Jun 28, 2021 · 6 min read · ✦ · Listen

Convolutional Autoencoder in Pytorch on MNIST dataset

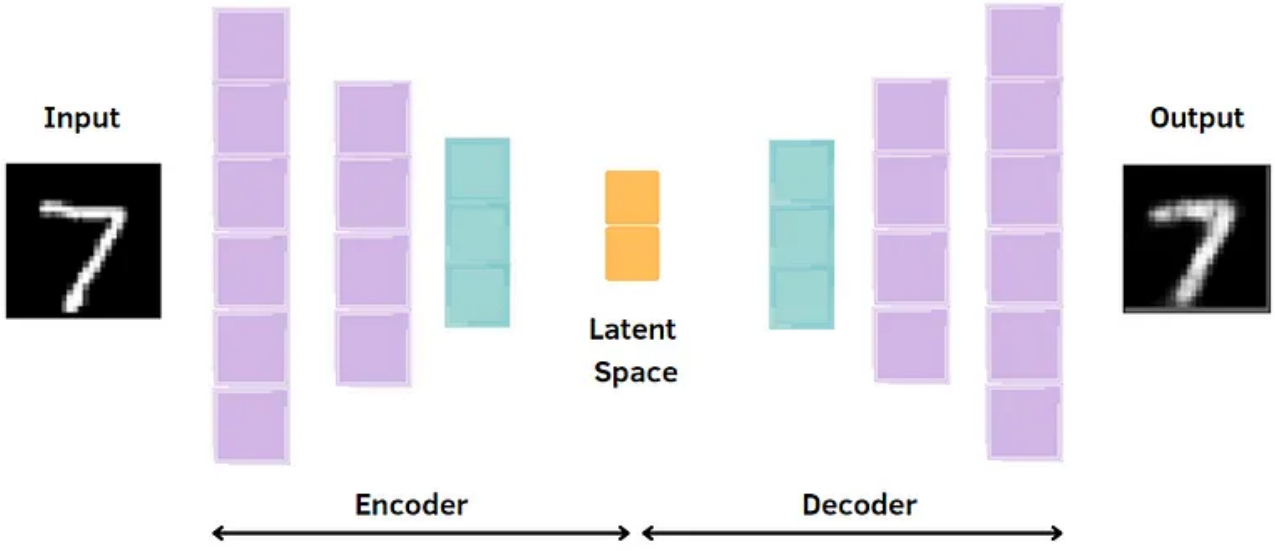


Illustration by Author

The post is the seventh in a series of guides to build 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 Opt*
6. *K Fold Cross Validation*
7. *Convolutional Autoencoder (this post)*
8. *Denoising Autoencoder*
9. *Variational Autoencoder*

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 a tutorial to another. In this series, I want to start from the simplest topics to the more advanced ones.

Autoencoder

The autoencoder is an unsupervised deep learning algorithm that learns encoded representations of the input data and then reconstructs the same input as output. It consists of two networks, Encoder and Decoder. The Encoder compresses the high-dimensional input into a low-dimensional latent code, called also latent code or encoded space, to extract the most relevant information from it, while the Decoder decompresses the encoded data and recreates the original input.

The goal of this architecture is to maximize the information when encoding and minimize the reconstruction error. But what is the reconstruction error? Its name is also reconstruction loss and is usually the mean-squared error between the

reconstructed input and the original input when the input is real-valued. In case the input data is categorical, the loss function used is the Cross-Entropy Loss.

Implementation in Pytorch

The following steps will be shown:

1. Import libraries and MNIST dataset
2. Define Convolutional Autoencoder
3. Initialize Loss function and Optimizer
4. Train model and evaluate model
5. Generate new samples from the latent code
6. Visualize the latent space with t-SNE

1. Import libraries and MNIST dataset

We can import the dataset using the library torchvision. We download the training and the test datasets and we transform the image datasets into Tensor. We don't need to normalize the images because the datasets contain colored images. After we divide the training dataset into training and validation sets.The `random_split` provides a random partition for these two sets. The `DataLoader` is used to create data loaders for the training, validation, and test sets, which are split into mini-batches. The `batchsize` is the number of samples used in one iteration during the training of the model.

```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

2. Define Convolutional Autoencoder

Here, we define the Autoencoder with Convolutional layers. It will be composed of two classes: one for the encoder and one for the decoder. The encoder will contain three

convolutional layers and two fully connected layers. Some batch norm layers are added as regularizers. The decoder will have the same architecture but in inverse order.

