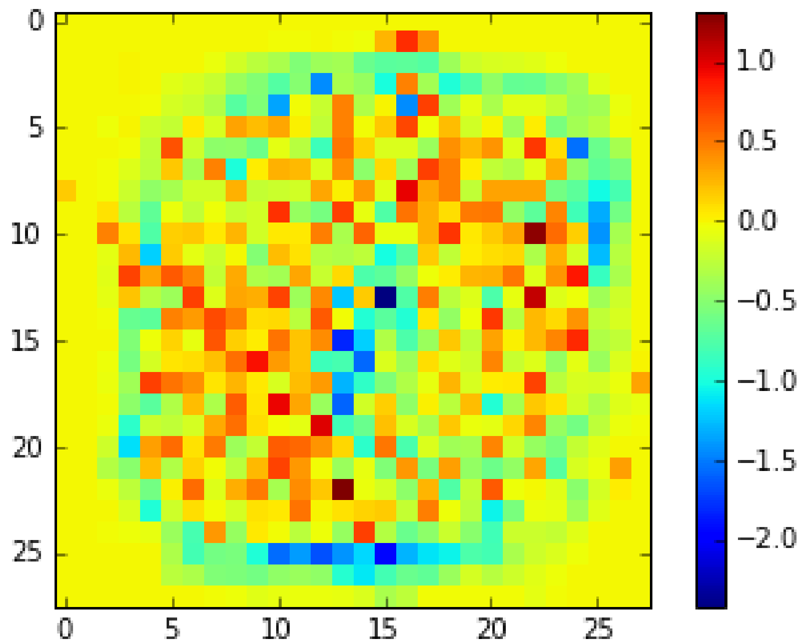
```
In [22]: preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
0.917857142857
```

```
In [23]: print preds[:10], yt[:10]
```

```
[ 0.  7.  7.  2.  2.  6.  5.  7.  5.  5.] [ 0.  7.  7.  2.  2.  6.  5.  7.  8.  5.]
```

```
In [24]: cfs0=clf.coef_[0].reshape((28,28))
         plt.imshow(cfs0,interpolation="nearest")
         plt.colorbar()
         plt.show()
```




```
In [10]: clf=LogisticRegressionCV()
         %time clf.fit(X,y)
         preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
CPU times: user 22min 21s, sys: 7.88 s, total: 22min 29s
Wall time: 3min 44s
0.919285714286
```

```
In [11]: clf.C_
```

```
Out[11]: array([ 0.35938137,  0.35938137,  0.35938137,  0.35938137,  0.35938137,
                 0.35938137,  0.04641589,  0.35938137,  0.04641589,  0.35938137])
```

1.2 Support Vector Machine

```
In [12]: from sklearn.svm import SVC
         clf = SVC()
         %time clf.fit(X,y)
         preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
CPU times: user 3min 23s, sys: 160 ms, total: 3min 23s
Wall time: 3min 23s
0.937142857143
```

```
In [13]: clf = SVC(C=100)
         %time clf.fit(X,y)
         preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
CPU times: user 1min 28s, sys: 112 ms, total: 1min 28s
Wall time: 1min 28s
0.96619047619
```

1.3 Random Forest

```
In [14]: from sklearn.ensemble import RandomForestClassifier as RFC
```

```
In [15]: clf=RFC(n_estimators=100)
         %time clf.fit(X,y)
         preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
CPU times: user 15 s, sys: 0 ns, total: 15 s
Wall time: 15 s
0.965833333333
```

```
In [24]: clf=RFC(n_estimators=500)
         %time clf.fit(X,y)
         preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
CPU times: user 1min 14s, sys: 80 ms, total: 1min 14s
Wall time: 1min 14s
0.9675
```

1.4 Gradient Boosted Trees

The sklearn GradientBoostingClassifier takes too long... Use `xgboost`

```
In [16]: from xgboost import XGBClassifier
```

```
In [17]: clf=XGBClassifier(max_depth=5, n_estimators=100)
         %time clf.fit(X,y)
         preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
CPU times: user 27min 3s, sys: 5.32 s, total: 27min 8s
Wall time: 2min 16s
0.96119047619
```

