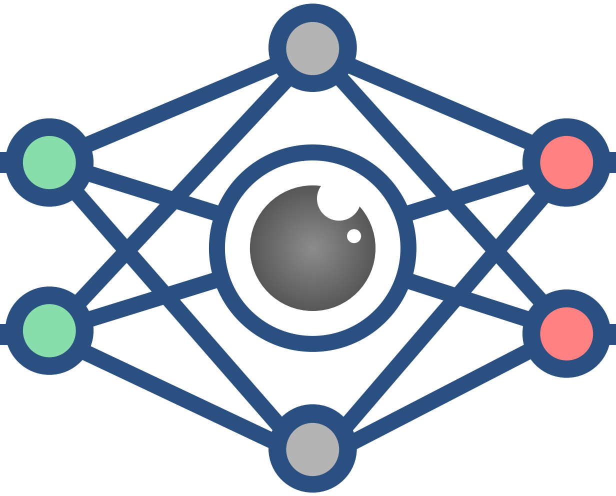
CS3485

Deep Learning for Computer Vision



Lec 16: Image Generation with GANs

(Tentative) Lecture Roadmap

Basics of Deep Learning

- Intro to Computer Vision
- Linear Classifiers and Perceptron
- Multilayer Perceptron and Intro to Deep Learning
- Optimization and Regularization
- Pytorch
- Convolutional Neural Networks
- Data Augmentation and Deep CNNs
- Transfer Learning and Residual Nets
- Inception Net and what CNNs learn
- Adversarial Examples and Self-supervision

Computer Vision Tasks

- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- Transformers and ChatGPT
- Image Generation by Prompt

Announcements

- Our baby is due today! I'm working from home as much as possible. Therefore, online office hours today (from 4 - 5pm).

Generative Models

- Last time, we saw an important unsupervised learning task called dimensionality reduction.
- Another very important unsupervised task is called **Generative Modelling** (or **Generative AI**):

Generative modeling is the task that aims at learning a model that is capable of generating unseen data instances according to the patterns learned in a given dataset.

- In CV, this task refers to learning how to **generate new images** from an available image dataset.
- An example of this is, for example, how can we generate new handwritten digits for MNIST?

The digits above were artificially generated (there are not in the original MNIST dataset.

Image Generation

- The task of **Image Generation** plays an important role in in the study of Computer VIsion and Human Perception as well, because, according to the physics nobel prize winner Richard Feynman,

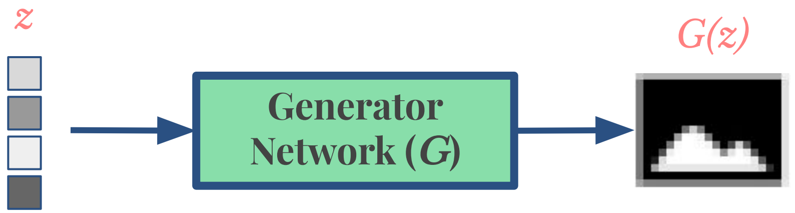
“What one cannot create, one does not understand”

- Today, we’ll dive into one of the most popular and powerful frameworks for that task in Deep Learning called **Generative Adversarial Networks (GANs),** published in 2014.
- GANs’ results have in fact **revolutionize** AI in many domains (like the one of artificial face generation)!

These people **don’t exist**! Check out more in here.

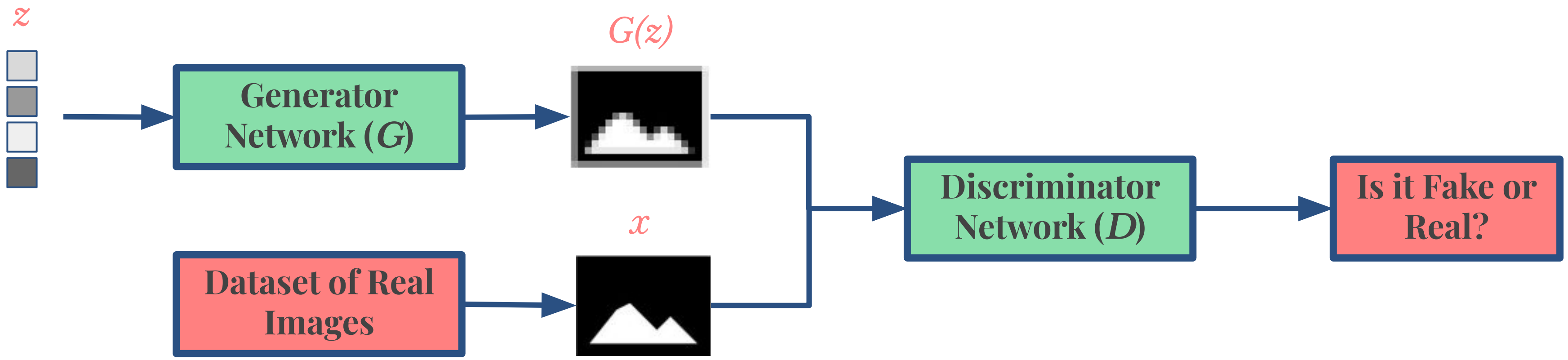
Generative Adversarial Networks

- In this setting, we have a dataset of images (no labels) at our disposal and we'd like to find a model/algorithm that generates a new image that resembles those in the dataset.
- Furthermore, we'd like to generate **different realistic images each time we run it**.
- GANs' approach to solve it is simple: we give a random vector z (like **noise** sampled from a normal distribution) to a network, called the **Generator** G, that outputs an image $G(z)$.



