```python
import numpy as np

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import sampler

import PIL

NOISE_DIM = 96

dtype = torch.FloatTensor
#dtype = torch.cuda.FloatTensor ## UNCOMMENT THIS LINE IF YOU'RE ON A GPU!

def sample_noise(batch_size, dim, seed=None):
    """
    Generate a PyTorch Tensor of uniform random noise.

    Input:
    - batch_size: Integer giving the batch size of noise to generate.
    - dim: Integer giving the dimension of noise to generate.

    Output:
    - A PyTorch Tensor of shape (batch_size, dim) containing uniform
      random noise in the range (-1, 1).
    """
    if seed is not None:
        torch.manual_seed(seed)

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    sample = torch.rand((batch_size, dim)) * 2 - 1

    return sample

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

def discriminator(seed=None):
    """
    Build and return a PyTorch model implementing the architecture above.
    """

    if seed is not None:
        torch.manual_seed(seed)

    model = None

    ##############################################################################
    # TODO: Implement architecture                                               #
    #                                                                            #
    # HINT: nn.Sequential might be helpful.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = nn.Sequential(
              Flatten(),
              nn.Linear(784, 256),
              nn.LeakyReLU(negative_slope=0.01, inplace=True),
              nn.Linear(256, 256),
              nn.LeakyReLU(negative_slope=0.01, inplace=True),
              nn.Linear(256, 1)
            )

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
```

```python
    return model

def generator(noise_dim=NOISE_DIM, seed=None):
    """
    Build and return a PyTorch model implementing the architecture above.
    """

    if seed is not None:
        torch.manual_seed(seed)

    model = None

    ##############################################################################
    # TODO: Implement architecture                                               #
    #                                                                            #
    # HINT: nn.Sequential might be helpful.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = nn.Sequential(
            nn.Linear(noise_dim, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 784),
            nn.Tanh()
    )

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return model

def bce_loss(input, target):
    """
    Numerically stable version of the binary cross-entropy loss function.

    As per https://github.com/pytorch/pytorch/issues/751
    See the TensorFlow docs for a derivation of this formula:
    https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits

    Inputs:
    - input: PyTorch Tensor of shape (N, ) giving scores.
    - target: PyTorch Tensor of shape (N,) containing 0 and 1 giving targets.

    Returns:
    - A PyTorch Tensor containing the mean BCE loss over the minibatch
      of input data.
    """
    neg_abs = - input.abs()
    loss = input.clamp(min=0) - input * target + (1 + neg_abs.exp()).log()
    return loss.mean()

def discriminator_loss(logits_real, logits_fake):
    """
    Computes the discriminator loss described above.

    Inputs:
    - logits_real: PyTorch Tensor of shape (N,) giving scores for the real data.
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Returns:
    - loss: PyTorch Tensor containing (scalar) the loss for the discriminator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # Target label vector, the discriminator shoud be aiming
```

```
        true_labels = torch.ones(logits_real.size()).type(dtype)

        # Discriminator loss has 2 parts: how well it classifies real images
        # and how well it classifies fake images.
        real_image_loss = bce_loss(logits_real, true_labels)
        fake_image_loss = bce_loss(logits_fake, 1 - true_labels)

        loss = real_image_loss + fake_image_loss


        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return loss

    def generator_loss(logits_fake):
        """
        Computes the generator loss described above.

        Inputs:
        - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

        Returns:
        - loss: PyTorch Tensor containing the (scalar) loss for the generator.
        """
        loss = None
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # Generator is trying to make the discriminator output 1 for all
        # its image. So we create a 'target' label vector of ones for computing
        # generator loss.
        true_labels = torch.ones(logits_fake.size()).type(dtype)

        # Compute the generator loss comparing
        loss = bce_loss(logits_fake, true_labels)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return loss

    def get_optimizer(model):
        """
        Construct and return an Adam optimizer for the model with learning rate 1e-3,
        beta1=0.5, and beta2=0.999.

        Input:
        - model: A PyTorch model that we want to optimize.

        Returns:
        - An Adam optimizer for the model with the desired hyperparameters.
        """
        optimizer = None
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        optimizer = optim.Adam(model.parameters(), lr=1e-3,betas=(0.5, 0.999))

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return optimizer

    def ls_discriminator_loss(scores_real, scores_fake):
        """
        Compute the Least-Squares GAN loss for the discriminator.

        Inputs:
        - scores_real: PyTorch Tensor of shape (N,) giving scores for the real data.
        - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

        Outputs:
        - loss: A PyTorch Tensor containing the loss.
        """
        loss = None
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```python
        true_label = torch.ones(scores_real.size()).type(dtype)

        real_image_loss = torch.mean((scores_real - true_label)**2)
        fake_image_loss = torch.mean(scores_fake**2)

        loss = 0.5 * fake_image_loss + 0.5 * real_image_loss

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return loss

    def ls_generator_loss(scores_fake):
        """
        Computes the Least-Squares GAN loss for the generator.

        Inputs:
        - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

        Outputs:
        - loss: A PyTorch Tensor containing the loss.
