Stacked Sparse Denoising Autoencoder

In this assignment, we will create a Stacked Sparse Denoising Autoencoder (SSDA) with the purpose of denoising grayscale images.

Not everything required to finish this assignment is covered in class but you should start working on "Imports", "Load Images", and "Create Patches" subsections that follow.

For this assignment, we recommend working in a Jupyter Notebook (http://jupyter.org). Jupyter Notebook is a browser based interactive programming environment and supports several languages (https://github.com/jupyter/jupyter/wiki/Jupyter-kernels).

For this assignment, you will most likely need to use a deep learning framework. Several frameworks exist. Here are a few of them (https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software). Based on personal and borrowed experience, I will recommend working with Keras if you have no idea what to choose. Brave or experienced users can feel free to use other frameworks.

I (Priyank) will cover an overview of python, numpy, jupyter, and a deep learning framework sometime during next week.

Keras will not work on NEU cluster, though, and you will have to train on your machine or Maria's desktop. If you don't have an access to yet, now is the best time to request it.

Additionally, we will also need to allocate time slots to each of the students to train their models on desktop should someone chooses to implement a complex model.

This homework is worth 100 points. However, there is no breakdown of points per steps. These steps are minimum that is required to score 100 points. Bonus points for showing interesting results and insights.

Credits for this assignment go to Maria's summer advisee Kaleigh O'hara.

This SSDA model is made up of two autoencoders. Rather than train the entire model at once, we train the two autoencoder layers individually, save their weights, and then use those weights to initialize the stacked autoencoder model.

A single autoencoder model includes an input layer, an encoding layer and a decoding layer. After training the first autoencoder and saving its weights, we run the original clean and noisy patches through the input and encoding layers of our training model. The output patches of this model become the input patches to the second autoencoder's training model.

We train the fully assembled SSDA model by initializing the two encoding layers to the weights saved from our individually trained autoencoder models. We provide the trained SSDA model with noisy test images and evaluate its performance.

Keras Blog - 'Deep autoencoder' used as example
https://blog.keras.io/building-autoencoders-in-keras.html (https://blog.keras.io/building-autoencoders-in-keras.html)

IMPORTS

In [1]:
```
#to resolve mkl bug when importing skimage (Seattle Computer Only)
# import mkl
# mkl.get_max_threads()

from keras.layers import Input, Dense
from keras.models import Model

import numpy as np

from keras.datasets import mnist
import numpy as np
import cv2 as cv2


import os as os  #some debugging code
# os.sys.path
```

```
Using TensorFlow backend.
```

LOAD IMAGES

For this assignment, we will use images from the Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500). The BSDS500 dataset provides three categories of images: train, val, and test, which we will use for training, validating, and testing the models.
https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html (https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html)

To reduce the dimensionality of the data, your program should convert the images to greyscale.
Load the BSDS500 train, validate, and test datasets:

In [2]:
```
train_path = "/Users/divyaagarwal/Desktop/Computervision/Assignment2Autoe
train_imgs = [f for f in os.listdir(train_path) if f[-3:] == 'jpg']
train_list = [cv2.imread(os.path.join(train_path, f), cv2.IMREAD_GRAYSCAL
train = np.array([t for t in train_list])
```

In [3]:
```
# TRAIN IMAGES (200 Total)
x_train = []

print train[0].shape

for i in range(len(train)):
    x_train.append(train[i].astype('float32'))


for i in range(len(x_train)):
    print x_train[i]
```

```
[[  80.  138.  143. ...,  130.  144.  147.]
 [ 127.  144.  136. ...,  151.  149.  150.]
 [ 137.  133.  133. ...,  150.  150.  153.]
 ...,
 [  11.   11.   11. ...,  135.  169.  159.]
 [  11.   11.   11. ...,  131.  182.  187.]
 [  11.   11.   11. ...,  140.  179.  136.]]
[[ 101.  145.  151. ...,  137.  134.  127.]
 [  96.  100.  126. ...,  107.  107.  106.]
 [  80.   93.  137. ...,   38.   39.   41.]
 ...,
 [ 125.  122.  121. ...,  121.  123.  127.]
 [ 127.  104.  102. ...,  121.  114.  108.]
 [ 103.  112.   97. ...,  116.  119.  117.]]
[[  42.   41.   41. ...,   86.   96.  141.]
 [  42.   42.   42. ...,  110.  135.  110.]
 [  42.   42.   43. ...,  126.   94.   83.]
 ...,
 [  87.   74.   65. ...,   98.   99.   99.]
 [  87.   72.   66. ...,   72.   97.  105.]
```

In [4]:
```
validate_path = "/Users/divyaagarwal/Desktop/Computervision/Assignment2Au
validate_imgs = [f for f in os.listdir(validate_path) if f[-3:] == 'jpg']
validate_list = [cv2.imread(os.path.join(validate_path, f), cv2.IMREAD_GR
validate = np.array([t for t in validate_list])
```

In [5]:
```
# VALIDATE IMAGES (100 Total)
x_validate = []

for i in range(len(validate)):
    x_validate.append(validate[i].astype('float32'))
```

In [6]:
```
test_path = "/Users/divyaagarwal/Desktop/Computervision/Assignment2Autoen
test_imgs = [f for f in os.listdir(test_path) if f[-3:] == 'jpg']
test_list = [cv2.imread(os.path.join(test_path, f), cv2.IMREAD_GRAYSCALE)
test = np.array([t for t in test_list])
```

In [7]:
```
# # TEST IMAGES (200 Total)

x_test = []

for i in range(len(test)):
    x_test.append(test[i].astype('float32'))
```

CREATE PATCHES

For each image in the train, validate, and test datasets, create clean and noisy patches using a window size of 8x8 and a step size of 8. Each noisy patch is created by applying random gaussian noise to its equivalent clean patch. Set the seed variable to allow comparison between program executions.

Hint for creating patches: [http://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/ (http://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/)](http://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/)

Output Format:

- Each pixel must be converted from int to float.
- Each patch must be a flattened numpy array of length 64.
- Clean and Noisy sets of Train patches have shape (480000, 64).
- Clean and Noisy sets of Validate patches have shape (240000, 64).
- Clean and Noisy sets of Test patches have shape (480000, 64).

Now creating Clean and Noisy Taining patches of shape (480000, 64).

