Published in DataSeries

Open in app ↗ Sign up Sign In

Search Medium

Eugenia Anello Follow

Jul 15, 2021 · 5 min read · ✦ · Listen

Save

Variational Autoencoder with Pytorch

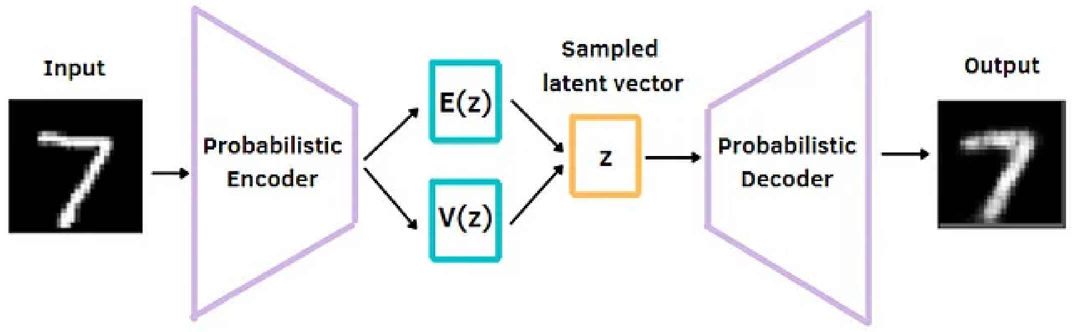


Illustration by Author

The post is the ninth *in a series of guides to building deep learning models with Pytorch. Below, there is the full series:*

1. *Pytorch Tutorial for Beginners*
2. *Manipulating Pytorch Datasets*

3. *Understand Tensor Dimensions in DL models*
4. *CNN & Feature visualizations*
5. *Hyperparameter tuning with Optuna*
6. *K Fold Cross Validation*
7. *Convolutional Autoencoder*
8. *Denoising Autoencoder*
9. *Variational Autoencoder (this post)*

134 | 5

The goal of the series is to make Pytorch more intuitive and accessible as possible through examples of implementations. There are many tutorials on the Internet to use Pytorch to build many types of challenging models, but it can also be confusing at the same time because there are always slight differences when you pass from one tutorial to another. In this series, I want to start from the simplest topics to the more advanced ones.

Variational autoencoder

The standard autoencoder can have an issue, constituted by the fact that the latent space can be irregular [1]. This means that close points in the latent space can produce different and meaningless patterns over visible units.

One solution to this issue is the introduction of the Variational Autoencoder. As the autoencoder, it is composed of two neural network architectures, encoder, and decoder.

But there is a modification of the encoding-decoding process. I will explain all the steps:

- We encode the input as a **distribution over the latent space**, instead of considering it as a single point. This encoded distribution is chosen to be normal so that the encoder can be trained to return the mean matrix and the covariance matrix.

- In the second step, we **sample a point from that encoded distribution.**
- After, we can **decode the sampled point** and calculate the reconstruction error
- We **backpropagate the reconstruction error through the network.** Since the sampling procedure is a discrete process, so it's not continuous, we need to apply a **reparameterisation trick** to make the backpropagation work:

$$z = E(z) + \epsilon \times \sqrt{V(z)} \text{ where } \epsilon \sim N(0, I_d)$$

VAE Loss Function

The loss for the VAE consists of two terms:

- the first term is the **reconstruction term**, which is obtained by comparing the input and its corresponding reconstruction.
- An additional term is the **regularization term**, which is also called **Kullback-Leibler divergence** between the distribution returned by the encoder and the standard normal distribution [3]. This term acts as a regularizer in the latent space, making the distributions returned by the encoder close to a standard normal distribution.

$$l(x, \hat{x}) = l_{reconstruction} + KL(z, N(0, I_d))$$

Implementation with Pytorch

As in the previous tutorials, the Variational Autoencoder is implemented and trained on the MNIST dataset.

Let's begin by importing the libraries and the datasets.

```python
import matplotlib.pyplot as plt # plotting library
import numpy as np # this module is useful to work with numerical arrays
import pandas as pd
import random
import torch
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader,random_split
from torch import nn
import torch.nn.functional as F
import torch.optim as optim

data_dir = 'dataset'

train_dataset = torchvision.datasets.MNIST(data_dir, train=True, download=True)
test_dataset  = torchvision.datasets.MNIST(data_dir, train=False, download=True)

train_transform = transforms.Compose([
transforms.ToTensor(),
])

test_transform = transforms.Compose([
transforms.ToTensor(),
])

train_dataset.transform = train_transform
test_dataset.transform = test_transform

m=len(train_dataset)

train_data, val_data = random_split(train_dataset, [int(m-m*0.2), int(m*0.2)])
batch_size=256

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size)
valid_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,shuffle=True)
```

.py hosted with ❤ by GitHub view raw

Now, we define a VariationalAutoencoder class, which combines the Encoder and Decoder classes [3].

