MedVAE: Generating Chest X-Ray Images using Variational Autoencoders (VAE)

Saksham Jindal
UC San Diego
sjindal@ucsd.edu

1 Introduction

Chest X-rays are one of the most common diagnostic tools used in medicine. They are used to detect a wide range of conditions, including pneumonia, heart disease, and cancer. However, chest X-rays can be difficult to interpret, even for experienced radiologists. This happens because images can be noisy and contain a lot of irrelevant information.

Variational autoencoders (VAEs) [2] are a type of deep learning model that can be used to generate images from latent representations. The primary motivation behind utilizing a VAE for image generation is its ability to learn a meaningful and continuous latent space representation of the input data. VAEs are trained on a dataset of images, and they learn to map the images to a latent space which captures essential features and variations present in the chest X-ray images, enabling us to generate new images by sampling from this space. Once the VAE is trained, it can be used to generate new images by sampling from the latent space. By training the VAE using an unsupervised approach, we can effectively model the underlying distribution of the data and generate novel images that exhibit similar characteristics.

In this paper, we propose to use VAEs to generate chest X-ray images. Our dataset comprises the medical dataset, obtained from [1], which consists of a chest x-ray images of patients diagnosed with pneumonia. We will train a VAE on a dataset of chest X-rays and use the trained VAE to generate new images. We will then evaluate the quality of the generated images and compare them to real chest X-rays.

2 Method

2.1 Variational Autoencoder

Variational autoencoders (VAEs) is a type of deep generative model that can be used to learn latent representations of data in a probablisitic manner. Compared to the autoencoder networks which learn a fixed or deterministic representation value, VAEs generate a probability distribution over the latent representations. The VAE architecture consists of two neural networks: an encoder and a decoder.

Encoder:The encoder is typically a deep neural network that is trained to map the observed data to a latent space. The encoder neural network $q_\phi(z|x)$ has weights ϕ and takes an input x and generates the parameters (mean μ and variance σ^2) of an approximate posterior distribution over the latent encoding $z : q_\phi(z|x) = \mathcal{N}(z; \mu, \sigma^2)$ where $\mathcal{N}$ represents a multivariate Gaussian distribution.

Decoder: The decoder is also typically a deep neural network that is trained to map the latent space back to the observed data. The decoder network $p_\theta(x|z)$ with weights θ then takes a sample z from the encoding distribution and reconstructs the original input x.

The objective of a VAE is to minimise the mean squared error (MSE) between the original data $\mathbf{x}$ and $\hat{\mathbf{x}}$, while also minimizing the KL divergence between the encoding distribution $q_\phi(z|x)$ and a standard normal prior distribution $p(z) = \mathcal{N}(0, 1)$. This objective can be written as:

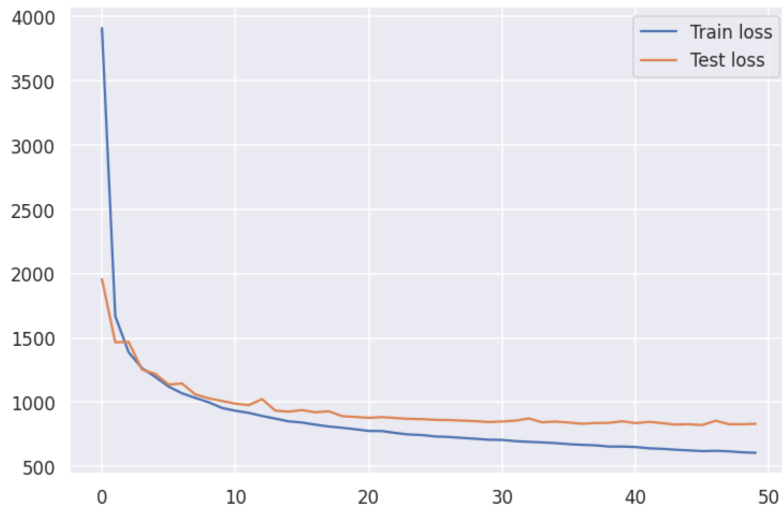


Figure 1: Total loss on the train and test set

$$\mathcal{L} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) \tag{1}$$

The first term is the reconstruction loss, which measures the dissimilarity between the original input data $\mathbf{x}$ and the reconstructed output $\hat{\mathbf{x}}$ generated by the decoder network. The second term is the KL divergence loss which encourages approximate posterior distribution to match a chosen prior distribution (standard Gaussian distribution here).

