Keras_mnist_analysis

August 22, 2018

0.1 Keras example: mnist analysis by simple NN

```
In [15]: %%time
         from keras.datasets import mnist
         (X_train0, y_train0), (X_test0, y_test0) = mnist.load_data()
```

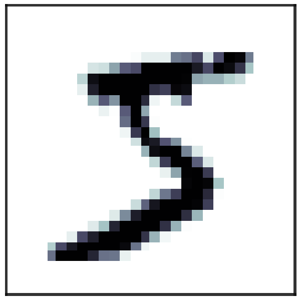
```
CPU times: user 162 ms, sys: 18.5 ms, total: 181 ms
Wall time: 180 ms
```

```
In [16]: print(X_train0.shape, X_train0.dtype)
         print(y_train0.shape, y_train0.dtype)
         print(X_test0.shape, X_test0.dtype)
         print(y_test0.shape, y_test0.dtype)
```

```
(60000, 28, 28) uint8
(60000,) uint8
(10000, 28, 28) uint8
(10000,) uint8
```

```
In [17]: import matplotlib.pyplot as plt
         import matplotlib as mpl
         %matplotlib inline
```

```
In [18]: plt.figure(figsize=(2, 2))
         plt.imshow(X_train0[0], cmap=mpl.cm.bone_r)
         plt.grid(False)
         plt.xticks([])
         plt.yticks([])
         plt.show()
```

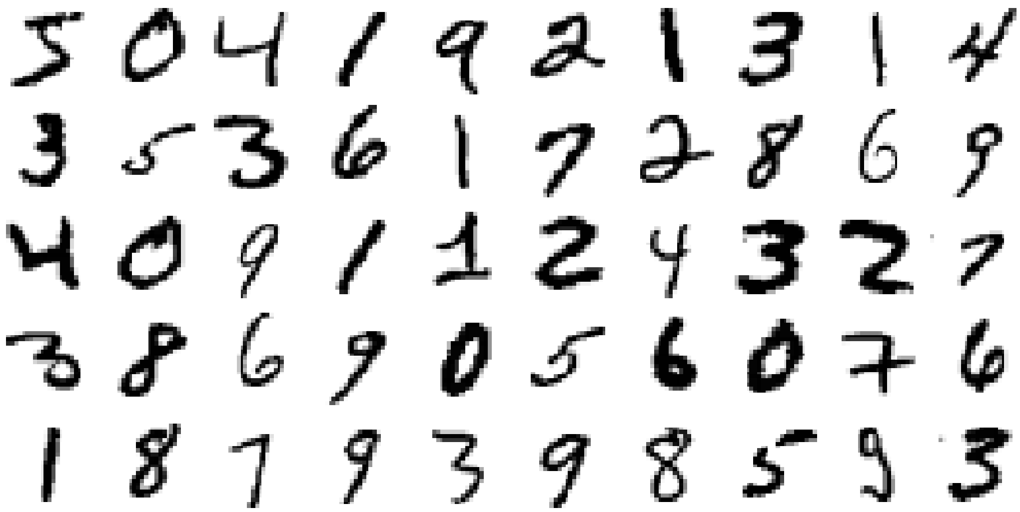


0.1.1 Show images of numbers

In [19]:
```
#
import numpy as np
# import matplotlib as mpl
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

In [20]:
```
plt.figure(figsize=(9,9))
example_images = np.r_[X_train0[:50]]
plot_digits(example_images, images_per_row=10)

plt.show()
```



0.1.2 float .

```
In [21]: X_train = X_train0.reshape(60000, 784).astype('float32') / 255.0
         X_test = X_test0.reshape(10000, 784).astype('float32') / 255.0
         print(X_train.shape, X_train.dtype)
```

```
(60000, 784) float32
```

0.1.3 y One-Hot-Encoding .