```
class Encoder(nn.Module):

    def __init__(self, encoded_space_dim,fc2_input_dim):
        super().__init__()

        ### Convolutional section
        self.encoder_cnn = nn.Sequential(
            nn.Conv2d(1, 8, 3, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.Conv2d(16, 32, 3, stride=2, padding=0),
            nn.ReLU(True)
        )

        ### Flatten layer
        self.flatten = nn.Flatten(start_dim=1)
### Linear section
        self.encoder_lin = nn.Sequential(
            nn.Linear(3 * 3 * 32, 128),
            nn.ReLU(True),
            nn.Linear(128, encoded_space_dim)
        )

    def forward(self, x):
        x = self.encoder_cnn(x)
        x = self.flatten(x)
        x = self.encoder_lin(x)
        return x
class Decoder(nn.Module):

    def __init__(self, encoded_space_dim,fc2_input_dim):
        super().__init__()
        self.decoder_lin = nn.Sequential(
            nn.Linear(encoded_space_dim, 128),
            nn.ReLU(True),
            nn.Linear(128, 3 * 3 * 32),
            nn.ReLU(True)
        )

        self.unflatten = nn.Unflatten(dim=1,
        unflattened_size=(32, 3, 3))

        self.decoder_conv = nn.Sequential(
```

```
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 3,
            stride=2, output_padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2,
            padding=1, output_padding=1),
            nn.BatchNorm2d(8),
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 3, stride=2,
            padding=1, output_padding=1)
```

```
### Define the loss function
loss_fn = torch.nn.MSELoss()

### Define an optimizer (both for the encoder and the decoder!)
lr= 0.001

### Set the random seed for reproducible results
torch.manual_seed(0)

### Initialize the two networks
d = 4

#model = Autoencoder(encoded_space_dim=encoded_space_dim)
encoder = Encoder(encoded_space_dim=d,fc2_input_dim=128)
decoder = Decoder(encoded_space_dim=d,fc2_input_dim=128)
params_to_optimize = [
    {'params': encoder.parameters()},
    {'params': decoder.parameters()}
]

optim = torch.optim.Adam(params_to_optimize, lr=lr, weight_decay=1e-05)

# Check if the GPU is available
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print(f'Selected device: {device}')

# Move both the encoder and the decoder to the selected device
encoder.to(device)
decoder.to(device)
```

.py hosted with ❤ by GitHub view raw

4. Train and evaluate model

We define a function to train the AE model. First, we pass the input images to the encoder. Later, the encoded data is passed to the decoder and then we compute the reconstruction loss with `loss_fn(x_hat,x)`. After we clear the gradient to not accumulate other values, we perform backpropagation and at the end, we compute the gradient by calling `opt.step()`.

```python
### Training function
def train_epoch(encoder, decoder, device, dataloader, loss_fn, optimizer):
    # Set train mode for both the encoder and the decoder
    encoder.train()
    decoder.train()
    train_loss = []
    # Iterate the dataloader (we do not need the label values, this is unsupervised learning)
    for image_batch, _ in dataloader: # with "_" we just ignore the labels (the second element of
        # Move tensor to the proper device
        image_batch = image_batch.to(device)
        # Encode data
        encoded_data = encoder(image_batch)
        # Decode data
        decoded_data = decoder(encoded_data)
        # Evaluate loss
        loss = loss_fn(decoded_data, image_batch)
        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Print batch loss
        print('\t partial train loss (single batch): %f' % (loss.data))
        train_loss.append(loss.detach().cpu().numpy())

    return np.mean(train_loss)
```

.py hosted with ❤ by GitHub view raw

Once the training function is created, we define a function to evaluate the performance of the model. As before, we pass the image to the encoder. The encoded image is passed to the decoder. Then, we store all the image batches and the reconstruction into two different lists, that will be used to calculate the test loss.