The encoder and decoder networks contain three convolutional layers and two fully connected layers. Some batch normal layers are added to have more robust features in the latent space. Differently from the standard autoencoder, the encoder returns mean and variance matrices and we use them to obtain the sampled latent vector. In the VariationalEncoder class, we obtain the Kullback-Leibler term.

```python
class VariationalEncoder(nn.Module):
    def __init__(self, latent_dims):
        super(VariationalEncoder, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(8, 16, 3, stride=2, padding=1)
        self.batch2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(16, 32, 3, stride=2, padding=0)
        self.linear1 = nn.Linear(3*3*32, 128)
        self.linear2 = nn.Linear(128, latent_dims)
        self.linear3 = nn.Linear(128, latent_dims)

        self.N = torch.distributions.Normal(0, 1)
        self.N.loc = self.N.loc.cuda() # hack to get sampling on the GPU
        self.N.scale = self.N.scale.cuda()
        self.kl = 0

    def forward(self, x):
        x = x.to(device)
        x = F.relu(self.conv1(x))
        x = F.relu(self.batch2(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.linear1(x))
        mu =  self.linear2(x)
        sigma = torch.exp(self.linear3(x))
        z = mu + sigma*self.N.sample(mu.shape)
        self.kl = (sigma**2 + mu**2 - torch.log(sigma) - 1/2).sum()
        return z
```

.py hosted with ❤ by GitHub view raw

After we define the Decoder class, which remains the same as the one I showed in the fifth tutorial of the series.

Below, we define the class that merges the encoder and decoder:

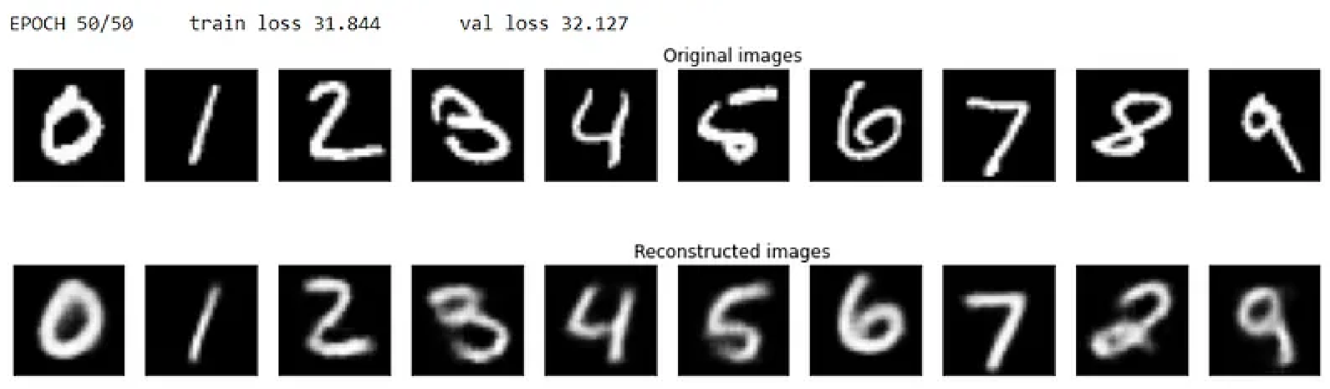
Finally, we can initialize the VariationalAutoencoder class, the optimizer, and the device to use the GPU in the code.

We define the functions to train and evaluate the Variational Autoencoder:

The loss is composed of two terms, as I described in the theory above. The reconstruction term is the sum of the squared differences between the input and its reconstruction. Some other versions use the BCE loss instead of MSE loss, but I preferred it in this way because it makes more sense.

I am also interested to see the input and its corresponding reconstruction in each epoch during the training of the VAE model. For this reason, I define a function to have these visualizations:

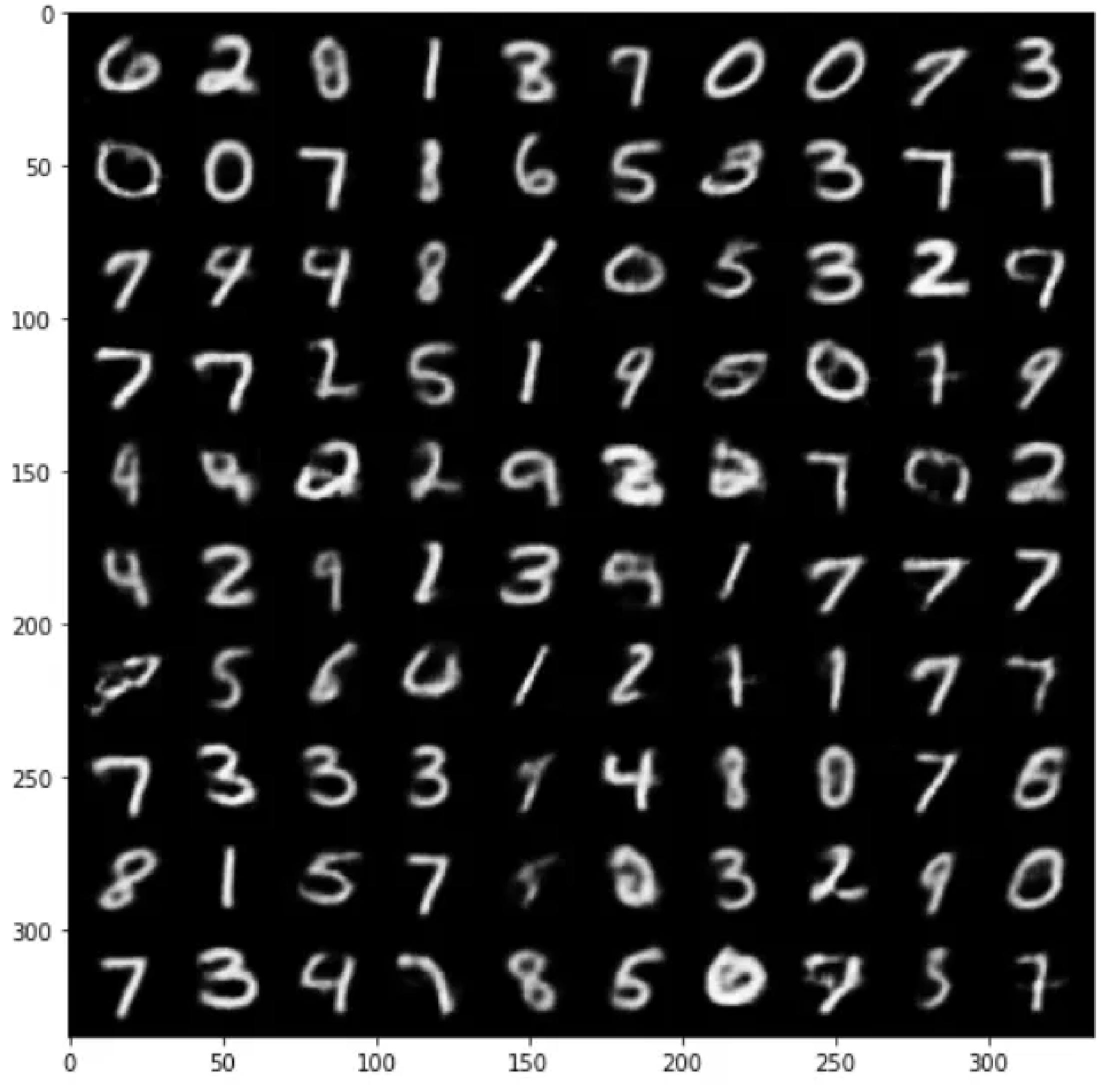
It’s time to finally train the VAE and evaluate in the validation set:



```
EPOCH 50/50     train loss 31.844       val loss 32.127
```

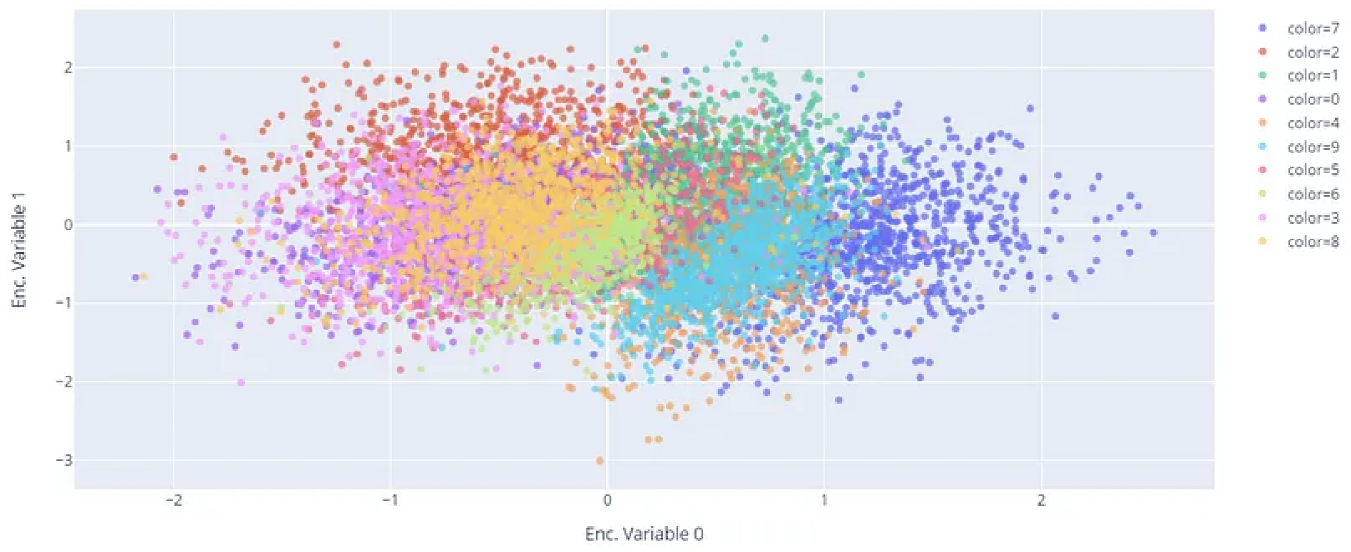
These are the results obtained after 50 epochs. We see that there is a high similarity between the input images and their reconstructions, even if there are still some imperfections. The reconstructed digits lose some details and are blurred compared to the original ones.

