What physicists want to know about advances in generative modeling

Michael Abergo

We hope to provide you all at KITP with an overview of one class of machine learning approaches, namely generative modeling. In this, we will discuss background on what generative models try to achieve and some recent advances in their approaches.

What are Generative Models?

One of the major goals of machine learning is to understand the essential parameters explaining why a dataset is the way it is. Most commonly, this is seen as a problem of building a model that can learn a probability distribution that *discriminates* some data from other data. In the case of functional mapping in which a $\mathbf{y}$ is associated to an input $\mathbf{x}$, this means learning a conditional probability $p(\mathbf{y}|\mathbf{x})$, where the likelihood of the output value is predicted given some input condition. In applicable cases like classification or regression, the goal is to discriminatively ascertain likely $\mathbf{y}$ values for a given $\mathbf{x}$. Other times, when a greater understanding of a functionally mapped space is desired beyond the distribution of some output conditioned on some input, the goal is to learn a joint probability $p(\mathbf{x}, \mathbf{y})$ -- to gain insight into the distribution that is responsible for *generating* the data.

Even more generally, there are many instances when there is no relational mapping or "labeling" behind the data distribution we are curious about. That is, we are only given some $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ that are distributed according some probability distribution $p(\mathbf{x})$. In such case, the objective is to model the true distribution $p(\mathbf{x})$ with some parameterized approximation $p_\theta(\mathbf{x})$ so that we can generalize on i) estimating the likelihood of some $\mathbf{x}$ in the domain and on ii) making novel samples that fall under the distribution. Approaches to this problem are detailed below.

Why would we want to do this?

Generative models can offer a variety of benefits or functional alternatives to discriminative approaches, some of which might be quite important for what physics want to accomplish:

- efficiently performing inference on multi-model and high-dimensional distributions
- meausring likelihoods for statistical and experimental analysis
- filling holes in existing data by extrapolating from the assembled parts of distributional knowledge
- sampling from high-dimensional distributions to make calculations
- super-resolution images

These techniques could provide insight in validating experimental setups, expedite simulation tasks, and approve data abundance/quality, among other things.

Evolution of neural generative models

Our perspectives on generative modeling with neural network approaches have evolved. Where did they start?

Unsupervised learning with neural network architectures began with Hopfield and Hinton in the 1980s with the Hopfield Network and Restrictred Boltzmann Machine (RBM). We will not go into much detail about them here, but their details can be reviewed here and here. In general, these are traditionally binary, shallow neural network models that store an associated memory of an event or learn to reconstruct input. The energy minimization principles behind them lend themselves nicely to a physics perspective, but the advent of deep learning and its interplay with statistical learning theory over the past 15 years have ushered in a new era of deep generative modeling. The main approach that we will maintain our attention toward are methods that seek to maximize the likelihood $p(x)$ of the data.

Different modern approaches to generative modeling

Each novel approach to generative modeling has its advantages and limitations. By their history and their approach to the objective of providing inference on a distribution, we can categorize generative models into three categories, as per Ian Goodfellow's NIPS 2016 GAN tutorial:

1. Explicit and **Tractable** Likelihood
    - Pixel methods (RNN, CNN)
    - Flow based models (NICE, Real NVP, Glow)
2. Explicit and **Approximate** (bounded) Likelihood
    - Variational Bayes (autoencoding)
    - Boltzmann Machines
3. **Implicit** Likelihood
    - GAN

Note: I might use the terms likelihood and density interchangeably here. Also note: I've included a few flow based models under the tractable likelihood category that I think are important to mention.

1. Explicit and Tractable Likelihood Models

Pixel RNN / CNN

Pixel generative models are techniques that break the full likelihood estimate $p(x)$ into the product of many sequential conditional probabilities. We can imagine that you perform inference pixel by pixel and use the chain rule of probability to amass an a measure of $p(x)$ such that:

$$p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_1,x_2)\ldots = p(x_1)\prod_{i=2}^{n} p(x_i|x_1,\ldots,x_{i-1}) \tag{16}$$

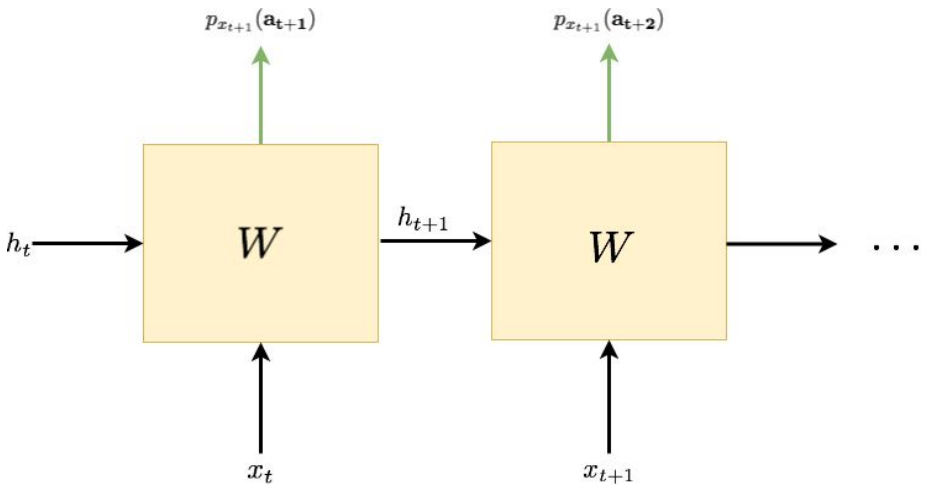
We can take a look at an example in the context of an RNN based on some recent work by Carrasquilla et al, "Reconstructing Quantum States with Generative Models."" Imagine that we have a 1D lattice of qubits that we want to perform measurements on to infer information about the distribution of measurement outcomes and relate that to reconstructing the quantum state.

Let's say we have a 50-qubit quantum system that is governed by the Transverse-Field Ising Model 1D Hamiltonian:

$$H(\sigma) = -J\sum_{i=1}^{L}\sigma_i^z\sigma_{i+1}^z - h\sum_i \sigma_i^x \tag{17}$$

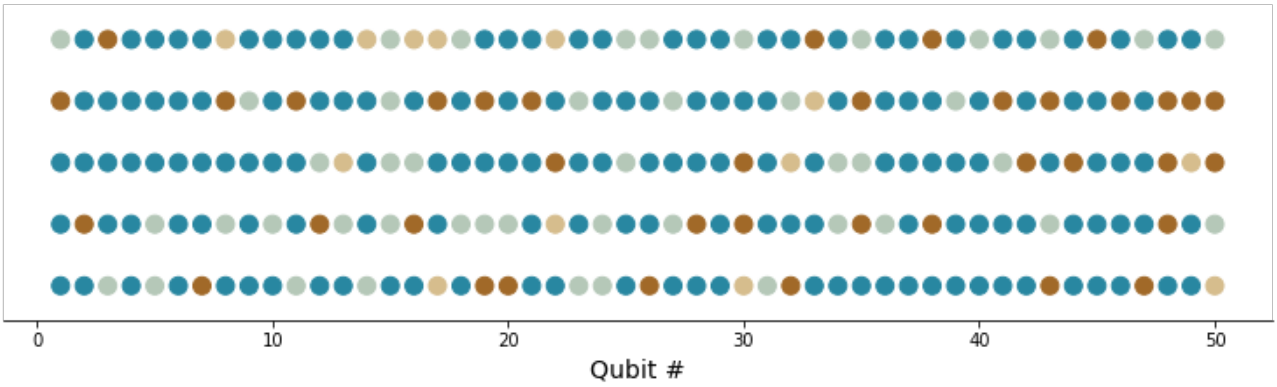
You want to be able to check on the fidelity of your setup to verify that you have properly prepared the quantum state. To do this, you want to perform inference on the density matrix of the mixed state -- to show that informationally complete measurement samples from your setup match the expected frequency associated with your intended density matrix ρ. With the right choice of measurements -- namely informationally complete Postive-Operator Valued Measurements for each qubit $M = \{M^{(\mathbf{a_1})} \otimes \cdots \otimes M^{(\mathbf{a_n})}\}$, we can link the probability of measurement outcomes with the invertible expression given by Born's Rule: $P(\mathbf{a}) = \mathrm{Tr}[M^{(\mathbf{a})}\rho]$. Thus, if we can gain insight into the distribution of measurement ourcomes $P(\mathbf{a})$, we can gain insight into the density matrix ρ. Here, each a_i in $\mathbf{a}$ can take on a discrete value to represent the measurement outcome for the i^{th} qubit. The number of possible outcomes is decided by which POVM measurements are used. Let's say we have the Pauli-4 POVM and we then have 4 possible measurement outcomes $a = 0, 1, 2, 3$. I will one-hot encode them to the vectors $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$. A given data example for our 50 qubits would be 50 of these one-hot vectors. We want to feed each of these sequentially into our RNN unit.

An RNN is made of a recurrent unit that takes in a hidden state h_t (used for updating computation at different sequential states) and an input vector x_t, which in our case is one of these one-hot vectors of measurement outcomes. The weight matrix W is involved with some computation with x_t and h_t (which will vary depending on what type of RNN unit you use, like a GRU or LSTM).



At the first step $t = 0$, we feed in some fixed initial h_0, x_0 of arbitrary value. The output of the RNN at the first time sequence is a probability distribution over measurement outcomes for the first qubit and an update to the hidden state h_t. That is, the output of the RNN on the first go will give us $\mathbf{p_t} = \begin{bmatrix} p_t(a_0) \\ p_t(a_1) \\ p_t(a_2) \\ p_t(a_3) \end{bmatrix}$ and our new h_t. During training, we then take the real first qubit measurement outcome from our training example and feed that in to the RNN unit as x_t with h_t to get our estimate $\mathbf{p_{t+1}}$. We do this for each sequential qubit in our model. We compute the loss by calculating the cross entropy between each $\mathbf{p_t}$ and the sample from our training data x_t for all t in our qubit chain and sum.

When we want to sample the model, we alter this process a bit. After outputting a set of probabilities for the measurement outcome of the first qubit, we draw a sample outcome from this set of probabilities and feed that into the next step of the RNN rather than the training sample outcome. That is, say we compute the probabilities for measurement outcomes of the first qubit and get $\mathbf{p_1} = \begin{bmatrix} .78 \\ .04 \\ .1 \\ .08 \end{bmatrix}$. We then sample from this and get $x_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, and this is what we feed as input into the RNN for the next pass estimating $\mathbf{p_2}$. Doing this for all qubits, you get a probaility distribution for each one that is dependent on the outcomes of those that preceded it. You can then start generating the measurements on 50 qubit lattices and using that to calculate expectations of observables for your density matrix ρ:



You can compute the Classical Fidelity F_c and the KL-divergence KL between P_{RNN} and P_{TFIM} (ground truth):

$$F_c = \sum_i \sqrt{P_{RNN}(i) \quad P_{TFIM}(i)} \approx \frac{1}{N_s} \sum_i e^{\frac{1}{2}(\log P_{RNN}(i) - \log P_{TFIM}(i))} \tag{19}$$

$$\mathrm{KL} = \sum_i P_{TFIM}(i) \log \frac{P_{TFIM}(i)}{P_{RNN}(i)} \approx \frac{1}{N_s} \sum_i (P_{TFIM}(i) - P_{RNN}(i))$$

For PixelCNNs rather than RNNs, imagine that the sequential steps come from 1D convolutions.

Advantages:

- Tractable likelihood estimate is useful for inference
- MLE provides stable training paradigms.
- Tractable sampling procedure.

Disadvantages:

- Estimating complete $p(\mathbf{x})$/sampling can be slow when data has to be treated as a long sequence
- Model can struggle to learn impact of very early parts of the probability chain on later conditional probabilities (can improve this with GRU and LSTM as RNN unit, or stacking them, but only goes so far)
- Ordering of sequence can be arbitrary
- Not all data amenable to this one-hot encoding.

Flow Based Models

A recently introduced class of generative modes with explicit likelihood estimate is those that make use of normalizing flows. Normalizing flows help learn an invertible transformation f (or set of invertible transformations for deeper models) such that f performs a mapping between probability distributions. If you can make a sufficiently versatile representation of f (say, with a neural network), then you should be able to transform most any probability distribution into another. Let's build the mathematical tools out for describing this. Say we have some $\mathbf{z} \sim p(\mathbf{z})$ and $\mathbf{x} \sim p(\mathbf{x})$ and we want to learn our generative mapping f which takes some $\mathbf{x}$ from the data space and transforms it to some $\mathbf{z}$ in the latent space . We can use the change of variables formula to do this if $f : \mathcal{X} \to \mathcal{Z}$ is invertible:

$$p(\mathbf{x}) = p_z(f(\mathbf{x}))\left|\det \frac{\partial f(x)}{\partial x^T}\right| \tag{4}$$

Note: because we assume $\mathbf{z}$ is multi-dimensional, we label $\frac{\partial f(x)}{\partial x^T}$ as the Jacobian of the f. This seems simple enough, so there must be a catch. And that catch is in coming up with a way to represent a highly flexible and complicated function f that one can easily invert. A number of papers by Dinh et al. (NICE and RealNVP) as well as a recent paper by Kingma et al. (Glow) build up and optimize some clever methods to make this inversion possible, the basics of which we'll describe here.

In general, it is difficult to learn invertible functions. Additionally, the above equation is computational inefficient because calculating the Jacobian of a large matrix is intractable. This can circumvented if f is constructed with what we'll call **additive or affine coupling layers**. NICE uses additive coupling layers, while RealNVP uses affine coupling layers. These layers are ways of splitting the invertible function into subparts so that the Jacobian calculation is tractable.

- Additive Coupling:

$$\begin{aligned} \mathbf{y} &= \begin{cases} y_{1:d} = x_{1:d} \\ y_{d+1:D} = x_{d+1:D} + b(x_{1:d}) \end{cases} \\ &\Downarrow \\ \mathbf{y}^{-1} &= \{ \begin{aligned} x_{1:d} &= y_{1:d} \\ x_{d+1:D} &= y_{d+1:D} - b(x_{1:d}) \end{aligned} \end{aligned} \tag{20}$$

- Affine Coupling:

$$\begin{aligned} \mathbf{y} &= \begin{cases} y_{1:d} = x_{1:d} \\ y_{d+1:D} = x_{d+1:D} \odot \exp\left(s(x_{1:d})\right) + t(x_{1:d}) \end{cases} \\ &\Downarrow \\ \mathbf{y}^{-1} &= \{ \begin{aligned} x_{1:d} &= y_{1:d} \\ x_{d+1:D} &= (y_{d+1:D} - t(y_{1:d})) \odot \exp\left(-s(y_{1:d})\right) \end{aligned} \end{aligned} \tag{6}$$

- Coupling + 1x1 Convolutions:
    - These are used in the Glow paper to help with image understanding. Details can be found in their paper and aren't esseential for understanding the flow-based modeling framework.

The coupling allows for us to compute a Jacobian defined by a triangular matrix, which is significantly more efficient than would otherwise be possible. The Jacobian J of these coupled functions looks like:

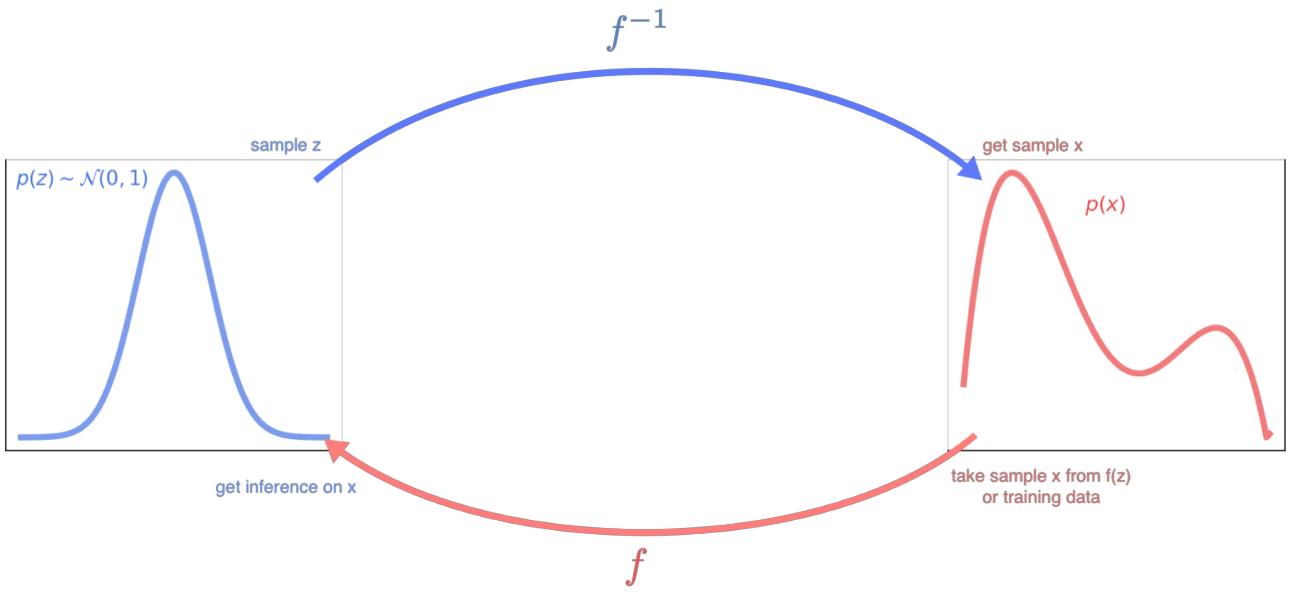
$$J = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} 1 & \dots & \dots & 0 \\ \frac{\partial y_2}{\partial x_1} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ \frac{\partial y_D}{\partial x_1} & \frac{\partial y_D}{\partial x_2} & \dots & \frac{\partial y_D}{\partial x_D} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0}_d \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}} & \frac{\partial y_{d+1:D}}{\partial x_{d+1:D}} \end{bmatrix} \tag{7}$$

where I_d is the identity matrix of dimension d. We write it like such to show the familiar form of a 2x2 matrix. This is really convenient! We need to calculate the determinant of this matrix to do our transformations and inverse transformations between probability density functions. If you recall that for a 2x23 matrix $M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, $\det M = \frac{1}{ad-bc}$, we can see that the determinant of our Jacobian is just the determinant of the smaller sub matrix defined by $\frac{\partial y_{d+1:D}}{\partial x_{d+1:D}}$. For as complex a transformation as the affine coupling, this only amounts to being able to compute:

$$\det J = \det \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}} & \text{diag}(\exp\left(s(x_{1:d})\right)) \end{bmatrix} = \exp(\sum_{k=1}^{D-d} s(x_{1:d})_k) \tag{8}$$

If you stack at least two of these coupling layers together, you can transform all of your data. Moreover, one can see that computing neither the Jacobians, nor the inverses, require explicit sub-computations of the scaling in NICE (function m) or the scaling and transformation in RealNVP (functions s and t). As such, these functions can be complicated and that won't impact the difficulty in calculating the inverse or the Jacobian.

With this flexibility of inversion, we can perform sampling and exact inference. If I draw some $\mathbf{z}$ from p_z and I know p_z is an analytically known distribution like a Gaussian $\mathcal{N}(0, \mathbb{I})$, I can pass it through f^{-1} to get a sample $\mathbf{x}$ from p_x. If I want to perform inference on a sample $\mathbf{x}$, I can pass it through f to understand its likelihood in the known latent density:



In practical optimization contexts, we maximize the log likelihood of the data rather so that the multiplication in Equation 4 can be split into additive terms whose optimization is more manageable to pursue:

$$\ln p(\mathbf{x}) = \ln\big(p_z(f(\mathbf{x}))\big) + \ln\left(\left|\det \frac{\partial f(x)}{\partial x^T}\right|\right) \tag{9}$$

Advantages:

- Efficient and exact inference $\rightarrow$ done through estimating the log density
- Efficient image reconstruction $\rightarrow$ invertibility

Disadvantages:

- Generally need to stack many coupling layers together, which, additively, can be computationally slow
- Images can be less resolved than with GAN, but recent updates like with Glow have challenged that claim!
- There are restrictions still on what functions we make because of needing a cheap Jacobian to compute, but Duvenaud et al. recently introduced a technique for using their Neural ODEs to do circumvent this

Note: Flow-based techniques can be combined with other approaches! For example, normalizing flows were introduced to VAEs in 2015 with the paper BLANNNKKKK by BLANKKKKK, and autoregressive models can be hybridized with flow techniques, as per Inverse Autoregressive Flows and Masked Autoregressive Flows.

2. Explicit and Approximate Likelihood Models

Variational Autoencoders

Under a similar vein as the flow based models in which we map our data $\mathcal{X} = \{\mathbf{x}\}_{i=1}^n$ to some known tractable variable $\mathcal{Z} = \{\mathbf{z}\}$, we can also approach this problem that the $\mathbf{z}$ variables are actually a set of hidden or latent variables that drive processes from which the data $\mathbf{x}$ are generated. That is, we can imagine that these latent variables drive and can be used to explain the observable data and how its distributed. This is called the latent variable model and has been used often in statistical modeling. By such, it is often used in research in the machine learning community, but it has a few limitations. We can describe the process of inference on $p(\mathbf{x})$ as calculating

$$p(\mathbf{x}) = \int_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}) = \int_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \tag{18}$$

We run into the problem of having to marginalize out all of $\mathbf{z}$ to exactly calculate p_x. For high-dimensional systems that we often encounter in machine learning. In such case, we also can't access it through non-integral Baye's Rule because the posterior density $p(\mathbf{z}|\mathbf{x})$ is also intractable. To overcome this, Kingma and Welling proposed a method of putting an optimizable lower bound on $p(\mathbf{x})$ so that one can have a model that provides approximate, bounded measure of the likelihood of the data.

Let's take this latent model and run with it a bit. We want to generate some $\mathbf{x}$ from $\mathbf{z}$ and thereby need a model which tell us about $p(\mathbf{x}|\mathbf{z})$. We can estimate this with a neural network defined by parameters θ as $p_\theta(\mathbf{x}|\mathbf{z})$ that will tell us about the probability of observing some $\mathbf{x}$ given some $\mathbf{z}$ to be *decoded*. We can choose our latent space to follow some simple distribution such as a Gaussian so $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbb{I})$. Because marginalizing out $\mathbf{z}$ is intractable, we have to start by performing inference on $p(\mathbf{x}|\mathbf{z})$. We can facilitate this by first encoding our data into a latent space by learning to model $p(\mathbf{z}|\mathbf{x})$. We can estimate this with a neural network defined by parameters ϕ as $q_\phi(\mathbf{z}|\mathbf{x})$, where q will be used to represent the density of our samples that are *encoded* into the latent space.

I have used the words decode and encode to hint at the structure of the model. These words are used often to describe the two subparts of an **autoencoder**. A conventional autoencoder is a deterministic framework for learning to encode data $\mathbf{x}$ into a dimensionally reduced space and then subsequently decode it for reconstruction. If the data can be reconstructed from the dimensionally reduced space, then the dimensionally reduced space captures the information behind the data well.

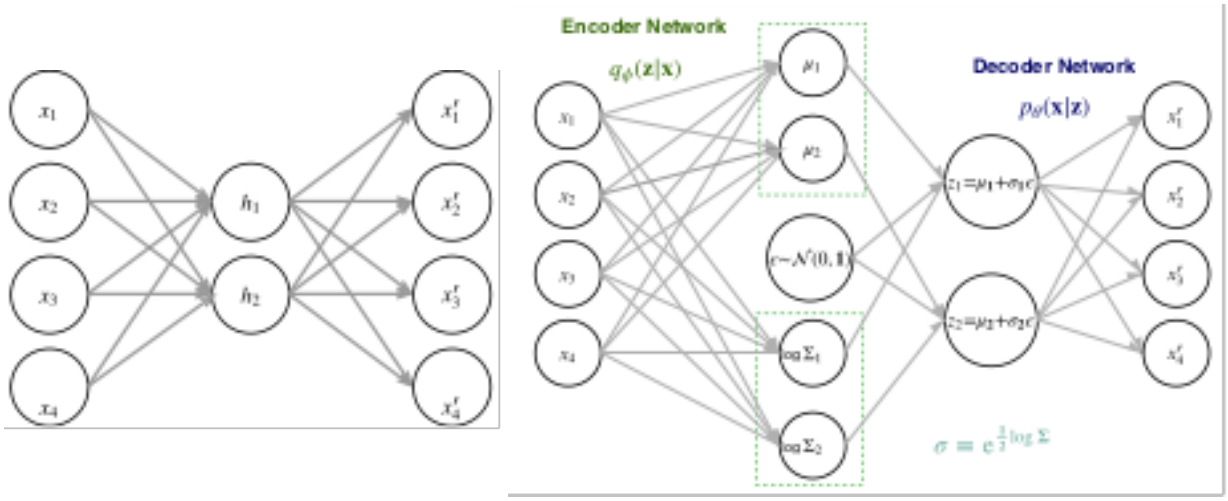
In our context, we want to put a probabilistic spin on this interpretation, where now the encoder represents our probability distribution $q_\phi(\mathbf{z}|\mathbf{x})$ and the decoder represents our $p_\theta(\mathbf{x}|\mathbf{z})$. We can write out an approximation for the log-likelihood of the data with these terms, as is done here:

$$
\begin{aligned}
\ln p_\theta(\mathbf{x}) &= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\ln p_\theta(\mathbf{x})] \\
&= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\ln \frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{z}|\mathbf{x})}\frac{q_\phi(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})}] \\
&= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\ln p_\theta(\mathbf{x}|\mathbf{z})] - \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\ln \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z})}] + \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\ln \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}|\mathbf{x})}] \\
&= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\ln p_\theta(\mathbf{x}|\mathbf{z})] - D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\,||\,p_\theta(\mathbf{z})) + D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\,||\,p_\theta(\mathbf{z}|\mathbf{x}))
\end{aligned}
\tag{11}
$$

We expand the log-likelihood into terms from our model and end up with three. The expectation over $\mathbf{z}$ equates the last two logarithmic fractions to KL-divergences. The **first** term is a posterior likelihood term that can be approximated on small minibatches with with a mean square error or a binary cross entropy. We can consider the reconstruction loss term as ensuring that real samples are embedded into the latent distribution and forcing the decoding function to be an approximate inverse of the encoding function. The **second** term is a measure of the similarity between the encoded distribution $q_\phi(\mathbf{z}|\mathbf{x})$ and our (generally) Gaussian prior, and the third term is intractable. However, $D_{KL}(\cdot||\cdot) \geq 0$, so the first two terms can still function as a differentiable lower bound on the log-likelihood $\ln p_\theta(\mathbf{x})$ -- also known as the also known as the *evidence lower bound* (ELBO).

$$
\ln p(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\ln p_\theta(\mathbf{x}|\mathbf{z})] - \beta\, D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\,||\,p_\theta(\mathbf{z})) \tag{12}
$$

Because it is differentiable, it can be optimized to maximize the lower bound of the likelihood. Using this variational principle, this has been dubbed a variational autoencoder (VAE).



Graphs of prototypical autoencoders and VAEs models are shown in the figure above to highlight their differences. An important distinction beyond the different objective function is in the encoding bottleneck. Instead of encoding down to some single latent representation, a VAE encodes to two bottlenecks -- a mean vector and a variance vector. The latent vector is then created by sampling the mean vector, the variance vector, and combining these samples with a noise term ϵ. This is to permit the VAE to generate new samples while still allowing the graph to be differentiable for backpropagation optimization and is known as the reparameterization trick. Autoencoders and VAEs are only cousins because of their encoding and decoding paradigm; otherwise, they are conceptually (Bayesian vs deterministic) and functionally (reductive vs generative) quite different.

You might be asking why the encoder maps to a μ and $\ln\Sigma$? This is to make the calculation of that second KL-term analytically calculable. Those two parameters define the Gaussian $q_\phi(\mathbf{z}|\mathbf{x})$ for us, so we can compute a closed form representation of $D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\,||\,p_\theta(\mathbf{z}))$ by treating μ and Σ as functions of $\mathbf{x}$:

$$
\begin{aligned}
D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\,||\,p_\theta(\mathbf{z})) &= D_{\mathrm{KL}}(\mathcal{N}\big(\mu(\mathbf{x}), \Sigma(\mathbf{x})\big)\,||\,\mathcal{N}(\mathbf{0}, \mathbb{I})) \\
&= \frac{1}{2}\bigg(\sum_d(\Sigma(\mathbf{x}) + \mu^2(\mathbf{x})) - \ln\prod_d\Sigma(\mathbf{x})\bigg) - \frac{1}{2}d \\
&= \frac{1}{2}\bigg(\sum_d\Sigma(\mathbf{x}) + \mu^2(\mathbf{x}) - 1 - \ln\Sigma(\mathbf{x})\bigg) \\
&= \frac{1}{2}\bigg(\sum_d\exp(\Sigma(\mathbf{x})) + \mu^2(\mathbf{x}) - 1 - \Sigma(\mathbf{x})\bigg)
\end{aligned}
\tag{13}
$$

where d is the dimensions of the latent space, and, in the last line, we switch to the exponent because one achieves more stable results computing $\ln\Sigma$ than Σ.

Advantages:

-

Disadvantages:

-

2. Implicit Likelihood Models

Generative Adversarial Networks

A common dilemma in generative modeling is choosing the right evaluation metric in your training regime,\footnote{See Goodfellow, et al 2016 \cite{Goodfellow2016} for further discussion of maximum likelihood approaches in generative modeling.} as likelihood calculations on latent variable and energy maximization techniques are generally intractable to compute \cite{Theis2016}. While proxy metrics related to likelihood have been used instead, a novel training technique known as adversarial training replaces the traditional likelihood estimation with a trainable network, whose task is to discriminate generated samples that came from $p_\theta(\mathbf{x})$ from those of the true $p(\mathbf{x})$. Generative adversarial networks (GANs) conceived in recent years follow this paradigm \cite{Goodfellow2014}.

GANs function as a competition between two competing neural networks - a generator and a discriminator -- which are both represented by functions that are differentiable with respect to inputs and weight parameters. Let the discriminator and generator have trainable network parameters ϕ and θ respectively. The generator takes in some latent noise vector $\mathbf{z}$ and seeks to output a conditional signal $p_\theta(\mathbf{x}|\mathbf{z})$. The discriminator takes in either the output of the generator which is a sample from the approximate distribution or a sample from the true distribution and outputs a sigmoid value of the probability that the given sample is real or counterfeit. The two functions share a cost function V -- one that the generator tries to minimize and the discriminator tries to maximize:

$$\min_G \max_D V(D, G) = \mathbb{E}_x[\ln(D_\phi(\mathbf{x}))] + \mathbb{E}_z[\ln(1 - D(G_\theta(\mathbf{z}))] \tag{14}$$

where ϕ and θ are the parameters of the discriminator and the generator, respectively. The training process on this value function with an optimal discriminator has been shown to be equivalent to minimizing the Jensen-Shannon divergence \cite{Goodfellow2014} given by:

$$D_{\text{JS}} = \frac{1}{2} D_{\text{KL}}(P_{real} || \frac{1}{2}(P_{real} + P_{gen})) + D_{\text{KL}}(P_{gen} || \frac{1}{2}(P_{real} + P_{gen})) \tag{15}$$

for KL divergence $D_{\text{KL}}(P||Q) = -\sum_i P(i) \ln \frac{P(i)}{Q(i)}$. The ultimate goal is for the generator to create samples that are coming from an approximate distribution so close to the real distribution that the discriminator can no longer tell them apart from real samples. It should be noted that because in real world applications the model does not have access to the full distribution $p(\mathbf{x})$, the generator can only best learn the training distribution $\hat{p}(\mathbf{x})$.