In [8]:
```
windowSize = (8,8)
stepSize = 8
# numPatches = 30*2400
numPatches = 480000
numPixels = 64

train_patch = np.empty(shape=(numPatches, numPixels))

k=0
for i in range(len(x_train)):
    image = x_train[i]
    for y in xrange(0, image.shape[0]-8-1, stepSize):
        for x in xrange(0, image.shape[1]-8-1, stepSize):
            train_patch[k] = image[y:y + windowSize[1], x:x + windowSize[
            k = k+1
```

In [9]:
```
print train_patch.shape
```

```
(480000, 64)
```

In [10]:
```
print train_patch[0]
```

```
[ 111.  115.  116.  119.  126.  131.  127.  121.  108.  109.  110.  11
6.
  130.  139.  135.  126.  118.  116.  117.  123.  133.  137.  133.  12
6.
  123.  122.  126.  133.  134.  131.  130.  132.  125.  126.  132.  13
8.
  137.  134.  132.  131.  136.  136.  139.  141.  142.  143.  136.  12
4.
  144.  140.  141.  142.  143.  146.  142.  131.  147.  140.  140.  14
2.
  138.  138.  142.  142.]
```

In [11]:
```
train_noisy_patch = np.empty(shape=(numPatches, numPixels))

# Noisy Patches
sigma = 25

for i in range(len(train_patch)):
    patch = train_patch[i]
    noise = np.random.random(patch.shape) * sigma
    newPatch = np.add(patch,noise)
    train_noisy_patch[i] = np.clip(newPatch, 0, 255)
```

In [12]:
```
print train_noisy_patch[0]
```

```
[ 131.63920714  132.41504264  133.67948457  131.26625144  140.37018471
  135.47408713  145.07620543  145.64272548  121.3820875   125.299633
  111.85493016  117.96745162  151.61998836  156.95828459  157.40030936
  129.50498264  138.08058669  140.33729645  138.6142917   132.51315001
  133.49610031  152.4027049   150.36458967  135.43869039  131.66695192
  137.84977743  141.63340962  138.97896745  135.65084801  149.95344208
  147.70855627  140.49755504  139.25529465  144.21616729  144.48890336
  155.28312836  140.510952    155.88989093  139.76908771  140.37330441
  144.44581561  157.65962077  156.3865672   162.75398887  156.26158942
  143.95375069  152.33854792  130.91858037  155.26951817  147.04163542
  148.91061268  165.98627358  144.99882658  152.26958255  142.5032973
  136.72919203  155.33356225  164.35548171  160.19363392  146.50334004
  153.58780958  138.94426201  152.86946939  143.61908463]
```

Now creating Clean and Noisy Validate patches of shape (480000, 64).

In [13]:
```
windowSize = (8,8)
stepSize = 8
#numPatches = 5*2400
numPatches = 240000
numPixels = 64

validate_patch = np.empty(shape=(numPatches, numPixels))

k=0
for i in range(len(x_validate)):
    image = x_validate[i]
    for y in xrange(0, image.shape[0]-8-1, stepSize):
        for x in xrange(0, image.shape[1]-8-1, stepSize):
            validate_patch[k] = image[y:y + windowSize[1], x:x + windowSi
            k = k+1
```

In [14]:
```
print validate_patch.shape
```

```
(240000, 64)
```

In [15]:
```
print validate_patch[0]
```

```
[ 200.  255.  255.  254.  253.  255.  250.  255.  253.  254.  252.  25
5.
  255.  252.  255.  254.  255.  255.  254.  255.  255.  255.  254.  25
5.
  254.  255.  253.  255.  253.  254.  255.  253.  255.  252.  255.  25
5.
  255.  254.  252.  187.  255.  255.  255.  249.  255.  254.  211.  13
3.
  253.  255.  255.  255.  253.  228.  172.  146.  255.  255.  253.  25
4.
  225.  184.  187.  184.]
```

In [16]:
```
validate_noisy_patch = np.empty(shape=(numPatches, numPixels))

# Noisy Patches
sigma = 25

for i in range(len(validate_patch)):
    patch = validate_patch[i]
    noise = np.random.random(patch.shape) * sigma
    newPatch = np.add(patch,noise)
    validate_noisy_patch[i] = np.clip(newPatch, 0, 255)
```

In [17]:
```
print validate_noisy_patch[0]
```

```
[ 209.04771447  255.          255.          255.          255.
255.
  253.59825248  255.          255.          255.          255.
255.
  255.          255.          255.          255.          255.
255.
  255.          255.          255.          255.          255.
255.
  255.          255.          255.          255.          255.
255.
  255.          255.          255.          255.          255.
255.
  255.          255.          255.          202.95793216  255.
255.
  255.          255.          255.          255.          211.55171391
  146.8584996   255.          255.          255.          255.
255.
  244.4638734   190.17535621  155.42413838  255.          255.
255.
  255.          247.8327585   186.79940819  198.46013997  187.56408489
]
```

Now creating Clean and Noisy Test patches of shape shape (480000, 64).

In [18]:
```
windowSize = (8,8)
stepSize = 8
#numPatches = 5*2400
numPatches = 480000
numPixels = 64

test_patch = np.empty(shape=(numPatches, numPixels))

k=0
for i in range(len(x_test)):
    image = x_test[i]
    for y in xrange(0, image.shape[0]-8-1, stepSize):
        for x in xrange(0, image.shape[1]-8-1, stepSize):
            test_patch[k] = image[y:y + windowSize[1], x:x + windowSize[0
            k = k+1
```

In [19]:
```
print test_patch.shape
```

```
(480000, 64)
```

In [20]:
```
print test_patch[0]
```

```
[  68.   69.   70.   71.   71.   71.   70.   70.   64.   66.   68.   7
0.
   69.   68.   68.   68.   66.   69.   73.   74.   73.   72.   73.   7
4.
   81.   83.   85.   86.   86.   87.   89.   91.  106.  105.  105.  10
5.
  107.  109.  112.  114.  122.  120.  118.  117.  118.  121.  123.  12
4.
  123.  122.  120.  119.  118.  118.  118.  118.  126.  126.  125.  12
3.
  121.  118.  116.  115.]
```

In [21]:
```
test_noisy_patch = np.empty(shape=(numPatches, numPixels))

# Noisy Patches
sigma = 25

for i in range(len(test_patch)):
    patch = test_patch[i]
    noise = np.random.random(patch.shape) * sigma
    newPatch = np.add(patch,noise)
    test_noisy_patch[i] = np.clip(newPatch, 0, 255)
```

In [22]:
```
print test_noisy_patch.shape
```

```
(480000, 64)
```

In [23]:
```
print test_noisy_patch[0]
```

```
[  69.8825226    88.9454007    71.75974664   95.89443798   73.56097471
   77.30254421   94.45172418   88.63319045   77.49728974   85.03947036
   90.91698649   81.12876598   75.10840943   73.65491074   70.33335455
   81.35020673   82.27549456   93.60554131   88.84674512   94.96799633
   76.87117045   76.89172666   90.91571872   86.15355358  100.653349
   85.05844962  103.19592927   88.8988022   104.03830643  110.2727726
  105.67517021   98.78557608  111.49248707  116.75635095  122.72021695
  108.68634919  121.40105804  130.93875668  119.66934557  127.9529882
  124.97947127  142.68872884  119.65153861  134.40535979  126.06842533
  140.57910437  144.21381653  147.41734104  135.39619264  131.13019447
  134.6437368   143.82939165  123.22631089  129.01074183  124.50801891
  122.00851116  135.38097067  133.31102122  128.69968748  133.44744292
  130.24687199  122.3277565   133.17853354  129.80312488]
```

SINGLE AUTOENCODER MODELS

Autoencoder Model 1

Train Model 1 - Fit model with random input weights and save output weights

Model takes flattened 8x8 patches, encodes to size 16x16, and decodes to size 8x8.

