CPE 490 590: Machine Learning for Engineering Applications

17 Generative Adversarial Network

Rahul Bhadani
Electrical & Computer Engineering, The University of Alabama in Huntsville

Outline

Motivation

Lecture Overview

- In this lecture we discussed generative models (briefly) and how to model distributions of data using density estimation techniques. One advantage to these generative models is that we can generate data from the distribution once we have the model.
- Our goal in this lecture is to introduce *Generative Adversarial Networks* (GANs) for generating data from a distribution $p_{data}(\mathbf{x})$. GANs are able to model complex distributions of data that would be incredible challenging to model using the density estimation approaches discussed in previous lectures.
- Modeling $p_{data}(\mathbf{x})$ is increasingly challenging for complex data, such as real scenes, people, etc.

Example of a Complex $p_{data}(\mathbf{x})$

Example of a Complex $p_{data}(\mathbf{x})$

Example of a Complex $p_{data}(\mathbf{x})$

Example of a Complex $p_{data}(\mathbf{x})$



Example of a Complex $p_{data}(\mathbf{x})$

None of these people are real!

GANs

There are hundreds(?) of different types of GANs; however, this lecture only presents the vanilla GAN.

Generative Adversarial Networks

Definitions

- $p_{data}(\mathbf{x})$: Probability distribution over the data. We assume we have access to a dataset $\{\mathbf{x}_i\}_{i=1}^{n}$; however, we do not assume we have labels nor do we need them because we're not doing classification.

Definitions

- $p_{data}(\mathbf{x})$: Probability distribution over the data. We assume we have access to a dataset $\{\mathbf{x}_i\}_{i=1}^{n}$; however, we do not assume we have labels nor do we need them because we're not doing classification.
- $p_g(\mathbf{z})$: Probability distribution over a latent variable $\mathbf{z}$. This of this distribution that is over random noise.

Definitions

- $p_{data}(\mathbf{x})$: Probability distribution over the data. We assume we have access to a dataset $\{\mathbf{x}_i\}_{i=1}^{n}$; however, we do not assume we have labels nor do we need them because we're not doing classification.
- $p_g(\mathbf{z})$: Probability distribution over a latent variable $\mathbf{z}$. This of this distribution that is over random noise.
- $G(\mathbf{z})$: G is the generator neural network that takes in a noise vector, $\mathbf{z}$, to produce a vector $\mathbf{x}_{fake}$. $\mathbf{x}_{fake}$ is a "fake" data sample that is generated from $\mathbf{z}$.

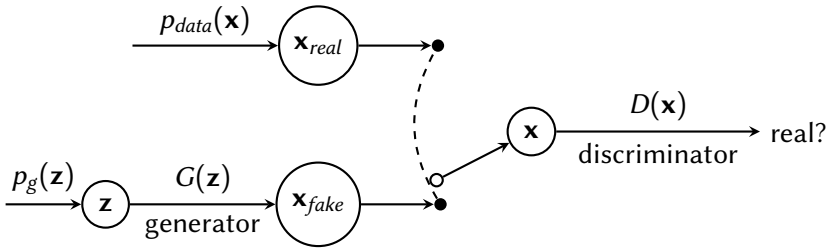
Definitions

- $p_{data}(\mathbf{x})$: Probability distribution over the data. We assume we have access to a dataset $\{\mathbf{x}_i\}_{i=1}^{n}$; however, we do not assume we have labels nor do we need them because we're not doing classification.
- $p_g(\mathbf{z})$: Probability distribution over a latent variable $\mathbf{z}$. This of this distribution that is over random noise.
- $G(\mathbf{z})$: G is the generator neural network that takes in a noise vector, $\mathbf{z}$, to produce a vector $\mathbf{x}_{fake}$. $\mathbf{x}_{fake}$ is a "fake" data sample that is generated from $\mathbf{z}$.
- $D(\mathbf{x})$: D is a discriminator network that takes in a data vector $\mathbf{x}$ and tries to discern whether $\mathbf{x}$ was sampled from $p_{data}(\mathbf{x})$ or generated by G.

