Deep Networks Observations

```
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers.convolutional import MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils
import matplotlib.pyplot as plt
import os
import numpy as np


batch_size = 32
num_classes = 10
epochs = 1

num_predictions = 20
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'keras_cifar10_trained_model.h5'

(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print("Training data:")
print( "Number of examples: ", X_train.shape[0])
print( "Number of channels:",X_train.shape[3] )
print( "Image size:", X_train.shape[1], X_train.shape[2])
print("\n")
print( "Test data:")
print( "Number of examples:", X_test.shape[0])
print( "Number of channels:", X_test.shape[3])
print( "Image size:",X_test.shape[1], X_test.shape[2])



plot = []
for i in range(1,10):
    plot_image = X_train[100*i,:,:,:]
    for j in range(1,10):
```

```
        plot_image = np.concatenate((plot_image, X_train[100*i+j,:,:,:]), axis=1)
    if i==1:
        plot = plot_image
    else:
        plot = np.append(plot, plot_image, axis=0)
print(plot.shape, np.max(plot), np.min(plot))
plt.imshow(plot/255)
plt.axis('off')
plt.show()


print("mean before normalization:", np.mean(X_train))
print("std before normalization:", np.std(X_train))

mean=[0,0,0]
std=[0,0,0]
newX_train = np.ones(X_train.shape)
newX_test = np.ones(X_test.shape)
for i in range(3):
    mean[i] = np.mean(X_train[:,:,:,i])
    std[i] = np.std(X_train[:,:,:,i])

for i in range(3):
    newX_train[:,:,:,i] = X_train[:,:,:,i] - mean[i]
    newX_train[:,:,:,i] = newX_train[:,:,:,i] / std[i]
    newX_test[:,:,:,i] = X_test[:,:,:,i] - mean[i]
    newX_test[:,:,:,i] = newX_test[:,:,:,i] / std[i]


X_train = newX_train
X_test = newX_test

print("mean after normalization:", np.mean(X_train))
print("std after normalization:", np.std(X_train))



batchSize = 50                  #-- Training Batch Size
num_classes = 10                #-- Number of classes in CIFAR-10 dataset
num_epochs = 10                 #-- Number of epochs for training
learningRate= 0.001             #-- Learning rate for the network
lr_weight_decay = 0.95          #-- Learning weight decay. Reduce the learn rate by 0.95 after
epoch
```

```
img_rows, img_cols = 32, 32       #-- input image dimensions

Y_train = np_utils.to_categorical(y_train, num_classes)
Y_test = np_utils.to_categorical(y_test, num_classes)



model = Sequential()                                    #-- Sequential container.

model.add(Convolution2D(6, 5, 5,                        #-- 6 outputs (6 filters), 5x5 convolution
kernel
                        border_mode='valid',
                        input_shape=( img_rows, img_cols, 3)))       #-- 3 input depth (RGB)
model.add(Activation('relu'))                           #-- ReLU non-linearity



# model.add(Convolution2D(8, 5, 5))                     #-- 16 outputs (16 filters), 5x5
convolution kernel
# model.add(Activation('relu'))                         #-- ReLU non-linearity


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



                        #-- A max-pooling on 2x2 windows
model.add(Convolution2D(16, 5, 5))                      #-- 16 outputs (16 filters), 5x5
convolution kernel
model.add(Activation('relu'))                           #-- ReLU non-linearity
model.add(MaxPooling2D(pool_size=(2, 2)))


# model.add(Convolution2D(26, 5, 5))                    #-- 26 outputs (16 filters), 5x5
convolution kernel
# model.add(Activation('relu'))                         #-- ReLU non-linearity
# model.add(MaxPooling2D(pool_size=(2, 2)))
#                       #-- A max-pooling on 2x2 windows

model.add(Flatten())                                    #-- eshapes a 3D tensor of 16x5x5 into 1D
tensor of 16*5*5
```

```
model.add(Dense(120))                                   #-- 120 outputs fully connected layer
model.add(Activation('relu'))                           #-- ReLU non-linearity
model.add(Dense(84))                                    #-- 84 outputs fully connected layer
model.add(Activation('relu'))                           #-- ReLU non-linearity
model.add(Dense(num_classes))                                   #-- 10 outputs fully connected layer
(one for each class)
model.add(Activation('softmax'))                          #-- converts the output to a
log-probability. Useful for classification problems

print(model.summary())


sgd = SGD(lr=learningRate, decay = lr_weight_decay)
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

#-- switch verbose=0 if you get error "I/O operation from closed file"
history = model.fit(X_train, Y_train, batch_size=batchSize, epochs=num_epochs,
          verbose=1, shuffle=True, validation_data=(X_test, Y_test))



plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

#-- summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

3 Layer Neural Network

Dermatology Dataset

