Generative Adversarial Networks (GAN)

Instructor

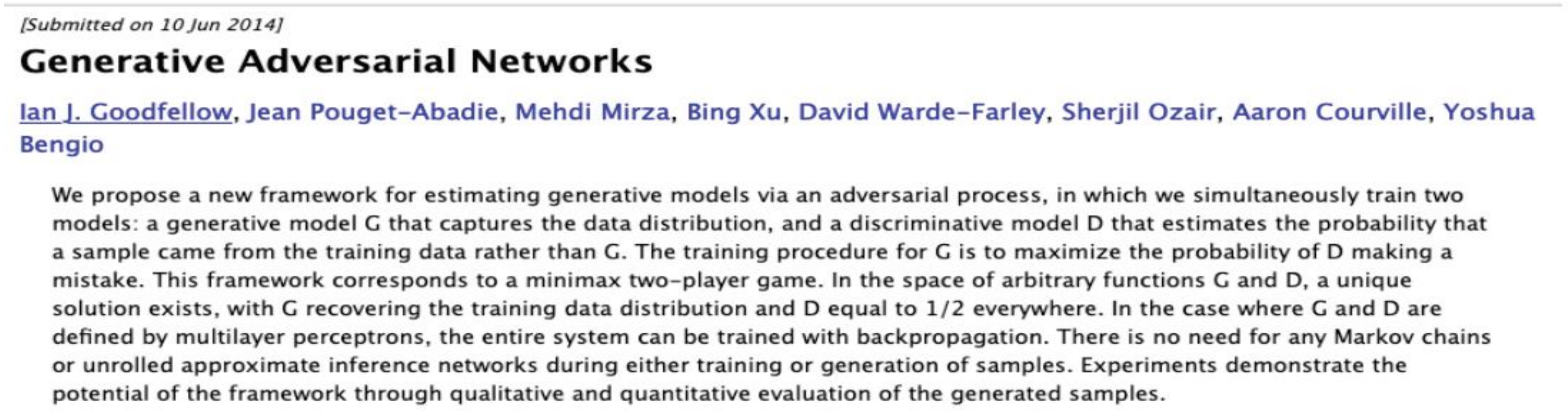
Abdul Qayyum, PhD

Date:20-10-2020

Generative Adversarial Networks (GAN)

- The original purpose is to generate new data
- Classically for generating new images, but applicable to wide range of domains
- Learns the training **set distribution** and can generate new images that have never been seen before
- In contrast, e,g,; autoregressive models or RNNs (generating one word at a time), **GANs generate the whole output all at once**

Generative Adversarial Networks (GAN)



[Submitted on 10 Jun 2014]

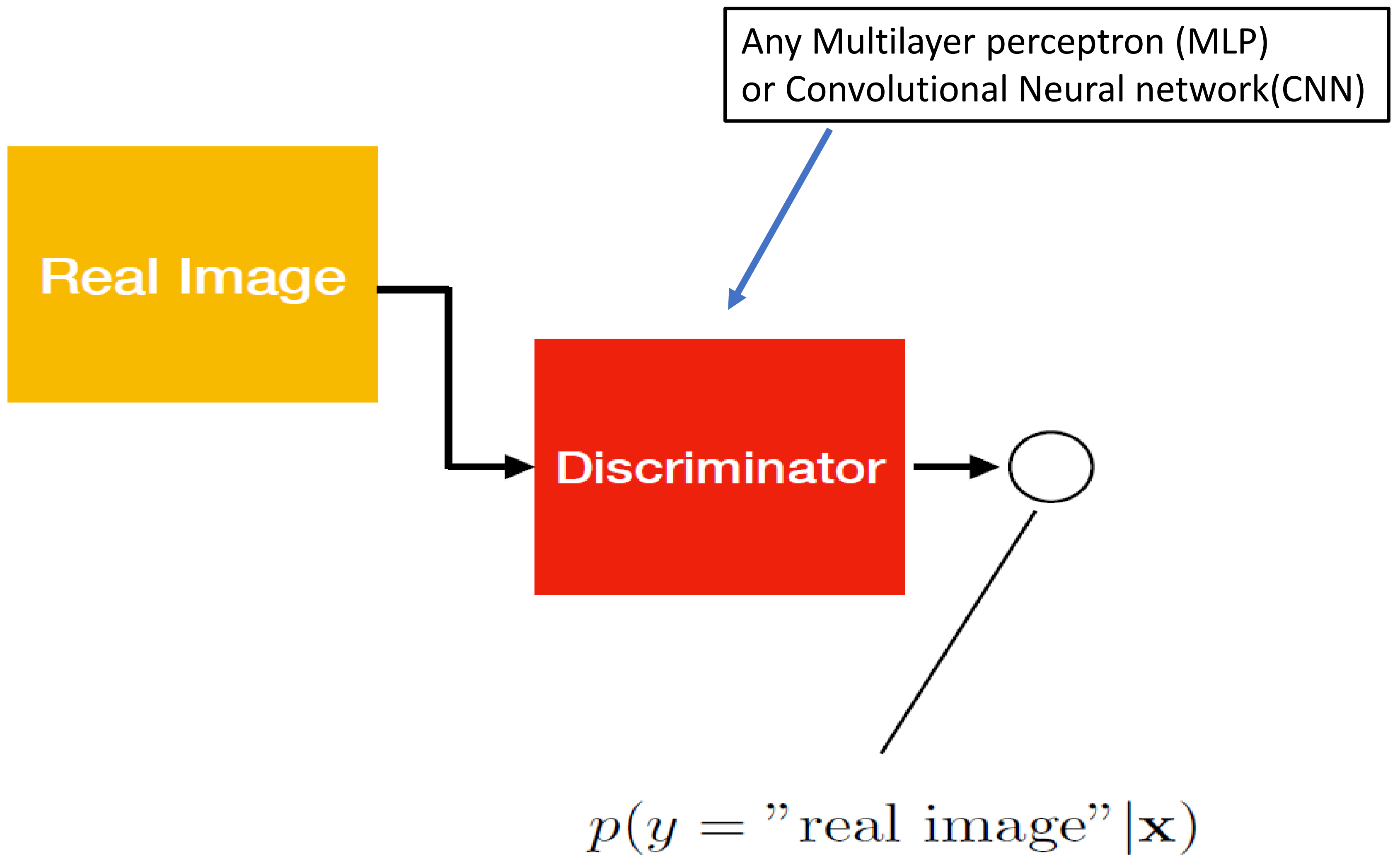
Generative Adversarial Networks

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to 1/2 everywhere. In the case where G and D are defined by multilayer perceptrons, the entire system can be trained with backpropagation. There is no need for any Markov chains or unrolled approximate inference networks during either training or generation of samples. Experiments demonstrate the potential of the framework through qualitative and quantitative evaluation of the generated samples.

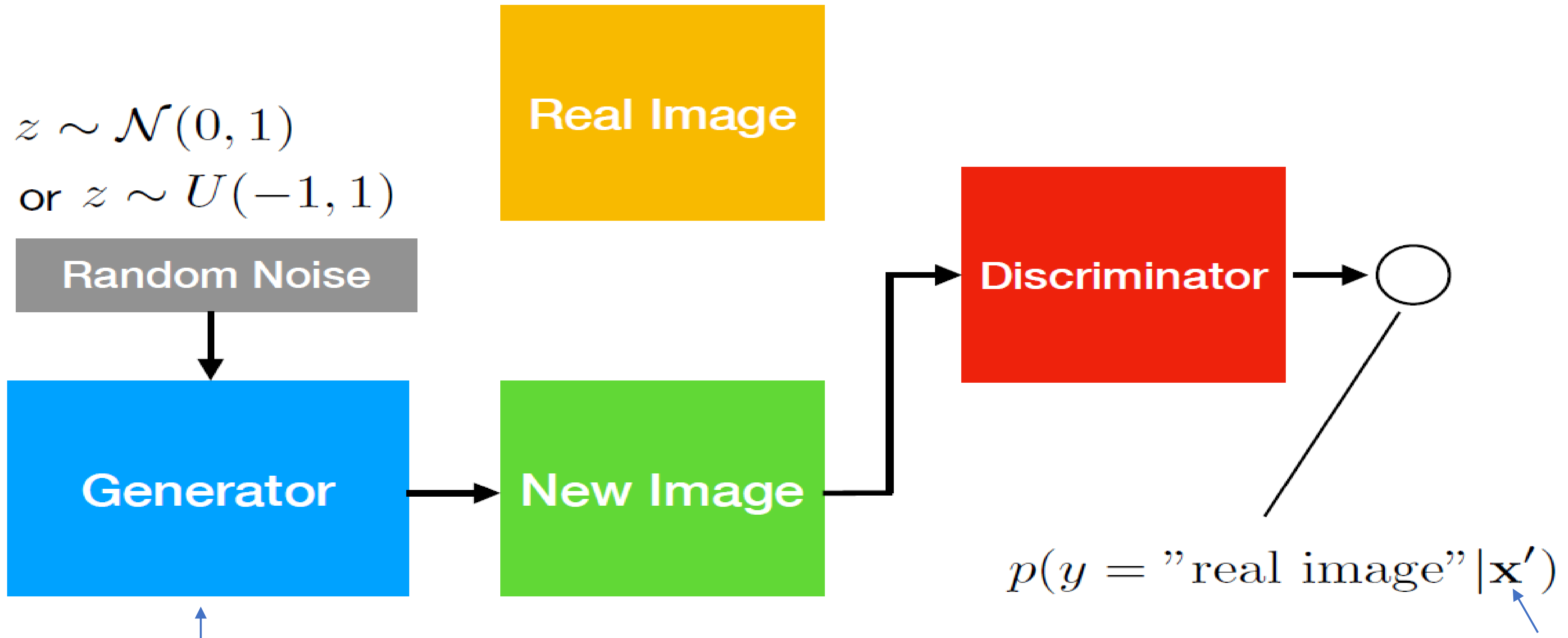
https://arxiv.org/abs/1406.2661

Generative Adversarial Networks (GAN)


Any Multilayer perceptron (MLP)
or Convolutional Neural network(CNN)
Real Image
Discriminator

$p(y = \text{"real image"} | \mathbf{x})$

Generative Adversarial Networks (GAN)

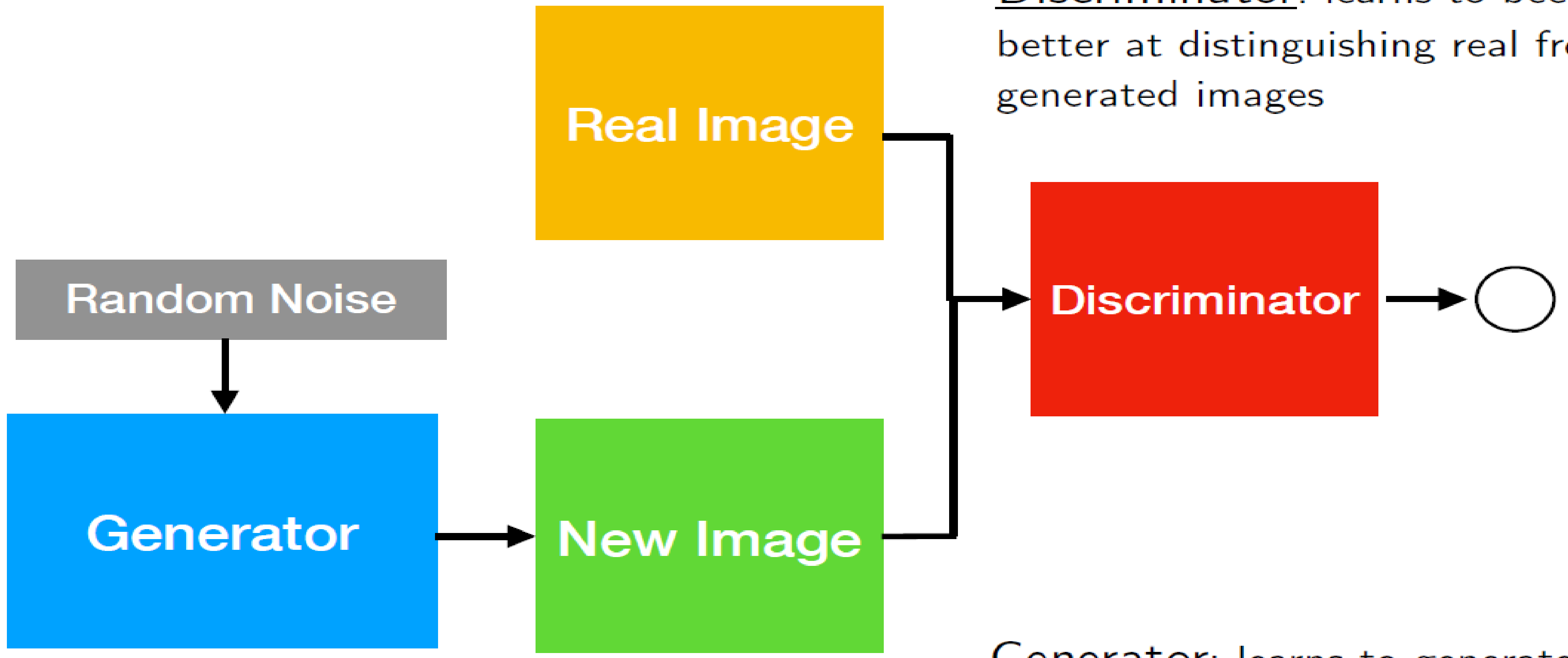


Any Multilayer perceptron (MLP)
or Convolutional Neural network(CNN)

Fake image

Generative Adversarial Networks (GAN)

Adversarial Game

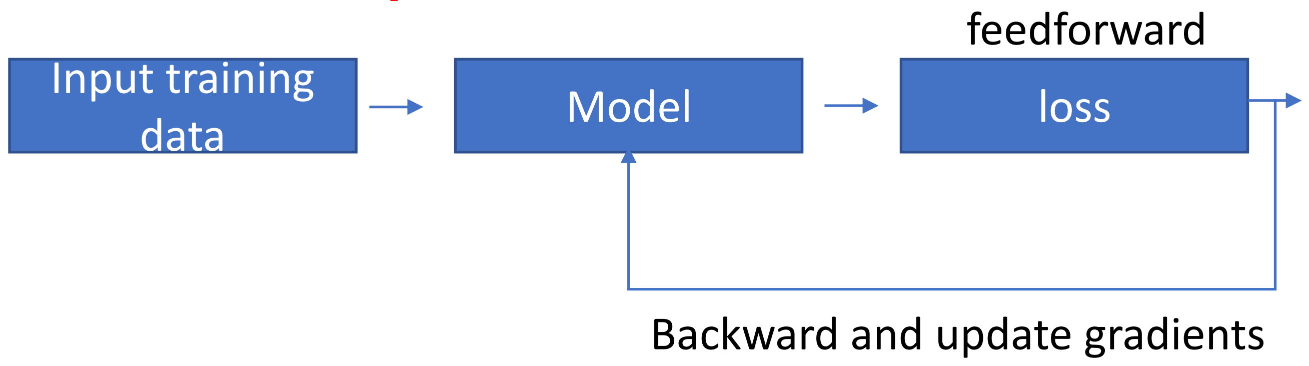


Discriminator: learns to become better at distinguishing real from generated images

