Imports and Basic Setup

```
# http://pytorch.org/
from os import path
from wheel.pep425tags import get_abbr_impl, get_impl_ver, get_abi_tag
import numpy as np
import time
from __future__ import division

platform = '{}{}-{}'.format(get_abbr_impl(), get_impl_ver(), get_abi_tag())

accelerator = 'cu80' if path.exists('/opt/bin/nvidia-smi') else 'cpu'

!pip install -q http://download.pytorch.org/whl/{accelerator}/torch-0.3.0.post4-{platform
import torch
```

```
import os
import torch
import torchvision
from torchvision import datasets, transforms
from torch.utils.data.sampler import SubsetRandomSampler
from matplotlib.backends.backend_pdf import PdfPages
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

```

```
!pip install Pillow==4.0.0
```

```
Collecting Pillow==4.0.0
  Downloading https://files.pythonhosted.org/packages/37/e8/b3fbf87b0188d22246678f8cd6
    100% |████████████████████████████████| 5.6MB 7.2MB/s
Requirement already satisfied: olefile in /usr/local/lib/python3.6/dist-packages (from
torchvision 0.2.1 has requirement pillow>=4.1.1, but you'll have pillow 4.0.0 which is
Installing collected packages: Pillow
  Found existing installation: Pillow 5.3.0
    Uninstalling Pillow-5.3.0:
      Successfully uninstalled Pillow-5.3.0
Successfully installed Pillow-4.0.0
```

```
!pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
import datetime

auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

Downsampler

```
# Downsampler(n_planes=3, factor=4, kernel_type='lanczos2', phase=0.5, preserve_size=True
from torch import nn
class Downsample(nn.Module):
  def __init__ (self, n_planes, factor, preserve_size):
    super(Downsample, self).__init__()
    kernel_width = 4 * factor + 1
    self.kernel = get_kernel(factor, kernel_width)
    downsampler = nn.Conv2d(n_planes, n_planes, kernel_size=self.kernel.shape, stride=fac
    downsampler.weight.data[:] = 0
    downsampler.bias.data[:] = 0
    kernel_torch = torch.from_numpy(self.kernel)
    for i in range(n_planes):
      downsampler.weight.data[i, i] = kernel_torch
    self.downsampler_ = downsampler

    if preserve_size:

        if  self.kernel.shape[0] % 2 == 1:
            pad = int((self.kernel.shape[0] - 1) / 2.)
        else:
            pad = int((self.kernel.shape[0] - factor) / 2.)

        self.padding = nn.ReplicationPad2d(pad)

    self.preserve_size = preserve_size

  def forward(self, input):
      if self.preserve_size:
          x = self.padding(input)
      else:
          x= input
      self.x = x
      return self.downsampler_(x)

def get_kernel(factor, kernel_width):
  support = 2
  kernel = np.zeros([kernel_width - 1, kernel_width - 1])
  center = (kernel_width + 1) / 2.

  for i in range(1, kernel.shape[0] + 1):
      for j in range(1, kernel.shape[1] + 1):

          di = abs(i + 0.5 - center) / factor
          dj = abs(j + 0.5 - center) / factor

          pi_sq = np.pi * np.pi

          val = 1
          if di != 0:
              val = val * support * np.sin(np.pi * di) * np.sin(np.pi * di / support)
              val = val / (np.pi * np.pi * di * di)

          if dj != 0:
              val = val * support * np.sin(np.pi * dj) * np.sin(np.pi * dj / support)
              val = val / (np.pi * np.pi * dj * dj)

          kernel[i - 1][j - 1] = val
  kernel /= kernel.sum()
  return kernel
```

Data import for superresolution dip network

```
import os

# Fetch the folder with all the input images
file_list = drive.ListFile({'q': "'11a5-Sek52Fq86iHXoK6KQ-Gvr1G8El0Y' in parents and tras

# Create an input directory (no need to execute cuz created)
os.mkdir('input')
```

```
# Store files in the input directory
i = 1
for file1 in sorted(file_list, key = lambda x: x['title']):
    print('Downloading {} from GDrive ({}/{})'.format(file1['title'], i, len(file_list)))
    file1.GetContentFile('input/' + file1['title'])
    i += 1
```

```
Downloading einstein.jpg from GDrive (1/58)
Downloading gaxe2_axbattler_01_input.png from GDrive (2/58)
Downloading gaxe2_axbattler_02_input.png from GDrive (3/58)
Downloading gaxe_skeleton_input.png from GDrive (4/58)
Downloading icon_atari_bomb_input.png from GDrive (5/58)
Downloading icon_disk_input.png from GDrive (6/58)
Downloading invaders_01_input.png from GDrive (7/58)
Downloading invaders_02_input.png from GDrive (8/58)
Downloading invaders_03_input.png from GDrive (9/58)
Downloading invaders_04_input.png from GDrive (10/58)
Downloading invaders_05_input.png from GDrive (11/58)
Downloading invaders_06_input.png from GDrive (12/58)
Downloading man_dancing.jpg from GDrive (13/58)
Downloading mana_granpa_input.png from GDrive (14/58)
Downloading mana_joch_input.png from GDrive (15/58)
Downloading mana_rabite_input.png from GDrive (16/58)
Downloading mana_randi_01_input.png from GDrive (17/58)
Downloading mana_randi_02_input.png from GDrive (18/58)
Downloading mana_salamando_input.png from GDrive (19/58)
```

```
images = os.listdir('input')
#Verify that files are present
images[:5]
```



```
Downloading sbm4_64_input.png from GDrive (36/58)
```

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

path = 'input/einstein.jpg'
image = mpimg.imread(path)
plt.imshow(image)
```



```
Downloading wlh31_keyboard_input.png from GDrive (56/58)
```

```
image.shape
```

```
from PIL import Image
def get_image(filename):
  image = Image.open('input/' + filename)
  image_array = np.array(image)
  image_array = image_array.transpose((2, 0, 1))
  return image_array/255
```

Build superresolution dip network

```
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

LeakyReLU = nn.LeakyReLU(0.2, inplace=True)

num_channels_down=[128, 128, 128, 128, 128]
num_channels_up=[128, 128, 128, 128, 128]
num_channels_skip=[4, 4, 4, 4, 4]
num_channels_input = 32
k_down = 3
k_up = 3
k_skip = 1

padding_down = int((k_down-1)/2)
padding_up = int((k_up-1)/2)
padding_skip = int((k_skip-1)/2)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        #------------------------------------------------------------------------
        #----------------------------- Down blocks ------------------------------
        #------------------------------------------------------------------------

        i = 0 # for first
        #first down block, others are copypaste
        self.down1_downsample = nn.Conv2d(num_channels_input, num_channels_down[i], k_dow
        self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 1
        self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 2
        self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 3
        self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
```

```
        self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 4
        self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        #----------------------------------------------------------------------
        #----------------------------- Skip blocks ----------------------------
        #----------------------------------------------------------------------

        i = 0 # for first
        #first skip block, others are copypaste
        self.skip1_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
        self.skip1_bn = nn.BatchNorm2d(num_channels_skip[i])

        i = 1
        self.skip2_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
        self.skip2_bn = nn.BatchNorm2d(num_channels_skip[i])

        i = 2
        self.skip3_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
        self.skip3_bn = nn.BatchNorm2d(num_channels_skip[i])

        i = 3
        self.skip4_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
        self.skip4_bn = nn.BatchNorm2d(num_channels_skip[i])

        i = 4
        self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
        self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])

        #----------------------------------------------------------------------
        #--------------------------- Misc but needed --------------------------
        #----------------------------------------------------------------------

        #will need this for the up layers
        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear') #at the end
        num_channels_up.append(0) #this way for the fifth layers the combined is just the
        #will need this at the end of network
        self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)

        #----------------------------------------------------------------------
        #------------------------------ Up blocks -----------------------------
        #----------------------------------------------------------------------

        i=4 # for fifth
        #fifth up block, others are copypaste
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
        self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=3
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up4_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up4_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=2
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up3_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up3_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])
```

```
        self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=1
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up2_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up2_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=0
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up1_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up1_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])

    def forward(self, x):
        #-------------------------------------------------------------------------
        #-------------------- Downsampling and recording skip --------------------
        #-------------------------------------------------------------------------
        x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
        x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))
        skips1 = LeakyReLU(self.skip1_bn(self.skip1_conv(x)))

        x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
        x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))
        skips2 = LeakyReLU(self.skip2_bn(self.skip2_conv(x)))

        x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
        x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))
        skips3 = LeakyReLU(self.skip3_bn(self.skip3_conv(x)))

        x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
        x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))
        skips4 = LeakyReLU(self.skip4_bn(self.skip4_conv(x)))

        x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
        x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
        skips5 = LeakyReLU(self.skip5_bn(self.skip5_conv(x)))

        #-------------------------------------------------------------------------
        #------------------------------ Upsampling -------------------------------
        #-------------------------------------------------------------------------

        #we start at the fifth layer (layers have reverse order)
        x = self.up5_bn_initial(skips5) #notice no concatenation here
        x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
        x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
        x = self.upsample(x)

        #now we upsample all the other layers
        #print("up4")
        #print(x.size())
        #print(skips4.size())
        x = self.up4_bn_initial(torch.cat((x, skips4),1))
        x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
        x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
        x = self.upsample(x)

        #print("up3")
        #print(x.size())
        #print(skips3.size())
        x = self.up3_bn_initial(torch.cat((x, skips3),1))
        x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
        x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
        x = self.upsample(x)