To estimate the ability to learn for the Variational Autoencoder, we can also generate new images by drawing latent vectors from the prior distribution, which is normal:



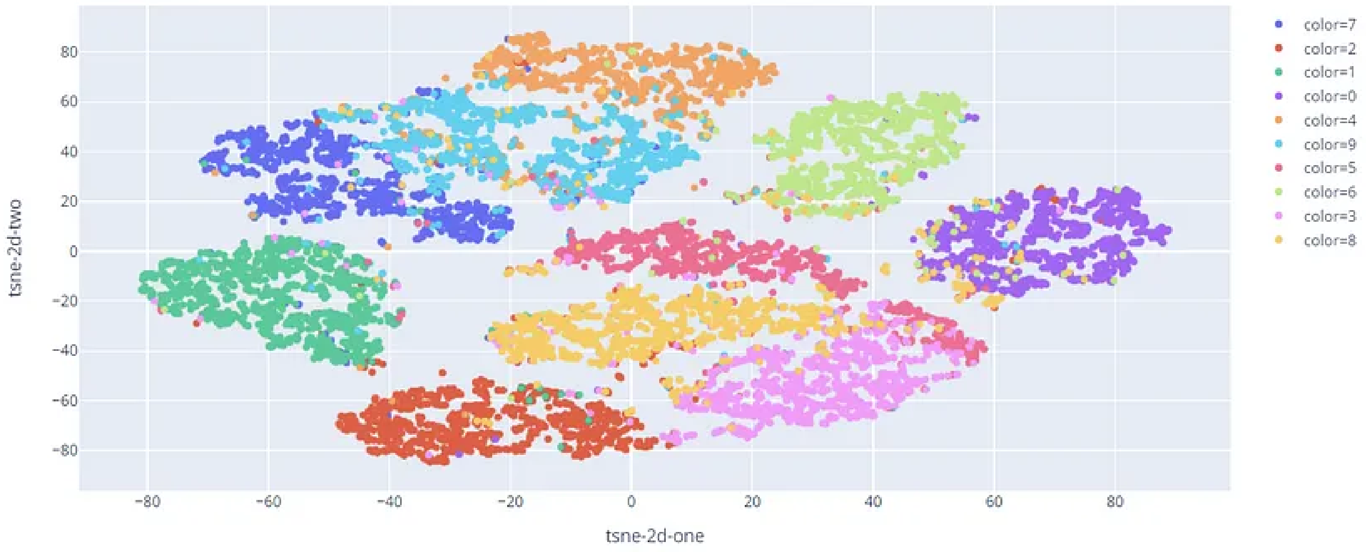
Most of the generated samples look like digits, so the Variational Autoencoder seems to have learned robust patterns from the latent space.

We can visualize the latent code learned by the variational decoder and we color by the ten classes of digits:



The range of values of the latent space is smaller and more centralized. The overall distribution seems close to a Gaussian distribution.

A better visualization can be obtained by applying the t-SNE, a reduction dimensionality method. With two components, I can visualize the latent code:



The resulting latent code seems to cluster the digits in different groups. There is also a slight overlapping between the different digits.

Final thoughts:

Congratulations! You have learned to implement and train a Variational Autoencoder with Pytorch. It's an extension of the autoencoder, where the only difference is that it encodes the input as a distribution over the latent space. Below there are some resources that helped me to study the VAE in a deeper way. I hope you found it useful. The GitHub code is here. Have a nice day.

References:

[1] https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

[2] https://atcold.github.io/pytorch-Deep-Learning/en/week08/08-3/

[3] https://avandekleut.github.io/vae/

Did you like my article? Become a member and get unlimited access to new data science posts every day! It's an indirect way of supporting me without any extra cost to you. If you are already a member, subscribe to get emails whenever I publish new data science and python guides!

Machine Learning Data Science Variational Autoencoder Pytorch Regularization