

Imports and Basic Setup

```
1 # http://pytorch.org/
2 from os import path
3 from wheel.pep425tags import get_abbr_impl, get_impl_ver, get_abi_tag
4 import numpy as np
5 import time
6 from __future__ import division
7
8 platform = '{}{}-{}'.format(get_abbr_impl(), get_impl_ver(), get_abi_tag())
9
10 accelerator = 'cu80' if path.exists('/opt/bin/nvidia-smi') else 'cpu'
11
12 !pip install -q http://download.pytorch.org/whl/{accelerator}/torch-0.3.0.post4-{platform}
13 import torch
```

```
1 import os
2 import torch
3 import torchvision
4 from torchvision import datasets, transforms
5 from torch.utils.data.sampler import SubsetRandomSampler
6 from matplotlib.backends.backend_pdf import PdfPages
7 import matplotlib.pyplot as plt
8 import matplotlib.image as mpimg
9 import numpy as np
10
```

```
1 !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
```

```
1 !pip install -U -q PyDrive
2 from pydrive.auth import GoogleAuth
3 from pydrive.drive import GoogleDrive
4 from google.colab import auth
5 from oauth2client.client import GoogleCredentials
6 import datetime
7
8 auth.authenticate_user()
9 gauth = GoogleAuth()
10 gauth.credentials = GoogleCredentials.get_application_default()
11 drive = GoogleDrive(gauth)
```

▼ Downsample

```
1 # Downsample(n_planes=3, factor=4, kernel_type='lanczos2', phase=0.5, preserve_size=True
2 from torch import nn
3 class Downsample(nn.Module):
4     def __init__(self, n_planes, factor, preserve_size):
5         super(Downsample, self).__init__()
6         kernel_width = 4 * factor + 1
7         self.kernel = get_kernel(factor, kernel_width)
8         downampler = nn.Conv2d(n_planes, n_planes, kernel_size=self.kernel.shape, stride=fac
9         downampler.weight.data[:] = 0
10        downampler.bias.data[:] = 0
11        kernel_torch = torch.from_numpy(self.kernel)
12        for i in range(n_planes):
13            downampler.weight.data[i, i] = kernel_torch
14        self.downampler_ = downampler
15
16        if preserve_size:
17
18            if self.kernel.shape[0] % 2 == 1:
19                pad = int((self.kernel.shape[0] - 1) / 2.)
20            else:
21                pad = int((self.kernel.shape[0] - factor) / 2.)
22
23            self.padding = nn.ReplicationPad2d(pad)
24
25        self.preserve_size = preserve_size
26
27    def forward(self, input):
28        if self.preserve_size:
29            x = self.padding(input)
30        else:
31            x= input
32        self.x = x
33        return self.downampler_(x)
34
35    def get_kernel(factor, kernel_width):
36        support = 2
37        kernel = np.zeros([kernel_width - 1, kernel_width - 1])
38        center = (kernel_width + 1) / 2.
39
40        for i in range(1, kernel.shape[0] + 1):
41            for j in range(1, kernel.shape[1] + 1):
42
43                di = abs(i + 0.5 - center) / factor
44                dj = abs(j + 0.5 - center) / factor
45
46                pi_sq = np.pi * np.pi
47
48                val = 1
49                if di != 0:
50                    val = val * support * np.sin(np.pi * di) * np.sin(np.pi * di / support)
51                    val = val / (np.pi * np.pi * di * di)
52
53                if dj != 0:
54                    val = val * support * np.sin(np.pi * dj) * np.sin(np.pi * dj / support)
55                    val = val / (np.pi * np.pi * dj * dj)
56
57                kernel[i - 1][j - 1] = val
58        kernel /= kernel.sum()
59        return kernel
```

▼ Data import for superresolution dip network

```
1 import os  
2  
3 # Fetch the folder with all the input images  
4 file_list = drive.ListFile({'q': "'11a5-Sek52Fq86iHXoK6KQ-Gvr1G8El0Y' in parents; and trac  
5  
6 # Create an input directory (no need to execute cuz created)  
7 os.mkdir('input')
```

```
1 # Store files in the input directory  
2 i = 1  
3 for file1 in sorted(file_list, key = lambda x: x['title']):  
4     print('Downloading {} from GDrive {}'.format(file1['title'], i, len(file_list)))  
5     file1.GetContentFile('input/' + file1['title'])  
6     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)
```

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

⇨

```
Downloading sun4_v4_input.jpg from GDrive (20/58)
```

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import matplotlib.image as mpimg
4 import numpy as np
5
6 path = 'input/einstein.jpg'
7 image = mpimg.imread(path)
8 plt.imshow(image)
```

⇨

(8c/8c) **אנו מודים לך!** **הנחתה נספחה לך!** **תודה!**

```

1 image.shape
→

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

```

▼ Build superresolution dip network

```

1 from torch.autograd import Variable
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import torch.optim as optim
5
6 LeakyReLU = nn.LeakyReLU(0.2, inplace=True)
7
8 num_channels_down=[128, 128, 128, 128, 128]
9 num_channels_up=[128, 128, 128, 128, 128]
10 num_channels_skip=[4, 4, 4, 4, 4]
11 num_channels_input = 32
12 k_down = 3
13 k_up = 3
14 k_skip = 1
15
16 padding_down = int((k_down-1)/2)
17 padding_up = int((k_up-1)/2)
18 padding_skip = int((k_skip-1)/2)
19
20 class Net(nn.Module):
21     def __init__(self):
22         super(Net, self).__init__()
23
24     #-----
25     #----- Down blocks -----
26     #-----
27
28     i = 0 # for first
29     #first down block, others are copypaste
30     self.down1_downsample = nn.Conv2d(num_channels_input, num_channels_down[i], k_down)
31     self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
32     self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_down)
33     self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
34
35     i = 1
36     self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k_down)
37     self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
38     self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_down)
39     self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
40
41     i = 2
42     self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k_down)
43     self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
44     self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_down)
45     self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
46
47     i = 3
48     self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k_down)
49     self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])

```

```

50     self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
51     self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
52
53     i = 4
54     self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
55     self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
56     self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
57     self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
58
59     #-----
60     #----- Skip blocks -----
61     #-----
62
63     i = 0 # for first
64     #first skip block, others are copypaste
65     self.skip1_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
66     self.skip1_bn = nn.BatchNorm2d(num_channels_skip[i])
67
68     i = 1
69     self.skip2_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
70     self.skip2_bn = nn.BatchNorm2d(num_channels_skip[i])
71
72     i = 2
73     self.skip3_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
74     self.skip3_bn = nn.BatchNorm2d(num_channels_skip[i])
75
76     i = 3
77     self.skip4_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
78     self.skip4_bn = nn.BatchNorm2d(num_channels_skip[i])
79
80     i = 4
81     self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
82     self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])
83
84     #-----
85     #----- Misc but needed -----
86     #-----
87
88     #will need this for the up layers
89     self.upsample = nn.Upsample(scale_factor=2, mode='bilinear') #at the end
90     num_channels_up.append(0) #this way for the fifth layers the combined is just the
91     #will need this at the end of network
92     self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)
93
94     #-----
95     #----- Up blocks -----
96     #-----
97
98     i=4 # for fifth
99     #fifth up block, others are copypaste
100    combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
101    self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
102    self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
103    self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
104    self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
105    self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])
106
107    i=3
108    combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
109    self.up4_bn_initial = nn.BatchNorm2d(combinedNumChannels)
110    self.up4_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
111    self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
112    self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
113    self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])
114
115    i=2
116    combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
117    self.up3_bn_initial = nn.BatchNorm2d(combinedNumChannels)
118    self.up3_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
119    self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])

```

