ECE 175B - Variational AutoEncoder

Anonymous Author(s)
Affiliation
Address
email

Abstract

Aleksandar Jeremic

This paper summarizes a Variational Auto Encoder with 4 Convolutional layers and fully connected layers to encode and decode, the depth was originally much larger but many issues with exploding and vanishing gradients along with long training time cause a pivot to smaller depth. The VAE was trained on chest xray images of size 128x128 and 256x256.

1 Introduction

In this paper a 4 layer convolutional VAE with a linear layer was used along with a latent space of dimension 2048 to reconstruct chest xray images, the VAE uses LeakyReLU and Batchnorm to help with vanishing and exploding gradients which was a major concern in the beginning of this work when there were over 8 layers in encode and decode stages

2 Hyper Parameters

In this paper I used both 256x256 and 128x128 scaled images from the chest xray dataset, Stochastic Gradient Descent and Adam optimization (Adam was the best) was used with a learning rate of 0.001 a batch size of 32. The VAE was implemented with multiple different epochs and the best results were chosen, the number of epochs of training ranged from 10-250 for the 128x128 simplistic model. The loss function involved binary cross entropy loss along with KL Divergence with a weight of 0.001, I also changed the size of the latent dim and found around 200 to give the best results.

3 Training loss Curves

We can see a training loss curve stabilizing after about 10 epochs, in this run the training loss began at a very small value and this iteration had problems with exploding and vanishing gradients which then destroy the training loss.

The loss also often diverged due to various reasons such as vanishing and exploding gradients and hyperparameter tuning

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

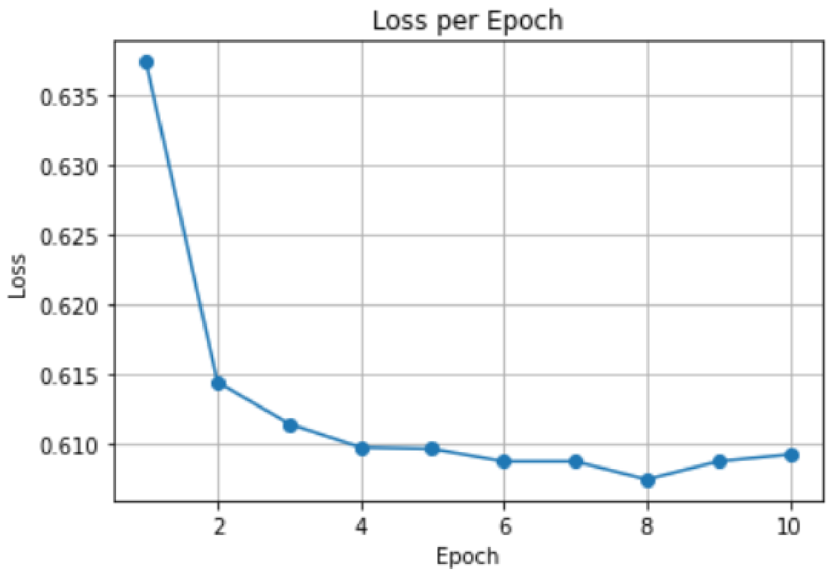

Figure 1: training loss

4 Reconstructed Images

Below are the images that were sent into the VAE, from there they are sent back to decode, these reconstructed images are fairly accurate however fail to recognize some of the ribs of the xrays, this is reasonable considering the most notable section of the xray is around the shoulders where there exists lots of differences in the way the shoulders are positioned. This is also quite a powerful reconstruction considering the small dataset which we trained on, only a few thousand images of size 256x256/128x128 and we could actually reconstruct some basic chest xray images!

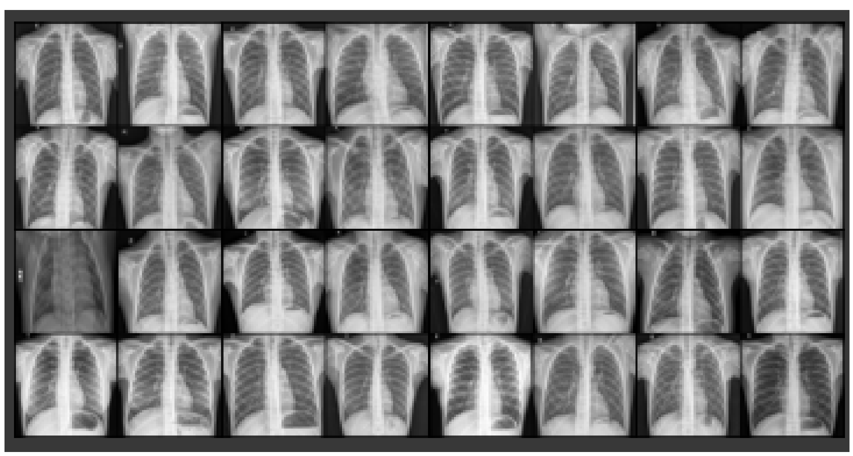

Figure 2: An example image.

here is the reconstructed images.

