Imports

In [ ]:
```
import numpy as np
import sys
import os
import tensorflow as tf
import matplotlib.pyplot as plt
from IPython import display
from skimage import io, color
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import TensorflowUtils as utils
%matplotlib inline
```

Space for global training options

In [2]:
```
# width and height of input images
input_size = 128

# batch size for training
batch_size = 40

# number of epochs to train for
epoch_num = 140

# learning rate for training
lr = 1e-4

# for the AdamOptimizer
beta = .9

# directory for input images
input_directory = 'LandscapeData/'

# directory for checkpoints (save / restore models)
checkpoint_directory = "CheckpointsHuber/"

# where to find and store the pre-trained VGG model
model_dir = "VGGModel/"
model_url = 'http://www.vlfeat.org/matconvnet/models/beta16/imagenet-vgg-ver
```

Function that takes the LAB layers and outputs the RGB image

In [3]:
```
# function takes the L, A, B, channels --> concatenates them, and converts t
def labChannelsToRGB(l, a, b):
    l[l > 99] = 99
    new_lab = np.stack((l, a, b), axis=2)
    new_lab = new_lab.astype('float64');
    return color.lab2rgb(new_lab)
```

Function to test an image and output the three relevant images

In [4]:
```
# function takes two images in the LAB-color-scheme and converts them to RGB
def showNetResults(predictedImage, initialImage):

    # get the Black-And-White Version of the Image
    l_img = predictedImage[:,:,0]

    # Convert the Initial Image to RGB
    initialImage   = color.lab2rgb(initialImage.astype('float64'))

    # Convert the Predicted Image to RGB
    predictedImage = color.lab2rgb(predictedImage.astype('float64'))

    # Create a Figure
    fig=plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')

    # Plot the Black-and-White Image
    plt.subplot(1,3,1)
    plt.title("Black and White Image")
    plt.imshow(l_img, cmap='gray')

    # Plot the Reconstructed / Predicted Image
    plt.subplot(1,3,2)
    plt.title("Reconstructed Image")
    plt.imshow(predictedImage)

    # Plot the Original / Ground-Truth Image
    plt.subplot(1,3,3)
    plt.title("Ground Truth Image")
    plt.imshow(initialImage)
```

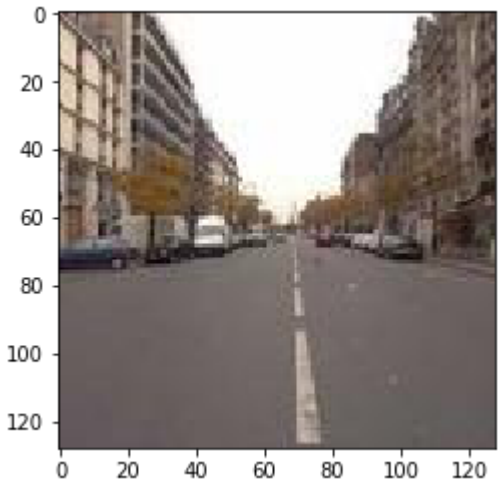
Get the data and split into training, validation, and testing sets

In [5]:
```
# get all of the names of the images in that directory and shuffle the names
input_images = np.asarray([x for x in os.listdir(input_directory) if x.endsw
np.random.shuffle(input_images)
num_inputs = len(input_images)

# separate data into training, validation, and testing
train_X, val_X, test_X = np.split(input_images, [int(.9*len(input_images)),
num_train = len(train_X)
num_val   = len(val_X)
num_test  = len(test_X)
```

Function that takes indices and outputs those images (RGB)

In [6]:
```
# Function just gets the images at the given indices and outputs them
def get_examples(indices):
    output = np.zeros((input_size, input_size, 3, len(indices)), 'uint8')
    for i, n in enumerate(indices):
        im =  io.imread(input_directory + input_images[n])
        output[:,:,:,i] = im
    return output

#  test this function and show the result
ind = np.random.choice(num_inputs, 2)
examples = get_examples(ind)
im = examples[:,:,:,0]
plt.imshow(im)
```

Out[6]: <matplotlib.image.AxesImage at 0x7feaa19fd278>



Function that takes a batch size and type of batch ("train", "val", "test") and outputs the input / output pairs

In [7]:
```
# function to get batch
#     num_samples --> the batch size
#     batch_type  --> can take the values (train, val, test)
def get_batch(num_samples, batch_type):
    assert(num_samples <= 100)

    # Create the input / output arrays to be filled with the proper data typ
    batch_input  = np.zeros((num_samples, input_size, input_size, 1), 'float
    batch_output = np.zeros((num_samples, input_size, input_size, 3), 'float

    # Select the indices based on whether we are getting a Training/Validati
    if batch_type == "test":
        batch = np.random.choice(num_test, num_samples)
    elif batch_type == 'val':
        batch = np.random.choice(num_val, num_samples)
    else:
        batch = np.random.choice(num_train, num_samples)

    # Enumerate through the batch and fill the array with the proper data
    for i, n in enumerate(batch):

        # get the data from the data of the proper type: Training/Validation
        if batch_type == "test":
            im =  color.rgb2lab(io.imread(input_directory+test_X[n]))
        elif batch_type == 'val':
            im =  color.rgb2lab(io.imread(input_directory+val_X[n]))
        else:
            im =  color.rgb2lab(io.imread(input_directory+train_X[n]))

        # put the L channel in the input and the whole image in the output
        batch_input[i,:,:,0] = im[:,:,0]
        batch_output[i,:,:,:] = im

    return batch_input, batch_output

# call to the function
[batch_input, batch_output] = get_batch(2, 'train')
print(batch_input.shape)
print(batch_output.shape)
```

```
(2, 128, 128, 1)
(2, 128, 128, 3)
```

Define the AutoEncoder network architecture

In [8]:
```
# Function to fill-in-the-blanks for the VGG pre-trained network
def vgg_net(weights, image):
    layers = (
        # 'conv1_1', 'relu1_1',
        'conv1_2', 'relu1_2', 'pool1',
        'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',
        'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3',
        'relu3_3', 'conv3_4', 'relu3_4', 'pool3',
        'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3',
        'relu4_3', 'conv4_4', 'relu4_4', 'pool4',
        'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3',
        'relu5_3', 'conv5_4', 'relu5_4'
    )

    net = {}
    current = image
    for i, name in enumerate(layers):
        kind = name[:4]
        if kind == 'conv':
            kernels, bias = weights[i + 2][0][0][0][0]
            kernels = utils.get_variable(np.transpose(kernels, (1, 0, 2, 3))
            bias = utils.get_variable(bias.reshape(-1), name=name + "_b")
            current = utils.conv2d_basic(current, kernels, bias)
        elif kind == 'relu':
            current = tf.nn.relu(current, name=name)
        elif kind == 'pool':
            current = utils.avg_pool_2x2(current)
        net[name] = current

    return net
```

In [9]:
```
# Function that builds the rest of the net
def generator(images, train_phase):

    # Ge the model data and set up
    print("setting up vgg initialized conv layers ...")
    model_data = utils.get_model_data(model_dir, model_url)
    weights = np.squeeze(model_data['layers'])

    # Build the remaining "decoder" that will colorize the image
    with tf.variable_scope("generator") as scope:

        # First Layer: 3x3 2dConv with bias follower by RELU
        #              Need this layer because the input is only 1 channel
        W0 = utils.weight_variable([3, 3, 1, 64], name="W0")
        b0 = utils.bias_variable([64], name="b0")
        conv0 = utils.conv2d_basic(images, W0, b0)
        hrelu0 = tf.nn.relu(conv0, name="relu")

        # Add in the VGG network
        image_net = vgg_net(weights, hrelu0)
        vgg_final_layer = image_net["relu5_3"]
        pool5 = utils.max_pool_2x2(vgg_final_layer)

        # Decoder Level 1: begin to upscale the image and decrease the numbe
        #                  Use conv2d_transpose_strided() with 4x4 filter
        deconv_shape1 = image_net["pool4"].get_shape()
        W_t1 = utils.weight_variable([4, 4, deconv_shape1[3].value, pool5.ge
        b_t1 = utils.bias_variable([deconv_shape1[3].value], name="b_t1")
        conv_t1 = utils.conv2d_transpose_strided(pool5, W_t1, b_t1, output_s
        fuse_1 = tf.add(conv_t1, image_net["pool4"], name="fuse_1")

        # Decoder Level 2: continue to upscale the image and decrease the nu
        deconv_shape2 = image_net["pool3"].get_shape()
        print(deconv_shape2)
        W_t2 = utils.weight_variable([4, 4, deconv_shape2[3].value, deconv_s
        b_t2 = utils.bias_variable([deconv_shape2[3].value], name="b_t2")
        conv_t2 = utils.conv2d_transpose_strided(fuse_1, W_t2, b_t2, output_
        fuse_2 = tf.add(conv_t2, image_net["pool3"], name="fuse_2")

        # Decoder Level 3: continue to upscale the image and decrease the nu
        shape = tf.shape(images)
        deconv_shape3 = tf.stack([shape[0], shape[1], shape[2], 2])
        W_t3 = utils.weight_variable([16, 16, 2, deconv_shape2[3].value], na
        b_t3 = utils.bias_variable([2], name="b_t3")
        pred = utils.conv2d_transpose_strided(fuse_2, W_t3, b_t3, output_sha

    # return the concatenation of the input with the output to make it the i
    return tf.concat(axis=3, values=[images, pred], name="pred_image")
```

In [10]:
```
# Function to define the training that the net will under go
def train(loss, var_list):
    # create and AdamOptimizer with a learning rate and beta parameter
    optimizer = tf.train.AdamOptimizer(lr, beta)

    # compute the gradients
    grads = optimizer.compute_gradients(loss, var_list=var_list)

    # Apply the gradients to the optimizer
    return optimizer.apply_gradients(grads)
```

Set up the network for training

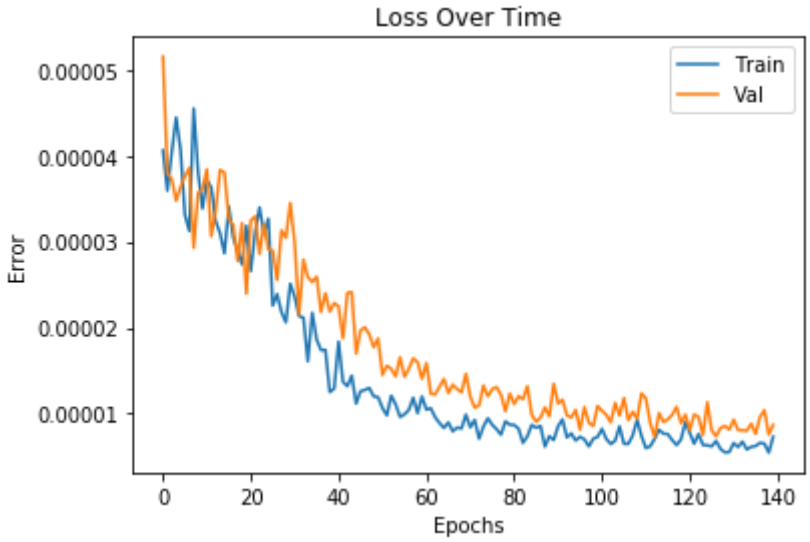
In [11]:
```
print("Setting up network...")

# Create placeholders for the input images and the output images
train_phase = tf.placeholder(tf.bool, name="train_phase")
images = tf.placeholder(tf.float32, shape=[None, None, None, 1], name='L_ima
lab_images = tf.placeholder(tf.float32, shape=[None, None, None, 3], name="I

# set pred_images to the output of the network
pred_image = generator(images, train_phase)

# define the loss function that we are minimizing as the L2-loss between the
gen_loss_mse = tf.reduce_mean(2 * tf.nn.l2_loss(pred_image - lab_images))
gen_loss_huber = tf.reduce_mean(tf.losses.huber_loss(lab_images, pred_image,

# initialize training variables
train_variables = tf.trainable_variables()

# train_op (which will be passes into the network) --> call the train() func
train_op = train(gen_loss_huber, train_variables)
```

```
Setting up network...
setting up vgg initialized conv layers ...
(?, ?, ?, 256)
```

In [12]:
```
avg_loss = []
val_loss = []
saver = tf.train.Saver()
```

In [13]:
```
# calculate the number of batches per epoch
batch_per_ep = int(num_train / batch_size)

# array for the losses
huber = 0.0

# begin tensor flow session
with tf.Session() as sess:
    # initialize variables
    sess.run(tf.global_variables_initializer())
    summary_op = tf.summary.merge_all()

    # get the saved model if it exists
    ckpt = tf.train.get_checkpoint_state(checkpoint_directory)
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, ckpt.model_checkpoint_path)
        print("Restoring Model")
    else:
        print("Creating New Model")

    # iterate through the number of epochs
    for ep in range(epoch_num):

        # iterate through the batches per epoch
        for batch_n in range(batch_per_ep):

            # get the batch out
            l_image, color_images = get_batch(batch_size, "train")

            # get the dictionary to feed into the training
            feed_dict = {images: l_image, lab_images: color_images, train_ph

            # run the dictionary through the network and output the mean-squ
            _, huber = sess.run([train_op, gen_loss_huber], feed_dict=feed_d

            if batch_n % 10 == 0:
                print("Epoch: %d, Batch: %d, Huber Loss: %g" % (ep, batch_n,

        # save the model each epoch
        _ = saver.save(sess, checkpoint_directory + "model.ckpt")

        # get error for validation set
        l_image, color_images = get_batch(batch_size, "val")
        feed_dict = {images: l_image, lab_images: color_images, train_phase:
        _, huber_val = sess.run([train_op, gen_loss_huber], feed_dict=feed_d

        # plot the training and validation error
        display.display(plt.gcf())
        display.clear_output(wait=True)
        avg_loss.append(huber)
        val_loss.append(huber_val)
        train_plt = plt.plot(avg_loss, label="Train")
        val_plt   = plt.plot(val_loss, label="Val")
        plt.legend()
        plt.title("Loss Over Time")
        plt.xlabel("Epochs")
        plt.ylabel("Error")
```

```
        plt.figure()
        plt.show()
        if ep % 20 == 19:
            lr = lr / 2
```

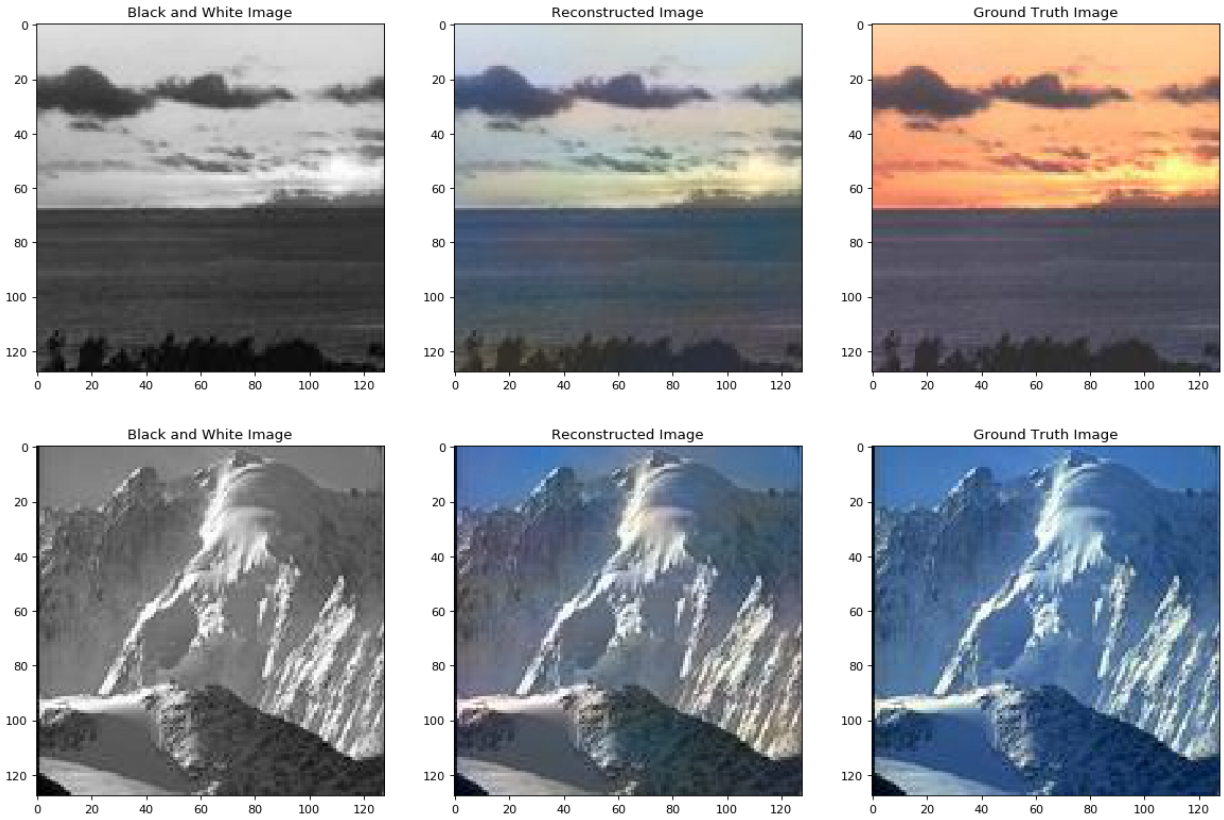


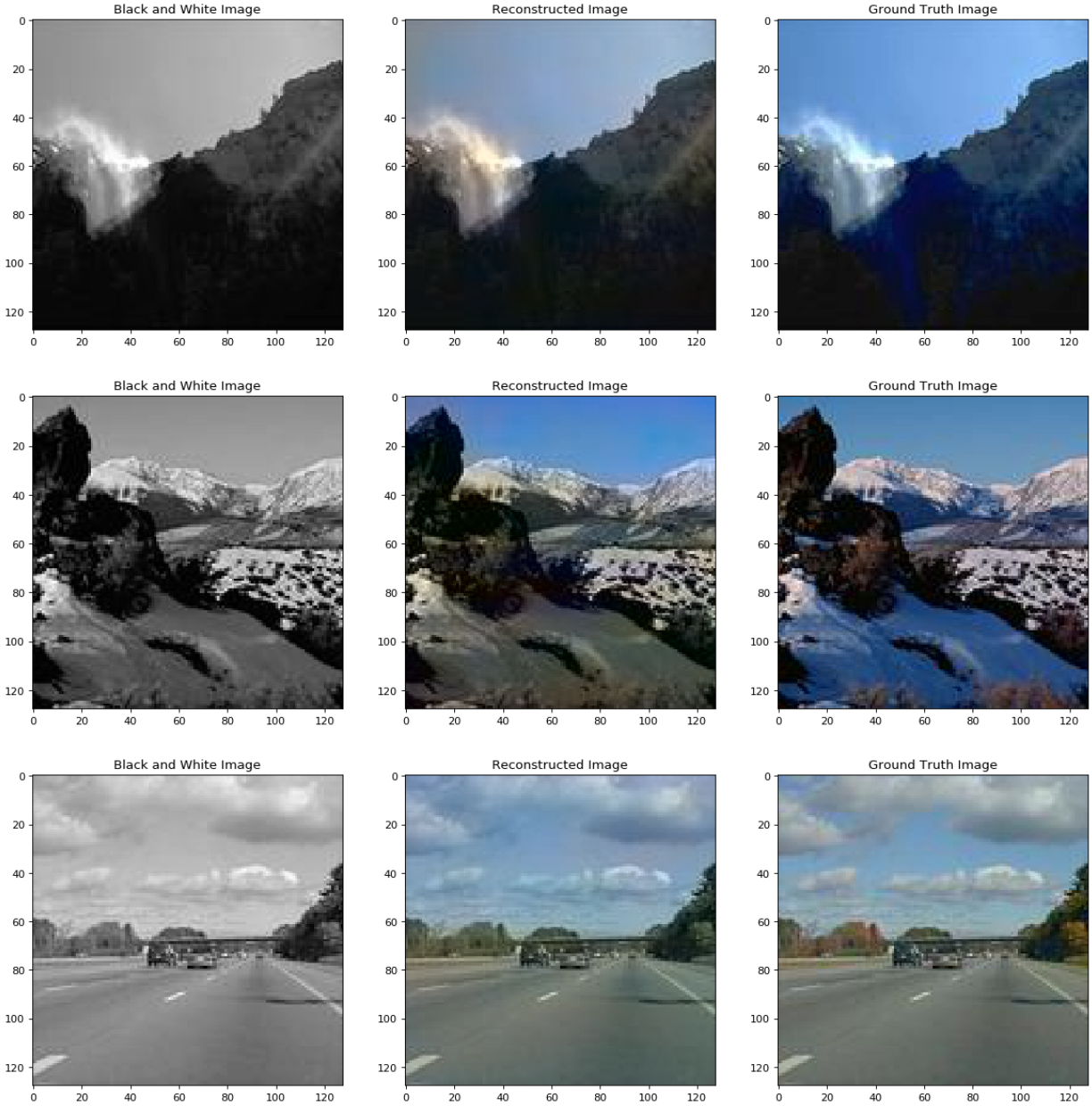

```
<matplotlib.figure.Figure at 0x7fea7ee23cf8>
```