1.5 Nearest neighbors

This **should not** be used for such high dimensional data (dim=**784**) but let's try nevertheless...

```
In [18]: from sklearn.neighbors import KNeighborsClassifier
```

```
In [19]: clf=KNeighborsClassifier()
         %time clf.fit(X,y)
         %time preds=clf.predict(Xt)
         print accuracy_score(yt,preds)
```

```
CPU times: user 5.17 s, sys: 160 ms, total: 5.33 s
Wall time: 5.33 s
CPU times: user 5min 24s, sys: 112 ms, total: 5min 24s
Wall time: 5min 24s
0.969047619048
```

Use PCA (*Principal Components Analysis*) to reduce dimensionality and only then apply Nearest neighbors

```
In [20]: from sklearn.decomposition import RandomizedPCA
```

```
In [21]: pca=RandomizedPCA(n_components=20)
```

```
In [22]: %time Xpca=pca.fit_transform(X)
         Xpcat=pca.transform(Xt)
```

```
CPU times: user 13.9 s, sys: 3.99 s, total: 17.9 s
Wall time: 4.26 s
```

```
In [23]: clf=KNeighborsClassifier()
         %time clf.fit(Xpca,y)
         %time preds=clf.predict(Xpcat)
         print accuracy_score(yt,preds)
```

```
CPU times: user 72 ms, sys: 8 ms, total: 80 ms
Wall time: 79.3 ms
CPU times: user 3.02 s, sys: 0 ns, total: 3.02 s
Wall time: 3.02 s
0.968571428571
```

1.6 Neural Networks

```
In [1]: from keras.models import Sequential
        from keras.layers.core import Dense, Dropout, Activation
        from keras.optimizers import SGD, Adam, RMSprop
        from keras.layers.advanced_activations import *
```

```
Using Theano backend.
```

