Stanford CS 236 Notes

Yikun Chi

September 26, 2022

Contents

Chapter 1

PyTorch Tutorials

1.1 Toy Example with PyTorch

Here is the setup:

$$
\begin{aligned}
x &= [x_1, x_2] \\
y &= [y_1, y_2] \\
z &= [x_1 y_1, x_2 y_2] \\
s &= x_1 y_1 + x_2 y_2
\end{aligned}
$$

We can find the derivative of s w.r.t x:

$$
\frac{\partial}{\partial x} s = [\frac{\partial}{\partial x_1} s, \frac{\partial}{\partial x_2} s] = [y_1, y_2]
$$

In PyTorch, this can be done as

```
x = torch.tensor([1,2], dtype = torch.float64)
y = torch.tensor([3,4], dtype = torch.float64)
x.requires_grad = True
y.require_grad = True

s = torch.sum(x * y)

s.backward()

x.grad
```

1.2 Reshaping

We can use tensor.view to reshape the tensor. For example, if I have a tensor of length 16, I can reshape it to $2 \times 2 \times 4$.

```
    x = torch.tensor([i for i in range(16)])
    print(x.shape)
    print(x.view(2,2,4))
```

1.3 Transfer between Devices

We can transfer a tensor from one device to the other.

```
    device - torch.device("cuda")
    y = torch.ones(2).to(device)
    z = torch.ones(2).cuda()
    x = z.cpu()
```

Chapter 2

Primer

2.1 Representing a Distribution

Goal
We want to learn a probability distribution $p(x)$ over images x such that

- Generation: if we sample x from the distribution, x should look like the training data
- Density estimation: p(x) should be high if x looks like training data, e.g.: a dog, and low otherwise
- Unsupervised representation learning: We should be able to learn what the input images have in common.

Simplifying Probability Distribution
Directly model joint distribution requires too many parameters. For example, modeling a single pixel's color will need $256^3 - 1$ number of parameters. So we want to simplify the parameters required through:

- chain rule: $p(x_1, ..., x_n) = p(x_1) * p(x_2|x_1) * p_(x_3|x_1, x_3) \cdots$
- Using DAG to represent a Bayesian network, introduce conditional independence. So we have $p(x|Parent(x))$ is independent from all others. Our joint distribution is $p(x_1, ..., x_n) = \prod_i p(x_i|Parent(x_i))$

2.2 Generative vs. Discriminative Model Example

2.2.1 Generative Model - Naive Bayes

Assume features are conditionally independent given the label, then we have

$$p(y, x_1, ..., x_n) = p(y) * \prod_{i=1}^{n} p(x_i|y)$$

We can estimate parameters from the training data, then predict with Bayes rule

$$p(Y = 1|x_1, ..., x_n) = \frac{p(Y = 1) \prod p(x_i|Y = 1)}{\sum_{y=\{0,1\}} p(Y = y) * \prod p(x_i|Y = y)}$$

We call this a generative model because we have learned the full distribution of the data and label. If we want, we can generate new data by sampling the distribution.

2.2.2 Discriminative Model - Logistic Regression

Discriminative model is just a different usage of chain rule. In chain rule, we have

$$\begin{aligned} p(Y, X) &= P(X|Y) * P(Y) && \text{(Generative Model)} \\ &= P(Y|X) * P(X) && \text{(Discriminative Model)} \end{aligned}$$

For discriminative model, we can say since X is normally given, we don't care about $P(X)$, so we only need to estimate $p(Y|X)$. In logistic regression, we model $Y = \sigma(AX + b)$

2.2.3 Geneative Model - Neural Model

Using Chain Rule, we have

$$p(x_1, x_2, x_3, x_4) = p(x_1) * p(x_2|x_1) * p(x_3|x_1, x_2) * p(x_4|x_1, x_2, x_3)$$

We already discussed it is possible to use Bayes net to simplify when assuming some conditional independence. In neural model approach, we use a neural network to approximate the conditional function. I.e.:

$$p(x_1, x_2, x_3, x_4) = p(x_1) * p(x_2|x_1) * p_{Nueral}(x_3|x_1, x_2) * p_{Nueral}(x_4|x_1, x_2, x_3)$$

Chapter 3

Autoregressive Model

Motivating Example: MNIST
Given a dataset D of handwritten digits, each image has $28 \times 28 = 784$ pixels. Each pixel is a binary variable. Goal is to learn a probability distribution $p(x) = p(x_1, ..., x_{784}$ such that when we sample $x \sim p(x)$, x looks like a digit.

3.1 Model Structure

We can pick an ordering of all random variable (e.g.: from top left corner to bottom right, raster scan). Now with chain rule, we have

$$p(x_1, ..., x_{784}) = p(x_1)p(x_2|x_1) \cdots p(x_n|x_1, ..., x_{n-1})$$

We choose to use a series of function with parameter $\alpha^{(n)}$ to estimate. e.g.:

$$p(x_1, ..., x_{784}) = p(x_1; \alpha^{(1)})p_{logit}(x_2|x_1; \alpha^{(2)}) \cdots p_{logit}(x_n|x_1, ..., x_{n-1}; \alpha^{(n)})$$

- $p(X_1 = 1) = \alpha^{(1)}$
- $p_{logit}(x_2 = 1|x_1, \alpha^{(2)}) = \sigma(\alpha_0^{(2)} + \alpha_1^{(2)} x_1)$

3.1.1 Example : Fully Visible Sigmoid Belief Netowrk (FVSBN)

The conditional variable $x_i|x_1, ..., x_{i-1}$ are Bernoulli with parameters. The approximation function is just the logit function.

$$\hat{x_i} = p(x_i = 1|x_1, ..., x_{i-1}; \alpha^{(i)}) = \sigma(\alpha_0^{(i)} + \sum_{j=1}^{i-1} \alpha_j^{(i)} x_j)$$

We can evaluate the joint probability by just multiplying everything together

$$p(x_1, x_2, x_3, x_4) = p(x_1) * p(x_2|x_1) * p(x_3|x_1, x_2) \cdots$$

For sampling, we just do sequential sampling. Pick x_1 first, and then x_2 and so on. Overall, we need $1 + 2 + 3 + ...n$, hence $O(n^2)$ number of parameter.

3.1.2 Example : Neural Autoregressive Density Estimation (NADE)

For NADE, we use a one layer neural network to replace logistic regression as estimation.

$$\hat{x}_i = p(x_i|x_1, ..., x_{i-1}; A_i, c_i, \alpha_i, b_i) = \sigma(\alpha_i(A_i x_{<i} + c_i) + b_i)$$

- $A_i x_{<i} + c_i$: the layer calculation
- in reality, we can tie the weight together. So A_2 has column w_1, then A_3, the network matrix for calculating x_3, will also have w_1, and an additional column w_2
- linear in n in terms of weights

3.1.3 Example: RENADE

For non-binary discrete random variable $X_i \in \{1, ..., K\}$, we can let $\hat{x}_i$ parameterize a categorical distribution.

$$\begin{aligned}
&h_i = \sigma(W_{<i} \cdot x_{<i} + c) \\
&p(x_i|x_1, ..., x_{i-1}) = Cat(p_i^{(1)}, ..., p_i^{(K)}) \\
&\hat{x}_i = (p_i^{(1)}, ..., p_i^{(K)}) = softmax(A_i h_i + b_i) \\
&softmax(a) = (\frac{exp(a^{(1)})}{\sum exp(a^{(i)})}, ..., \frac{exp(a^{(k)})}{\sum exp(a^{(i)})})
\end{aligned}$$

For continuous random variable, we want to let $\hat{x}_i$ parameterize a continuous distribution, such as mixture of K Gaussians, i.e.:

$$\begin{aligned}
&p(x_i|x_1, ..., x_{i-1}) = \sum_{j=1}^{K} \frac{1}{K} \mathcal{N}(\mu_i^j, \sigma_i^j) \\
&h_i = \sigma(W_{<i} \cdot x_{<i} + c) \\
&\hat{x_i} = (\mu_i^1, ..., \mu_i^K, \sigma_i^1, ..., \sigma_i^K) = f(h_i)
\end{aligned}$$

- Note here j is not power, just a notiation for parameters for x_j
- we can use exponential to ensure the stand deviation is positive.

3.1.4 Example: Masked Autoencoders for Distribution Estimation (MADE)

An autoencoder has two part, an encoder $e(\cdot)$, such as $e(x) = \sigma(W^2(W^1x + b^1) + b^2)$, and a decoer $d(\cdot)$ such that $d(e(X)) \approx x$. For loss function, we can use cross entroy for binary data $\sum_{x \in D} \sum_i -x_i log\hat{x}_i - (1 - x_i)log(1 - \hat{x}_i)$ and MSE for continuous data $\sum_{x \in D} \sum_i (x_i - \hat{x}_i)^2$.

Note a vanilla autoencoder is not a generative model. But we can modify the graph to make sure it corresponds to a valid Bayesian Network (DAG structure). For example, if the ordering is 1, 2, 3, then $\hat{x}_1$ can not depend on any input x, $\hat{x}_2$ can only depend on x_1, and etc.

We can achieve use this by using mask to disallow certain paths.

