From Autoencoders to betaVAE: A Survey

Pradeep Singh

Computational Science Research Center,
San Diego State University

Dec 2018

Abstract

Autocoders are a family of neural network models that aims to learn compressed latent representation of high-dimensional data. In this project, I study, review and implement autoencoders in various forms: basic autoencoder, denoising, sparse, and contractive autoencoders, and then Variational Autoencoder (VAE) and its modification beta-VAE. The goal of this project is to study and understand how autoencoders (and it's variants) work.

1 Introduction

Autoencoders are a family of artificial neural network that learns (without any supervision) how to encode (and decode) data from high-dimensional space to low-dimensional space and vice-versa. Autoencoders were invented to reconstruct high-dimensional data using a neural network model with a narrow bottleneck layer in the middle. A nice byproduct is dimension reduction: the bottleneck layer captures a compressed latent encoding. Such a low-dimensional representation can be used as en embedding vector in various applications (i.e. search), help data compression, or reveal the underlying data generative factors.

This idea was originated in the 1980s, and later promoted by the seminal paper by Hinton & Salakhutdinov, 2006.

Autoencoders, by design, reduces data dimensions by learning how to ignore the noise in the data. Every autoencoders is consists of 4 main parts:

- Encoder: Neural network model that learns how to reduce the input dimensions and compress the input data into an latent representation.
- Bottleneck: Hidden layer that contains the compressed representation of the input data.
- Decoder: Neural network model that learns how to reconstruct the data from the encoded representation to original space.
- Reconstruction Loss: Measure for how well decoder can reconstruct output in comparison to original input.

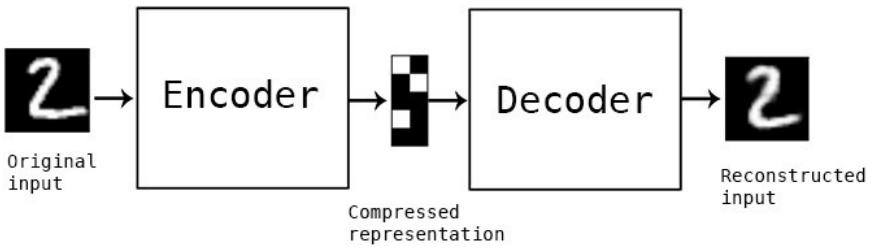


Figure 1: Simple architecture for an autoencoder, consisting of encoder and decoder network. Image source: Will Badr, 2019

The training then involves using back propagation in order to minimize the network's reconstruction loss.

2 Vanilla Autoencoder

Vanilla autoencoder, also known as simple autoencoder is consist of two networks: encoder and decoder.

1. Encoder network: It translates the original high-dimension input into the latent low-dimensional code. The input size is larger than the output size.
2. Decoder network: The decoder network recovers the data from the code, likely with larger and larger output layers.

The encoder network essentially accomplishes the dimensionality reduction, just like how we would use Principal Component Analysis (PCA) or Matrix Factorization (MF) for. In addition, the autoencoder is explicitly optimized for the data reconstruction from the code. A good intermediate representation not only can capture latent variables, but also benefits a full decompression process.

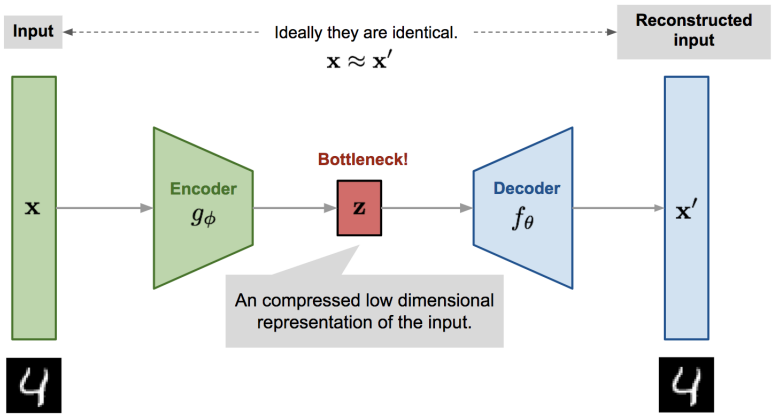


Figure 2: An autoencoder model architecture. Image source: Lilian Weng, 2018

The model contains an encoder function $g(.)$ parameterized by ϕ and a decoder function $f(.)$ parameterized by θ. The low-dimensional code learned for input x in the bottleneck layer is z= and the reconstructed input is $x = f_\theta(g_\phi(x))$.

The parameters (θ, ϕ) are learned together to output a reconstructed data sample same as the original input, $x \approx f_\theta(g_\phi(x))$, or in other words, to learn an identity function. There are various metrics to quantify the difference between two vectors, such as cross entropy when the activation function is sigmoid, or as simple as MSE loss:

$$L_{AE}(\theta, \phi) = \frac{1}{n} \sum_{n=1}^{n} x^{(i)} - f_\theta(g_\phi(x^{(i)}))^2$$

Vanilla autoencoder can be of two types depending upon how many units hidden layers have;

- Undercomplete: If hidden layers dimesnionality is less than input/ previous hidden layer. This is also a way of constraining autoencoders to learn the most salient features of the training data.
- Overcomplete: If hidden layers dimesnionality is more than input/ previous hidden layer.

Since, all that autoencoder does is, it learns the identity function (in other words; it learns how to copy data from input to the output), it is very prone to overfitting. In cases like overcomplete autoencoders, where model has large capacity to learn noise and data, we will need some sort of regularization methods to put constraint on model, so that it learns some useful representation of data rather than just learning the data itself.

3 Regularized Autoencoder

Since the autoencoders learn the identity function, it could "overfit" when there are more network parameters than the number of data points. To avoid overfitting and improve the robustness, we can introduce some regularization in the training process. This will put constraints on the model and will help model in learning some useful representation of data.

3.1 Denoising Autoencoder

Denoising Autoencoder (Vincent et al. 2008) is basically a vanilla autoencoder only, where input is partially corrupted by adding some noise and then the model is trained to recover the original input.

Let's say we have some input x which is partially corrupted by adding noises to or masking some values of the input vector in a stochastic manner, $\hat{x} \sim M_D(\hat{x}|x)$.

$$\hat{x}^{(i)} \sim M_D(\hat{x}^{(i)}|x^{(i)})$$

$$L_{DAE}(\theta, \phi) = \frac{1}{n} \sum_{n=1}^{n} x^{(i)} - f_\theta(g_\phi(\hat{x}^{(i)}))^2$$

where M_D defines the stochastic mapping from the true data samples x to the noisy or corrupted ones $\hat{x}$. The data corruption (addition of noise) is not specific to a particular type of corruption process (i.e. masking noise, Gaussian noise, salt-and-pepper noise, etc.), it can also be equipped with prior knowledge (eg: some prior distribution).

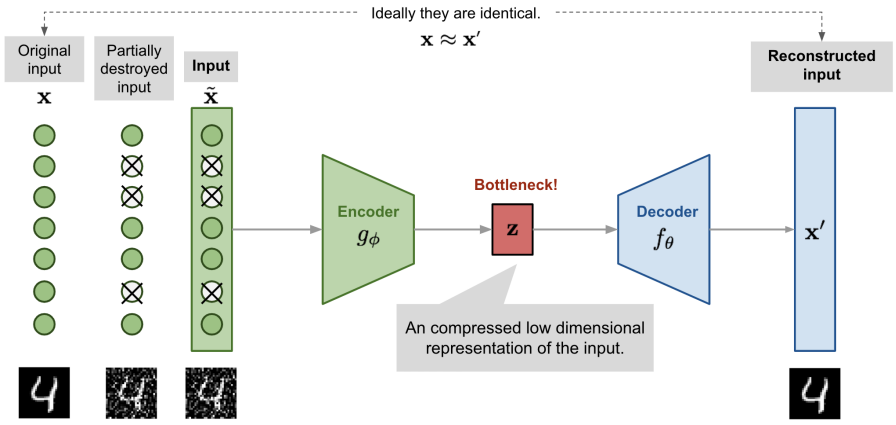


Figure 3: An Denoise Autoencoder model architecture. Image source: Lilian Weng, 2018

3.2 Sparse Autoencoder

Sparse Autoencoder applies a "sparse" constraint on the hidden activation units to avoid overfitting and improve robustness. It forces the model to only have a small number of hidden units being activated at the same time, or in other words, one hidden neuron should be inactivate most of time.

Let's say there are s neurons in the l^{th} hidden layer and the activation function for the j^{th} neuron in l^{th} layer is labelled as $a_j^{(l)}(.)$, $j = 1, \ldots, s_l$. The fraction of activation of this neuron $\hat{\rho}$ is expected to be a small number ρ, known as sparsity parameter; a common config is $\rho = 0.05$.

$$\hat{\rho}_j^{(l)} = \frac{1}{n} \sum_{n=1}^{n} [a_j^l(x^i)] \approx \rho$$

This constraint is achieved by adding a penalty term into the loss function. The KL-divergence D_{KL} measures the difference between two Bernoulli distributions, one with mean ρ and the other with mean $\hat{\rho}_j^{(l)}$. The hyperparameter β controls how strong the penalty we want to apply on the sparsity loss.

$$L_{SAE}(\theta) = L(\theta) + \beta \sum_{l=1}^{L} \sum_{j=1}^{s_l} D_{KL}(\rho || \hat{\rho}_j^{(l)})$$

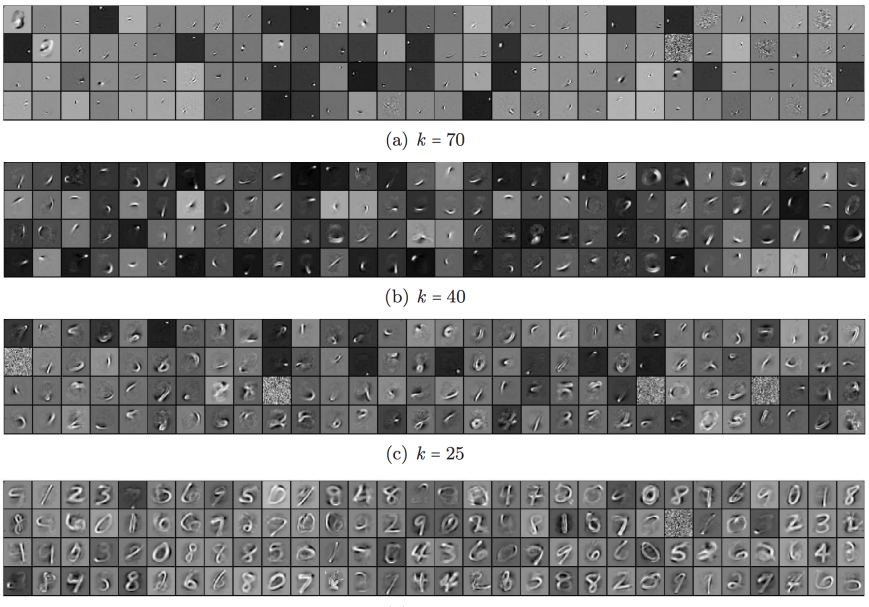
$$L_{SAE}(\theta) = L(\theta) + \beta \sum_{l=1}^{L} \sum_{j=1}^{s_l} \rho log \frac{\rho}{\hat{\rho}_j^{(l)}} + (1-\rho) log \frac{1-\rho}{1-\hat{\rho}_j^{(l)}}$$

3.3 k-Sparse Autoencoder

One modification of sparse autoencoders would be k-Sparse autoencoder (Makhzani and Frey, 2013), where sparsity is enforced by only keeping the top k highest activations in the bottleneck layer with linear activation function.

First we run forward through the encoder network to get the compressed code: $\mathbf{z} = g(\mathbf{x})$. Sort the values in the code vector $\mathbf{z}$. Only the k-largest values are kept while other neurons are set to 0. This can be done in a ReLU layer with an adjustable threshold too.

Once we have a sparsified code: $z = Sparsify(z)$, we can compute the output and the loss from the sparsified code, $L = \| x - f(z) \|_2^2$. And, then use back-propagation to train the model. NOTE: back-propagation only goes through the top k activated hidden units!



(a) $k = 70$

(b) $k = 40$

(c) $k = 25$

(d) $k = 10$

Figure 4: Filters of the k-sparse autoencoder for different sparsity levels k, learnt from MNIST with 1000 hidden units. (Image source: Makhzani and Frey, 2013)

3.4 Contractive Autoencoder

Similar to sparse autoencoder, Contractive Autoencoder (Rifai, et al, 2011) encourages the learned representation to stay in a contractive space for better robustness.
It adds a term in the loss function to penalize the representation being too sensitive to the input, and thus improve the robustness to small perturbations around the training data points. The sensitivity is measured by the Frobenius norm of the Jacobian matrix of the encoder activations with respect to the input:

$$\| J_f(X) \|_F^2 = \sum_{ij} \left(\frac{\partial h_j(X)}{\partial x_i} \right)^2$$

where h_j is one unit output in the compressed code $z = f(x)$.

This penalty term is the sum of squares of all partial derivatives of the learned encoding with respect to input dimensions. The authors claimed that empirically this penalty was found to carve a representation that corresponds to a lower-dimensional non-linear manifold, while staying more invariant to majority directions orthogonal to the manifold.

4 Probabilistic Autoencoders

In previous sections, we have seen different kinds of autoencoders, which take data as input and discover some latent state representation of that data. More specifically, our input data is converted into an encoding vector where each dimension represents some learned attribute about the data.

The most important detail to grasp here is that our encoder network is outputting a single value for each encoding dimension. The decoder network then subsequently takes these values and attempts to recreate the original input. This might not be sufficient (or good enough) for many real world examples and tasks. Thus, rather than building an encoder which outputs a single value to describe each latent state attribute, we should formulate our encoder to describe a probability distribution for each latent attribute.

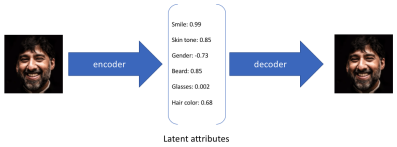


Figure 5: Simple autoencoder for encoding faces. Image source: Jeremy Jordan

Intuition: Let's say we've trained an autoencoder model on a large dataset of faces with a encoding dimension of 6. An ideal autoencoder will learn descriptive attributes of faces such as skin color, whether or not the person is wearing

glasses, etc. in an attempt to describe an observation in some compressed representation.

In the example in figure 5, we've described the input image in terms of its latent attributes using a single value to describe each attribute. However, we may prefer to represent each latent attribute as a range of possible values. For instance, what single value would you assign for the smile attribute if you feed in a photo of the Mona Lisa? Using a variational autoencoder, we can describe latent attributes in probabilistic terms.

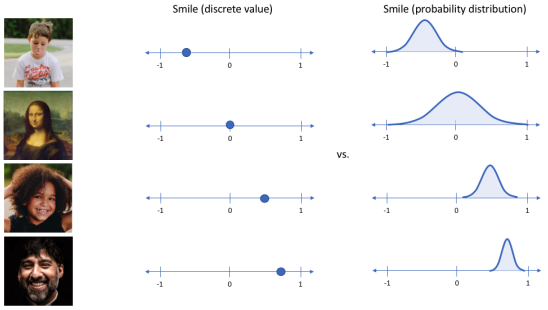


Figure 6: Representing smile in terms of discrete and continous values. Image source: Jeremy Jordan

With this approach, we'll now represent each latent attribute for a given input as a probability distribution. When decoding from the latent state, we'll randomly sample from each latent state distribution to generate a vector as input for our decoder model.

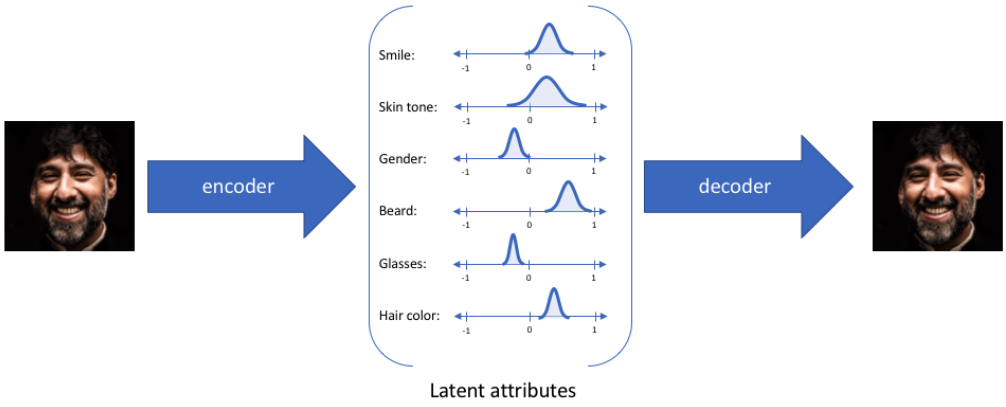


Figure 7: Probabilistic Autoencoder. Image source: Jeremy Jordan

By constructing our encoder model to output a range of possible values (a statistical distribution) from which we'll randomly sample to feed into our decoder model, we're essentially enforcing a continuous, smooth latent space representation. For any sampling of the latent distributions, 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.

4.1 VAE: Variational Autoencoder

A variational autoencoder (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.

The idea of Variational Autoencoder (Kingma Welling, 2014), short for VAE, is actually less similar to all the autoencoder models above, but deeply rooted in the methods of variational bayesian and graphical model. Instead of mapping the input $\mathbf{x}$ into a fixed vector $\mathbf{z}$, we want to map it into a distribution. Let's label this distribution as p_θ, parameterized by θ. The relationship between the data input $\mathbf{x}$ and the latent encoding vector $\mathbf{z}$ can be fully defined by:

1. Prior $p_\theta(\mathbf{z})$
2. Likelihood $p_\theta(\mathbf{x}|\mathbf{z})$
3. Posterior $p_\theta(\mathbf{z}|\mathbf{x})$

Assuming that we know the real parameter θ^* for this distribution, in order to generate a sample that looks like a real data point $\mathbf{x}^(\mathbf{i})$, we follow these steps:

1. First, sample a $\mathbf{z}^{(i)}$ from a prior distribution $p_\theta(\mathbf{z})$.
2. Then a value $\mathbf{x}^{(i)}$ is generated from a conditional distribution $p_\theta(\mathbf{x}|\mathbf{z} = \mathbf{z}^{(i)})$.

The optimal parameter θ^* is the one that maximizes the probability of generating real data samples:

$$\theta^* = \arg\max_\theta \prod_{i=1}^{n} p_\theta(\mathbf{x}^{(i)})$$

Commonly we use the log probabilities to convert the product on RHS to a sum:

$$\theta^* = \arg\max_\theta \sum_{i=1}^{n} p_\theta(\mathbf{x}^{(i)})$$

Now let's update the equation to better demonstrate the data generation process so as to involve the encoding vector:

$$p_\theta(\mathbf{x}^{(i)}) = \int p_\theta(\mathbf{x}^{(i)}|\mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$$

Unfortunately it is not easy to compute $p_\theta(\mathbf{x}^{(i)})$ in this way, as it is very expensive to check all the possible values of $\mathbf{z}$ and sum them up. To narrow down the value space to facilitate faster search, we would like to introduce a new approximation function to output what is a likely code given an input $\mathbf{x}$, $q_\phi(\mathbf{z}|\mathbf{x})$, parameterized by ϕ.
Now the structure looks a lot like an autoencoder:

1. The conditional probability $p_\theta(\mathbf{x}|\mathbf{z})$ defines a generative model, similar to the decoder $f_\theta(\mathbf{x}|\mathbf{z})$ introduced above.

2. The approximation function $q_\phi(\mathbf{z}|\mathbf{x})$ is the probabilistic encoder, playing a similar role as $g_\phi(\mathbf{z}|\mathbf{x})$ above.

4.1.1 Loss Function: ELBO

The estimated posterior $q_\phi(\mathbf{z}|\mathbf{x})$ should be very close to the real one $p_\theta(\mathbf{z}|\mathbf{x})$. We can use Kullback-Leibler divergence to quantify the distance between these two distributions. KL divergence $D_{KL}(X||Y)$ measures how much information is lost if the distribution Y is used to represent X.
In our case we want to minimize $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}|\mathbf{x}))$ with respect to ϕ.
But why use $D_{KL}(q_\phi \parallel p_\theta)$ (reversed KL) instead of $D_{KL}(p_\theta \parallel q_\phi)$ (forward KL)?

- Forward KL divergence: $D_{KL}(P \parallel Q) = E_{z\sim P(z)} log \frac{P(z)}{Q(z)}$ we have to ensure that $Q(z) > 0$ wherever $P(z) > 0$. The optimized variational distribution $q(z)$ has to cover over the entire $p(z)$.

- Reversed KL divergence: $D_{KL}(Q \parallel P) = E_{z\sim Q(z)} log \frac{Q(z)}{P(z)}$; minimizing the reversed KL divergence squeezes the $Q(z)$ under $P(z)$.

Let's now expand the equation, $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}|\mathbf{x}))$

$$
\begin{aligned}
D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}|\mathbf{x})) &= \int q_\phi(\mathbf{z}|\mathbf{x}) log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) log \frac{q_\phi(\mathbf{z}|\mathbf{x}) p_\theta(\mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} d\mathbf{z} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \left(log p_\theta(\mathbf{x}) + log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} \right) d\mathbf{z} \\
&= log p_\theta(\mathbf{x}) + \int q_\phi(\mathbf{z}|\mathbf{x}) log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} d\mathbf{z} \\
&= log p_\theta(\mathbf{x}) + \int q_\phi(\mathbf{z}|\mathbf{x}) log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z})} d\mathbf{z} \\
&= log p_\theta(\mathbf{x}) + E_{z\sim q_\phi(\mathbf{z}|\mathbf{x})} \left[log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}} - log p_\theta(\mathbf{x}|\mathbf{z}) \right] \\
&= log p_\theta(\mathbf{x}) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}) - E_{z\sim q_\phi(\mathbf{z}|\mathbf{x})} log p_\theta(\mathbf{x}|\mathbf{z})
\end{aligned}
$$

so, we have:

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}|\mathbf{x})) = logp_\theta(\mathbf{x}) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}) - E_{z\sim q_\phi(\mathbf{z}|\mathbf{x})} logp_\theta(\mathbf{x}|\mathbf{z})$$

Once we rearrange the left and right hand side of the equation,

$$logp_\theta(\mathbf{x}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}|\mathbf{x})) = E_{z\sim q_\phi(\mathbf{z}|\mathbf{x})} logp_\theta(\mathbf{x}|\mathbf{z}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z})$$

The LHS of the equation is exactly what we want to maximize when learning the true distributions: we want to maximize the (log-)likelihood of generating real data (i.e. $logp_\theta(\mathbf{x})$) and also minimize the difference between the real and estimated posterior distributions (the term D_{KL} works like a regularizer). NOTE that $p_\theta(\mathbf{x})$ is fixed with respect to q_ϕ.

The negation of the above defines our loss function:

$$L_{VAE}(\theta, \phi) = -logp_\theta(\mathbf{x}) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}|\mathbf{x}))$$

$$= -E_{z\sim q_\phi(\mathbf{z}|\mathbf{x})} logp_\theta(\mathbf{x}|\mathbf{z}) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z})$$

$$\theta^*, \phi^* = \arg\max_{\theta,\phi} L_{VAE}$$

In Variational Bayesian methods, this loss function is known as the variational lower bound, or evidence lower bound. The "lower bound" part in the name comes from the fact that KL divergence is always non-negative and thus L_{VAE} is the lower bound of $logp_\theta(\mathbf{x})$.

$$-L_{VAE} = logp(\mathbf{x}) D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) p_\phi(\mathbf{z}|\mathbf{x}) \leq logp_\theta(\mathbf{x})$$

Therefore by minimizing the loss, we are maximizing the lower bound of the probability of generating real data samples.

4.1.2 Reparameterization Trick

The expectation term in the loss function invokes generating samples from $z \sim q_\phi(\mathbf{z}|\mathbf{x})$. Sampling is a stochastic process and therefore we cannot backpropagate the gradient. To make it trainable, the reparameterization trick is introduced: It is often possible to express the random variable $\mathbf{z}$ as a deterministic variable $\mathbf{z} = \Gamma_\phi(\mathbf{x}, \epsilon)$, here ϵ is an auxiliary independent random variable, and the transformation function Γ_ϕ is parameterized by ϕ, which converts ϵ to z.

For example, a common choice of the form of $q_\phi(\mathbf{z}|\mathbf{x})$ is a multivariate Gaussian with a diagonal covariance structure:

$$\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x}^{(\mathbf{i})}) = \mathcal{N}(\mathbf{z}; \mu^{(i)}, \sigma^{2(i)} I)$$

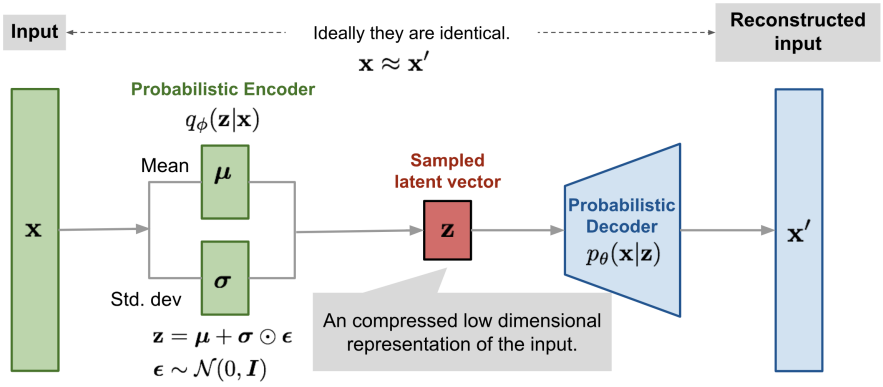


Figure 8: An variational autoencoder model with the multivariate Gaussian. Image source: Lilian Weng, 2018

$$\mathbf{z} = \mu + \sigma \odot \epsilon \text{ w,here, } \epsilon \sim \mathcal{N}(0, I)$$

where $\odot$ refers to element-wise product.

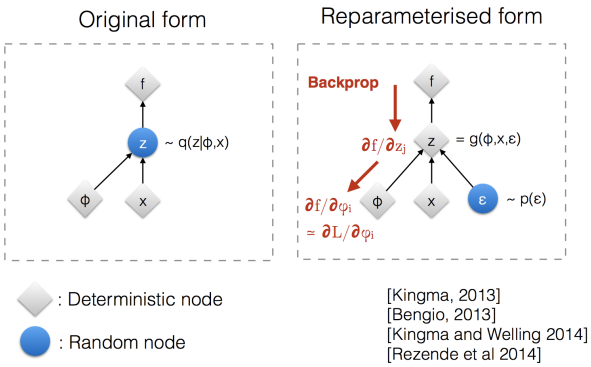


Figure 9: Illustration of how the reparameterization trick makes the z sampling process trainable.(Image source: Slide 12 in Kingma's NIPS 2015 workshop talk)

4.2 Conditional Variational Autoencoder (CVAE)

Conditional Variational Autoencoder (CVAE) is an extension of Variational Autoencoder (VAE). We've seen that by formulating the problem of data generation as a bayesian model, we could optimize its variational lower bound to learn the model.

However, we have no control on the data generation process on VAE. This could be problematic if we want to generate some specific data. As an example, suppose we want to convert a unicode character to handwriting. In vanilla

VAE, there is no way to generate the handwriting based on the character that the user inputted. Concretely, suppose the user inputted character '2', how do we generate handwriting image that is a character '2'? We couldn't. Hence, CVAE was developed.

VAE essentially models latent variables and data directly, whereas CVAE models latent variables and data, both conditioned to some random variables. Recall in VAE, the objective is,

$$logp_\theta(\mathbf{x}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z}|\mathbf{x})) = E_{z\sim q_\phi(\mathbf{z}|\mathbf{x})} logp_\theta(\mathbf{x}|\mathbf{z}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z})$$

i.e., we want to optimize the log likelihood of our data $P(\mathbf{x})$ under some "encoding" error.

The original VAE model has two parts: the encoder $q_\phi(\mathbf{z}|\mathbf{x})$ and the decoder $p_\theta(\mathbf{z}|\mathbf{x})$. Looking closely at the model, we could see why can't VAE generate specific data, as per our example above. It's because the encoder models the latent variable $\mathbf{z}$ directly based on $\mathbf{x}$, it doesn't care about the different type of $\mathbf{x}$. For example, it doesn't take any account on the label of $\mathbf{x}$. Similarly, in the decoder part, it only models $\mathbf{x}$ directly based on the latent variable $\mathbf{z}$.

We could improve VAE by conditioning the encoder and decoder to another thing(s). Let's say that other thing is c, so the encoder is now conditioned to two variables x and c: $q_\phi(\mathbf{z}|\mathbf{x}, c)$. The same with the decoder, it's now conditioned to two variables z and c: $p_\theta(\mathbf{x}|\mathbf{z}, c)$.

Hence, our variational lower bound objective is now in this following form:

$$logp_\theta(\mathbf{x}|c) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}, c) \parallel p_\theta(\mathbf{z}|\mathbf{x}, c)) = E_{z\sim q_\phi(\mathbf{z}|\mathbf{x},c)} logp_\theta(\mathbf{x}|\mathbf{z}, c) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}, c) \parallel p_\theta(\mathbf{z}|c)$$

i.e. we just conditioned all of the distributions with a variable c.
Now, the real latent variable is distributed under $p_\theta(\mathbf{z}|c)$. That is, it's now a *conditional probability distribution*. Think about it like this: for each possible value of c, we would have a $p_\theta(\mathbf{z})$. We could also use this form of thinking for the decoder.

4.3 beta-VAE

If each variable in the inferred latent representation $\mathbf{z}$ is only sensitive to one single generative factor and relatively invariant to other factors, we will say this representation is disentangled or factorized. One benefit that often comes with disentangled representation is *good interpretability* and easy generalization to a variety of tasks.

For example, a model trained on photos of human faces might capture the gentle, skin color, hair color, hair length, emotion, whether wearing a pair of glasses and many other relatively independent factors in separate dimensions. Such a disentangled representation is very beneficial to facial image generation.

β-VAE (Higgins et al., 2017) is a modification of Variational Autoencoder with a special emphasis to discover disentangled latent factors. Following the same incentive in VAE, we want to maximize the probability of generating real

data, while keeping the distance between the real and estimated posterior distributions small (say, under a small constant δ):

$$\max_{\theta,\phi} E_{\mathbf{x}\sim D}[E_{\mathbf{x}\sim q_\phi(\mathbf{z}|\mathbf{x})} log p_\theta(\mathbf{x}|\mathbf{z})]$$

$$\text{subject to} D_{KL}(q_\phi(\mathbf{z}|\mathbf{x} \parallel p_\theta(\mathbf{z})) < \delta$$

We can rewrite it as a Lagrangian with a Lagrangian multiplier β under the KKT condition. The above optimization problem with only one inequality constraint is equivalent to maximizing the following equation $F(\theta, \phi, \beta)$:

$$F(\theta, \phi, \beta) = E_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})} log p_\theta(\mathbf{x}|\mathbf{z}) - \beta(D_{KL} q_\phi(\mathbf{z}|\mathbf{x} \parallel p_\theta(\mathbf{z})) - \delta)$$

$$F(\theta, \phi, \beta) = E_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})} log p_\theta(\mathbf{x}|\mathbf{z}) - \beta D_{KL} q_\phi(\mathbf{z}|\mathbf{x} \parallel p_\theta(\mathbf{z})) + \beta\delta$$

$$F(\theta, \phi, \beta) \geq E_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})} log p_\theta(\mathbf{x}|\mathbf{z}) - \beta D_{KL} q_\phi(\mathbf{z}|\mathbf{x} \parallel p_\theta(\mathbf{z})$$

The loss function of β-VAE is defined as:

$$L_{BETA}(\phi, \beta) = -E_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})} log p_\theta(\mathbf{x}|\mathbf{z}) + \beta D_{KL} q_\phi(\mathbf{z}|\mathbf{x} \parallel p_\theta(\mathbf{z})$$

where the Lagrangian multiplier β is considered as a hyperparameter.

Since the negation of $L_{BETA}(\phi, \beta)$ is the lower bound of the Lagrangian $F(\phi, \beta, \theta)$, minimizing the loss is equivalent to maximizing the Lagrangian and thus works for our initial optimization problem.

When β=1, it is same as VAE. When β >1, it applies a stronger constraint on the latent bottleneck and limits the representation capacity of $\mathbf{z}$. For some conditionally independent generative factors, keeping them disentangled is the most efficient representation. Therefore a higher β encourages more efficient latent encoding and further encourages the disentanglement. Meanwhile, a higher β may create a trade-off between reconstruction quality and the extent of disentanglement.

References

1. Geoffrey E. Hinton, and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." Science 313.5786 (2006): 504-507.

2. Pascal Vincent, et al. "Extracting and composing robust features with denoising autoencoders." ICML, 2008.

3. Pascal Vincent, et al. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.". Journal of machine learning research 11.Dec (2010): 3371-3408.

4. Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors." arXiv preprint arXiv:1207.0580 (2012).

5. Sparse Autoencoder by Andrew Ng.

6. Alireza Makhzani, Brendan Frey (2013). "k-sparse autoencoder". ICLR 2014.

7. Salah Rifai, et al. "Contractive auto-encoders: Explicit invariance during feature extraction." ICML, 2011.

8. Diederik P. Kingma, and Max Welling. "Auto-encoding variational bayes." ICLR 2014.

9. Tutorial - What is a variational autoencoder? on jaan.io

10. Youtube tutorial: Variational Autoencoders by Arxiv Insights

11. "A Beginner's Guide to Variational Methods: Mean-Field Approximation" by Eric Jang.

12. Carl Doersch. "Tutorial on variational autoencoders." arXiv:1606.05908, 2016.

13. Irina Higgins, et al. "-VAE: Learning basic visual concepts with a constrained variational framework." ICLR 2017.

14. Christopher P. Burgess, et al. "Understanding disentangling in beta-VAE." NIPS 2017.

15. From Autoencoder to Beta-VAE, Lilian Weng, 2018.

16. Variational autoencoders, Jeremy Jordan, 2018.