# DEEP NEURAL NETWORKS

# MOD1: GAN MOTIVATED

# PROBABILISTIC MODELLING

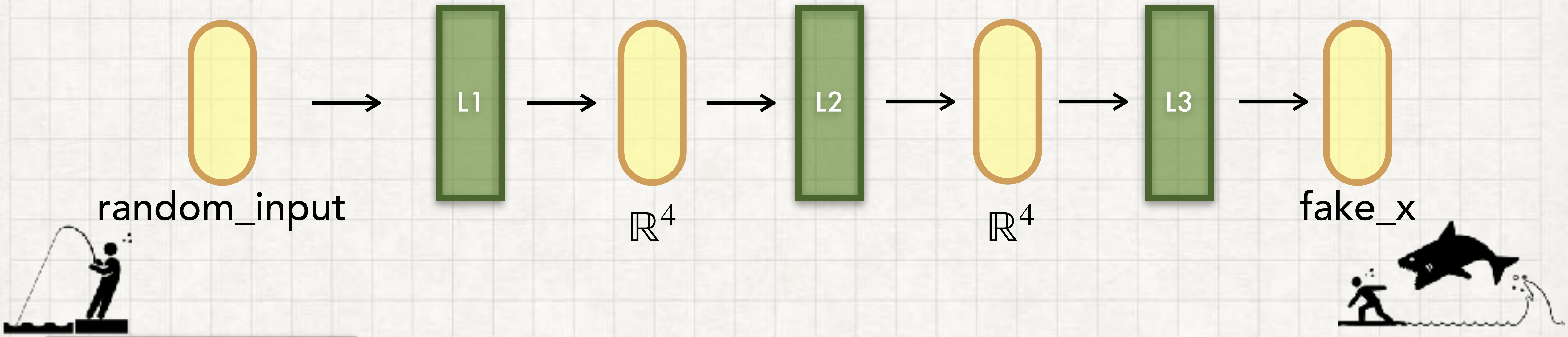
- Attempt 1
  - For every $x \in \mathcal{X}$, model p(x)

### 3.1.1 Probabilistic Density (PDF) Function Approx.

```
# def
lin1 = nn.Linear(in_features=1, out_features=4)
lin2 = nn.Linear(in_features=4, out_features=1)

# forward
p = sigmoid((lin2(active(lin1(x)))) # prob in [0, 1]

…
for i in range(MAX_TRAIN):
    pred = pdf_net_(x_trn)
    loss = mse_loss(pred, y_trn)
    loss.backward()
    …
```

# PROBABILISTIC MODELLING

- Attempt 2
  - Let us generate samples in $\mathscr{X}$. We start by guessing.



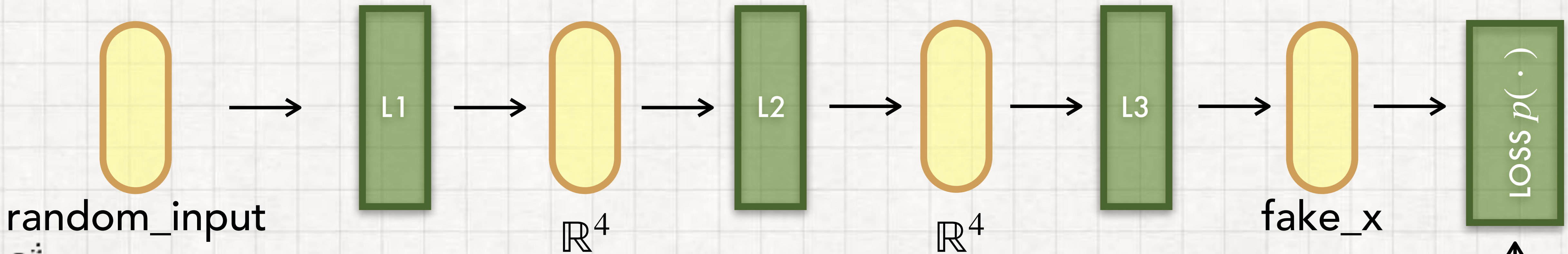

## 3.1.2.Generator

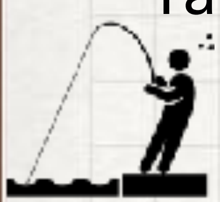
```
# def:
lin1 = Linear(in_features=1, out_features=4)
lin2 = Linear(in_features=4, out_features=4)
lin3 = nn.Linear(in_features=4, out_features=1)

# forward(…):
random_input = torch.rand()
out = lin3(activ(lin2(activ(lin1(random_input)))))
```

