Machine Learning 2 Homework 4

Group: DLBSP

May 16, 2019

1 Task 1

(a)

$$\frac{dE}{dW} = 2\eta W + \sum_{i}^{N} 2(x_i - Ws_i)s_i^T \tag{1}$$

(b)

λ is a vector with the same length as s_i

$$\frac{dE}{ds_i} = \lambda + 2W^T(x_i - Ws_i) \tag{2}$$

Task 2

(a)

If we choose

$$g(r_i) = r_i - \tanh(r_i) \tag{3}$$

1.1

as a replacement function for s_i, we will still be able to achieve sort of sparsity by optimizing our objective function, since the terms $g(r_i)$ and $\|r_i\|^2$ complement each other in a way, that the former maps small values r_i to zeros and the latter constraints its values to be small. though not making the vector itself sparse.

(b)

The L2-norm is more suitable for gradient descent than the L1-norm. The L1-norm has a constant gradient, except for the minimum, where no gradient is defined. The gradient of the L2-norm declines with the proximity to the minimum. That means, by minimizing s_i directly one may end up in a poor local minimum, whereas in the reparameterized objective minimum is unique due to convexity. On the other hand use of $g(r)$ in objective function may lead to more complicated computations, especially after we take the partial derivative and expand the terms.

1.2

Task 3

(a)

$$E = \eta \|W\|_F^2 + \sum_{i=1}^{N} \|x_i - W g(V^T x_i)\|^2 + \lambda \|V^T x_i\|^2 \tag{4}$$

$$\frac{\partial E}{\partial V} = -2 \sum_{i=1}^{N} W x_i g'(V^T x_i)(x_i - W g(V^T x_i)) + 2\lambda V^T x_i \tag{5}$$

(b)

Direct optimization of s_i is computationally hard since the code (dictionary) must be overcomplete, that means it should be quite large. Autoencoders use backpropagation as a way of training which is relatively effective.

Denoising Autoencoders

In this exercise, we would like to train on the MNIST dataset a denoising auto-encoder made one layer of ReLU nonlinearity. Once the auto-encoder is implemented and trained, the reconstruction error, the sparsity of the solution and the parameters of the model will be analyzed.

The next few lines of code load the necessary Python modules for this exercise sheet, and read the MNIST dataset from scikit-learn. Since the autoencoders are unsupervised models, only the input MNIST data is necessary. The input data is normalized to have mean zero and standard deviation one.

In [1]:
```
# Load Libraries
import numpy
import numpy as np
import matplotlib
from matplotlib import pyplot as plt
%matplotlib inline
from sklearn.datasets import fetch_mldata

# Extract MNIST data
mnist = fetch_mldata('MNIST original')
X = mnist['data']
X = X - X.mean()
X = X / X.std()
```

```
/home/simon/miniconda3/envs/science/lib/python3.7/site-packages/sklearn/utils/deprecation.py:77: DeprecationWarning: Function fetch_mldata is deprecated; fetch_mldata was deprecated in version 0.20 and will be removed in version 0.22
  warnings.warn(msg, category=DeprecationWarning)
/home/simon/miniconda3/envs/science/lib/python3.7/site-packages/sklearn/utils/deprecation.py:77: DeprecationWarning: Function mldata_filename is deprecated; mldata_filename was deprecated in version 0.20 and will be removed in version 0.22
  warnings.warn(msg, category=DeprecationWarning)
```

The MNIST data consists of 70000 handwritten digits of 28x28 pixels, stored in a matrix `X` of size 70000x784. The following code displays in a mosaic format 100 randomly selected MNIST digits. This visualization is obtained by successive reshapings and transpositions of the input matrix.

In [2]:
```
R = numpy.random.randint(0,70000,[100])
plt.imshow(X[R].reshape(10,10,28,28).transpose(0,2,1,3).reshape(10*28,10*28),cmap='gray')
plt.show()
```

As it can be observed, the MNIST dataset is composed of handwritten digits from 0 to 9 with varying writing styles. Modeling these digits (e.g. predicting their labels, estimating their distribution, or removing noise added to them) is a non-trivial task for which one needs machine learning.

