Generative models for continuous random variables: GAN and GAIL

Eilif Solberg

TEK5040/TEK9040

Outline

Section 1

Introduction

Subsection 1

Why model of data distribution?

Why model of data distribution?

Analyzing data:

- Figure out the uncommon or rare elements
  - Anomalies, outliers, errors
- Find typical elements / prototypes

Prediction:

- How likely is something to happen?
- RL: If we can create model for environment, don't have to explore, but can just do planning.

Generalization: learning a familiy of conditional distributions.

- Recall classification: learned family of distributions $Y|X$ with shared parameters.

Why model of data distribution?

Analyzing data:

- Figure out the uncommon or rare elements
  - Anomalies, outliers, errors
- Find typical elements / prototypes

Prediction:

- How likely is something to happen?
- RL: If we can create model for environment, don't have to explore, but can just do planning.

Generalization: learning a familiy of conditional distributions.

- Recall classification: learned family of distributions $Y|X$ with shared parameters.

Both knowing how likely something is and being able to generate samples can be useful depending on the situation.

Subsection 2

Fitting a probability distribution

Fitting a probability distribution

Given data points

$$-5.17, -1.01, -2.43, -6.01, -4.16, 0.3, -7.85, -3.86, -0.47 \ldots$$

- How do we fit a probability distribution?

Fitting a probability distribution

Given data points

$$-5.17, -1.01, -2.43, -6.01, -4.16, 0.3, -7.85, -3.86, -0.47 \ldots$$

- How do we fit a probability distribution?
- What do we mean by fitting a probability distribution?

Solution I

Define parametric family of functions, $p_\theta, \theta \in \Theta$, and then find the parameters that maximizes the *likelihood* of the data, or equivalently, the log-likelihood

$$\text{argmax}_\theta \sum_{i=1}^{N} \log p_\theta(x_i)$$

Solution I

Define parametric family of functions, $p_\theta, \theta \in \Theta$, and then find the parameters that maximizes the *likelihood* of the data, or equivalently, the log-likelihood

$$\text{argmax}_\theta \sum_{i=1}^{N} \log p_\theta(x_i)$$

This minimizes the Kullback-Leibler divergence to the data distribution, i.e. $KL(p_{\text{data}} \parallel p_\theta)$.

Visualizing data distribution

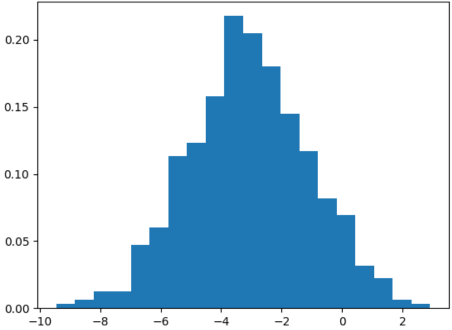


Figure: Histogram for data

Visualizing data distribution

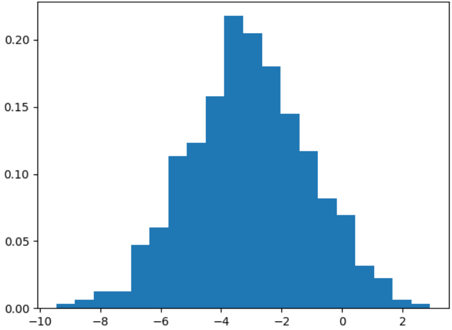


Figure: Histogram for data

- Normally distributed?

Fit distribution

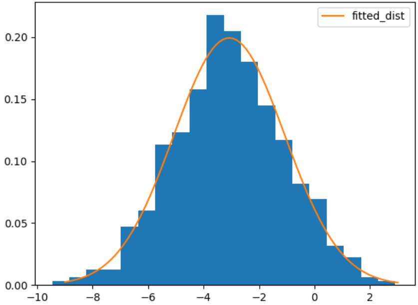
Assuming normal distribution

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad \hat{\sigma^2} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{\mu})^2$$

Fit distribution

Assuming normal distribution

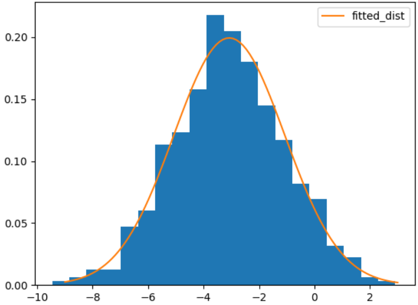
$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad \hat{\sigma^2} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{\mu})^2$$



Fit distribution

Assuming normal distribution

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad \hat{\sigma^2} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{\mu})^2$$



Generally may require iterative procedure to maximize likelihood.