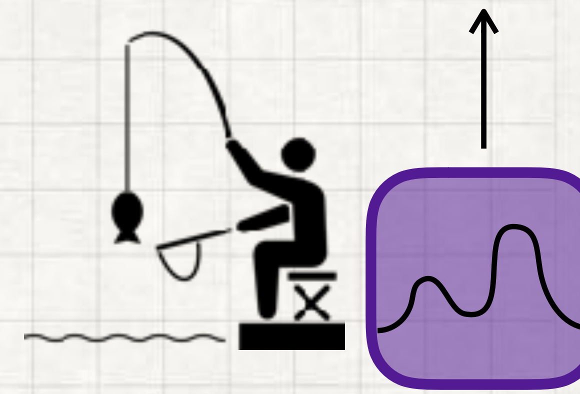
# PROBABILISTIC MODELLING

- Attempt 2

  - Generate samples in $\mathscr{X}$ - Then adjust model to maximise $p(x_{fake})$



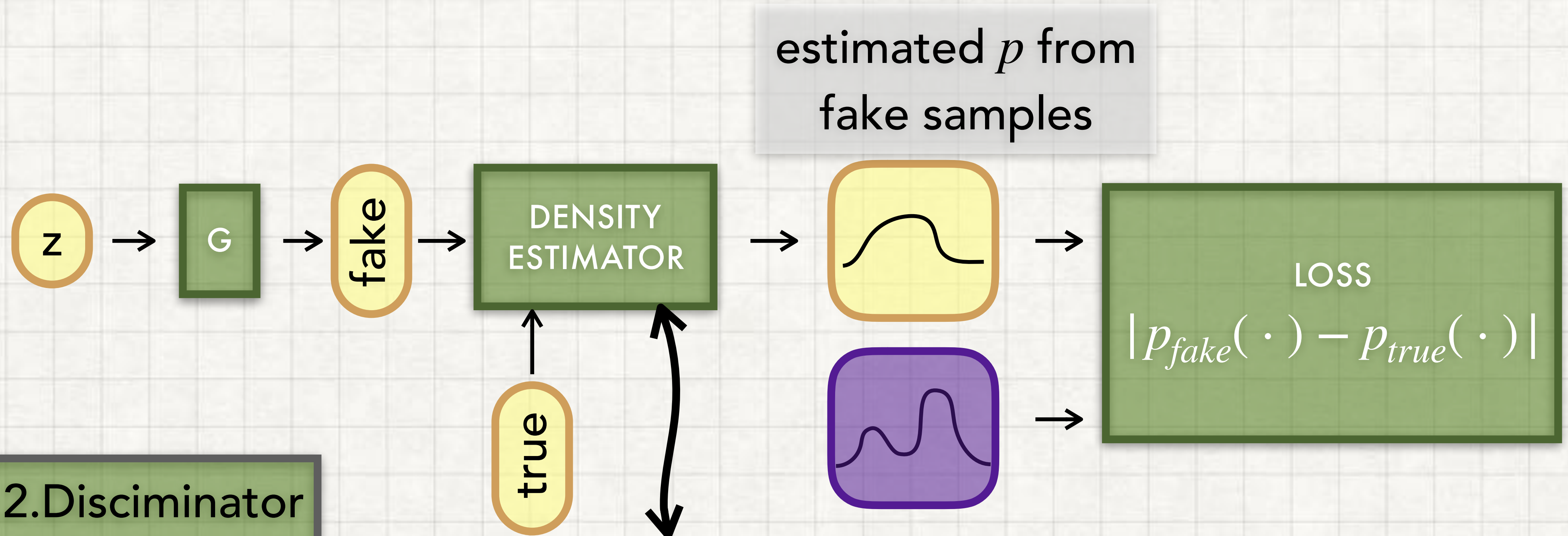


## 3.1.2.Generator-Naive Improve

```
# backward(…):
fake_x = model_g(random_input)
fake_prob = logpdf_fn_t(fake_x.squeeze())
loss = - fake_prob.mean()
loss.backward()
# adjust model according to grad
optim_g.step()
```