Implementing an auto-encoder (20 P)

We consider a simple two-layer denoising autoencoder where given an input `x` representing a digit to which noise has been added, it produces a reconstruction $\hat{x}$ that should be as close as possible to the original digit (i.e. a denoised version of `x`). The auto-encoder can be depicted as follow:

The code below defines a class `AutoEncoder` that provides the functionalities for storing the model parameters, performing the reconstruction, and training the model. It consists of four methods:

- The method `__init__(self)` initializes the parameters of the autoencoder to some random values of distributions selected heuristically.
- The method `forward(self,x)` applies to a noisy input `x` a forward pass on the autoencoder and outputs the denoised reconstruction of that input.
- The method `backward(self,dEdy)` receives as input the error gradient with respect to the autoencoder output `dEdy`, and outputs the error gradient with respect to the parameters of the network.
- The method `update(self,lr)` performs a gradient step with a certain learning rate `lr` based on the gradient computation obtained in the `backward` method.

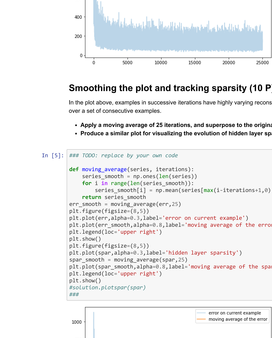
In [3]:
```
class AutoEncoder:

    def __init__(self,nbinput,nbhid):
        self.nbinput = nbinput
        self.nbhid = nbhid
        self.W = numpy.random.normal(0,0.01,[nbinput,nbhid])
        self.B = numpy.zeros([nbhid])+0.5
        self.V = numpy.random.normal(0,0.01,[nbhid,nbinput])
        self.C = numpy.zeros([nbinput])

    def forward(self,x):
        self.x = x
        self.z = numpy.maximum(0,numpy.dot(x,self.W)+self.B)
        y = numpy.dot(self.z,self.V)+self.C
        return y

    def backward(self,dEdy):
        ### TODO: Replace by your own code
        self.dEdV = np.outer(self.z, dEdy)
        self.dEdC = dEdy
        select = (1+sign(self.z))*0.5 # 1 if self.z > 0 else 0
        selectW = np.outer(np.ones(self.nbinput),select)
        self.dEdW = np.outer(self.x,self.V@dEdy) * selectW
        self.dEdB =  self.V@dEdy*select
        ###

    def update(self,lr):
        self.V = self.V - lr*self.dEdV
        self.W = self.W - lr*self.dEdW
        self.C = self.C - lr*self.dEdC
        self.B = self.B - lr*self.dEdB
```

The code below trains the denoising autoencoder for 25000 iterations using stochastic gradient descent, and a Gaussian noise of scale 1.0 on each dimension of the input digit. The objective to minimize is the squared Euclidean norm between the autoencoder output and the original digit (without noise).

- **Implement the method `backward(self,dEdy)` of the class `AutoEncoder`.**
- **Run the code below to train the autoencoder.**

Training an autoencoder can take a long time. You may want to reduce the number of training steps `nbit` during the development phase, and set it back to the original value afterwards. The training procedure keeps track of the error at each iteration on the current example and stores it in an array `err`. It also keeps track of the sparsity in the hidden layer (measured as the fraction of non-zero activations) and store these values in an array `spar`. Finally, the code plots the evolution of the error at each iteration.

In [4]:
```
ae = AutoEncoder(784,225)

err  = [] # error at each iteration
spar = [] # sparsity at each iteration

nbit = 25000

for i in range(1,nbit+1):

    # Choose a random data point
    x0 = X[numpy.random.randint(70000)]

    # Train the autoencoder with this data point
    x = x0+numpy.random.normal(0,1.0,x0.shape)
    y = ae.forward(x)
    ae.backward(y-x0)
    ae.update(0.0005)

    # Keep track of the reconstruction error and sparsity
    err  += [((y-x0)**2).sum()]
    spar += [(ae.z>0).sum()*1.0/len(ae.z)]

# Plot reconstruction error over time
plt.figure(figsize=(8,5))
plt.plot(err,alpha=0.3,label='error on current example')
plt.legend(loc='upper right')
plt.show()
```

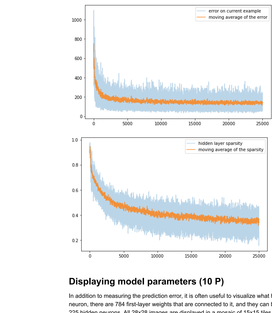


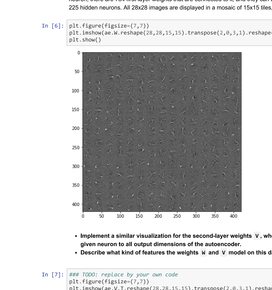
Smoothing the plot and tracking sparsity (10 P)

In the plot above, examples in successive iterations have highly varying reconstruction errors. Instead, we would like to measure some average of the error over a set of consecutive examples.

