Unsupervised Learning

Latent Spaces and Generative models

Daniel Yukimura

yukimura@impa.br

October 3, 2018

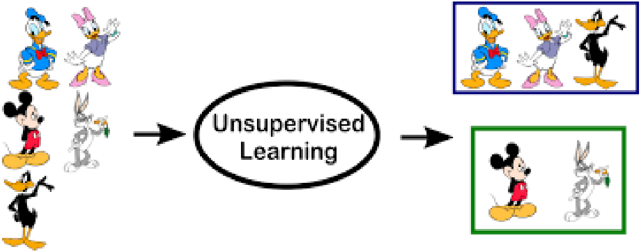
Unsupervised Learning

- **Motivation:** Most of the data available nowadays is actually unlabeled. This data coming from multiple sources might be hiding a lot of useful structure and statistical properties.
- **Usual strategy:** Dimensionality reduction, summarization, PCA,...
- **Modern strategy:** Generative modelling, statistical modelling, Deep Learning ...

Unsupervised Learning

Example: Clustering

Goal: Group similar elements of a dataset.
If each data point has a hidden class associated to it $(X_i, c_i) \in \mathcal{X} \times [k]$, we want to find these classes observing only $\mathcal{D} = \{X_i\}_{i=1}^n$.



Generative Modeling

- In Machine Learning the term **Generative Modeling(GM)** refers to methods for learning the probability distribution p_{data} from data examples.
- In a indirect form, we can learn a way of **sampling** from the distribution, without explicitly estimating it. This kind of GM we'll be of special interest.

Generative Modeling

Applications

- Image Synthesis - the photorealistic kind.
- Texture Synthesis.
- Super-resolution.
- Image-to-Image translation: Colorization, segmentation, photo generation from sketches, labels, edges, etc...
- Art generation

Latent Variable Models

Definition 1 (Latent Variable Models)

A latent variable model (LVM) p is a probability distribution over two sets of random variables V and H.

$$p(v, h) = \mathbb{P}(V = v, H = h)$$

- V are the **visible variables** (the ones observed at learning time in a data set).
- H are the **latent variables** (the ones representing underlying concepts).

Latent Variable Models

Example: Mixture models

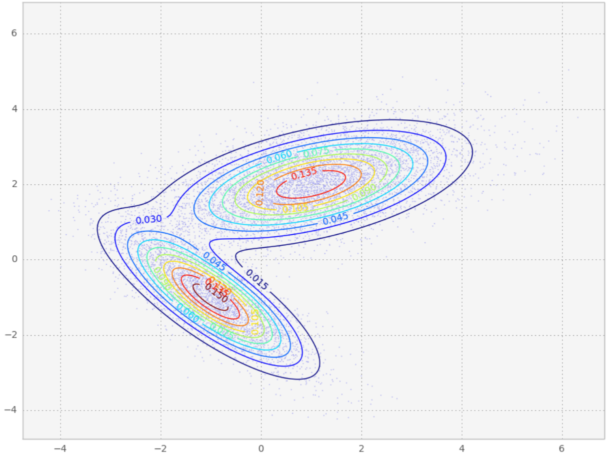
Here H is a discrete variable over the set $\{1, \ldots, m\}$ ($H \sim Cat(\pi)$).

$$p(v) = \sum_{k=1}^{m} \pi_k p(v|k) = \sum_{k=1}^{m} \pi_k p_k(v)$$

- **Gaussian Mixture:** Assume $p_k(v) = \mathcal{N}(v|\mu_k, \Sigma_k)$
- **Clustering from Mixtures:** (Bayes Rule)

$$\mathbb{P}(H = k|V = v; \theta) = \frac{p_H(k|\theta)p(v|k;\theta)}{\sum_{k'=1}^{m} p_H(k'|\theta)p(v|k';\theta)}$$

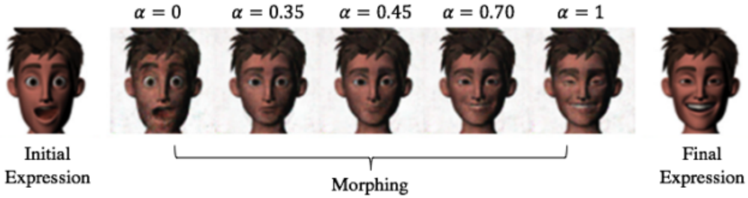
Example: Mixture Models



Latent Variable Models

Latent Spaces

The space where our latent variables $H \in \mathcal{H}$ live, is known as the **latent space**. Given good representations for these spaces and how they affect the distribution is very useful, as we'll see in the future.



Chen et al.