Solution II

- Define a random variable Z on $\mathbb{R}^k$ for some k, e.g. $Z \sim \mathcal{N}(0, I)$. Let p_z denote distribution.
- Define $G : \mathbb{R}^k \to \mathbb{R}^d$
- Generate samples by
    1. Draw z from p_z.
    2. Map $z \to G(z)$.

Solution II continued

Advantages:

- Easy to generate samples
- Works even if X does not have density on $\mathbb{R}^d$.
- Can use complex functions, e.g. neural networks to represent distribution

Disadvantages:

- Not straightforward to find likelihood of samples.

Solution II continued

Advantages:

- Easy to generate samples
- Works even if X does not have density on $\mathbb{R}^d$.
- Can use complex functions, e.g. neural networks to represent distribution

Disadvantages:

- Not straightforward to find likelihood of samples.

How do we fit G though?

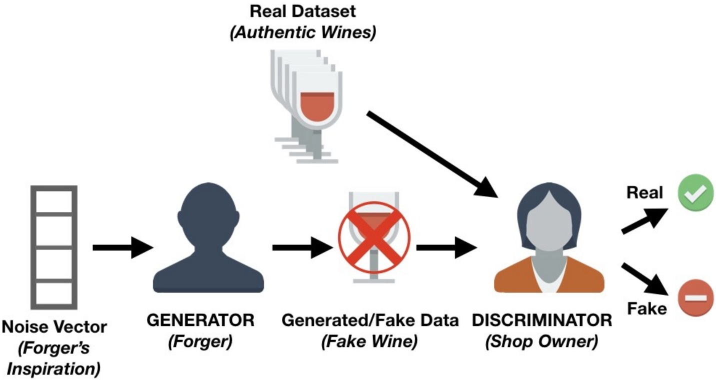
Section 2

Generative Adverserial Networks (GAN)

Subsection 1

Introduction

Analogy



https://towardsdatascience.com/demystifying-generative-adversarial-networks-c076d8db8f44

Adverserial networks

Generative Adversarial Nets (2014)

Adverserial networks

Generative Adversarial Nets (2014)

- Two networks: generator G and discriminator D.
- Discriminator: try to classify an input as real or fake (generated), outputs probability in $[0, 1]$, where 1 means real.
- Generator: try to fool discriminator
- Minimax game:

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

Minimax solutoin

Let p_g be the density function of the distribution induced by G and p_{data} be the density of data distribution[1]. Optimal D is given by

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$$

[1] We assume this exists here.

Minimax solutoin

Let p_g be the density function of the distribution induced by G and p_{data} be the density of data distribution[1]. Optimal D is given by

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$$

Inserting D^* in minimax equation and rewriting leads to

$$\min_G 2 * JSD(p_{\text{data}} \parallel p_g) - \log(4)$$

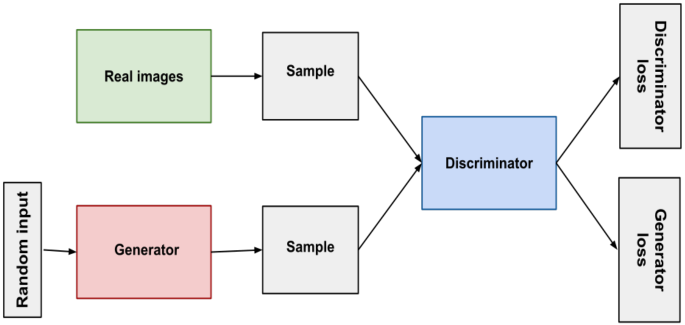
where JSD is the Jensen-Shannon divergence. The minimum value is achieved at $p_g = p_{\text{data}}$.

[1] We assume this exists here.

