Variational Auto Encoder Latent Space Activity and Visualization

Jesse Bettencourt
Department of Mathematics
University of Toronto
Toronto, ON
jessbett@math.toronto.edu

Matt Craddock
Department of Computer Science
University of Toronto
Toronto, ON
matt.craddock@mail.utoronto.ca

Abstract

In this project we implement a Variational Auto Encoder (VAE) in Tensorflow. We compare experimental results for modifications to the VAE, including Importance Weighting, Batch Normalization, and Warm-Up. In particular, we consider how these additional methods affect the training rate and latent dimension activity. Finally, we demonstrate latent visualization techniques available with VAEs.

1 Introduction to Variational Autoencoders

Variational Auto Encoders (VAEs) were introduced by Kingma & Welling 2013 as generative analogues to the standard deterministic auto encoder [5]. As with deterministic auto encoders, VAEs pair a bottom-up inference network called an encoder with a top-down generative network called a decoder.

VAEs employ a probabilistic interpretation of these encoder and decoder networks. We assume that our data set $\{x^{(i)}\}_{i=1}^{N}$ are N i.i.d. samples of some variable x. Further, we assume that the data was generated by a random process with continuous latent variable z. So we have that our data x was generated by some conditional distribution $p_\theta(x|z)$, where p_θ is a probability distribution with parameters θ. This provides a probabilistic interpretation of the decoder network, where given a latent variable or 'code' z we generate a sample x in the data space. Similarly, the role of the encoder would be to take a sample x from data space and give us a latent z sampled from the posterior density distribution $p_\theta(z|x)$.

However, this is where problems arise in the probabilistic interpretation. It is common that the posterior density distribution $p_\theta(z|x)$ is intractable. In order to learn an encoder-decoder network pair, VAEs instead learn a different inference model, $q_\phi(z|x)$, which approximates the true, intractable posterior distribution. Note that our approximate inference model, the encoder distribution, has parameters ϕ distinct from the θ of the true posterior and decoder network. Training a VAE will amount to jointly learning these these parameters.

Variational auto encoders are further characterized by their training criterion. Instead of learning an often intractable log-likelihood, the training objective $\mathcal{L}(x)$ is a tractable lower bound to the log-likelihood:

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] = -\mathcal{L}(x) \tag{1}$$

$$\mathcal{L}(x) = D_{KL}\left(q_\phi(z|x)||p_\theta(z)\right) - \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] \tag{2}$$

Where D_{KL} is the Kullback-Leibler divergence. It will be useful later to directly identify the two components of our objective. The reconstruction error term $\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]$ is

present in deterministic auto encoders, and represents the likelihood that the input data would be reconstructed by the model. The variational regularization term $D_{KL}\left(q_\phi(z|x)||p_\theta(z)\right)$ represents the KL-divergence between the encoder induced latent distribution and the true prior on the latent distribution. This term encourages our approximate posterior $q_\phi(z|x)$ to be close to $p_\theta(z)$.

Finally, one last detail to discuss in the process of training a VAE is the reparametrization trick. Since the reconstruction error term is estimated by sampling $z \sim q_\phi(z|x)$, there is a problem with using gradient training methods through the sampling process. To address this, Kingma & Welling describe an alternative method for generating the samples, simply to let $z = g_\phi(x, \epsilon)$ be a deterministic function of ϕ and ϵ be some independent noise.

For example, in our implementation which we will soon describe in detail, we assume that the true posterior distribution can be approximated by a multivariate Gaussian with diagonal covariance. Therefore we let $q_\phi(z|x) = \mathcal{N}(z; \mu, \sigma^2 I)$. So the outputs of our encoder network are the μ and σ of our approximate posterior. Now in order to train with gradient methods, we reparameterize as described, letting $z = \mu + \sigma\epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$.

2 Implementing Variational Autoencoders

2.1 Network Architecture

Each of the following methods were included into what will we call the 'Vanilla VAE'. The network architecture for the Vanilla VAE and all additional methods is similar to the architecture introduced in the Section 3 example of Auto-Encoding Variational Bayes [5].

