

MNIST Inpainting with Energy-Based Learning

In this exercise, we consider the task of inpainting incomplete handwritten digits, and for this, we would like to make use of neural networks and the Energy-Based Learning framework.

```
In [1]: import torch
import torch.nn as nn
import utils
import numpy
%matplotlib inline
```

As a first step, we load the MNIST dataset

```
In [2]: Xr,Xt = utils.getdata()
```

We consider the following perturbation process that draws some region near the center of the image randomly and set the pixels in this area to some gray value.

```
In [3]: def removepatch(X):
mask = torch.zeros(len(X),28,28)
for i in range(len(X)):
    j = numpy.random.randint(-4,5)
    k = numpy.random.randint(-4,5)
    mask[i,11+j:17+j,11+k:17+k] = 1
mask = mask.view(len(X),784)
return (X*(1-mask)).data,mask
```

The outcome of the perturbation process can be visualized below:

```
In [4]: %matplotlib inline
xmask = removepatch(Xt[:10])[0]
utils.vis10(xmask)
```



PCA Reconstruction (20 P)

A simple technique for inpainting an image is principal component analysis. It consists of taking the incomplete image and projecting it on the d principal components of the training data.

Task:

- Implement a function that takes a collection of test examples \mathbf{z} and projects them on the d principal components of the training data \mathbf{x} .

```
In [5]: def pca(z,x,d):

# -----
# TODO: replace by your code
# -----

import solutions
y = solutions.pca(z,x,d)

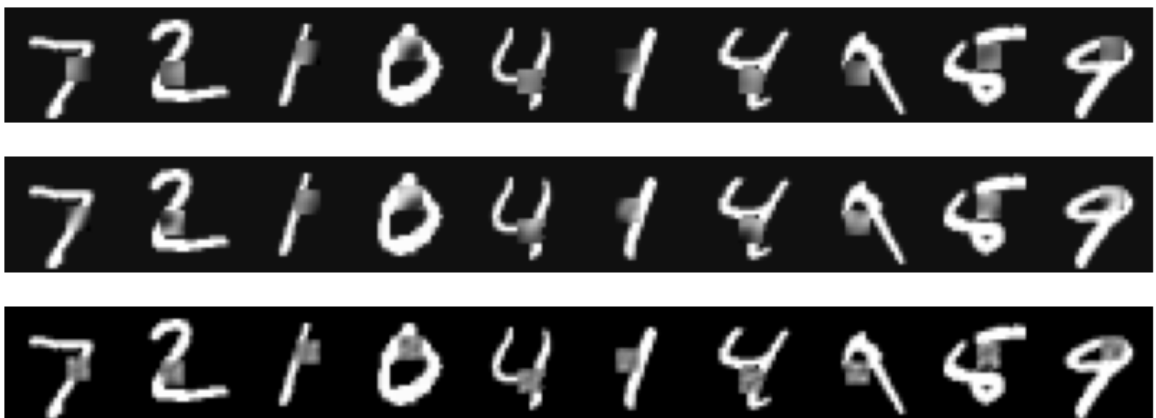
# -----

return y
```

The PCA-based inpainting technique is tested below on 10 test points for which a patch is missing. We observe that the patch-like perturbation is less severe when d is low, but the reconstructed part of the digit appears blurry. Conversely, if setting d high, more details become available, but the missing pattern appears more prominent.

```
In [6]: Xn,m = removepatch(Xt[:10])

utils.vis10(pca(Xn,Xr,1)*m+Xn*(1-m))
utils.vis10(pca(Xn,Xr,10)*m+Xn*(1-m))
utils.vis10(pca(Xn,Xr,100)*m+Xn*(1-m))
```



Energy-Based Learning (20 P)

We now consider the energy-based learning framework where we learn an energy function to discriminate between correct and incorrect reconstructions.

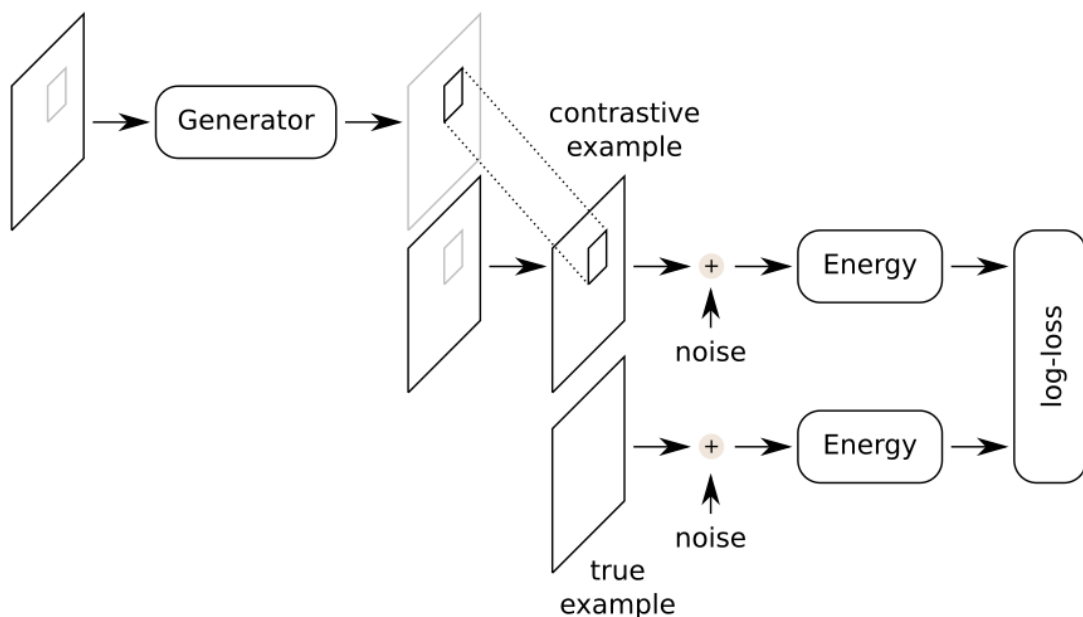
```
In [7]: torch.manual_seed(0)
enet = nn.Sequential(
    nn.Linear(784,1024),nn.Hardtanh(),
    nn.Linear(1024,1024),nn.Hardtanh(),
    nn.Linear(1024,1),
)
```