Try the previous one...

```
In [4]: model = Sequential()
        model.add(Dense(64, input_dim=X.shape[1], activation='relu'))
        model.add(Dense(64, activation='relu'))
        model.add(Dense(10))
        model.add(Activation('softmax'))
```

```
In [5]: model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])
```

```
In [6]: %time model.fit(X, y, nb_epoch=30, batch_size=64, validation_data=(Xt,yt))
```

```
Train on 33600 samples, validate on 8400 samples
Epoch 1/30
33600/33600 [==============================] - 1s - loss: 0.4059 - acc: 0.8846 - val_loss: 0.2006 - val_
Epoch 2/30
33600/33600 [==============================] - 1s - loss: 0.1761 - acc: 0.9478 - val_loss: 0.1517 - val_
Epoch 3/30
33600/33600 [==============================] - 1s - loss: 0.1280 - acc: 0.9622 - val_loss: 0.1375 - val_
Epoch 4/30
33600/33600 [==============================] - 1s - loss: 0.1010 - acc: 0.9698 - val_loss: 0.1273 - val_
Epoch 5/30
```

```
33600/33600 [==============================] - 1s - loss: 0.0813 - acc: 0.9758 - val_loss: 0.1200 - val_
Epoch 6/30
33600/33600 [==============================] - 1s - loss: 0.0688 - acc: 0.9795 - val_loss: 0.1099 - val_
Epoch 7/30
33600/33600 [==============================] - 1s - loss: 0.0577 - acc: 0.9824 - val_loss: 0.1133 - val_
Epoch 8/30
33600/33600 [==============================] - 1s - loss: 0.0481 - acc: 0.9851 - val_loss: 0.1127 - val_
Epoch 9/30
33600/33600 [==============================] - 1s - loss: 0.0397 - acc: 0.9879 - val_loss: 0.1155 - val_
Epoch 10/30
33600/33600 [==============================] - 1s - loss: 0.0340 - acc: 0.9898 - val_loss: 0.1074 - val_
Epoch 11/30
33600/33600 [==============================] - 1s - loss: 0.0287 - acc: 0.9914 - val_loss: 0.1170 - val_
Epoch 12/30
33600/33600 [==============================] - 1s - loss: 0.0243 - acc: 0.9929 - val_loss: 0.1140 - val_
Epoch 13/30
33600/33600 [==============================] - 1s - loss: 0.0218 - acc: 0.9935 - val_loss: 0.1138 - val_
Epoch 14/30
33600/33600 [==============================] - 1s - loss: 0.0194 - acc: 0.9941 - val_loss: 0.1241 - val_
Epoch 15/30
33600/33600 [==============================] - 1s - loss: 0.0174 - acc: 0.9949 - val_loss: 0.1145 - val_
Epoch 16/30
33600/33600 [==============================] - 1s - loss: 0.0136 - acc: 0.9961 - val_loss: 0.1337 - val_
Epoch 17/30
33600/33600 [==============================] - 1s - loss: 0.0150 - acc: 0.9954 - val_loss: 0.1288 - val_
Epoch 18/30
33600/33600 [==============================] - 1s - loss: 0.0111 - acc: 0.9970 - val_loss: 0.1260 - val_
Epoch 19/30
33600/33600 [==============================] - 1s - loss: 0.0117 - acc: 0.9964 - val_loss: 0.1308 - val_
Epoch 20/30
33600/33600 [==============================] - 1s - loss: 0.0108 - acc: 0.9970 - val_loss: 0.1313 - val_
Epoch 21/30
33600/33600 [==============================] - 1s - loss: 0.0117 - acc: 0.9961 - val_loss: 0.1542 - val_
Epoch 22/30
33600/33600 [==============================] - 1s - loss: 0.0086 - acc: 0.9972 - val_loss: 0.1245 - val_
Epoch 23/30
33600/33600 [==============================] - 1s - loss: 0.0072 - acc: 0.9980 - val_loss: 0.1330 - val_
Epoch 24/30
33600/33600 [==============================] - 2s - loss: 0.0044 - acc: 0.9987 - val_loss: 0.1627 - val_
Epoch 25/30
33600/33600 [==============================] - 1s - loss: 0.0169 - acc: 0.9942 - val_loss: 0.1444 - val_
Epoch 26/30
33600/33600 [==============================] - 2s - loss: 0.0077 - acc: 0.9979 - val_loss: 0.1432 - val_
Epoch 27/30
33600/33600 [==============================] - 1s - loss: 0.0060 - acc: 0.9983 - val_loss: 0.1520 - val_
Epoch 28/30
33600/33600 [==============================] - 1s - loss: 0.0112 - acc: 0.9966 - val_loss: 0.1544 - val_
Epoch 29/30
33600/33600 [==============================] - 2s - loss: 0.0050 - acc: 0.9987 - val_loss: 0.1586 - val_
Epoch 30/30
33600/33600 [==============================] - 2s - loss: 0.0031 - acc: 0.9994 - val_loss: 0.1544 - val_
CPU times: user 1min 42s, sys: 3.09 s, total: 1min 45s
Wall time: 53.4 s
```

Out[6]: <keras.callbacks.History at 0x7fe38767cf50>

In [9]:
```
preds=model.predict_classes(Xt)
print
print accuracy_score(yt,preds)
```

```
8400/8400 [==============================] - 0s

0.971904761905
```

Try a bigger one with Dropout layers...

