Exercises for the course
Machine Learning 2
Summer semester 2021

Abteilung Maschinelles Lernen
Institut für Softwaretechnik und theoretische Informatik
Fakultät IV, Technische Universität Berlin
Prof. Dr. Klaus-Robert Müller
Email: klaus-robert.mueller@tu-berlin.de

Exercise Sheet 10

Exercise 1: Mixture Density Networks (20 + 10 P)

In this exercise, we prove some of the results from the paper Mixture Density Networks by Bishop (1994). The mixture density network is given by

$$p(\boldsymbol{t}|\boldsymbol{x}) = \sum_{i=1}^{m} \alpha_i(\boldsymbol{x})\phi_i(\boldsymbol{t}|\boldsymbol{x})$$

with the mixture elements

$$\phi_i(\boldsymbol{t}|\boldsymbol{x}) = \frac{1}{(2\pi)^{c/2}\sigma_i(\boldsymbol{x})^c} \exp\Big(-\frac{\|\boldsymbol{t}-\boldsymbol{\mu}_i(\boldsymbol{x})\|^2}{2\sigma_i(\boldsymbol{x})^2}\Big).$$

The contribution to the error function of one data point q is given by

$$E^q = -\log\Big\{\sum_{i=1}^{m} \alpha_i(\boldsymbol{x}^q)\phi_i(\boldsymbol{t}^q|\boldsymbol{x}^q)\Big\}$$

We also define the posterior distribution

$$\pi_i(\boldsymbol{x},\boldsymbol{t}) = \frac{\alpha_i\phi_i}{\sum_{j=1}^{m}\alpha_j\phi_j}$$

which is obtained using the Bayes theorem.

(a) *Compute* the gradient of the error E^q w.r.t. the mixture parameters, i.e. show that

(i) $\dfrac{\partial E^q}{\partial \alpha_i} = -\dfrac{\pi_i}{\alpha_i}$

(ii) $\dfrac{\partial E^q}{\partial \mu_{ik}} = \pi_i\Big(\dfrac{\mu_{ik}-t_k}{\sigma_i^2}\Big)$

(b) We now assume that the neural network produces the mixture coefficients as:

$$\alpha_i = \frac{\exp(z_i^\alpha)}{\sum_{j=1}^{M}\exp(z_j^\alpha)}$$

where z^α denotes the outputs of the neural network (after the last linear layer) associated to these mixture coefficients. *Compute* using the chain rule for derivatives (i.e. by reusing some of the results in the first part of this exercise) the derivative $\partial E^q/\partial z_i^\alpha$.

Exercise 2: Conditional RBM (20 + 10 P)

The conditional restricted Boltzmann machine is a system of binary variable comprising inputs $\boldsymbol{x} \in \{0,1\}^d$, outputs $\boldsymbol{y} \in \{0,1\}^c$, and hidden units $\boldsymbol{h} \in \{0,1\}^K$. It associates to each configuration of these binary variables the energy:

$$E(\boldsymbol{x},\boldsymbol{y},\boldsymbol{h}) = -\boldsymbol{x}^\top W\boldsymbol{h} - \boldsymbol{y}^\top U\boldsymbol{h}$$

and the probability associated to each configuration is then given as:

$$p(\boldsymbol{x},\boldsymbol{y},\boldsymbol{h}) = \frac{1}{Z}\exp(-E(\boldsymbol{x},\boldsymbol{y},\boldsymbol{h}))$$

where Z is a normalization constant that makes probabilities sum to one.

(a) Let $\mathrm{sigm}(t) = \exp(t)/(1+\exp(t))$ be the sigmoid function. *Show* that

(i) $p(h_k = 1 \,|\, \boldsymbol{x}, \boldsymbol{y}) = \mathrm{sigm}\big(\boldsymbol{x}^\top W_{:,k} + \boldsymbol{y}^\top U_{:,k}\big)$

(ii) $p(y_j = 1 \,|\, \boldsymbol{h}, \boldsymbol{x}) = \mathrm{sigm}\big(U_{j,:}^\top \boldsymbol{h}\big)$

(b) *Show* that

$$p(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{Z} \exp(-F(\boldsymbol{x}, \boldsymbol{y}))$$

where

$$F(\boldsymbol{x}, \boldsymbol{y}) = -\sum_{k=1}^{K} \log\left(1 + \exp\left(\boldsymbol{x}^\top W_{:,k} + \boldsymbol{y}^\top U_{:,k}\right)\right)$$

is the free energy and where Z is again a normalization constant.

Exercise 3: Programming (40 P)

Download the programming files on ISIS and follow the instructions.