Generator: learns to generate better images to fool the discriminator

Generative Adversarial Networks (GAN)

How to train simple neural network



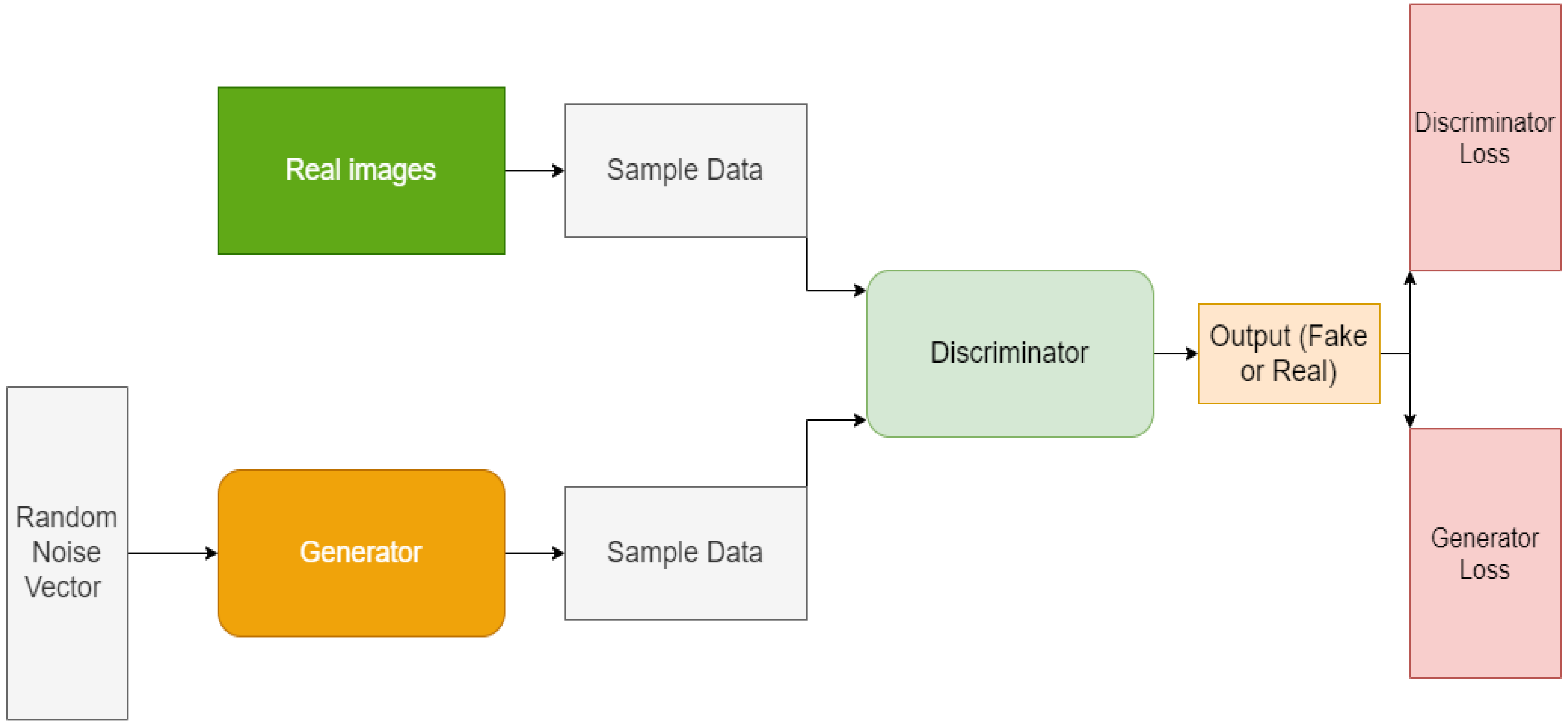
Step1: Input data and labels
Step 2: Define the model and get output
Setp3: calculate the loss (feedforward + backward)
setp4: update the gradient
Step5: repeat the process