Subsection 2

Training

GAN overview



https://developers.google.com/machine-learning/gan/generator

Discriminator loss (minibatch)

Assume discriminator D parametrized by η.

$$-\Big(\frac{1}{m}\sum_{i=1}^{m}\log(D_\eta(x_i^{real})) + \frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(x_i^{fake}))\Big)$$

Discriminator loss (minibatch)

Assume discriminator D parametrized by η.

$$-\Big(\frac{1}{m}\sum_{i=1}^{m}\log(D_\eta(x_i^{real})) + \frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(x_i^{fake}))\Big)$$

which can also be written as

$$-\Big(\frac{1}{m}\sum_{i=1}^{m}\log(D_\eta(x_i)) + \frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(G_\theta(z_i)))\Big)$$

Discriminator loss (minibatch)

Assume discriminator D parametrized by η.

$$-\Big(\frac{1}{m}\sum_{i=1}^{m}\log(D_\eta(x_i^{real})) + \frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(x_i^{fake}))\Big)$$

which can also be written as

$$-\Big(\frac{1}{m}\sum_{i=1}^{m}\log(D_\eta(x_i)) + \frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(G_\theta(z_i)))\Big)$$

Note: This is just the normal cross-entropy loss.

Generator loss (minibatch)

Negative of discriminator loss

$$\frac{1}{m}\sum_{i=1}^{m}\log(D_\eta(x_i)) + \frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(G_\theta(z_i)))$$

Generator loss (minibatch)

Negative of discriminator loss

$$\frac{1}{m}\sum_{i=1}^{m}\log(D_\eta(x_i)) + \frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(G_\theta(z_i)))$$

As generator cannot influence first term, we may simplify to

$$\frac{1}{m}\sum_{i=1}^{m}\log(1 - D_\eta(G_\theta(z_i)))$$

Algorithm

Algorithm 1 GAN training, k is a hyperparameter (e.g. 1).

for number of training iterations **do**

  for k steps **do**

    Sample minibatch of m noise samples $\{z_1, \ldots, z_m\}$ from noise prior p_z.

    Sample minibatch of m examples $\{x_1, \ldots, x_m\}$ from data generating distribution $p_{\text{data}}(x)$.

    Update the discriminator by ascending its stochastic gradient:

$$\nabla_\eta \frac{1}{m} \sum_{i=1}^{m} \left[\log D_\eta\left(x_i\right) + \log\left(1 - D_\eta\left(G_\theta\left(z_i\right)\right)\right)\right].$$

  end for

  Sample minibatch of m noise samples $\{z_1, \ldots, z_m\}$ from noise prior p_z.

  Update the generator by descending its stochastic gradient:

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D_\eta\left(G_\theta\left(z_i\right)\right)\right)$$

end for

Training I

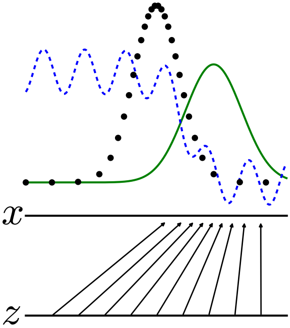
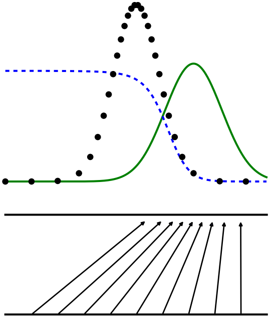
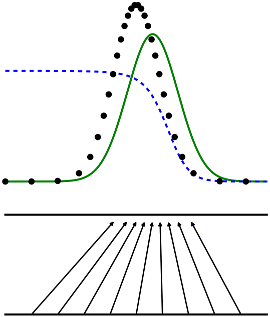


Figure: Green: p_g, black: p_{data}, blue: discriminator score.