2.2 Experiments

Data Preprocessing: We obtained chest x-ray images of varying sizes and aspect ratios from the dataset which were uniformly resized to a size of (256, 256). The given dataset had 5216, 624 and 16 images in the training, test and validation set respectively. We use the test set for validation of our generative network and report the results on the test and validation set.

Network Archtecture: We use series of convolution layers for filter extraction (without batch normalization) in the encoder and transpose convolution for upsampling in the decoder. More details on the architecture can be found in the code attached.

Training: We used a batch size of 32 with the Adam optimizer with a learning rate of 0.001. We tested learning rates on a log scale from 0.1, 0.01, 0.001 and 0.0001. 0.001 worked our best for the given batch size. We trained the model for 50 epochs and stopped the training citing convergence of loss curve on the test set.

Metrics: To evaluate the quality and diversity of generated images in our model, we employed two popular metrics: Inception Score (IS) and Fréchet Inception Distance (FID). These metrics provide quantitative measures that help assess the performance of generative models. Fréchet Inception Distance (FID) is a metric that is used to assess the similarity between the distribution of generated images and the distribution of real images. Inception Score (IS) is a metric that is used to assess the quality and diversity of generated images from a generative model.

3 Results

We present the results of our experiments using the Variational Autoencoder (VAE) to generate chest X-ray images. We evaluate the quality and diversity of the generated images and compare them to real chest X-rays using quantitative metrics. Additionally, we provide visual examples of the generated images for qualitative assessment.

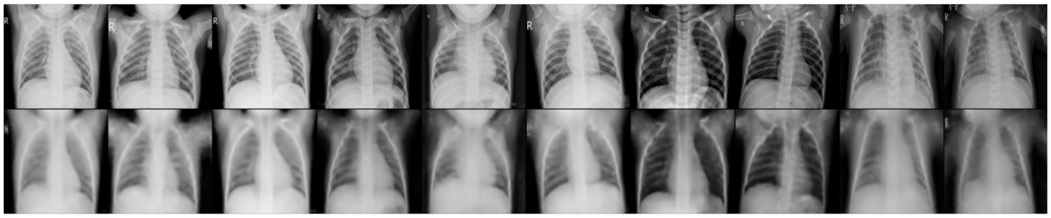


Figure 2: Reconstruction on the test set. Above: input images. Below: reconstructed images

3.1 Quantitative Evaluation

As discussed in previous section, we utilized two popular metrics: Inception Score (IS) and Fréchet Inception Distance (FID). These metrics provide objective measures for evaluating the performance of generative models. We computed both IS and FID scores on our generated images using a test set and a validation set. We obtained a FID score of 0.160502 and Inception Score of 1.9068 on the test set. On the validation set, we obtained a FID score of 0.1168 and Inception score of 1.60268. The FID score of 0.160502 on the test set 0.1168 on the validation set indicates a reasonably close similarity between the distribution of the generated images and the distribution of real chest X-ray images. The Inception Score of 1.9068 on the test set is relatively better than 1.6028 on the validation set, though both are moderate scores, indicates that the model is still able to generate images that are visually similar to real images and have diverse patterns.

3.2 Qualitative Assessment

To provide a visual assessment of the generated images, we present some examples in Figure 1. The top row shows the input chest X-ray images from the test set, while the bottom row displays the corresponding reconstructed images generated by the VAE. As observed in Figure 1, the VAE successfully reconstructs the structure and composition of the input chest X-ray images, capturing the essential features. This indicates that the generative model was effective in in preserving the composition of the image. However, there is a noticeable loss of finer details in the reconstructed images compared to the original input images.

4 Discussion

In this paper, we proposed a method for generating chest X-ray images using variational autoencoders (VAEs). We trained a VAE on a dataset of chest X-rays, and we used the trained VAE to generate new images. We evaluated the quality of the generated images using quantitative metrics, and we also provided visual examples of the generated images.

The results of our experiments show that the VAE is able to generate chest X-ray images that are visually similar to real chest X-rays. The VAE also achieves good scores on quantitative metrics, such as the Inception Score and the Fréchet Inception Distance.

However, there are still some limitations to the VAE. For example, the VAE can sometimes generate images that are blurry or noisy. A possible way of avoid blurry or noisy reconstruction is to use positional encoding with fourier features [3] in future work.

References

[1] Daniel Kermany, Michael Goldbaum, Wenjia Cai, Carolina Valentim, Hui-Ying Liang, Sally Baxter, Alex McKeown, Ge Yang, Xiaokang Wu, Fangbing Yan, Justin Dong, Made Prasadha, Jacqueline Pei, Magdalena Ting, Jie Zhu, Christina Li, Sierra Hewett, Jason Dong, Ian Ziyar, and Kang Zhang. Identifying medical diagnoses and treatable diseases by image-based deep learning, 02 2018.