```

120     self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
121     self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])
122
123     i=1
124     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
125     self.up2_bn_initial = nn.BatchNorm2d(combinedNumChannels)
126     self.up2_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
127     self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
128     self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
129     self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])
130
131     i=0
132     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
133     self.up1_bn_initial = nn.BatchNorm2d(combinedNumChannels)
134     self.up1_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
135     self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
136     self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
137     self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])
138
139 def forward(self, x):
140     #-----
141     #----- Downsampling and recording skip -----
142     #-
143     x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
144     x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))
145     skips1 = LeakyReLU(self.skip1_bn(self.skip1_conv(x)))
146
147     x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
148     x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))
149     skips2 = LeakyReLU(self.skip2_bn(self.skip2_conv(x)))
150
151     x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
152     x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))
153     skips3 = LeakyReLU(self.skip3_bn(self.skip3_conv(x)))
154
155     x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
156     x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))
157     skips4 = LeakyReLU(self.skip4_bn(self.skip4_conv(x)))
158
159     x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
160     x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
161     skips5 = LeakyReLU(self.skip5_bn(self.skip5_conv(x)))
162
163     #-
164     #----- Upsampling -----
165     #-
166
167     #we start at the fifth layer (layers have reverse order)
168     x = self.up5_bn_initial(skips5) #notice no concatenation here
169     x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
170     x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
171     x = self.upsample(x)
172
173     #now we upsample all the other layers
174     #print("up4")
175     #print(x.size())
176     #print(skips4.size())
177     x = self.up4_bn_initial(torch.cat((x, skips4),1))
178     x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
179     x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
180     x = self.upsample(x)
181
182     #print("up3")
183     #print(x.size())
184     #print(skips3.size())
185     x = self.up3_bn_initial(torch.cat((x, skips3),1))
186     x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
187     x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
188     x = self.upsample(x)
189

```

```

190     #print("up2")
191     #print(x.size())
192     #print(skips2.size())
193     x = self.up2_bn_initial(torch.cat((x, skips2),1))
194     x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
195     x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
196     x = self.upsample(x)
197
198     #print("up1")
199     #print(x.size())
200     #print(skips1.size())
201     x = self.up1_bn_initial(torch.cat((x, skips1),1))
202     x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
203     x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
204     x = self.upsample(x)
205
206
207     #-----
208     #----- Finalize -----
209     #-----
210
211     #now we get x to the right dimensions
212     x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here
213
214     return x

```

```

1 #generate noisy image for Yoshi
2 #original dimensions (3,32,22)
3 #output image (3,256,176)
4 #input_depth = 32
5 #dimensions = [256,176]
6 #noiseType = "uniform"
7
8 def createNoisyImage(input_depth,dimensions,var=0.1,noiseType='uniform'):
9     tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
10    networkInput = Variable(torch.zeros(tensorShape))
11
12    if noiseType == "uniform":
13        networkInput.data.uniform_()
14    elif noiseType == "normal":
15        networkInput.data.normal_()
16
17    networkInput.data = networkInput.data * var
18
19    return networkInput
20
21 #this needs to be image size times 4!
22 superResolutionBase = createNoisyImage(32,[256,256],0.1,"uniform")
23 #inputs are: input_depth, [img.shape[1],img.shape[2]], var, noiseType
24
25 base = superResolutionBase.type(torch.cuda.FloatTensor).detach()
26 savedBase = base.data.clone()
27 baseNoise = base.data.clone()

```

```

1 def addRandomNoise(inputs):
2     inputs.data = savedBase + baseNoise.normal_()*1.0/30
3     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

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

⇨

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

⇨

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

```
1 net = Net()
2 net.cuda()
3
4 criterion = nn.MSELoss()
5 optimizer = optim.Adam(net.parameters(), lr = 0.01)

1 #Set up Training
2 #Loss Function = MSE
3 #Get image in Tensor form. This is the label
4 dtype = torch.cuda.FloatTensor
5
6 print('Started Training')
7 starttime = time.time();
8
9 maxepochs = 1000
10 numepochs = 0
11
12 # horizon = 5
13
14 acc = []
15 trainLoss = []
16
17 downsample = Downsample(n_planes=3, factor=4, preserve_size=True).type(dtype)
```

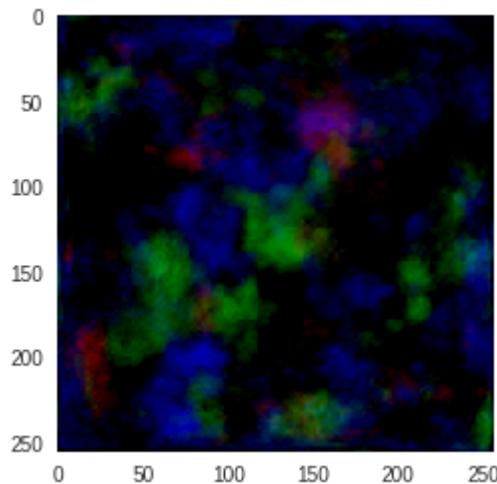
```

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 = downampler(outputs)
38     loss = criterion(downsmpledOutputs[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() - starttime), 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() - starttime))

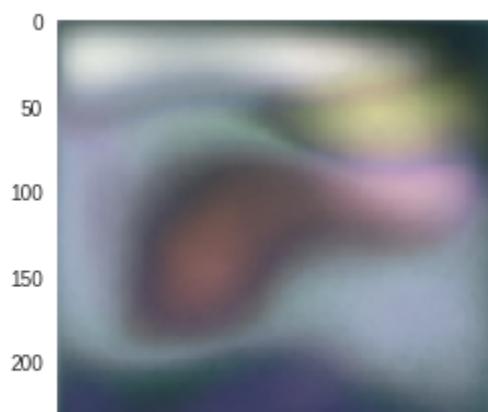
```



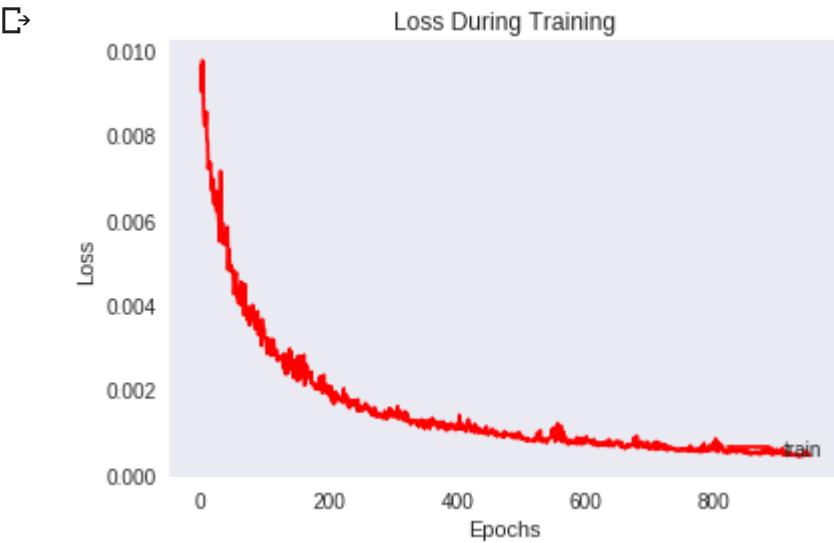
Started Training



```
[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
```



```
1 plt.figure()
2 plt.title('Loss During Training')
3 plt.xlabel('Epochs')
4 plt.ylabel('Loss')
5 #because loss is very high in the early iteration,
6 #we cut off the first 50 to make the graph meaningful
7 plt.plot(trainLoss[50:], 'r', label="train")
8 plt.legend(loc=4)
9 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>

```

1 #1AY63mgg2926Z2XtxXvzez3hxdWhU0KDb -- main folder (no useful files in there)
2 #1gNp7MLzRkLDOwgmdT-j0XLAVQ9zACR_z -- original8frames
3 #1n7iCwVNkUtMZE9TyRMXvo2bCPn82-Od1 -- 8otherframes (not used yet!), these frames don't
4 #1KqhEYNBtiQRsM-WmoWcSEdt1V5iprxFc -- contains original8frames plus processed data (mas
5
6 # Fetch the folder with all the annie hall images
7 file_list = drive.ListFile({'q': "'1gNp7MLzRkLDOwgmdT-j0XLAVQ9zACR_z' in parents and trac

```

```

1 i = 1
2 for file1 in sorted(file_list, key = lambda x: x['title']):
3     print('Downloading {} from GDrive ({}/{})'.format(file1['title'], i, len(file_list)))
4     file1.GetContentFile(file1['title'])
5     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)

```

```

1 #This function makes a mask from an image where the pixels corresponding to
2 #the subtitles in the image are 1 in the mask, and other pixels are 0
3

```

