Generative Models

Reference:

- introtodeeplearning.com
- http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec20.pdf
- deeplearningbook.org

Which face is fake?

Supervised vs unsupervised learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn function to map

$$x \rightarrow y$$

Examples: Classification, regression, object detection, semantic segmentation, etc.

Unsupervised Learning

Data: x
x is data, no labels!

Goal: Learn some *hidden* or *underlying structure* of the data

Examples: Clustering, feature or dimensionality reduction, etc.

Supervised vs unsupervised learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn function to map

$$x \rightarrow y$$

Examples: Classification, regression, object detection, semantic segmentation, etc.

Unsupervised Learning

Data: x
x is data, no labels!

Goal: Learn some *hidden* or *underlying structure* of the data

Examples: Clustering, feature or dimensionality reduction, etc.

Generative modeling

Goal: Take as input training samples from some distribution and learn a model that represents that distribution

Density Estimation

Sample Generation

Input samples

Training data $\sim P_{data}(x)$

Generated samples

Generated $\sim P_{model}(x)$

How can we learn $P_{model}(x)$ similar to $P_{data}(x)$?

Why generative models? Debiasing

Capable of uncovering **underlying latent variables** in a dataset

Homogeneous skin color, pose

VS

Diverse skin color, pose, illumination

How can we use latent distributions to create fair and representative datasets?

Why generative models? Outlier detection

- **Problem:** How can we detect when we encounter something new or rare?
- **Strategy:** Leverage generative models, detect outliers in the distribution
- Use outliers during training to improve even more!

95% of Driving Data:
(1) sunny, (2) highway, (3) straight road

Detect outliers to avoid unpredictable behavior when training

Edge Cases

Harsh Weather

Pedestrians

What is a latent variable?

Myth of the Cave

What is a latent variable?

Can we learn the **true explanatory factors**, e.g. latent variables, from only observed data?

Autoencoders

Autoencoders

- An autoencoder is a feed-forward neural net whose job is to take an input x and predict x.
- To make this non-trivial, we need to add a bottleneck layer whose dimension is much smaller than the input.

Why autoencoders?

- Map high-dimensional data to low dimensions for visualization Compression (i.e. reducing the file size)
- Learn abstract features in an unsupervised way so you can apply them to a supervised task
- Unlabeled data can be much more plentiful than labeled data

Deep Autoencoders

- Deep nonlinear autoencoders learn to project the data, not onto a subspace, but onto a nonlinear **manifold** (a space that locally looks like some Euclidean space)
- This is a kind of nonlinear dimensionality reduction

Linear Basis

Nonlinear Manifold

Autoencoders: background

Unsupervised approach for learning a **lower-dimensional** feature representation from unlabeled training data

Why do we care about a low-dimensional z?

"Encoder" learns mapping from the data, x, to a low-dimensional latent space, z

Autoencoders: background

How can we learn this latent space?
Train the model to use these features to **reconstruct the original data**

"Decoder" learns mapping back from latent, z, to a reconstructed observation, $\hat{x}$

Autoencoders: background

How can we learn this latent space?
Train the model to use these features to **reconstruct the original data**

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

Loss function doesn't use any labels!!

Autoencoders: background

How can we learn this latent space?
Train the model to use these features to **reconstruct the original data**

Symmetric structures are usually used in encoder and decoder
Q: If encoder uses convolution layers, what layers can be used in decoder?

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

Loss function doesn't use any labels!!

Deconvolution (Transposed Convolution)

Convolution:
Filter: 3x3
Input: 6x6
Output: 4:4

Conv2DTranspose*:*
Filter: 3x3
Input: 4x4 with *2x2 padding*
Output: 6x6

For details of Transposed Convolution, check

- https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8
- https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d

Dimensionality of latent space → reconstruction quality

Autoencoding is a form of compression!
Smaller latent space will force a larger training bottleneck

2D latent space

5D latent space

Ground Truth

Deep Autoencoders

- Nonlinear autoencoders can learn more powerful codes for a given dimensionality, compared with linear autoencoders (PCA)

Autoencoders for representation learning

Bottleneck hidden layer forces network to learn a compressed latent representation

Reconstruction loss forces the latent representation to capture (or encode) as much "information" about the data as possible

Autoencoding = **Auto**matically **encoding** data

Denoising Autoencoder (DAE)

- Autoencoder: learn an appropriate dimensionality reduction of inputs similar to our training set
  - Sensitive to noise in training set
  - Encoder may not be generalizable to inputs other than x
- Denoising Autoencoder (DAE)
  - Input x is *corrupted* by a corruption process (e.g. injecting Gaussian noise)

$$x' = corrupt(x)$$

  - Then the network is judged on how well it reconstructs the *original input* from the *corrupted input* x' :

$L(x, \tilde{x}) = \| x - \tilde{x} \|^2$, where $\tilde{x} = decode[encode(corrupt(x))]$

- The objective is to learn a generalizable encoding and decoding:
  - Sensitive enough to recreate the original observation
  - But insensitive enough to the training data

From Autoencoder to Variational Autoencoder (VAE)

- Autoencoders learn how to compress the data based on feature correlations discovered from data during training
- However, ***what do they tell us about the underlying properties of input x***?
- An ideal encoder:

- Let's focus on decoder. If we plugin any value to the decode, can it generate interesting results?

https://www.jeremyjordan.me/variational-autoencoders/

Variational Autoencoders (VAEs)

Variational Autoencoder (VAE)

- VAE provides a ***probabilistic*** manner for describing an observation in latent space
- Thus, rather than building an encoder which outputs *a single value* to describe each latent state attribute, we'll formulate our encoder to describe **a probability distribution** for each latent attribute.

Intuition of VAE

- By constructing encoder to output a range of possible values (a statistical distribution), we're essentially enforcing a continuous, smooth latent space representation.
- For any sampling of the latent distribution, we're expecting our decoder model to be able to accurately reconstruct the input.
- Thus, values which are nearby to one another in latent space should correspond with very similar reconstructions.

VAEs: key difference with traditional autoencoder

VAEs: key difference with traditional autoencoder

VAEs: key difference with traditional autoencoder

Variational autoencoders are a probabilistic twist on autoencoders!

Sample from the mean and standard dev. to compute latent sample

VAE optimization

Encoder computes: $q_\phi(\mathrm{z}|x)$ Decoder computes: $p_\theta(\mathrm{x}|z)$

VAE optimization

Encoder computes: $q_\phi(z|x)$

Decoder computes $p_\theta(x|z)$

$$\mathcal{L}(\phi, \theta) = \text{(reconstruction loss)} + \text{(regularization term)}$$

VAE optimization

Encoder computes: $q_{\phi}(z|x)$

Decoder computes $p_{\theta}(x|z)$

$$\mathcal{L}(\phi, \theta, x) = \text{(reconstruction loss)} + \text{(regularization term)}$$

VAE optimization

Encoder computes: $q_\phi(z|x)$

Decoder computes $p_\theta(x|z)$

$$\mathcal{L}(\phi, \theta, x) = \boxed{\text{(reconstruction loss)}} + \text{(regularization term)}$$

VAE optimization

Encoder computes: $q_\phi(z|x)$

Decoder computes $p_\theta(x|z)$

$$\mathcal{L}(\phi, \theta, x) = \text{(reconstruction loss)} + \boxed{\text{(regularization term)}}$$

This is also called Evidence Lower Bound (ELBO). For math details about loss, see pages in the backup

Priors on the latent distribution

$$D\left(q_\phi(z|x) \parallel p_\theta(z)\right)$$

Inferred latent distribution

Fixed prior on latent distribution

Common choice of prior:

$$p(z) = \mathcal{N}(\mu = 0, \sigma^2 = 1)$$

- Encourages encodings to distribute encodings evenly around the center of the latent space
- Penalize the network when it tries to "cheat" by clustering points in specific regions (ie. memorizing the data)

Check https://www.jeremyjordan.me/variational-autoencoders/ for illustrative result of regularization

Intuition of Regularization

Only reconstruction loss

Only KL divergence

Combination

Without regularization, our network can "cheat" by learning narrow distributions

Penalizing KL divergence acts as a regularizing force

With a small enough variance, this distribution is effectively only representing a single value

Attract distribution to have zero mean

Ensure sufficient variance to yield a smooth latent space

Priors on the latent distribution

Assume both probabilities follow normal distributions. See the math details in page 65

$$D\left(p_\phi(z|x) \parallel p_\theta(z)\right)$$

$$= \frac{1}{2}\sum_{j=0}^{k-1}\left(\sigma_j^2 + \mu_j^2 - 1 - \log\sigma_j^2\right)$$

KL-divergence between the two distributions

Common choice of prior:

$$p(z) = \mathcal{N}(\mu = 0, \sigma^2 = 1)$$

- Encourages encodings to distribute encodings evenly around the center of the latent space
- Penalize the network when it tries to "cheat" by clustering points in specific regions (ie. memorizing the data)

VAEs computation graph

$$\mathcal{L}(\phi, \theta, x) = \text{(reconstruction loss)} + \text{(regularization term)}$$

VAEs computation graph

Problem: We cannot backpropagate gradients through sampling layers!

Encoder computes: $q_\phi(\mathrm{z}|x)$

Decoder computes $p_\theta(\mathrm{x}|z)$

$$\mathcal{L}(\phi, \theta, x) = \text{(reconstruction loss)} + \text{(regularization term)}$$

Reparametrizing the sampling layer

Key Idea:

$$\cancel{z \sim \mathcal{N}(\mu, \sigma^2)}$$

Consider the sampled latent vector as a sum of

- a fixed μ vector,
- and fixed σ vector, scaled by random constants drawn from the prior distribution

$$\Rightarrow z = \mu + \sigma \odot \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0,1)$

Reparametrizing the sampling layer

Original form

Reparametrizing the sampling layer

Original form

Reparametrized form

Reparametrisation trick

- Naïvely plugging in the sampling operation into the encoder *will not work*, because the sampling operation **has no gradient**!
- To make it work, we can instead sample upfront $\vec{\varepsilon} \sim \mathcal{N}(\vec{0}, \mathbf{I})$, feed it as an *external input*, and transform appropriately:

$$\vec{z} = \vec{\mu} + \vec{\varepsilon} \odot \vec{\sigma}$$

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(batch_size, m),
                              mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon
z = Lambda(sampling)([z_mean, z_log_var])
```

VAEs: Latent perturbation

Slowly increase or decrease a **single latent variable**
Keep all other variables fixed

Head pose

Different dimensions of z encodes **different interpretable latent features**

VAEs: Latent perturbation

Ideally, we want latent variables that are uncorrelated with each other

Enforce diagonal prior on the latent variables to encourage independence

Disentanglement

VAE summary

1. Compress representation of world to something we can use to learn

VAE summary

1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)

VAE summary

1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)
3. Reparameterization trick to train end-to-end

VAE summary

1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)
3. Reparameterization trick to train end-to-end
4. Interpret hidden latent variables using perturbation

VAE summary

1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)
3. Reparameterization trick to train end-to-end
4. Interpret hidden latent variables using perturbation
5. Generating new examples

BACKUP SLIDES

A probabilistic generative framework

- We will think of the code, $\vec{z}$, as an object that captures the essential properties of our input data, $\vec{x}$.
- We can make it easy to generate new, useful, codes—namely, we can engineer the *prior* distribution of codes $\mathbb{P}(\vec{z})$ to be a "tractable" probability distribution (such as a $\mathcal{N}(\vec{0}, \mathbf{I})$, the unit Gaussian).
- Once we have the code, generating data implies sampling the probability distribution $\mathbb{P}(\vec{x}|\vec{z})$. This can also be designed to be tractable (e.g. a learnt decoder neural network).
- This allows us to generate new data points!

Feel free to skip this heavy mathematical part if you feel difficult

A probabilistic generative framework

- Let's assume that the code prior $\mathbb{P}(\vec{z})$ and the generating distribution $\mathbb{P}(\vec{x}|\vec{z})$ are both parametrised by θ (these could be, say, the weights of a neural network).

- We will denote the prior by $p_\theta(\vec{z})$ and the generating distribution by $p_\theta(\vec{x}|\vec{z})$ to make this explicit.

- What do we have so far?

$$\xrightarrow{p_\theta(\vec{z})} \vec{z} \xrightarrow[\text{decoder}]{p_\theta(\vec{x}|\vec{z})} \vec{x}$$

Learning the generative framework

- Note that, having specified $p_\theta(\vec{z})$ and $p_\theta(\vec{x}|\vec{z})$, we can also derive an expression for $\mathbb{P}(\vec{x}) = p_\theta(\vec{x})$!
- This roughly corresponds to: "how likely is it that our model will generate $\vec{x}$?" and sounds like a great objective to maximise!
- We could train our model to maximise $p_\theta(\vec{x})$ over all the samples $\vec{x}$ in our training set.

Maximising the *evidence*

- Unfortunately, computing this quantity (sometimes called the *evidence*) requires *integrating over all possible codes*:

$$p_\theta(\vec{x}) = \mathbb{P}(\vec{x}) = \int_{\vec{z}\in\mathcal{Z}} \mathbb{P}(\vec{x}, \vec{z})\, \mathrm{d}\vec{z} = \int_{\vec{z}\in\mathcal{Z}} p_\theta(\vec{x}|\vec{z}) p_\theta(\vec{z})\, \mathrm{d}\vec{z}$$

  and this is <u>intractable</u> in all except the simplest of cases!

- We could try approximating it using *Monte Carlo* methods, but this does not scale for large datasets and large networks.

- It is possible to make this objective work—and we will see how! For now, we need to address an even more serious issue…

Performing inference

- For *inference* purposes, we also would very much like to be able to *attach codes* to known inputs $\vec{x}$.
- This would allow us to use the code for other purposes, such as *dimensionality reduction*, obtaining new inputs similar to $\vec{x}$ (e.g. for *data augmentation*) by modifying the code, etc.
- This involves sampling codes from the *posterior distribution* $\mathbb{P}(\vec{z}|\vec{x})$, and is almost always **insanely hard**!

Why is the posterior *so hard*?

- At a glance, it might seem like we could use Bayes' theorem to help us evaluate the posterior:

$$p_\theta(\vec{z}|\vec{x}) = \mathbb{P}(\vec{z}|\vec{x}) = \frac{\mathbb{P}(\vec{x}|\vec{z})\mathbb{P}(\vec{z})}{\mathbb{P}(\vec{x})} = \frac{p_\theta(\vec{x}|\vec{z})p_\theta(\vec{z})}{p_\theta(\vec{x})}$$

- However, the pesky *evidence* reappears in the denominator :(and this time we need to *sample* from something involving it, not just evaluate it...

  We approximate it using another distribution q, which we'll define such that it has a tractable distribution.

Variational autoencoder (*Kingma & Welling, 2015*)

- Our model of the world now looks like this:

$$\vec{x} \xrightarrow[\text{encoder}]{q_\phi(\vec{z}|\vec{x})} \vec{z} \xrightarrow[\text{decoder}]{p_\theta(\vec{x}|\vec{z})} \vec{x}'$$

- Looks like an *autoencoder*, doesn't it?
- Indeed—when p_θ and q_ϕ are specified by neural networks, this is the general setup of a **variational autoencoder** (VAE).
- The loss function is now a bit more complicated…

Towards a VAE loss

- As discussed, a VAE has **two objectives** that need to be *simultaneously satisfied*. For a training example, $\vec{x}$:
  - We want to make our *decoder* highly likely to generate this example—this implies **maximising**

$$\boxed{\log p_\theta(\vec{x})}$$

  - Simultaneously, we would like our *encoder* to not stray too far from the *true posterior* $p_\theta(\vec{z}|\vec{x})$. This implies **minimising**

$$\boxed{D_{KL}\left(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}|\vec{x})\right)}$$

where D_{KL} is the Kullback-Leibler (KL) divergence:

$$D_{KL}(q(x)\|p(x)) = \int_{x\in\mathcal{X}} q(x) \log \frac{q(x)}{p(x)} \mathrm{d}x$$

The evidence lower bound

- Combined, these objectives give us the **evidence lower bound** (ELBO), which our network needs to maximise:

$$ELBO(\theta, \phi) = \log p_\theta(\vec{x}) - D_{KL}\left(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}|\vec{x})\right)$$

- The troublesome posterior $p_\theta(\vec{z}|\vec{x})$ is **still** here...
- Luckily, we can rewrite ELBO in a form that *completely eliminates the posterior*!

$$ELBO(\theta, \phi) = \underbrace{\mathbb{E}_{\vec{z} \sim q_\phi(\vec{z}|\vec{x})}[\log p_\theta(\vec{x}|\vec{z})]}_{\text{Reconstruction accuracy}} - \underbrace{D_{KL}(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}))}_{\text{Regularisation}}$$

Derivation to follow in the next two slides—likely omitted.

ELBO derivation

Recall: $ELBO(\theta, \phi) = \log p_\theta(\vec{x}) - D_{KL}\left(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}|\vec{x})\right)$

$$
\begin{aligned}
\log p_\theta(\vec{x}) &= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log p_\theta(\vec{x}) \, \mathrm{d}\vec{z} \\
&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{p_\theta(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} \\
&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} \frac{q_\phi(\vec{z}|\vec{x})}{p_\theta(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} \\
&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} + \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{q_\phi(\vec{z}|\vec{x})}{p_\theta(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} \\
&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} + D_{KL}(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}|\vec{x}))
\end{aligned}
$$

Looks like we can cancel out the term containing the posterior! :)

ELBO derivation

Recall: $ELBO(\theta, \phi) = \log p_\theta(\vec{x}) - D_{KL}\left(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}|\vec{x})\right)$

$$\log p_\theta(\vec{x}) = \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} + D_{KL}(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}|\vec{x}))$$

Finally,

$$\begin{aligned}
ELBO(\theta, \phi) &= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} \\
&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}|\vec{z}) p_\theta(\vec{z})}{q_\phi(\vec{z}|\vec{x})} \, \mathrm{d}\vec{z} \\
&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log p_\theta(\vec{x}|\vec{z}) \, \mathrm{d}\vec{z} - \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{q_\phi(\vec{z}|\vec{x})}{p_\theta(\vec{z})} \mathrm{d}\vec{z} \\
&= \boxed{\mathbb{E}_{\vec{z} \sim q_\phi(\vec{z}|\vec{x})}[\log p_\theta(\vec{x}|\vec{z})] - D_{KL}(q_\phi(\vec{z}\|\vec{x}) \| p_\theta(\vec{z}))}
\end{aligned}$$

The ELBO function—analysed

$$ELBO(\theta, \phi) = \underbrace{\mathbb{E}_{\vec{z} \sim q_\phi(\vec{z}|\vec{x})}[\log p_\theta(\vec{x}|\vec{z})]}_{\text{Reconstruction accuracy}} - \underbrace{D_{KL}(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}))}_{\text{Regularisation}}$$

- The first term can be interpreted as the usual *autoencoder reconstruction loss*—given input $\vec{x}$, use the encoder to sample a code $\vec{z}$ from $q_\phi(\vec{z}|\vec{x})$, then use the decoder to compute $p_\theta(\vec{x}|\vec{z})$. We want to maximise this value!
- Minimising the second term forces the distribution of generated codes to remain close to the prior—prevents the network from "cheating" (by, e.g., assigning distant codes to every example)!

Practical choice of $p_\theta(\vec{x}|\vec{z})$

- Now I will present a specific example of a VAE implementation (which is also most common).
- Our decoder will generate samples $\vec{x}'$ coming from a Gaussian distribution, where each component is sampled independently with standard deviation one.
- Essentially, $p_\theta(\vec{x}|\vec{z}) = \mathcal{N}(\vec{\mu}, \mathbf{I})$.
- For computing the mean sample $\vec{\mu}$, we will utilise a neural network (with weights θ) of exactly the same kind as before—with **logistic output units**.
- We can then use the cross-entropy of the encoder input $\vec{x}$ and $\vec{\mu}$ as the reconstruction loss (just as before).

Practical choice of $q_\phi(\vec{z}|\vec{x})$

- For the encoder, we will use m Gaussian distributions to generate each of the m elements of the code.
- The parameters of these distributions ($\vec{\mu}$, $\vec{\sigma}$) are computed by a neural network (with weights ϕ and **linear output units**). This implies $q_\phi(\vec{z}|\vec{x}) = \mathcal{N}(\vec{\mu}, diag(\vec{\sigma}^2))$.
- If we use the prior $p_\theta(\vec{z}) = \mathcal{N}(\vec{0}, \mathbf{I})$, then the regularisation term has an expression we can easily work with!

$$-D_{KL}(\mathcal{N}(\vec{\mu}, diag(\vec{\sigma}^2))\|\mathcal{N}(\vec{0}, \mathbf{I})) = 1/2 \sum_{j=1}^{m} 1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2$$

Let $p = N(\mu_1, \sigma_1)$, $q = N(\mu_2, \sigma_2)$, $KL(p||q) = ?$

$$KL(p||q) = \int \left[\log(p(x)) - log(q(x))\right] p(x)dx$$

$$= \int \left[-\frac{1}{2}\log(2\pi) - \log(\sigma_1) - \frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2 + \frac{1}{2}\log(2\pi) + \log(\sigma_2) + \frac{1}{2}\left(\frac{x-\mu_2}{\sigma_2}\right)^2\right]$$
$$\times \frac{1}{\sqrt{2\pi}\sigma_1}\exp\left[-\frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2\right]dx$$

$$= \int \left\{\log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2}\left[\left(\frac{x-\mu_2}{\sigma_2}\right)^2 - \left(\frac{x-\mu_1}{\sigma_1}\right)^2\right]\right\} \times \frac{1}{\sqrt{2\pi}\sigma_1}\exp\left[-\frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2\right]dx$$

$$= E_1\left\{\log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2}\left[\left(\frac{x-\mu_2}{\sigma_2}\right)^2 - \left(\frac{x-\mu_1}{\sigma_1}\right)^2\right]\right\}$$

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2\sigma_2^2}E_1\left\{(X-\mu_2)^2\right\} - \frac{1}{2\sigma_1^2}E_1\left\{(X-\mu_1)^2\right\}$$

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2\sigma_2^2}E_1\left\{(X-\mu_2)^2\right\} - \frac{1}{2}$$

(Now note that
$(X-\mu_2)^2 = (X-\mu_1+\mu_1-\mu_2)^2 = (X-\mu_1)^2 + 2(X-\mu_1)(\mu_1-\mu_2) + (\mu_1-\mu_2)^2)$

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{1}{2\sigma_2^2}\left[E_1\left\{(X-\mu_1)^2\right\} + 2(\mu_1-\mu_2)E_1\left\{X-\mu_1\right\} + (\mu_1-\mu_2)^2\right] - \frac{1}{2}$$

$$= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{\sigma_1^2+(\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}$$

Generative Adversarial Networks (GANs)

What if we just want to sample?

Idea: don't explicitly model density, and instead just sample to generate new instances.

Problem: want to sample from complex distribution – can't do this directly!

Solution: sample from something simple (noise), learn a transformation to the training distribution.

Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a way to make a generative model by having two neural networks compete with each other.

The **discriminator** tries to identify real data from fakes created by the generator.

The **generator** turns noise into an imitation of the data to try to trick the discriminator.

Intuition behind GANs

Generator starts from noise to try to create an imitation of the data.

Intuition behind GANs

Discriminator looks at both real data and fake data created by the generator.

Discriminator

Generator

Fake data

Intuition behind GANs

Discriminator looks at both real data and fake data created by the generator.

Discriminator

Generator

Real data

Fake data

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Generator tries to improve its imitation of the data.

Intuition behind GANs

Generator tries to improve its imitation of the data.

Intuition behind GANs

Generator tries to improve its imitation of the data.

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.

Intuition behind GANs

Generator tries to improve its imitation of the data.

Intuition behind GANs

Generator tries to improve its imitation of the data.

Intuition behind GANs

Generator tries to improve its imitation of the data.

Intuition behind GANs

Discriminator tries to identify real data from fakes created by the generator.
Generator tries to create imitations of data to trick the discriminator.

Training GANs

Discriminator tries to identify real data from fakes created by the generator.
Generator tries to create imitations of data to trick the discriminator.

Train GAN jointly via **minimax** game:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log \left(1 - D_{\theta_d} \left(G_{\theta_g}(z) \right) \right) \right]$$

Discriminator wants to maximize objective s.t. $D(x)$ close to 1, $D(G(z))$ close to 0.
Generator wants to minimize objective s.t. $D(G(z))$ close to 1.

Why GANs?

A. Courville, 6S191 2018.

Why GANs?

more traditional max-likelihood approach

GAN

A. Courville, 6S191 2018.

Generating new data with GANs

After training, use generator network to create **new data** that's never been seen before.

GANs: Recent Advances

Progressive growing of GANs (NVIDIA)

Karras et al., ICLR 2018.

Progressive growing of GANs: results

Karras et al., ICLR 2018.

Deep Generative Modeling: Summary

Autoencoders and Variational Autoencoders (VAEs)

Learn **lower-dimensional** latent space and **sample** to generate input reconstructions

Generative Adversarial Networks (GANs)

Competing **generator** and **discriminator** networks