[2] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.

[3] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains, 2020.

In [64]:
```
import math
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision.utils import make_grid

device = torch.device('cuda:2') if torch.cuda.is_available() else torch.device('cpu')
```

In [65]:
```
BATCH_SIZE = 32
```

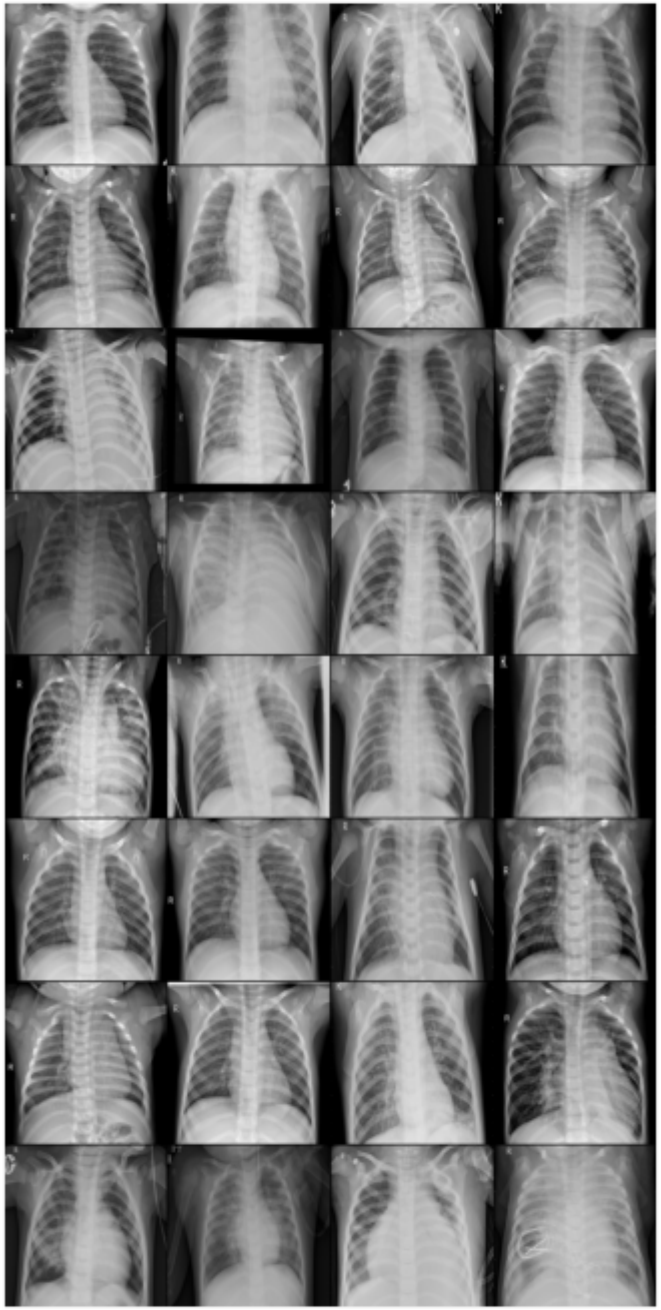
In [66]:
```
class_names = ['PNEUMONIA', 'NORMAL']

data_dir = './chest_xray'
TEST = 'test'
TRAIN = 'train'
VAL ='val'

# define transforms
transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize((256,256)),
    torchvision.transforms.ToTensor(),
])

# datasets
trainset = torchvision.datasets.ImageFolder(os.path.join(data_dir, TRAIN),transform = transform)
testset = torchvision.datasets.ImageFolder(os.path.join(data_dir, TEST),transform = transform)
validset = torchvision.datasets.ImageFolder(os.path.join(data_dir, VAL),transform = transform)
```

In [67]:
```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE, shuffle=False)
valloader = torch.utils.data.DataLoader(validset, batch_size=BATCH_SIZE, shuffle=False)

# check the dataset
print('Trainset size:', len(trainset))
```

```
print('Testset size:', len(testset))
print('Validset size:', len(validset))
```

```
Trainset size: 5216
Testset size: 624
Validset size: 16
```

In [68]:
```
# visualise random 4 images with the labels
def show_batch(images, labels):
    fig, ax = plt.subplots(figsize=(12, 12))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(images, nrow=4).permute(1, 2, 0))
    plt.show()


# show some images
images, labels = next(iter(trainloader))
show_batch(images, labels)
```



