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 additional methods 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 encoder with a top-down generative network called decoder.
VAEs employ probabilistic interpretation of these encoder and decoder networks. We assume that our dataset $\{x^{(i)}\}_{i=1}^{N}$ 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 distribution with parameters θ. This provides a probabilistic interpretation of the encoder 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 $p_\theta(z|x)$.

Matt, this is how we'll leave comments

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 different to 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:

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

where

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

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 even 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 some prior on the latent distribution. This term encourages our approximate posterior $q_\phi(z|x)$ to be close to $p_\theta(x)$.
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 describe, letting $z = \mu + \sigma\epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$.

2 Implementing Variational Autoencoders

In this section we will detail methods for VAE implementation and effect on learning rate.

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, simple, fully-connected neural networks, namely Multi Layer Perceptrons (MLP). Both feature two deterministic, or hidden, layers each with 200 dimensions (or nodes) per layer. A stochastic, or latent, layer with dimensions n_z on top the deterministic layers. In hidden layers the activation is the softplus function.
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 [**?**, 7, 1]. In all examples we are training on the MNIST handwritten digit dataset. We learn the MLP weights and bias parameters, representing the ϕ and θ distribution parameters, with Adam optimization minimizing $\mathcal{L}(x)$ with parameters $\beta_1 = 0.9, \beta_2 = 0.9, \epsilon = 10^{-4}$ with batch size 100, learning rate 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 a reasonable range for initial values. Especially for deep networks there is an initial value trade-off. If the initial weights are too small then the signal will shrink through the layers and and the influence will trend too small to be useless. If the weights are too large then the signal growth through the layers will trend to 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}. Then the variance is defined as $\text{Var}(w) = \frac{2}{n_{in}+n_{out}}$.

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 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:

$$\log p_\theta(x \mid z) = \log \mathcal{N}(x; \mu, \sigma^2 I)$$

where

$$\begin{aligned} \mu &= \text{sigmoid}(W_\mu h_2 + b_\mu) \\ \log \sigma^2 &= \tanh(W_\sigma h_2 + b_\sigma) \\ h_2 &= \text{softplus}(W_2 h_1 + b_2) \\ h_1 &= \text{softplus}(W_1 z + b_1) \end{aligned}$$

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:

$$\log p_\theta(x \mid z) = \sum_{i=1}^{D} x_i \log y_i + (1 - x_i) \cdot \log(1 - y_i)$$

where

$$\begin{aligned} y &= \text{sigmoid}(W_\mu h_2 + b_\mu) \\ h_2 &= \text{softplus}(W_2 h_1 + b_2) \\ h_1 &= \text{softplus}(W_1 z + b_1) \end{aligned}$$

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 we expected that the Bernoulli decoder would out-preform the Gaussian decoder. The results of this experiment can be found in Figure 1a.

2.4 Importance Weighting

In their 2015 publication *Importance Weighted Autoencoders*, Burda, Grosse, and Salakhutdinov observe that Kingma & Welling's $\mathcal{L}(x)$ lower bound on the log-likelihood from Eq.2 makes strong assumptions about the posterior inference leading to overly simplified representations. They propose an improvement called Importance Weighted Auto Encoders (IWAE) which is a generative model using the VAE network architecture, but 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 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 $\{z_1, \dots, z_K\}$ 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:

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

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

2.5 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 essential method for learning deep VAEs [**?**]. 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 affects training and, as we will discuss later, the latent space dimensionality.
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 respond 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 addresses this problem 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:

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

where μ_B and σ is the mean and standard deviation of the layer's activation function input across the batch, ϵ is a small constant 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 [**?**].
See the result of training with batch normalization in Figure

2.6 Warm Up

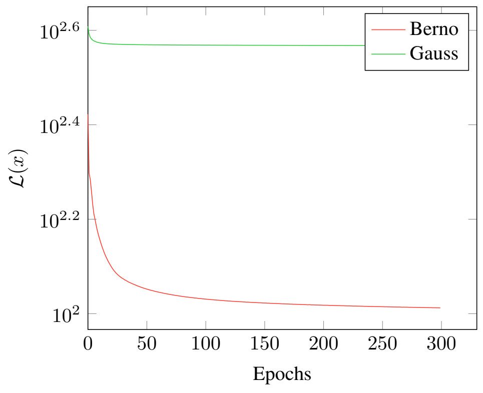
Recall that the log-likelihood lower bound in Eq.2 contains a reconstruction term and a variational regularization term. Further, notice that without that variational regularization term the lower bound becomes that for a standard deterministic autoencoder. It has been observed that the variational regularization 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, particularly we are interested in how training maintains or prunes 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 will not 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 [**?**].
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 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:

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

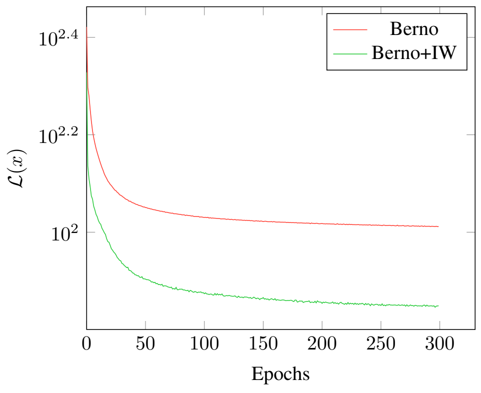
Again, note that this causes the first epoch to initialize a standard deterministic autoencoder, then linearly introduce the variational behaviour. Further, observe that after N_T epochs the model remains a fully variational autoencoder. This warm-up can also be applied to the lower bound objective of the IWAE identically, by linearly scaling the variational regularization term.
See the result of training with warm-up in Figure

2.7 Latent Space Dimensionality

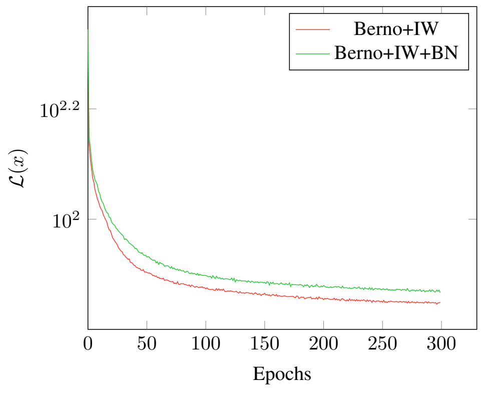
Finally, we were interested in determining how the choice of latent space dimensionality affected the training of our model. As described previously, the number of latent dimensions, n_z is given by the number of nodes in the stochastic layers of the model.
To experiment with this parameter, we trained multiple Importance Weighted VAEs with dimensionality $n_z \in \{2, 10, 20, 50, 100\}$. See the effect of dimensionality on training in Figure 1e. To summarize our results, models with 2 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.



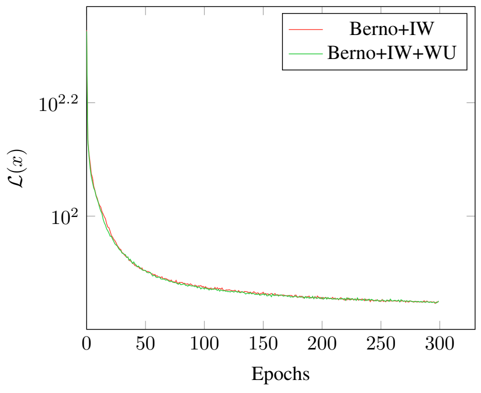
(a) Bernoulli v.s Gaussian decoder distributions



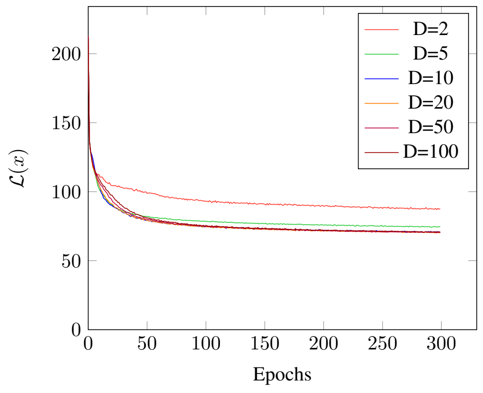
(b) Variational v.s. Importance Weighted (IW)



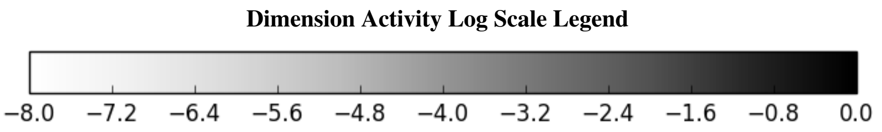
(c) Effect of Batch Normalization (BN)

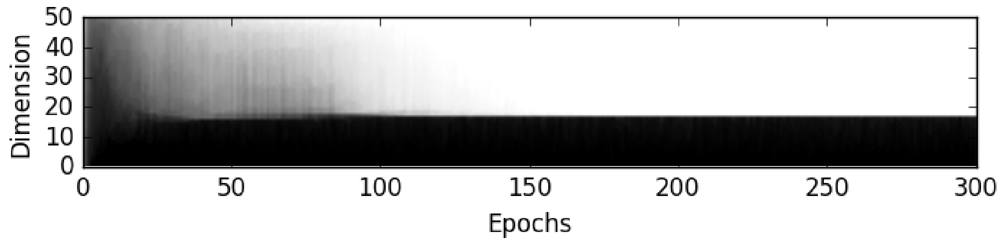


(d) Effect of Warm-Up

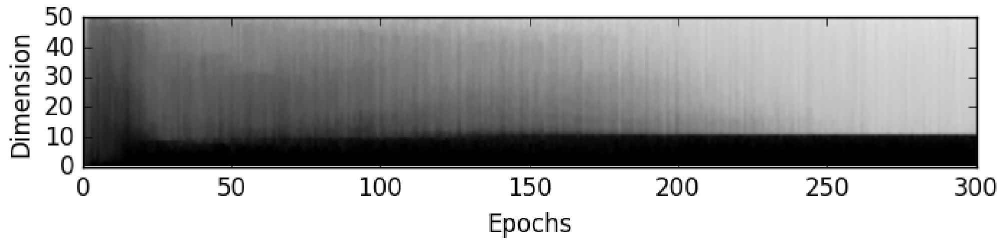


(e) Latent Dimensionality

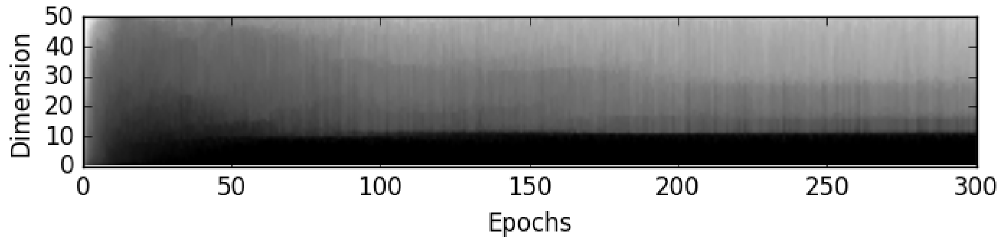




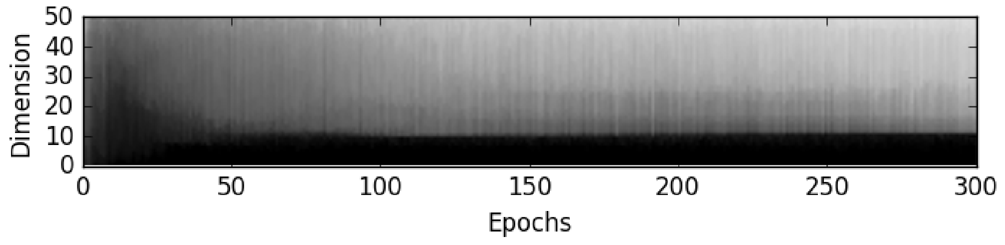
(a) Vanilla VAE with Bernoulli Decoder Distribution



(b) Importance Weighted (IW)



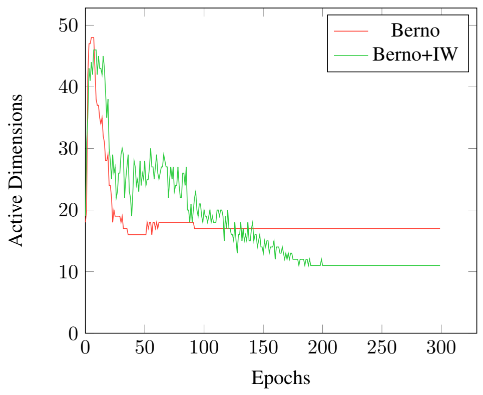
(c) IW + Batch Normalization (BN)



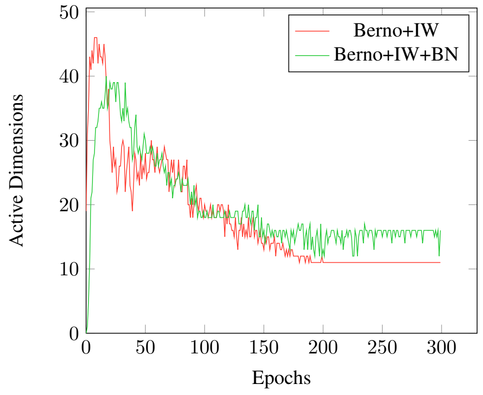
(d) IW + Warm Up (WU)

3 Latent Dimension Activity

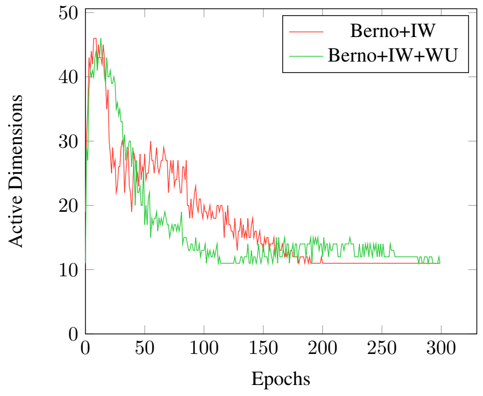
In the previous section we noticed that the



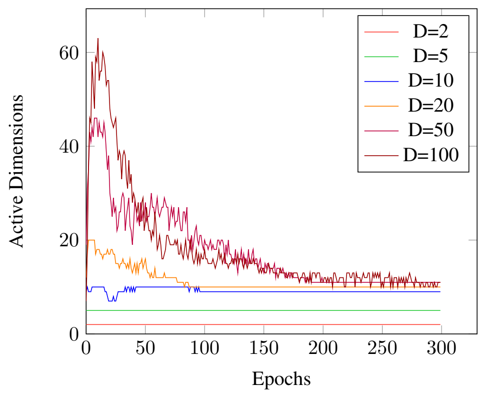
(a) Effect of Importance Weighted (IW)



(b) Effect of Batch Normalization (BN)



(c) Effect of Warm-Up



(d) Latent Dimensionality

3.1 Activity Metric

3.2 Effects of Implementation on Activity

3.2.1 Importance Weighting

3.2.2 Batch Normalization

3.2.3 Warm Up

3.2.4 Initial Latent Dimension

4 Latent Space Visualization

4.1 Data Reconstruction

4.2 Data Latent Transformation

4.3 Latent Hyperplane Lattice Generation

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.