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 4

Exercise 1: Sparse Coding (20 + 20 P)

Let $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N \in \mathbb{R}^d$ be a dataset of N examples. Let $\boldsymbol{s}_i \in \mathbb{R}^h$ be the source associated to example $\boldsymbol{x}_i$, and W be a matrix of size $d \times h$ that linearly reconstructs the examples from the sources. We wish to minimize the objective:

$$J = \underbrace{\frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{x}_i - W\boldsymbol{s}_i\|^2}_{\text{reconstruction}} + \lambda \cdot \underbrace{\frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{s}_i\|_1}_{\text{sparsity}} + \eta \cdot \underbrace{\|W\|_F^2}_{\text{regularization}}$$

with respect to the weights W and the sources $\boldsymbol{s}_1, \dots, \boldsymbol{s}_N$. The objective consists of three terms: The reconstruction term is the standard mean square error, the sparsity term consists of a standard L_1 penalty on the sources, and the last regularization term prevents the sparsity term from becoming ineffective.

(a) *Show* that for fixed sources, the optimal matrix W is given in closed form as:

$$W = \Sigma_{XS}\big(\Sigma_{SS} + \eta I\big)^{-1}$$

where

$$\Sigma_{XS} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i \boldsymbol{s}_i^\top \qquad \text{and} \qquad \Sigma_{SS} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{s}_i \boldsymbol{s}_i^\top.$$

(b) We now consider the optimization of sources. Due to the 1-norm in the sparsity term, we cannot find a closed form solution. However, we consider a local relaxation of the optimization problem where the 1-norm of the sparsity term is linearized as

$$\|\boldsymbol{s}_i\|_1 = \boldsymbol{q}_i^\top \boldsymbol{s}_i$$

with $\boldsymbol{q}_i \in \{-1, 0, 1\}^d$ a constant vector. This relaxation makes the objective function quadratic with $\boldsymbol{s}_i$.

Show that under this local relaxation, the solution of the optimization problem is given in closed form as:

$$\boldsymbol{s}_i = \big(W^\top W\big)^{-1}\big(W^\top \boldsymbol{x}_i - \lambda \cdot \boldsymbol{q}_i/2\big)$$

Although this solution is not the true minimum of J (e.g. it is not sparse), it can serve as the end-point of some line-search method for finding good source vectors $\boldsymbol{s}_i$.

Exercise 2: Auto-Encoders (20 P)

In this exercise, we would like to show an equivalence between linear autoencoders with tied weights (same parameters for the encoder and decoder) and PCA. We consider the special case of an autoencoder with a single hidden unit. In that case, the autoencoder consists of the two layers:

$$s_i = \boldsymbol{w}^\top \boldsymbol{x}_i \qquad \text{(encoder)}$$
$$\hat{\boldsymbol{x}}_i = \boldsymbol{w} \cdot s_i \qquad \text{(decoder)}$$

where $\boldsymbol{w} \in \mathbb{R}^d$. We consider a dataset $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ assumed to be centered (i.e. $\sum_i \boldsymbol{x}_i = \boldsymbol{0}$), and we define the objective that we would like to mininize to be the mean square error between the data and the reconstruction:

$$J(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i\|^2$$

Furthermore, to make the objective closer to PCA, we can always rewrite the weight vector as $\boldsymbol{w} = \alpha \boldsymbol{u}$ where $\boldsymbol{u}$ is a unit vector (of norm 1) and α is some positive scalar, and search instead for the optimal parameters $\boldsymbol{u}$ and α.

(a) *Show* that the optimization problem can be equally rewritten as

$$\arg\min_{\alpha,\boldsymbol{u}} \; J(\boldsymbol{w}) \quad = \quad \arg\max_{\alpha,\boldsymbol{u}} \; \boldsymbol{u}^\top S \boldsymbol{u}$$

where $S = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i \boldsymbol{x}_i^\top$, which is a common formulation of PCA.

Exercise 3: Programming (40 P)

Download the programming files on ISIS and follow the instructions.