In [71]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

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

        # Encoder layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=4, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
        self.fc1 = nn.Linear(256 * 16 * 16, 512)
        self.fc2 = nn.Linear(256 * 16 * 16, 512)

        # Decoder layers
        self.fc3 = nn.Linear(512, 256 * 16 * 16)
        self.deconv1 = nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1)
        self.deconv2 = nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1)
        self.deconv3 = nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1)
        self.deconv4 = nn.ConvTranspose2d(32, 3, kernel_size=4, stride=2, padding=1)

    def encode(self, x):
        h1 = F.relu((self.conv1(x)))
        h2 = F.relu((self.conv2(h1)))
        h3 = F.relu((self.conv3(h2)))
        h4 = F.relu((self.conv4(h3)))
        h4 = h4.view(-1, 256 * 16 * 16)
        return self.fc1(h4), self.fc2(h4)

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

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        h3 = h3.view(-1, 256, 16, 16)
```

```
        h4 = F.relu((self.deconv1(h3)))
        h5 = F.relu((self.deconv2(h4)))
        h6 = F.relu((self.deconv3(h5)))
        x_recon = torch.sigmoid((self.deconv4(h6)))
        return x_recon

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar

def vae_loss(x_recon, x, mu, logvar):
    mse_loss = nn.MSELoss(reduction='sum')(x_recon, x)
    kld_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return mse_loss + kld_loss
```

In [72]:
```
from tqdm import tqdm

def train(model, trainloader, optimizer, epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in (enumerate(trainloader)):
        data = data.to(device)
        optimizer.zero_grad()
        x_recon, mu, logvar = model(data)
        loss = vae_loss(x_recon, data, mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{}]\tLoss: {:.3f}'.format(
                epoch, batch_idx * len(data), len(trainloader.dataset), loss.item() / len(data)))
    print('====> Epoch: {} Average loss: {:.4f}'.format(
          epoch, train_loss / len(trainloader.dataset)))
    return train_loss

def test(model, testloader):
    model.eval()
    test_loss = 0
```

```
    with torch.no_grad():
        for data, _ in (testloader):
            data = data.to(device)
            x_recon, mu, logvar = model(data)
            test_loss += vae_loss(x_recon, data, mu, logvar).item()

    test_loss /= len(testloader.dataset)
    print('====> Test set loss: {:.4f}'.format(test_loss))
    return test_loss

# train the model
model = VAE().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)


EPOCHS = 20
print_every_epoch = 1
train_loss = []
test_loss = []

for epoch in range(1, EPOCHS + 1):
    train_loss.append(train(model, trainloader, optimizer, epoch))
    if epoch % print_every_epoch == 0:
        test_loss.append(test(model, testloader))
```

```
Train Epoch: 1 [0/5216] Loss: 11289.796
Train Epoch: 1 [3200/5216]      Loss: 2116.344
====> Epoch: 1 Average loss: 3906.8194
====> Test set loss: 1955.6297
Train Epoch: 2 [0/5216] Loss: 1788.370
Train Epoch: 2 [3200/5216]      Loss: 1745.300
====> Epoch: 2 Average loss: 1667.9467
====> Test set loss: 1464.8053
Train Epoch: 3 [0/5216] Loss: 1374.875
Train Epoch: 3 [3200/5216]      Loss: 1381.791
====> Epoch: 3 Average loss: 1388.4206
====> Test set loss: 1468.4538
Train Epoch: 4 [0/5216] Loss: 1476.313
Train Epoch: 4 [3200/5216]      Loss: 1178.011
====> Epoch: 4 Average loss: 1264.7234
====> Test set loss: 1253.7550
Train Epoch: 5 [0/5216] Loss: 1202.415
Train Epoch: 5 [3200/5216]      Loss: 1106.659
====> Epoch: 5 Average loss: 1196.1227
====> Test set loss: 1220.0580
Train Epoch: 6 [0/5216] Loss: 1070.840
Train Epoch: 6 [3200/5216]      Loss: 1163.831
====> Epoch: 6 Average loss: 1121.0916
====> Test set loss: 1137.2060
Train Epoch: 7 [0/5216] Loss: 1129.333
Train Epoch: 7 [3200/5216]      Loss: 994.929
====> Epoch: 7 Average loss: 1068.9983
====> Test set loss: 1144.7448
Train Epoch: 8 [0/5216] Loss: 1106.987
Train Epoch: 8 [3200/5216]      Loss: 985.492
====> Epoch: 8 Average loss: 1033.0517
====> Test set loss: 1060.0231
Train Epoch: 9 [0/5216] Loss: 938.920
Train Epoch: 9 [3200/5216]      Loss: 994.811
====> Epoch: 9 Average loss: 999.5244
====> Test set loss: 1029.8588
Train Epoch: 10 [0/5216]        Loss: 924.823
Train Epoch: 10 [3200/5216]     Loss: 873.398
====> Epoch: 10 Average loss: 955.0527
====> Test set loss: 1009.1665
```