- **Apply a moving average of 25 iterations, and superpose to the original error curve the resulting smoothed error curve.**
- **Produce a similar plot for visualizing the evolution of hidden layer sparsity over time.**

In [5]:
```
### TODO: replace by your own code

def moving_average(series, iterations):
    series_smooth = np.ones(len(series))
    for i in range(len(series_smooth)):
        series_smooth[i] = np.mean(series[max(i-iterations+1,0):i+1])
    return series_smooth
err_smooth = moving_average(err,25)
plt.figure(figsize=(8,5))
plt.plot(err,alpha=0.3,label='error on current example')
plt.plot(err_smooth,alpha=0.8,label='moving average of the error')
plt.legend(loc='upper right')
plt.show()
plt.figure(figsize=(8,5))
plt.plot(spar,alpha=0.3,label='hidden layer sparsity')
spar_smooth = moving_average(spar,25)
plt.plot(spar_smooth,alpha=0.8,label='moving average of the sparsity')
plt.legend(loc='upper right')
plt.show()
#solution.plotspar(spar)
###
```

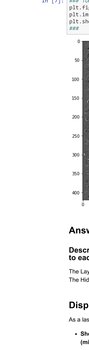




Displaying model parameters (10 P)

In addition to measuring the prediction error, it is often useful to visualize what hidden neurons are representing in the pixel space. For each hidden layer neuron, there are 784 first-layer weights that are connected to it, and they can be visualized as a pixel image. The following code shows these images for all 225 hidden neurons. All 28x28 images are displayed in a mosaic of 15x15 tiles, by application of a series of reshaping and transposition operations.

In [6]:
```
plt.figure(figsize=(7,7))
plt.imshow(ae.W.reshape(28,28,15,15).transpose(2,0,3,1).reshape(28*15,28*15),cmap='gray')
plt.show()
```



- **Implement a similar visualization for the second-layer weights `V`, where each image of the mosaic is composed of the parameter connecting a given neuron to all output dimensions of the autoencoder.**
- **Describe what kind of features the weights `W` and `V` model on this dataset, and how the two sets of features relate to each other.**

In [7]:
```
### TODO: replace by your own code
plt.figure(figsize=(7,7))
plt.imshow(ae.V.T.reshape(28,28,15,15).transpose(2,0,3,1).reshape(28*15,28*15),cmap='gray')
plt.show()
###
```

Answer to the Question :

Describe what kind of features the weights W and V model on this dataset, and how the two sets of features relate to each other.

The Layers contain common features of hand written numbers. The Features of W and V are very similar, W produces activations for an input, which lead to z. The Hidden Activation z gets translated back to number Features by V.

Displaying reconstructions of data points (10 P)

As a last analysis, we can verify that the noise was properly added to the input digit and that the autoencoder correctly performs denoising of the input digit.

- **Show for 10 randomly selected MNIST digits: the original digit (top), the same digit with additive noise that we give as input to the autoencoder (middle), and the reconstructed digit at the output of the autoencoder (bottom).**

In [8]:
```
### TODO: replace by your own code plot(x,oe)
R = numpy.random.randint(0,70000,[10])

X_noise = X[R]+numpy.random.normal(0,1.0,X[R].shape)
X_forward = ae.forward(X_noise)
plt.figure(figsize=(7,7))
per = (0,2,1)
canvas = np.zeros((28*3,28*10))
canvas[0:28] = X[R].reshape(10,28,28).transpose(per).reshape(28*10,28).T
canvas[28*1:28*2] = X_noise.reshape(10,28,28).transpose(per).reshape(28*10,28).T
canvas[56:84] = X_forward.reshape(10,28,28).transpose(per).reshape(28*10,28).T
plt.figure(figsize=(7,7))
plt.imshow(canvas,cmap='gray')
plt.show()
###
```

```
<figure size 504x504 with 0 Axes>
```

Index of comments

1.1 choose g(ri) = [ri1^2,...,rih^2]

1.2 - l2 norm is differentiable and convex, so SGD is suitable
- A positivity constraint disappears, which makes optimization easier

3.1 self.z !

3.2 W extract noise-resistant class features, V uses them to reconstruct class instances