In [10]:
```
model = Sequential()
model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))
```

In [11]:
```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [12]:
```
%time model.fit(X, y, nb_epoch=20, batch_size=128, validation_data=(Xt,yt))
```

```
Train on 33600 samples, validate on 8400 samples
Epoch 1/20
33600/33600 [==============================] - 5s - loss: 0.3254 - acc: 0.9013 - val_loss: 0.1364 - val_
Epoch 2/20
33600/33600 [==============================] - 5s - loss: 0.1246 - acc: 0.9616 - val_loss: 0.1031 - val_
Epoch 3/20
33600/33600 [==============================] - 5s - loss: 0.0873 - acc: 0.9715 - val_loss: 0.0957 - val_
Epoch 4/20
33600/33600 [==============================] - 8s - loss: 0.0663 - acc: 0.9787 - val_loss: 0.0943 - val_
Epoch 5/20
33600/33600 [==============================] - 8s - loss: 0.0518 - acc: 0.9832 - val_loss: 0.0788 - val_
Epoch 6/20
33600/33600 [==============================] - 8s - loss: 0.0412 - acc: 0.9861 - val_loss: 0.0917 - val_
Epoch 7/20
33600/33600 [==============================] - 8s - loss: 0.0350 - acc: 0.9884 - val_loss: 0.0863 - val_
Epoch 8/20
33600/33600 [==============================] - 7s - loss: 0.0329 - acc: 0.9892 - val_loss: 0.0817 - val_
Epoch 9/20
33600/33600 [==============================] - 7s - loss: 0.0268 - acc: 0.9909 - val_loss: 0.0962 - val_
Epoch 10/20
33600/33600 [==============================] - 8s - loss: 0.0253 - acc: 0.9914 - val_loss: 0.0920 - val_
Epoch 11/20
33600/33600 [==============================] - 7s - loss: 0.0239 - acc: 0.9922 - val_loss: 0.0826 - val_
Epoch 12/20
33600/33600 [==============================] - 8s - loss: 0.0188 - acc: 0.9932 - val_loss: 0.0990 - val_
Epoch 13/20
33600/33600 [==============================] - 8s - loss: 0.0200 - acc: 0.9931 - val_loss: 0.0925 - val_
Epoch 14/20
```

```
33600/33600 [==============================] - 8s - loss: 0.0197 - acc: 0.9934 - val_loss: 0.1053 - val_
Epoch 15/20
33600/33600 [==============================] - 8s - loss: 0.0183 - acc: 0.9940 - val_loss: 0.0962 - val_
Epoch 16/20
33600/33600 [==============================] - 8s - loss: 0.0178 - acc: 0.9943 - val_loss: 0.1169 - val_
Epoch 17/20
33600/33600 [==============================] - 8s - loss: 0.0191 - acc: 0.9933 - val_loss: 0.1121 - val_
Epoch 18/20
33600/33600 [==============================] - 7s - loss: 0.0150 - acc: 0.9949 - val_loss: 0.1065 - val_
Epoch 19/20
33600/33600 [==============================] - 8s - loss: 0.0137 - acc: 0.9955 - val_loss: 0.0965 - val_
Epoch 20/20
33600/33600 [==============================] - 7s - loss: 0.0152 - acc: 0.9950 - val_loss: 0.1156 - val_
CPU times: user 5min 6s, sys: 3.5 s, total: 5min 9s
Wall time: 2min 35s
```

```
Out[12]: <keras.callbacks.History at 0x7fe37c38e110>
```

```
In [13]: preds=model.predict_classes(Xt)
         print
         print accuracy_score(yt,preds)
```

```
8400/8400 [==============================] - 0s

0.978571428571
```

1.7 Convolutional Neural Network

```
In [2]: from keras.layers import Convolution2D, MaxPooling2D, Flatten

        model = Sequential()

        model.add(Convolution2D(32, 3, 3,
                                border_mode='valid',
                                input_shape=(1, 28, 28)))
        model.add(Activation('relu'))
        model.add(Convolution2D(32, 3, 3))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        model.add(Flatten())
        model.add(Dense(128))
        model.add(Activation('relu'))
        model.add(Dropout(0.5))
        model.add(Dense(10))
        model.add(Activation('softmax'))
```