```
Train Epoch: 11 [0/5216]        Loss: 933.762
Train Epoch: 11 [3200/5216]     Loss: 964.687
====> Epoch: 11 Average loss: 933.3594
====> Test set loss: 988.3110
Train Epoch: 12 [0/5216]        Loss: 905.859
Train Epoch: 12 [3200/5216]     Loss: 1004.626
====> Epoch: 12 Average loss: 916.8143
====> Test set loss: 976.8629
Train Epoch: 13 [0/5216]        Loss: 1048.693
Train Epoch: 13 [3200/5216]     Loss: 919.313
====> Epoch: 13 Average loss: 892.7239
====> Test set loss: 1023.2171
Train Epoch: 14 [0/5216]        Loss: 914.347
Train Epoch: 14 [3200/5216]     Loss: 894.951
====> Epoch: 14 Average loss: 872.1892
====> Test set loss: 934.4654
Train Epoch: 15 [0/5216]        Loss: 901.026
Train Epoch: 15 [3200/5216]     Loss: 846.736
====> Epoch: 15 Average loss: 850.9668
====> Test set loss: 925.9690
Train Epoch: 16 [0/5216]        Loss: 744.669
Train Epoch: 16 [3200/5216]     Loss: 853.390
====> Epoch: 16 Average loss: 841.8159
====> Test set loss: 938.2212
Train Epoch: 17 [0/5216]        Loss: 830.741
Train Epoch: 17 [3200/5216]     Loss: 862.305
====> Epoch: 17 Average loss: 825.0516
====> Test set loss: 921.1635
Train Epoch: 18 [0/5216]        Loss: 924.312
Train Epoch: 18 [3200/5216]     Loss: 786.032
====> Epoch: 18 Average loss: 810.8661
====> Test set loss: 929.0139
Train Epoch: 19 [0/5216]        Loss: 847.451
Train Epoch: 19 [3200/5216]     Loss: 757.061
====> Epoch: 19 Average loss: 800.9129
====> Test set loss: 891.4531
Train Epoch: 20 [0/5216]        Loss: 818.563
Train Epoch: 20 [3200/5216]     Loss: 792.382
====> Epoch: 20 Average loss: 789.3923
====> Test set loss: 884.9710
```

In [76]:
```
for epoch in range(21, 30 + 1):
    train_loss.append(train(model, trainloader, optimizer, epoch))
    if epoch % print_every_epoch == 0:
        test_loss.append(test(model, testloader))
```

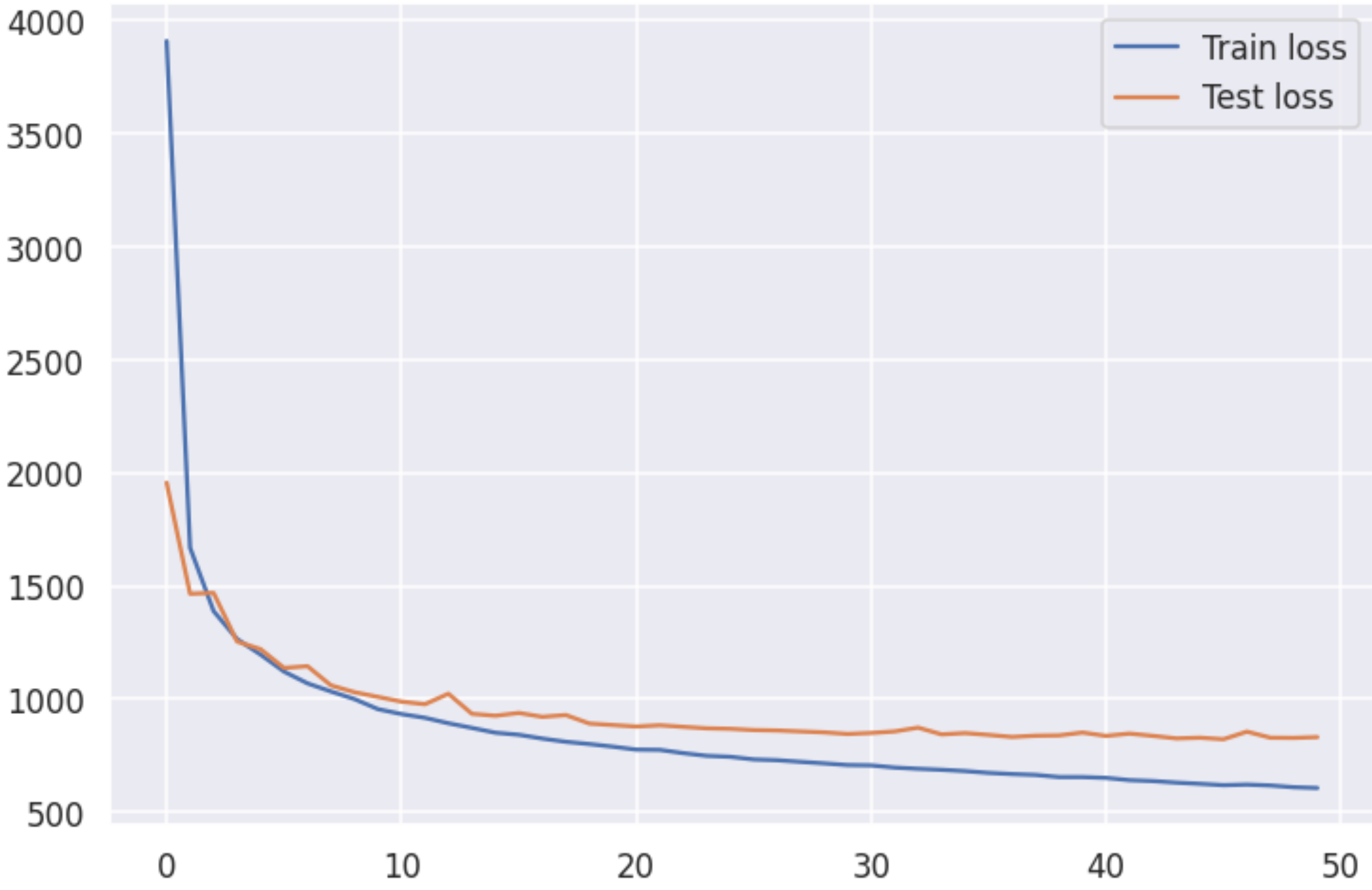
```
Train Epoch: 21 [0/5216]        Loss: 784.870
Train Epoch: 21 [3200/5216]     Loss: 798.450
====> Epoch: 21 Average loss: 776.3277
====> Test set loss: 878.2844
Train Epoch: 22 [0/5216]        Loss: 704.334
Train Epoch: 22 [3200/5216]     Loss: 781.415
====> Epoch: 22 Average loss: 775.1710
====> Test set loss: 883.9722
Train Epoch: 23 [0/5216]        Loss: 748.816
Train Epoch: 23 [3200/5216]     Loss: 729.082
====> Epoch: 23 Average loss: 760.7296
====> Test set loss: 876.8732
Train Epoch: 24 [0/5216]        Loss: 689.028
Train Epoch: 24 [3200/5216]     Loss: 761.563
====> Epoch: 24 Average loss: 748.8828
====> Test set loss: 870.3555
Train Epoch: 25 [0/5216]        Loss: 756.213
Train Epoch: 25 [3200/5216]     Loss: 719.312
====> Epoch: 25 Average loss: 744.6167
====> Test set loss: 868.0855
Train Epoch: 26 [0/5216]        Loss: 705.564
Train Epoch: 26 [3200/5216]     Loss: 710.530
====> Epoch: 26 Average loss: 732.9642
====> Test set loss: 862.6265
Train Epoch: 27 [0/5216]        Loss: 762.068
Train Epoch: 27 [3200/5216]     Loss: 713.795
====> Epoch: 27 Average loss: 729.5735
====> Test set loss: 860.8354
Train Epoch: 28 [0/5216]        Loss: 690.287
Train Epoch: 28 [3200/5216]     Loss: 672.328
====> Epoch: 28 Average loss: 722.2965
====> Test set loss: 856.5312
Train Epoch: 29 [0/5216]        Loss: 688.800
Train Epoch: 29 [3200/5216]     Loss: 747.492
====> Epoch: 29 Average loss: 715.5255
====> Test set loss: 852.4863
Train Epoch: 30 [0/5216]        Loss: 666.164
Train Epoch: 30 [3200/5216]     Loss: 713.815
====> Epoch: 30 Average loss: 708.1032
====> Test set loss: 845.4841
```

In [81]:
```
for epoch in range(31, 50 + 1):
    train_loss.append(train(model, trainloader, optimizer, epoch))
    if epoch % print_every_epoch == 0:
        test_loss.append(test(model, testloader))
```