```

```
        #print("up2")
        #print(x.size())
        #print(skips2.size())
        x = self.up2_bn_initial(torch.cat((x, skips2),1))
        x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
        x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
        x = self.upsample(x)

        #print("up1")
        #print(x.size())
        #print(skips1.size())
        x = self.up1_bn_initial(torch.cat((x, skips1),1))
        x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
        x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
        x = self.upsample(x)


        #---------------------------------------------------------------------
        #------------------------------ Finalize -----------------------------
        #---------------------------------------------------------------------

        #now we get x to the right dimensions
        x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here

        return x
```

```
#generate noisy image for Yoshi
#original dimensions (3,32,22)
#output image (3,256,176)
#input_depth = 32
#dimensions = [256,176]
#noiseType = "uniform"

def createNoisyImage(input_depth,dimensions,var=0.1,noiseType='uniform'):
  tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
  networkInput = Variable(torch.zeros(tensorShape))

  if noiseType == "uniform":
    networkInput.data.uniform_()
  elif noiseType == "normal":
    networkInput.data.normal_()

  networkInput.data = networkInput.data * var

  return networkInput

#this needs to be image size times 4!
superResolutionBase = createNoisyImage(32,[256,256],0.1,"uniform")
#inputs are: input_depth, [img.shape[1],img.shape[2]], var, noiseType

base = superResolutionBase.type(torch.cuda.FloatTensor).detach()
savedBase = base.data.clone()
baseNoise = base.data.clone()
```

```
def addRandomNoise(inputs):
  inputs.data = savedBase + baseNoise.normal_()*1.0/30
  return inputs
```

Train superresolution dip network

We train it on a small image (64x64) for prototyping but the same network was used successfully on larger

```
#plotting only
superResolutionInput = superResolutionBase.data[0,:,:,:]
#print(superResolutionInput)
plt.imshow(np.array(superResolutionInput)[30])
```

```
#size is [1, channels, height, width]
lowres_image = Variable(torch.Tensor(get_image('einstein.jpg')))
img = get_image('einstein.jpg')
img.shape
#plt.imshow(img)
```

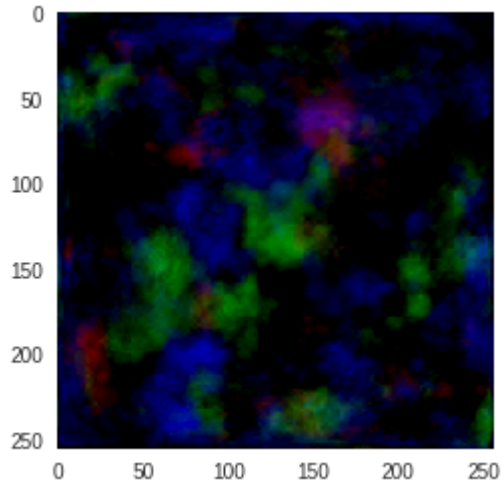
Naturally training is testing here, so we save outputs to images and upload these images to gdrive

```
net = Net()
net.cuda()

criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
```

```
#Set up Training
#Loss Function = MSE
#Get image in Tensor form. This is the label
dtype = torch.cuda.FloatTensor

print('Started Training')
startime = time.time();

maxepochs = 1000
numepochs = 0

# horizon = 5

acc = []
trainLoss = []

downsampler = Downsample(n_planes=3, factor=4, preserve_size=True).type(dtype)
```

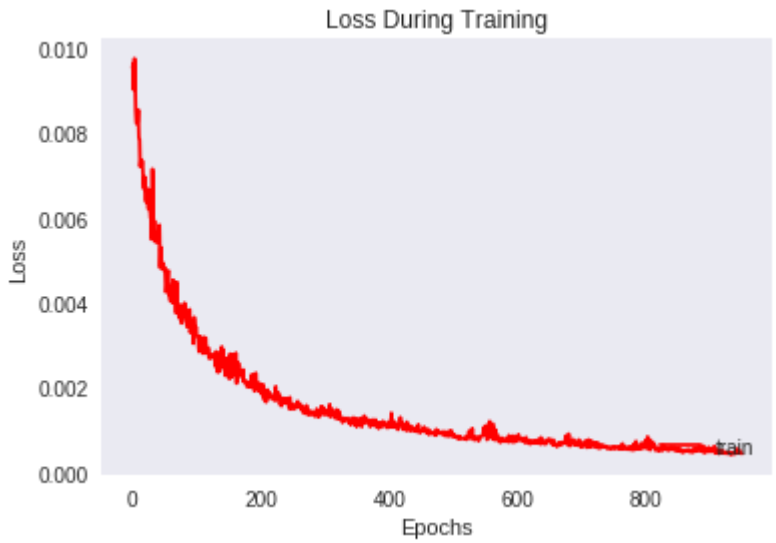
```
18
19 for epoch in range(maxepochs):  # loop over the data multiple times
20
21     # get the input
22     inputs = superResolutionBase #this is the z = uniform noise input
23
24     # get the label
25     label = lowres_image
26
27     # wrap them in Variable
28     inputs, label = inputs.cuda(), label.cuda()
29
30     # zero the parameter gradients
31     optimizer.zero_grad()
32
33     inputs = addRandomNoise(inputs)
34
35     # forward + backward + optimize
36     outputs = net(inputs)
37     downsampledOutputs = downsampler(outputs)
38     loss = criterion(downsampledOutputs[0], label)
39     loss.backward()
40     optimizer.step()
41
42     #save the image every 20 epochs
43     if epoch%20==0:
44         pp = PdfPages('einstein' + str(epoch) + '.pdf')
45
46         arr = np.array(outputs[0].data)
47         arr = arr.transpose((1,2,0))
48         arr = np.clip(arr, 0, 1)
49         plt.rcParams["axes.grid"] = False
50         plt.imshow(arr)
51         plt.savefig(pp, format='pdf')
52         pp.close()
53
54         file1 = drive.CreateFile()
55         file1.SetContentFile('einstein' + str(epoch) + '.pdf')
56         file1.Upload() # Upload the file.
57         plt.show()
58
59     #print some results
60     if epoch == 0:
61         k = -1
62     else:
63         k = -2
64     print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - startime), loss.data[0]
65     numepochs = numepochs + 1
66
67     trainLoss.append(loss.data[0])
68
69 print("Final train loss")
70 print(trainLoss[len(trainLoss)-1])
71
72 print('Finished Training. See training time on next line')
73 print(int(time.time() - startime))
```

```
Started Training
```




```
[1, 2 s] loss: 0.396
[2, 2 s] loss: 0.400
[3, 2 s] loss: 0.140
[4, 2 s] loss: 0.298
[5, 2 s] loss: 0.083
[6, 3 s] loss: 0.084
[7, 3 s] loss: 0.071
[8, 3 s] loss: 0.051
[9, 3 s] loss: 0.042
[10, 3 s] loss: 0.038
[11, 3 s] loss: 0.035
[12, 3 s] loss: 0.033
[13, 3 s] loss: 0.033
[14, 3 s] loss: 0.032
[15, 3 s] loss: 0.029
[16, 3 s] loss: 0.029
[17, 3 s] loss: 0.026
[18, 3 s] loss: 0.025
[19, 3 s] loss: 0.023
[20, 4 s] loss: 0.022
```


```
plt.figure()
plt.title('Loss During Training')
plt.xlabel('Epochs')
plt.ylabel('Loss')
#because loss is very high in the early iteration,
#we cut off the first 50 to make the graph meaningful
plt.plot(trainLoss[50:], 'r', label="train")
plt.legend(loc=4)
plt.show()
```



Subtitle extraction for subtitle inpainting dip network (SIDN)

We are creating the masks used in inpainting dip. Specifically, we are creating filters for subtitle extracting. The original images themselves are frames from the movie Annie Hall: https://www.youtube.com/watch?v=3tcH2zSqVCc

```
#1AY63mgg2926Z2XtxXvzez3hxdWhU0KDb -- main folder (no useful files in there)
  #1gNp7MLzRkLDOwgmdT-j0XLAVQ9zACR_z -- original8frames
  #1n7iCwVNkUtMZE9TyRMXvo2bCPn82-Od1 -- 8otherframes (not used yet!), these frames don't
  #1KqhEYNBtiQRsM-WmoWcSEdt1V5iprxFc -- contains original8frames plus processed data (mas

# Fetch the folder with all the annie hall images
file_list = drive.ListFile({'q': "'1gNp7MLzRkLDOwgmdT-j0XLAVQ9zACR_z' in parents and tras
```

```
i = 1
for file1 in sorted(file_list, key = lambda x: x['title']):
    print('Downloading {} from GDrive ({}/{})'.format(file1['title'], i, len(file_list)))
    file1.GetContentFile(file1['title'])
    i += 1
```

```
Downloading snapshot1.png from GDrive (1/8)
Downloading snapshot2.png from GDrive (2/8)
Downloading snapshot3.png from GDrive (3/8)
Downloading snapshot4.png from GDrive (4/8)
Downloading snapshot5.png from GDrive (5/8)
Downloading snapshot6.png from GDrive (6/8)
Downloading snapshot7.png from GDrive (7/8)
Downloading snapshot8.png from GDrive (8/8)
```

```
#This function makes a mask from an image where the pixels corresponding to
#the subtitles in the image are 1 in the mask, and other pixels are 0

