EE 046202 - Technion - Unsupervised Learning & Data Analysis

Tal Daniel

Tutorial 10 - Generative Adversarial Networks (GANs)

* Image Source (https://becominghuman.ai/with-gans-world-s-first-ai-generated-painting-to-recent-advancement-of-nvidia-b08ddfda45b1)

Agenda

In [1]:

```
# imports for the tutorial
import time
import numpy as np
import matplotlib.pyplot as plt

# pytorch
import torch
import torch.nn.functional as F
from torchvision import datasets
from torchvision import transforms
import torch.nn as nn
from torch.utils.data import DataLoader

if torch.cuda.is_available():
    torch.backends.cudnn.deterministic = True
```

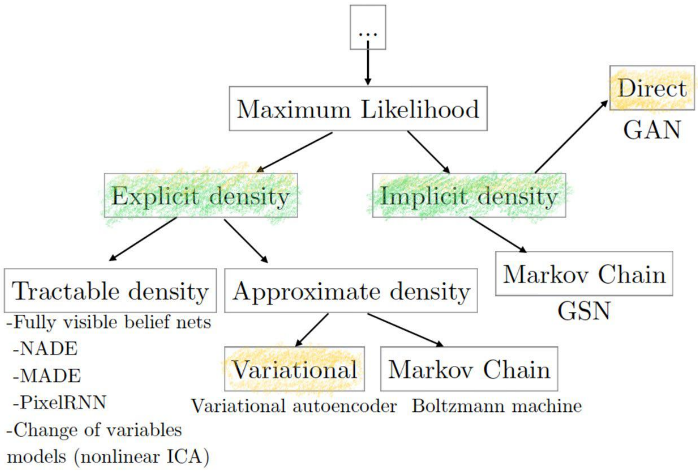
What Are Generative Adversarial Networks (GANs)?

- **Generative** - learn a generative model that can generate new data.
- **Adversarial** - trained in an *adversarial* setting (there is some competition during the model's training).
- **Networks** - the model is implemented using deep neural networks.

GANs were first introduced in [Generative Adversarial Networks (http://papers.nips.cc/paper/5423-generative-adversarial-nets)](http://papers.nips.cc/paper/5423-generative-adversarial-nets), NIPS 2014, by Goodfellow et al.

Discriminative vs. Generative

- So far, we have only seen *discriminative* models
  - Given an image X, predict a label Y
  - That is, we learn $P(Y \mid X)$
- The problem with discriminative models:
  - During training, labels are required, as it is a *supervised* setting.
  - Can't model $P(X)$, i.e., the probability of seeing a certain image.
  - As a result, can't *sample* from $P(X)$, i.e., **can't generate new images**.
- **Generative** models can overcome these limitations!
  - They can model $P(X)$, implicitly (e.g. GANs) or explicitly (e.g. Variational Autoencoders - VAEs).
  - Given a trained model, can generate new images (or data in general).

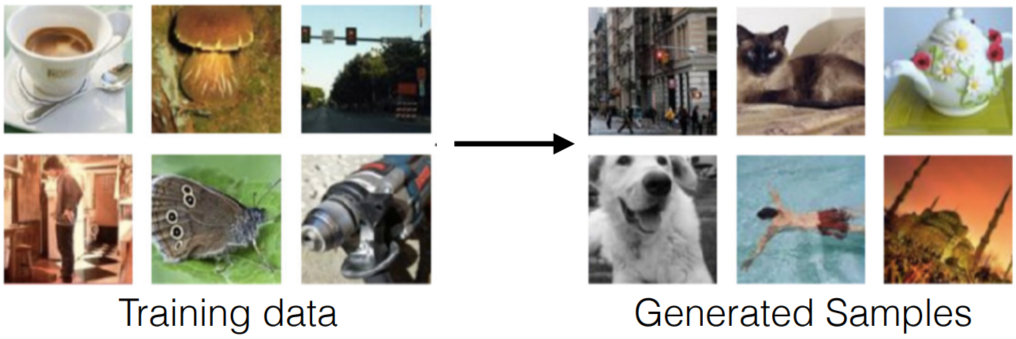


[Image Source (https://arxiv.org/abs/1701.00160)](https://arxiv.org/abs/1701.00160)

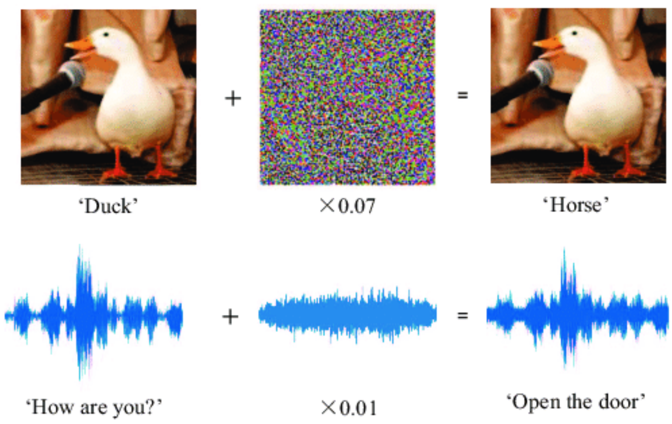
- **Explicit** density estimation: explicitly define and solve for $p_{model}(x)$.
- **Implicit** density estimation: learn a model that can sample from $p_{model}(x)$ without explicitly defining it.