```
Train Epoch: 31 [0/5216]        Loss: 716.197
Train Epoch: 31 [3200/5216]     Loss: 719.987
====> Epoch: 31 Average loss: 706.6529
====> Test set loss: 849.8202
Train Epoch: 32 [0/5216]        Loss: 682.095
Train Epoch: 32 [3200/5216]     Loss: 657.601
====> Epoch: 32 Average loss: 697.0394
====> Test set loss: 856.6563
Train Epoch: 33 [0/5216]        Loss: 678.404
Train Epoch: 33 [3200/5216]     Loss: 696.818
====> Epoch: 33 Average loss: 691.5706
====> Test set loss: 873.4023
Train Epoch: 34 [0/5216]        Loss: 696.069
Train Epoch: 34 [3200/5216]     Loss: 713.138
====> Epoch: 34 Average loss: 687.2731
====> Test set loss: 843.6204
Train Epoch: 35 [0/5216]        Loss: 708.348
Train Epoch: 35 [3200/5216]     Loss: 622.536
====> Epoch: 35 Average loss: 681.4753
====> Test set loss: 849.2965
Train Epoch: 36 [0/5216]        Loss: 625.251
Train Epoch: 36 [3200/5216]     Loss: 653.756
====> Epoch: 36 Average loss: 673.5354
====> Test set loss: 841.3298
Train Epoch: 37 [0/5216]        Loss: 642.590
Train Epoch: 37 [3200/5216]     Loss: 659.590
====> Epoch: 37 Average loss: 668.2521
====> Test set loss: 832.2691
Train Epoch: 38 [0/5216]        Loss: 675.517
Train Epoch: 38 [3200/5216]     Loss: 667.443
====> Epoch: 38 Average loss: 664.5860
====> Test set loss: 837.8425
Train Epoch: 39 [0/5216]        Loss: 595.445
Train Epoch: 39 [3200/5216]     Loss: 650.212
====> Epoch: 39 Average loss: 655.1117
====> Test set loss: 839.0833
Train Epoch: 40 [0/5216]        Loss: 633.389
Train Epoch: 40 [3200/5216]     Loss: 626.263
====> Epoch: 40 Average loss: 654.9597
====> Test set loss: 851.6873
```

