Formulation of the Variational Autoencoder and Evidence Lower Bound and an Application to the MNIST Dataset

Eric M. Fischer
University of California Los Angeles
Los Angeles, CA 90095
emfischer712@ucla.edu
303 759 361

Abstract

*We will provide a rigorous statistical formulation for the Variational Autoencoder (**VAE**), which includes a derivation for the variational lower bound, or evidence lower bound (ELBO). The variational lower bound is central to the algorithm and enables optimization of a tractable objective function, the Stochastic Gradient Variational Bayes (SGVB) estimator, specifically created for the VAE. We will then apply the variational autoencoder to the MNIST database of handwritten digits to perform encoding, decoding, and display some generated data samples.*

1. Introduction

An underlying motivation for this study was to explore the intersection of Bayesian inference and neural networks. Before the resurgence of neural networks due to advances in computing power, Bayesian inference due to its high computational cost was applied mostly to small datasets for extracting information and performing regularization [13]. Now Bayesian inference and neural networks are commonly leveraged together in Bayesian neural networks to tackle demanding computational problems. As the VAE employs a Bayesian neural network, let us cover some associated terminology.

First, a Bayesian network (Bayes network, belief network, decision network, Bayes(ian) model, or probabilistic directed acyclic graphical model) is a probabilistic graphical model that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG). The nodes represent variables in the Bayesian sense: they may be observable quantities, latent variables, unknown parameters, or hypotheses. Edges represent conditional dependencies, and accordingly nodes without a connecting edge are conditionally independent. Each node has an associated probability function that accepts as input a set of values for the node's parent variables and gives as output the probability, or probability distribution, of the variable represented by the node [3] [19]. So a Bayesian network is a graphical model that encodes probabilistic relationships among variables of interest, taking prior knowledge and data and estimating the posterior probabilities of outcomes. Predictions are made by marginalizing over distributions of parameters.

This is in contrast to an artificial neural network, which uses maximum likelihood estimation (MLE) to determine network parameters (weights and biases) and hence make predictions. Neural networks do not have any such structure like the Bayesian network that gives valuable information about the conditional dependence between variables [10]. A prominent advantage of the neural network, however, is that it can appropriately handle any correlation or dependence between input variables. Bayesian networks like Naive Bayes assume that all input variables are independent, which if untrue will negatively impact the performance of the network. By modeling probability distributions, however, the Bayesian network is more robust and is regularly utilized for inference, modeling, and prediction, whereas the neural network is mostly used for prediction [18] [17].

The VAE as mentioned uses a Bayesian neural network, which extends a standard artificial neural network with posterior inference. A Bayesian neural network, by placing a prior distribution on its weights, is able to capture a measure of uncertainty in predictions that a neural network cannot. From a probabilistic perspective, this is crucial. The MLE that neural networks perform to determine weight values ignores any uncertainty in e.g. the weight values or the even the choice of MLE as an optimization method [14].

Also, neural networks are well-known to be susceptible to overfitting due to no quantification of uncertainty, and thus they employ techniques such as L2 regularization (which from a Bayesian perspective is equivalent to induc-

ing prior distributions on the weights). Neural network optimization in this case is akin to searching for maximum *a posteriori* estimators rather than MLE. Although this works well in practice, from a probabilistic perspective it only alleviates the overfitting issue but does not solve it [11] [4]. Hence, we have a motivation for Bayesian neural networks: to lend a probabilistic interpretation of the neural network with which we can perform posterior inference.

2. Variational Autoencoder

Although we will introduce the variational autoencoder with a multivariate normal distribution to expound the full color of the model, in this study a binomial distribution was used to model the MNIST digit data.

Broadly speaking, autoencoders and variational autoencoders provide ways to perform unsupervised learning with neural networks. Other ways to perform unsupervised learning include both linear techniques such as Principal Component Analysis (PCA) and Factor Analysis (FA) and non-linear techniques such as Locally Linear Embedding (LLE), and T-distributed Stochastic Neighbor Embedding (tSNE).

Autoencoders and variational autoencoders produce lower-dimensional representations of input data. Given an input $x \in \mathbb{R}^n$, they will produce $h \in \mathbb{R}^d$ where $d < n$. The key difference is that the VAE is probabilistic and a generative model. This means we can obtain generated samples $x \in \mathbb{R}^n$ by sampling from its distribution [8] [6]. VAEs belong to a class of generative models that can generate examples of data by learning statistics about it.

2.1. Autoencoder

An autoencoder is a type of artificial neural network that learns efficient data codings, i.e. performs feature detection, in an unsupervised manner. It learns an encoding, or representation, for a set of data usually for the purposes of dimensionality reduction. In addition to the this reduction, the autoencoder has a reconstruction side, in which it generates from a reduced encoding a representation as similar as possible to the original to the original input [2]. The autoencoder has become very popular in learning generative models of data, although Generative Adversarial Networks (GANs) briefly discussed later have shown more success with data generation in particular [15].

Hence the basic structure of an autoencoder involves an encoder and a decoder. The encoder performs a dimensionality reduction step on the data $x \in \mathbb{R}$ to obtain features $h \in \mathbb{R}^d$. The decoder maps the features $h \in \mathbb{R}^d$ to closely reproduce the input, producing $\hat{x} \in \mathbb{R}^n$. The autoencoder solves the following problem:

Let $x \in \mathbb{R}^n$, $f(\cdot) : \mathbb{R}^n \to \mathbb{R}^d$ and $g(\cdot) : \mathbb{R}^d \to \mathbb{R}^n$.

$$\hat{x} = g(f(x))$$

We define a loss function $\mathcal{L}(x, \hat{x})$ and minimize $\mathcal{L}$ with respect to the parameters of $f(\cdot)$ and $g(\cdot)$. We could use various loss functions, but a common one is the squared loss:

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

The functions $f()$ and $g()$ represent deep neural networks and the encoder and decoder, respectively [8].

3. VAE Description

Variational autoencoders, introduced by Kingma and Welling in 2014, do not learn $f()$ and $g()$ directly but rather probabilistic versions of $f()$ and $g()$. That is, they learn distributions of the features (or activations) z given the input x and the input x given the features z. Concretely, the VAE learns:

$p(z|x)$: distribution of the features given the input
$p(x|z)$: distribution of the input given the features

These distributions can express complex, nonlinear transformations as they are parameterized by neural networks [8], This also means we can train them by stochastic gradient descent or various other first-order optimization methods that have demonstrated success in neural networks such as Adam, RMSprop, and stochastic gradient descent with Nesterov momentum [7].

