DCGAN in PyTorch

- get data

```
import torch
from torchvision import datasets, transforms

transform = transforms.ToTensor() # transforming img to tensor
train_data = datasets.SVHN(root='data', split='train',
                           download=True, transform = transform)
```

SVHN → dataset for real-world street numbers

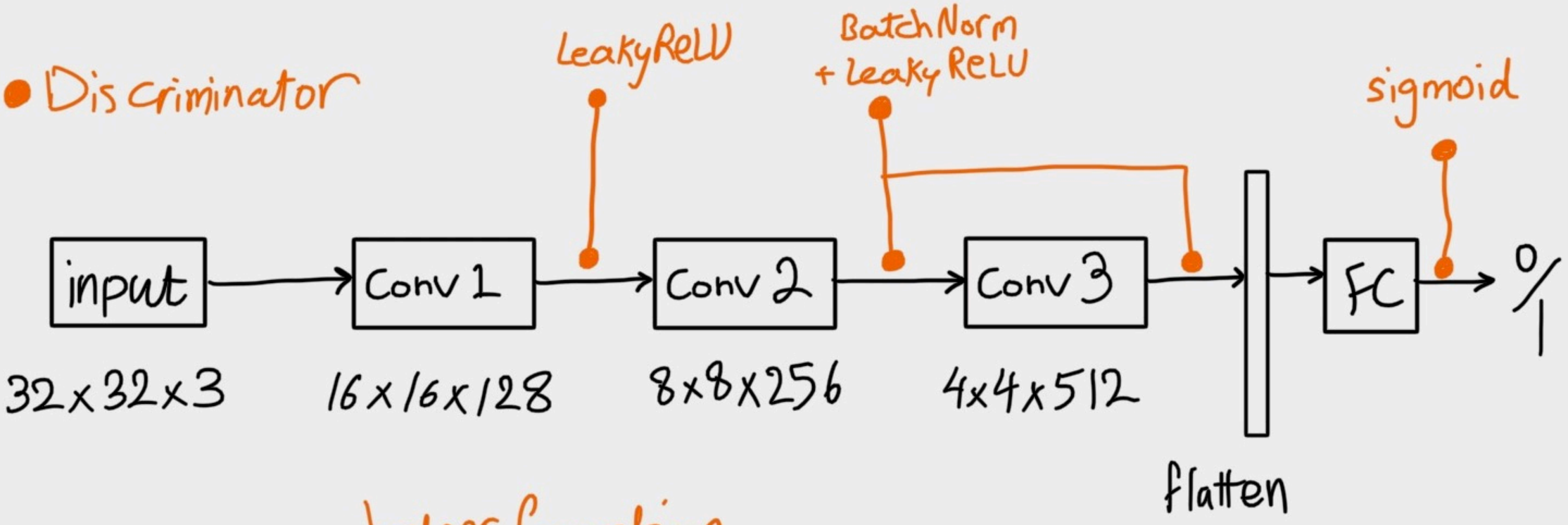
```
bsize = 128
train_loader = torch.utils.data.DataLoader ( dataset = train_data,
                                              batch_size = bsize,
                                              shuffle= True,
                                              num_workers = 0 )
```

- Pre-processing

```
def scale (x, feature_range=(-1,1)):
    min, max = feature_range
    x = x * (max - min) + min
    return x
```

(-1,1) → scale to feature range

- Discriminator



helper function

```
def Conv(in_chan, out_chan, kernel_size,
         stride=2, padding=2, batch_norm=True):
    layers = []
    Conv_layer = nn.Conv2d(in_chan, out_chan, kernel_size,
                           stride, padding, bias=False)
    layers.append(Conv_layer)
    if batch_norm:
        layers.append(nn.BatchNorm2d(out_chan))
    return nn.Sequential(*layers)
```

Create a Conv layer

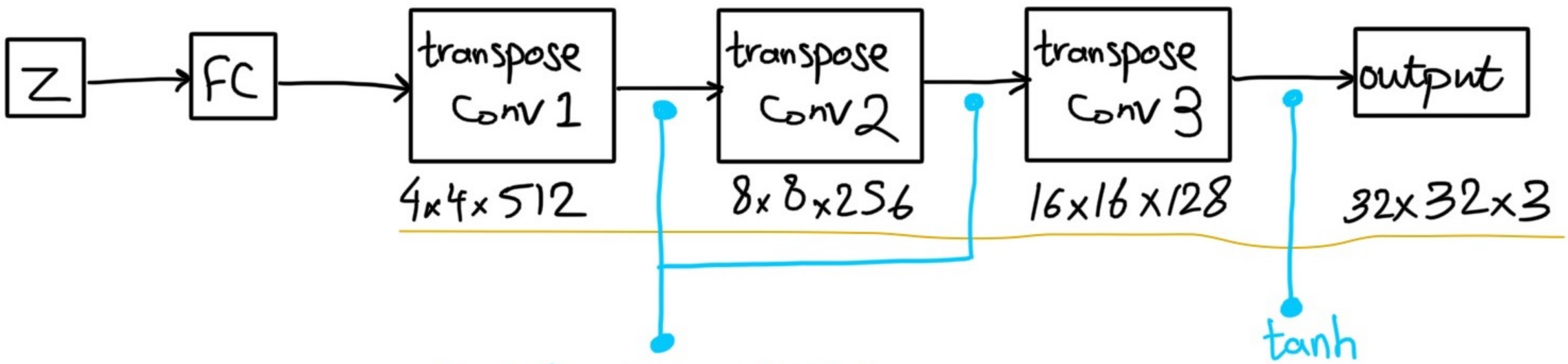
stack them

add batch norm

Create a sequence of layers

```
class D(nn.Module):
    def __init__(self, conv_dim=32):
        super(D, self).__init__()
        self.conv_dim = conv_dim

        # input: 32x32
16x16:  self.conv1 = conv(3, conv_dim, 4, batch_norm=False)
8x8:    self.conv2 = conv(conv_dim, conv_dim*2, 4)
4x4:    self.conv3 = conv(conv_dim*2, conv_dim*4, 4)

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

    def forward(self, x):
        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)

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

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

- Generator



helper to make transpose Conv

```
def deconv(in_chan, out_chan, kernel_size,
           stride=2, padding=1, batch_norm = True):
    layers = []
    conv_layer = nn.ConvTranspose2d(in_chan, out_chan, kernel_size,
                                    stride, padding, bias= False)
    layers.append(conv_layer)

    if batch_norm:
        layers.append(nn.BatchNorm2d(out_chan))

    return nn.Sequential(*layers)
```

only diff

```
class G(nn.Module):
    def __init__(self, z_size, conv_dim = 32):
        super(G,self).__init__()
        self.conv_dim = conv_dim
        self.fc = nn.Linear(z_size, conv_dim*4*4*4)
        self.tconv1 = deconv(conv_dim*4, conv_dim*2, 4)
        self.tconv2 = deconv(conv_dim*2, conv_dim, 4)
        self.tconv3 = deconv(conv_dim, 3, 4, batch_norm=false)
```

this should gen enough output

transpose Conv layers

```
def forward(self, x):
    out = self.fc(x)
    out = out.view(-1, self.conv_dim*4, 4,4)
    out = F.relu(self.tconv1(out))
    out = F.relu(self.tconv2(out))
    out = F.tanh(self.tconv3(out))

    return out
```

batch size

depth

32x32x3

- glue D and G together

```
conv_dim = 32
z_size = 100

d = D(conv_dim)
g = G(z_size, conv_dim)

is_gpu = torch.cuda.is_available()
if is_gpu:
    g.cuda()
    d.cuda()
```

- define losses

```
def real_loss(D_out, smooth=False):
    b = D_out.size(0)
    if smooth:
        labels = torch.ones(b) * 0.9
    else:
        labels = torch.ones(b)
    if is_gpu: labels = labels.cuda()

criterion = nn.BCEWithLogitsLoss()
return criterion(D_out.squeeze(), labels)
```

```
def real_loss(D_out):
    b = D_out.size(0)
    labels= torch.zeros(b)
    if is_gpu:   labels = labels.cuda()

    criterion = nn.BCEWithLogitsLoss()
    return criterion(D_out.squeeze(), labels)
```

- define optimizer

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

→ from a paper or so experiment

```
d_opt = optim.Adam(d.parameters(), lr, [beta1, beta2])
g_opt = optim.Adam(g.parameters(), lr, [beta1, beta2])
```

- Train the models

```
d.train()
g.train()
for epoch in range(epochs):
    for idx, (real_im, _) in enumerate(train_loader):
        bsize = real_im.size(0)
        real_im = scale(real_im)
```

discriminator

```
d_optim.zero_grad()
d_real = d(real_images)
d_real_loss = real_loss(d_real, smooth=True)
```

seems to be better w/o smoothing!

```
z = np.random.uniform(-1, 1, size=(batch_size, z_size))
z = torch.from_numpy(z).float()
fake_images = G(z)
d_fake = d(fake_images)
d_fake_loss = fake_loss(d_fake)

d_loss = d_real_loss + d_fake_loss
d_loss.backward()
d_optim.step()
```

generator

```
g_optim.zero_grad()

z = np.random.uniform(-1, 1, size=(batch_size, z_size))
Z = torch.from_numpy(z).float()
fake_images = g(z)

d_fake = d(fake_images)
g_loss = real_loss(d_fake)
g_loss.backward()
g_optim.step()
```

```
losses.append((d_loss.item(), g_loss.item()))
samples.append(G(fixed_z))
```

generated images ← samples.append(G(fixed_z))