UDACITY

‹ Return to Classroom

Generate Faces

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Dear Student,

well done ⭐ you have completed the face generation assignment with flying colours.

I appreciated the effort you have made to implement the generator as well as the discriminator and achieve the required loss values.

```
Epoch [ 100/ 100] | d_loss: 0.2489 | g_loss: 0.7366
```

You can also improve the current implementation by taking reference from this (link) [https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html]

To understand all functionalities in brief. you can also check this blog link

Best of luck 👍 for your next assignment. till that time, happy learning 📚

Required Files and Tests

✓

The project submission contains the project notebook, called "dlnd_face_generation.ipynb".

Good 👏 all required files are available in the submission.
✅ python notebook

✅ test file

✓

All the unit tests in project have passed.

Great 👍 both unit tests have been passed successfully.
✅ test discriminator
✅ test generator

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.

Good 👏 you have used the Image Folder wrapper. mainly the `image_size = 32` as per the requirement. plus 1 for using `batch size = 32`.
you should keep the batch size in the power of 2. you can take reference from the link

You have also used PyTorch ImageFolder wrapper to read data and applied data transform.
But you can also apply data augmentation and normalization in the transform as follows:

```
transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ])
```

✓

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.

Great, the range for min and max is between -1 to 1.

```
Min:  tensor(-0.9765)
Max:  tensor(0.9843)
```

Good implementation 👍

```
    min, max = feature_range
    x = (max - min) * x + min
```

you can also try to make it one-liner code as follows:

```
return (feature_range[1] - feature_range[0]) * x + feature_range[0]
```

Build the Adversarial Networks

✓

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

good implementation of discriminator model 👌 you have chosen leaky relu activation for all layers and didn't consider batch normalizing at the beginning of the model.

```
class Discriminator(nn.Module):

    def __init__(self, conv_dim):
        """
        Initialize the Discriminator Module
        :param conv_dim: The depth of the first convolutional layer
        """
        super(Discriminator, self).__init__()

        # complete init function
        self.conv1 = conv(3, conv_dim, batch_norm=False)
        self.conv2 = conv(conv_dim, conv_dim*2)
        self.conv3 = conv(conv_dim*2, conv_dim*4)

        self.fc = nn.Linear(conv_dim*4*4*4, 1)

    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: Discriminator logits; the output of the neural network
        """
        # define feedforward behavior
        x = F.leaky_relu(self.conv1(x), 0.2)
        x = F.leaky_relu(self.conv2(x), 0.2)
        x = F.leaky_relu(self.conv3(x), 0.2)

        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

you can also refer to an explanation from here

✓

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

good implementation of generator model 👌 The network is the reverse sequence of the discriminator model and used tanh activation at the end of the model.

```
class Generator(nn.Module):

    def __init__(self, z_size, conv_dim):
        """
        Initialize the Generator Module
        :param z_size: The length of the input latent vector, z
        :param conv_dim: The depth of the inputs to the *last* transpose convolutional l
ayer
        """
        super(Generator, self).__init__()

        self.fc = nn.Linear(z_size, conv_dim*4*4*4)

        self.tconv1 = d_conv(conv_dim*4, conv_dim*2)
        self.tconv2 = d_conv(conv_dim*2, conv_dim)
        self.tconv3 = d_conv(conv_dim, 3, batch_norm=False)

    def forward(self, x):
        """
        Forward propagation of the neural network
        :param x: The input to the neural network
        :return: A 32x32x3 Tensor image as output
        """
        # define feedforward behavior
        x = self.fc(x)
        x = x.view(x.size(0), -1, 4, 4)

        x = F.relu(self.tconv1(x))
        x = F.relu(self.tconv2(x))

        x = torch.tanh(self.tconv3(x))
        return x
```

you can also refer to the explanation from here

✓

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.

Good 👏 you have initialized convolution and linear layer weights.

```
    if classname.find('Linear') != -1 or classname.find('Conv') != -1:
        m.weight.data.normal_(0, 0.02)
```

You can also improve your model performance by initializing weights as follows:

```
if classname.find("Conv") != -1 or classname.find('Linear') != -1:
    torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
elif classname.find("BatchNorm2d") != -1:
    torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
    torch.nn.init.constant_(m.bias.data, 0.0)
```

Optimization Strategy

✓

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

Good that you have used MSE.

```
def real_loss(D_out):
    '''Calculates how close discriminator outputs are to being real.
       param, D_out: discriminator logits
       return: real loss'''
   #  return criterion(D_out.squeeze(),torch.ones(batch_size).cuda())
    return torch.mean((D_out-1)**2)
```

But you can also check the performance by using BCE

```
def real_loss(D_out):
    '''Calculates how close discriminator outputs are to being real.
       param, D_out: discriminator logits
       return: real loss'''
    batch_size = D_out.size(0)
    # real labels = 1
    labels = torch.ones(batch_size)

    # move labels to GPU if available
    if train_on_gpu:
        labels = labels.cuda()

    # numerically stable loss
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)

    return loss
```

You can also try label smoothing to reduce overfitting.

```
def real_loss(D_out, smooth=False):
    '''Calculates how close discriminator outputs are to being real.
       param, D_out: discriminator logits
       return: real loss'''

    batch_size = D_out.size(0)
    # label smoothing
    if smooth:  # smooth, real labels = 0.9 or as given by smooth parameter
        labels = torch.ones(batch_size)*0.9
    else:
        labels = torch.ones(batch_size) # real labels = 1

    # move labels to GPU if available
    if train_on_gpu:
        labels = labels.cuda()

    # binary cross entropy with logits loss
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss
```

You can check the link to understand how label smoothing works.
This is a reference to accelerate your model but this is just a reference for future cases link

✓

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

appropriate hyperparameters.

Great 👍 all hyperparameters are chosen correctly.

Even you have chosen Adam optimizer which performs overall better.

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

# Create optimizers for the discriminator D and generator G
d_optimizer = optim.Adam(D.parameters(), lr, [beta1, beta2])
g_optimizer = optim.Adam(G.parameters(), lr, [beta1, beta2])
```

for reference, you can visit the site.

Training and Results

✓

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

Good work, you have achieved the required loss values below 2.

```
Epoch [ 100/ 100] | d_loss: 0.2489 | g_loss: 0.7366
```

✓

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.

Good 👏 the generator model is deep enough to recognize facial features

✓

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

The generator is creating proper realistic faces.

✓

The question about model improvement is answered.

superb 🔥

- you have pointed out that the dataset contains most of the white faces.
- A large model requires more time to train compared to a smaller one. a large model indeed has more features

that can help to generalize the model even better. but, keep in mind that if the small model can work with nearly the same accuracy as the bigger one, we should consider the smaller model.

- increasing images may help to extract deep features but it will also take more time to get trained.

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

START