```

```
#This algorithm is general enough that it works as long as:
  # The pixels from the subtitles are (relatively) uniform in color
  # The range of pixels across the rest of the pictures excludes the uniform color used f

#The two paramters to set are
  # The color of the subtitles (see comments lines 18-19 and line 22)
  # The radius of pixels you want to mask around detected subtitle pixels (line 26)

#This is not perfect, but we were not very conservative around the subtitles but very con
#The dataset (this scene from Annie Hall: https://www.youtube.com/watch?v=3tcH2zSqVCc) th
def makeMask(img):
  out1 = np.zeros(img[:,:,0].shape)
  out2 = np.zeros(img[:,:,0].shape)

  #catch yellow ones (by yellow we mean our mildly arbitrary range)
  #we used https://imagecolorpicker.com/
  for i in range(500,img.shape[0]):
    for j in range(img.shape[1]):
      if img[i,j,0]>.9 and img[i,j,1]>.9 and img[i,j,2]>.05 and img[i,j,2]<.6:
        out1[i,j] = 1.

  #make everyone within radius of yellow one yellow
  radius = 5 #higher is more liberal with masking nonsubtitle pixels, but if it's too low
  for i in range(radius,img.shape[0]-radius-1):
    for j in range(radius,img.shape[1]-radius-1):
      for k in range(-radius,radius+1):
        for l in range(-radius,radius+1):
          if out1[i+k,j+l]==1:
            out2[i,j] = 1.

  #apply the same mask on 3 channels
  finalOut = np.zeros(img.shape)
  finalOut[:,:,0] = 1-out2
  finalOut[:,:,1] = 1-out2
  finalOut[:,:,2] = 1-out2

  return finalOut
```

```
#Do NOT run this again -- it takes a lot of itme and we have the outputs saved in gdrive
#This code is used both for testing and saving stuff, you can comment stuff out

%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import scipy.misc

for i in range(1,9):
  print("Dealing with image " + str(i))

  #show original picture
  image = mpimg.imread('snapshot' + str(i) + '.png')
  plt.rcParams["axes.grid"] = False
  plt.imshow(image)
  plt.show()

  #make mask and subs/nosubs image
  myMask = makeMask(image)
  onlysubs = np.multiply(image, 1-myMask) #use this if you wanna see just the subs!
  withoutsubs = np.multiply(image, myMask) #use this if you wanna see img without subs!

  #show subs/nosubs image
  plt.imshow(onlysubs)
  plt.show()
  plt.imshow(withoutsubs)
  plt.show()

  #check and diagnose in more depth
```

```
  plt.imshow(withoutsubs[560:650,300:900,:])
  plt.show()
  plt.imshow(onlysubs[560:650,300:900,:])
  plt.show()

  #save the images
  scipy.misc.imsave('snapshot' + str(i) + '_mask.png', myMask)
  scipy.misc.imsave('snapshot' + str(i) + '_withoutsubs.png', withoutsubs)
  scipy.misc.imsave('snapshot' + str(i) + '_onlysubs.png', onlysubs)
```

```
Dealing with image 1
```


```
#make sure we have all the files
dir = os.listdir('.')
dir.sort()
print(dir)
```

```
for i in range(1,9):
  file1 = drive.CreateFile()
  file2 = drive.CreateFile()
  file3 = drive.CreateFile()

  file1.SetContentFile('snapshot' + str(i) + '_mask.png')
  file2.SetContentFile('snapshot' + str(i) + '_withoutsubs.png')
  file3.SetContentFile('snapshot' + str(i) + '_onlysubs.png')

  file1.Upload()
  file2.Upload()
  file3.Upload()
```

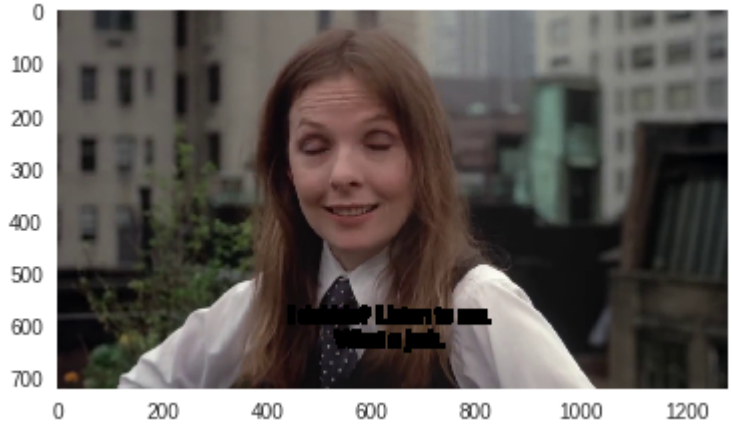
```
#show any image to make sure it got saved
image = mpimg.imread('snapshot7_withoutsubs.png')
plt.imshow(image)
```

Load 8frames processed data for SIDN

```
#Download the original and processed 8frames data
file_list = drive.ListFile({'q': "'1KqhEYNBtiQRsM-WmoWcSEdt1V5iprxFc' in parents and tras
i = 1
for file1 in sorted(file_list, key = lambda x: x['title']):
    print('Downloading {} from GDrive ({}/{})'.format(file1['title'], i, len(file_list)))
    file1.GetContentFile(file1['title'])
    i += 1
```

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["axes.grid"] = False
import matplotlib.image as mpimg
import numpy as np
import scipy.misc

#for loop to loop through dataset
#for i in range(1,9):
i = 1
original = mpimg.imread('snapshot' + str(i) + '.png')
mask = mpimg.imread('snapshot' + str(i) + '_mask.png')
withoutsubs = mpimg.imread('snapshot' + str(i) + '_withoutsubs.png')
onlysubs = mpimg.imread('snapshot' + str(i) + '_onlysubs.png')

#plot any of the extracted images
plt.imshow(withoutsubs)
mask.shape
```

```
(720, 1280, 3)
```




```
#crop withoutsubs and mask to make both axes a multiple of 32
withoutsubsCropped = withoutsubs[8:712,:,:]
maskCropped = mask[8:712,:,:]
```

Build SIDN

```
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

LeakyReLU = nn.LeakyReLU(0.2, inplace=True)

num_channels_down=[16, 32, 64, 128, 128]
num_channels_up=[16, 32, 64, 128, 128]
num_channels_skip=[0, 0, 0, 0, 4]
num_channels_input = 32
k_down = 7
k_up = 7
k_skip = 1

padding_down = int((k_down-1)/2)
padding_up = int((k_up-1)/2)
padding_skip = int((k_skip-1)/2)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        #------------------------------------------------------------------------
        #---------------------------- Down blocks -------------------------------
        #------------------------------------------------------------------------

        i = 0 # for first
        #first down block, others are copypaste
        self.down1_downsample = nn.Conv2d(num_channels_input, num_channels_down[i], k_dow
        self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 1
        self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
```

```
        self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 2
        self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 3
        self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 4
        self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        #----------------------------------------------------------------------
        #---------------------------- Skip blocks -----------------------------
        #----------------------------------------------------------------------

        i = 4 # for 5th
        #first skip block, others are copypaste
        self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
        self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])

        #----------------------------------------------------------------------
        #--------------------------- Misc but needed --------------------------
        #----------------------------------------------------------------------

        #will need this for the up layers
        self.upsample = nn.Upsample(scale_factor=2, mode='nearest') #at the end
        num_channels_up.append(0) #this way for the fifth layers the combined is just the
        #will need this at the end of network
        self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)

        #----------------------------------------------------------------------
        #------------------------------ Up blocks -----------------------------
        #----------------------------------------------------------------------

        i=4 # for fifth
        #fifth up block, others are copypaste
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
        self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=3
        self.up4_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up4_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
        self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=2
        self.up3_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up3_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
        self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=1;
        self.up2_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up2_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
```

```
        self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=0
        self.up1_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up1_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
        self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])

    def forward(self, x):
        #---------------------------------------------------------------------
        #-------------------- Downsampling and recording skip ----------------
        #---------------------------------------------------------------------
        x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
        x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))

        x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
        x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))

        x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
        x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))

        x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
        x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))

        x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
        x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
        x = self.skip5_conv(x)
        skips5 = LeakyReLU(self.skip5_bn(x))

        #---------------------------------------------------------------------
        #------------------------------ Upsampling ---------------------------
        #---------------------------------------------------------------------