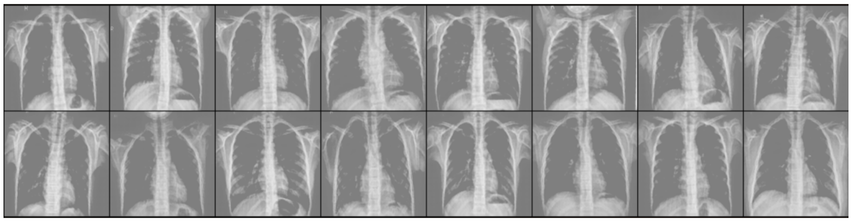

Figure 3: An example image.

5 FID score and IS Loss

FID and IS scoring is a form of accuracy for generative networks such as GANs and VAEs, FID measures the difference between real and generated images and compares how similar they are, IS scores are the diversity and quality of the generated images. I used scoring code directly from the website provided https: //pytorch-ignite.ai/blog/gan-evaluation-with-fid-and-is/

I had alot of trouble with working FID and IS scores, I would consistently get an error of having an imaginary component and would crash the runtime, this would cause me to lose images which made it very hard to gather consistent results when my training would often diverge. I believe this was in part due to the main exploding and vanishing gradients which could be solved using a larger dataset or a shallower algorithm.

6 Conclusion

The VAE is a Machine Learning Model which uses a latent space representation of input images to understand information about the image, from the latent space the trained model can recreate images similar to the ones that were inputted and trained by the model. Alternatively sampling from a distribution in the latent space is equivalent to picking a latent space representation of the training set and then reconstruction from that sample is generating a similar image to the training set.

7 Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torchvision.utils import save_image

import torch.nn.functional as F


from google.colab import drive
drive.mount('/content/drive')



#!unzip /content/drive/MyDrive/chest_xray.zip -d /content/drive/
    MyDrive/

import os


dataset_dir = '/content/drive/MyDrive/chest_xray/chest_xray'

data_dir = '/content/drive/MyDrive/chest_xray'

```

```
base_path = '/content/drive/MyDrive/chest_xray'

data_transforms = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
])

train_dir = os.path.join(dataset_dir, 'train')
test_dir = os.path.join(dataset_dir, 'test')
val_dir = os.path.join(dataset_dir, 'val')

USE_GPU = True
dtype = torch.float32

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

print_every = 100

print('using device:', device)

train_dataset = ImageFolder(train_dir, transform=data_transforms)
test_dataset = ImageFolder(test_dir, transform=data_transforms)
val_dataset = ImageFolder(val_dir, transform=data_transforms)

batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size,
    shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=
    False)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=
    False)

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.conv1_1 = nn.Conv2d(1, 64, (3, 3), padding=1, stride=2)
        self.conv1_2 = nn.Conv2d(64, 64, (3, 3), padding=1)

        self.conv2_1 = nn.Conv2d(64, 128, (3, 3), padding=1, stride=2)
        self.conv2_2 = nn.Conv2d(128, 128, (3, 3), padding=1)

        self.conv3_1 = nn.Conv2d(128, 256, (3, 3), padding=1, stride
            =2)
        self.conv3_2 = nn.Conv2d(256, 256, (3, 3), padding=1)
        self.conv3_3 = nn.Conv2d(256, 256, (3, 3), padding=1)

        self.conv4_1 = nn.Conv2d(256, 256, (3, 3), padding=1, stride
            =2)
        self.conv4_2 = nn.Conv2d(256, 256, (3, 3), padding=1)
        self.conv4_3 = nn.Conv2d(256, 256, (3, 3), padding=1)

        self.conv5_1 = nn.Conv2d(512, 512, (3, 3), padding=1, stride
            =2)
        self.conv5_2 = nn.Conv2d(512, 512, (3, 3), padding=1)
        self.conv5_3 = nn.Conv2d(512, 512, (3, 3), padding=1)
        self.conv5_4 = nn.Conv2d(512, 512, (3, 3), padding=1)

        self.bn1_1 = nn.BatchNorm2d(64)
        self.bn1_2 = nn.BatchNorm2d(64)
        self.bn2_1 = nn.BatchNorm2d(128)
```

```
        self.bn2_2 = nn.BatchNorm2d(128)

        self.bn3_1 = nn.BatchNorm2d(256)
        self.bn3_2 = nn.BatchNorm2d(256)
        self.bn3_3 = nn.BatchNorm2d(256)

        self.bn4_1 = nn.BatchNorm2d(256)
        self.bn4_2 = nn.BatchNorm2d(256)
        self.bn4_3 = nn.BatchNorm2d(256)

        self.bn5_1 = nn.BatchNorm2d(512)
        self.bn5_2 = nn.BatchNorm2d(512)
        self.bn5_3 = nn.BatchNorm2d(512)
        self.bn5_4 = nn.BatchNorm2d(512)

        self.relu = nn.LeakyReLU(0.01)
        self.sigmoid = nn.Sigmoid()

        self.fc1 = nn.Linear(256 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 256 * 8 * 8)

        self.fc3 = nn.Linear(256 * 8 * 8, 512)
        self.fc4 = nn.Linear(256 * 8 * 8, 512)