Parameters:

- verbose = 0 (to prevent keras bug "model.fit ValueError: I/O operation on closed file")
- optimizer = 'adadelta'
- loss = 'mean_squared_error'
- encoder activation = 'relu'
- decoder activation = 'sigmoid'

In [24]:
```
# this is the size of our encoded representations
encoding_dim = 256

# this is our input placeholder
input_img = Input(shape=(64,))
# "encoded" is the encoded representation of the input
encoded_model1 = Dense(encoding_dim, activation='relu', kernel_initialize
                bias_initializer='zeros')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded_model1 = Dense(64, activation='sigmoid', kernel_initializer='rand
                bias_initializer='zeros')(encoded_model1)

# this model maps an input to its reconstruction
autoencoder_model1 = Model(input_img, decoded_model1)
```

In [25]:
```
autoencoder_model1.compile(optimizer='adadelta', loss='mean_squared_error
```

In [26]:
```
train_patch = train_patch / 255.
validate_patch = validate_patch / 255.
test_patch = test_patch / 255.
train_noisy_patch = train_noisy_patch / 255.
test_noisy_patch = test_noisy_patch / 255.
validate_noisy_patch = validate_noisy_patch / 255.
```

In [27]:
```
history = autoencoder_model1.fit(train_patch, train_patch,
                                 epochs=35,
                                 batch_size=1000,
                                 shuffle=True,
                                 validation_data=(validate_patch,validate_patch)
```

```
Train on 480000 samples, validate on 240000 samples
Epoch 1/35
480000/480000 [==============================] - 6s - loss: 0.0490 - a
cc: 0.0186 - val_loss: 0.0250 - val_acc: 0.0259
Epoch 2/35
480000/480000 [==============================] - 6s - loss: 0.0118 - a
cc: 0.0577 - val_loss: 0.0087 - val_acc: 0.0535
Epoch 3/35
480000/480000 [==============================] - 5s - loss: 0.0078 - a
cc: 0.0597 - val_loss: 0.0078 - val_acc: 0.0675
Epoch 4/35
480000/480000 [==============================] - 5s - loss: 0.0071 - a
cc: 0.0679 - val_loss: 0.0071 - val_acc: 0.0738
Epoch 5/35
480000/480000 [==============================] - 5s - loss: 0.0064 - a
cc: 0.0735 - val_loss: 0.0064 - val_acc: 0.0777
Epoch 6/35
480000/480000 [==============================] - 5s - loss: 0.0058 - a
cc: 0.0787 - val_loss: 0.0059 - val_acc: 0.0832
Epoch 7/35
480000/480000 [==============================] - 5s - loss: 0.0053 - a
cc: 0.0840 - val_loss: 0.0055 - val_acc: 0.0899
Epoch 8/35
480000/480000 [==============================] - 5s - loss: 0.0050 - a
cc: 0.0903 - val_loss: 0.0052 - val_acc: 0.0957
Epoch 9/35
480000/480000 [==============================] - 5s - loss: 0.0048 - a
cc: 0.0962 - val_loss: 0.0050 - val_acc: 0.1005
Epoch 10/35
480000/480000 [==============================] - 5s - loss: 0.0046 - a
cc: 0.1021 - val_loss: 0.0049 - val_acc: 0.1082
Epoch 11/35
480000/480000 [==============================] - 5s - loss: 0.0045 - a
cc: 0.1079 - val_loss: 0.0047 - val_acc: 0.1138
Epoch 12/35
480000/480000 [==============================] - 5s - loss: 0.0043 - a
cc: 0.1137 - val_loss: 0.0046 - val_acc: 0.1194
Epoch 13/35
480000/480000 [==============================] - 5s - loss: 0.0042 - a
cc: 0.1192 - val_loss: 0.0045 - val_acc: 0.1263
Epoch 14/35
480000/480000 [==============================] - 5s - loss: 0.0041 - a
cc: 0.1243 - val_loss: 0.0043 - val_acc: 0.1320
Epoch 15/35
480000/480000 [==============================] - 5s - loss: 0.0040 - a
```

```
cc: 0.1296 - val_loss: 0.0042 - val_acc: 0.1371
Epoch 16/35
480000/480000 [==============================] - 5s - loss: 0.0038 - a
cc: 0.1343 - val_loss: 0.0041 - val_acc: 0.1448
Epoch 17/35

480000/480000 [==============================] - 5s - loss: 0.0037 - a
cc: 0.1391 - val_loss: 0.0040 - val_acc: 0.1468
Epoch 18/35
480000/480000 [==============================] - 6s - loss: 0.0036 - a
cc: 0.1431 - val_loss: 0.0039 - val_acc: 0.1497
Epoch 19/35
480000/480000 [==============================] - 7s - loss: 0.0035 - a
cc: 0.1481 - val_loss: 0.0038 - val_acc: 0.1556
Epoch 20/35
480000/480000 [==============================] - 5s - loss: 0.0034 - a
cc: 0.1522 - val_loss: 0.0037 - val_acc: 0.1600
Epoch 21/35
480000/480000 [==============================] - 5s - loss: 0.0034 - a
cc: 0.1568 - val_loss: 0.0036 - val_acc: 0.1657
Epoch 22/35
480000/480000 [==============================] - 5s - loss: 0.0033 - a
cc: 0.1608 - val_loss: 0.0036 - val_acc: 0.1681
Epoch 23/35
480000/480000 [==============================] - 7s - loss: 0.0032 - a
cc: 0.1644 - val_loss: 0.0035 - val_acc: 0.1739
Epoch 24/35
480000/480000 [==============================] - 6s - loss: 0.0031 - a
cc: 0.1676 - val_loss: 0.0034 - val_acc: 0.1773
Epoch 25/35
480000/480000 [==============================] - 5s - loss: 0.0031 - a
cc: 0.1711 - val_loss: 0.0034 - val_acc: 0.1808
Epoch 26/35
480000/480000 [==============================] - 5s - loss: 0.0030 - a
cc: 0.1745 - val_loss: 0.0033 - val_acc: 0.1851
Epoch 27/35
480000/480000 [==============================] - 5s - loss: 0.0030 - a
cc: 0.1778 - val_loss: 0.0032 - val_acc: 0.1859
Epoch 28/35
480000/480000 [==============================] - 5s - loss: 0.0029 - a
cc: 0.1812 - val_loss: 0.0032 - val_acc: 0.1906
Epoch 29/35
480000/480000 [==============================] - 5s - loss: 0.0029 - a
cc: 0.1842 - val_loss: 0.0031 - val_acc: 0.1937
Epoch 30/35
480000/480000 [==============================] - 5s - loss: 0.0028 - a
cc: 0.1874 - val_loss: 0.0031 - val_acc: 0.1968
Epoch 31/35
480000/480000 [==============================] - 5s - loss: 0.0028 - a
cc: 0.1909 - val_loss: 0.0030 - val_acc: 0.2004
```

```
Epoch 32/35
480000/480000 [==============================] - 6s - loss: 0.0027 - a
cc: 0.1940 - val_loss: 0.0030 - val_acc: 0.2031
Epoch 33/35
480000/480000 [==============================] - 6s - loss: 0.0027 - a
cc: 0.1971 - val_loss: 0.0029 - val_acc: 0.2059
Epoch 34/35
480000/480000 [==============================] - 6s - loss: 0.0026 - a
cc: 0.2001 - val_loss: 0.0029 - val_acc: 0.2079
Epoch 35/35
480000/480000 [==============================] - 5s - loss: 0.0026 - a
cc: 0.2033 - val_loss: 0.0028 - val_acc: 0.2132
```