        #we start at the fifth layer (layers have reverse order)
        x = self.up5_bn_initial(skips5) #notice no concatenation here
        x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
        x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
        x = self.upsample(x)

        #now we upsample all the other layers
        #print("up4")
        #print(x.size())
        #print(skips4.size())
        x = self.up4_bn_initial(x)
        x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
        x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
        x = self.upsample(x)

        #print("up3")
        #print(x.size())
        #print(skips3.size())
        x = self.up3_bn_initial(x)
        x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
        x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
        x = self.upsample(x)

        #print("up2")
        #print(x.size())
        #print(skips2.size())
        x = self.up2_bn_initial(x)
        x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
        x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
        x = self.upsample(x)

        #print("up1")
        #print(x.size())
        #print(skips1.size())
```

```
        x = self.up1_bn_initial(x)
        x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
        x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
        x = self.upsample(x)


        #--------------------------------------------------------------------
        #------------------------------ Finalize ----------------------------
        #--------------------------------------------------------------------

        #now we get x to the right dimensions
        x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here

        return x
```

```
#generate noisy image as input
#noiseType = "uniform"

def createNoisyImage(input_depth,dimensions,var=0.1,noiseType='uniform'):
  tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
  networkInput = Variable(torch.zeros(tensorShape))

  if noiseType == "uniform":
    networkInput.data.uniform_()
  elif noiseType == "normal":
    networkInput.data.normal_()

  networkInput.data = networkInput.data * var

  return networkInput

#this needs to be image size times 4!
superResolutionBase = createNoisyImage(32,[maskCropped.shape[0],maskCropped.shape[1]],0.1
#inputs are: input_depth, [img.shape[1],img.shape[2]], var, noiseType

base = superResolutionBase.type(torch.cuda.FloatTensor).detach()
savedBase = base.data.clone()
baseNoise = base.data.clone()
```

```
maskCropped.shape[0]
```

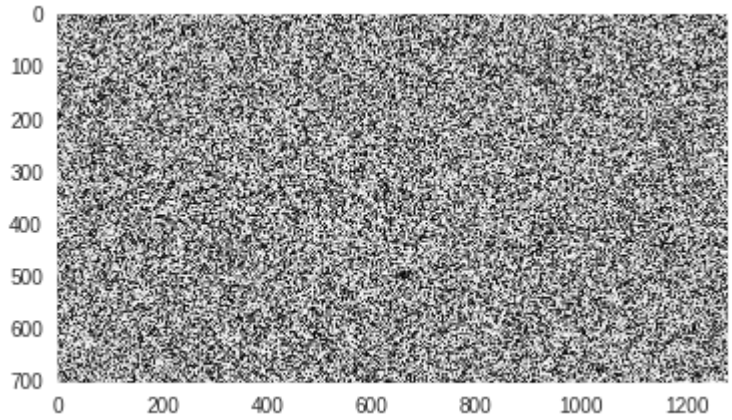
```
704
```

```
def addRandomNoise(inputs): #we don't add noise this time!
  inputs.data = savedBase #+ baseNoise.normal_()*1.0/30
  return inputs
```

Train SIDN

```
#plotting only
superResolutionInput = superResolutionBase.data[0,:,:,:]
#print(superResolutionInput)
plt.imshow(np.array(superResolutionInput)[30])
```

```
<matplotlib.image.AxesImage at 0x7f7c2cee3160>
```




```
#size is [1, channels, height, width
maskedTarget = Variable(torch.Tensor(withoutsubsCropped))
maskCroppedTensor = Variable(torch.Tensor(maskCropped))

maskedTarget = maskedTarget.cuda()
maskCroppedTensor = maskCroppedTensor.cuda()
```

Naturally training is testing here, so we save outputs to images and upload these images to gdrive

```
# Saving any model
def save_model(model, filename):
  torch.save(model, filename)

def load_model(filename):
  model = torch.load(filename)
  model.cuda()
  return model
```

```
#Saving a random model
net = Net()
net.cuda()
save_model(net, 'random_init.pt')
```

```
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:158: UserWarning: Couldn
  "type " + obj.__name__ + ". It won't be checked "
```

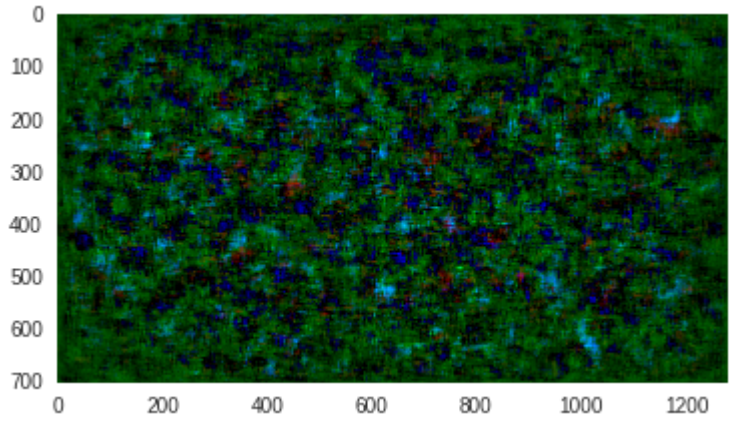
```
net = Net()
net.cuda()

criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
```

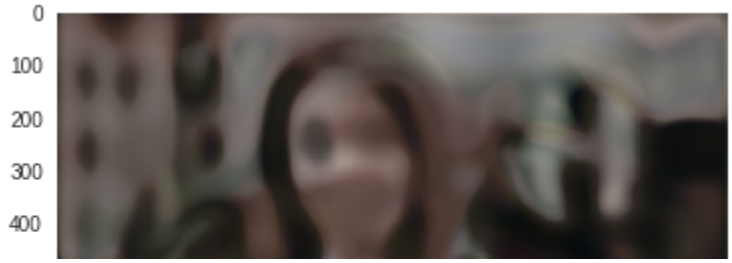
```
def train_model(net, mask_fname, withoutsubs_fname, criterion, optimizer, maxepochs=3000,
  mask = mpimg.imread(mask_fname)
  withoutsubs = mpimg.imread(withoutsubs_fname)
  withoutsubsCropped = withoutsubs[8:712,:,:]
  maskCropped = mask[8:712,:,:]
  maskedTarget = Variable(torch.Tensor(withoutsubsCropped))
  maskCroppedTensor = Variable(torch.Tensor(maskCropped))
```

```
    maskedTarget = maskedTarget.cuda()
    maskCroppedTensor = maskCroppedTensor.cuda()

    startime = int(time.time())
    trainLoss = []

    for epoch in range(maxepochs):  # loop over the data multiple times

        # get the input
        inputs = superResolutionBase #this is the z = uniform noise input

        # get the label
        label = maskedTarget

        # wrap them in Variable
        inputs, label = inputs.cuda(), label.cuda()

        # zero the parameter gradients
        optimizer.zero_grad()

        inputs = addRandomNoise(inputs)

        #print("backpropagation time")
        # forward + backward + optimize
        outputs = net(inputs)
        outputs=outputs[0]
        outputs = outputs.permute(1,2,0)
        loss = criterion(torch.mul(outputs, maskCroppedTensor), label)
        loss.backward()
        optimizer.step()

        #if epoch%5==0:
          #print("We are at epoch " + str(epoch))

        #save the image every 100 epochs
        if epoch%100==0:
            print(epoch)
    #         pp = PdfPages('subsTest' + str(epoch) + '.pdf')

            arr = np.array(outputs.data)
            arr = np.clip(arr, 0, 1)
            plt.rcParams["axes.grid"] = False
            plt.imshow(arr)
    #         plt.savefig(pp, format='pdf')
    #         pp.close()
            plt.imsave(fname + 'subsTest' + str(epoch) + '.png', arr)
            file1 = drive.CreateFile()
            file1.SetContentFile(fname + 'subsTest' + str(epoch) + '.png')
            file1.Upload() # Upload the file.
            plt.show()