Generative Adversarial Networks

- The challenge is on how we train G. For that reason, GANs have another network called **Discriminator** D, that classifies images in real or fake.
- If the image y is a generated image, $D(y) = 0$, meaning it is **fake**. If y is **real**, $D(y) = 1$.
- This means that, if D is well trained, we should have $D(x) = 1$, for any image x from our image dataset, and $D(G(z)) = 0$, for any random vector z.

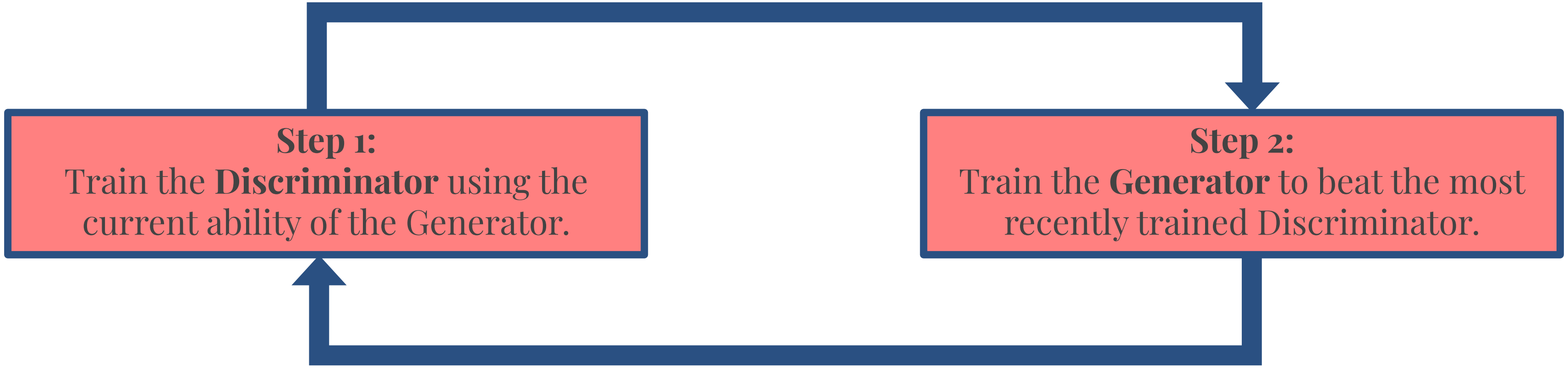


Intuition behind Generative Adversarial Networks

- The “Adversarial” from GAN is such because the Discriminator and the Generator are supposed the be **competing networks** (adversaries) that are trying beat each other.
- The allegory for this clash is usually that of a **artistic forger** and an **“art” detective**:
  - The **Generator (the forger)** tries to create artwork that fools the detective to think that those pieces are real!
  - The **Discriminator (the detective)** learns what to discriminate between real and the forged artwork by learning from some examples of fake and real data.
- In this process both the forger and the detective get better at what they do!
  - With a better detective around, the forger needs to learn how to draw more realistic paintings compared to the real artwork.
  - With the access to better forged artwork, the detective needs to learn to detect finer details that differentiate the real and the forged artwork.
- After repeating this back and forth a few times, the forger learns to draw very realistically!

Training the GAN

- The Training of a GAN involves two steps that alternate:



Step 1:
Train the **Discriminator** using the current ability of the Generator.
Step 2:
Train the **Generator** to beat the most recently trained Discriminator.

- The final goal is to generate images from the Generator such that **they are classified incorrectly by a well trained Discriminator**.
- We repeat this process until the generated images are (subjectively) “realistic enough”.

Math in GANs

- The Generator/Discriminator strife can expressed by two optimization problems*:
  - **For G fixed**, we want to find D that outputs *1* for inputs coming from the dataset (i.e., maximizes the $D(x)$) and *0* to those inputs coming from the generator (i.e., minimizes $D(G(z))$):

$$\max_{D} \mathbb{E}_{x\sim Data}[\log(D(x))] + \mathbb{E}_{z\sim Noise}[\log(1 - D(G(z)))]$$

  - **For D fixed**, we find G that fools the discriminator as much as possible, meaning, it makes D output *1* to images generated by it, i.e., maximizes $D(G(z))$ (which is like minimizing $-D(G(z))$):

$$\min_{G} \mathbb{E}_{z\sim Noise}[\log(1 - D(G(z)))]$$

- In GAN's literature, we usually put those problems together in a min-max optimization:

$$\min_{G} \max_{D} \mathbb{E}_{x\sim Data}[\log(D(x))] + \mathbb{E}_{z\sim Noise}[\log(1 - D(G(z)))]$$

* The logs are in the formulas to make the statistics behind them precise. For the intuition above, you can "forget" they are in there.