```python
### Testing function
def test_epoch(encoder, decoder, device, dataloader, loss_fn):
    # Set evaluation mode for encoder and decoder
    encoder.eval()
    decoder.eval()
    with torch.no_grad(): # No need to track the gradients
        # Define the lists to store the outputs for each batch
        conc_out = []
        conc_label = []
        for image_batch, _ in dataloader:
            # Move tensor to the proper device
            image_batch = image_batch.to(device)
            # Encode data
            encoded_data = encoder(image_batch)
            # Decode data
            decoded_data = decoder(encoded_data)
            # Append the network output and the original image to the lists
            conc_out.append(decoded_data.cpu())
            conc_label.append(image_batch.cpu())
        # Create a single tensor with all the values in the lists
        conc_out = torch.cat(conc_out)
        conc_label = torch.cat(conc_label)
        # Evaluate global loss
        val_loss = loss_fn(conc_out, conc_label)
    return val_loss.data
```

.py hosted with ❤ by GitHub view raw

We also would like to see the reconstructed images during each epoch of the training. The goal is to understand how the autoencoder is learning from the input images.

```python
def plot_ae_outputs(encoder,decoder,n=10):
    plt.figure(figsize=(16,4.5))
    targets = test_dataset.targets.numpy()
    t_idx = {i:np.where(targets==i)[0][0] for i in range(n)}
    for i in range(n):
      ax = plt.subplot(2,n,i+1)
      img = test_dataset[t_idx[i]][0].unsqueeze(0).to(device)
      encoder.eval()
      decoder.eval()
      with torch.no_grad():
         rec_img  = decoder(encoder(img))
      plt.imshow(img.cpu().squeeze().numpy(), cmap='gist_gray')
      ax.get_xaxis().set_visible(False)
      ax.get_yaxis().set_visible(False)
      if i == n//2:
        ax.set_title('Original images')
      ax = plt.subplot(2, n, i + 1 + n)
      plt.imshow(rec_img.cpu().squeeze().numpy(), cmap='gist_gray')
      ax.get_xaxis().set_visible(False)
      ax.get_yaxis().set_visible(False)
      if i == n//2:
         ax.set_title('Reconstructed images')
    plt.show()
```

.py hosted with ❤ by GitHub view raw

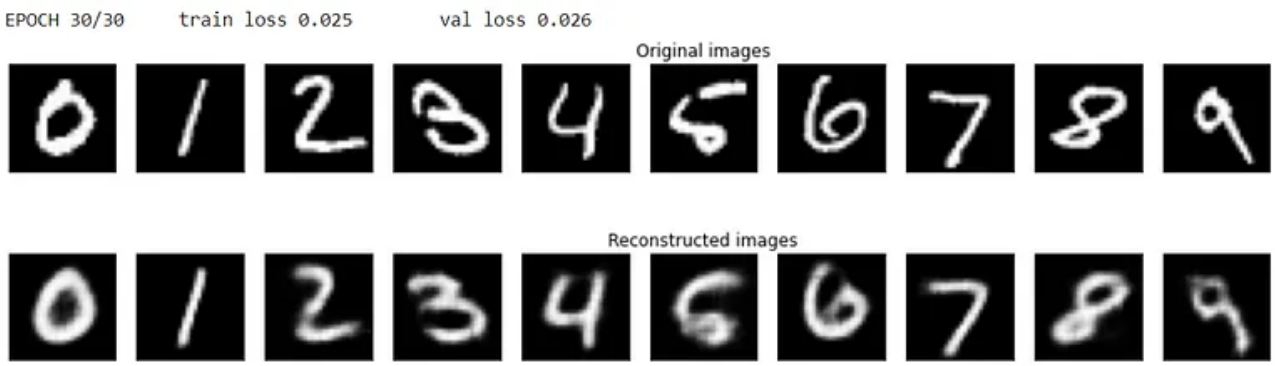
Let's break the test code into little pieces:

- `test_dataset[i][0].unsqueeze(0)` is used to extract the ith image from the test dataset and then it will be increased by 1 dimension on the 0 axis. This step is needed to pass the image to the autoencoder.
- `decoder(encoder(img))` is used to obtain the reconstructed image
- `plt.imshow(img.cpu().squeeze().numpy())` is used to plot the original image. `squeeze()` removes the dimension added before and is essential to visualize the image. `numpy()` transforms a tensor into a n-dimensional array, which is the only type of object accepted by the function `plt.imshow`. `numpy()` returns a copy of the tensor object into CPU memory.

Now we can finally begin to train the model on the training set and evaluate it on the validation set.