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
For Ibatch,Ilabel in dataloader:
        imagebatch=Ibatch.cuda()
        Gtlabel=Ilabel.cuda()
        Optimizer.zero_grad() # zero gradient
       # forward+backward+update the gradient
        ypredic=model(imagebatch)
        Loss=lossfun(Gtlabel,ypredic) # compute loss
         loss.backward() # calculate the gradients
         Optimizer.step() # update the gradients
```

Generative Adversarial Networks (GAN)

GANs Contain Two Neural Networks



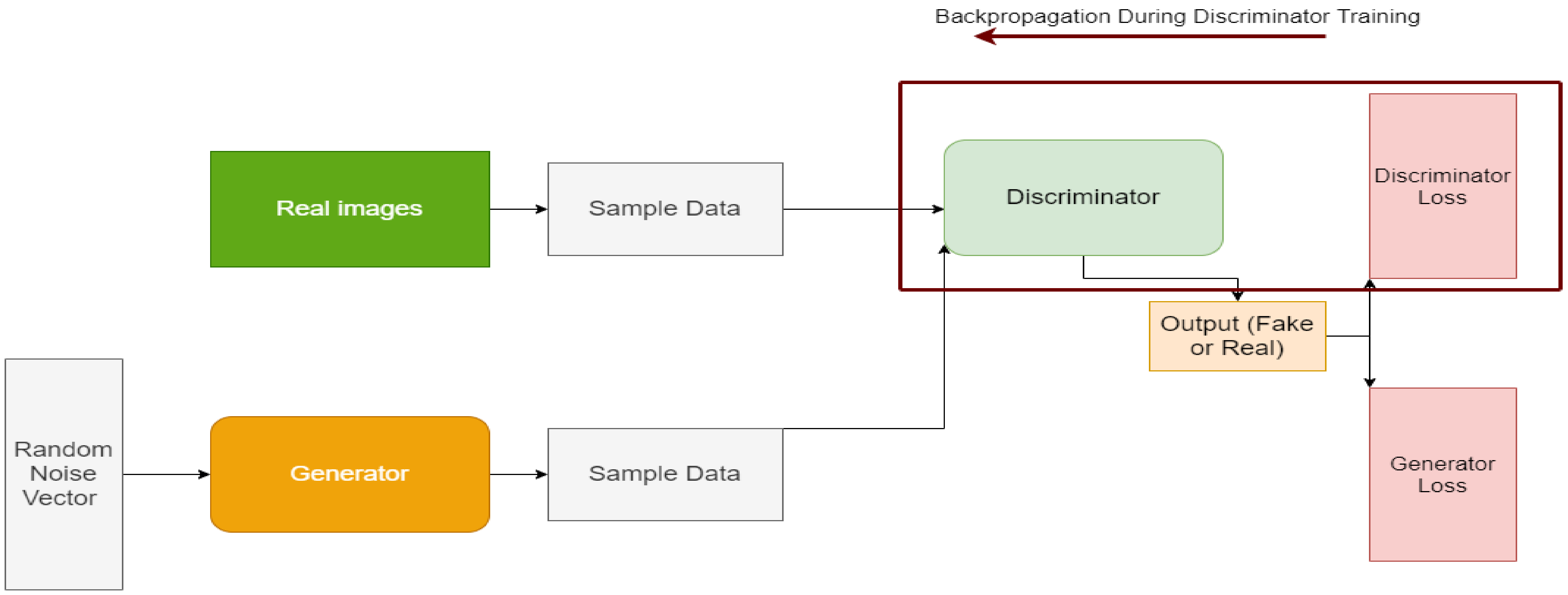
Architecture of a Generative Adversarial Network.

https://debuggercafe.com/introduction-to-generative-adversarial-networks-gans/

Generative Adversarial Networks (GAN)

The Discriminator in GAN

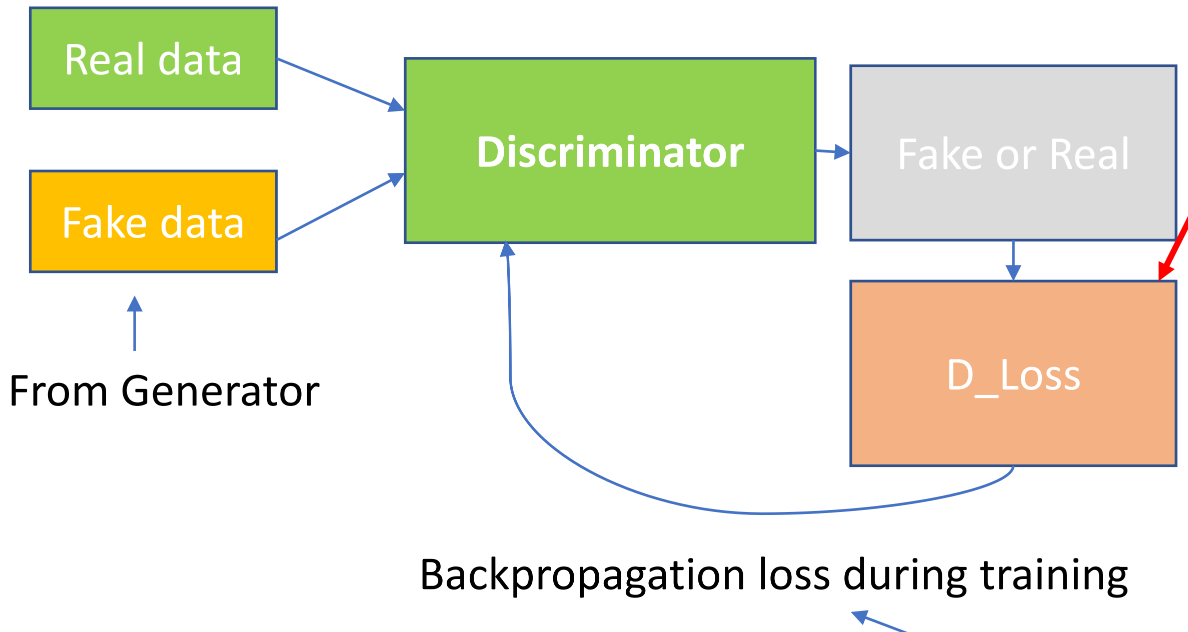
The discriminator neural network in a GAN architecture tries to **differentiate between the real data** and the **data generated by a generator.** In fact, we can say that the discriminator is a binary classifier that classifies between **positive and negative examples.**



Training the discriminator in a Generative Adversarial Network.

Generative Adversarial Networks (GAN)

Training the Discriminator



The discriminator trains, the generator does not train. While training the discriminator, **we only backpropagate using the discriminator loss values**. **The loss values of the generator are frozen when the discriminator trains**.

How training the discriminator helps the generator?

- If the discriminator misclassifies the data, then it is penalized.
- **Misclassification** can happen in two ways. Either when the discriminator classifies the real data as fake or when it classifies the fake data as real.

- **The backpropagation in discriminator happens through the discriminator network.**

The updated gradients from the discriminator training are not passed to the generator.

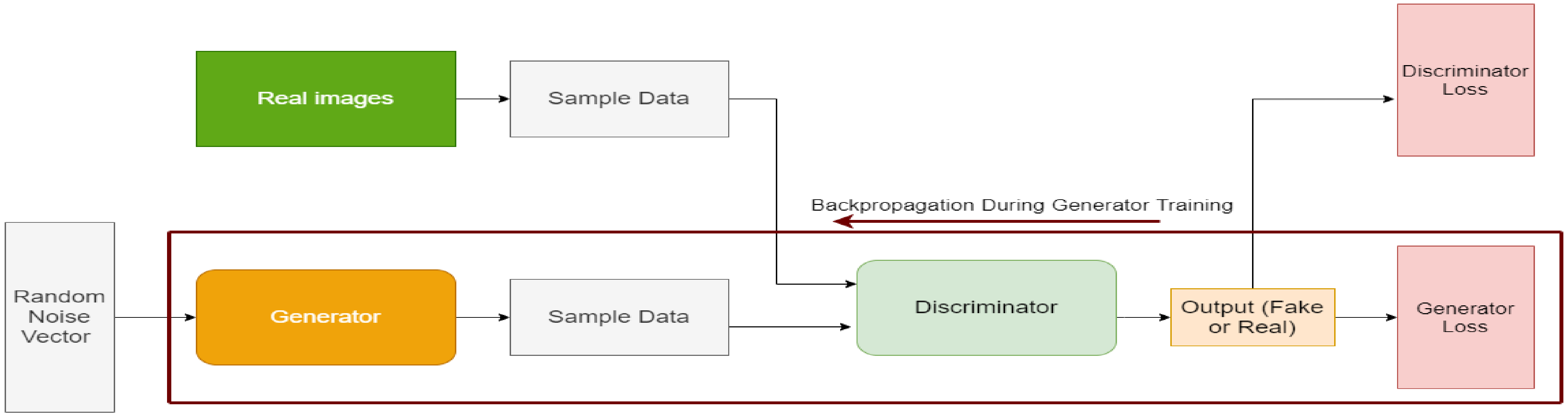
Generative Adversarial Networks (GAN)

The Generator in GAN

The generator neural network tries to **generate new data from the random noise (latent vector).**

With each iteration, it tries to produce more and more realistic data. These generated data also **act as negative examples for the discriminator training**. **Also, the fake data generation process is linked to the feedback that the generator gets from the discriminator.**

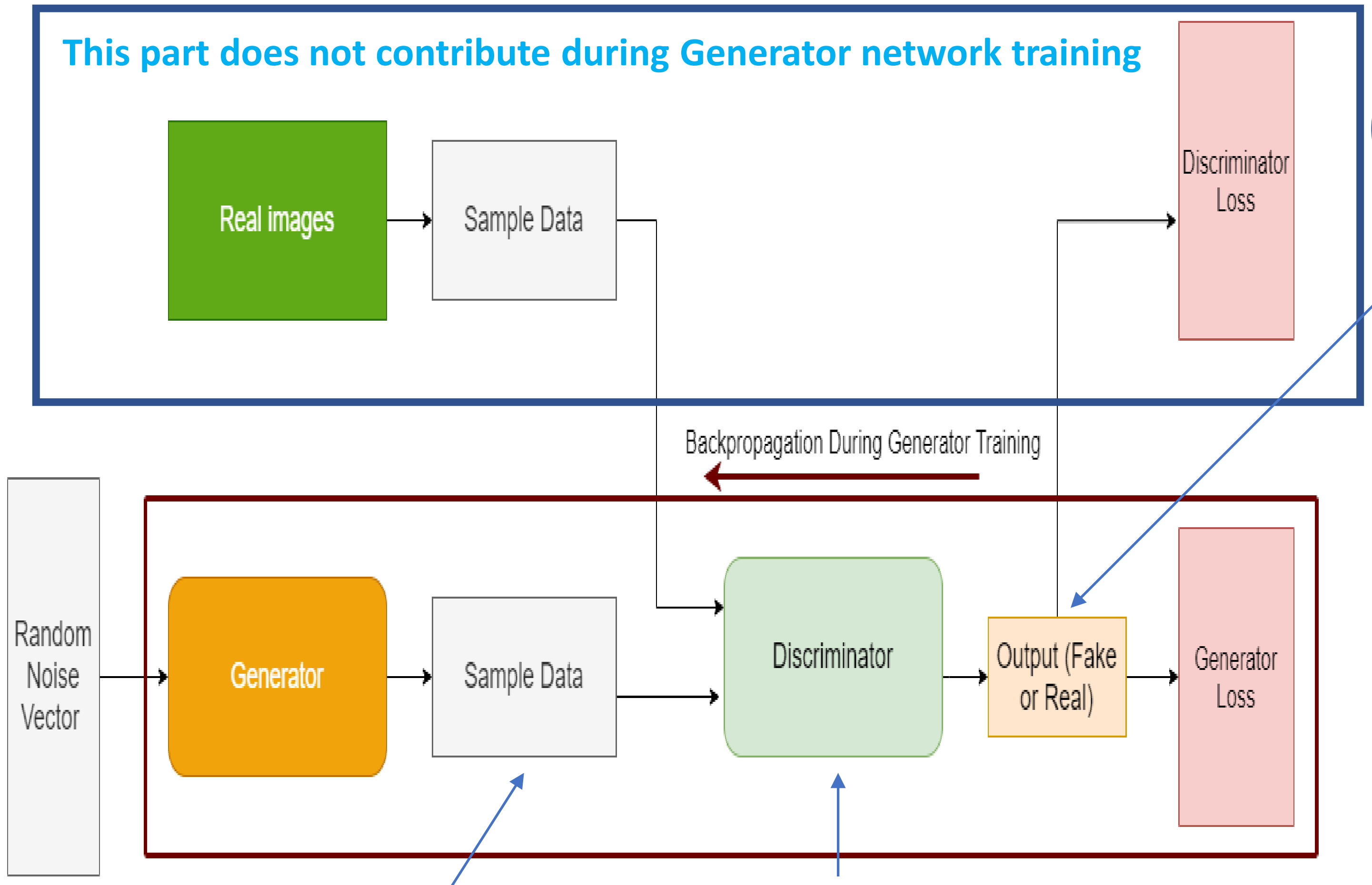
As training progresses, the generator will eventually be able to **fool the discriminator into classifying the fake data as real data.**



The generator training happens.

Generative Adversarial Networks (GAN)

The Generator in GAN



Step 3.

The output from the discriminator which acts as a feedback to the generator. The discriminator feedback is used to **penalize the generator for producing bad data instances.** **Also, we have the generator loss from the generator outputs. This generator loss penalizes the generator itself if it is not able to fool the discriminator.**

Fake data

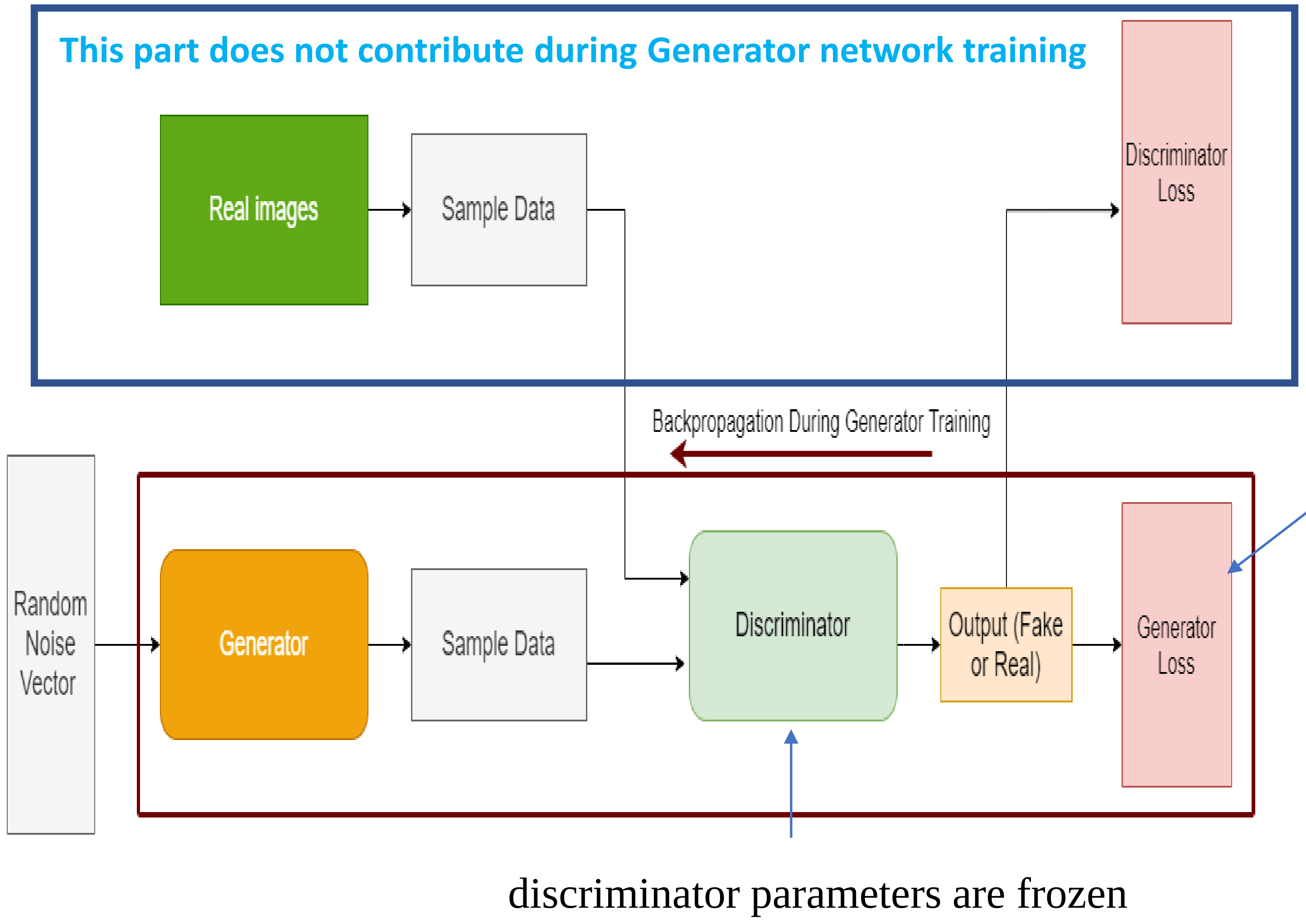
Step 1.

Step 2.

At this step, the discriminator tries to classify the generated data as fake.

Generative Adversarial Networks (GAN)

The Generator in GAN


This part does not contribute during Generator network training
Real images
Sample Data
Discriminator Loss
Backpropagation During Generator Training
Random Noise Vector
Generator
Sample Data
Discriminator
Output (Fake or Real)
Generator Loss
discriminator parameters are frozen

Step 4.

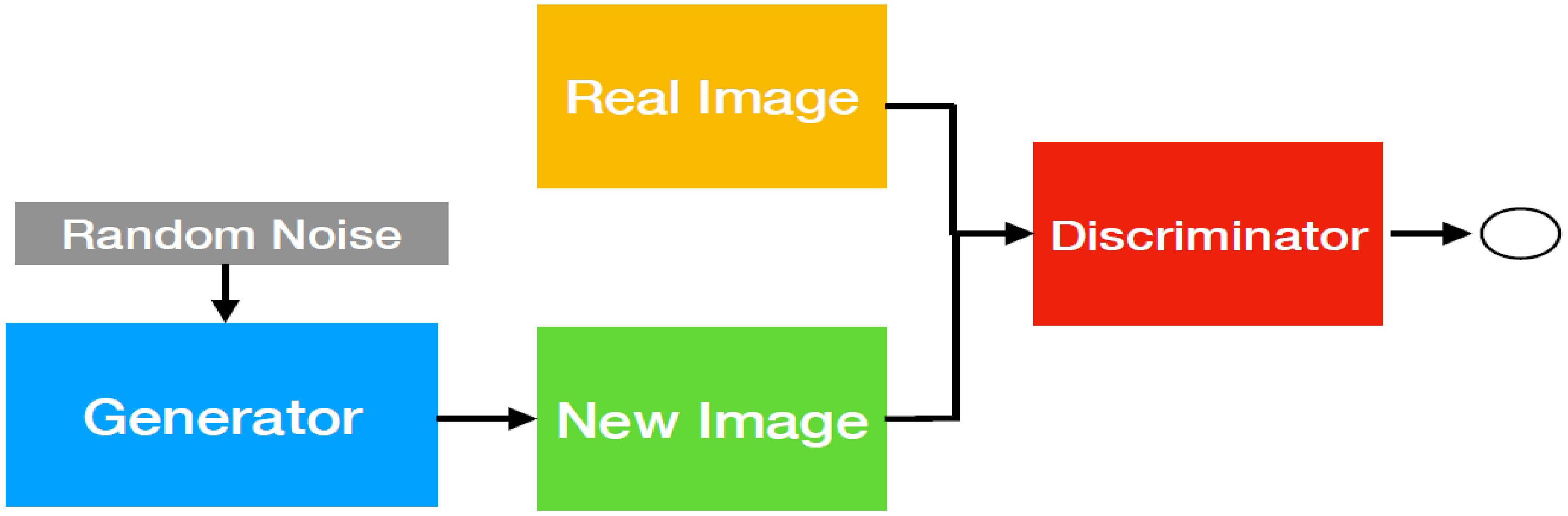
You can see that after we get the **generator loss, the backpropagation happens both through the discriminator and the generator**. We know that while training the generator and updating the weights we need to backpropagate through the generator.

But why backpropagate through the discriminator also?

Although the discriminator parameters are frozen, still we need feedback about the performance of the generator. This we get by backpropagating through the discriminator also. The discriminator parameters do not get updated in this step but we get feedback about the generator.

Generative Adversarial Networks (GAN)

When Does a GAN Converge?



Generative Adversarial Networks (GAN)

GAN Objective

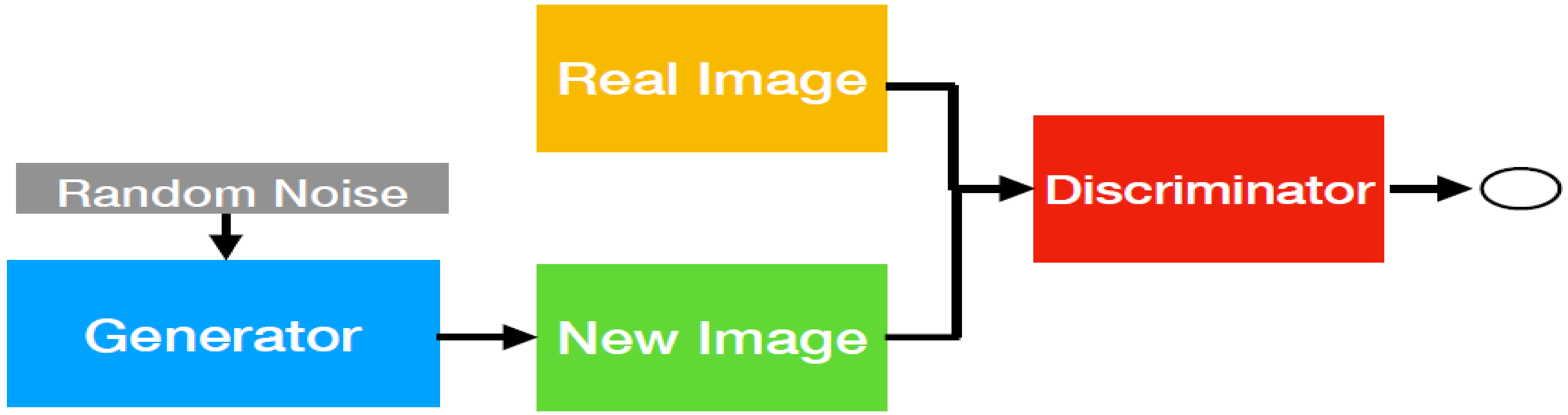