There are various benefits to learning distributions like the VAE does. If the data is noisy, a model of the distribution of the data can be more useful for a given problem. Also, the relationship between observed and latent variables is often nonlinear, in which case the VAE provides a way to do inference. And as mentioned previously, because the VAE model learns $p(x|z)$ and accordingly samples z before subsequently x, it is a generative model that can generate data with similar statistics to the input [8].

The following is a conceptual diagram of the VAE.

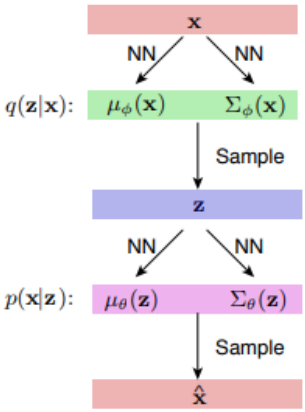


Figure 1. Conceptual diagram of the VAE [8]

Note we do not infer activations z directly from the input x and infer the (encoded and decoded) output $\hat{x}$ directly from z, as one does with a standard autoencoder. With the VAE, instead of performing the inference, or learning, $z = f(x)$, we learn the distribution $q(z|x)$. Similarly, instead of learning $\hat{x} = g(z)$, we learn the distribution $p(x|z)$.

Accordingly, to obtain values for z and $\hat{x}$, we sample from the learned distributions. We can see how VAE is a probabilistic variant of the autoencoder [8].

4. VAE Formulation

We will formulate the VAE from the context of a generative model. If we would like to generate samples from a distribution $p(x)$, instead of $p(x)$ directly, we can use a latent variable model.

Latent variable models model the data x as coming from an unobserved, or latent, variable z. In the event it is difficult to model $p(x)$ directly, one can choose a distribution $p(z)$ and model $p(x|z)$. The conditional distribution $p(x|z)$ can be defined as some mapping $x = g(z)$ that is determined by the prior distribution $p(z)$ and the function $g()$ [5].

Once we know $p(z)$ and $p(x|z)$, we can generate samples from $p(x)$ as follows:

1. Generate a random sample from $z_s \sim p(z)$
2. Generate a sample $x_s \sim p(x|z = z_s)$

Crucially, the sample of x_s in Step 2 is indeed from $p(x)$. This is because the sample has probability $p(x, z_s)$, so by repeatedly sampling (i.e. drawing a sample for each of the sampled z_s variables), the resulting probability density of the sampled x_s variables in Step 2 is:

$$p(x) \approx \frac{1}{n} \sum_{i=1}^{n} p(x, z_s^{(i)})$$

which approximates the distribution of x [8].

As a note on designing the prior distribution $p(z)$, in almost all cases we can set $z \sim \mathcal{N}(0, I)$ and consider it a non-stringent constraint on the distribution of the latent variables. This is because, as long as $x|z$ is expressive enough (e.g. it has enough nonlinear transformations), z can be a simple distribution.

In general with the VAE, we can design both $p(z)$ and $p(x|z)$ to optimize the generative process for $p(x)$. And by keeping $p(z)$ simple, we can concentrate on optimizing $x|z$ to generate the distribution $p(x)$.

So we let the expressive capacity of the nonlinear transformation $x = g(z)$ find a complex fit to the data $p(x)$.

To summarize, a variational autoencoder models the distribution $p(x)$ not directly but through a latent variable, so that modeling the complex distribution is reformulated as a nonlinear transformation of a simpler latent variable z [8].

4.1. Modeling $p(x)$

To find the parameters θ of the model, we could follow a standard process with neural networks. We define a loss function, calculate the gradients of it with respect to the paremeters θ, and apply a version of stochastic gradient descent to minimize the loss function given the gradient. The most intuitive function to optimize is the likelihood function. We can maximize the likelihood of observing the data:

$$L(\theta|x) = \prod_{i=1}^{m} p_\theta(x^{(i)})$$

For one example x, maximizing the likelihood and finding $p_\theta(x)$ would thus entail:

$$\begin{aligned} p_\theta(x) &= \int p_\theta(x, z) dz \\ &= \int p_\theta(x|z) p(z) dz \end{aligned}$$

But with the VAE, as it is a neural network x is a nonlinear function of z, so the distribution $p_\theta(x, z)$ cannot be written analytically and hence $p_\theta(x)$ is intractable. So we cannot use an approach like the expectation maximization algorithm to optimize the parameters θ, which is a helpful way to evaluate the integral when e.g. $x|z$ and x are normal distributions and x is a linear function of z.

Another approach to obtain samples from $p_\theta(x)$ to find the parameters θ might be Monte Carlo sampling. Namely, we could effectively sample from the distribution $p(x)$ by sampling z and then calculating $x|z$. But this is only tractable when x is relatively low-dimensional and this is often not the case. If x were a high-dimensional 32x32x3 pixel image, for example, the curse of dimensionality makes it such that we need to obtain a very large number of samples to get an accurate representation of x.

So we eliminate approaches like the expectation maximization algorithm or Monte Carlo sampling to calculate $p(x)$ for the maximum likelihood objective. Instead, we will arrive at a different objective function for VAEs: the evidence lower bound (ELBO), or variational lower bound.

Before doing so, note that our model for $p_\theta(x)$ is a mixture of infinitely many Gaussians. This is more powerful than a Gaussian mixture model (GMM) that is considered too restrictive and other approaches one might consider for modeling $p_\theta(x)$. With infinitely many Gaussians, each image sample x has a corresponding latent variable z and conditional Guassian distribution $p_\theta(x|z)$. Note that even if the Gaussians are factorized, i.e. have independent components for each dimension, the mixture is not [1].

4.2. Arriving at the Variational Lower Bound

Using the evidence lower bound (ELBO) as our objective function allows us to perform variational inference in the VAE. The idea is that, as we cannot write $\prod_i p(x^{(i)})$, we can instead derive a lower bound for it. Then assuming that that lower bound is tractable, we can optimize the parameters θ with respect to it. Hence the goal is to maximize the lower bound to (hopefully) increase the likelihood. Note the similarity here with expectation maximization [8].

To arrive at ELBO, first recall our model for $p_\theta(x)$, which is intractable as it has an intractable integral:

$$p_\theta(x) = \int p_\theta(x|z)p(z)dz$$

The first step to overcoming this issue and deriving ELBO is noting that $p(x,z) = p(z|x)p(x) = p(x|z)p(z)$, so we can also express $p(x)$ as:

$$p(x) = \frac{p(x|z)p(z)}{p(z|x)}$$

Now we have another formulation for $p(x)$, but the distribution $p(z|x)$ in the denominator is also intractable. To overcome this, we can approximate $p(z|x)$ with another distribution we introduce, $q(z|x)$ [8].

The distribution $q(z|x)$ takes the form of a nonlinear neural network with normally distributed statistics, just as the distribution $x|z$ does. Concretely,

$$q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \Sigma_\phi(x)$$
$$p_\theta(x|z) = \mathcal{N}(\mu_\theta(z), \Sigma_\theta(z)$$

If we then include a penalty term in the objective function for the divergence between $p(z|x)$ and $q(z|x)$, we can make it such that $q(z|x)$ is as close to $p(z|x)$ as possible. We can measure the divergence with Kullback-Leibler (KL) divergence, which is the expectation of the logarithmic difference between two distributions. The KL divergence of $q(z)$ and $p(z)$ is defined as follows:

$$\text{KL}(q||p) = -\sum_z q(z) log \frac{p(z)}{q(z)}$$

As we next define ELBO, let us summarize the architecture of the VAE at this stage.

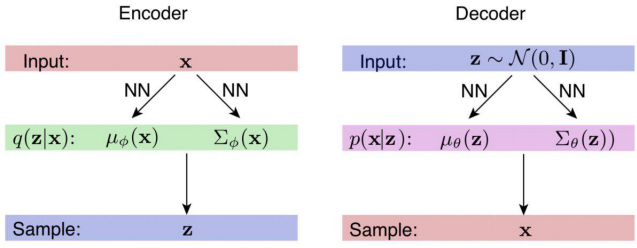


Figure 2. VAE before ELBO

Thus the encoder network samples latent variables z with the following:

1. Accept an input x
2. Calculate $q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \Sigma_\phi(x))$
3. Sample $z \sim q(z|x)$

And the decoder network samples latent variables x with the following [8]:

1. Accept an sampled latent variable $z \sim \mathcal{N}(0, I)$
2. Calculate $p_\theta(x|z) = \mathcal{N}(\mu_\theta(z), \Sigma_\theta(z)$
3. Sample $x \sim p(x|z)$

Note that the parameters ϕ belongs to the encoder and the parameters θ to the decoder. As such, the VAE paper refers to the parameters ϕ as variational parameters and the parameters θ as generative parameters [9].

4.3. Variational Lower Bound

The first step in deriving the evidence lower bound is observing that we can state the following for one sample $x^{(i)}$, which for simplicity we refer to as x here and for the rest of the proof:

$$\log p_\theta(x) = \mathbb{E}_{z\sim q(z|x)} \log p_\theta(x)$$

This is intuitive once digested, as the term $\log p_\theta(x)$ has no relation to the expectation of z, i.e. it is independent of z. Also by the linearity of expectation, starting with the right-hand side (RHS) of the equality, we can see:

$$\begin{aligned}\mathbb{E}_{z\sim q(z|x)} \log p_\theta(x) &= \log p_\theta(x) \mathbb{E}_{z\sim q(z|x)} 1 \\ &= \log p_\theta(x)\end{aligned}$$

For the rest of the proof, we denote this introduced term $\mathbb{E}_{z\sim q(z|x)}$ as $\mathbb{E}_z$:

$$\begin{aligned}\log p_\theta(x) &= \mathbb{E}_z \log p_\theta(x) \\ &\overset{(1)}{=} \mathbb{E}_z \log \frac{p(x|z)p(z)}{p(z|x)} \\ &= \mathbb{E}_z \log \left(\frac{p(x|z)p(z)}{p(z|x)} \frac{q(z|x)}{q(z|x)}\right) \\ &\overset{(2)}{=} \mathbb{E}_z \log p(x|z) - \mathbb{E}_z \log \frac{q(z|x)}{p(z)} + \\ &\quad \mathbb{E}_z \log \frac{q(z|x)}{p(z|x)} \\ &\overset{(3)}{=} \mathbb{E}_z \log p(x|z) - \text{KL}(q(z|x)||p(z)) + \\ &\quad \text{KL}(q(z|x)||p(z|x)) \\ &\geq \mathbb{E}_z \log p(x|z) - \text{KL}(q(z|x)||p(z))\end{aligned}$$

The critical steps of the derivation are labeled (1), (2), and (3). For (1), we use the chain rule for probability. For

(2), we expand the logarithm and use the linearity of expectation. For (3), we use the definition of KL divergence. And the final inequality can be drawn due to the KL divergence property $\mathrm{KL}(q||p) \geq 0$ for any q, p [8].

Note as well that a KL divergence term $\mathrm{KL}q(z|x)||p(z|x)$ has been dropped. This term expresses the divergence between the approximation $q(z|x)$ and the distribution it seeks to approximate, $p(z|x)$. Hence it can quantify the penalty in estimating the log-likelihood of the data by using $q(z|x)$ instead of $p(z|x)$. In conclusion, the lower bound on the log-likelihood of the data is:

$$\begin{aligned}\log p_\theta(x) &\geq \mathbb{E}_z \log p(x|z) - \mathrm{KL}(q(z|x)||p(z)) \\ &= \mathcal{L}_{vae}(x)\end{aligned}$$

$\mathcal{L}_{vae}(x)$ is called the evidence lower bound (ELBO) or variational lower bound.

Importantly, we now have a loss function, namely the lower bound $\mathcal{L}_{vae}(x)$, which is tractable (the first and second terms can be computed). The first term $\mathbb{E}_z \log p_{(x|z)}$ can be easily approximated with batches of samples during learning. Namely, we calculate $q(z|x^{(i)})$, sample z from this distribution, and subsequently calculate $\log p(x^{(i)}|z)$. These 3 steps are diagrammed in the computational graph below.

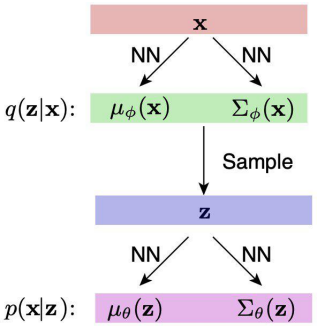


Figure 3. ELBO calculation for $\log p(x|z)$

The second term, the KL divergence, is a simple analytic calculation using the parameters u_ϕ, u_θ, Σ_ϕ, Σ_θ. When the probability distributions $q()$ and $p()$ are Guassian, it can be calculated as follows [8]:

$$\mathrm{KL}(\mathcal{N}(\mu_0, \Sigma_0), \mathcal{N}(\mu_1, \Sigma_1) =$$

$$\frac{1}{2}\left[tr(\Sigma_1^{-1}\Sigma_0) + (\mu_1 - \mu_0)^T \Sigma_1^{-1}(\mu_1 - \mu_0) - d + \log \frac{\det \Sigma_1}{\det \Sigma_0}\right]$$

The derivation of the evidence lower bound (ELBO) is central to the VAE algorithm. In general, it allows the VAE to use a variational approach for latent representation learning (*variational* Bayesian methods approximate intractable integrals in Bayesian inference), and it leads to a specific estimator for the VAE training algorithm called the Stochastic Gradient Variational Bayes (SGVB) estimator, which we introduce after the necessary reparameterization step.

4.4. Reparameterization Technique

For the first term $\mathbb{E}_z \log p(x|z)$ of the ELBO, or variational lower bound, that involves sampling $z \sim q(z|x^{(i)})$, the VAE reparameterization technique must be employed. This is because neural networks cannot perform backpropagation (necessary to perform weight updates) through sampling procedures like this sampling of z. There must be another way to calculate z that will allow backpropagation to be performed.

The solution is to sample $\epsilon \sim \mathcal{N}(0, \mathrm{I})$ and then calculate analytically:

$$z = u_\phi(x^{(i)}) + \Sigma_\phi^{\frac{1}{2}}(x^{(i)})\epsilon$$

As here z is just a linear transformation of ϵ with mean $u_\phi(x^{(i)})$ and covariance $\Sigma_\phi(x^{(i)})$, it indeed constitutes a sample from $q(z|x^{(i)})$ and now the sampling can take place for ϵ as opposed to z. This solves the issue of not being able to backpropagate, as the neural network does not need to backpropagate through ϵ, contrary to z [8].

Now that the VAE model is reparameterized to sample through the parameter ϵ instead of z, the ELBO variational lower bound can be optimized.

4.5. Final VAE Model

The final computational graph to calculate the ELBO for the VAE is below.

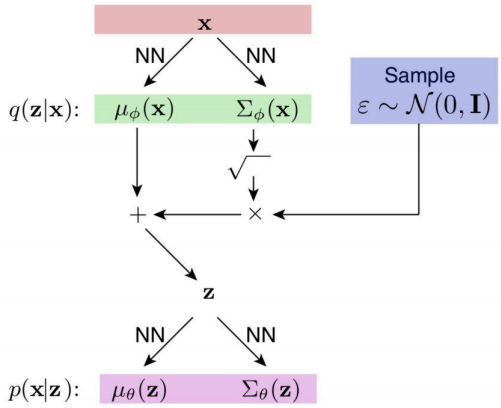


Figure 4. VAE after ELBO

As we backpropagate through the parameters θ and ϕ in optimizing the ELBO, we obtain two gradients:

$$\nabla_\theta \mathcal{L}_{vae} \text{ and } \nabla_\phi \mathcal{L}_{vae}$$

With these gradients, we can use stochastic gradient descent or other optimization methods to optimize the parameters ϕ

and θ of the encoder and decoder neural networks, respectively.

In conclusion, we have the Stochastic Gradient Variational Bayes (SGVB) estimator which we can optimize for the variational autoencoder:

$$\mathcal{L}_{vae} = \mathbb{E}_z \log p(x|z) - \mathrm{KL}(q(z|x)||p(z))$$

Now that we have derived the VAE and its ELBO objective function, we can observe some brief results of the VAE applied to the MNIST dataset. Specifically, after briefly introducing the dataset, we observe the results of the decoding, encoding, and data generation processes.

5. MNIST Dataset

The MNIST database of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image. The dataset is optimal for learning pattern recognition methods on real-world data while expending minimal effort on preprocessing [12].

Figure 5. A subset of the MNIST database of handwritten digits

The original black and white (bilevel, not grayscale) images from NIST were size-normalized to fit in a 20x20 pixel box while preserving their aspect ratio.

The images were centered in a 28x28 pixel image by computing the center of mass of the pixels for each image and then translating the image to position this calculated point at the center of a 28x28 pixel field.

The handwritten digits were obtained from a combination of U.S. Census Bureau employees and high-school students, the samples from high-school students providing more variation [12].

6. VAE on the MNIST Dataset

A diagram of the variational autoencoder architecture applied to the MNIST dataset is provided below.

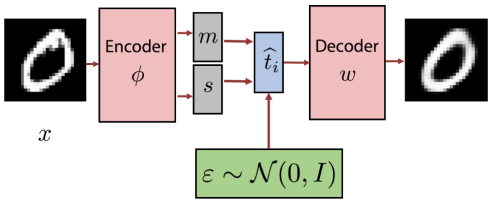


Figure 6. Variational autoencoder

Note how the sampling takes place for ϵ and not z, which was the goal of the reparameterization technique in order to be able to optimize the ELBO variational lower bound.

7. Results

Now we can review the results of the variational autoencoder decoding, encoding, and data generation processes.

7.1. VAE Encoding and Decoding

This is an example of an encoded digit, the result after passing an input image x through the encoding process to undergo dimensionality reduction.

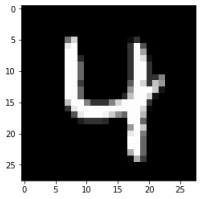


Figure 7. VAE encoding output

And we display the same image after passing it through the decoder, which produces an output image using the encoding from the encoder. The aim of the decoder, if not performing the related task of data generation, is to output an image that looks as similar as possible to the provided input image x.

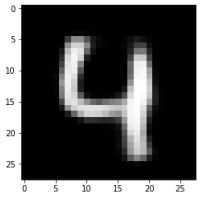


Figure 8. VAE decoding output

Lastly, below is a visualization of some MNIST images before and after being passed to the variational autoencoder for dimensionality reduction.

Figure 9. Images before and after autoencoding

In the left column are input images x and in the right column the encoded and subsequently decoded output images.

7.2. VAE Data Generation

We can also observe some generated images by the VAE.

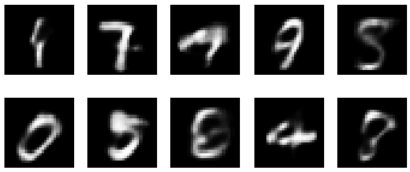


Figure 10. VAE Generated Data

These images were produced without any corresponding input image x.

8. Related Work

There exist many types of generative models in the literature, but a popular one with several key advantages is the generative adversarial network (**GAN**). Namely, the GAN generator has few restrictions in that it could easily be made into a convolutional neural network (**CNN**) or Long Short Term Memory (**LSTM**) network, it also does not require a variational lower bound, and GANs are asymptotically consistent. Additionally, as mentioned previously, GANs are well-known for generating more realistic data samples [8]. (Interestingly, there still does not exist a proof that VAEs are asymptotically consistent.)

A key difference between the VAE and the GAN is with respect to how they generate data. We saw that with the VAE, we had to learn some probability $p_{model}(x)$ to generate data. In a GAN, which can be seen as a game between a generator (generative model) and discriminator (generator's opponent), the discriminator learns an approximation of the ratio $p_{data}(x)/p_{model}(x)$. This is the central idea of the GAN and makes it distinctly different than other generative models that learn $p_{model}(x)$ directly, or indirectly via latent variable models. By learning this ratio, the discriminator can better judge the quality of samples from the generator, i.e. whether they are real or generated images [8].

Other than the variational autoencoder, extended autoencoders in the literature include sparse autoencoders, denoising autoencoders, and contractive autoencoders, amongst others.

9. Future Work

Most prominently, we would like to extend this project to work with an infinite mixture of Gaussian distributions, in addition to the mixture of binomial distributions used for the MNIST data in this paper. This would allow us to reperform similar experiments with the CIFAR-10 dataset, applying our created variational autoencoder to natural images with diverse structure.

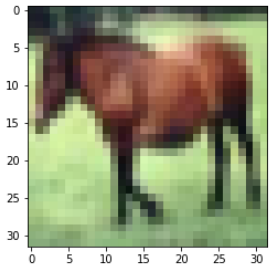


Figure 11. Encoded and decoded CIFAR-10 horse

Also, it would be very interesting to perform a comparison between a VAE and GAN applied to the CIFAR-10 dataset or related computer vision tasks, as we mentioned how GANs are subjectively better at generating data samples [8].

Lastly, research at the intersection of convolutional neural networks (CNN) and the variational autoencoder for the purposes of data generation is fascinating and immense. One paper in particular, by Jianwen Xie, Yang Lu, Song-Chun Zhu, and Ying Nian Wu at the University of California Los Angeles, in which a generative CNN is derived from a discriminative CNN in an effort to improve generated results, would be a great source for more realistic data generation using VAEs with CNNs [16]. Research by Dmitry Vetrov in scalable methods for Bayesian neural networks would also be useful for research regarding scalability [13].

10. Conclusion

In conclusion, our statistical formulation for the Variational Autoencoder (**VAE**) has given us a deeper understanding for the necessity of the variational lower bound, or evidence lower bound (ELBO), specific to the VAE. By

deriving a tractable objective function called the Stochastic Gradient Variational Bayes (SGVB) estimator, we can optimize the VAE model. We also applied the variational autoencoder to the MNIST database of handwritten digits and observed the fascinating results of the encoding, decoding, and data generation processes.

References

[1] Jaan Altosaar. Tutorial - what is a variational autoencoder? https://jaan.io/what-is-variational-autoencoder-vae-tutorial/.

[2] Autoencoder. Autoencoder — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Autoencoder.

[3] Bayesian Network. Bayesian network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Bayesian_network.

[4] Chistopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer. http://users.isr.ist.utl.pt/~wurmd/Livros/school/Bishop.

[5] Carl Doersch. Tutorial on variational autoencoders. https://arxiv.org/pdf/1606.05908.pdf.

[6] David Duvenaud. Differentiable inference and generative models. http://www.cs.toronto.edu/~duvenaud/courses/csc2541/.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] Jonathon C. Kao. Generative adversarial networks. https://seas.ucla.edu/~kao/nndl/lectures/gans.pdf.

[9] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *arXiv*, May 2014. https://arxiv.org/pdf/1312.6114.pdf.

[10] Volodymyr Kuleshov. Cs 228: Probabilistic graphical models. https://ermongroup.github.io/cs228-notes/.

[11] CS 229: Machine Learning. Machine learning. http://cs229.stanford.edu/.

[12] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[13] Daniil Polykovskiy and Alexander Novikov. Bayesian methods for machine learning. https://www.coursera.org/learn/bayesian-methods-in-machine-learning.

[14] Ying Nian Wu, Ruiqi Gao, Tian Han, and Song-Chun Zhu. A tale of three probabilistic families: Discriminative, descriptive and generative models. *Quarterly of Applied Mathematics*. http://www.stat.ucla.edu/~ywu/QAM2018.pdf.

[15] Jianwen Xie, Yang Lu, Song-Chun Zhu, and Ying Nian Wu. A theory of generative content. *Annals of Statistics*. http://www.stat.ucla.edu/~jxie/download/GenerativeConvNet.pdf.

[16] Jianwen Xie, Yang lu, Song-Chun Zhu, and Ying Nian Wu. A theory of generative convnet. http://www.stat.ucla.edu/~jxie/download/GenerativeConvNet.pdf.

[17] Qing Zhou. Causal dags: Inference and learning. http://www.stat.ucla.edu/~zhou/courses/Stats201C_DAG_Slides.pdf.

[18] Qing Zhou. Introduction to graphical models. http://www.stat.ucla.edu/~zhou/courses/Stats201C_Graph_Slides.pdf.

[19] Qing Zhou. Random graphs for modeling network data. http://www.stat.ucla.edu/~zhou/courses/Stats201C_Network_Slides.pdf.

```
try:
    import google.colab
    IN_COLAB = True
except:
    IN_COLAB = False
if IN_COLAB:
    print("Downloading Colab files")
    ! shred -u setup_google_colab.py
    ! wget https://raw.githubusercontent.com/hse-aml/bayesian-methods-for-ml/master/setup_google_colab.py
    import setup_google_colab
    setup_google_colab.load_data_week5()
```

```
Downloading Colab files
shred: setup_google_colab.py: failed to open for writing: No such file or directory
--2019-06-12 09:06:23--  https://raw.githubusercontent.com/hse-aml/bayesian-methods-for-ml/master
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.133,
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... connect
HTTP request sent, awaiting response... 200 OK
Length: 1308 (1.3K) [text/plain]
Saving to: 'setup_google_colab.py'

setup_google_colab. 100%[===================>]   1.28K  --.-KB/s    in 0s

2019-06-12 09:06:23 (215 MB/s) - 'setup_google_colab.py' saved [1308/1308]
```

```
import tensorflow as tf
import keras
import numpy as np
import matplotlib.pyplot as plt

from keras.layers import Input, Dense, Lambda, InputLayer, concatenate
from keras.models import Model, Sequential
from keras import backend as K
from keras import metrics
from keras.datasets import mnist
from keras.utils import np_utils
from w5_grader import VAEGrader
```

```
Using TensorFlow backend.
```

```
def vlb_binomial(x, x_decoded_mean, t_mean, t_log_var):
    """Returns the value of negative Variational Lower Bound

    The inputs are tf.Tensor
        x: (batch_size x number_of_pixels) matrix with one image per row with zeros and ones
        x_decoded_mean: (batch_size x number_of_pixels) mean of the distribution p(x | t), real numbers f
        t_mean: (batch_size x latent_dim) mean vector of the (normal) distribution q(t | x)
        t_log_var: (batch_size x latent_dim) logarithm of the variance vector of the (normal) distributio

    Returns:
        A tf.Tensor with one element (averaged across the batch), VLB
    """
    print(tf.shape(x))
    print(x.get_shape().as_list())
    print(x_decoded_mean.get_shape().as_list())
    print(t_mean.get_shape().as_list())
    print(t_log_var.get_shape().as_list())

    # Bernoulli: p^k *(1-p)^(1-k) for k in {0, 1}
    # k: ground truth
    # p: decoded value
    xent_loss = K.sum(x * K.log(x_decoded_mean + 1e-10) + (1 - x) * K.log(1 - x_decoded_mean + 1e-10), ax
    print(xent_loss.get_shape().as_list())

    kl_loss = 0.5 * K.sum((1 + t_log_var - K.square(t_mean) - K.exp(t_log_var)), axis=1)
    print(kl_loss.get_shape().as_list())

    # average across minuibatch
    vae_loss = - K.mean(xent_loss + kl_loss)
    return vae_loss
```

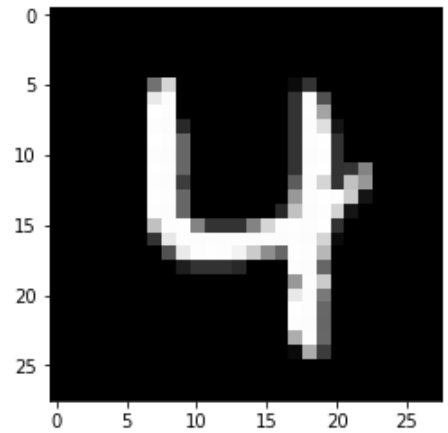
```
# Start tf session so we can run code.
sess = tf.InteractiveSession()
# Connect keras to the created session.
K.set_session(sess)


batch_size = 100
original_dim = 784 # Number of pixels in MNIST images.
latent_dim = 10 #3 # d, dimensionality of the latent code t.
intermediate_dim = 256 # Size of the hidden layer.
epochs = 3


x = Input(batch_shape=(batch_size, original_dim))
x1 = Input(batch_shape=(1, original_dim))
def create_encoder(input_dim):
    # Encoder network.
    # We instantiate these layers separately so as to reuse them later
    encoder = Sequential(name='encoder')
    encoder.add(InputLayer([input_dim]))
    encoder.add(Dense(intermediate_dim, activation='relu'))
    encoder.add(Dense(2 * latent_dim))
    return encoder
encoder = create_encoder(original_dim)

get_t_mean = Lambda(lambda h: h[:, :latent_dim])
get_t_log_var = Lambda(lambda h: h[:, latent_dim:])
h = encoder(x)
t_mean = get_t_mean(h)
t_log_var = get_t_log_var(h)

h1 = encoder(x1)
t1_mean = get_t_mean(h1)
t1_log_var = get_t_log_var(h1)

# Sampling from the distribution
#     q(t | x) = N(t_mean, exp(t_log_var))
# with reparametrization trick.
def sampling(args):
    """Returns sample from a distribution N(args[0], diag(args[1]))

    The sample should be computed with reparametrization trick.

    The inputs are tf.Tensor
        args[0]: (batch_size x latent_dim) mean of the desired distribution
        args[1]: (batch_size x latent_dim) logarithm of the variance vector of the desired distribution

    Returns:
        A tf.Tensor of size (batch_size x latent_dim), the samples.
    """
    t_mean, t_log_var = args
    epsilon = K.random_normal(shape=K.shape(t_mean), mean=0.0, stddev=1.0)
    return t_mean + K.exp(0.5 * t_log_var) * epsilon


t = Lambda(sampling)([t_mean, t_log_var])
t1 = Lambda(sampling)([t1_mean, t1_log_var])


def create_decoder(input_dim):
    # Decoder network
    # We instantiate these layers separately so as to reuse them later
    decoder = Sequential(name='decoder')
    decoder.add(InputLayer([input_dim]))
    decoder.add(Dense(intermediate_dim, activation='relu'))
    decoder.add(Dense(original_dim, activation='sigmoid'))
    return decoder
decoder = create_decoder(latent_dim)
x_decoded_mean = decoder(t)
x1_decoded_mean = decoder(t1)


loss = vlb_binomial(x, x_decoded_mean, t_mean, t_log_var)
vae = Model(x, x_decoded_mean)
# Keras will provide input (x) and output (x_decoded_mean) to the function that
# should construct loss, but since our function also depends on other
# things (e.g. t_means), it is easier to build the loss in advance and pass
# a function that always returns it.
vae.compile(optimizer=keras.optimizers.RMSprop(lr=0.001), loss=lambda x, y: loss)
```

```
Tensor("Shape_6:0", shape=(2,), dtype=int32)
[100, 784]
[100, 784]
[100, 10]
[100, 10]
[100]
[100]
```

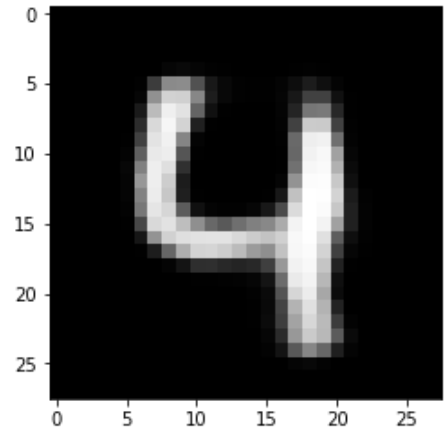
```
# train the VAE on MNIST digits
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# One hot encoding.
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))


epochs = 20
hist = vae.fit(x=x_train, y=x_train,
               shuffle=True,
               epochs=epochs,
               batch_size=batch_size,
               validation_data=(x_test, x_test),
               verbose=2)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
```

```
img_flat = x_test[49]
img = img_flat.reshape((28,28))
plt.imshow(img,  cmap='gray')
plt.show()

enc, enc_mean, enc_log_var = sess.run([h1, t1_mean, t1_log_var], feed_dict={x1: [img_flat]})
print(enc)
print(enc_mean)
print(enc_log_var)
print(h1)

dec = sess.run(x1_decoded_mean, feed_dict={t1: enc_mean})
res_img = dec.reshape((28,28))
plt.imshow(res_img,  cmap='gray')
plt.show()
```

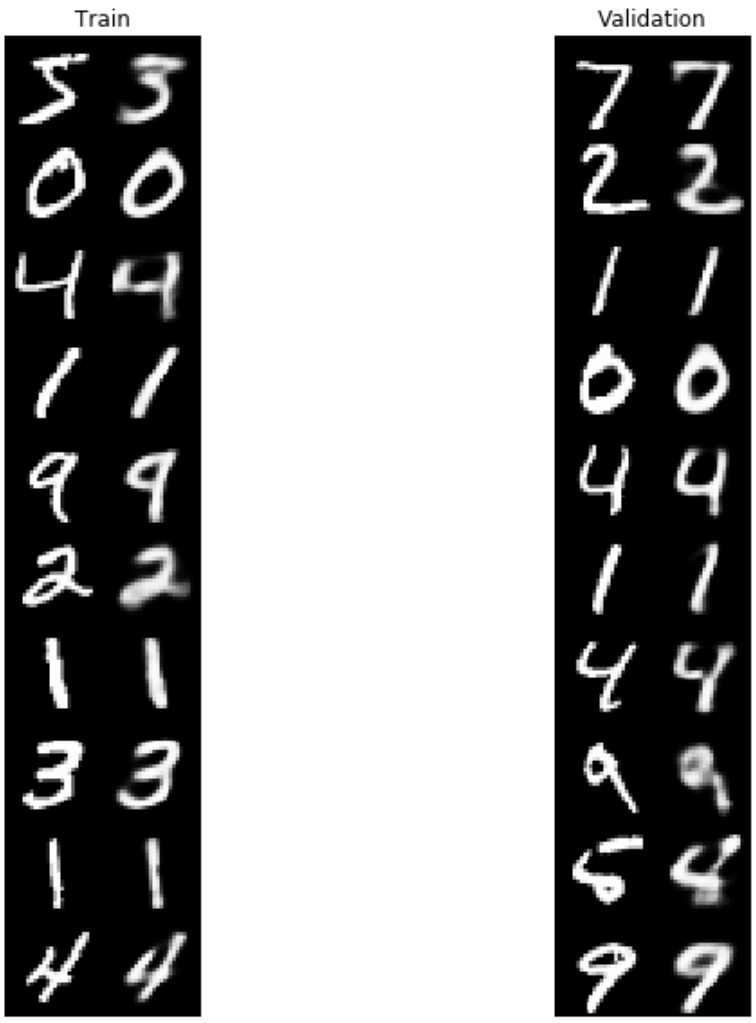


```
[[-0.28441328 -1.3568563   0.5647244  -0.77163976 -0.9561359   2.3299193
   0.4285398  -1.5410898  -1.6350224   0.7509855  -3.110401   -3.1845288
  -3.6625056  -3.6305943  -2.5657816  -3.3691223  -3.0476518  -3.2734396
  -3.321457   -3.748587  ]]
[[-0.28441328 -1.3568563   0.5647244  -0.77163976 -0.9561359   2.3299193
   0.4285398  -1.5410898  -1.6350224   0.7509855 ]]
[[-3.110401  -3.1845288 -3.6625056 -3.6305943 -2.5657816 -3.3691223
  -3.0476518 -3.2734396 -3.321457  -3.748587 ]]
Tensor("encoder_8/dense_18/BiasAdd:0", shape=(1, 20), dtype=float32)
```



```
fig = plt.figure(figsize=(10, 10))
for fid_idx, (data, title) in enumerate(
            zip([x_train, x_test], ['Train', 'Validation'])):
    n = 10  # figure with 10 x 2 digits
    digit_size = 28
    figure = np.zeros((digit_size * n, digit_size * 2))
    decoded = sess.run(x_decoded_mean, feed_dict={x: data[:batch_size, :]})
    for i in range(10):
        figure[i * digit_size: (i + 1) * digit_size,
               :digit_size] = data[i, :].reshape(digit_size, digit_size)
```

```
        figure[i * digit_size: (i + 1) * digit_size,
               digit_size:] = decoded[i, :].reshape(digit_size, digit_size)
    ax = fig.add_subplot(1, 2, fid_idx + 1)
    ax.imshow(figure, cmap='Greys_r')
    ax.set_title(title)
    ax.axis('off')
plt.show()
```




```
n_samples = 10  # To pass automatic grading please use at least 2 samples here.
# sampled_im_mean is a tf.Tensor of size 10 x 784 with 10 random
# images sampled from the vae model.
sample_t = K.random_normal(shape=(n_samples, latent_dim), mean=0.0, stddev=1.0)
sampled_im_mean = decoder(sample_t)
```

```
sampled_im_mean_np = sess.run(sampled_im_mean)
# Show the sampled images.
plt.figure()
for i in range(n_samples):
    ax = plt.subplot(n_samples // 5 + 1, 5, i + 1)
    plt.imshow(sampled_im_mean_np[i, :].reshape(28, 28), cmap='gray')
    ax.axis('off')
plt.show()
```

```
# One-hot labels placeholder.
x = Input(batch_shape=(batch_size, original_dim))
label = Input(batch_shape=(batch_size, 10))

xl = concatenate([x, label])

encoder = create_encoder(original_dim + 10)

h = encoder(xl)
cond_t_mean = get_t_mean(h)
cond_t_log_var = get_t_log_var(h)

print(xl.get_shape().as_list())
print(h.get_shape().as_list())
print(cond_t_mean.get_shape().as_list())
print(cond_t_log_var.get_shape().as_list())


t = Lambda(sampling)([cond_t_mean, cond_t_log_var])
tl = concatenate([t, label])

decoder = create_decoder(latent_dim + 10)
cond_x_decoded_mean = decoder(tl)

#cond_t_mean =  # Mean of the latent code (without label) for cvae model.
#cond_t_log_var = # Logarithm of the variance of the latent code (without label) for cvae model.
#cond_x_decoded_mean =  # Final output of the cvae model.
```

```
[100, 794]
[100, 20]
[100, 10]
[100, 10]
```

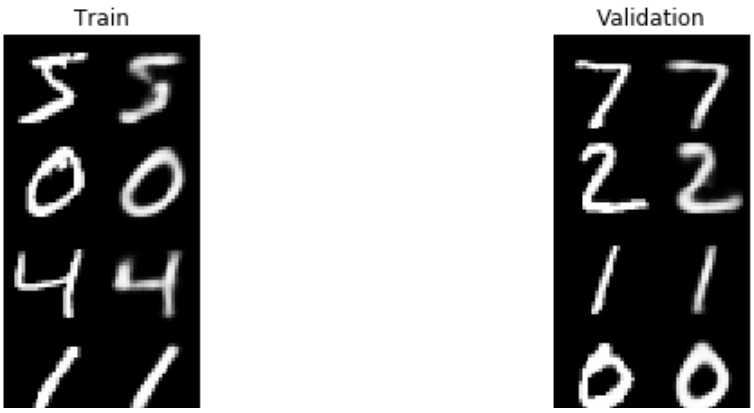
```
conditional_loss = vlb_binomial(x, cond_x_decoded_mean, cond_t_mean, cond_t_log_var)
cvae = Model([x, label], cond_x_decoded_mean)
cvae.compile(optimizer=keras.optimizers.RMSprop(lr=0.001), loss=lambda x, y: conditional_loss)
```

```
Tensor("Shape_7:0", shape=(2,), dtype=int32)
[100, 784]
[100, 784]
[100, 10]
[100, 10]
[100]
[100]
```

```
hist = cvae.fit(x=[x_train, y_train],
                y=x_train,
                shuffle=True,
                epochs=epochs,
                batch_size=batch_size,
                validation_data=([x_test, y_test], x_test),
                verbose=2)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
 - 2s - loss: 155.4058 - val_loss: 129.0842
Epoch 2/20
 - 2s - loss: 124.1803 - val_loss: 120.4274
Epoch 3/20
 - 2s - loss: 118.4696 - val_loss: 116.5869
Epoch 4/20
 - 2s - loss: 115.4052 - val_loss: 113.4894
Epoch 5/20
 - 2s - loss: 113.3610 - val_loss: 111.9370
Epoch 6/20
 - 2s - loss: 111.9178 - val_loss: 110.4100
Epoch 7/20
 - 2s - loss: 110.7301 - val_loss: 109.6823
Epoch 8/20
 - 2s - loss: 109.8311 - val_loss: 108.2383
Epoch 9/20
 - 2s - loss: 109.0987 - val_loss: 107.7660
Epoch 10/20
 - 2s - loss: 108.4247 - val_loss: 108.0507
Epoch 11/20
 - 2s - loss: 107.8561 - val_loss: 106.8372
Epoch 12/20
 - 2s - loss: 107.3355 - val_loss: 106.8308
Epoch 13/20
 - 2s - loss: 106.8826 - val_loss: 105.6550
```

```
fig = plt.figure(figsize=(10, 10))
for fid_idx, (x_data, y_data, title) in enumerate(
            zip([x_train, x_test], [y_train, y_test], ['Train', 'Validation'])):
    n = 10  # figure with 10 x 2 digits
    digit_size = 28
    figure = np.zeros((digit_size * n, digit_size * 2))
    decoded = sess.run(cond_x_decoded_mean,
                       feed_dict={x: x_data[:batch_size, :],
                                  label: y_data[:batch_size, :]})
    for i in range(10):
        figure[i * digit_size: (i + 1) * digit_size,
               :digit_size] = x_data[i, :].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               digit_size:] = decoded[i, :].reshape(digit_size, digit_size)
    ax = fig.add_subplot(1, 2, fid_idx + 1)
    ax.imshow(figure, cmap='Greys_r')
    ax.set_title(title)
    ax.axis('off')
plt.show()
```




```
# Prepare one hot labels of form
#   0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 ...
# to sample five zeros, five ones, etc
curr_labels = np.eye(10)
curr_labels = np.repeat(curr_labels, 5, axis=0)  # Its shape is 50 x 10.
# cond_sampled_im_mean is a tf.Tensor of size 50 x 784 with 5 random zeros,
# then 5 random ones, etc sampled from the cvae model.
sample_t = K.random_normal(shape=(50, latent_dim), mean=0.0, stddev=0.5)
sample_tl = concatenate([sample_t, tf.convert_to_tensor(curr_labels, dtype=tf.float32)])
cond_sampled_im_mean = decoder(sample_tl)
```

```
cond_sampled_im_mean_np = sess.run(cond_sampled_im_mean)
# Show the sampled images.
plt.figure(figsize=(10, 10))
global_idx = 0
for digit in range(10):
    for _ in range(5):
        ax = plt.subplot(10, 5, global_idx + 1)
        plt.imshow(cond_sampled_im_mean_np[global_idx, :].reshape(28, 28), cmap='gray')
        ax.axis('off')
        global_idx += 1
plt.show()
```

```
from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 6s 0us/step
```

```
plt.imshow(x_train[7, :])
plt.show()
```