```

4 #This algorithm is general enough that it works as long as:
5   # The pixels from the subtitles are (relatively) uniform in color
6   # The range of pixels across the rest of the pictures excludes the uniform color used f
7
8 #The two parameters to set are
9   # The color of the subtitles (see comments lines 18-19 and line 22)
10  # The radius of pixels you want to mask around detected subtitle pixels (line 26)
11
12 #This is not perfect, but we were not very conservative around the subtitles but very con
13 #The dataset (this scene from Annie Hall: https://www.youtube.com/watch?v=3tch2z5qVCc) th
14 def makeMask(img):
15     out1 = np.zeros(img[:, :, 0].shape)
16     out2 = np.zeros(img[:, :, 0].shape)
17
18     #catch yellow ones (by yellow we mean our mildly arbitrary range)
19     #we used https://imagecolorpicker.com/
20     for i in range(500, img.shape[0]):
21         for j in range(img.shape[1]):
22             if img[i, j, 0] > .9 and img[i, j, 1] > .9 and img[i, j, 2] > .05 and img[i, j, 2] < .6:
23                 out1[i, j] = 1.
24
25     #make everyone within radius of yellow one yellow
26     radius = 5 #higher is more liberal with masking nonsubtitle pixels, but if it's too low
27     for i in range(radius, img.shape[0]-radius-1):
28         for j in range(radius, img.shape[1]-radius-1):
29             for k in range(-radius, radius+1):
30                 for l in range(-radius, radius+1):
31                     if out1[i+k, j+l]==1:
32                         out2[i, j] = 1.
33
34     #apply the same mask on 3 channels
35     finalOut = np.zeros(img.shape)
36     finalOut[:, :, 0] = 1-out2
37     finalOut[:, :, 1] = 1-out2
38     finalOut[:, :, 2] = 1-out2
39
40     return finalOut

```

```

1 #Do NOT run this again -- it takes a lot of time and we have the outputs saved in gdrive
2 #This code is used both for testing and saving stuff, you can comment stuff out
3
4 %matplotlib inline
5 import matplotlib.pyplot as plt
6 import matplotlib.image as mpimg
7 import numpy as np
8 import scipy.misc
9
10 for i in range(1,9):
11     print("Dealing with image " + str(i))
12
13 #show original picture
14 image = mpimg.imread('snapshot' + str(i) + '.png')
15 plt.rcParams["axes.grid"] = False
16 plt.imshow(image)
17 plt.show()
18
19 #make mask and subs/nosubs image
20 myMask = makeMask(image)
21 onlysubs = np.multiply(image, 1-myMask) #use this if you wanna see just the subs!
22 withoutsubs = np.multiply(image, myMask) #use this if you wanna see img without subs!
23
24 #show subs/nosubs image
25 plt.imshow(onlysubs)
26 plt.show()
27 plt.imshow(withoutsubs)
28 plt.show()
29
30 #check and diagnose in more depth

```

```
31 plt.imshow(withoutsubs[560:650,300:900,:])
32 plt.show()
33 plt.imshow(onlysubs[560:650,300:900,:])
34 plt.show()
35
36 #save the images
37 scipy.misc.imsave('snapshot' + str(i) + '_mask.png', myMask)
38 scipy.misc.imsave('snapshot' + str(i) + '_withoutsubs.png', withoutsubs)
39 scipy.misc.imsave('snapshot' + str(i) + '_onlysubs.png', onlysubs)
```

→

Dealing with image 1



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

⇨

```
1 for i in range(1,9):
2     file1 = drive.CreateFile()
3     file2 = drive.CreateFile()
4     file3 = drive.CreateFile()
5
6     file1.SetContentFile('snapshot' + str(i) + '_mask.png')
7     file2.SetContentFile('snapshot' + str(i) + '_withoutsubs.png')
8     file3.SetContentFile('snapshot' + str(i) + '_onlysubs.png')
9
10    file1.Upload()
11    file2.Upload()
12    file3.Upload()
```

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

⇨

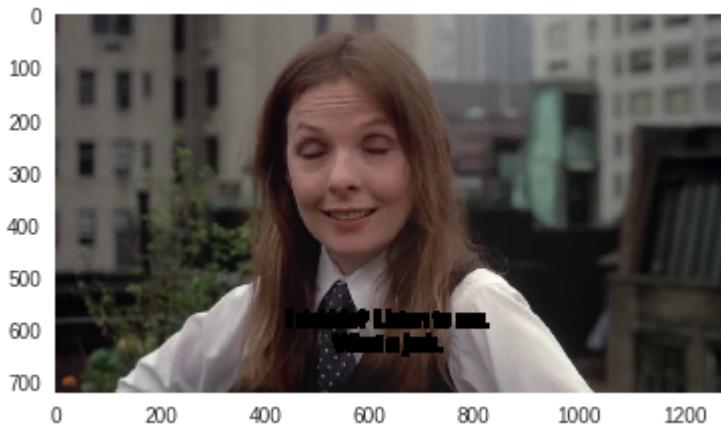
▼ Load 8frames processed data for SIDN

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

→

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 plt.rcParams["axes.grid"] = False
4 import matplotlib.image as mpimg
5 import numpy as np
6 import scipy.misc
7
8 #for loop to loop through dataset
9 #for i in range(1,9):
10 i = 1
11 original = mpimg.imread('snapshot' + str(i) + '.png')
12 mask = mpimg.imread('snapshot' + str(i) + '_mask.png')
13 withoutsubs = mpimg.imread('snapshot' + str(i) + '_withoutsubs.png')
14 onlysubs = mpimg.imread('snapshot' + str(i) + '_onlysubs.png')
15
16 #plot any of the extracted images
17 plt.imshow(withoutsubs)
18 mask.shape
```

↳ (720, 1280, 3)



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

▼ Build SIDN

```
1 from torch.autograd import Variable
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import torch.optim as optim
5
6 LeakyReLU = nn.LeakyReLU(0.2, inplace=True)
7
8 num_channels_down=[16, 32, 64, 128, 128]
9 num_channels_up=[16, 32, 64, 128, 128]
10 num_channels_skip=[0, 0, 0, 0, 4]
11 num_channels_input = 32
12 k_down = 7
13 k_up = 7
14 k_skip = 1
15
16 padding_down = int((k_down-1)/2)
17 padding_up = int((k_up-1)/2)
18 padding_skip = int((k_skip-1)/2)
19
20 class Net(nn.Module):
21     def __init__(self):
22         super(Net, self).__init__()
23
24         #-----
25         #----- Down blocks -----
26         #-----
27
28         i = 0 # for first
29         #first down block, others are copypaste
30         self.down1_downsample = nn.Conv2d(num_channels_input, num_channels_down[i], k_dow
31         self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
32         self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
33         self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
34
35         i = 1
36         self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
37         self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
```

```

38     self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
39     self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
40
41     i = 2
42     self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
43     self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
44     self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
45     self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
46
47     i = 3
48     self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
49     self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
50     self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
51     self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
52
53     i = 4
54     self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
55     self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
56     self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
57     self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
58
59     #-----
60     #----- Skip blocks -----
61     #-----
62
63     i = 4 # for 5th
64     #first skip block, others are copypaste
65     self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
66     self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])
67
68     #-----
69     #----- Misc but needed -----
70     #-----
71
72     #will need this for the up layers
73     self.upsample = nn.Upsample(scale_factor=2, mode='nearest') #at the end
74     num_channels_up.append(0) #this way for the fifth layers the combined is just the
75     #will need this at the end of network
76     self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)
77
78     #-----
79     #----- Up blocks -----
80     #-----
81
82     i=4 # for fifth
83     #fifth up block, others are copypaste
84     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
85     self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
86     self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
87     self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
88     self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
89     self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])
90
91     i=3
92     self.up4_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
93     self.up4_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
94     self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
95     self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
96     self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])
97
98     i=2
99     self.up3_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
100    self.up3_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
101    self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])
102    self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
103    self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])
104
105    i=1;
106    self.up2_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
107    self.up2_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride

```

```

108     self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
109     self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
110     self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])
111
112     i=0
113     self.up1_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
114     self.up1_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
115     self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
116     self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
117     self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])
118
119 def forward(self, x):
120     #-----
121     #----- Downsampling and recording skip -----
122     #-----
123     x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
124     x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))
125
126     x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
127     x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))
128
129     x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
130     x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))
131
132     x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
133     x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))
134
135     x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
136     x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
137     x = self.skip5_conv(x)
138     skips5 = LeakyReLU(self.skip5_bn(x))
139
140     #-----
141     #----- Upsampling -----
142     #-----
143
144     #we start at the fifth layer (layers have reverse order)
145     x = self.up5_bn_initial(skips5) #notice no concatenation here
146     x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
147     x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
148     x = self.upsample(x)
149
150     #now we upsample all the other layers
151     #print("up4")
152     #print(x.size())
153     #print(skips4.size())
154     x = self.up4_bn_initial(x)
155     x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
156     x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
157     x = self.upsample(x)
158
159     #print("up3")
160     #print(x.size())
161     #print(skips3.size())
162     x = self.up3_bn_initial(x)
163     x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
164     x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
165     x = self.upsample(x)
166
167     #print("up2")
168     #print(x.size())
169     #print(skips2.size())
170     x = self.up2_bn_initial(x)
171     x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
172     x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
173     x = self.upsample(x)
174
175     #print("up1")
176     #print(x.size())
177     #print(skips1.size())

```

```

178     x = self.up1_bn_initial(x)
179     x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
180     x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
181     x = self.upsample(x)
182
183
184     #-----
185     #----- Finalize -----
186     #-----
187
188     #now we get x to the right dimensions
189     x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here
190
191     return x

```

```

1 #generate noisy image as input
2 #noiseType = "uniform"
3
4 def createNoisyImage(input_depth,dimensions,var=0.1,noiseType='uniform'):
5     tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
6     networkInput = Variable(torch.zeros(tensorShape))
7
8     if noiseType == "uniform":
9         networkInput.data.uniform_()
10    elif noiseType == "normal":
11        networkInput.data.normal_()
12
13    networkInput.data = networkInput.data * var
14
15    return networkInput
16
17 #this needs to be image size times 4!
18 superResolutionBase = createNoisyImage(32,[maskCropped.shape[0],maskCropped.shape[1]],0.1
19 #inputs are: input_depth, [img.shape[1],img.shape[2]], var, noiseType
20
21 base = superResolutionBase.type(torch.cuda.FloatTensor).detach()
22 savedBase = base.data.clone()
23 baseNoise = base.data.clone()

```

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

⇨ 704

```
1 def addRandomNoise(inputs): #we don't add noise this time!
2     inputs.data = savedBase #+ baseNoise.normal_()*1.0/30
3
4 return inputs
```

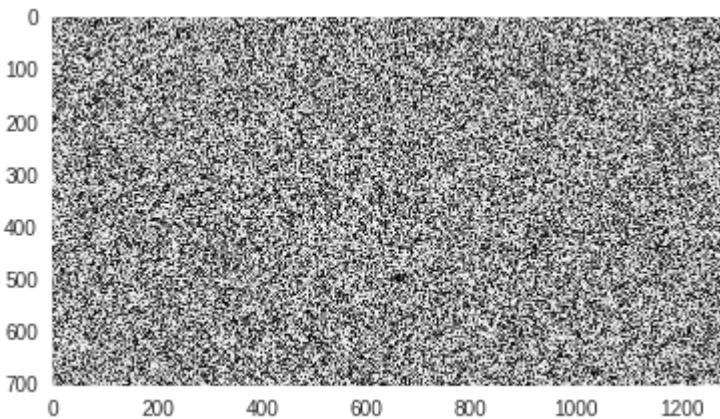
▼ Train SIDN

```

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

```

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



```
1 #size is [1, channels, height, width
2 maskedTarget = Variable(torch.Tensor(withoutsubsCropped))
3 maskCroppedTensor = Variable(torch.Tensor(maskCropped))
4
5 maskedTarget = maskedTarget.cuda()
6 maskCroppedTensor = maskCroppedTensor.cuda()
```

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

```
1 # Saving any model
2 def save_model(model, filename):
3     torch.save(model, filename)
4
5 def load_model(filename):
6     model = torch.load(filename)
7     model.cuda()
8     return model
```

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

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

```
1 net = Net()
2 net.cuda()
3
4 criterion = nn.MSELoss()
5 optimizer = optim.Adam(net.parameters(), lr = 0.01)
```

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

```

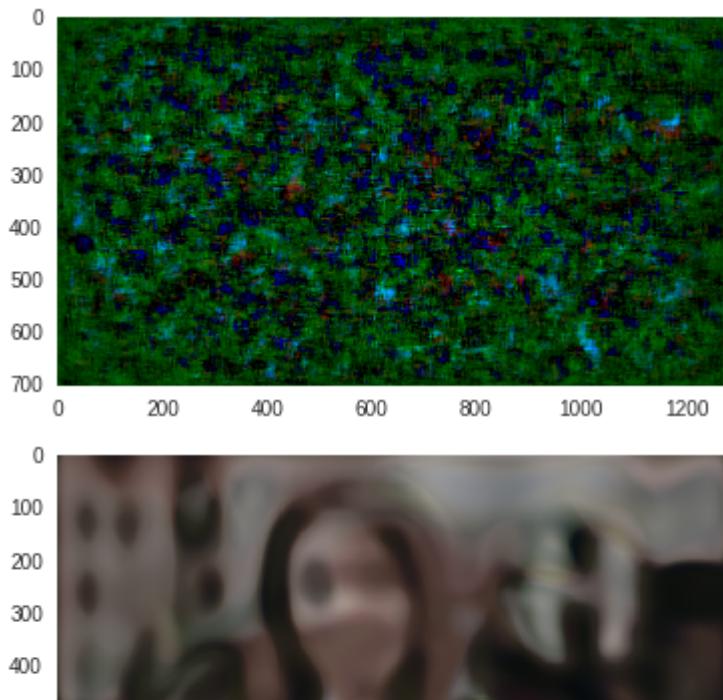
8
9     maskedTarget = maskedTarget.cuda()
10    maskCroppedTensor = maskCroppedTensor.cuda()
11
12    starttime = int(time.time())
13    trainLoss = []
14
15    for epoch in range(maxepochs): # loop over the data multiple times
16
17        # get the input
18        inputs = superResolutionBase #this is the z = uniform noise input
19
20        # get the label
21        label = maskedTarget
22
23        # wrap them in Variable
24        inputs, label = inputs.cuda(), label.cuda()
25
26        # zero the parameter gradients
27        optimizer.zero_grad()
28
29        inputs = addRandomNoise(inputs)
30
31        #print("backpropagation time")
32        # forward + backward + optimize
33        outputs = net(inputs)
34        outputs=outputs[0]
35        outputs = outputs.permute(1,2,0)
36        loss = criterion(torch.mul(outputs, maskCroppedTensor), label)
37        loss.backward()
38        optimizer.step()
39
40        #if epoch%5==0:
41            #print("We are at epoch " + str(epoch))
42
43        #save the image every 100 epochs
44        if epoch%100==0:
45            print(epoch)
46            pp = PdfPages('subsTest' + str(epoch) + '.pdf')
47
48            arr = np.array(outputs.data)
49            arr = np.clip(arr, 0, 1)
50            plt.rcParams["axes.grid"] = False
51            plt.imshow(arr)
52            #plt.savefig(pp, format='pdf')
53            #pp.close()
54            plt.imsave(fname + 'subsTest' + str(epoch) + '.png', arr)
55            file1 = drive.CreateFile()
56            file1.SetContentFile(fname + 'subsTest' + str(epoch) + '.png')
57            file1.Upload() # Upload the file.
58            plt.show()
59
60        #print some results
61        if epoch == 0:
62            k = -1
63        else:
64            k = -2
65        print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - starttime), loss.data[0])
66        trainLoss.append(loss.data[0])
67
68    print("Final train loss")
69    print(trainLoss[len(trainLoss)-1])
70
71    print('Finished Training. See training time on next line')
72    print(int(time.time() - starttime))
73    save_model(net, fname+'trained.pt')
74    file1 = drive.CreateFile()
75    file1.SetContentFile(fname+'trained.pt')
76    file1.Upload() # Upload the file.
77    return net