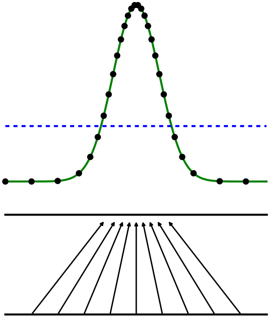
Training II



Training III



Training IV



Subsection 3

TensorFlow example

TensorFlow example I

```
import numpy as np
from scipy.stats import norm
import tensorflow as tf
from matplotlib import pyplot as plt
```

TensorFlow example II

```
class Generator(tf.keras.Model):
  def __init__(self):
    super(Generator, self).__init__()
    self.w = tf.Variable(1, dtype=tf.float32)
    self.b = tf.Variable(0, dtype=tf.float32)

  def call(self, z):
    x = self.w*z + self.b
    return x

class Discriminator(tf.keras.Model):
  def __init__(self, hidden_units=8):
    super(Discriminator, self).__init__()
    self.dense = tf.keras.layers.Dense(hidden_units)
    self.logits = tf.keras.layers.Dense(1,
    ↪  kernel_initializer=tf.keras.initializers.zeros())

  def call(self, x):
    x = tf.expand_dims(x, axis=-1)
    logits = self.logits(tf.nn.relu(self.dense(x)))
    logits = tf.squeeze(logits, axis=-1)
    p = 1 / (1 + tf.math.exp(-logits))
    return p
```

TensorFlow example III

```python
# parameters true distribution
mu = -4
sigma = 2

def visualize(G, D):
  interval = np.linspace(-10, 10, 100)
  d_values = D(interval)
  g_dist = norm.pdf(interval, loc=G.b.numpy(), scale=G.w.numpy())
  true_dist = norm.pdf(interval, loc=mu, scale=sigma)
  plt.plot(interval, true_dist, label="true_dist")
  plt.plot(interval, g_dist, label="G_dist")
  plt.plot(interval, d_values, label="D, p_true_data(x)")
  plt.legend()
  plt.show()
```

TensorFlow example IV

```
N = 32
x = np.random.normal(loc=mu, scale=sigma, size=N)
indices = np.array(range(N))

G = Generator()
D = Discriminator()

D_learning_rate = 0.1
G_learning_rate = 0.1
D_optimizer = tf.keras.optimizers.Adam(D_learning_rate)
G_optimizer = tf.keras.optimizers.Adam(G_learning_rate)

batch_size = 16
critic_iters = 1
iterations = 100
plot_interval = 1
```

TensorFlow example V

```
for iteration in range(iterations):
  if iteration % plot_interval == 0: visualize(G, D)

  # discriminator update
  for _ in range(critic_iters):
    # sample real data (from data distribution)
    np.random.shuffle(indices)
    real = x[indices[:batch_size]]
    z = tf.random.normal(shape=[batch_size])
    fake = G(z)
    with tf.GradientTape() as tape:
      loss_real = tf.reduce_mean(-tf.math.log(D(real)))
      loss_fake = tf.reduce_mean(-tf.math.log(1-D(fake)))
      D_loss = loss_real + loss_fake
    grads = tape.gradient(D_loss, D.trainable_variables)
    D_optimizer.apply_gradients(zip(grads, D.trainable_variables))

  # generator update
  z = tf.random.normal(shape=[batch_size])
  with tf.GradientTape() as tape:
    G_loss = tf.reduce_mean(tf.math.log(1-D(G(z))))
  grads = tape.gradient(G_loss, G.trainable_variables)
  G_optimizer.apply_gradients(zip(grads, G.trainable_variables))
```

Subsection 4

Does it work?

4 years of GAN progress

The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation

Does it end there?

Does it end there?

A Style-Based Generator Architecture for Generative Adversarial Networks

Subsection 5

Conditional GANs

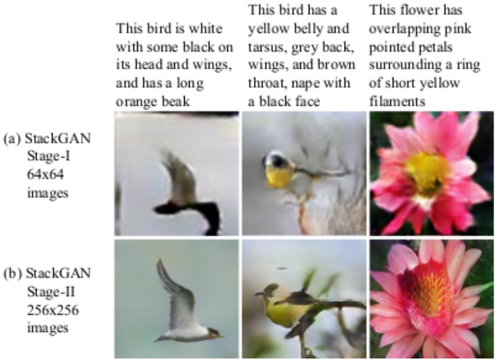
Conditional GANs

- Before: data were samples $x_1, x_2, \ldots, x_N$
- Now: data are sample pairs $(x_1, c_1), (x_2, c_2), \ldots, (x_N, c_N)$
- Generator and discriminator get c as extra input:
  - $G(z, c)$
  - $D(x, c)$

Subsection 6

Applications

Text-to-image



StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks

- (x, c) = (image, corresponding sentence)

