In [3]:
```
import pandas as pd
import json
import urllib
import numpy as np
```

In [4]:
```
import tensorflow as tf
import numpy as np
import os
import sys
import random
from PIL import Image
from PIL import ImageEnhance
import PIL.ImageOps
from six.moves import cPickle as pickle
from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, MaxPooling2D, Flatte
n, Conv3D, MaxPooling3D, LSTM
from keras.layers import Dense, Dropout
```

```
Using TensorFlow backend.
```

In [5]:
```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, MaxPooling2D, Flatte
n, Conv3D, MaxPooling3D
from keras.layers import Dense, Dropout

from keras.optimizers import SGD

from keras import backend as K
```

Image Data Processing

In the data processing step, we firstly joined the tmdb file and the clustering result data by tmdb id. And then we filtered out the movies without tmdb id and poster link. After these steps, we have 63823 samples. According to the poster link, we then retrieved 26,000 posters through urllib package. After downloading the posters, we began to covert the raw image data into pixel matrices. Each image will be converted to RGB style matrices. There are three number of channels representing R, G, B values. So the final input values should be like 154x154x3. And the label should be in the form of nx7.

Read pre-possed data

In [4]:
```
df=pd.read_csv('/home/ubuntu/data/whole_data.csv')
```

```
/home/ubuntu/.local/lib/python2.7/site-packages/IPython/core/interactiv
eshell.py:2717: DtypeWarning: Columns (0,1,4,12,23) have mixed types. S
pecify dtype option on import or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)
```

In [5]:
```
df=df[df['tmdb_id'].isnull()==False]
```

In [5]:
```
len(df['poster_path'])
```

Out[5]:
```
63823
```

Download the image

In [ ]:
```
# download img
for i in range(len(df)):

urllib.urlretrieve('https://image.tmdb.org/t/p/w154'+df.poster_path[i],'/
e/ubuntu/poster/'+str(df.tmdb_id[i])+'.jpg')
```

Read the response variables

In [12]:
```
l=
['tmdb_id','cluster_1','cluster_2','cluster_3','cluster_4','cluster_5','c
ter_6','cluster_7']
df_3 = df.ix[:,l]
df_3.head()
```

Out[12]:

	tmdb_id	cluster_1	cluster_2	cluster_3	cluster_4	cluster_5	cluster_6	cluster_7
0	100	0	0	0	1	0	1	0
1	10001	0	0	1	1	0	0	0
2	10002	1	0	0	1	0	1	0
3	10003	1	1	0	0	1	0	1
4	10004	0	1	0	1	1	1	0

Generate the dataset

In [48]:
```
IMAGE_WIDTH=154
IMAGE_HEIGHT=154
NUM_CHANNELS=3
PIXEL_DEPTH=255.0
DATA_PATH = '/home/ubuntu/poster/'
poster=[0 for i in range(26000)]
```

In [49]:
```
tmdb_id=[0 for i in range(26000)]
```

Convert the image to pixel matrix

In [51]:
```
def to_rgb(im):
    w, h = im.shape
    ret = np.empty((w, h, 3), dtype=np.uint8)
    ret[:, :, :] = im[:, :, np.newaxis]
    return ret
def read_image_from_file(file_path):
    img = Image.open(file_path).convert('RGB')
      #downsample image
    img = img.resize((154,154), Image.ANTIALIAS)
    pixel_values = np.array(img.getdata())
    return np.reshape(pixel_values, [IMAGE_WIDTH,IMAGE_HEIGHT, NUM_CHANN
ELS])
def scale_pixel_values(dataset):
    return (dataset - 255.0 / 2.0) / 255.0
```

In [52]:
```
i=0
j=0
for filename in os.listdir(DATA_PATH):
    if i <26000:
        idx= int(filename.split('.')[0])
        image=read_image_from_file(DATA_PATH+filename)
        poster[i]=image
        tmdb_id[j]=idx
        i+=1
        j+=1
```

In [10]:
```
df_img = pd.DataFrame({'img':poster, 'tmdb_id':tmdb_id})
```

In [13]:
```
#df_whole=pd.merge(df_img, df_y,on = 'tmdb_id', how='inner')
df_whole=pd.merge(df_img, df_3,on = 'tmdb_id', how='inner')
```