```
num_epochs = 30
diz_loss = {'train_loss':[],'val_loss':[]}
for epoch in range(num_epochs):
   train_loss =train_epoch(encoder,decoder,device,
   train_loader,loss_fn,optim)
   val_loss = test_epoch(encoder,decoder,device,test_loader,loss_fn)
   print('\n EPOCH {}/{} \t train loss {} \t val loss {}'.format(epoch + 1, num_epochs,train_loss,
   diz_loss['train_loss'].append(train_loss)
   diz_loss['val_loss'].append(val_loss)
   plot_ae_outputs(encoder,decoder,n=10)
```

.py hosted with ❤ by GitHub view raw



It's possible to notice that the autoencoder is able to reconstruct well the images after 30 epochs, even if there are some imperfections. But since this model is really simple, it performed very well. Now the model is trained and we want to do a final evaluation of the test set:

```
test_epoch(encoder,decoder,device,test_loader,loss_fn).item()
```

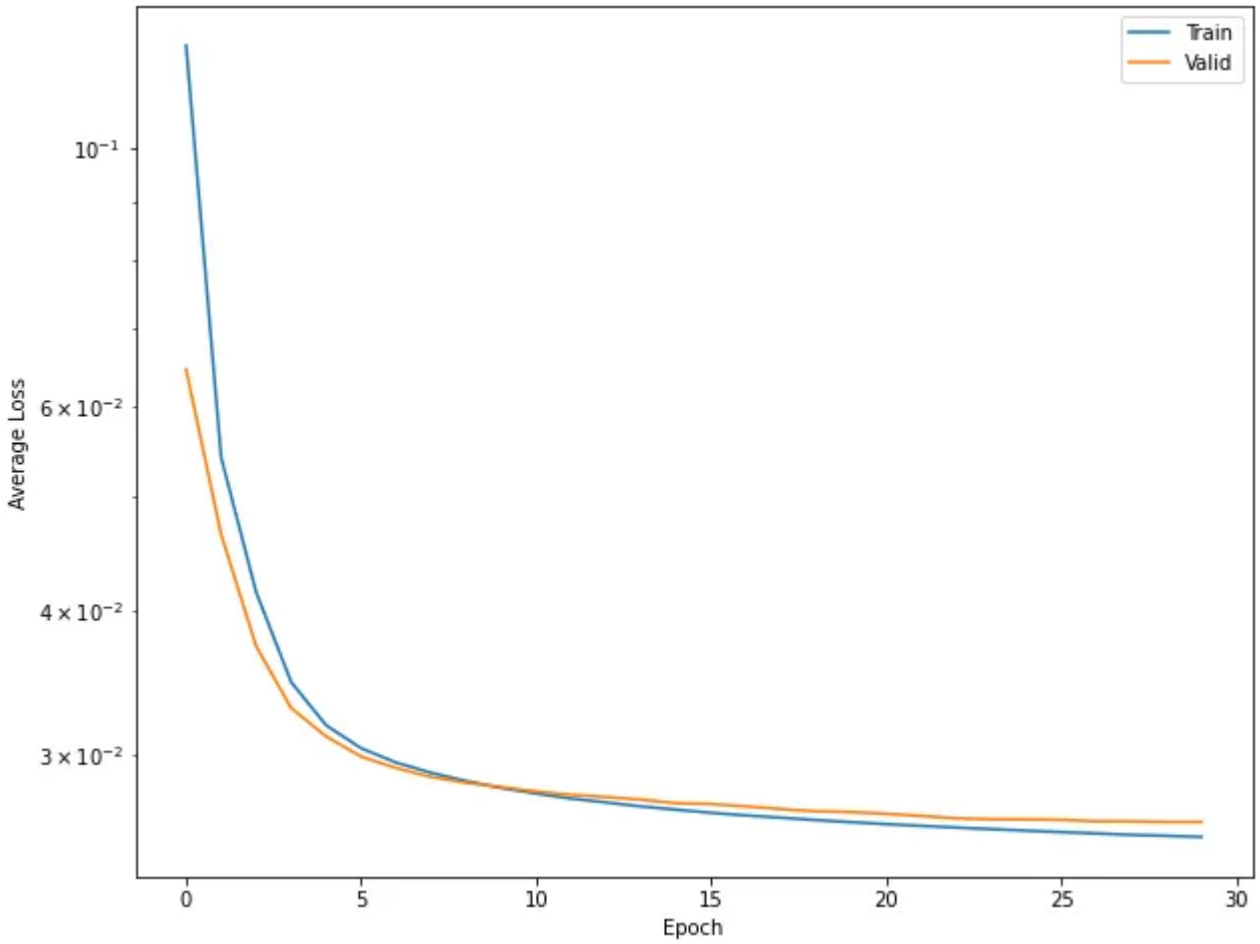
.py hosted with ❤ by GitHub view raw