        self.deconv1_1 = nn.ConvTranspose2d(512, 512, (3, 3), padding
            =1)
        self.deconv1_2 = nn.ConvTranspose2d(512, 512, (3, 3), padding
            =1)
        self.deconv1_3 = nn.ConvTranspose2d(512, 512, (3, 3), padding
            =1)
        self.deconv1_4 = nn.ConvTranspose2d(512, 512, (3, 3), padding
            =1, stride=2, output_padding=1)

        self.deconv2_1 = nn.ConvTranspose2d(256, 256, (3, 3), padding
            =1)
        self.deconv2_2 = nn.ConvTranspose2d(256, 256, (3, 3), padding
            =1)
        self.deconv2_3 = nn.ConvTranspose2d(256, 256, (3, 3), padding
            =1, stride=2, output_padding=1)

        self.deconv3_1 = nn.ConvTranspose2d(256, 256, (3, 3), padding
            =1)
        self.deconv3_2 = nn.ConvTranspose2d(256, 256, (3, 3), padding
            =1)
        self.deconv3_3 = nn.ConvTranspose2d(256, 128, (3, 3), padding
            =1, stride=2, output_padding=1)

        self.deconv4_1 = nn.ConvTranspose2d(128, 128, (3, 3), padding
            =1)
        self.deconv4_2 = nn.ConvTranspose2d(128, 64, (3, 3), padding
            =1, stride=2, output_padding=1)

        self.deconv5_1 = nn.ConvTranspose2d(64, 64, (3, 3), padding=1)
        self.deconv5_2 = nn.ConvTranspose2d(64, 1, (3, 3), padding=1,
            stride=2, output_padding=1)

        self.bn1_1_deconv = nn.BatchNorm2d(512)
        self.bn1_2_deconv = nn.BatchNorm2d(512)
        self.bn1_3_deconv = nn.BatchNorm2d(512)
        self.bn1_4_deconv = nn.BatchNorm2d(512)

        self.bn2_1_deconv = nn.BatchNorm2d(256)
        self.bn2_2_deconv = nn.BatchNorm2d(256)
        self.bn2_3_deconv = nn.BatchNorm2d(256)

```

```
        self.bn3_1_deconv = nn.BatchNorm2d(256)
        self.bn3_2_deconv = nn.BatchNorm2d(256)
        self.bn3_3_deconv = nn.BatchNorm2d(128)

        self.bn4_1_deconv = nn.BatchNorm2d(128)
        self.bn4_2_deconv = nn.BatchNorm2d(64)

        self.bn5_1_deconv = nn.BatchNorm2d(64)
        self.bn5_2_deconv = nn.BatchNorm2d(1)

    def encode(self, x):
        x = self.conv1_1(x)
        x = self.bn1_1(x)
        x = self.relu(x)

        x = self.conv2_1(x)
        x = self.bn2_1(x)
        x = self.relu(x)

        x = self.conv3_1(x)
        x = self.bn3_1(x)
        x = self.relu(x)

        x = self.conv4_1(x)
        x = self.bn4_1(x)
        x = self.relu(x)

        x = x.view(x.size(0), -1)
        mean = self.fc3(x)
        log_var = self.fc4(x)
        return mean, log_var

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        x = self.fc2(z)
        x = x.view(x.size(0), 256, 8, 8)

        x = self.deconv2_3(x)
        x = self.bn2_3_deconv(x)
        x = self.relu(x)

        x = self.deconv3_3(x)
        x = self.bn3_3_deconv(x)
        x = self.relu(x)

        x = self.deconv4_2(x)
        x = self.bn4_2_deconv(x)
        x = self.relu(x)

        x = self.deconv5_2(x)
        x = self.bn5_2_deconv(x)
        x = self.relu(x)

        x = self.sigmoid(x)
        return x

    def forward(self, x):
        mu, log_var = self.encode(x)
        z = self.reparameterize(mu, log_var)
        x_reconstructed = self.decode(z)
        return x_reconstructed, mu, log_var
```

```
model = VAE().to(device)

optimizer = optim.Adam(model.parameters(), lr=0.001)

kld_rate = .001



num_epochs = 20

for epoch in range(num_epochs):
    running_loss = 0.0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)

        x, mu, log_var = model(data)

        loss = F.binary_cross_entropy(x, data) -0.5 *kld_rate * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        if batch_idx % 100 == 99:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                  .format(epoch + 1, num_epochs, batch_idx + 1, len(train_loader), running_loss / 100))
            running_loss = 0.0


torch.save(model.state_dict(), "vae_model.pth")

torch.save(model.state_dict(), os.path.join(os.getcwd(), "model2.pth"))
model.load_state_dict(torch.load(os.path.join(os.getcwd(), "model2.pth")))

with torch.no_grad():
    model.eval()
    for i, (data, _) in enumerate(test_loader):
        data = data.to(device)
        reconstructed ,_ ,_ = model(data)
        if i == 0:
            save_image(data, os.path.join(data_dir, "original_images.png"))
            save_image(reconstructed, os.path.join(data_dir, "reconstructed_images.png"))
        break
```