```
Train Epoch: 41 [0/5216]        Loss: 654.033
Train Epoch: 41 [3200/5216]     Loss: 659.851
====> Epoch: 41 Average loss: 651.4501
====> Test set loss: 836.8791
Train Epoch: 42 [0/5216]        Loss: 646.999
Train Epoch: 42 [3200/5216]     Loss: 662.930
====> Epoch: 42 Average loss: 641.4194
====> Test set loss: 847.0670
Train Epoch: 43 [0/5216]        Loss: 636.798
Train Epoch: 43 [3200/5216]     Loss: 618.143
====> Epoch: 43 Average loss: 637.4127
====> Test set loss: 837.0255
Train Epoch: 44 [0/5216]        Loss: 633.413
Train Epoch: 44 [3200/5216]     Loss: 670.472
====> Epoch: 44 Average loss: 630.4280
====> Test set loss: 825.7518
Train Epoch: 45 [0/5216]        Loss: 604.555
Train Epoch: 45 [3200/5216]     Loss: 592.863
====> Epoch: 45 Average loss: 625.2166
====> Test set loss: 828.8593
Train Epoch: 46 [0/5216]        Loss: 607.246
Train Epoch: 46 [3200/5216]     Loss: 605.299
====> Epoch: 46 Average loss: 619.3499
====> Test set loss: 822.1335
Train Epoch: 47 [0/5216]        Loss: 644.067
Train Epoch: 47 [3200/5216]     Loss: 598.841
====> Epoch: 47 Average loss: 621.8162
====> Test set loss: 855.6692
Train Epoch: 48 [0/5216]        Loss: 613.484
Train Epoch: 48 [3200/5216]     Loss: 613.787
====> Epoch: 48 Average loss: 617.8894
====> Test set loss: 828.9340
Train Epoch: 49 [0/5216]        Loss: 622.985
Train Epoch: 49 [3200/5216]     Loss: 594.099
====> Epoch: 49 Average loss: 610.0464
====> Test set loss: 828.1646
Train Epoch: 50 [0/5216]        Loss: 594.247
Train Epoch: 50 [3200/5216]     Loss: 611.555
====> Epoch: 50 Average loss: 606.7001
====> Test set loss: 831.4890
```

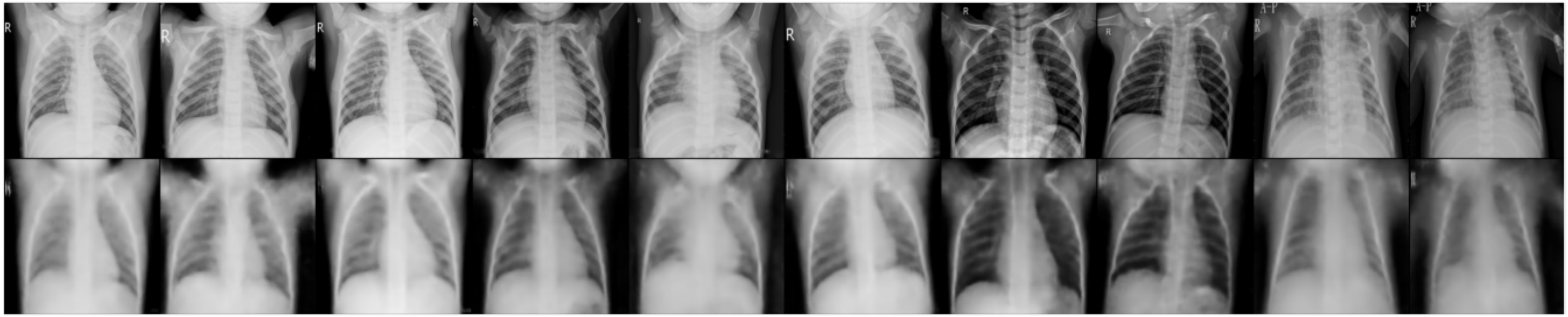
In [82]:
```
# plot train and test loss