```

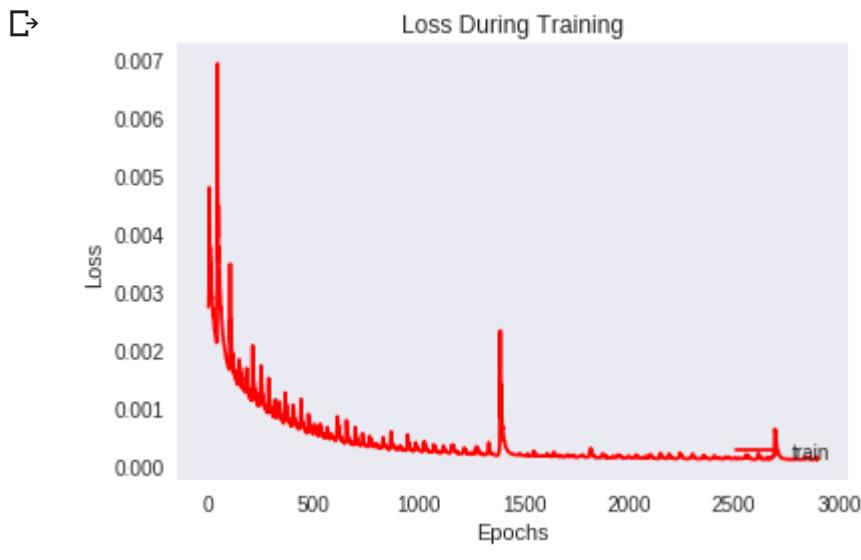
```
1 #Set up Training
2 #Loss Function = MSE
3 #Get image in Tensor form. This is the label
4 dtype = torch.cuda.FloatTensor
5
6 print('Started Training')
7 starttime = time.time();
8
9 maxepochs = 3000
10 numepochs = 0
11
12 # horizon = 5
13
14 acc = []
15 trainLoss = []
16 criterion = nn.MSELoss()
17 optimizer = optim.Adam(net.parameters(), lr = 0.01)
18 i=1
19 mask_fname = 'snapshot' + str(i) + '_mask.png'
20 withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'
21 train_model(net, mask_fname, withoutsubs_fname, criterion, optimizer, maxepochs=3000, fna
```



Started Training



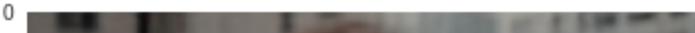
```
1 plt.figure()
2 plt.title('Loss During Training')
3 plt.xlabel('Epochs')
4 plt.ylabel('Loss')
5 plt.plot(trainLoss[100:], 'r', label="train")
6 plt.legend(loc=4)
7 plt.show()
```



```
1 # Transfer Learning
2 model = load_model('first_traintrained.pt')
3 acc = []
4 trainLoss = []
5
6 criterion = nn.MSELoss()
7 optimizer = optim.Adam(net.parameters(), lr = 0.01)
8 i=3
9 mask_fname = 'snapshot' + str(i) + '_mask.png'
10 withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'
11 model_after_transfer = train_model(net, mask_fname, withoutsubs_fname, criterion, optimiz
```

□
→

0



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





‣ Creating and Saving Masks for Report

↳ 3 cells hidden



▼ Build Autoencoder SIDN

```

1  from torch.autograd import Variable
2  import torch.nn as nn
3  import torch.nn.functional as F
4  import torch.optim as optim
5
6  LeakyReLU = nn.LeakyReLU(0.2, inplace=True)
7
8  num_channels_down=[16, 32, 64, 128, 128]
9  num_channels_up=[16, 32, 64, 128, 128]
10 num_channels_skip=[0, 0, 0, 0, 4]
11 num_channels_input = 32
12 num_train_images = 4
13 k_down = 7
14 k_up = 7
15 k_skip = 1
16
17 padding_down = int((k_down-1)/2)
18 padding_up = int((k_up-1)/2)
19 padding_skip = int((k_skip-1)/2)
20
21 class Net(nn.Module):
22     def __init__(self):
23         super(Net, self).__init__()
24
25         #-----
26         #----- Down blocks -----
27         #-----
28
29         i = 0 # for first
30         #first down block, others are copypaste
31         self.down1_downsample = nn.Conv2d(num_channels_input+num_train_images, num_channe
32         self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
33         self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
34         self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
35
36         i = 1
37         self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k_
38         self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])

```

```

39     self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
40     self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
41
42     i = 2
43     self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
44     self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
45     self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
46     self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
47
48     i = 3
49     self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
50     self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
51     self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
52     self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
53
54     i = 4
55     self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
56     self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
57     self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
58     self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
59
60     #-----
61     #----- Skip blocks -----
62     #-----
63
64     i = 4 # for 5th
65     #first skip block, others are copypaste
66     self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip, s
67     self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])
68
69     #-----
70     #----- Misc but needed -----
71     #-----
72
73     #will need this for the up layers
74     self.upsample = nn.Upsample(scale_factor=2, mode='nearest') #at the end
75     num_channels_up.append(0) #this way for the fifth layers the combined is just the
76     #will need this at the end of network
77     self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)
78
79     #-----
80     #----- Up blocks -----
81     #-----
82
83     i=4 # for fifth
84     #fifth up block, others are copypaste
85     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
86     self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
87     self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
88     self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
89     self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
90     self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])
91
92     i=3
93     self.up4_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
94     self.up4_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
95     self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
96     self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
97     self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])
98
99     i=2
100    self.up3_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
101    self.up3_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
102    self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])
103    self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
104    self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])
105
106    i=1;
107    self.up2_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
108    self.up2_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride

```

```

109     self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
110     self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
111     self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])
112
113     i=0
114     self.up1_bn_initial = nn.BatchNorm2d(num_channels_up[i+1])
115     self.up1_conv1 = nn.Conv2d(num_channels_up[i+1], num_channels_up[i], k_up, stride
116     self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
117     self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
118     self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])
119
120 def forward(self, x):
121     #-----
122     #----- Downsampling and recording skip -----
123     #-----
124     x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
125     x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))
126
127     x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
128     x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))
129
130     x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
131     x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))
132
133     x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
134     x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))
135
136     x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
137     x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
138     x = self.skip5_conv(x)
139     skips5 = LeakyReLU(self.skip5_bn(x))
140
141     #-----
142     #----- Upsampling -----
143     #-----
144
145     #we start at the fifth layer (layers have reverse order)
146     x = self.up5_bn_initial(skips5) #notice no concatenation here
147     x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
148     x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
149     x = self.upsample(x)
150
151     #now we upsample all the other layers
152     #print("up4")
153     #print(x.size())
154     #print(skips4.size())
155     x = self.up4_bn_initial(x)
156     x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
157     x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
158     x = self.upsample(x)
159
160     #print("up3")
161     #print(x.size())
162     #print(skips3.size())
163     x = self.up3_bn_initial(x)
164     x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
165     x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
166     x = self.upsample(x)
167
168     #print("up2")
169     #print(x.size())
170     #print(skips2.size())
171     x = self.up2_bn_initial(x)
172     x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
173     x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
174     x = self.upsample(x)
175
176     #print("up1")
177     #print(x.size())
178     #print(skips1.size())

```

```

179     x = self.up1_bn_initial(x)
180     x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
181     x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
182     x = self.upsample(x)
183
184
185     #-----
186     #----- Finalize -----
187     #-----
188
189     #now we get x to the right dimensions
190     x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here
191
192     return x

```

```

1 #generate noisy image as input
2 #noiseType = "uniform"
3 from torch.autograd import Variable
4
5 def createNoisyImage(input_depth,dimensions,var=0.1,noiseType='uniform'):
6     tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
7     networkInput = Variable(torch.zeros(tensorShape))
8
9     if noiseType == "uniform":
10         networkInput.data.uniform_()
11     elif noiseType == "normal":
12         networkInput.data.normal_()
13
14     networkInput.data = networkInput.data * var
15
16     return networkInput
17
18 def createNoisyImagesWithOneHot(numImages, input_depth,dimensions,var=0.1,noiseType='unif
19 mynoisy = createNoisyImage(input_depth,dimensions,var,noiseType)
20
21 tensorShape = [numImages,input_depth+numImages,dimensions[0],dimensions[1]]
22 fourDinput = Variable(torch.zeros(tensorShape))
23
24 for i in range(0, numImages):
25     ithoutput = mynoisy
26     ithoutput = padit(mynoisy, numImages, i)
27     fourDinput[i] = ithoutput[0]
28
29 return fourDinput
30
31 def padit(originalImage, numImages, padIndex):
32     outImage = originalImage
33
34     oneImageShape = [1,1,originalImage.shape[2],originalImage.shape[3]]
35     myzeros = Variable(torch.zeros(oneImageShape))
36     myones = Variable(torch.ones(oneImageShape))
37
38     for j in range(0,numImages):
39         if j==padIndex:
40             outImage = torch.cat((outImage, myones),1)
41         else:
42             outImage = torch.cat((outImage, myzeros),1)
43     return outImage
44
45 oneHotInput = createNoisyImagesWithOneHot(4, 32,[maskCropped.shape[0],maskCropped.shape[1

