Dissertation Type: enterprise

DEPARTMENT OF COMPUTER SCIENCE

On Beyond Gaussian Variational Autoencoders in Language Generative Tasks

Adam Stein

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Tuesday 4th June, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Adam Stein, Tuesday 4^{th} June, 2019

Contents

Executive Summary

In machine learning, we rely on the data that we have to train our models with. Often the performance of our solutions is limited by the amount of data that we possess. By generating new artificial data points based on the data we have, we can improve the performance of our models. This process is known as "augmenting" our datasets. Variational autoencoders are a state-of-the-art neural network architecture that can be used learn a representation of sentences. We can use this representation to generate new artificial sentences to augment our datasets in natural language processing tasks. To build this representation, the variational autoencoder encodes the input into a distribution. The Gaussian distribution has long been the default choice for this encoding but there has been recent literature suggesting that it is poor at representing some types of data [3].

In this thesis, we experiment with using the von Mises-Fisher and Caucy distributions within the variational autoencoder as an alternative to the Gaussian. We hypothesise that these different distributions will improve the performance of models used to generate sentences. For our investigation we use the Yelp dataset [28] and focus on creating a model that generates artificial reviews based on the training data.

The highlights of this project are:

- The first documented von Mises-Fisher variational autoencoder implementation with learned parameters for an natural language processing based task.
- The first documented Cauchy variational autoencoder.
- The implementation of two different methods used to generate sentences using variational autoencoders.
- A comparative study to quantify the effects of using different distributions within the variational autoencoder.

Supporting Technologies

- I used the Tensorflow framework to build my neural network models.
- I used the SciPy library for an implementation of the modified Bessel function with numpy arrays.
- I used publicly available code from the authors of the Prototype and Edit model paper for design inspiration `https://github.com/kelvinguu/neural-editor`.
- I used datasets provided by the authors of the Prototype and Edit model paper `https://worksheets.codalab.org/worksheets/0xa915ba2f8b664ddf8537c83bde80cc8c/`.
- I used pre-trained GloVe-300d word embeddings that are available online `https://nlp.stanford.edu/projects/glove/`.
- I had access to AWS computing instances provided by Fractal Labs Ltd.

Notation and Acronyms

KL	:	KullbackLeibler
vMF	:	von Mises-Fisher
PE	:	Prototype and Edit
LSTM	:	Long Short-Term Memory
VAE	:	Variational Autoencoder
	$\vdots$	
$\mathcal{L}$	:	Loss function of a model
$\mathbb{E}[.]$	:	Expectation over a random variable
$\mathcal{N}(.)$	:	Gaussian distribution
$vMF(.)$	:	von Mises-Fisher distribution
$\mathcal{C}(.)$	:	Cauchy distribution
$U(.)$	:	Uniform distribution
$KL(p\|\|q)$	:	KL divergence between distributions p and q
z	:	A latent variable that represents the representation learnt by a VAE

Acknowledgements

I would like to express my deepest gratitude to Fractal Labs Ltd and particularly Nicholas Heller. This project would not have been possible without the resources and assistance provided by Fractal Labs Ltd. I would also like to sincerely thank Andrea Karlová whose expertise and insight have made her an excellent mentor. She encouraged me to push myself academically and has been critical to the project's success.

Chapter 1

Contextual Background

Natural language processing (NLP) tasks relate to a large number of problems involving text and speech. In this thesis, we focus on a particular problem within NLP: natural language generation (NLG). We aim to design models to generate new artificial sentences based on a training set of sentences. The model should be able to generalise well to generate new unseen sentences whilst maintaining a certain quality in terms of grammar and sentence meaning.

1.1 Problem Definition

In machine learning, the data that we use is extremely important. Performance of models is frequently limited by the size of the training set that we provide to them for training. To move beyond our current performance levels, we need to be able to increase the size of these training sets but often this is not possible. In many tasks extra data does not exist or would be too costly to attain. A common approach across the field is to "augment" our data; this means adding some type of noise and slightly modifying a data point to create another valid data point. This technique has been shown to dramatically increase performance of models and prevent a phenomenon known as overfitting which occurs when a model struggles to generalise to unseen test cases [26]. A simple example of this data augmentation is in image classification: imagine a model that classifies types of birds but we only have a limited number of pictures per class. By flipping all the images in the horizontal direction we have doubled the amount of training data available. This type of approach is not so straightforward in NLP tasks. If we slightly modify a sentence we could end up with a sentence that does not make grammatical sense or had its meaning dramatically changed. We want to able to create a model that creates new fictitious sentences based around sentences that we already have in our training sets. By doing this we will be able to dramatically increase the performance of our models, particularly in areas where data is limited.

1.2 Generating Sentences

There have been a number of proposed models to address the sentence generation problem but none solve the task sufficiently. There have been considerable advancements in the field of machine learning in recent years and the most successful models utilise neural network architecture to solve the problem of sentence generation.

1.2.1 Rule-based Substitutions

A simple rule-based approach could be used to generate slight alterations to sentences by substituting certain words. This could be changing the tense of a sentence or swapping certain words with synonyms. This basic approach is effective at generating a small number of sentences but they will be too similar to the existing training set to dramatically improve model performance. To produce sufficiently diverse sentences we will need to utilise neural networks.

1.2.2 Bag-of-Words Model

The bag-of-words model was introduced to model text documents. It builds a set of words that appear in the text and disregards the order in which they appear as well as any grammar. The number of times that each word appears in the body of text is maintained. A neural network architecture is then used to decode the bag-of-words representation into text. Even for short sequences of text, it is difficult for the network to learn a strong relationship between the bag of words and the order they should appear. In addition to this, for any sentence we want to generate, we need to hand define the bag-of-words we want it to consist of. This is a difficult process to automate and may result in poorly constructed sentences even with a well-trained decoder.

1.2.3 Recurrent Neural Network Language Model

A Recurrent Neural Network Language Model (RNNLM) utilises recurrent neural architectures that are designed to capture sequence data to generate text. A RNNLM generates sentences one word at a time based on the words seen previously within a window. We start with a seed word and the network fills in words probabilistically to generate whole sentences. This is far more effective than the bag-of-words model and no longer requires us to define a set of words that we would like to appear in our generated sentence. Despite this model being effective at producing grammatical sentences, it tends to often produce common utterances found in the training set rather than new unseen sentences. Attempts to vary the diversity of sentences often results in less coherent generated sentences [6].

1.2.4 Conventional Autoencoder

Generating new fictitious data based on training data is not an NLP specific problem; a model that solves this task is referred to as a language generative model, often enhanced with latent variables. Generative models with latent variables aim to learn to encode input (sentences in our case) into a hidden variable z. This z can then be decoded to reconstruct the original sentence. By taking a trained model and sampling a random z, we should be able to decode this variable to then construct a new sentence that we may not have seen before. This architecture is known as an autoencoder and was introduced by Rumelhart et al. [18]. The autoencoder performs far better at producing diverse sentences that make grammatical sense. Similar to the bag-of-words model, the autoencoder builds a representation of a sequence of text. However, in this case, the encoding is learned through unsupervised learning using neural networks. Autoencoders build a global representation of a sentence allowing it to capture high level features such as topic and style [2]. We can generate new sentences by sampling randomly from our representation and decoding.

1.2.5 Variational Autoencoder

The variational autoencoder (VAE) is an extension on the conventional autoencoder introduced by Kingma et al [10]. Variational autoencoders differ from the conventional autoencoder by sampling z from a learned distribution rather than through a deterministic process. The state-of-the-art models generate sentences using the variational autoencoder approach and have far outperformed any model previously proposed [2]. Given that the variational autoencoder has dramatically better performance in text generation tasks, it is sensible to focus on extending this architecture rather than any other model.

1.3 Models

One of the best performing models for generating text is one proposed by Bowman et al [2]. The approach follows a basic variational autoencoder architecture with slight alterations made to be able to perform well when generating sentences. It uses the common-choice of a Gaussian distribution when sampling for our representation z. Following on from this, Kevin Guu et al. took a novel approach by trying to focus on editing a sampled sentence rather than generating a sentence from scratch [6]. A very key part of this paper was the authors moving away from a Gaussian distribution and replacing it with the von Mises-Fisher (vMF) distribution. The rationale behind this was that the common-choice Gaussian distribution is not very good at NLP tasks relative to other distributions. We will discuss this further in Chapter 2. It has also been suggested by Davidson et al. that the vMF distribution may be a better distribution for variational autoencoders in other fields outside NLP [3]. The success of the Prototype and Edit model proposed by Guu was definitely attributed to the change in approach as well as the change in distribution.

1.4 Distribution Experimentation

If we experiment using different distributions in these two different models then we can quantify improvements from using the von Mises-Fisher distribution in the sentence generation task. We propose to use two different methods of sentence generation to build a quantifiable consensus of the best distribution to use in NLG. If the vMF distribution is proven to be superior over the Gaussian distribution then logic would dictate it could be used to improve the performance of other models for different problems in NLP. Furthermore, by setting up this framework of comparison, we can experiment with additional distributions which may prove superior to both the Gaussian and vMF.

The Cauchy distribution has a number of qualities that suggest it may be superior to the Gaussian distribution at modelling certain types of data. It has previously been used to compress data into visual representations [24] but has never been used in the variational autoencoder. We use our framework to introduce the Cauchy distribution for both models to see how well it performs in the task of sentence generation.

The Gaussian distribution has long been the default choice and little experimentation has been done with using other distributions within variational autoencoders. By using the vMF or Cauchy distribution we may be able to improve the performance of both models such that we can perform even better at the task of generating sentences to augment our datasets.

Changing the distributions is not a trivial task and requires complex mathematics to derive equations and implementation of sampling algorithms. A general vMF variational autoencoder has already been proposed by Davidson et al. [3] but a number of alterations will need to be made for high dimensional text data. We will introduce the first documented learned vMF variational autoencoder for a NLP task. The Cauchy distribution has been an entirely unexplored distribution; we will mathematically prove the suitability of the distribution and outline how it can be practically developed for our two methods.

The problem of the Gaussian not being an appropriate distribution is not unique to NLP tasks. The variational autoencoder was not designed specifically for NLP tasks and has a vast number of use cases across machine learning. These distributions may have merits in other fields of machine learning with relatively high dimensional data. We aim to add to the growing literature promoting alternative distribution choices to encourage researchers to give the area further study for their specific tasks.

1.5 Dataset

For both models, we will focus on generating artificial reviews using the publicly available Yelp dataset [28]. We use a set 1.2 million reviews for different businesses for our language models. The Yelp dataset was specifically chosen by Guu et al. in the Prototype and Edit model because there are a large number of sentences that are semantically similar. This is important in the training process such that the network can learn how to make small edits on sentences. We provide a few example reviews to give more of an insight to the form of the dataset:

1. I didn't think it was bad per se
2. Definitely a place to take your significant other or with friends for a casual dinner and drinks .
3. If you get a chance to visit , I highly recommend !
4. This place was my favorite place EVER .

The authors of the Prototype and Edit paper simplified the dataset further by changing words outside the 10,000 most used works to an out of vocabulary token. They also used a named entity recognition tool to change words recognised as entities to their class [20]. Below is an example sentence before and after the process:

- Before: One of London's hidden gems.
- After: ⟨CARDINAL⟩ of ⟨GPE⟩'s hidden treasures.

This altered training set has also been publicly released and will be used to train both models [7]. It is important to use the same dataset as the authors so that we can easily compare the performance of our model in comparison to the author's benchmark.

We aim to generate fictitious reviews that seem both grammatically sound and plausible. The plausibility of a sentence is hard to quantitatively evaluate and refers to the general topic around a sentence; for example a review that says "the pizza was great" or "the service was great" is far more plausible than "the floor was great". Even though all three make grammatical sense, any human would label the first two as far more likely to be genuine reviews.

1.6 Aims

1.6.1 Understanding the Variational Autoencoder

In the area of machine learning, variational autoencoders are considered one of the most complex types of architectures. Particularly for this project, a real in-depth understanding is going to be required of all mathematical theory rather than a high level understanding. The first aim of the project is studying the paper that initially proposed the architecture by Kingma et al [10]. With such a complex topic, a thorough understanding of the content is the priority before any implementation.

1.6.2 Understanding the Bowman et al. Model [2]

Following the completion of this, we will begin investigating the Bowman model. This model is far simpler than the Prototype and Edit model and is a much more intuitive extension on the Kingma paper but for sentences rather than the general case. The paper specifically focuses on a number of optimisation problems and understanding this content will be paramount as common problems are likely to emerge when introducing new distributions.

1.6.3 Understanding the Prototype and Edit Model [6]

Instead of immediately implementing the Bowman model, it makes more sense to first focus on understanding the theory behind the Prototype and Edit (PE) model. The PE model takes a more unconventional approach but a large amount of code will be shared between the two models. This will enable us to optimally design both models. Once again, the theory and the mathematical computations in the PE model are complex and having a thorough understanding before execution is essential. We will briefly outline the relevant theory in Chapter 3.

1.6.4 Implementation of the Models

Once we recall the selected preliminary concepts, we implement the Bowman model and the PE model. We aim to implement this in a way to have as much code shared between both models as possible. This is not just good coding practice but will also provide a much more meaningful comparison between the two different models. If the majority of the programmed architecture is identical for both models then the comparison will be a pure result of approach. There is code that is publicly available for the PE model but we decided not to utilise it because of the above reason, as well as it being a hard code base to manipulate and extend [5].

1.6.5 Implementation of Von Mises-Fisher VAE

Following on from this, we implement the vMF VAE such that it can be used as a configuration in both models. This will be an extension on the Prototype and Edit model which currently uses fixed hyperparameters in the distribution. The mathematical theory behind the model has already been proposed by Davidson et al. [3] but we will focus on practically implementing the model for high dimensional data.

1.6.6 Implementation of Cauchy VAE

We will then focus on introducing the use of the Cauchy distribution in both models. There is no literature exploring the theoretical use of the Cauchy within a VAE and therefore we will focus on deriving necessary equations for its implementation. We will also outline any practical implementation details for the sentence generation task.

1.6.7 Comparison of Distributions

After implementing all distribution configurations, we will run experiments on both models with all distributions. We aim to analyse and compare their performance relative to one another and build consensus on the best distribution for the sentence generation task.

1.7 Summary of Aims and Objectives

The high level objective of this project is to compare two different approaches to using variational autoencoders for sentence generation and attempt to improve their performance by changing the probability distributions they use. The aims and objectives are:

1. Gain a thorough theoretical and practical understanding of variational autoencoders.
2. Implement two different approaches for variational autoencoders that generate sentences.
3. Implement a von Mises-Fisher variational autoencoder for generating text.
4. Implement a Cauchy variational autoencoder for generating text.
5. Explore how different probability distributions affect the performance of the models.

Chapter 2

Technical Background

In this thesis, we will experiment using different probability distributions within a variational autoencoder to generate sentences. In this chapter, we will outline the necessary technical knowledge to solve the problem of interest. We begin by recalling the basic terms in probability theory and introduce the distributions we will use throughout this thesis. Next we briefly discuss different architectures of neural networks used in natural language modelling. We conclude this chapter with a brief overview of common metrics to evaluate similarity. This project is aimed at thoroughly understanding the theory behind variational autoencoders as well as implementing and extending models from two different papers; because of this, all background behind variational autoencoders and the two papers will be found in chapter 3.

2.1 Probability Theory

2.1.1 Random Variables

A random variable is a variable that can take on values probabilistically. It is a description of all possible outcomes of an experiment; it is combined with a probability distribution to specify the fraction of time we would expect to see that result if we did an infinite number of trials. A discrete random variable has a finite number of possible outcomes and a continuous variable has an infinite number of possibilities over some range which could also be infinite.

Expectations

The expectation of a random variable is denoted using $\mathbb{E}$ and specifies the expected value of a random variable. This is not the most likely value to be sampled but a sum of possible outcomes weighted by their probability. The $\mathbb{E}$ is often sub-scripted with the variable that we are finding the expectation over.

$$\text{Discrete:} \quad \mathbb{E}_x[p(x)] = \sum_{x \in \mathcal{X}} x \cdot p(x) \tag{2.1}$$

$$\text{Continuous:} \quad \mathbb{E}_x[p(x)] = \int_{-\infty}^{\infty} x \cdot p(x) dx \tag{2.2}$$

Conditional and Joint Probability

Often we have multiple random variables in an experiment. Conditional probability is a formalisation of one random variable influencing the outcome of another. If we already know the outcome of one variable then we can create a new distribution of the other based on the information we know. This concept is utilised in machine learning where we are often aim to create models that calculate the probability of a target label y given an input x denoted by $p(y|x)$. We use joint probability to formalise the probability for the outcome of multiple random outcomes happening as a result of an experiment. This is denoted by $p(x^{(1)}, .., x^{(n)})$. It is often the case that we need only a subset of the variables in a joint distribution. We can use marginalisation and integrate out variables which we do not need. For example, eliminate the variable x_n, we compute: $p(x^{(1)}, .., x^{(n-1)}) = \int_{-\infty}^{\infty} p(x^{(1)}, .., x^{(n)})\ dx^{(n)}$. Joint distributions can be also decomposed into conditional distributions:

$$p(x^{(1)},..,x^{(n)}) = p(x^{(1)}) \prod_{i=2}^{n} p(x^{(i)} \mid x^{(1)},..,x^{(i-1)}) \tag{2.3}$$

Bayes Theorem

In machine learning, we often want to learn a conditional distribution representation based on our data. Bayesian inference is the process of learning the conditional distribution of a random variable(y) based on some observed evidence (x) known as the posterior. We have a likelihood $p(x|y)$ as well as a prior $p(y)$.

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{2.4}$$

For additional reading on random variables see Goodfellow et al [4].

2.1.2 Latent Variables and Generative Models

A latent variable is used to represent a phenomenon that is unobservable. In latent variable models, we assume that the hidden latent variable influances the data that we observe. Throughout, the hidden latent variable will be referred to as z. There is a probability distribution $p(x|z)$ for all data points that describes the generative property of our latent variable. A generative model learns to encode our data into this representation $p(z|x)$ as well as how to decode it $p(x|z)$. For sentence generation, if we can learn a good representation for our latent variable then we can utilise it to generate new semantically similar sentences that make sense. To do this, we will need to learn how to encode our sentences into the latent variable that generated them. We can then add some noise to generate new unseen artificial sentences using a decoder.

2.1.3 Monte Carlo Sampling

Calculating expectation values for random variables tend to result in large integrals that are computationally expensive, especially in generative models. Sampling allows us to approximate of these large calculations to make our computations more tractable. Monte Carlo sampling treats sums or integrals as expectations to generate an unbiased estimator y' over a function f.

$$y = \sum_{x} p(x)f(x) = \mathbb{E}[f(x)] \text{ or } y = \int p(x)f(x)dx = \mathbb{E}[f(x)] \tag{2.5}$$

We approximate y by drawing a number of samples and taking the average.

$$y' = \frac{1}{n}\sum_{i=1}^{n} f(x_i) \tag{2.6}$$

$$\lim_{n\to\infty} y' = y$$

2.1.4 Rejection Sampling

For some probability distributions it is not trivial to generate samples. If we take our probability density function $p(x)$ then uniformly sampling in this space is equivalent to sampling from $p(x)$. Rejection sampling uses another distribution $q(x)$ such that $Mq(x)$ is an upper bound of $p(x)$ for $M > 1$. Typically, we select $q(x)$ as a distribution that is easy to sample; we sample from $q(x)$ and check if the sampled point is below the probability density function of $p(x)$. If this is the case then we accept the sample, otherwise we reject the sample and repeat the process.

1. $x^{(i)} \sim q(x)$
2. $u \sim U([0, 1])$
3. if $u \leq \frac{x^{(i)}}{Mq(x^{(i)})}$ then accept sample $x^{(i)}$ for $p(x)$
4. else return to step 1

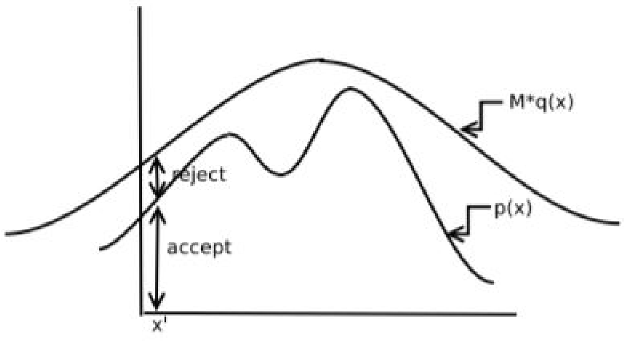

Figure 2.1: Rejection sampling from a distribution $p(x)$[9]

2.2 Distributions

Our main focus is on investigating methods of learning a generative model to infer a latent variable z. The performance of our models is strongly influenced by the prior distribution over this latent space. Over the course of this project, we investigate the effect on performance on using the Gaussian, von Mises-Fisher and Cauchy as the distribution in the variational autoencoder.

2.2.1 Gaussian Distribution

The Gaussian (or normal) distribution, denoted by $\mathcal{N}$, is a common distribution used in machine learning. It consists of two parameters μ and σ^2 that represent the mean and variance. The mean dictates the most likely value to be sampled from the distribution and the variance controls the speed that the probability decays over our sample space. The distribution is symmetric around the mean. A multivariate normal distribution is an extension of the Gaussian distribution in the space $\mathbb{R}^n$. In this case, the variance parameter is replaced with with a covariance matrix Σ. The mean is still represented by μ but is now a vector rather than a single value.

$$x \sim \mathcal{N}(\mu, \Sigma) \qquad p(x) = \sqrt{\frac{1}{(2\pi)^n det(\Sigma)}} exp\Big(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\Big) \tag{2.7}$$

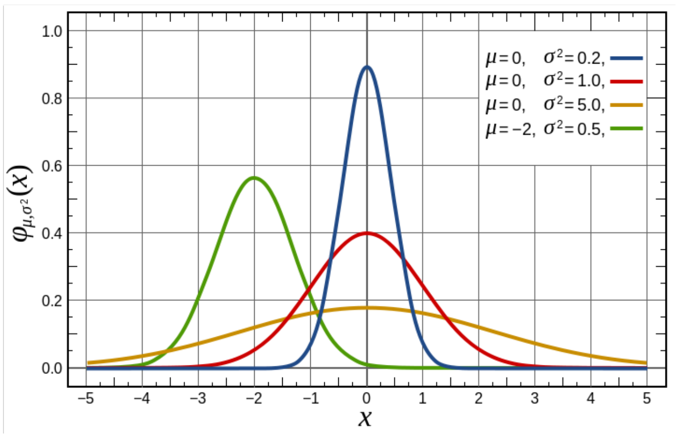

Figure 2.2: A diagram illustrating how the shape of a Gaussian changes with different values of μ and σ^2 [14]

The Gaussian distribution has a number of properties that make it a popular choice [14]. The standard version of the central limit theorem states that any distribution with finite second moments can be approximated as a Gaussian distribution given a large enough number of samples. This means that the Gaussian distribution can be applied to a wide range of problems. In Bayesian inference, the Gaussian distribution is a conjugate prior meaning that the product of two Gaussian distributions is also a Gaussian. Being conjugate means that we can perform Bayesian inference to calculate a Gaussian posterior by multiplying a Gaussian prior and likelihood. On the other hand, the Gaussian distribution has unfavourable limitations [3]:

1. **Origin Gravity:** "In low dimensions, the Gaussian density presents a mass around the origin, encouraging points to cluster in the centre. This is particularly problematic when the data is divided into multiple clusters. Although an ideal latent space should separate clusters for each class, the normal prior will encourage all the cluster centres towards the origin. An ideal prior would only stimulate the variance of the posterior without forcing its mean to be close to the centre. A prior satisfying these properties is a uniform over the entire space. However such a uniform prior, is not well defined on the hyperplane."

2. **Soap Bubble Effect:** "It is a well known phenomenon that the standard Gaussian distribution in high dimensions tends to resemble a uniform distribution on the surface of a hypersphere, with the vast majority of its mass concentrated on the hyperspherical shell."

2.2.2 Von Mises-Fisher Distribution

The Gaussian distribution studied in the previous section defined probability over a flat planar subspace of the vector space, also called a hyperplane. Davidson et al. proposed that some types of data may be better represented by a distribution designed to handle spherical representations [3]. We will be approaching language modelling with a directional data approach using word embeddings and therefore explore the use of a distribution over the hypersphere. The von Mises-Fisher(vMF) distribution is a probability distribution often seen as a Gaussian distribution on a hypersphere. A hypersphere is a set of points equal distance away from the centre making a spherical shape in the vector space. Guu argued that that the vMF distribution is particularly appropriate for text data because the tails decay with cosine similarity [6]. The use of the vMF distribution is less computationally convenient, although we hypothesise that it may be better at capturing the structure of our data manifold.

Bessel Function

A Bessel function is often used in systems that use a spherical coordinate system. A Bessel function of the first kind order v, denoted by $J_v(x)$, is used to describe the solution to the Bessel differential equation:

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + y(x^2 - v^2) = 0 \tag{2.8}$$

The modified Bessel function of the first order, $I_v(x)$, is an extension on $J_v(x)$ and is used in the vMF distribution.

Initialisation

$$f_p(x; \mu, \kappa) = C_p(\kappa)\exp(\kappa\mu^T x) \tag{2.9}$$

$$C_p(\kappa) = \frac{\kappa^{p/2-1}}{(2\pi)^{p/2} I_{p/2-1}(\kappa)}. \tag{2.10}$$

The parameters μ and κ are called the mean direction and concentration parameter, respectively. Similarly to the Gaussian distribution, the vMF distribution is conjugate. We have a uniform distribution over the hypersphere when the concentration parameter is set to 0.

There is a "vanishing surface problem" which suggests that there may be weaknesses in the vMF distribution in high dimensions because points get spread to thinly [3]. So far there has been little investigation into the suitability of the vMF distribution for directional data that is in high dimensions, like text.

2.2.3 Cauchy Distribution

Another alternative to the Gaussian distribution is the Cauchy distribution. The Cauchy distribution is a heavy-tailed distribution. This means that unlike the Gaussian, the tails of the distribution are not exponentially bounded and instead decay over a parabola. This results in a much slower decay of the distribution making it better at representing some types of data. Success has been seen with using

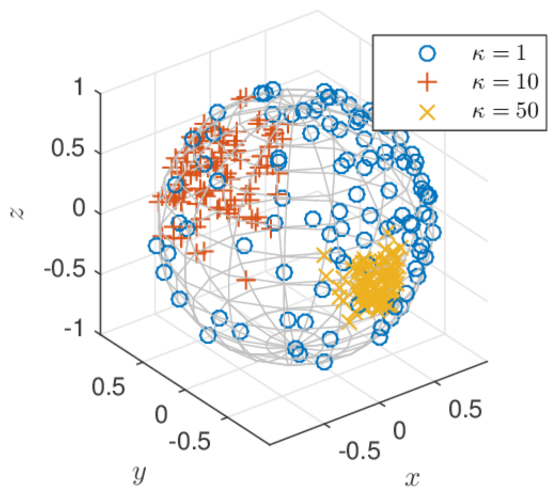

Figure 2.3: Illustrating change in spread of data points given different values of μ and κ in a vMF distribution[11]

the Cauchy distribution to compress data into visual representations because of this property [24]. The Cauchy distribution is similar to the Gaussian in having a location parameter μ and a scale parameter γ.

$$p(x) = \frac{1}{\pi\gamma}\left[\frac{\gamma^2}{(x-\mu)^2+\gamma^2}\right] \tag{2.11}$$

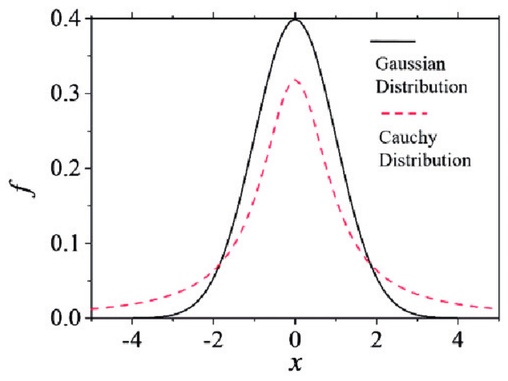

Figure 2.4: A comparison between the probability density functions of the Gaussian and Cauchy distribution

2.3 Deep Neural Networks

Next, let us explore the neural network architectures that we will use to construct our generative model for sentences.

2.3.1 Fully Connected Layer

A fully connected layer (also referred to as a dense or linear layer) is a specific type of neural architecture. The architecture transforms input of size m to an output of size n. It consists of a matrix of weights $w \in \mathbb{R}^m \times \mathbb{R}^n$ and a bias $b \in \mathbb{R}^n$ vector that linearly transforms a vector such that $y = wx + b$; both parameters are learned through back propagation. A fully connected layer can be used to transform the size of a vector and is particularly useful when compressing data. In-between layers, we add a non-linear activation function to learn more complex relationships within the data.

2.3.2 LSTM

Recurrent neural networks are a subset of neural network architectures that are designed to capture the notion of sequence data in a way that regular networks cannot. They do this by having a network loop that allows information to persist. Sentences are considered sequence data as they comprise of a set of words.

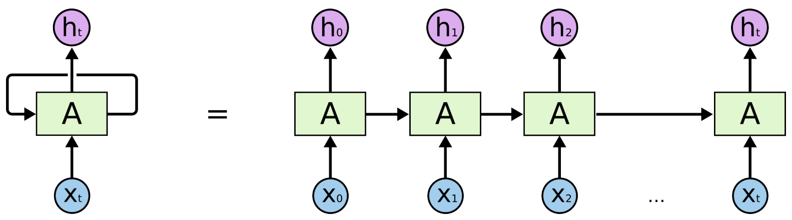

Figure 2.5: A recurrent neural network $\boldsymbol{A}$ that takes in an input x_t and the previous state to produce an output h_t [13]

The LSTM, introduced by Schmidhuber [8], is a specific type of recurrent neural networks designed to capture both short and long term dependencies in sequences data. A cell state C passes through the network allowing data to persist. At each time step, the network learns how to optimally use this state as well as update it for the next time step.

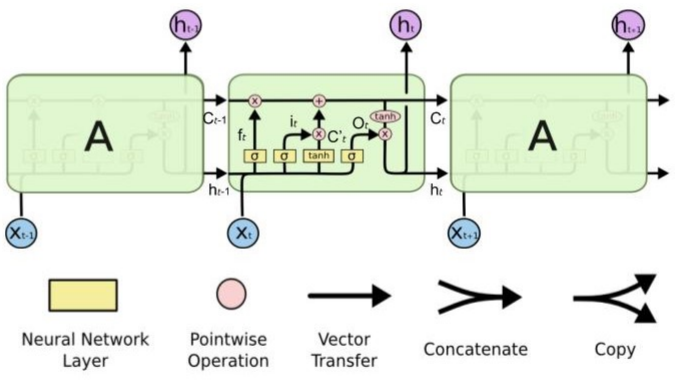

Figure 2.6: LSTM Network[13]

An LSTM cell has the following architecture: a dense layer with a sigmoid function to produce a value f_t between 0 and 1 known as the forget gate. The forget gate is used to control how much information is retained from our previous state C_{t-1}.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.12}$$

A dense layer with a *tanh* function layer creates a vector of new candidate values, C'_t, that could be added to the state. A sigmoid layer called the "input gate layer, i, decides a weighting between 0 and 1 for this new value C'_t. This allows the network to use x_t to update the state of the network.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.13}$$

$$C'_t = \sigma(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.14}$$

Combining this with our forget gate means that our state gets updated with the following equation at each time step t:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot C'_t \tag{2.15}$$

Finally, the architecture generates an output based on another network layer:

$$O_t = \sigma(W_O \cdot [h_{t-1}, x_t] + b_O) \tag{2.16}$$

$$h_t = O_t \cdot tanh(C_t) \tag{2.17}$$

Bidirectional LSTM

A natural extension of the LSTM architecture is to combine two LSTMs both feeding the data in opposite directions. This was introduced by Schuster et al. and is known as a bidirectional LSTM [19]. Figure 2.6 demonstrates a forward pass through the sequence whereas a bidirectional LSTM will run through a sequence with a forward pass $x_0, x_1, ..., x_n$ as well as a separate pass with the sequence reversed $x_n, x_{n-1}, ..., x_0$. The outputs from the forward pass h_{ft} and backward pass h_{bt} are then concatenated together to produce h_t. The bidirectional technique allows the network to take into account the context from both the past and the future in an infinite window to best produce an output. This is particularly useful in natural language processing (NLP) tasks because the network captures the entire sentence when looking at a specific word rather than just taking into consideration the relationship of the past tokens.

2.3.3 Autoencoders

An autoencoder is used to reconstruct high dimensional data from a learned latent representation and was first introduced by Rumelhart at al. [18]. This latent representation is often a much more compressed vector representation of the initial input data. This latent representation can be used to learn underlying generative features in the data.

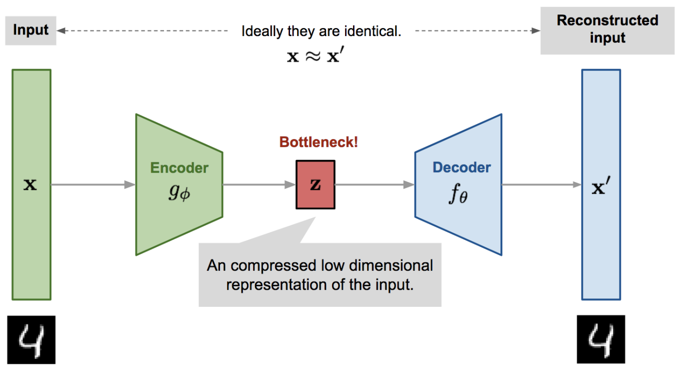
Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Figure 2.7: An autoencoder being used to reconstruct the drawing of digits[27]

The autoencoder consists of two neural architectures: the encoder and decoder with a fully connected layer in-between. These architectures can be anything from an LSTM to a convolutional neural network. Our encoder takes the input x and generates our latent representation z such that $z = g_\phi(x)$. Our decoder architecture then aims to reconstruct our initial input x using the vector z such that $x' = f_\theta(z)$. The reconstruction cost is the difference between x and x' and we train the model to minimise the squared difference averaged over a batch. Our cost function is given as:

$$\mathcal{L}(\phi, \theta) = \frac{1}{n}\sum_{i=1}^{n}(x^{(i)} - f_\theta(g_\phi(x^{(i)})))^2 \tag{2.18}$$

For $f_\theta(.)$ to perform a successful reconstruction of x, then our encoder $g_\phi(.)$ must create a useful representation for z.

2.3.4 Autoencoders in NLP

For certain NLP problems such as machine translation, LSTMs are not a good fit. Translation is not as simple as translating one word at a time; a sentence in English may be far shorter than one in French for example. We use autoencoders to solve these types of problems. The model consists of an LSTM encoder and LSTM decoder; input is encoded into our latent vector $z = h_n$ where x_n is the last element of a sequence. This encoder is often bidirectional to improve performance. The state is then passed to the decoder along with z as its input. During the decoding stage, we simply pass the previous time step's output as input without looking at the original data, this can be seen in Figure 2.8.

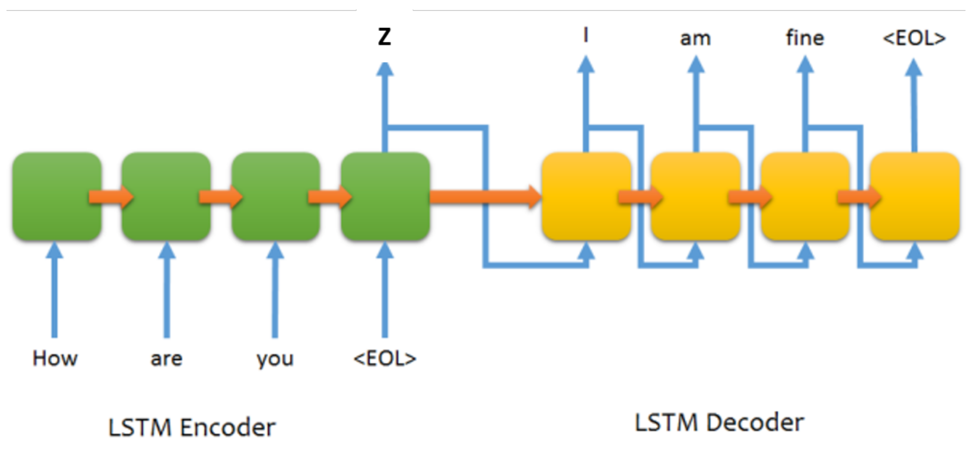

Figure 2.8: A sequence-to-sequence model trained to answer questions posed to it as input. The question is encoded into a vector z [17]

Attention

For long sequences of text, autoencoder models can struggle. Words at the start of the sentences go through many LSTMs cells when being encoded which means it can be very easy for key information to be lost. To attempt to preserve this key information during decoding, Vaswani et al. introduced attention [25]. Attention allows the decoder to additionally use parts of the initial input to help improve the quality of its decoding. We do not want to explicitly define what parts of the sentence may be useful so instead we introduce a set of learned parameters a for weightings of the encoder output at each time step. We then sum together the encoder outputs, h_i, multiplied with their associated weight to produce an associated context vector v_t. This context vector is appended to the input at each time step of the decoder.

$$v_t = \sum_{i=1}^{n} a_{t,i} \cdot h_i \tag{2.19}$$

Augmentation

Input augmentation is another approach to improve the performance of autoencoders. Augmentation assists with the persistence of important information during the decoding stage by concatenating a vector to the input at each time. This vector is often our encoded z but can be any piece of useful information.

2.4 Word Embeddings

Vectors are the input and output of a neural network; in NLP problems the input and output are usually supposed to be text. This means that we need a vector representation for the words in our vocabulary so that we can feed them as input into our networks. A naive approach would be to use one-hot-vectors to encode our words. A one-hot-vector is a vector with the length being the number of classes (in our case the number of words in our vocabulary). There is a 0 in every position except a single 1 in the index of the class. For example if $V : \{rock, paper, scissors\}$ then the associated one-hot-vectors are $\{100, 010, 001\}$. In one-hot-vectors there is no concept of word similarity which makes it far harder to solve NLP tasks. In addition to this, we run into memory issues very quickly on tasks with a large vocabulary. To overcome this problem, trained word embeddings have been introduced. These are vector representations of words designed to capture the idea that the distance between similar words should be less than words unlikely to appear in a similar context. For example, the vector for “computer” should be closer to the vector for “programming” than the vector for “king”. Multiple different approaches have been proposed to solve this problems but the conventional two leaders are either “GloVe” [15] or “word2vec” [22]. The trained embeddings for a vast vocabulary of words are publicly available to download and use. We use the GloVe embeddings because the approach for training seems more sensible; the GloVe embeddings have a 300 feature vector for each word in the vocabulary.

2.5 Similarity Measures

2.5.1 KullbackLeibler Divergence

Throughout this project, we will need to compare the similarity of probability distributions. This could be to measure the quality of an approximation or to see how well a calculated distribution matches our prior belief of what it should look like. We measure this using the KullbackLeibler (KL) divergence. If the KL divergence of two distributions is recorded as 0 then the two distributions are regarded as identical.

Discrete Probability Distributions

$$KL(p||q) = -\sum_{x \in \mathcal{X}} p(x)\ log\Big(\frac{p(x)}{q(x)}\Big) \tag{2.20}$$

Continuous Probability Distributions

$$KL(p||q) = \int_{-\infty}^{\infty} p(x)\ log\Big(\frac{p(x)}{q(x)}\Big)\, dx \tag{2.21}$$

$$= \mathbb{E}_p[log\ p(x)] - \mathbb{E}_p[log\ q(x)]$$

The KL divergence also has a couple of notable properties. The first is that $D_{KL}(p||q) \geq 0$ for all distributions p and q. As well as this, the KL divergence is not a symmetric measure: $D_{KL}(p||q) \neq D_{KL}(q||p)$.

2.5.2 Jaccard Distance

The Jaccard distance is often used as a threshold to find sentences that are similar within a training set. It is a measure of how similar two sentences $(X,\ Y)$ are in terms of the words they contain.

$$d_J(X, Y) = 1 - \frac{|\ X \cap Y\ |}{|\ X \cup Y\ |} \tag{2.22}$$

A large weakness with using this method is that two sentences with a very small Jaccard distance can be radically different in terms of the meaning. Consider the example sentences "The pizza was very good" and "The pizza was not very good" which has a distance of 0.17 but the two sentences are completely opposite in meaning [6].

2.5.3 Cosine similarity

The cosine similarity is a measure used to compare the similarity between two vectors by measuring the cosine angle between them. It is commonly used to compare word embeddings:

$$cos(\theta) = \frac{A \cdot B}{||A||||B||} \tag{2.23}$$

2.6 Summary

We now have covered all necessary theory required to begin to understand how we can extend the conventional autoencoder into the variational autoencoder. We have also explored surrounding theory in NLP to understand the practical method used in the two papers that we will be investigating as well as their mathematical underpinning. The Deep Neural Network section alone should be sufficient in gaining a thorough understanding of the outlined practical implementations in the next chapter.

Chapter 3

Project Execution

In the last chapter, we introduced key concepts in probability theory as well as deep learning to provide a technical background to the main body of this thesis. In this chapter we aim to understand and implement two different methods for generating sentences from the Yelp dataset training corpus. We will begin by focusing on how to extend the autoencoder architecture discussed in 2.3.3 into a variational autoencoder (VAE) [10] and then investigate how this architecture has been used by Bowman [2] and Guu [6]. After discussing the current work in detail as well as a practical implementation that we have designed, we will move onto extensions of their work by experimenting with the use of the von Mises-Fisher and Cauchy distribution with the VAE architecture.

3.1 Variational Autoencoder

We introduced the concept of the autoencoder in section 2.3.3. Autoencoders work well at compressing data into a latent representation z but they tend to overfit to the training set. Overfitting is when a model optimises well for the training data but struggles to generalise well for unseen data in the test set. In a conventional autoencoder, our input x is encoded into single points in our latent representation of z. This allows the network to easily over optimise for the provided training set rather than creating a meaningful representation. In the case of encoding sentences, a meaningful representation of z should learn about high level features such as grammar and topic. If we sampled z between two random sentences in our training set, we would expect to decode a sentence that makes grammatical sense. We would also expect the content to be some intuitive hybrid between the two sentences. When our input is encoded as fixed points are model is not forced to work in this more continuous space and if we performed the described sampling process, quite often we would decode nonsense.

To prevent this overfitting, Kingma outlined a new approach of adding a variational element to the autoencoder [10]. Instead of having input encoded into a single point, he proposed z should encoded as a distribution. Input is encoded into an elliptical region as rather than a singular point forcing our autoencoder to learn a more continuous representation of z. Although this is harder to optimise on the training set, the model will be able to generalise far better.

We specify a prior $p_\theta(z)$ of what we believe the shape our distribution of z will look like; for example $p_\theta(z) = \mathcal{N}(z; 0, I)$ is a common assignment. We have a likelihood $p_\theta(x|z)$ and a posterior $p_\theta(z|x)$ in our generative model.

$$p_\theta(x) = \int p_\theta(x|z)p_\theta(z)\ dz \tag{3.1}$$

$$p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)} \tag{3.2}$$

Calculating $p_\theta(x)$ would involve summing over all possible values of z but there are near infinite values; this means that calculating our posterior is intractable. The Autoencoding Variational Bayes (AEVB) algorithm aims to approximate this posterior by defining a recognition model $q_\phi(z|x)$ to approximate our intractable posterior. AEVB makes inference and learning of these type of models efficient by using a Stochastic Gradient Variational Bayes (SGVB) estimator to optimise a recognition model. We then extend the conventional autoencoder by replacing our deterministic encoder with the probabilistic recognition model. We sample a z value from the model and then use the decoder to reconstruct x.

3.1.1 Variational Bound

The distribution $p_\theta(z|x)$ is considered intractable so we have introduced the recognition model $q_\phi(z|x)$ as our approximation. We want to be able to quantify the quality of the approximation such that we can optimise ϕ to improve our recognition model. To quantify the difference between our distributions we use the KL divergence. The KL divergence aims to compare the information in the distributions p and q to quantify how similar they are.

$$KL(q||p) = \int q(z)\ log \frac{q(z)}{p(z)} dz \tag{3.3}$$

The problem is that we do not know exactly what the distribution of $p(z|x)$ looks like. However, we can have a prior $p_\theta(z)$ of what we think the latent space of z should look like. We perform the following steps to remove the dependency on $p(z|x)$.

$$\begin{aligned}
KL(q_\phi(z|x)||p_\theta(z|x)) &= \int q_\phi(z|x)\ log \frac{q_\phi(z|x)}{p_\theta(z|x)} dz \qquad (3.4)\\
&= \int q_\phi(z|x)\ log \frac{q_\phi(z|x)p_\theta(x)}{p_\theta(x,z)} dz\\
&= log\ p_\theta(x) + \int q_\phi(z|x)\ log \frac{q_\phi(z|x)}{p_\theta(x,z)} dz\\
&= log\ p_\theta(x) + \int q_\phi(z|x)\ log \frac{q_\phi(z|x)}{p_\theta(x|z)p_\theta(z)} dz\\
&= log\ p_\theta(x) + \mathbb{E}_{z\sim q_\phi}\left[log \frac{q_\phi(z|x)}{p_\theta(z)}\right] - \mathbb{E}_{z\sim q_\phi}[log\ p_\theta(x|z)]\\
&= log\ p_\theta(x) + KL(q_\phi(z|x)||p_\theta(z)) - \mathbb{E}_{z\sim q_\phi(z|x)}[log\ p_\theta(x|z)]
\end{aligned}$$

As specified in the previous section, $p(x)$ is considered intractable meaning that the dependency on log $p(x)$ makes the divergence computationally infeasible. The KL divergence can not be computed and therefore can not be optimised for. Instead we introduce the Evidence Lower Bound term (ELBO):

$$ELBO(x, \theta, \phi) = log\ p_\theta(x) - KL(q_\phi(z|x)||p_\theta(z|x)) \tag{3.5}$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}[log\ p_\theta(x|z)] - KL(q_\phi(z|x)||p_\theta(z))$$

Maximising ELBO is equivalent to minimising the KL divergence [1] and therefore allows us to optimise our parameters θ and ϕ. Usually we would use Monte-Carlo sampling to estimate the expectation over $q_\phi(z|x)$ but the variance of drawn samples will be too high. This prevents us from getting a good estimate in a reasonable amount of time. We need to alter our expectation such that we can back propagate with respect to ϕ reliably.

3.1.2 The SGVB estimator

Under certain conditions outlined in 3.1.3, we can reparameterise the random variable $z \sim q_\phi(z|x)$ using a differentiable transformation $g_\phi(\epsilon, x)$ where ϵ is an auxiliary noise variable sampled from a distribution. We use this reparameterisation trick for a simplified Monte-Carlo estimate. When training, we sample L values for ϵ from $p(\epsilon)$ and average the result. By separating out the stochastic process from ϕ, we have now made our function differentiable with respect to ϕ.

$$\mathbb{E}_{q_\phi(z|x)}[f(z)] = \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))] \approx \frac{1}{L}\sum_{l=1}^{L} f(g_\phi(\epsilon, x) \text{ where } \epsilon \sim p(\epsilon) \tag{3.6}$$

We apply this technique to our evidence lower bound to create the generic Stochastic Gradient Variational Bayes (SGVB) estimator (ELBO′).

$$ELBO'(x, \theta, \phi) = \frac{1}{L}\sum_{l=1}^{L} log\ p_\theta(x|z) - KL(q_\phi(z|x)||p_\theta(z)) \tag{3.7}$$

$$z = g_\phi(\epsilon, x) \text{ and } \epsilon \sim p(\epsilon)$$

The result is a loss function that is computationally tractable and measures both the reconstruction cost of our input x as well as ensuring that our recognition model holds close to our prior of how the latent space should look.

3.1.3 The Reparameterisation Trick

This reparameterisation is useful for our case since it can be used to rewrite an expectation w.r.t $q_\phi(z|x)$ such that the Monte-Carlo estimate of the expectation is differentiable w.r.t. ϕ. Here are the list of conditions required such that we can choose a differentiable function $g_\phi(.)$ and an auxiliary variable $\epsilon \sim p(\epsilon)$ for our $q_\phi(z|x)$ outlined by Kingma [10]:

1. **Tractable inverse CDF:** "In this case, let $\epsilon \sim U(0, I)$, and let $g_\phi(\epsilon, x)$ be the inverse CDF of $q_\phi(z|x)$. Examples: Exponential, Cauchy, Logistic, Rayleigh, Pareto, Weibull, Reciprocal, Gompertz, Gumbel and Erlang distributions."

2. **Analogous to the Gaussian example:** "For any location-scale family of distributions we can choose the standard distribution (with location = 0, scale = 1) as the auxiliary variable ϵ, and let $g(.) = \text{location} + \text{scale} \cdot \epsilon$. Examples: Laplace, Elliptical, Students t, Logistic, Uniform, Triangular and Gaussian distributions."

3. **Composition:** "It is often possible to express random variables as different transformations of auxiliary variables. Examples: Log-Normal (exponentiation of normally distributed variable), Gamma (a sum over exponentially distributed variables), Dirichlet (weighted sum of Gamma variates), Beta, Chi-Squared, and F distributions."

3.1.4 Gaussian Variational Autoencoder

Finally, we tie all of this together to illustrate a variational autoencoder neural architecture example using a Gaussian distribution [10]. We use a neural network as our probabilistic encoder $q_\phi(z|x)$ which is an approximation of posterior in our generative model $p_\theta(x, z)$. We want to optimise our parameters ϕ and θ with the AEVB algorithm that has been outlined.

We set a prior $p_\theta(z) = \mathcal{N}(0, I)$. We then create likelihood function $p_\theta(x|z)$ to be some multivariate Gaussian whose parameters are calculated from z. This is done by using a number of fully connected layers and eventually ending in a softmax layer to generate a distribution. The posterior $p_\theta(z|x)$ is intractable and therefore must be approximated to $q_\phi(z|x)$. If we assume that the true posterior is of Gaussian form, then our approximation is also set up to be a multivariate Gaussian.

$$q_\phi(z|x) = \mathcal{N}(\mu, \sigma^2 I) \tag{3.8}$$

μ and σ are outputs from an encoder architecture consisting again of a number of fully connected layers which are then connected to two layers of dimension J. ϕ is the set of all trainable weights for this encoding architecture. We can easily compute the KL divergence without estimation given our choice of $p_\theta(z)$.

$$-KL(q_\phi(z|x)||p_\theta(z)) = \frac{1}{2}\sum_{j=1}^{J}(1 + log\ \sigma_j^2 - \mu_j^2 - \sigma_j^2) \tag{3.9}$$

where J is the dimensionality of z and j denotes the index of a vector. Going back to equation 3.7, this provides us with our objective function for our variational autoencoder.

$$ELBO'(x, \theta, \phi) = \frac{1}{L}\sum_{l=1}^{L} log\ p_\theta(x|z) + \frac{1}{2}\sum_{j=1}^{J}(1 + log\ \sigma_j^2 - \mu_j^2 - \sigma_j^2) \tag{3.10}$$

$$\text{where } z = \mu + \sigma \cdot \epsilon \text{ and } \epsilon \sim \mathcal{N}(0, I)$$

We back propagate with respect to both θ and ϕ to train our recognition and decoder respectively.

3.1.5 Summary of Autoencoders

Both conventional and variational autoencoders are designed to learn a representation of a latent variable z using unsupervised learning on some input. Points are encoded using a learned posterior recognition model $q_\phi(z|x)$; this distribution is normally a diagonal Gaussian. Conventional autoencoders will learn codes as a single point in latent space whereas the VAE learns codes as a distribution (elliptic regions) in the space of the latent variable z. A prior $p_\theta(z)$ is used to regularise this learned distribution; this prevents the variance becoming vanishingly small and turning the VAE into a conventional autoencoder. Learning codes as elliptical regions tends to create models that generalise better because we have a more complete representation of z. The architecture is illustrated in Figure 3.1

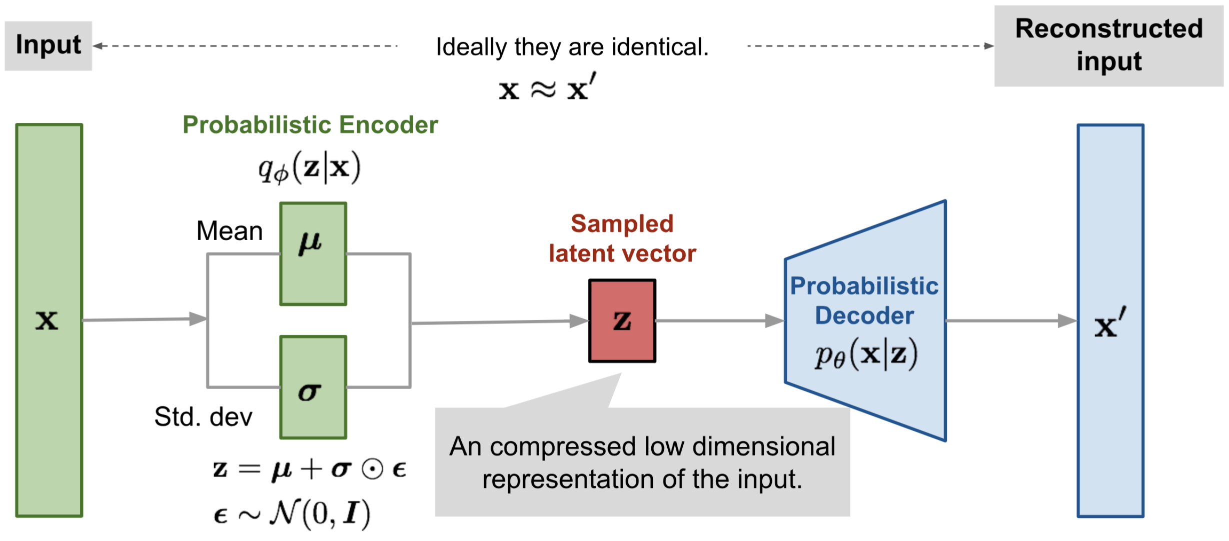
Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
$\mathbf{x}$
Mean
$\boldsymbol{\mu}$
Std. dev
$\boldsymbol{\sigma}$
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

Figure 3.1: A diagram of the architecture of a variational autoencoder [27]

3.2 Generating Sentences in Continuous Space

Standard recurrent neural network language models generate sentences one word at a time but do not work with a global representation of a sentence. Bowman decided to apply the theory from the variational autoencoder to create a model to generate sentences [2]. The learned latent representation allowed the model to learn more complex features such as style and topic.

3.2.1 Problem Statement

The model aims to generate diverse sentences that make semantic and syntactic sense. The hope is that by using a latent variable z, we will be able to represent high level features of sentences. Sampling between two different sentences should provide coherent sentences with an intuitive mix between high level features of the sampled sentences. This is something that previous models have struggled to capture.

3.2.2 Model Architecture

Sentences are encoded using a basic two layer LSTM architecture. The final output is then used as input to two dense linear layers which represent our function $g_\phi(.)$ specified in the last section. One linear layer represents μ, and the other σ, which we will use to parameterise our Gaussian distribution $q_\phi(z|x)$. We sample z from this distribution which we then decode using a three layer LSTM architecture. The authors of the paper specified that they tried numerous extensions on the decoder but found they provided no significant improvement.

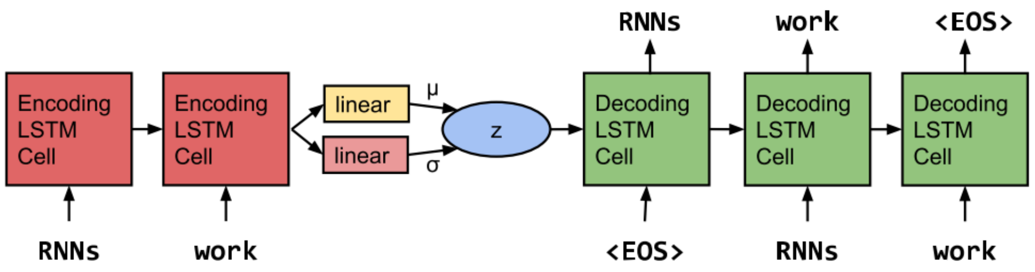

Figure 3.2: A figure taken from the Bowman et al [2] illustrating the model architecture of the VAE for sentences [2]

3.2.3 Approach

The objective function that we aim to optimise is taken from equation 3.10 with a standard Gaussian prior:

$$\mathcal{L}(x,\theta,\phi) = \frac{1}{L}\sum_{l=1}^{L} log\ p_\theta(x|z) + \frac{1}{2}\sum_{j=1}^{J}(1 + log\ \sigma_j^2 - \mu_j^2 - \sigma_j^2) \tag{3.11}$$

$$\text{where } p_\theta(z) = \mathcal{N}(0,1),\ z = \mu + \sigma \cdot \epsilon \text{ and } \epsilon \sim \mathcal{N}(0,I)$$

Our parameters for the encoder and decoder (ϕ and θ) are trained concurrently using stochastic gradient ascent. It is a common approach to train ϕ and θ alternately but Bowman et al. addressed some optimisation problems in an attempt to avoid this.

3.2.4 Optimisation Problems

A model that uses the variational element to the autoencoder in a useful way will have a non-zero KL term in the loss function. It will have to utilise the variational properties to learn a better encoding of our latent variable z. Unfortunately, with the current definition of our model architecture and approach, this proposed model in the vast majority of cases will incorrectly optimise the KL term to zero. This is because in the initial stages, it is very easy for the model to ignore the encoder output and purely focus on greedily optimising for the KL term. The result is that little to no information is passed through from the encoder to the decoder and the system reaches an undesirable stable equilibrium. Two methods have been proposed to address this undesired KL divergence shrinking problem.

KL Cost Annealing

A simple approach to stop the issue is by having a variable weight function f for the KL term. f is a pre-defined function that takes in the training step, s, and returns a value for the weight of the KL term in the cost function. This changes the cost function to the following:

$$\mathcal{L}(x,\theta,\phi,s) = \mathbb{E}_{q_\phi(z|x)}[log\ p_\theta(x|z)] - f(s) \cdot KL(q_\phi(z|x)||p(z)) \tag{3.12}$$

$$\text{where } p(z) = \mathcal{N}(0,1)$$

The paper recommends using a sigmoid annealing function for f but does not specify specific values for this. The details of the chosen function will be outlined in the implementation section in this chapter.

Word Dropout

In addition to this KL annealing term, the authors experimented with weakening the decoder to help act as a regulariser for the network. This is done by removing some information about the input sentence x using word dropout. We use the additional fixed parameter $k \in [0,1]$. called *keep probability*. Denote UNK as unknown word token and define the dropout function:

$$d(x,k) = \begin{cases} x & \text{if } r \leq k \\ \langle \text{UNK} \rangle & \text{otherwise} \end{cases} \tag{3.13}$$

$$\text{such that } r \sim U([0,1])$$

We apply a function d to each element of the input sequence individually to change input at some time steps to the unknown token.By losing information about the input, it forces the network to learn how to generalise better and perform better inference. To perform well it will be forced to make better use of the latent variable z and therefore will use the distribution better.

3.2.5 Algorithm for Training

1. Create minibatch x from training set X
2. Pad and replace unknown words in x
3. Embed x using GloVe pre-trained embeddings
4. Encode x using LSTM encoder
5. Calculate μ, σ^2 using two different deep layers
6. Compute: $z = \mu + \sigma \cdot \epsilon \quad \epsilon \sim \mathcal{N}(0, I)$
7. Decode z using LSTM decoder
8. Use softmax layer to calculate word probability at each time step
9. Compute: $D_{KL} = \frac{1}{2} \sum_{j=1}^{J} (1 + log\ \sigma_j^2 - \mu_j^2 - \sigma_j^2)$
10. Set the annealing cost function: $w = f(s)$
11. Compute the objective function: $\mathcal{L} = log\ p(x) + w \cdot D_{KL}$
12. Use back propagation to optimise all model parameters

3.3 Prototype and Edit Model

Most neural models for generating text work in a left to right manner from scratch.. These models often lead to favouring common utterances and lack diversity. Trying to deviate from this has frequently compromised the grammar of generated sentences. The prototype and edit model takes a novel approach to sentence generation by combining a sampled prototype sentence x' along with latent vector z representing an edit [6]. This proposed model focuses more on editing existing sentences rather than creating a generative model that builds sentences from scratch.

3.3.1 Problem Statement

The model aims to incorporate the idea of semantic smoothness by only applying small controlled edits to a given prototype sentence; multiple edits are used to deviate further away from the initial sampled prototype. The model also aims to capture consistent edit behaviour implying that the same edit vector z should apply the same semantic change to two entirely different sentences. We aim to maximise the likelihood of a sentence x being produced such that:

$$p(x) = \sum_{x' \in X} p(x|x')p(x') \tag{3.14}$$

$$p(x|x') = \mathbb{E}_{z \sim p(z)}[p_\theta(x|x', z)] \tag{3.15}$$

3.3.2 Approach

The neural editor $p_\theta(x|x', z)$ aims to be trained by maximising the marginal likelihood specified in 3.14 via gradient ascent but the objective function is intractable. A sum over all possible prototypes x' is expensive and the expectation over the prior $p(z)$ has no closed form so two approximations are made.

Approximating the sum on prototypes

The sum is lower bounded by only summing over x' that are semantically similar to x. Prototypes that are not semantically similar to x' will have extremely small probabilities and therefore will not have much influence in the resulting distribution. Removing them from consideration will make the sum computationally feasible. Training pairs are considered semantically similar if their Jaccard distance is under a certain threshold.

The neighbourhood set for a given prototype sentence is defined such that

$$\mathcal{N}(x) \overset{def}{=} \{x' \in \mathcal{X} : d_J(x, x') < 0.5\} \tag{3.16}$$

Consider the lower bound of $\log p(x)$:

$$log\ p(x) = log[\sum_{x' \in X} p(x|x')p(x')] \geq log[\sum_{x' \in \mathcal{N}(x)} p(x|x')p(x')] \tag{3.17}$$

because $\mathcal{N}(x) \subseteq \mathcal{X}$ and we are only summing over $\mathcal{N}(x)$, which reduces the size of $\log p(x)$. Next denote $R(x) = log(\frac{|\mathcal{N}(x)|}{|\mathcal{X}|})$.

Because prototype sentences are uniformly sampled we have $p(x') = \frac{1}{|\mathcal{X}|}$. Substituting back into 3.17 we get:

$$log(x) \geq log\left[\mid \mathcal{N}(x) \mid^{-1} \sum_{x' \in \mathcal{N}(x)} p(x|x') \right] + R(x) \geq \mid \mathcal{N}(x) \mid^{-1} \sum_{x' \in \mathcal{N}(x)} log\ p(x|x') + R(x), \tag{3.18}$$

where we used Jensen equality to take $\mid \mathcal{N}(x) \mid^{-1}$ out of the log to lower bound 3.18. Because $\mid \mathcal{N}(x) \mid$ is a constant over x, we take our lower bound of $\log p(x)$ to be:

$$LEX(x) \overset{def}{=} \sum_{x' \in \mathcal{N}(X)} log\ p(x|x') \tag{3.19}$$

$LEX(x)$ is still computationally intractable because $p(x|x')$ still requires the expectation over the edit prior $p(z)$ which has no closed form 3.15

Approximating expectation on edit vectors

We aim to approximate the intractable equation:

$$p(x|x') = \mathbb{E}_{z \sim p_\theta(z)}[p_\theta(x|x', z)], \tag{3.20}$$

where the term $p_\theta(z)$ has no closed form.

A common approach in this situation would be to use Monte-Carlo sampling $z \sim p(z)$. Unfortunately in this case, the variance of the approximations will be unacceptably high to provide a reliable estimate. This is because $p_\theta(x|x', z)$ will be almost zero for nearly all sampled z. Instead an inverse neural editor $q_\phi(z|x', x)$ is defined to concentrate the probability of an edit on a few important values of z. We then combine this with the variational bound discussed in 3.1.1 to provide a tractable approximation.

$$log\ p(x|x') \geq \mathbb{E}_{z \sim q_\phi(z|x,x')}[log\ p_\theta(x|x', z)] - KL(q_\phi(z|x, x')||p_\theta(z)) \tag{3.21}$$

$$\overset{def}{=} ELBO(x, x', \theta, \phi)$$

Combining with the lower bound from LEX(x), this provides us with the final objective function that we aim to maximize:

$$\mathcal{L}(x, \theta, \phi) = \sum_{x' \in \mathcal{N}(\mathcal{X})} ELBO(x, x', \theta, \phi)$$

This objective is optimised using stochastic gradient ascent with respect to θ and ϕ alternately.

3.3.3 Model Architecture

Neural Editor $p_\theta(x|x', z)$: Sequence-to-sequence model with attention. The initial sequence being taken as input is embedded using the GloVe word embeddings. The edit vector z is concatenated at every time step to the input of the decoder. The encoder is a 3 layer bidirectional LSTM and the decoder is a 3 layer LSTM.

Edit Prior $p_\theta(z)$: We sample the edit vector z from the prior by first sampling its scalar length $z_{norm} \sim U(0, 10)$ and then sampling its direction z_{dir} (a unit vector) from the uniform distribution on the unit sphere. We define $z = z_{norm} \cdot z_{dir}$

Inverse Neural Editor $q_\phi(z|x, x')$: The inverse neural editor takes in a prototype sentence and a similar sentence and then generates a distribution $q_\phi(z|x, x')$ where z is the edit vector. We define a function f that generates an edit vector:

$$f(x, x') = \sum_{w \in I} \Phi(w) \ \oplus \ \sum_{w \in D} \Phi(w), \tag{3.22}$$

where I is the set of inserted words by the edit step and D is the set of deleted words by the edit operation. Φ is the embedding mapping which transforms the word w into the word vector $\Phi(w)$.

Recall that $z = z_{dir} \cdot z_{norm}$. We inject the noise around the edit vector and model the directional part and normalising part of the recognition model as:

$$q_\phi(z_{dir} \mid x', x) = vMF(z_{dir}; f_{dir}, k) \tag{3.23}$$

$$q_\phi(z_{norm} \mid x', x) = U(z_{norm}; [f_{norm}, f_{norm} + \epsilon]) \tag{3.24}$$

$$z = z_{dir} \cdot z_{norm} \tag{3.25}$$

The von Mises-Fisher distribution (vMF) is an unconventional choice of distribution for a variational autoencoder; a Gaussian being far more common choice. Edit vectors are sums of word vectors and cosine distances are often used to measure distances between word vectors. Encoding distances between edit vectors by the cosine distance seems like a sensible approach. The vMF distribution captures this idea, as the log likelihood decays with cosine similarity [6].

3.3.4 Sampling from von Mises-Fisher

The authors of the Prototype and Edit model provided the following pseudo-code of how to use rejection sampling in the vMF distribution [5]. They have used a reparameterisation trick which will be discussed in detail in section 3.5.2. The procedure is the following:

1. Sample $v \sim U(\mathcal{S}^{m-2})$
2. Sample $\omega \sim g(\omega|\kappa, m)$:
   - $a = \frac{1}{4}(m - 1 + 2\kappa + \sqrt{4\kappa^2 + (m-1)^2})$
   - $b = \frac{-2\kappa + \sqrt{4\kappa^2 + (m-1)^2}}{m-1}$
   - $d = \frac{4ab}{1-b} - (m-1)ln(m-1)$
   - Repeat until condition met (rejection sampling):
     - $\epsilon \sim Beta(\frac{m-1}{2}, \frac{m-1}{2})$
     - $\omega = \frac{1-(1+b)\epsilon}{1-(1-b)\epsilon}$
     - $t = \frac{2ab}{1-(1-b)\epsilon}$
     - $u \sim U(0, 1)$
     - Accept ω iff $(m-1)ln(t) - t + d > ln(u)$
3. $z' = (\sqrt{1-\omega^2} v^T)^T$
4. $R = Householder(e_1, \mu)$

- $e_1 = [1\ 0\ 0\ 0...]$
- $r = (e_1 - \mu)/||e_1 - \mu||$
- $R = I - 2rr^T$, where I is the identity matrix and T is transpose

5. $z = Rz'$

where $U(\mathcal{S}^{m-2})$ is the uniform distribution over the $m-2$ hypersphere. We can sample from it by generating a random multivariate vector from sampling $U[0,1]$ at each index and then normalising the length to one.

3.3.5 Differentiating Inverse Neural Editor

We begin by introducing some new notation for clarity

$$\mathcal{L}_{gen} = \mathbb{E}_{z\sim q_\phi(z|x,x')}[log\ p_\theta(x|x', z)]$$

$$\mathcal{L}_{KL} = KL(q_\phi(z|x,x')||p_\theta(z)) \tag{3.26}$$

$$\mathcal{L}(x, \theta, \phi) = \mathcal{L}_{gen} - \mathcal{L}_{KL}$$

For our inverse neural editor we want to maximise $\mathcal{L}(x, \theta, \phi)$ with respect to ϕ.

$$\nabla_\phi \mathcal{L}(x, \theta, \phi) = \nabla_\phi \mathcal{L}_{gen} - \nabla_\phi \mathcal{L}_{KL} \tag{3.27}$$

Calculating $\nabla_\phi \mathcal{L}_{gen}$

We use the reparameterisation trick from section 3.1.3. We rewrite $z \sim q_\phi(z|x,x')$ as $z = g_\phi(\epsilon, x, x')$ and $\epsilon \sim p(\epsilon)$ allowing us to rewrite $\nabla_\phi \mathcal{L}_{gen}$ as:

$$\nabla_\phi \mathcal{L}_{gen} = \mathbb{E}_{\epsilon\sim p(\epsilon)}[\nabla_\phi\ log\ p_\theta(x|x', g_\phi(\epsilon, x, x'))] \tag{3.28}$$

We know that g_ϕ is differentiable with respect to ϕ. Bringing ∇_ϕ inside the brackets allows us to calculate $\nabla_\phi \mathcal{L}_{gen}$ using standard back propagation.

Calculating $\nabla_\phi \mathcal{L}_{KL}$

We know that:

$$\mathcal{L}_{KL} = KL(q_\phi(z_{norm}|x,x')||p(z_{norm})) + KL(q_\phi(z_{dir}|x,x')||p(z_{dir})) \tag{3.29}$$

The prior of both terms are unconventionally chosen such that they are not dependant on ϕ.

$$KL(q_\phi(z_{norm}|x,x')||p(z_{norm})) = KL(U(f_{norm}, f_{norm} + \epsilon)||U(0, 10))$$

$$KL(q_\phi(z_{dir}|x,x')||p(z_{dir})) = KL(\text{vMF}(f, \kappa)||\text{vMF}(f, 0))$$

Guu et al. state that the the differential of first term is trivially 0 with respect to ϕ [6]. For the second term, μ is calculated using ϕ but is present in both the prior and the approximation and has no bearing. The term κ is a fixed hyperparameter and therefore does not relate to ϕ. This means that $\nabla_\phi \mathcal{L}_{KL} = 0$ for all input. The rationale for this is allowing us to directly control the effect of $\mathcal{L}_{KL}$ on the system by tuning hyperparameters. We can encourage a large $\mathcal{L}_{KL}$ with high values of κ and ϵ that will generalise better but will find it tougher to learn and vice versa. This is an approach designed to stop optimisation problems mentioned in the last section that led to the KL term reducing to 0.

3.3.6 Algorithm for Training

1. Create minibatch x' from training set X
2. Pad and replace unknown words in x'
3. Embed x' using GloVe pre-trained embeddings
4. Encode x' using LSTM encoder to e
5. Generate insertion and deletion sets I, D
6. $f(x, x') = \sum_{w \in I} \Phi(w) \ \oplus \ \sum_{w \in D} \Phi(w)$
7. Generate distribution $q_\phi(z|x, x')$
8. Sample z
9. Decode $z \oplus e$ using LSTM decoder
10. Softmax layer to calculate word probability at each time step
11. $\mathcal{L} = log\ p(x)$
12. Optimise parameters for encoder and decoder architecture using back propagation with respect to $\mathcal{L}$
13. Repeat steps 4-11 and optimise parameters for Φ with respect for newly calculated $\mathcal{L}$ using back propagation

3.4 Practical Implementation Details

We have now outlined in detail the theoretical intuition behind the approach in both papers. In this section, we will go into more detail expanding the finer details of a practical implementation of the two models. We will begin by discussing the functionality that is shared by both models in terms of the neural architecture and other useful functions and classes. We will then illustrate more specific parts of the code for each model and be explicit with the values for all the hyperparameters. We will use the Python programming language with the Tensorflow machine learning framework. It is worth noting that there is publicly available code for the Prototype and Edit model written using the PyTorch framework from Guu et al [5]. This code utilised a number of downloaded third party libraries and had to be run within a docker container for all environments and directories to work well. The code was well designed to replicate the results of the paper but was extremely difficult to manipulate and debug. Given that the Bowman model would need to be programmed from scratch it made more sense to construct this and then extend it to the Prototype and Edit model. Furthermore, if we design both with a lot of shared architecture, then we can provide a far better comparison between them. The Prototype and Edit code has been used for design inspiration in places but all work detailed here and submitted is my own work unless mentioned otherwise. We will use the published solution as a benchmark to evaluate the relative performance of the implemented solution.

3.4.1 Shared Functionality

Token Embedder

The Token Embedder is a class designed to perform the necessary pre and post processing of our plain-text data. Before being fed to the network, our plain-text sentences must first be converted into their GloVe word embeddings. We build a large dictionary of vocab based on the publicly available GloVe embeddings to convert text efficiently. Words not found in the vocabulary are replaced with the unknown embedding and sentences are padded using a pad embedding so that all sentences are a fixed length. Following processing, the token embedder is used to convert word indexes produced by the architecture into plain-text generated sentences.

LSTM

We implemented a new abstract LSTM class. This class simplifies the extensive set of inbuilt Tensorflow functionality to create a new LSTM class with only the required functionality. The constructor takes multiple parameters and flags to build custom LSTM configurations easily. Its main purpose is to ensure consistency in both models and to allow very easy experimentation with different hyperparameters. The length of un-padded sentences are held and used by the LSTM class on each forward pass. This is used to ensure that the padding of sentences is not used to train the weights of the architecture. We use the Tensorflow functionality for building the underlying simple cell as opposed to creating this completely from scratch.

Encoder

The encoder class was built as a further abstraction over the LSTM class and is designed to take in a batch of embedded sentences and produce a batch of encoded z vectors. The state of the architecture is also returned so that it can be used in the decoding stage if necessary.

Decoder

Similarly to the encoder, the decoder architecture is built on top of the LSTM class with more specific functionality. The decoder is first initialised with just a basic implementation but functions are provided to add the attention and augmentation extensions. Augmentation requires a user to call a function, passing the tensor they would like to augment the decoder with as an argument. This tensor is then concatenated to the input of the decoder at each time step. There is inbuilt Tensorflow functionality for attention but it does not provide the level of manipulation required so we implemented it from scratch. We initialise a set of weights for each time step; after a sentence has been encoded we use the outputs from the encoder to calculate attention values by multiplying the weights by the encoder output. A softmax layer controls the weights for the attention to ensure that particular input does not dwarf the effect of any other features of the decoder such as our z vector.

The key difference between the encoder and the decoder is that the decoder is set up to run one time step at a time such that the decoded output can be fed back in as input. The encoder uses inbuilt Tensorflow functions to pass the batch of input in all at once. The decoder can be initialised using the encoder's state or a fresh state.

Autoencoder

The autoencoder ties together the encoder and decoder architecture into one simple model that is easy for users to interact with. It takes a large number of hyperparameters making it simple to try different configurations for an architecture. The variational component can be added using designed distribution functions. In LSTM autoencoders, it is often common to pass the state of the encoder to the decoder. This seemed like a poor approach in the Bowman model as the model should be able to generate sentences exclusively from z. By allowing the encoder to pass its state, we would lose the ability to generate sentences randomly by sampling z. In the prototype and edit model, we are training to edit the sentence rather than recreate it so it makes sense to pass the state to the decoder.

Loss Function

The cost function for both architectures involves calculating the log probability of the target sentence. This is done by multiplying the probability of the correct word at each time step and then taking the log. The problem with this approach is that at early training steps, the probability is so small that they were being rounded to 0. Instead we utilise the fact that:

$$log(\prod_{i=0}^{N} p(w_i) + \epsilon_1) = \sum_{i=0}^{N} log\ (p(w_i) + \epsilon_2) \tag{3.30}$$

We use a tiny value ϵ to prevent the result from the input of the log being too small which would result in ∞. This approach still causes problems because the optimiser tended to focus on setting all decoded words to the pad token as this provided relatively good early performance. To combat this, we add a mask which set $log\ (p(w_i) + \epsilon_2)$ to 0 if w_i is the pad token.

Greedy Decoder

During the training process, it is useful to be able to see the quality of the sentences that are being produced rather than just observing the loss. We implement a simple greedy decoder that produced the most likely sentence in plain-text, enabling visual debugging and monitoring progress. Decoded sentences often had repeating words caused by a requirement to reach a padded length. When decoding sentences to be displayed we remove any repeating sequences of words.

3.4.2 Generating Sentences in Continuous Space

Training Set

The training set for this model was a long list of unique plain-text sentences taken from yelp reviews. Each sentence example is separated by a new line with there being an 80/10/10 training, validation, test split.

Architecture

When implementing the architecture we make slight changes to the hyperparameters specified in the paper to gain stronger results for our dataset. For the encoder, we use a bidirectional 3 layer LSTM with 512 hidden units. For the decoder, we use a 3 layer LSTM with 512 hidden units. We augment the data at each time step with z. For the variational aspect, we use the conventional prior $p_\theta(z) = \mathcal{N}(0, 1)$. Words were embedded using the GloVe with sentences being padded to a length of 15. To handle the optimisation problems we add a sigmoid KL cost annealing that reaches 1 after roughly 5000 steps. The model is trained for 50000 steps with a learning rate of 0.001 on batches of 64. We use a word dropout of 0.05.

Optimisation

The parameters for our encoder ϕ and decoder θ are trained simultaneously at each time step s using the Adam optimiser to maximise our cost function:

$$\mathcal{L}(x, \theta, \phi, s) = \frac{1}{B} \sum_{i=0}^{B} m(log\ (p(w_i) + \epsilon)) - f(s) \cdot KL(q_\phi(z|x)||p(z)), \tag{3.31}$$

where B is our batch size, f is our KL cost annealing sigmoid and m is our masking function such that:

$$m(log\ (p(w_i) + \epsilon)) = \begin{cases} 0 & \text{if } w_i = \langle \text{PAD} \rangle \\ log\ (p(w_i) + \epsilon) & \text{otherwise} \end{cases}$$

KL Cost Annealing

Bowman et al. recommended using a sigmoid shaped function for the weight for the KL divergence [2]. The spread of this sigmoid varies depending on how many steps it takes the KL divergence to stabilise. We experimented with the model without the annealing schedule and found that the KL divergence stabilised around step 5000.

Word Dropout

Word dropout is applied at every training step to the batch of training examples. This means that words are randomly replaced with the unknown label every time the same sentence is seen. Sentences from the testing or validation set do not have any word dropout applied to them when predictions are made.

3.4.3 Prototype and Edit Model

Training Set

As mentioned in section 3.3.2, we approximate the sum over all possible target sentences by only using training example within a Jaccard threshold. This is a computationally expensive operation but Guu et al. have already prepared the dataset for the Yelp dataset. The format of the dataset is the source

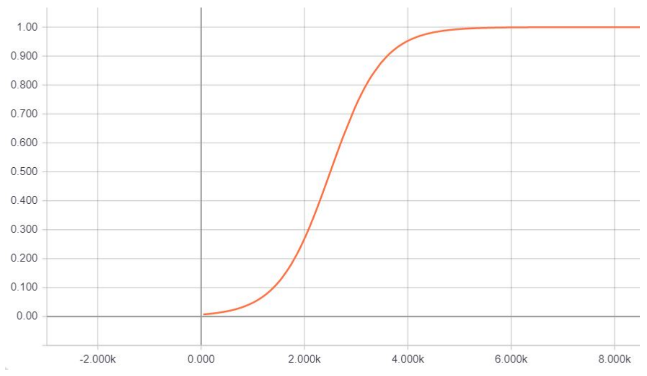

Figure 3.3: KL weight term used in the Bowman model over number of steps

sentence and associated target sentence that are separated by a tab. This dataset is split into training, validation and testing in roughly an 80/10/10 split.

Architecture

Editor
The encoder architecture was a bidirectional 3 layer LSTM with 256 hidden units. The decoder architecture was an 3 layer LSTM with 512 hidden units and had attention and augmentation. The attention vector was passed through a linear layer to be a 128 length vector for each training example. The input at each time was augmented with an "agenda" which is an 128 sized vector that is generated using a linear layer from the concatenation of the edit vector and the encoded prototype sentence. Words were embedded using the GloVe with sentences being padded to a length of 15. The model was trained for 15000 steps with a learning rate of 0.001 on batches of 128.

Inverse Neural Editor
The inverse neural editor takes in a batch of insertions and deletions and builds a batch of edit vectors that have been sampled from the vMF distribution. As mentioned in section 3.3.3 the function for this transformation before noise is added is:

$$f(x, x') = \sum_{w \in I} \Phi(w) \; \oplus \; \sum_{w \in D} \Phi(w), \tag{3.32}$$

where w are the plain-text words from the insertion set I and deletion set D. We want to utilise the GloVe word embeddings but also allow these to be altered to specialise in this new problem of editing. We set up the definition of Φ in the following way:

$$\Phi(w) = a_\phi(t_{GloVe}(w)) \tag{3.33}$$

with t being a token embedder that uses loaded trained embeddings to convert a word into its associated embedding. a_ϕ is a linear dense layer with learned parameters ϕ. This also allows us to change our edit vector to any arbitrary shape, 128 seemed like a sensible size. The sampling process required the hyperparameters ϵ and κ which were set to 0.1 and 100 respectively. The code to sample from the created vMF distribution was taken from the authors publicly available code.

Optimisation

As mentioned in section 3.3.5, $\nabla_\phi \mathcal{L}_{KL} = 0$ and therefore can be ignored from the cost function. We use the same cost function and masking trick as mentioned earlier in this section to improve training, this provides us with:

$$\mathcal{L}(x, x', \theta, \phi) = \frac{1}{B} \sum_{i=1}^{B} m(log\ (p_\theta(x|x', g_\phi(\epsilon, x, x')) + \epsilon)) \tag{3.34}$$

We alternately train the parameters θ and ϕ, meaning that every train step consists of two runs of back propagation on the same batch. On the first run, only θ is updated, which is all the parameters used in our encoding and decoding LSTMs. On the second run, we update only ϕ, which is just the weights used in the linear layer of Φ.

3.5 Von Mises-Fisher Variational Autoencoder

The use of the vMF distribution over the Gaussian distribution in the Prototype and Edit model was an unconventional choice but one with sound theoretical underpinning. The large weakness in this approach however was the fixed κ term used when training. Guu argued that by keeping this term fixed we could easily control the trade off between $\mathcal{L}_{KL}$ and $\mathcal{L}_{gen}$. Although this point has some merit, it would be far better to follow the theory outlined by Kingma and allow the model to learn this balance itself. Davidson et al. proposed an in-depth design for a vMF VAE which they named the $\mathcal{S}$-VAE [3]. They outline in detail why the Gaussian distribution can often be a weak choice of distribution for certain types of data, these reasons have been discussed in Chapter 2. In this section, we introduce the first ever variational autoencoder to represent text using a learned von Mises-Fisher representation.

3.5.1 KL Divergence

When using the Gaussian distribution, we used two dense layers to calculate μ and σ. If we selected the prior to be the common choice of $\mathcal{N}(0, I)$ then we could easily calculate our KL term, $KL(q(z|x)||p(z) = -\frac{1}{2}\sum_{j=1}^{J}(1 + log\ \sigma_j^2 - \mu_j^2 - \sigma_j^2)$. When using the vMF distribution, we set our prior $p(z)$ to be $vMF(\mu, 0)$ which is a uniform distribution over the hypersphere. We then learn our concentration parameter κ for a given input. We have:

$$KL(q(z|x)||p(z)) = KL(vMF(\mu,\kappa)||vMF(\mu,0)), \tag{3.35}$$

$$KL(vMF(\mu,\kappa)||vMF(\mu,0)) = \kappa\frac{\mathcal{I}_{m/2}(\kappa)}{\mathcal{I}_{m/2-1}(\kappa)} + log\ \mathcal{C}_m(\kappa) - log\Big(\frac{2\pi^{m/2}}{\Gamma(m/2)}\Big)^{-1}, \tag{3.36}$$

$$\mathcal{C}_m(\kappa) = \frac{\kappa^{m/2-1}}{(2\pi)^{m/2}\mathcal{I}_{m/2-1}},$$

where m is our dimension of z, $\mathcal{I}_v$ denotes the modified Bessel function of the first kind at order v and Γ denotes factorial [3].

3.5.2 Reparameterisation Trick

As discussed in section 3.1.2, we need to calculate $\mathbb{E}_{q_\phi(z|x)}[f(z)]$ but using Monte-Carlo estimates will provide unacceptably high variance. In the case of the Gaussian distribution, we introduce an auxiliary noise variable ϵ using a reparameterisation trick. Unfortunately this trick is not applicable for the von Mises-Fisher distribution. Naesseth et al. proposed a reparameterisation trick using rejection sampling that can be used for the vMF distribution [12].

Reparameterisation

We want to calculate the $\mathbb{E}_{q_\phi(z|x)}[f(z)]$, we start with $\mathbb{E}_{g(\omega|\phi)}[f(z)]$. We introduce the auxiliary noise variable $\epsilon \sim p(\epsilon)$ with $\omega = h(\epsilon, \phi)$. We denote our reparameterisation with $r(\omega|\phi)$ with $\phi = (\mu, \kappa)$. We define:

$$\pi_1(\epsilon|\phi) = p(\epsilon)\frac{g(h(\epsilon,\phi)|\phi)}{r(h(\epsilon,\phi)|\phi)} \tag{3.37}$$

If our reparameterisation is identical to the original function, we take $\pi_1(\epsilon|\phi) = p(\epsilon)$:

$$\mathbb{E}_{g(\omega|\phi)}[f(\omega)] = \mathbb{E}_{\pi_1(\epsilon|\phi)}[f(h(\epsilon,\phi))] \tag{3.38}$$

We can then take then take the differential inside the expectation such that we can compute the gradient using the log derivative trick [12]:

$$\nabla_\phi\mathbb{E}_{g(\omega|\phi)}[f(\omega)] = \nabla_\phi\mathbb{E}_{\pi_1(\epsilon|\phi)}[f(h(\epsilon,\phi))] \tag{3.39}$$

$$= \mathbb{E}_{\pi_1(\epsilon|\phi)}[\nabla_\phi f(h(\epsilon,\phi))] + \mathbb{E}_{\pi_1(\epsilon|\phi)}\Big[f(h(\epsilon,\phi))\nabla_\phi \log\Big(\frac{g(h(\epsilon,\phi)|\phi)}{r(h(\epsilon,\phi)|\phi)}\Big)\Big]$$

Unfortunately in the case of the vMF distribution, we also sample v from a distribution. Therefore we must also introduce $v \sim \pi_2(v)$ and a transform $\mathcal{T}$ such that $z = \mathcal{T}(w, v; \phi)$ [3]:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|\phi)}[f(z)] = \nabla_\phi \mathbb{E}_{\pi_1(\epsilon|\phi),\pi_2(v)}[f(\mathcal{T}(h(\epsilon,\phi), v; \phi))] \tag{3.40}$$

$$= \mathbb{E}_{\pi_1(\epsilon|\phi),\pi_2(v)}[\nabla_\phi f(\mathcal{T}(h(\epsilon,\phi), v; \phi))] + \mathbb{E}_{\pi_1(\epsilon|\phi),\pi_2(v)}\Big[f(\mathcal{T}(h(\epsilon,\phi), v; \phi))\nabla_\phi \log\Big(\frac{g(\mathcal{T}(h(\epsilon,\phi), v; \phi)|\phi)}{r(\mathcal{T}(h(\epsilon,\phi), v; \phi)|\phi)}\Big)\Big]$$

Furthermore if we can design our reparameterisation $r(w|\phi)$ such that is equivalent to our initial distribution $g(w|\phi)$, we can simplify the equation. Our term $\pi_1(\epsilon|\phi)$ simplifies to $p(\epsilon)$ (equation 3.37); similarly we can simplify $\pi_2(v)$ to $p(v)$. Critically if $r(.)$ and $g(.)$ are equivalent, our log term in the second term evaluates to 0, thus removing whole term. This means that if we can design an equivalent reparameterisation:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|\phi)}[f(z)] = \mathbb{E}_{(\epsilon,v)\sim p(\epsilon),p(v)}[\nabla_\phi f(\mathcal{T}(h(\epsilon,\phi), v; \phi))] \tag{3.41}$$

Rejection Sampling for vMF [3]

Now that we have a mathematical basis on how to reparameterise our expectation, we will explain how we sampled from the vMF distribution for the PE model [6] and demonstrate the reparameterisation in practice.

1. Sample $v \sim U(\mathcal{S}^{m-2})$
2. Sample $\omega \sim g(\omega|\kappa, m)$
   - $a = \frac{1}{4}(m - 1 + 2\kappa + \sqrt{4\kappa^2 + (m-1)^2})$
   - $b = \frac{-2\kappa + \sqrt{4\kappa^2+(m-1)^2}}{m-1}$
   - $d = \frac{4ab}{1-b} - (m-1)ln(m-1)$
   - Repeat until condition met (rejection sampling):
     - $\epsilon \sim Beta(\frac{m-1}{2}, \frac{m-1}{2})$
     - $\omega = \frac{1-(1+b)\epsilon}{1-(1-b)\epsilon}$
     - $t = \frac{2ab}{1-(1-b)\epsilon}$
     - $u \sim U(0, 1)$
     - Accept ω iff $(m-1)ln(t) - t + d > ln(u)$
3. $z' = (\sqrt{1-\omega^2} v^T)^T$
4. $R = Householder(e_1, \mu)$
   - $e_1 = [1\ 0\ 0\ 0...]$
   - $r = (e_1 - \mu)/||e_1 - \mu||$
   - $R = I - 2rr^T$, where I is the identity matrix and T is transpose
5. $z = Rz'$

Remember that in our reparameterisation we set $\omega = h(\phi, \epsilon)$. We use the *Beta* distribution to draw our sample of ϵ based on our m parameter which is not dependant on ϕ. The remainder of the process is clearly linear and thus we have satisfied our requirement of h such that it is differentiable w.r.t. ϕ.

We define our prior $p(v)$ to be a uniform distribution over the $m-2$ sphere. We define our transformation $\mathcal{T}(v, \omega; \phi)$ to be steps from 3 to 5 to incorporate the sampled v to generate z. The stochastic elements of the transformation $\mathcal{T}$ are based purely on our prior over v and therefore is still differentiable w.r.t. ϕ. Our function f represents the decoder architecture to generate a result based on our sample z.

This process has now demonstrated how we can calculate our encoding such that we have separated out the probabilistic element away from the variable ϕ which we wish to differentiate with respect to. We can successfully move our differential inside the expectation making the calculation tractable, equation 3.41.

3.5.3 Approximating KL Divergence

Davidson et al. provided the following formulae for a the KL divergence in the vMF VAE [3]:

$$KL(vMF(\mu,\kappa)||vMF(\mu,0)) = \kappa\frac{\mathcal{I}_{m/2}(\kappa)}{\mathcal{I}_{m/2-1}(\kappa)} + log\ \mathcal{C}_m(\kappa) - log\Big(\frac{2\pi^{m/2}}{\Gamma(m/2)}\Big)^{-1} \tag{3.42}$$

$$\mathcal{C}_m(\kappa) = \frac{\kappa^{m/2-1}}{(2\pi)^{m/2}\mathcal{I}_{m/2-1}(k)}$$

$$\nabla KL(vMF(\mu,\kappa)||vMF(\mu,0)) = \frac{1}{2}\kappa\Big(\frac{\mathcal{I}_{m/2+1}(\kappa)}{\mathcal{I}_{m/2-1}(\kappa)} - \frac{\mathcal{I}_{m/2}(\kappa)(\mathcal{I}_{m/2-2}(\kappa) - \mathcal{I}_{m/2}(\kappa))}{\mathcal{I}_{m/2-1}(\kappa)^2} + 1\Big) \tag{3.43}$$

Both the Bowman model and the Prototype and Edit model have relatively high dimensionality in their z representation. This leads to values becoming too large to represent because they are being put to the power of a very large number e.g. $\kappa^{m/2-1}$. Furthermore, because of the shape of the Bessel function, $\mathcal{I}_v(\kappa)$ will be infinitely small as κ gets small causing divide by 0 errors. To solve the first issue, we simplify equation 3.42 using log rules.

$$KL(vMF(\mu,\kappa)||vMF(\mu,0)) = \kappa\frac{\mathcal{I}_{m/2}(\kappa)}{\mathcal{I}_{m/2-1}(\kappa)} + log\ \mathcal{C}_m(\kappa) - (log\ 2 + \frac{m}{2}(log\ \pi) - \sum_{i=1}^{m/2} log\ i)^{-1} \tag{3.44}$$

$$log(\mathcal{C}_m(\kappa)) = (\frac{m}{2} - 1)log(\kappa) - \frac{m}{2}log(2\pi) - log(\mathcal{I}_{m/2-1}(\kappa))$$

We have managed to prevent specific values from getting to large by rearranging the equation using log rules. Unfortunately the Bessel function produces infinitely small values when κ moves towards 0. We solve this by capping κ such that it is always greater than $\frac{m}{8}$. After experimentation, this was the lowest threshold to prevent $\mathcal{I}_{m/2-1}(\kappa)$ being rounded to 0 using a 64-bit float and causing an error. To ensure we continue to learn κ rather than it being stuck at the threshold, we do the thresholding by adding $\frac{m}{8}$ to our calculated values for κ from our linear layer.

3.5.4 Algorithm For Using vMF in VAE

We illustrate the algorithm to use the von Mises-Fisher distribution in the variational autoencoder for text based input in batches of size b.

1. Encode input using LSTM architecture into vector z' of size $b \times m$
2. Feed z' through fully connected layer of size m to create μ sized $b \times m$
3. Feed z' through fully connected layer of size 1 to create κ' sized $b \times 1$
4. $\kappa = \kappa' + \frac{m}{8}$
5. Sample z using rejection sampling
6. Decode input using LSTM decoder
7. $D_{KL} = \kappa\frac{\mathcal{I}_{m/2}(\kappa)}{\mathcal{I}_{m/2-1}(\kappa)} + log\ \mathcal{C}_m(\kappa) - (log\ 2 + \frac{m}{2}(log\ \pi) - \sum_{i=1}^{m/2} log\ i)^{-1}$
8. $w = f(s)$ where f is annealing schedule on step s
9. $\mathcal{L} = log\ p(x) + w \cdot D_{KL}$
10. Use back propagation to optimise all model parameters

3.5.5 Practical Implementation

Prototype and Edit Model Alterations

We added a configuration to the Prototype and Edit model to now learn the κ parameter. We add the KL cost to the loss function and additionally added in the KL cost annealing strategy from the Bowman model to solve optimisation issues. All other elements of the model remained the same.

Bessel Function in Tensorflow

There is currently no functionality for the Bessel function of any arbitrary order in Tensorflow. Currently only $\mathcal{I}_0(x)$ and $\mathcal{I}_1(x)$ are available because of their use in smoothing functions. The Bessel function is defined in package *special* within SciPy library. We use the Tensorflow ops package to wrap this function which takes numpy arrays as input to instead take tensors. The Tensorflow framework is unable to differentiate this wrapped Bessel functions with respect to our parameters, so we manually define the gradient of our KL divergence (equation 3.43) using the Tensorflow custom_gradient package.

Distributions Configurations

Crucially we now have multiple distribution configurations for both of our models that we would like to seamlessly swap between when running a set of experiments. We introduce a distribution class for each of the distributions that we are experimenting with in this thesis. They all contain an initialisation function that takes a list of their parameters, a sample function and a function that calculates the KL divergence given the current initialisation. We then add a distribution key to our config JSON and different distributions can be easily attached to the models without any code changes in-between experiments.

3.6 Cauchy Variational Autoencoder

The Gaussian distribution has light tails meaning that the distribution decays exponentially. This can lead to a crowding problem where data points are packed too closely together at the origin and the model struggles to learn how to decode samples [24]. The Cauchy distribution has heavier tails meaning that it decays far slower. The distribution was selected because of this property to compress data into visual representations in the t-SNE algorithm [24]. We propose that the Cauchy distribution may be more suitable for learning a representation of our latent variable z.

3.6.1 KL Divergence

There is no documented use of the Cauchy distribution as a variational autoencoder so we must derive the KL divergence between our calculated distribution $\mathcal{C}(\mu, \gamma)$ and our prior $\mathcal{C}(0, I)$. Calculating the KL divergence in the multivariate case is mathematically difficult. We derive a lower bound in the one dimensional case and then make another approximation to generate a multivariate KL equation. The following derivation was done in collaboration with Andrea Karlová, the industrial co-supervisor of this project.

Let us recall that the density of one-dimensional Cauchy distribution is given by:

$$p(x|\mu, \gamma) = \frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)} \tag{3.45}$$

We would like to compute the KL-divergence term

$$KL(\mathcal{C}(\mu, \gamma)||\mathcal{C}(0,1)) = H[p(\mu, \gamma), p(0,1)] - H[p(\mu, \gamma)], \tag{3.46}$$

which we use in the objective function of the VAE.

Deriving Entropy

$$H[p(\mu, \gamma)] = -\int \frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)} log \left[\frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)}\right] dx \tag{3.47}$$

$$log \left[\frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)}\right] = log \left[\frac{\gamma}{\pi}\right] + log \left[\gamma^2 + (x-\mu)^2\right]^{-1}$$

$$H[p(\mu,\gamma)] = -log \left[\frac{\gamma}{\pi}\right] \int \frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)}dx + \int \frac{\gamma log \left[\gamma^2 + (x-\mu)^2\right]}{\pi(\gamma^2 + (x-\mu)^2)}dx$$

$$\int \frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)}dx = 1$$

We use a substitution $y = x - \mu$ to simplify the integration. We also simplify the limits of our integral because the function is symmetric around 0.

$$H[p(\mu,\gamma)] = -log \left[\frac{\gamma}{\pi}\right] + 2 \cdot \frac{\gamma}{\pi} \int_0^\infty \frac{log \left[\gamma^2 + y^2\right]}{\gamma^2 + y^2}dy$$

We calculate the integration using an integration table and then perform some basic log transformations to simplify the equation.

$$H[p(\mu,\gamma)] = log \left[4\pi\gamma\right]$$

Deriving Cross Entropy

$$H[p(\mu,\gamma), p(0,1)] = -\frac{s}{\pi} \int \frac{log \left[\pi(1+x^2)\right]^{-1}}{\gamma^2 + (x-\mu)^2}dx \tag{3.48}$$

$$log[\pi] \int \frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)}dx + \frac{\gamma}{\pi} \int \frac{log \left[1+x^2\right]}{\gamma^2 + (x-\mu)^2}dx$$

$$\int \frac{\gamma}{\pi(\gamma^2 + (x-\mu)^2)}dx = 1$$

We use a substitution $y = x - \mu$ to simplify the integration. We also simplify the limits of our integral because the function is symmetric around 0:

$$H[p(\mu,\gamma), p(0,1)] = log \left[\pi\right] + 2 \cdot \frac{\gamma}{\pi} \int_0^\infty \frac{log \left[1 + (y+\mu)^2\right.}{\gamma^2 + y^2}dy$$

$$\geq log \left[\pi\right] + 2 \cdot \frac{\gamma}{\pi} \int_0^\infty \frac{log \left[1 + y^2 + \mu^2\right.}{\gamma^2 + y^2}dy$$

We calculate the integration using an integration table and then perform some basic log transformations to simplify the equation:

$$\geq 2log \left[\sqrt{1+\mu} + \gamma\right] + log \left[\pi\right]$$

Deriving KL Divergence Bound

Finally we can consider of the KL-divergence. Recall that:

$$KL(\mathcal{C}(\mu,\gamma)||\mathcal{C}(0,1)) = H[p(\mu,\gamma), p(0,1)] - H[p(\mu,\gamma)] \tag{3.49}$$

$$KL(\mathcal{C}(\mu,\gamma)||\mathcal{C}(0,1)) \geq 2log \left[\sqrt{1+\mu} + \gamma\right] + log \left[\pi\right] - log \left[4\pi\gamma\right] = log \left[\frac{\sqrt{1+\mu}+\gamma}{4\gamma}\right] \tag{3.50}$$

We have successfully estimated the lower bound of the KL-divergence for the one-dimensional Cauchy distribution. Generalising into the $n-$dimensional case requires using complex variables and other more complicated special functions. We decided to use a lower estimate of the joint multi-dimensional distribution. We approximate it by the sum of the one-dimensional Cauchy distributions. We are loosing large amounts of the probability mass but we believe that this approach is sufficient for the primary investigation.

$$KL(\mathcal{C}(\mu,\gamma)||\mathcal{C}(0,I)) \approx \sum_{i=1}^{n} log \left[\frac{\sqrt{1+\mu_i}+\gamma_i}{4\gamma_i}\right] \tag{3.51}$$

We are forced to ensure that $\mu_i > -1$ and that γ_i does not shrink to zero causing a divide error.

3.6.2 Reparameterisation Trick

As with the other two distributions that we have observed, we must use a reparameterisation trick to separate out the noise ϵ from encoding process away from our parameter ϕ. This allows us to move the derivative inside the expectation, making the computation tractable. Kingma et al. outlined that the Cauchy distribution could be reparameterised using the inverse of its CDF [10].

$$F(x;\mu,\gamma) = \frac{1}{\pi} arctan\Big(\frac{2(x-\mu)}{\gamma}\Big) + \frac{1}{2} \tag{3.52}$$

$$F(x;\mu,\gamma)^{-1} = \mu + \gamma\ tan[\pi(x-0.5)]$$

The reparameterisation $z = g_\phi(\epsilon, x)$ does not require any rejection sampling scheme because of the simpler transformation. We update our loss function:

$$\mathcal{L}(x,\theta,\phi) = \frac{1}{L}\sum_{l=1}^{L} log\ p_\theta(x|z) + KL(\mathcal{C}(\mu,\gamma)||\mathcal{C}(0,I)) \tag{3.53}$$

$$\text{where } z = \mu + \gamma\ tan[\pi(\epsilon - 0.5)] \text{ and } \epsilon \sim U(0,I)$$

3.6.3 Practical Implementation

We have already successfully set up our framework to easily change the distribution within each model. The reparameterisation trick is far simpler than that used in the vMF distribution; this meant that once the KL divergence was derived, there were no implementation difficulties.

3.7 Summary

We have now gained a thorough understanding of the variational autoencoder as well as exploring two models that implement it to generate sentences. We have introduced the use of the von Mises-Fisher distribution and the Cauchy distribution in the variational autoencoder and added them as configurations to both models. We now focus on comparing the implementations of our models to any benchmarks and then explore the effect that each choice of distribution has on performance.

Chapter 4

Critical Evaluation

4.1 Introduction

In the last chapter, we introduced the variational autoencoder and two proposed models that use the architecture to generate sentences: The Bowman model [2] and the PE model [6]. We first illustrated a practical implementation of these two papers and then moved into extending them to improve their performance. In the Bowman model, we explored the use of the von Mises-Fisher and Cauchy distribution instead of the Gaussian in the variational component of our autoencoder. In the PE model, we believed that having a fixed value of κ in the vMF distribution was a poor approach and implemented an alternative approach that learned an optimal value based on the input. We then also introduced the Cauchy variational autoencoder.

We begin by analysing the performance of both of our models. We evaluate based on a set of metrics as well as observing training curves and particular success and failure cases. Once we are satisfied with our initial implementations, we will investigate the effect of our extension to the Prototype and Edit model. We reasoned that having the κ parameter fixed in the vMF distribution was a poor approach and we developed an extension such that the model learned how to alter κ depending on the input.

We then focus on our primary area of evaluation: comparing the performance of both models using the Gaussian, von Mises-Fisher and Cauchy distribution. We test a number of configurations with different representation sizes to explore how results vary over different dimensions. We will identify the strengths and weaknesses in each model and how this reflects in the results to identify the optimum distribution choice to use for natural language generation.

After concluding our distribution experiments, we will focus on the learned representation of sentences from our best model. We will relate this back to our initial motivation of generating new artificial sentences for data augmentation by evaluating the quality of generated sentences.

The complexity of the models as well as the size of the training sets meant the training time per model was extremely long even using accelerated computing technology. This limits us in the number of configurations that we can fairly compare with one another. The suggestion of further possible areas of evaluation will be listed under future works in Chapter 5.

4.2 Performance Metrics

We introduce a number of varied performance metrics to compare the relative performance of our different configurations.

4.2.1 Reconstruction Loss

We will primarily use the average reconstruction loss (RL) to compare the performance of our models given by:

$$RL = -\frac{1}{N}\sum_{x\in X} ln\ p(x) \tag{4.1}$$

It is a measure of how well our trained model is predicts samples in the test set. A better model has a smaller reconstruction loss because it places a higher probability on generating the target sentence. Reconstruction loss is our strongest method for comparison because we are measuring the certainty of the sentence that we are generating rather than just comparing decoded sentences. Two different models may decode a sentence correctly to match a target but reconstruction loss will further reward a model based on the confidence it had in this correct estimate.

Perplexity is traditionally used instead of reconstruction loss in NLP literature as an evaluation metric. It is defined as:

$$H(X) = \frac{RL}{ln\ 2} \tag{4.2}$$

$$perplexity = 2^{H(X)}$$

Because of our limited training capabilities and the exponential relationship between the reconstruction loss and perplexity, we decided it was not a clear comparator. Although our perplexity would reduce to more reasonable values after a large amount of training, it was $\mathcal{O}(10^6)$ for the number of steps we were doing.

4.2.2 Jaccard Distance

We have already discussed the use of the Jaccard distance in the Prototype and Edit model to build our dataset; it will also act as a good similarity measure between our generated sentence from our target sentence. The Jaccard distance is a good measure for evaluating how well a network is at generating uncommon vocabulary. See Chapter 2 for the formal definition.

4.2.3 Levenshtein Edit Distance

As well as focusing on the set of words in a sentence, it is important to evaluate the order in which they appear. The Levenshtein edit distance is another similarity measure used to compare the distances between two sentences. We compare our sentences by calculating the minimum number of edits we need to perform to transform the generated sentence into the target sentence. An edit is either inserting, deleting or substituting a word at a specific index with two identical sentences having a Levenshtein edit distance as 0.

As an illustrated example consider the target sentence “the pizza really was great” and a generated sentence “the pizza was really good”. The distance between these sentences is 3: removing the word “was”, inserting the word “was” after “really”, substituting “good” for “great”.

4.2.4 Word Accuracy

The Levenshtein edit distance does not reward the proportion of correctly identified words in the right position. It calculates the average number of mistakes per sentence. We introduce a new metric, word accuracy, that rewards words being in the correct position relative to the length of the sentence. We compare both the generated sentence and the target sentence one word at a time and then find the percentage of correct words for the sentence.

4.2.5 BLEU Score

It is also important to focus on the number of correct sequences, known as n-grams in a sentence. The BLEU (BiLingual Evaluation Understudy) is a metric introduced by Papineni et al. as a method of evaluating the correctness of a translation quantitatively [16]. We measure the number of n-grams from our generated sentence that are found in our target sentence. We create a set of n-grams (n values of 1,2,3 and 4) for our generated sentence and our target sentence. We then calculate the proportion of n-grams in the generated set from the target set as our BLEU score.

4.3 Bowman Model Implementation

In the previous chapter we discussed the importance of implementing the Prototype and Edit model from scratch; it was important to set up a framework such that we could easily change configurations as well as having a large crossover in architecture between the two models. Before beginning our investigation, it is paramount that we compare the performance of our implementation to the public results.

Unfortunately, the only published results in the Bowman paper use the Penn Treebank data which is not free to access. This means that we do not have a way of fully verifying the quality of our model in comparison to the published implementation. There is also no publicly available code for the Bowman model for us to run on the Yelp dataset either. To evaluate the performance of our model, we measure all of our performance metrics and assess how well the model performs at reconstructing input sentences. We also analyse the training curves of our model to observe if the model is performing correctly. For this experiment, we use our default configuration which is clearly outlined in section 3.4.2. The results can be seen in Table 4.1.

z Dimension	Steps	Reconstruction Loss	Word Accuracy	Jaccard	BLEU	Levenshtein
1024	50000	11.12	74.3%	0.24	0.62	2.62
512	50000	13.1	69.8%	0.28	0.57	3.1
256	50000	17.02	60.6%	0.33	0.5	3.7

Table 4.1: Results of experiments using our Bowman model implementation with the Gaussian distribution

4.3.1 Training Analysis

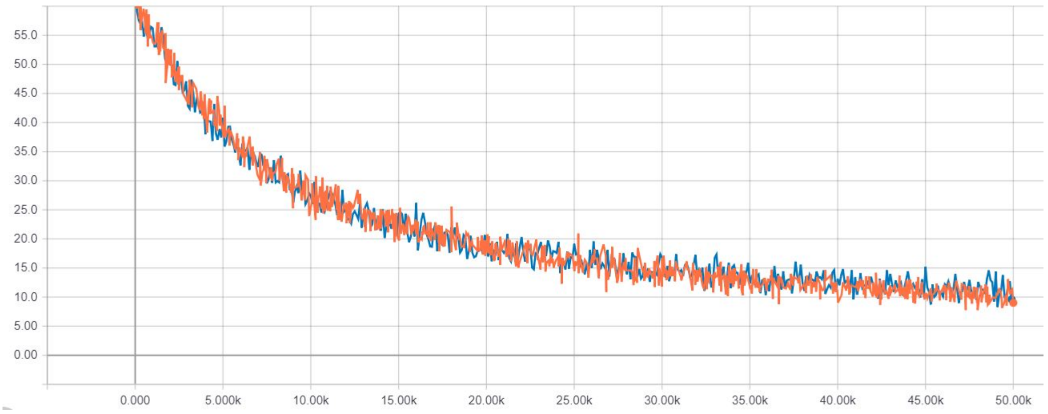

Figure 4.1: The training (orange) and testing (blue) reconstruction cost over the number of training steps for the Bowman model with a z dimension of 1024.

Figure 4.1 illustrates the decrease in reconstruction loss in the test set during the training process. The model is clearly optimising well and shows no signs of overfitting. Although the loss is beginning to plateau, the training curve implies that we could get further better performance if we increased the number of training steps.

Whilst training, the network is balancing the reconstruction loss term with the KL divergence. We would expect the KL divergence to be non-zero and to change slightly over time to learn an optimum balance. A higher KL divergence should result in a smaller reconstruction cost because we are providing more freedom to the network to model the representation of z. Figure 4.2 demonstrates the KL divergence term during training. We added a sigmoid annealing schedule to the optimisation process; a weight on the KL term slowly increased from 0 to 1 over the first 5000 steps. The divergence drops off very rapidly and then remains extremely close to 0 only varying slightly. We would expect the KL term to vary more

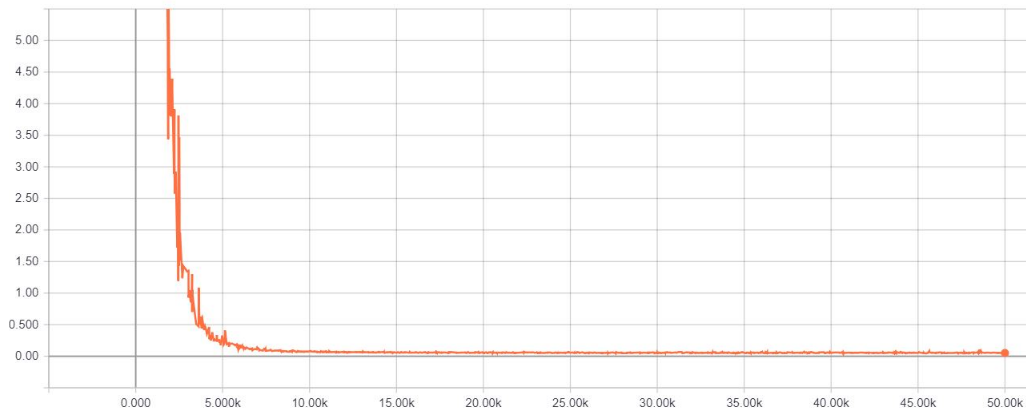

Figure 4.2: The KL divergence over the number of training steps for the Bowman model with a z dimension of 1024

than this as it implies we are relying too much on optimising our representation to match the prior. This could be having negative effects in minimising our reconstruction cost. We may be stuck in a local minima in the balance between the two terms. Experimenting with the annealing schedule further or adding a fixed KL scaler to the loss function could improve results. Unfortunately because of the long training times required, we were unable to investigate these potential improvements.

4.3.2 Qualitative Results

We have provided quantitative measures of how successful the Bowman model has been at reconstructing sentences. We now illustrate some specific examples of success and failure cases that are not fully illustrated by the published results.

Success Cases

A number of our sentences were perfectly reconstructed by our model. Instead of observing these examples, we instead illustrate examples where our model failed to reconstruct the sentence perfectly but still generated sentences that are syntactically correct.

1. - Source: unquestionably the best ⟨norp⟩ i 've found in the area .
   - Predicted: absolutely the best ⟨norp⟩ i 've found in the area .
2. - Source: cover charge is around \$ ⟨money⟩ .
   - Predicted: parking is around \$ ⟨money⟩ .
3. - Source: i will definitely be back to try more ⟨norp⟩ dishes !
   - Predicted: i will definitely be back to try more dishes ! ;)

In the first example, we can see that the model has made a singular mistake when attempting to generate the source sentence. It replaced the word "unquestionably" with "absolutely". Although we have failed in recreating the exact sentence, the network has placed probability over another similar word. The sentence still remains on topic and still makes grammatical sense. In the second example, the network's representation has captured that the source sentence is related to some charge. The selection of the word "parking" over "cover charge" has kept the generated sentence on topic and plausible. In the third example, we can see that the network can produce sentences that are grammatically correct even if a word has been completely missed in the decoding process. In the last two examples, we were just substituting words but in this case we missed the word "⟨norp⟩" entirely. However, the network still generated a plausible sentence and appended a "winking face emoji" which is fitting with the style of the review.

Failure Cases

We now illustrate some example sentences that were poor reconstructions to explore possible weaknesses in the model.

1. - Source: for ⟨gpe⟩ , it 's the best dim sum restaurant in town .
   - Predicted: it 's the best sum restaurant in ⟨gpe⟩ .
2. - Source: they finally showed up ⟨time⟩ late .
   - Predicted: they showed early ⟨time⟩ early late .
3. - Source: not thrilled and probably wo n't be back anytime soon .
   - Predicted: not disappointed and probably wo n't be back anytime soon .
4. - Source: great spot - yes , a chain , but good food and quick !
   - Predicted: good , a good , good food !

In the first example, the model has struggled to reconstruct the same structure of the sentence. The generated sentence still picked a structure that maintained the grammar as well as the topic so is not considered a failure. However, the model has failed to recall the unusual word "dim" which is a crucial part of the sentence, making it implausible. In the second example, the model has failed to generate a sentence that makes grammatical sense. The generated sentence has words such as "early" and "late" that we would expect to see near "⟨time⟩" but not in an order that makes sense. In the third example, the model has replaced the word "thrilled" with "disappointed" when reconstructing the source sentence. The selection of the word "disappointed" shows the model was able to capture the overall negative sentiment to the sentence but was unable to interpret the effect of the word "not". As a result, the generated sentence is grammatically correct but is clearly implausible. The final example is a good illustration of a common failure in which certain sequences of words repeat multiple times within a sentence.

4.3.3 Evaluation

It is unfortunate that we are unable to compare our implementation with any published results but the results we have gathered suggest that we can be confident with our implementation. The model performs very well at reconstructing sentences and the training curves suggest that this performance would only improve given more time. Any failed reconstructions often still make grammatical sense and remain on topic with there being very few failure cases. The KL term remains relatively constant during the training process which could be a weakness; further experimentation with the KL cost annealing could improve the results further.

4.4 Prototype and Edit Model Implementation

The hyperparameters used to recreate the results in the paper can be found on the authors public GitHub repository [5]. We match these hyperparameters in our own implementation of the PE model to provide a fair comparison between the two. We use the exact same train and test sets in both models. The PE model is complex and the Yelp dataset is also extremely large; this results in a very long time to train the model. Consequently, we only train the model once with the correct configuration rather than a common approach which would be to train the model multiple times and average the performance.

Model	Steps	Training Hours	Edit Dimension	Sentence Dimension	Loss
Guu PE	5000	7	128	512	14.8
Stein PE	15000	6	128	512	26.2

Table 4.2: Comparison of the public Guu code [5] and our implementation over the full Yelp dataset

4.4.1 Analysis

Our implementation falls far short of the results from the benchmark code, table 4.2. Not only does our model optimise slower but the reconstruction loss also plateaus at a much earlier stage. When investigating the code base, it becomes clear that the authors have far extended their code base beyond the basic approach that has been outlined in the paper [6]. Because of a lack of comments in their code, it is very difficult to clearly identify what these extensions are doing such that we can extend our own code. A large concern is the difference in training time between each iteration. The public code trained about three times slower, implying that there must have been a large number of extra hidden parameters to the network. The main aim of this project was to compare the effect of using different distributions within the variational autoencoder architecture when generating sentences. Although we have not implemented the architecture optimally, we are still approaching the problem differently and the model is still a good method of comparison in our investigation. Our implementation of the model was not optimising well so we make a number of alterations to improve its performance.

4.4.2 Alterations

To stop the training loss from plateauing, we first reduce the size of our training set. This would allow the model to overfit to the training set, making optimisation far easier. We reduce the training set from 4 million sentences to just 30,000. We also increase the hidden dimension of the LSTM and the dimension of the vector used to encode the edit. We alter the optimisation process to train all the parameters together rather than splitting it into the set ϕ and θ. We experimented using three fully connected layers with relu activation functions to better learn the edit encoding as well as adding extra layers elsewhere in the network but the result of these changes were marginal. All these changes provided a small increase in performance from our initial attempt on the full dataset.

Model	Steps	Training Hours	Edit Dimension	Sentence Dimension	Loss
Stein PE v1	10000	6	128	512	26.2
Stein PE v2	10000	7	256	1024	21.6

Table 4.3: Comparing the performance of our previous configuration with our new model on the full Yelp dataset

Model	Steps	Edit Dimension	Sentence Dimension	Train RL	Test RL
Stein PE v1	25000	128	512	4.2	62.1
Stein PE v2	25000	256	1024	1.1	57.7

Table 4.4: Comparing the performance of our previous configuration with our new model on the reduced Yelp dataset

4.4.3 Evaluation

The implemented model fell far short of the benchmark that was provided by the authors of the Prototype and Edit model. Although these results are disappointing, our implementation is still a varied approach of utilising the variational autoencoder to generate sentences. It is still therefore an adequate mean of comparison between the different distributions that we will be investigating. Because of its relative poor performance in comparison to the Bowman model, we will focus only on the quantitative performance of the Prototype and Edit model rather than evaluating any generated sentences.

4.5 Prototype and Edit Model leaning κ

Although the Prototype and Edit model uses the von Mises-Fisher distribution, the authors decided to keep the κ parameter fixed. This goes against the theory suggested for the variational autoencoder that models learn this variational parameter to balance the reconstruction loss with the KL divergence term. In the previous chapter, we outlined the extension upon the PE model to learn this variable κ as well as outlining any optimisation problems that we had whilst training. We expect this will improve the performance of the model because the PE model will be able to learn an optimal balance between regularising the network towards a prior distribution and minimising the reconstruction cost.

Distribution	Steps	Edit Dimension	Sentence Dimension	Train RL	Test RL
vMF ($\kappa = 100$)	10000	256	1024	21.3	21.6
vMF (Learned κ)	10000	256	1024	26.2	26.3

Table 4.5: Comparing the performance of the Prototype and Edit model with a fixed κ and a learned κ on the full Yelp dataset

Distribution	Steps	Edit Dimension	Sentence Dimension	Train RL	Test RL
vMF ($\kappa = 100$)	25000	256	1024	1.1	57.7
vMF (Learned κ)	25000	256	1024	4.6	53.4

Table 4.6: Comparing the performance of the Prototype and Edit model with a fixed κ and a learned κ on the reduced Yelp dataset

4.5.1 Analysis

In the Prototype Edit model, the fixed κ model performs better on the full dataset but overfits more in the small dataset. We can see from Figure 4.3 that the extended model optimises κ to a value much lower than 100. There is initially great instability because of the annealing schedule but it then begins to plateau around 32 which is our κ threshold. Once again, more experimentation is necessary to attribute this to a poorly chosen annealing schedule or if this is the optimum behaviour. A lower concentration parameter means that sampled points are spread more evenly around the hypersphere. It is then no surprise that a model that has a lower value of κ will initially optimise slower when given a large amount of data. Having said this, by learning a lower value of κ, our network is forced to generalise better and this is demonstrated in the results from the small dataset. The model that learns κ has a higher training reconstruction loss but a better test reconstruction loss.

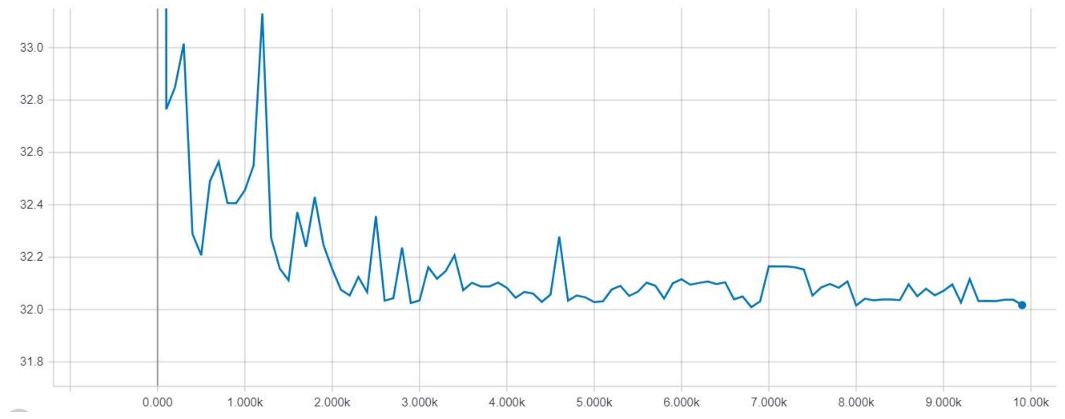

Figure 4.3: The learned value of κ over the number of training steps for the Prototype and Edit model with a z dimension of 1024 on the full training set

A variational autoencoder should be learning the correct balance between the reconstruction loss and the KL cost. We have seen that in both models, the network places too much priority on initially optimising the KL term to be close to its prior despite our annealing schedule. We have stated many times that experimenting with different annealing schedules could help solve this issue but this solution may still be insufficient for the von Mises-Fisher distribution. It can be seen from Figure 4.4 that even when our learned κ is close to our threshold, our KL divergence is extremely large because of our selected prior $p(z) = vMF(\mu, 0)$. Because of difficulties in sampling uniformly in the hypersphere in large dimensions [3], any learned κ is going to be much much larger than 0. Our KL term is always going to be dwarfing the reconstruction loss which means it is overly favoured by the optimiser. To combat this, we could experiment with selecting a different more realistic prior or alternatively adding a fixed scaler β such that KL term was closer in magnitude to the reconstruction loss.

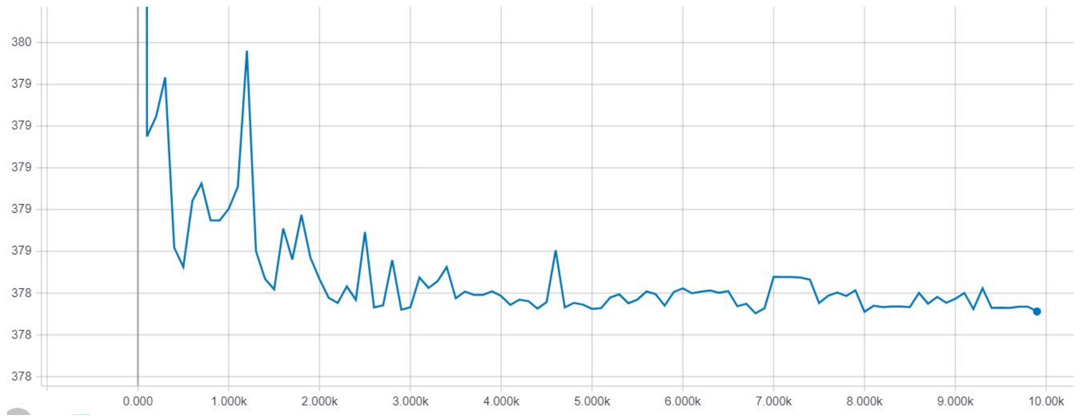

Figure 4.4: The KL divergence over the number of training steps for the Prototype and Edit model with a z dimension of 1024 on the full training set

4.5.2 Evaluation

The fixed κ model performs better on the large dataset than the learned representation because it had a higher value of κ. The learned representation however has showed its improved ability to generalise when the model begins to overfit to the training data. We have discussed possible improvements to the learned κ model to improve performance and these alteration will be discussed further in Chapter 5.

4.6 Distribution Evaluation

We will now move onto our primary area of evaluation which is the main focus of this thesis. We will investigate the effect on performance of the variational autoencoder for sentence generation using three different distributions: the Gaussian, the von Mises-Fisher and the Cauchy. We start by explaining all our hypothesised relative strengths and weaknesses of each distribution in turn for the task of natural language generation before we compare their performance. We are going to be comparing with both models to hopefully build censuses of the best distribution to use in the sentence generation task regardless of the method. We hope to show that the Gaussian distribution, despite being by far the dominant choice, is not always the optimum choice of distribution in the variational autoencoder.

We use J to denote the dimensionality of our latent representation z. The strength of particular distribution choices may be dependant on the size of J. A lower dimensionality will lead to a more condensed representation of the data. This could lead to better generalisation because the model will be forced to come up with a stronger representation without much unfilled space. Having said this, if J is too small than there may not be enough space to encode all our sentences well. Furthermore, we know that in general, some distributions are better than others depending on the scale of dimensions. We want to explore the effect of different distributions over multiple different values of J to explore this trade off for both models.

The set of J sizes for our representation that we wish to compare will be different for both models. In the Bowman model, we need to learn to represent a whole sentence whereas the PE model only needs to learn a small transformation. This means that the Bowman model inherently needs to have a larger size because there is more data to capture. We will explore $J = [256, 512, 1024]$ for Bowman and $J = [128, 256]$ for the PE model. We will first begin by summarising our three varieties of the variational autoencoder.

4.6.1 Gaussian Variational Autoencoder

Ever since the variational autoencoder was first described by Kingma et al [10], the Gaussian distribution has been the conventional choice because of a number of benefits. The first of which is that we can use a basic reparameterisation trick to make our loss function easily and reliably differentiable with respect to ϕ.

$$\mathbb{E}_{q_\phi(z|x)}[f(z)] = \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))] \tag{4.3}$$

$$g_\phi(\epsilon, x) = \mu + \sigma \cdot \epsilon \text{ and } \epsilon \sim \mathcal{N}(0, I)$$

In addition to its convenience, the Gaussian distribution is extremely good at representing all data because it can be used as an approximation for any distribution. The standard central limit theory states that any distribution can be approximated to a Gaussian distribution given a large enough number of samples.

However, the Gaussian distribution is far from perfect and has limitations especially when working with high dimensional data like word embeddings. The “soap bubble effect” is a well known phenomenon that the Gaussian distribution in high dimensions tends to resemble a uniform distribution on the surface of a hypersphere, with the vast majority of its mass concentrated on the hyperspherical shell [3]. This leads us to believe that for NLP tasks, the Gaussian distribution may not be the most effective distribution at representing our data and has been our motivation to experiment with new distributions not commonly used.

4.6.2 Von Mises-Fisher Variational Autoencoder

Recent work published has suggested the use of the von Mises-Fisher distribution for data that is seen to be spherical rather than lying on a hyperplane [3]. Figure 4.5 shows how the use of the von Mises-Fisher distribution can provide a far stronger representation when the original data is spherical in nature.

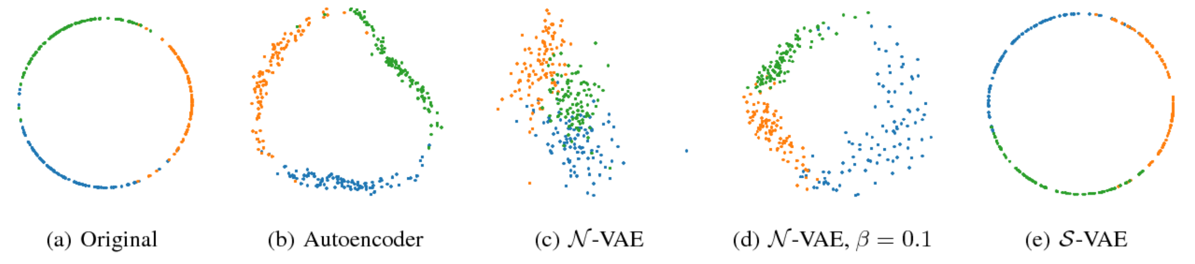

Figure 4.5: A plot demonstrating the latent representation built by a Gaussian autoencoder($\mathcal{N}$-VAE) in comparison to one using vMF($\mathcal{S}$-VAE) for spherical data [3]. β is a KL cost scaling constant

Word embeddings are seen as being a spherical data representation because of the shown effectiveness of using cosine similarity to compare points in the representation. This implies that for NLP tasks the vMF distribution may be a more optimum distribution. This was pioneered by Guu et al in the Prototype and Edit model [6]. We have already demonstrated on how we have extended their intuition to create a model that now learns the parameters of the distribution.

We have shown in the previous chapter the additional complexity both mathematically and in a practical implementation when the dimension used for z is high. The distribution can become unstable at high dimensions [3]. This may imply that the vMF may not be as good a fit for the Bowman model as it is for the PE model.

4.6.3 Cauchy Variational Autoencoder

The Cauchy distribution has heavy tails meaning that the distribution decays far slower than the Gaussian. The Gaussian distribution is known to suffer from the crowding problem in which data points get packed too closely together often at the origin. The heavy tailed Cauchy distribution allows data points to spread out more. This property has led to its success when being used to compress high dimensional data into visual 2D representations [24]. By the same logic, we believe that the Cauchy distribution may be a more appropriate distribution to use within the variational autoencoder. Similarly to the Gaussian distribution, the cauchy distribution has a very mathematically convenient reparameterisation.

$$\mathbb{E}_{q_\phi(z|x)}[f(z)] = \mathbb{E}_{p(\epsilon)}[f(g_\phi(\epsilon, x))] \tag{4.4}$$

$$g_\phi(\epsilon, x) = \mu + \gamma\ tan[\pi(\epsilon - 0.5)] \text{ and } \epsilon \sim \mathcal{U}(0, I)$$

We have unfortunately had to make a large simplification when calculating the KL divergence between our calculated distribution and our prior. Having said this, we still believe the model will be able to balance the reconstruction cost and this KL to learn an effective representation.

Distribution	z Dimension	Steps	Reconstruction Loss	Word Accuracy	Jaccard	BLEU	Levenshtein
Gaussian	1024	50000	11.12	74.3%	0.24	0.62	2.62
Gaussian	512	50000	13.1	69.8%	0.28	0.57	3.1
Gaussian	256	50000	17.02	60.6%	0.33	0.5	3.7
vMF	1024	50000	9.8	77.1%	0.2	0.63	2.3
vMF	512	50000	13.37	68.4%	0.29	0.54	3.1
vMF	256	50000	18.77	57.5%	0.37	0.5	4.2
Cauchy	1024	50000	13.2	70.9%	0.32	0.48	3.4
Cauchy	512	50000	16.9	62.6%	0.35	0.36	3.7
Cauchy	256	50000	20.96	53.7%	0.42	0.22	4.6

Table 4.7: Results from the Bowman model experiments on the full Yelp dataset

Distribution	Steps	Edit Dimension	Sentence Dimension	Train RL	Test RL
Gaussian	25000	256	1024	3.4	57.6
Gaussian	25000	128	1024	3.7	58.2
vMF	25000	256	1024	4.6	53.4
vMF	25000	128	1024	3.6	56.5
Cauchy	25000	256	1024	3.9	58.9
Cauchy	25000	128	1024	3.6	59.0

Table 4.8: Results from the Prototype and Edit model experiments on a reduced Yelp dataset

4.6.4 Analysis

Models with a higher dimension for z perform better because the model has more space to build its latent representation. For the task of generating new artificial sentences, a model with a higher reconstruction cost and lower z dimension may perform better because the representation is more compressed and there is less empty space. This forces the network to learn a stronger representation meaning that any decoded sentence in the space is more likely to make grammatical sense. Finding the optimum value for z for our specific task requires more investigation which is outside the scope of this these; we accept the model with the smallest reconstruction loss as our most successful model. In this section, we focus on evaluating the quantitative suitability of the von Mises-Fisher and Cauchy distribution. We focus more on the quality of our learned representation and the ability of the model at generating artificially sentences in sections 4.8 and 4.9 respectively.

4.6.5 Suitability of Von Mises-Fisher VAE

Bowman Model

The vMF distribution VAE significantly outperforms the Gaussian VAE in higher dimensions. We have discussed the limitations in both models in the high dimensional case with the Gaussian "soap and bubble effect" and the vMF "vanishing surface problem". By setting the lower threshold on κ for practical implementation reasons, we have mitigated this issue. In higher dimensions, our model is forced to increase the concentration of an encoded sentence preventing data being spread too thinly in the high dimensional space. Interestingly, this has not impacted the models ability to generalise to the test set when overfitting may have been expected. In lower dimensions, the vMF VAE performed worse than the Gaussian distribution. We have already discussed weaknesses in our method of learning κ in section 4.5.1. Possible weaknesses in the annealing schedule as well as the KL term dwarfing the reconstruction loss is forcing the κ close to its lower threshold. When the value of κ is this low, the model is struggling to learn a good representation.

Prototype and Edit Model

In the Prototype and Edit model, we are only perturbing the directional element (z_{dir}) of our encoded edit. Given that the vMF distribution is particularly strong at modelling this type of data, it is no surprise that it performs better than the Gaussian distribution despite potential weaknesses in learning κ. Because we are using a smaller dataset in this model, we are observing the ability to optimise as well as generalise to unseen data. In both variations of the edit dimension, the vMF distribution is learning a representation that generalises better to unseen data. In the higher dimensional case, this generalisation comes at a cost of the ability to optimise as well to the training set. However, in lower dimensions the vMF distribution is stronger in both optimising and generalising compared to the Gaussian. This demonstrates that it is a more optimal distribution of choice in the Prototype and Edit model.

Evaluation

In earlier chapters, we have proposed the weaknesses of the Gaussian distribution and the potential benefits of using the von Mises-Fisher distribution in the variational autoencoder. We have gained quantitative results to demonstrate this in high dimensional cases or if only modelling the direction component of a latent vector. The method proposed for learning has weaknesses and can be improved which highlights the potential further increase in performance. For the task of text generation, we have shown that the vMF distribution is more appropriate than the Gaussian.

4.6.6 Suitability of Cauchy VAE

Bowman Model

The Cauchy distribution consistently had the lowest performance across all dimensions and is clearly not appropriate at representing our data. Interestingly, the KL divergence varied much more during training than both the Gaussian and the vMF VAE. This implies that the network was performing a good balance between the reconstruction cost and KL divergence. The Cauchy distribution decays at a much slower rate than the Gaussian meaning that each data point must take into consideration encodings in a wider region rather than its immediate neighbours. This should lead to a more complete representation and should help the model generate more diverse sentences because of the wider context. This is quantitatively shown when comparing the Gaussian-512 and Cauchy-1024. Both have near identical reconstruction loss but the Cauchy has much higher Jaccard and Levenshtein distance. The model is generating more diverse vocabulary not in the source sentence because of its wider context. It can be seen from Figure 4.6 and 4.7 that the Cauchy VAE is far more volatile than the other distribution choices. This is because, by nature, the Cauchy distribution points are occasionally sampled far away from the mean.

Prototype and Edit Model

Similarly, the Cauchy distribution performed worst in the Prototype and Edit model. It struggled both to optimise to the training set as well as generalise to unseen data. This compounds our findings from the Bowman model that the Cauchy distribution is not an effective choice for generating sentences.

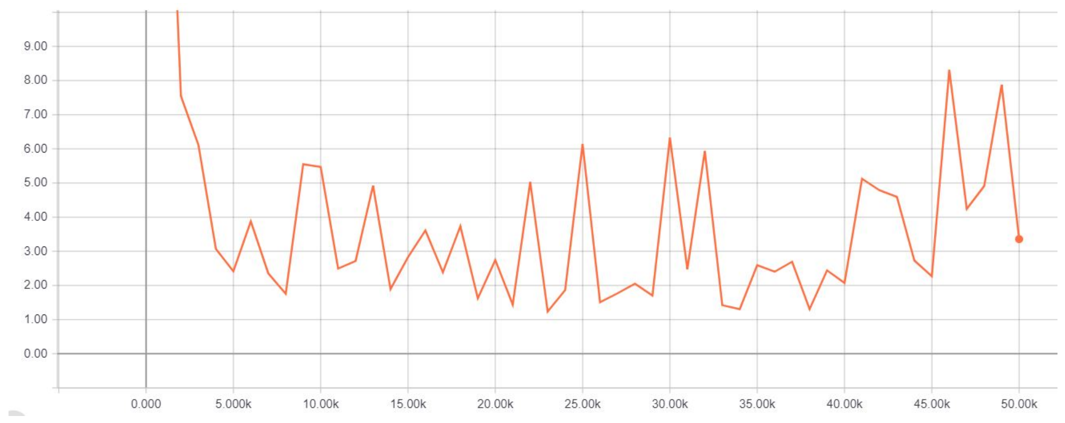

Figure 4.6: The KL divergence over the number of training steps for the Cauchy Bowman model with a z dimension of 1024 on the full training set

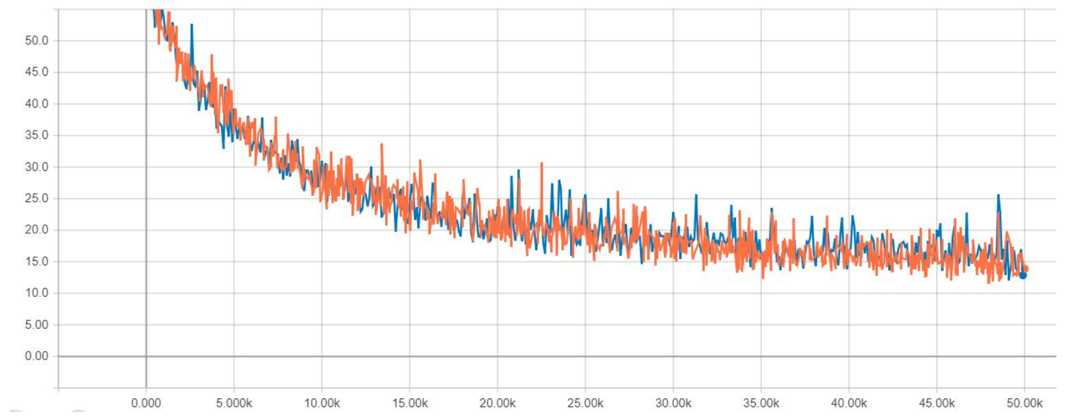

Figure 4.7: The training(orange) and testing(blue) loss over the number of the number of training steps for the Cauchy Bowman model with a z dimension of 1024 on the full training set.

Evaluation

Overall the Cauchy distribution appears to be an inappropriate choice for reconstructing text data. The volatility of the sampling is preventing the network from optimising as well as the other distributions. We made an approximation to calculate the KL divergence equation and this may also be having detrimental effects on performance.

4.6.7 Bowman Regularisation

When working on the full dataset, the von Mises-Fisher out-performed the other two distributions. The full training set for the Bowman model contains 1.2 million reviews and so even after 50,000 mini-batches of 64, each training example has only been seen a handful of times. We used a reduced training set in the Prototype and Edit model because of optimisation problems but we thought it would be interesting to explore how the performance of each distribution changes when we use a reduced training set for the Bowman model. The model will see the same training examples far more times and will begin to overfit towards the training set; we would like to see which distribution performs best on the test set when this happens. The results can be found in Table 4.9.

4.6.8 Evaluation

The von Mises-Fisher distribution has once again been shown to be the strongest distribution. It both optimises and generalises better than the Gaussian. The Cauchy distribution generalises better but this comes at a considerable cost for optimisation because of the volatility of the drawn samples.

Distribution	z Dimension	Steps	Train RL	Test RL
Gaussian	512	50000	1.5	35.5
vMF	512	50000	1.2	34.5
Cauchy	512	50000	4.1	32.6

Table 4.9: Comparing the effect of different distributions in the Bowman model on the reduced Yelp dataset

4.7 Final Models

We have learned that we gain the best performance at reconstructing sentences when using the von Mises-Fisher distribution with a hidden dimension of 1024. We now execute a longer train on this configuration to analyse further the generative qualities of this model.

4.7.1 Quantitative Results

Distribution	z Dimension	Steps	Reconstruction Loss	Word Accuracy	Jaccard	BLEU	Levenshtein
vMF	1024	100000	8.6	81.5%	0.17	0.8	2.09

4.7.2 Training Curves

We demonstrate how a number of metrics change over time in the training set (orange) and testing set (blue).

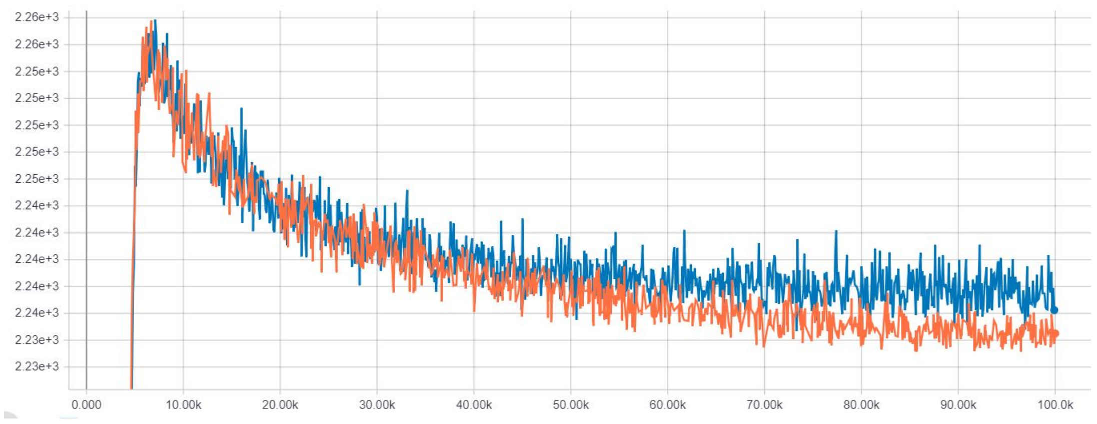

Figure 4.8: The training loss over the number of the number of training steps. The initial dramatic rise is due to the KL cost annealing schedule gradually increasing the KL term weight before it reaches 1 around step 5000.

4.8 Sentence Representation

For the network to be able to reconstruct sentences as well as it does, it must have learned a good latent representation for generating sentences. We have discussed in theory that this representation should include features such as topic and sentiment. We can use our trained model to see this effect in practice. We could verify basic hypothesis observing distances inbetween different encodings but it is far more powerful to compress our latent variable into a visual 2D representation.

4.8.1 UMAP

The UMAP algorithm is the state-of-the-art algorithm used to compress high dimensional data into visual representations. UMAP aims to build a graph of all our data points and then maintain this graph's

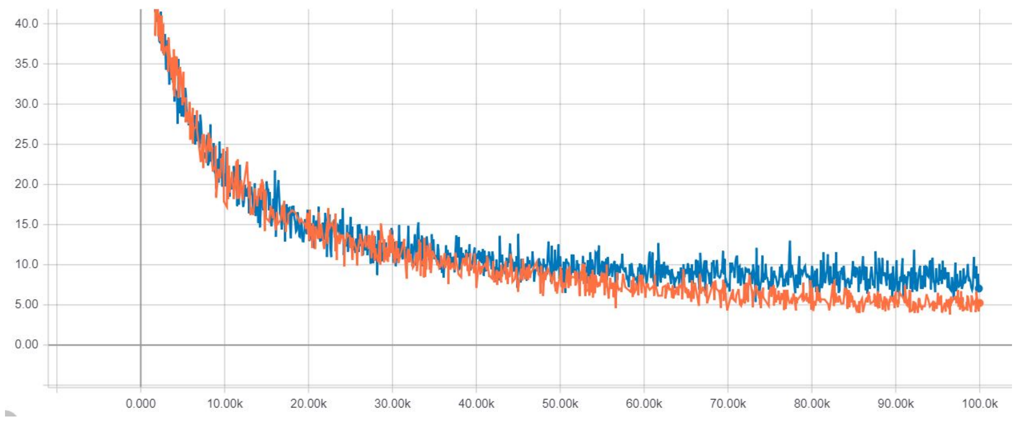

Figure 4.9: The reconstruction loss over the number of training steps. It can been seen that the model is beginning to overfit to the training data. The test set reconstruction loss is plateauing between 8 and 9.

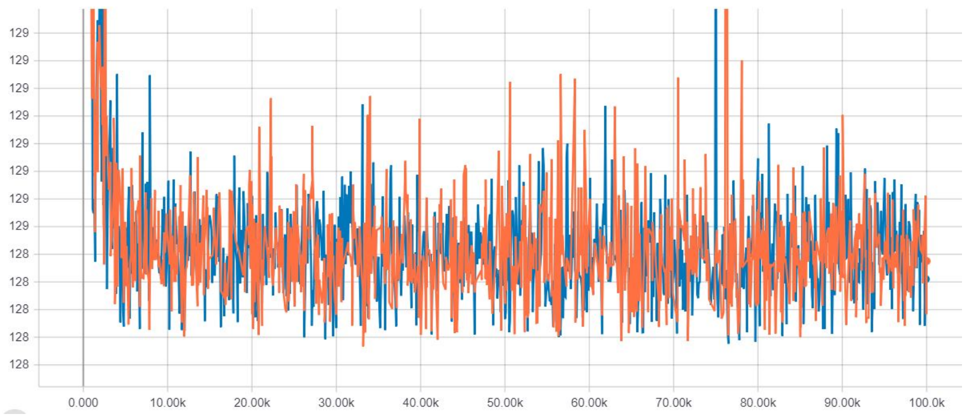

Figure 4.10: The learned parameter κ over the number of training steps. We initially see high values that make it easier for the model to optimise because of the annealing schedule. It quickly stabilises around the lower bound of 128.

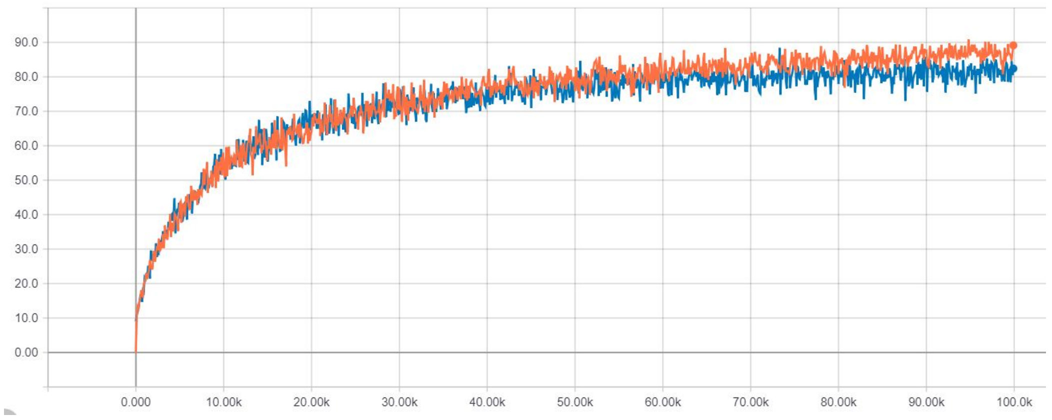

Figure 4.11: The word accuracy over the number of training steps. We can begin to see overfitting towards the training set.

structure in lower dimensions. A detailed explanation of the UMAP method can be found at [23].

4.8.2 Method

We encode 2000 sentences from our training set using our trained model; we then compress this data using UMAP such that each encoding is only 2 dimensions. We select a few individual sentences to reason further about the underlying features behind our representation of z. The resulting representation can be seen in Figure 4.12. We encode more sentences than just the displayed examples because more data points will lead to a better learned compression of our high dimensional representation.

We reason that the learnt representation encompasses the sentiment, style and topic of a sentence as well as the words it contains. To demonstrate this, we create 4 arbitrary classes and hand label 36 sentences that we believe fall into one of these categories.

- Negative: A review that would be seen as negative e.g. "Not worth anything near \$ ⟨money⟩."
- Great Service: A review that suggests the received service was exceptional e.g. "Amazing service, plus the staff is very kind."
- Great Food: A review that suggests the food at at a restaurant is good e.g. "The quality of food is high, and the variety is great."
- Food Item: A review that simply states the food that was ordered without any description of its quality e.g. "I got the eggplant parm special."

4.8.3 Visual Representation

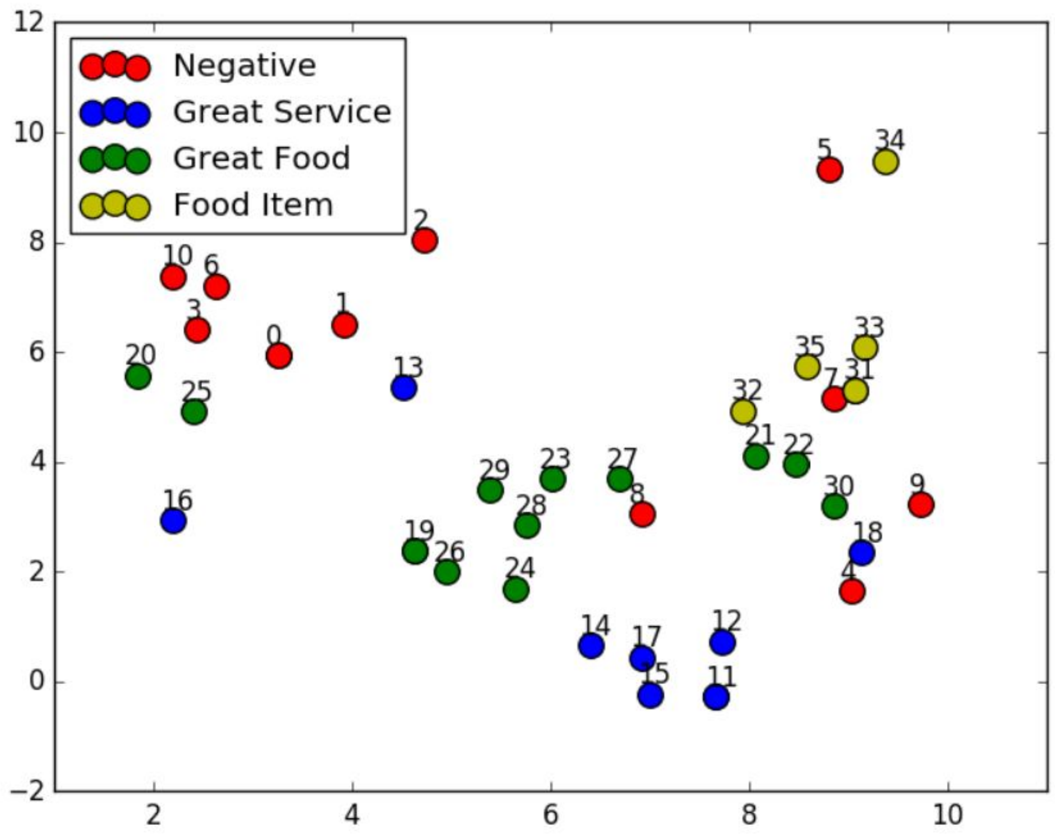

Figure 4.12: A UMAP compression of our learned representation of z into a 2D space with a few hand selected data points

Key	Sentence
1	He also acted irritated when we asked for him to split out checks.
2	⟨cardinal⟩ words come to mind , loud , dim and dirty .
3	Do n't waste your time or hard earned money visiting this establishment.
4	I usually do n't write reviews but my experience here was so bad.
5	Worst customer service at this place .
6	Not worth anything near $ ⟨money⟩.
7	Suffice to say, i wo n't be returning here.
8	My friend had the filet mignon and was very disappointed.
9	The food was just bland , bland , bland .
10	.... customer service sucks !
11	I don't think I 'll be heading back here.
12	The waitresses were friendly and attentive to our questions/comments.
13	Awesome service though, very attentive and reasonable prices.
14	Best customer service I've ever gotten at a nail salon.
15	Amazing service, plus the staff is very kind.
16	Both bartenders were very nice, personable, helpful, and friendly.
17	The best customer service that I ever experienced.
18	The service was great– our waitress was fast and friendly.
19	In ⟨cardinal⟩ words : excellent customer service!
20	The salsas are good and the chips are fresh when you order them.
21	The food was very good , better than I expected to be honest!
22	⟨gpe⟩ ⟨person⟩ has pretty good sushi.
23	The ⟨org⟩ dessert menu was incredible.
24	This place has the best ⟨gpe⟩ food here.
25	The food is great and the entertainment is awesome.
26	But either way, this place has some really good food.
27	The quality of food is high, and the variety is great.
28	Easily the best ⟨norp⟩ deli in ⟨gpe⟩.
29	I'm a ⟨norp⟩ food snob & would highly recommend this place!
30	⟨gpe⟩'s has some of the best tortillas ever.
31	i had the biscuits and gravy with sausage patties.
32	I got the eggplant parm special.
33	We also ordered the chips and pico de gallo.
34	It came with rice, beans and ⟨cardinal⟩ tortilla.
35	- two crab cakes for $ ⟨money⟩.
36	I ordered a turkey and avocado sandwich that came on a buttery croissant.

4.8.4 Analysis

Despite the categories being arbitrarily chosen and the expected inaccuracies from the representation being reduced from 1024 to 2, clear clusters have been formed. There are a number of points that fall outside their intended cluster in unintuitive ways. An example of this is the sentence 13: "Awesome service though, very attentive and reasonable prices.". This sentence is extremely positive but is close to a large number of negative reviews. With the small number of data points displayed as well as the effect of the compression to 2D, it is very hard to reason exactly why this is the case. There are a few other of these failure cases but we elect to focus more generally because of the lack of displayed data.

The least compressed cluster is the "Negative" category. This is not surprising considering how broad the variety of negative sentiment sentences can be in comparison to the other categories. The "Food Item" cluster is the most densely packed. This is most likely a result of all the sentences having a very similar structure e.g. starting with "I had" or "We ordered". The use of the pre-trained GloVe word embeddings will also have greatly aided in the ease of generalising across different food items.

Figure 4.12 clearly demonstrates the learned representation has encoded high level features such as topic and style. We can be increasingly confident in the quality of decoded sentences because of our strong representation. By adding noise to an encoded sentence, we would expect to decode a new sentence that remains on topic. This is essential for data augmentation because we want to be able to label generated sentences automatically by using the same label as the source sentence.

4.9 Sentence Generation

We aimed to design a system that could artificially generate sentences based on a training set such that we could augment our datasets. We have discussed the quantitative performance of our Bowman model at great length but we now focus on using it to generate new artificial sentences and evaluate their plausibility in addition to verifying they make grammatical sense.

4.9.1 Performance Metrics

To evaluate the quality of generated sentences we will use a variety of both quantitative and qualitative metrics.

Jaccard Distance

We introduced the Jaccard distance earlier in this section as a metric to measure the quality of a reconstruction. It is also a very effective tool and measuring the diversity of words between two sentences. We aim for our generated sentence to have a high Jaccard distance from their source sentence.

Levenshtein Distance

Another useful quantitative metric to evaluate our generated sentences is the Leuvensein distance. The Jaccard distance measures how different the words are between the source and generated sentences but we would also like a measure on the scale of transformation. For example a source sentence of “I thought the pizza was great” and a generated sentence “The pizza was great I thought” have a Jaccard distance of 0 but a high Leuvensein distance. Both of these measures are important at verifying the diversity of the generated sentence.

Plausibility

Throughout this thesis we have discussed the importance of generated sentences being plausible. Often this measure is ranked on a 1-3 scale. We believe that this scale is too ambiguous and simplify it: A sentence scores 0 if it is not plausible and 1 if it is. We clarify that plausibility in our case is a plausible review rather than being plausible within the English language. We clarify this further with examples of implausible sentences:

- “i ’ve tried ⟨cardinal⟩ kinds their cupcakes . :-) unfortunately”: Implausible because the word “unfortunately” is out of place given the positive sentiment of the rest of the review
- “i gave it a ⟨cardinal⟩ for ⟨cardinal⟩ , easy to ⟨cardinal⟩ , unk”: Sentences that make so little sense that we can not understand their context are also marked implausible

Although this qualitative measure is still somewhat subjective it should still provide a strong measure of the quality of the produced sentences.

Syntactic correctness

We also evaluate the syntactic correctness of our generated sentences using a 3 point scale. A sentence that makes no sense at all is given a score of 0; a sentence with a couple of minor mistakes is given a score of 1; a sentence with perfect grammar is given 2. This is a slightly less ambiguous measure opposed to plausibility because the English language is well defined.

4.9.2 Method

Given the requirement for human evaluation for our qualitative measures, we limit our evaluation to 128 samples from the validation set. We encode our samples into their z representation using our trained model. We then add some Gaussian noise to our encoding and then generate a new sentence based off our noisy z. The sentences generated should be moderately different to the sentence that was initially encoded. We vary the amount of noise added to our encoding and observe the trade offs between word diversity and the quality of the generated sentence.

4.9.3 Results

Noise variance	Jaccard	Leuventstein	Plausible	Grammatical Points
0	0.08	1.0	96.1%	94.2%
0.005	0.22	2.5	83.6%	75.4%
0.01	0.47	5.1	68.0%	50.3%

Table 4.10: Comparing the effect of generated sentences based on the level of noise added to encodings

4.9.4 Evaluation

As we increase the noise variance, our sentences become less plausible and less grammatically correct but our diversity measures increase. The ideal balance between diversity and correctness will vary depending on the NLP task. More training data is generally going to increase model performance but investigation is required to understand the trade off between the benefits of having a larger dataset through augmentation against having a number of sentences that may be grammatically incorrect. We focus on examining some individual success and failure cases where $\sigma^2_{noise} = 0.005$ to illustrate the effectiveness of the variational autoencoder at generating artificial sentences.

Success cases

1.
   - Source: I highly recommend this clinic.
   - Predicted: I highly recommend this dentist.
2.
   - Source: Not the very best sushi I've ever had but decent.
   - Predicted: Not quite the best sushi i 've ever had but decent.
3.
   - Source: This place was fun with a really cool vibe.
   - Predicted: This was clean with a really nice vibe.

The model was successful in generating some artificial sentences based on a source sentence that could be used to augment our datasets. These were mostly simple appropriate word substitutions shown in example 1 where “clinic” was replaced with “dentist”. In example 2, we can see that these substitutions are occasionally more than one word; in this case we have replaced “the very” with “quite” to form an appropriate sentence. In a few rare cases, the model will generate a new sentence with a different appropriate adjective shown in example 3.

Failure cases

1.
   - Source: There is plenty of parking here, and indoor and outdoor seating.
   - Predicted: There is plenty of there, there, and outdoor seating.
2.
   - Source: This is great hangout place for the college crowd.
   - Predicted: this is great hangout for hangout college crowd.

There were a number of failure cases when generating new artificial sentences. The majority of the failure cases are sentences being generated that do not make grammatical sense. These can be seen in both examples 1 and 2. These sentences tend to be those that contain less common vocabulary causing the model to struggle. The model either replaces there uncommon words with common words such as “there” as seen in example 1 or it overcompensates and places the unusual word in multiple times as seen in example 2. The main failure of the model is that we did not get any cases where sentences were reordered which would be a beneficial alteration. Within our small set of noisy samples, only small word substitutions were made.

4.10 Summary

We have evaluated in detail the implementation of both the Bowman and Prototype and Edit model. We have also explored the suitability of different distributions in each model and also the overall ability of the vMF Bowman model at generating artificial sentences. In the next chapter, we will summarise our findings and discuss possible extensions of the work outlined in this thesis.

Chapter 5

Conclusion

5.1 Contributions

In this thesis, we have explored the use of the variational autoencoder for generating artificial sentences for data augmentation. We utilised two different methods introduced by Bowman et al. [2] and Guu et al. [6] known in this paper as the "Bowman model" and the "Prototype and Edit model" respectively. We implemented both models from scratch using Python with the Tensorflow framework. The Bowman model is a variational autoencoder that encodes sentences as Gaussian distributions. Davidson et al. illustrated the weaknesses of the Gaussian distribution for certain types of data because it favours pushing points to the origin as well as being poor at modelling data in high dimensions. Guu et al. argued that the von Mises-Fisher distribution was a more appropriate distribution for text data because of the shape of the distribution. They replaced the conventional Gaussian distribution with the von Mises-Fisher distribution but crucially kept the concentration parameter, κ, fixed rather than it being learned from the input. They argue that this allows the developer to manually control the reconstruction cost with the KL divergence. Although this point has some merit and may be useful in some cases, we believe that we achieve better performance by allowing the network to learn this concentration parameter. We extended both of our models by adding configurations to allow the networks to encode sentences into vMF distributions with learned parameters of μ and κ. Furthermore, we added a Gaussian configuration to the Prototype and Edit model to verify the claim that the von Mises-Fisher is a more appropriate distribution.

Setting up the von Mises-Fisher distribution was not a trivial change in the models. The mathematical theory has already been outlined by Davidson et al [3]. We had to make a number of practical implementation changes to the outlined process to effectively learn κ. This included rearranging the KL divergence equation, adding a lower threshold to κ and adding a KL cost annealing schedule to the Prototype and Edit model. This is the first documented use of a von Mises-Fisher VAE to solve an NLP task.

We then introduced the Cauchy distribution which we argued may be superior to Gaussian because of its heavier tails. There is no documented use of the Cauchy distribution within the variational autoencoder. Because of the nature of the Cauchy distribution, it is mathematically very difficult to derive the equation for the KL divergence. We make an approximation by deriving the upper bound of the KL divergence and using this within our loss function. We then implemented a reparameterisation trick for the distribution to allow us to use it successfully within the variational autoencoder. We added the Cauchy distribution configuration to both models.

5.2 Distribution Investigation

Once we had implemented the use of the three distributions in both models, we performed an in-depth investigation into the most appropriate distribution for generating sentences. We compared performance over a number of different dimensions for both models as well as using different sized training sets. Overall we showed that the von Mises-Fisher distribution performed better than the common-choice Gaussian. This is an extremely key finding as it adds to the growing literature that Gaussian distribution is not the most appropriate distribution choice for every task.

The Cauchy distribution was far less successful because of its extreme variability when drawing samples. The Cauchy VAE showed signs of using wider contexts when generating sentences. Although this configuration performed poorly at reconstructing sentences, it may be learning a more robust representation. Furthermore, there may be other tasks in machine learning that have datasets where the Cauchy is a better fit. By deriving an approximation of the KL divergence equation, we have contributed by allowing other researches to investigate the Cauchy distribution within the variational autoencoder further.

5.3 Generating Sentences Using Variational Autoencoders

We investigated both the Bowman and Prototype and Edit models to create a VAE that could generate new artificial sentences to be used as training data in NLP tasks. We have implemented an extension of the Bowman model using the vMF distribution in a highly configurable manner that solves this task well. We demonstrated its ability at building a strong, high-level representation of sentences and how this representation can be used to generate sentences. We have contributed publicly-available code to the research community to allow NLP researchers to augment their datasets [21].

5.4 Evaluation of Aims

At the beginning of this project, we outlined a number a of key aims and objectives. We will now evaluate the progress on each of these aims.

1. **Gain a thorough theoretical and practical understanding of variational autoencoders**

   We spent a large amount of time understanding the theory behind the variational autoencoder proposed by Kingma et al [10]. Gaining a strong theoretical background was required for future steps in the project. We believe that this aim was achieved well and can be demonstrated by our explanation in Section 3.1.

2. **Implement two different approaches for variational autoencoders that generate sentences**

   We aimed to implement both model architectures from scratch to have two models that utilised two different methods of generating text using a variational autoencoder. The Bowman model was the simpler of the two methods and we believe that we implemented this well. We had no way to fully verify our implementation because the published results used a dataset that is not freely available and there was no public code provided by its authors. We analysed the training curves and the results we achieved and felt that we could be confident in our implementation. However, we did identify a potential weakness in our annealing schedule which could be potentially improved to increase model performance.

   The Prototype and Edit model implementation was far less successful. After fully understanding the paper, the method appeared to be only a small extension upon the Bowman to learn a representation of an edit as well as just the sentence. There is code publicly available for the Prototype and Edit model which we used for design inspiration. After our implementation did not perform nearly as well as expected, we explored this code further to identify any optimisation tricks that were not specified in the paper. The public code used a number of downloaded third-party libraries as well as using the PyTorch framework which we were unfamiliar with. The code was also poorly documented which made it extremely difficult to identify the tactics that the authors had used in building their successful model beyond what they theorised. We successfully added a number of small alterations on the initially proposed architecture but our model still fell far short of the benchmark. The implementation was still useful to quantitatively compare the performance of distribution but failed as a model to generate sentences.

3. **Implement a von Mises-Fisher variational autoencoder for generating text**

   Overall, we were successful in implementing the von Mises-Fisher VAE for both models. We successfully derived a new equation for the KL divergence that is possible to optimise for in high dimensional spaces as well as managing to create a modified Bessel function compatible with tensors. We did have to make a small approximation by adding a lower bound to the learned κ parameter to stop a divide by 0 zero error.

4. **Implement a Cauchy variational autoencoder for generating text**

   We were also successful in implementing a Cauchy variational autoencoder for both models. We reparameterised the equation using the method outlined by Kingma [10]. The KL divergence equation was mathematically difficult to derive so we elected to use the upper bounded equation which was far simpler. We deemed this to be an acceptable approximation but our implementation is not as mathematically robust in comparison to the other two distributions.

5. **Explore how different probability distributions affect the performance of the models**

   In Chapter 4, we performed an in-depth exploration into the performance of our three different distributions. We analysed a number of quantitative metrics over varying dimensions and assessed training curves generated from the optimisation phase. We aimed to build consensus over both models on which distribution was most appropriate for the sentence generation task. We achieved this in identifying the vMF distribution as the most suitable in the majority of cases.

   However, the training times of our models were extremely long which somewhat limited the configurations we were able to experiment with. Furthermore, we identified weaknesses in the behaviour of the KL divergence in the Gaussian and vMF distribution which limited the validity of our work. A more optimum annealing schedule or a fixed scaler for the vMF KL term could have slightly altered our results. Overall, we believe that we performed a strong investigation.

5.5 Future Work

We have discussed the achievements and shortcomings of the project as well as the outcomes of our investigations. In this section, we explore our findings from Chapter 4 further by suggesting improvements to our models. We also suggest areas that should be examined to verify our results and provide interesting insight.

5.5.1 Improve Prototype and Edit Model

The first element of future work would be to improve the Prototype and Edit model performance. Our model fell far below the performance benchmark set up the publicly available code. This still provided a useful comparison between our different distribution configurations but a better working model would have allowed us to do more meaningful analysis. The model struggled to optimise on the entire training set and we were forced to use a significantly reduced dataset. Experimenting more with hyperparameters and different optimisers may dramatically improve the performance from our implementation.

5.5.2 KL Cost Annealing Schedule

In Chapter 3, we discussed in detail the potential weakness of our KL cost annealing schedule. We decided to use a sigmoid that increased to a weight of 1 after roughly 5000 training steps. We devised this by observing the magnitude of the KL term at different time steps and deliberately elected to use an annealing schedule very similar to Bowman [2]. In our experiments on our implementation, we found that the KL divergence in the Gaussian and vMF configuration optimised to the prior very quickly and did not change much as training progressed. It is possible that this is the optimum balance between the KL divergence term and the reconstruction loss but is more likely a poorly chosen annealing schedule. Stretching the sigmoid over a larger number of steps or using a different function may improve the optimisation process and allow the network to better balance the reconstruction loss and the KL divergence term. This should improve the performance of the model and may provide different results when analysing the effects of different distributions within our models.

5.5.3 vMF KL Divergence

In higher dimensions, the KL divergence of the von Mises-Fisher distribution will always be far larger than our reconstruction cost because of our lower threshold for κ. This means that that even with a better annealing schedule, our model optimiser will overly focus on reducing the κ term rather than finding an optimum representation to minimise the reconstruction cost. To extend our model further, we could experiment the effect of adding a scalar term β for our KL divergence as well as our annealing schedule such that:

$$Loss = ReconstructionLoss + \beta \cdot f(s) \cdot KL \tag{5.1}$$

where f is our annealing schedule on step, s

This scalar term would control the KL divergence term so that it is more similar in magnitude to the reconstruction cost. This would prevent the model from focusing exclusively on reducing κ rather than considering its effect on the strength of the representation.

5.5.4 Dynamic vMF threshold

Furthermore, in our vMF VAE we had to add a lower threshold to our κ term of $\frac{m}{8}$ where m is the dimension of our z vector. We calculated this from experimentation with our KL divergence calculation as values of κ too small would cause a divide by zero error. Although this threshold worked for all our dimension sizes that we experimented with it, a linear threshold is not sensible for an exponential function. The threshold is inappropriate when $m > 1024$ and κ could be lower than the threshold for smaller values of m without an error. Experimenting with this threshold could provide a more optimum flexibility of our κ parameter and combined with our annealing schedule could aid in improving the vMF VAE further.

5.5.5 Representation and Generation Investigation

In sections 4.8 and 4.9, we evaluated the learnt representation for our model using our vMF model. We discussed in Chapter 4 that a lower reconstruction loss does not necessarily equate to a better representation for our model when it comes to generating new unseen sentences. It would be interesting to investigate this effect further by doing more qualitative evaluation on the quality of sentences generated with different distribution configuration.

5.5.6 Exploring Data Augmentation Improvements

The motivation for this thesis was to develop a model that can generate artificial sentences to be used to augment datasets in NLP tasks. We were able to create an effective model to complete this task. An interesting area for investigation would be to see how much using the Bowman model to increase training sets could improve the performance of models solving NLP tasks. This would provide more insight into understanding how to balance the trade off between diversity and syntactic correctness in generated sentences.

5.5.7 Different Problems

Throughout this thesis we have argued that the vMF distribution is a more effective distribution to model text data compared to the Gaussian. Through our investigation we found this hypothesis correct for the task of generating text. There are a number of problem within NLP that utilise variational autoencoders other than generating sentences, examples include question answering or machine translation. Exploring using different distribution configurations within these other problems would further help to verify our hypothesis and extend the findings of this thesis.

5.6 Summary

In this thesis, we have implemented two different methods for generating sentences using variational autoencoders. We reasoned that the von Mises-Fisher and the Cauchy distribution could be more appropriate at modelling text data. We have introduced the first documented use of a vMF VAE for an NLP task as well as proposing the first documented Cauchy VAE. We experimented over a number of configurations within both of our models and found the von Mises-Fisher to be the strongest performing model.

By showing the superiority of the von Mises-Fisher distribution for modelling word vectors in the sentence generation problem, we have encouraged its use in other NLP tasks. Using this project as reference, this extension could easily be applied to other state-of-the-art variational autoencoder NLP models and improve their performance. Furthermore, we have also added to the growing literature that the Gaussian distribution is not the optimal choice for all types of data. By using different distributions, we can develop a more refined approach for representing our data and push the performance boundaries of variational autoencoders across all fields of machine learning.

Bibliography

[1] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[2] Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

[3] Tim R Davidson, Luca Falorsi, Nicola De Cao, Thomas Kipf, and Jakub M Tomczak. Hyperspherical variational auto-encoders. *arXiv preprint arXiv:1804.00891*, 2018.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[5] Kelvin Guu. Prototype and edit model github, 2018.

[6] Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. Generating sentences by editing prototypes. *Transactions of the Association of Computational Linguistics*, 6:437–450, 2018.

[7] Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. Generating sentences by editing prototypes codalab, `https://worksheets.codalab.org/worksheets/0xa915ba2f8b664ddf8537c83bde80cc8c/`, 2019.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] Michael I. Jordan. Bayesian modeling and inference lecture: Monte carlo sampling, April 2010.

[10] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[11] Gerhard Kurz and Uwe Hanebeck. *Stochastic Sampling of the Hyperspherical von MisesFisher Distribution Without Rejection Methods*. 10 2015.

[12] Christian A Naesseth, Francisco JR Ruiz, Scott W Linderman, and David M Blei. Reparameterization gradients through acceptance-rejection sampling algorithms. *arXiv preprint arXiv:1610.05683*, 2016.

[13] Christopher Olah. Understanding lstm networks.

[14] Abhishek Parbhakar and Abhishek Parbhakar. Why data scientists love gaussian?, May 2018.

[15] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[16] Diana Pérez, Enrique Alfonseca, and Pilar Rodríguez. Application of the bleu method for evaluating free-text answers in an e-learning environment. In *LREC*, 2004.

[17] Fariz Rahman. Seq2seq, 2017.

[18] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[19] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11), November 1997.

[20] SpaCy. spacy annotation specifications, https://spacy.io/api/annotation#named-entities, 2019.

[21] Adam Stein. Thesis implementation, https://github.com/AdamStein593/Generative-Language-Models.

[22] Greg Corrado Tomas Mikolov, Kai Chen and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[23] UMAP. How umap works, https://umap-learn.readthedocs.io/en/latest/how_umap_works.html.

[24] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[26] Jason Wang and Luis Perez. *The Effectiveness of Data Augmentation in Image Classification using Deep Learning*. 2017.

[27] Lilian Weng. From autoencoder to beta-vae, Aug 2018.

[28] Yelp. Yelp open dataset, https://www.yelp.com/dataset, 2019.