In [14]:
```
df_whole.head()
```

Out[14]:

	img	tmdb_id	cluster_1	cluster_2	cluster_3	cluster_4	cluster_5	cluster_6	cluster_7
0	[[[253, 249, 248], [253, 249, 248], [253, 249,...	43819	1	0	0	1	0	0	0
1	[[[19, 12, 23], [23, 16, 26], [26, 21, 30], [2...	36996	0	1	0	1	0	0	1
2	[[[182, 153, 145], [182, 153, 145], [179, 151,...	16520	1	0	1	1	0	0	0
3	[[[242, 244, 243], [245, 247, 245], [242, 245,...	29960	0	1	0	0	0	0	0
4	[[[50, 51, 43], [21, 22, 14], [22, 23, 15], [1...	41678	0	0	0	1	0	0	0

In [15]:
```
df_whole.to_pickle('whole_img.p')
```

In [8]:
```
df_whole= pd.read_pickle('whole_img.p')
df_whole.head()
```

Out[8]:

	img	tmdb_id	cluster_1	cluster_2	cluster_3	cluster_4	cluster_5	cluster_6	cluster_7
0	[[[253, 249, 248], [253, 249, 248], [253, 249,...	43819	1	0	0	1	0	0	0
1	[[[19, 12, 23], [23, 16, 26], [26, 21, 30], [2...	36996	0	1	0	1	0	0	1
2	[[[182, 153, 145], [182, 153, 145], [179, 151,...	16520	1	0	1	1	0	0	0
3	[[[242, 244, 243], [245, 247, 245], [242, 245,...	29960	0	1	0	0	0	0	0
4	[[[50, 51, 43], [21, 22, 14], [22, 23, 15], [1...	41678	0	0	0	1	0	0	0

Generate the train, validation and test set

In [9]:
```
train_X=np.zeros((10000,154,154, 3))
```

In [10]:
```
x= df_whole.img[:10000].values
```

In [11]:
```
for i in range(10000):
    train_X[i]=x[i]
```

In [12]:
```
train_Y=df_whole.ix[:9999,2: ].values
```

In [15]:
```
vali_X=np.zeros((5000,154,154, 3))
x= df_whole.img[10000:15000].values
for i in range(5000):
    vali_X[i]=x[i]
vali_Y=df_whole.ix[10000:14999,2: ].values
```

In [16]:
```
x_test= np.zeros((5000,154,154, 3))
x= df_whole.img[15000:20000].values
for i in range(5000):
    x_test[i]=x[i]
y_test=df_whole.ix[15000:19999,2: ].values
```

Model from scratch

Model 1

As for the model from scratch, we started our idea by considering build convolutional neural network. The reason that we choose CNN is because the raw data we have is the image data. We began our experiment with one input layer(conv2D) and one con2D hidden layer and one output layer with 7 output units(sigmoid).

The input data is the image pixel data matrices. Each image has 154x154 dimensions and 3 channels. The response variable is n x 7. Each image will have seven response variables. The seven columns are the cluster results we generated in the previous milestone.

In [7]:
```
input_shape = ( 154,154, 3)
```

In [42]:
```
model3 = Sequential()
# input: 100x100 images with 3 channels -> (100, 100, 3) tensors.
# this applies 32 convolution filters of size 3x3 each.
model3.add(Conv2D(32, (3, 3), activation='relu',
input_shape=input_shape))
model3.add(Conv2D(32, (3, 3), activation='relu'))


model3.add(Flatten())
model3.add(Dense(7,activation='sigmoid'))

# prints out a summary of the model architecture
model3.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_21 (Conv2D)           (None, 152, 152, 32)      896
_________________________________________________________________
flatten_7 (Flatten)          (None, 739328)            0
_________________________________________________________________
dense_13 (Dense)             (None, 7)                 5175303
=================================================================
Total params: 5,176,199.0
Trainable params: 5,176,199.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [43]:
```
sgd = SGD(lr=0.1, momentum=0.9)
model3.compile(loss='binary_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
history = model3.fit(train_X, train_Y,
                     batch_size=batch_size,
                     epochs=epochs,
                     verbose=1,
                     validation_data=(vali_X, test_Y))
```