        #print some results
        if epoch == 0:
            k = -1
        else:
            k = -2
        print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - startime), loss.data[
        trainLoss.append(loss.data[0])

    print("Final train loss")
    print(trainLoss[len(trainLoss)-1])

    print('Finished Training. See training time on next line')
    print(int(time.time() - startime))
    save_model(net, fname+'trained.pt')
    file1 = drive.CreateFile()
    file1.SetContentFile(fname+'trained.pt')
    file1.Upload() # Upload the file.
    return net
```

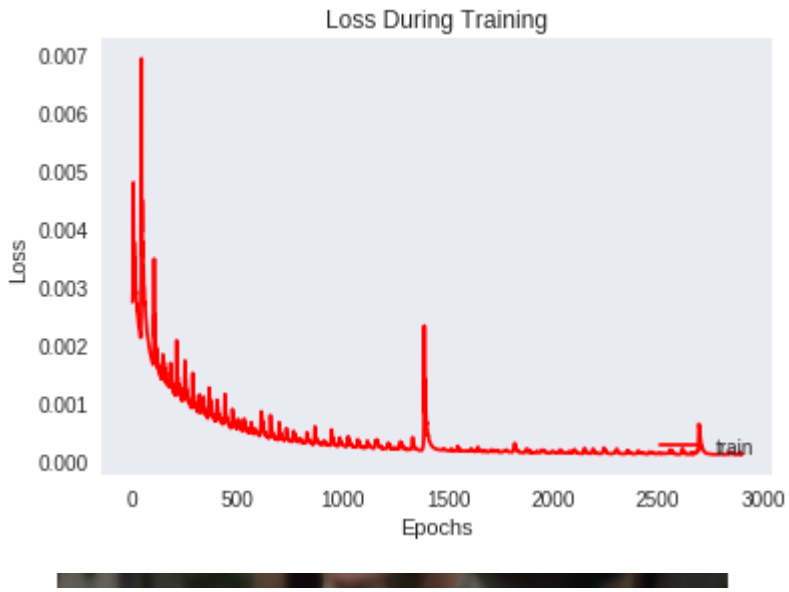
```
#Set up Training
#Loss Function = MSE
#Get image in Tensor form. This is the label
dtype = torch.cuda.FloatTensor

print('Started Training')
startime = time.time();

maxepochs = 3000
numepochs = 0

# horizon = 5

acc = []
trainLoss = []
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
i=1
mask_fname = 'snapshot' + str(i) + '_mask.png'
withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'
train_model(net, mask_fname, withoutsubs_fname, criterion, optimizer, maxepochs=3000, fna
```

```
Started Training
```






```
plt.figure()
plt.title('Loss During Training')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(trainLoss[100:], 'r', label="train")
plt.legend(loc=4)
plt.show()
```



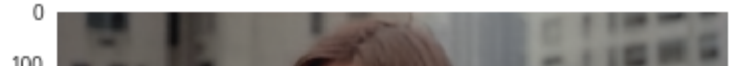
```
# Transfer Learning
model = load_model('first_traintrained.pt')
acc = []
trainLoss = []

criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
i=3
mask_fname = 'snapshot' + str(i) + '_mask.png'
withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'
model_after_transfer = train_model(net, mask_fname, withoutsubs_fname, criterion, optimiz
```

0

0 

```
#training from scratch is slower to produce good inpainting
model = Net()
model.cuda()
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
i=3
mask_fname = 'snapshot' + str(i) + '_mask.png'
withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'
model_without_transfer = train_model(net, mask_fname, withoutsubs_fname, criterion, optim
```

2700



Creating and Saving Masks for Report

3 cells hidden

Build Autoencoder SIDN

```
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

LeakyReLU = nn.LeakyReLU(0.2, inplace=True)

num_channels_down=[16, 32, 64, 128, 128]
num_channels_up=[16, 32, 64, 128, 128]
num_channels_skip=[0, 0, 0, 0, 4]
num_channels_input = 32
num_train_images = 4
k_down = 7
k_up = 7
k_skip = 1

padding_down = int((k_down-1)/2)
padding_up = int((k_up-1)/2)
padding_skip = int((k_skip-1)/2)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        #------------------------------------------------------------------------
        #---------------------------- Down blocks -------------------------------
        #------------------------------------------------------------------------

        i = 0 # for first
        #first down block, others are copypaste
        self.down1_downsample = nn.Conv2d(num_channels_input+num_train_images, num_channe
        self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 1
        self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
```

```
        self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 2
        self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 3
        self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 4
        self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        #----------------------------------------------------------------------
        #----------------------------- Skip blocks ----------------------------
        #----------------------------------------------------------------------

        i = 4 # for 5th
        #first skip block, others are copypaste
        self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
        self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])

        #----------------------------------------------------------------------
        #--------------------------- Misc but needed --------------------------
        #----------------------------------------------------------------------

        #will need this for the up layers
        self.upsample = nn.Upsample(scale_factor=2, mode='nearest') #at the end
        num_channels_up.append(0) #this way for the fifth layers the combined is just the
        #will need this at the end of network
        self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)

        #----------------------------------------------------------------------
        #------------------------------ Up blocks -----------------------------
        #----------------------------------------------------------------------

        i=4 # for fifth
        #fifth up block, others are copypaste
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
        self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=3
        self.up4_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up4_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
        self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=2
        self.up3_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up3_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
        self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=1;
        self.up2_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up2_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
```

```
        self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=0
        self.up1_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
        self.up1_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
        self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])

    def forward(self, x):
        #---------------------------------------------------------------------
        #-------------------- Downsampling and recording skip ----------------
        #---------------------------------------------------------------------
        x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
        x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))

        x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
        x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))

        x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
        x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))

        x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
        x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))

        x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
        x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
        x = self.skip5_conv(x)
        skips5 = LeakyReLU(self.skip5_bn(x))

        #---------------------------------------------------------------------
        #------------------------------ Upsampling ---------------------------
        #---------------------------------------------------------------------

        #we start at the fifth layer (layers have reverse order)
        x = self.up5_bn_initial(skips5) #notice no concatenation here
        x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
        x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
        x = self.upsample(x)

        #now we upsample all the other layers
        #print("up4")
        #print(x.size())
        #print(skips4.size())
        x = self.up4_bn_initial(x)
        x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
        x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
        x = self.upsample(x)

        #print("up3")
        #print(x.size())
        #print(skips3.size())
        x = self.up3_bn_initial(x)
        x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
        x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
        x = self.upsample(x)

        #print("up2")
        #print(x.size())
        #print(skips2.size())
        x = self.up2_bn_initial(x)
        x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
        x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
        x = self.upsample(x)

        #print("up1")
        #print(x.size())
        #print(skips1.size())
```

```
        x = self.up1_bn_initial(x)
        x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
        x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
        x = self.upsample(x)


        #----------------------------------------------------------------------
        #------------------------------ Finalize ------------------------------
        #----------------------------------------------------------------------

        #now we get x to the right dimensions
        x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here

        return x
```

```
#generate noisy image as input
#noiseType = "uniform"
from torch.autograd import Variable

def createNoisyImage(input_depth,dimensions,var=0.1,noiseType='uniform'):
  tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
  networkInput = Variable(torch.zeros(tensorShape))

  if noiseType == "uniform":
    networkInput.data.uniform_()
  elif noiseType == "normal":
    networkInput.data.normal_()

  networkInput.data = networkInput.data * var

  return networkInput

def createNoisyImagesWithOneHot(numImages, input_depth,dimensions,var=0.1,noiseType='unif
  mynoisy = createNoisyImage(input_depth,dimensions,var,noiseType)

  tensorShape = [numImages,input_depth+numImages,dimensions[0],dimensions[1]]
  fourDinput = Variable(torch.zeros(tensorShape))

  for i in range(0, numImages):
      ithoutput = mynoisy
      ithoutput = padit(mynoisy, numImages, i)
      fourDinput[i] = ithoutput[0]

  return fourDinput

def padit(originalImage, numImages, padIndex):
  outImage = originalImage

  oneImageShape = [1,1,originalImage.shape[2],originalImage.shape[3]]
  myzeros = Variable(torch.zeros(oneImageShape))
  myones = Variable(torch.ones(oneImageShape))

  for j in range(0,numImages):
    if j==padIndex:
      outImage = torch.cat((outImage, myones),1)
    else:
      outImage = torch.cat((outImage, myzeros),1)
  return outImage

oneHotInput = createNoisyImagesWithOneHot(4, 32,[maskCropped.shape[0],maskCropped.shape[1
```

```
#Checking that we have the right 0 and 1 images at the right places
print(oneHotInput.shape)

samInput = oneHotInput.data[2,:,:,:]
print(samInput[34])
```

```
#plt.imshow(np.array(samInput)[33])
```

```
torch.Size([4, 36, 704, 1280])

      1     1     1  ...      1     1     1
      1     1     1  ...      1     1     1
      1     1     1  ...      1     1     1
         ...           ⋱           ...
      1     1     1  ...      1     1     1
      1     1     1  ...      1     1     1
      1     1     1  ...      1     1     1
[torch.FloatTensor of size 704x1280]
```

```
#size is [1, channels, height, width
maskedTarget = Variable(torch.Tensor(withoutsubsCropped))
maskCroppedTensor = Variable(torch.Tensor(maskCropped))

maskedTarget = maskedTarget.cuda()
maskCroppedTensor = maskCroppedTensor.cuda()
```

Train Autoencoder SIDN

This has different attempts to do transfer learning. See report for details.

Naturally training is testing here, so we save outputs to images and upload these images to gdrive

```
# Saving any model
def save_model(model, filename):
  torch.save(model, filename)

def save_model_statedict(model, filename):
  torch.save(model.state_dict(), filename)

def load_model(filename):
  model = torch.load(filename)
  model.cuda()
  return model
```

```
#Saving a random model
net = Net()
net.cuda()
save_model(net, 'random_init.pt')
```

```
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:158: UserWarning: Couldn
  "type " + obj.__name__ + ". It won't be checked "
```