Text-to-image II

StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks

Image impainting

Figure: Conditional image

Figure: L2 loss

Figure: Sample with GAN loss

Context Encoders: Feature Learning by Inpainting

- (x, c) = (image, image with missing data)

Super-resolution

Figure: Bicubic

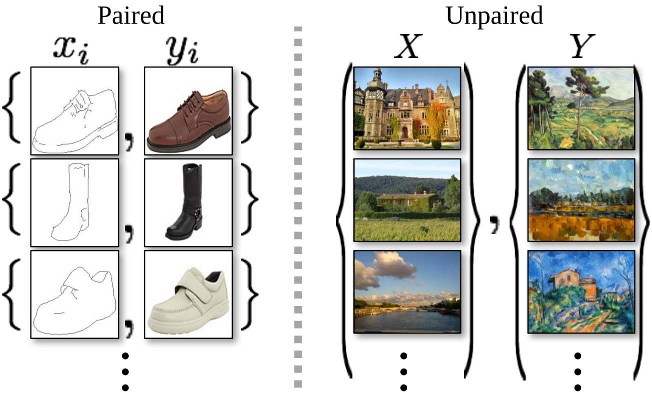
Figure: SRGAN

Figure: original

Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

- (x, c) = (image, lower resolution image)

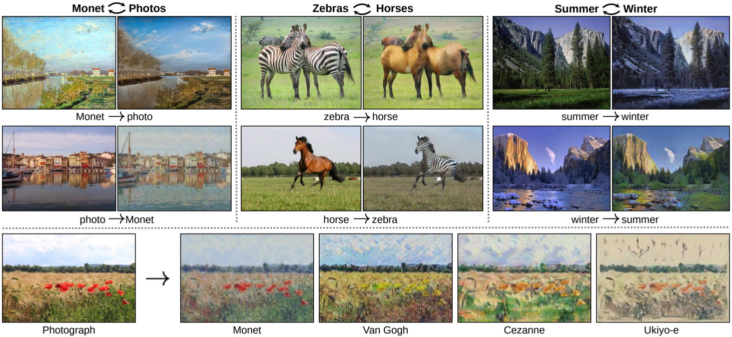
CycleGAN: unpaired image-to-image translation



Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

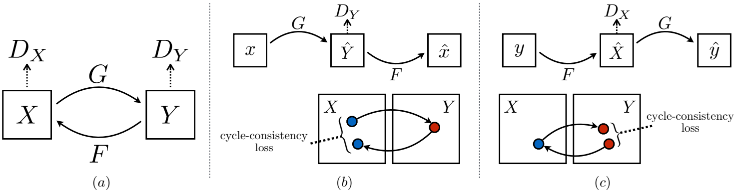
- TensorFlow tutorial on CycleGan

CycleGAN



Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

CycleGAN: how?



- Where is z?

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

CycleGAN: how?

- Two generators F and G, each with their own discriminator.
  - Takes samples from other distribution as input, not z!
- Learn F such that $x \sim X$, $F(x)$ should be distributed as Y.
- Learn G such that $y \sim Y$, $G(y)$ should be distributed as X.
- Need additional constraints to get *pairing* that we need for translation. Propose cycle-consistency:
  - Losses on $G(F(x)) - x$ and $F(G(y)) - y$
  - In order to reconstruct the information must be retained in the target domain, so should perhaps be a similar image?
    - Likely some conflict between the different goals. . .

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Subsection 7

Evaluation (not curriculum)

Challenges in evaluation

- No likelihood to evaluate

Challenges in evaluation

- No likelihood to evaluate
- Look at discriminator?
  - Optimal discriminator loss is $2 * JSD(p_{\text{data}}, p_g) - log(4)$ where JSD is the Jensen-Shannon divergence.

Challenges in evaluation

- No likelihood to evaluate
- Look at discriminator?
  - Optimal discriminator loss is $2 * JSD(p_{\text{data}}, p_g) - log(4)$ where JSD is the Jensen-Shannon divergence.
- Discriminator likely imperfect

Challenges in evaluation

- No likelihood to evaluate
- Look at discriminator?
  - Optimal discriminator loss is $2 * JSD(p_{\text{data}}, p_g) - log(4)$ where JSD is the Jensen-Shannon divergence.
- Discriminator likely imperfect
- A single number not enough? Quality vs diversity

Visual inspection

For image data we may look at images...