```
Train on 10000 samples, validate on 5000 samples
Epoch 1/10
10000/10000 [==============================] - 20s - loss: 3.9037 - ac
c: 0.7549 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 2/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 3/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 4/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 5/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 6/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 7/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 8/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 9/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 10/10
10000/10000 [==============================] - 20s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
```

In [46]:
```
score = model3.evaluate(x_test, y_test, verbose=0)
score[1]
```

Out[46]: 0.76628572473526002

Pre-tuned model VGG-like model

After we tried to build our own models from scratch, we began to explore some pre-tuned models. VGG-like models are one of our considerations.VGG16 and VGG19 are common models for image processing and classification in deep learning. So we decided to go with VGG-like convnet model. The inpute shape is 154x154x3. There are 10000 training samples, 5000 validation samples, and 5000 test samples.

The structure of this convnet is:

Conv2D (input layer) -> Conv2D -> MaxPooling2D -> Conv2D -> Conv2D -> MaxPooling2D -> Dense (relu) -> Dense(output, sigmoid, 7)

As for the input layer, we set a 5x5 conv window with stride 1. We set 3x3 window with stride 1 for other conv layer. We have the reLU activation function for each conv layer. And we have the factor that used to downscaling to be 2x2 for maxpooling2D layer.

In [34]:
```
model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu',
input_shape=input_shape))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))


model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(7,activation='sigmoid'))

# prints out a summary of the model architecture
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_11 (Conv2D)           (None, 150, 150, 32)      2432
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 148, 148, 32)      9248
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
dropout_9 (Dropout)          (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 72, 72, 32)        9248
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 70, 70, 32)        9248
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 35, 35, 32)        0
_________________________________________________________________
dropout_10 (Dropout)         (None, 35, 35, 32)        0
_________________________________________________________________
flatten_5 (Flatten)          (None, 39200)             0
_________________________________________________________________
dense_9 (Dense)              (None, 64)                2508864
_________________________________________________________________
dropout_11 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_10 (Dense)             (None, 7)                 455
=================================================================
Total params: 2,539,495.0
Trainable params: 2,539,495.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [35]:
```
batch_size= 500
epochs=10
```

In [36]:
```
sgd = SGD(lr=0.1, momentum=0.9)
model.compile(loss='binary_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
history = model.fit(train_X, train_Y,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(vali_X, test_Y))
```

```
Train on 10000 samples, validate on 5000 samples
Epoch 1/10
10000/10000 [==============================] - 52s - loss: 4.4895 - ac
c: 0.7168 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 2/10
10000/10000 [==============================] - 47s - loss: 4.5019 - ac
c: 0.7194 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 3/10
10000/10000 [==============================] - 47s - loss: 4.5237 - ac
c: 0.7180 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 4/10
10000/10000 [==============================] - 47s - loss: 4.5253 - ac
c: 0.7179 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 5/10
10000/10000 [==============================] - 47s - loss: 4.5418 - ac
c: 0.7169 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 6/10
10000/10000 [==============================] - 47s - loss: 4.5157 - ac
c: 0.7185 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 7/10
10000/10000 [==============================] - 47s - loss: 4.5306 - ac
c: 0.7176 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 8/10
10000/10000 [==============================] - 47s - loss: 4.5424 - ac
c: 0.7168 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 9/10
10000/10000 [==============================] - 47s - loss: 4.5294 - ac
c: 0.7176 - val_loss: 4.6418 - val_acc: 0.7105
Epoch 10/10
10000/10000 [==============================] - 47s - loss: 4.5347 - ac
c: 0.7173 - val_loss: 4.6418 - val_acc: 0.7105
```

In [37]:
```
score = model.evaluate(x_test, y_test, verbose=0)
```

In [38]:
```
score[1]
```

Out[38]:
```
0.70742858753204341
```

Model 2

As for the next exploration, we decide to add more layers to our final model. We then add 2 more cov2D layers and one more Maxpooling2D layer.

The structure then became like:

Conv2D (input layer) -> Conv2D -> MaxPooling2D -> Conv2D -> Conv2D -> MaxPooling2D -> Conv2D -> Conv2D -> MaxPooling2D -> Dense (relu) -> Dense(output, sigmoid, 7)

As for the input layer, we set a 5x5 conv window with stride 1. We set 3x3 window with stride 1 for other conv layer. We have the reLU activation function for each conv layer. And we have the factor that used to downscaling to be 2x2 for maxpooling2D layer.

Since there more layers, the total number of parameters in this network increased significantly. So we decided to increase the dropout rate to avoid overfitting.