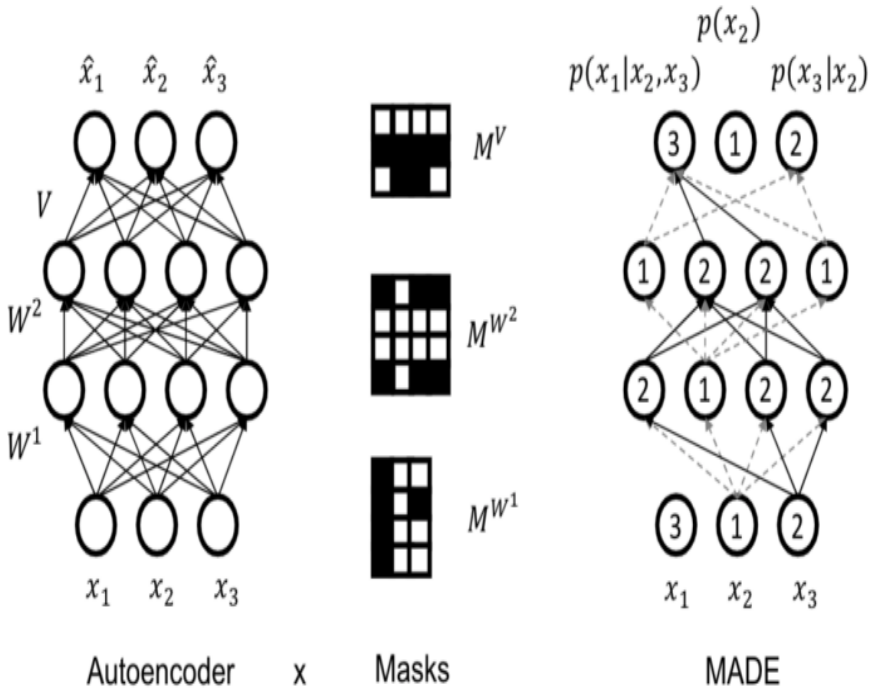


Figure 3.1: MADE Structure

In the structure of MADE, we first label all the input nodes $(1, 2, 3)$. Then for all the hidden nodes, we can arbitarily assign them a number $1 or 2$. If it is 1, it means that this node only allow to take input from node with label 1. If it is 2, it means this node are allowed to take input from nodes with label 1 and 2.

We then alter the mask for autoencoder to correspond to the disabled path.

3.1.5 Example: Character RNN

In the model $p(x_t|x_{1:t-1};\alpha^t)$, the history $x_{1:t-1}$ keeps getting longer. One idea is to keep a summary and recursively update it.

$$\begin{aligned} \text{Summary update rule: } & h_{t+1} = tanh(W_{hh}h_t + W_{xh}x_{t+1}) \\ \text{Prediction: } & o_{t+1} = W_{hy}h_{t+1} \\ \text{Summary Initialization: } & h_0 = b_0 \end{aligned}$$

- Hidden layer h_t is a summary of the inputs seen till time t
- Output layer o_{t-1} specifies parameters for conditional $p(x_t|x_{1:t-1})$
- Parameterized by b_0 (initialization), and matrices W_{hh}, W_{xh}, W_{hy}. Constant number of parameters w.r.t n.

3.1.6 Additional Examples

Some additional advanced example:

- We can replace RNN with Transformer. It has attention mechanisms to adaptively focus on relevant context, and avoid recursive computation. It needs mased self-attention to preserve autoregressive structure
- Pixel RNN uses LSTMs + masking
- Pixel CNN uses CNN with masking.

3.2 Training Generative Model with Maximum Likelihood Learning

Setting Assume the domain is governed by some underlying distribution P_{data}. We are given a dataset D of m samples from P_{data}. The standard assumption is *iid* samples. We are also given a family of models M, and our task is to learn some good model that defines a distribution $p_{\hat{M}}$. We treat it as a density estimation (learning the full distribution).

3.2.1 Kullback-LeiblerL Divergence

KL Divergence can measure the distance between true distribution p_{data} and the distribution from our fitted model family q.

$$\begin{aligned} D(p||q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\ &= E_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right] \\ &= E_{x\sim P}\left[\log p(x)\right] - E_{x\sim P}\left[\log q(x)\right] \end{aligned}$$

- $d(p||q) \geq 0$ for all p, q
- $D(p||q) = 0$ iff $p = q$
- KL Divergence is asymmetric. I.e.: $D(p||q) \neq D(q||p)$
- Measures the expected number of extra bits required to describe samples from $p(x)$ using a code based on q instead of p
- Notice at line 3 the first term is not dependent on p_{data}, so $\overset{argmin}{p_\theta} D(P_{data}||P_\theta) = \overset{argmin}{p_\theta} - E_{x\sim P_{data}}(\log p_\theta(x)) = \overset{argmax}{p_\theta} E_{x\sim P_{data}}(\log p_\theta(x))$
- i.e.: We ask P_θ to assign the high probability to instances sampled from P_{data}, this is a maximum likelihood estimation.

3.2.1.1 Using Jensen Inequality to Prove Non-negative of KL Divergence

$$\begin{aligned} D(p||q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\ &= E_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right] \\ &= E_{x\sim P}\left[-\log \frac{q(x)}{p(x)}\right] \\ &\geq -\log E_{x\sim P}\left[\frac{q(x)}{p(x)}\right] && \text{(Concavity of Log function)} \\ &= -\log\left(\sum_x p(x)\frac{q(x)}{p(x)}\right) \\ &= -\log\left(\sum_x q(x)\right) \\ &= -\log\left(1\right)) \\ &= 0 \end{aligned}$$

3.2.2 Maximum Likelihood

Because we don't know P_{data}, we have to appxomiate the expected log-likelihood $E_{x\sim P_{data}}[\log P_\theta(x)]$ with the empirical log-likelihood:

$$E_D[\log P_\theta(x)] = \frac{1}{|D|}\sum_{x\in D} logP_\theta(x)$$

So the maximum likelihood learning is

$$\overset{max}{P_\theta}\frac{1}{|D|}\sum_{x\in D} logP_\theta(x)$$

This is equivalent to getting a distribution to assign high probability to the data (notice if we take the log on both side in below equation, we get our original objective)

$$P_\theta(x^{(1)}, x^{(2)}, ..., x^{(m)}) = \prod_{x\in D} P_\theta(x)$$

3.2.3 Monte Carlo Estimation

We can use Monte Carlo Estimation to find the estimate of a function w.r.t a distribution when we don't know the distribution and only have samples from it.

1. Express the quantity of interest as the expected value of a random variable: $E_{x\sim P}[g(x)] = \sum_x g(x)p(x)$
2. Generate T samples from the distribution P with respect to which the expectation was taken
3. estimate the expected value from the sampling using $\hat{g}(x^1, ..., x^T) \overset{\triangle}{=} \frac{1}{T}\sum_{t=1}^{T} g(x^t)$

- Notice here $\hat{g}$ is the a random variable.
- $\hat{g}$ is unbiased. i.e.: $E_p[\hat{g}] = E_p[g(x)]$
- By law of large numbers, $\hat{g}$ converges
- Its variance is $\frac{V_p[g(x)]}{T}$. So as sample increases, the variance will decrease.

3.2.4 MLE Scoring for Autoregressive Model

Given an autoregressive model with n variables and factorization, for a single x sample, we can write it's probability as

$$p_\theta(x) = \prod_{i=1}^{n} p_{neural}(x_i|x_{<i};\theta_i)$$

So the overall likelihood of the entire dataset is

$$L(\theta, D) = \prod_{j=1}^{m} P_\theta(x^{(j)}) = \prod_{j=1}^{m}\prod_{i=1}^{n} p_{neural}(x_i^{(j)}|x_{<i}^{(j)};\theta_i)$$
$$l(\theta) = \sum_{j=1}^{m}\sum_{i=1}^{n} \log p_{neural}(x_i^{(j)}|x_{<i}^{(j)};\theta_i)$$

We then use numerical solution (SGD) to find the max of this likelihood or the log likelihood.

- Notice that for a specific θ_i, say θ_t, $\nabla_{\theta_t} l(\theta) = \sum_{j=1}^{m} \nabla_{\theta_t} \sum_{i=1}^{n} \log p_{neural}(x_i^{(j)}|x_{<i}^{(j)};\theta_i) = \sum_{j=1}^{m} \nabla_{\theta_t} \log p_{neural}(x_t^{(j)}|x_{<t}^{(j)};\theta_t)$. Hence the parameters for the t' th conditional parameters only appears in the t'th conditional. Without parameter sharing, we are essentially optimizing each conditional parameter θ_i separately.

What if $m = |D|$ is huge, we can use SGD instead of GD

$$\begin{aligned}\nabla_\theta(l(\theta)) &= \sum_{j=1}^{m}\sum_{i=1}^{n} \nabla_\theta \log p_{neural}(x_i^{(j)}|x_{<i}^{(j)};\theta_i) \\ &= m * \sum_{j=1}^{m} * \frac{1}{m} \sum_{i=1}^{n} \nabla_\theta \log p_{neural}(x_i^{(j)}|x_{<i}^{(j)};\theta_i) \\ &= m * E_{x^{(j)}\sim D}\left[\sum_{i=1}^{n} \nabla_\theta \log p_{neural}(x_i^{(j)}|x_{<i}^{(j)};\theta_i)\right]\end{aligned}$$

Estimate the expectation by taking a sample from the dataset and evaluate the gradient

3.2.5 Overfitting

Empirical risk minimization can easily overfit the data. The extreme example is that the model memerize the data, and have uniform probability over the training set. So we typically restrict the hypothesis space of distribution that we search over or add the regularization term to bias the model to simpler structure.

- Smaller neural networks with less parameters
- weight sharing
- conditional independence assumption
- Regularization

Chapter 4

Latent Variable Models

Motivation: We want to use neural networks to model the conditionals distribution ($p(x|z)$), z is the latent variables. We sample the latent variable z from some prior, then generate data x based on trained neural networks and the latent variable input. The hope is that after training, z will correspond to a meaningful latent factors of variation (features). Feature can be computed via $p(z|x)$ and we can then move on to other tasks such as $p(y|z)$

4.1 Primer: Mixture of Gaussians: Shallow Example

Bayes net is defined as $z \to x$. $z \sim (1, 2, ..., K)$ and $p(x|z = k) \sim \mathcal{N}(\mu_k, \Sigma_k)$.