$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Generative Adversarial Networks (GAN)

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-D(G(\boldsymbol{z})))]$$

<u>Discriminator gradient for update (gradient ascent):</u>

$$\nabla_{\mathbf{w}_D} \frac{1}{n}\sum_{i=1}^{n}\left[\overbrace{\log D\left(\boldsymbol{x}^{(i)}\right)}^{\substack{\text{predict well on real images}\\ \text{=> probability close to 1}}} + \overbrace{\log\left(1-D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)}^{\substack{\text{predict well on fake images}\\ \text{=> probability close to 0}}}\right]$$

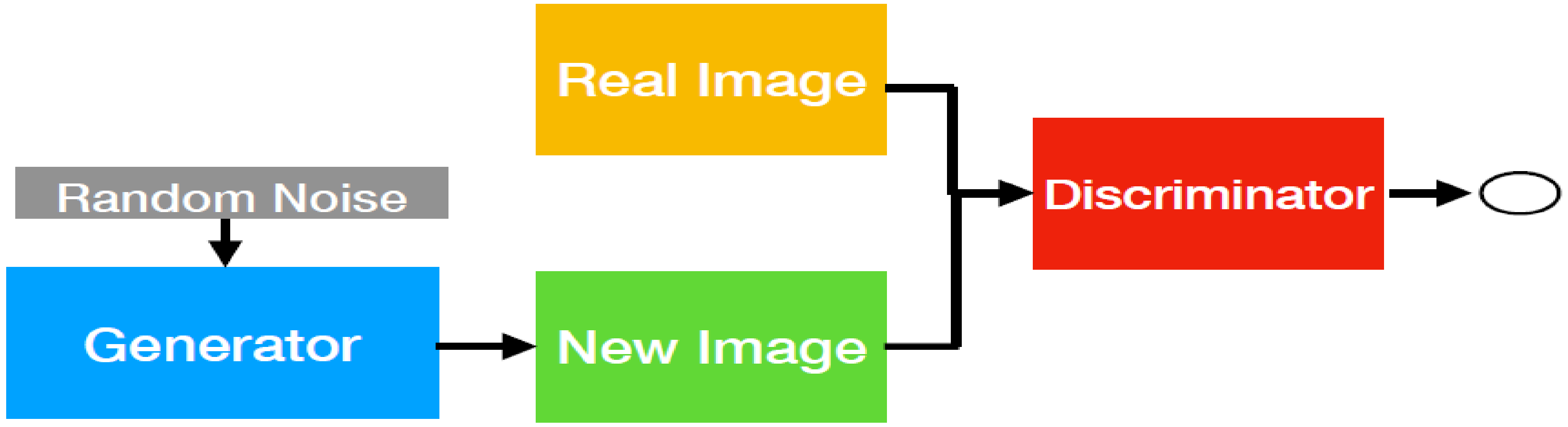


Generative Adversarial Networks (GAN)

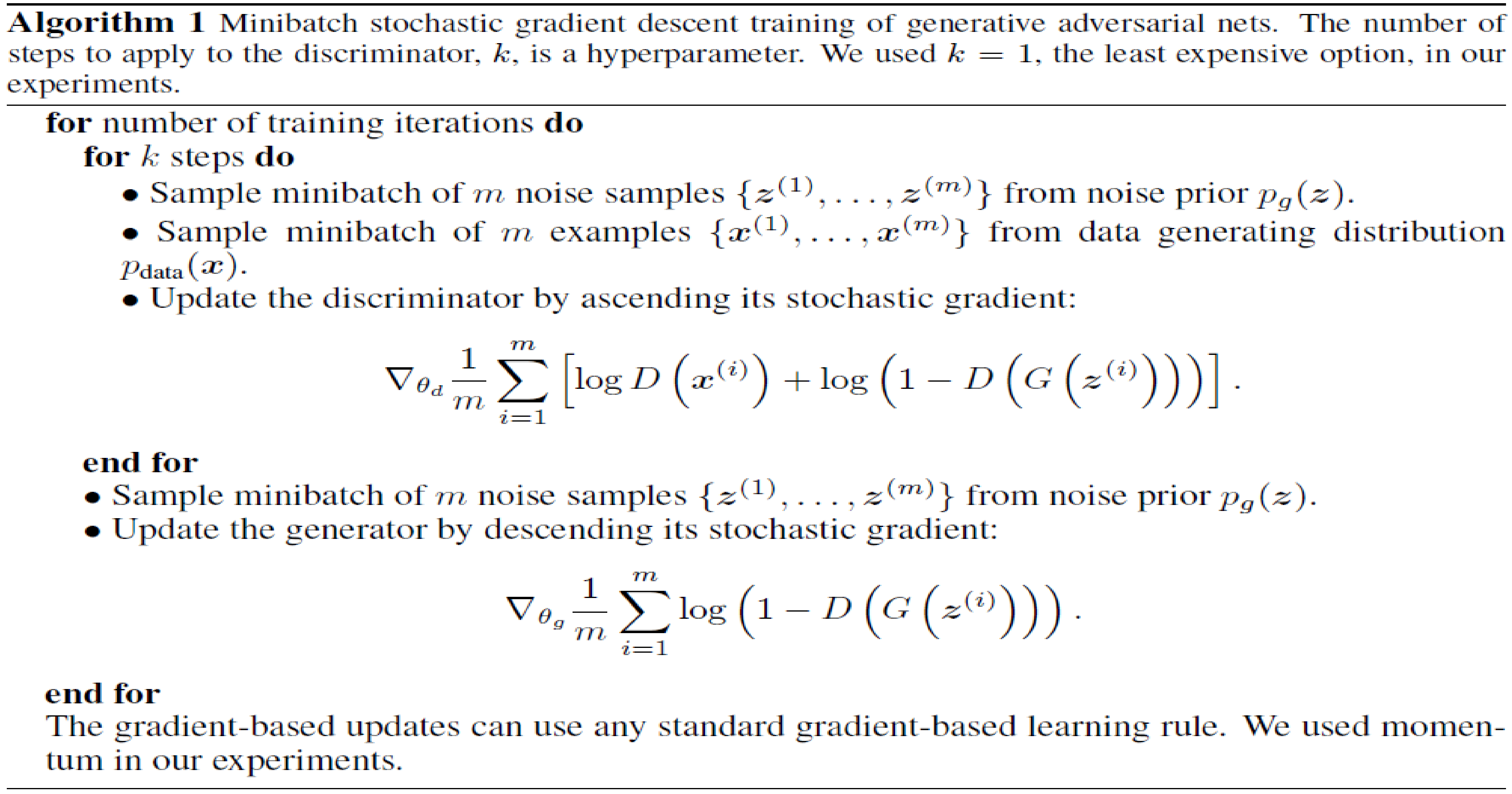
$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

<u>Generator gradient for update (gradient descent):</u>

$$\nabla_{\mathbf{w}_G} \frac{1}{n}\sum_{i=1}^{n} \log\left(1 - \overbrace{D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)}^{\substack{\text{predict badly on fake images}\\ \text{=> probability close to 1}}}\right)$$



Generative Adversarial Networks (GAN)



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 $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
        • Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
        • Update the discriminator by ascending its stochastic gradient:
            $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right].$$
    end for
    • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:
        $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$
end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

- Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets." In *Advances in Neural Information Processing Systems*, pp. 2672-2680. 2014.

Generative Adversarial Networks (GAN)

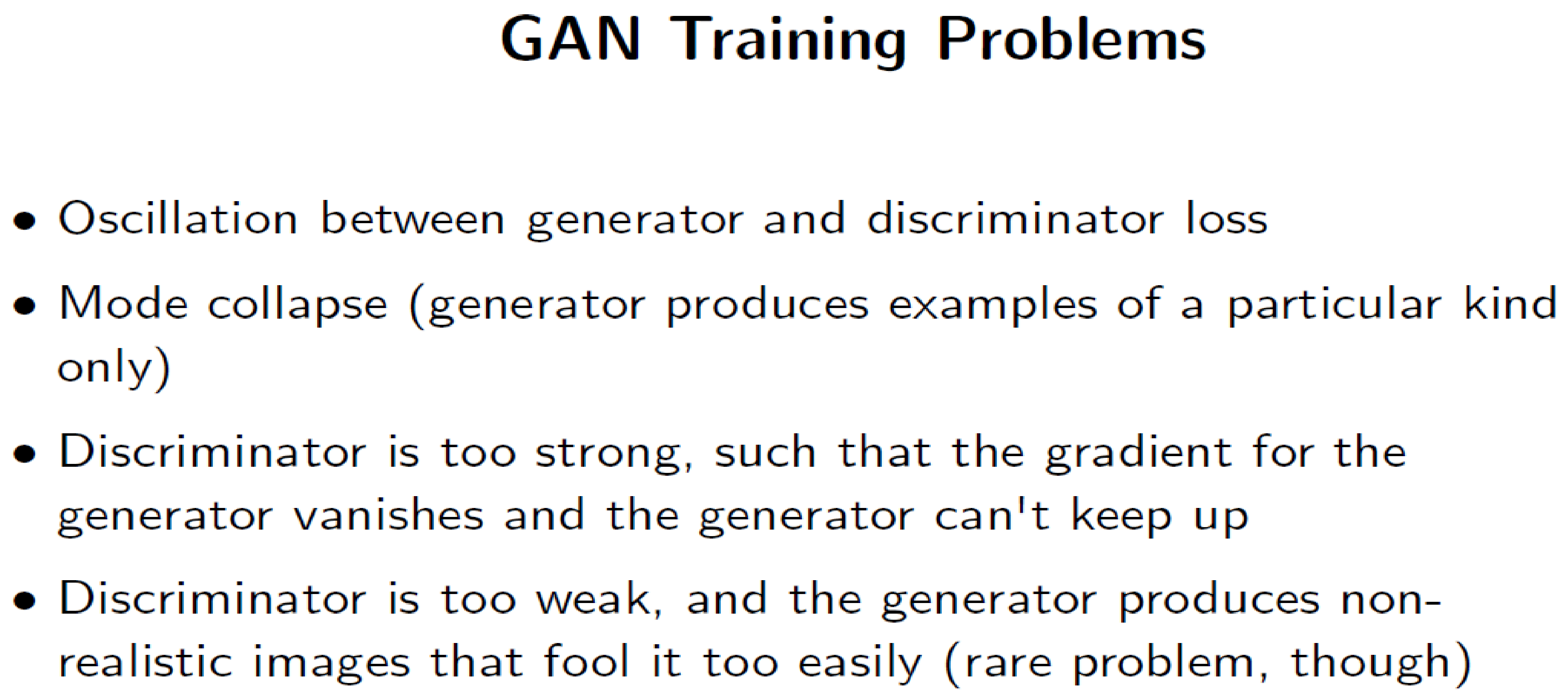
GAN Convergence

- Converges when Nash-equilibrium (Game Theory concept) is reached in the minmax (zero-sum) game

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

- Nash-Equilibrium in Game Theory is reached when the actions of one player won't change depending on the opponent's actions
- Here, this means that the GAN produces realistic images and the discriminator outputs random predictions (probabilities close to 0.5)

Generative Adversarial Networks (GAN)



GAN Training Problems

- Oscillation between generator and discriminator loss
- Mode collapse (generator produces examples of a particular kind only)
- Discriminator is too strong, such that the gradient for the generator vanishes and the generator can't keep up
- Discriminator is too weak, and the generator produces non-realistic images that fool it too easily (rare problem, though)

Generative Adversarial Networks (GAN)

GAN Training Problems

- Discriminator is too strong, such that the gradient for the generator vanishes and the generator can't keep up
- Can be fixed as follows:

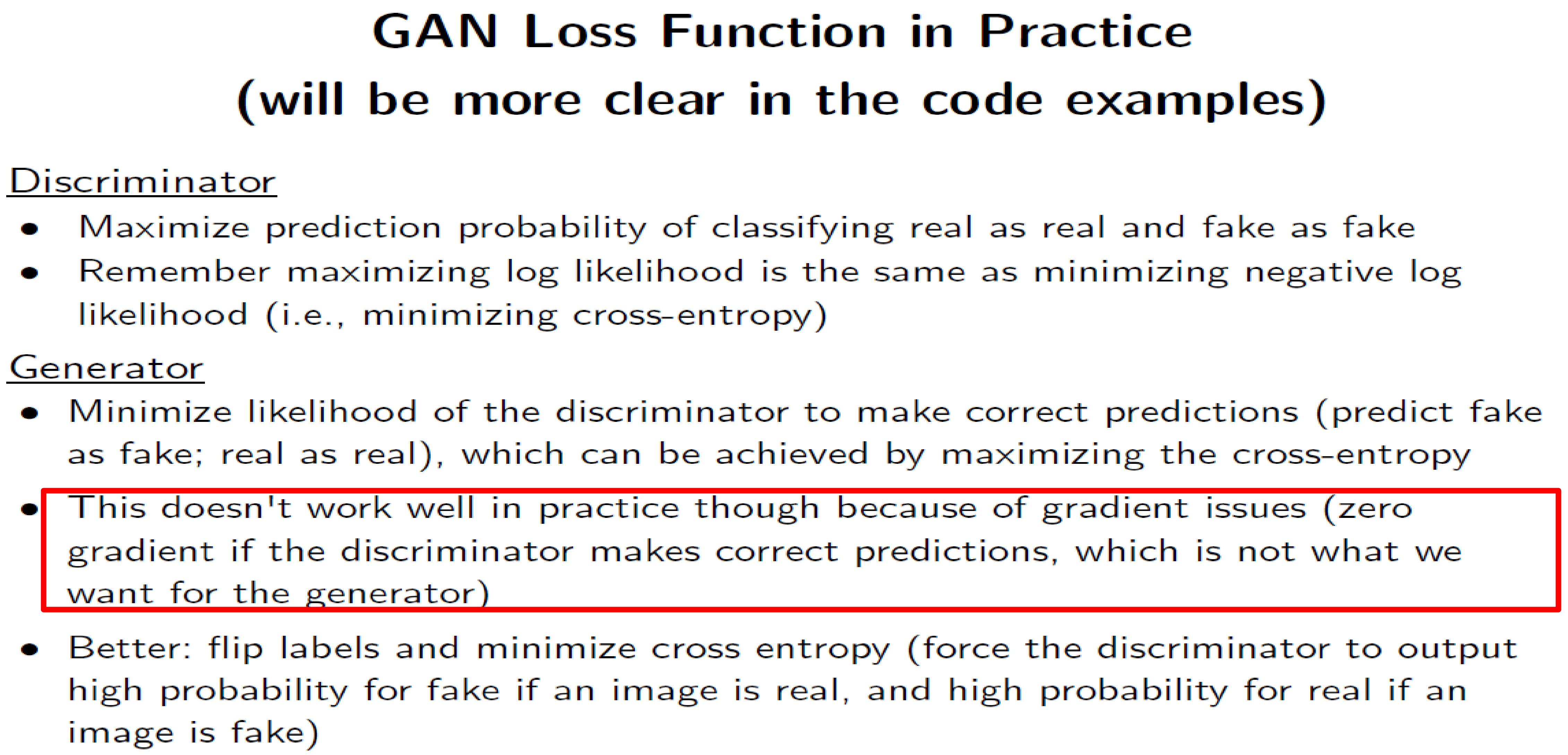
Instead of gradient descent with

$$\nabla_{\mathbf{w}_G} \frac{1}{n} \sum_{i=1}^{n} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

Do gradient ascent with

$$\nabla_{\mathbf{w}_G} \frac{1}{n} \sum_{i=1}^{n} \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$$

Generative Adversarial Networks (GAN)



GAN Loss Function in Practice
(will be more clear in the code examples)

<u>Discriminator</u>

- Maximize prediction probability of classifying real as real and fake as fake
- Remember maximizing log likelihood is the same as minimizing negative log likelihood (i.e., minimizing cross-entropy)

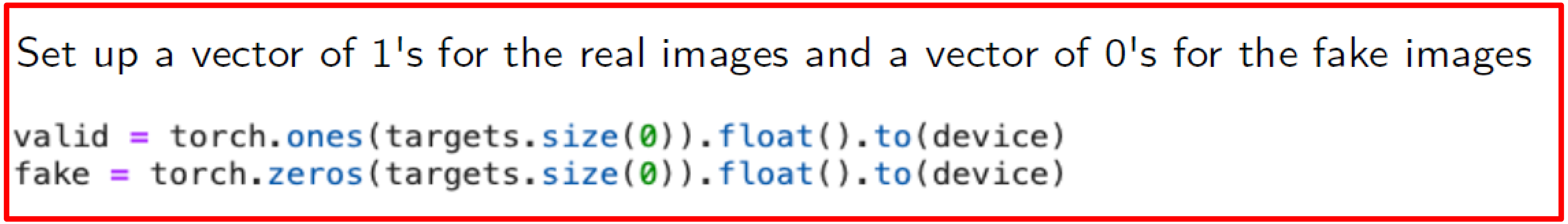
<u>Generator</u>

- Minimize likelihood of the discriminator to make correct predictions (predict fake as fake; real as real), which can be achieved by maximizing the cross-entropy
- This doesn't work well in practice though because of gradient issues (zero gradient if the discriminator makes correct predictions, which is not what we want for the generator)
- Better: flip labels and minimize cross entropy (force the discriminator to output high probability for fake if an image is real, and high probability for real if an image is fake)

Generative Adversarial Networks (GAN)

GANs In Practice (1)

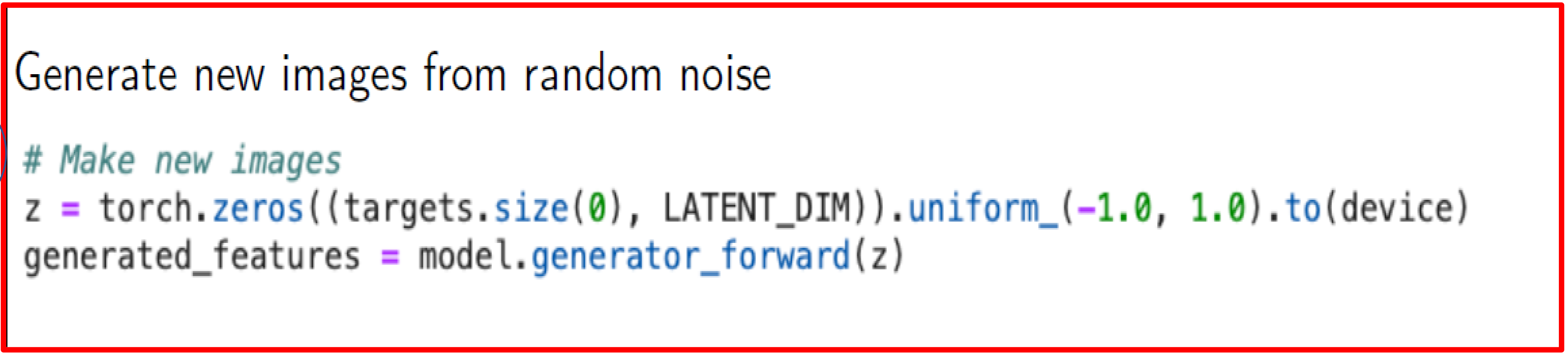
Step1



Set up a vector of 1's for the real images and a vector of 0's for the fake images