Definitions

- $p_{data}(\mathbf{x})$: Probability distribution over the data. We assume we have access to a dataset $\{\mathbf{x}_i\}_{i=1}^{n}$; however, we do not assume we have labels nor do we need them because we're not doing classification.
- $p_g(\mathbf{z})$: Probability distribution over a latent variable $\mathbf{z}$. This of this distribution that is over random noise.
- $G(\mathbf{z})$: G is the generator neural network that takes in a noise vector, $\mathbf{z}$, to produce a vector $\mathbf{x}_{fake}$. $\mathbf{x}_{fake}$ is a "fake" data sample that is generated from $\mathbf{z}$.
- $D(\mathbf{x})$: D is a discriminator network that takes in a data vector $\mathbf{x}$ and tries to discern whether $\mathbf{x}$ was sampled from $p_{data}(\mathbf{x})$ or generated by G.
- Let θ_g and θ_d represent the parameters for the generator and discriminator neural networks, respectively.

Generative Adversarial Net



Generative Adversarial Net

- We have two neural network in a GAN
  - The network G's goal is to generate sample that look like they came from $p_{data}(\mathbf{x})$.
  - The network D's goal is to effectively discriminate between real samples from $p_{data}(\mathbf{x})$ and fake samples from G.
- The goals of G and D are different! We can view this scenario as a game between D and G.

Formally Defining the Optimization Task

The optimization of θ_g and θ_d can be cast as a min-max task, which is given by

$$\min_G \max_D \left\{ \mathbb{E}_{\mathbf{x} \sim p_{data}} [\log(D(\mathbf{x}))] + \mathbb{E}_{\mathbf{z} \sim p_g} [\log(1 - D(G(\mathbf{z})))] \right\}$$

Formally Defining the Optimization Task

- If $\mathbf{x} \sim p_{data}(\mathbf{x})$ then the discriminator should maximize $\log D(\mathbf{x})$
- If $\mathbf{z} \sim p_g(\mathbf{z})$ then the generator seeks to generate a sample that makes the discriminator think it is a normal sample.
- We can use stochastic gradient descent and ascent to adjust the parameters of the network.

Formally Defining the Optimization Task

Theorem 1. The global minimum of the virtual training criterion $C(G)$ is achieved if and only if $p_g = p_{data}$. At that point, $C(G)$ achieves the value $-\log(4)$. This point is a Nash Equilibrium.

Pseudo Code for Training a GAN

for $t = 1, \ldots, T$

for $k = 1, \ldots, K$

- Sample m noise data $\mathbf{z}_1, \ldots, \mathbf{z}_m$ from $p_g(\mathbf{z})$
- Sample m data samples $\mathbf{x}_1, \ldots, \mathbf{x}_m$ from $p_{data}(\mathbf{x})$
- Update the discriminator by ascending its stochastic gradient:

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

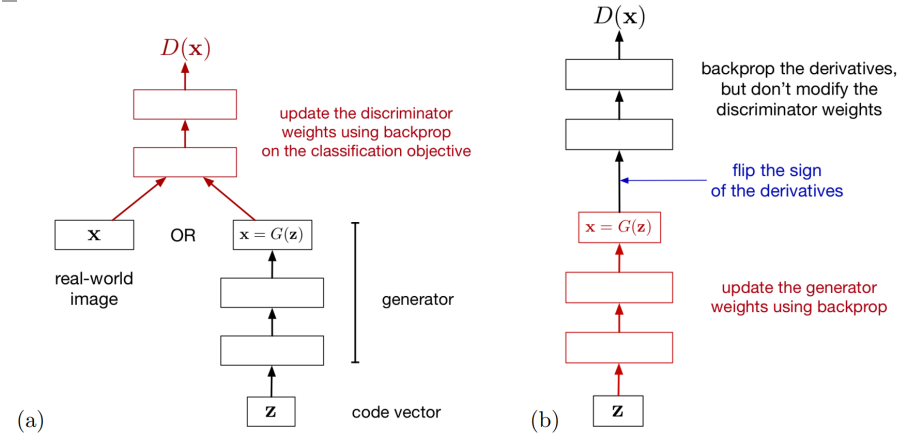
end for

- Sample m noise data $\mathbf{z}_1, \ldots, \mathbf{z}_m$ from $p_g(\mathbf{z})$
- Update the generator by descending its stochastic gradient:

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

end for

Formally Defining the Optimization Task



From Roger Grosse's GAN lecture notes.

