

◀ Return to "Deep Learning" in the classroom

# Generate Faces

REVIEW

CODE REVIEW

HISTORY

## Meets Specifications

Dear Learner,

Overall a very good work. You have constructed all necessary components and expressions as part of both the networks and integrated it as part of training logic.

- More or less a balanced loss value at the end of the training. Well Done. This has resulted in generated a good realistic images.
- Perfect convergence of GAN model can be elusive as there are completing architectures and a fool proof loss functions is under the area of research. But we can perform experiments to get better result. Appreciate the effort you had made in this

## Further you can explore to modify the architecture

- It is observed that you have not Drop out layers , which can be incorporated. Check the impact of batch Normalization
- Initialization logic can be changed and you can also experiment by changing the SD for the logic and also using different random functions
- ADAM added as optimizer for both the networks. It is a good choice for GAN based Network. This is one of the area where you can experiment by using variants of ADAM and check the performances.
- Next Level should be for you to experiments with a more advanced architectures in GAN , which includes Cyclic GAN architecture.

[https://pytorch.org/tutorials/beginner/dcgan\\_faces\\_tutorial.html](https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)

All the best

## Required Files and Tests



The project submission contains the project notebook, called "dlnd\_face\_generation.ipynb".

Project notebook submitted as part of the attachment



All the unit tests in project have passed.

All unit test passed. Good work

## Data Loading and Processing



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

Correctly done. Transformation methods incorporated. Returns a DataLoader,

```
def get_data_loader(batch_size, image_size, data_dir='processed_celeba_small/'):
    """
    Batch the neural network data using DataLoader
    :param batch_size: The size of each batch; the number of images in a batch
    :param img_size: The square size of the image data (x, y)
    :param data_dir: Directory where image data is located
    :return: DataLoader with batched data
    """
    transform = transforms.Compose([transforms.Resize(image_size), transforms.ToTensor()])

    dataset = datasets.ImageFolder(data_dir, transform=transform)

    # TODO: Implement function and return a dataloader
    data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                              batch_size=batch_size,
                                              shuffle=True)

    return data_loader
```



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.

Functions constructed with correct expression for scaling the images

Functions constructed with correct expression for scaling the images.

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

## Build the Adversarial Networks



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

Model architecture for the Discriminator is constructed well. However , it lacks any regularization configurations.

- Could have included Dropout layer

Drop out can be included

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

    self.conv_dim = conv_dim

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

    # Final layer
    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
    out = F.leaky_relu(self.conv1(x), 0.2)
    out = F.leaky_relu(self.conv2(out), 0.2)
    out = F.leaky_relu(self.conv3(out), 0.2)

    # flatten
    out = out.view(-1, self.conv_dim * 4 * 4 * 4)

    out = self.fc(out)
    return out
```




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

Good work. You have constructed the generator class with required architecture. Similar to discriminator network, a layers for regularization can be introduced and experimented with



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.

## Initialization logic constructed with correct expressions



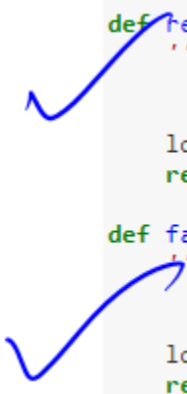
```
def weights_init_normal(m):  
    """  
    Applies initial weights to certain layers in a model .  
    The weights are taken from a normal distribution  
    with mean = 0, std dev = 0.02.  
    :param m: A module or layer in a network  
    """  
  
    # classname will be something like:  
    # `Conv`, `BatchNorm2d`, `Linear`, etc.  
    classname = m.__class__.__name__  
  
    # TODO: Apply initial weights to convolutional and linear layers  
    if classname.find('Conv') != -1 or classname.find('Linear') != -1:  
        nn.init.normal_(m.weight.data, 0, 0.02)  
    if hasattr(m, 'bias') and m.bias is not None:  
        m.bias.data.fill_(0)
```

## Optimization Strategy



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

- Good work on providing expressions as part of the loss functions.



```
def real_loss(D_out):  
    '''Calculates how close discriminator outputs are to being real.  
    param, D_out: discriminator logits  
    return: real loss'''  
    loss = torch.mean((D_out - 1)**2)  
    return loss  
  
def fake_loss(D_out):  
    '''Calculates how close discriminator outputs are to being fake.  
    param, D_out: discriminator logits  
    return: fake loss'''  
    loss = torch.mean(D_out**2)  
    return loss
```



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

- ADAM added as optimizer for both the networks. It is a good choice for GAN based Network. This is one of the area where you can experiment by using variants of ADAM and check the performances.

## Training and Results



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

Training logic is perfect. You have integrated both the model architecture



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 with model convergence.

More or less a balanced loss value at the end of the training. Well Done.

Epoch	[	100/	100]		d_loss:	0.0795		g_loss:	0.9339
Epoch	[	100/	100]		d_loss:	0.0696		g_loss:	0.9262
Epoch	[	100/	100]		d_loss:	0.1816		g_loss:	0.7482
Epoch	[	100/	100]		d_loss:	0.0673		g_loss:	0.8812
Epoch	[	100/	100]		d_loss:	0.1819		g_loss:	0.7866
Epoch	[	100/	100]		d_loss:	0.1121		g_loss:	0.9750

Balanced



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

Great work. Model convergences has resulted in generating realistic faces.  
There are still some issues but overall a good output



The question about model improvement is answered.

Well answered. You have correctly identified the reasons for the performance of the model

RETURN TO PATH

---