Adversarial Training

- **Goal**: given training data, generate new samples from the same distribution.

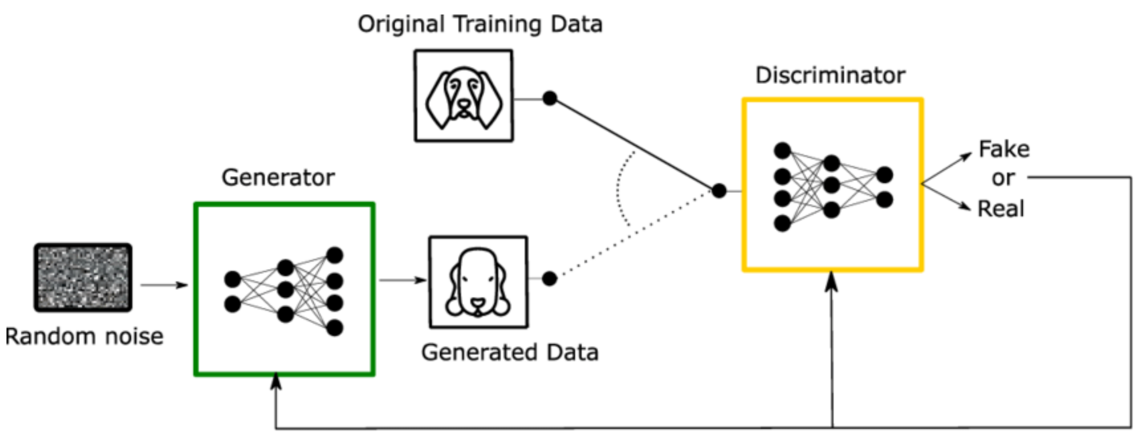


- In **general adversarial setting** (can also be discriminative):
  - We can generate adversarial samples to *fool* a dsicriminative model.
  - Using adversarial samples, we can make models more **robust**.
  - Doing this will require the adversarial samples to be of better quality over time.
    - This will require more effort in generating such quality samples!
  - Repeating this process will result in a better *discriminative* model.



- [Image Source (https://www.researchgate.net/publication/325370539_Protecting_Voice_Controlled_Systems_Using_Sound_Source_Identification_Based_on_Acoustic_Cu](https://www.researchgate.net/publication/325370539_Protecting_Voice_Controlled_Systems_Using_Sound_Source_Identification_Based_on_Acoustic_Cu)

- **GANs** extend this idea to *generative* models:
  - **Generator**: generate fake samples, tries to fool the *Discriminator*.
  - **Discriminator**: tries to dsitinguish between real and fake samples.
  - Train them **against** each other!
  - Repeat this and get a better *Generator* over time.



- [Image Source (https://www.researchgate.net/publication/334100947_Partial_Discharge_Classification_Using_Deep_Learning_Methods-Survey_of_Recent_Progress)](https://www.researchgate.net/publication/334100947_Partial_Discharge_Classification_Using_Deep_Learning_Methods-Survey_of_Recent_Progress)

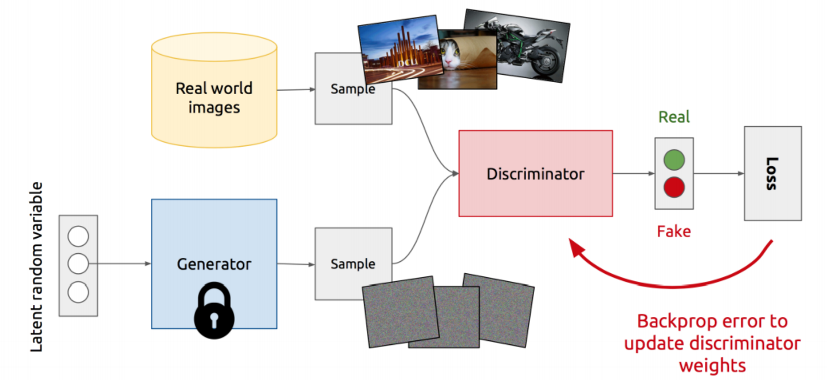
- [Image Source (https://towardsdatascience.com/comprehensive-introduction-to-turing-learning-and-gans-part-2-fd8e4a70775)](https://towardsdatascience.com/comprehensive-introduction-to-turing-learning-and-gans-part-2-fd8e4a70775)