```

```

1 #Checking that we have the right 0 and 1 images at the right places
2 print(oneHotInput.shape)
3
4 samInput = oneHotInput.data[2,:,:,:]
5 print(samInput[34])

```

```
6 #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]
```

```
1 #size is [1, channels, height, width  
2 maskedTarget = Variable(torch.Tensor(withoutsubsCropped))  
3 maskCroppedTensor = Variable(torch.Tensor(maskCropped))  
4  
5 maskedTarget = maskedTarget.cuda()  
6 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

```
1 # Saving any model  
2 def save_model(model, filename):  
3     torch.save(model, filename)  
4  
5 def save_model_statedict(model, filename):  
6     torch.save(model.state_dict(), filename)  
7  
8 def load_model(filename):  
9     model = torch.load(filename)  
10    model.cuda()  
11    return model
```

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

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

```
1 #reset your net  
2 net = Net()  
3 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)
    )

```

```

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

```

```
6     file1.GetContentFile(file1['title'])
7     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)
```

```
1 her1 = mpimg.imread('her1.png')[8:712,:,:]
2 her2 = mpimg.imread('her2.png')[8:712,:,:]
3 her3 = mpimg.imread('her3.png')[8:712,:,:]
4 her4 = mpimg.imread('her4.png')[8:712,:,:]
5 images = [her1, her2, her3, her4]
6 images = np.array(images)
7 multiTarget = Variable(torch.Tensor(images))
8
9 fname = "multiTraining3k"
10
11 torch.save(oneHotInput, fname+'oneHotVariable.pt')
12 file1 = drive.CreateFile()
13 file1.SetContentFile(fname+'oneHotVariable.pt')
14 file1.Upload() # Upload the file.
```

```
1 #changing the epochs to 1000 for the sake of this file
2 def train_model_multiImage(net, oneHotInput, target, numImages, criterion, optimizer, max
3     target = target.cuda()
4     oneHotInput = oneHotInput.cuda()
5
6     for epoch in range(maxepochs): # loop over the data multiple times
7         # zero the parameter gradients
8         optimizer.zero_grad()
9
10        #print("backpropagation time")
11        # forward + backward + optimize
12        #THIS SHOULD BE A LOOP
13        loss = 0
14        #for i in range(0, numImages):
15        outputs = net(oneHotInput)
16        #outputs = outputs[0]
17        outputs = outputs.permute(0,2,3,1)
18        loss = criterion(outputs, target)
19        loss.backward()
20
21        optimizer.step()
22
23        #print some results
24        if epoch == 0:
25            k = -1
26        else:
27            k = -2
28        print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - starttime), loss.data[0])
29        trainLoss.append(loss.data[0])
30
31    print("Final train loss")
32    print(trainLoss[len(trainLoss)-1])
33
34    print('Finished Training. See training time on next line')
35    print(int(time.time() - starttime))
36    save_model(net, fname+'trained.pt')
```

```

37 file1 = drive.CreateFile()
38 file1.SetContentFile(fname+'trained.pt')
39 file1.Upload() # Upload the file.
40 return net

1 #changing the epochs to 1000 for the sake of this file
2 def train_model_loadMultiModel(net, numImages, inputs, mask_fname, withoutsubs_fname, cri
3 mask = mpimg.imread(mask_fname)
4 withoutsubs = mpimg.imread(withoutsbs_fname)
5 withoutsubsCropped = withoutsubs[8:712,:,:]
6 maskCropped = mask[8:712,:,:]
7 maskedTarget = Variable(torch.Tensor(withoutsbsCropped))
8 maskCroppedTensor = Variable(torch.Tensor(maskCropped))
9
10 maskedTarget = maskedTarget.cuda()
11 maskCroppedTensor = maskCroppedTensor.cuda()
12 inputs = inputs.cuda()
13
14 for epoch in range(maxepochs): # loop over the data multiple times
15     # get the label #NEEDS TO BE 4D TENSOR
16     label = maskedTarget
17
18     # wrap them in Variable
19     inputs, label = inputs.cuda(), label.cuda()
20
21     # zero the parameter gradients
22     optimizer.zero_grad()
23
24     #print("backpropagation time")
25     # forward + backward + optimize
26     #THIS SHOULD BE A LOOP
27     loss = 0
28     outputs = net(inputs)
29     outputs = outputs[0]
30     outputs = outputs.permute(1,2,0)
31     loss = criterion(torch.mul(outputs, maskCroppedTensor), label)
32     loss.backward()
33
34     optimizer.step()
35
36     #if epoch%5==0:
37         #print("We are at epoch " + str(epoch))
38
39     #save the image every 100 epochs
40     if epoch%50==0:
41         pp = PdfPages('subsTest' + str(epoch) + '.pdf')
42
43         arr = np.array(outputs.data)
44         arr = np.clip(arr, 0, 1)
45         plt.rcParams["axes.grid"] = False
46         plt.imshow(arr)
47         plt.savefig(pp, format='pdf')
48         pp.close()
49         plt.imsave(fname + 'subsTest' + str(epoch) + '.png', arr)
50         file1 = drive.CreateFile()
51         file1.SetContentFile(fname + 'subsTest' + str(epoch) + '.png')
52         file1.Upload() # Upload the file.
53         plt.show()
54
55     #print some results
56     if epoch == 0:
57         k = -1
58     else:
59         k = -2
60     print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - starttime), loss.data[0])
61     trainLoss.append(loss.data[0])
62
63 print("Final train loss")

```

```
64 print(trainLoss[len(trainLoss)-1])
65
66 print('Finished Training. See training time on next line')
67 print(int(time.time() - starttime))
68 save_model(net, fname+'trained.pt')
69 file1 = drive.CreateFile()
70 file1.SetContentFile(fname+'trained.pt')
71 file1.Upload() # Upload the file.
72 return net

1 #Set up Training
2 #Loss Function = MSE
3 #Get image in Tensor form. This is the label
4 dtype = torch.cuda.FloatTensor
5
6 print('Started Training')
7 starttime = time.time();
8
9 numepochs = 0
10
11 acc = []
12 trainLoss = []
13 criterion = nn.MSELoss()
14 optimizer = optim.Adam(net.parameters(), lr = 0.01)
15
16 train_model_multiImage(net, oneHotInput, multiTarget, 4, criterion, optimizer, maxepochs=
```

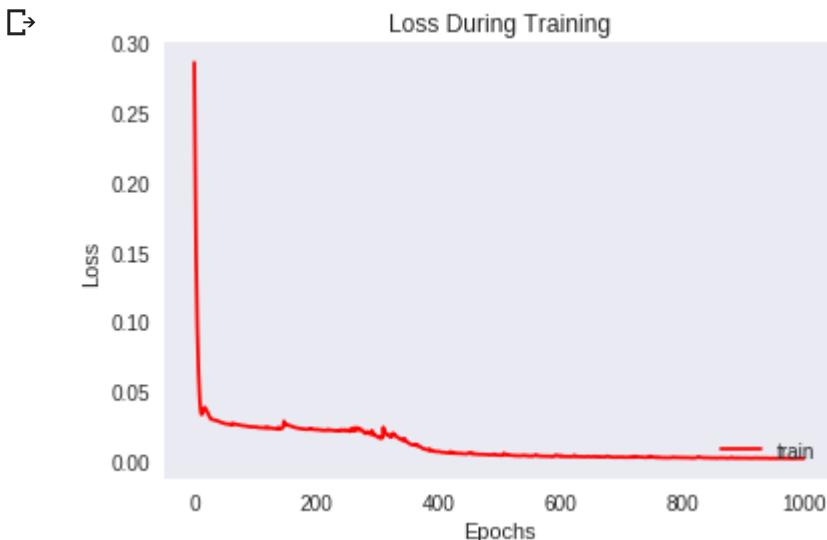


