CS109b - Project - Team 14
Milestone 4: Deep Learning
April 18, 2017¶

Introduction

During earlier stages of this project, our team got acquainted with the movie domain and data from three sources: IMDb, TMDb, and MovieLens. After EDA, data acquisition, cleaning, and modeling with classic machine-learning approaches, we now tackle the challenge of deep learning in order to leverage movie posters. Our models included one from scratch and one pre-trained, so that we could compare performance and increase team learning. This report describes the models, reports on results, and discusses lessons learned.

From-scratch Network

The from-scratch network used in this project found a middle ground between the sample code used for teaching purposes and the complexity of VGG-16 (the well-known pre-trained network). The basic architecture of VGG-16 was used in a reduced form, consisting of the following sections in aggregate:

- 2D convolutions mixed with pooling, to extract basic features from the posters
- A flattening layer, to prepare the convolution results for the fully-connected layers
- Fully-connected layers, to extract higher-order features from the posters
- An output layer, to make probability-style predictions of genre classes

Pilot training was completed on a MacBook Pro (mid-2014). Final training was completed on AWS using a p2.xlarge instance of the EC2 service. Persistent storage was supported by the AWS S3 service.

In addition to the sections described above, dropout regularization was inserted in several layers to help with generalizing training results. Details of the network architecture are reproduced below from the Jupyter notebook used for model implementation.

The model parameters for the from-scratch model are described in the table below. Broadly speaking, these parameters are shared with the pre-trained model for the sake of team comparison (although network weights are obviously not shared).

From-scratch Network Architecture

Layer (type)	Output Shape	Param #
dropout_26 (Dropout)	(None, 128, 128, 3)	0
conv2d_31 (Conv2D)	(None, 124, 124, 64)	4864
conv2d_32 (Conv2D)	(None, 120, 120, 64)	102464
conv2d_33 (Conv2D)	(None, 116, 116, 64)	102464
max_pooling2d_11 (MaxPooling	(None, 58, 58, 64)	0
dropout_27 (Dropout)	(None, 58, 58, 64)	0
conv2d_34 (Conv2D)	(None, 56, 56, 128)	73856
conv2d_35 (Conv2D)	(None, 54, 54, 128)	147584
conv2d_36 (Conv2D)	(None, 52, 52, 128)	147584
max_pooling2d_12 (MaxPooling	(None, 26, 26, 128)	0
dropout_28 (Dropout)	(None, 26, 26, 128)	0
flatten_6 (Flatten)	(None, 86528)	0
dense_21 (Dense)	(None, 256)	22151424
dropout_29 (Dropout)	(None, 256)	0
dense_22 (Dense)	(None, 256)	65792
dropout_30 (Dropout)	(None, 256)	0
dense_23 (Dense)	(None, 256)	65792
dense_24 (Dense)	(None, 19)	4883

From-scratch model parameters

Parameter	Value	Description and Rationale
Epochs	Min 20 Max 150	With early stop after plateau of 20 epochs, to balance between accuracy and computation time.
Batch_size	256	The batch size was kept reasonably high to lower runtime.
Image Size	128 * 128 pixels RGB	This size was reduced from 244*244 pixels to reduce training time while preserving sufficient accuracy.
Optimizer	Stochastic Gradient Descent	Support for multi label classes (Adam was found not to impact metrics noticeably in casual testing)
Loss Function	Binary Cross Entropy	Support for multi-label classes
Learning Rate	Start: .1 to 10^-6	Decreases after plateau of 15 epochs to enable efficient early termination
Momentum	Start .9 to .99	Controlled with Nesterov momentum to focus on key gradient direction.
Sample Size	36,000 movies	The research MovieLens set of ID's used to download posters from TMDb (before upsampling minority classes)
Data Splits	60% Training 20% Validation 20% Testing	A traditional allocation that trains sufficiently while supporting two phases of evaluation
Number of Classes	19 Genres	Action, Adventure, Animation, Comedy, Crime, Documentary, Drama, Family, Fantasy, History, Horror, Music, Musical, Mystery, Romance, Sci-Fi, Thriller, War & Western (a unified and streamlined list from IMDb and TMDb)
Up Sampled Classes	8 Genres	Several minority classes were upsampled with an additional augmentation, as determined by EDA in M1: Animation, History, Music, Musical, History, Sci-Fi, War & Western
Class Weights	Vector of 19 floats	Relative weightings to account for minority classes

Performance

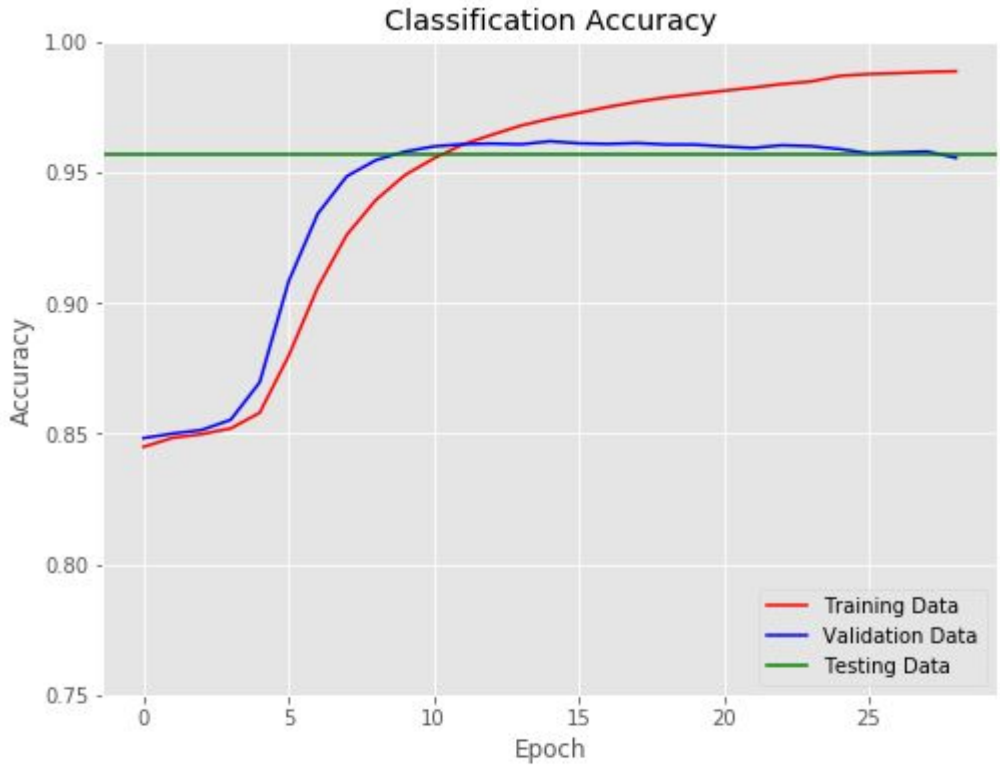
Because of the complexity of deep learning, a number of performance metrics were used to evaluate different dimensions of the model. These metrics included accuracy, precision, recall, ‘top-class’, Hamming distance, and ROC curve. We also tabulated the number of predicted hits per genre, to ensure that minority classes were represented . By paying attention to the shifting overview of these metrics during model tuning, we were able to get some sense of the most promising avenues to pursue. (Similar metrics were used for both from-scratch and pre-trained models for comparison purposes.)

An important caveat is that the results reported here are in progress. The most promising approach resulted in inflated metrics, on account of sub-optimal global upsampling and training/validation/testing partitioning. This approach will be fixed at our earliest opportunity. In the meantime, all results reported here will be reduced by 7%, in order to align the reported accuracy with that of the null model. In other words, the results reported here are “too good to be true”, but they represent a promising direction for model evolution.

In addition to traditional metrics, we created an informal metric called “top class”. This metric represents the success in finding the highest-scoring genre for a given movie among that movie’s actual genre tags. Anecdotally, a Netflix user often seeks a movie with a particular genre for an evening’s viewing - classification is more important for such a single genre, rather than for a full set of genre tags. For this reason, we tend to think that precision has greater real-world utility than recall for movie genre prediction.

Metric (on Testing Set)	**Value**
Accuracy	0.89
Precision	0.69
Recall	0.68
“Top Class”	0.78
Hamming Loss	0.11

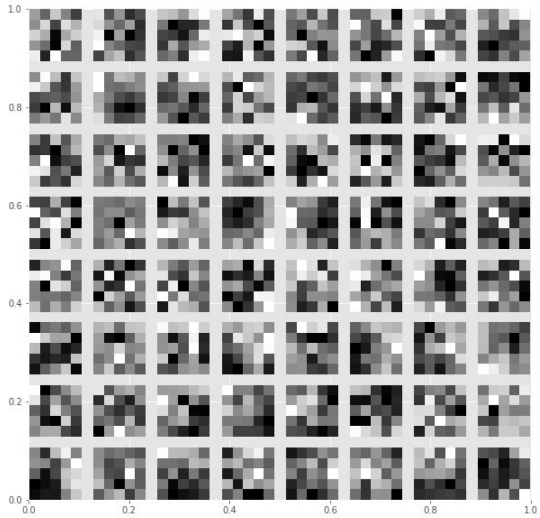
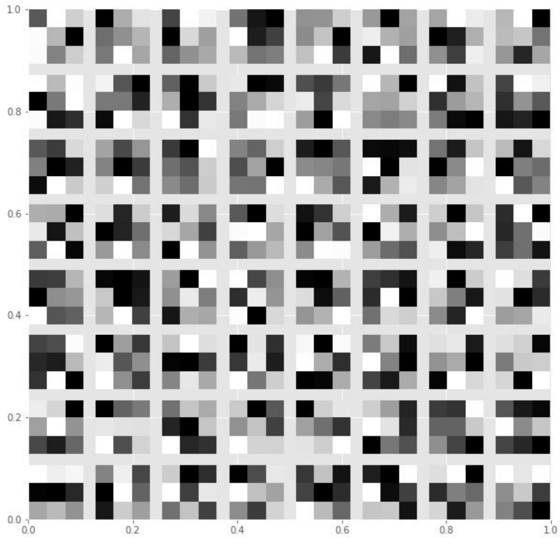
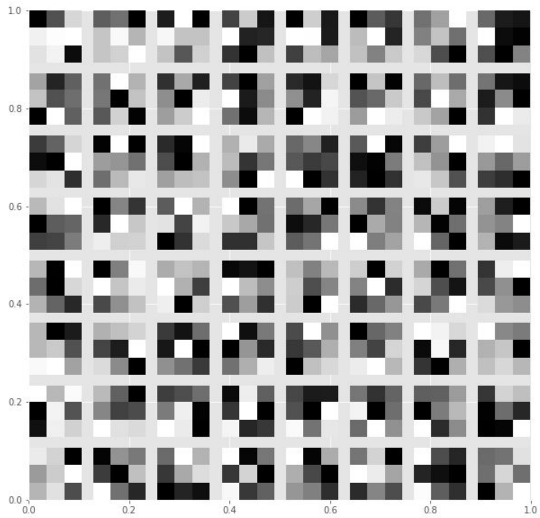
Training Process



Features Learned

Feature learning by a neural network can be visualized through the weights of network units. The outputs from convolution layers can be visualized at various architectural levels, though the fully-connected layers unfortunately lack direct ties to the original visual form of the images. Below we show sample filter weights as images for convolutions 1, 4, and 6 (the last).

In examining the images below, the learning path is clear. The samples of the first convolution show perhaps fine-grained features. Determining the nature of the features is difficult, but one gets an impression of edges and combinations of various sorts. The samples of the fourth convolution show clearer patterns in the tiles. It appears that the network is learning edges, L-shapes, parallel lines, and so on. Finally, the samples of the sixth convolution show the clearest patterns in this network. Apparent features include the ones previously mentioned, as well as enclosed shapes. This improvement in pattern clarity between layers 1 and 8 is striking. The clarity is a reflection of the network learning process.

1st Convolution	4th Convolution
	
6th Convolution	
	

Pre-Trained Network

As our base model, we selected the Keras implementation of the VGG16 model from the Visual Geometry Group of the Department of Engineering Science, at Oxford University. The VGG model has the advantages of being well trained, widely used and easy to conceptualise. By re-training an existing model, we were able to achieve transfer learning.

As the base model had been pre-trained against millions of images, we were able retrain the model to learn more quickly than a model from scratch. The predictions are also far more accurate on a relatively small sample of 36,000 movies.

The weights for the base model had been pretrained against ImageNet. The top three layers of the VGG base model containing 1,000 features was removed. New top layers were then added to support our 19 multi-label genres classification. The base layers were flagged as non- trainable before training our top layers.

VGG uses padding with convolutional layers, in order to preserve the spatial size of original imagery, thereby allowing greater depth.

The original VGG model uses softmax for prediction layer. As our genre model requires a multi-label response, we selected binary cross-entropy instead.

Our initial learning rate for the from-scratch model was relatively aggressive at 0.1. More conservatively, we lowered the initial learning rate on the re-trained model while “warming it up”. The base parameters had already been tuned. A learning rate that was too high might have eroded the learning previously gained through pre-training. Both models lowered the learning rate after 15 sequential epochs with no improvement, down to a minimum of 10^{-6}.