```
0.026236288249492645
```

We can also observe how the reconstruction losses decrease over the epochs:

```python
# Plot losses
plt.figure(figsize=(10,8))
plt.semilogy(diz_loss['train_loss'], label='Train')
plt.semilogy(diz_loss['val_loss'], label='Valid')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
#plt.grid()
plt.legend()
#plt.title('loss')
plt.show()
```

.py hosted with ❤ by GitHub view raw

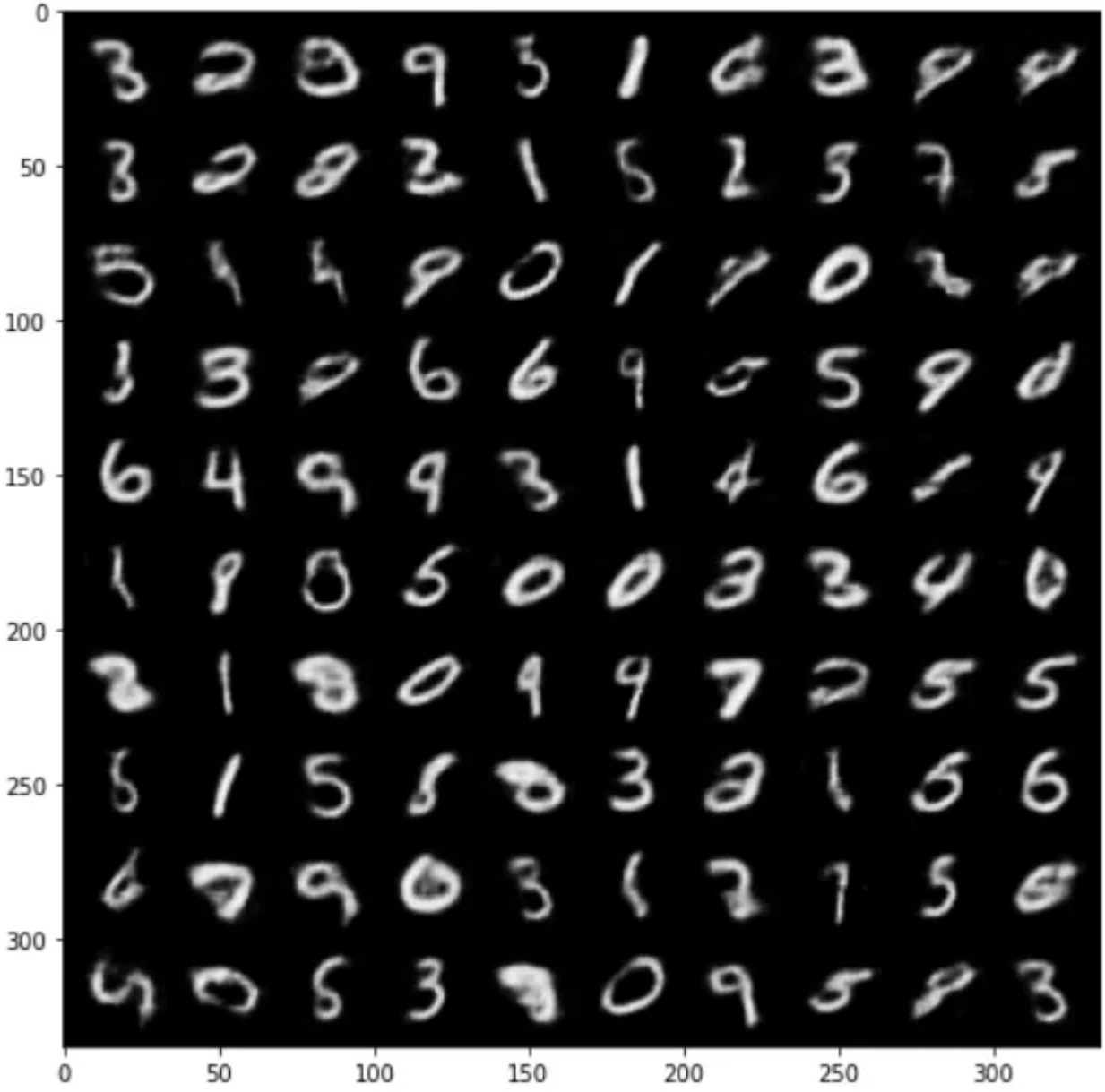


5. Generate new samples from the random latent code

To generate new images from the latent code, we sample randomly from a normal distribution with the mean and standard deviation of the encoded data. These samples will be passed to the decoder, which will create the reconstructed images.