The losses in GANs

- The previous formulas can be easily translated to two losses Cross Entropy losses.
- In our case Binary Cross Entropy (BCE)*, since the discriminator only outputs two possible classes: real or fake.
  - For a real image x from the dataset and and random vector z, we want $D(x)$ to be as close as possible to 1 and $D(G(z))$ to as close as possible to 0. The Discriminator loss is then given by:

$$l_D(x, z) = l_{BCE}(D(x), 1) + l_{BCE}(D(G(z)), 0) \quad (G \text{ fixed})$$

  - For a random vector z, we want $D(G(z))$ to be as close as possible to 1. The Generator loss is then given by:

$$l_G(z) = l_{BCE}(D(G(z)), 1) \quad (D \text{ fixed})$$

- Finally, we'll consider that z is a vector whose values are normally distributed with mean 0 and variance 1.

* Reminder: the BCE loss for one prediction $\hat{y}$ / true label y pair is $l_{BCE}(\hat{y}, y) = -y^\top \log(\hat{y}) = -y_0 \log(\hat{y}_0) - (1 - y_1) \log(1 - \hat{y}_1)$

GANs in Pytorch

- We'll now create a GAN with the weights described in the previous slides to generate new MNIST digits.
- To that end, we first load the training images from MNIST:

```python
import torch

from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
from torchvision import transforms

device = "cuda" if torch.cuda.is_available() else "cpu"

transform = transforms.Compose([
       transforms.ToTensor(),
       transforms.Normalize(mean=(0.5,), std=(0.5,)) # The image values will be approximately on a
                                                     # [-0.5, 1.5] range, since std = 0.5.
])

train dataset = MNIST('~/data', train=True, download=True, transform=transform)
data_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
```

GANs in Pytorch

- Now, we create our two networks, both being Multilayer Perceptrons, for simplicity:
  - The `generator` network transforms a (random) input of size *100* into an output of size *784* (remember that MNIST images are *28*×*28*). We use `Tanh()`, because its range is [*-1, 1*] (a bit similar to our image value range) and because it works best in practice.
  - The `discriminator` network has a *784* vector as input and outputs either *0* or *1* (fake or real).

```
import torch.nn as nn
```

```
generator = nn.Sequential(
              nn.Linear(100, 128),
              nn.ReLU(),
              nn.Linear(128, 512),
              nn.ReLU(),
              nn.Linear(512, 1024),
              nn.ReLU(),
              nn.Linear(1024, 784),
              nn.Tanh()
          ).to(device)
```

```
discriminator = nn.Sequential(
                  nn.Linear(784, 1024),
                  nn.ReLU(),
                  nn.Linear(1024, 512),
                  nn.ReLU(),
                  nn.Linear(512, 128),
                  nn.ReLU(),
                  nn.Linear(128, 1),
                  nn.Sigmoid()
              ).to(device)
```

GANs in Pytorch

- We just need to define two **separate optimizers** for the discriminator and for the generator (Adam as always):

```python
import torch.optim as optim

d_optimizer = optim.Adam(discriminator.parameters(), lr=0.0002)
g_optimizer = optim.Adam(generator.parameters(), lr=0.0002)
```

- And we also define a function to plot the eventual newly generated MNIST data:

```python
from torchvision.utils import make_grid
from torch_snippets import show

def plot_samples():
   z = torch.randn(64, 100).to(device)
   sample_images = generator(z).data.cpu().view(64, 1, 28, 28)
   grid = make_grid(sample_images, nrow=8, normalize=True)
   show(grid.cpu().detach().permute(1,2,0), sz=5)
```

GANs in Pytorch

- We also create a function that outputs a matrix of random normally distributed numbers of dimension $n \times 100$, where n is the number of images in a training batch.

```
def noise(batch_size):
   n = torch.randn(batch_size, 100)
   return n.to(device)
```

- We define our loss (BCE):

```
loss = nn.BCELoss()
```

- And now, we can create a function to train the generator. Note that its goal is to make the discriminator output 1 (or as close as possible to it) when it input is made of a batch of fake data.

```
def generator_train_step(fake_data):
   n = len(real_data)
   vec_ones = torch.ones(n, 1).to(device)

   g_optimizer.zero_grad()

   prediction = discriminator(fake_data)
   error = loss(prediction, vec_ones)
   error.backward()

   g_optimizer.step()
   return error
```

