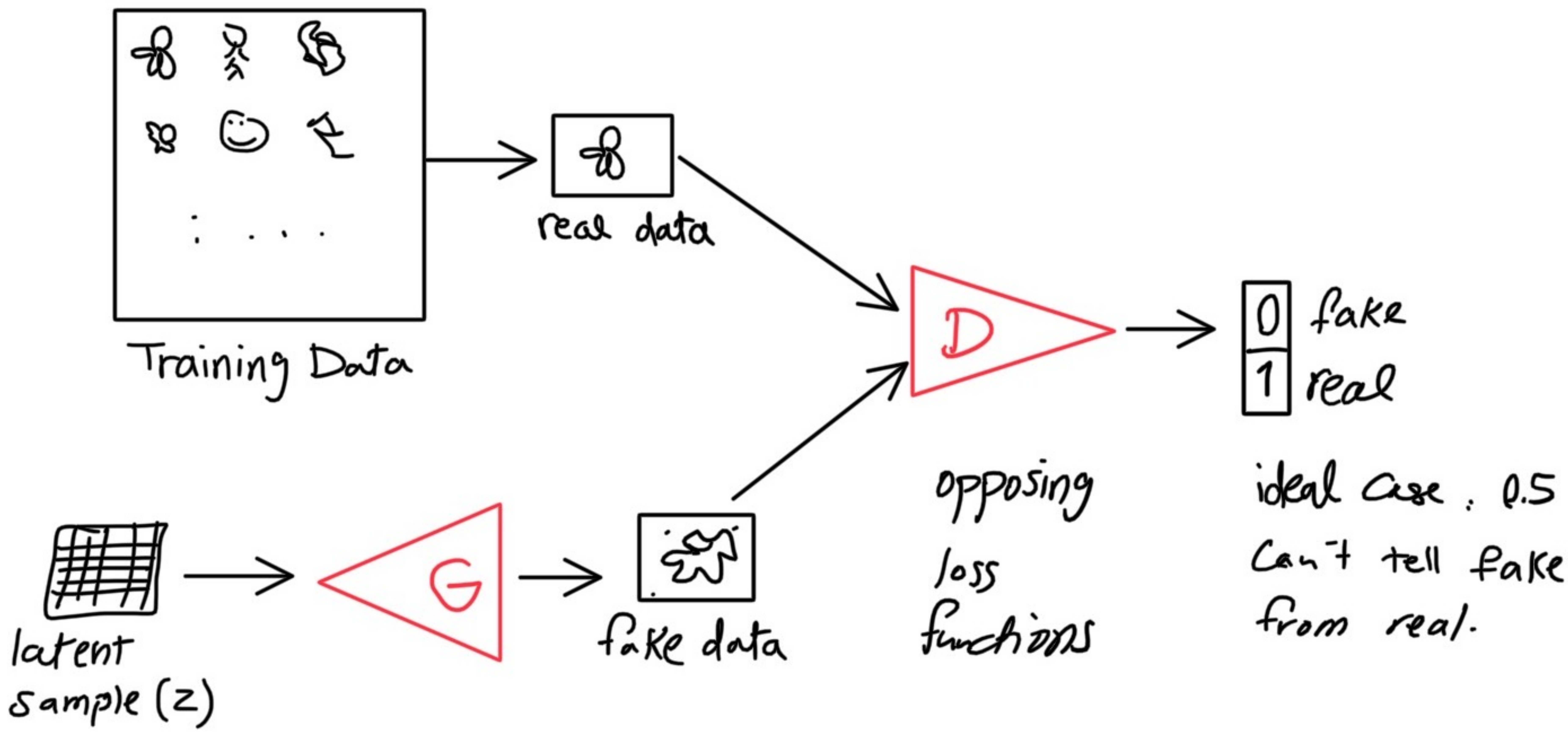
GANs in PyTorch — Simple Example




```
import numpy as np, torch
import matplotlib.pyplot as plt
from torchvision import datasets
import torchvision.transforms as transforms
```

- handle the data

```
bsize = 64
transform = transforms.ToTensor()
train_data = datasets.MNIST(root='data', train = True,
                            download=True,
                            transform=transform)
train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=bsize,
                                           num_workers =0)
```