Pre-Trained Base Architecture

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	(None, 128, 128, 3)	0
block1_conv1 (Conv2D)	(None, 128, 128, 64)	1792
block1_conv2 (Conv2D)	(None, 128, 128, 64)	36928
block1_pool (MaxPooling2D)	(None, 64, 64, 64)	0
block2_conv1 (Conv2D)	(None, 64, 64, 128)	73856
block2_conv2 (Conv2D)	(None, 64, 64, 128)	147584
block2_pool (MaxPooling2D)	(None, 32, 32, 128)	0
block3_conv1 (Conv2D)	(None, 32, 32, 256)	295168
block3_conv2 (Conv2D)	(None, 32, 32, 256)	590080
block3_conv3 (Conv2D)	(None, 32, 32, 256)	590080
block3_pool (MaxPooling2D)	(None, 16, 16, 256)	0
block4_conv1 (Conv2D)	(None, 16, 16, 512)	1180160
block4_conv2 (Conv2D)	(None, 16, 16, 512)	2359808
block4_conv3 (Conv2D)	(None, 16, 16, 512)	2359808
block4_pool (MaxPooling2D)	(None, 8, 8, 512)	0
block5_conv1 (Conv2D)	(None, 8, 8, 512)	2359808
block5_conv2 (Conv2D)	(None, 8, 8, 512)	2359808
block5_conv3 (Conv2D)	(None, 8, 8, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

Pre-Trained Top Architecture (Trainable)

flatten_12 (Flatten)	(None, 8192)	0
dense_36 (Dense)	(None, 4096)	33558528
dropout_40 (Dropout)	(None, 4096)	0
dense_37 (Dense)	(None, 4096)	16781312
dropout_41 (Dropout)	(None, 4096)	0
dense_38 (Dense)	(None, 19)	77843

Pre-trained Model Parameters

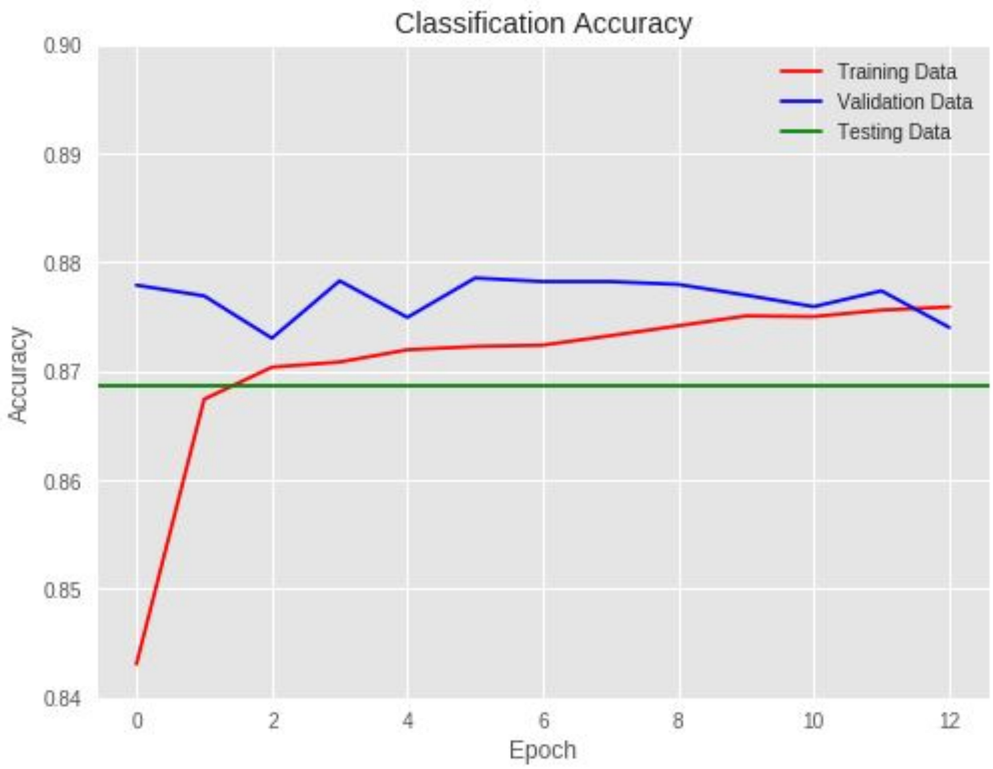
Parameter	**Value**	**Description and Rationale**
Epochs	Max 150	With early stop after plateau of 20 epochs, to balance between accuracy and computation time.
Batch_size	256	The batch size was kept reasonably high to lower runtime.
Image Size	128 * 128 pixels	This size was reduced from 244*244 pixels to reduce training time while preserving sufficient accuracy.
Optimizer	Stochastic Gradient Descent	Support for multi label classes (Adam was found not to impact metrics noticeably in casual testing)
Loss Function	Binary Cross Entropy	Support for multi-label classes
Learning Rate	Scratch: 0.1-10-6 Retrain: .01-10^-6	Decreases after plateau of 15 epochs to enable efficient early termination
Momentum	Start .9 to .99	Controlled with Nesterov momentum to focus on key gradient direction.
Sample Size	36,000 movies	The research MovieLens set of ID's used to download posters from TMDb (before upsampling minority classes)
Data Splits	60% Training 20% Validation 20% Testing	A traditional allocation that trains sufficiently while supporting two phases of evaluation
Number of Classes	19 Genres	Action, Adventure, Animation, Comedy, Crime, Documentary, Drama, Family, Fantasy, History, Horror, Music, Musical, Mystery, Romance, Sci-Fi, Thriller, War & Western (a unified and streamlined list from IMDb and TMDb)
Up Sampled Classes	8 Genres	Several minority classes were upsampled with an additional augmentation, as determined by EDA in M1: Animation, History, Music, Musical, History, Sci-Fi, War & Western
Class Weights	Vector of 19 floats	Relative weightings to account for minority classes

Performance

When run against the full 36,000 movies, the model produced superior results. We submitted the notebook with a smaller sample of 5,000 and an earlier stop, in order to produce useful visuals. Both sets of modeling results are show below for convenience.

Metric (on Testing Set)	**5K sample**	**37K sample**
Accuracy	0.87	0.88
Precision	0.13	0.45
Recall	0.17	0.27
“Top Class”	0.54	0.56
Hamming Loss	0.15	0.12

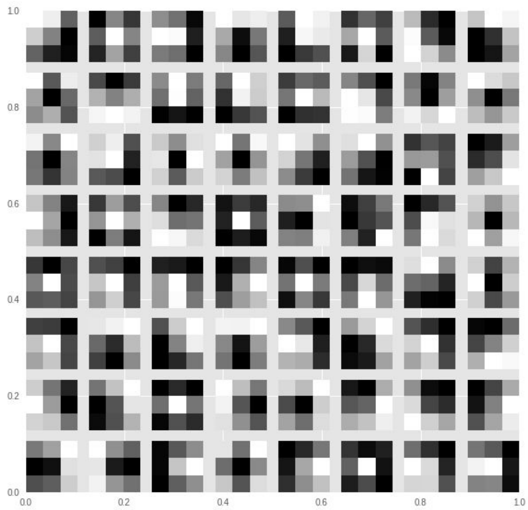
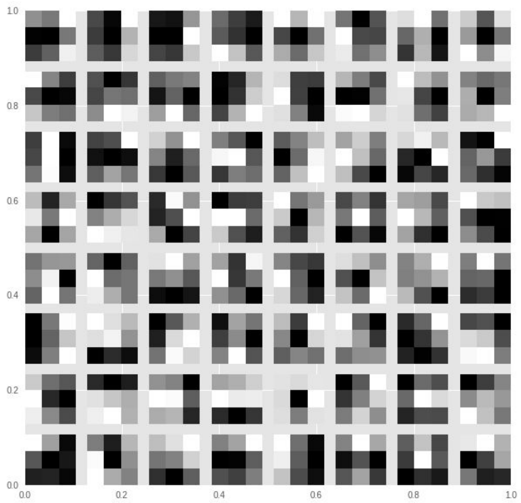
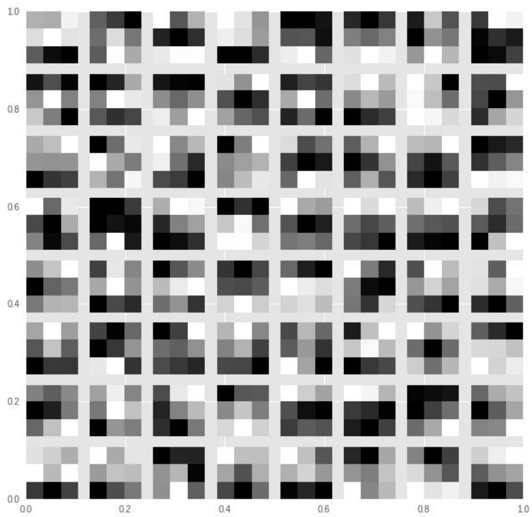
Training Process



Features Learned

Feature learning by a neural network can be visualized through the weights of network units. The outputs from convolution layers can be visualized at various architectural levels, though the fully-connected layers unfortunately lack direct ties to the original visual form of the images. Below we show sample filter weights as images for convolutions 1, 6th, and 13 (the last).

In examining the images below, the learning path is visible, though not dramatic. The samples from all three convolutions appear to show edges, L-shapes, parallel lines, enclosed shapes, and so on. The clarity of the first convolution's visuals are striking. In contrast to the fine-grained patterns shown by the first layer of the from-scratch network, the patterns shown by the first layer below are clear. It seems that the larger input volume and the longer training time of the pre-trained model allowed it to learn features more relatively clearly near the beginning of the network.

1st Convolution	**6th Convolution**
	
13th Convolution	
	

Results

The pretrained model showed a noticeable improvement in predicting minority classes, relative to the from-scratch model. Even using upsampling and class weights, the from-scratch model was challenged by minority genres.

The validation accuracy curve of the pre-trained model was initially dramatically steep. Within 3 epochs, values had reached 87% before the climb became shallower. The validation accuracy curve of the from-scratch model increased at a slower rate, taking more epochs to reach the same level.

The from-scratch model showed overfitting, with validation scores declining while training scores still increased. Applying dropout regularisation significantly reduced the overfitting problem. Though comparison would have been instructive, there was unfortunately no opportunity to test without regularisation on the pre-trained model.

Given more opportunity, we would have invested more time in optimising the L2 penalty parameters, in addition to leveraging dropout regularization. Given more time, we would also investigate ensemble models to pool trained neural networks (as well as perhaps integrating random-forest and QDA models from Milestone 3).

Team development interwove the from-scratch and the pre-trained models to some extent. Mastering a from-scratch model was essential for acquiring the skill necessary to work with a pre-trained model. At the same time, the architecture of the pre-trained model informed directions for pursuing an effective from-scratch model.

Exploratory Idea

To explore ideas beyond the original from-scratch and VGG architectures, we pursued two ideas.

First, as mentioned, we upsampled minority classes in order to account for an unbalanced data set. Our initial upweighting was relatively rough, using a constant factor for all minority classes (as determined through EDA). As a next step, we would apply upweighting in inverse proportion to class weights, in order to achieve more nuance.

Second, we explored image augmentation in a preliminary way. Our approach was simply to flip minority images about the vertical axis. While coordinating development effort with the upsampling task, this augmentation added diversity to the image set. Our metrics didn't detect clear improvement from this augmentation, so deeper transformations are likely needed. The most obvious next step would be applying multiple crops to some subset of posters.

Conclusion

Deep learning proved powerful, but not a panacea. Both models learned features from movie posters sufficient for predicting genres with fair precision, but only minor improvements in accuracy over the null model. The pre-trained model had a significant advantage in performance on account of vast training input and time, as mentioned previously.

It is surprising however that even a untuned model performed relatively well. Making gains in accuracy was challenging, though. Through training, we saw improvements on the order of 3% in accuracy. We found that fine-tuning the learning rate and momentum had the most significant impact on our model performance, among available hyperparameters. (Naturally, choices for batch size, image size, and number of epochs had a pragmatic impact as well.) Given more computational time and resources, we would liked to have explored a greater range of parameters, particularly those for dropout regularization.

Relative to classical machine learning, the sheer complexity of deep learning can be overwhelming. We would also like to note the challenges of managing the model training process. At the same, the power to infer a model from unstructured data is impressive. Working with classic and deep-learning paradigms during this project has thrown the contrasts into sharp relief. During Milestone 5 (the final report), we will explore these ideas more deeply, while wrapping up the project.