To be able to generate good contrastive examples (i.e. incorrect reconstructions that are still plausible enough to confuse the energy-based model and for which

meaningful gradient signal can be extracted), we consider a generator network that takes as input the incomplete images.

```
In [8]: gnet = nn.Sequential(
    nn.Linear(784,1024),nn.Hardtanh(),
    nn.Linear(1024,1024),nn.Hardtanh(),
    nn.Linear(1024,784),nn.Hardtanh(),
)
```

The whole architecture is depicted in the diagram below:



The two networks are then jointly optimized. We add a small additive noise (Gaussian with mean zero and variance one) before the energy model to tilt training slightly in favor of the generator that is harder to train. The structure of the optimization problem is already provided to you, however, the code that computes the forward pass from the input data up to the error function are missing.

Task:

- Write the code that computes the error function. Here, we use a single optimizer and must therefore implement the gradient flip trick described in the slides. A similar trick can be used to only let the gradient flow into the generator only via the missing image patch and not through all pixels.

```
In [9]: import torch.optim as optim

N = 2500
mb = 25

optimizer = optim.SGD(list(enet.parameters())+list(gnet.parameters()), lr

for epoch in numpy.arange(1,40):
    for i in range(N//mb):
```

```

optimizer.zero_grad()

# Take a minibatch and train it
x = Xr[mb*i:mb*(i+1)].data*1.0
z,m = removepatch(x)

# Build the forward pass from the input until the loss function

# -----
# TODO: replace by your code
# -----
import solutions
err = solutions.err(x,m,z,gnet,enet)
# -----

# Compute the gradient and perform one step of gradient descent
err.backward()
optimizer.step()

print('%3d %.3f'%(epoch,err.data.numpy()))

```

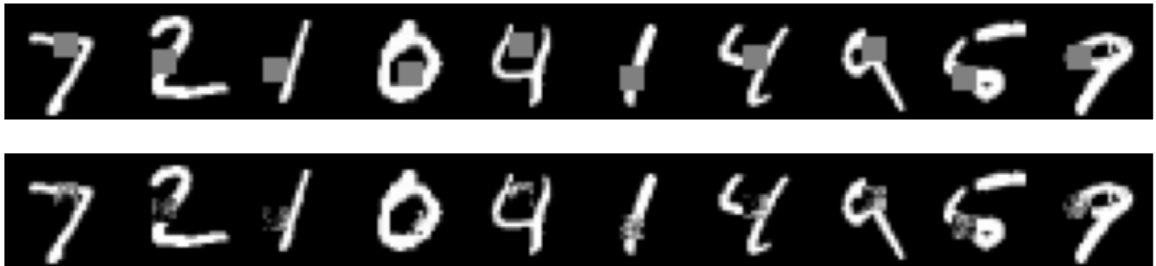
```

1 0.736
2 0.741
3 0.695
4 0.664
5 0.735
6 0.741
7 0.718
8 0.737
9 0.657
10 0.737
11 0.690
12 0.601
13 0.639
14 0.633
15 0.671
16 0.621
17 0.627
18 0.711
19 0.672
20 0.710
21 0.558
22 0.709
23 0.702
24 0.627
25 0.585
26 0.726
27 0.671
28 0.631
29 0.671
30 0.607
31 0.724
32 0.652
33 0.624
34 0.700
35 0.620
36 0.590
37 0.694
38 0.643
39 0.638

```

After optimizing for a sufficient number of epochs, the solution has ideally come close to some nash equilibrium where both the generator and energy-based model perform well. In particular, the generator should generate examples that look similar to the true examples. The code below plots the incomplete digits and the reconstruction obtained by the generator network.

```
In [10]: x = Xt[:10]
z,m = removepatch(x)
utils.vis10(z)
utils.vis10(gnet(z)*m+z*(1-m))
```



As we can see, although some artefacts still persist, the reconstructions are quite plausible and look better than those one gets with the simple PCA-based approach. Note however that the procedure is also more complex and computationally more demanding than a simple PCA-based reconstruction.