- Quality may be eaiser to evaluate then diversity

FID

- Extract *features* and estimate difference in distributions in these.
- Assuming features are normally distributed (quite strong assumption!), can measure Fréchet distance (also known as Wasserstein-2 distance), which is given by

$$d^2((m, C), (m_g, C_g)) = \|m - m_g\|_2^2 + trace(C + C_g - 2(CC_g)^{1/2})$$

where (m, C) and (m_g, C_g) are the mean and covariance of the features of the real and generated data respectively, and the *trace* of a matrix is the sum of its diagonal elements.

FID

- Extract *features* and estimate difference in distributions in these.
- Assuming features are normally distributed (quite strong assumption!), can measure Fréchet distance (also known as Wasserstein-2 distance), which is given by

$$d^2((m, C), (m_g, C_g)) = \|m - m_g\|_2^2 + trace(C + C_g - 2(CC_g)^{1/2})$$

where (m, C) and (m_g, C_g) are the mean and covariance of the features of the real and generated data respectively, and the *trace* of a matrix is the sum of its diagonal elements.

- To make comparable: Use Inception architecture with weights from `http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz`.
  - Known as Fréchet Inception Distance (FID)

GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium

FID example

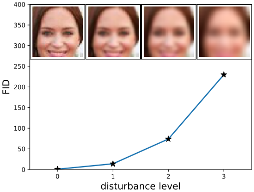


Figure: Blur

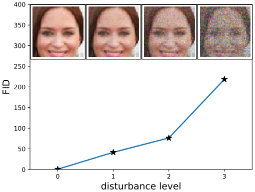


Figure: Gaussian noise

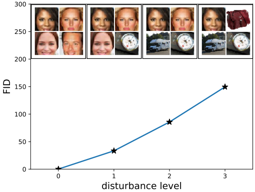


Figure: Mixed in ImageNet images

- Empirically proved to correlate well(?) with visual inspection.

GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium

Subsection 8

Challenges (not curriculum)

Challenges

- Not obvous how to track progress or measure performance
- Training can be unstable due to interactions between G and D
  - Input distribution of D changes over time
  - Loss function of G changes over time
- Choice of optimizer is important
  - Standard SGD not normally used, Adam popular
- Mode collapse - G only able to capture some of modes in data

Generator loss I

Let D_l denote the logits of D, i.e. $D(x) = \sigma(D_l(x))$ where σ is the sigmoid function.

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G_\theta(z_i))) = \frac{1}{m} \sum_{i=1}^{m} D(G_\theta(z_i)) \nabla_\theta D_l(G_\theta(z_i))$$

When generator is poor, may have $D(G_\theta(z_i)) \approx 0$, and thus gradients ≈ 0.

Generator loss I

Let D_l denote the logits of D, i.e. $D(x) = \sigma(D_l(x))$ where σ is the sigmoid function.

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G_\theta(z_i))) = \frac{1}{m} \sum_{i=1}^{m} D(G_\theta(z_i)) \nabla_\theta D_l(G_\theta(z_i))$$

When generator is poor, may have $D(G_\theta(z_i)) \approx 0$, and thus gradients ≈ 0. The original GAN paper recommends modified generator loss

$$-\frac{1}{m} \sum_{i=1}^{m} \log(D_\eta(G_\theta(z_i)))$$

to mitigate vanishing gradients issue.

Generator loss I

Let D_l denote the logits of D, i.e. $D(x) = \sigma(D_l(x))$ where σ is the sigmoid function.