GANs in Pytorch

- Then we also create a function to train the discriminator, which takes into account batches of both real and fake data and follows the loss described previously:

```
def discriminator_train_step(real_data, fake_data):
   n = len(real_data)
   vec_ones = torch.ones(n, 1).to(device)
   vec_zeros = torch.zeros(n, 1).to(device)

   d_optimizer.zero_grad()

   prediction_real = discriminator(real_data)
   error_real = loss(prediction_real, vec_ones)
   error_real.backward()

   prediction_fake = discriminator(fake_data)
   error_fake = loss(prediction_fake, vec_zeros)
   error_fake.backward()

   d_optimizer.step()
   return error_real + error_fake
```

GANs in Pytorch

- Now, our final step is to take turns training the generator and the discriminator for batches of images and plot generated samples as we go:

```python
num_epochs = 200
for epoch in range(num_epochs):
   N = len(data_loader)
   for _, (images, _) in enumerate(data_loader):
       n_images = len(images)
       real_data = images.view(n_images, -1).to(device)

       fake_data = generator(noise(n_images)).to(device).detach()
       d_loss = discriminator_train_step(real_data, fake_data)

       fake_data = generator(noise(n_images)).to(device)
       g_loss = generator_train_step(fake_data)

   if (epoch + 1) % 5 == 0:
       plot_samples()
       print(f"Epoch: {epoch + 1}")
```

GANs in Pytorch

- Now, our final step is to take turns training the generator and the discriminator for batches of images and plot generated samples as we go:

```
num_epochs = 200
for epoch in range(num_epochs):
   N = len(data_loader)
   for _, (images, _) in enumerate(data_loader):
       n_images = len(images)
       real_data = images.view(n_images, -1).to(device)

       fake_data = generator(noise(n_images)).to(device).detach()
       d_loss = discriminator_train_step(real_data, fake_data)

       fake_data = generator(noise(n_images)).to(device)
       g_loss = generator_train_step(fake_data)

   if (epoch + 1) % 5 == 0:
       plot_samples()
       print(f"Epoch: {epoch + 1}")
```

We first flatten the images.

Here we transform noise into new, generated data to train the discriminator. Note that it is important to use `detach()` on the generated data. On detaching, we are creating a fresh copy of the tensor so that when `error.backward()` is called in `discriminator_train_step`, the tensors associated with the generator (which create `fake_data`) are not affected (they are fixed).

We then train the discriminator on the real and on the newly generated data.

GANs in Pytorch

- Now, our final step is to take turns training the generator and the discriminator for batches of images and plot generated samples as we go:

```
num_epochs = 200
for epoch in range(num_epochs):
   N = len(data_loader)
   for _, (images, _) in enumerate(data_loader):
       n_images = len(images)
       real_data = images.view(n_images, -1).to(device)

       fake_data = generator(noise(n_images)).to(device).detach()
       d_loss = discriminator_train_step(real_data, fake_data)

       fake_data = generator(noise(n_images)).to(device)
       g_loss = generator_train_step(fake_data)

   if (epoch + 1) % 5 == 0:
       plot_samples()
       print(f"Epoch: {epoch + 1}")
```

Now we generate more data to train the generator now. Note that we don't need to detach this part, since we do want the generator tensors to be affected (we are training then here, after all).

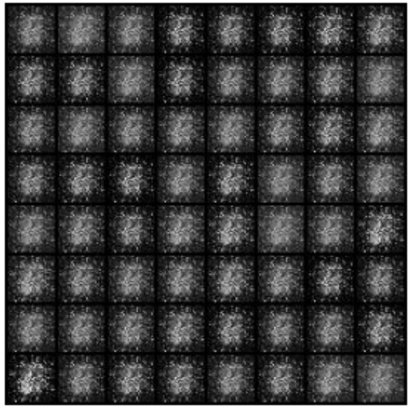
We now train the generator with the newly generated fake data.

We now just need to plot samples from our generator every once in a while.

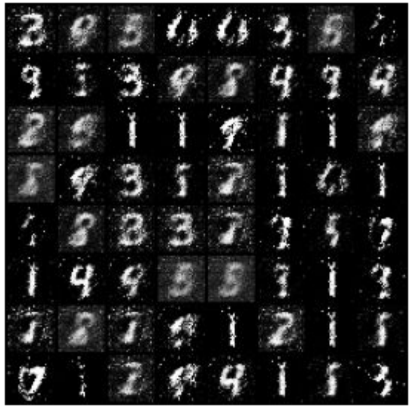
GANs in Pytorch

- Here are some results of the previous code (because of the randomness involved, you may see very different samples):

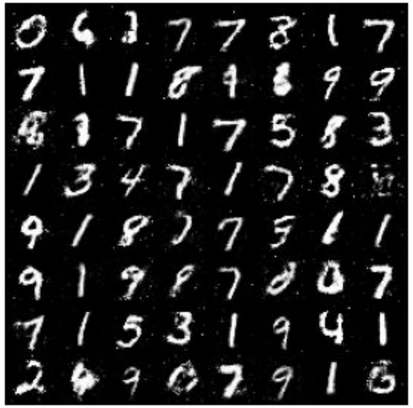
Samples after 5 epochs



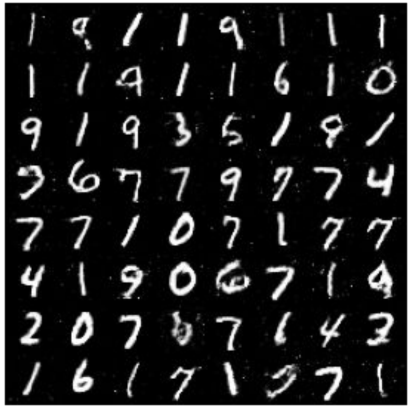
Samples after 20 epochs



Samples after 100 epochs



Samples after 200 epochs



- Note how the samples get **sharper** as we train the network for longer.