In [28]:

```
# save the weights
weights_model1 =[]
weights_model1.append(autoencoder_model1.get_weights())

print len(weights_model1[0])
```

```
4
```

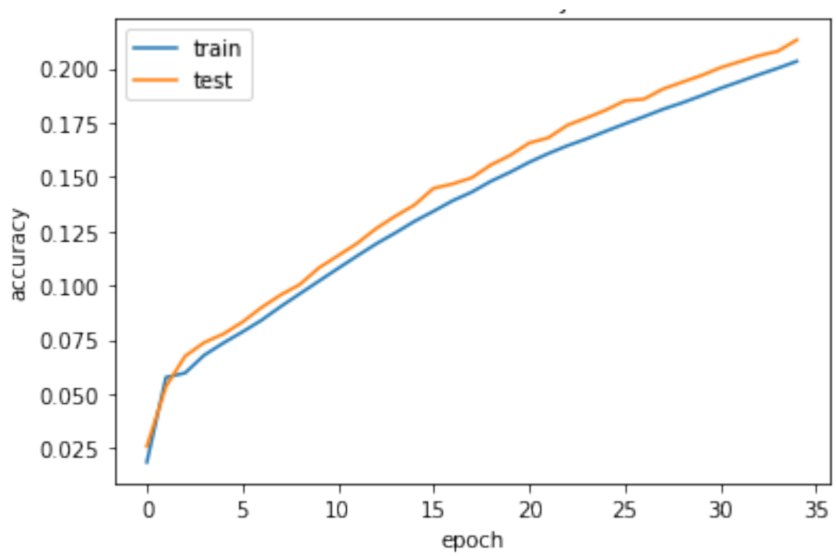
Plot the Loss from Model 1

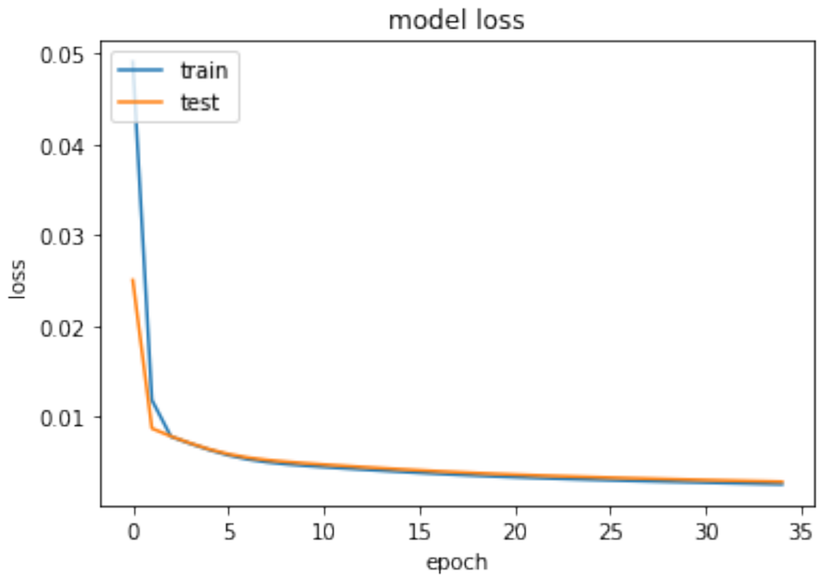
In [29]:

```
import matplotlib.pyplot as plt

print(history.history.keys())

# summary for accuracy

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()

# summary 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()
```

```
['acc', 'loss', 'val_acc', 'val_loss']
```

model accuracy





Feedforward Model 1

Feed clean and noisy patches through the input and encoding layers of model 1.

Note: The weights of the encoding layer must equal the encoding layer weights obtained from the trained model 1.
This model will transform your flattened 8x8 images to flattened 16x16 images, which will be used as input to Autoencoder Model 2.

In [30]:
```
# this model maps an input to its encoded representation
encoder_model1 = Model(input_img, encoded_model1)

#load weights from previous model
encoder_model1.set_weights([weights_model1[0][0], weights_model1[0][1]])

#compile the model
encoder_model1.compile(optimizer='adadelta', loss='mean_squared_error')

# save the O/P 16x16
encoder_clean = encoder_model1.predict(train_patch)

print encoder_clean.shape

# save the O/P 16x16
encoder_noisy = encoder_model1.predict(train_noisy_patch)

print encoder_noisy.shape
```

```
(480000, 256)
(480000, 256)
```

Autoencoder Model 2

Train Model 2 - Fit model with random input weights and save output weights

Model takes flattened 16x16 patches, encodes to size 32x32, and decodes to size 16x16.

Parameters:

- verbose = 0 (to prevent keras bug "model.fit ValueError: I/O operation on closed file")
- optimizer = 'adadelta'
- loss = 'mean_squared_error'
- encoder activation = 'relu'
- decoder activation = 'sigmoid'

In [34]:
```
# this is the size of our encoded representations
encoding_dim = 1024

# this is our input placeholder
input_img = Input(shape=(256,))

# "encoded" is the encoded representation of the input
encoded_model2 = Dense(encoding_dim, activation='relu', kernel_initialize
                bias_initializer='zeros')(input_img)

# "decoded" is the lossy reconstruction of the input
decoded_model2 = Dense(256, activation='sigmoid', kernel_initializer='ran
                bias_initializer='zeros')(encoded_model2)

# this model maps an input to its reconstruction
autoencoder_model2 = Model(input_img, decoded_model2)
```

In [35]:
```
autoencoder_model2.compile(optimizer='adadelta', loss='mean_squared_error
```