```
valid = torch.ones(targets.size(0)).float().to(device)
fake = torch.zeros(targets.size(0)).float().to(device)
```

Step2

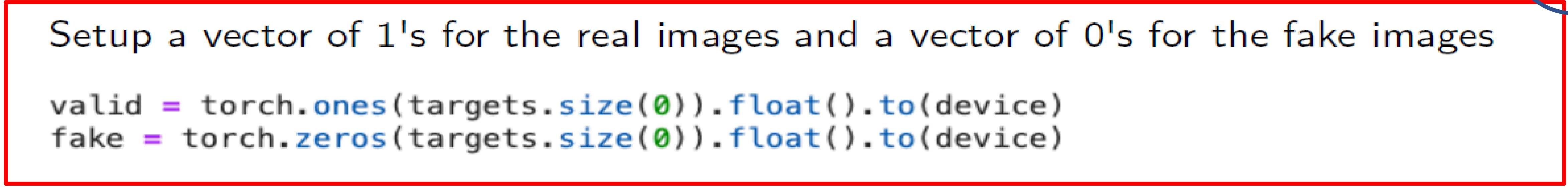


Generate new images from random noise

```
# Make new images
z = torch.zeros((targets.size(0), LATENT_DIM)).uniform_(-1.0, 1.0).to(device)
generated_features = model.generator_forward(z)
```

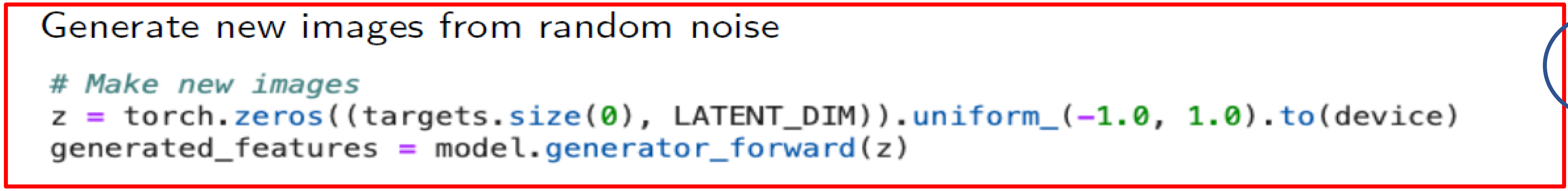
Generative Adversarial Networks (GAN)

Step1


Setup a vector of 1's for the real images and a vector of 0's for the fake images

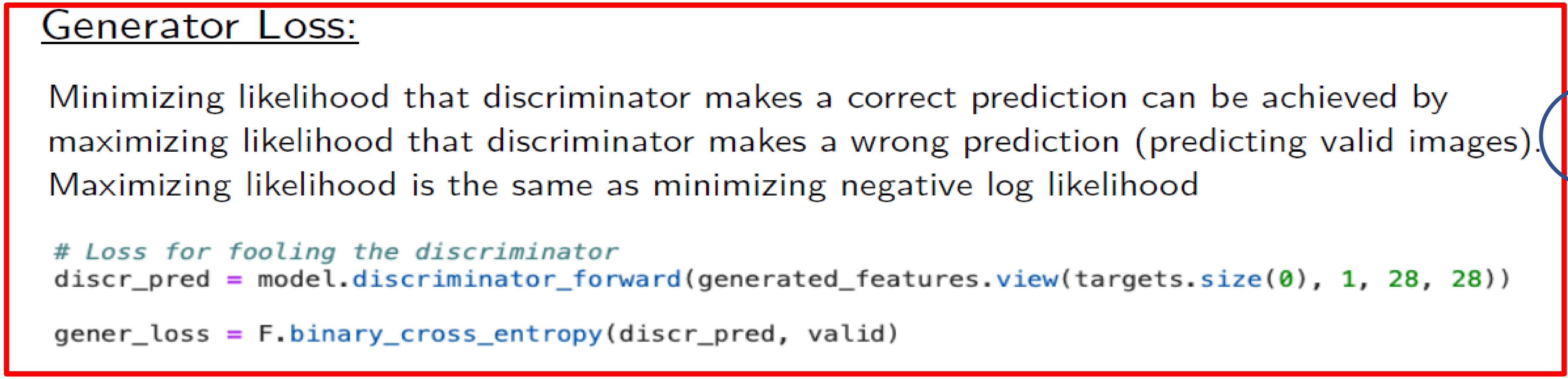
```
valid = torch.ones(targets.size(0)).float().to(device)
fake = torch.zeros(targets.size(0)).float().to(device)
```

Step2


Generate new images from random noise

```
# Make new images
z = torch.zeros((targets.size(0), LATENT_DIM)).uniform_(-1.0, 1.0).to(device)
generated_features = model.generator_forward(z)
```

Step3


Generator Loss:

Minimizing likelihood that discriminator makes a correct prediction can be achieved by maximizing likelihood that discriminator makes a wrong prediction (predicting valid images). Maximizing likelihood is the same as minimizing negative log likelihood

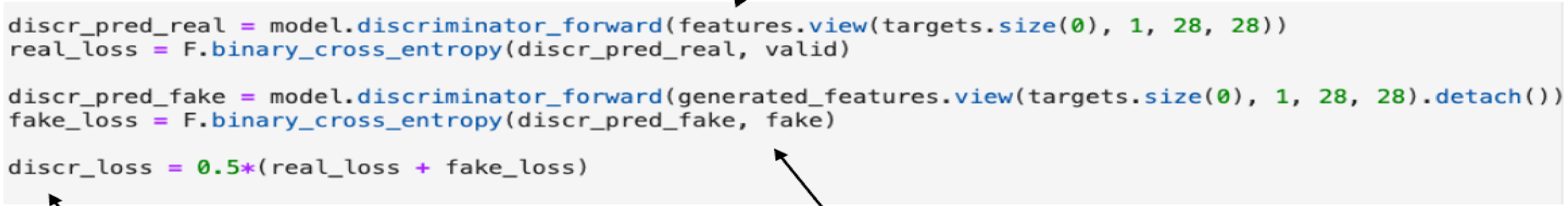
```
# Loss for fooling the discriminator
discr_pred = model.discriminator_forward(generated_features.view(targets.size(0), 1, 28, 28))

gener_loss = F.binary_cross_entropy(discr_pred, valid)
```

Generative Adversarial Networks (GAN)

Discriminator Loss:

Step4

Train discriminator to recognize real images as real



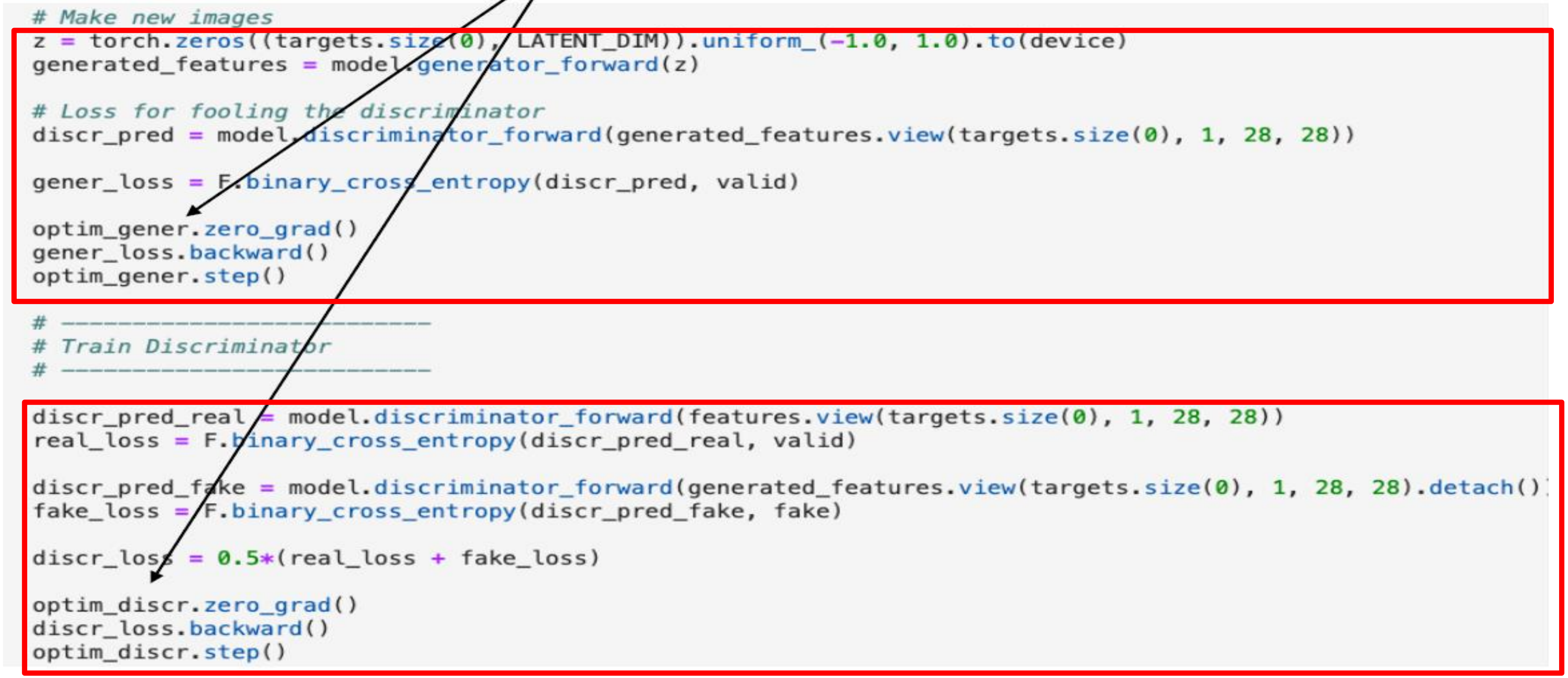
```
discr_pred_real = model.discriminator_forward(features.view(targets.size(0), 1, 28, 28))
real_loss = F.binary_cross_entropy(discr_pred_real, valid)

discr_pred_fake = model.discriminator_forward(generated_features.view(targets.size(0), 1, 28, 28).detach())
fake_loss = F.binary_cross_entropy(discr_pred_fake, fake)

discr_loss = 0.5*(real_loss + fake_loss)
```

Train discriminator to recognize generated images as fake

Combine fake & real parts

Generative Adversarial Networks (GAN)

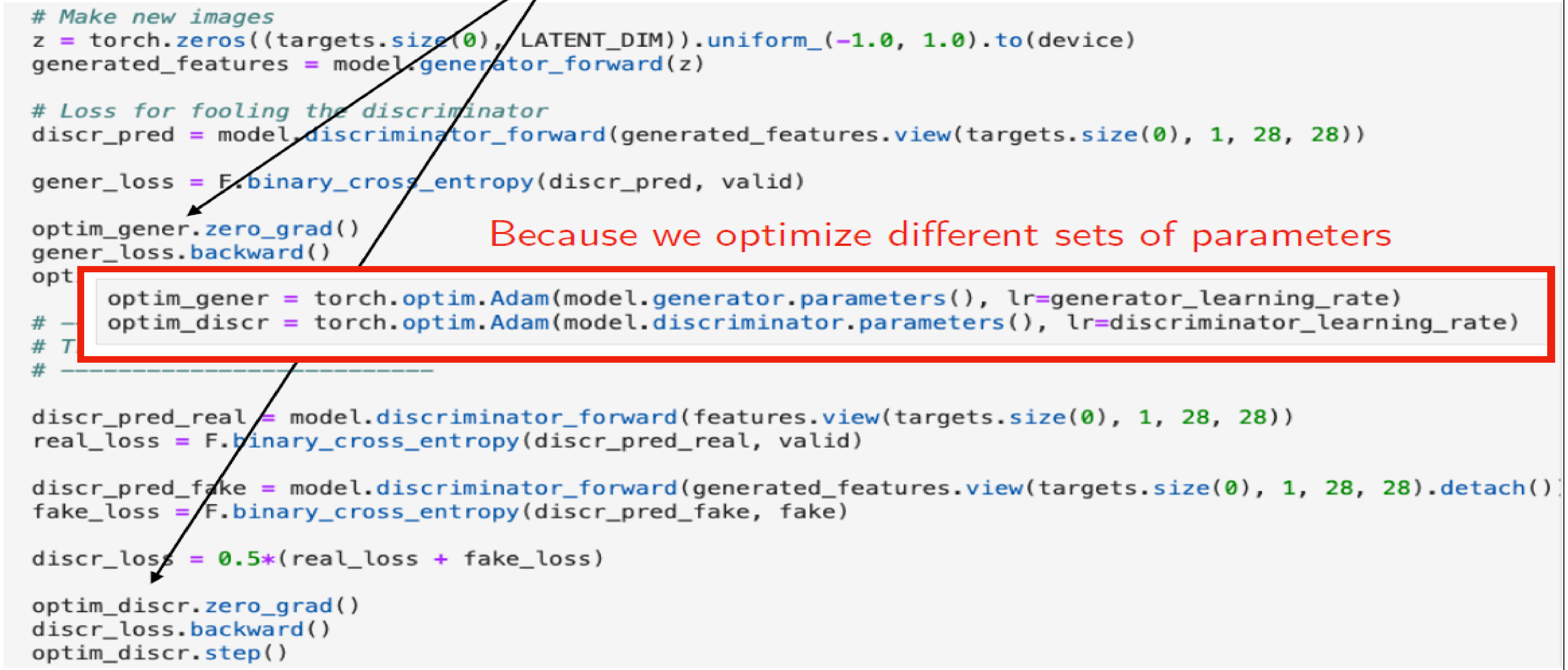
Use separate optimizers for generator and discriminator