The generative process is to pick a mixture componenet k by sampling z, and then sample x from the conditional distribution.

4.2 Variation autoencoder

VAE is a mixture of an infinite number of Gaussian.

$$Z \sim N(0, 1)$$
$$p(x|z) = \mathcal{N}\left(\mu_\theta(z)\right), \Sigma_\theta(z))$$

- Note here $\mu_\theta, \Sigma_\theta$ are neural networks with z as input. We may want to use exp to modify the output of Σ neural networks to make sure the network output is positive semi definite.
- Z are unobserved at train time
- Suppose we have a model for the joint distribution. Given training data $\tilde{x}$, the likelihood $p(X = \tilde{x}; \theta)$ is $\int_z p(X = \tilde{x}, Z = z; \theta)dz$ is often hard to calculate.

4.3 MLE Learning Problem with VAE and ELBO

4.3.1 Marginal Likelihood Problem

Given joint distribution $p(X, Z; \theta)$. We have a dataset D where for each datapoint the X varibales are observed and the variable Z are never observed. The maximum likelihood learning requires

$$\log \prod_{x \in D} p(x; \theta) = \sum_{x \in D} \log p(x; \theta) = \sum_{x \in D} \log \int_z p(X = x, Z = z; \theta) dz$$

This marginalizing over all possible value of z in reality are difficult and intractable.

4.3.2 Naive Monte Carlo

As discussed above, getting marginal distribution of $p(x)$ is hard. So can we just do naive monte carlo sampling?

$$\begin{aligned} p_\theta(x) &= \sum_{z \in Z} p_\theta(x, z) \\ &= |\mathcal{Z}| \sum_{z \in \mathcal{Z}} \frac{1}{|\mathcal{Z}|} p_\theta(x, z) \\ &= |\mathcal{Z}| E_{z \sim Uniform(\mathcal{Z})}[p_\theta(x, z)] \end{aligned}$$

We can think of it as an intractable expectation, and we can use monte carlo to estimate.

- Sample $z^{(1)}, z^{(2)}, z^{(3)}, ...$ uniformly at random
- Approximate expectation with sample average $\sum_z p_\theta(x, z) \approx |\mathcal{Z}| \frac{1}{k} \sum_{i=1}^{k} p_\theta(x, z^{(i)})$
- In theory this works, but in practice it doesn't work. Because for most of the z, $p_\theta(x, z)$ is very low. So the model variance will be huge. So we need a clever way to select $z^{(1)}$ to reduce variance of the estimator

4.3.3 Importance Sampling

We use importance sampling to improve on naive monte carlo by sampling from another distribution q instead of uniformally at random

$$\begin{aligned} p_\theta(x) &= \sum_z p_\theta(x, z) \\ &= \sum_z \frac{q(z)}{q(z)} p_\theta(x, z) \\ &= E_{z \sim q(z)}[\frac{p_\theta(x, z)}{q(z)}] \end{aligned}$$

So now we can use Monte Carlo, but sample from $q(z)$

- Sample $z^{(1)}, z^{(2)}, z^{(3)}, ...$ from $q(z)$
- Approximate expectation with sample average $p_\theta(x) \approx \frac{1}{k} \sum_{j=1}^{k} \frac{p_\theta(x, z^{(j)})}{q(z^{(j)})}$

4.3.4 Evidence Lower Bound (ELBO)

We have shown that:

$$p_\theta(x) \approx \frac{1}{k} \sum_{j=1}^{k} \frac{p_\theta(x, z^{(j)})}{q(z^{(j)})}$$

In MLE learning setting, we care about the log likelihood, so naively we hope that

$$\log(p_\theta(x)) = \log\left(E_{z\sim q(z)}[\frac{p_\theta(x,z)}{q(z)}]\right) \approx \log\left(\frac{1}{k}\sum_{j=1}^{k}\frac{p_\theta(x,z^{(j)})}{q(z^{(j)})}\right)$$

However, we can image if we only take 1 sample, i.e. $k = 1$, our expected value is

$$E_{z^{(1)}\sim q(z)}[\log(\frac{p_\theta(x,z^{(1)})}{q(z^{(1)})})]$$

But what we care about is

$$\log\left(E_{z^{(1)}\sim q(z)}[\frac{p_\theta(x,z^{(1)})}{q(z^{(1)})}]\right)$$

Formally, the problem is that

$$\begin{aligned}
\log(\sum_{z\in\mathcal{Z}} p_\theta(x,z)) &= \log\left(\sum_{z\in\mathcal{Z}}\frac{q(z)}{q(z)}p_\theta(x,z)\right) \\
&= \log E_{z\sim q(z)}\left[\frac{p_\theta(x,z)}{q(z)}\right] \\
&= \log E_{z\sim q(z)}\left[f(z)\right] \qquad (f(z) = \tfrac{p_\theta(x,z)}{q(z)}) \\
&= \log(\sum_z q(z) * f(z)) \\
&\geq \sum_z q(z)\log f(z)
\end{aligned}$$

(The expectation (estimated by sample mean) of the log)

This is due to the log being a concave function. So essentially, we have the following lower bound:

$$\log\left(E_{z\sim q(z)}[\frac{p_\theta(x,z)}{q(z)}]\right) \geq \ E_{z\sim q(z)}\left[\log\frac{p_\theta(x,z)}{q(z)}\right]$$

- On the left is our target, the log of the marginal likelihood of the data
- On the right is our approximation, the empirical average from monte carlo simulation by sampling from $q(z)$
- So the true likelihood is at least better than the likelihood estimated from monte carlo

To summarize, suppose $q(z)$ is any probability distribution over the hidden variables. Given the ELBO holds for any q, we have

$$\begin{aligned}
&\log p(x;\theta) \\
&\geq \sum_z q(z)\log(\frac{p_\theta(x,z)}{q(z)}) && \text{(Empirical estimation using monte carlo)} \\
&= \sum_z q(z)\log p_\theta(x,z) - \sum_z q(z)\log q(z) \\
&= \sum_z q(z)\log p_\theta(x,z) + H(q) && \text{(ELBO)}
\end{aligned}$$

So the takeaway is we want to pick q such that it has high entropy, and that put a lot of probability mass on z that is consistent with given data x.

The inequality become equality if $q = p(z|x;\theta)$. So intuitively, we want to sample likely completion, i.e.: $q = p(z|x)$, but this posterior could be very hard to compute because we have to invert the neural network.
We use KL Divergence to measure the distance between an arbitrary $q(z)$ and ideal $q(z) = p(z|x)$

$$D_{KL}(q(z)||p(z|x;\theta)) = -\sum_z q(z)\log p(z,x;\theta) + \log p(x;\theta) - H(q) \geq 0$$

We want to pick a model family e.g.: $q(z;\phi) = \mathcal{N}(\phi_1,\phi_2)$ and make it as close to the true $p(z|x;\theta)$ as possible. The smaller the KL divergence we can achieve, the close ELBO will be to the actual log likelihood $\log p(x;\theta)$

Another way to see is that in general:

$$\log p(z,x;\theta) = ELBO + D_{KL}(q(z)||p(z|x;\theta))$$

4.3.5 Example / Another Perspective

Given an image, assume the top part is missing (think of it as latent variable). Assume $p(z,x;\theta)$ is clsoe to $p_{data}(z,x)$ (This is very important, and not true at the beginning of learning). Suppose $q(z;\phi)$ is tractable probability distribution

over the hidden variables z parameterized by ϕ. (Assume latent variable is binary)

$$p(z;\phi) = \prod_{\text{unobserved variable } z_i} (\phi_i)^{z_i}(1-\phi_i)^{1-z_i}$$

- Given the bottom image, at least intuitively we know $\phi_i = 0.5$ is not a good idea.

We want to jointly optimize θ and ϕ by maximizing likelihood of the data.

4.4 Variational Learning

The ELBO holds for any $q(z;\phi)$

$$\log p(x;\theta) \geq \sum_z q(z;\phi)\log p(z,x;\theta) + H(q(z;\phi)) = L(x;\theta,\phi)$$

- Note that given z, the probability of x is easy to calculate due to VAE assumption.

Ideally, we want to use maximum likelihood learning over the entire dataset

$$\begin{aligned} l(\theta;D) &= \sum_{x^i \in D} \log p(x^i;\theta) && \text{(Ideal goal, but this is hard to compute)} \\ &\geq \sum_{x^i \in D} L(x^i;\theta,\phi^i) && \text{(Use ELBO to approximate)} \end{aligned}$$

- So the optimal loss we can achieve on the original log likelihood is definitely better than the likelihood we can achieve on the ELBO (i.e.: if ELBO likelihood is good enough, then the true likelihood is even better).
- Note we use different variational parameters ϕ^i for every data point x^i because the true posterior $p(z|x^i;\theta)$ is different across data points. (Think of an image, if the top missing part is the latent variable, each image has a different top).

