VAE Problem Set

Group U8: Lake Yin, Zhiqi Wang

April 24, 2023

In the lecture of VAE, we've learned that the variational autoencoder (VAE) as a tweak of autoencoder with given objective. In this problem set we will explore how ELBO is the objective function of VAE.

Problem 1

The equation of log likelihood we used in the lecture is:

$$\log p_\theta(x) = \mathcal{L}_{\theta,\phi}(x) + KL(q_\phi(z|x)||p_\theta(z|x))$$

The first term on RHS is the ELBO, which is the objective function to use for VAE. Rewrite this equation to explain why that would be called evidence lower bound (ELBO).

Problem 2

Here's the code implementation of VAE in PyTorch. (https://github.com/ZhiqiEliWang/csci4968-VAE-project). In assignment 4, we have done an reconstruction of Fashion-MNIST dataset. In this problem, we will build a VAE on assignment 4 based on our implementation of VAE. Import the data and partition it into a test set and training set.

1. Encoder class

Write a class `Encoder` that is a subclass of PyTorch `nn.module` that implements the encoder section of a VAE. To do this, the class should implement the following functions:

Constructor

An encoder constructor `__init__(self, c1, c2, lat, k)`, where `c1` is the output channel size of the first convolution layer, `c2` is the output channel size of the second convolution layer, `lat` is the latent dimension size, and `k` is the kern3l size. Do not forget `super(Encoder, self).__init__()`. The constructor initialize two convolution layers with `Conv2d`, as well as two linear layers, `mean` and `std`. These two linear layers should both be the same size and accept a flattened version of the final output from the convolution layers. The linear layer size can be determined with the formula

$$c(d-(k-1))^2$$

where c is the number of output channels for the last layer, d is the height or width of the image input, and k is the kernel size. We recommend a kernel size of 3 and output channel sizes of 16 and 32 for the first and second convolution layers, respectively.

Forward function

A function `forward(self, x)`, where `x` is a batch of images. The function should apply the convolution layers to `x` in order with ReLu activation, then flatten the output into one dimension. Then, the function should calculate `mu` using the `mean` linear layer on the flattened output and calculate `sigma` by applying the exponential function to output of passing flatten output through the `std` layer. The function should return `mu` and `sigma` as a tuple.

2. Decoder class

Write a class `Decoder` that is a subclass of PyTorch `nn.module` that implements the decoder section of a VAE. To do this, the class should implement the following functions:

Constructor

An encoder constructor `__init__(self, c1, c2, lat, k)`, where `c1` is the output channel size of the first encoder convolution layer, `c2` is the output channel size of the second encoder convolution layer, `lat` is the latent dimension size, and `k` is the kernel size. Do not forget `super(Decoder, self).__init__()`. The constructor initialize a linear layer that matches the size of the `mean` or `std` encoder layers, and two transpose convolution layers with `ConvTranspose2d` with output channel sizes in reverse order to the encoder convolution layers. (So `c2` and `c1`, in that order.)

Forward function

A function `forward(self, z)`, where `z` is a batch of latent vectors. The function should apply ReLu to `z`, then unflatten `z` into a batch of images with size `(b, c2, d, d)` where `b` is the batch size. Then, the function should apply the transpose convolution layers and return the output. The first transpose convolution layer should have ReLu activation and the second layer should have sigmoid activation.

3. VAE class

Write a class `VAE` that is a subclass of PyTorch `nn.module` that implements the a full VAE. To do this, the class should implement the following functions:

Constructor

An encoder constructor `__init__(self, c1, c2, lat, k)`, where `c1` is the output channel size of the first encoder convolution layer, `c2` is the output channel size of the second encoder convolution layer, `lat` is the latent dimension size, and `k` is the kernel size. Do not forget `super(VAE, self).__init__()`. The constructor initialize an `Encoder` object and a `Decoder` object.

Reconstruction loss function

A function `r_loss(self, x, y)`, where `x` is a batch of autoencoder output images and `y` is a batch of ground truth images. The function should calculate the loss between the output and ground truth with total sum binary crossentropy formula

$$-[x \odot \log y + (1 - x) \odot \log(1 - y)]$$

where $\odot$ is the element-wise multiplication operator.

KL divergence function

A function `kl_loss(self, mu, sigma)`, calculates the KL divergence given `mu` and `sigma` between two probability distributions. As discussed, the goal of this function is to force the latent space into a normal distribution. The KL divergence for a normal distribution in this case can be calculated as

$$\mu^2 + \sigma^2 - \log \sigma - \frac{1}{2}$$

Gaussian function

A function `gaussian(self, mu, sigma)`, which performs the reparameterization trick on `mu` and `sigma`. This function should return the output of

$$\mu + \sigma \odot \mathcal{N}$$

where $\mathcal{N}$ is a normal distribution tensor with shape equal to `sigma`, a mean of 0, and a standard deviation of 1.

Forward function

A function `forward(self, x)`, where `x` is a batch of images. The function should apply the encoder object to `x` to obtain `mu` and `sigma`, then apply `gaussian` to `mu` and `sigma` to obtain `z`, and then apply the decoder to `z` to obtain `y`. Calculate `loss` by adding together `r_loss(x, y)` and `kl_loss(mu, sigma)` and store `loss` as an instance variable. Return `y`.

Backward function

A function `backward(self)`, which initiates backpropagation. It should call the `backward` function for `loss` by calling `self.loss.backward()`.

2. Train VAE

The VAE should be trained using the PyTorch `Adam` optimizer approximately as follows:

```
vae = VAE(c1, c2, lat, k)
adam = Adam(vae.parameters())

for e in range(num_epochs):
    loss = 0
    for x_batch, _ in training_data:
        adam.zero_grad()
        vae.forward(x_batch)

        loss += vae.loss
        vae.backward()
        adam.step()

    print(f"Training loss [{e + 1}/{num_epochs}]: {loss:0.4f}")
```

3. Feature selection

Apply the trained encoder `vae.encoder` and `gaussian` function to the test set images to obtain a list of embeddings. Select and store the indices of the two latent dimensions with the highest variance.

4. Verify normal distribution

Slice the list of embeddings using the two indices. Plot both of these slices using Matplotlib `hist2d(slice1, slice2, bins=50)`. Verify that the distribution is centered on 0 and roughly resembles a normal distribution.

4. Perturb along one dimension

Generate new images from the latent space. Create a list of 7 evenly spaced values from 0.05 to 0.95. Apply the SciPy `norm.ppf` function to this list to obtain numbers from the normal distribution. For each number `k`, create a vector of size `d` with all entries set to 0. Set the entry at one index to `k`. This index should be from one of the indices obtained earlier. Apply the trained decoder `vae.decoder` to these vectors and display the output images as a 1 by 7 grid. Verify that the images change consistently from one end to the other. As a bonus, do this with both previously obtained indices simultaneously to create a 7 by 7 grid.