Implementing an Autoencoder

In this exercise, we would like to train on a popular face dataset, a sparse auto-encoder. We consider the simple two-layer autoencoder network:

$$\begin{aligned} \boldsymbol{z}_i &= \max(0, V\boldsymbol{x}_i + \boldsymbol{b}) && \text{(layer 1)} \\ \hat{\boldsymbol{x}}_i &= W\boldsymbol{z}_i + \boldsymbol{a} && \text{(layer 2)} \end{aligned}$$

where W, V are matrices of parameters of the encoder and the decoder, and $\boldsymbol{b}, \boldsymbol{a}$ are additional bias parameters. We seek to maximize the objective:

$$\min_W \; \underbrace{\frac{1}{N}\sum_{i=1}^{N}\|\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i\|^2}_{\text{reconstruction}} + \lambda \cdot \underbrace{\frac{1}{N}\sum_{i=1}^{N}\|\boldsymbol{z}_i\|_1}_{\text{sparsity}} + \epsilon \cdot \underbrace{\sum_{j=1}^{h}\Big(\frac{1}{N}\sum_{i=1}^{N}[\boldsymbol{z}_i]_j\Big)^{-1}}_{\text{"entropy"}} + \eta \cdot \underbrace{\|W\|_F^2}_{\text{regularization}}$$

The objective is composed of four terms: The reconstruction term is the standard mean square error between the data points and their reconstructions. The sparsity term applies a l1-norm to drive activation to zero in the representation. The "entropy" term that ensures that at least a few examples activate each source dimension. The regularization term ensures that the sparsity term remains effective.

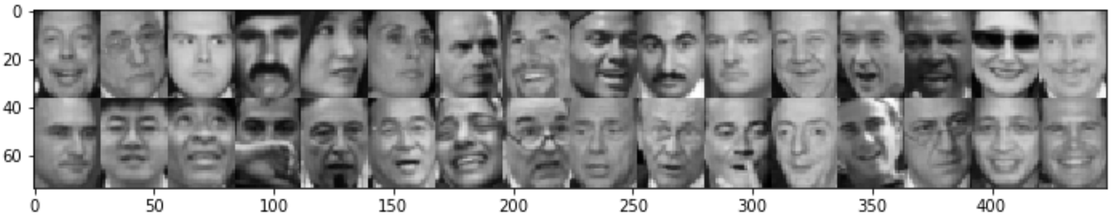
Loading the dataset

We first load the Labeled Faces in the Wild (LFW) dataset. The LFW is a popular face recognition dataset which is readily available from scikit learn. When loading the dataset, we specify some image downscaling in order to limit the computation resources. The following code visualizes a few images that we have extracted from the LFW dataset.

```
In [1]: import sklearn,sklearn.datasets
        import matplotlib
        %matplotlib inline
        from matplotlib import pyplot as plt

        data = sklearn.datasets.fetch_lfw_people(resize=0.3)['images']

        plt.figure(figsize=(12,2.5))
        plt.imshow(data[:32].reshape(2,16,37,28).transpose(0,2,1,3).reshape(2*37,16*28),cmap=
        'gray')
        plt.show()
```



Implementing the autoencoder (20 P)

We now would like to train an autoencoder on this data. As a first step, we standardize the data, which is a usual step before training a ML model. (Note that contrarily to other component analyses such as ICA, the data does not need to be whitened.)

```
In [2]: X = data.reshape(len(data),-1)
        X = X - X.mean(axis=0)
        X = X / X.std()
```

To learn the autoencoder, we need to optimize the objective function above. This can be done using by gradient descent, or some enhanced gradient-based optimizer such as Adam. Because a manual computation of the gradients can be difficult and error-prone, we will make use of automatic differentiation readily provided by the PyTorch software. PyTorch uses its own structures for storing the data and the model parameters. (You can consult the tutorials at https://pytorch.org/tutorials/ (https://pytorch.org/tutorials/) to learn the basics.)

We first convert the data into a PyTorch tensor.

In [3]:
```
import torch

X  = torch.FloatTensor(X)
```

Recall that the four terms that compose the objective function are given by:

$$\text{rec} = \frac{1}{N}\sum_{i=1}^{N}\|\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i\|^2 \qquad \text{spa} = \Big(\frac{1}{N}\sum_{i=1}^{N}\|\boldsymbol{z}_i\|_1\Big)$$

$$\text{ent} = \sum_{j=1}^{h}\Big(\frac{1}{N}\sum_{i=1}^{N}[\boldsymbol{z}_i]_j\Big)^{-1} \qquad \text{reg} = \|W\|_F^2$$

Task:

- **Create the function `get_objective_terms` that computes these terms.**

The function receives as input:

- A `FloatTensor` `X` of size $m \times d$ containing a data minibatch of m examples.
- A `FloatTensor` `V` of size $d \times h$ containing the weights of the encoder.
- A `FloatTensor` `W` of size $h \times d$ containing the weights of the decoder.
- A `FloatTensor` `b` of size h containing the bias of the encoder.
- A `FloatTensor` `a` of size d containing the bias of the decoder.