        """
        loss = None
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        true_label = torch.ones(scores_fake.size()).type(dtype)

        loss = 0.5 * torch.mean((scores_fake - true_label)**2)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return loss

    def build_dc_classifier(batch_size):
        """
        Build and return a PyTorch model for the DCGAN discriminator implementing
        the architecture above.
        """

        ##############################################################################
        # TODO: Implement architecture                                               #
        #                                                                            #
        # HINT: nn.Sequential might be helpful.                                      #
        ##############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        model = nn.Sequential(
            Unflatten(batch_size, 1, 28, 28),
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=5, stride=1),
            nn.LeakyReLU(negative_slope=0.01, inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1),
            nn.LeakyReLU(negative_slope=0.01, inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            Flatten(),
            nn.Linear(in_features=64*4*4, out_features=64*4*4),
            nn.LeakyReLU(negative_slope=0.1, inplace=True),
            nn.Linear(in_features=64*4*4, out_features=1)
        )

        return model
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################


    def build_dc_generator(noise_dim=NOISE_DIM):
        """
        Build and return a PyTorch model implementing the DCGAN generator using
        the architecture described above.
```

```
    """

    ##########################################################################
    # TODO: Implement architecture                                           #
    #                                                                        #
    # HINT: nn.Sequential might be helpful.                                  #
    ##########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    batch_size = 128

    model =  nn.Sequential(
        nn.Linear(in_features=noise_dim, out_features=1024),
        nn.ReLU(inplace=True),
        nn.BatchNorm1d(num_features=1024),
        nn.Linear(in_features=1024, out_features=7*7*128),
        nn.ReLU(inplace=True),
        nn.BatchNorm1d(num_features=7*7*128),
        Unflatten(N=batch_size, C=128, H=7, W=7),
        nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=4, \
            stride=2, padding=1),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(num_features=64),
        nn.ConvTranspose2d(in_channels=64, out_channels=1, kernel_size=4, \
            stride=2, padding=1),
        nn.Tanh(),
        Flatten()
    )

    return model
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##########################################################################
    #                              END OF YOUR CODE                          #
    ##########################################################################

def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss, \
    loader_train, show_every=250, batch_size=128, noise_size=96, num_epochs=10):
    """
    Train a GAN!

    Inputs:
    - D, G: PyTorch models for the discriminator and generator
    - D_solver, G_solver: torch.optim Optimizers to use for training the
      discriminator and generator.
    - discriminator_loss, generator_loss: Functions to use for computing the
      generator and discriminator loss, respectively.
    - show_every: Show samples after every show_every iterations.
    - batch_size: Batch size to use for training.
    - noise_size: Dimension of the noise to use as input to the generator.
    - num_epochs: Number of epochs over the training dataset to use for training.
    """
    images = []
    iter_count = 0
    for epoch in range(num_epochs):
        for x, _ in loader_train:
            if len(x) != batch_size:
                continue
            D_solver.zero_grad()
            real_data = x.type(dtype)
            logits_real = D(2* (real_data - 0.5)).type(dtype)

            g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
            fake_images = G(g_fake_seed).detach()
            logits_fake = D(fake_images.view(batch_size, 1, 28, 28))

            d_total_error = discriminator_loss(logits_real, logits_fake)
            d_total_error.backward()
            D_solver.step()

            G_solver.zero_grad()
```

```
            g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
            fake_images = G(g_fake_seed)

            gen_logits_fake = D(fake_images.view(batch_size, 1, 28, 28))
            g_error = generator_loss(gen_logits_fake)
            g_error.backward()
            G_solver.step()

            if (iter_count % show_every == 0):
                print('Iter: {}, D: {:.4}, G:{:.4}'.format(iter_count,\
                    d_total_error.item(),g_error.item()))
                imgs_numpy = fake_images.data.cpu().numpy()
                images.append(imgs_numpy[0:16])

            iter_count += 1

    return images


class ChunkSampler(sampler.Sampler):
    """Samples elements sequentially from some offset.
    Arguments:
        num_samples: # of desired datapoints
        start: offset where we should start selecting from
    """
    def __init__(self, num_samples, start=0):
        self.num_samples = num_samples
        self.start = start

    def __iter__(self):
        return iter(range(self.start, self.start + self.num_samples))

    def __len__(self):
        return self.num_samples


class Flatten(nn.Module):
    def forward(self, x):
        # read in N, C, H, W
        N, C, H, W = x.size()
        # "flatten" the C * H * W values into a single vector per image
        return x.view(N, -1)

class Unflatten(nn.Module):
    """
    An Unflatten module receives an input of shape (N, C*H*W) and reshapes it
    to produce an output of shape (N, C, H, W).
    """
    def __init__(self, N=-1, C=128, H=7, W=7):
        super(Unflatten, self).__init__()
        self.N = N
        self.C = C
        self.H = H
        self.W = W
    def forward(self, x):
        return x.view(self.N, self.C, self.H, self.W)

def initialize_weights(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.ConvTranspose2d):
        nn.init.xavier_uniform_(m.weight.data)

def preprocess_img(x):
    return 2 * x - 1.0

def deprocess_img(x):
    return (x + 1.0) / 2.0

def rel_error(x,y):
```

```
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def count_params(model):
    """Count the number of parameters in the current TensorFlow graph """
    param_count = np.sum([np.prod(p.size()) for p in model.parameters()])
    return param_count
```