Abstract

In this project, we modified the Variational Autoencoder ("VAE") model proposed in [2] to produce binary latent encodings that are used as semantic hash codes. We compared semantic hash codes produced by our VAE models with those produced by the plain autoencoders proposed in [1]. Our main contribution is to demonstrate that VAEs can be constructed with non-Gaussian latent variables. We trained and evaluated the information retrieval performance of our VAE models on the MNIST dataset and were able to demonstrate that our modified VAEs are able to generate meaningful binary semantic hash codes.

1 Introduction

1.1 Semantic Hashing

The goal of semantic hashing is to associate a binary hash code with each input, where semantically similar inputs have very similar binary hash codes (i.e. small Hamming distance). This allows for fast information retrieval of similar items by flipping some number of bits and looking up these similar hash codes in a hash table. Learning the binary hash codes can be done with an autoencoder as proposed in [1], where the bottleneck layer is made up of sigmoid units that are coaxed into operating in the extreme regions where the outputs are very nearly 0 or 1. This is done by injecting a large amount of Gaussian noise during training. Semantic Hashing with plain autoencoders for retrieving documents and images has been studied in [1, 4, 5, 6, 7].

It is important to note that the primary goal of semantic hashing is constant-time lookup of items in a massive corpus of items. Other information retrieval methods, like Latent Dirichlet Analysis, may be able to get better scores on metrics like precision and recall, but will be much slower than semantic hashing for large corpuses.

1.2 Variational Autoencoders

VAEs are best explained by separately discussing the role of the encoder and decoder. The decoder network can be understood as a probabilistic graphical model, where some number of latent variables are responsible for generating data. Many algorithms exist to do learning and inference with probabilistic graphical models, but most end up being intractable for high dimensional data or complicated distributions. A clever approach is to use a neural network to learn the parameters that govern the posterior distributions of the latent variables. This is the role of the encoder in the VAE.

Figure 1: High-level architecture of a VAE

From a system-level perspective, a fully trained VAE works as illustrated in Figure 1: Given an input, the encoder estimates the parameters of the posterior probability distributions governing each latent variable. Some number of samples (often just one) are drawn from these latent variable distributions and fed through the decoder to create a
reconstruction of the input data.

A necessary reparameterization trick is to express the sampling operation in a way that makes it differentiable with respect to the parameters of $p(Z)$, which allows the overall system to be trained through gradient descent optimization. This is done by injecting random Gaussian noise $\varepsilon = N(0, 1)$ into the bottleneck layer and transforming it into a sample from $p(Z|X)$ by the deterministic and differentiable function $z = \mu + \varepsilon \sigma$

During training, the optimization objective is to maximize the variational lower bound on $p(X)$, which is accomplished by minimizing the sum of reconstruction error and the KL divergence between prior and posterior distributions of the latent variables. Plain autoencoders only aim to minimize reconstruction error, while VAEs also aim to produce good probabilistic models of the latent variables and data.

A repeated theme throughout our project is that we use this additional KL divergence term and its sub-components as knobs that we can tweak to get the overall system to behave in a manner we desire.

1.3 Project Outline and Main Contributions

When we began this project our goal was to build a great informational retrieval system, and we did so by building a very deep but plain autoencoder. We explored autoencoders primarily built with ReLU, tanh, and sigmoid activations. We experimented with the number of code units at the bottleneck layer and how much noise to inject. Throughout these experiments we learned about the trade-offs between precision and recall in informational retrieval, and how our autoencoder design choices play into those trade-offs. We envisioned that this report would be filled with precision vs recall and Receiver Operating Characteristic (“ROC”) curves, and tables pitting our performance against state of the art.

However, when we began our literature review we discovered VAEs, and noticed that they too work by injecting noise into the bottleneck layer. We wanted to explore the connection between the injected noise in semantic hashing and VAE. In order to explore this connection, we set out to do semantic hashing with a VAE, and this became the primary goal of this project.

Though much of this report will document our experiments with plain autoencoders and the tradeoffs in information retrieval tasks, we feel that our main contribution is in extending the mathematics of VAEs to work with non-Gaussian and nearly binary latent variables.

2 Semantic Hashing with Plain Autoencoders

As illustrated in Figure 2, we implemented semantic hashing with plain fully connected autoencoders that closely followed the methodology in [1]. Due to recent advances in neural network optimization, we were able to directly train the autoencoder instead of having to pre-train a Restricted-Boltzmann-Machine model as in [1].

![Figure 2: High-level architecture of a semantic hashing plain autoencoder](image)

3 Semantic Hashing with Variational Autoencoders

3.1 Hierarchical Latent Variable Model ("VAE-Normal-Tanh")

Because VAEs were originally designed to work with continuous Gaussian latent variables, our first approach was to use the same and to transform them into binary codes. Figure 3 shows our framework, where a layer of Gaussian latent variables are squeezed through a sigmoid layer to create a subsequent layer of approximately binary latent variables. This framework is naturally suited to existing VAE implementations that predict the means and variances of Gaussian latent variables.
Figure 3: High-level architecture of a VAE with Hierarchical Latent Variables

The challenge is to coax the system into predicting suitable values of \( \mu, \sigma \) that result in extreme, nearly binary latent variables when passed through the sigmoid. If we say that our prior belief is that the Gaussian latent variables have very large variances (large in relation to the width of the sigmoid), then the optimization routine will try to minimize KL divergence and return posterior variances that are also very large. Like we saw with plain autoencoders, the network compensates for large incoming variances by learning to make the inputs to the sigmoid very extreme, resulting in nearly binary outputs. We view these as a second layer of latent variables whose task is to feed the decoder network to build good reconstructions of the input.

3.2 Beta Latent Variable Model (“VAE-Beta”)

Our second approach is to have just one layer of latent variables that represent our nearly binary code units. Beta distributions are an appropriate choice, and are flexible enough to model both our prior beliefs that a bit can be either a 0 or 1, and our posterior beliefs that a bit is a 0 or 1.

Bernoulli distributions would model the behavior more directly, but discrete probability distributions do not lend themselves well to stochastic gradient descent, as the infinitesimal quantity \( \frac{d}{dz} \) does not exist if \( z=\{0,1\} \).

Figure 4: We can model the behavior of binary code units with continuous beta distributions with appropriate shape parameters.

Figure 5: High-level architecture of a VAE with Beta Latent Variables

The first major challenge we faced was to compute the KL divergence between a beta prior and a beta posterior, which was much more difficult to compute than the Gaussian case. Nonetheless, the KL divergence can be computed as [8]:

\[
KL(q(z|x)||p(z)) = \ln \frac{B(\epsilon, \epsilon)}{B(\alpha, \beta)} - (\alpha - \epsilon)\psi(\alpha) - (\beta - \epsilon)\psi(\beta) + (2\epsilon - \alpha - \beta)\psi(\alpha + \beta)
\]

\( B(\cdot) \): Beta function
\( \psi(\cdot) \): Digamma function
\( \alpha, \beta \): Parameters of beta distribution
\( \epsilon \): Small value used for both \( \alpha, \beta \) in the prior

Our second major challenge was to recast the sampling operation in a way that is differentiable with respect to the posterior distribution parameters to allow the entire system to be trained via gradient descent. In other words, we
needed a "reparameterization trick" of our own. The general approach is to randomly sample from a uniform
distribution and feed it into the inverse cumulative probability (cdf) function. This is a deterministic function that
can be differentiated with respect to the parameters of the distribution. We note that not all probability
distributions have inverse cdfs with a closed analytical form. The Gaussian distribution is perhaps the most
famous example of a pdf that has no closed-form cdf, and the reparameterization trick of Kingma and Welling [2]
is a clever way to get around that.

We found that the inverse cdf of a beta distribution was rather difficult to work with, as it involves beta and
gamma functions, and a derivative that involves digamma functions. To simplify our implementation, we used the
Kumaraswamy-distribution (α, β), which is very closely approximates the beta distribution but with an inverse cdf
that is a very simple analytical function given by:  \( Z = (1 - (1 - \epsilon)^{\frac{1}{\alpha}})^{\frac{1}{\beta}} \), where \( \epsilon \sim \text{Uniform}(0,1) \)

4 Experimental Methodology and Dataset

4.1 Model Architecture

In this section, we provide the architectural details of the models we experimented with. Please note that we have
chosen to use tanh rather than sigmoid as activation functions in our models.

4.1(a) Fully Connected Plain Autoencoder:

In the encoder, we decrease the number of hidden units in the layer by 50% from the previous layer until we reach
the number of latent variables {i.e. 6, 12, 20, 49}. We do the reverse in the decoder. A tanh activation function is
applied at each layer except the final decoder layer, where we chose to use the soft-plus activation as recommend
by [3]. Consistent with [1], we add random Gaussian noise to the values of the innermost encoder layer before
applying the tanh transform.

4.1(b) VAE with Hierarchical Latent Variables:

In the encoder, we apply a linear transform and a ReLU activation to the input layer resulting in a hidden unit
layer with 400 hidden units. We then apply a linear transform to the 400 hidden units to estimate the parameters (μ, σ) for the posterior distribution of each latent variable.

In the sampler, we sample the latent variables from the posterior normal distribution by sampling by \( \mathcal{N}(0,1) \) and
transforming the sample with a function parameterized by the parameters of the posterior distribution. We then
subject the latent variable to a tanh transform to binarize the output.

In the decoder, we first apply a linear transform and a ReLU activation to the latent variables resulting in a hidden
unit layer with 400 hidden units. We then apply a linear transform and a sigmoid to the 400 hidden units resulting
in an output layer of 784 units.

We use the sum of the KL divergence between the posterior and prior distribution of the latent variables (normal
or uniform) and the reconstruction error as our objective function for training the VAE through gradient-descent.

4.1(c) VAE with Beta Latent Variables:

Similar to the VAE with Hierarchical Latent Variable Model, the encoder learns the parameters of the
Kumaraswamy distribution (α, β) for each latent variable. In the sampler, we sample the latent variables from the
Kumaraswamy using the reparameterization trick described in Section 3.2. The decoder is identical to the one for
VAE with Hierarchical Latent Variable Model. We use the sum of the KL divergence between the prior
distribution and posterior beta-distribution of the latent variables and the reconstruction error as our objective
function.
4.2 Dataset

We chose to use the MNIST dataset of handwritten digits for training and evaluating our models. The MNIST dataset is split into a training set, validation set and test set of 50,000, 10,000 and 10,000 examples respectively. In the future, we would like to apply our models to textual corporuses and image datasets in order to evaluate the information-retrieval performance under more realistic conditions. However, we found the MNIST dataset to be a good dataset for us to visualize how the various models work in reconstruction, sampling and information retrieval tasks and performance preliminary information retrieval evaluations.

5 Experimental Results

We performed various experiments in order to understand how the various models perform on the information-retrieval task as well as to improve our understanding of the models. We trained our models using the training set, selected hyperparameters (such as level of noise) using the validation set and evaluated the models on the test set. We have chosen to provide a representative but non-exhaustive sample of visualizations to illustrate our observations.

5.1 Information Retrieval

Given a query image, the information retrieval task is to return similar/relevant images. In order to quantitatively assess the information-retrieval performance of the various models, we use the class-label of the handwritten digits as a proxy for the “ground truth” similarity between two handwritten digits. Given a query image, all images in the test dataset of the same class are considered relevant and all images of different class are considered irrelevant. This is not a perfect measure of relevance as it does not account for stylistic similarity/difference between handwritten digits, which may be relevant to users. Similar to [1], we used LDA and PCA as base-line comparisons for the autoencoder and VAE models.

We computed the Area under the Receiver Operating Characteristic (“AUC”) for each model and each setting of number of latent variables and noise level. For each model and each setting of number of latent variables, we select the noise level that resulted in the highest AUC. We present the results in the following Figure 6.

![AUC for different models and different number of latent variables](image)

Figure 6: AUC for different models and different number of latent variables

We observed that the fully-connected autoencoder achieved the highest AUC and the AUC appeared to increase slightly as we increased the number of latent variables. VAE-Normal-Tanh had high AUC when there are few latent variables but the AUC decreased significantly as we increased the number of latent variables. The VAE-Beta had fairly consistent AUC across different number of latent variables. We see that for Variational-Autoencoder models, binarizing the latent representation using tanh or an approximate beta-distribution resulted in higher AUC than the standard Variational-Autoencoder with normally distributed latent variables.
We compared the AUC achieved by our models against the base-line of Fisher’s LDA and PCA. When we used the cosine distance of the Fisher LDA representation of images to retrieve similar images, we saw that Fisher LDA was able to achieve a very high AUC of around 0.93. When we used the Fisher LDA representation of the images as a binary hash code (0 if the LDA value < mean and 1 otherwise), we see the AUC score drop significantly to around 0.85. The best-performing model we tested (plain autoencoder with 49 latent variables) is able to get close to but not match the binarized LDA. This is reasonable since the LDA representations were specifically tuned to separate the training examples by class label whereas the autoencoder models were trained without knowledge of class labels. We also see the fully connected autoencoder is able to achieve similar AUC score as PCA and many of the models we tested can significantly beat binarized PCA.

In order to get a better understanding of how the number of latent variables affect the precision and recall of the model, we examine how the precision and recall changes with hamming distance as we vary the number latent variables in the model. We also examined the ROC of each model as we changed valued the number of latent variables. See Figure 7. We observed that as we increased the number latent variables, the precision decreased and the recall increased more slowly as we increased the hamming distance. The impact of different number of latent variables is not as apparent in the ROC.

![Figure 7: Precision vs. Recall vs. Hamming Distance and ROC for Plain Autoencoder](image)

We also perform some exploratory qualitative assessment by looking at samples of handwriting images retrieved. We experimented with different models and different number of latent variables. We saw that as we increased the number of latent variables, the precision generally increased and the results returned for the small hamming distances tend to be of the same class or visually similar to the query image. Conversely, as we decreased the number of latent variables, the recall rate generally increased. As the results were visually similar across the autoencoder and variational-autoencoder models, we present two examples below in Figures 8 and 9:

**Query:**  

![Query Image]

**Results:**

![Result Images]

*Figure 8: Autoencoder with 49 latent variables and gaussian noise set to 4. The first, second, third and fourth row correspond to hamming distance of 0, 1, 2, 3 respectively.*
5.2 Latent Representations

We consider the latent representations of the handwriting digits produced by the various models. We consider the marginal distribution of the latent variables and a 2-dimensional visualization of the latent variables using t-SNE in Figure 10. We also consider the mutual-information between the latent variables representation of a handwritten digit and its class label in Figure 11.

Introducing noise into the latent layer of the autoencoder or increasing the noise parameter in the prior for the latent variable in the VAE cause the marginal distribution of the latent variable to concentrate in a small area around -1 and 1. We see that the distribution of latent variables concentrate around -1 and 1 more quickly when
we increased the noise level in the autoencoder model as compared to when we increased the noise parameter in the prior distribution of latent variables in the VAE model. This is expected because the noise in the autoencoder model is not dependent on actual data while the noise in the VAE model is dependent on the data as well as the noise parameter in the prior distribution for the latent variable.

Figure 11: Mutual information between the latent variable and the class label for fully connected autoencoder with noise set to 4.

It appears that as we increase the number of latent variables, the importance of individual variables tend to decrease. Also, when there are a large number of latent variables (49, 98), only a portion of the latent variables (approximately 20-40%) appear to have significant mutual information with the class label. We observed that the distribution of the mutual information between latent variables and the class label varies between the different models. However, the general trend appears to hold.

5.3 Image Generation

VAEs are generative models, meaning that you can draw samples from the latent variables and pass those sample through the decoder network to generate data. This often serves as a powerful visual sanity check to make sure the system is acting sensibly.

One typically samples from the posterior distribution over the latent variables after training so that the generated images lie on or near the data manifold. We can model binary codes as multivariate Bernoulli random variables, and constructing the posterior is a matter of collecting the sufficient statistics of bit counts and correlations in the training set. However, we left this approach for future work.

Instead, we sampled from our latent space uniformly by generating random bit streams. We did so knowing that some percentage of these random bit streams could produce non-sensical outputs that do not resemble real images. We wanted to get a rough idea of the volume of our data manifold as a percentage of the overall space, and to see how this changes as a function of the number of bits.

We begin with 6-bit codes, meaning the entire latent space has only $2^6 = 64$ possible values whose generated outputs can be easily plotted in Figure 12. Note that almost all of the codes produce images that are recognizable to the human eye. In this case, the data manifold occupies almost 100% of the latent space. Also note that the different digits are not equally represented. For example, there is an overabundance of 2s, 6s and 0s, but very few
1s, 4s, and 7s. An explanation for this behavior is that we are doing unsupervised learning based largely on RMSE, and from this perspective there is not much difference between a 4 and a 9, for example.

**Figure 12:** Left: autoencoder with 6 latent variables and gaussian noise with $\sigma$ set to 4. Middle: VAE-normal-tanh with 6 latent variables and prior $\sigma$ set to 4. Right: VAE-beta with 6 latent variables and prior alpha and beta parameters set to 0.1. Please note that as we move one square from left to right, the latent encoding changes by exactly 1-bit.

Next we examine 20-bit codes, which have approximately 1 million different bit permutations. That is too many to plot, so we repeatedly created 100 random bit streams and plotted the results in a 10x10 grid. Figure 13 shows just one such sample of 100 random bit streams that we feel is representative. Remarkably, most of these random bit streams still result in images that resemble real handwritten digits. Even the nonsense examples vaguely resemble particular digits.

Finally, we examined 49-bit codes, generating 100 random bit streams at a time as before. Once again, we were surprised to see that most randomly generated bit streams lead to recognizable digits. Figure 14 shows a representative sample of 100 codes plotted in a 10x10 grid.

**Figure 13:** Left: autoencoder with 20 latent variables. Right: autoencoder with 49 latent variables.

An explanation for this behavior is that most bits don't really matter, as the data has some true underlying dimensionality. This is obvious if you are thinking about latent variable models and their relation to dimensionality reduction techniques, but it can be quite easy to overlook if you are thinking about trying to avoid collisions by allocating more space to a hash table. To validate this hypothesis, we quantified how important each bit is by looking at information gain or the GINI coefficient. Here we take advantage of the fact that we have class labels for MNIST data, even though we are building an unsupervised learning algorithm.

**Figure 11** show a Pareto plot of the GINI coefficient for the importance of each bit. With 6-bit codes, all bits are very important. As we increase the number of bits, we eventually start to get bits that don't matter very much. By the time we get to 49-bit codes, most of the bits don't seem to matter much at all. It is unknown whether these bits capture unimportant noise, or minor stylistic flourishes in handwriting.
5.4 Reconstruction

We explore the qualitatively and quantitatively the impact of modelling choices and injection of noise on the reconstruction. While we are interested in building autoencoder and VAE models that are good at information-retrieval, studying the reconstruction results give us insights into how well the various models are at preserving information in the original image.

As expected, the RMSE generally increases as we decrease the number of latent variables or increase the noise level. The VAE with normal-distributed latent variables without the tanh transform had the lowest RMSE which is expected because the latent variables were not forced into near-binary code and therefore had higher capacity for information. We also observed that VAE-normal-tanh model tend to have lower RMSE than the other models when there are few latent variables but its RMSE does not improve significantly when we increase the number of latent variables past 12. The autoencoder and VAE-Approx-Beta models, on the other hand, had higher RMSE when the number of latent variables is small but the RMSE improved significantly and begin to approach the RMSE of the VAE-Normal model as the number of latent variables increased. This suggest that the VAE-Normal-Tanh model is able to use the latent variables more efficiently when the number of latent variables is small. However, the autoencoder, VAE-Uniform-Tanh and VAE-Approx-Beta models are able to take better advantage of increased number of latent variables to get more informative encoding when we increase the number of latent variables.

6 Conclusion and Future Work

In this project, we were able to reproduce the “plain” autoencoder used by [1] and build working variational autoencoders that used different strategies to obtain binary latent encodings. In future works, we would like to apply variational autoencoders to more interesting datasets such as document corpuses (eg. Reuters) and images corpuses (eg. ImageNet). We would also like to experiment with different architectural choices for the encoder and decoder in order to further optimize the information-retrieval performance. We would like to gain a better understanding of the data manifold in the space of latent variables by modelling posterior distribution of the latent variables given the training data and generating images from latent variables sampled from this posterior distribution. Another possible extension of our work is to experiment with other probability distributions for the latent variables.
7 References