On Training GANs

Challenge

One issue associated with training a GAN is the form of the loss. Observe above that the generator weights only get updated by the gradients of the second term:

$$\log(1 - D(G(\mathbf{z})))$$

- The problem with this is that if the generator sample is really bad, then the discriminator's prediction is close to zero, and since $\log(1 - D)$ is very flat when $D \approx 0$ there is not enough "signal" to move the generator weights meaningfully.
- Increasing learning rates do not seem to help. This is called the *saturation problem* in GANs.
- To fix this, while updating generator weights, it is common to heuristically replace the second term in the GAN loss with $-\log D(G(\mathbf{z}))$

On Training GANs

Challenge

Stably training GANs was a challenge faced by the community for quite some time (and continues to be a challenge), and a common resolution is to use *Wasserstein GANs.*

- The GAN loss is a form of distance between probability distributions (called the Jensen-Shannon divergence), and this can be generated to other distances.
- A common alternative is the Earth-mover or Wasserstein distance, leading to a different type of GAN model called Wasserstein GAN. The WGAN was shown to improve stability of the optimization process

GANs (in PyTorch)

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers
        self.model = nn.Sequential(
            *block(latent_dim, 128, normalize=False),
            *block(128, 256), *block(256, 512), *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))), nn.Tanh())
    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), *img_shape)
        return img
```

GAN in PyTorch

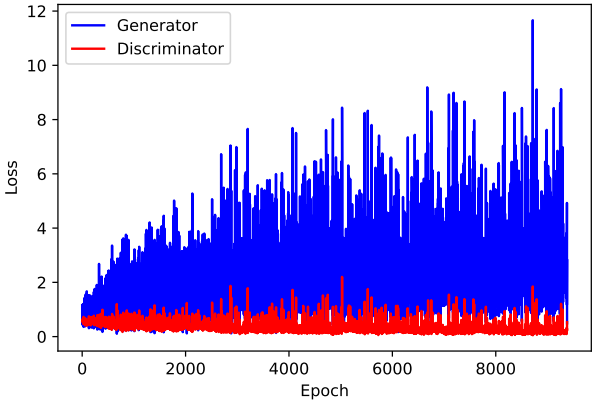
GANs (in PyTorch)

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )
    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return validity
```

GANs (in PyTorch)

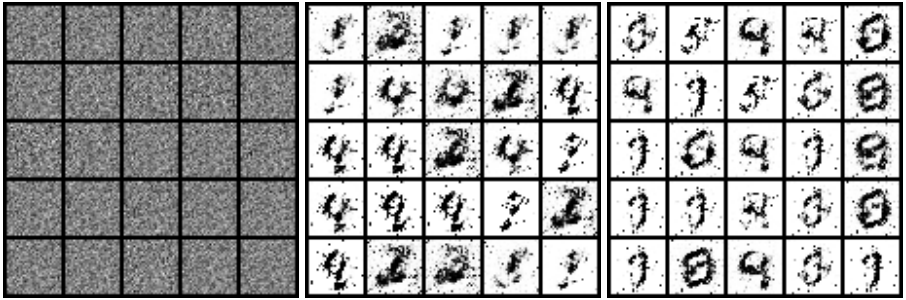
```
for epoch in range(n_epochs):
    for i, (imgs, _) in enumerate(dataloader):
        # Adversarial ground truths
        valid = Variable(Tensor(imgs.size(0), 1).fill_(1.0), requires_grad=False)
        fake = Variable(Tensor(imgs.size(0), 1).fill_(0.0), requires_grad=False)
        # Configure input
        real_imgs = Variable(imgs.type(Tensor))
        # -----------------
        #  Train Generator
        optimizer_G.zero_grad()
        # Sample noise as generator input
        z = Variable(Tensor(np.random.normal(0, 1, (imgs.shape[0], latent_dim))))
        # Generate a batch of images
        gen_imgs = generator(z)
        # Loss measures generator's ability to fool the discriminator
        g_loss = adversarial_loss(discriminator(gen_imgs), valid)
        g_loss.backward()
        optimizer_G.step()
        # ---------------------
        #  Train Discriminator
        optimizer_D.zero_grad()
        # Measure discriminator's ability to classify real from generated samples
        real_loss = adversarial_loss(discriminator(real_imgs), valid)
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()
```