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 impainting 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 `z` and projects them on the d principal components of the training data `x`.**

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

    # -------------------------------
    # TODO: replace by your code
    # -------------------------------
    import solution
    y = solution.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,10)*m+Xn*(1-m))
utils.vis10(pca(Xn,Xr,60)*m+Xn*(1-m))
utils.vis10(pca(Xn,Xr,360)*m+Xn*(1-m))
```

Energy-Based Learning (20 P)

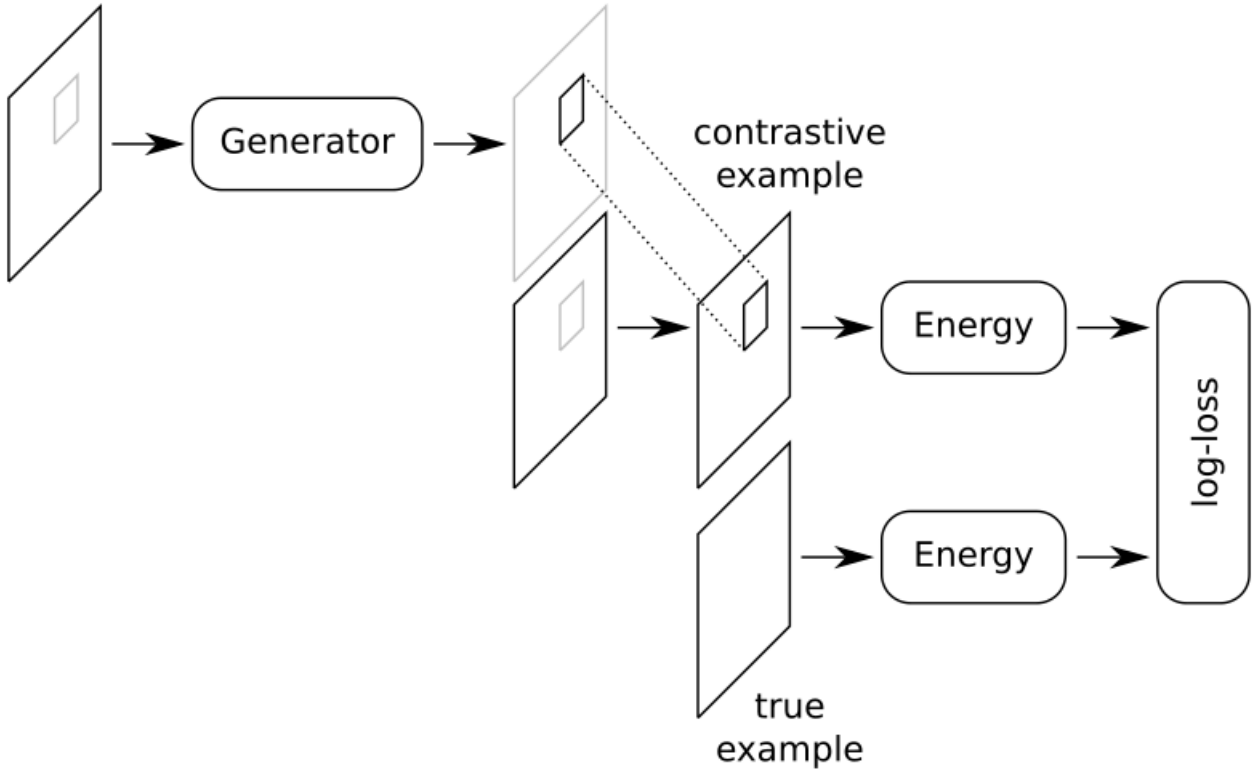
We now consider the energy-based learning framework where we learn an energy function to discriminate between correct and incorrect reconstructions.

In [7]:
```
torch.manual_seed(0)
enet = nn.Sequential(
    nn.Linear(784,256),nn.Hardtanh(),
    nn.Linear(256,256),nn.Hardtanh(),
    nn.Linear(256,1),
)
```

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(784,256),nn.Hardtanh(),
    nn.Linear(256,256),nn.Hardtanh(),
    nn.Linear(256,784),nn.Hardtanh()
)
```

The whole architecture is depicted in the diagram below:



The two networks are then jointly optimized. 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 = 10000
mb = 100

optimizer = optim.SGD(list(enet.parameters())+list(gnet.parameters()), lr=0.05)

for epoch in numpy.arange(100):

    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 solution
        err = solution.err(x,m,z,gnet,enet)
        # ------------------------------

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

    if epoch%10==0: print(epoch,err)
```

```
/home/gregoire/.local/lib/python3.8/site-packages/torch/autograd/__init__.py:130: Use
rWarning: CUDA initialization: The NVIDIA driver on your system is too old (found ver
sion 10010). Please update your GPU driver by downloading and installing a new versio
n from the URL: http://www.nvidia.com/Download/index.aspx Alternatively, go to: http
s://pytorch.org to install a PyTorch version that has been compiled with your version
of the CUDA driver. (Triggered internally at  /pytorch/c10/cuda/CUDAFunctions.cpp:10
0.)
  Variable._execution_engine.run_backward(

