



Technische Universität Berlin

Fakultät IV

Institute of Software Engineering and Theoretical Computer Science

Supervisor: Cesar Odeja

WS 2020/21

Stick-Breaking Variational Auto-encoders

Projects in Machine Learning and Artificial Intelligence

Santosh Dhirwani - 396807

Adu Matory - 404600

Konstantin Mehl - 368392

21.03.21

Contents

1. Introduction	4
2. Background: Variational Auto-encoders	4
2.1. Architecture	4
2.2. Stochastic Sampling/The Reparameterization Trick	5
2.3. Evidence Lower Bound (ELBO) loss function	7
3. Background: Stick-Breaking Variational Auto-encoders	8
3.1. Stochastic Gradient Variational Bayes	8
3.2. Stick-Breaking Processes	8
4. GEM Random Variables	9
4.1. Gamma Random Variables	9
4.2. Kumaraswamy Distribution	9
4.3. Gauss-Logit Parameterization	10
5. Stick-Breaking Variational Auto-encoders	10
6. Semi-supervised Variant	11
7. Implementation	12
7.1. Density Estimation	13
7.2. Discriminative Qualities	14
7.2.1. k-Nearest Neighbor (kNN) Classifier	14
7.2.2. t-Distributed Stochastic Neighbor Embedding (t-SNE)	15
7.3. Semi-supervised	15
8. Experimentation	16
8.1. Unsupervised	16
8.2. Semi-supervised	17
8.3. Limitations	17
8.4. Results	18
8.5. Observations	23
9. Future Work	24

10. Project Organization	25
10.1. Important Deadlines	25
10.2. Tasks	25
10.3. GitHub	26
10.4. Communication With The Team	27
10.5. Lessons Learned	29
11. Conclusions	29
12. Acknowledgements	30
References	31
Appendices	32
A. Challenges	32
B. Code Requirements	33
C. Experimental Setup	34
D. Code Repositories	35
E. Report Authorship	35

A variational auto-encoder is an unsupervised algorithm that performs dimensionality reduction, encoding input data features as a distribution with fewer parameters than features. In this report, we explore the utility of stick-breaking auto-encoders, a generalization to vanilla variational auto-encoders, which theoretically enables a well-regularized latent space. In this report, we also confirm the results of (Nalisnick & Smyth, 2017) and demonstrate that stick-breaking variational auto-encoders, in comparison to vanilla auto-encoders, show some increased performance, including better preservation of latent space class structure and differentiable control of their potentially infinite capacity.

1. Introduction

Leveraging the inferential capabilities of variational Bayesian methods, variational auto-encoders are a recent, yet quintessential example of dimensionality reduction and generative models. A variational auto-encoder is designed to learn a dataset’s latent space. In other words, it learns approximate causes of observed data or, rather, the parameters of approximate distribution from which we assume samples from the data-set were generated. Not only can a variational auto-encoder efficiently learn a data-set’s latent space, it can also efficiently generate meaningful, yet never-before-seen data that is similar to the observed data. Variational auto-encoders have application for data compression, brain decoding (Ozdenizci, Wang, Koike-Akino, & Erdogmus, 2019), image generation (Razavi, van den Oord, & Vinyals, 2019), and language modeling (Li et al., 2020). However, Gaussian variational auto-encoders might be limited in capturing certain features of observed data when learning when learning their latent space.

In this project, we aimed to confirm the results of (Nalisnick & Smyth, 2017) and demonstrate that stick-breaking variational auto-encoders, in comparison to vanilla auto-encoders, show some increased performance, including better preservation of latent space class structure and differentiable control of their potentially infinite capacity.

2. Background: Variational Auto-encoders

2.1. Architecture

An auto-encoder is an unsupervised algorithm that performs dimensionality reduction, encoding input data features with fewer parameters than features. The architecture of an auto-encoder consists of two, stacked multi-layer perceptrons. The first, an encoder (aka density network) finds a lower dimensional representation on an input. The second, a decoder (aka inference network), reconstructs the input from the lower-dimensional representation.

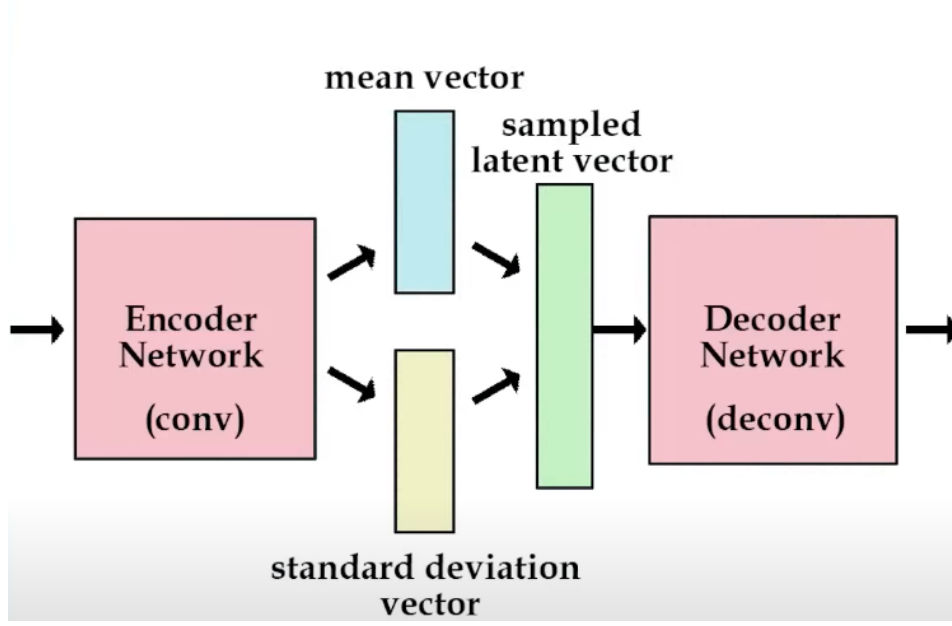


Figure 1: Gaussian Variational Auto-encoder (VAE)
(Insights, 2018)

In comparison, a variational auto-encoder encodes input data as a distribution. In the case of a vanilla variational auto-encoder, the encoder network learns to represent each data point by the two parameters of a Gaussian distribution, the mean and variance. The process is easily generalized to a multivariate Gaussian distribution. If the latent distribution is chosen to be multidimensional, the encoder network learns a mean vector and a variance vector (Figure 1).

2.2. Stochastic Sampling/The Reparameterization Trick

Variational autoencoders are also generative models, allowing the generation of new, stochastic samples by sampling from the latent space with the reparameterization trick. The feed-forward architecture is shown in (Figure 2). To generate samples from the latent space, a noise term ϵ is drawn from a noise distribution and applied to the learned parameters before they are decoded. In the case of a Gaussian variational auto-encoder we can reparametrize our mean vector μ and variance vector σ with our noise term ϵ to obtain the sampled latent vector z which is input to our decoder network.

$$z = \mu + \sigma \odot \epsilon \quad (1)$$

$$\epsilon \sim \mathcal{N}(0, 1) \quad (2)$$

Stochastic sampling with the reparametrization trick affords the network to learn the latent distribution's parameters with back-propagation, while simultaneously ensuring the latent space is regularized, or in other words, that it is both continuous and complete. Continuity describes the property that two points close together (according to some distance measure, e.g. Euclidean distance) in the latent space give two similar contents once decoded. Completeness describes the property that from a distribution, a point sampled from the latent distribution should give meaningful content once decoded. We more clearly see how these properties are achieved once we consider how loss is calculated for variational auto-encoders.