References

Very Deep Convolutional Networks for Large-Scale Image Recognition
K. Simonyan, A. Zisserman
arXiv:1409.1556

CS 109B - Course Project - Team 14

Deep Learning - Posters

1. Prepare AWS

Install packages

In [143]:
```
# import pip
# def install(package):
#     pip.main(['install', package])
# install('boto')
# install('opencv-python')
# install('h5py')
# # # # install('scikit-image')
```

Retrieve images and data from Amazon's S3

In [144]:
```
# import boto
# import boto.s3.connection
# access_key = 'AKIAIULN726KJGOR6YJQ'
# secret_key = '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'

# conn = boto.connect_s3(
#         aws_access_key_id = access_key,
#         aws_secret_access_key = secret_key,
#         host = 's3.amazonaws.com',
#         #is_secure = False,               # uncomment if you are not
using ssl
#         calling_format = boto.s3.connection.OrdinaryCallingFormat(),
#         )

# bucket = conn.get_bucket('cs109b.dkm.posters.250sq')
# key = bucket.get_key('Posters_250_250.zip')
# key.get_contents_to_filename('Posters_250_250.zip')

# bucket = conn.get_bucket('cs109b.dkm.imdb.data')
# key = bucket.get_key('imdb_movies_trim.csv')
# key.get_contents_to_filename('imdb_movies_trim.csv')

# import os, zipfile
# home_dir = os.path.expanduser("~")
# os.chdir(home_dir)
# my_zipfile = zipfile.ZipFile('Posters_250_250.zip')
# my_zipfile.extractall()
```

2. Load Data

Load packages and set display options

In [145]:
```
import numpy as np
import pandas as pd
import os

from __future__ import print_function
import cv2
from scipy import ndimage, misc
from sklearn.cross_validation import train_test_split as sk_split
from sklearn.metrics import hamming_loss

import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, MaxPooling2D, Flat
ten, Dropout, ZeroPadding2D
from keras.optimizers import SGD, Adam
from keras import backend as K
from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import CSVLogger
from keras.callbacks import EarlyStopping
from keras.callbacks import ModelCheckpoint

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

from IPython.display import display, HTML, Markdown
%matplotlib inline
plt.style.use('ggplot')
def printmd(string):
    display(Markdown(string))
```

Set global constants

In [146]:
```
# input image dimensions
img_rows, img_cols = 128, 128

# smaller batch size means noisier gradient, but more updates per epoc
h
#Dom: keep high for laptop, lower for AWS
batch_size = 128

# 18 genres in our data set
num_classes = 19

# number of iterations over the complete training data
epochs = 150

#(80,75) equivilant to test 20 , train 60, validate 20
train_percent = 80
validation_percent = 75   #note: percentage of training not the entire
set
```

Load and clean Y data

In [147]:
```
movie_fields = ['imdb_id',
                'Action','Adventure','Animation','Comedy','Crime','Doc
umentary','Drama',
                'Family','Fantasy','History','Horror','Music', 'Musica
l', 'Mystery',
                'Romance','Sci-Fi','Thriller','War','Western']
genres = movie_fields[1:]
small_genres = ['Western', 'Musical', 'Music', 'History', 'Animation',
'War', 'History', "Sci-Fi"]

#ignore ,'Game-Show','News','Reality-TV', 'Biography','Adult','Film-No
ir'
df_im_0 = pd.read_csv( './imdb_movies_trim.csv',
                    encoding = 'utf-8',
                    usecols = movie_fields)

# filter out no-genre movies
df_im = df_im_0[df_im_0.iloc[:, 1:].any(axis = 1)]

#reorder the columns
df_im = df_im[movie_fields]

#sort by movie id
df_im = df_im.sort_values(by = 'imdb_id')

# prep for augmentation
small_genres_idx = []
for i in xrange(len(small_genres)):
    small_genres_idx.append(movie_fields.index(small_genres[i]))
```

3. Load and Prepare Images

Load and resize images

In [148]:
```
images = []
imdb_ids = []
imdb_id_image_not_found = []

num_images = df_im.shape[0]
#num_images = 100
for i in xrange(num_images):

    # retrieve movie object from imdb
    imdb_id = int(df_im.iloc[i,0])
    filepath = './Posters_250_250/' + str(imdb_id).zfill(7) + '.jpg'

    # image found, so load, resize, and collect it
    if os.path.isfile(filepath) :

        image = cv2.imread(filepath)
        image = cv2.resize(image,(img_rows, img_cols))
        # image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # needed if l
oading images with scikit-image
        image = np.array(image).reshape((3, img_rows, img_cols))

        # add image and movie id to list
        images.append([image])
        imdb_ids.append(imdb_id)

        # image augmentation on minority genres
        if sum(df_im.iloc[i, small_genres_idx]) >= 1 :
            for i in range(10):
                image_flip = cv2.flip(image, 1) # vertical-axis flip
                images.append([image_flip])
                imdb_ids.append(imdb_id)

    else:
        # add id to failure list
        imdb_id_image_not_found.append(imdb_id)

# collect the image list into an array
x = np.array(images)
x = x[:, 0, :, :]
images = None
df_imdb_ids = pd.DataFrame({'id': imdb_ids })

print()
print ('images loaded', len(imdb_ids))
print ('images failed', len(imdb_id_image_not_found))
```

```
images loaded 129059
images failed 8
```

Split into test and train data sets

In [149]:
```
#Build y taking care to align rows with x
#merge ensures that if images were not found they do not cause a mis-a
slignment of x and y
y = pd.merge(left = df_imdb_ids , right = df_im , left_on = 'id', righ
t_on = 'imdb_id')

#drop first two columns (id and imdb_id) leaving just the 18 encoded g
enres
y = y.ix[:,2:]

#split y into test and train
y_train, y_test = sk_split(y, train_size = train_percent / 100.0)

#take out a portion for validation from the training set
y_train, y_valid = sk_split(y_train, train_size = validation_percent /
100.0)

#use post split indicies from y to split x into the same groups
x_test   = x[y_test.index]
x_train  = x[y_train.index]
x_valid  = x[y_valid.index]

#convert label dataframes to numpy arrays
y_train = y_train.values
y_test  = y_test.values
y_valid = y_valid.values

print()
print('x_train shape:', x_train.shape)
print('x_valid shape:', x_valid.shape)
print('x_test shape:' , x_test.shape)
print()
print('y_train shape:', y_train.shape)
print('y_valid shape:', y_valid.shape)
print('y_test shape:' , y_test.shape)


#free up memory
x = y = None
```

```
x_train shape: (77435, 3, 128, 128)
x_valid shape: (25812, 3, 128, 128)
x_test shape: (25812, 3, 128, 128)

y_train shape: (77435, 19)
y_valid shape: (25812, 19)
y_test shape: (25812, 19)
```

Using tensorflow as backend so expect order of array as is (n = sample size, img_rows, img_cols, colour = 3)

In [150]:
```
if K.image_data_format() ==  'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 3, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 3, img_rows, img_cols)
    x_valid = x_valid.reshape(x_valid.shape[0], 3, img_rows, img_cols)
    input_shape = (3, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 3)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 3)
    x_valid = x_valid.reshape(x_valid.shape[0], img_rows, img_cols, 3)
    input_shape = (img_rows, img_cols, 3)
```

Centre and normalise images

In [151]:
```
# normalize image values to [0,1] (Keras sample code doesn't center)
x_train = x_train.astype('float32')
x_valid = x_valid.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_valid /= 255
x_test /= 255

print()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (77435, 128, 128, 3)
77435 train samples
25812 test samples
```

4. Train Model

Create dictionary of class weights

In [152]:
```
# create dictionary of class weights
class_count = y_train.sum(axis = 0) + 1
class_weight = y_train.shape[0] / (num_classes * class_count)
class_weight_dict = dict(zip(np.arange(num_classes), class_weight))
```

Create model shell

In [153]:
```
# create an empty network model
model = Sequential()
model.add(Dropout(0.2, input_shape = input_shape))

# --- input layer ---
model.add(Conv2D(64, kernel_size = (5, 5), activation = 'relu'))
model.add(Conv2D(64, kernel_size = (5, 5), activation = 'relu'))
model.add(Conv2D(64, kernel_size = (5, 5), activation = 'relu'))

# --- max pool ---
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.2))

# --- next layer ---
model.add(Conv2D(128, kernel_size = (3, 3), activation = 'relu'))
model.add(Conv2D(128, kernel_size = (3, 3), activation = 'relu'))
model.add(Conv2D(128, kernel_size = (3, 3), activation = 'relu'))

# --- max pool ---
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.2))

# flatten for fully connected classification layer
model.add(Flatten())

# --- fully connected layer ---
model.add(Dense(256, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(256, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(256, activation = 'relu'))

# --- classification ---
model.add(Dense(num_classes, activation = 'sigmoid'))

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dropout_26 (Dropout)         (None, 128, 128, 3)       0         
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 124, 124, 64)      4864      
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 120, 120, 64)      102464    
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 116, 116, 64)      102464    
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 58, 58, 64)        0         
_________________________________________________________________
dropout_27 (Dropout)         (None, 58, 58, 64)        0         
_________________________________________________________________
conv2d_34 (Conv2D)           (None, 56, 56, 128)       73856     
_________________________________________________________________
conv2d_35 (Conv2D)           (None, 54, 54, 128)       147584    
_________________________________________________________________
conv2d_36 (Conv2D)           (None, 52, 52, 128)       147584    
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 26, 26, 128)       0         
_________________________________________________________________
dropout_28 (Dropout)         (None, 26, 26, 128)       0         
_________________________________________________________________
flatten_6 (Flatten)          (None, 86528)             0         
_________________________________________________________________
dense_21 (Dense)             (None, 256)               22151424  
_________________________________________________________________
dropout_29 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_22 (Dense)             (None, 256)               65792     
_________________________________________________________________
dropout_30 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_23 (Dense)             (None, 256)               65792     
_________________________________________________________________
dense_24 (Dense)             (None, 19)                4883      
=================================================================
Total params: 22,866,707.0
Trainable params: 22,866,707.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile model

In [154]:
```
# use basic categorical crossentropy with stochastic gradient decent
# evaluate model in terms of accuracy
# lr : learning rate (start from 0.1 and move as low as 10^-6)
# momentum: start with .5 and tune to .9

sgd = SGD(lr = 0.1, momentum = 0.9, nesterov = True)
# adam = Adam(lr = 0.001, beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-0
8, decay = 0.0)
model.compile(loss = 'binary_crossentropy',
              optimizer = sgd,
              metrics = ['accuracy'])
```

Train model and score with validation data

In [155]:
```
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.1,
                              patience = 10, min_lr = 0.000001)

early_stop = EarlyStopping (monitor = 'val_loss', min_delta = .0001, p
atience = 15, verbose = 0, mode = 'auto')

csv_logger = CSVLogger('training.log')

model_checkpoint = ModelCheckpoint(filepath = "./best_model.hdf5", mon
itor = 'val_loss',
                                   verbose = 0, save_best_only = True,
                                   save_weights_only = False, mode = '
auto', period = 1)

history = model.fit(x_train, y_train,
                    batch_size = batch_size,
                    epochs = epochs,
                    callbacks = [reduce_lr, early_stop, csv_logger, mo
del_checkpoint],
                    verbose = 1,
                    validation_data = (x_valid, y_valid),
                    class_weight = class_weight_dict)
```

```
Train on 77435 samples, validate on 25812 samples
Epoch 1/150
77435/77435 [==============================] - 1052s - loss: 0.1248
- acc: 0.8449 - val_loss: 0.3953 - val_acc: 0.8483
Epoch 2/150
77435/77435 [==============================] - 1040s - loss: 0.1178
- acc: 0.8484 - val_loss: 0.3835 - val_acc: 0.8500
Epoch 3/150
77435/77435 [==============================] - 1037s - loss: 0.1139
- acc: 0.8497 - val_loss: 0.3704 - val_acc: 0.8513
```

```
Epoch 4/150
77435/77435 [==============================] - 1033s - loss: 0.1100
- acc: 0.8519 - val_loss: 0.3575 - val_acc: 0.8552
Epoch 5/150
77435/77435 [==============================] - 1035s - loss: 0.1006
- acc: 0.8579 - val_loss: 0.3186 - val_acc: 0.8696
Epoch 6/150
77435/77435 [==============================] - 1036s - loss: 0.0807
- acc: 0.8798 - val_loss: 0.2359 - val_acc: 0.9083
Epoch 7/150
77435/77435 [==============================] - 1035s - loss: 0.0620
- acc: 0.9059 - val_loss: 0.1797 - val_acc: 0.9340
Epoch 8/150
77435/77435 [==============================] - 1036s - loss: 0.0489
- acc: 0.9260 - val_loss: 0.1450 - val_acc: 0.9484
Epoch 9/150
77435/77435 [==============================] - 1036s - loss: 0.0400
- acc: 0.9393 - val_loss: 0.1271 - val_acc: 0.9545
Epoch 10/150
77435/77435 [==============================] - 1039s - loss: 0.0335
- acc: 0.9488 - val_loss: 0.1190 - val_acc: 0.9578
Epoch 11/150
77435/77435 [==============================] - 1036s - loss: 0.0289
- acc: 0.9552 - val_loss: 0.1125 - val_acc: 0.9598
Epoch 12/150
77435/77435 [==============================] - 1036s - loss: 0.0254
- acc: 0.9605 - val_loss: 0.1106 - val_acc: 0.9607
Epoch 13/150
77435/77435 [==============================] - 1035s - loss: 0.0228
- acc: 0.9643 - val_loss: 0.1101 - val_acc: 0.9609
Epoch 14/150
77435/77435 [==============================] - 1034s - loss: 0.0204
- acc: 0.9678 - val_loss: 0.1126 - val_acc: 0.9606
Epoch 15/150
77435/77435 [==============================] - 1039s - loss: 0.0187
- acc: 0.9704 - val_loss: 0.1103 - val_acc: 0.9618
Epoch 16/150
77435/77435 [==============================] - 1039s - loss: 0.0172
- acc: 0.9726 - val_loss: 0.1138 - val_acc: 0.9610
Epoch 17/150
77435/77435 [==============================] - 1039s - loss: 0.0158
- acc: 0.9749 - val_loss: 0.1156 - val_acc: 0.9607
Epoch 18/150
77435/77435 [==============================] - 1039s - loss: 0.0145
- acc: 0.9769 - val_loss: 0.1199 - val_acc: 0.9611
Epoch 19/150
77435/77435 [==============================] - 1036s - loss: 0.0135
- acc: 0.9785 - val_loss: 0.1196 - val_acc: 0.9605
Epoch 20/150
77435/77435 [==============================] - 1033s - loss: 0.0127
```