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G_\theta(z_i))) = \frac{1}{m} \sum_{i=1}^{m} D(G_\theta(z_i)) \nabla_\theta D_l(G_\theta(z_i))$$

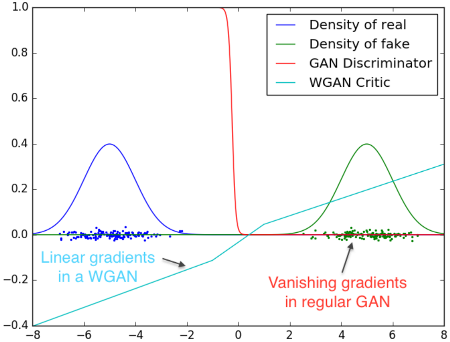
When generator is poor, may have $D(G_\theta(z_i)) \approx 0$, and thus gradients ≈ 0. The original GAN paper recommends modified generator loss

$$-\frac{1}{m} \sum_{i=1}^{m} \log(D_\eta(G_\theta(z_i)))$$

to mitigate vanishing gradients issue. This does however introduce other potential problems: Sample weighting as an explanation for mode collapse in generative adversarial networks

Loss function II

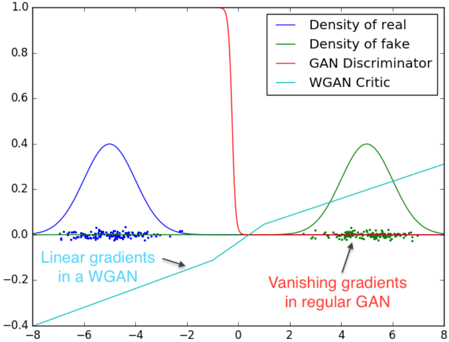
Original GAN loss have some theoretical (and practical?) issues



Wasserstein GAN

Loss function II

Original GAN loss have some theoretical (and practical?) issues



Wasserstein GAN

Note: regularizing discriminator can also mitigate problem.

Wasserstein loss

Wasserstein GAN discriminator loss:

$$-(\frac{1}{m}\sum_{i=1}^{m} D_\eta(x_i) - D_\eta(G_\theta(z_i))$$

- Note that D_η no longer outputs a probability, but any real number is valid.

Wasserstein GAN generator loss:

$$-\frac{1}{m}\sum_{i=1}^{m} D_\eta(G_\theta(z_i))$$

- Assumes some Lipschitz-constraints on D_η.

Wasserstein loss

Wasserstein GAN discriminator loss:

$$-(\frac{1}{m}\sum_{i=1}^{m} D_\eta(x_i) - D_\eta(G_\theta(z_i))$$

- Note that D_η no longer outputs a probability, but any real number is valid.

Wasserstein GAN generator loss:

$$-\frac{1}{m}\sum_{i=1}^{m} D_\eta(G_\theta(z_i))$$

- Assumes some Lipschitz-constraints on D_η.
- G now tries to minimize Wasserstein distance between generated distribution and data distribution.

Wasserstein GAN

Section 3

Generative Adverserial Imitation Learning (GAIL)

Subsection 1

Introduction

Why imitation learning?

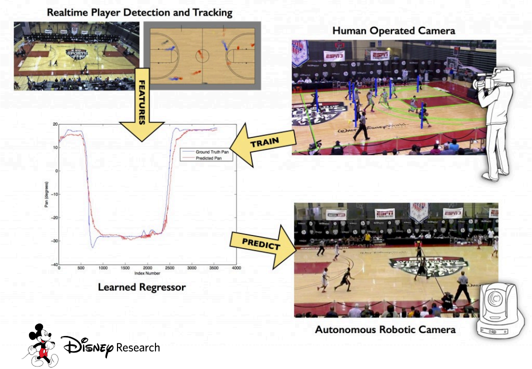


Figure: Source: New Frontiers in Imitation Learning.