GAN Loss


Generator
Discriminator
Loss
Epoch
0
2
4
6
8
10
12
0
2000
4000
6000
8000

Examples

GAN example



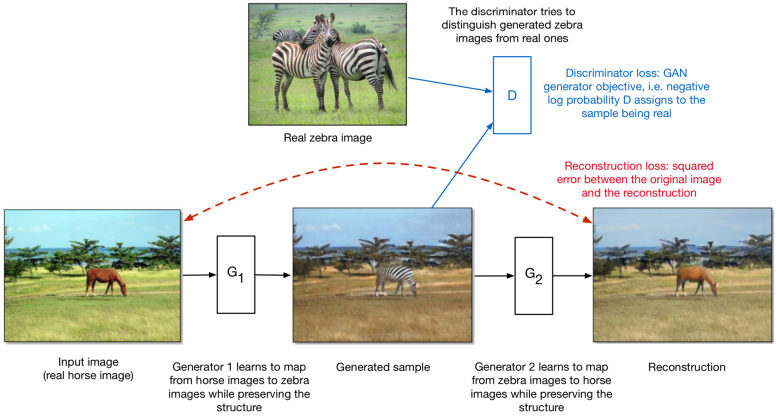
GAN example: Sketch to Figure

GAN example: Cycle GANs

- Image-to-image translation involves generating a new synthetic version of a given image with a specific modification, such as translating a summer landscape to winter

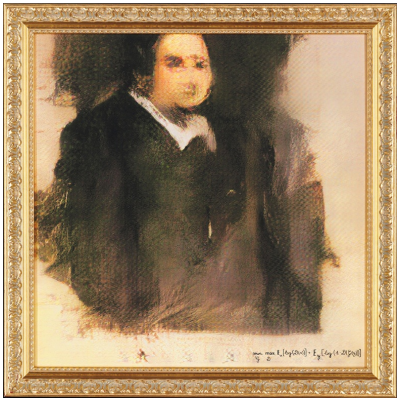
GAN example: Cycle GANs

GAN example: Cycle GANs



Total loss = discriminator loss + reconstruction loss

First AI-Generated Portrait Ever Sold at Auction Shatters Expectations: $432,500



Cosmopolitan Meets DALL-E 2

DALL-E 2: Text-to-Image Synthesis

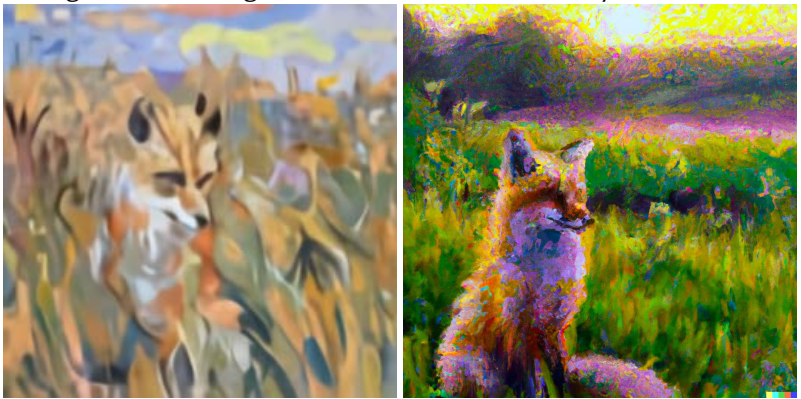
- DALL-E 2 is a new AI system that can create realistic images and art from a description in natural language. DALL-E 2 can create original, realistic images and art from a text description. It can combine concepts, attributes, and styles.
- DALL-E's uses multimodal version of GPT-3 with 12 billion parameters. The concept is that it swaps text for pixels and it is trained on trained on text-image pairs from the Internet. DALL-E 2 uses 3.5 billion parameters.

DALL-E 2

“a painting of a fox sitting in a field at sunrise in the style of Claude Monet”

DALL-E 2

"a painting of a fox sitting in a field at sunrise in the style of Claude Monet"



Challenges with GANs

Issues with GANs (Google's Comments)

- Research has suggested that if your discriminator is too good, then generator training can fail due to vanishing gradients. In effect, an optimal discriminator doesn't provide enough information for the generator to make progress.
  - The Wasserstein loss is designed to prevent vanishing gradients even when you train the discriminator to optimality.
- The generator may learn to produce a single output. The generator is always trying to find the one output that seems most plausible to the discriminator. This form of GAN failure is called *mode collapse*.
- GANs frequently fail to converge, as discussed in the module on training.

References

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio (2013)

Generative Adversarial Networks

Advances in Neural Information Processing Systems.

The End