

```
1 # https://github.com/eugeniarling/Medium-Articles/blob
  /main/Pytorch/denAE.ipynb
2 import PIL.Image
3 import matplotlib.pyplot as plt
4 import numpy as np # this module is useful to work
  with numerical arrays
5 import pandas as pd # this module is useful to work
  with tabular data
6 import random # this module will be used to select
  random samples from a collection
7 import os # this module will be used just to create
  directories in the local filesystem
8 from tqdm import tqdm # this module is useful to plot
  progress bars
9 import plotly.io as pio
10 from sklearn.cluster import KMeans
11 import torch
12 import torchvision
13 from torchvision import transforms
14 from torch.utils.data import DataLoader, random_split
15 from torch import nn
16 import torch.nn.functional as F
17 import torch.optim as optim
18 from sklearn.manifold import TSNE
19 import plotly.express as px
20 from matplotlib import image as mpimg
21 from PIL import Image
22 from collections import Counter
23
24 data_dir = 'dataset'
25 ### With these commands the train and test datasets,
  respectively, are downloaded
26 ### automatically and stored in the local "data_dir"
  directory.
27 train_dataset = torchvision.datasets.MNIST(data_dir,
  train=True, download=True)
28 test_dataset = torchvision.datasets.MNIST(data_dir,
  train=False, download=True)
29
30
31 fig, axs = plt.subplots(5, 5, figsize=(8,8))
```

```
32 for ax in axs.flatten():
33     # random.choice allows to randomly sample from a
    list-like object (basically anything that can be
    accessed with an index, like our dataset)
34     img, label = random.choice(train_dataset)
35     ax.imshow(np.array(img), cmap='gist_gray')
36     ax.set_title('Label: %d' % label)
37     ax.set_xticks([])
38     ax.set_yticks([])
39 plt.tight_layout()
40 plt.show()
41
42 train_transform = transforms.Compose([
43     transforms.ToTensor(),
44 ])
45 test_transform = transforms.Compose([
46     transforms.ToTensor(),
47 ])
48
49 # Set the train transform
50 train_dataset.transform = train_transform
51 # Set the test transform
52 test_dataset.transform = test_transform
53
54 m=len(train_dataset)
55
56 #random_split randomly split a dataset into non-
    overlapping new datasets of given lengths
57 #train (55,000 images), val split (5,000 images)
58 train_data, val_data = random_split(train_dataset, [
    int(m-m*0.2), int(m*0.2)])
59
60 batch_size=256
61
62 # The dataloaders handle shuffling, batching, etc...
63 train_loader = torch.utils.data.DataLoader(train_data
    , batch_size=batch_size)
64 valid_loader = torch.utils.data.DataLoader(val_data,
    batch_size=batch_size)
65 test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=batch_size, shuffle=True)
```

```

66
67
68
69
70 class Encoder(nn.Module):
71
72     def __init__(self, encoded_space_dim,
73         fc2_input_dim):
74         super().__init__()
75
76         ### Convolutional section
77         self.encoder_cnn = nn.Sequential(
78             # First convolutional layer
79             nn.Conv2d(1, 8, 3, stride=2, padding=1),
80             #nn.BatchNorm2d(8),
81             nn.ReLU(True),
82             # Second convolutional layer
83             nn.Conv2d(8, 16, 3, stride=2, padding=1
84         ),
85             nn.BatchNorm2d(16),
86             nn.ReLU(True),
87             # Third convolutional layer
88             nn.Conv2d(16, 32, 3, stride=2, padding=0
89         ),
90             #nn.BatchNorm2d(32),
91             nn.ReLU(True)
92     )
93
94     ### Flatten layer
95     #self.flatten = torch.flatten(start_dim=1)
96
97     ### Linear section
98     self.encoder_lin = nn.Sequential(
99         # First linear layer
100         nn.Linear(3 * 3 * 32, 128),
101         nn.ReLU(True),
102         # Second linear layer
103         nn.Linear(128, encoded_space_dim)
104     )
105
106     def forward(self, x):

```

```

104         # Apply convolutions
105         x = self.encoder_cnn(x)
106         # Flatten
107         x = torch.flatten(x, start_dim=1)
108         # # Apply linear layers
109         x = self.encoder_lin(x)
110         return x
111
112
113
114
115
116 class Decoder(nn.Module):
117
118     def __init__(self, encoded_space_dim,
119                  fc2_input_dim):
119         super().__init__()
120
121         ### Linear section
122         self.decoder_lin = nn.Sequential(
123             # First linear layer
124             nn.Linear(encoded_space_dim, 128),
125             nn.ReLU(True),
126             # Second linear layer
127             nn.Linear(128, 3 * 3 * 32),
128             nn.ReLU(True)
129         )
130
131
132         ### Convolutional section
133         self.decoder_conv = nn.Sequential(
134             # First transposed convolution
135             nn.ConvTranspose2d(32, 16, 3, stride=2,
136                               output_padding=0),
137             nn.BatchNorm2d(16),
138             nn.ReLU(True),
139             # Second transposed convolution
140             nn.ConvTranspose2d(16, 8, 3, stride=2,
141                               padding=1, output_padding=1),
142             nn.BatchNorm2d(8),
143             nn.ReLU(True),