```
In [22]: y_train0[:5]
```

```
Out[22]: array([5, 0, 4, 1, 9], dtype=uint8)
```

```
In [23]: from keras.utils import np_utils

         Y_train = np_utils.to_categorical(y_train0, 10)
         Y_test = np_utils.to_categorical(y_test0, 10)
         Y_train[:5]
```

```
Out[23]: array([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
                [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]], dtype=float32)
```

0.2

0.2.1 Keras .

1. Sequential
2. add layer .
   - Dense layer
   - .
   - .
   - input_dim .
   - activation activation
3. compile .
   - loss Loss
   - optimizer
   - metrics
4. fit
   - nb_epoch epoch
   - batch_size mini batch size
   - metrics
   - Jupyter Notebook verbose=2 progress bar .

In [24]:
```
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD
import numpy as np
```

In [25]:
```
# Learning model
np.random.seed(0)

model = Sequential()
model.add(Dense(15, input_dim=784, activation="sigmoid"))  # firsr layer
model.add(Dense(10, activation="sigmoid")) # output layer
```

model_to_dot summary layers .

In [26]:
```
# !pip install pydot
```

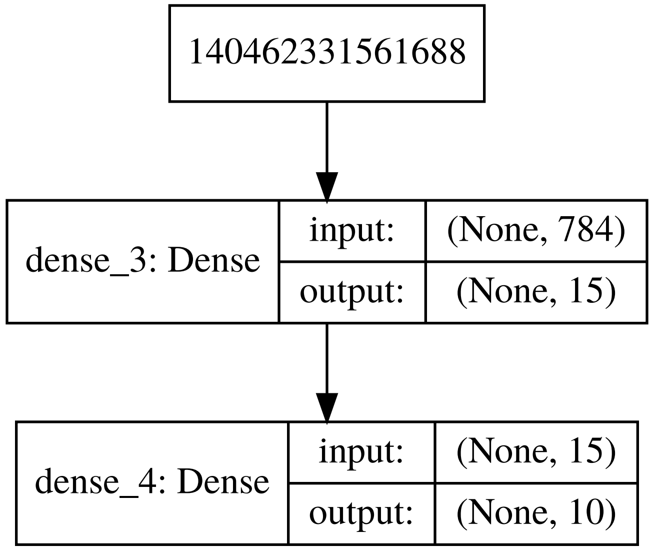
In [27]:
```
# !pip install GraphViz
```

In [28]:
```
import pydot
import graphviz
```

In [29]:
```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model, show_shapes=True).create(prog='dot', format='svg'))
```

Out[29]:




```
In [30]: from keras.utils import plot_model
         plot_model(model, to_file='model.png')

In [31]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 15)                11775
_________________________________________________________________
dense_4 (Dense)              (None, 10)                160
=================================================================
Total params: 11,935
Trainable params: 11,935
Non-trainable params: 0
_________________________________________________________________
```

```
In [32]: l1 = model.layers[0]
         l2 = model.layers[1]

In [33]: l1.name, type(l1), l1.output_shape, l1.activation.__name__, l1.count_params()

Out[33]: ('dense_3', keras.layers.core.Dense, (None, 15), 'sigmoid', 11775)

In [34]: l2.name, type(l1), l2.output_shape, l2.activation.__name__, l2.count_params()

Out[34]: ('dense_4', keras.layers.core.Dense, (None, 10), 'sigmoid', 160)
```

0.3 fit

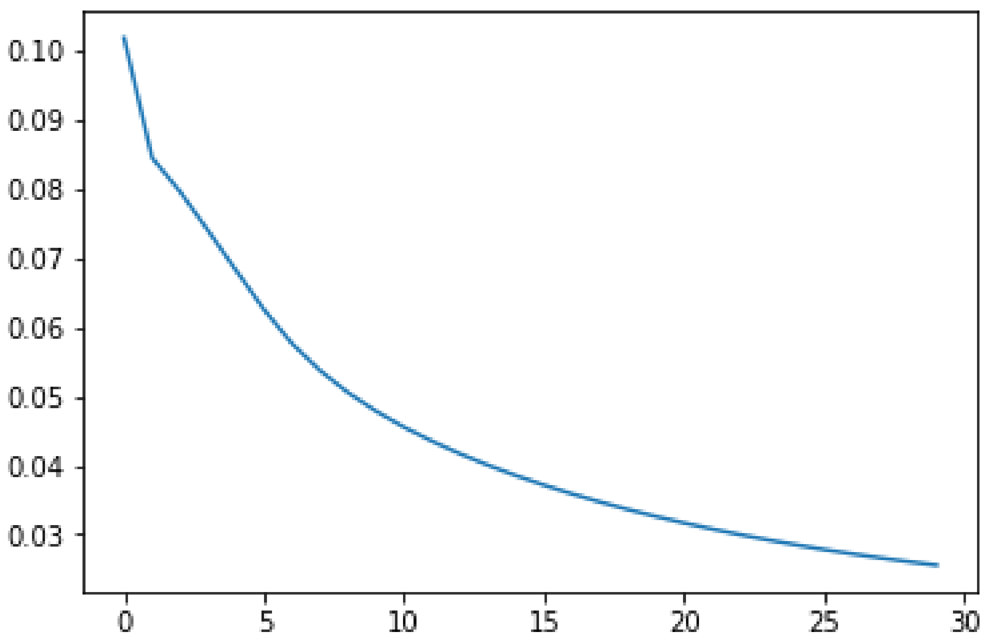
```
In [35]: model.compile(optimizer=SGD(lr=0.2), loss='mean_squared_error', metrics=["accuracy"])
```