🎮 GANs - A Game Theory Persepective - Nash Equilibrium

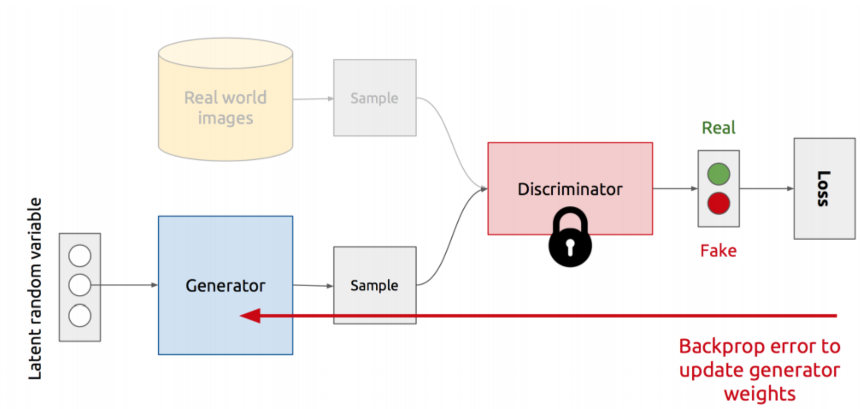
- GAN is based on a **zero-sum** cooperative game (minimax).
  - In short, if one wins the other loses.
- In game theory, the GAN model **converges** when the *discriminator* and the *generator* reach a **Nash equilibrium**.
- **Nash equilibrium** - as both sides want to beat the other, a Nash equilibrium happens when *one player will not change its action regardless of what the opponent may do*.
- **Cost functions may not converge using gradient descent in a minimax game**.

GANs Training Steps

- Training the **Discriminator**
  - **Freeze** the *Generator* and generate fake samples (that is, when backpropagating, don't update the generator weights)



- Training the **Generator**
  - **Freeze** the *Discriminator*, and update the Generator to get a higher score (like the real data) from the Discriminator



Formulation & Algorithm

- For a Discriminator (binary classifier) D, a Generator G and a reward function V, the GAN's objective function:

$$\min_G \max_D V(D, G)$$

- It is formulated as a **minimax game**, where:
  - The **Discriminator** D is trying to *maximize* its reward $V(D, G)$
  - The **Generator** G is trying to *minimize* the Discriminator's reward (or maximize its loss)
    - Why? Because minimizing the Discriminator's reward means that the Discriminator can not tell the difference between real and fake samples, thus, the Generator is "winning".

- In our case, the reward function V:

$$V(D,G) = \mathbb{E}_{x\sim p(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim q(z)}\left[\log(1 - D\left(G(z)\right))\right]$$

  - Recall from ML course that for binary classification (real or fake) we use the [Binary Cross Entropy (BCE) (https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy)](https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy) loss function.
- The **Nash equilibrium** is reached when:
  - $P_{data}(x) = P_{gen}(x), \forall x$
  - $D(x) = \frac{1}{2}$ (completely random classifier).

Generator

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

m: Number of samples
z: Random noise samples
x: Real samples

How realistic are the generated samples?

G wants to maximize this.

Discriminator

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

Make sure real samples are classified as being real.

D wants to maximize this.

Make sure generated samples are classified as unreal.

D wants to minimize this.

- [Image Source (https://towardsdatascience.com/comprehensive-introduction-to-turing-learning-and-gans-part-2-fd8e4a70775)](https://towardsdatascience.com/comprehensive-introduction-to-turing-learning-and-gans-part-2-fd8e4a70775)

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**

Discriminator updates

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 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

Generator updates

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ 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.

Exercise - Convergence of GANs to Nash Equilibrium

Recall that the goal is that the Generator generates an example that is indistinguishable from the real data. Mathematically, probability density functions (i.e. the probability measure induced by the random variable on its range) are equal:

$$p_G(x) = p_{data}(x)$$

The optimization problem is (the value function of the min-max game):

$$V(G, D) := \mathbb{E}_{x\sim p_{data}(x)} \log(D(x)) + \mathbb{E}_{z\sim p_z(z)} \log(1 - D(G(z)))$$

The theorem: "The global minimum of the virtual training criterion $C(G) = \max_D V(G, D)$ is acheived **if and only if** $p_G = p_{data}$."

1. What is the optimal Discriminator D^*_G for *some* generator G?
2. Given an optimal Discriminator D^*_G is optimal, what is the optimal Generator G?

From the [Radon-Nikodym Theorem (https://en.wikipedia.org/wiki/Radon%E2%80%93Nikodym_theorem)](https://en.wikipedia.org/wiki/Radon%E2%80%93Nikodym_theorem) it satisfies:

$$\mathbb{E}_{z\sim p_z(z)} \log(1 - D(G(z))) = \mathbb{E}_{x\sim p_G(x)} \log(1 - D(x))$$

1. What is the intuition for the above equality?
2. Let $D(x) = y, p_{data}(x) = a, p_G(x) = b$, write down the value function $V(G, D)$ with a, b, y (without the expectancy $\mathbb{E}$).
3. Let $V(G, D) = \int_x f(y)dx$ (y is a function of x). Find the optimal Discriminator.
4. For the optimal Discriminator you found in (5), if the Generator is also optimal, what is the value of D? What is the meaning of this?
5. We showed that if $p_G = p_{data}$, the theorem is correct (and the minimum is acheived). Prove the second direction, that is, show that $p_G = p_{data}$. Hint: use the following:

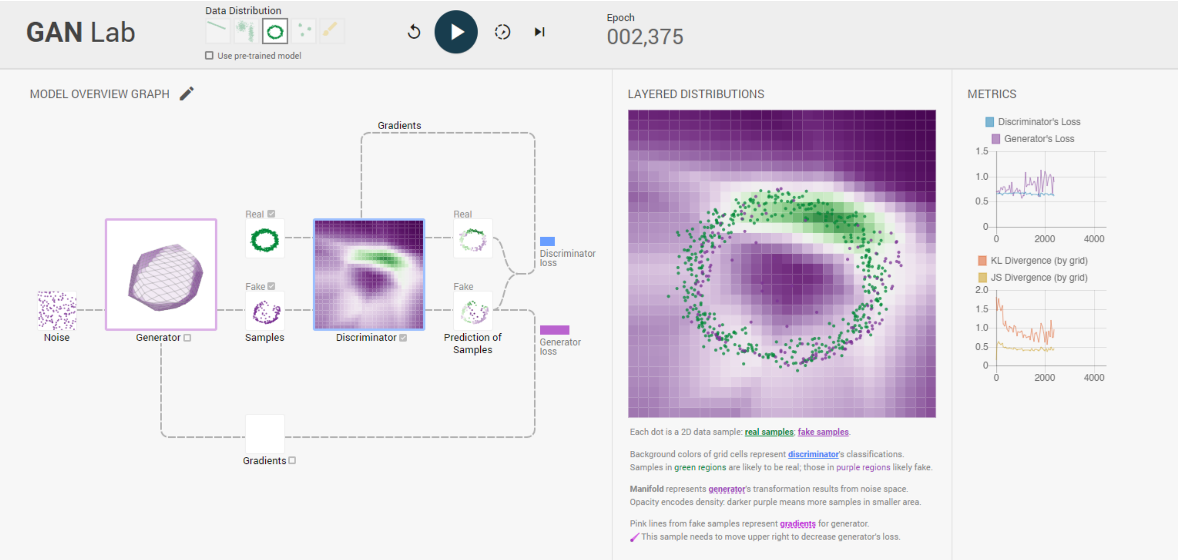
$$JSD(p_G \mid\mid p_{data}) = JSD(p_{data} \mid\mid p_G) = \frac{1}{2}KL\left(p_{data} \mid\mid \left(\frac{p_{data} + p_G}{2}\right)\right) + \frac{1}{2}KL\left(p_G \mid\mid \left(\frac{p_{data} + p_G}{2}\right)\right)$$

Solution

1. The Discriminator tries to maximize the value function (to identify between fake and real points), which means:
$$D_G^* = arg\max_D V(G, D)$$
2. The optimal G minimizes the value function when $D = D_G^*$, thus:
$$G^* = arg\min_G V(G, D_G^*)$$
3. Basically, sampling $z \sim p_z(z)$ and transforming it to x is just like saying, let's sample $x \sim p_G(x)$, where $p_G(x)$ is the transforming distribution (eventually, we create x's, so the sampling of z's is already included in the process of creating x).
4. Solution:
$$V(G, D) = \int_x (a \log y + b \log(1 - y)) dx$$
5. Let's do a bit of calculus, recall that we want to maximize the value function for D, so let's do it:
$$f^{'}(y) = 0 \rightarrow \frac{a}{y} - \frac{b}{1-y} = 0 \rightarrow y = \frac{a}{a+b}$$
Recall that $a, b >= 0$ as they are density functions (we assume non-zero points). Let's verify that it is a maximum point:
$$f^{''}(y = \frac{a}{a+b}) = -\frac{a}{\left(\frac{a}{a+b}\right)^2} - \frac{b}{\left(1 - \frac{a}{a+b}\right)^2} < 0$$
GOOD! So if
$$D(x) = \frac{p_{data}}{p_{data} + p_G},$$
we acheive the maximum.
6. The goal is $p_G = p_{data}$, so if this is satisfied, we get:
$$D_G^* = 0.5$$
This means that the Discriminator is completely confused, outputting 0.5 for examples from both p_{data} and p_G.
7. On the one hand, if we *assume* that $p_G = p_{data}$ then:
$$\begin{aligned} V(G, D_G^*) &= \int_x p_{data}(x) \log 0.5 + p_G(x) \log(1 - 0.5) dx \\ &= -log2 * 1 - log2 * 1 = -log4 \end{aligned}$$
We need to show that this value is the minimum. Let's look from the other direction, that is, we now don't assume that $p_G = p_{data}$. For any G, we can plug in D_G^* into $C(G)$:
$$C(G) = \int_x p_{data}(x) \log\left(\frac{p_{data}(x)}{p_{data}(x) + p_G(x)}\right) + p_G(x) \log\left(\frac{p_G(x)}{p_{data}(x) + p_G(x)}\right) dx$$
Using the hint, we know that we need to get to the form of the JSD, so let's do it:
$$\begin{aligned} C(G) &= \int_x a \log\left(\frac{a}{a+b}\right) + b \log\left(\frac{b}{a+b}\right) dx \\ &= \int_x a \log\left(\frac{2a}{2(a+b)}\right) + b \log\left(\frac{2b}{2(a+b)}\right) dx \\ &= \int_x a \log(0.5) + a \log\left(\frac{2a}{a+b}\right) + b \log(0.5) + b \log\left(\frac{2b}{a+b}\right) dx \\ &= -\log 4 + 2JSD(p_G \mid\mid p_{data}) \end{aligned}$$
We also know that
$$JSD \geq 0$$
So we know that the minimum value $C(G)$ can get is $-\log 4$ and when does that happen? ONLY when $JSD(p_G \mid\mid p_{data}) = 0$, which only happens when
$$p_G = p_{data},$$
and we are DONE!