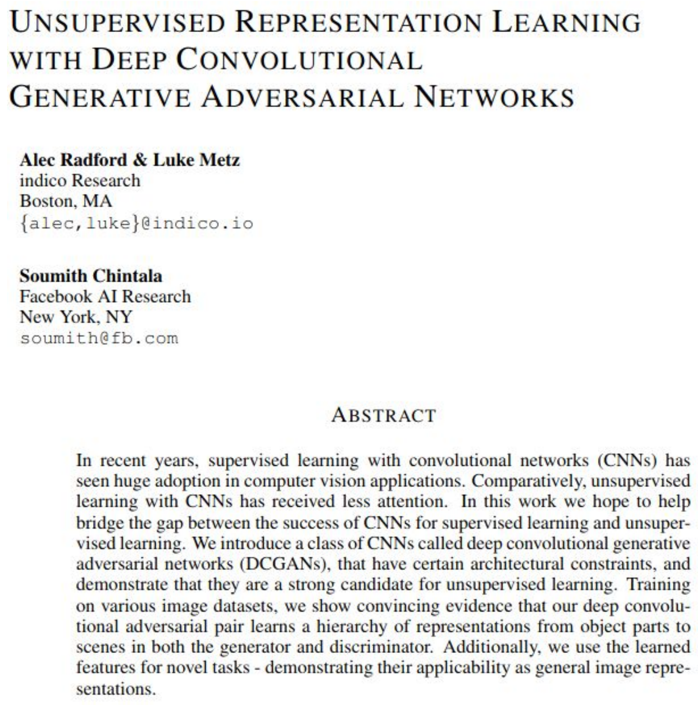
Exercise (*In pairs*)

Click here to open code in Colab

- Use the same code from before to generate some samples from the Fashion MNIST dataset. It may take around 5 min to get through a 50 epochs.

Deep Convolutional GANs

- In practice, many GAN implementations heavily use Convolutional Layers instead of Dense Layers for problems in Computer Vision.
- The main reason for this interest is due to the CNN capacity to **learn very good visual features** (as we saw with VGG and GoogLeNet) and generate **very photorealistic images**.
- Historically, this approach was introduced to the GAN community with **Deep Convolutional GAN (DCGAN)**, proposed in 2015.
- Its author’s intention was to create a **Self-Supervised model** for Image Generation!



UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS

Alec Radford & Luke Metz
indico Research
Boston, MA
{alec,luke}@indico.io

Soumith Chintala
Facebook AI Research
New York, NY
soumith@fb.com

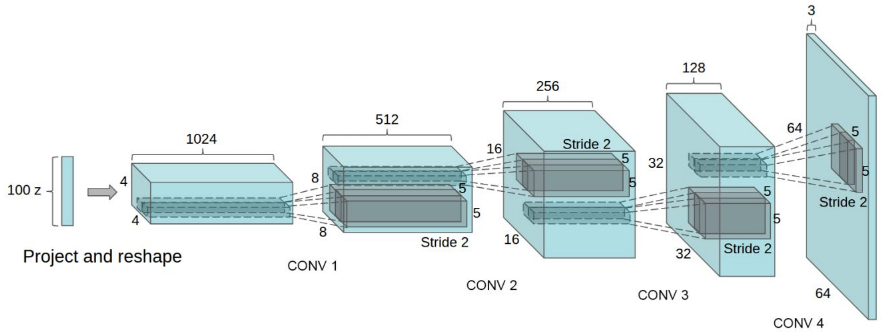
ABSTRACT

In recent years, supervised learning with convolutional networks (CNNs) has seen huge adoption in computer vision applications. Comparatively, unsupervised learning with CNNs has received less attention. In this work we hope to help bridge the gap between the success of CNNs for supervised learning and unsupervised learning. We introduce a class of CNNs called deep convolutional generative adversarial networks (DCGANs), that have certain architectural constraints, and demonstrate that they are a strong candidate for unsupervised learning. Training on various image datasets, we show convincing evidence that our deep convolutional adversarial pair learns a hierarchy of representations from object parts to scenes in both the generator and discriminator. Additionally, we use the learned features for novel tasks - demonstrating their applicability as general image representations.