0 tensor(0.6744, grad_fn=<MeanBackward0>)
10 tensor(0.1035, grad_fn=<MeanBackward0>)
20 tensor(0.2146, grad_fn=<MeanBackward0>)
30 tensor(0.3614, grad_fn=<MeanBackward0>)
40 tensor(0.3134, grad_fn=<MeanBackward0>)
50 tensor(0.3515, grad_fn=<MeanBackward0>)
60 tensor(0.4389, grad_fn=<MeanBackward0>)
70 tensor(0.3787, grad_fn=<MeanBackward0>)
80 tensor(0.4541, grad_fn=<MeanBackward0>)
90 tensor(0.4365, grad_fn=<MeanBackward0>)
```

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.

Exercise Sheet 10

1 Mixture Density Networks

a) $$\frac{\partial E^q}{\partial \alpha_i} = \frac{\partial}{\partial \alpha_i} - \log\left\{\sum_{i=1}^{n_k} \alpha_i(x^q)\,\phi_i(t^q|x^q)\right\}$$

$$= -\frac{\phi_i}{\sum \alpha_i \phi_i} = -\frac{\alpha_i}{\alpha_i}\frac{\phi_i}{\sum \alpha_i \phi_i} = -\frac{\pi_i}{\alpha_i}$$

$$\frac{\partial E^q}{\partial \mu_{ik}} = \frac{\partial E^q}{\partial \phi_i}\left(\frac{\partial \phi_i}{\partial \mu_{ik}}\right) = \frac{\alpha_i}{-\sum \alpha_i \phi_i} \cdot \phi_i\left(-\frac{\mu_{ik}(x) - t_k}{\sigma_i(x^2)}\right)$$

$$= \pi_i\left(\frac{\mu_i(x) - t_k}{\sigma_i(x^2)}\right)$$

b) $$\frac{\partial E^q}{\partial z_i^\alpha} = \frac{\partial E^q}{\partial \alpha_j}\cdot\frac{\partial \alpha_j}{\partial z_i^\alpha} = \sum_j \frac{\partial E^q}{\partial \alpha_j}\cdot\frac{\partial \alpha_j}{\partial z_i^\alpha}$$

$\mu = e^{z_i^\alpha}$

$v = \sum_i e^{z_i^\alpha}$

$\left(\frac{u}{v}\right)' = \frac{uv' - vu'}{v^2}$

$$f' = \frac{\sum_j e^{z_k^\alpha}\cdot \mathbb{I}(i=k)\, e^{z_i^\alpha} - e^{z_i^\alpha} e^{z_j^\alpha}}{\left(\sum_j e^{z_k^\alpha}\right)^2} = \mathbb{I}(i=k)\,\alpha_i - \alpha_i\alpha_j$$

$$\frac{\partial E^q}{\partial z_i^\alpha} = \sum_j -\frac{\pi_j}{\alpha_j}\left(\mathbb{I}(i=j)\alpha_i - \alpha_i\alpha_j\right) = -\pi_i + \sum_j \pi_j \alpha_i$$

$$= -\pi_i + \alpha_i = \alpha_i - \pi_i$$

2 Conditional RBM

a) p i) $p(h_k = 1 \mid x, y) = p(h_k = 1, x, y) / p(x, y)$

$$= \frac{\sum_{h_k} p(h_k = 1, h_{-k}, x, y)}{\sum_{q \in \{0,1\}} \sum_{h_{-k}} p(h_k = q, h_{-k}, x, y)}$$

$$= \frac{\sum_{h_{-k}} \frac{1}{Z} e^{x^T W_{:,k} h_k + y^T U_{:,k} h_k - E(x, y, h_{-k})}}{\sum_{q \in \{0,1\}} \sum_{h_{-k}} \frac{1}{Z} e^{x^T W_{:,k} q + y^T U_{:,k} q - E(x, y, h_{-k})}}$$

$$= \frac{e^{x^T W_{:,k} + y^T U_{:,k}}}{1 + e^{x^T W_{:,k} + y^T U_{:,k}}} = \text{sigm}(x^T W_{:,k} + y^T U_{:,k})$$

ii) $p(y_j = 1 \mid h, x) = \text{sigm}(U_{j,:}^T h) = \frac{p(y_j, h, x)}{p(h, x)}$

$$= \frac{\sum_{y_{-j}} p(y_j = 1, y_{-j}, h, x)}{\sum_{q \in \{0,1\}} \sum_{y_{-j}} p(y_j = q, y_{-j}, h, x)} = \frac{e^{U_{j,:}^T h}}{1 + e^{U_{j,:}^T h}} = \text{sigm}(U_{j,:}^T h)$$

b) $p(x, y) = \sum_h p(x, y, h) = \sum_h \frac{1}{Z} e^{-E(x, y, h)}$

$$= \sum_{h_1} \frac{1}{Z} \prod_{h_2} e^{-E(x, y, h_k)} = \frac{1}{Z} \prod_{h_k} \left(1 + e^{-E(x, y, h_k = 1)}\right)$$

$$= \frac{1}{Z} e^{\sum_h \log\left(1 + e^{-E(x, y, h_k = 1)}\right)} = \frac{1}{Z} e^{-F(x, y)}$$