```
#reset your net
net = Net()
net.cuda()
```

```
Net(
  (down1_downsample): Conv2d (36, 16, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3
  (down1_bn_downsample): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
  (down1_justconv): Conv2d (16, 16, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (down1_bn_justconv): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
  (down2_downsample): Conv2d (16, 32, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3
  (down2_bn_downsample): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (down2_justconv): Conv2d (32, 32, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (down2_bn_justconv): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (down3_downsample): Conv2d (32, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3
  (down3_bn_downsample): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (down3_justconv): Conv2d (64, 64, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (down3_bn_justconv): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (down4_downsample): Conv2d (64, 128, kernel_size=(7, 7), stride=(2, 2), padding=(3,
  (down4_bn_downsample): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down4_justconv): Conv2d (128, 128, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3
  (down4_bn_justconv): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down5_downsample): Conv2d (128, 128, kernel_size=(7, 7), stride=(2, 2), padding=(3,
  (down5_bn_downsample): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down5_justconv): Conv2d (128, 128, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3
  (down5_bn_justconv): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (skip5_conv): Conv2d (128, 4, kernel_size=(1, 1), stride=(1, 1))
  (skip5_bn): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True)
  (upsample): Upsample(scale_factor=2, mode=nearest)
  (makeFinalOutput): Conv2d (16, 3, kernel_size=(1, 1), stride=(1, 1))
  (up5_bn_initial): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True)
  (up5_conv1): Conv2d (4, 128, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (up5_bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up5_conv2): Conv2d (128, 128, kernel_size=(1, 1), stride=(1, 1))
  (up5_bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up4_bn_initial): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up4_conv1): Conv2d (128, 128, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (up4_bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up4_conv2): Conv2d (128, 128, kernel_size=(1, 1), stride=(1, 1))
  (up4_bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up3_bn_initial): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up3_conv1): Conv2d (128, 64, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (up3_bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (up3_conv2): Conv2d (64, 64, kernel_size=(1, 1), stride=(1, 1))
  (up3_bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (up2_bn_initial): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (up2_conv1): Conv2d (64, 32, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (up2_bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (up2_conv2): Conv2d (32, 32, kernel_size=(1, 1), stride=(1, 1))
  (up2_bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (up1_bn_initial): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (up1_conv1): Conv2d (32, 16, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (up1_bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
  (up1_conv2): Conv2d (16, 16, kernel_size=(1, 1), stride=(1, 1))
  (up1_bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
)
```

```
#Download the original and processed 8frames data
file_list = drive.ListFile({'q': "'1n7iCwVNkUtMZE9TyRMXvo2bCPn82-Od1' in parents and tras
i = 1
for file1 in sorted(file_list, key = lambda x: x['title']):
    print('Downloading {} from GDrive ({}/{})'.format(file1['title'], i, len(file_list)))
```

```
    file1.GetContentFile(file1['title'])
    i += 1
```

```
Downloading her1.png from GDrive (1/8)
Downloading her2.png from GDrive (2/8)
Downloading her3.png from GDrive (3/8)
Downloading her4.png from GDrive (4/8)
Downloading him1.png from GDrive (5/8)
Downloading him2.png from GDrive (6/8)
Downloading him3.png from GDrive (7/8)
Downloading him4.png from GDrive (8/8)
```

```
her1 = mpimg.imread('her1.png')[8:712,:,:]
her2 = mpimg.imread('her2.png')[8:712,:,:]
her3 = mpimg.imread('her3.png')[8:712,:,:]
her4 = mpimg.imread('her4.png')[8:712,:,:]
images = [her1, her2, her3, her4]
images = np.array(images)
multiTarget = Variable(torch.Tensor(images))

fname = "multiTraining3k"

torch.save(oneHotInput, fname+'oneHotVariable.pt')
file1 = drive.CreateFile()
file1.SetContentFile(fname+'oneHotVariable.pt')
file1.Upload() # Upload the file.
```

```
#changing the epochs to 1000 for the sake of this file
def train_model_multiImage(net, oneHotInput, target, numImages, criterion, optimizer, max
  target = target.cuda()
  oneHotInput = oneHotInput.cuda()

  for epoch in range(maxepochs):  # loop over the data multiple times
      # zero the parameter gradients
      optimizer.zero_grad()

      #print("backpropagation time")
      # forward + backward + optimize
      #THIS SHOULD BE A LOOP
      loss = 0
      #for i in range(0, numImages):
      outputs = net(oneHotInput)
      #outputs = outputs[0]
      outputs = outputs.permute(0,2,3,1)
      loss = criterion(outputs, target)
      loss.backward()

      optimizer.step()

      #print some results
      if epoch == 0:
          k = -1
      else:
          k = -2
      print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - startime), loss.data[
      trainLoss.append(loss.data[0])

  print("Final train loss")
  print(trainLoss[len(trainLoss)-1])

  print('Finished Training. See training time on next line')
  print(int(time.time() - startime))
  save_model(net, fname+'trained.pt')
```

```
  file1 = drive.CreateFile()
  file1.SetContentFile(fname+'trained.pt')
  file1.Upload() # Upload the file.
  return net
```

```
#changing the epochs to 1000 for the sake of this file
def train_model_loadMultiModel(net, numImages, inputs, mask_fname, withoutsubs_fname, cri
  mask = mpimg.imread(mask_fname)
  withoutsubs = mpimg.imread(withoutsubs_fname)
  withoutsubsCropped = withoutsubs[8:712,:,:]
  maskCropped = mask[8:712,:,:]
  maskedTarget = Variable(torch.Tensor(withoutsubsCropped))
  maskCroppedTensor = Variable(torch.Tensor(maskCropped))

  maskedTarget = maskedTarget.cuda()
  maskCroppedTensor = maskCroppedTensor.cuda()
  inputs = inputs.cuda()

  for epoch in range(maxepochs):  # loop over the data multiple times
      # get the label #NEEDS TO BE 4D TENSOR
      label = maskedTarget

      # wrap them in Variable
      inputs, label = inputs.cuda(), label.cuda()

      # zero the parameter gradients
      optimizer.zero_grad()

      #print("backpropagation time")
      # forward + backward + optimize
      #THIS SHOULD BE A LOOP
      loss = 0
      outputs = net(inputs)
      outputs = outputs[0]
      outputs = outputs.permute(1,2,0)
      loss = criterion(torch.mul(outputs, maskCroppedTensor), label)
      loss.backward()

      optimizer.step()

      #if epoch%5==0:
        #print("We are at epoch " + str(epoch))

      #save the image every 100 epochs
      if epoch%50==0:
  #         pp = PdfPages('subsTest' + str(epoch) + '.pdf')

          arr = np.array(outputs.data)
          arr = np.clip(arr, 0, 1)
          plt.rcParams["axes.grid"] = False
          plt.imshow(arr)
  #         plt.savefig(pp, format='pdf')
  #         pp.close()
          plt.imsave(fname + 'subsTest' + str(epoch) + '.png', arr)
          file1 = drive.CreateFile()
          file1.SetContentFile(fname + 'subsTest' + str(epoch) + '.png')
          file1.Upload() # Upload the file.
          plt.show()

      #print some results
      if epoch == 0:
          k = -1
      else:
          k = -2
      print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - startime), loss.data[
      trainLoss.append(loss.data[0])

  print("Final train loss")
```

```
  print(trainLoss[len(trainLoss)-1])

  print('Finished Training. See training time on next line')
  print(int(time.time() - startime))
  save_model(net, fname+'trained.pt')
  file1 = drive.CreateFile()
  file1.SetContentFile(fname+'trained.pt')
  file1.Upload() # Upload the file.
  return net
```

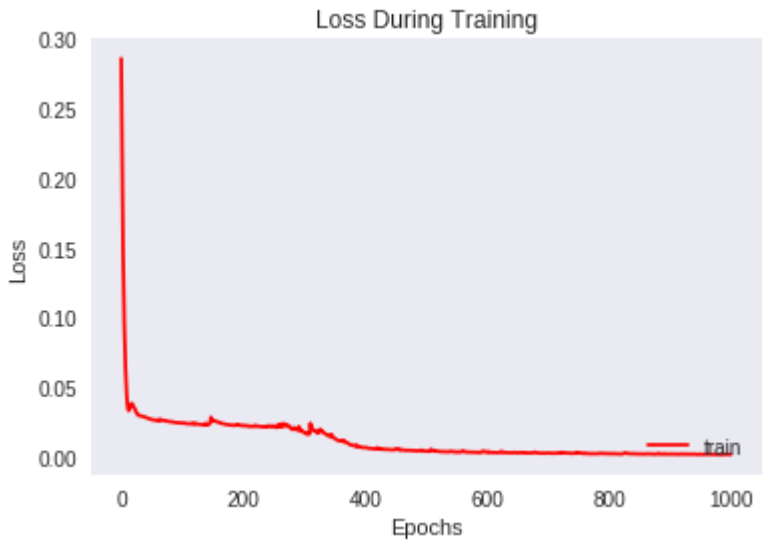
```
#Set up Training
#Loss Function = MSE
#Get image in Tensor form. This is the label
dtype = torch.cuda.FloatTensor

print('Started Training')
startime = time.time();

numepochs = 0

acc = []
trainLoss = []
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)

train_model_multiImage(net, oneHotInput, multiTarget, 4, criterion, optimizer, maxepochs=
```