- Visualize the data

```
dataiter = iter(train_loader)
im, lb = dataiter.next()  → get one batch
im = im.numpy()


im=np.squeeze(im[0])  → get one image from batch
                        (28×28 image)

fig =plt.figure(figsize = (3, 3))
ax = fig.add_subplot(111)
ax.imshow(im, cmap='gray')
```

- # Define the Discriminator

```
import torch.nn as nn, torch.nn.functional as F
class Discriminator(nn.Module):
    def __init__(self, input_size, hidden_dim, output_size):
        super(Discriminator, self).__init__()
```

hidden layers

```
        self.fc1 = nn.Linear(input_size, hidden_dim*4)
        self.fc2 = nn.Linear(hidden_dim*4, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, hidden_dim)
```

smaller dim

final layer

```
        self.fc4 = nn.linear(hidden_dim, output_size)
```

sigle value

aka down sample

```
        self.dropout = nn.Dropout(0.3)
```

```
def forward(self, x):
    x = x.view(-1, 28*28)
    x = F.leaky_relu(self.fc1(x), 0.2)
    x = self.dropout(x)
    x = F.leaky_relu(self.fc2(x), 0.2)
    x = self.dropout(x)
    x = F.leaky_relu(self.fc3(x), 0.2)
    x = self.dropout(x)

    out = self.fc4(x)
    return out
```

→ flatten image

apply hidden layers w/ LReLU

↓

we need to have LReLU here in hidden layers

good practice to add dropout after each fc layer.

→ final layer

```
class Generator(nn.Module):
    def __init__(self, input_size, hidden_dim, output_size):
        super(Generator, self).__init__()
```

similar to Discriminator

```
        self.fc1 = nn.Linear(input_size, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, hidden_dim*4)

        self.fc4 = nn.linear(hidden_dim*4, out_size)

        self.dropout = nn.Dropout(0.3)
```

larger dim

aka up sample

$\underbrace{\texttt{out\_size}}$ Can be reshaped into an image.

```
    def forward(self, x):
```

flattening is <u>not</u> needed here.

similar to Discriminator

```
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        x = self.fc4(x)

        out = F.tanh(x)
        return out
```

→ apply tanh to last layer scales values to [-1, 1]

- Hyperparameters

input_size = 28*28 → image size
d_out_size = 1 → fake or real prob.
d_hid_size = 32 → last hidden layer of D

z_size = 100 → latent vector given to G.
g_out_size = 28*28
g_hid_size = 32 → first hidden layer of G

- Make model instances

D= Discriminator (input_size, d_hid_size, d_out_size)
G= Generator (z_size, g_hid_size, g_out_size)

- Loss Functions

numerically stable ?

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

→ we know that for real images, label = 1.

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

remove empty dims

```
def fake_loss(D_out)
    bs = D_out.size(0)
    labels = torch.zeros(bs)
```

→ we know that for fake images, label = 0.

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

remove empty dims

- Optimizer

```
import torch.optim as optim
lr= 0.002
d_opt= optim.Adam(D.parameters(), lr)
g_opt= optim.Adam(G.parameters(), lr)
```

- Training

Discriminator

1. Compute d_loss for real images
2. Generate fake images
3. Compute d-loss for fake images
4. loss is fake loss + real loss
5. backprop & optim step for D's weights

Generator

1. Generate fake images
2. Compute d-loss for fake images, with flipped labels
3. backprop & optim step for G's weights

* Traing Loop

```
import pickle as pkl


epochs = 100
samples, losses = [], []
```

Some fixed sample data to debug the model.

```
sample_size = 16
fixed_Z = np.random.uniform(-1, 1, size=(sample_size, z_size))
fixed_Z = torch.from_numpy(fixed_Z).float()


D.train()
G.train()
for epoch in range(epochs):
    for idx, (real_im, _) in enumerate(train_loader):
        bsize = real_im.size(0)

      ★ real_im = real_im * 2 - 1
```

rescale the input images from [0,1) to [-1,1)

DISCRIMINATOR

---

train w/ real images

```
        d_optimizer.zero_grad()
        D_real = D(real_images)
        d_real_loss = real_loss(D_real, smooth=True)
```

train w/ fake images

```
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)
```

generates fake images

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

GENERATOR

```
g_optimizer.zero_grad()
z = np.random.uniform(-1, 1, size=(batch_size, z_size))
Z = torch.from_numpy(z).float()
fake_images = G(z)
```

generates fake images

```
D_fake = D(fake_images)
g_loss = real_loss(D_fake)
```

adversarial loss

* here, we computed loss of D on fake images. G aims to make it so that fake images get labels closer to 1.