4.4.1 Learning via Stochastic Variational Inference

Motivating Example
MNIST dataset. Assume $p(z,x;\theta)$ is clsoe to $p_{data}(z,x)$. Let's say z is now the underlying digits, hence $z \in \{0,1,...,9\}$. Let $q(z;\phi^i)$ be a categorical probability distribution over z, parameterized by $\phi^i = [p_0,...,p_9]$

$$q(z;\phi^j) = \prod_{k\in\{0,...,0\}} (\phi_k^i)^{\mathbb{1}[z=k]}$$

Note we want to optimize variational parameter per data point. So for an image of 1, we want the categorical probability distribution to put a lot of focus on p_1

Learning via stochastic variational inference (SVI)
We can optimize $\sum_{x^i \in D} L(x^i; \theta, \phi^i)$ as a function of $\theta, \phi^1, ..., \phi^M$ using SGD

$$\begin{aligned} L(x^i; \theta, \phi^i) &= \sum_z q(z; \phi^i) \log p(z, x^i; \theta) + H(q(z; \phi^i)) \\ &= E_{q(z;\phi^j)}[\log p(z, x^i; \theta) - \log q(z; \phi^i)] \end{aligned}$$

The process will be

1. Initialize $\theta, \phi^1, ..., \phi^M$
2. Randomly sample a data point x^i from D
3. Optimize $L(x^i; \theta, \phi^i)$ as a function of ϕ^i:
   (a) Repeat until convergence $\phi^i = \phi^i + \eta \nabla_{\phi^i} L(x^i; \theta, \phi^i)$ to get ϕ^{i*}
   (b) If computationally expensive, we can just choose to take some steps, instead of go all the way to convergence.
4. Compute $\nabla_\theta L(x^i; \theta, \phi^{i*})$
5. Update θ in the gradient direction, go back to step 2

But this process is not guaranteed to converge. (We can make progress to lower bound, but our goal is still bad since we are trying to maximize the goal and the goal's lower bound at the same time)

Also note that this loss L is an expectation. So we cannot directly compute it. We need to do Monte Carlo sampling again.

4.4.2 Calculating Gradient for conditional distribution parameter theta

Now let's say we already sampled a data $x \in D$. So everyting below is for a specific sampled given x. We use Monte Carlo sampling to compute the gradient because in the loss function there is an expectation and it has no closed form solution.

To evaluate the bound, we sample $z^1, ..., z^k$ from $q(z; \phi)$, and estimate

$$E_{q(z;\phi)}[\log p(z, x; \theta) - \log q(z; \phi)] \approx \frac{1}{k} \sum_{i=1}^{k} \log p(z^i, x; \theta) - \log q(z^i; \phi))$$

z^k is not the power, just the ith sampled z

The gradient w.r.t θ is easy

$$\begin{aligned}\nabla_\theta E_{q(z;\phi)}[\log p(z,x;\theta) - \log q(z;\phi)] &= E_{q(z;\phi)}[\nabla_\theta \log p(z,x;\theta)] \\ &= \frac{1}{k}\sum_{i=1}^{k} \nabla_\theta \log p(z^i,x;\theta) \\ &\qquad \text{(Monte Carlo again)}\end{aligned}$$

- Expectation is w.r.t z, which does not depend on θ, so we can bring the gradient operator ∇_θ inside the expectation

4.4.3 Reparameterization in Gradient Calculation

The gradient w.r.t ϕ is not so easy because the expectation depend on ϕ. So when we are changing ϕ, the expectation w.r.t z, which is drawn from a distribution depended on ϕ. So we introduce reparameterization techniques.

Abstractly, we want to compute a gradient w.r.t ϕ of

$$E_{q(z;\phi)}[r(z)] = \int q(z;\phi) r(z) dz$$

where z is now continuous.
Suppose $q(z;\phi) = \mathcal{N}(\mu, \sigma^2 I)$ with parameters $\phi = (\mu, \sigma)$ there are two equivalent way of sampling

- Sample $z \sim q_\phi(z)$
- Sample $\epsilon \sim \mathcal{N}(0,1)$, then $z = \mu + \sigma\epsilon = g(\epsilon;\phi)$

Using this equivalence, we can transform the expectation:

$$E_{z\sim q(z;\phi)}[r(z)] = E_{\epsilon\sim\mathcal{N}(0,1)}[r(g(\epsilon;\phi))] = \int p(\epsilon) r(\mu + \sigma\epsilon) d\epsilon$$

So now, we can compute the gradient as below:

$$\begin{aligned}\nabla_\phi E_{z\sim q(z;\phi)}[r(z)] &= \nabla_\phi E_\epsilon[r(g(\epsilon;\phi))] \\ &= E_\epsilon[\nabla_\phi r(g(\epsilon;\phi))] \quad \text{(Expectation does not depend on } \phi) \\ &\approx \frac{1}{k}\sum_{i=1}^{k} \nabla_\phi r(g(\epsilon^i;\phi)) \quad \epsilon^i \sim \mathcal{N}(0,1) \\ &\qquad \text{(Monte Carlo by sampling } \epsilon)\end{aligned}$$

Now if r and g are differentiable w.r.t ϕ and ϵ is easy to sample from, then we can run the Monte Carlo.

4.4.4 Calculating Gradient for ELBO parameter phi

We have : (remember below is all for a sampled $x \in D$, we are optimize ϕ for each data point x)

$$\begin{aligned} L(x;\theta,\phi) &= \sum_z q(z;\phi)\log p(z,x;\theta) + H(q(z;\phi)) \\ &= E_{q(z;\phi)}[\log p(z,x;\theta) - \log q(z;\phi)] \end{aligned}$$

Compare to previous reparameterization case, our case is a bit complicated because we have $r(z,\phi) = \log p(z,x;\theta) - \log q(z;\phi)$ instead of just $r(z)$. But we can still use reparameterization. Assume $z = \mu + \sigma\epsilon = g(\epsilon;\phi)$ like before, then

$$E_{q(z;\phi)}[r(z,\phi)] = E_\epsilon[r(g(\epsilon;\phi),\phi)] \approx \frac{1}{k}\sum_{i=1}^{k} r(g(\epsilon^i;\phi),\phi)$$

4.4.5 Amortized Inference

So far we have used a set of variational parameters ϕ^j fpr each data point x^i. Does not scale to large datasets. So we want to learn a single parametric function f_λ that maps each x to a set of good variational parameters. So training a model to generate ϕ^* given x^i. We approximate the posteriors $q(z|x^i)$ using this distribution $q_\lambda(z|x)$. In the literature, $q(z;f_\lambda(x^i))$ often denoted as $q_\phi(z|x)$. So overall our loss become

$$E_{q_\phi(z;\phi)}[\log p(z,x;\theta) - \log q_\phi(z|x)]$$

The procedure is

1. Initialize $\theta^{(0)}, \phi^{(0)}$
2. Randomly sample a data point x^i from D
3. Compute $\nabla_\theta L(x^i;\theta,\phi)$ and $\nabla_\phi L(x^i;\theta,\phi)$ using reparameterization
4. Update θ, ϕ in the gradient direction

4.5 VAE at High Level

$$\begin{aligned} L(x;\theta,\phi) &= E_{q_\phi(z|x)}[\log p(z,x;\theta) - \log q_\phi(z|x)] \\ &== E_{q_\phi(z|x)}[\log p(z,x;\theta)] - D_{KL}(q_\phi(z|x)||p(z)) \end{aligned}$$

1. We take a data point x^i
2. Map it to $\hat{z}$ by sampling from $q_\phi(z|x^i)$ (encoder)
3. Reconstruct $\hat{x}$ by sampling from $p(x|\hat{z};\theta)$ (decoder)

The first term of the training objective encourages $\hat{x} \approx x^i$, and the second term encourages $\hat{z}$ to be likely under the prior $p(z)$

Chapter 5

Normalizing Flow Model

Our latent variable has to have a model distribution $p_\theta(x)$ that is

- easy to evaluate, closed form density, useful for training
- easy to sampel, useful for generation

Key idea behind flow model: Map simple distributions (easy to sample and evaluate density) to complex distributions through an inevitable transformation

5.1 Change of Variables Formula

If $X = f(Z)$ and $f(\cdot)$ is monotone with inverse $Z = f^{-1}(X) = h(X)$, then we have

$$P_X(x) = p_z(h(x)) * |h'(x)|$$

Example 1
Let Z be a uniform random variable $\mathcal{U}[0, 2]$ with density p_Z. Let $X = f(Z) = 4Z$. So $h(x) = f^{-1}(x) = \frac{1}{4}x$

$$p_X(X = 4) = p_Z(h(4)) * |h'(4)| = p_Z(1) * \frac{1}{4} = \frac{1}{2} * \frac{1}{4}$$

Example 2
Let Z be a uniform random variable $\mathcal{U}[0, 2]$ with density p_Z. Let $X = f(Z) = \exp(Z)$. So $h(x) = f^{-1}(x) = \log(x)$

$$P_X(X = x) = p_Z(\log(x)) * |h'(x)| = \frac{1}{2} * \frac{1}{x}$$

5.1.1 Geometry: Determinants and volumes

Let Z be a uniform random vector $[0,1]^n$. Let $X = AZ$ for a square invertible matrix A with inverse $W = A^{-1}$. Geometrically, the matrix A maps the unit from hypercube to paralleltotope. The volume of the parallelotope is equal to $|det(A)|$, and X is uniformally distributed over the parallelotope. Hence

$$p_X(x) = p_Z(Wx)/|det(A)| = p_Z(Wx) * |det(W)|$$

- Note here $p_Z(Wx)$ is easy to compute, but $p_X(x)$ is complex

5.1.2 Generic Case: Non-linear transformation

For linear transformations specified via matrix multiplication A, change in volume is given by the determinant of A. But for non-linear transformation $f(\cdot)$, the linearized change in volume is given by the determinant of the Jacobian of $f(\cdot)$.