In your function, the parameter ϵ can be hardcoded to 0.01. The function should return the four terms (`rec` , `spa` , `ent` , `reg`) of the objective. (These terms will be merged later on in a single objective function.) While implementing the `get_objective_terms` function, make sure to use PyTorch functions so that the gradient information necessary for automatic differentiation is retained. For example, converting arrays to numpy will not work as this will remove the gradient information.

In [4]:
```
def get_objective_terms(X,V,W,b,a):

    # ------------------------------------------------------
    # TODO: replace by your code
    # ------------------------------------------------------
    import solution
    rec,spa,ent,reg = solution.get_objective_terms(X,V,W,b,a)
    # ------------------------------------------------------

    return rec,spa,ent,reg
```

Training the autoencoder

Now that the terms of the objective function have been implemented, the model can be trained to minimize the objective. The code below calls the function `get_objective_terms` repeatedly (once per iteration). Automatic differentiation is used to compute the gradient, and we use Adam (a state-of-the-art optimizer for neural networks) to optimize the parameters. The number of units in the representation is hard-coded to $h = 400$, and we use the parameter $\eta = 1$ for the regularizer.

In [5]:

```
import torch.optim
import torch.nn
import numpy

def train(X,lambd=0):

    d = X.shape[1]
    h = 400

    eps = 0.01 * lambd # hard-coded parameter
    eta = 1 * lambd    # hard-coded parameter

    V = torch.nn.Parameter(d**-.5*torch.randn([d,h]))
    W = torch.nn.Parameter(torch.zeros([h,d]))
    b = torch.nn.Parameter(torch.zeros([h]))
    a = torch.nn.Parameter(torch.zeros([d]))

    optimizer = torch.optim.Adam((V,W,b,a), lr=0.0001)

    print('%7s %8s %8s %8s %8s'%('nbit','rec','spa','ent','reg'))

    for i in range(0,10001):

        optimizer.zero_grad()

        x= X[numpy.random.permutation(len(X))[:100]]

        rec,spa,ent,reg = get_objective_terms(x,V,W,b,a)

        (rec + lambd*spa + eps*ent + eta*reg).backward()

        if i%1000 == 0: print('%7d %8.2f %8.2f %8.2f %8.2f'%(i,rec.data,spa.data,ent.
data,reg.data))

        optimizer.step()

    return V,W,b,a
```

Dense Autoencoder

We first train an autoencoder with parameter $\lambda = 0$, that is, a standard autoencoder without sparsity. The parameters of the learn autoencoder are stored in the variables `V`, `W`, `b`, `a`. Running the code may take a few minutes. You may temporarily reduce the number of iterations when testing your implementation.

In [7]:

```
V1,W1,b1,a1 = train(X,lambd=0)
```

```
   nbit      rec      spa      ent      reg
      0  1045.22   153.52  1099.49     0.00
   1000   128.94   458.57   356.76    48.74
   2000    86.48   456.00   366.91    75.44
   3000    63.23   438.00   377.32    94.49
   4000    62.44   450.24   381.65   110.32
   5000    54.00   450.36   370.90   124.42
   6000    54.19   466.40   362.62   137.63
   7000    46.84   436.62   389.91   150.74
   8000    40.91   441.50   391.96   164.41
   9000    36.63   423.16   407.59   179.50
  10000    30.43   425.86   407.38   196.64
```

We observe that the reconstruction term decreases strongly, indicating that the autoencoder becomes increasingly better at reconstructing the data. The sparsity term, however, increases, indicating that the standard autoencoder does not learn a sparse representation.

Sparse Autoencoder

We now would like to train a sparse autoencoder. For this, we set the sparsity parameter to $\lambda = 1$ and re-run the training procedure. We store the learned parameters in another set of variables.

```
In [8]: V2,W2,b2,a2 = train(X,lambd=1)
```

```
  nbit       rec       spa       ent       reg
     0   1086.80    157.35   1077.17      0.00
  1000    176.41    170.94   1313.81     79.80
  2000    155.85    159.27   1674.05     76.04
  3000    146.50    154.00   1950.38     72.20
  4000    142.15    155.02   2055.47     70.96
  5000    131.36    156.07   2003.10     70.35
  6000    136.57    157.66   1966.48     70.26
  7000    136.83    154.05   2090.28     69.95
  8000    125.52    147.72   2086.20     69.97
  9000    132.11    159.23   1948.78     69.84
 10000    127.77    156.40   2025.83     69.77
```

We observe that setting the parameter λ to a non-zero keeps the sparsity term low, which indicates that a sparser representation has been learned. In turn, we also loose a bit of reconstruction accuracy compared to the original autoencoder. This can be expected since the sparsity imposes additional constraints on the solution.