In [36]:
```
history = autoencoder_model2.fit(encoder_clean,
                                 encoder_clean,
                                 batch_size=1000,
                                 epochs=40,
                                validation_split=0.33)
```

```
Train on 321599 samples, validate on 158401 samples
Epoch 1/40
321599/321599 [==============================] - 31s - loss: 0.0535 -
val_loss: 0.0338
Epoch 2/40
321599/321599 [==============================] - 31s - loss: 0.0386 -
val_loss: 0.0288
Epoch 3/40
321599/321599 [==============================] - 31s - loss: 0.0374 -
val_loss: 0.0276
Epoch 4/40
321599/321599 [==============================] - 31s - loss: 0.0369 -
val_loss: 0.0269
Epoch 5/40
321599/321599 [==============================] - 32s - loss: 0.0365 -
val_loss: 0.0265
Epoch 6/40
321599/321599 [==============================] - 38s - loss: 0.0362 -
val_loss: 0.0261
Epoch 7/40
```

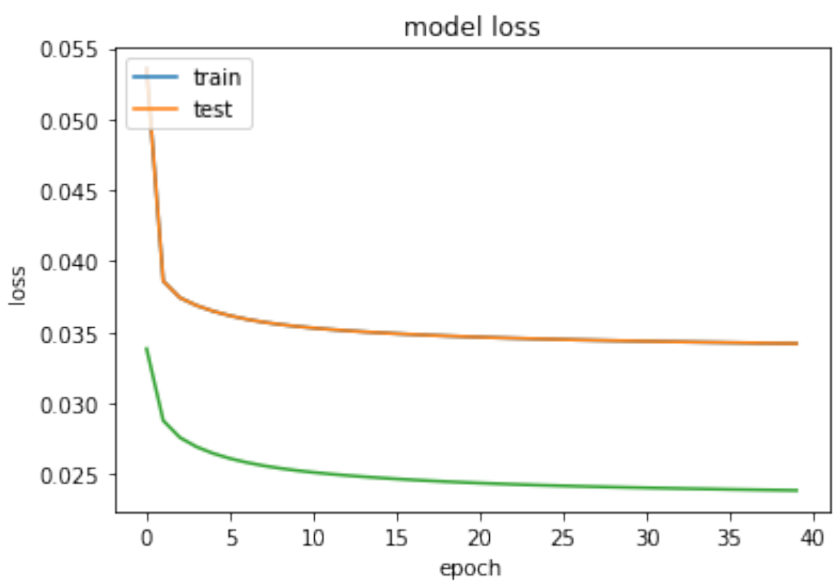
In [37]:
```
autoencoder_model2.save_weights("/Users/divyaagarwal/Desktop/Computervisi
```

In [38]:
```
# save the weights
weights_model2 =[]
weights_model2.append(autoencoder_model2.get_weights())

print len(weights_model2[0])
```

```
4
```

In [42]:
```
import matplotlib.pyplot as plt

print(history.history.keys())

plt.plot(history.history['loss'])

# summary 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()
```

```
['loss', 'val_loss']
```



Feedforward Model 2

Feed the clean and noisy patches from the feedforward model 1 through the input and encoding layers of model 2.

Note: The weights of the encoding layer must equal the encoding layer weights obtained from the trained model 2.
This model will transform your flattened 16x16 images to flattened 32x32 images, which can be used for testing and debugging the trained model.

In [43]:
```
# this is the size of our encoded representations
encoding_dim = 1024
input_img = Input(shape=(256,))

encoded_model2 = Dense(encoding_dim, activation='relu')(input_img)

# this model maps an input to its encoded representation
encoder_model2 = Model(input_img, encoded_model2)

#set weights
encoder_model2.set_weights([weights_model2[0][0], weights_model2[0][1]])

#compile
encoder_model2.compile(optimizer='adadelta', loss='mean_squared_error')

# save the O/P 32x32
encoder_clean2 = encoder_model2.predict(encoder_clean)

print encoder_clean2.shape

# save the O/P 32x32
encoder_noisy2 = encoder_model2.predict(encoder_noisy)

print encoder_noisy2.shape
```

```
(480000, 1024)
(480000, 1024)
```

STACKED SPARSE DENOISING AUTOENCODER

Train the SSDA

The SSDA model will take patches of size 8x8 as input.

The first encoding layer expands an 8x8 patch to 16x16. The second encoding layer expands a 16x16 patch to 32x32. We then introduce the two decoding layers, which decrease the image from 32x32 to 16x16 to 8x8. The purpose of our model is take a noisy 8x8 image as input and return a clean 8x8 image as output.

The weights of the two encoding layers must be initialized with the encoding layer weights that were saved from the corresponding single autoencoder models above.

In [44]:
```
input_img1 = Input(shape=(64,))

encoded1 = Dense(256, activation='relu')(input_img1)

encoded2 = Dense(1024, activation='relu')(encoded1)

# "decoded" used encoded i/p of noisy data 32x32 dim
decoded1 = Dense(256, activation='sigmoid', kernel_initializer='random_no
                bias_initializer='zeros')(encoded2)

decoded2 = Dense(64, activation='sigmoid', kernel_initializer='random_nor
                bias_initializer='zeros')(decoded1)
```

In [46]:
```
# send the model
autoencoder_ssda = Model(input_img1, decoded2)


# set the weights 1
autoencoder_ssda.layers[1].set_weights([weights_model1[0][0], weights_mod
# set the weights 2
autoencoder_ssda.layers[2].set_weights([weights_model2[0][0], weights_mod

# compile it now
autoencoder_ssda.compile(optimizer='adadelta', loss='mean_squared_error',

# weights_ssda =[]
# weights_ssda.append(autoencoder_ssda.get_weights())

# autoencoder_ssda.set_weights([weights_model1[0][0],
#                             weights_model1[0][1],
#                             weights_model2[0][0],
#                             weights_model2[0][1],
#                             weights_ssda[0][4],
#                             weights_ssda[0][5],
#                             weights_ssda[0][6],
#                             weights_ssda[0][7]])
```

In [47]:
```
history = autoencoder_ssda.fit(train_noisy_patch,
                               train_patch,
                               batch_size=1000,
                               epochs=40,
                               validation_data=(validate_noisy_patch, val
```

```
Train on 480000 samples, validate on 240000 samples
Epoch 1/40
480000/480000 [==============================] - 65s - loss: 0.0165 -
acc: 0.0231 - val_loss: 0.0094 - val_acc: 0.0339
Epoch 2/40
480000/480000 [==============================] - 54s - loss: 0.0080 -
acc: 0.0537 - val_loss: 0.0081 - val_acc: 0.0521
Epoch 3/40
480000/480000 [==============================] - 57s - loss: 0.0069 -
acc: 0.0510 - val_loss: 0.0069 - val_acc: 0.0517
Epoch 4/40
480000/480000 [==============================] - 77s - loss: 0.0059 -
acc: 0.0533 - val_loss: 0.0061 - val_acc: 0.0559
Epoch 5/40
480000/480000 [==============================] - 57s - loss: 0.0054 -
acc: 0.0576 - val_loss: 0.0057 - val_acc: 0.0590
Epoch 6/40
480000/480000 [==============================] - 55s - loss: 0.0051 -
acc: 0.0617 - val_loss: 0.0054 - val_acc: 0.0638
```

