UDACITY

Return to "Deep Learning" in the classroom

DISCUSS ON STUDENT HUB

Generate Faces

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Good job overall with the training part and choices you made around ideal dcgan implementation.
Also, do try to work with bigger images and other GANs for multiple of use cases ;
For instance StyleGAN has some pretty interesting outputs - > https://towardsdatascience.com/how-to-train-stylegan-to-generate-realistic-faces-d4afca48e705
Best of luck and hope you apply all that you have learnt in the nanodegree to some interesting projects at your workplace and/or at your home right now :)

Required Files and Tests

The project submission contains the project notebook, called “dlnd_face_generation.ipynb”.

All files present.

All the unit tests in project have passed.

Both tests have passed

Data Loading and Processing

The function `get_dataloader` should transform image data into resized, Tensor image types and return a DataLoader that batches all the training data into an appropriate size.

```
transform = transforms.Compose([transforms.Resize((image_size,image_size)),
                                transforms.ToTensor()])

    # get images directory
    image_path = os.path.join('./' + data_dir)

    # define dataset
    dataset = datasets.ImageFolder(image_path, transform)

    # create and return DataLoader
    data_loader = DataLoader(dataset=dataset, batch_size=batch_size, shuffle=True
)
```

Done well with the transform.resize function
Do remember that if y ou only pass one integer value to resize function, the output may not be (input,input) as dimension as pytorch's resize works differently.
https://pytorch.org/docs/stable/torchvision/transforms.html#torchvision.transforms.Resize

Pre-process the images by creating a `scale` function that scales images into a given pixel range. This function should be used later, in the training loop.

Done well.
This is needed as it is mentioned in the original DCGan paper that they normalized all pixel values between 0 and 1 and it gives much better results.
Do read the full paper if you get time it is filled with insights to train and understand the architecture.
https://arxiv.org/pdf/1511.06434.pdf

In [7]:
```
# TODO: Complete the scale function
def scale(x, feature_range=(-1, 1)):
    ''' Scale takes in an image x and returns that image, scaled
       with a feature_range of pixel values from -1 to 1.
       This function assumes that the input x is already scaled from 0-1.'''
    # assume x is scaled to (0, 1)
    # scale to feature_range and return scaled x

    return 2*x-1
```

In [8]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
# check scaled range
# should be close to -1 to 1
img = images[0]
scaled_img = scale(img)

print('Min: ', scaled_img.min())
print('Max: ', scaled_img.max())
```

```
Min:  tensor(-0.9686)
Max:  tensor(0.7255)
```

Build the Adversarial Networks

The Discriminator class is implemented correctly; it outputs one value that will determine whether an image is real or fake.

```
# define feedforward behavior
 x = F.relu(self.conv1(x))
 x = F.relu(self.conv2(x))
 x = F.relu(self.conv3(x))
 x = F.relu(self.conv4(x))
 # last conv layer without relu
 x = self.conv5(x)
 x = x.view(-1, 1)
 return x
```

Good job on using leaky relu and batchnorm as discussed by Soumith Chintala in his GanHacks guidelines as well
https://github.com/soumith/ganhacks

The Generator class is implemented correctly; it outputs an image of the same shape as the processed training data.

```
# define feedforward behavior
x = self.fc(x)
x = x.view(-1, self.conv_dim*8, 2, 2)

x = F.relu(self.t_conv1(x))
x = F.relu(self.t_conv2(x))
x = F.relu(self.t_conv3(x))
x = torch.tanh(self.t_conv4(x))
```

Batch norm and tanh at the end is just what is needed for a good implementation for this project. So good job following the guidelines to the dot.

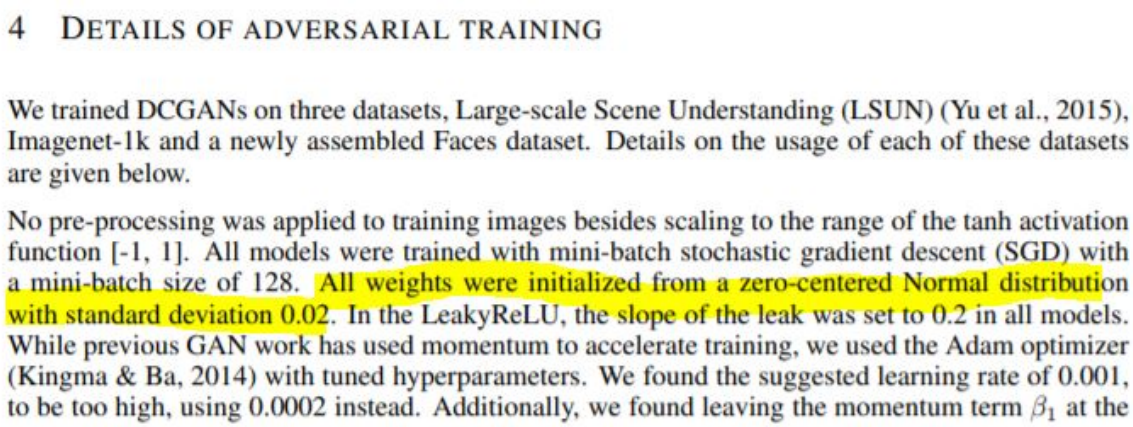
This function should initialize the weights of any convolutional or linear layer with weights taken from a normal distribution with a mean = 0 and standard deviation = 0.02.