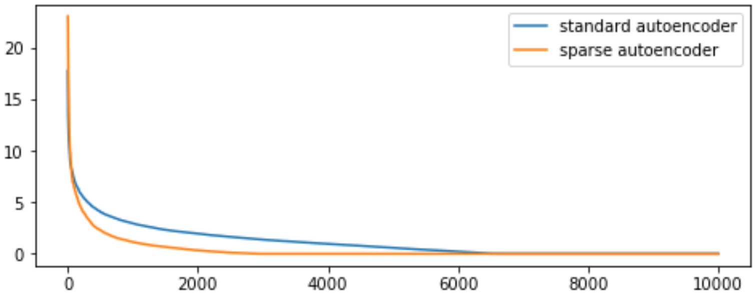
Analyzing autoencoder sparsity (10 P)

As a first analysis, we would like to verify how truly sparse the representation we have learned is.

Task:

- **Create a line plot where the two lines represents all activations (for the 25 first examples in the dataset) sorted from largest to smallest of the respective autoencoder models.**

```
In [9]: # ------------------------------------------------------
        # TODO: replace by your code
        # ------------------------------------------------------
        import solution
        solution.plot_sparsity(X[:25],V1,V2,b1,b2)
        # ------------------------------------------------------
```



We observe that the sparse autoencoder has a much larger proportion of weights that are close to zero. Hence, the our model has learned a sparse representation. One possible use of sparsity is to compress the data while retaining most of the information.

Inspecting the representation (10 P)

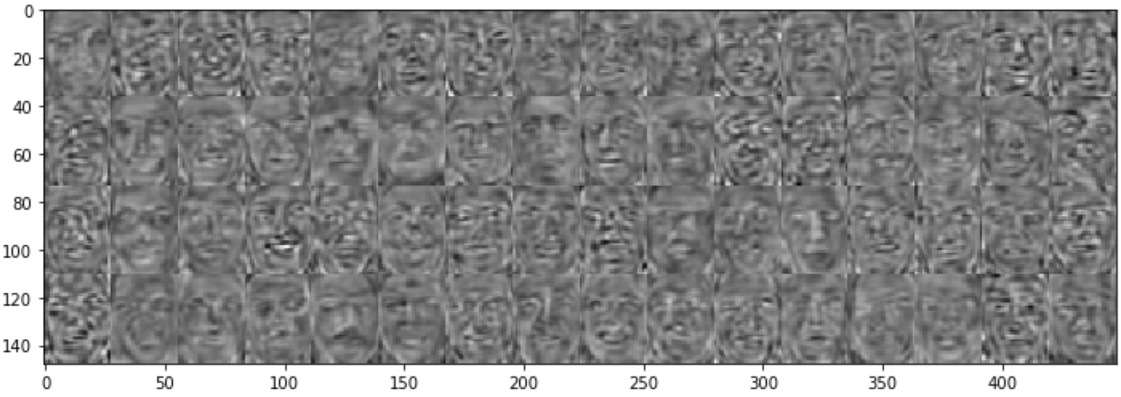
As a second analysis, we would like to visualize what the decoder has learned.

Task:

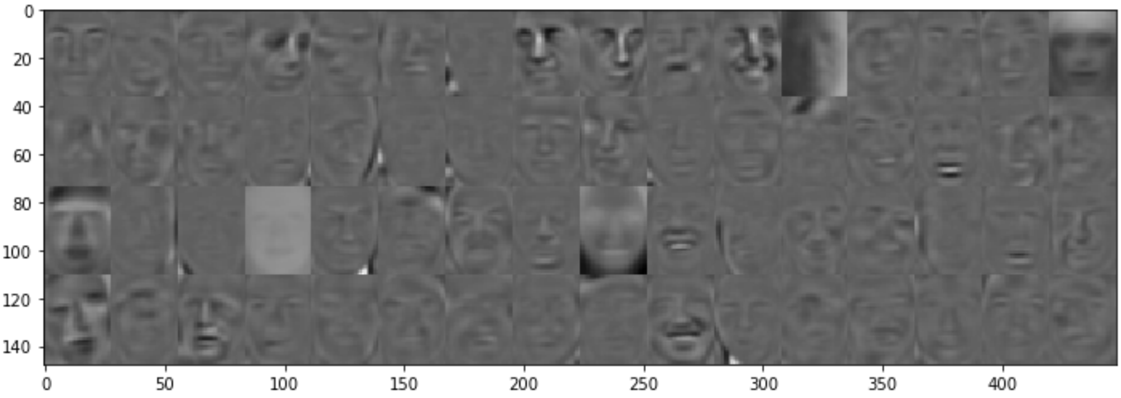
- **Write code that displays the first 64 decoding filters of the two models, in a similar mosaic format as it was used above to display some examples from the dataset.**

```
In [10]: # ---------------------------------------------------
         # TODO: replace by your code
         # ---------------------------------------------------
         import solution
         solution.view_decoder(W1,W2)
         # ---------------------------------------------------
```

```
decoder weights for the standard autoencoder
```



```
decoder weights for the sparse autoencoder
```