```
[580, 857 s] loss: 0.003
[581, 858 s] loss: 0.003
[582, 859 s] loss: 0.003
[583, 861 s] loss: 0.003
[584, 862 s] loss: 0.003
[585, 864 s] loss: 0.003
[586, 865 s] loss: 0.003
[587, 867 s] loss: 0.003
[588, 868 s] loss: 0.003
[589, 870 s] loss: 0.003
[590, 871 s] loss: 0.003
[591, 873 s] loss: 0.003
[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
```

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

```
161 002 s1 loss: 0.003
```

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



```
1 ~~~, ~~~ ~, ~~~. ~~~~
```

```
1 oneHotInput = createNoisyImagesWithOneHot(4, 36,[maskCropped.shape[0],maskCropped.shape[1]]  
2  
1 #initial training (not shown).  
2 model = Net()  
3 model.cuda()  
4 criterion = nn.MSELoss()  
5 optimizer = optim.Adam(net.parameters(), lr = 0.01)  
6 i=3  
7 mask_fname = 'snapshot' + str(i) + '_mask.png'  
8 withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'  
9 model_without_transfer = train_model(net, mask_fname, withoutsubs_fname, criterion, optim
```

```
1 #main training with transfer  
2  
3 def makeInputsWithZeroOneHot(onehotinput, numImages):  
4     #assumes 32 channels in noisy input  
5     noisyImage = onehotinput[0:1,0:32,:,:]  
6     outImage = padit(noisyImage, numImages, -1)  
7     return outImage  
8  
9 zeroOneHotInput = makeInputsWithZeroOneHot(oneHotInput, 4)  
10  
11 # Try multi  
12 #model = load_model('multiTraintrained.pt')  
13 acc = []  
14 trainLoss = []  
15  
16 criterion = nn.MSELoss()  
17 optimizer = optim.Adam(net.parameters(), lr = 0.01)  
18 i=3  
19 mask_fname = 'snapshot' + str(i) + '_mask.png'  
20 withoutsubs_fname = 'snapshot' + str(i) + '_withoutsubs.png'  
21 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/python2.6/dist-packages/torch/compilation.py:152: UserWarning: Couldn't

```

▼ General Inpainting (not text inpaining)

```
copyfrom input to /home/duygu/PycharmProjects/DeepImagePrior/
```

```

1 import requests
2 import shutil
3 path = 'library.png'
4 r = requests.get('https://github.com/DmitryUlyanov/deep-image-prior/raw/master/data/inpai
5 if r.status_code == 200:
6     with open(path, 'wb') as f:
7         r.raw.decode_content = True
8         shutil.copyfileobj(r.raw, f)
9 path = 'library_mask.png'
10 r = requests.get('https://github.com/DmitryUlyanov/deep-image-prior/raw/master/data/inpai
11 if r.status_code == 200:
12     with open(path, 'wb') as f:
13         r.raw.decode_content = True
14         shutil.copyfileobj(r.raw, f)

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

```

```

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

```



```

1 from torch.autograd import Variable
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import torch.optim as optim
5
6 LeakyReLU = nn.LeakyReLU(0.2, inplace=True)
7
8 num_channels_down=[16, 32, 64, 128, 128, 128]
9 num_channels_up=[16, 32, 64, 128, 128, 128]
10 num_channels_skip=[0, 0, 0, 0, 0, 0]
11 num_channels_input = 1
12 k_down = 5
13 k_up = 3
14
15 padding_down = int((k_down-1)/2)
16 padding_up = int((k_up-1)/2)
17 # padding_skip = int((k_skip-1)/2)
18
19 class Net(nn.Module):
20     def __init__(self):
21         super(Net, self).__init__()
22
23         #-----
24         #----- Down blocks -----
25         #-----
26
27         i = 0 # for first
28         #first down block, others are copypaste
29         self.down1_downsample = nn.Conv2d(num_channels_input, num_channels_down[i], k_down)
30         self.down1_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
31         self.down1_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_down)
32         self.down1_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
33
34         i = 1
35         self.down2_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k_down)
36         self.down2_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
37         self.down2_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_down)
38         self.down2_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
39
40         i = 2

```

```

41     self.down3_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
42     self.down3_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
43     self.down3_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
44     self.down3_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
45
46     i = 3
47     self.down4_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
48     self.down4_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
49     self.down4_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
50     self.down4_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
51
52     i = 4
53     self.down5_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
54     self.down5_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
55     self.down5_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
56     self.down5_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
57
58     i = 5
59     self.down6_downsample = nn.Conv2d(num_channels_down[i-1], num_channels_down[i], k
60     self.down6_bn_downsample = nn.BatchNorm2d(num_channels_down[i])
61     self.down6_justconv = nn.Conv2d(num_channels_down[i], num_channels_down[i], k_dow
62     self.down6_bn_justconv = nn.BatchNorm2d(num_channels_down[i])
63
64     #----- Skip blocks -----
65
66
67 #     i = 0 # for first
68 #     #first skip block, others are copypaste
69 #     self.skip1_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
70 #     self.skip1_bn = nn.BatchNorm2d(num_channels_skip[i])
71
72 #     i = 1
73 #     self.skip2_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
74 #     self.skip2_bn = nn.BatchNorm2d(num_channels_skip[i])
75
76 #     i = 2
77 #     self.skip3_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
78 #     self.skip3_bn = nn.BatchNorm2d(num_channels_skip[i])
79
80 #     i = 3
81 #     self.skip4_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
82 #     self.skip4_bn = nn.BatchNorm2d(num_channels_skip[i])
83
84 #     i = 4
85 #     self.skip5_conv = nn.Conv2d(num_channels_down[i], num_channels_skip[i], k_skip,
86 #     self.skip5_bn = nn.BatchNorm2d(num_channels_skip[i])
87
88     #-----
89     #----- Misc but needed -----
90
91
92     #will need this for the up layers
93     self.upsample = nn.Upsample(scale_factor=2, mode='nearest') #at the end
94     num_channels_up.append(num_channels_down[i]) #this way for the fifth layers the c
95     #will need this at the end of network
96     self.makeFinalOutput = nn.Conv2d(num_channels_up[0],3,1)
97
98     #-----
99     #----- Up blocks -----
100
101
102     i=5 # for fifth
103     #fifth up block, others are copypaste
104     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
105     self.up6_bn_initial = nn.BatchNorm2d(combinedNumChannels)
106     self.up6_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
107     self.up6_bn1 = nn.BatchNorm2d(num_channels_up[i])
108     self.up6_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
109     self.up6_bn2 = nn.BatchNorm2d(num_channels_up[i])
110

```