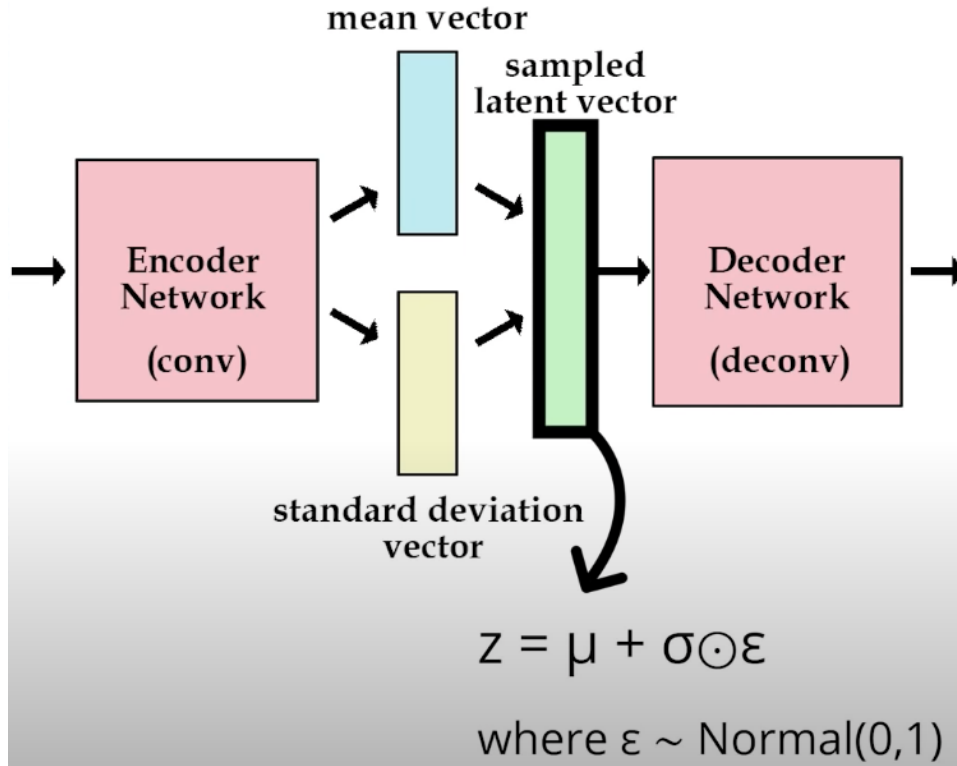


Figure 2: Gaussian Variational Auto-encoder (VAE)
(Insights, 2018)

2.3. Evidence Lower Bound (ELBO) loss function

The variational auto-encoder loss function is known as the evidence lower bound (ELBO) \mathcal{L} and has two components: a reconstruction loss and a regularization loss. Given prior parameters θ , learned parameters ϕ , and an input data sample x_i , and some S Monte Carlo samples \hat{z}_i from the latent space

$$\mathcal{L}(\theta, \phi; x_i) = \underbrace{D_{KL}(q_\phi(z_{i,s}|x_i)||p(z))}_{\text{regularization loss}} - \underbrace{\frac{1}{S} \sum_{s=1}^S \log p_\theta(x_i|\hat{z}_{i,s})}_{\text{reconstruction loss}} \quad (3)$$

with

$$D_{KL}(P||Q) = \int \log P(x) \frac{P(x)}{Q(x)} dx \quad (4)$$

The reconstruction loss is calculated as the negative log probability of the input given the reconstructions. Minimal loss is achieved when a sample is (a) the same as its reconstruction and (b) similar to reconstructions of other samples. Considering the reparametrization trick, which almost definitely yields noisy reconstructions, it becomes apparent that perfect reconstruction is impossible and a latent space that has continuity will minimize the reconstruction loss. Further, this term encourages overlapping of possible latent distributions in latent space, as a latent space that is complete will minimize the reconstruction loss.

The regularization loss is defined as the Kullback-Leibler divergence D_{KL} between the learned latent distribution and a chosen prior distribution. Minimal loss is achieved when the latent distribution is close as possible to the chosen prior distribution. Under the assumption that our data is generated from a highly complex, intractable distribution, a simple, smooth distribution (e.g. Gaussian, Gamma) is often chosen as the prior, which prevents over-fitting by constraining the complexity of the learned latent distribution.

These two components in the loss function work together to drive regularized learning of the latent space of the input data. We will later explore the implications this regularization has for the preservation of class structure and sampling from specific latent dimensions.

3. Background: Stick-Breaking Variational Auto-encoders

3.1. Stochastic Gradient Variational Bayes

Our Stick-Breaking Variational Auto-encoders will be trained similarly to the Variational Auto-encoders using Stochastic Gradient Variational Bayes (SGVB). To perform this training we will consider the ELBO loss function mentioned above that utilizes the Monte Carlo expectation to approximate the reconstruction error. In order to propagate gradients through our network, we will again need to make use of the reparametrization trick thus it is vital for our latent variable to be represented in a differentiable, non-centered parameterization (DNCP). In other words, our prior distribution must expose its parameters.

3.2. Stick-Breaking Processes

In this section, we will introduce stick-breaking processes whose weights will replace the Gaussian as our networks prior. A stick-breaking prior is defined as a random measure that is of the form $G(\cdot) = \sum_{k=1}^{\infty} \pi_k \delta_{\zeta_k}$. Here δ_{ζ_k} is some discrete measure whereas the π_k s are random weights independent of the base distribution. We chose π such that $0 \leq \pi_k \leq 1$ for each k and $\sum_k \pi_k = 1$ almost surely. It is easy to construct these weights using the stick-breaking process introduced by Sethuraman in 1994. To do so we first draw some weights $v_k \sim \text{Beta}(\alpha, \beta)$ which we then iteratively turn into π_k as follows.

$$\pi_k = \begin{cases} v_1, & \text{if } k = 1 \\ v_k \prod_{j < k} (1 - v_j), & \text{for } k > 1 \end{cases} \quad (5)$$

The special case that we draw v_k from $\text{Beta}(1, \alpha_0)$ results in the stick-breaking construction for the Dirichlet Process. Here we call the joint distribution over the infinite sequence of values of π the Griffiths, Engen and McCloskey distribution with concentration parameter α_0 . In short $(\pi_1, \pi_2, \dots) \sim \text{GEM}(\alpha_0)$

4. GEM Random Variables

In future, we will want to use SGVB to train our Variational Auto-encoder that uses weights of a stick-breaking process as its latent representation. This quickly runs into trouble as the Beta distribution does not admit a differentiable non-centered parameterization which we require. Thus we will need a surrogate distribution from which we can draw v . Luckily the authors of the primary literature (Nalisnick & Smyth, 2017) propose three viable options.

4.1. Gamma Random Variables

The first option was originally proposed in the initial SGVB paper. It suggests calculating the beta distribution as a composition of Gamma random variables as for $v = x/(x + y)$ with $x \sim \text{Gamma}(\alpha, 1)$ and $y \sim \text{Gamma}(\beta, 1)$ we know that $v \sim \text{Gamma}(\alpha, \beta)$. This however is not entirely non-problematic as the Gamma distribution does not have a DNCP representation with respect to its shape parameter. We can approximate the inverse CDF of the Gamma distribution like below for $\hat{u} \sim \text{Uniform}(0, 1)$.

$$F^{-1}(\hat{u}) \approx \frac{(\hat{u}a\Gamma(a))^{\frac{1}{a}}}{b} \quad (6)$$

However this approximation only holds if the shape parameter is close to zero and we will need to use a different approach once the shape parameter becomes equal to or larger than one. Hence we will look at some other possible methods to approximate the Beta.

4.2. Kumaraswamy Distribution

A promising posterior is the Kumaraswamy distribution whose density function is defined on the unit interval as

$$\text{Kumaraswamy}(x; a, b) = abx^{a-1}(1 - x^a)^{b-1} \quad (7)$$

for $x \in (0, 1)$ and $a, b > 0$. In the case that either a , b or both are equal to one the Kumaraswamy is equal to the Beta distribution and for similar parameters the Kumaraswamy appears as a Beta with higher entropy. The Kumaraswamy's closed form

inverse CDF naturally gives us a DNCP from which we can easily draw samples using

$$x \sim (1 - u^{\frac{1}{b}})^{\frac{1}{a}} \quad (8)$$

for $u \sim \text{Uniform}(0, 1)$. Additionally it is quite easy to approximate the KL-divergence of the Kumaraswamy and the Beta distribution in closed form.

4.3. Gauss-Logit Parameterization

The final method was inspired by the Probit Stick-Breaking Process. Here we simply draw a Gaussian which has a DNCP as we have already shown. The Gaussian is then squashed between zero and one using a function $g(\cdot)$. In this case using the logistic function $g(x) = 1/(1 + e^{-x})$ is simplest as it has a closed form solution.

5. Stick-Breaking Variational Auto-encoders

We have now presented all the requirements in order to define Stick-Breaking Variational Auto-encoders (SB-VAE). The main difference to standard Variational Auto-encoders is that we now draw our latent variables from the GEM distribution which turns the hidden representation into an infinite series of stick-breaking weights. Thus the reconstruction process remains largely unchanged and can be written as

$$\pi_i \sim \text{GEM}(\alpha_0), x_i \sim p_\theta(x_i|\pi_i) \quad (9)$$

Here π_i represents the stick-breaking weights drawn from the GEM distribution with concentration parameter α_0 and the likelihood model $p_\theta(x_i|\pi_i)$ describes the density network with parameters θ used for reconstruction. The inference process however requires some additional work. First we need our dense inference network to compute the parameters of K distributions. Using these we can then sample values $v_{i,k}$ from one of our three distributions that were presented in the previous section. Finally π_i can be computed in linear time using the iterative scheme detailed above.

$$\pi_i = \left(v_{i,1}, v_{i,2}(1 - v_{i,1}), \dots, \prod_{j=1}^{K-1} (1 - v_{i,j}) \right) \quad (10)$$

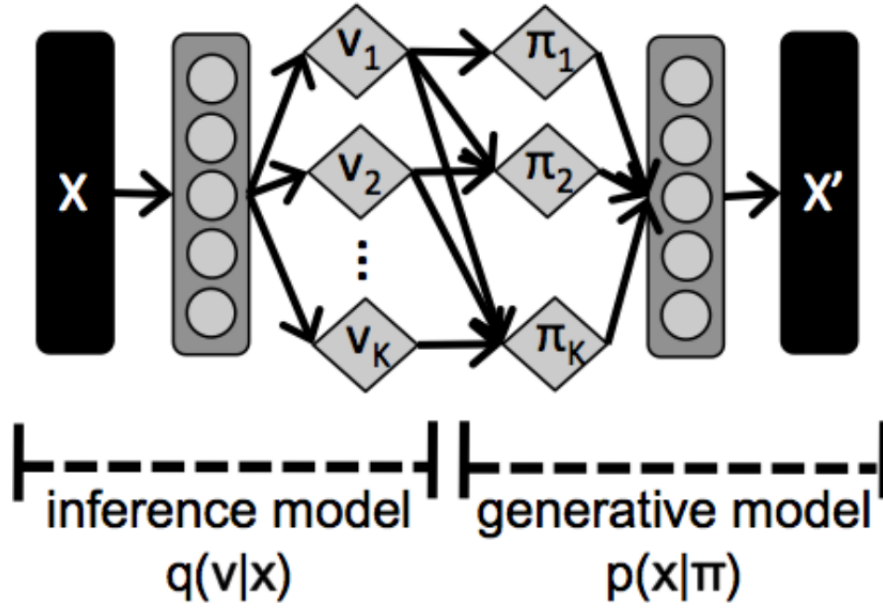


Figure 3: SB-VAE architecture

Here the only distinction that must be made is that the final fraction $v_{i,K}$ is always set to one in order to ensure that $\sum_k \pi_k = 1$ holds. We can then optimize the SB-VAE using standard SGVB techniques and minimize our loss function by adjusting the network parameters as needed.

6. Semi-supervised Variant

The VAE also has a semi-supervised relative known as the M2 model (Kingma, Rezende, Mohamed, & Welling, 2014a). An analogous approach to this M2 model was proposed by the author in the primary literature (Nalisnick & Smyth, 2017) known as the semi-supervised variant. In this section we explain the theory of the semi-supervised variant.

The feed-forward architecture can be seen in Figure 4. This model is similar to the variational auto-encoder, the change being that a class label is introduced as the second latent variable y_i that is marginalized when unobserved.

The semi-supervised classification experiments can be run with the semi-supervised deep generative model with stick-breaking latent variables. For our experiments, we used the MNIST data-set and reduced the number of labeled training examples to 10%, 5% and 1% to the total training set size.

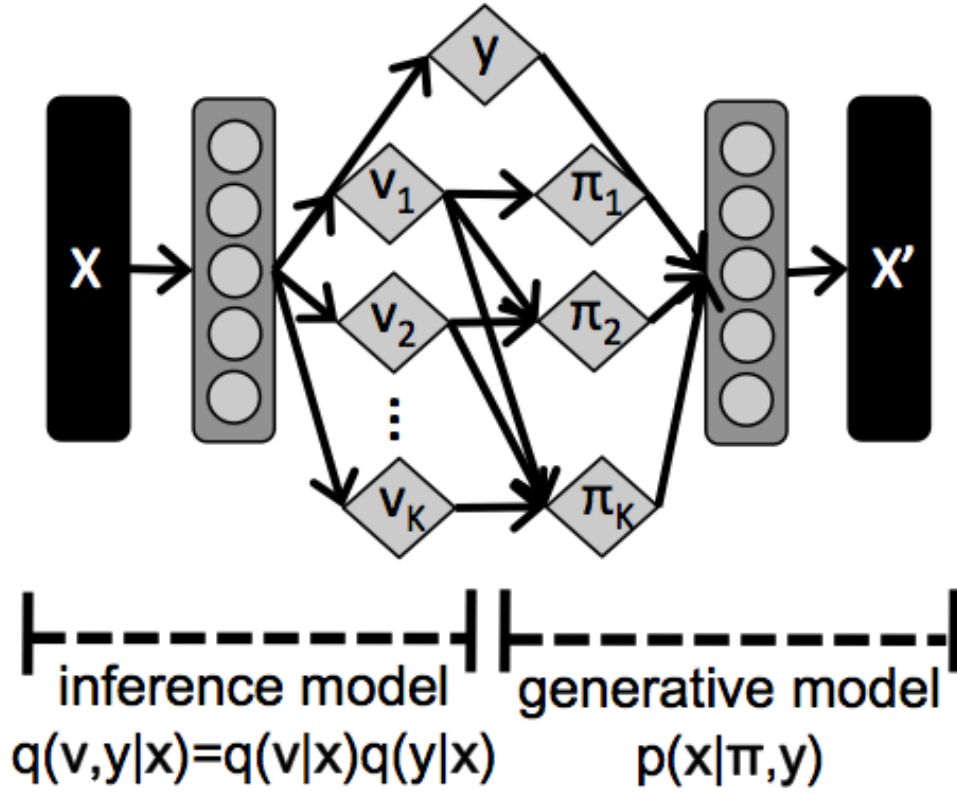


Figure 4: Semi-supervised Variant

7. Implementation

In this section we will explain in detail the implementation and the various decisions that we took during this project. The requisites of the implementation and instructions to run can be found in the appendix section A and B.

The pseudocode for the forward pass in our SB-VAE models, give some input data x , is seen in (Figure 5).

```

class StickBreakingVAE():
    def forward():
        param1, param2 = self.encode(x)
        samples = self.get_latent_distribution_samples(param1, param2)
        v = self.set_v_K_to_one(samples)
        pi = self.get_stick_segments(v)
        reconstructed_x = self.decode(pi)

        return reconstructed_x

    def ELBO_loss(self, reconstructed_x, x, param1, param2):
        reconstruction_loss = - sum(x * recon_x.log() + (1 - x) * (1 - recon_x).log())
        regularization_loss = kl_divergence(param1, param2, prior_param1, prior_param2)

        return reconstruction_loss + regularization_loss

```

Figure 5: Pseudo code: Model class for Stick-breaking VAE

The pseudocode for the training loop for our SB-VAE models is seen in (Figure 6).

```

model = StickBreakingVAE()

for i in range(epochs):
    random_idx = randint(len(data))
    mc_sample = data[random_idx]
    reconstructed_sample, param1, param2 = model(mc_sample)
    loss = model.ELBO_loss(reconstructed_sample, mc_sample, param1, param2)
    loss.backward()

```

Figure 6: Pseudo code: Training loop for Stick-breaking VAE

7.1. Density Estimation

Density estimation involves three sub-categories: (1) Data Reconstruction Ability, (2) Marginal likelihood and (3) Dirichlet’s increasing dimensionality. In this sub-section, we talk about implementing these two categories.

The first step in analyzing the behavior of the three SB-VAE parameterizations and observe how they compare to the Gaussian VAE, it by examining their data reconstruction ability and preserve the class structure. The complete implementation and all optimization details can be found in the GitHub repositories. The same architecture and optimization hyper-parameters were used in all experiments to isolate the effects of Gaussian versus stick-breaking latent variables.

The marginal likelihood for each model is reported by using 100 samples via the MC

approximation

$$\log p(x_i) \approx \log \frac{1}{S} \sum_s p(x_i | \hat{z}_{i,s}) p(\hat{z}_{i,s}) / q(\hat{z}_{i,s}) \quad (11)$$

In the case of Dirichlet’s increasing dimensionality, MNIST digits are drawn from the SB-VAE by sampling from the prior where $v_k \text{Beta}(1, 5)$, and Gauss VAE samples are shown for comparison. The SB-VAE samples using all fifty dimensions will also be observed and reported.

7.2. Discriminative Qualities

7.2.1. k-Nearest Neighbor (kNN) Classifier

k-Nearest Neighbor (kNN) classifiers were implemented using scikit-learn package (Pedregosa et al., 2011). With these classifiers, we tested the discriminative capacity of the latent space of trained models. We used mean accuracy on the given test data and labels as a performance metric.

Briefly, a kNN classifier comprises a three-step algorithm: (1) calculate distance, (2) find the closest neighbors, and (3) vote for the labels. First, for each data point x , the k closest data points (aka nearest neighbors) are found and the point is classified by the majority ‘vote’ of its k neighbors. Each nearest neighbor votes for its class and the class with the most votes is taken as the class prediction for x . Closeness is determined by a distance measure, such as Euclidean distance.

A kNN classifier is a lazy learner because it does not learn a discriminative function from the training data. Rather, it memorizes the training data set. The logistic regression algorithm is a good example for this case, as it learns the model weights during the training time. In contrast, a kNN classifier basically stores a training set during training time. It is a simple, instance-based algorithm.

The number of neighbors is the core deciding factor for a kNN classifier’s performance. Choosing the right value of k is called parameter tuning. If the value of k is too small, the bias it is based on is too noisy, and the resulting classification may be skewed. If k is too big then it will take a very long time to learn and one might run into process or resources issues. Ideally, it is recommended to choose the value of k as the square root of N , where N is the total number of data points. An odd value is often selected for k to

avoid confusion between 2 classes of data.

We evaluated performance in the following way. First, samples from both the train and test sets were encoded by the trained VAE. The latent samples from the train set were then used to fit the classifier and those from the test set were used to evaluate performance. Each trained model’s latent space was evaluated on three kNN classifiers, each with a different values of k : $k = 3, 5, 10$.

7.2.2. t-Distributed Stochastic Neighbor Embedding (t-SNE)

The t-SNE algorithm was also implemented using the scikit-learn package (Pedregosa et al., 2011). This implementation was done to further experiment with the discriminative qualities of the models and support our findings.

The t-SNE algorithm, a non-linear dimensionality reduction algorithm, detects patterns in data by finding observed clusters based on data point similarity with different features (Violante, 2018). However, it is a dimensionality reduction algorithm and not a clustering algorithm. The input features are no longer recognizable since the multi-dimensional data is mapped to a lower-dimensional space. As a result, you can’t draw any conclusions solely based on t-output. SNE’s As a result, it’s mostly a data exploration and visualization method. It is capable of explicitly categorizing the different types of inputs and avoids the ”crowding problem” that occurs when using Principal Component Analysis (PCA). A popular example of t-SNE implementation is facial expression recognition.

In short, the t-SNE algorithm determines a resemblance measure between pairs of instances in high and low dimensional space. It then uses a cost function to try to maximize these two similarity measures.

7.3. Semi-supervised

In this analogous approach for the semi-supervised variant of the SB-VAE, the second latent variable y_i , that represents a class label, has a categorical distribution

$$q_\phi(y_i|x_i) = Cat(y|g_y(x_i)) \quad (12)$$

where g_y is a non-linear function of the inference network (Nalisnick & Smyth, 2017). Although the distribution of ”y” is written as independent of ”z”, within the inference

network, the two share parameters and thus act to regularize one another. As in the finite dimensional version, we assume the same factorization of the posterior and use the same aims (Kingma, Rezende, Mohamed, & Welling, 2014b). Because y_i is present in some but not all observations, semi-supervised DGMs must be trained with various goals depending on whether or not the label is present. If the label is present, we optimize according to (Kingma et al., 2014b).

$$\tilde{J}(\theta, \phi; x_i, y_i) = \frac{1}{S} \sum_{s=1}^S \log p_{\theta}(x_i | \pi_{i,s}, y_i) - KL(q_{\phi}(\pi_i | x_i) || p(\pi_i; \alpha_0)) + \log q_{\phi}(y_i | x_i) \quad (13)$$

where $\log q_{\phi}(y_i | x_i)$ is the likelihood of the label. If the label is missing, we optimize the following

$$\tilde{J}(\theta, \phi; x_i) = \frac{1}{S} \sum_{s=1}^S \sum_{y_j} q_{\phi}(y_j | x_i) [\log p_{\theta}(x_i | \pi_{i,s}, y_j)] + \mathbb{H}[q_{\phi}(y_i | x_i)] - KL(q_{\phi}(\pi_i | x_i) || p(\pi_i; \alpha_0)) \quad (14)$$

where $\mathbb{H}[q_{\phi}(y_i | x_i)]$ is the entropy of the variational distribution of "y".

8. Experimentation

In this section we describe the experiments that were performed after training the four models. Here, four sets of experiments were performed. The first three sets are unsupervised experiments, namely Marginal likelihood, Density Estimation, and Discriminative qualities using kNN and t-SNE algorithms. The fourth set is semi-supervised classification. All of these experiments were performed on the MNIST data-set.

8.1. Unsupervised

In the experiments performed there are two categories, name unsupervised and semi-supervised. This section explains the unsupervised experiments conducted in this project. The unsupervised category includes three sub-categories: (1) Data Reconstruction Ability,

(2) Density Estimation and (3) Discriminative Qualities.

In order to compare the data reconstruction ability, we will investigate the behavior of the three SB-VAE parameterizations and compare them to the Gaussian VAE. The only distinction was in the prior: $p(z) = N(0, 1)$ for Gaussian latent variables and $p(v) = \text{Beta}(1, \alpha_0)$ (Nalisnick & Smyth, 2017) for stick-breaking latent variables. The concentration parameter α_0 was cross validated over the range of $\alpha_0 \in 1, 3, 5, 8$. As mentioned earlier, we are only working with the MNIST data-set in this project, the cross-validation led to choosing the value of $\alpha_0 = 5$. Both models, inference and generative, contain one hidden layer of 500 units. The truncation level of SB-VAE is set to $K = 25$, so it can use the same number of latent variables as the Gauss VAE.

Density Estimation experiments are needed to show the optimization progress of each model. In this part, we will report the test expected reconstruction error (the first term in the ELBO) vs the training progress (number of epochs).

8.2. Semi-supervised

We also conducted semi-supervised classification, replicating and expanding on the experiments described in the semi-supervised DGMs paper (Kingma et al., 2014b). The number of classified training examples was decreased to 10%, 5%, and 1% of the total training set size using the MNIST data-set. As a consequence of the complete removal of labels at random, class imbalance was almost inevitably added. We matched DGMs with stick-breaking (SB-DGM) and Gaussian (Gauss-DGM) latent variables against each other and a baseline k-Nearest Neighbors classifier with the value of k as 5, in a similar way to what we did in the set of unsupervised experiments. The dimensionality / truncation degree of the latent variable was set to 50. One secret layer of 500 hidden units is used in the MNIST networks. Identity feature skip-connections exist in the last three secret layers. The value of the cross-validation parameter α_0 was chosen as 5.

8.3. Limitations

As students, we have limited access to resources, so we could not run experiments on XL AWS EC2 instances, as suggested in the original stick-breaking paper (Nalisnick & Smyth, 2017). All of the experiments conducted in this project were implemented on a bare metal laptop. This limited the capability to train the models for more than 500

epochs. In order to reach 500 epochs on a local setup, it took roughly 16 hours. Due to the fixed length of a semester, we could only conduct experiments on the MNIST data-set, as suggested by our supervisor. The other options to compare are Frey Faces and MNIST + rot data-set.

8.4. Results

In this section we present and discuss results based on the experiments and experimental setup presented in the above sub sections.

- **Data Reconstruction Ability:** Figure 7 shows the results of the predicted reconstruction error (i.e. the first term in the ELBO) vs training success (epochs) for MNIST data-set, for each model’s optimization progress. Both models optimize in a similar way, with the exception that the SB-VAE learns at a significantly slower rate for all parameterizations compared to the Gauss VAE. The Gauss-Logit VAE learns the fastest amongst all models.

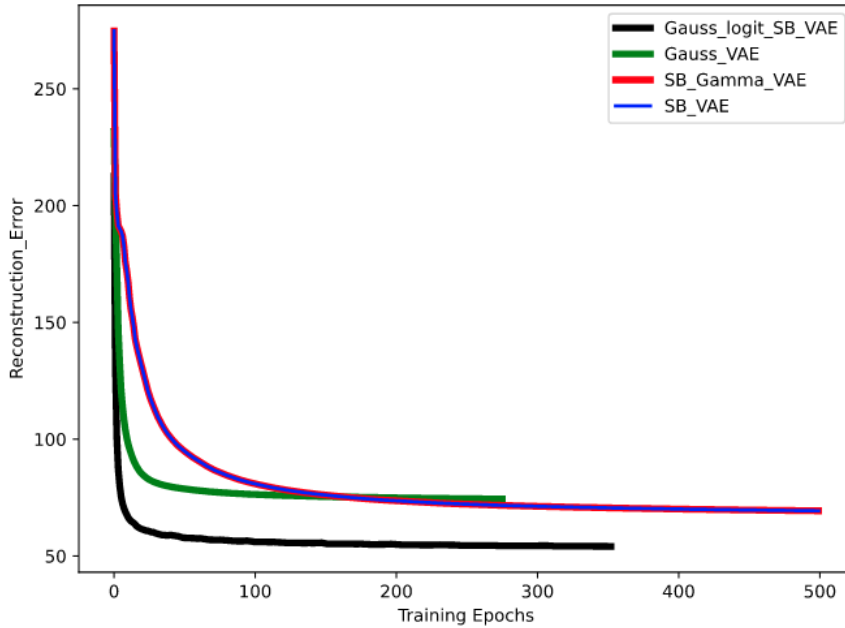


Figure 7: Reconstruction Error vs Training Epoch on MNIST data-set

- **Density Estimation:**

The density estimation involves two parts: marginal likelihood of each model and the nature of the latent variables by sampling from Dirichlet’s increasing dimensionality (Ferguson, 1973).

The results of the marginal likelihood experiments are displayed in Table 1 below. The column $-\log p(x_i)$ shows the likelihood and you can also see the best epoch where this likelihood was achieved, along with the training time spent in minutes. As shown in Table 1, Gauss VAE has a better likelihood than Kumaraswamy SB-VAE (~ 74 vs ~ 86). Among the three stick-breaking implementations, Logit SB-VAE outperforms both Kumaraswamy SB-VAE and Gamma SB-VAE on the MNIST data-set.

Model	$-\log p(x_i)$	Best epoch	Training time (minutes)
Gauss VAE	-74.5738	244	111.89
Kumar SB-VAE	-86.0617	498	298.55
Logit SB-VAE	-54.1776	321	149.25
Gamma SB-VAE	-69.3720	493	229.82

Table 1: Marginal Likelihood Results

Figure 8 shows the results of the experiments conducted towards Dirichlet Dimensionality. The bottom block shows SB-VAE samples that use all fifty dimensions of the truncated posterior. In the two top-left columns, samples from Dirichlet’s confined to a subset of the dimensions are shown to see if the latent features are focusing on lower-dimensional simplices. It happens indeed that adding a latent variable results in samples, that are significantly different but still coherent. If you look closely at the figure, you can observe that for example, the sixth dimension seems to capture the "1" class, the second and third dimensions seem to capture the "7" and "9" classes, whereas the eighth dimension seems to capture the "3" class. The seventh dimension models notably thick digits. In this section, we achieved very similar results to the primary literature.

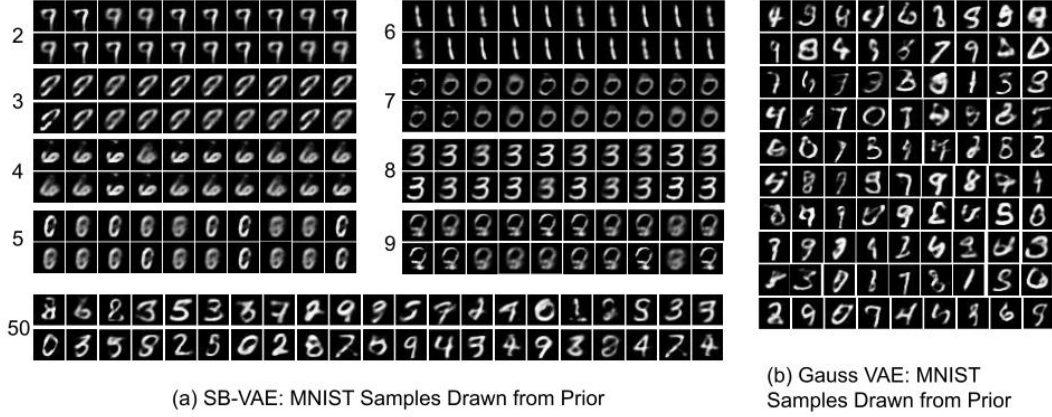


Figure 8: Samples from the SB-VAE and Gauss VAE trained on MNIST.

- Discriminative Qualities:** The maximum throughput found is 77.7 with a block interval of 1 second and a block gas limit of 9.99 million. The execution time was more than 13 hours. As we can see in figure 7, our tool again manages to find the peak for each interval and stop after stagnation or drop of performance. In this experiment we can notice that, apart of having less throughput values than in the single region experiment, the throughput results are more chaotic and less "stable". The multi region extra latency may be the cause of these irregular results. In figure 8 we can again see the trend of the single region, the more we increase the block interval the more steps it takes to reach the peak.

kNN Classifier:

In order to assess the discriminative qualities of the models' latent spaces, we used the kNN algorithm that we implemented and ran the classifier on sampled MNIST latent variables. We conducted the experiments for three different k-values: 3, 5 and 10 and for three types: Kumar SB-VAE, Gauss VAE and Raw Pixels. You can see in the table below how each model performs for the different k values. The SB-VAE outperforms the Gauss VAE at all k values, meaning that although the Gauss VAE converges to a higher probability, the SB-VAE's latent space captures the class structure much better.

Model	k = 3	k = 5	k = 10
Kumar SB-VAE	5.63	5.38	5.61
Gauss VAE	23.73	17.66	13.58
Raw Pixels	3.33	3.39	3.56

t-SNE Algorithm: We further assessed the discriminative qualities of the SB-VAE’s latent space by implementing a t-SNE algorithm. In figure 9, the results of a Gauss VAE are shown on the MNIST data-set that we implemented. The figure 10 shows the results of SB-VAE on the MNIST data-set that was implemented as explained in the previous section. If we compare these two figures, it is clearly visible that the digit classes denoted by color in the stick-breaking latent space are clustered with noticeably more cohesion and separation, compared to the Gauss VAE.

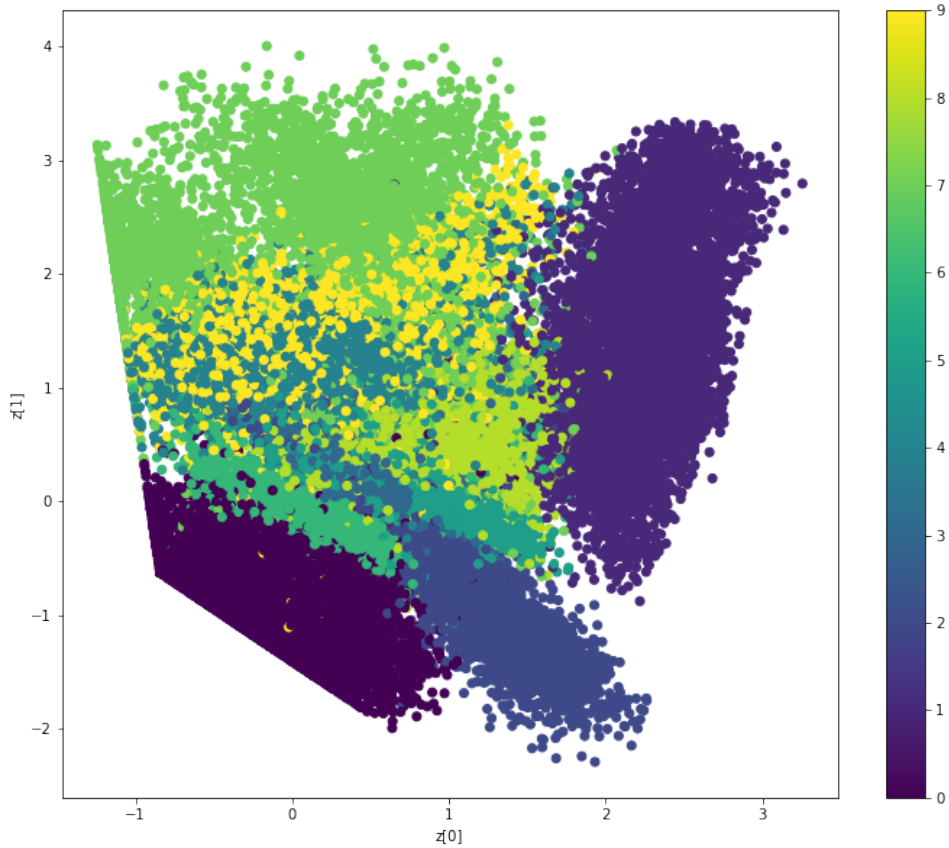


Figure 9: MNIST Gauss VAE

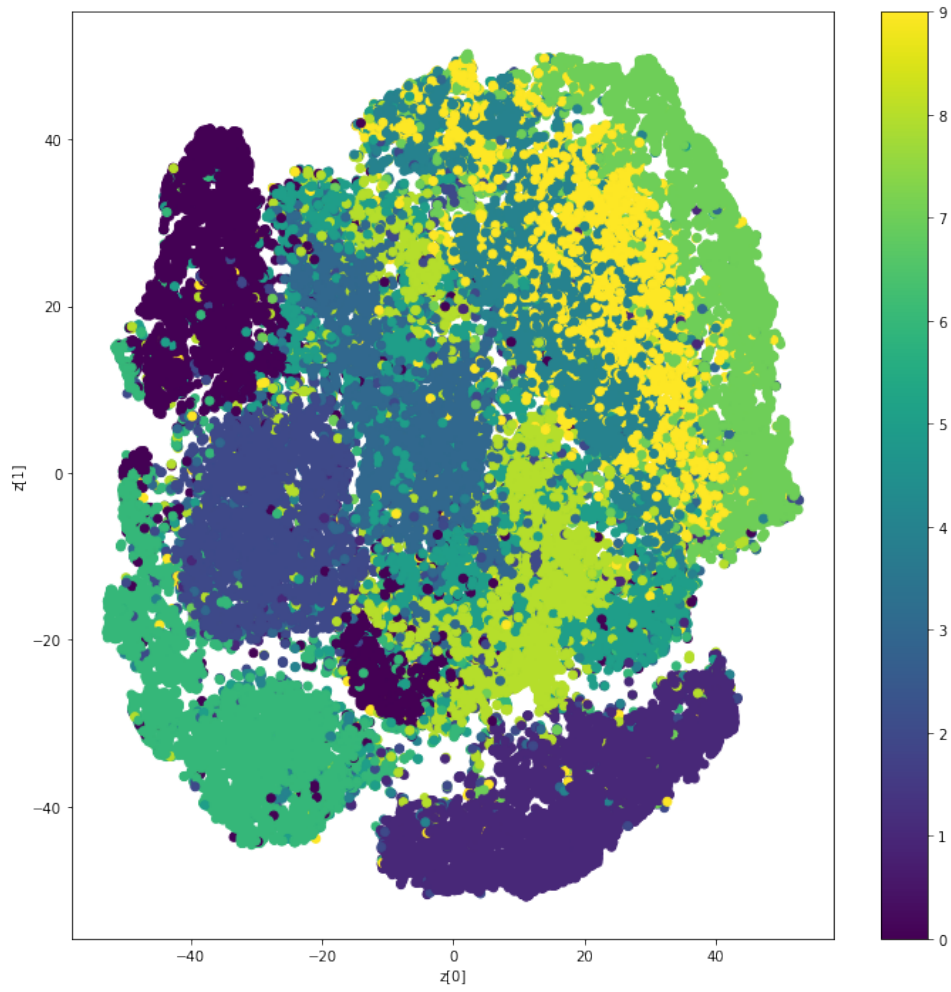


Figure 10: MNIST SB-VAE

- **Semi-supervised Classification:**

Model	MNIST (N = 45,000)		
	10%	5%	1%
SB-DGM	5.25	5.73	7.44
Gauss-DGM	4.03	4.3	9.5
kNN	6.0	8.0	19.0

Table 2: Percent error with 10%, 5% and 1% labels present for training

8.5. Observations

After analysing the results of our evaluation experiments we saw the following observations:

- **Gauss-Logit SB-VAE has better likelihood than all stick-breaking implementations:** The results achieved by us show that the Gaussian VAE has a better marginal likelihood compared to the Kumaraswamy SB-VAE. In our experimental setup, among the three stick-breaking parameterizations, the Gauss-Logit SB-VAE showed the best results. This was a bit surprising for us because the Gauss-Logit parameterization is restricted. The results achieved for the Gamma SB-VAE were expected because of the flaw that it is approximate.
- **Gauss-Logit SB-VAE learns faster compared to other models:** As mentioned earlier, we trained the models until 500 epochs and the reason for not going further in this part of experimentation was that there was no significant change observed after 200 epochs. The Kumar SB-VAE and the Gamma VAE learn at a slightly slower pace compared to others and this was expected because the recursive definition of the latent variables likely causes coupled gradients.
- **Kumaraswamy SB-VAE performs best for all kNN choices of k:** Kumaraswamy SB-VAE performs better than Gauss VAE at all choices of k which leads us to conclude that although the Gauss VAE converges to a better likelihood, the Kumaraswamy SB-VAE's latent space captures the class structure better. The results for Raw Pixels were achieved as expected, as the class structure is usually more apparent in low-sparsity data before it is compressed.
- **SB-DGM performs best only when 1% of MNIST labels are available:** The results we achieved in this part of experimentation were very close to those achieved in the primary literature (Nalisnick & Smyth, 2017). SB-DGM performs much better when only 1% MNIST labels are available. On the other hand, Gauss DGM achieves a superior error rate only on the easiest tasks which are on MNIST data-set with 5% and 10% labelled data.

In summary, we successfully implemented a stick-breaking variational auto-encoder, not only in Theano, but also in PyTorch and Keras. We were also able to implement two more parameterizations along with the Gaussian VAE. Furthermore, experiments were conducted successfully in the defined experimental setup. However the training time is quite long as the setup is on a bare metal laptop. The experimentation can be performed to achieve better results on a stronger setup and we are also aware that the

implementations can be further improved to be more robust and intuitive too.

9. Future Work

We are aware of the fact that our contributions and implementation are far from perfect. Nevertheless, in this section we outline several improvements that can be implemented in future. The set of improvements that can be done are the following:

- **Implement further models:** If we had more than one semester i.e., six months to work on this project, the first thing that would be exciting for us to experiment with, is models with convolutional layers and multiple stochastic layers. As convolutional layers are the major building blocks of a convolutional neural network, this will allow us to further develop and conduct experiments in this domain.
- **Full Dirichlet processes:** All of the ideas discussed in the primary paper (Nalisnick & Smyth, 2017) and re-implemented by us in this project, are not only immediately useful in their current form, but they are also a crucial first step towards combining black box variational inference and Bayesian non-parametrics, resulting in scalable models with differentiable power regulation. Applying Stochastic Gradient Variational Bayes (SGVB) to complete Dirichlet processes (Teh, 2010) with non-trivial base steps, in particular, is a fascinating next step.
- **Neural Networks:** Furthermore, differentiable stick-breaking has the ability to improve the dynamism and adaptivity of neural networks in a probabilistically principled way, a topic of recent concern (Graves, 2017).
- **Scalable models:** After working on this project for roughly six months, we agree that Kumaraswamy distribution is the first step towards building scalable models that have differentiable control of their capacity.
- **Experimentation with cloud services:** As mentioned earlier in this report, we conducted experiments on a bare metal laptop with a graphic card. It will be very interesting to see how the models perform when you train them for more than 1000 epochs on an XL AWS EC2 instance for example.

10. Project Organization

In this section we describe how exactly we organized our work as a team in order to fulfill the goals of this project. It includes important deadlines, task organization and scheduling, code organization and team communication.

10.1. Important Deadlines

Kickoff Meeting (11.11.2020):

In the Kickoff Meeting, all members introduced themselves and described their relevant experiences. The meeting was very helpful to structure the literature review process and also to plan the meetings in future.

Weekly Team Meeting (every Friday at 18.00):

We scheduled a weekly team meeting on Fridays. In this meeting we discussed the current state of the project, sharing obstacles that we found, planning for the next week's milestones, and getting input from the supervisor. All three team members attended this meeting.

Midterm Presentation (12.01.2021):

In the middle of January we presented our work to all the supervisors and other project groups. After presenting our achievements we shared what would be the next steps and the goal of this project.

Final presentation (05.03.2021):

The final presentation was held in the beginning of March. In this meeting, we presented all of our experiments with the four models, and the three implementations that we completed successfully. The findings were discussed in detail along with the challenges faced and the future work that we suggested.

10.2. Tasks

On reviewing the literature, all team members managed to build up a very good understanding of the topic and the problem statements that were to be tackled. The author of

the primary paper (Nalisnick & Smyth, 2017) implemented the SB-VAE using Theano and Python2. As a team, we made the decision to follow the same technology stack. However, after the mid-term presentation, we took a step back and tried to rethink the technology stack. As Theano and Python2 are outdated at the time we are working on this project, it is also important to provide results with a more modern setup. Therefore, we decided to implement everything using Python3 in Pytorch and Keras. It turned out to be a great decision because one of the team members was comfortable with python2, one was comfortable with Pytorch and one with Keras.

10.3. GitHub

We used GitHub for maintaining the source code and version control. It turned out to be a good idea because all three of us had previous experience with GitHub and were comfortable in using it. The tasks that came out of the team meetings, were listed on GitHub as issues and assigned to a team member.

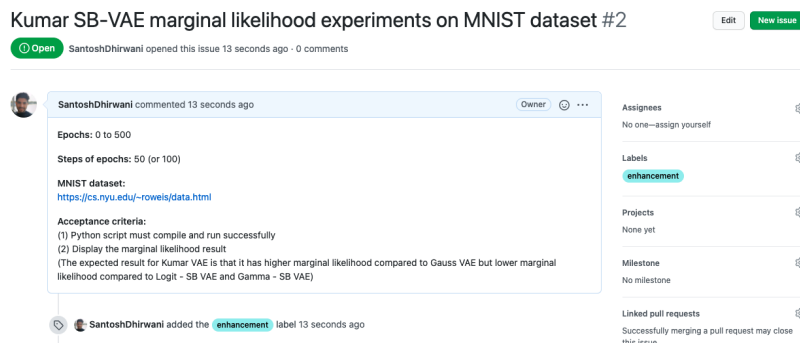


Figure 11: Sample list of tasks on Issues page.

We discussed each and every task in detail and set an acceptance criteria to make sure that the task is aligned with our final goal. Each team member had to create their own branch for every new issue, to keep the master branch clean from unfinished code. The member would then create a pull request when he is done and assign the remaining two team members as reviewers.

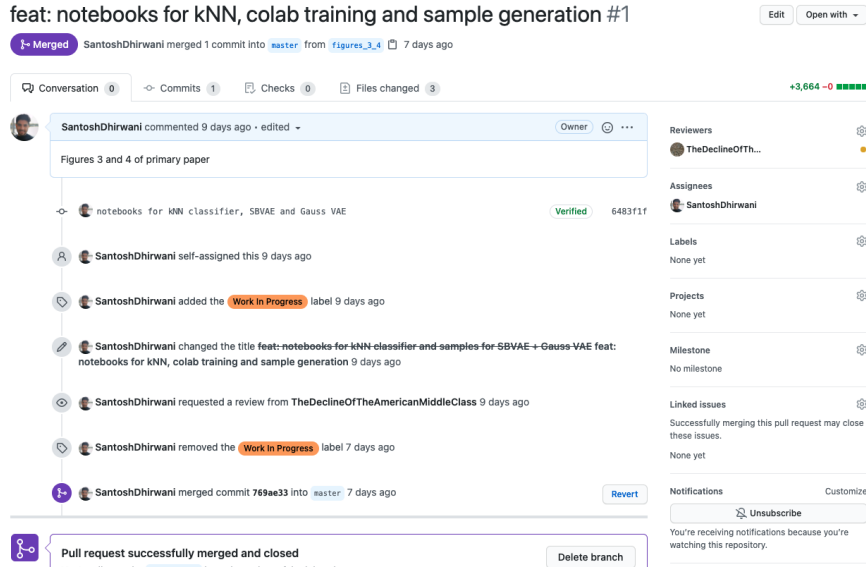


Figure 12: Sample Pull Request.

Pull requests had to be approved by at least one developer before it could be merged. To review the pull request, team members needed to follow the instructions mentioned inside the relevant pull request, to correctly test it. If there were any changes required, it was requested during the review and also questions were asked as comments if something was unclear. Once the pull request was approved, the member would then merge the branch into master, after which that branch was automatically deleted. Also, before opening a new pull request, we always fetched the latest version of the master branch to avoid merge conflicts.

10.4. Communication With The Team

To manage our communication, we decided to use Slack as the main platform. In slack we have several channels to communicate specific matters :

- GitHub: this channel is connected with our GitHub repository to send notification about new/closed issues and new/closed pull requests.
- General: used for communicating with all team members including the supervisor.
- Random: used for sharing related literature that we found.
- Unreported GitHub features: this channel was used to share the features that were not reported in the primary paper. (Nalisnick & Smyth, 2017)

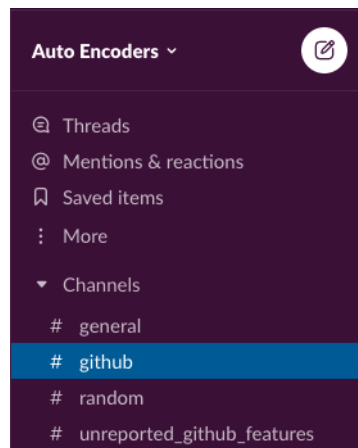


Figure 13: Slack Channels.

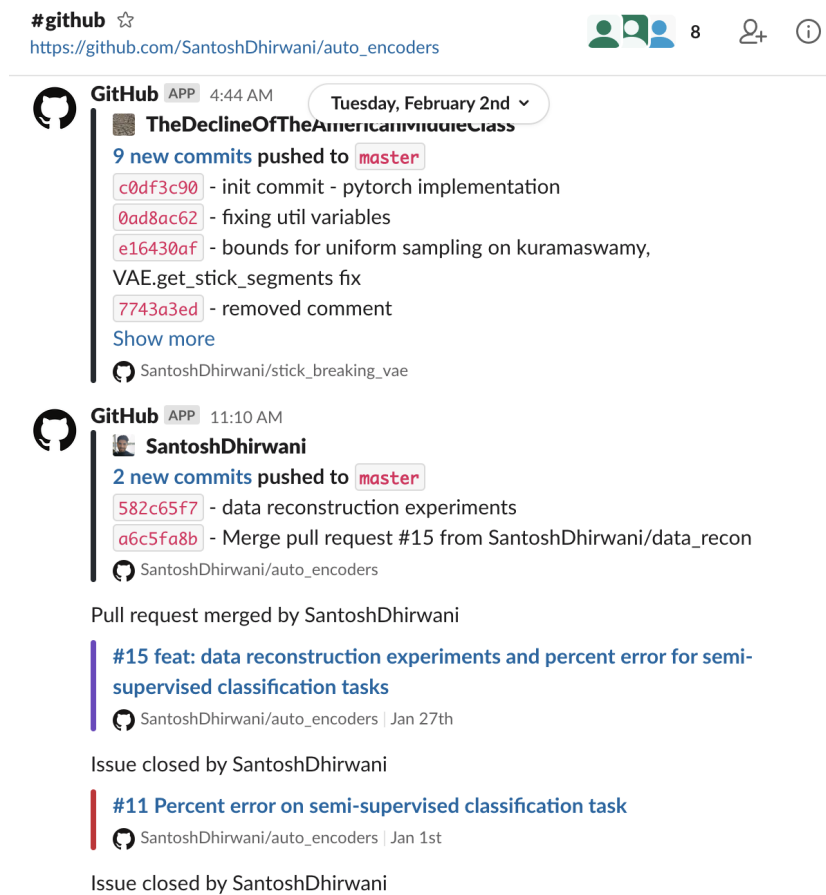


Figure 14: GitHub-Slack Integration.

10.5. Lessons Learned

From this project, we have learned things about how to work in a team. However, there were also some small challenges that we faced in the beginning. We found that coordinating amongst the team is quite difficult if we are not communicating well. Since each team member has their own personal schedule, it was hard to expect us to work together at the same time. Also, initially, sometimes we found out some of us were working on the same task. Therefore, we decided to discuss these coordination issues and solve them on our Slack channel. We centralized our communication platform and we were able to communicate with team members and the supervisor easily.

Moreover defining a task was not an easy job. There has been a long discussion to set priorities for tasks, the whole team needed to clearly agree which tasks were the priority. Discussing our progress helped us to understand the current state of each member. If there was some misunderstanding we could point out the issue and solve it to make a clearer expectation of the task.

11. Conclusions

We aptly demonstrated the utility of using stick-breaking process to expand the capacity of vanilla Gaussian auto-encoders and better identify structure in estimate causes of our data. During our experiments, we observed the Kumaraswamy SB-VAE has better marginal likelihood than all stick-breaking implementations, performs best for all kNN choices of “k”, shows more distinct results for t-SNE projections, and its deep generative model performs best only when 1% of MNIST labels are available. However, the Gauss-Logit SB-VAE learns faster compared to the other models. We discussed what organization methods we used during the project and we envisioned multiple future applications for this approach to dimensionality reduction and deep generative models. We look forward to applying what we’ve learned to our own fields of interest, from mathematics research to analysis of neural data to app development.

12. Acknowledgements

We would like to thank our supervisors, Cesar Ojeda, Theo Galy-Fajou, and Prof. Dr. Manfred Opper, for providing us with stimulating material, instructions, and a platform to grow our interest in machine learning. We would also like to thank all the students in the class for their engaging presentations, comments, and questions. This project was not only edifying, but a wonderful opportunity to collaborate with our colleagues, an especially rare experience considering the unprecedented social restrictions caused by the pandemic. Thank you.

References

- Ferguson, T. S. (1973). A Bayesian Analysis of Some Nonparametric Problems. *The Annals of Statistics*, 1(2), 209 – 230. Retrieved from <https://doi.org/10.1214/aos/1176342360> doi: 10.1214/aos/1176342360
- Graves, A. (2017). *Adaptive computation time for recurrent neural networks*.
- Insights, A. (2018, February 25). *Variational autoencoders*. Retrieved from <https://www.youtube.com/watch?v=9zKuYvjFFS8>
- Kingma, D. P., Rezende, D. J., Mohamed, S., & Welling, M. (2014a). Semi-supervised learning with deep generative models. *CoRR*, abs/1406.5298. Retrieved from <http://arxiv.org/abs/1406.5298>
- Kingma, D. P., Rezende, D. J., Mohamed, S., & Welling, M. (2014b). *Semi-supervised learning with deep generative models*.
- Li, C., Gao, X., Li, Y., Peng, B., Li, X., Zhang, Y., & Gao, J. (2020). Optimus: Organizing Sentences via Pre-trained Modeling of a Latent Space. Retrieved from <http://arxiv.org/abs/2004.04092>
- Nalisnick, E., & Smyth, P. (2017). *Stick-breaking variational autoencoders*.
- Ozdenizci, O., Wang, Y., Koike-Akino, T., & Erdogmus, D. (2019). Transfer Learning in Brain-Computer Interfaces with Adversarial Variational Autoencoders. *2019 9th International IEEE/EMBS Conference on Neural Engineering (NER)*, 207–210. Retrieved from <http://arxiv.org/abs/1812.06857>
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Razavi, A., van den Oord, A., & Vinyals, O. (2019). Generating Diverse High-Fidelity Images with VQ-VAE-2. Retrieved from <http://arxiv.org/abs/1906.00446>
- Teh, Y. W. (2010). Dirichlet process. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of machine learning* (pp. 280–287). Boston, MA: Springer US. Retrieved from https://doi.org/10.1007/978-0-387-30164-8_219 doi: 10.1007/978-0-387-30164-8_219
- Violante, A. (2018). An introduction to t-sne with python example. *Towards Data Science*. Retrieved from <https://towardsdatascience.com/an-introduction-to-t-sne-with-python-example-5a3a293108d1>

Appendices

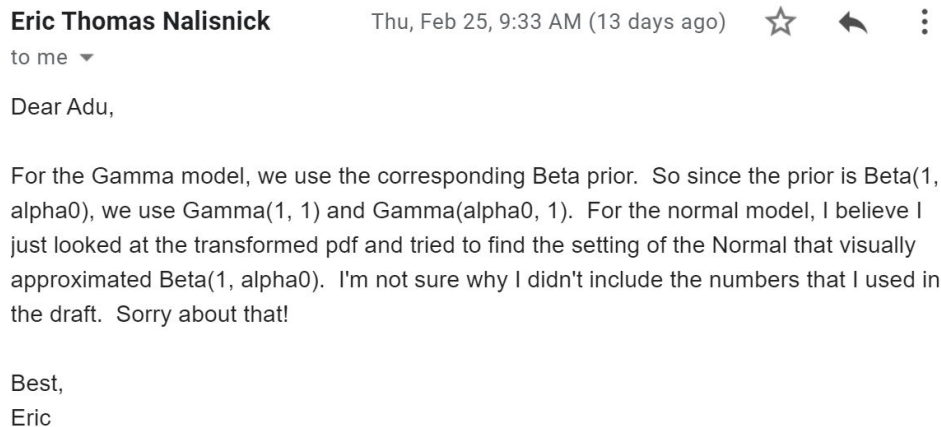
A. Challenges

There were a few challenges associated with implementing and experimenting with the stick-breaking variational autoencoders. Notably, the experimentation was very time-consuming, especially when training the models. In particular, the Taylor series approximation for the Kumaraswamy-Beta KL-divergence was computationally heavy.

Deciding the best deep-learning back-end to use was difficult. Ultimately, each member of the team implemented models in their deep-learning back-end of choice. We may have been able to run more experiments if we had taken a better approach to delegation of tasks, so as to avoid redundancy. However, given we were able to successfully train models and run experiments on all implementations given the project's time limitations, our approach was suitable.

NaN values sometimes occurred in our loss calculation. We identified two causes:

- When a randomly drawn first value for a sample v_k is small (< 0.01), sum-to-1 errors can arise in the stick-breaking process. Restarting training from a previously saved checkpoint often allowed training to continue unencumbered.
- Using ReLu as the final activation function in our encoder network sometimes caused vanishing weights. This problem was solved by using SoftPlus activation function, whose output only approaches zero for very large negative values, so NaN gradients are almost never back-propagated during gradient descent. Further, we implemented gradient clipping to avoid any potential of exploding gradients.



Eric Thomas Nalisnick Thu, Feb 25, 9:33 AM (13 days ago) ☆ ↩ ⋮
to me ▼

Dear Adu,

For the Gamma model, we use the corresponding Beta prior. So since the prior is $\text{Beta}(1, \alpha_0)$, we use $\text{Gamma}(1, 1)$ and $\text{Gamma}(\alpha_0, 1)$. For the normal model, I believe I just looked at the transformed pdf and tried to find the setting of the Normal that visually approximated $\text{Beta}(1, \alpha_0)$. I'm not sure why I didn't include the numbers that I used in the draft. Sorry about that!

Best,
Eric

Figure 15: Clarification with Authors

Further, not all training hyper-parameters, prior parameters, architectural choices, and KL divergence calculation methods were reported in the original paper. We discovered the architecture and hyper-parameters by combing through the project GitHub referenced in the paper, however, for the Gauss-Logit and Gamma distributions, we only determined the necessary prior parameters and the calculation of KL divergence by writing the authors with a request for clarification (Figure 15). Though we ultimately succeeded in filling in the blanks, the process of understanding which pieces we were missing took time and critical thinking.

B. Code Requirements

The following packages are required for each implementation of our Stick-breaking variational autoencoders.

- PyTorch implementation: Python 3 and packages: torch, matplotlib, Pillow, numpy, pandas, xarray, torchvision, botorch, scikit-learn
- Keras Implementation: Python 3, keras, tensorflow, matplotlib, scikit-learn
- Theano Implementation: Python 2, theano, cPickle, gzip, tarfile, zipfile, urllib

C. Experimental Setup

evaluation section The configurations for all experiments run on our three implementations are below.

- Hardware:
 - Operating System: Windows (Local Machine)
 - Processor: 4 cores
 - RAM: 8 GB
 - Graphic Card: NVIDIA GeForce GTX 1070 (Theano), AMD Ryzen 5 3500U (Keras/PyTorch)
- Dataset:
 - Name: MNIST
 - Data type: Handwritten digits
 - Data classes: All ten integers in range $[0, 9]$
 - Data resolution: 28 x 28 pixels
 - Train/test split: 45,000/10,000 samples
- Hyperparameters:
 - Epochs: up to 500 (early stopping, with 30-epoch lookahead)
 - Monte Carlo samples: 1, 10
 - Learning rate: .0001, .0003
 - Minibatch: 100 samples
 - AdaM optimizer ($\alpha, b1, b2$): .0003, .95, .999
 - Latent dimensions: 500
 - Activation functions: Sigmoid for the final layer of the decoder network when training on fully labeled MNIST data, SoftPlus on the final layer of the encoder network, ReLU elsewhere
- Choice of Priors:
 - Beta (α, β): 1, 5

- Gaussian (μ, σ) : 0, 1
- Gauss-logit (μ, σ) : -1.6, 1
- Uniform (low, high): .01, .99 (used when drawing a noise term)

D. Code Repositories

All our implementations are available via GitHub repositories, hyperlinks below.

- Theano
- Keras and PyTorch

E. Report Authorship

In this section we list the authors of the document and the sections written by each of them:

- Adu Matory: Introduction, Background, Conclusions, Appendices
- Konstantin Mehl: GEM Random Variables, Stick-Breaking Variational Auto-encoders
- Santosh Dhirwani: Semi-supervised Variant, Implementation, Experimentation, Future Work, Project Organization