Save SSDA Model Weights

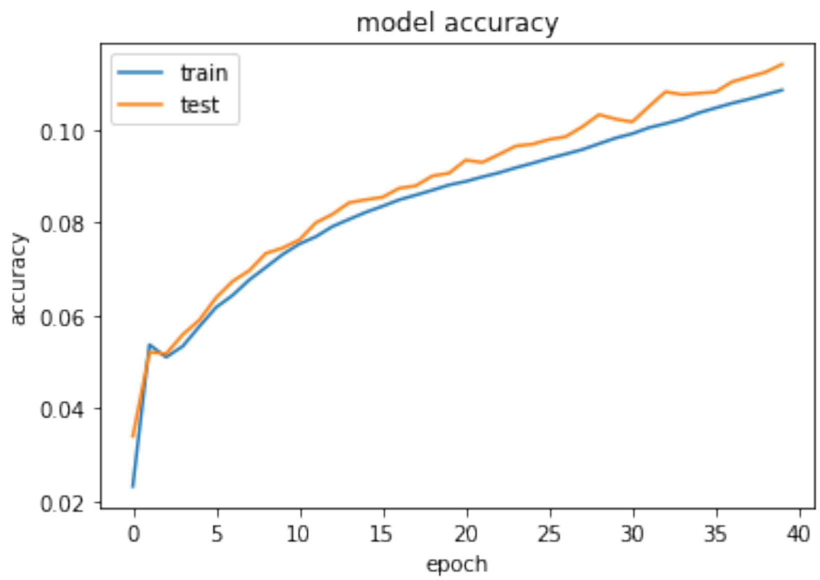
In [49]:
```
autoencoder_ssda.save_weights("/Users/divyaagarwal/Desktop/Computervision
```

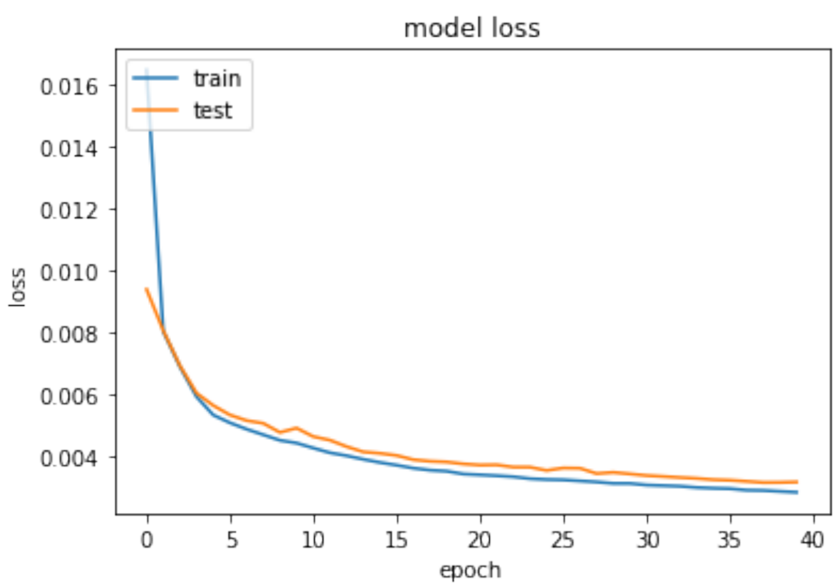
Plot the Loss from SSDA

In [50]:
```
import matplotlib.pyplot as plt

print(history.history.keys())

# summarize history for accuracy

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()
```

```
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()
```

```
['acc', 'loss', 'val_acc', 'val_loss']
```





Run SSDA on Noisy Test Images and Calculate Mean Squared Error

In [51]:
```
final_prediction = autoencoder_ssda.predict(test_noisy_patch)
```

In [52]:
```
autoencoder_ssda.save_weights("/Users/divyaagarwal/Desktop/Computervision
#mean_squared_error(test_patch, final_prediction)
```

VISUALIZE RESULTS

Use the image 119082.jpg from the validate image set to visualize effectiveness of the SSDA model.

Load and Display the image

In [53]:
```
import matplotlib.pyplot as plt
img = validate[10]
print img
plt.imshow(img,  cmap='gray')
plt.axis('off')
plt.show()
```

```
[[ 38  37  37 ..., 171 170 170]
 [ 31  30  30 ..., 169 170 171]
 [ 29  26  25 ..., 171 170 170]
 ...,
 [255 243 231 ..., 248 253 246]
 [125  71  41 ..., 252 255 246]
 [ 66  77  55 ..., 206 182 114]]
```

Create patches from the image

Select a patch with significant contrast (for example, a clearly defined vertical or horizontal line).

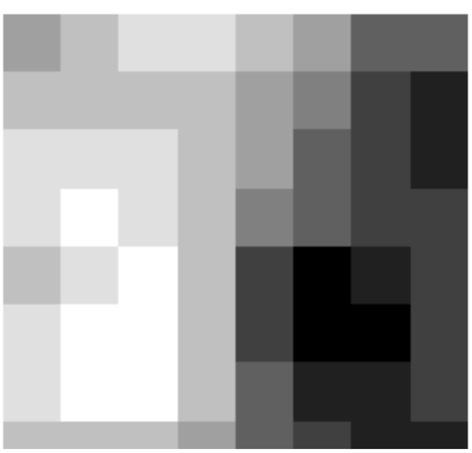
In [54]:
```
clean_patch_img = validate_patch[2400*9 : 2400*10 -1]
print clean_patch_img[0]
```

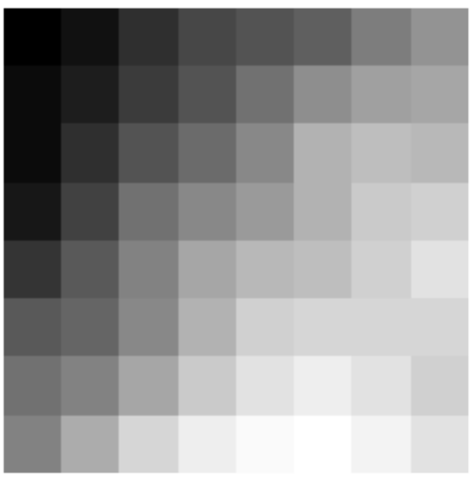
```
[ 0.23529412  0.23921569  0.24313725  0.24313725  0.23921569  0.235294
12
  0.22745098  0.22745098  0.23921569  0.23921569  0.23921569  0.239215
69
  0.23529412  0.23137255  0.22352941  0.21960784  0.24313725  0.243137
25
  0.24313725  0.23921569  0.23529412  0.22745098  0.22352941  0.219607
84
  0.24313725  0.24705882  0.24313725  0.23921569  0.23137255  0.227450
98
  0.22352941  0.22352941  0.23921569  0.24313725  0.24705882  0.239215
69
  0.22352941  0.21568627  0.21960784  0.22352941  0.24313725  0.247058
82
  0.24705882  0.23921569  0.22352941  0.21568627  0.21568627  0.223529
41
  0.24313725  0.24705882  0.24705882  0.23921569  0.22745098  0.219607
84
  0.21960784  0.22352941  0.23921569  0.23921569  0.23921569  0.235294
12
  0.22745098  0.22352941  0.21960784  0.21960784]
```