Smooth Imitation Learning for Online Sequence Prediction

Imitation learning by behaviour cloning

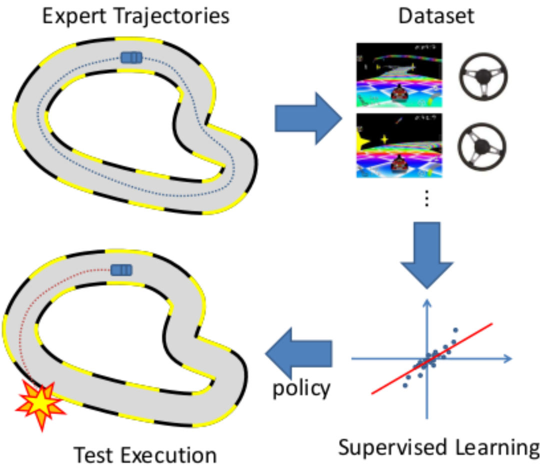


Figure: Source: Interactive Learning for Sequential Decisions and Predictions

Equivalence of policy and occupancy measure

Previously defined ρ_π as the unnormalized discounted visitation frequencies

$$\rho_\pi(s) = \sum_{t=0}^{\infty} \gamma^t P_\pi(S_0 = s)$$

Equivalence of policy and occupancy measure

Previously defined ρ_π as the unnormalized discounted visitation frequencies

$$\rho_\pi(s) = \sum_{t=0}^{\infty} \gamma^t P_\pi(S_0 = s)$$

Slightly abusing notation we define the *occupancy measure* as

$$\rho_\pi(s, a) = \pi(a|s) \sum_{t=0}^{\infty} \gamma^t P_\pi(S_0 = s)$$

Equivalence of policy and occupancy measure

Previously defined ρ_π as the unnormalized discounted visitation frequencies

$$\rho_\pi(s) = \sum_{t=0}^{\infty} \gamma^t P_\pi(S_0 = s)$$

Slightly abusing notation we define the *occupancy measure* as

$$\rho_\pi(s, a) = \pi(a|s) \sum_{t=0}^{\infty} \gamma^t P_\pi(S_0 = s)$$

It turns out that if we have a policy π' with the same occupancy measure as for π, i.e. $\rho_{\pi'} = \rho_\pi$, then $\pi' = \pi$.

GAIL

Generative Adversarial Imitation Learning (2016)

- To imitate a policy we may imitate state-action frequencies
- Data samples are (state, action) pairs from expert policy
  - $(s_1, a_1), (s_2, a_2), ..., (s_N, a_N)$
  - Note: not conditional GAN, $x_i = (s_i, a_i)$

GAIL

Generative Adversarial Imitation Learning (2016)

- To imitate a policy we may imitate state-action frequencies
- Data samples are (state, action) pairs from expert policy
  - $(s_1, a_1), (s_2, a_2), ..., (s_N, a_N)$
  - Note: not conditional GAN, $x_i = (s_i, a_i)$
- Generator is here a deterministic policy $\pi(s) \rightarrow a$. By interacting with the environment we get generated/"fake" state-action pairs $(s_1, a_1)_g, (s_2, a_2)_g, \ldots, (s_N, a_N)_g$.
- Discriminator takes state, action pairs and try to classify them as real or fake.

GAIL vs GAN

Differences from standard GAN:

- Sequential process
- Policy network don't have directly control over samples, interaction with (possibly stochastic) environment.
- Can't train "generator" by backpropagating through discriminator, instead trained with reinforcement learning with discriminator feedback as reward signal. E.g. generator gets high reward for samples the discriminator finds more real.
- Entropy loss term

GAIL vs GAN

Differences from standard GAN:

- Sequential process
- Policy network don't have directly control over samples, interaction with (possibly stochastic) environment.
- Can't train "generator" by backpropagating through discriminator, instead trained with reinforcement learning with discriminator feedback as reward signal. E.g. generator gets high reward for samples the discriminator finds more real.
- Entropy loss term

Discriminator update is unchanged.

GAIL algorithm

Algorithm 2 Generative adversarial imitation learning

1: **Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, η_0
2: **for** $i = 0, 1, 2, \ldots$ **do**
3: Sample trajectories $\tau_i \sim \pi_{\theta_i}$
4: Update the discriminator parameters from η_i to η_{i+1} with the gradient

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\eta \log(D_\eta(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_\eta \log(1 - D_\eta(s,a))] \tag{1}$$

5: Take a policy step from θ_i to θ_{i+1}, using the TRPO rule with cost function $\log(D_{\eta_{i+1}}(s,a))$. Specifically, take a KL-constrained natural gradient step with

$$\begin{aligned} &\hat{\mathbb{E}}_{\tau_i}\left[\nabla_\theta \log \pi_\theta(a|s) Q(s,a)\right] - \lambda \nabla_\theta H(\pi_\theta), \\ &\text{where } Q(\bar{s},\bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{\eta_{i+1}}(s,a)) \,|\, s_0 = \bar{s}, a_0 = \bar{a}] \end{aligned} \tag{2}$$

6: **end for**

Note: TRPO is an RL algorithm, you may switch this out with e.g. a PPO iteration (recall second RL lecture).