```
# Make new images
z = torch.zeros((targets.size(0), LATENT_DIM)).uniform_(-1.0, 1.0).to(device)
generated_features = model.generator_forward(z)

# Loss for fooling the discriminator
discr_pred = model.discriminator_forward(generated_features.view(targets.size(0), 1, 28, 28))

gener_loss = F.binary_cross_entropy(discr_pred, valid)

optim_gener.zero_grad()
gener_loss.backward()
optim_gener.step()

# --------------------------
# Train Discriminator
# --------------------------

discr_pred_real = model.discriminator_forward(features.view(targets.size(0), 1, 28, 28))
real_loss = F.binary_cross_entropy(discr_pred_real, valid)

discr_pred_fake = model.discriminator_forward(generated_features.view(targets.size(0), 1, 28, 28).detach()
fake_loss = F.binary_cross_entropy(discr_pred_fake, fake)

discr_loss = 0.5*(real_loss + fake_loss)

optim_discr.zero_grad()
discr_loss.backward()
optim_discr.step()
```

Generative Adversarial Networks (GAN)

Use separate optimizers for generator and discriminator



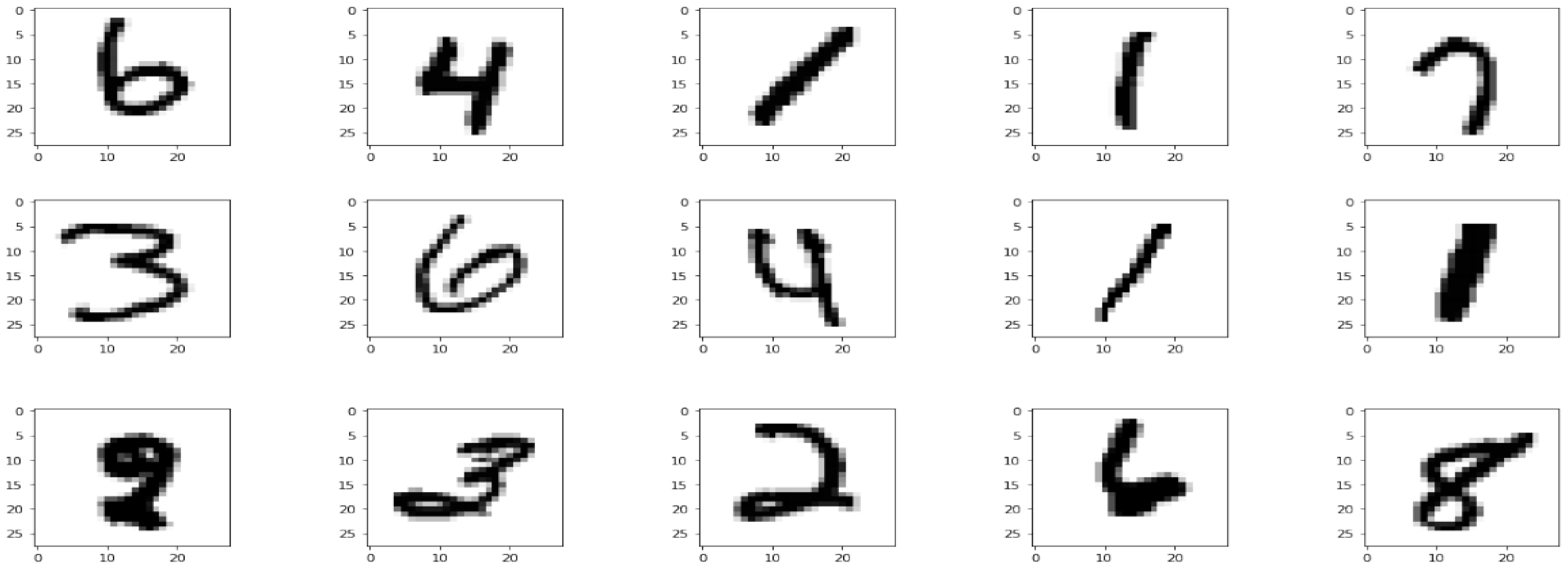
```
# Make new images
z = torch.zeros((targets.size(0), LATENT_DIM)).uniform_(-1.0, 1.0).to(device)
generated_features = model.generator_forward(z)

# Loss for fooling the discriminator
discr_pred = model.discriminator_forward(generated_features.view(targets.size(0), 1, 28, 28))

gener_loss = F.binary_cross_entropy(discr_pred, valid)

optim_gener.zero_grad()
gener_loss.backward()
opt

# -
# T
# -------------------------

discr_pred_real = model.discriminator_forward(features.view(targets.size(0), 1, 28, 28))
real_loss = F.binary_cross_entropy(discr_pred_real, valid)

discr_pred_fake = model.discriminator_forward(generated_features.view(targets.size(0), 1, 28, 28).detach()
fake_loss = F.binary_cross_entropy(discr_pred_fake, fake)

discr_loss = 0.5*(real_loss + fake_loss)

optim_discr.zero_grad()
discr_loss.backward()
optim_discr.step()
```

Because we optimize different sets of parameters