So the mapping between Z and X, given by $f : \mathbb{R}^n \to \mathbb{R}^n$, is invertible such that $X = f(Z)$ and $Z = f^{-1}(X)$, we have:

$$p_X(x) = p_Z(f^{-1}(x)) * |det(\frac{\partial}{\partial x} f^{-1}(x))|$$

- Unlike VAEs, x, z need to be continuous, and have the same dimension. This is a huge restriction.
- For any invertible matrix A, $det(A^{-1}) = det(A)^{-1}$, so we have $p_X(x) = p_Z(z) * |det(\frac{\partial}{\partial z} f(z)|^{-1}$

5.1.3 Generic Case Example: Two dimensional

Let Z_1, Z_2 be continuous random variables with joint density p_{Z_1,Z_2}.
Let $u = (u_1, u_2)$ be a transform
Let $v = (v_1, v_2)$ be the inverse transformation.
Let $X_1 = u_1(Z_1, Z_2)$ and $X_2 = u_2(Z_1, Z_2)$, then we have $Z_1 = v_1(X_1, X_2)$ and $Z_2 = v_2(X_1, X_2)$. Then the joint probability can be represented as :

$$\begin{aligned} p_{X_1,X_2}(x_1,x_2) &= p_{Z_1,Z_2}(v_1(x_1,x_2), v_2(x_1,x_2)) * \left| det \begin{pmatrix} \frac{\partial}{\partial X_1} v_1(x_1,x_2) & \frac{\partial}{\partial X_2} v_1(x_1,x_2) \\ \frac{\partial}{\partial X_1} v_2(x_1,x_2) & \frac{\partial}{\partial X_2} v_2(x_1,x_2) \end{pmatrix} \right| \\ &= p_{Z_1,Z_2}(z_1,z_2) * \left| det \begin{pmatrix} \frac{\partial}{\partial Z_1} u_1(z_1,z_2) & \frac{\partial}{\partial Z_2} u_1(z_1,z_2) \\ \frac{\partial}{\partial Z_1} u_2(z_1,z_2) & \frac{\partial}{\partial Z_2} u_2(z_1,z_2) \end{pmatrix} \right| \end{aligned}$$

5.2 Flow Model Structure

Consider a directed, latent-variable model over observed variables X and latent variables Z. In a normalizing flow model, we assume $X = f_\theta(Z)$, AND

$Z = f_\theta^{-1}(X)$. So the mapping function is deterministic and invertible.

Using change of variables, the marginal likelihood $p(x)$ is given by

$$p_X(x;\theta) = p_Z(f_\theta^{-1}(x)) * |det(\frac{\partial}{\partial x} f_\theta^{-1}(x))|$$

So now with multiple layers of invertible transformation, we have

$$z_m = f_\theta^m \circ ... \circ f_\theta^1(z_0) = f_\theta(z_0)$$

By change of variables, our marginal likelihood of x (z_m) can be calculated as

$$p_X(x;\theta) = p_Z(f_\theta^{-1}(x)) * \prod_{m=1}^{M} |det(\frac{\partial}{\partial z_m}(f_\theta^m)^{-1}(z_m))|$$

5.3 Learning and Inference

We can learn via maximum likelihood over the dataset D

$$\max_\theta p_X(D;\theta) = \sum_{x \in D} \log p_Z(f_\theta^{-1}(x)) + \log |det(\frac{\partial}{\partial x} f_\theta^{-1}(x))|$$

We can evaluate the likelihood via invese transformation $x \to z$ and change of variables formula as seen above.

The sampling process can be achieved by $z \sim p_Z(z)$ and then do $x = f_\theta(z)$, and we can obtain the latent representation via the inverse transformation $z = f^{-1}(x)$

5.4 Consideration for Invertible Transformation

First our transformation have to be invertible. So if we use an neural netowrk, it has to be an invertible neural network. Second, computing likelihood requires the evaluation of determinants of $n \times n$ Jacobian matrices, which is $O(n^3)$. So we want to choose transformations so that the resulting Jacobian matrix has special structure (e.g.: triangular matrix)

One example could be $x_i = f_i(z)$ only depends on $z_{\leq i}$, then the Jacobian matrix is lower triangular matrix, and we can just multiple the diagonal entry to get the determinant.

5.4.1 Planar Flows

Planar flow transformation parameterized by $\theta = (w, u, b)$:

$$x = f_\theta(z) = z + uh(w^T z + b)$$

- h is non-linearity invertible function

The absolute value of the determinant of the Jacobian is given by

$$|det(\frac{\partial}{\partial z} f_\theta(z)| = |det(I + h'(w^T z + b)uw^T)| = |1 + h'(w^T z + b)u^T w|$$

We also need to restrict parameters and non-linearity for the mapping to be invertible, for example, $h = tanh()$ and $h'(w^T z + b)u^T w \geq -1$

5.4.2 NICE

Partition the variable z into two disjoint subsets $z_{1:d}$ and $z_{d+1:n}$ for any $1 \leq d < n$. The forward mapping $z \to x$ is defined as :

$x_{1:d} = z_{1:d}$ (identity transformation)
$x_{d+1:n} = z_{d+1:n} + m_\theta(z_{1:d})$
$m_\theta(\cdot)$ is a neural network with parameters θ, d input units, and $n - d$ output units

This way, the inverse mapping can be seen as:

$$z_{1:d} = x_{1:d}$$
$$z_{d+1:n} = x_{d+1:n} - m_\theta(x_{1:d})$$

The Jacobian of forward mapping:

$$J = \begin{pmatrix} I_d & 0 \\ \frac{\partial}{\partial z_{1:d}} x_{d+1:n} & I_{n-d} \end{pmatrix}$$
$$det(J) = 1$$

We then add multiple layers together with arbitrary partition of variables in each layer. Note so far we are doing volume preserving transformation since the determinant is 1. So the final layer of NICE applies a re-scaling transformation with learnable parameters s_i for each dimension of x:

$$\text{Forward: } x_i = s_i z_i$$
$$\text{Inverse: } z_i = \frac{x_i}{s_i}$$
$$\text{Jacobian: } J = diag(s) = \prod s_i$$

Everything is trained by MLE

5.4.3 Real-NVP: Non-volume preserving extension of NICE

Forward mapping $z \to x$:

$x_{1:d} = z_{1:d}$ (identity transformation)

$x_{d+1:n} = z_{d+1:n} \odot \exp(\alpha_\theta(z_{1:d})) + \mu_\theta(z_{1:d})$

Same as NICE, but we add elementwise exponential scaling

$\alpha_\theta(\cdot)$ and $\mu_\theta(\cdot)$ are both neural networks with d input and $n-d$ output

Inverse mapping:

$$z_{1:d} = x_{1:d}$$
$$z_{d+1:n} = (x_{d+1:n} - \mu_\theta(x_{1:d})) \odot \exp(-\alpha_\theta(x_{1:d}))$$

The Jacobian of forward mapping is

$$J = \begin{pmatrix} I_d & 0 \\ \frac{\partial}{\partial z_{1:d}} x_{d+1:n} & diag(\exp(\alpha_\theta(z_{1:d}))) \end{pmatrix}$$
$$det(J) = \prod_{i=d+1}^{n} \exp(\alpha_\theta(z_{1:d}))) = \exp(\sum_{i=d+1^n} \alpha_\theta((z_{1:d})_i))$$

5.4.4 Masked Autoregressive Flow

Continuous Autoregressive Models as flow models
Consider a Gaussian autoregressive model:

$$p(x) = \prod_{i=1}^{n} p(x_i | x_{<i})$$

such that $p(x_i|x_{<i}) = \mathcal{N}(\mu_i(x_i, ..., x_{i-1}), \exp(\alpha_i(x_1, ..., x_{i-1}))^2)$ here $\mu_i(\cdot)$ and $\alpha_i(\cdot)$ are neural networks.
We can think of the sampler for this model as :

- Sample $z_i \sim \mathcal{N}(0, 1)$ for $i = 1, ..., n$
- Let $x_1 = \exp(\alpha_1)z_1 + \mu_1$, compute $\mu_2(x_1), \alpha_2(x_1)$
- Let $x_2 = \exp(\alpha_2)z_2 + \mu_2$, compute $\mu_3(x_1, x_2), \alpha_3(x_1, x_2)$

So essentially, we are transform a series of standard Gaussian to the x_i generated from model via invertible transformation parameterized by μ and α

MAF
Forward mapping from $z \to x$ is the same process as described above.
Inverse mapping from $x \to z$ (need this during training to compute likelihood)

- Compute all μ_i, α_i in parallel (e.g.: MADE)
- Let $z_1 = (x_1 - \mu_1)/\exp(\alpha_1)$
- Let $z_2 = (x_2 - \mu_2)/\exp(\alpha_2)$

Jacobian is lower diagonal, so efficient determinant computation.

5.4.5 Inverse Autoregressive Flow

This is just the reverse of MAF:
Forward mapping from $z \to x$ can be done in parallel:

- Sample $z_i \sim \mathcal{N}(0, 1)$ for $i = 1, ..., n$
- Compute all μ_i, α_i can be done in parallel
- Let $x_1 = \exp(\alpha_1)z_1 + \mu_1$
- Let $x_2 = \exp(\alpha_2)z_2 + \mu_2$

Inverse mapping from $x \to z$ are in sequential

- Let $z_1 = (x_1 - \mu_1)/\exp(\alpha_1)$, Compute $\mu_2(z_1), \alpha_2(z_1)$
- Let $z_2 = (x - 2 - \mu_2)/\exp(\alpha_2)$, Compute $\mu_3(z_1, z_2), \alpha_3(z_1, z_2)$

5.4.6 Parallel Wavenet

Two part training with a teacher and student model. Teacher is parameterized by MAF, which can be efficiently trained via MLE. Once teacher is trained, initialize a student model parameterized by IAF, which enables efficient sampling (but can no longer efficiently evaluate density for external datapoints).
Student distribution is trained to minimize the KL divergence between student s and teacher t. i.e.:

$$D_{KL}(s, t) = E_{x\sim s}[\log s(x) - \log t(x)]$$

This requires us to know:

- Samples x from student model (IAF) and get the density of x. (This is easy because x is generated by IAF model itself, so we already know all the z_i. No need to estimate it sequentially.
- Density of x from teacher model (MAF), easy to evaluate by design

So the overall process is to train teacher model via MLE, and then train student model to minimize KL divergence. At test-time use student model for testing.