```

111     i=4 # for fifth
112     #fifth up block, others are copypaste
113     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i]; #num_channels_
114     self.up5_bn_initial = nn.BatchNorm2d(combinedNumChannels)
115     self.up5_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
116     self.up5_bn1 = nn.BatchNorm2d(num_channels_up[i])
117     self.up5_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
118     self.up5_bn2 = nn.BatchNorm2d(num_channels_up[i])
119
120     i=3
121     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
122     self.up4_bn_initial = nn.BatchNorm2d(combinedNumChannels)
123     self.up4_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
124     self.up4_bn1 = nn.BatchNorm2d(num_channels_up[i])
125     self.up4_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
126     self.up4_bn2 = nn.BatchNorm2d(num_channels_up[i])
127
128     i=2
129     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
130     self.up3_bn_initial = nn.BatchNorm2d(combinedNumChannels)
131     self.up3_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
132     self.up3_bn1 = nn.BatchNorm2d(num_channels_up[i])
133     self.up3_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
134     self.up3_bn2 = nn.BatchNorm2d(num_channels_up[i])
135
136     i=1
137     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
138     self.up2_bn_initial = nn.BatchNorm2d(combinedNumChannels)
139     self.up2_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
140     self.up2_bn1 = nn.BatchNorm2d(num_channels_up[i])
141     self.up2_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
142     self.up2_bn2 = nn.BatchNorm2d(num_channels_up[i])
143
144     i=0
145     combinedNumChannels = num_channels_up[i+1] + num_channels_skip[i];
146     self.up1_bn_initial = nn.BatchNorm2d(combinedNumChannels)
147     self.up1_conv1 = nn.Conv2d(combinedNumChannels, num_channels_up[i], k_up, stride=
148     self.up1_bn1 = nn.BatchNorm2d(num_channels_up[i])
149     self.up1_conv2 = nn.Conv2d(num_channels_up[i], num_channels_up[i], 1, stride=1, p
150     self.up1_bn2 = nn.BatchNorm2d(num_channels_up[i])
151
152     def forward(self, x):
153         #-----
154         #----- Downsampling and recording skip -----
155         #-----
156         x = LeakyReLU(self.down1_bn_downsample(self.down1_downsample(x)))
157         x = LeakyReLU(self.down1_bn_justconv(self.down1_justconv(x)))
158         #    skips1 = LeakyReLU(self.skip1_bn(self.skip1_conv(x)))
159
160         x = LeakyReLU(self.down2_bn_downsample(self.down2_downsample(x)))
161         x = LeakyReLU(self.down2_bn_justconv(self.down2_justconv(x)))
162         #    skips2 = LeakyReLU(self.skip2_bn(self.skip2_conv(x)))
163
164         x = LeakyReLU(self.down3_bn_downsample(self.down3_downsample(x)))
165         x = LeakyReLU(self.down3_bn_justconv(self.down3_justconv(x)))
166         #    skips3 = LeakyReLU(self.skip3_bn(self.skip3_conv(x)))
167
168         x = LeakyReLU(self.down4_bn_downsample(self.down4_downsample(x)))
169         x = LeakyReLU(self.down4_bn_justconv(self.down4_justconv(x)))
170         #    skips4 = LeakyReLU(self.skip4_bn(self.skip4_conv(x)))
171
172         x = LeakyReLU(self.down5_bn_downsample(self.down5_downsample(x)))
173         x = LeakyReLU(self.down5_bn_justconv(self.down5_justconv(x)))
174         #    skips5 = LeakyReLU(self.skip5_bn(self.skip5_conv(x)))
175
176         x = LeakyReLU(self.down6_bn_downsample(self.down6_downsample(x)))
177         x = LeakyReLU(self.down6_bn_justconv(self.down6_justconv(x)))
178
179         #-----
180         #----- Upsampling -----

```

```

181 #-----
182
183     x = self.up6_bn_initial(x) #notice no concatenation here
184     x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
185     x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
186     x = self.upsample(x)
187
188     #we start at the fifth layer (layers have reverse order)
189     x = self.up5_bn_initial(x) #notice no concatenation here
190     x = LeakyReLU(self.up5_bn1(self.up5_conv1(x)))
191     x = LeakyReLU(self.up5_bn2(self.up5_conv2(x)))
192     x = self.upsample(x)
193
194     #now we upsample all the other layers
195     #print("up4")
196     #print(x.size())
197     #print(skips4.size())
198     x = self.up4_bn_initial(x)
199     x = LeakyReLU(self.up4_bn1(self.up4_conv1(x)))
200     x = LeakyReLU(self.up4_bn2(self.up4_conv2(x)))
201     x = self.upsample(x)
202
203     #print("up3")
204     #print(x.size())
205     #print(skips3.size())
206     x = self.up3_bn_initial(x)
207     x = LeakyReLU(self.up3_bn1(self.up3_conv1(x)))
208     x = LeakyReLU(self.up3_bn2(self.up3_conv2(x)))
209     x = self.upsample(x)
210
211     #print("up2")
212     #print(x.size())
213     #print(skips2.size())
214     x = self.up2_bn_initial(x)
215     x = LeakyReLU(self.up2_bn1(self.up2_conv1(x)))
216     x = LeakyReLU(self.up2_bn2(self.up2_conv2(x)))
217     x = self.upsample(x)
218
219     #print("up1")
220     #print(x.size())
221     #print(skips1.size())
222     x = self.up1_bn_initial(x)
223     x = LeakyReLU(self.up1_bn1(self.up1_conv1(x)))
224     x = LeakyReLU(self.up1_bn2(self.up1_conv2(x)))
225     x = self.upsample(x)
226
227
228 #-----
229 #----- Finalize -----
230 #
231
232     #now we get x to the right dimensions
233     x = self.makeFinalOutput(x) #not sure if we need the leaky ReLU here
234
235     return x

```

```

1 #Saving a random model
2 net = Net()
3 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)

```

```

1 def createNoisyImage(input_depth,dimensions,var=0.1,noisType='uniform'):
2     tensorShape = [1,input_depth,dimensions[0],dimensions[1]]
3     networkInput = Variable(torch.zeros(tensorShape))
4

```

```

5   if noiseType == "uniform":
6       networkInput.data.uniform_()
7   elif noiseType == "normal":
8       networkInput.data.normal_()
9
10  networkInput.data = networkInput.data * var
11
12  return networkInput
13
14
15
16 superResolutionBase = createNoisyImage(1,[448,704],0.1,"uniform")
17 base = superResolutionBase
18 savedBase = base.data.clone().cuda()
19 baseNoise = base.data.clone().cuda()
20
21 def train_model_inpainting(net, mask_fname, image_fname, criterion, optimizer, maxepochs=100):
22     mask = mpimg.imread(mask_fname)
23     mask = mask[None, :]
24     image = mpimg.imread(image_fname)
25     imageTensor = Variable(torch.Tensor(image))
26     imageTensor = imageTensor.permute(2, 0, 1)
27     maskTensor = Variable(torch.Tensor(mask))
28     imageTensor = imageTensor.cuda()
29     maskTensor = maskTensor.cuda()
30
31     # get the input
32     inputs = superResolutionBase #this is the z = uniform noise input
33
34     for epoch in range(maxepochs): # loop over the data multiple times
35         for n in [x for x in net.parameters() if len(x.size()) == 4]:
36             n.data += n.data.clone()*n.data.std()/50
37
38         # get the label
39         label = imageTensor
40
41         # wrap them in Variable
42         inputs, label = inputs.cuda(), label.cuda()
43
44         # zero the parameter gradients
45         optimizer.zero_grad()
46
47         # inputs = addRandomNoise(inputs)
48
49         #print("backpropagation time")
50         # forward + backward + optimize
51         outputs = net(inputs)
52         outputs=outputs[0]
53         # outputs = outputs.permute(1,2,0)
54         loss = criterion(outputs * maskTensor, label * maskTensor)
55         loss.backward()
56         optimizer.step()
57
58         #if epoch%5==0:
59             #print("We are at epoch " + str(epoch))
60
61         #save the image every 100 epochs
62         if epoch%100==0:
63             pp = PdfPages('subsTest' + str(epoch) + '.pdf')
64
65             arr = np.array(outputs.data)
66             arr = arr.transpose(1,2,0)
67             arr = np.clip(arr, 0, 1)
68             plt.rcParams["axes.grid"] = False
69             plt.imshow(arr)
70             plt.savefig(pp, format='pdf')
71             pp.close()
72             plt.imsave(fname + 'subsTest' + str(epoch) + '.png', arr)
73             file1 = drive.CreateFile()
74             file1.SetContentFile(fname + 'subsTest' + str(epoch) + '.png')

```

```

75     file1.Upload() # Upload the file.
76     plt.show()
77     pass
78
79     #print some results
80     if epoch == 0:
81         k = -1
82     else:
83         k = -2
84     print('[%d, %d s] loss: %.3f '% (epoch + 1, int(time.time() - starttime), loss.data[0])
85     trainLoss.append(loss.data[0])
86
87     print("Final train loss")
88     print(trainLoss[len(trainLoss)-1])
89
90     print('Finished Training. See training time on next line')
91     print(int(time.time() - starttime))
92     save_model(net, fname+'trained.pt')
93     file1 = drive.CreateFile()
94     file1.SetContentFile(fname+'trained.pt')
95     file1.Upload() # Upload the file.
96     return net

```

```

1 #Set up Training
2 #Loss Function = MSE
3 #Get image in Tensor form. This is the label
4 dtype = torch.cuda.FloatTensor
5
6 print('Started Training')
7 starttime = time.time();
8
9 maxepochs = 3000
10 numepochs = 0
11
12 # horizon = 5
13
14 acc = []
15 trainLoss = []
16 criterion = nn.MSELoss()
17 optimizer = optim.Adam(net.parameters(), lr = 0.01)
18 i=1
19 mask_fname = 'library_mask.png'
20 image_fname = 'library.png'
21 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
14547 424 s1 loss: 0.008
```

[4548, 424 s] loss: 0.000
[4548, 424 s] loss: 0.008