```
optim_gener = torch.optim.Adam(model.generator.parameters(), lr=generator_learning_rate)
optim_discr = torch.optim.Adam(model.discriminator.parameters(), lr=discriminator_learning_rate)
```

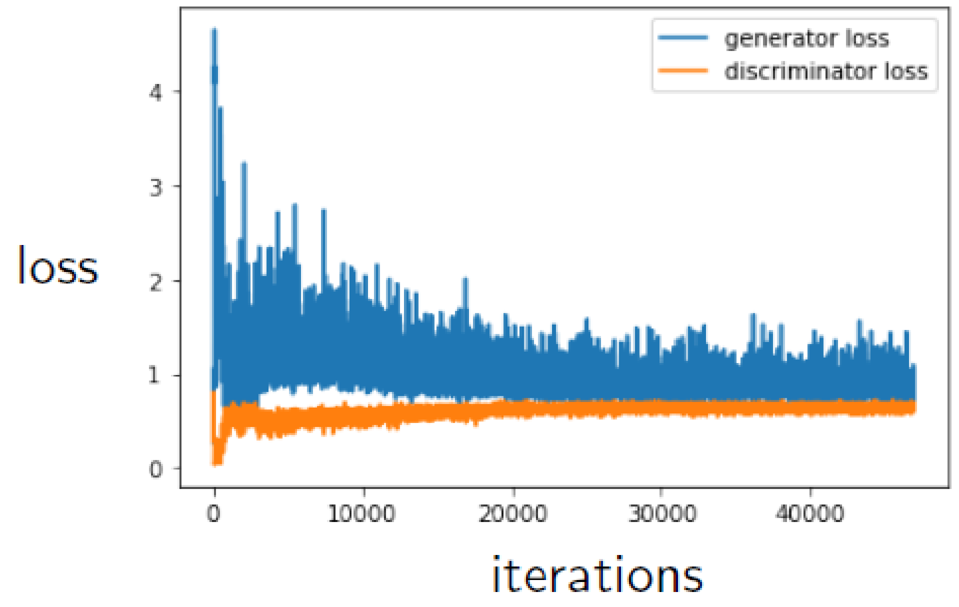
Generative Adversarial Networks (GAN)

For Reference: Some Real MNIST Images

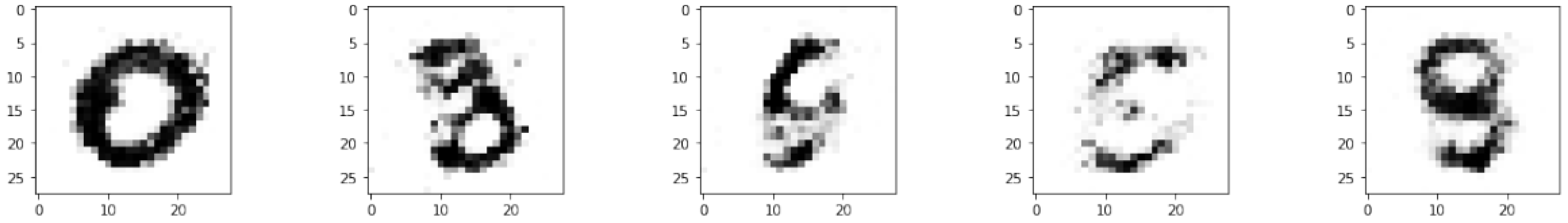


Generative Adversarial Networks (GAN)

MLP GAN to generate MNIST

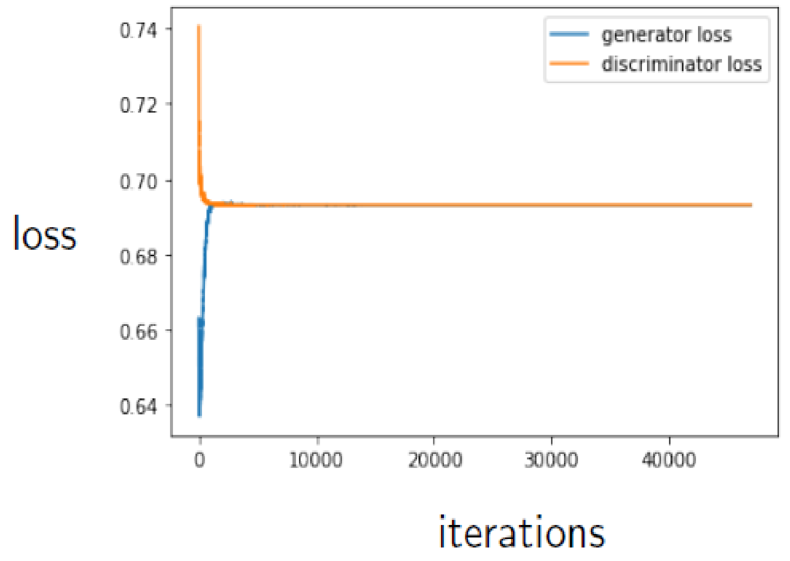


Generated images

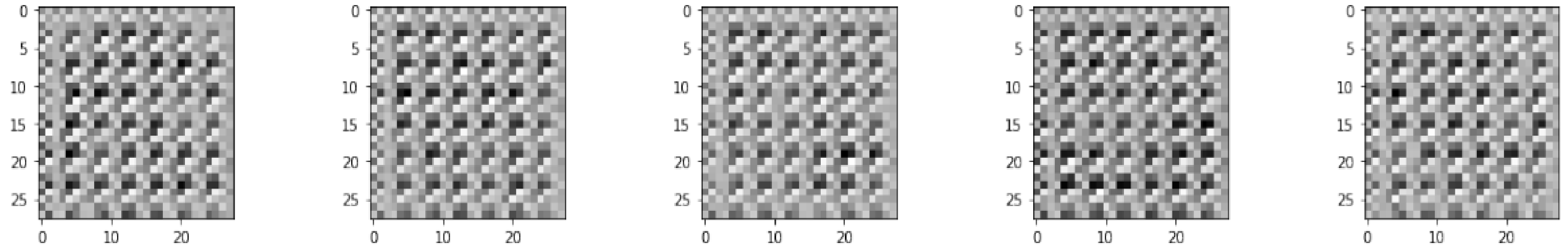


Generative Adversarial Networks (GAN)

Failed Convolutional GAN to generate the MNIST images

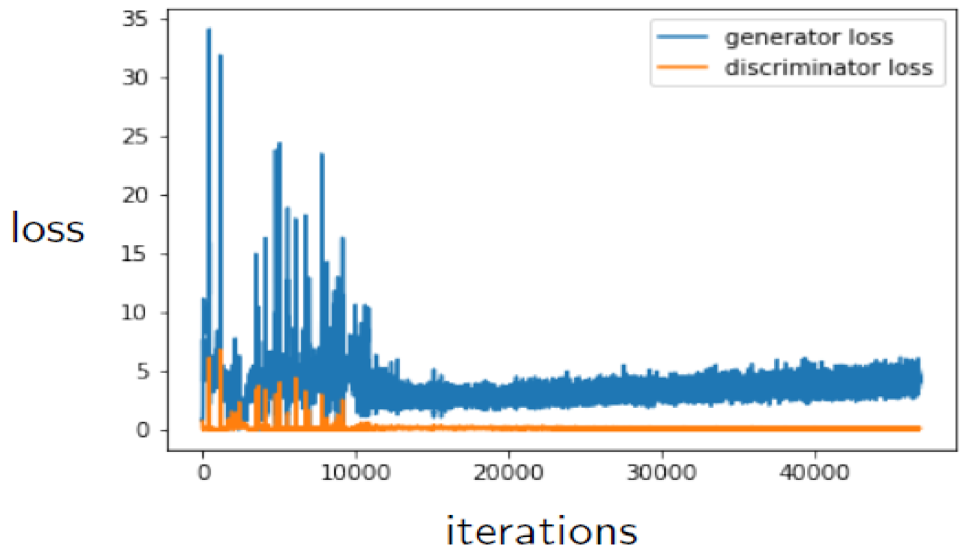


Generated images

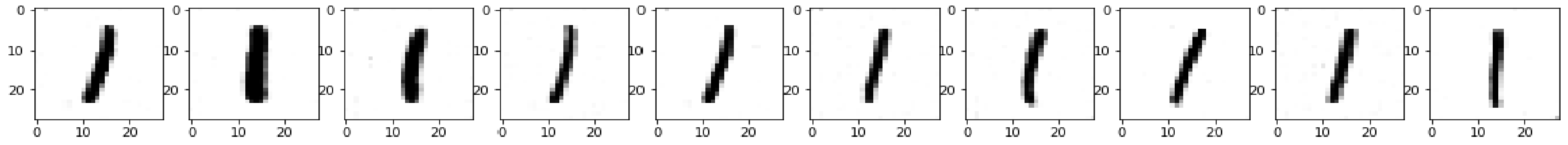


Generative Adversarial Networks (GAN)

MNIST Convolutional GAN with model collapse

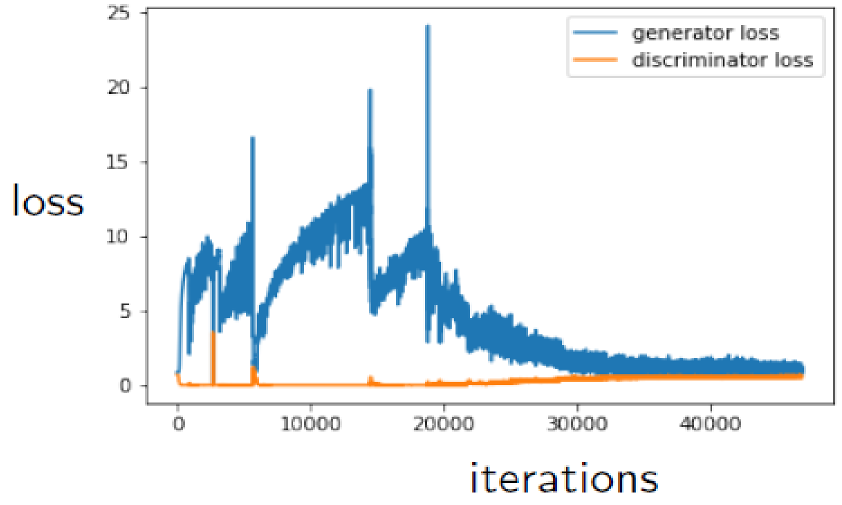


Generated images

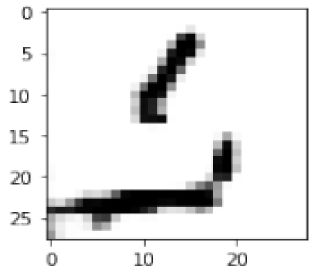


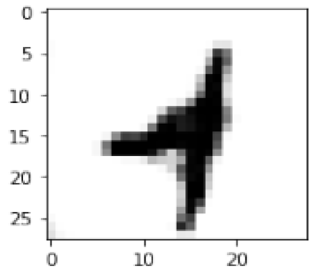
Generative Adversarial Networks (GAN)

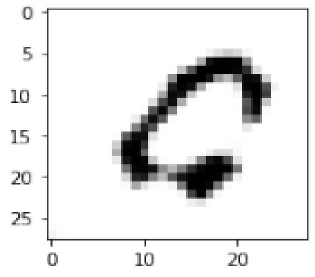
MNIST Convolutional GAN – not great results

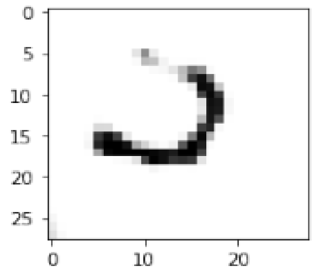


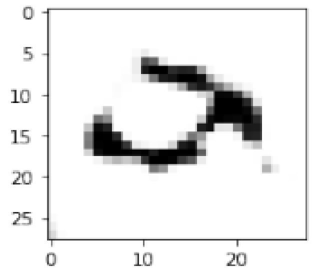
Generated images





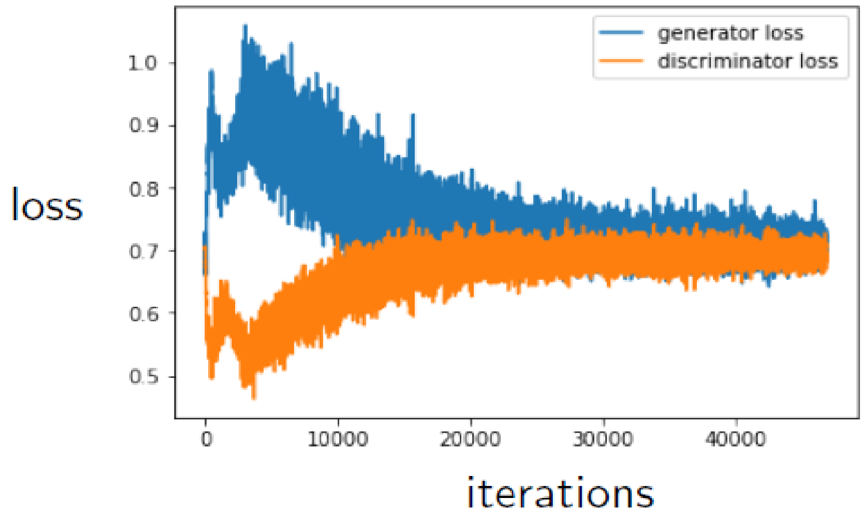






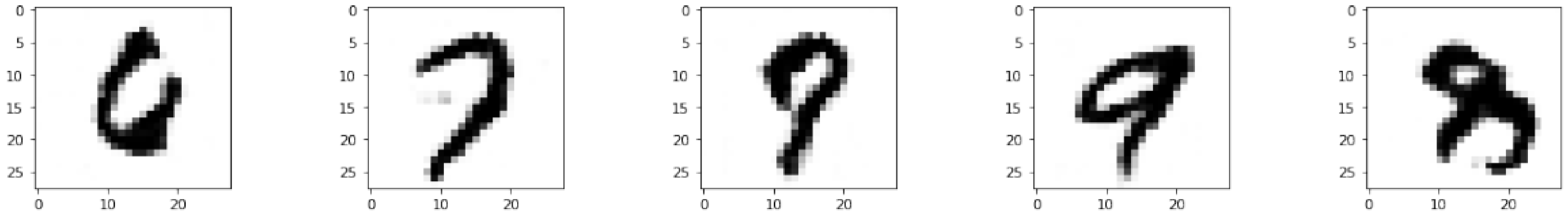
Generative Adversarial Networks (GAN)

MNIST Convolutional GAN – relatively good



Compared to the previous slide,
this has BatchNorm

Generated images

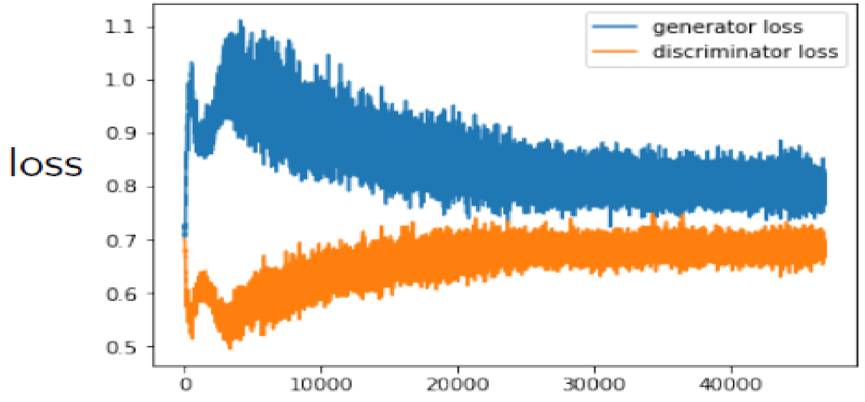


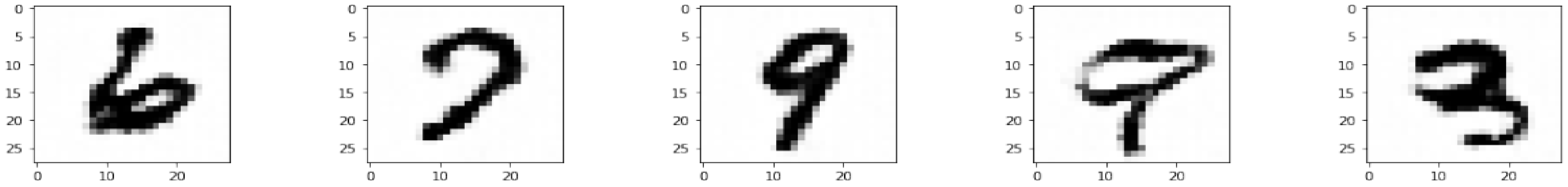
Generative Adversarial Networks (GAN)

MNIST Convolutional GAN – relatively good

Like previous slide but with label smoothing: Replace real images (1's) by 0.9 based on idea in

- Salimans, Tim, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. "Improved techniques for training GANs." In *Advances in Neural Information Processing Systems* pp. 2234-2242. 2016.





Generative Adversarial Networks (GAN)

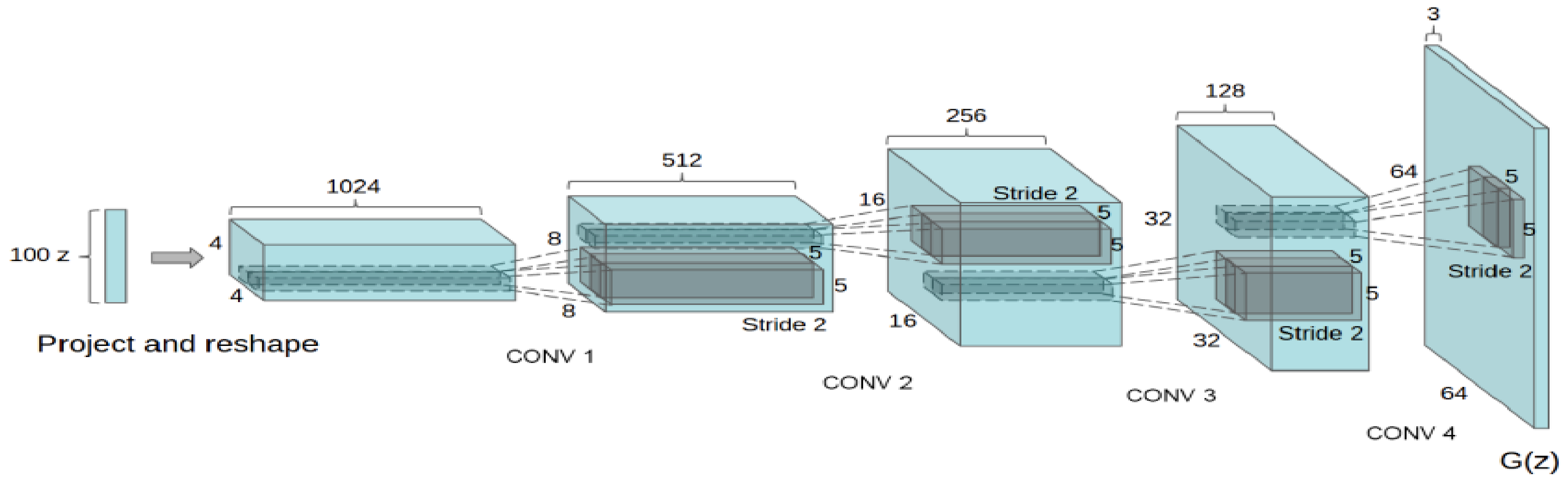
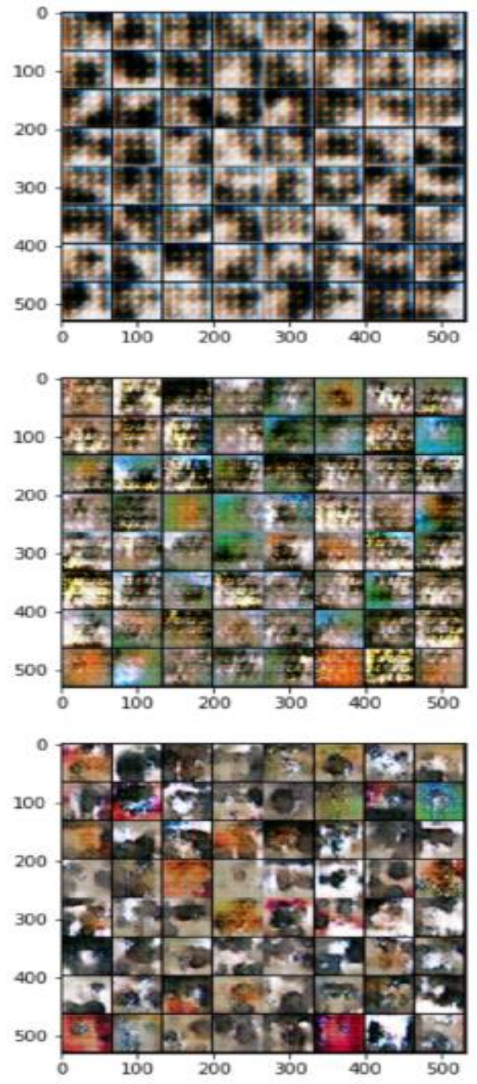


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434.

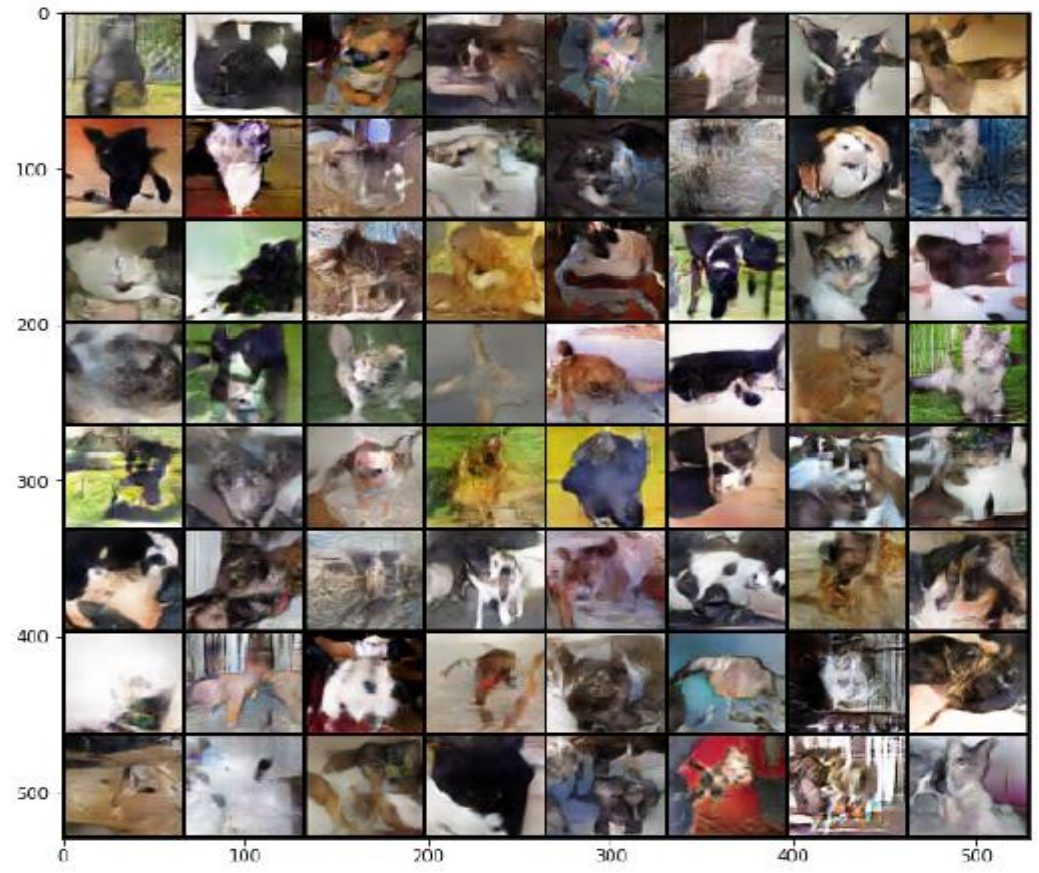
Generative Adversarial Networks (GAN)

Cats vs Dogs



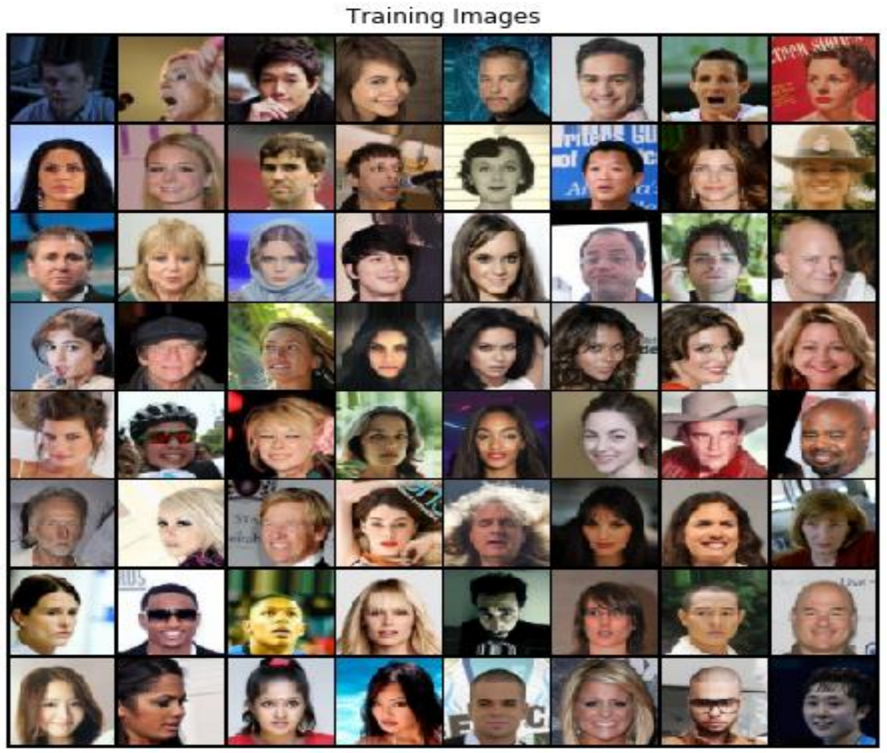
Generative Adversarial Networks (GAN)

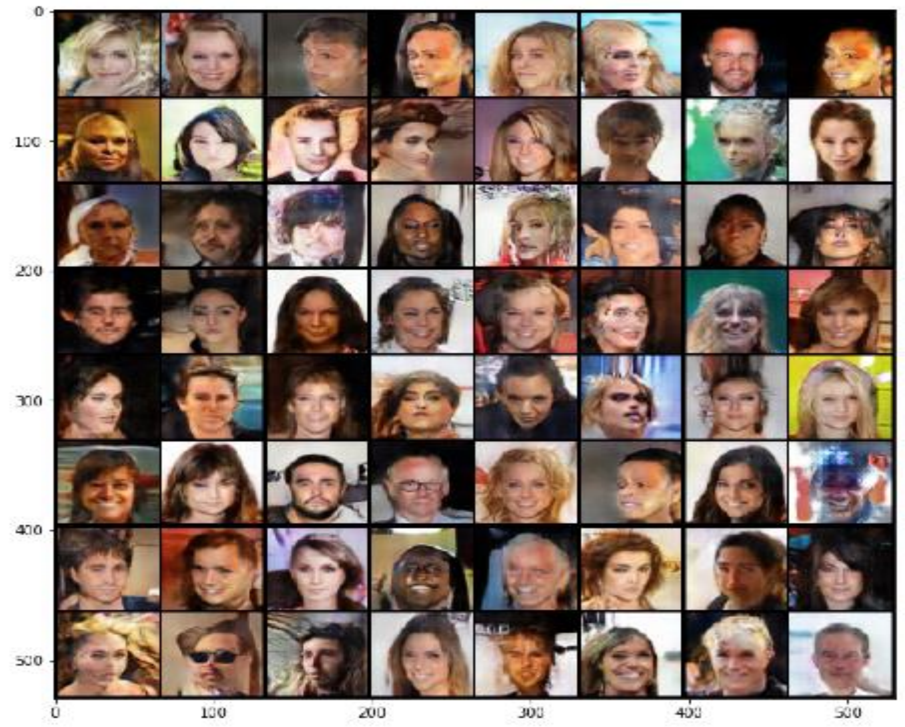
Cats vs Dogs



Generative Adversarial Networks (GAN)

CelebA Face Images





Generative Adversarial Networks (GAN)

Loss Function and Training of GANs
By now, we know that we need to train two neural networks to complete the GAN training.

The generator and the discriminator train alternately. While the discriminator trains, the generator does not. Conversely, when the generator trains, the discriminator does not.

Also, training of GANs is a very difficult process and often not stable.
There are many pitfalls and problems that we can face while trying to train GAN perfectly.
We will get into those points.
For now, we will focus on the loss function that is used to train GANs.

Loss Function in GANs
In this section, we will talk about the ***minimax loss*** that is described in the original paper on GANs.
First of all, let's take a look at the loss function.

$$L(\theta(G),\theta(D))=ExlogD(x)+Ez(1–D(G(z)))$$

You may recognize the above loss function as a form of the very common ***binary-cross entropy loss***. So, what do all the terms mean.

https://debuggercafe.com/introduction-to-generative-adversarial-networks-gans/

Generative Adversarial Networks (GAN)

Training the Discriminator

First, we get the real data and the real labels (real labels are all 1s). The length of the real label should be equal to the batch size.

Then we do a forward pass by feeding the real data to the discriminator neural network. This gives us the real outputs from the real data.

Calculate the discriminator loss for the real outputs and labels and backpropagate it.

Get the fake data using the noise vector and doing a forward pass through the generator. Get fake labels as well.

Using the fake data, do a forward pass through the discriminator. Calculate the loss using the fake data outputs and the fake labels.

Backpropagate the fake data loss. Then calculate the total discriminator loss by adding real data loss and fake data loss. Update the discriminator optimizer parameters.

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Training the Generator

- *For the generator training, first, get the fake data by doing a forward pass through the generator. Get the real labels (all 1s).*
- *Then do a forward pass through the discriminator using the fake data and the labels.*
- *Calculate the loss and backpropagate them.*
- *But this time, update the generator optimizer parameters.*

The last few steps may seem a bit confusing. Especially, ***"why do we need to forward pass the fake data through the discriminator to update the generator parameters?"*** This is because, the discriminator would tell how well the generator did while generating the fake data. Do take some time to think about this point. All of this will become even clearer while coding. So, hang on for a bit.

Generative Adversarial Networks (GAN)

Importing All the Required Modules and Libraries

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import imageio
import numpy as np
import matplotlib
from torchvision.utils import make_grid, save_image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from tqdm import tqdm
matplotlib.style.use('ggplot')
```