```
- acc: 0.9799 - val_loss: 0.1261 - val_acc: 0.9605
Epoch 21/150
77435/77435 [==============================] - 1034s - loss: 0.0119
- acc: 0.9811 - val_loss: 0.1307 - val_acc: 0.9598
Epoch 22/150
77435/77435 [==============================] - 1043s - loss: 0.0112
- acc: 0.9823 - val_loss: 0.1306 - val_acc: 0.9592
Epoch 23/150
77435/77435 [==============================] - 1049s - loss: 0.0104
- acc: 0.9837 - val_loss: 0.1250 - val_acc: 0.9603
Epoch 24/150
77435/77435 [==============================] - 1048s - loss: 0.0098
- acc: 0.9847 - val_loss: 0.1327 - val_acc: 0.9599
Epoch 25/150
77435/77435 [==============================] - 1049s - loss: 0.0085
- acc: 0.9868 - val_loss: 0.1422 - val_acc: 0.9588
Epoch 26/150
77435/77435 [==============================] - 1048s - loss: 0.0080
- acc: 0.9875 - val_loss: 0.1499 - val_acc: 0.9572
Epoch 27/150
77435/77435 [==============================] - 1048s - loss: 0.0078
- acc: 0.9878 - val_loss: 0.1510 - val_acc: 0.9575
Epoch 28/150
77435/77435 [==============================] - 1048s - loss: 0.0074
- acc: 0.9883 - val_loss: 0.1500 - val_acc: 0.9579
Epoch 29/150
77435/77435 [==============================] - 1048s - loss: 0.0074
- acc: 0.9885 - val_loss: 0.1596 - val_acc: 0.9554
```

5. Score Model

In [156]:
```
### prepare metrics calculations

avg_tags_per_movie = y_train.sum(axis = 1).mean()
y_test_predict_proba = model.predict(x_test, verbose = 0)

threshold_accus = np.zeros(101)
tp_rate = np.zeros(101)
fp_rate = np.zeros(101)
hamming_losses = np.zeros(101)

for i in range(101):
    y_test_predict = y_test_predict_proba > i / 100.0
    threshold_accus[i] = (y_test == y_test_predict).sum() / float((num
_classes * y_test.shape[0]))
    tp_rate[i] = ((y_test == 1) & (y_test_predict == 1)).sum() / float
((num_classes * y_test.shape[0]))
    fp_rate[i] = ((y_test == 0) & (y_test_predict == 1)).sum() / float
((num_classes * y_test.shape[0]))
    hamming_losses[i] = hamming_loss(y_test, y_test_predict)

opt_hamming_threshold = hamming_losses.argmin() / 100.0
```

In [157]:
```
# score after training
score = model.evaluate(x_test, y_test, verbose = 0)
print('Test loss:', round(score[0], 4))
print('Test accuracy:', round(score[1], 4))
```

```
Test loss: 0.1555
Test accuracy: 0.9565
```

In [158]:
```
y_test_predict = y_test_predict_proba > .5
test_accu = (y_test == y_test_predict).sum() / float((num_classes * y_
test.shape[0]))
print('Test accuracy using probability threshold of 0.50 :', round(tes
t_accu, 4))
```

```
Test accuracy using probability threshold of 0.50 : 0.9565
```

In [159]:
```
y_test_predict = y_test_predict_proba > opt_hamming_threshold
test_accu = (y_test == y_test_predict).sum() / float((num_classes * y_
test.shape[0]))
print('Test accuracy using probability threshold of', round(opt_hammin
g_threshold, 2), ':', round(test_accu, 4))
```

```
Test accuracy using probability threshold of 0.73 : 0.9585
```

In [160]:
```
y_test_predict = np.zeros([y_test.shape[0], num_classes])
test_accu = (y_test == y_test_predict).sum() / float((num_classes * y_
test.shape[0]))
print('Test accuracy using classifier that always predicts 0 :', round
(test_accu, 4))
```

```
Test accuracy using classifier that always predicts 0 : 0.8482
```

In [161]:
```
y_test_predict = y_test_predict_proba > opt_hamming_threshold
temp = pd.DataFrame(y_test_predict.sum(axis = 0), columns = ['Count'])
temp.index = genres
temp
```

Out[161]:

	Count
Action	3932
Adventure	3509
Animation	3037
Comedy	4879
Crime	1067
Documentary	1920
Drama	11378
Family	2344
Fantasy	2083
History	2968
Horror	2248
Music	2496
Musical	1838
Mystery	918
Romance	3322
Sci-Fi	5369
Thriller	3172
War	3227
Western	1953

In [162]:
```
from sklearn.metrics import precision_score as ps

y_test_predict = y_test_predict_proba > opt_hamming_threshold
avg_precision = ps(y_test, y_test_predict, average = 'samples')
print('Average test precision (samples):', round(avg_precision, 4))
```

```
Average test precision (samples): 0.8017
```

In [163]:
```
from sklearn.metrics import recall_score as rs

y_test_predict = y_test_predict_proba > opt_hamming_threshold
avg_recall = rs(y_test, y_test_predict, average = 'samples')
print('Average test recall (samples):', round(avg_recall, 4))
```

```
Average test recall (samples): 0.7413

/usr/lib/python2.7/dist-packages/sklearn/metrics/classification.py:1
076: UndefinedMetricWarning: Recall is ill-defined and being set to
0.0 in samples with no true labels.
  'recall', 'true', average, warn_for)
```

In [164]:
```
y_test_predict_top_class = np.argmax(y_test_predict_proba, axis = 1)
test_top_class_accu_list = [(y_test[i, y_test_predict_top_class[i]] ==
1) for i in range(y_test.shape[0])]
test_top_class_accu = np.sum(test_top_class_accu_list) / float(y_test.
shape[0])
print('Test accuracy matching top class:', round(test_top_class_accu,
4))
```

```
Test accuracy matching top class: 0.8403
```

In [165]:
```
y_test_predict = y_test_predict_proba > 0.33
test_hamming_loss = hamming_loss(y_test, y_test_predict)
print('Test Hamming loss:', round(test_hamming_loss, 4))
```

```
Test Hamming loss: 0.0478
```

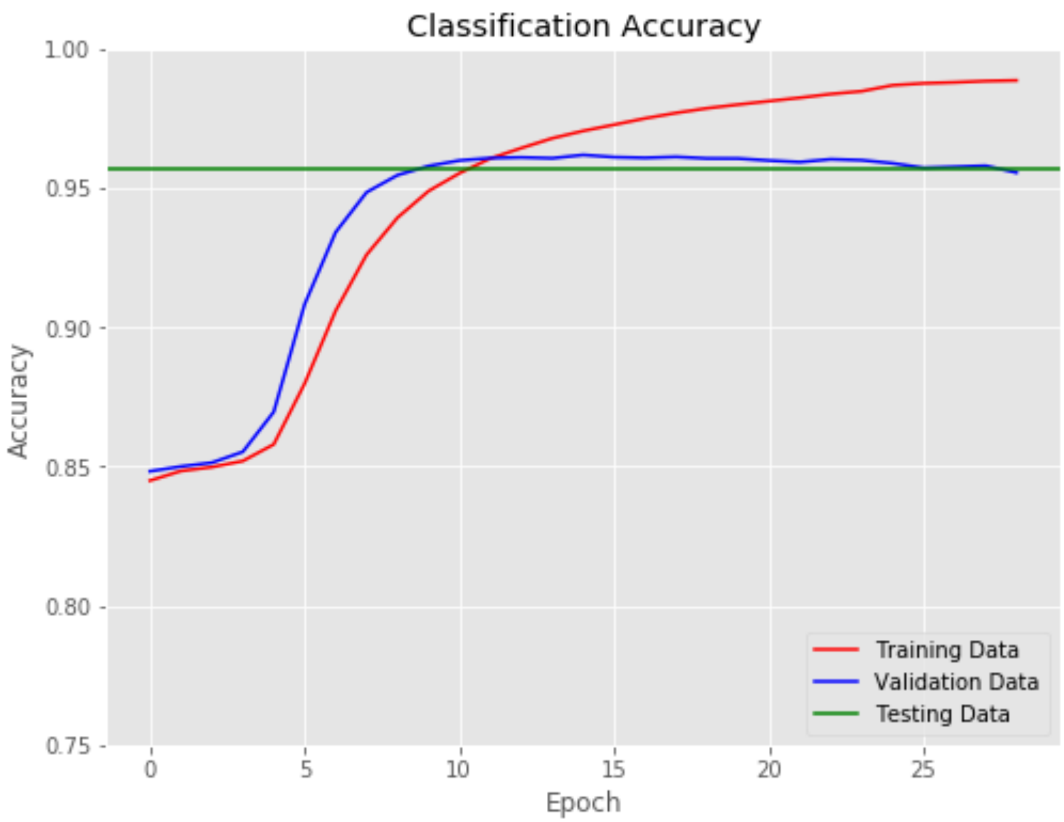
6. Visualization

Training process

In [170]:
```
### plot training process
print()
fig = plt.figure(figsize = (8, 6))
ax = fig.add_subplot(1, 1, 1)

ax.plot(history.history['acc'], color = 'red', label = 'Training Data'
)
ax.plot(history.history['val_acc'], color = 'blue', label = 'Validatio
n Data')
ax.axhline(y = score[1], color = 'green', label = 'Testing Data')
ax.set_ylim(0.75, 1)
ax.legend(loc = 'lower right')
ax.set_xlabel("Epoch")
ax.set_ylabel("Accuracy")
ax.set_title("Classification Accuracy")

plt.show()
```

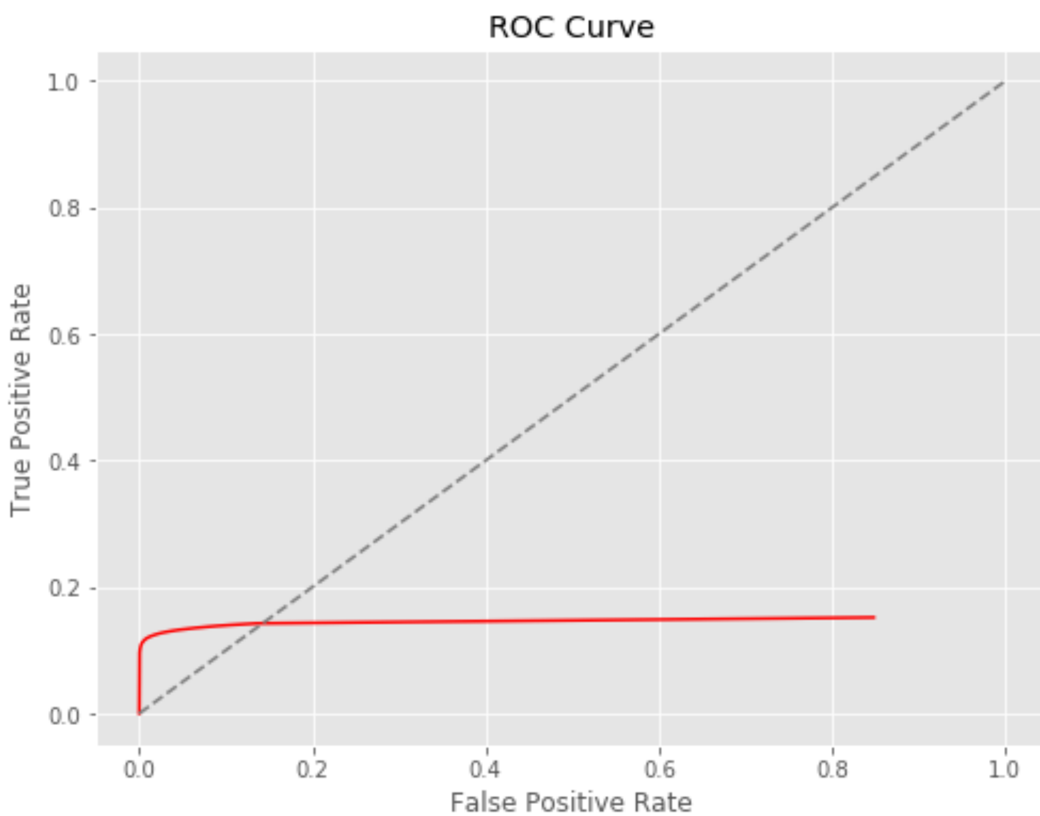


ROC Curve

In [167]:
```
### ROC curve
print()
fig = plt.figure(figsize = (8, 6))
ax = fig.add_subplot(1, 1, 1)