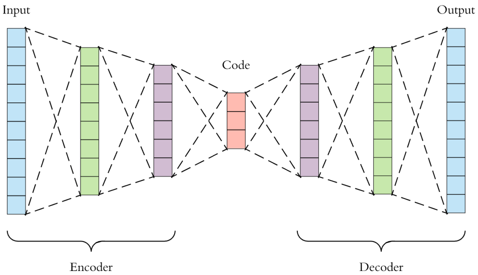
Example: Autoencoder

Autoencoders are a classical way of setting latent spaces.
We model two functions

$$f : \mathcal{X} \to \mathcal{H} \quad g : \mathcal{H}\mathcal{X}$$

respectively an encoder and a decoder, minimizing

$$\hat{f}, \hat{g} = \operatorname*{argmin}_{f,g} \|X - (g \circ f)(X)\|$$



Latent Variable Models

Generator Functions

Consider the latent variables as coming from a known distribution $H \sim p_H$ over the latent space $\mathcal{H}$. Then, a **generator function**

$$g : \mathcal{H} \to \mathcal{X} \tag{1}$$

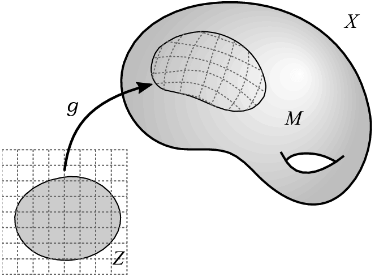
generates (or samples) X given H

$$X \stackrel{d}{=} g(H) \tag{2}$$

Latent Variable Models

Generator Functions

- g models a map of p_H into p_{data}.
- **Goal**: Given a sample $\{X_i\}_i \sim p_{data}$ produce a parametric generator function.



Generative Adversarial Networks

- **Goal:** Learning a generator function $G : \mathcal{H} \to \mathcal{X}$ parametrized by a neural network.
  We are not able to look at the latent variables, therefore how can we learn G?
- **Idea**: By pairing with another neural network $D : \mathcal{X} \to [0, 1]$, we can set an **adversarial training** framework.
- **Adversarial Training:** Design a game between machines where the equilibrium solves a learning problem.

Generative Adversarial Networks

- **Goal:** Learning a generator function $G : \mathcal{H} \to \mathcal{X}$ parametrized by a neural network.
  We are not able to look at the latent variables, therefore how can we learn G?
- **Idea**: By pairing with another neural network $D : \mathcal{X} \to [0, 1]$, we can set an **adversarial training** framework.
- **Adversarial Training:** Design a game between machines where the equilibrium solves a learning problem.

Generative Adversarial Networks

- **Goal:** Learning a generator function $G : \mathcal{H} \to \mathcal{X}$ parametrized by a neural network.
  We are not able to look at the latent variables, therefore how can we learn G?
- **Idea**: By pairing with another neural network $D : \mathcal{X} \to [0, 1]$, we can set an **adversarial training** framework.
- **Adversarial Training:** Design a game between machines where the equilibrium solves a learning problem.

Adversarial Training

Nash equilibrium:

To each player i we associate an strategy θ_i and a cost function $c_i(\theta)$. The **Nash equilibrium** is a special collection of strategies θ such that for each $i \in [n]$

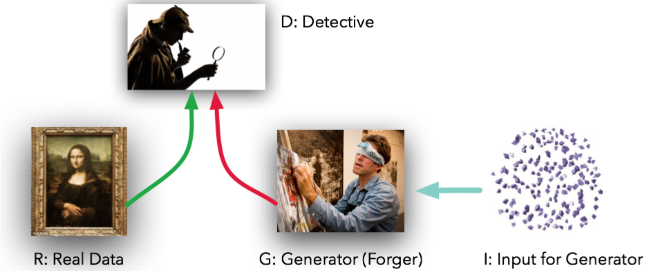
$$c_i(\theta) \leq c_i(\tilde{\theta}_i; \theta_{-i}), \tag{3}$$

is satisfied for all possible choices of $\tilde{\theta}_i$ and where θ_{-i} denotes θ minus coordinate i.

Generative Game

Players:

- **Generator**: $G : \mathcal{H} \to \mathcal{X}$.
- **Discriminator**: $D : \mathcal{X} \to [0, 1]$



Generative Game

Players:

- **Generator**: $G : \mathcal{H} \to \mathcal{X}$.
- **Discriminator**: $D : \mathcal{X} \to [0, 1]$

- G is a candidate for generative function.
- D distinguish if samples come from p_{data} or p_G.

$$c_D(G, D) = -\frac{1}{2} \left(\mathbb{E}[\log D(X)] + \mathbb{E}[\log(1 - D(G(H)))] \right) \quad (4)$$

where $H \sim p_H$ has a distribution we are able to sample.

Generative Game

Players:

- **Generator**: $G : \mathcal{H} \rightarrow \mathcal{X}$.
- **Discriminator**: $D : \mathcal{X} \rightarrow [0, 1]$

- G is a candidate for generative function.
- D distinguish if samples come from p_{data} or p_G.

$$c_D(G, D) = -\frac{1}{2}\left(\mathbb{E}[\log D(X)] + \mathbb{E}[\log\left(1 - D(G(H))\right)]\right) \tag{4}$$

where $H \sim p_H$ has a distribution we are able to sample.

Generative Game

- Considering a zero-sum game

$$c_G(G, D) = -c_D(G, D). \tag{5}$$

- The equilibrium coincides with the solution of the minmax problem

$$G^* = \underset{G}{\operatorname{argmin}} \max_D \mathbb{E}[\log D(X)] + \mathbb{E}[\log (1 - D(X_G))]. \tag{6}$$

Generative Game

- Considering a zero-sum game

$$c_G(G, D) = -c_D(G, D). \tag{5}$$

- The equilibrium coincides with the solution of the minmax problem

$$G^* = \operatorname*{argmin}_G \max_D \mathbb{E}[\log D(X)] + \mathbb{E}[\log(1 - D(X_G))]. \tag{6}$$

Generative Adversarial Networks

We model each player as a neural network

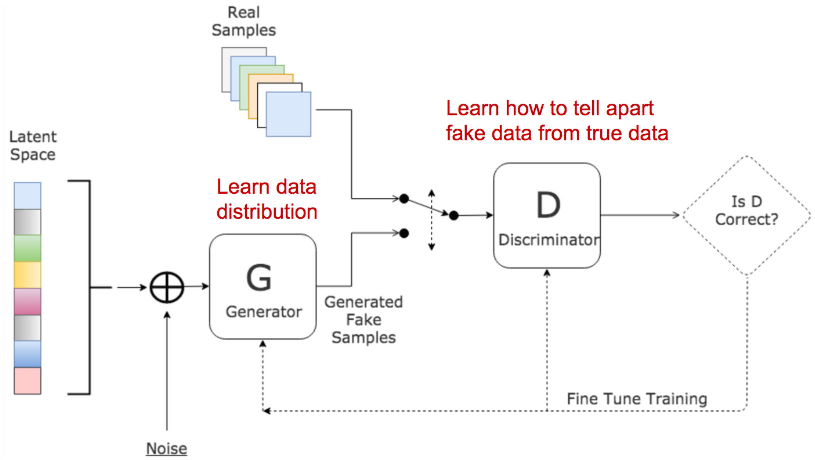
$$G(\cdot) = G(\cdot, \theta^G), \qquad D(\cdot) = D(\cdot, \theta^D) \tag{7}$$

Training GANs:

Using gradient descent (through backpropagation)

- Improve θ^G minimizing c_G.
- Improve θ^D minimizing c_D.

Generative Adversarial Networks



Applications

NEW! BigGAN

Large Scale GAN Training for High Fidelity Natural Image Synthesis, Brock et al. 2018

Algorithm: SGD for GANs

- **For** number of training iterations
  - **Sample** minibatch $\{h^{(1)}, \ldots, h^{(m)}\}$ from p_H.
  - **Sample** minibatch $\{x^{(1)}, \ldots, x^{(m)}\}$ from the data set.
  - **Update** the discriminator network D by the stochastic gradient:

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

  - **Sample** minibatch $\{h^{(1)}, \ldots, h^{(m)}\}$ from p_Z.
  - **Update** the generator network G by the stochastic gradient:

$$\nabla_{\theta^G} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(h^{(i)}\right)\right)\right)$$

- **Return** G

Modeling in PyTorch

```
h_dim, nb_hidden = 8, 100
batch_size, lr = 10, 1e-3

model_G = nn.Sequential(nn.Linear(h_dim, nb_hidden),
                        nn.ReLU(),
                        nn.Linear(nb_hidden, 2))

model_D = nn.Sequential(nn.Linear(2, nb_hidden),
                        nn.ReLU(),
                        nn.Linear(nb_hidden, 1),
                        nn.Sigmoid())
```

Modeling in PyTorch

```
...
optimizer_G = optim.Adam(model_G.parameters(), lr = lr)
optimizer_D = optim.Adam(model_D.parameters(), lr = lr)

for e in range(nb_epochs):
    for t, real_batch in enumerate(real_samples.split(batch_size)):
        z = real_batch.new(real_batch.size(0), z_dim).normal_()
        fake_batch = model_G(z)
        D_scores_on_real = model_D(real_batch)
        D_scores_on_fake = model_D(fake_batch)
        ...
```

Modeling in PyTorch

```
...
if t%2 == 0:
    loss = (1 - D_scores_on_fake).log().mean()
    optimizer_G.zero_grad()
    loss.backward()
    optimizer_G.step()
else:
    loss = - (1 - D_scores_on_fake).log().mean() \
    - D_scores_on_real.log().mean()
    optimizer_D.zero_grad()
    loss.backward()
    optimizer_D.step()
```

Generative Adversarial Networks

There are two pathological behaviors that often appear when training a standard GAN:

- The model doesn't converge, it keeps **oscillating**. There is no loss minimization, and there are no guarantees that the chosen procedure to reach the equilibrium in fact works.
- "**Mode Collapse**": When G models very well only a small sub-population, concentrating in modes that the discriminator can't tell its fake, but still doesn't represent the data.

Moreover, there are no standard metrics for performance or accuracy for generative models.