```
    if classname.find('Linear') != -1 or classname.find('Conv') != -1 :
```

```
        m.weight.data.normal_(0, 0.02)
```

Done well. Do remember that by doing this we are initializing weights for all the conv and linear linears (including transpose conv (deconv) layers as well.

Excerpt from the paper regarding training



4 DETAILS OF ADVERSARIAL TRAINING

We trained DCGANs on three datasets, Large-scale Scene Understanding (LSUN) (Yu et al., 2015), Imagenet-1k and a newly assembled Faces dataset. Details on the usage of each of these datasets are given below.

No pre-processing was applied to training images besides scaling to the range of the tanh activation function [-1, 1]. All models were trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128. All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02. In the LeakyReLU, the slope of the leak was set to 0.2 in all models. While previous GAN work has used momentum to accelerate training, we used the Adam optimizer (Kingma & Ba, 2014) with tuned hyperparameters. We found the suggested learning rate of 0.001, to be too high, using 0.0002 instead. Additionally, we found leaving the momentum term β_1 at the

Optimization Strategy

The loss functions take in the outputs from a discriminator and return the real or fake loss.

Done well. You can also look at implementing custom loss functions like WGAN's loss which stabilizes the learning rate a much more than DCGAN.
WGAN Paper - https://arxiv.org/abs/1701.07875

There are optimizers for updating the weights of the discriminator and generator. These optimizers should have appropriate hyperparameters.

```
lr = 0.0002
beta1=0.5
beta2=0.999
```

I guess you followed this article ->https://machinelearningmastery.com/how-to-train-stable-generative-adversarial-networks/
If not, this is also a good starting point for stable training params.

Training and Results

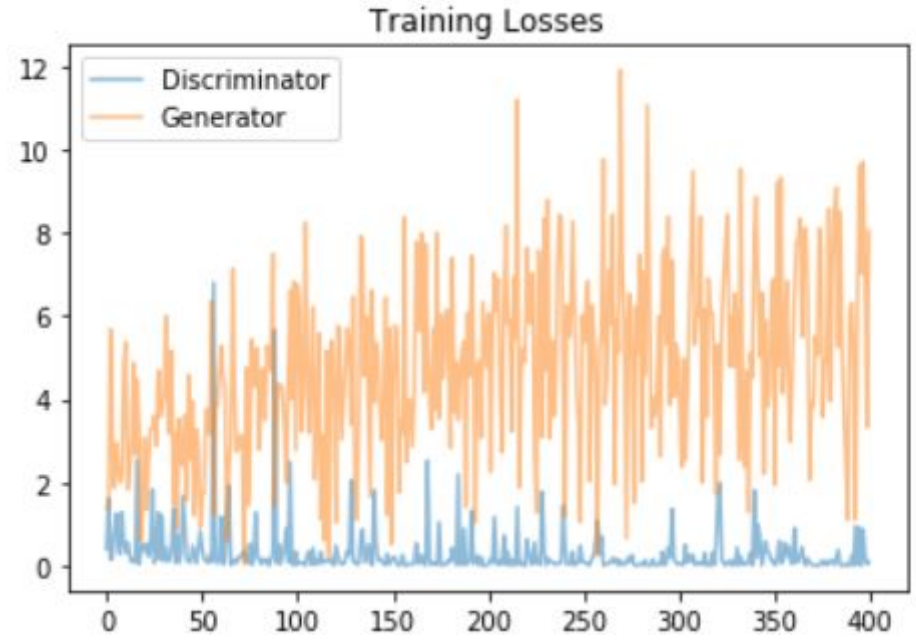
Real training images should be scaled appropriately. The training loop should alternate between training the discriminator and generator networks.

```
Epoch [ 10/ 50] | d_loss: 0.3492 | g_loss: 0.0471
```

Somewhere near 10th epoch you get the best convergence and after that the generator loss just explodes. The generator loss is a bit too high but as discrimator loss has hit the plateau, the network would be able to generate somewhat proper faces.
I'd recommend a good way to see the training is also to generate faces per epoch and display them in output to see how the model is learning to generate faces (it starts with a very noisy image, then shapes start building and it gives you an intuitive idea as to how it is converging or diverging at a particular epoch.)

There is not an exact answer here, but the models should be deep enough to recognize facial features and the optimizers should have parameters that help wth model convergence.

You can try to go with SGD or tune Adam's parameters to get a stable convergence but this will do as well.



The project generates realistic faces. It should be obvious that generated sample images look like faces.

There are no ghoulish faces and quite sharp faces as well, I'll give you credit for that.

The question about model improvement is answered.

You are right about the size of the image, but given that if you increase the size and also have deep NNs, the

training time would jump up significantly if you don't have the right GPU or memory.
There is this comparative paper as well which you can read - https://arxiv.org/pdf/1801.04406.pdf

DOWNLOAD PROJECT

RETURN TO PATH

Rate this project