```
In [3]: model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])
```

```
In [4]: print model.summary()
```

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
```

```
====================================================================================================
convolution2d_1 (Convolution2D)    (None, 32, 26, 26)  320         convolution2d_input_1[0][0]
____________________________________________________________________________________________________
activation_1 (Activation)          (None, 32, 26, 26)  0           convolution2d_1[0][0]
____________________________________________________________________________________________________
convolution2d_2 (Convolution2D)    (None, 32, 24, 24)  9248        activation_1[0][0]
____________________________________________________________________________________________________
activation_2 (Activation)          (None, 32, 24, 24)  0           convolution2d_2[0][0]
____________________________________________________________________________________________________
maxpooling2d_1 (MaxPooling2D)      (None, 32, 12, 12)  0           activation_2[0][0]
____________________________________________________________________________________________________
dropout_1 (Dropout)                (None, 32, 12, 12)  0           maxpooling2d_1[0][0]
____________________________________________________________________________________________________
flatten_1 (Flatten)                (None, 4608)        0           dropout_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                    (None, 128)         589952      flatten_1[0][0]
____________________________________________________________________________________________________
activation_3 (Activation)          (None, 128)         0           dense_1[0][0]
____________________________________________________________________________________________________
dropout_2 (Dropout)                (None, 128)         0           activation_3[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                    (None, 10)          1290        dropout_2[0][0]
____________________________________________________________________________________________________
activation_4 (Activation)          (None, 10)          0           dense_2[0][0]
====================================================================================================
Total params: 600810
____________________________________________________________________________________________________
None
```

We need to reshape the input data back into 28x28 images with a single channel (black/white)

In [16]:
```
X2=X.reshape((len(X),1,28,28))
X2t=Xt.reshape((len(Xt),1,28,28))
```

In [17]:
```
%time model.fit(X2, y, nb_epoch=12, batch_size=128, validation_data=(X2t,yt))
```

```
Train on 33600 samples, validate on 8400 samples
Epoch 1/12
33600/33600 [==============================] - 75s - loss: 0.3215 - acc: 0.8999 - val_loss: 0.0711 - val
Epoch 2/12
33600/33600 [==============================] - 89s - loss: 0.1099 - acc: 0.9669 - val_loss: 0.0549 - val
Epoch 3/12
33600/33600 [==============================] - 75s - loss: 0.0821 - acc: 0.9760 - val_loss: 0.0430 - val
Epoch 4/12
33600/33600 [==============================] - 72s - loss: 0.0636 - acc: 0.9806 - val_loss: 0.0378 - val
Epoch 5/12
33600/33600 [==============================] - 75s - loss: 0.0541 - acc: 0.9830 - val_loss: 0.0384 - val
Epoch 6/12
33600/33600 [==============================] - 72s - loss: 0.0468 - acc: 0.9850 - val_loss: 0.0385 - val
Epoch 7/12
33600/33600 [==============================] - 70s - loss: 0.0441 - acc: 0.9860 - val_loss: 0.0374 - val
Epoch 8/12
33600/33600 [==============================] - 71s - loss: 0.0405 - acc: 0.9869 - val_loss: 0.0379 - val
Epoch 9/12
33600/33600 [==============================] - 68s - loss: 0.0346 - acc: 0.9888 - val_loss: 0.0367 - val
```

```
Epoch 10/12
33600/33600 [==============================] - 68s - loss: 0.0305 - acc: 0.9906 - val_loss: 0.0340 - val
Epoch 11/12
33600/33600 [==============================] - 68s - loss: 0.0293 - acc: 0.9903 - val_loss: 0.0342 - val
Epoch 12/12
33600/33600 [==============================] - 67s - loss: 0.0253 - acc: 0.9916 - val_loss: 0.0337 - val
CPU times: user 28min 19s, sys: 53.6 s, total: 29min 12s
Wall time: 14min 38s
```

```
Out[17]: <keras.callbacks.History at 0x7fe37442bdd0>
```

```
In [20]: preds=model.predict_classes(X2t)
         print
         print accuracy_score(yt,preds)
```

```
8400/8400 [==============================] - 5s

0.990119047619
```

On a modern GPU, training takes only **42 seconds** (with NVIDIA cuDNN library installed)…