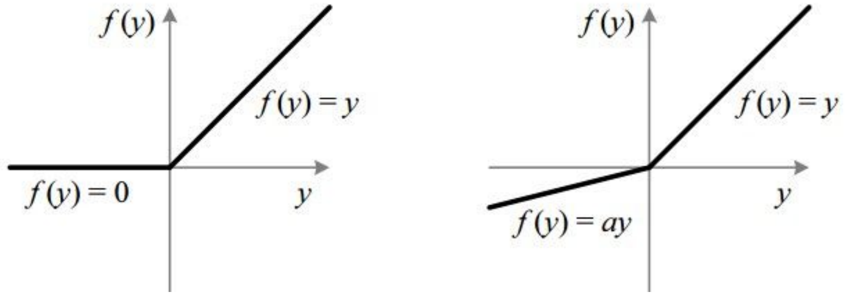
Deep Convolutional GANs

- DCGAN makes the following some important changes to the traditional architecture (also called **Vanilla GAN**) we saw previously. Some of these changes are:
  - Replacing any pooling layers with (strided) **ConvLayers** in the discriminator and (strided) **Transpose ConvLayers** in the generator.
  - Removing fully connected hidden layers in order to get **deeper architectures**.
  - Using only ReLU activation in generator and **Leaky ReLU activation** in the discriminator.

Generator for DCGAN



ReLU (left) vs. Leaky ReLU (right)



Deep Convolutional GANs

- The authors of DCGAN implemented Leaky ReLU to solve a known issue with usual ReLU: **The Dying ReLU Problem (DRP)**.
- The DRP refers to the scenario when many ReLU neurons only output values of 0, because their input happens to usually be negative numbers.
- When this happens, the gradients fail to flow during backpropagation, and the **weights don’t get updated**. Ultimately a large part of the network becomes inactive, and it **is unable to learn** further.
- Leaky ReLU solves this problem by having **negative inputs lead to non-zero outputs**.
- In PyTorch, you can implement the Leaky ReLU activation using the following:

```
nn.LeakyReLU(negative_slope=0.01)
```

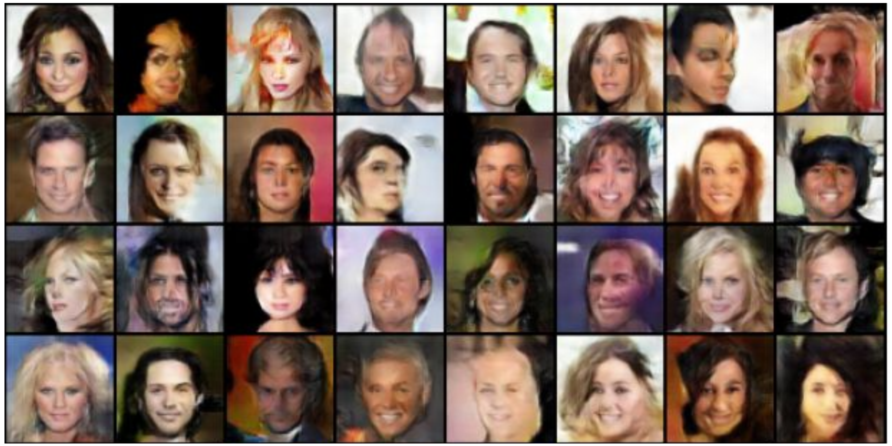
where `negative_slope` is the same as a in the previous slide.

DCGAN results

- Using this approach DCGAN is able to learn faster and more efficiently and eventually generate interestingly looking images.
- These results were taken from this **official** DCGAN tutorial (one of the few) from PyTorch.