Defining the Learning Parameters

```
# learning parameters
batch_size = 512
epochs = 200
sample_size = 64 # fixed sample size
nz = 128 # latent vector size
k = 1 # number of steps to apply to the discriminator
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

Preparing the Dataset

```
transform = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.5,),(0.5,)),
])
to_pil_image = transforms.ToPILImage()
```

```
train_data = datasets.MNIST(
root='../input/data',
train=True,
download=True,
transform=transform
)
train_loader = DataLoader(train_data, batch_size=batch_size,
shuffle=True)
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

The Generator Neural Network
Let’s start with building the generator neural network.
It is going to be a very simple network with Linear layers, and LeakyReLU activations in-between.

```
class Generator(nn.Module):
def __init__(self, nz):
super(Generator, self).__init__()
self.nz = nz
self.main = nn.Sequential(
nn.Linear(self.nz, 256),
nn.LeakyReLU(0.2),
nn.Linear(256, 512),
nn.LeakyReLU(0.2),
nn.Linear(512, 1024),
nn.LeakyReLU(0.2),
nn.Linear(1024, 784),
nn.Tanh(),
)
def forward(self, x):
return self.main(x).view(-1, 1, 28, 28)
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

The Discriminator Neural Network
Here we will define the discriminator neural network.
Remember that the discriminator is a binary classifier. Therefore, we will have to take that into consideration while building the discriminator neural network.

```
class Discriminator(nn.Module):
def __init__(self):
super(Discriminator, self).__init__()
self.n_input = 784
self.main = nn.Sequential(
nn.Linear(self.n_input, 1024),
nn.LeakyReLU(0.2),
nn.Dropout(0.3),
nn.Linear(1024, 512),
nn.LeakyReLU(0.2),
nn.Dropout(0.3),
nn.Linear(512, 256),
nn.LeakyReLU(0.2),
nn.Dropout(0.3),
nn.Linear(256, 1),
nn.Sigmoid(),
)
def forward(self, x):
x = x.view(-1, 784)
return self.main(x)
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

Initialize the Neural Networks and Define the Optimizers

Before moving further, we need to initialize the generator and discriminator neural networks.

```
generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)
print('##### GENERATOR #####')
print(generator)
print('######################')
print('\n##### DISCRIMINATOR #####')
print(discriminator)
print('######################')
```

```
# optimizers
optim_g = optim.Adam(generator.parameters(), lr=0.0002)
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

We will also need to define the loss function here. We will use the Binary Cross Entropy Loss Function for this problem.

```
# loss function
criterion = nn.BCELoss()
```

While training the generator and the discriminator, we need to store the epoch-wise loss values for both the networks. We will define two lists for this task. We will also need to store the images that are generated by the generator after each epoch. For that also, we will use a list.

```
losses_g = [] # to store generator loss after each epoch
losses_d = [] # to store discriminator loss after each epoch
images = [] # to store images generatd by the generator
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

Defining Some Utility Functions

For training the GAN in this tutorial, **we need the real image data and the fake image data from the generator.**
To calculate the loss, we also need real labels and the fake labels. Those will have to be tensors whose size should be equal to the batch size.
Let's define two functions, which will create tensors of 1s (ones) and 0s (zeros) for us whose size will be equal to the batch size.

```
# to create real labels (1s)
def label_real(size):
data = torch.ones(size, 1)
return data.to(device)
# to create fake labels (0s)
def label_fake(size):
data = torch.zeros(size, 1)
return data.to(device)
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

Defining Some Utility Functions

For generating fake images, we need to provide the generator with a noise vector. The size of the noise vector should be equal to nz (128) that we have defined earlier. To create this noise vector, we can define a function called create_noise().

```
# function to create the noise vector
def create_noise(sample_size, nz):
return torch.randn(sample_size, nz).to(device)
```

The function create_noise() accepts two parameters, sample_size and nz. It will return a vector of random noise that we will feed into our generator to create the fake images.

There is one final utility function. We need to save the images generated by the generator after each epoch. Now, they are torch tensors. To save those easily, we can define a function which takes those batch of images and saves them in a grid-like structure.

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

Defining Some Utility Functions

There is one final utility function. We need to save the images generated by the generator after each epoch. Now, they are torch tensors. To save those easily, we can define a function which takes those batch of images and saves them in a grid-like structure.

```
# to save the images generated by the generator
def save_generator_image(image, path):
save_image(image, path)
```

The above are all the utility functions that we need. In the following sections, we will define functions to train the generator and discriminator networks.

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

Function to Train the Discriminator

First, we will write the function to train the discriminator, then we will move into the generator part. Let’s write the code first, then we will move onto the explanation part.

```
# function to train the discriminator network
def train_discriminator(optimizer, data_real, data_fake):
b_size = data_real.size(0)
real_label = label_real(b_size)
fake_label = label_fake(b_size)
optimizer.zero_grad()
output_real = discriminator(data_real)
loss_real = criterion(output_real, real_label)
output_fake = discriminator(data_fake)
loss_fake = criterion(output_fake, fake_label)
loss_real.backward()
loss_fake.backward()
optimizer.step()
return loss_real + loss_fake
```

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

Generative Adversarial Networks (GAN)

Function to Train the Discriminator

At line 3, we get the batch size of the data. Then we use the batch size to create the fake and real labels at lines 4 and 5.

Before doing any training, we first set the gradients to zero at line 7.

At line 9, we get the output_real by doing a forward pass of the real data (data_real) through the discriminator. Line 10 calculates the loss for the real outputs and the real labels.

Similarly, at line 12, we get fake outputs using fake data. And line 13, calculates the loss for the fake outputs and the fake labels.

Lines 16 to 18 backpropagate the gradients for the fake and the real loss and update the parameters as well.

Finally, at line 20, we return the total loss for the discriminator network.

I hope that the above steps make sense. If you are feeling confused, then please spend some time to analyze the code before moving further.

https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/