Test the trained network

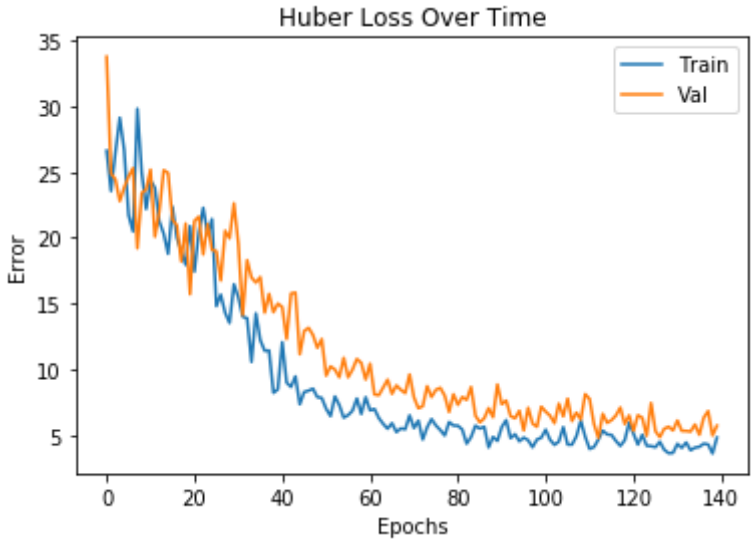
In [24]:
```
# Get the batch and run it through the network
with tf.Session() as sess:
    # get the previous model
    ckpt = tf.train.get_checkpoint_state(checkpoint_directory)
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, ckpt.model_checkpoint_path)
    else:
        # this should never fail
        assert(False)

    # run num_tests through the network and then display them
    num_tests = 5
    l_image, color_images = get_batch(num_tests, "test")
    feed_dict = {images: l_image, lab_images: color_images, train_phase: Fal
    [pred, huber] = sess.run([pred_image, gen_loss_huber], feed_dict=feed_di
    print(huber)
    for i in range(num_tests):
        showNetResults(pred[i,:,:,:], color_images[i,:,:,:])
```

```
INFO:tensorflow:Restoring parameters from CheckpointsHuber/model.ckpt
25.0134
```




Black and White Image
Reconstructed Image
Ground Truth Image
Black and White Image
Reconstructed Image
Ground Truth Image
Black and White Image
Reconstructed Image
Ground Truth Image

In [22]:
```
num = batch_size * input_size * input_size
t_loss = [num*x for x in avg_loss]
v_loss = [num*x for x in val_loss]
train_plt = plt.plot(t_loss, label="Train")
val_plt   = plt.plot(v_loss, label="Val")
plt.legend()
plt.title("Huber Loss Over Time")
plt.xlabel("Epochs")
plt.ylabel("Error")
plt.figure()
plt.show()
```




```
<matplotlib.figure.Figure at 0x7fea7e50a240>
```

In [ ]:
```
print
```