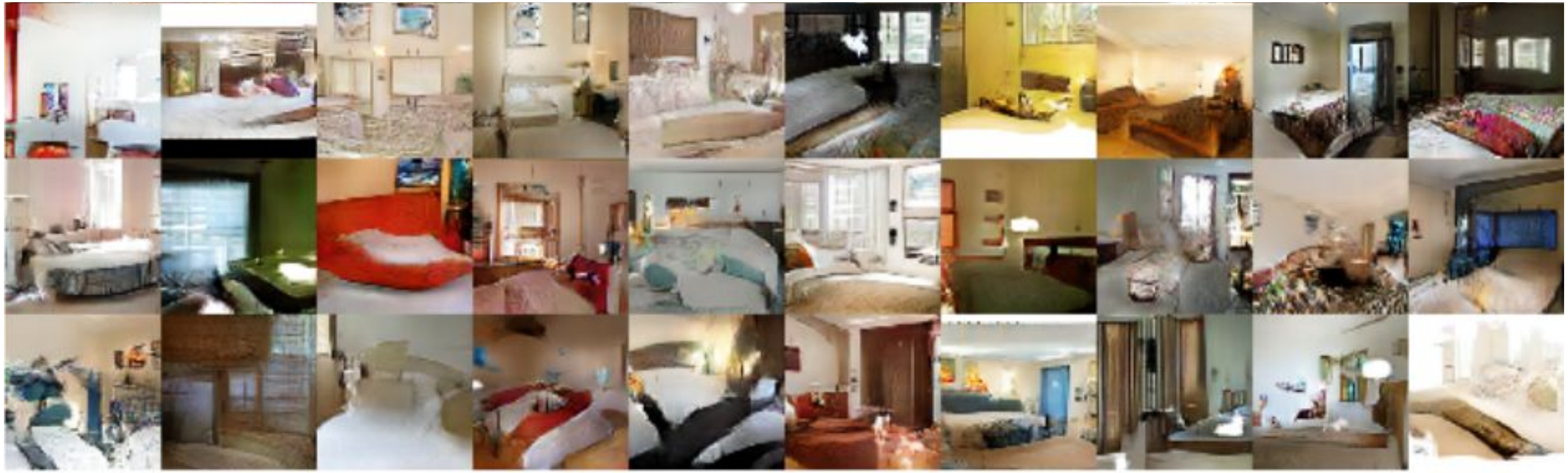
Real images used in training

Images generated by DCGAN



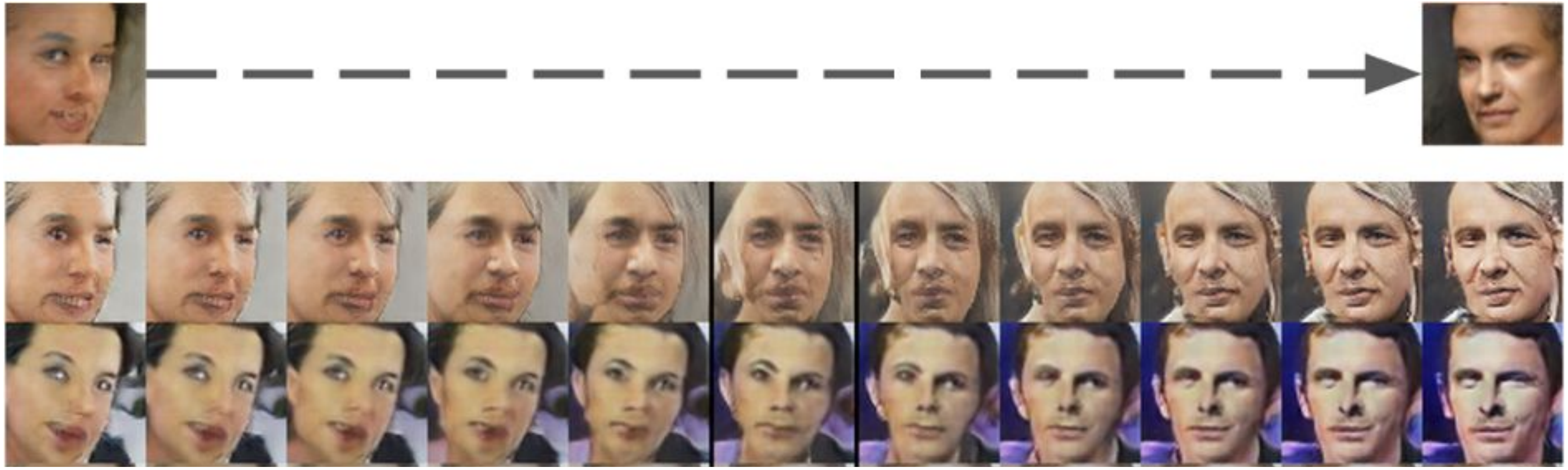
DCGAN results

- In the DCGAN paper, the authors showed that their method is able to generate (somewhat) realistic bedroom scenes when trained on the LSUN Dataset (for indoor scenes):