ax.plot(fp_rate, tp_rate, color = 'red')
ax.plot((0, 1), (0, 1), linestyle = 'dashed', color = 'gray')
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("True Positive Rate")
ax.set_title("ROC Curve")

plt.show()
```



Features Learned

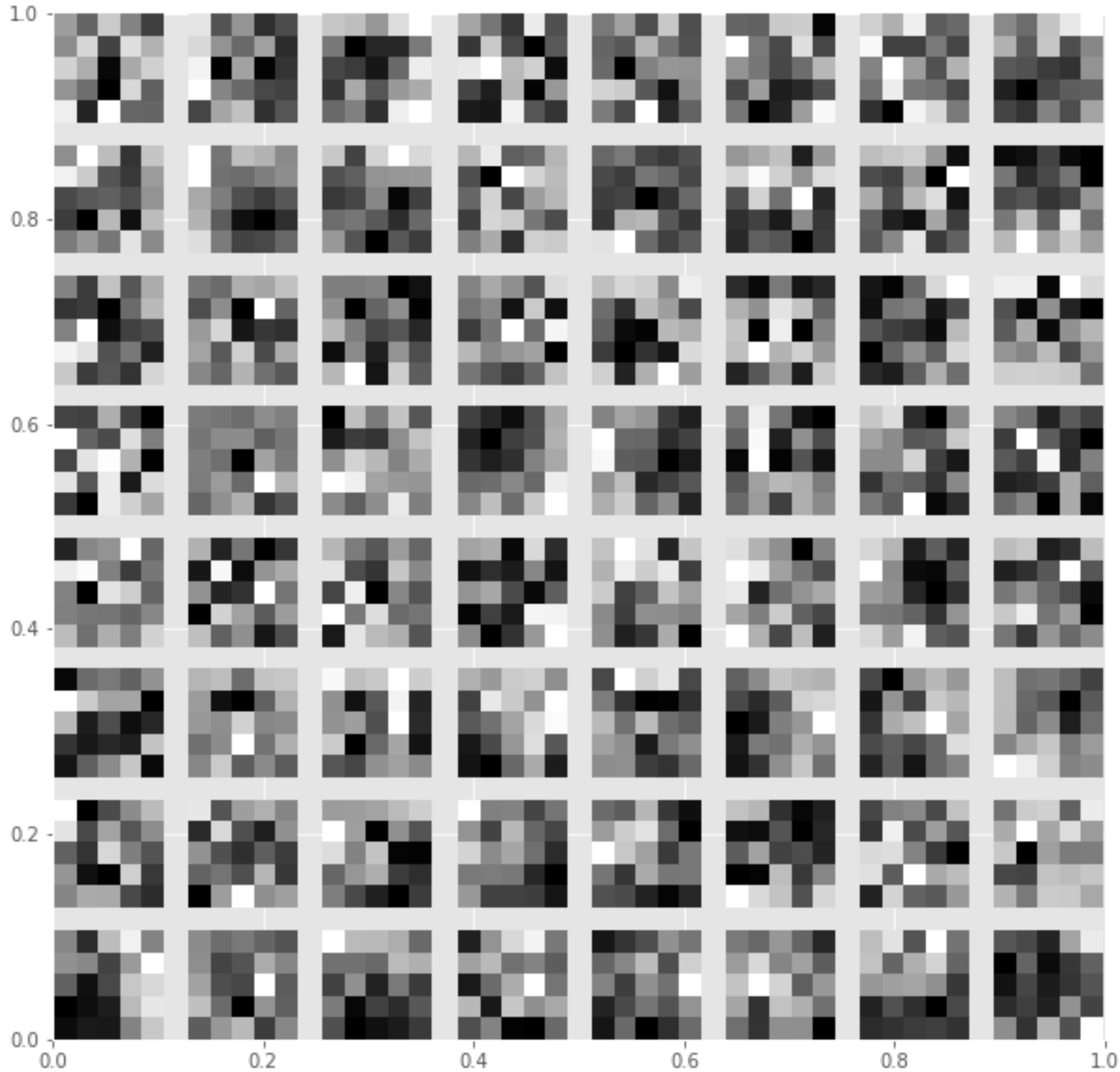
Layer 1

In [208]:
```
# get layer weights
layer = model.layers[1]
weights = layer.get_weights()

# set up plot
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)

# populate plot
index = 0
for i in range(8):
    for j in range(8):
        w = weights[0][:,:,0,index]
        w = w.reshape(5,5)
        index += 1
        ax = fig.add_subplot(8,8,index)
        ax.axis('off')
        plt.imshow(w, cmap = 'gray')

# show plot
plt.show()
```



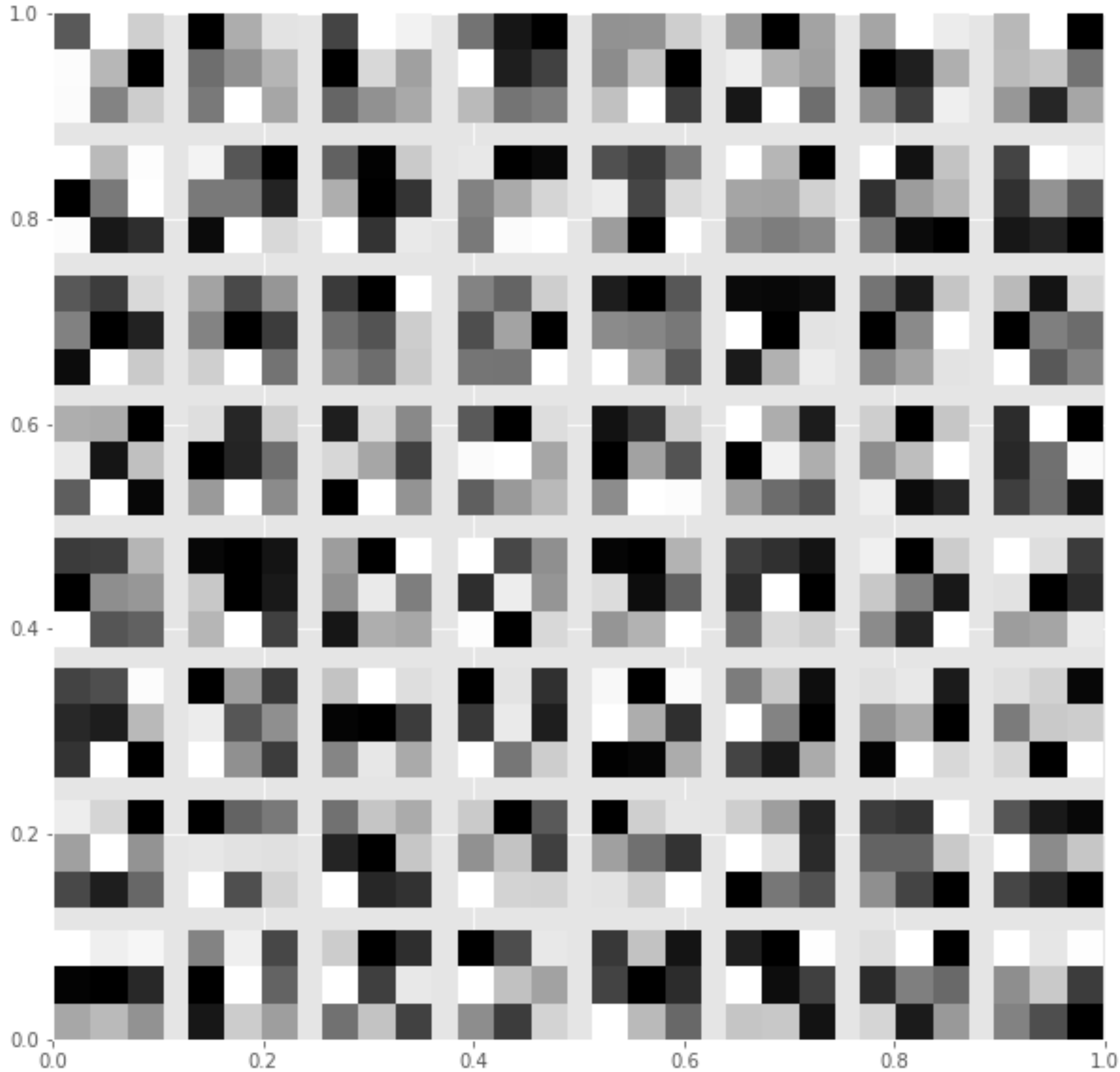
Layer 6

In [209]:
```
# get layer weights
layer = model.layers[6]
weights = layer.get_weights()

# set up plot
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)

# populate plot
index = 0
for i in range(8):
    for j in range(8):
        w = weights[0][:,:,0,index]
        w = w.reshape(3,3)
        index += 1
        ax = fig.add_subplot(8,8,index)
        ax.axis('off')
        plt.imshow(w, cmap = 'gray')

# show plot
plt.show()
```



Layer 8

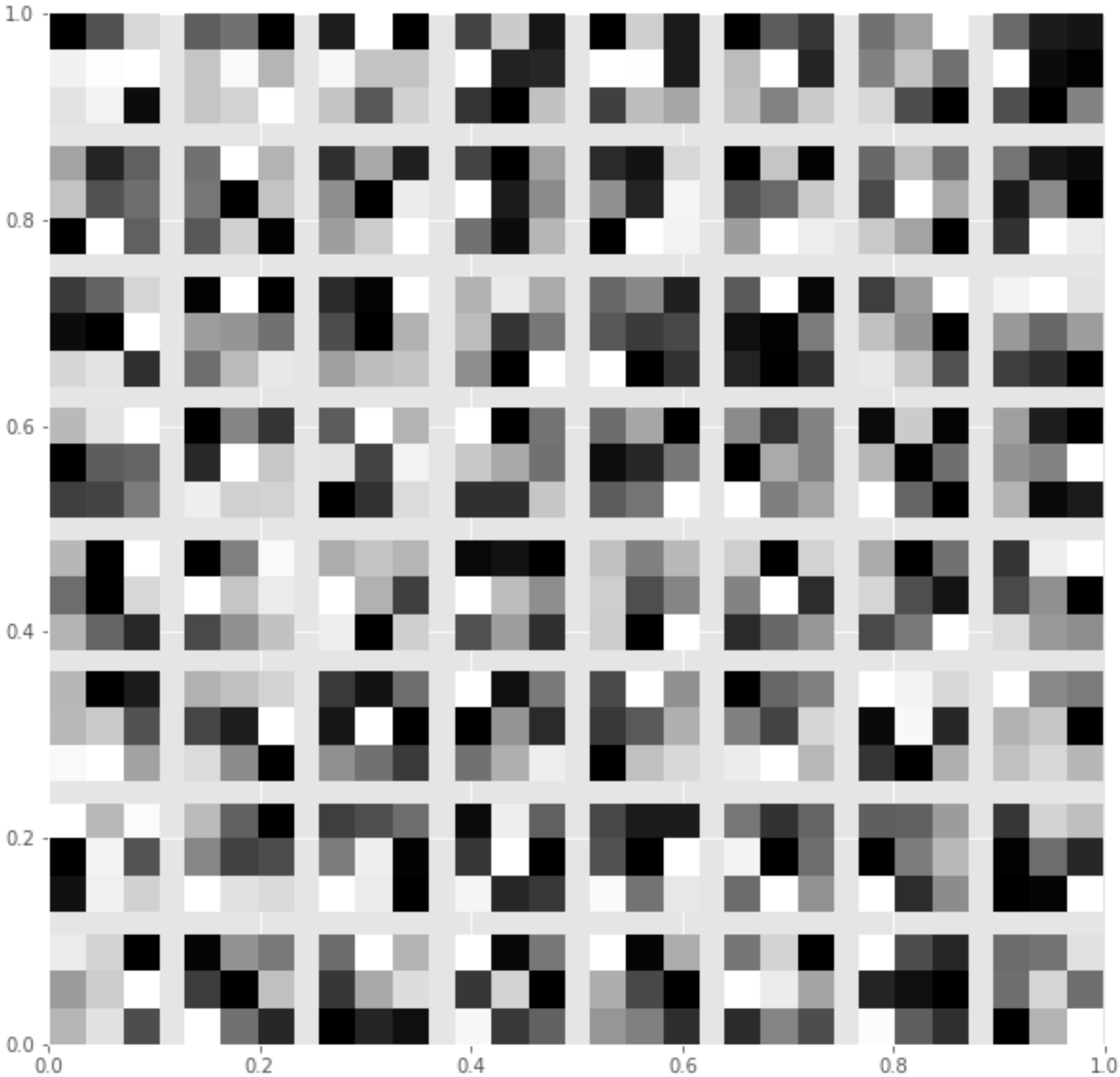
In [207]:

```
# get layer weights
layer = model.layers[8]
weights = layer.get_weights()

# set up plot
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)

# populate plot
index = 0
for i in range(8):
    for j in range(8):
        w = weights[0][:,:,0,index]
        w = w.reshape(3,3)
        index += 1
        ax = fig.add_subplot(8,8,index)
        ax.axis('off')
        plt.imshow(w, cmap = 'gray')