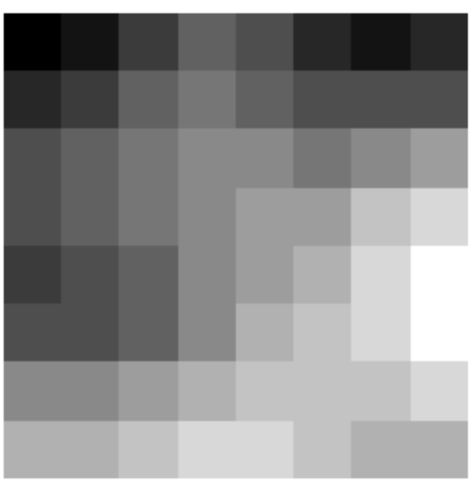
In [55]:
```
noisy_patch_img = validate_noisy_patch[2400*9 : 2400*10 -1]
```

Display the clean patch

In [56]:
```
for i in range(4):
    patch = clean_patch_img[i]
    patch = patch.reshape(8,8) * 255.0
    plt.imshow(patch,  cmap='gray')
    plt.axis('off')
    plt.show()
```

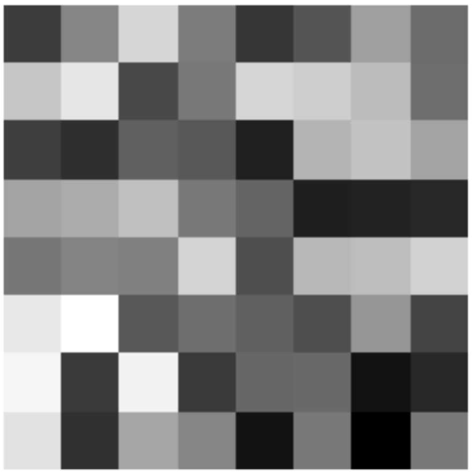


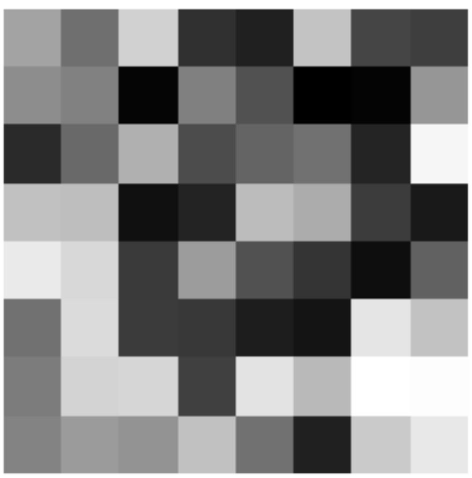


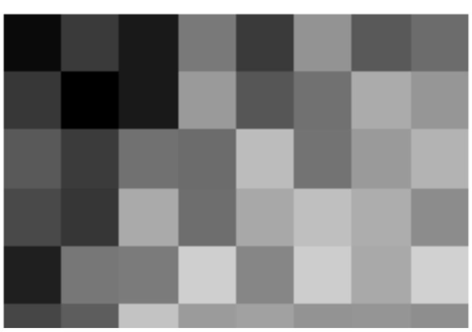


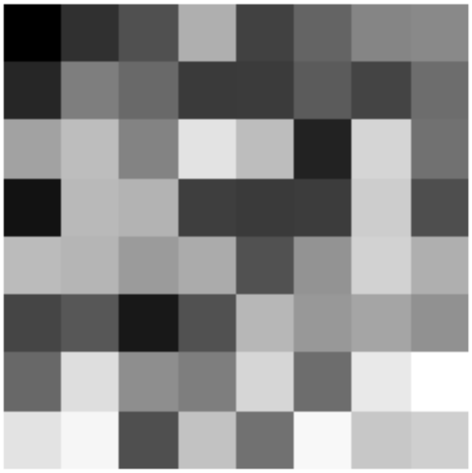
Display the noisy patch

In [57]:
```
for i in range(4):
    patch = noisy_patch_img[i]
    patch = patch.reshape(8,8) * 255.0
    plt.imshow(patch,  cmap='gray')
    plt.axis('off')
    plt.show()
```







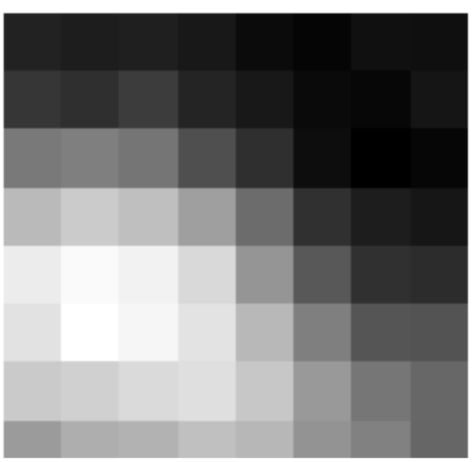


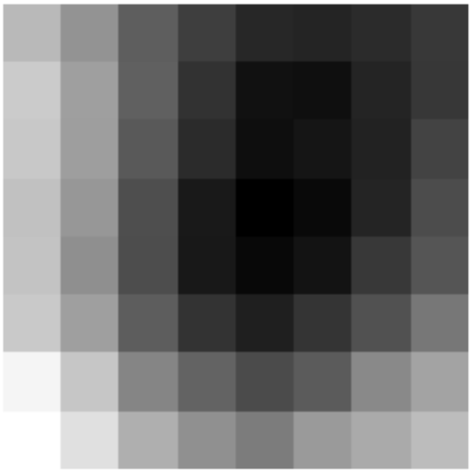
Feed the noisy patch through the trained SSDA model and display the resulting denoised image

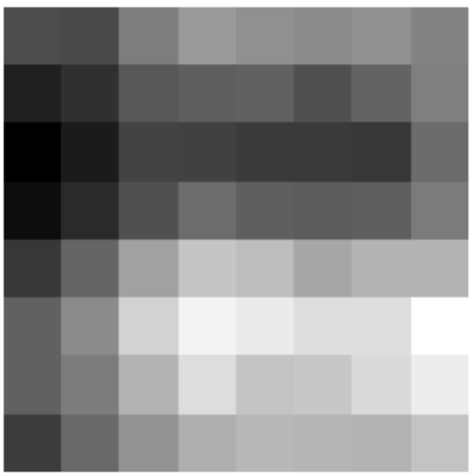
Does the denoised patch resemble the clean patch?