# PROBABILISTIC MODELLING

- Attempt 3
  - Generate samples in $\mathscr{X}$ - Then adjust model to maximise $p(x_{fake})$

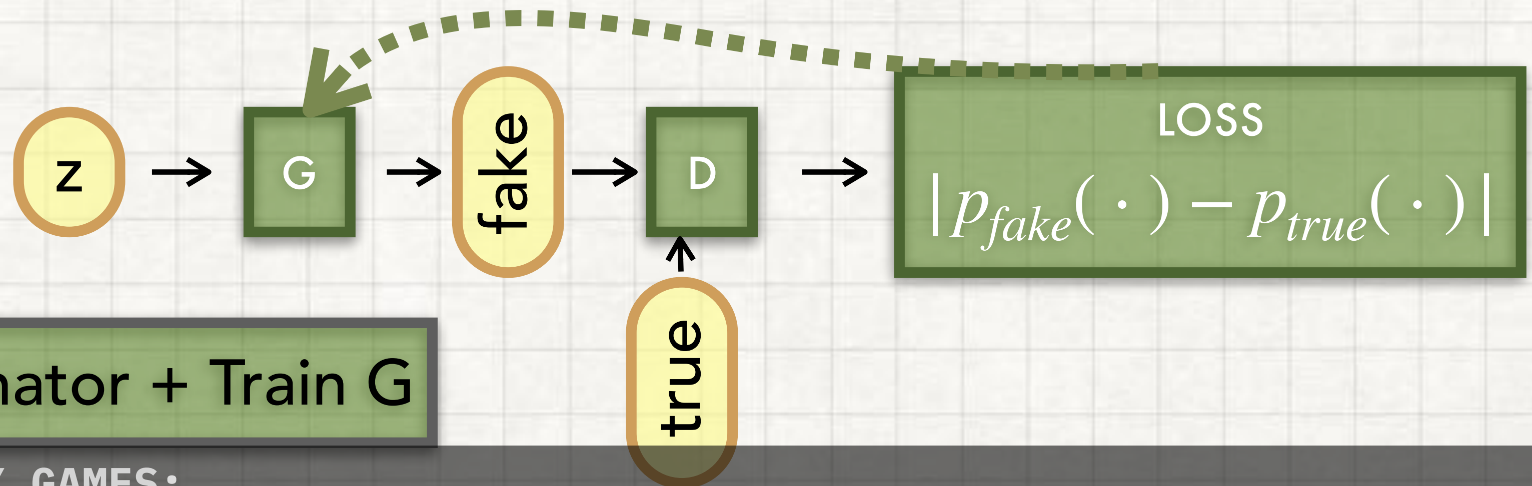



3.1.2.Disciminator

```
def fit_fake_density(fake_density_model, fake_samples, x_values):
    # make mixing true/false sample dataset,
    # x_values are from X-space
    train_dataset = FakeSampleInXSpaceDataset(fake_samples, x_values)

    # in training …
        pred = fake_density_model(x.unsqueeze(dim=1))
        loss =  criterion(pred.squeeze(), y) # pred: probability of being true samples
        loss.backward()
        optim.step()
```

# PROBABILISTIC MODELLING

- Attempt 3
  - Generate samples in $\mathscr{X}$ - Then adjust model to maximise $p(x_{fake})$