5.4.7 MintNet

Building invertible neural networks with masked convolutions and let the Jacobian matrix to be triangular and entries are positive.

5.4.8 Gaussianization Flows

Let $X = f_\theta(Z)$ be a flow model with Gaussian prior $Z \sim \mathcal{N}(0, I) = P_Z$, and let $\tilde{X} \sim p_{data}$ be a random vector distributed according to the true data distribution. Flow models are trained with maximum likelihood to minimize the KL divergence $D_{KL}(p_{data}||p_\theta(X))$. Alternatively, we can think of the goal is to transform the distribution of data to Gaussian. We know that $U = F_{data}(\tilde{X})$ is uniform. We can transform U into a Gaussian using the inverse CDF trick : $\phi^{-1}(U) = \phi^{-1}(F_{data}(\tilde{X}))$. So overall the process is

1. Individual transform each variable into Gaussian
2. Apply a rotation matrix to the transformed data
3. Repeat Step 1, and Step 2 (stacking learnable Gaussian copula) to transform data into a normal distribution

Chapter 6

GAN

6.1 Motivation

Example 1: Great test log-likelihood, poor samples. For a discrete noise mixture models, let $p_\theta(x) = 0.01p_{data}(x) + 0.99p_{noise}(x)$

$$\begin{aligned}
\log p_\theta(x) &= \log(0.01p_{data}(x) + 0.99p_{noise}(x)) \\
&\geq \log(0.01p_{data}(x)) \\
&= \log p_{data}(x) - \log 100 \\
&\implies E_{p_{data}}[\log p_\theta(x)] \geq E_{p_{data}}[\log p_{data}(x)] - \log 100] \\
E_{p_{data}}[\log p_\theta(x)] &\leq E_{p_{data}}[\log p_{data}(x)] \\
&\qquad \text{(By non-negative KL divergence)}
\end{aligned}$$

As we increase the dimension of x, absolute value of $\log p_{data}(x)$ increases proportionally. So eventually we will have high likelihood, but garbage samples.

6.2 Comparing distributions via samples but not likelihood

Given $S_1 = \{x \sim P\}$ and $S_2 = \{x \sim Q\}$, a two -sample test considers the following hypotheses:

- Null: $P = Q$
- Alternative: $P \neq Q$

A test statistics T is a function of samples, and can be used to compare s_1, s_2. Such as means, and variance.

6.3 Settings

We can always assume that s_1 is from the data, i.e.: $s_1 = \{x \sim p_{data}\} = \{x \in D\}$. In addition, our generative model with distribution p_θ should permits efficient sampling, so we have $s_2 = \{x \sim p_\theta\}$. We use a neural network discriminator to conduct two-sample test. The discriminator will try to maximizes the two-sample test objective.

6.4 GAN Objective

Discriminator objective (predict 1 for x sampled from data, predict 0 for x generated from generator):

$$max_D V(G, D) = E_{x\sim P_{data}}[\log D(x)] + E_{x\sim P_G}[\log(1 - D(X)]$$

Generator objective (generator takes in noise z and maps to x):

$$min_G max_D V(G, D) = E_{x\sim P_{data}}[\log D(x)] + E_{x\sim P_G}[\log(1 - D(X)]$$

i.e.: Generator objective is to minimize the loss described for discriminator

6.5 Jenson-Shannon Divergence

JSD is a symmetric version of KL divergence and its squareroot satisfies the triangle inequality.

$$D_{JSD}(p, q) = \frac{1}{2}(D_{KL}(p, \frac{p+q}{2}) + D_{KL}(q, \frac{p+q}{2}))$$

with the optimal discriminator D^*_G, we see GAN minimizes a scaled and shifted Jensen-Shannon divergence $\overset{min}{G}\ 2D_{JSD}(P_{data}, P_G) - \log 4$

6.6 Training GAN

Sample minibatch of m training points $x^{(1)}, ..., x^{(m)}$ from dataset D. Sample minibatch of m noise vector $z^{(1)}, ..., z^{(m)}$ from p_z.
Update the discriminator parameters ϕ by stochastic gradient ascent

$$\nabla_\phi V(G_\theta, D_\phi) = \frac{1}{m}\nabla_\phi \sum_{i=1}^{m}[\log D_\phi(x^{(i)}) + \log(1 - D_\phi(G_\theta(z^{(i)})))]$$

Update the generator parameters θ by stochastic gradient descent

$$\nabla_\theta V(G_\theta, D_\phi) = \frac{1}{m}\nabla_\theta \sum_{i=1}^{m}[\log(1 - D_\phi(G_\theta(z^{(i)})))]$$

6.7 f-GAN

Given two densities p and q, the f-divergence is given by

$$D_f(p,q) = E_{x\sim q}[f(\frac{p(x)}{q(x)})]$$

where f is any convex, lower-semicontinuous function with $f(1) = 0$

- Convex: Line joining any two points lies above the function. (Non-negative second derivative)
- Lower-semicontinuous: At discontinuous point, function value at any point x_0 is close to $f(x_0)$ or greater than $f(x_0)$
- By convex and Jensen's inequality: $E_{x\sim q}[f(\frac{p(x)}{q(x)})] \geq f(E_{x\sim q}[\frac{p(x)}{q(x)}]) = f(1) = 0$ i.e.: f-divergence is always non-negative

In order to use f-divergences as a two-sample test objective for likelihood free learning, we need to be able to estimate it only via samples.

Frenchel Conjugate: for any function $f(\cdot)$. its convex conjugate is defined as

$$f*(t) = \sup_{u\in domain_f} (ut0f(u))$$

- f^* is always convex and lower semi-continuous
- $f^{**} \leq f$
- $f^{**} = f$ when $f(\cdot)$ is convex, lower semicontinuous. Equivalently, $f(u) = f^{**}(u) = sup_{t\in domin_{f^*}}(tu - f^*(t))$

We can obtain a lower bound to any f-divergence via its Fenchel conjugate. Because f-divergence might not necessarily directly computable.

$$\begin{aligned}
D_f(p,q) &= E_{x\sim q}[f(\frac{p(x)}{q(x)})] && \text{(Definition of } f\text{- divergence)}\\
&= E_{x\sim q}[sup_{t\in domain_{f^*}}(t\frac{p(x)}{q(x)} - f^*(t))]\\
&:= E_{x\sim q}[T^*(x)\frac{p(x)}{q(x)} - f^*(T^*(x))]\\
&\quad \text{(Let } T^*(x) \text{ be the optimal t that maximizes the value)}\\
&= \int_X T^*(x) * p(x) - f^*(T^*(x))q(x) \quad dx\\
&= sup_T \int_X T(x) * p(x) - f^*(T(x))q(x) \quad dx\\
&\geq sup_{T\in\mathcal{T}} \int_X T(x) * p(x) - f^*(T(x))q(x) \quad dx\\
&\quad (\mathcal{T} \text{ is an arbitrary class of function})\\
&= sup_{T\in\mathcal{T}} E_{x\sim p}[T(x)] - E_{x\sim q}[f^*(T(x))]
\end{aligned}$$

Note this lower bound is likelihood-free w.r.t p and q because we can use Monte Carlo to estiamte the expected value. Conceptually, maxing T means to push up the value of T on samples from p (real data) and push down the value for x from q (generator).
So we can establish f-GAN as :

- Choose any f-divergence
- Let $p = p_{data}$, and $q = p_G$
- Parameterize T by ϕ and G by θ
- The objective is $min_\theta max_\phi F(\theta, \phi) = E_{x\sim p_{data}}[T_\phi(x)] - E_{x\sim p_{G_\theta}}[f^*(T_\phi(x))]$
- Generator G_θ tries to minimize the divergence estimate, and discriminator T_ϕ tries to tighten the lower bound

Limitation of f-GAN:

- The support of q has to cover the support of p. Otherwise discontinuity arises in $f-$ divergences

6.8 Wasserstein GAN

Wasserstein distance is defined as:

$$D_w(p, q) = \inf_{\gamma \in \prod(p,q)} E_{(x,y)\sim\gamma}[\|x - y\|_1]$$

Where $\prod(p, q)$ contains all joint distributions of (x, y) where the marginal distribution of x is $p(x)$ and the marginal distribution of y is $q(y)$. $\gamma(y|x)$ is a probabilistic earth moving plan that warps $p(x)$ to $q(y)$.

Kantorovich-Rubinstein duality describes the Wasserstein distance in the form of optimization problem.

$$D_w(p, q) = \sup_{\|f\|_{L\leq 1}} E_{x\sim p}[f(x)] - E_{x\sim q}[f(x)]$$

$\|f\|_{L\leq 1}$ means that the Lipschitz constant of $f(x)$ is 1. i.e.:

$$\forall x, y : |f(x) - f(y)| \leq \|x - y\|_1$$

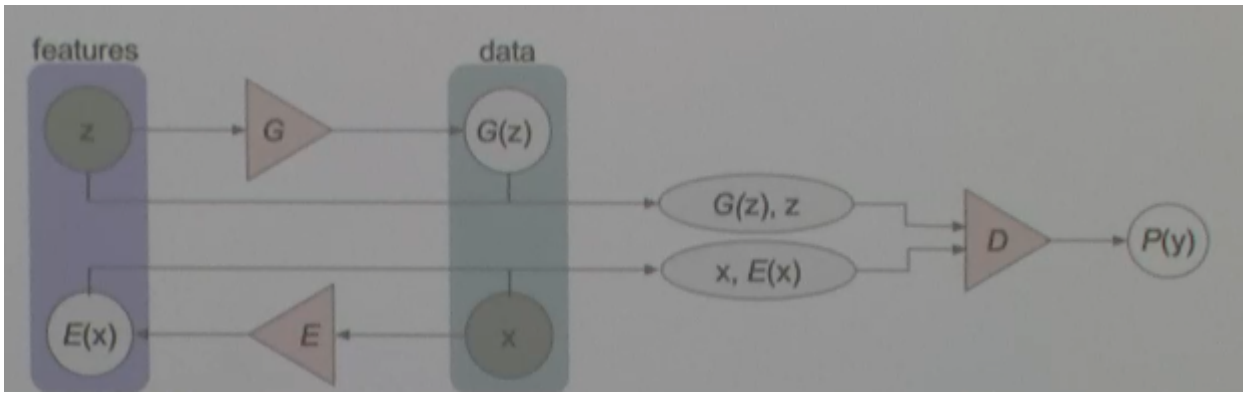
So the Wasserstein GAN with discriminator $D_\phi(x)$ and generator $G_\theta(z)$

$$\min_\theta \max_\phi E_{x\sim p_{data}}[D_\phi(x)] - E_{z\sim p(z)}[D_\phi(G_\theta(z))]$$

The Lipschitzness of $D_\phi(x)$ is enforced through weight clipping or gradient penalty.

6.9 BiGAN and latent representations

BiGAN framework:



- Encoder network E only observes $x \sim p_{data}(x)$ during training to learn a mapping $E : x \to z$.
- The generator network only observes the samples from the prior $z \sim p(z)$ to learn a mapping $G : z \to x$
- The discriminator D now observes samples from the generative model $z, G(z)$ and the encoding distribution $E(x), x$, and try to discriminate.
- After training is complete, new samples are generated via G and latent representation encoder E

6.10 Translating across domains

Let's say we have distribution X and distribution Y. CycleGAN structure:

- Conditional Generative models $G : X \to Y$
- Conditional Generative model $F : Y \to X$
- Discriminator D_y compares the observed dataset Y and the generated samples $\hat{Y} = G(X)$
- Discriminator D_x compares the observed dataset X and the generated sample $\hat{X} = F(Y)$
- Cycle consistency: $F(G(X)) \approx X$ and $G(F(Y)) \approx Y$

The overall loss function is

$$\min_{F,G,D_x,D_y} L_{GAN}(G, D_y, X, Y) + L_{GAN}(F, D_x, X, Y) + \lambda(E_X[\|F(G(X)) - X\|_1] + E_Y[\|G(F(Y)) - Y\|_1]$$

Chapter 7

Energy Based Models

7.1 Creating Arbitrary Probability Distributions

Let $g_\theta(x) \geq 0$, then we can define a probability distribution as follows:

$$p_\theta(x) = \frac{1}{\int g_\theta(x)\, dx} g_\theta(x)$$

This way, if we can choose $g_\theta(x)$ such that we know the volume (integration result), we can easily create the probability distribution.

We can create more complicated probability distribution via the following approach:
Autoregressive: Given $p(x, y) = p(x) * p(y|x)$, We can let each component to be a arbitrary probability distribution

$$\int_x \int_y p_\theta(x) p_{\theta'(x)}(y)\, dydx = \int_x p_\theta(x) \int_y p_{\theta'(x)}(y)\, dydx$$

Latent variable model: Mixtures of normalized objects:

$$\alpha p_\theta(x) + (1 - \alpha) p_{\theta'}(x)$$

7.2 Energy-based Model

Let $f_\theta(x)$ to be any arbitrary function, define the probability distribution as

$$p_\theta(x) = \frac{1}{\int \exp(f_\theta(x))\, dx} \exp(f_\theta(x)) = \frac{1}{Z(\theta)} \exp(f_\theta(x))$$

- $Z(\theta) = \int \exp(f_\theta(x))\, dx$ is called the partition function. Now we can assume this function is not easy to evaluate.
- $-f_\theta(x)$ is called the energy.

- Sample from $p_\theta(x)$ is hard
- Evaluating and optimizing likelihood $p_\theta(x)$ is hard
- But notice the ratio $\frac{p_\theta(x)}{p_theta(x')} = \exp(f_\theta(x) - f_\theta(x'))$ is easy

7.3 Training EBM with Contrastive Divergence

Original goal is

$$\max_\theta \log \frac{\exp(f_\theta(x))}{Z(\theta)} = \max_\theta f_\theta(x_{train}) - \log Z(\theta)$$

Looking at the gradient of the log-likelihood, we have

$$\begin{aligned}
&\nabla_\theta f_\theta(x_{train}) - \nabla_\theta \log Z(\theta) \\
&= \nabla_\theta f_\theta(x_{train}) - \frac{\nabla_\theta Z(\theta)}{Z(\theta)} \\
&= \nabla_\theta f_\theta(x_{train}) - \frac{1}{Z(\theta)} \int \nabla_\theta \exp(f_\theta(x))\, dx \\
&= \nabla_\theta f_\theta(x_{train}) - \frac{1}{Z(\theta)} \int \exp(f_\theta(x)) \nabla_\theta f_\theta(x)\, dx \\
&= \nabla_\theta f_\theta(x_{train}) - \int \frac{\exp(f_\theta(x))}{Z(\theta)} \nabla_\theta f_\theta(x)\, dx \\
&= \nabla_\theta f_\theta(x_{train}) - E_{x \sim sample}[\nabla_\theta f_\theta(x_{sample})]
\end{aligned}$$

So now the first term can be evaluated using training data, and second term can be evaluate by Monte Carlo Sampling from the distribution $p_\theta(x) = \frac{\exp(f_\theta(x))}{Z(\theta)}$

7.3.1 Sampling from EBMs - Metropolis-Hasting Markov Chain Monte Carlo (MCMC)

- $x^0 = \pi(x)$
- $x' = x^t + noise$
- $x_{t+1} = x'$ if $f_\theta(x') \geq f_\theta(x^t)$. Otherwise, set $x^{t+1} = x'$ with probability $\exp(f_\theta(x') - f_\theta(x^t))$, and $x^{t+1} = x^t$ with the remaining probability.
- quite slow

7.3.2 Sampling in Contrastive Divergence

We can not directly calculate $p_\theta(x) = \frac{\exp(f_\theta(x))}{Z(\theta)}$, so this means we cannot directly sample from it. The solution is to use Langevin MCMC

- Let $\pi(x)$ be a prior distribution that is easy to sample from, e.g.: standard normal

- $x^0 \sim \pi(x)$
- Repeat $x^{t+1} \sim x_t + \epsilon \nabla_x \log p_\theta(x^t) + \sqrt{2\epsilon} z^t$ where $z^t \sim \mathcal{N}(0, I)$.
- if $\epsilon \to 0$ (normally decay with $\frac{1}{t}$), and $T \to \infty$, we have $x_T \sim p_\theta(x)$. In reality we repeat 10 100 times.
- Note that gradiet w.r.t x is easy. In log likelihood term, we have $\nabla_x f_\theta(x) - \nabla_x \log Z(\theta) = \nabla_x (f_\theta(x))$
- Somewhat slow

7.4 Training EBM without Sampling: Score function

Intuition
So far we have been trying to minimize the KL divergence of the data distribution and the generator distribution (equivalent to contrastive divergence). This process is very slow due to sampling process. So we want to use a different metrics.

Score Function
The Stein score function is defined as the gradient w.r.t input x:

$$\begin{aligned} s_\theta(x) &:= \nabla_x \log p_\theta(x) \\ &= \nabla_x f_\theta(x) - \nabla_x \log Z(\theta) = \nabla_x f_\theta(x) \qquad \text{(for EBM)} \end{aligned}$$

So if we use a deep neural network to parameterize f_θ, then the score function is just the back propagation.

Fisher Divergence
The Fisher Divergence between $p(x)$ and $q(x)$:

$$D_F(p, q) := \frac{1}{2} E_{x \sim p}[\|\nabla_x \log p(x) - \nabla_x \log q(x)\|_2^2]$$

- If $p = q$, then $D_F(p, q) = 0$, and vice versa.
- $D_F(p, q)$ is not symmetric

Fisher Divergence for Training EBM
So in training EBM with Fisher Divergence, we want to minimize the divergence between p_{data} and p_θ

$$\begin{aligned}&\frac{1}{2}E_{x\sim p_{data}}[\|\nabla_x \log p_{data}(x) - s\theta(x)\|_2^2]\\&= \frac{1}{2}E_{x\sim p_{data}}[\|\nabla_x \log p_{data}(x) - \nabla_x f_\theta(x)\|_2^2]\end{aligned}$$

We can obtain $\nabla_x \log p_{data}(x)$ using integration by parts. Using univariate case as an example:

$$\begin{aligned}&\frac{1}{2}E_{x\sim p}[\|\nabla_x \log p_{data}(x) - \nabla_x p_\theta(x)\|_2^2]\\&= \frac{1}{2}\int p_{data}(x)[(\nabla_x \log p_{data}(x) - \nabla_x p_\theta(x))^2]\,dx\\&= \frac{1}{2}\int p_{data}(x)(\nabla_x \log p_{data}(x))^2\,dx + \frac{1}{2}\int p_{data}(x)(\nabla_x \log p_\theta(x))^2\,dx -\\&\quad \int p_{data}(x)\nabla_x \log p_{data}(x)\nabla_x \log p_\theta(x)\,dx\end{aligned}$$

Notice that the first term $\frac{1}{2}\int p_{data}(x)(\nabla_x \log p_{data}(x))^2\,dx$ is not dependent on the model parameters, so it will not have any contribution during optimization (since we are tuning θ). The second term $\frac{1}{2}\int p_{data}(x)(\nabla_x \log p_\theta(x))^2\,dx$ is easy to handle, because it essentially an expectation, which we can use Monte Carlo Sampling.

Now for the last term:

$$\begin{aligned}&-\int p_{data}(x)\nabla_x \log p_{data}(x)\nabla_x \log p_\theta(x)\,dx\\&= -\int p_{data}(x)\frac{1}{p_{data}(x)}\nabla_x p_{data}(x)\nabla_x \log p_\theta(x)\,dx\\&= -\int \nabla_x p_{data}(x)\nabla_x \log p_\theta(x)\,dx\\&= -p_{data}(x)\nabla_x \log p_\theta(x)|_{x=-\infty}^{\infty} + \int p_{data}\nabla_x^2 \log p_\theta(x)\,dx && \text{(Integration by parts)}\\&= \int p_{data}\nabla_x^2 \log p_\theta(x)\,dx && \text{(Some weak assumption on } p_{data}\text{)}\end{aligned}$$

Now this term is also an expectation, which can be estimated via Monte Carlo

sampling. In summary for multivariate case,

$$\begin{aligned}
&\min_\theta \frac{1}{2} E_{x\sim p_{data}}[\|\nabla_x \log p_{data}(x) - \nabla_x \log p_\theta(x)\|_2^2] \\
&= \min_\theta E_{x\sim p_{data}}[\frac{1}{2}\|\nabla_x \log p_\theta(x)\|_2^2 + tr(\nabla_x^2 \log p_\theta(x))] \\
&\qquad\qquad \text{(Second component is Hessian of } \log p_\theta(x))
\end{aligned}$$

In practice

- Sample a min-batch of datapoints $\{x_1, ..., x_n\} \sim p_{data}(x)$
- Estimate the score matching loss with the empirical mean $\frac{1}{n}\sum_{i=1}^n \left(\frac{1}{2}\|\nabla_x f_\theta(x_i)\|_2^2 + tr(\nabla_x^2 f_\theta(x_i)\right)$
  - Note $\nabla_x p_\theta(x) = \nabla_x f_\theta(x) - \nabla_x \log Z(\theta) = \nabla_x f_\theta(x)$
- Stochastic gradient descent
- Note in all the steps, we did not do any sampling from EBM

But computing the trace of Hessian $tr(\nabla_x^2 \log p_\theta(x))$ is in general very expensive for large models. We will discuss solution later.

7.4.1 Score matching for learning implicit VAEs

Given an implicit VAE model $p(z), p_\theta(x|z), q_\phi(z|x)$, and let sampling from $q_\phi(z|x)$ is easy, but actual probability calculation is hard. E.g.: $q_\phi(z|x) = \delta(z = f_\phi(x, \epsilon))$ where ϵ is a Gaussian noise.
Given the setup, the goal would be maximize the ELBO:

$$E_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)p(z)] - E_{z\sim q_\phi(z|x)} \log q_\phi(z|x)$$

But calculating $\log q_\phi(z|x)$ is hard given the setup. So we want to estimate the gradient of the entropy term by training an energy-based model:

$$\begin{aligned}
\nabla_\phi H(q_\phi(z|x)) &= -\nabla_\phi E_{z\sim q_\phi(z|x)}[\log q_\phi(z|x)] \\
&= -\nabla_\phi E_\epsilon[\log q_\phi(f_\phi(x,\epsilon)|x)] \\
&= -E_\epsilon[\nabla_\phi \log q_\phi(f_\phi(x,\epsilon)|x)] \\
&= -E_\epsilon[\nabla_z \log q_\phi(z|x)|_{z=f_\phi(x,\epsilon)} \nabla_\phi f_\phi(x,\epsilon)]
\end{aligned}$$

Note that the first term $\nabla_z \log q_\phi(z|x)|_{z=f_\phi(x,\epsilon)}$ is the score function of $q_\phi(z|x)$. So this can be an energy based model. The second term $\nabla_\phi f_\phi(x,\epsilon)$ normally can be obtained via back propagation.

7.5 Training EBM without Sampling: Noise Contrastive Estimation

We can learn an energy-based model by contrasting it with a noise distribution:

- Data distribution: $p_{data}(x)$
- Noise distribution: $p_n(x)$ Should be analytically tractable and easy to sample from
- Training a discriminator $D_\theta(x) \in [0, 1]$ to distinguish between data samples and noise samples
- Training objective: $\max_\theta E_{x\sim p_{data}}[\log D_\theta(x)] + E_{x\sim p_n}[\log(1 - D_\theta(x))]$
- Optimal discriminator $D_{\theta*}(x)$ has the form $\frac{p_{data}(x)}{p_{data}(x)+p_n(x)}$
- So we can parameterize a discriminator to be in the form of $\frac{p_\theta(x)}{p_\theta(x)+p_n(x)}$. In the end, the $p_{\theta*}$ from optimal discriminator will be equal to $p_{data}(x)$

In Energy-based model:

$$p_\theta(x) = \frac{\exp(f_\theta(x))}{Z(\theta)}$$

The constraint $Z(\theta) = \int e^{f_\theta(x)}\, dx$ is hard to satisfy. So we model $Z(\theta)$ with an additional trainable parameter Z that disregards the constraint $Z = \int e^{f_\theta(x)}\, dx$.

Now the model form become

$$p_{\theta,Z}(x) = \frac{\exp(f_\theta(x))}{Z}$$

Note that right now this $p_{\theta,Z}$ is unormalized. But it doesn't matter in Noise Contrastive Estimation. But the optimal $Z*$ will become the proper partition function.

In summary, we build a discrminator $D_{\theta,Z}(x)$ for probabilistic model $p_{\theta,Z}(x)$ as

$$D_{\theta,Z}(x) = \frac{\frac{\exp(f_\theta(x))}{Z}}{\frac{\exp(f_\theta(x))}{Z} + p_n(x)} = \frac{\exp(f_\theta(x))}{\exp(f_\theta(x)) + p_n(x)Z}$$

The Noise contrastive estimation training objective is

$$\max_{\theta,Z} E_{x\sim p_{data}}[\log D_{\theta,Z}(x)] + E_{x\sim p_n}[\log(1 - D_{\theta,Z}(x))]$$
$$= \max_{\theta,Z} E_{x\sim p_{data}}[f_\theta(x) - \log(\exp(f_\theta(x)) + Zp_n(x))] + E_{x\sim p_n}[\log(Zp_n(x)) - \log(\exp(f_\theta(x)) + Zp_n(x))]$$

We use log-sum-exp trick for numerical stability:

$$\begin{aligned}\log(\exp(f_\theta(x)) + Zp_n(x)) &= \log(\exp(f_\theta(x)) + \exp(\log Z + \log p_n(x))) \\ &= logsumexp(f_\theta(x), \log Z + \log p_n(x))\end{aligned}$$

Training overall procedure:

- Sample a mini-batch of datapoints $x_1, ..., x_n \sim p_{data}(x)$
- Sample a mini-batch of noise samples $y_1, ..., y_n \sim p_n(y)$
- Estimate the Noise Contrasitive Estimation loss
  - $\frac{1}{n}\sum_{i=1}^{n}[f_\theta(x_i) - logsumexp(f_\theta(x_i), \log Z + \log p_n(x_i)) + \log Z + p_n(y_i) - logsumexp(f_\theta(y_i), \log Z + \log p_n(y_i))]$
- Stochastic gradient ascent
- Notice this process does not need to sample from the EBM

7.5.1 Flow contrastive estimation

We can parameterize the noise distribution with a normalizing flow model $p_{n,\phi}(x)$. This we we can expand our noise distribution family. Subsequently parameterize the distriminator $D_{\theta,Z,\phi}(x)$ as

$$D_{\theta,Z,\phi}(x) = \frac{\frac{\exp(f_\theta(x)}{Z}}{\frac{\exp(f_\theta(x)}{Z} + p_{n,\phi}(x)} = \frac{\exp(f_\theta(x))}{\exp(f_\theta(x)) + p_{n,\phi}(x)Z}$$

We can train the flow model to minimize $D_{JS}(p_{data}, p_{n,\phi})$:

$$\min_\phi \max_{\theta,Z} E_{x\sim p_{data}}[\log D_{\theta,Z,\phi}(x)] + E_{x\sim p_{n,\phi}}[\log(1 - D_{\theta,Z,\phi}(x)]$$

7.6 Adversarial Training for EBMs

Energy-based model has the form:

$$p_\theta(x) = \frac{\exp(f_\theta(x))}{Z(\theta)}$$

We can develop an upper bound with a variational distribution $q_\phi(x)$:

$$\begin{aligned}
E_{x\sim pdata}[\log p_\theta(x)] &= E_{x\sim pdata}[f_\theta(x) - \log Z(\theta)] \\
&= E_{x\sim pdata}[f_\theta(x)] - \log \int e^{f_\theta(x)}\, dx \\
&= E_{x\sim pdata}[f_\theta(x)] - \log \int q_\phi(x) * \frac{\exp(f_\theta(x))}{q_\phi(x)}\, dx \\
&\leq E_{x\sim pdata}[f_\theta(x)] - \int q_\phi(x) * \log \frac{\exp(f_\theta(x))}{q_\phi(x)}\, dx \\
&\leq E_{x\sim pdata}[f_\theta(x)] - \int q_\phi(x)(f_\theta(x) - \log q_\phi(x))\, dx \\
&= E_{x\sim pdata}[f_\theta(x)] - E_{x\sim q_\phi(x)}[f_\theta(x)] + H(q_\phi(x))
\end{aligned}$$

So the Adversarial training can be defined as max min operation (first we need to tighten the upper bound, then we maximize the likelihood of the bound):

$$\max_\theta \min_\phi E_{x\sim p_{data}}[f_\theta(x)] - E_{x\sim q_\phi(x)}[f_\theta(x)] + H(q_\phi(x))$$

Note here for $q_\phi(x)$, we need to sample from it very efficiently, and we need to compute its entropy efficiently. But note we only need to compute the entropy for the sample of $q_\phi(x)$ (so we can do inverse autoregressive flow model)