```
In [36]: %%time
         hist = model.fit(X_train, Y_train,
                          epochs=30, batch_size=100,
                          validation_data=(X_test, Y_test),
                          verbose=1)
```

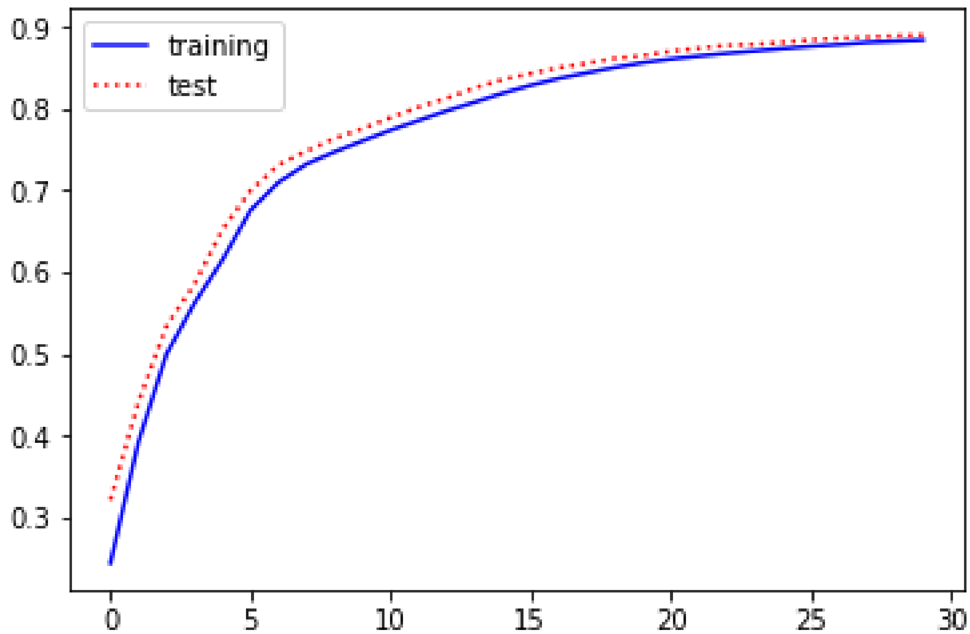
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 2s 30us/step - loss: 0.1019 - acc: 0.2440 - val_
Epoch 2/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0845 - acc: 0.3921 - val_
Epoch 3/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0796 - acc: 0.4997 - val_
Epoch 4/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0740 - acc: 0.5620 - val_
Epoch 5/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0682 - acc: 0.6149 - val_
Epoch 6/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0625 - acc: 0.6759 - val_
Epoch 7/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0576 - acc: 0.7101 - val_
Epoch 8/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0537 - acc: 0.7325 - val_
Epoch 9/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0505 - acc: 0.7474 - val_
Epoch 10/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0478 - acc: 0.7608 - val_
Epoch 11/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0455 - acc: 0.7739 - val_
Epoch 12/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0435 - acc: 0.7859 - val_
Epoch 13/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0417 - acc: 0.7977 - val_
Epoch 14/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0400 - acc: 0.8083 - val_
Epoch 15/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0385 - acc: 0.8192 - val_
Epoch 16/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0371 - acc: 0.8288 - val_
Epoch 17/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0359 - acc: 0.8371 - val_
Epoch 18/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0347 - acc: 0.8440 - val_
Epoch 19/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0336 - acc: 0.8506 - val_
```

```
Epoch 20/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0326 - acc: 0.8565 - val_
Epoch 21/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0316 - acc: 0.8610 - val_
Epoch 22/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0308 - acc: 0.8646 - val_
Epoch 23/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0299 - acc: 0.8679 - val_
Epoch 24/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0292 - acc: 0.8709 - val_
Epoch 25/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0285 - acc: 0.8738 - val_
Epoch 26/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0278 - acc: 0.8765 - val_
Epoch 27/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0272 - acc: 0.8787 - val_
Epoch 28/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0266 - acc: 0.8814 - val_
Epoch 29/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0261 - acc: 0.8829 - val_
Epoch 30/30
60000/60000 [==============================] - 1s 10us/step - loss: 0.0256 - acc: 0.8844 - val_
CPU times: user 36.2 s, sys: 4.04 s, total: 40.3 s
Wall time: 19.2 s
```