```

```

142         # Third transposed convolution
143         nn.ConvTranspose2d(8, 1, 3, stride=2,
padding=1, output_padding=1)
144     )
145
146     def forward(self, x):
147         # Apply linear layers
148         x = self.decoder_lin(x)
149         # Unflatten
150         x = nn.Unflatten(dim=1, unflattened_size=(32
, 3, 3))(x)
151         # Apply transposed convolutions
152         x = self.decoder_conv(x)
153         # Apply a sigmoid to force the output to be
between 0 and 1 (valid pixel values)
154         x = torch.sigmoid(x)
155         return x
156
157
158
159
160 ### Set the random seed for reproducible results
161 torch.manual_seed(0)
162
163 ### Initialize the two networks
164 d = 4
165
166 encoder = Encoder(encoded_space_dim=d, fc2_input_dim=
128)
167 decoder = Decoder(encoded_space_dim=d, fc2_input_dim=
128)
168
169
170 ### Define the loss function
171 loss_fn = torch.nn.MSELoss()
172
173 ### Define an optimizer (both for the encoder and
the decoder!)
174 lr= 0.001 # Learning rate
175
176

```

```

177 params_to_optimize = [
178     {'params': encoder.parameters()},
179     {'params': decoder.parameters()}
180 ]
181
182 #device = torch.device("cuda") if torch.cuda.
    is_available() else torch.device("cpu")
183 device = torch.device("cpu")
184
185 # print(f'Selected device: {device}')
186
187 optim = torch.optim.Adam(params_to_optimize, lr=lr)
188
189 # Move both the encoder and the decoder to the
    selected device
190 encoder.to(device)
191 decoder.to(device)
192 #model.to(device)
193 encoded_all = []
194
195
196 def add_noise(inputs, noise_factor=0.3):
197     noise = inputs+torch.randn_like(inputs)*
        noise_factor
198     noise = torch.clamp(noise,0.,1.)
199     return noise
200
201 ### Training function
202 def train_epoch_den(encoder, decoder, device,
    dataloader, loss_fn, optimizer, noise_factor=0.3):
203     # Set train mode for both the encoder and the
        decoder
204     encoder.train()
205     decoder.train()
206     train_loss = []
207     # Iterate the dataloader (we do not need the
        label values, this is unsupervised learning)
208     for image_batch, _ in dataloader: # with "_" we
        just ignore the labels (the second element of the
        dataloader tuple)
209         # Move tensor to the proper device

```

```

210         image_batch = image_batch.to(device)
211         image_noisy = add_noise(image_batch,
    noise_factor)
212         image_noisy = image_noisy.to(device)
213         # Encode data
214         encoded_data = encoder(image_noisy)
215         # Decode data
216         decoded_data = decoder(encoded_data)
217         # Evaluate loss
218         loss = loss_fn(decoded_data, image_batch)
219         # Backward pass
220         optimizer.zero_grad()
221         loss.backward()
222         optimizer.step()
223         # Print batch loss
224         #print('\t partial train loss (single batch
    ): %f' % (loss.data))
225         train_loss.append(loss.detach().cpu().numpy
    ())
226
227     return np.mean(train_loss)
228
229
230 ### Testing function
231 def test_epoch_den(encoder, decoder, device,
    dataloader, loss_fn, noise_factor=0.3):
232     # Set evaluation mode for encoder and decoder
233     encoder.eval()
234     decoder.eval()
235     with torch.no_grad(): # No need to track the
    gradients
236         # Define the lists to store the outputs for
    each batch
237         conc_out = []
238         conc_label = []
239         for image_batch, _ in dataloader:
240             # Move tensor to the proper device
241             image_noisy = add_noise(image_batch,
    noise_factor)
242             image_noisy = image_noisy.to(device)
243             # Encode data