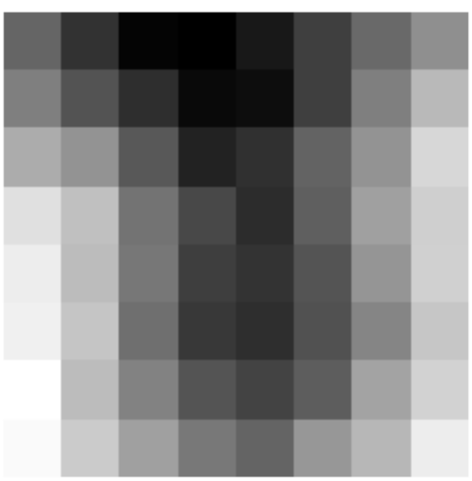
In [58]:
```
output = final_prediction[2400*9 : 2400*10-1]
```

In [59]:
```
for i in range(4):
    patch = output[i]
    patch = patch.reshape(8,8) * 255.0
    plt.imshow(patch,  cmap='gray')
    plt.axis('off')
    plt.show()
```









Plot a single image patch and compare

ANALYZE RESULTS

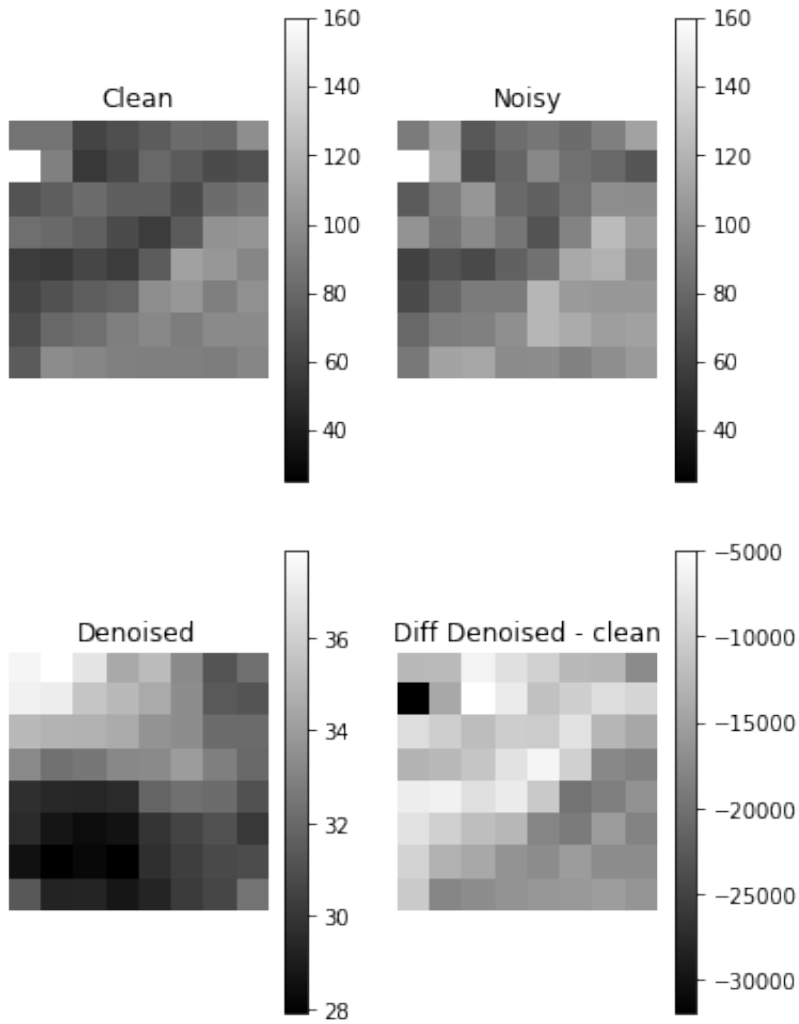
Comment on the Effectiveness of your model.
Can you improve the model by changing any of the parameters?

In [68]:
```
print output[1234].reshape(8,8) * 255
```

```
[[ 37.47543716  37.89933395  36.84737015  34.51482773  35.21337891
   33.31004715  31.19306374  32.2495079 ]
 [ 37.3486557   37.1876297   35.6241684   35.13164139  34.55157852
   33.40540314  31.43386269  31.19120026]
 [ 35.12536621  34.93384552  34.81932831  34.60026932  33.70185471
   33.41449738  32.11282349  32.08383179]
 [ 33.25758743  32.35405731  32.54502869  33.20717621  33.28340149
   34.02289963  32.88191223  31.97543907]
 [ 29.74655914  29.48270226  29.38748741  29.52987671  31.82883072
   32.26580048  32.11250687  31.10915756]
 [ 29.54504776  28.71029854  28.3489666   28.59669495  29.91965485
   30.5398941   31.06070328  30.09921074]
 [ 28.55385017  27.87735748  28.20727158  27.90884018  29.74308014
   30.32805252  30.73750305  30.86085129]
 [ 31.33398628  29.25794983  29.29073715  28.75881386  29.35293961
   30.19287682  30.62119484  32.3893013 ]]
```

In [81]:
```
#add vmax and vmin : todo

plt.figure(1)
ax =plt.subplot(121)
patch_clean = clean_patch_img[1234].reshape(8,8) * 255.0
plt.imshow(patch_clean, cmap='gray', vmin=25, vmax=160)
ax.set_title('Clean')
plt.axis('off')
plt.colorbar()

ax = plt.subplot(122)
patch_noisy = noisy_patch_img[1234].reshape(8,8) * 255.0
plt.imshow(patch_noisy, cmap='gray', vmin=25, vmax=160)
ax.set_title('Noisy')
plt.axis('off')
plt.colorbar()
plt.show()

ax = plt.subplot(121)
```

```
patch_denoised = output[1234].reshape(8,8) * 255.0
plt.imshow(patch_denoised, cmap='gray')
ax.set_title('Denoised')
plt.axis('off')
plt.colorbar()

ax =plt.subplot(122)
patch_noisy_clean_diff = patch_denoised - patch_clean
patch_noisy_clean_diff = patch_noisy_clean_diff.reshape(8,8) * 255.0
plt.imshow(patch_noisy_clean_diff, cmap='gray') #vmin=-10000, vmax=-60000
ax.set_title('Diff Denoised - clean')
plt.colorbar()
plt.axis('off')
plt.show()
```



In [73]:
```
#Calculate psnr 1
psnr1 = -10. * np.log10(np.mean(np.square(patch_denoised - patch_clean)))
```

In [74]:
```
#calculate psnr 2
psnr2 = -10. * np.log10(np.mean(np.square(patch_noisy - patch_clean)))
```

In [75]:
```
print "patch_denoised - patch_clean"
print psnr1
print "patch_noisy - patch_clean"
print psnr2
```

```
patch_denoised - patch_clean
-34.5942049317
patch_noisy - patch_clean
-22.7518845712
```

The model could be more effective if it could be trained on higher epoch.
By increasing the epoch the efficincy can be increased. We might we overfitting with the small dataset, we could improve and avoid overfitting it by using a bigger datasets. Also using a bigger CNN might increase the accuracy and decrease the loss.