In [37]:
```python
# Plot performance
plt.plot(hist.history['loss'])
plt.show()
```



In [38]:
```
plt.plot(hist.history['acc'], 'b-', label="training")
plt.plot(hist.history['val_acc'], 'r:', label="test")
plt.legend()
plt.show()
```



0.4

get_weights . w b .

In [39]:
```
w1 = l1.get_weights()
w1[0].shape, w1[1].shape
```

Out[39]: ((784, 15), (15,))

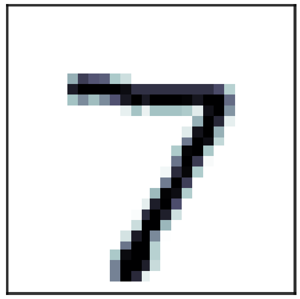
In [40]:
```
w2 = l2.get_weights()
w2[0].shape, w2[1].shape
```

Out[40]: ((15, 10), (10,))

0.5

predict y y predict_classes classification .

```
In [41]: plt.figure(figsize=(2, 2))
         plt.imshow(X_test0[0], cmap=mpl.cm.bone_r)
         plt.grid(False)
         plt.xticks([])
         plt.yticks([])
         plt.show()
```



```
In [42]: model.predict(X_test[:1, :])

Out[42]: array([[0.01766999, 0.01917328, 0.01319782, 0.02869662, 0.01251966,
                 0.05721793, 0.01232018, 0.9418675 , 0.00762814, 0.05536831]],
               dtype=float32)

In [43]: model.predict_classes(X_test[:1, :], verbose=0)

Out[43]: array([7])
```

0.6

save hdf5 load .

```
In [44]: model.save('my_model.hdf5')
         # del model

In [45]: from keras.models import load_model

         model2 = load_model('my_model.hdf5')
         model2.predict_classes(X_test[:1, :], verbose=0)

Out[45]: array([7])

In [46]: model2.predict_classes(X_test[:10, :], verbose=0)

Out[46]: array([7, 2, 1, 0, 4, 1, 4, 9, 6, 9])

In [47]: y_test0[:10]

Out[47]: array([7, 2, 1, 0, 4, 1, 4, 9, 5, 9], dtype=uint8)
```

0.6.1

```
In [48]: # Wrong prediction
         model2.predict_classes(X_test[8:9, :], verbose=1)

1/1 [==============================] - 0s 787us/step


Out[48]: array([6])

In [49]: y_test0[8]

Out[49]: 5

In [50]: x_pred = model2.predict_classes(X_test, verbose=1)

10000/10000 [==============================] - 0s 10us/step


In [51]: t_count = np.sum(x_pred==y_test0) # True positive
         f_count = np.sum(x_pred!=y_test0) # False positive
         f_count==10000-t_count

Out[51]: True

In [52]: t_count,f_count

Out[52]: (8904, 1096)

In [53]: accuracy = t_count/10000*100
         accuracy

Out[53]: 89.03999999999999
```

0.6.2 Accuracy of predicting test numbers is around 89% in simple neural network model.