```python
def show_image(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

encoder.eval()
decoder.eval()

with torch.no_grad():
    # calculate mean and std of latent code, generated takining in test images as inputs
    images, labels = iter(test_loader).next()
    images = images.to(device)
    latent = encoder(images)
    latent = latent.cpu()

    mean = latent.mean(dim=0)
    print(mean)
    std = (latent - mean).pow(2).mean(dim=0).sqrt()
    print(std)

    # sample latent vectors from the normal distribution
    latent = torch.randn(128, d)*std + mean

    # reconstruct images from the random latent vectors
    latent = latent.to(device)
    img_recon = decoder(latent)
    img_recon = img_recon.cpu()

    fig, ax = plt.subplots(figsize=(20, 8.5))
    show_image(torchvision.utils.make_grid(img_recon[:100],10,5))
    plt.show()
```

.py hosted with ❤ by GitHub view raw



It should notice that this procedure allows the sampling to be in the same region as the latent code but at the same time there are digits that make no sense. This aspect is explained by the fact that the **latent space of the autoencoder is extremely irregular:** close points in the latent space can produce very different and meaningless patterns over visible units. For this reason, the autoencoder doesn't perform well for generative purposes.

6. Visualize the latent space with t-SNE

After we can observe dynamic visualization to see the latent space learned by the autoencoder. First, we create the encoded samples using the test set.

```python
encoded_samples = []
for sample in tqdm(test_dataset):
    img = sample[0].unsqueeze(0).to(device)
    label = sample[1]
    # Encode image
    encoder.eval()
    with torch.no_grad():
        encoded_img  = encoder(img)
    # Append to list
    encoded_img = encoded_img.flatten().cpu().numpy()
    encoded_sample = {f"Enc. Variable {i}": enc for i, enc in enumerate(encoded_img)}
    encoded_sample['label'] = label
    encoded_samples.append(encoded_sample)
encoded_samples = pd.DataFrame(encoded_samples)
encoded_samples
```

.py hosted with ❤ by GitHub view raw

Let's plot the latent space representation using plotly express library:

```python
import plotly.express as px

px.scatter(encoded_samples, x='Enc. Variable 0', y='Enc. Variable 1',
           color=encoded_samples.label.astype(str), opacity=0.7)
```

.py hosted with ❤ by GitHub view raw

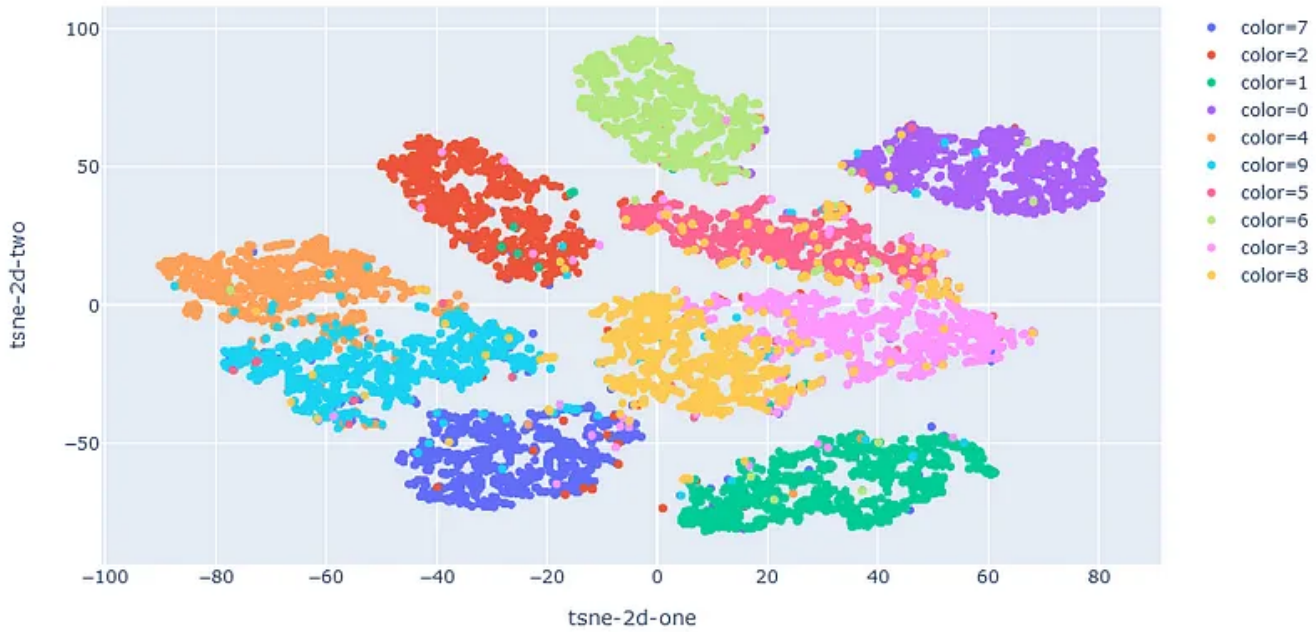
Visualization of latent space

From this plot, we see that similar digits are clustered together. For example “4” overlap with “9” and “5”.

In order to have an easier representation to read, we can apply a dimensionality reduction, called t-SNE, to visualize the latent code in a 2d space. We’ll fix the number of components equal to 2 for this reason.

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2)
tsne_results = tsne.fit_transform(encoded_samples.drop(['label'],axis=1))
fig = px.scatter(tsne_results, x=0, y=1,
                 color=encoded_samples.label.astype(str),
                 labels={'0': 'tsne-2d-one', '1': 'tsne-2d-two'})
fig.show()
```

.py hosted with ❤ by GitHub view raw



You can see that it distinguishes clearly one digit from the other. There are some exceptions with points that fall in other categories, but t-SNE still remains an improvement compared to the previous representation.

Final thoughts:

Congratulations! You have learned to implement a Convolutional autoencoder. There aren't many tutorials that talk about autoencoders with convolutional layers with Pytorch, so I wanted to contribute in some way. The autoencoder provides a way to compress images and extract the most important information. There are also many extensions of this model to improve the performance, some of these are the Denoising Autoencoder, the Variational Autoencoder, and the Generative Adversarial Networks. The GitHub code is here. Thanks for reading. Have a nice day.

Reference:

[1] https://github.com/smartgeometry-ucl/dl4g

Machine Learning Data Science Convolutional Autoencoder Pytorch