2D Demo

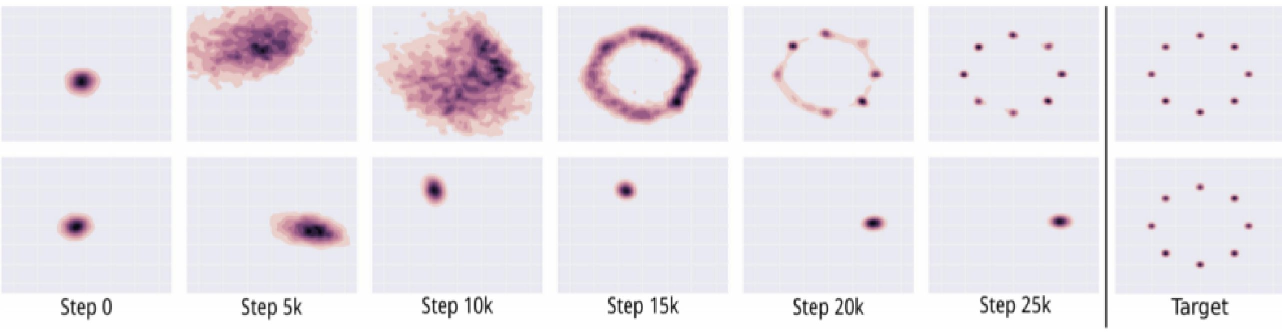
[GAN Lab](https://poloclub.github.io/ganlab/)



[(https://poloclub.github.io/ganlab/)](https://poloclub.github.io/ganlab/)

GANs (Serious) Problems

- **Non-convergence**: the model parameters oscilate, destabilize and (almost) never converge.
- **Mode Collapse**: the *Generator* collapses, which produces limited varieties of samples.
  - For example, on a 2D eight-Gaussians dataset:



  - [Image Source (https://mc.ai/gan-unrolled-gan-how-to-reduce-mode-collapse/)](https://mc.ai/gan-unrolled-gan-how-to-reduce-mode-collapse/)

- **Vanisihng/Diminishing Gradient**: the discriminator gets *too good* such that the generator gradient vanishes and learns nothing.
  - Proof: HW
  - Possible remedy: replace the problematic term with a **non-saturating** loss

$$\mathbb{E}_{z\sim q(z)}\left[\log(1-D\left(G(z)\right))\right] \rightarrow -\mathbb{E}_{z\sim q(z)}\left[\log D\left(G(z)\right)\right]$$

- **GANs are highly sensitive to hyper-parameters!**
  - Even the slightest change in hyper-parameters may lead to any of the above, e.g. even changing the learning rate from 0.0002 to 0.0001 may lead to instability.

Vanilla-GAN on MNIST with PyTorch

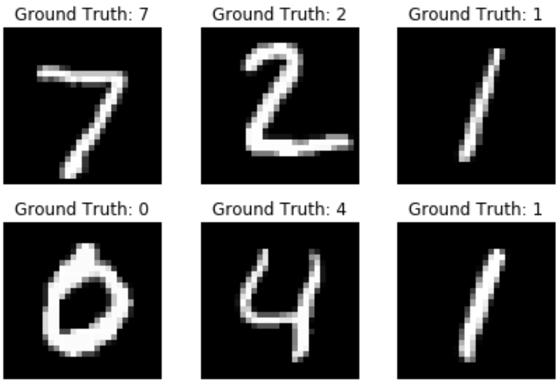
- Based on example by [Sebastian Raschka (https://github.com/rasbt/deeplearning-models)](https://github.com/rasbt/deeplearning-models)

In [2]:
```
##########################
### SETTINGS
##########################

# Device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Hyperparameters
# Remember that GANs are highly sensitive to hyper-parameters
random_seed = 123
generator_learning_rate = 0.001
discriminator_learning_rate = 0.001
NUM_EPOCHS = 100
BATCH_SIZE = 128
LATENT_DIM = 100  # latent vectors dimension [z]
IMG_SHAPE = (1, 28, 28)  # MNIST has 1 color channel, each image 28x8 pixels
IMG_SIZE = 1
for x in IMG_SHAPE:
    IMG_SIZE *= x
```

In [3]:

```python
##########################
### MNIST DATASET
##########################

# Note transforms.ToTensor() scales input images
# to 0-1 range
train_dataset = datasets.MNIST(root='./datasets',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

test_dataset = datasets.MNIST(root='./datasets',
                              train=False,
                              transform=transforms.ToTensor())


train_loader = DataLoader(dataset=train_dataset,
                          batch_size=BATCH_SIZE,
                          shuffle=True)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
                         shuffle=False)

# Checking the dataset
for images, labels in train_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break

# let's see some digits
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
print("shape: \n", example_data.shape)
fig = plt.figure()
for i in range(6):
    ax = fig.add_subplot(2,3,i+1)
    ax.imshow(example_data[i][0], cmap='gray', interpolation='none')
    ax.set_title("Ground Truth: {}".format(example_targets[i]))
    ax.set_axis_off()
plt.tight_layout()
```

```
Image batch dimensions: torch.Size([128, 1, 28, 28])
Image label dimensions: torch.Size([128])
shape: 
 torch.Size([128, 1, 28, 28])
```



In [4]:
```
##########################
### MODEL
##########################

class GAN(torch.nn.Module):

    def __init__(self):
        super(GAN, self).__init__()

        # generator: z [vector] -> image [matrix]
        self.generator = nn.Sequential(
            nn.Linear(LATENT_DIM, 128),
            nn.LeakyReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(128, IMG_SIZE),
            nn.Tanh()
        )

        # discriminator: image [matrix] -> label (0-fake, 1-real)
        self.discriminator = nn.Sequential(
            nn.Linear(IMG_SIZE, 128),
            nn.LeakyReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )


    def generator_forward(self, z):
        img = self.generator(z)
        return img

    def discriminator_forward(self, img):
        pred = model.discriminator(img)
        return pred.view(-1)
```

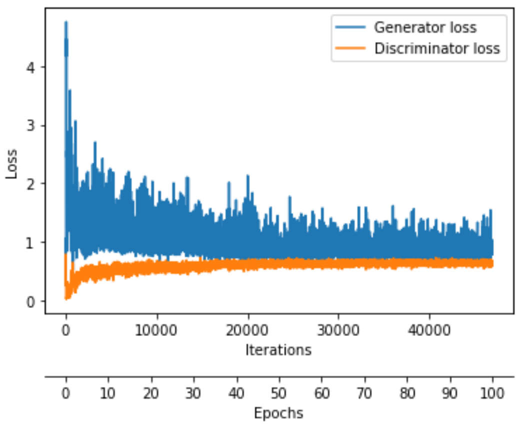
In [5]:
```
# constant the seed
torch.manual_seed(random_seed)

# build the model, send it ti the device
model = GAN().to(device)

# optimizers: we have one for the generator and one for the discriminator
# that way, we can update only one of the modules, while the other one is "frozen"
optim_gener = torch.optim.Adam(model.generator.parameters(), lr=generator_learning_rate)
optim_discr = torch.optim.Adam(model.discriminator.parameters(), lr=discriminator_learning_rate)
```

In [ ]:

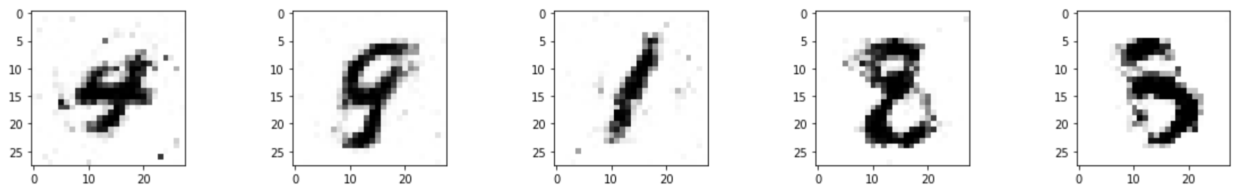
```
##########################
### Training
##########################

start_time = time.time()

discr_costs = []
gener_costs = []
for epoch in range(NUM_EPOCHS):
    model = model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = (features - 0.5) * 2.0  # normalize between [-1, 1]
        features = features.view(-1, IMG_SIZE).to(device)
        targets = targets.to(device)

        # generate fake and real labels
        valid = torch.ones(targets.size(0)).float().to(device)
        fake = torch.zeros(targets.size(0)).float().to(device)

        ### FORWARD PASS AND BACKPROPAGATION

        # --------------------------
        # Train Generator
        # --------------------------

        # 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)

        # here we use the `valid` labels because we want the discriminator to "think"
        # the generated samples are real
        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(-1, IMG_SIZE))
        real_loss = F.binary_cross_entropy(discr_pred_real, valid)

        # here we use the `fake` labels when training the discriminator
        discr_pred_fake = model.discriminator_forward(generated_features.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()

        discr_costs.append(discr_loss)
        gener_costs.append(gener_loss)


        ### LOGGING
        if not batch_idx % 100:
            print ('Epoch: %03d/%03d | Batch %03d/%03d | Gen/Dis Loss: %.4f/%.4f'
                   %(epoch+1, NUM_EPOCHS, batch_idx,
                     len(train_loader), gener_loss, discr_loss))

    print('Time elapsed: %.2f min' % ((time.time() - start_time)/60))

print('Total Training Time: %.2f min' % ((time.time() - start_time)/60))
```

In [7]:
```python
##########################
### Evaluation
##########################

ax1 = plt.subplot(1, 1, 1)
ax1.plot(range(len(gener_costs)), gener_costs, label='Generator loss')
ax1.plot(range(len(discr_costs)), discr_costs, label='Discriminator loss')
ax1.set_xlabel('Iterations')
ax1.set_ylabel('Loss')
ax1.legend()

# Set scond x-axis
ax2 = ax1.twiny()
newlabel = list(range(NUM_EPOCHS+1))
iter_per_epoch = len(train_loader)
newpos = [e*iter_per_epoch for e in newlabel]

ax2.set_xticklabels(newlabel[::10])
ax2.set_xticks(newpos[::10])

ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 45))
ax2.set_xlabel('Epochs')
ax2.set_xlim(ax1.get_xlim())
```

Out[7]: (-2344.9500000000003, 49243.95)

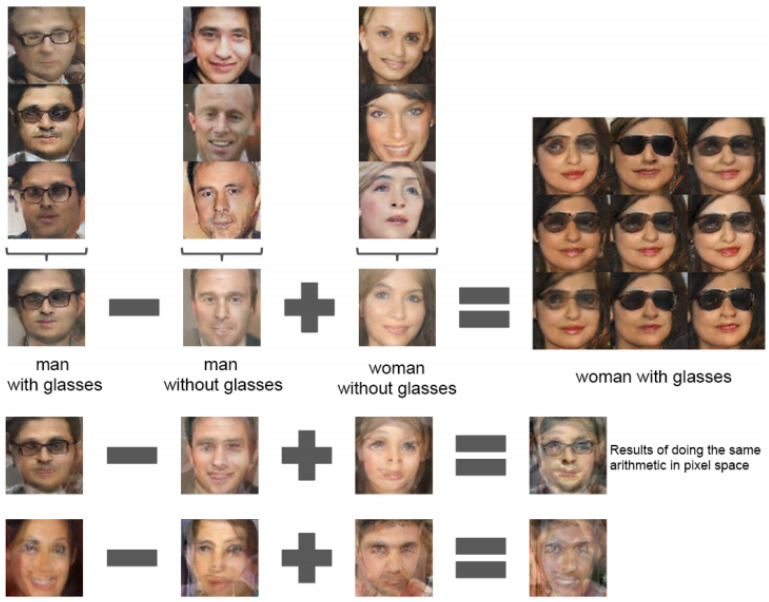


In [11]:
```python
##########################
### VISUALIZATION
##########################

model.eval()
# Make new images
z = torch.zeros((5, LATENT_DIM)).uniform_(-1.0, 1.0).to(device)
generated_features = model.generator_forward(z)
imgs = generated_features.view(-1, 28, 28)

fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(20, 2.5))


for i, ax in enumerate(axes):
    axes[i].imshow(imgs[i].to(torch.device('cpu')).detach(), cmap='binary')
```



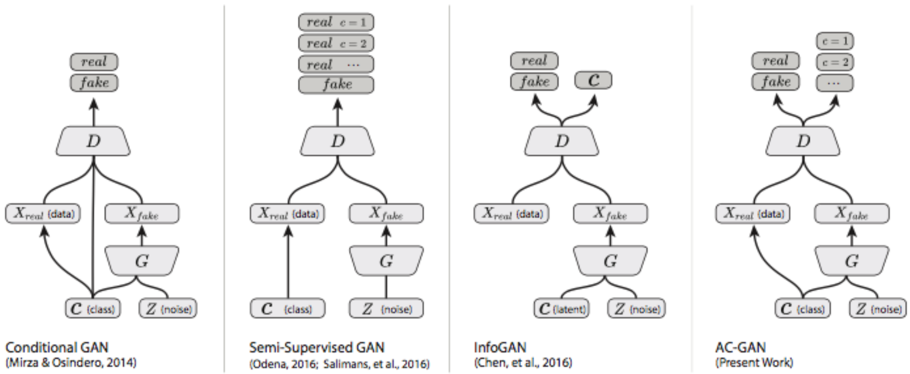
The Latent Space

- As we learn how to transform a latent vector, z, to images, we actullay learn a latent continuous space.
- This continuous spcae allows us to perform interpolation and arithmetics.
- As this space is continuous, unlike the original data (images), it was found that some operations (like summing) perform really well when done on the latent space.
- As you can see below, those operations were demonstrated in the paper [Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, Alec Radford, Luke Metz, Soumith Chintala, ICLR 2016 (https://arxiv.org/abs/1511.06434)](https://arxiv.org/abs/1511.06434)



Conditional GANs

- As you probably have noticed, we don't too much control over the latent space, e.g., with vanilla-GAN trained on MNIST we can't control what digit we are generating.
- **Conditional-GANs** - a simple modification to the original GAN framework that *conditions* the model on additional information for better multi-modal learning.
- In practice, we usually use the labels of the datasets to perform the conditioning.
  - For example, on MNIST we will use the one-hot vector representation of the digit ($1 \rightarrow [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$) along with the images from that class.
- Leads to many practical applications of GANs when we have *explicit supervision available*.
- There is more than one way to perform conditioning, some approaches are presented below.