We observe that the filters of the standard autoencoder are quite difficult to interpret, whereas the sparse autoencoder produces filters with a stronger focus a single facial or background features such as the mouth, the nose, the bottom left/right corners, or the overall lighting condition. The features of the sparse autoencoder are also more interpretable for a human.

Exercise Sheet 4

1 Sparse Coding

a) J is convex → setting gradient to 0 will yield minimum

$$J = \frac{1}{N} \sum_{i=1}^{N} \|x_i - W s_i\|^2 + \lambda \cdot \frac{1}{N} \sum_{i=1}^{N} \|s_i\|_1 + \eta \|W\|_F^2$$

$$\frac{\partial J}{\partial W} = \frac{\partial}{\partial W} \frac{1}{N} \sum_{i=1}^{N} \|x_i - W s_i\|^2 + \eta \|W\|_F^2 \overset{!}{=} 0$$

$$\Leftrightarrow \frac{\partial}{\partial W} \frac{1}{N} \sum_{i=1}^{N} (x_i - W s_i)^T (x_i - W s_i) + \eta \|W\|_F^2 = 0$$

$$\Leftrightarrow \frac{1}{N} \sum_{i=1}^{N} -2(x_i - W s_i) s_i^T + 2\eta W = 0$$

$$\Leftrightarrow \frac{1}{N} \sum_{i=1}^{N} -(x_i - W s_i) s_i^T + \eta W = 0$$

$$\Leftrightarrow \frac{1}{N} \sum_{i=1}^{N} W(\eta \mathbb{1} + s_i s_i^T) = \frac{1}{N} \sum_{i=1}^{N} x_i s_i^T$$

$$\Leftrightarrow W(\Sigma_{ss} + \eta \mathbb{1}) = \Sigma_{xs} \quad \Leftrightarrow W = \Sigma_{xs} (\Sigma_{ss} + \eta \mathbb{1})^{-1}$$

b) $$\frac{\partial}{\partial s_i} J \Rightarrow \frac{\partial}{\partial s_i} \frac{1}{N} \sum_{i=1}^{N} \|x_i - W s_i\|^2 + \lambda \frac{1}{N} \sum_{i=1}^{N} q_i^T s_i \overset{!}{=} 0$$

$$\Leftrightarrow \frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial s_i} (x_i - W s_i)^T (x_i - W s_i) + \lambda \frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial s_i} q_i^T s_i = 0$$

$$\Leftrightarrow \frac{2}{N} W^T (x_i - W s_i) = \frac{\lambda}{N} q_i^T$$

$$\Leftrightarrow W^T x_i - W^T W s_i = \frac{\lambda}{2} q_i^T$$

$$\Leftrightarrow s_i = (W^T W)^{-1} (W^T x_i - \frac{\lambda}{2} q_i^T)$$

2 Auto-Encoders

a) $$\underset{\alpha,u}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} \| x_i - \hat{x}_i \|^2 = \underset{\alpha,u}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} \| x_i - w(u^T x_i) \|^2$$

$$= \underset{\alpha,u}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} \| x_i - \alpha u (\alpha u^T x_i) \|^2$$

$$= \underset{\alpha,u}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} \| x_i - \alpha^2 u u^T x_i \|^2$$

$$= \underset{\alpha,u}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} (x_i^T - \alpha^2 u u^T x_i)^T (x_i - \alpha^2 u u^T x_i)$$

$$= \underset{\alpha,u}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} (\alpha^4 - 2\alpha^2)(u^T x_i x_i^T u)$$

$$= \underset{\alpha,u}{\operatorname{argmin}} (\alpha^4 - 2\alpha^2)\left(u^T \left(\frac{1}{N}\sum_{i=1}^{N} x_i x_i^T\right) u\right)$$

$$= \underset{\alpha,u}{\operatorname{argmin}} (\alpha^4 - 2\alpha^2)(u^T S u) = \underset{\alpha,u}{\operatorname{argmax}}\ u^T S u$$