```

```

244         encoded_data = encoder(image_noisy)
245         # Decode data
246         decoded_data = decoder(encoded_data)
247         # Append the network output and the
        original image to the lists
248         conc_out.append(decoded_data.cpu())
249         conc_label.append(image_batch.cpu())
250         # Create a single tensor with all the values
        in the lists
251         conc_out = torch.cat(conc_out)
252         conc_label = torch.cat(conc_label)
253         # Evaluate global loss
254         val_loss = loss_fn(conc_out, conc_label)
255         return val_loss.data
256
257 def plot_ae_outputs_den(encoder, decoder, n=5,
        noise_factor=0.3):
258     plt.figure(figsize=(10, 4.5))
259     for i in range(n):
260
261         ax = plt.subplot(3, n, i+1)
262         img = test_dataset[i][0].unsqueeze(0)
263         image_noisy = add_noise(img, noise_factor)
264         image_noisy = image_noisy.to(device)
265
266         encoder.eval()
267         decoder.eval()
268
269         with torch.no_grad():
270             rec_img = decoder(encoder(image_noisy))
271
272         plt.imshow(img.cpu().squeeze().numpy(), cmap='
        gist_gray')
273         ax.get_xaxis().set_visible(False)
274         ax.get_yaxis().set_visible(False)
275         if i == n//2:
276             ax.set_title('Original images')
277         ax = plt.subplot(3, n, i + 1 + n)
278         plt.imshow(image_noisy.cpu().squeeze().numpy
        (), cmap='gist_gray')
279         ax.get_xaxis().set_visible(False)

```



```

280         ax.get_yaxis().set_visible(False)
281         if i == n//2:
282             ax.set_title('Corrupted images')
283
284         ax = plt.subplot(3, n, i + 1 + n + n)
285         plt.imshow(rec_img.cpu().squeeze().numpy(),
cmap='gist_gray')
286         ax.get_xaxis().set_visible(False)
287         ax.get_yaxis().set_visible(False)
288         if i == n//2:
289             ax.set_title('Reconstructed images')
290     plt.subplots_adjust(left=0.1,
291                        bottom=0.1,
292                        right=0.7,
293                        top=0.9,
294                        wspace=0.3,
295                        hspace=0.3)
296     plt.show()
297
298     ### Training cycle
299     noise_factor = 0.3
300     num_epochs = 30
301     history_da={'train_loss':[], 'val_loss':[]}
302
303     for epoch in range(num_epochs):
304         print('EPOCH %d/%d' % (epoch + 1, num_epochs))
305         ### Training (use the training function)
306         train_loss=train_epoch_den(
307             encoder=encoder,
308             decoder=decoder,
309             device=device,
310             dataloader=train_loader,
311             loss_fn=loss_fn,
312             optimizer=optim,noise_factor=noise_factor)
313         ### Validation (use the testing function)
314         val_loss = test_epoch_den(
315             encoder=encoder,
316             decoder=decoder,
317             device=device,
318             dataloader=valid_loader,
319             loss_fn=loss_fn,noise_factor=noise_factor)

```

```

320     # Print Validationloss
321     history_da['train_loss'].append(train_loss)
322     history_da['val_loss'].append(val_loss)
323     print('\n EPOCH {}/{} \t train loss {:.3f} \t
    val loss {:.3f}'.format(epoch + 1, num_epochs,
    train_loss, val_loss))
324     plot_ae_outputs_den(encoder, decoder, noise_factor
    =noise_factor)
325
326
327 # put your image generator here
328 random_tensor = torch.rand((9,4))
329 decoder_imgs = decoder(random_tensor).detach()
330
331 fig, axs = plt.subplots(3, 3, figsize=(5,5))
332 i=0
333 for ax in axs.flatten():
334     # random.choice allows to randomly sample from a
    list-like object (basically anything that can be
    accessed with an index, like our dataset)
335     img = decoder_imgs[i].squeeze().numpy()
336     ax.imshow(img, cmap='gist_gray')
337     ax.set_title('Object: %d' % i)
338     ax.set_xticks([])
339     ax.set_yticks([])
340     i+=1
341 plt.tight_layout()
342 plt.show()
343
344 #Creating Clusters
345 k = 10
346 all_encod = np.ones((48000,4))
347 i=0
348 cluster_loader = torch.utils.data.DataLoader(
    train_data, batch_size=1)
349 labels= np.ones((48000))
350 for image, label in cluster_loader:
351     image = image.to(device)
352     encoded_data = encoder(image).detach().flatten
    ().numpy()
353     all_encod[i] = encoded_data

```

```
354     labels[i] = label.numpy()
355     i+=1
356 clusters = KMeans(k, random_state = 40).fit_predict(
    all_encod)
357 dict={}
358 for i in range(10):
359     label_list=[]
360     for j in range(len(clusters)):
361         if clusters[j] == i:
362             label_list.append(labels[j])
363     dict[i] = Counter(label_list).most_common()[0][0]
    ]
364
365 correct=0
366 wrong=0
367 for cl,lb in zip(clusters,labels):
368     if dict[cl] - lb ==0:
369         correct+=1
370     else:
371         wrong+=1
372 accuracy = ((correct/(correct+wrong))*100)
373 print(accuracy)
374
375
376
377
378
379
380
381
382
383
384
385
386
387
```