The encoder and decoder networks are symmetric, shallow, fully-connected neural networks, namely Multi Layer Perceptrons (MLP). Both feature two deterministic layers each with 200 dimensions (or nodes) per layer. A stochastic, or latent, layer with dimensions n_z receives the output from the final deterministic layer. In the nodes of the hidden layers the activation is determined by the softplus function, $\ln\left(1_e^x\right)$.

Practically, the probabilistic encoder network takes data from the input space and encodes a representation into a latent space with dimension n_z. In particular, the latent representation, $q_\phi(z|x) = \mathcal{N}(z; \mu, \sigma^2 I)$ is a Gaussian distribution over the possible latent values of z from which data x could have been generated. The probabilistic decoder network takes a latent representation and produces a distribution $p_\theta(x \mid z)$ over possible data values x generated by z.

We implement this network and the following additional methods in Tensorflow. Our implementation follows from examples in Tensorflow and Theano [8, 7, 1]. In all examples we are training on the MNIST handwritten digit data set. We learn the MLP weights and bias parameters, representing the ϕ and θ distribution parameters, with Adam optimization minimizing $\mathcal{L}(x)$ with parameters $\beta_1 = 0.9, \beta_2 = 0.9, \epsilon = 10^{-4}$ a batch size of 100, and learning rate of 0.001, trained for 300 epochs.

2.2 Xavier Initialization

All parameters in the MLP were initialized with the Xavier-Glorot method outlined in [3]. Xavier-Glorot initialization is shown to improve learning in deep networks by establishing an effective range for initial values. In general, especially for deep networks, there is an initial value trade-off. If the initial weights are too small then the signal magnitude will decrease through the layers and and the influence will tend too small to be useful. If the weights are too large then the signal will grow as it passes through the layers and will tend to be too large to be representative.

Xavier-Glorot initialization addresses this trade-off by sampling initialization weights from a Gaussian distribution with zero mean and variance as a function of the network connections for the node. The variance for weight w of a neuron is given as a function of the number of neurons feeding into it n_{in} and the number of neurons the result feeds to n_{out}. The function is defined to be $\text{Var}(w) = \frac{2}{n_{in}+n_{out}}$.

2.3 Decoder Distribution

As mentioned previously, the probabilistic decoder network takes a latent representation z and produces a distribution over possible data values, $p_\theta(x \mid z)$. In our initial description of the network architecture we specified that output of the encoder MLP is Gaussian, but we made no specification to the decoder output distribution.

Two choices for decoder distributions, Gaussian and Bernoulli, were outlined in *Auto-Encoding Variational Bayes*, where the authors suggest that the choice of preferred decoder distribution depends on the type of data [5].

2.3.1 Gaussian Decoder and Encoder Structure

For continuous, real-valued data, Kingma & Welling suggest letting $p_\theta(x \mid z)$ be a multivariate Gaussian distribution. This gives the following structure for the decoder distribution with two hidden deterministic layers h_1 and h_2:

$$\begin{aligned}
\log p_\theta(x \mid z) &= \log \mathcal{N}(x; \mu, \sigma^2 I) \\
\mu &= \text{sigmoid}(W_\mu h_2 + b_\mu) \\
\log \sigma^2 &= \tanh(W_\sigma h_2 + b_\sigma) \\
h_2 &= \text{softplus}(W_2 h_1 + b_2) \\
h_1 &= \text{softplus}(W_1 z + b_1)
\end{aligned}$$

Note that the parameters $\{W_1, W_2, W_\mu, W_\sigma, b_1, b_2, b_\mu, b_\sigma\}$ here are the learned parameters of the decoder MLP, and represent the decoder distribution parameter, θ in $p_\theta(x \mid z)$. Further, since our encoder distribution is always a multivariate Gaussian distribution, we use this structure for the encoder, where the z and x are swapped and the weights and biases represent the encoder distribution parameter, ϕ in $q_\phi(z \mid x)$.

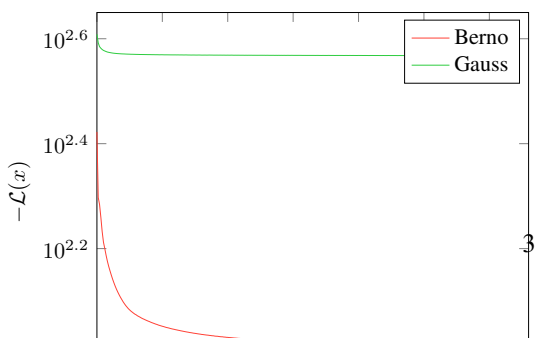
2.3.2 Bernoulli Decoder Structure

For binary data, Kingma & Welling suggest letting $p_\theta(x \mid z)$ be a multivariate Bernoulli distribution. This gives the following structure for our decoder distribution with two hidden deterministic layers h_1 and h_2:

$$\begin{aligned}
\log p_\theta(x \mid z) &= \sum_{i=1}^{D} x_i \log y_i + (1 - x_i) \cdot \log(1 - y_i) \\
y &= \text{sigmoid}(W_\mu h_2 + b_\mu) \\
h_2 &= \text{softplus}(W_2 h_1 + b_2) \\
h_1 &= \text{softplus}(W_1 z + b_1)
\end{aligned}$$

Again, here the Bernoulli decoder distribution parameter θ is represented by the learned MLP weights and biases $\theta = \{W_1, W_2, W_\mu, b_1, b_2, b_\mu\}$.

2.3.3 Comparing Decoder Distributions



Our findings support the recommendation by Kingma & Welling that Bernoulli decoder distribution performs better on binary data than a Gaussian decoder. The one-hot MNIST data used to train our VAE is binary, and thus we expected that the Bernoulli decoder would outpreform the Gaussian decoder. The results of

this experiment can be found in Figure 2.1. As expected, the Bernoulli distribution was able to learn a much tighter lower bound on the log-likelihood. It is important to note that VAE with the Gaussian decoder has converged to its lower bound, suggesting that even with longer training it would not learn a better model than the VAE with Bernoulli decoder distribution. Therefore, when implementing VAEs it is important to choose a decoder distribution that is representative of your data type.
Note that all further methods were implemented on VAEs with Bernoulli decoder distributions.

2.4 Importance Weighting

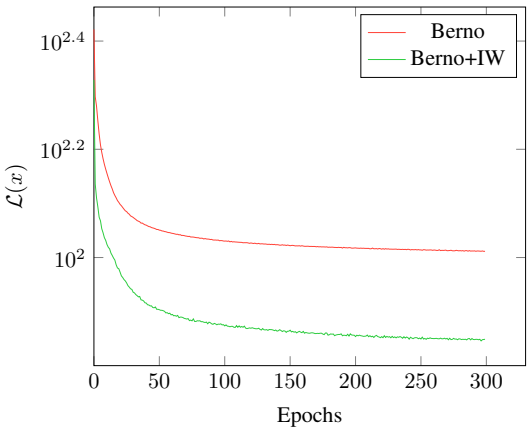


Figure 2.2: Variational v.s. Importance Weighted (IW)

In their 2015 publication *Importance Weighted Autoencoders*, Burda, Grosse, and Salakhutdinov observed that Kingma & Welling's $\mathcal{L}(x)$ lower bound on the log-likelihood from Eq.2 made strong assumptions about the posterior inference, leading to overly simplified representations. They proposed an improvement called Importance Weighted Auto Encoders (IWAE), a generative model using the VAE network architecture with a few key improvements to model generalizability.

The critical feature of IWAE is that the encoder network uses multiple importance weighted samples to approximate the posterior, where the original VAE uses a single sample. This allows IWAE to approximate complex posteriors which are not available under the stronger VAE posterior assumptions.

By considering multiple importance weighted samples we introduce a new lower bound on the log-likelihood which is strictly tighter than Eq.2.

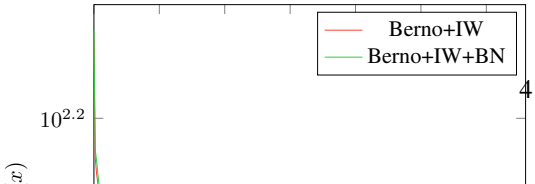
Given K independent samples $\{z_1, \ldots, z_K\}$ from the encoder distribution $z_k \sim q_\phi(z|x)$ we define the new lower bound given by the K-sample importance weighting expectation of the log-likelihood:

$$\mathcal{L}_K(x) = \mathbb{E}_{z_1,\ldots,z_K \sim q_\phi(z|x)} \left[\log \frac{1}{K} \sum_{k=1}^{K} \frac{p_\theta(x, z_k)}{q_\phi(z_k|x)} \right] \tag{3}$$

The term inside the sum is the normalized importance weights for the joint distribution. Note in particular that the case $K = 1$ corresponds exactly to Kingma & Welling's $\mathcal{L}(x)$ from Eq.2. Burda, Grosse, and Salakhutdinov show that this importance weighted lower bound is strictly tighter than the vanilla lower bound. In particular, that $\log p(x) \geq \mathcal{L}_{K+1}(x) > \mathcal{L}_K(x)$ Our implementation of IWAE with this lower bound supports their findings. We observed that with $K = 5$ samples IWAE significantly improves the learned log-likelihood bound during training. The results of this experiment can be found in Figure 2.2.
Since importance weighting so significantly improved the performance of our Auto Encoder, all further methods were implemented on VAEs with importance weighting.

2.5 Batch Normalization

Batch normalization is a recent method developed to improve stability and convergence



speed in deep networks [4]. In the recent paper *How to Train Deep Variational Autoencoders and Probabilistic Ladder Networks* by Sønderby et al. 2016, the authors show that batch normalization is an important method for learning deep VAEs [8]. That is, deep generative models with several latent layers. Though we are only considering a shallow model with a single latent layer, we were interested to see how batch normalization affected training as well as latent space activity (discussed in sections to follow).

Batch normalization was developed to improve learning stability during deep network training. As parameters change during learning the layer output distributions change for each hidden layer, requiring later layers to adjust parameters in response to these distribution changes. The problem that batch normalization attempts to address is that changes in early layer output distributions can cause noisy changes to later layers.

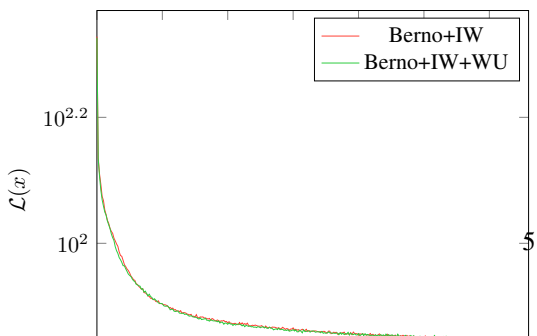
Batch normalization functions by normalizing the inputs of the activation function for each layer so that the inputs across each training batch have a mean of 0 and a variance of 1. However, batch normalization restricts the representation of each layer by assuming this normal distribution. To alleviate some of this restriction, batch normalization introduces learnable parameters to scale the variance of the normal distribution, γ and shift the mean, β. Therefore, we transform each activation function input, x_i, with the batch normalization given by:

$$\text{BN}(x_i) = \gamma \left(\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \beta$$

where μ_B and σ_B are the mean and standard deviation of the layer's activation function input across the batch, ϵ is a small constant addition to the variance to avoid division by 0, and γ and β are learnable scale and shift parameters. Note in particular that when learning with batch normalization, the learned shift parameter β for each layer replaces the need for bias terms added to that layer's activation inputs.

To train our model we add batch normalization before the activation for all layers in our auto-encoder except the output layers, as described in [8]. See the result of training with batch normalization in Figure 2.3. Our experiments resulted in VAEs trained with batch normalization performing poorer than VAEs without batch normalization. While the literature suggests that batch normalization significantly improves training for deep networks, we found no improvements to our shallow VAEs. We suggest that the reason for this result is that the benefit of batch normalization is to stabilize the propagation of signal throughout the layers of a deep network. However, to achieve this stabilization, batch normalization imposes assumptions on the distribution of activation inputs for each layer. We suggest that batch normalization reduced performance on our VAEs because they were too shallow to receive any benefit from signal stabilization, and were made less effective by the normalization assumption.

2.6 Warm Up



Recall that the log-likelihood lower bound in Eq.2 contains a reconstruction term and a variational regularization term. Further, notice that without that variational regularization term the lower bound becomes that of a standard deterministic autoencoder. It has been observed that the variational regularization term causes

some latent dimensions to become inactive or 'pruned' during training [6, 2]. In the later sections of this report we consider the activity of the latent dimensions, and particularly how training prunes or preserves latent dimensions.

Pruning non-informative dimensions later in training could be considered advantageous for automatic relevance determination. However, if latent dimensions are pruned too early in training they will not have a chance to learn informative representations. Once the dimensions become inactive in training, they are unlikely be reactivated. This problem of early latent dimension pruning is particularly troublesome for deep VAEs, because deep latent layers depend on the shallow latent dimensions in the network. If shallow latent dimensions are pruned early, deep latent layers will not be able to learn useful representations [8].

To avoid the problem of early pruning due to the variational regularization, we 'warm up' our VAE. Warm-up is achieved by initializing the learning process with the objective of a standard deterministic autoencoder, and then linearly introducing the variational regularization. This way the latent dimensions have a chance to learn useful representations as in a deterministic autoencoder before being possibly pruned by variational regularization.
We introduce a warm-up parameter β to our objective function which increases linearly from 0 to 1 during the first N_T epochs of training:

$$-\mathcal{L}(x)_T = -\beta D_{KL}\left(q_\phi(z|x)||p_\theta(z)\right) + \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] \tag{4}$$

Again, note that this causes the first epoch to initialize a standard deterministic autoencoder objective function, then linearly introduce the variational behaviour. Further, observe that after N_T epochs the model becomes a fully variational autoencoder. This warm-up can also be applied to the lower bound objective of the IWAE identically, by linearly scaling the variational regularization term. See the result of training with warm-up in Figure 2.4. We observe that VAEs with warmup learn at the same rate and converge to the same lower bound as VAEs without warmup. However, we will discuss later how warm-up improves early latent dimension pruning, as desired.

2.7 Latent Space Dimensionality

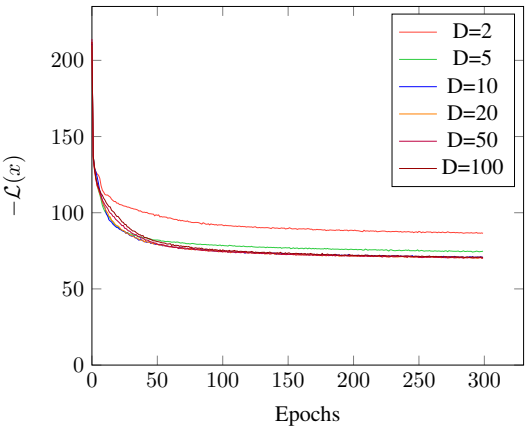


Figure 2.5: Latent Dimensionality

Finally, we were interested in determining how the choice of latent space dimensionality affected the training of our model. As described previously, the number of latent dimensions, n_z is given by the number of nodes in the stochastic layer of the model.

To experiment with this parameter, we trained multiple Importance Weighted VAEs with dimensionality $n_z \in \{2, 5, 10, 20, 50, 100\}$. See the effect of dimensionality on training in Figure 2.5. Our experiments show that models with 2 or 5 latent dimensions were not trained as successfully as models with higher latent dimensions. Further, it is interesting to note that all models with $n_z \geq 10$ converged to the same log-likelihood bound. This suggests to us that the additional latent dimensionality was not useful for learning latent representations of the MNIST data. In fact, $n_z = 10$ converged slightly faster than higher latent dimensional models, further suggesting that the additional dimensions were superfluous to training.

We were particularly interested in this question of latent space dimensionality. It is especially suggestive that models with $n_z \geq 10$ perform identically, given that there are 10 classes of digits in MNIST. This motivates our inquiry for the next section where we explore usefulness of latent dimensions by describing their activity.

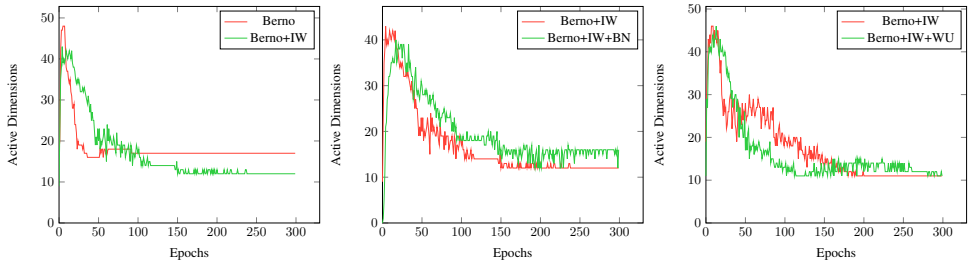
3 Latent Dimension Activity

3.1 Activity Metric

In the previous sections we discussed the concept of activity. Here we define it explicitly, using a method set forward by Burda, Grosse, & Salakhutdinov [2]. Given that the distribution parameters of nodes in the latent space combine to form the latent representation of the data, they observed that if the parameters remain the same or very similar in the latent representations of all data in the training set, the "contribution" of those nodes to the understanding of the data is minor. As such, they defined a measurement of activity which amounts to a calculation of the variance of the expected value of the latent distribution:

$$Cov_{\mathbf{x}}\left(\mathbb{E}_{u\ q(u|\mathbf{x})}[u]\right) > \epsilon \tag{5}$$

Where u represents a single dimension in the latent space, q is the generative distribution of the encoder network q_ϕ, and ϵ is a threshold value. Any dimension whose activity (represented by the left-hand side of the equation) exceeds ϵ is defined as "active" for the purposes of our experiment. In the experiment from [2], an activation of $\epsilon = 10^{-2}$ was used, but for this experiment a higher threshold of $\epsilon = 10^{-1}$ proved to be more useful as many dimensions surpassed the lower threshold with strong random variation. The effect of the additional features described above averaged over multiple trials can be seen in the plots below.



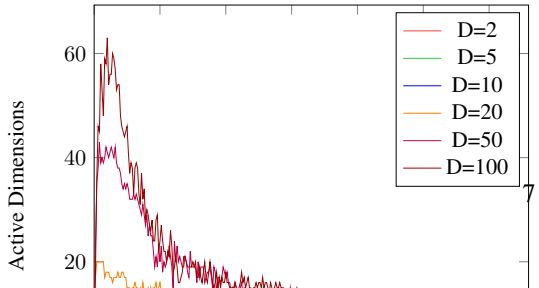
(a) Effect of Importance Weighting (IW)

(b) Effect of Batch Normalization (BN)

(c) Effect of Warm-Up

3.2 Effects of Implementation on Activity

It is clear from the above that, as training converges, importance weighting tends to reduce the number of active dimensions, while batch normalization increases it. Warmup was not seen to affect the final dimensionality count, but did accelerate its convergence. The dimension-reducing effect of importance weighting is likely a result of its more accurate loss function, which reduces the capacity for error in the network and forces a more specific lower entropy latent representation of the data. The dimension-increasing effects of batch normalization are likely a result of its tendency to force a distribution on the output of the deterministic layers. This introduces noise in the form of an additional influence on the data that is passed to the latent layer, which it must then compensate for. The effect of warmup is likely a result of the distinct loss function, which punishes frivolous and incorrect information in the latent layer disproportionately in the early training stages.

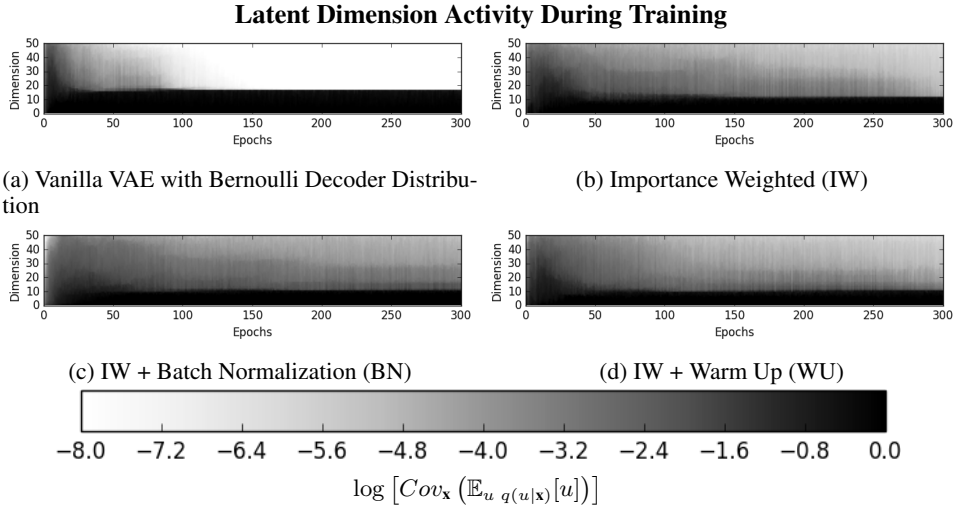


An interesting result can be seen in Figure 3.2, in which the dimensionality is varied for models trained with importance weighting and neither batch normalization or

warmup. As the dimensionality of the model increases, the number of active dimensions as the training converges tends to be constant at $min(D_{init}, 10)$ where D_{init} is the number of dimensions being trained (the "initial" dimensionality of the latent space before pruning). This suggests that a fixed number of dimensions in the latent space may be ideal for representing a particular data set. In this case, 10. This is consistent with the results in Figure 2.5, which indicated that an increase in the dimensionality beyond 10 had little effect on the loss function of the trained model.

3.3 Visualizing Latent Space Activity during Training

In their research Sønderby et al. [8] put forward a shaded rectangle plot for latent space activation that we have adapted below in Figures 3.3a to 3.3d. In these plots, the x-axis represents the number of training epochs that have passed, the y-axis represents the index of a dimension, and the shade of a pixel at a particular x-y coordinate represents the log of the activation defined in 5 of the yth dimension after the xth epoch. The log-activation is calculated by sampling the variance of the dimensional mean over a random subset of the data. In this experiment, the dimensions are sorted based on their activation after the first epoch, which means that, in general, more active (and therefore darker) dimensions are near the bottom of the chart, with less active or inactive dimensions near the top. These shade plots provide a more holistic view of the activity than simply plotting the number of dimensions whose activity exceeds the threshold. For examples, in Figures 3.3b, 3.3c, and 3.3d it is clear that the dimensions labelled as inactive by the threshold measurement are making a nonzero contribution to the output of the latent layer, which may be useful in models with multiple latent stochastic layers, and based on the cost functions plotted in Figures 2.2, 2.3, and 2.4 is certainly useful in improving the accuracy of the model.



(a) Vanilla VAE with Bernoulli Decoder Distribution

(b) Importance Weighted (IW)

(c) IW + Batch Normalization (BN)

(d) IW + Warm Up (WU)

4 Latent Space Visualization

In this section we demonstrate three techniques for visualizing the latent representation learned by the VAE.

4.1 Data Reconstruction

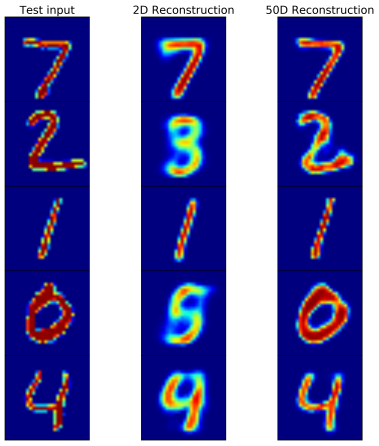


Figure 4.1: Reconstruction Visualization

The first and most apparent technique for visualizing the model learned by our VAEs is to generate images of reconstructed input data. In this visualization we give a single test input to our VAE. The test input is first encoded to a representation in the latent space by the encoder network. This latent representation is then decoded to a reconstructed value in the data space.
We generated examples of reconstruction visualizations for VAEs with 2 and 50 latent dimensions, see Figure 4.1. As we would expect, the 50 dimensional VAE was able to learn better representations for every test input than the 2 dimensional VAE. However, the 2 dimensional VAE performed comparably well on certain input classes, namely digits 1 and 7, but was completely unable to represent other classes, digits 2 and 0.

To better understand why our 2 dimensional VAE was unable to learn representations for these classes, we utilize other visualization techniques which illustrate the distribution of the data classes in the latent space.

4.2 Data Latent Encoding

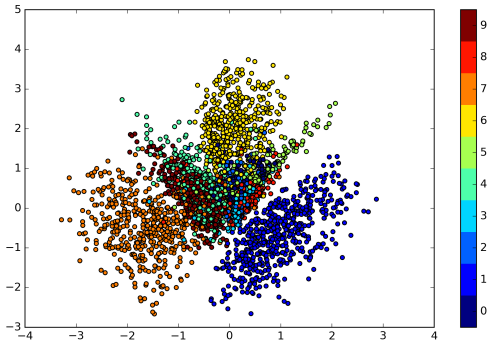


Figure 4.2: Data Encoding Visualization

This visualization technique utilizes just the encoder network of our VAE. To understand how our data classes are encoded into the latent distribution, we input a subset of our data and plot their code values in latent space. For an example of this visualization with our 2 dimensional VAE, see Figure 4.2. This visualization method clearly demonstrates which classes the VAE successfully modelled, and which classes were poorly represented. From our previous visualization, we observed that digits 1 and 7 were clearly reconstructed, and the data encoding visualization demonstrates why. Data classes which are clearly distinguished in latent space from other classes were better constructed by our VAE. Notice that the classes for digits 2 and 3 are not distinguished in the latent encoding, this corresponds to the incorrect reconstruction we observed previously.

4.3 Latent Manifold Lattice Generation

The final visualization technique utilizes just the decoder network of our VAE. To visualize our learned manifold we choose latent parameters on a lattice of a hyperplane of our latent space. Then we decode the values on our lattice and visualize the lattice in data space. See an example of this visualization in Figure 4.3. Notice that this manifold exactly corresponds to the encoding of our data classes into latent space, e.g. 6s at the top and 1s in the lower right. This is an extremely valuable visualization method because it illustrates how model transitions between data classes. Further, it

accessibly demonstrates instances of problematic representation, such as the transition from 7s to 9s to 4s. This visualization technique demonstrates clearly how the latent space encodes representations of our data.

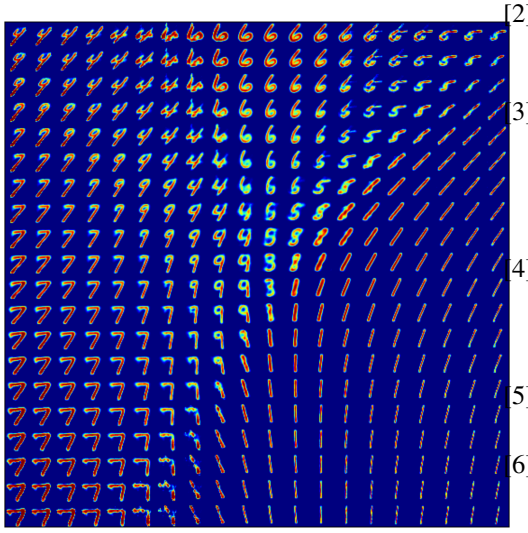


Figure 4.3: Latent Manifold Visualization

References

[1] Arun Ahuja. Generative Models in Tensorflow.

[2] Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance Weighted Autoencoders. pages 1–12, 2015.

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 9:249–256, 2010.

[4] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv*, 2015.

[5] Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. (Ml):1–14, 2013.

[6] David MacKay. Local minima, symmetry-breaking, and model pruning in variational free energy minimization. *Inference Group, Cavendish Laboratory,*, pages 1–10, 2001.

[7] Jan Metzen. Variational Autoencoder in TensorFlow, 2015.

[8] Casper Kaae Sønderby, Tapani Raiko, Lars Maaløe, Søren Kaae Sønderby, and Ole Winther. How to Train Deep Variational Autoencoders and Probabilistic Ladder Networks. 2016.