Someone asked me how create an Auto-Associative Neural Network in a different discussion group, so I thought I'd use the TGS Salt contest data to explain.

An auto-associative network has the same number of input dimensions as it does output dimensions, because it is being trained to precisely recreate the input at the output. This is a boring math exercise, except in the case when some middle layer of the feed forward network has much fewer dimensions than the input.

[many dimensions n] → [much fewer dimensions m] → [many dimensions n]

This has the effect of compressing the original data from n dimensions down to m dimensions. The auto-associative network is therefore a CODEC, with the encoder being the front half, and the decoder being the back half.

It also provides the researcher a qualitative way of picking a lower bound on the number of dimensions for internal layers of their feed-forward networks.

How? Just try different values for m, and then look at the output. If you can no longer recognize (classify) the output, then probably neither can a deep learning algorithm, and so m is probably too small.

Make sure that the smallest layer in your network (through which all data must pass) is larger than that value of m you found to be "too small."

- References - originally forked from AshishPatel and collaborator: Have you check this approach... 🎶 🎶 🎶? - forked from seasalt or not??? by AshishPatel (+0/–0)
- P.S. The code also shows a method of taking the seismic 2d image data and turning it into a series of 1d columns. I was working on this as part of my contest entry. I was experimenting with doing some first-pass signal analysis transformations on the seismic data as the one-dimensional time varying data that it is (the time varying signal of the bounced back sound) like an ultrasound. I wanted to sanity check not only internal network dimensions, but also any extractable features that would make the classification run smoother.

Start with some included components:

Hide code

In [1]:
```
import os
import sys
import random
import warnings
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline
import cv2
from tqdm import tqdm_notebook, tnrange
from itertools import chain

from skimage.io import imread, imshow, concatenate_images
from skimage.transform import resize
from skimage.morphology import label

from keras.models import Model, load_model
from keras.layers import Input
from keras.layers.core import Lambda
from keras.layers.convolutional import Conv2D, Conv2DTranspose
from keras.layers.pooling import MaxPooling2D
from keras.layers.merge import concatenate
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras import backend as K
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D

import tensorflow as tf
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Flatten, Conv2D, Dropout, SpatialDropout2D

from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img
```

```
/opt/conda/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the secon
d argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treate
d as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Limit the training set to only 500 images for speed. Feel free to use the whole set (buy commenting out the last line below), etc...

Hide code

In [2]:
```
im_width = 101
im_height = 101
im_chan = 1
path_train = '../input/train/'

# create a "list" of filenames of all the image files
train_ids = next(os.walk(path_train+"images"))[2]
# LIMIT THE SIZE OF THE TRAINING AND TESTING SETS FOR SPEED - Comment out the below to use the whole se
t
train_ids=train_ids[:500]
```

Instead of using the 101×101 image graphics from the contest, will only use individual columns of the data (101×1), thereby creating one-hundred and one times more examples.

Apologies for not using TensorFlow below and saving a few pennies of electricity (would have been much quicker, but for this small data set it's OK)

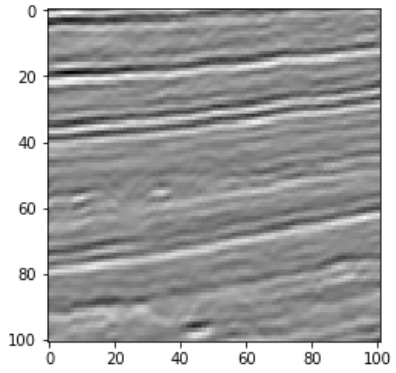
In [3]:
```
X_train = np.zeros((len(train_ids) * im_width, im_height,1), dtype=np.uint8)
print('Loading train images and converting them into columns... ')
sys.stdout.flush()

# for loop counts n, and also pulls from train_ids list using "progress bar" tqdm_notebook()
for n, id_ in tqdm_notebook(enumerate(train_ids), total=len(train_ids)):
    path = path_train
    img = load_img(path + '/images/' + id_)
    x = img_to_array(img)[:,:,1]
    for y in range(0,im_width):
        for z in range(0,im_height):
            X_train[n*im_width + y,z] = x[z,y]

print(X_train.shape)
print('Done!')
```

```
Loading train images and converting them into columns...



(50500, 101, 1)
Done!
```

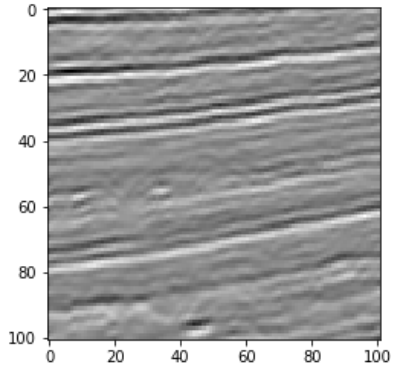
Pick random image and make sure the columns reconstruct to look the same (sanity check).

In [4]:
```
print("SANITY CHECK")
# Check if training data looks all right
# by reconstructing an image from the columns
ix = random.randint(0, len(train_ids))
print("original image")
original_image = load_img(path_train + '/images/' + train_ids[ix])
plt.imshow(original_image)
plt.show()

print("reconstructed columns image")
ix = int(ix*im_width)
image = np.zeros((im_width,im_height), dtype=np.uint8)
for x in range(0,im_width):
    for y in range(0,im_height):
        image[y,x]=X_train[ix+x,y]
plt.imshow(np.dstack((image, image, image)))
plt.show()

plt.show()
```

```
SANITY CHECK
original image
```




```
reconstructed columns image
```



Auto-Associatve Network Example

- Input (columns of data 101 x 1)
- Middle layers 101 or fewer "relu" activation function perceptrons
- Output layer - 101 "linear" activated perceptrons

In [5]:
```
# use a simple sequential model
model = Sequential()
# this first layer (connecting the input to the output layer) will have fewer than 101 neurons.

#################################################################################################
#     CHANGE THE FIRST PARAMETER (20) BELOW                                                     #
#     THIS IS THE VALUE OF m                                                                    #
# TRY A VERY SMALL VALUE (DOWN to 1)                                                            #
#     SEE THAT YOU CAN NO LONGER RECOGNIZE THE IMAGE, MUCH LESS FIND THE SALT                   #
# TRY A VERY LARGE VALUE (UP TO 101)                                                            #
#     SEE THAT THE ORIGINAL IMAGE IS NEARLY PERFECTLY RECREATED                                 #
#################################################################################################
model.add(Dense(20, activation='relu', input_shape=(im_height,)))

# this output layer has to have 101 neurons, and needs to be linearly activated
model.add(Dense(101, activation='linear'))
model.compile(loss='mean_squared_error',
              optimizer='adam')
# Diplay the model summary
print("model summary")
model.summary()
print("layer shapes of weights and bias arrays")
for x in range(0,len(model.layers) - 1):
    print(x)
    print(model.get_layer(index=x+1).get_weights()[0].shape)
    print(model.get_layer(index=x+1).get_weights()[1].shape)
```

```
model summary
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 20)                2040
_________________________________________________________________
dense_2 (Dense)              (None, 101)               2121
=================================================================
Total params: 4,161
Trainable params: 4,161
Non-trainable params: 0
_________________________________________________________________
layer shapes of weights and bias arrays
0
(20, 101)
(101,)
```

In [6]:

```
earlystopper = EarlyStopping(patience=2, verbose=1)
checkpointer = ModelCheckpoint('tgs-salt-columns-autoassoc', verbose=1, save_best_only=True)
results = model.fit(X_train[:,:,0], X_train[:,:,0], validation_split=0.1, batch_size=100, epochs=300,
                    callbacks=[earlystopper, checkpointer])
```

```
Train on 45450 samples, validate on 5050 samples
Epoch 1/300
45450/45450 [==============================] - 3s 56us/step - loss: 2109.7281 - val_loss: 624.5854

Epoch 00001: val_loss improved from inf to 624.58539, saving model to tgs-salt-columns-autoassoc
Epoch 2/300
45450/45450 [==============================] - 2s 39us/step - loss: 620.1900 - val_loss: 474.9817

Epoch 00002: val_loss improved from 624.58539 to 474.98167, saving model to tgs-salt-columns-autoa
ssoc
Epoch 3/300
45450/45450 [==============================] - 2s 39us/step - loss: 479.9183 - val_loss: 389.1242

Epoch 00003: val_loss improved from 474.98167 to 389.12417, saving model to tgs-salt-columns-autoa
ssoc
Epoch 4/300
45450/45450 [==============================] - 2s 39us/step - loss: 404.1405 - val_loss: 340.3691

Epoch 00004: val_loss improved from 389.12417 to 340.36910, saving model to tgs-salt-columns-autoa
ssoc
Epoch 5/300
45450/45450 [==============================] - 2s 38us/step - loss: 359.1691 - val_loss: 304.4398

Epoch 00005: val_loss improved from 340.36910 to 304.43979, saving model to tgs-salt-columns-autoa
ssoc
Epoch 6/300
45450/45450 [==============================] - 2s 39us/step - loss: 328.7294 - val_loss: 282.9373

Epoch 00006: val_loss improved from 304.43979 to 282.93735, saving model to tgs-salt-columns-autoa
ssoc
Epoch 7/300
45450/45450 [==============================] - 2s 39us/step - loss: 309.8895 - val_loss: 270.0114

Epoch 00007: val_loss improved from 282.93735 to 270.01143, saving model to tgs-salt-columns-autoa
ssoc
Epoch 8/300
45450/45450 [==============================] - 2s 38us/step - loss: 299.3494 - val_loss: 264.3641

Epoch 00008: val_loss improved from 270.01143 to 264.36412, saving model to tgs-salt-columns-autoa
ssoc
Epoch 9/300
45450/45450 [==============================] - 2s 39us/step - loss: 294.0230 - val_loss: 262.3719

Epoch 00009: val_loss improved from 264.36412 to 262.37193, saving model to tgs-salt-columns-autoa
ssoc
Epoch 10/300
45450/45450 [==============================] - 2s 38us/step - loss: 291.0695 - val_loss: 259.2924

Epoch 00010: val_loss improved from 262.37193 to 259.29244, saving model to tgs-salt-columns-autoa
ssoc
Epoch 11/300
45450/45450 [==============================] - 2s 38us/step - loss: 288.2272 - val_loss: 241.8902

Epoch 00011: val_loss improved from 259.29244 to 241.89020, saving model to tgs-salt-columns-autoa
ssoc
```

```
Epoch 12/300
45450/45450 [==============================] - 2s 39us/step - loss: 264.7174 - val_loss: 236.5714

Epoch 00012: val_loss improved from 241.89020 to 236.57142, saving model to tgs-salt-columns-autoa
ssoc
Epoch 13/300
45450/45450 [==============================] - 2s 39us/step - loss: 262.6978 - val_loss: 235.0842

Epoch 00013: val_loss improved from 236.57142 to 235.08416, saving model to tgs-salt-columns-autoa
ssoc
Epoch 14/300
45450/45450 [==============================] - 2s 39us/step - loss: 262.3919 - val_loss: 235.3563

Epoch 00014: val_loss did not improve
Epoch 15/300
45450/45450 [==============================] - 2s 38us/step - loss: 261.9016 - val_loss: 234.4776

Epoch 00015: val_loss improved from 235.08416 to 234.47760, saving model to tgs-salt-columns-autoa
ssoc
Epoch 16/300
45450/45450 [==============================] - 2s 38us/step - loss: 261.6483 - val_loss: 233.9125

Epoch 00016: val_loss improved from 234.47760 to 233.91254, saving model to tgs-salt-columns-autoa
ssoc
Epoch 17/300
45450/45450 [==============================] - 2s 39us/step - loss: 261.6905 - val_loss: 234.5067

Epoch 00017: val_loss did not improve
Epoch 18/300
45450/45450 [==============================] - 2s 39us/step - loss: 261.5603 - val_loss: 233.7732

Epoch 00018: val_loss improved from 233.91254 to 233.77319, saving model to tgs-salt-columns-autoa
ssoc
Epoch 19/300
45450/45450 [==============================] - 2s 39us/step - loss: 261.3181 - val_loss: 233.1573

Epoch 00019: val_loss improved from 233.77319 to 233.15734, saving model to tgs-salt-columns-autoa
ssoc
Epoch 20/300
45450/45450 [==============================] - 2s 39us/step - loss: 261.3632 - val_loss: 233.2744

Epoch 00020: val_loss did not improve
Epoch 21/300
45450/45450 [==============================] - 2s 38us/step - loss: 261.2875 - val_loss: 233.1490

Epoch 00021: val_loss improved from 233.15734 to 233.14900, saving model to tgs-salt-columns-autoa
ssoc
Epoch 22/300
45450/45450 [==============================] - 2s 39us/step - loss: 261.2273 - val_loss: 233.2199

Epoch 00022: val_loss did not improve
Epoch 23/300
45450/45450 [==============================] - 2s 39us/step - loss: 261.0716 - val_loss: 233.5473

Epoch 00023: val_loss did not improve
Epoch 00023: early stopping
```

In [7]:
```
# Run all the data through the created network
model = load_model('tgs-salt-columns-autoassoc')
preds_train_t = model.predict(X_train[:,:,0], verbose=1)
```

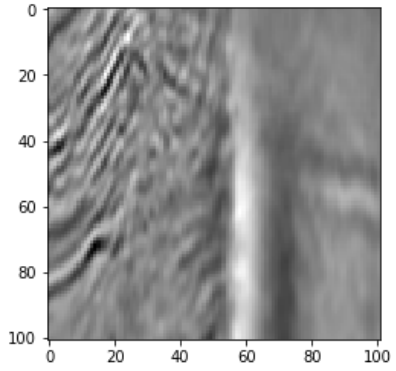
```
50500/50500 [==============================] - 2s 30us/step
```

Let's see how we did! If the auto-associative network worked perfectly, then the output image should look EXACTLY like the input image.

Naturally, this won't be the case if the number of neurons in the "center" layer is less than 101.

In [8]:
```
# RUN THIS ONE A FEW TIMES TO VIEW DIFFERENT IMAGES AND JUDGE FOR YOURSELF
# Pick a random example
ix = random.randint(0, int(len(train_ids)))
ix = int((ix-1)*im_width) # subtract 1 to give radix 0

print("original image from reconstructed input columns")
image = np.zeros((im_width,im_height), dtype=np.uint8)
for x in range(0,im_width):
    for y in range(0,im_height):
        image[y,x]=X_train[ix+x,y]
plt.imshow(np.dstack((image, image, image)))
plt.show()

print("auto-associated image (encoded image) from reconstructed output columns")
pred_mask = np.zeros((im_width,im_height))
for x in range(0,im_width):
    for y in range(0,im_height):
        pred_mask[y,x]=preds_train_t[ix+x,y]
# USE SCALING ONLY FOR AUTO-ASSOC, NOT FOR SALT DETECTION or IF CONTRAST ENHANCEMENT NEEDED
scale = pred_mask.max() - pred_mask.min()
pred_mask = (pred_mask - pred_mask.min()) / scale
plt.imshow(np.dstack((pred_mask, pred_mask, pred_mask)))
plt.show()
```

```
original image from reconstructed input columns
```




```
auto-associated image (encoded image) from reconstructed output columns
```