In [44]:
```
model1 = Sequential()
model1.add(Conv2D(32, (5, 5), activation='relu',
input_shape=input_shape))
model1.add(Conv2D(32, (3, 3), activation='relu'))
model1.add(MaxPooling2D(pool_size=(2, 2)))
model1.add(Dropout(0.25))

model1.add(Conv2D(32, (3, 3), activation='relu'))
model1.add(Conv2D(32, (3, 3), activation='relu'))
model1.add(MaxPooling2D(pool_size=(2, 2)))
model1.add(Dropout(0.5))

model1.add(Conv2D(32, (3, 3), activation='relu'))
model1.add(Conv2D(32, (3, 3), activation='relu'))
model1.add(MaxPooling2D(pool_size=(2, 2)))
model1.add(Dropout(0.5))

model1.add(Flatten())
model1.add(Dense(64, activation='relu'))
model1.add(Dropout(0.5))
model1.add(Dense(7,activation='sigmoid'))

# prints out a summary of the model architecture
model1.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_22 (Conv2D)           (None, 150, 150, 32)      2432
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 148, 148, 32)      9248
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 74, 74, 32)        0
_________________________________________________________________
dropout_16 (Dropout)         (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 72, 72, 32)        9248
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 70, 70, 32)        9248
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 35, 35, 32)        0
_________________________________________________________________
dropout_17 (Dropout)         (None, 35, 35, 32)        0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 33, 33, 32)        9248
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 31, 31, 32)        9248
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 15, 15, 32)        0
_________________________________________________________________
dropout_18 (Dropout)         (None, 15, 15, 32)        0
_________________________________________________________________
flatten_8 (Flatten)          (None, 7200)              0
_________________________________________________________________
dense_14 (Dense)             (None, 64)                460864
_________________________________________________________________
dropout_19 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_15 (Dense)             (None, 7)                 455
=================================================================
Total params: 509,991.0
Trainable params: 509,991.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [45]:
```
sgd = SGD(lr=0.1, momentum=0.9)
model1.compile(loss='binary_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
history = model1.fit(train_X, train_Y,
                   batch_size=batch_size,
                   epochs=epochs,
                   verbose=1,
                   validation_data=(test_X, test_Y))
```

```
Train on 10000 samples, validate on 5000 samples
Epoch 1/10
10000/10000 [==============================] - 49s - loss: 3.7998 - ac
c: 0.7591 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 2/10
10000/10000 [==============================] - 49s - loss: 3.6759 - ac
c: 0.7716 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 3/10
10000/10000 [==============================] - 48s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 4/10
10000/10000 [==============================] - 48s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 5/10
10000/10000 [==============================] - 48s - loss: 3.6738 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 6/10
10000/10000 [==============================] - 48s - loss: 3.6738 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 7/10
10000/10000 [==============================] - 48s - loss: 3.6736 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 8/10
10000/10000 [==============================] - 48s - loss: 3.6738 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 9/10
10000/10000 [==============================] - 48s - loss: 3.6733 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
Epoch 10/10
10000/10000 [==============================] - 48s - loss: 3.6738 - ac
c: 0.7717 - val_loss: 3.6192 - val_acc: 0.7751
```

In [47]:
```
score = model1.evaluate(x_test, y_test, verbose=0)
score[1]
```

Out[47]: 0.76628572473526002

Summary

We divided the dataset into three parts. We have 10000 training samples and 5000 validation samples and 5000 test samples. The performance of each model is listed below.

	by-scratch model	**pre-tuned model 1**	**pre-tuned model 2**
Accuracy	0.766	0.707	0.766
Number of Parameters	5,176,199.0	2,539,495.0	509,991

We found that the by-scracth model and the pre-tuned model 2 achieved similar accuracy, and pre-tuned model 1 achieved slightly low accuracy on test data set. However, as we can see from the number of the parameters above, the total number of the parameters for the by-scratch model is 10 times greater than the pre-tuned model 2, when the accuracy of these two models are the same. So we believe that the pre-tuned model 2 is a better choice.

We also did other tuning for each models. Firstly, we did tried to tune the learning rate of the stachastic gradient descent process. However, we found that there is no siginificant difference between different learning rate. Next, we then tried to scaled the input data and found that there is no significant difference on accuracy, too.

Another idea & Future work

In this network setting, the response variable is the 7 cluster results. Each column of these 7 columns are binary column, indicating whether this movie is in this cluster of genres or not. The 7 clusters are not mutually exclusive. That is, a movie could be in cluster 1 and 7 at the same time. So the classification in this setting is the multi-label classification. Another idea in the modeling step is to convert the multi-label classification to multi-class classificaation problem. The basic idea is that each cluster represents some genres and each movie have some genres, we then calculate the proportion of this genre in each cluster and we decide the label of this movie with the cluster has the largest proportion. Then we transfer the multi-label problem to the multi-class problem. Then the output layer should only have one unit rather than 7 unit. And instead of the sigmoid unit. We choosed the softmax activation.

Also we would like to download more posters. Currently, we only used 10000 samples to train our model. We then want to use 50000 posters to train. Alos, we would like to do the data augmentation to reduce the bias.

In [ ]: