Exploring Variational Auto-Encoders for Topic Modelling

Apoorva Jarmale,Kaavya Gowthaman,Oj Sindher,Sarang Pande,Vartika Tewari

April 1, 2021

Project Milestone Report (DS 5230: USML — Spring 2021)

Abstract

Topic models are one of the most popular methods for learning representations of text, but a major challenge is that any change to the topic model requires mathematically deriving a new inference algorithm. In this project, we implement a model called ProdLDA, that replaces the mixture model in LDA with a product of experts. This model tackles the problems caused for autoencoding variational Bayes by the Dirichlet prior and by component collapsing. We find that Autoencoded Variational Inference For Topic Model matches traditional methods in accuracy with much better inference time. The data used for the analysis is 20 Newsgroup data set. The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. Each sub directory in the bundle represents a newsgroup; each file in a sub directory is the text of some newsgroup document that was posted to that newsgroup. In project milestone report, we are implementing variational autoencoder(standard LDA) on MNIST data set.

1 Introduction

Response:

The variational inference methods tries to solve the major challenge of applying topic models to the text data, which is the computational cost of computing the posterior distribution, as posterior inference over the hidden variables θ and z is intractable due to the coupling between the θ and β under the multinomial assumption (Dickey, 1983). Though these inference methods have been used and developed by a large body of work, they posses a common issue when applying them to new models, that is, it relies on the practitioner's ability to derive the closed form updates, which can be impractical and sometimes impossible. This has motivated the development of black-box inference methods (Ranganath et al., 2014; Mnih Gregor, 2014; Kucukelbir et al., 2016; Kingma Welling, 2014; Srivastava et al., 2017).

Autoencoding variational Bayes (AEVB) is a black-box inferencing method which trains an inference network (Dayan et al., 1995) to map a document to an approximate posterior distribution, though practically AEVB faces several challenges, such as, we need to determine the reparameterization function for q(θ) in order to use the "reparameterization trick" Rezende et al. (2014), which is essential because it allows the backpropagation of gradient to flow from deterministic nodes instead of the random nodes. The ProdLDA model, which is advancement of AEVB makes them stable, and it also has been able to address

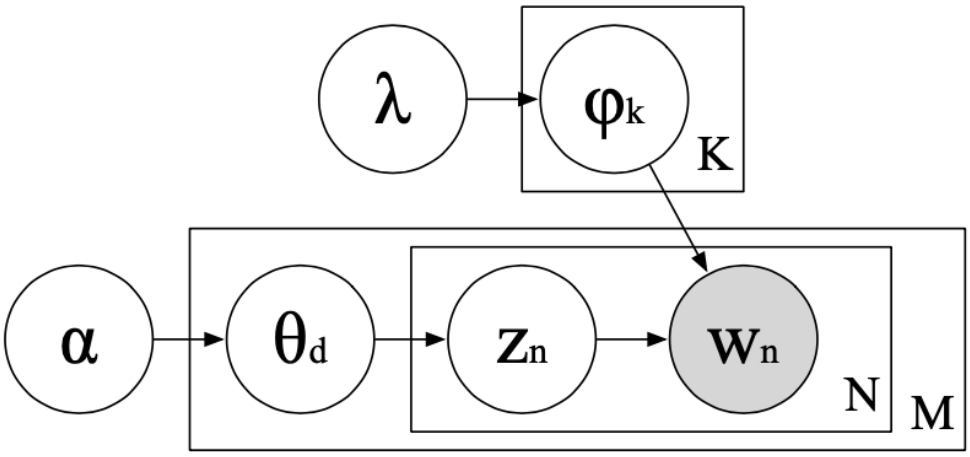


Figure 1: Plate diagram for LDA. Nodes show random variables, lines show (possible) probabilistic dependence, rectangles show repetition, and shading shows observation

the drawbacks of implementing topic models by either using variational inference methods such as mean field or Gibbs sampling methods, as shown by Srivastava et al. (2017) in their research paper where they presented ProdLDA model for the first time. It is very interesting that how changing one assumption in LDA model results in drastic improvement in topic coherence. Basically, ProdLDA model just replaces the mixture assumption at the word-level in LDA with a weighted product of experts. In other words, ProdLDA is an instance of the exponential-family PCA (Collins et al., 2001) class, and relates to the exponential-family harmoniums (Welling et al., 2004) but with non-Gaussian priors.

We would be implementing the ProdLDA model which is a very effective AEVB inference method for text data. The model described promises advantages like topic coherence, computational efficiency, and a Black-box methodology i.e. it does not require rigorous mathematical derivations to handle changes in the model, and can be easily applied to a wide-range of topic models.

2 Background

Topic Models are used to assign topics to documents. There are various techniques for topic modelling , some of the earliest techniques include probabilistic latent semantic analysis (PLSA)[2], Latent Dirichlet allocation (LDA)[1]. Latent Drichlet Allocation assumes the generative process of the data. Each document contains some topics and each word can be mapped to one of the document's topics. Other topic models are generally extensions of LDA.

In this project we will focus on ProdLDA[4] which uses Variational Autoencoders to do topic modeling.

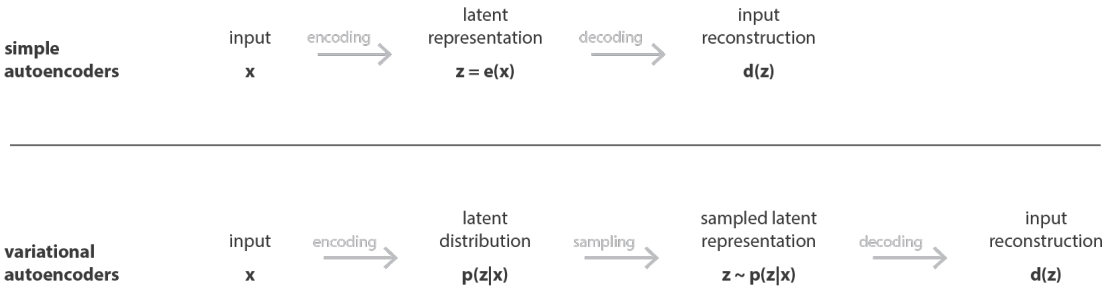


Figure 2: Difference between autoencoder and variational autoencoder

2.1 Variational Inference using Autoencoders

2.1.1 Autoencoders

Autoencoders are neural networks which tries to recreate the input as its output. It consists of two parts encoder and decoder. The encoder maps the input to an embedding and the decoder maps the embedding to the reconstructed input. They are mostly used for dimentionality reduction or feauture learning.

2.1.2 Variational Autoencoders[3]

To make a generative model we add regularisation in the training process of the autoencoder. So instead of encoding an input as a single point, we encode it as a distribution over the latent space. This variation now uses the point sampled from a distribution for the decoding process.

ELBO Loss: The loss function for the VAE is called the ELBO.

$$min\mathbb{E}_q \log p(z) - \mathbb{E}_q \log p(x|z)$$

The first term is the KL divergence. The second term is the reconstruction term.

3 Preliminary Results

For the preliminary results, we worked on understanding and running a simple variational autoencoder. VAE implemented here uses the setup found in most VAE papers:a multivariate Normal distribution for the conditional distribution of the latent vectors given and input image and a multivariate Bernoulli distribution for the conditional distribution of images given the latent vector. Using a Bernoulli distribution, the reconstruction loss (negative log likelihood of a data point in the output distribution) reduces to the pixel-wise binary cross-entropy. This VAE has been used with the MNIST dataset. We learnt that in this case, a convolutional encoder and decoder, gives better performance than fully connected versions that have the same number of parameters.
Following is the parameter setting :

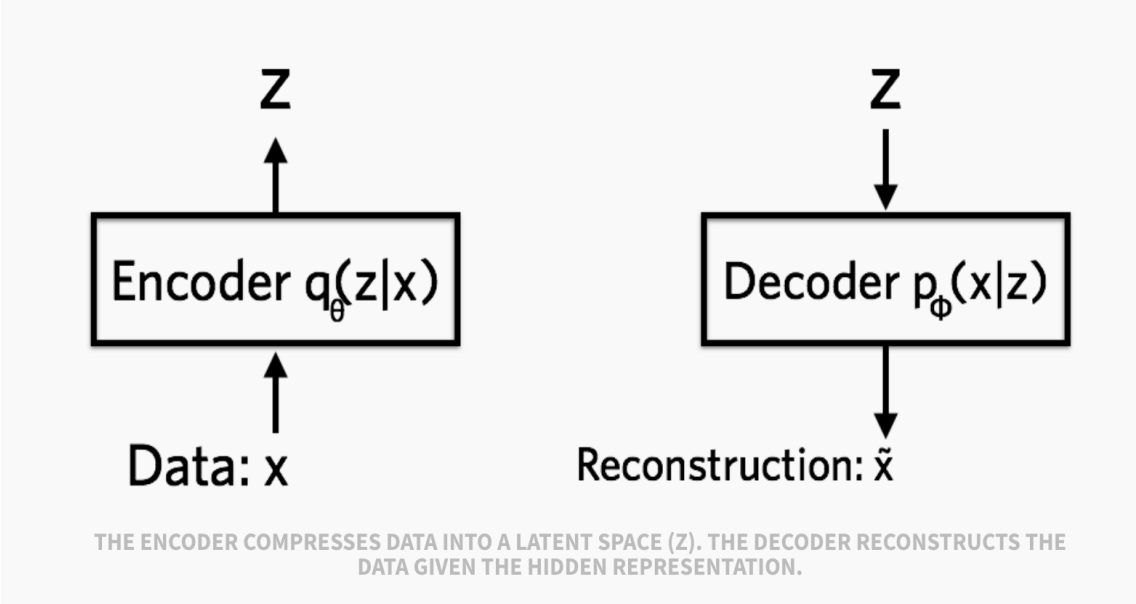
```
# 2-d latent space, parameter count in same order of magnitude
latent_dims = 2
num_epochs = 100
batch_size = 128
capacity = 64
learning_rate = 1e-3
variational_beta = 1
use_gpu = True
```

As a preliminary dataset, we have considered MNIST. MNIST images show digits from 0-9 in 28x28 grayscale images. We do not center them at 0, because we will be using a binary cross-entropy loss that treats pixel values as probabilities in [0,1]. We create both a training set and a test set as follows :

```
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST

img_transform = transforms.Compose([
    transforms.ToTensor()
])

train_dataset = MNIST(root='./data/MNIST', download=True, train=True, transform=img_transform)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

test_dataset = MNIST(root='./data/MNIST', download=True, train=False, transform=img_transform)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/MNIST/raw/train-images-idx3-ubyte.gz
9913344/? [05:05<00:00, 32448.60it/s]

Extracting ./data/MNIST/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/MNIST/raw/train-labels-idx1-ubyte.gz
29696/? [01:43<00:00, 288.06it/s]

Extracting ./data/MNIST/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/MNIST/raw/t10k-images-idx3-ubyte.gz
1649664/? [00:51<00:00, 31809.37it/s]

Extracting ./data/MNIST/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/MNIST/raw/t10k-labels-idx1-ubyte.gz
5120/? [01:50<00:00, 46.53it/s]
```

encoder is a neural network. Its input is a datapoint x, its output is a hidden representation z, and it has weights and biases θ.
The encoder learns an efficient compression of the data into this lower-dimensional space.Suppose we denote the encoder $q_\theta(z|\, x)$. We note that the lower-dimensional space is stochastic: the encoder outputs parameters to $q_\theta(z|\, x)$, which is a Gaussian probability density. We can sample from this distribution to get noisy values of the representations z.
The decoder is another neural net. Its input is the representation z, it outputs the parameters to the probability distribution of the data, and has weights and biases θ. The decoder is denoted by $p_\theta(x|\, z)$.



THE ENCODER COMPRESSES DATA INTO A LATENT SPACE (Z). THE DECODER RECONSTRUCTS THE DATA GIVEN THE HIDDEN REPRESENTATION.

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        c = capacity
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=c, kernel_size=4, stride=2, padding=1) # out: c x 14 x 14
        self.conv2 = nn.Conv2d(in_channels=c, out_channels=c*2, kernel_size=4, stride=2, padding=1) # out: c x 7 x 7
        self.fc_mu = nn.Linear(in_features=c*2*7*7, out_features=latent_dims)
        self.fc_logvar = nn.Linear(in_features=c*2*7*7, out_features=latent_dims)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = x.view(x.size(0), -1) # flatten batch of multi-channel feature maps to a batch of feature vectors
        x_mu = self.fc_mu(x)
        x_logvar = self.fc_logvar(x)
        return x_mu, x_logvar

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        c = capacity
        self.fc = nn.Linear(in_features=latent_dims, out_features=c*2*7*7)
        self.conv2 = nn.ConvTranspose2d(in_channels=c*2, out_channels=c, kernel_size=4, stride=2, padding=1)
        self.conv1 = nn.ConvTranspose2d(in_channels=c, out_channels=1, kernel_size=4, stride=2, padding=1)
    def forward(self, x):
        x = self.fc(x)
        x = x.view(x.size(0), capacity*2, 7, 7) # unflatten batch of feature vectors to a batch of multi-channel feature maps
        x = F.relu(self.conv2(x))
        x = torch.sigmoid(self.conv1(x)) # last layer before output is sigmoid, since we are using BCE as reconstruction loss
        return x
```

The VAE is implemented with the reparametrization trick shown as follows :

```
class VariationalAutoencoder(nn.Module):
    def __init__(self):
        super(VariationalAutoencoder, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        latent_mu, latent_logvar = self.encoder(x)
        latent = self.latent_sample(latent_mu, latent_logvar)
        x_recon = self.decoder(latent)
        return x_recon, latent_mu, latent_logvar

    def latent_sample(self, mu, logvar):
        if self.training:
            # the reparameterization trick
            std = logvar.mul(0.5).exp_()
            eps = torch.empty_like(std).normal_()
            return eps.mul(std).add_(mu)
        else:
            return mu
```

The loss function of the variational autoencoder is the negative log-likelihood with a regularizer. The loss function l_i for datapoint x_i is :

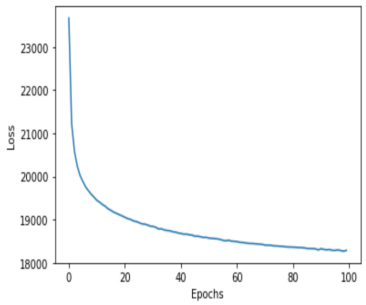
$$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i \mid z)] + \mathbb{KL}(q_\theta(z \mid x_i) \mid\mid p(z))$$

The first term is the reconstruction loss, or expected negative log-likelihood of the ith datapoint. The expectation is taken with respect to the encoder's distribution over the representations. This term encourages the decoder to learn to reconstruct the data.
The second term is a regularizer . This is the Kullback-Leibler divergence between the encoder's distribution $q_\theta(z|x)$ and p(z). This divergence measures how much information is lost when using q to represent p. It is one measure of how close q is to p.In the variational autoencoder, p is specified as a standard Normal distribution with mean zero and variance one, or p(z) = Normal(0,1)p(z)=Normal(0,1). If the encoder outputs representations z that are different than those from a standard normal distribution, it will receive a penalty in the loss.
We train the variational autoencoder using gradient descent to optimize the loss with respect to the parameters of the encoder and decoder θ and ϕ. For stochastic gradient descent with step size ρ, the encoder parameters are updated using $\theta \leftarrow \theta - \rho \frac{\partial l}{\partial \theta}$ and the decoder is updated similarly.

```
optimizer = torch.optim.Adam(params=vae.parameters(), lr=learning_rate, weight_decay=1e-5)
# set to training mode
vae.train()
train_loss_avg = []
print('Training ...')
for epoch in range(num_epochs):
    train_loss_avg.append(0)
    num_batches = 0

    for image_batch, _ in train_dataloader:

        image_batch = image_batch.to(device)

        # vae reconstruction
        image_batch_recon, latent_mu, latent_logvar = vae(image_batch)

        # reconstruction error
        loss = vae_loss(image_batch_recon, image_batch, latent_mu, latent_logvar)

        # backpropagation
        optimizer.zero_grad()
        loss.backward()

        # one step of the optmizer (using the gradients from backpropagation)
        optimizer.step()

        train_loss_avg[-1] += loss.item()
        num_batches += 1

    train_loss_avg[-1] /= num_batches
    print('Epoch [%d / %d] average reconstruction error: %f' % (epoch+1, num_epochs, train_loss_avg[-1]))
```

After training,we observe the reconstruction error decreasing and stabilising as shown:

```
[ ] import matplotlib.pyplot as plt
    plt.ion()

    fig = plt.figure()
    plt.plot(train_loss_avg)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.show()
```



We evaluated on the test set we had initialized earlier and check the reconstruction error :

```
[ ]  # set to evaluation mode
     vae.eval()

     test_loss_avg, num_batches = 0, 0
     for image_batch, _ in test_dataloader:

         with torch.no_grad():

             image_batch = image_batch.to(device)

             # vae reconstruction
             image_batch_recon, latent_mu, latent_logvar = vae(image_batch)

             # reconstruction error
             loss = vae_loss(image_batch_recon, image_batch, latent_mu, latent_logvar)

             test_loss_avg += loss.item()
             num_batches += 1

     test_loss_avg /= num_batches
     print('average reconstruction error: %f' % (test_loss_avg))
```

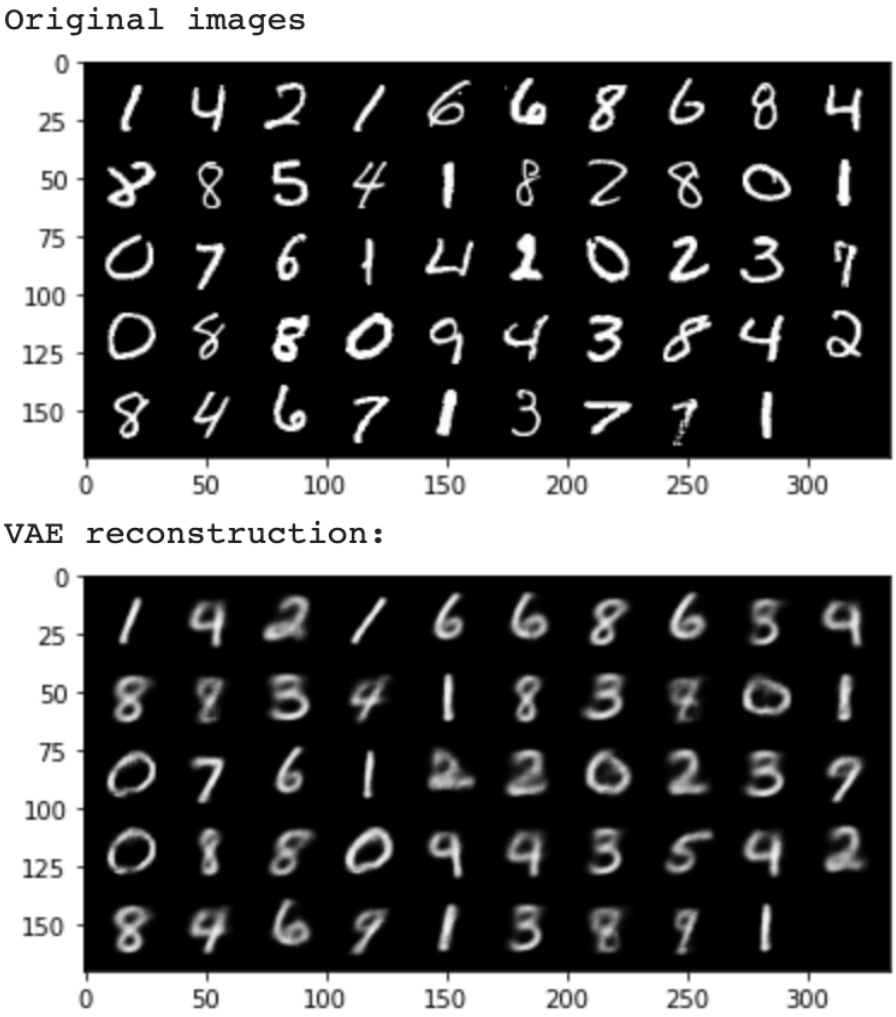
```
average reconstruction error: 19174.169347
```

Next, we visualize and compare the reconstruction with our original image. This function takes as an input the images to reconstruct and the name of the model with which the reconstructions are performed.

```
import numpy as np
import matplotlib.pyplot as plt
plt.ion()
import torchvision.utils
vae.eval()
def to_img(x):
    x = x.clamp(0, 1)
    return x
def show_image(img):
    img = to_img(img)
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
def visualise_output(images, model):
    with torch.no_grad():
        images = images.to(device)
        images, _, _ = model(images)
        images = images.cpu()
        images = to_img(images)
        np_imagegrid = torchvision.utils.make_grid(images[1:50], 10, 5).numpy()
        plt.imshow(np.transpose(np_imagegrid, (1, 2, 0)))
        plt.show()
images, labels = iter(test_dataloader).next()
# First visualise the original images
print('Original images')
show_image(torchvision.utils.make_grid(images[1:50],10,5))
plt.show()
# Reconstruct and visualise the images using the vae
print('VAE reconstruction:')
visualise_output(images, vae)
```

.

The results are as follows:



Link to code : Colab (Jupyter Notebook)

4 Exploratory Data Analysis

Exploratory Data Analysis on the 20 Newsgroup Dataset can be found here: Colab (Jupyter Notebook)

5 Discussion

Now having an understanding of how variational inference works and implementing a variational autoencoder for MNIST data. We will next recreate the ProdLDA implementation using Pytorch and apply it to 20Newsgroup dataset.

We will analyse our results and compare them with the paper's implementation, and also make a baseline with LDA.

References

[1] David M Blei, Andrew Y Ng, and Michael I Jordan. "Latent dirichlet allocation". In: *the Journal of machine Learning research* 3 (2003), pp. 993–1022.

[2] Thomas Hofmann. "Probabilistic latent semantic indexing". In: *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. 1999, pp. 50–57.

[3] Diederik P Kingma and Max Welling. "An introduction to variational autoencoders". In: *arXiv preprint arXiv:1906.02691* (2019).

[4] Akash Srivastava and Charles Sutton. "Autoencoding variational inference for topic models". In: *arXiv preprint arXiv:1703.01488* (2017).