```
import csv
import sys
import random
import random
import numpy as np
import cPickle
import gzip
import numpy as np
import random


def vectorized_result(j):
    e = np.zeros((3, 1))
    e[int(j)] = 1.0
    return e

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))


class Neural_Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y,x) for x, y in zip(sizes[:-1], sizes[1:])]

        # print self.weights[0].shape
        # print self.weights[1].shape
```

```
    def cost_derivative(self, output_activations, y):
        return (output_activations - y)


    def cross_entropy(self,batch_size, output, expected_output):
        return (-1/batch_size) * np.sum(expected_output * np.log(output) + (1 - expected_output) *
np.log(1-output))


    def backprop(self, x, y):

            nabla_b = [np.zeros(b.shape) for b in self.biases]
            nabla_w = [np.zeros(w.shape) for w in self.weights]

            activation = x
        activations = [x]
        zs = []

        # print "activation"
        # print activation
        #
        # print "activations"
        # print activations

        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)

            # print "z"
            # print z

            activation = sigmoid(z)
            activations.append(activation)

        delta = self.cross_entropy(len(self.weights),activations[-1], y) * \
            sigmoid_prime(zs[-1])


        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        # print "before",delta.shape
```

```
    # print delta



    for l in xrange(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta

        # print delta.shape
        # print activations[-l-1].shape
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())


    # print "nabla_b"
    # print nabla_b
    #
    # print "nabla_w"
    # print nabla_w

        return (nabla_b, nabla_w)




def update_mini_batch(self, mini_batch, eta):


    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # print self.weights[1][0]

    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)




        nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
```

```
    self.weights = [w - (eta / len(mini_batch)) * nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b - (eta / len(mini_batch)) * nb for b, nb in zip(self.biases, nabla_b)]

def feedforward(self, a):
    """Return the output of the network if ``a`` is input."""
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a) + b)
    return a

def check_postition(self,data):

    for i,x in enumerate(data):
        if(x==1.0):
            return i+1


def evaluate(self, test_data):
    """Return the number of test inputs for which the neural
    network outputs the correct result. Note that the neural
    network's output is assumed to be the index of whichever
    neuron in the final layer has the highest activation."""
    test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data]

    count=0

    predicted=[]
    original=[]
    for x,y in test_results:
        # print x
        predicted.append(x)

        t=self.check_postition(y)
        original.append(t)

       # # print t
       # if(x==t):
       #     count=count+1

    # print "original"
    # print original
    # print "predicted"
    # print predicted
```

```
    correct = 0
    for i in range(len(original)):
        if original[i] == predicted[i]:
            correct += 1

    return correct



def SGD(self, training_data, epochs, mini_batch_size, eta):

    n = len(training_data)

    test_data=""


    print "epoch",epochs

    print "mini_batch_size",mini_batch_size

    random.shuffle(training_data)
    mini_batches = [training_data[k:k + mini_batch_size] for k in xrange(0, n, mini_batch_size)]

    for k in range(0,len(mini_batches)):

            if k==0:
                test_data = mini_batches[0]
                mini_batches.pop(0)
            else:
                mini_batches.append(test_data)
                test_data = mini_batches[k]
                mini_batches.pop(k)


            flag = 0

            for j in xrange(epochs):
                # print "epoch", j
                if flag == 1:
                    break
```

```
                for mini_batch in mini_batches:
                    # print "inside loop"
                    self.update_mini_batch(mini_batch, eta)

                    # print mini_batch[0]

                if test_data:
                    # print "outside loop////////////////////////////////////"
                    val = self.evaluate(test_data)
                    # print "after validation"
                    #
                    percent = val / float(len(test_data)) * 100
                    print "Number of fold :",k,"epoch :",j,"percentage",percent

                    if (percent > 80):
                        flag = 1









if __name__=="__main__":

    file = open('dermatology.data', 'r')

    list_of_labels=[]
    list_of_dataset=[]
    list_of_data=[]

    for line in file:
        data=[]
        if "?" in line:
            continue
        line = line.split(",")
        if(len(line)>0):
            for val in line:
```