```
[580, 857 s] loss: 0.003
[581, 858 s] loss: 0.003
[582, 859 s] loss: 0.003
[583, 861 s] loss: 0.003
[584, 862 s] loss: 0.003
[585, 864 s] loss: 0.003
[586, 865 s] loss: 0.003
[587, 867 s] loss: 0.003
[588, 868 s] loss: 0.003
[589, 870 s] loss: 0.003
[590, 871 s] loss: 0.003
[591, 873 s] loss: 0.003
[592, 874 s] loss: 0.004
[593, 876 s] loss: 0.004
[594, 877 s] loss: 0.004
[595, 879 s] loss: 0.004
[596, 880 s] loss: 0.004
[597, 882 s] loss: 0.004
[598, 883 s] loss: 0.004
[599, 885 s] loss: 0.004
[600, 886 s] loss: 0.004
[601, 888 s] loss: 0.004
[602, 889 s] loss: 0.004
[603, 891 s] loss: 0.003
[604, 892 s] loss: 0.004
[605, 894 s] loss: 0.003
```

```
save_model_statedict(net, fname+'m1_3k_initTraining.pt')
file1 = drive.CreateFile()
file1.SetContentFile('multiTrain3ktrained.pt')
file1.Upload() # Upload the file.
```

```
[611, 902 s] loss: 0.003
```

```
plt.figure()
plt.title('Loss During Training')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(trainLoss, 'r', label="train")
plt.legend(loc=4)
plt.show()
```




```
[890, 933 s] loss: 0.003
```

```
oneHotInput = createNoisyImagesWithOneHot(4, 36,[maskCropped.shape[0],maskCropped.shape[1
```

```
#initial training (not shown)
model = Net()
model.cuda()
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
i=3
mask_fname = 'snapshot' + str(i) + '_mask.png'
withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'
model_without_transfer = train_model(net, mask_fname, withoutsubs_fname, criterion, optim
```

```
#main training with transfer

def makeInputsWithZeroOneHot(onehotinput, numImages):
  #assumes 32 channels in noisy input
  noisyImage = onehotinput[0:1,0:32,:,:]
  outImage = padit(noisyImage, numImages, -1)
  return outImage

zeroOneHotInput = makeInputsWithZeroOneHot(oneHotInput, 4)

# Try multi
#model = load_model('multiTraintrained.pt')
acc = []
trainLoss = []

criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
i=3
mask_fname = 'snapshot' + str(i) + '_mask.png'
withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'
model_after_transfer = train_model_loadMultiModel(net, 4, zeroOneHotInput, mask_fname, wi
```

```
[971, 3656 s] loss: 0.000
[972, 3657 s] loss: 0.000
[973, 3658 s] loss: 0.000
[974, 3658 s] loss: 0.000
[975, 3659 s] loss: 0.000
[976, 3660 s] loss: 0.000
[977, 3660 s] loss: 0.000
[978, 3661 s] loss: 0.000
[979, 3661 s] loss: 0.000
[980, 3662 s] loss: 0.000
[981, 3663 s] loss: 0.000
[982, 3663 s] loss: 0.000
[983, 3664 s] loss: 0.000
[984, 3665 s] loss: 0.000
[985, 3665 s] loss: 0.000
[986, 3666 s] loss: 0.000
[987, 3667 s] loss: 0.000
[988, 3667 s] loss: 0.000
[989, 3668 s] loss: 0.000
[990, 3668 s] loss: 0.000
[991, 3669 s] loss: 0.000
[992, 3670 s] loss: 0.000
[993, 3670 s] loss: 0.000
[994, 3671 s] loss: 0.000
[995, 3672 s] loss: 0.000
[996, 3672 s] loss: 0.000
[997, 3673 s] loss: 0.000
[998, 3674 s] loss: 0.000
[999, 3674 s] loss: 0.000
[1000, 3675 s] loss: 0.000
Final train loss
0.00035066052805632353
Finished Training. See training time on next line
3675
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:158: UserWarning: Couldn
```

General Inpainting (not text inpaining)

```
<ipython-input-72-1c0da7cfb174> in <module>()
```

```
import requests
import shutil
path = 'library.png'
r = requests.get('https://github.com/DmitryUlyanov/deep-image-prior/raw/master/data/inpai
if r.status_code == 200:
    with open(path, 'wb') as f:
        r.raw.decode_content = True
        shutil.copyfileobj(r.raw, f)
path = 'library_mask.png'
r = requests.get('https://github.com/DmitryUlyanov/deep-image-prior/raw/master/data/inpai
if r.status_code == 200:
    with open(path, 'wb') as f:
        r.raw.decode_content = True
        shutil.copyfileobj(r.raw, f)
```

```
mask = mpimg.imread(mask_fname)
mask = mask[:, :, None]
image = mpimg.imread(image_fname)
mask = 1 - mask
```

```
out = np.clip(mask+image,0,1)
plt.imshow(out)
plt.imsave('masked_image.png', out)
```


```
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

LeakyReLU = nn.LeakyReLU(0.2, inplace=True)

num_channels_down=[16, 32, 64, 128, 128, 128]
num_channels_up=[16, 32, 64, 128, 128, 128]
num_channels_skip=[0, 0, 0, 0, 0, 0]
num_channels_input = 1
k_down = 5
k_up = 3

padding_down = int((k_down-1)/2)
padding_up = int((k_up-1)/2)
# padding_skip = int((k_skip-1)/2)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        #------------------------------------------------------------------------
        #----------------------------- Down blocks ------------------------------
        #------------------------------------------------------------------------

        i = 0 # for first
        #first down block, others are copypaste
        self.down1_downsample = nn.Conv2d(num_channels_input, num_channels_down[i], k_dow
        self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 1
        self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 2
```

```
        self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 3
        self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 4
        self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])

        i = 5
        self.down6_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
        self.down6_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
        self.down6_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
        self.down6_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
        #----------------------------------------------------------------------
        #----------------------------- Skip blocks ----------------------------
        #----------------------------------------------------------------------

#         i = 0 # for first
#         #first skip block, others are copypaste
#         self.skip1_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
#         self.skip1_bn = nn.BatchNorm2d(num_channels_skip[i])

#         i = 1
#         self.skip2_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
#         self.skip2_bn = nn.BatchNorm2d(num_channels_skip[i])

#         i = 2
#         self.skip3_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
#         self.skip3_bn = nn.BatchNorm2d(num_channels_skip[i])

#         i = 3
#         self.skip4_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
#         self.skip4_bn = nn.BatchNorm2d(num_channels_skip[i])

#         i = 4
#         self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
#         self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])

        #----------------------------------------------------------------------
        #--------------------------- Misc but needed --------------------------
        #----------------------------------------------------------------------

        #will need this for the up layers
        self.upsample = nn.Upsample(scale_factor=2, mode='nearest') #at the end
        num_channels_up.append(num_channels_down[i]) #this way for the fifth layers the c
        #will need this at the end of network
        self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)

        #----------------------------------------------------------------------
        #----------------------------- Up blocks ------------------------------
        #----------------------------------------------------------------------

        i=5 # for fifth
        #fifth up block, others are copypaste
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
        self.up6_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up6_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up6_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up6_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up6_bn2 = nn.BatchNorm2d(num_channels_up[i])

```

```
        i=4 # for fifth
        #fifth up block, others are copypaste
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
        self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=3
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up4_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up4_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=2
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up3_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up3_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=1
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up2_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up2_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])

        i=0
        combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
        self.up1_bn_initial = nn.BatchNorm2d(combinedNumChannels)
        self.up1_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
        self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
        self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
        self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])

    def forward(self, x):
        #------------------------------------------------------------------------
        #-------------------- Downsampling and recording skip -------------------
        #------------------------------------------------------------------------
        x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
        x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))
#         skips1 = LeakyReLU(self.skip1_bn(self.skip1_conv(x)))

        x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
        x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))
#         skips2 = LeakyReLU(self.skip2_bn(self.skip2_conv(x)))

        x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
        x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))
#         skips3 = LeakyReLU(self.skip3_bn(self.skip3_conv(x)))

        x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
        x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))
#         skips4 = LeakyReLU(self.skip4_bn(self.skip4_conv(x)))

        x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
        x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
#         skips5 = LeakyReLU(self.skip5_bn(self.skip5_conv(x)))

        x = LeakyReLU(self.down6_bn_downsample(self.down6_downsample(x)))
        x = LeakyReLU(self.down6_bn_justconv(self.down6_justconv(x)))

        #------------------------------------------------------------------------
        #------------------------------ Upsampling ------------------------------
```

```
        #-------------------------------------------------------------------

        x = self.up6_bn_initial(x) #notice no concatenation here
        x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
        x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
        x = self.upsample(x)

        #we start at the fifth layer (layers have reverse order)
        x = self.up5_bn_initial(x) #notice no concatenation here
        x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
        x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
        x = self.upsample(x)

        #now we upsample all the other layers
        #print("up4")
        #print(x.size())
        #print(skips4.size())
        x = self.up4_bn_initial(x)
        x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
        x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
        x = self.upsample(x)