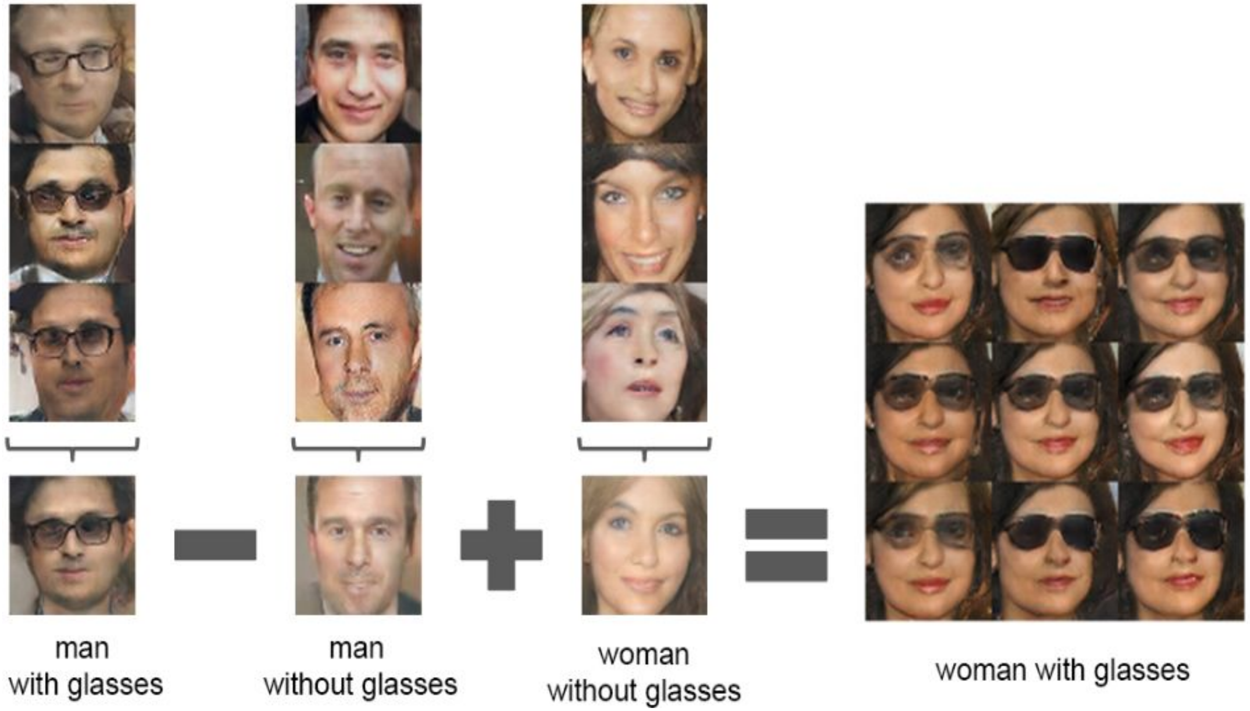
DCGAN results

- They also showed that **interpolating** latent representations (the z's), we also interpolate the generated images from those representations.
- Below, it is possible to transition from one pose to another!



DCGAN results

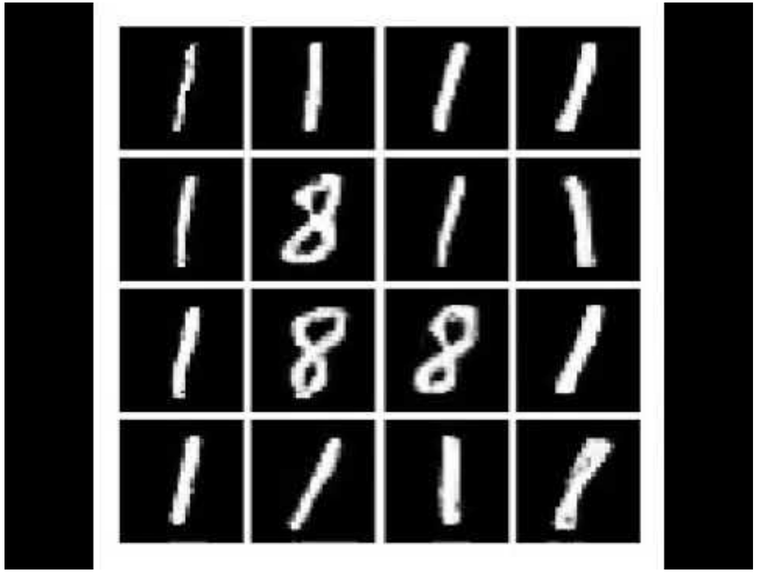
- It is even possible to do arithmetic with the latent vectors and get that arithmetic reflected in visual concepts.
- For example, + and - on the latent space represent removal or introduction of visual concepts on the image space.



Mode Collapse when training GANs

- A common challenge of training GANs (at least in its **vanilla form**, which is what we saw so far) is its susceptibility to **Mode Collapse.**
- Mode Collapse happens when the generator **fails** to generate samples that are as diverse as the distribution of the real-world data.
- It happens because the generator finds a way to generate data that easily beats the discriminator and then it **focus on generating only that kind of data**.
- That kind of data (which usually is one available data class) is called a mode of the data.

GAN Mode Collapse when training a vanilla GAN on MNIST data



Mode Collapse when training GANs

- One intuitive way to see how Mode Collapse and other problems when training Vanilla GANs can happen is found in its discriminator:
  - It is like a teacher who only says “pass”/“fail” to a student and nothing more.
  - It therefore does not give much of information to the generator about how to improve itself.
- A typical solution to this issue is to use Wasserstein GAN, published in 2017, where it replaces the discriminator by a **critic**:
  - It doesn’t simply checks “fake or real” (discrete options), by it gives an **continuous unbounded output** (any real number), such that, the higher that number, the more real it is.
  - This gives the generator more information from the discriminator to work on (think of a teacher which doesn’t just say “pass”/“fail”, but also says how close you are from passing and failing).
  - To do so, they use a different (by even simpler) loss function, called **Wasserstein Loss**, which has (beautiful) connections to **Optimal Transport Theory**.
- *Before we finish*: there is this cool website, called GAN Lab, where you can visualize what GANs are learning in their training process.

Video: *AI Generated Music and Music Video*