3.1.2.Discriminator + Train G

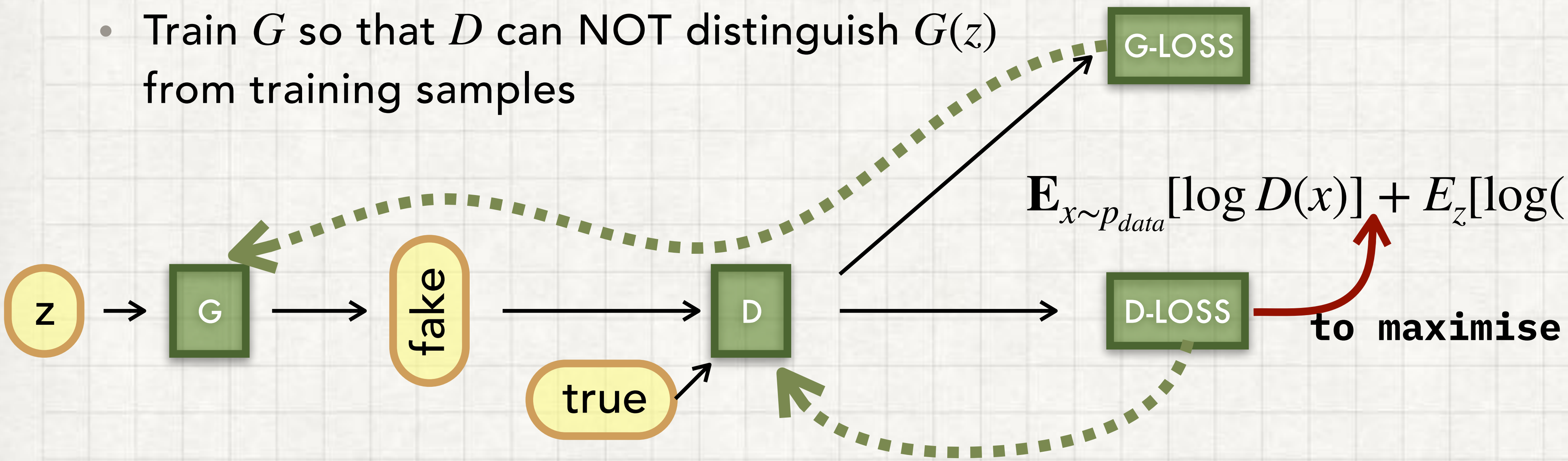
```
while game < MAX_GAMES:
    # 1. Train G for a while
    for i in range(MAX_TRAIN_GEN):
        fake_x = model_g(random_input)
        loss = - logpdf_fn_t(fake_x).mean()
        …
    # 2. Fit a PDF to this G-generated samples
    fake_x = model_g(random_input).detach()
    fit_fake_density(fake_density_model, fake_x, x_grid)

    # 3. Density estimator checks G-generated distrib. ~ true data distrib.
    for i in range(MAX_ADJUST_ITERS):

        fake_x = model_g(random_input) # No detach.
        gen_sample_fake_likelihood = fake_density_model(fake_x)
        gen_sample_true_likelihood = logpdf_fn_t(fake_x)
        loss = (gen_sample_fake_likelihood - gen_sample_true_likelihood).mean()
        …
```

# GAN FRAMEWORK

- GAN Framework
  - $G$ generates samples in $\mathscr{X}$ by mapping random noises $G(z)$
  - Train $D$ to distinguish $G(z)$ from training samples (**Remove the density estimation step**)
  - Train $G$ so that $D$ can NOT distinguish $G(z)$ from training samples

# GAN FRAMEWORK

- GAN Framework
  - $G$ generates samples in $\mathscr{X}$ by mapping random noises $G(z)$
  - Train $D$ to distinguish $G(z)$ from training samples (**Remove the density estimation step**)
  - Train $G$ so that $D$ can NOT distinguish $G(z)$ from training samples




$$\mathbf{E}_{x \sim p_{data}}[\log D(x)] + E_z[\log(1 - D(G(z)))]$$

# GAN IMPLEMENTATION

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, $k$, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

**for** number of training iterations **do**

  **for** $k$ steps **do**

  - Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
  - Sample minibatch of $m$ examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right]$$

  **end for**

  - Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

**end for**

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

```
for i in range(max_train_iters):
    random_inputs = torch.randn(fake_sample_num)
    with torch.no_grad(): # take G as fixed when training D
        fake_samples = generator(random_inputs)

    x = torch.cat((train_samples, fake_samples), dim=0)
    gnd = torch.cat((ones(train_sample_num),zeros(fake_sample_num)))

    #train discriminator
    for j in range(2):
        pred = discrim(x)
        # discriminator loss
        loss_discrim = -(pred * (gnd - 0.5) * 2).sum()
        …
    # train generator
    gnd = torch.ones(fake_sample_num)
    random_inputs = torch.randn(fake_sample_num).unsqueeze(dim=1)
    fake_samples = generator(random_inputs)
    output = discrim(fake_samples).squeeze()
    # generator loss
    loss_gen = - output.mean()
    …
```

I. Goodfellow et al. "Generative Adversarial Nets", 2014

# MOD2: GAN OF IMAGE DATASET

# THANKS