- [Conditional Generative Adversarial Nets, Mehdi Mirza, Simon Osindero (https://arxiv.org/abs/1411.1784)](https://arxiv.org/abs/1411.1784)
- [Conditional GANs (https://assemblingintelligence.wordpress.com/2017/05/10/conditional-gans/)](https://assemblingintelligence.wordpress.com/2017/05/10/conditional-gans/)

GANs Today

- GANs are **HARD to train** and many researche studies try to improve training stability.
- **WGAN** - Wasserstein GANs use the Wasserstein (Earth Movers) distance as the loss function. Training is more stabilized than vanilla-GAN.
  - **WGAN-GP** - improves upon the original WGAN by using *Gradient Penalty* in the loss function (instead of *value clipping*)
  - WGAN Paper (https://arxiv.org/abs/1701.07875), PyTorch Code (https://github.com/Zeleni9/pytorch-wgan)
  - WGAN-GP Paper (https://arxiv.org/abs/1704.00028), PyTorch Code (https://github.com/Zeleni9/pytorch-wgan)
- **EBGAN** - Energy-Based GANs use *autoencoders* in their architecture (with the autoencoder loss).
  - EBGAN Paper (https://arxiv.org/abs/1609.03126), PyTorch Code (https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/ebgan/ebgan.py)
- **BEGAN** - Boundary Equilibrium GANs combines *autoencoders* and Wassertein distance to balance the generator and discriminator during training.
  - BEGAN Paper (https://arxiv.org/abs/1703.10717), PyTorch Code (https://github.com/anantzoid/BEGAN-pytorch)
- **Mimicry** - a lightweight PyTorch library aimed towards the reproducibility of GAN research - GitHub (https://github.com/kwotsin/mimicry)

Tips for Training GANs

All tips are here: Tips for Training GANs (https://github.com/soumith/ganhacks)

- Normalize the inputs - usually between $[-1, 1]$. Use TanH for the Generator output.
- Use the modified loss function to avoid the vanishing gradients.
- Use a spherical Z - sample from a Gaussian distribution instead of uniform distribution.
- BatchNorm (when batchnorm is not an option use instance normalization).
- Avoid Sparse Gradients: ReLU, MaxPool - the stability of the GAN game suffers if you have sparse gradients.
  - LeakyReLU is good (in both G and D)
  - For Downsampling, use: Average Pooling, Conv2d + stride
  - For Upsampling, use: PixelShuffle, ConvTranspose2d + stride

- Use Soft and Noisy Labels
  - Label Smoothing, i.e. if you have two target labels: Real=1 and Fake=0, then for each incoming sample, if it is real, then replace the label with a random number between 0.7 and 1.2, and if it is a fake sample, replace it with 0.0 and 0.3.
  - Make the labels the noisy for the discriminator: occasionally flip the labels when training the discriminator
- Track failures early:
  - D loss goes to 0 -- failure mode.
  - Check norms of gradients: if they are over 100 things are not good...
  - When things are working, D loss has low variance and goes down over time vs. having huge variance and spiking.
- **Don't** balance loss via statistics (unless you have a good reason to)
  - For example, don't do that: `while lossD > A: train D` or `while lossG > B: train G`

Cool GAN Projects (with Code)

- gans-awesome-applications (https://github.com/nashory/gans-awesome-applications)
- pytorch-generative-model-collections (https://github.com/znxlwm/pytorch-generative-model-collections)

Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- Introduction to GANs - Introduction to GANs, NIPS 2016 | Ian Goodfellow, OpenAI (https://www.youtube.com/watch?v=9JpdAg6uMXs)
- Generative Models - Stanford CS231n - Lecture 13 | Generative Models (https://www.youtube.com/watch?v=5WoItGTWV54)
- Deep Generative Modeling - MIT 6.S191 (2019): Deep Generative Modeling (https://www.youtube.com/watch?v=yFBFl1cLYx8)
- Wasserstein GANs - Nuts and Bolts of WGANs, Kantorovich-Rubistein Duality, Earth Movers Distance (https://www.youtube.com/watch?v=31mqB4yGgQY)
- Energy-Based GANs - Energy-Based Adversarial Training and Video Prediction, NIPS 2016 | Yann LeCun, Facebook AI Research (https://www.youtube.com/watch?v=x4sI5qO6O2Y)

Credits

- Slides from CS 598 LAZ (http://slazebni.cs.illinois.edu/spring17/)
- Slides by Lihi Zelnik-Mannor
- Slides from CMU - 16720B – Computer Vision (http://ci2cv.net/16720b/)
- Some material from Alexander Amini and Ava Soleimany, MIT 6.S191: Introduction to Deep Learning, IntroToDeepLearning.com (http://introtodeeplearning.com/)
- Proof of Nash Equilibrium in GANs (https://srome.github.io/An-Annotated-Proof-of-Generative-Adversarial-Networks-with-Implementation-Notes/)
- Icons from Icon8.com (https://icons8.com/) - https://icons8.com (https://icons8.com)