# show plot
plt.show()
```



In [ ]:

CS 109B - Course Project - Team 14

Deep Learning - Posters

1. Prepare AWS

Install packages

In [59]:

```
# import pip
# def install(package):
#     pip.main(['install', package])
# # install('boto')
# # install('opencv-python')
# install('h5py')
# # # install('scikit-image')
```

Retrieve images and data from Amazon's S3

In [60]:

```
# import boto
# import boto.s3.connection
# access_key = 'AKIAIULN726KJGOR6YJQ'
# secret_key = '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'

# conn = boto.connect_s3(
#         aws_access_key_id = access_key,
#         aws_secret_access_key = secret_key,
#         host = 's3.amazonaws.com',
#         #is_secure = False,               # uncomment if you are not using ssl
#         calling_format = boto.s3.connection.OrdinaryCallingFormat(),
#         )

# bucket = conn.get_bucket('cs109b.dkm.posters.250sq')
# key = bucket.get_key('Posters_250_250.zip')
# key.get_contents_to_filename('Posters_250_250.zip')

# bucket = conn.get_bucket('cs109b.dkm.imdb.data')
# key = bucket.get_key('imdb_movies_trim.csv')
# key.get_contents_to_filename('imdb_movies_trim.csv')

# import os, zipfile
# home_dir = os.path.expanduser("~")
# os.chdir(home_dir)
# my_zipfile = zipfile.ZipFile('Posters_250_250.zip')
# my_zipfile.extractall()
```

2. Load Data

Load packages and set display options

In [61]:

```
#from keras.models import Model
from keras.layers.normalization import BatchNormalization

from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input

import numpy as np
import pandas as pd
import os

from __future__ import print_function
import cv2
from scipy import ndimage, misc
from sklearn.cross_validation import train_test_split as sk_split
from sklearn.metrics import hamming_loss

import keras
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense, Activation, Conv2D, MaxPooling2D, Flatten, Dropo
ut, ZeroPadding2D
from keras.optimizers import SGD, Adam
from keras import backend as K
from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import CSVLogger
from keras.callbacks import EarlyStopping
from keras.callbacks import ModelCheckpoint

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('white')

from IPython.display import display, HTML, Markdown
%matplotlib inline
plt.style.use('ggplot')
def printmd(string):
    display(Markdown(string))
```

Set global constants

In [62]:

```
# input image dimensions
img_rows, img_cols = 128, 128

# smaller batch size means noisier gradient, but more updates per epoch
#Dom: keep high for laptop, lower for AWS
batch_size = 256

# 18 genres in our data set
num_classes = 19

# number of iterations over the complete training data
epochs = 100

#(80,75) equivilant to test 20% , train 60%, validate 20%
train_percent = 80
validation_percent = 75   #note: percentage of training (not the entire set)

#path containing the image files
image_path = './Posters_250_250/'
```

Load and clean Y data

In [63]:

```
movie_fields = ['imdb_id',
                'Action','Adventure','Animation','Comedy','Crime','Documentary',
'Drama',
                'Family','Fantasy','History','Horror','Music', 'Musical', 'Myste
ry',
                'Romance','Sci-Fi','Thriller','War','Western']
genres = movie_fields[1:]
small_genres = ['Western', 'Musical', 'Music', 'History', 'Animation', 'War', 'H
istory', "Sci-Fi"]

#ignore ,'Game-Show','News','Reality-TV', 'Biography','Adult','Film-Noir'
df_im_0 = pd.read_csv( './imdb_movies_trim.csv',
                    encoding = 'utf-8',
                    usecols = movie_fields)

# filter out no-genre movies
df_im = df_im_0[df_im_0.iloc[:, 1:].any(axis = 1)]

#reorder the columns
df_im = df_im[movie_fields]

#sort by movie id
df_im = df_im.sort_values(by = 'imdb_id')

# prep for augmentation
small_genres_idx = []
for i in xrange(len(small_genres)):
    small_genres_idx.append(movie_fields.index(small_genres[i]))
```

3. Load and Prepare Images

Helper function to Load and resize images

In [64]:

```
#inputs
#  image path
#  df_set dataframe (id, [genres])
#  Augment
#  num_images  default = 0, all images, otherwise limits the number of images
#returns
#  x numpy matrix of flaterned images
#  y numpy matrix of responses
def load_images(df_set, image_path, augment, num_images):
    images = []
    imdb_ids = []
    imdb_id_image_not_found = []
```

```
    #use whole dataset if not capped
    if (num_images == 0):
        num_images = df_set.shape[0]

    for i in xrange(num_images):

        # retrieve movie object from imdb
        imdb_id = int(df_set.iloc[i,0])
        filepath = image_path  + str(imdb_id).zfill(7) + '.jpg'

        # image found, so load, resize, and collect it
        if os.path.isfile(filepath) :

            image = cv2.imread(filepath)
            image = cv2.resize(image,(img_rows, img_cols))
            image = np.array(image).reshape((3, img_rows, img_cols))

            # add image and movie id to list
            images.append([image])
            imdb_ids.append(imdb_id)

            # up-sample and augment images for minority genres (only on training
)

            if (augment & (sum(df_set.iloc[i, small_genres_idx]) >= 1)) :
                image_flip = cv2.flip(image, 1) # vertical-axis flip
                images.append([image_flip])
                imdb_ids.append(imdb_id)

        else:
            # add id to failure list
            imdb_id_image_not_found.append(imdb_id)

    # collect the image list into an array
    x = np.array(images)
    x = x[:, 0, :, :]
    images = None

    df_imdb_ids = pd.DataFrame({'id': imdb_ids })

    #Build y taking care to align rows with x
    #merge ensures that if images were not found they do not cause a mis-aslignm
ent of x and y
    y = pd.merge(left = df_imdb_ids , right = df_set , left_on = 'id', right_on
= 'imdb_id')

    #drop first two columns (id and imdb_id) leaving just the 18 encoded genres
    y = y.ix[:,2:]

    print ('images loaded', len(imdb_ids))
    print ('images failed', len(imdb_id_image_not_found))

    return x, y.values
```

Split Responses into test, train and Validation

In [65]:

```
#split full dataset into test & train
y_train, y_test = sk_split(df_im, train_size = train_percent / 100.0)

#futher split train into train and validation
y_train, y_valid = sk_split(y_train, train_size = validation_percent / 100.0)


# create dictionary of class weights
class_count = y_train.sum(axis = 0) + 1
class_weight = y_train.shape[0] / (num_classes * class_count)
class_weight_dict = dict(zip(np.arange(num_classes), class_weight))
```

Load images

In [66]:

```
#load images for datasets and convert to numpy arrays
print()
print ('Load Training data')
x_train, y_train = load_images(y_train, image_path, augment = True, num_images =
4000)
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)

print()
print ('Load Validation data')
x_valid, y_valid = load_images(y_valid, image_path, augment = False, num_images
= 800)
print('x_valid shape:', x_valid.shape)
print('y_valid shape:', y_valid.shape)

print()
print ('Load Test data')
x_test , y_test  = load_images(y_test , image_path, augment = False, num_images
= 800)
print('x_test shape:' , x_test.shape)
print('y_test shape:' , y_test.shape)
```

```
Load Training data
images loaded 5045
images failed 1
x_train shape: (5045, 3, 128, 128)
y_train shape: (5045, 19)

Load Validation data
images loaded 800
images failed 0
x_valid shape: (800, 3, 128, 128)
y_valid shape: (800, 19)

Load Test data
images loaded 800
images failed 0
x_test shape: (800, 3, 128, 128)
y_test shape: (800, 19)
```

Using tensorflow as backend so expect order of array as is (n = sample size, img_rows, img_cols, colour = 3)

In [67]:

```
if K.image_data_format() ==  'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 3, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 3, img_rows, img_cols)
    x_valid = x_valid.reshape(x_valid.shape[0], 3, img_rows, img_cols)
    input_shape = (3, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 3)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 3)
    x_valid = x_valid.reshape(x_valid.shape[0], img_rows, img_cols, 3)
    input_shape = (img_rows, img_cols, 3)
```

Centre and normalise images

In [68]:

```
# normalize image values to [0,1] (Keras sample code doesn't center)
x_train = x_train.astype('float32')
x_test  = x_test.astype('float32')
x_valid = x_valid.astype('float32')

x_train /= 255
x_test  /= 255
x_valid /= 255

print()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_valid.shape[0], 'validation samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (5045, 128, 128, 3)
5045 train samples
800 validation samples
800 test samples
```

4. Train Model

Load and Modify Pretrained Model

In [69]:

```
#load vgg16 without the 3 fully-connected layers at the top of the network
base_model = VGG16(weights = 'imagenet', include_top = False, input_shape = inpu
t_shape)

# set all other layers from the pretained model to non-trainable
for layer in base_model.layers:
    layer.trainable = False

# create new classification layers for our genre classification ---
x = Flatten()(base_model.output)
x = Dense(4096, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(4096, activation='relu')(x)
x = Dropout(0.5)(x)
#x = BatchNormalization()(x)
predictions = Dense(num_classes, activation = 'sigmoid', )(x)

#Place our genre classification layer on top of the existing model
model = keras.models.Model(inputs = base_model.input, outputs = predictions)

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 128, 128, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 128, 128, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 128, 128, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 64, 64, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 64, 64, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 64, 64, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 32, 32, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 32, 32, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 32, 32, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 32, 32, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 16, 16, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 16, 16, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
```

```
block4_conv3 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 8, 8, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
_________________________________________________________________
flatten_5 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_13 (Dense)             (None, 4096)              33558528
_________________________________________________________________
dropout_9 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_14 (Dense)             (None, 4096)              16781312
_________________________________________________________________
dropout_10 (Dropout)         (None, 4096)              0
_________________________________________________________________
dense_15 (Dense)             (None, 19)                77843
=================================================================
Total params: 65,132,371.0
Trainable params: 50,417,683.0
Non-trainable params: 14,714,688.0
_________________________________________________________________
```

Compile model

In [70]:

```
# use basic categorical crossentropy with stochastic gradient decent
# evaluate model in terms of accuracy
# lr : learning rate (start from 0.1 and move as low as 10^-6)
# momentum: start with .5 and tune to .9


sgd = SGD(lr = 0.1, momentum = 0.9, nesterov = True)
model.compile(loss = 'binary_crossentropy',
              optimizer = sgd,
              metrics = ['accuracy'])
```

Train model and score with validation data

In [71]:

```
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.01,
                              patience = 10, min_lr = 0.000001)

early_stop = EarlyStopping (monitor = 'val_loss', min_delta = .0001, patience =
15, verbose = 0, mode = 'auto')

csv_logger = CSVLogger('training.log')

model_checkpoint = ModelCheckpoint(filepath = "./best_model.hdf5", monitor = 'va
l_loss',
                                   verbose = 0, save_best_only = True,
                                   save_weights_only = False, mode = 'auto', per
iod = 1)

history = model.fit(x_train, y_train,
                    batch_size = batch_size,
                    epochs = epochs,
                    callbacks = [reduce_lr, early_stop, csv_logger, model_checkp
oint],
                    verbose = 1,
                    validation_data = (x_valid, y_valid),
                    class_weight = class_weight_dict)
```

```
Train on 5045 samples, validate on 800 samples
Epoch 1/100
5045/5045 [==============================] - 53s - loss: 0.2217 - ac
c: 0.8431 - val_loss: 0.3541 - val_acc: 0.8779
Epoch 2/100
5045/5045 [==============================] - 47s - loss: 0.1858 - ac
c: 0.8674 - val_loss: 0.3446 - val_acc: 0.8769
Epoch 3/100
5045/5045 [==============================] - 42s - loss: 0.1801 - ac
c: 0.8704 - val_loss: 0.3465 - val_acc: 0.8730
Epoch 4/100
5045/5045 [==============================] - 47s - loss: 0.1771 - ac
c: 0.8708 - val_loss: 0.3412 - val_acc: 0.8783
Epoch 5/100
5045/5045 [==============================] - 42s - loss: 0.1744 - ac
c: 0.8720 - val_loss: 0.3461 - val_acc: 0.8749
Epoch 6/100
5045/5045 [==============================] - 42s - loss: 0.1729 - ac
c: 0.8723 - val_loss: 0.3416 - val_acc: 0.8786
Epoch 7/100
5045/5045 [==============================] - 47s - loss: 0.1708 - ac
c: 0.8724 - val_loss: 0.3400 - val_acc: 0.8782
Epoch 8/100
5045/5045 [==============================] - 47s - loss: 0.1693 - ac
c: 0.8733 - val_loss: 0.3398 - val_acc: 0.8782
Epoch 9/100
5045/5045 [==============================] - 47s - loss: 0.1674 - ac
c: 0.8742 - val_loss: 0.3395 - val_acc: 0.8780
Epoch 10/100
5045/5045 [==============================] - 47s - loss: 0.1668 - ac
c: 0.8751 - val_loss: 0.3392 - val_acc: 0.8770
Epoch 11/100
5045/5045 [==============================] - 42s - loss: 0.1648 - ac
c: 0.8750 - val_loss: 0.3423 - val_acc: 0.8759
Epoch 12/100
5045/5045 [==============================] - 47s - loss: 0.1636 - ac
c: 0.8756 - val_loss: 0.3392 - val_acc: 0.8774
Epoch 13/100
5045/5045 [==============================] - 43s - loss: 0.1619 - ac
c: 0.8759 - val_loss: 0.3439 - val_acc: 0.8740
```

In [72]:

```
model.save('VGG_retrain.h5')  # creates a HDF5 file
#model = load_model('VGG_retrain.h5')
```

5. Score Model

In [73]:

```
### prepare metrics calculations

avg_tags_per_movie = y_train.sum(axis = 1).mean()
y_test_predict_proba = model.predict(x_test, verbose = 0)

threshold_accus = np.zeros(101)
tp_rate = np.zeros(101)
fp_rate = np.zeros(101)

#Finding the threshold that mininises the hamming loss
for i in range(101):
    y_test_predict = y_test_predict_proba > i / 100.0
    threshold_accus[i] = hamming_loss(y_test, y_test_predict)
    tp_rate[i] = ((y_test == 1) & (y_test_predict == 1)).sum() / float((num_clas
ses * y_test.shape[0]))
    fp_rate[i] = ((y_test == 0) & (y_test_predict == 1)).sum() / float((num_clas
ses * y_test.shape[0]))

opt_hamming_threshold = threshold_accus.argmin() / 100.0
```

In [74]:

```
# score after training
score = model.evaluate(x_test, y_test, verbose = 0)
print('Test loss:', round(score[0], 4))
print('Test accuracy:', round(score[1], 4))
```

```
Test loss: 0.3577
Test accuracy: 0.8687
```

In [75]:

```
y_test_predict = y_test_predict_proba > .5
test_accu = (y_test == y_test_predict).sum() / float((num_classes * y_test.shape
[0]))
print('Test accuracy using probability threshold of 0.50 :', round(test_accu, 4)
)
```

```
Test accuracy using probability threshold of 0.50 : 0.8687
```

In [78]:

```
y_test_predict = y_test_predict_proba > opt_hamming_threshold
test_accu = (y_test == y_test_predict).sum() / float((num_classes * y_test.shape
[0]))
print('Test accuracy using probability threshold of', round(opt_hamming_threshol
d, 2), ':', round(test_accu, 4))
```

```
Test accuracy using probability threshold of 0.73 : 0.8779
```

In [79]:

```
y_test_predict = np.zeros([y_test.shape[0], num_classes])
test_accu = (y_test == y_test_predict).sum() / float((num_classes * y_test.shape
[0]))
print('Test accuracy using classifier that always predicts 0 :', round(test_accu
, 4))
```

```
Test accuracy using classifier that always predicts 0 : 0.8724
```

In [81]:

```
y_test_predict = y_test_predict_proba > opt_hamming_threshold
temp = pd.DataFrame(y_test_predict.sum(axis = 0), columns = ['Count'])
temp.index = genres
temp
```

Out[81]:

	Count
Action	0
Adventure	0
Animation	0
Comedy	214
Crime	0
Documentary	0
Drama	203
Family	0
Fantasy	0
History	0
Horror	11
Music	0
Musical	0
Mystery	0
Romance	10
Sci-Fi	0
Thriller	2
War	0
Western	0

In [83]:

```
from sklearn.metrics import precision_score as ps

y_test_predict = y_test_predict_proba > opt_hamming_threshold
avg_precision = ps(y_test, y_test_predict, average = 'samples')
print('Average test precision (samples):', round(avg_precision, 4))
```

```
Average test precision (samples): 0.301

/usr/lib/python2.7/dist-packages/sklearn/metrics/classification.py:1
074: UndefinedMetricWarning: Precision is ill-defined and being set
to 0.0 in samples with no predicted labels.
  'precision', 'predicted', average, warn_for)
```

In [84]:

```
from sklearn.metrics import recall_score as rs

y_test_predict = y_test_predict_proba > opt_hamming_threshold
avg_recall = rs(y_test, y_test_predict, average = 'samples')
print('Average test recall (samples):', round(avg_recall, 4))
```

```
Average test recall (samples): 0.179

/usr/lib/python2.7/dist-packages/sklearn/metrics/classification.py:1
076: UndefinedMetricWarning: Recall is ill-defined and being set to
0.0 in samples with no true labels.
  'recall', 'true', average, warn_for)
```

In [85]:

```
y_test_predict_top_class = np.argmax(y_test_predict_proba, axis = 1)
test_top_class_accu_list = [(y_test[i, y_test_predict_top_class[i]] == 1) for i
in range(y_test.shape[0])]
test_top_class_accu = np.sum(test_top_class_accu_list) / float(y_test.shape[0])
print('Test accuracy matching top class:', round(test_top_class_accu, 4))
```

```
Test accuracy matching top class: 0.5375
```

In [86]:

```
y_test_predict = y_test_predict_proba > 0.33
test_hamming_loss = hamming_loss(y_test, y_test_predict)
print('Test Hamming loss:', round(test_hamming_loss, 4))
```

```
Test Hamming loss: 0.1549
```

6. Visualization

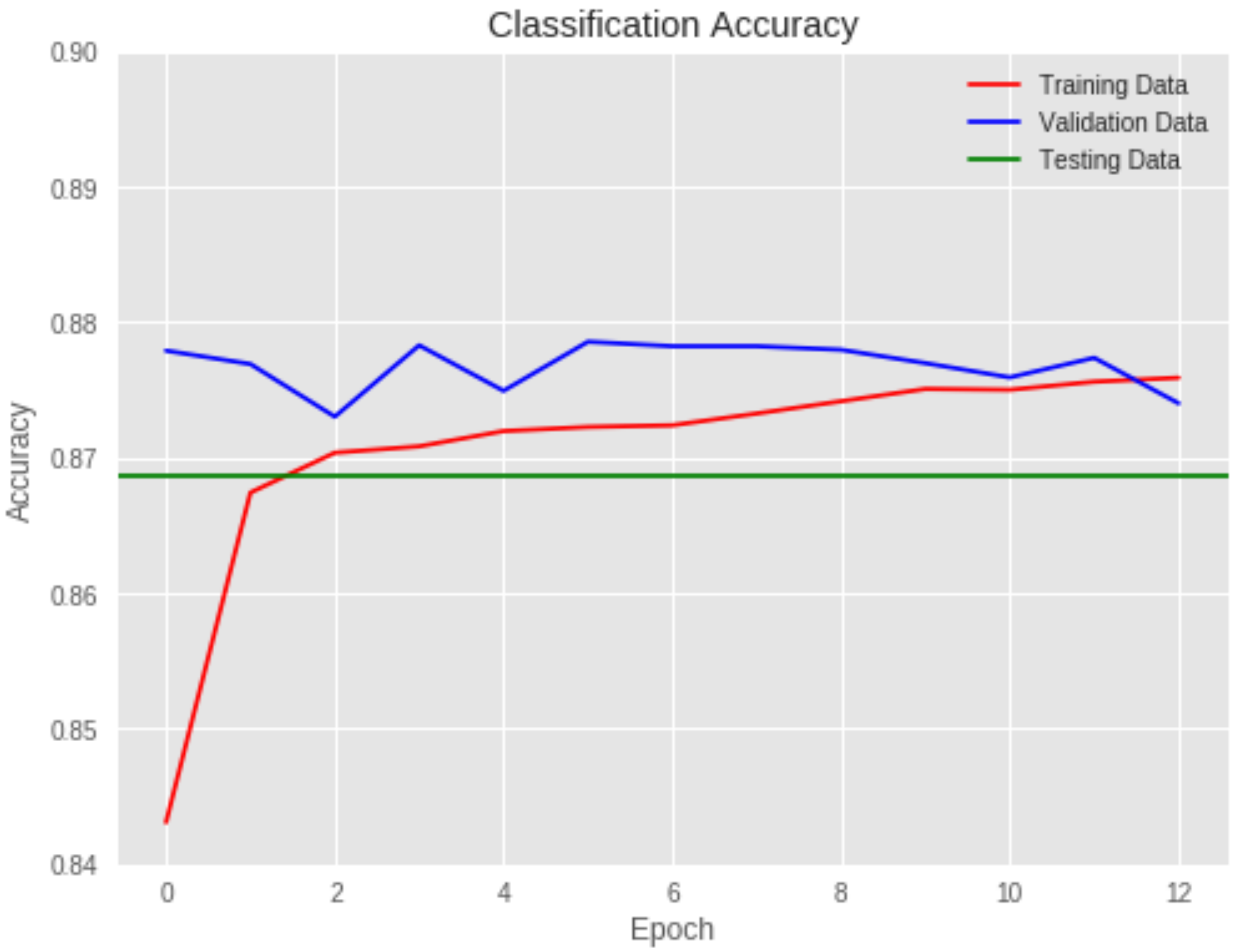
Training process

In [93]:

```
### plot training process
print()
fig = plt.figure(figsize = (8, 6))
ax = fig.add_subplot(1, 1, 1)

ax.plot(history.history['acc'], color = 'red', label = 'Training Data')
ax.plot(history.history['val_acc'], color = 'blue', label = 'Validation Data')
ax.axhline(y = score[1], color = 'green', label = 'Testing Data')
ax.set_ylim(0.84, .9)
ax.legend(loc = 'upper right')
ax.set_xlabel("Epoch")
ax.set_ylabel("Accuracy")
ax.set_title("Classification Accuracy")

plt.show()
```



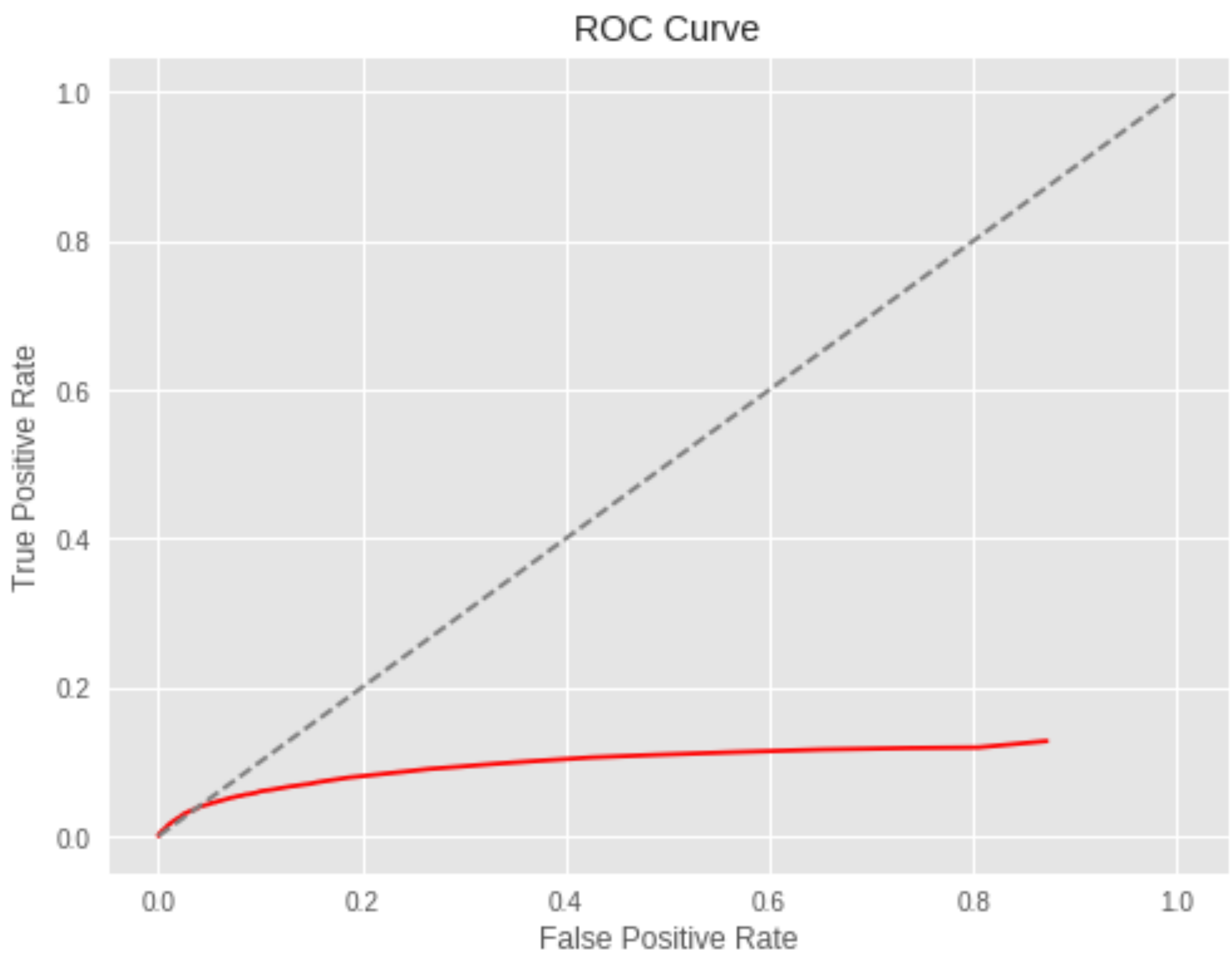
ROC Curve

In [94]:

```
### ROC curve
print()
fig = plt.figure(figsize = (8, 6))
ax = fig.add_subplot(1, 1, 1)

ax.plot(fp_rate, tp_rate, color = 'red')
ax.plot((0, 1), (0, 1), linestyle = 'dashed', color = 'gray')
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("True Positive Rate")
ax.set_title("ROC Curve")

plt.show()
```



Features Learned

Layer 1

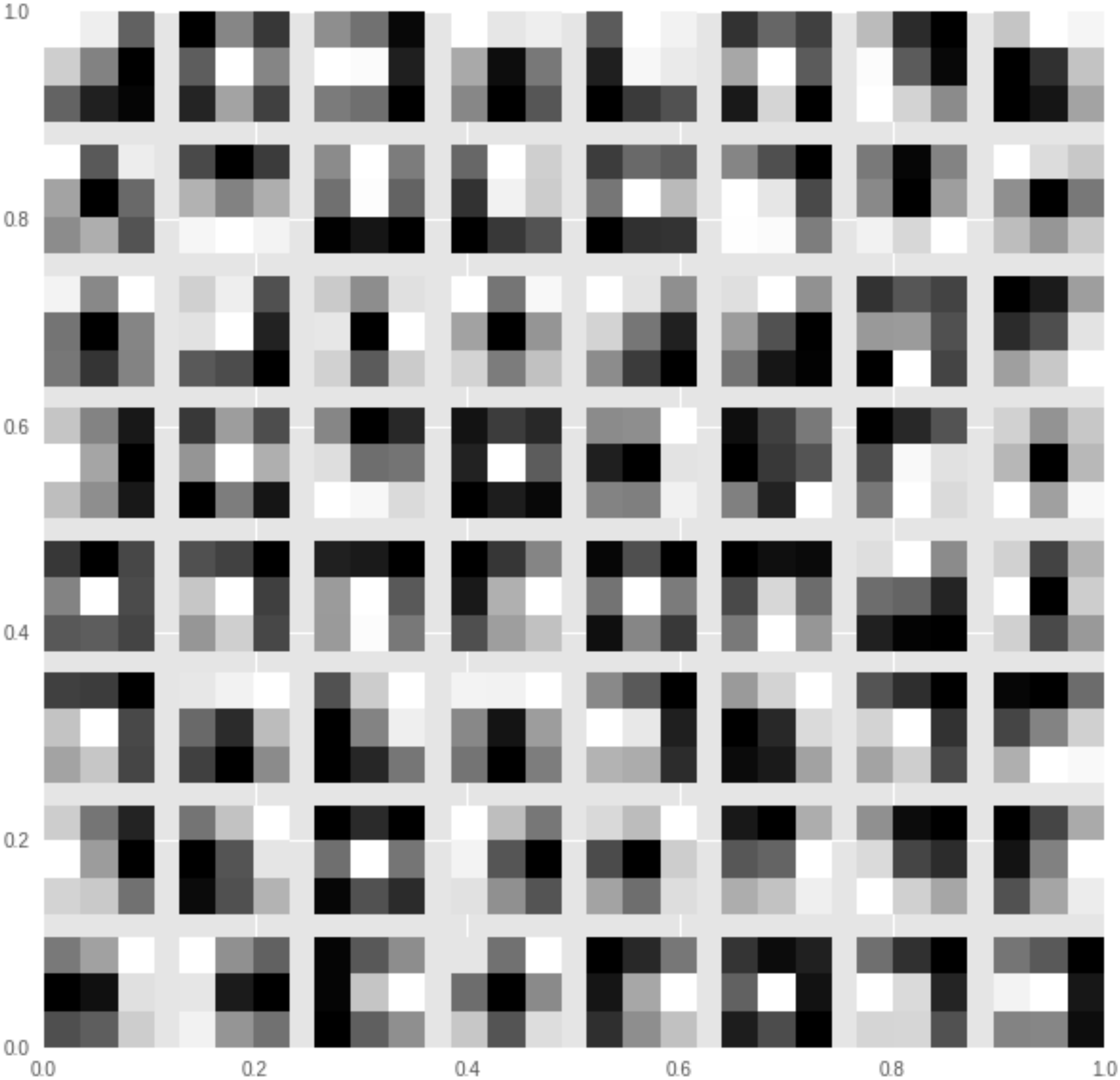
In [125]:

```
# get layer weights
layer = model.layers[1]
weights = layer.get_weights()

# set up plot
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)

# populate plot
index = 0
for i in range(8):
    for j in range(8):
        w = weights[0][:,:,0,index]
        w = w.reshape(3,3)
        index += 1
        ax = fig.add_subplot(8,8,index)
        ax.axis('off')
        plt.imshow(w, cmap = 'gray')

# show plot
plt.show()
```



Layer 9

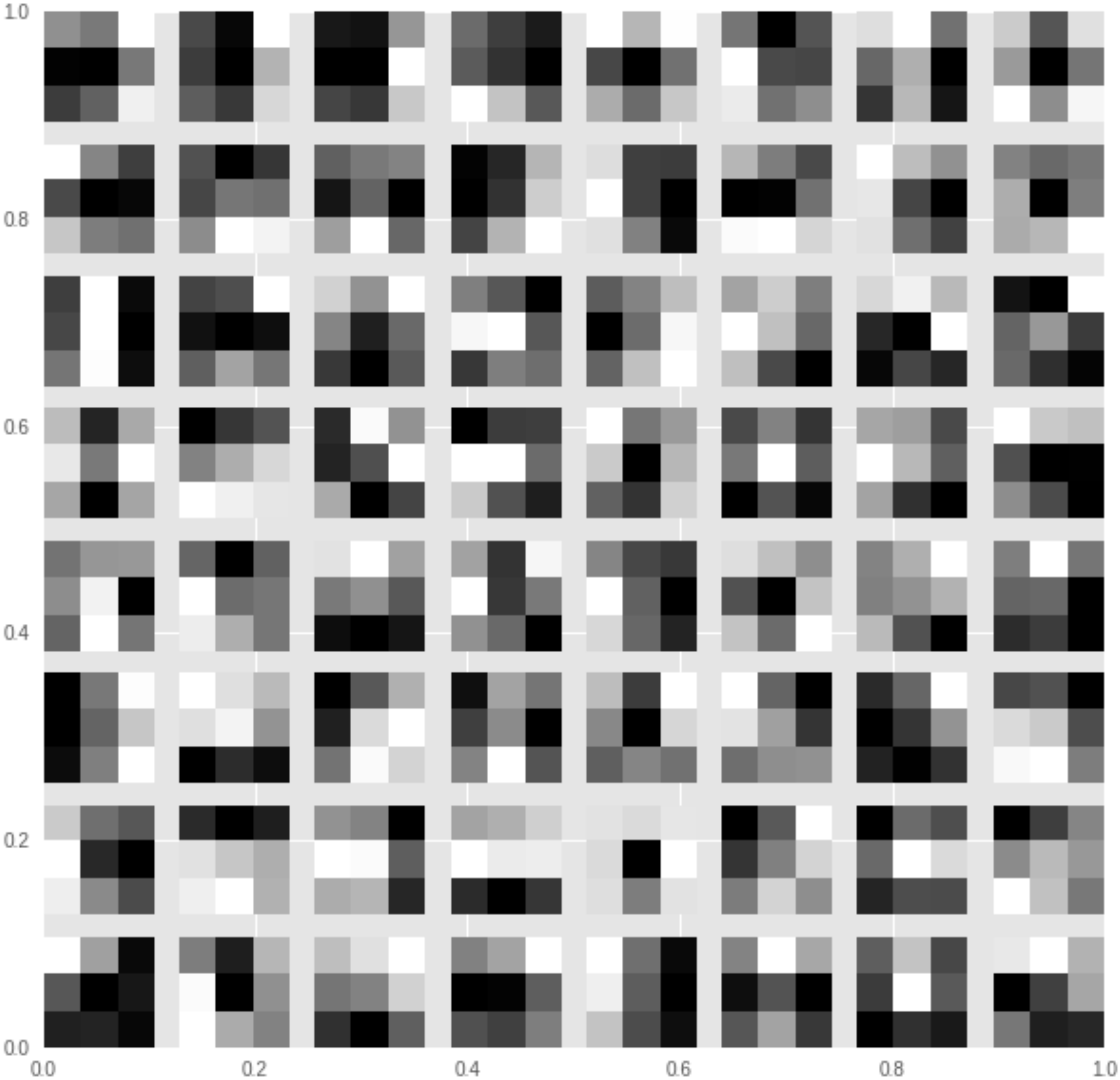
In [126]:

```
# get layer weights
layer = model.layers[9]
weights = layer.get_weights()

# set up plot
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)

# populate plot
index = 0
for i in range(8):
    for j in range(8):
        w = weights[0][:,:,0,index]
        w = w.reshape(3,3)
        index += 1
        ax = fig.add_subplot(8,8,index)
        ax.axis('off')
        plt.imshow(w, cmap = 'gray')

# show plot
plt.show()
```



Layer 17

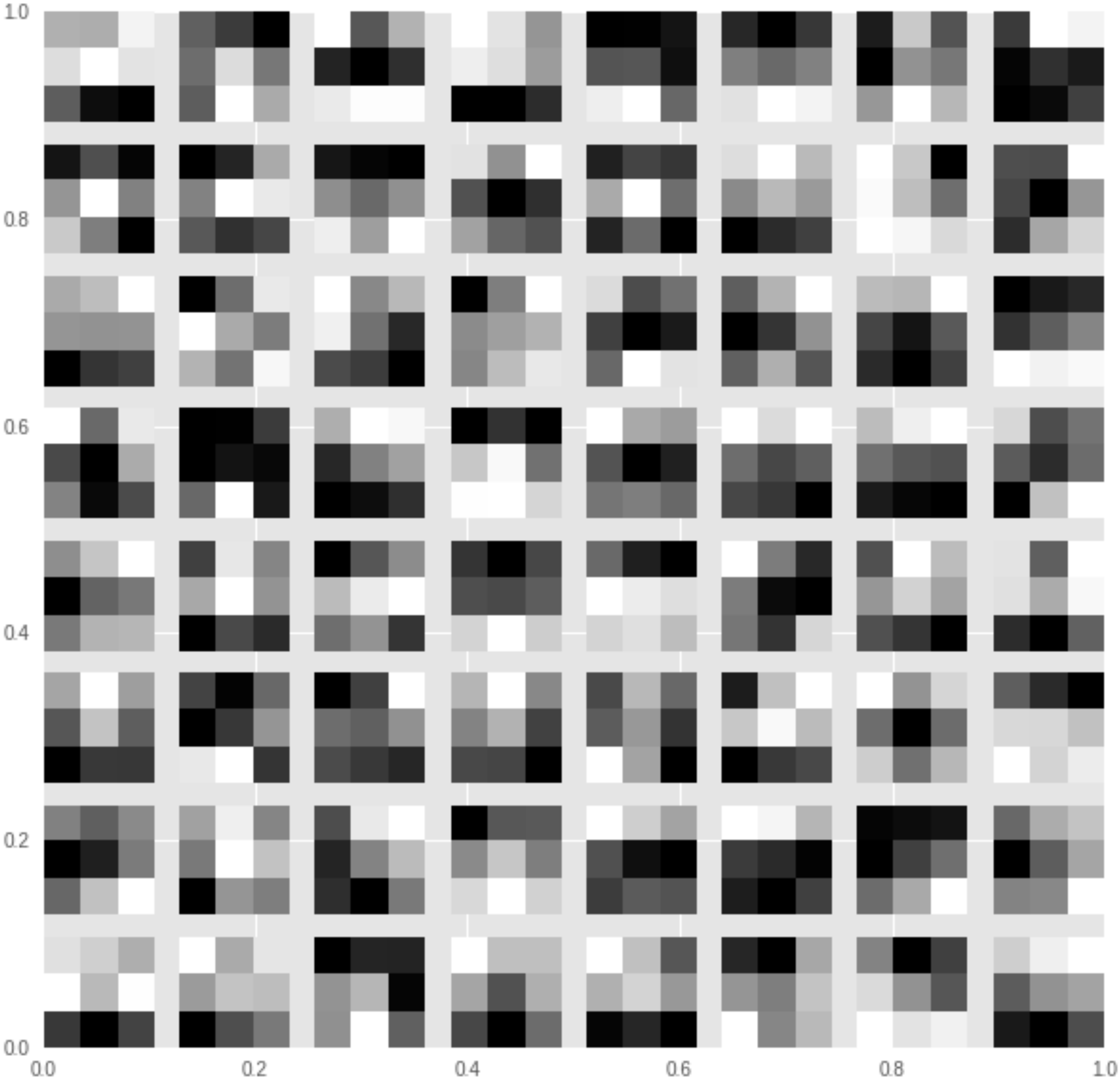
In [128]:

```
# get layer weights
layer = model.layers[17]
weights = layer.get_weights()

# set up plot
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)

# populate plot
index = 0
for i in range(8):
    for j in range(8):
        w = weights[0][:,:,0,index]
        w = w.reshape(3,3)
        index += 1
        ax = fig.add_subplot(8,8,index)
        ax.axis('off')
        plt.imshow(w, cmap = 'gray')

# show plot
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0