```
            if(val==" "):
                continue
            else:
                data.append(val)
                # print(val)
                # print(float(val))

        # print(line)
        list_of_data.append(data)
    # print(list_of_data)
    DataSetList=[]
    for row in list_of_data:

        data = []
        for ele in row:
            # print(ele)
            data.append(float(ele))

        DataSetList.append(data)
    list_of_labels=[]
    list_of_dataset=[]
    for row in DataSetList:
        if row[0]==0:
            continue
        else:
            temp = vectorized_result(row[0]-1)
            list_of_labels.append(temp)
            list_of_dataset.append(row[1:len(row)])

list_of_dataset = np.asarray(list_of_dataset)
list_of_labels = np.array(list_of_labels)
temp_dataset=[]
for data in list_of_dataset:
    x=np.array(data).reshape(34,1)
    # print x
    temp_dataset.append(x)
list_of_dataset=temp_dataset
training_data = zip(list_of_dataset,list_of_labels)
# print training_data[0]
network = Neural_Network([34,16,3])
print "Number of hidden layers 16"
batch_size = len(training_data)/5
network.SGD(training_data,5, batch_size, 0.0001)
```

Pendigit Dataset

```
import csv
import sys
import random
import random
import numpy as np

import cPickle
import gzip

import numpy as np
import random



def vectorized_result(j):
    e = np.zeros((4,1))
    e[int(j)] = 1.0
    return e

# def sigmoid(z):
#     """The sigmoid function."""
#     z=2*z
#     return 2.0/(1.0+np.exp(-z))

# def sigmoid_prime(z):
#     """Derivative of the sigmoid function."""

#     z=2*z
#     f= 2.0/(1.0+np.exp(-z))
#     return 1-f**2


def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

def softmax(z):
```

```
    z = np.exp(z)
    g = z/np.sum(z)
    return g


def softmax_prime(z):


    return z*(1-z)

class Neural_Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y,x) for x, y in zip(sizes[:-1], sizes[1:])]

        # print self.weights[0].shape
        # print self.weights[1].shape


    def cost_derivative(self, output_activations, y):
        return (output_activations - y)


    def cross_entropy(self,batch_size, output, expected_output):
        return (-1/batch_size) * np.sum(expected_output * np.log(output) + (1 - expected_output) *
np.log(1-output))


    def backprop(self, x, y):

            nabla_b = [np.zeros(b.shape) for b in self.biases]
            nabla_w = [np.zeros(w.shape) for w in self.weights]

            activation = x
        activations = [x]
        zs = []

        # print "activation"
        # print activation
```

```
#
# print "activations"
# print activations


# print "length "
# print len(self.biases)


i=0
for b, w in zip(self.biases, self.weights):
    z = np.dot(w, activation)+b
    zs.append(z)

    # print "z"
    # print z

    if (i!=len(self.weights)-1):
        activation = sigmoid(z)
        activations.append(activation)
    else:
        activation = softmax(z)
        activations.append(activation)

delta = self.cross_entropy(len(self.weights),activations[-1], y) * \
    sigmoid_prime(zs[-1])


nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())

# print "before",delta.shape
# print delta



for l in xrange(2, self.num_layers):
    z = zs[-l]
    sp = sigmoid_prime(z)
    delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
    nabla_b[-l] = delta

    # print delta.shape
```

```
        # print activations[-l-1].shape
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())


    # print "nabla_b"
    # print nabla_b
    #
    # print "nabla_w"
    # print nabla_w

        return (nabla_b, nabla_w)

def update_mini_batch(self, mini_batch, eta):


    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # print self.weights[1][0]

    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)




        nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

    self.weights = [w - (eta / len(mini_batch)) * nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b - (eta / len(mini_batch)) * nb for b, nb in zip(self.biases, nabla_b)]

def feedforward(self, a):
    """Return the output of the network if ``a`` is input."""
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a) + b)
    return a

def check_postition(self,data):

    for i,x in enumerate(data):
        if(x==1.0):
            return i+1
```

```
def evaluate(self, test_data):
    """Return the number of test inputs for which the neural
    network outputs the correct result. Note that the neural
    network's output is assumed to be the index of whichever
    neuron in the final layer has the highest activation."""
    test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data]

    count=0

    predicted=[]
    original=[]
    for x,y in test_results:
        # print x
        predicted.append(x)

        t=self.check_postition(y)
        original.append(t)

       # # print t
       # if(x==t):
       #     count=count+1

    # print "original"
    # print original
    # print "predicted"
    # print predicted

    correct = 0
    for i in range(len(original)):
        if original[i] == predicted[i]:
            correct += 1

    return correct




def SGD(self, training_data, epochs, mini_batch_size, eta):
```

```
n = len(training_data)

test_data=""


print "epoch",epochs

print "mini_batch_size",mini_batch_size

random.shuffle(training_data)
mini_batches = [training_data[k:k + mini_batch_size] for k in xrange(0, n, mini_batch_size)]

for k in range(0,len(mini_batches)):

        if k==0:
            test_data = mini_batches[0]
            mini_batches.pop(0)
        else:
            mini_batches.append(test_data)
            test_data = mini_batches[k]
            mini_batches.pop(k)


        flag = 0

        for j in xrange(epochs):
            # print "epoch", j
            if flag == 1:
                break




            for mini_batch in mini_batches:
                # print "inside loop"
                self.update_mini_batch(mini_batch, eta)

                # print mini_batch[0]

            if test_data:
                # print "outside loop/////////////////////////////////////"
                val = self.evaluate(test_data)
                # print "after validation"
```

```
                    #
                    percent = val / float(len(test_data)) * 100
                    print "Fold No:",k,"epoch :",j,"percentage",percent

                    if (percent > 80):
                        flag = 1




if __name__=="__main__":



    file = open('pendigit.tes', 'r')

    list_of_labels=[]
    list_of_dataset=[]

    list_of_data=[]
##############################################################################
###############################################
    for line in file:
        data=[]
        if "?" in line:
            continue
        line = line.split(",")
        if(len(line)>0):
            for val in line:

                if(val==" "):
                    continue
                else:
                    data.append(val)
                    # print(val)
                    # print(float(val))

        # print(line)
        list_of_data.append(data)
```

```
    # print(list_of_data)


    DataSetList=[]
    for row in list_of_data:
        data = []
        for ele in row:
            # print(ele)
            data.append(float(ele))

        DataSetList.append(data)

    list_of_labels=[]
    list_of_dataset=[]
    for row in DataSetList:
       if row[len(row)-1]==0 or row[len(row)-1]==1 or row[len(row)-1]==2 or row[len(row)-1]==3:
            temp = vectorized_result(row[len(row)-1])
            list_of_labels.append(temp)
            list_of_dataset.append(row[0:len(row)-1])



list_of_dataset = np.asarray(list_of_dataset)

list_of_labels = np.array(list_of_labels)

temp_dataset=[]

for data in list_of_dataset:
    x=np.array(data).reshape(16,1)
    # print x
    temp_dataset.append(x)

list_of_dataset=temp_dataset

training_data = zip(list_of_dataset,list_of_labels)
network = Neural_Network([16,16,4])
print "Number of hidden layers 16"
batch_size = len(training_data)/5
network.SGD(training_data,5,batch_size, 0.001)
```

Observations for 1.2

1

What are the number of parameters in convolution layers with K filters each of size 3*w*h.

(3*w*h+1)*k

2

What are the number of parameters in a max pooling operation?

zero

3

Which of the operations contain most number of parameters?

Fully connected layer (FC)

4

Which operation consume most amount of memory?

Fully connected layers at the end

5

Effect of Learning Rate:

Increasing Learning Rate

Model accuracy takes more epoch to reach certain accuracy level which was attained earlier (0.001)to(0.0001)

Model loss takes more epoch to reach certain accuracy level which was attained earlier (0.001)to(0.0001)

6

Batch-size:

Model was able to achieve 50% accuracy during testing when batch size was 50,when changed to 1,it was not able to achieve 50 % accuracy even after 8 epochs.

Model loss curve is also not smooth its fluctuating too much and it never reaches minimum level.

7

Number of convolution filters:

Model accuracy increased for each epochs compared previous time(6,16)to(12,20)

Model Loss (1.2)is reached in less epoch than previous time(6,16)to(12,20)

8

Number of layers:

Model accuracy curve looks more smooth and matches closer to the training curve

In the beginning loss seems to be high,but during each epoch the curve is decreasing continuously and looks somewhat similar to training

9

Activation functions:

The curve is not linear and minimum loss is not able to achieve in this(sigmoid) case which we are able to achieve with relu

10

Training data:

Adding Salt and Pepper noise

Rotation (at finer angles)