        #print("up3")
        #print(x.size())
        #print(skips3.size())
        x = self.up3_bn_initial(x)
        x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
        x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
        x = self.upsample(x)

        #print("up2")
        #print(x.size())
        #print(skips2.size())
        x = self.up2_bn_initial(x)
        x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
        x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
        x = self.upsample(x)

        #print("up1")
        #print(x.size())
        #print(skips1.size())
        x = self.up1_bn_initial(x)
        x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
        x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
        x = self.upsample(x)


        #-------------------------------------------------------------------------
        #------------------------------ Finalize ---------------------------------
        #-------------------------------------------------------------------------

        #now we get x to the right dimensions
        x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here

        return x
```

```
#Saving a random model
net = Net()
net.cuda()
```

```
Net(
  (down1_downsample): Conv2d (1, 16, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2)
  (down1_bn_downsample): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
  (down1_justconv): Conv2d (16, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (down1_bn_justconv): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
  (down2_downsample): Conv2d (16, 32, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2
  (down2_bn_downsample): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (down2_justconv): Conv2d (32, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (down2_bn_justconv): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (down3_downsample): Conv2d (32, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2
  (down3_bn_downsample): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (down3_justconv): Conv2d (64, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (down3_bn_justconv): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (down4_downsample): Conv2d (64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2,
  (down4_bn_downsample): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down4_justconv): Conv2d (128, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2
  (down4_bn_justconv): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down5_downsample): Conv2d (128, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2,
  (down5_bn_downsample): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down5_justconv): Conv2d (128, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2
  (down5_bn_justconv): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down6_downsample): Conv2d (128, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2,
  (down6_bn_downsample): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (down6_justconv): Conv2d (128, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2
  (down6_bn_justconv): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (upsample): Upsample(scale_factor=2, mode=nearest)
  (makeFinalOutput): Conv2d (16, 3, kernel_size=(1, 1), stride=(1, 1))
  (up6_bn_initial): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up6_conv1): Conv2d (128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (up6_bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up6_conv2): Conv2d (128, 128, kernel_size=(1, 1), stride=(1, 1))
  (up6_bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up5_bn_initial): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up5_conv1): Conv2d (128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (up5_bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up5_conv2): Conv2d (128, 128, kernel_size=(1, 1), stride=(1, 1))
  (up5_bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up4_bn_initial): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up4_conv1): Conv2d (128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (up4_bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up4_conv2): Conv2d (128, 128, kernel_size=(1, 1), stride=(1, 1))
  (up4_bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up3_bn_initial): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
  (up3_conv1): Conv2d (128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (up3_bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (up3_conv2): Conv2d (64, 64, kernel_size=(1, 1), stride=(1, 1))
  (up3_bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (up2_bn_initial): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (up2_conv1): Conv2d (64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (up2_bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (up2_conv2): Conv2d (32, 32, kernel_size=(1, 1), stride=(1, 1))
  (up2_bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
  (up1_bn_initial): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
```

```
def createNoisyImage(input_depth,dimensions,var=0.1,noiseType='uniform'):
  tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
  networkInput = Variable(torch.zeros(tensorShape))

```

```
  if noiseType == "uniform":
    networkInput.data.uniform_()
  elif noiseType == "normal":
    networkInput.data.normal_()

  networkInput.data = networkInput.data * var

  return networkInput



superResolutionBase = createNoisyImage(1,[448,704],0.1,"uniform")
base = superResolutionBase
savedBase = base.data.clone().cuda()
baseNoise = base.data.clone().cuda()

def train_model_inpainting(net, mask_fname, image_fname, criterion, optimizer, maxepochs=
  mask = mpimg.imread(mask_fname)
  mask = mask[None, :]
  image = mpimg.imread(image_fname)
  imageTensor = Variable(torch.Tensor(image))
  imageTensor = imageTensor.permute(2, 0, 1)
  maskTensor = Variable(torch.Tensor(mask))
  imageTensor = imageTensor.cuda()
  maskTensor = maskTensor.cuda()

  # get the input
  inputs = superResolutionBase #this is the z = uniform noise input

  for epoch in range(maxepochs):  # loop over the data multiple times
      for n in [x for x in net.parameters() if len(x.size()) == 4]:
        n.data += n.data.clone().normal_()*n.data.std()/50

      # get the label
      label = imageTensor

      # wrap them in Variable
      inputs, label = inputs.cuda(), label.cuda()

      # zero the parameter gradients
      optimizer.zero_grad()

#       inputs = addRandomNoise(inputs)

      #print("backpropagation time")
      # forward + backward + optimize
      outputs = net(inputs)
      outputs=outputs[0]
#       outputs = outputs.permute(1,2,0)
      loss = criterion(outputs * maskTensor, label * maskTensor)
      loss.backward()
      optimizer.step()

      #if epoch%5==0:
        #print("We are at epoch " + str(epoch))

      #save the image every 100 epochs
      if epoch%100==0:
  #         pp = PdfPages('subsTest' + str(epoch) + '.pdf')

        arr = np.array(outputs.data)
        arr = arr.transpose(1,2,0)
        arr = np.clip(arr, 0, 1)
        plt.rcParams["axes.grid"] = False
        plt.imshow(arr)
#         plt.savefig(pp, format='pdf')
#         pp.close()
        plt.imsave(fname + 'subsTest' + str(epoch) + '.png', arr)
        file1 = drive.CreateFile()
        file1.SetContentFile(fname + 'subsTest' + str(epoch) + '.png')
```

```
        file1.Upload() # Upload the file.
        plt.show()
        pass

      #print some results
      if epoch == 0:
          k = -1
      else:
          k = -2
      print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - startime), loss.data[
      trainLoss.append(loss.data[0])

  print("Final train loss")
  print(trainLoss[len(trainLoss)-1])

  print('Finished Training. See training time on next line')
  print(int(time.time() - startime))
  save_model(net, fname+'trained.pt')
  file1 = drive.CreateFile()
  file1.SetContentFile(fname+'trained.pt')
  file1.Upload() # Upload the file.
  return net
```

```
#Set up Training
#Loss Function = MSE
#Get image in Tensor form. This is the label
dtype = torch.cuda.FloatTensor

print('Started Training')
startime = time.time();

maxepochs = 3000
numepochs = 0

# horizon = 5

acc = []
trainLoss = []
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr = 0.01)
i=1
mask_fname = 'library_mask.png'
image_fname = 'library.png'
train_model_inpainting(net, mask_fname, image_fname, criterion, optimizer, maxepochs=5000
```

[4490, 418 s] loss: 0.008
[4491, 418 s] loss: 0.008
[4492, 419 s] loss: 0.008
[4493, 419 s] loss: 0.008
[4494, 419 s] loss: 0.008
[4495, 419 s] loss: 0.008
[4496, 419 s] loss: 0.008
[4497, 419 s] loss: 0.008
[4498, 419 s] loss: 0.008
[4499, 419 s] loss: 0.008
[4500, 419 s] loss: 0.008
Your cell size is close to the size limit (9M). Large new outputs may be dropped. So f
[4502, 421 s] loss: 0.008
[4503, 421 s] loss: 0.008
[4504, 421 s] loss: 0.008
[4505, 421 s] loss: 0.008
[4506, 421 s] loss: 0.008
[4507, 421 s] loss: 0.008
[4508, 421 s] loss: 0.008
[4509, 421 s] loss: 0.008
[4510, 421 s] loss: 0.008
[4511, 421 s] loss: 0.008
[4512, 421 s] loss: 0.008
[4513, 421 s] loss: 0.008
[4514, 421 s] loss: 0.008
[4515, 422 s] loss: 0.008
[4516, 422 s] loss: 0.008
[4517, 422 s] loss: 0.008
[4518, 422 s] loss: 0.008
[4519, 422 s] loss: 0.008
[4520, 422 s] loss: 0.008
[4521, 422 s] loss: 0.008
[4522, 422 s] loss: 0.008
[4523, 422 s] loss: 0.008
[4524, 422 s] loss: 0.008
[4525, 422 s] loss: 0.008
[4526, 422 s] loss: 0.008
[4527, 423 s] loss: 0.008
[4528, 423 s] loss: 0.008
[4529, 423 s] loss: 0.008
[4530, 423 s] loss: 0.008
[4531, 423 s] loss: 0.008
[4532, 423 s] loss: 0.008
[4533, 423 s] loss: 0.008
[4534, 423 s] loss: 0.008
[4535, 423 s] loss: 0.008
[4536, 423 s] loss: 0.008
[4537, 423 s] loss: 0.008
[4538, 423 s] loss: 0.008
[4539, 423 s] loss: 0.008
[4540, 424 s] loss: 0.008
[4541, 424 s] loss: 0.008
[4542, 424 s] loss: 0.008
[4543, 424 s] loss: 0.008
[4544, 424 s] loss: 0.008
[4545, 424 s] loss: 0.008
[4546, 424 s] loss: 0.008
[4547, 424 s] loss: 0.008

[4547, 424 s] loss: 0.008
[4548, 424 s] loss: 0.008