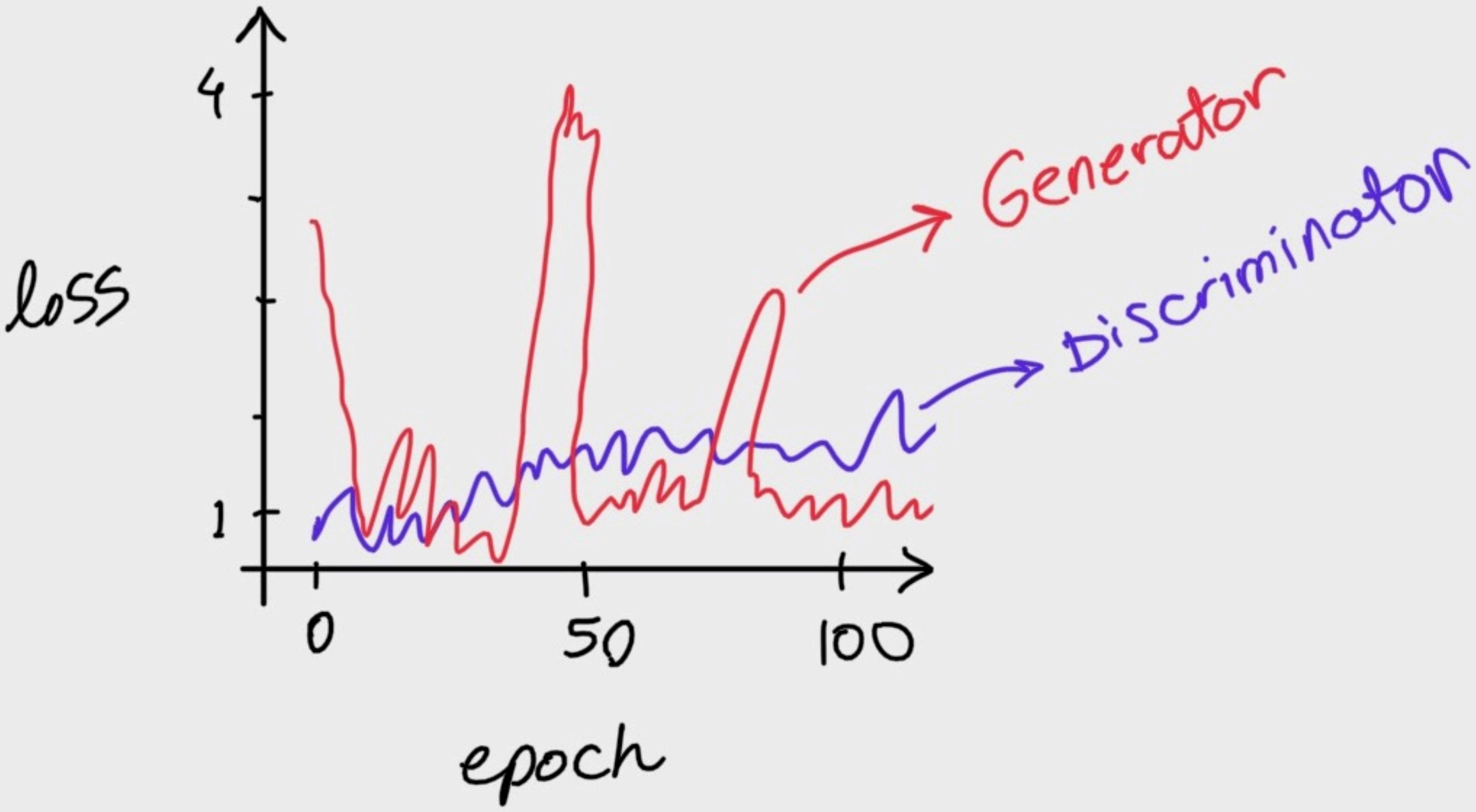
```
g_loss.backward()
g_optim.step()
```

```
losses.append((d_loss.item(), g_loss.item()))
```

```
samples.append(G(fixed_z))
```
→ gen some fake images

```
with open('samples.pkl', 'wb') as f:
    pkl.dump(samples, f)
```
and save them

How does loss look over time?



its normal to see these fluctuations, because the two models are competing against each other.