import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
sns.set_style('darkgrid')
sns.set(rc={'figure.figsize':(12, 8)})
sns.set_context('talk')

plt.plot(np.array(train_loss)/len(trainloader.dataset), label='Train loss')
plt.plot(np.arange(0, 50, print_every_epoch), test_loss, label='Test loss')
plt.legend()
plt.show()
```



In [83]:

```
# visualize reconstructions
import matplotlib.pyplot as plt
import numpy as np

def show_image(x):
    fig, ax = plt.subplots(figsize=(24, 24))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(np.transpose(x.detach().cpu().numpy(), (1, 2, 0)))
    plt.show()

def show_reconstruction(model, testloader, n=10):
```

```
    model.eval()
    images, _ = next(iter(testloader))
    images = images[:n]
    with torch.no_grad():
        x_recon, _, _ = model(images.to(device))
    x_concat = torch.cat([images.cpu(), x_recon.cpu()])
    show_image(make_grid(x_concat, nrow=n))

show_reconstruction(model, valloader, n = 10)
```



In [84]:

```
from ignite.metrics import FID, InceptionScore
from ignite.engine import Engine
import PIL.Image as Image

def interpolate(batch):
    import torchvision.transforms as transforms
    arr = []

    for img in batch:
        pil_img = transforms.ToPILImage()(img)
        resized_img = pil_img.resize((299,299), Image.BILINEAR)
        arr.append(transforms.ToTensor()(resized_img))
    return torch.stack(arr)

def evaluation_step(engine, batch):
    model.eval()
    x, _ = batch
    with torch.no_grad():
```

```
        x_recon, _, _ = model(x.to(device))
        fake = interpolate(x_recon)
        real = interpolate(x)
    return fake, real


fid_metric = FID(device=device)
is_metric = InceptionScore(device=device, output_transform=lambda x: x[0])
evaluator = Engine(evaluation_step)
fid_metric.attach(evaluator, "fid")

# run the evaluator on your test data loader
is_metric.attach(evaluator, "is")
evaluator.run(testloader, max_epochs=1) # use your test data loader, NOT training data loader
metrics = evaluator.state.metrics
fid_score = metrics['fid']
is_score = metrics['is']
print("====> For the test data loader:")
print("FID score: {}".format(fid_score))
print("Inception score: {}".format(is_score))

# run the evaluator on the val data loader
is_metric.attach(evaluator, "is")
evaluator.run(valloader, max_epochs=1) # use your test data loader, NOT training data loader
metrics = evaluator.state.metrics
fid_score = metrics['fid']
is_score = metrics['is']
print("====> For the val data loader:")
print("FID score: {}".format(fid_score))
print("Inception score: {}".format(is_score))
```

```
====> For the test data loader:
FID score: 0.1605020390753168
Inception score: 1.9068931714571449
====> For the val data loader:
FID score: 0.11687438211117764
Inception score: 1.60268167316294
```

In [ ]:

```
```

In [ ]: