

01.112 MACHINE LEARNING (2018)

HOMEWORK 3

LIANG ZHENG

The problems which require your solutions are marked (a), (b), and so on. Each problem is worth one point. This assignment has a total of 8 points.

1. MATRIX FACTORIZATION

Find the rank-one matrix $A \in \mathbb{R}^{3 \times 3}$ that is closest to the incomplete matrix

$$\begin{pmatrix} 0 & 1 & * \\ 1 & * & 1 \\ * & 1 & 2 \end{pmatrix},$$

i.e. A should minimize the squared error to the observed entries. You may find A by writing your own code, or using a suitable Python library, or even solving for it analytically. Let $A = UV^\top$ be the matrix factorization of A , where $U, V \in \mathbb{R}^{3 \times 1}$ are the factors.

- (a) Write down A .
- (b) Write down U .
- (c) Write down V .

2. SUPPORT VECTOR MACHINES

In this problem, you will use a non-linear support vector machine (SVM) with the radial basis kernel to classify points in the following data set.

<https://www.dropbox.com/s/wt45tvn9ig3o7vu/kernel.csv?dl=1>.

The data set contains a 100×3 table of numbers where the first column is the label vector Y and the remaining two columns form the feature matrix X . There are 100 samples in the training data. First, we import and plot the data.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
csv = 'https://www.dropbox.com/s/wt45tvn9ig3o7vu/kernel.csv?dl=1'
data = np.genfromtxt(csv, delimiter=',')
X = data[:,1:]
```

Date: October 9, 2018.

```
Y = data[:,0]
plt.scatter(X[:,0],X[:,1],c=Y)
plt.show()
```

Recall that the dual problem for the linear SVM is as follows.

$$\begin{aligned} \text{minimize} \quad & \ell(\alpha) := \sum_{(x,y)} \alpha_{x,y} - \frac{1}{2} \sum_{(x,y)} \sum_{(x',y')} \alpha_{x,y} \alpha_{x',y'} y y' K(x, x') \\ \text{subject to} \quad & \alpha_{x,y} \geq 0 \text{ for all training data } (x, y). \end{aligned}$$

where the kernel or similarity function $K(x, x') = x^\top x'$ is the dot product. To perform non-linear classification, we only need to change the kernel to something else. A popular choice is the radial basis kernel

$$K(x, x') = \exp(-\gamma \|x - x'\|^2), \quad \gamma = 0.5.$$

The classifier may be written as $h(x) = \text{sign } f(x)$ where $f(x)$ is the *decision function*

$$f(x) = \sum_{(x',y')} \alpha_{x',y'} y' K(x, x').$$

Fortunately, you are not required to implement an optimization algorithm to solve the above dual problem, because there are many libraries written specifically for this purpose.

- (a) Use the `sklearn.svm.SVC` module in the Python `scikit-learn` package to train a kernel support vector machine via the radial basis kernel. Remember to set `gamma` to 0.5 and `kernel` to `rbf` when initializing the object.
- (b) Evaluate the kernel SVM's decision function. You may use the `decision_function` method in `sklearn.svm.SVC`. You should write a function that takes coordinates `x1, x2` of a point $x \in \mathbb{R}^2$ and the trained `sklearn.svm.SVC` object `clf`, and return the \mathbb{R} -valued output of the decision function.

```
def decision(x1,x2,clf):
    x = np.array([x1,x2])
    ...
    return value
```

- (c) Use the following code to visualize the classifier and the data points.

```
vdecision = np.vectorize(decision,excluded=[2])
x1list = np.linspace(-8.0,8.0,100)
x2list = np.linspace(-8.0,8.0,100)
X1, X2 = np.meshgrid(x1list,x2list)
Z = vdecision(X1,X2,clf)
cp = plt.contourf(X1,X2,Z)
plt.colorbar(cp)
plt.scatter(X[:,0],X[:,1],c=Y,cmap='gray')
plt.show()
```

3. DEEP LEARNING

In this assignment, you will train a sparse autoencoder to discover features from natural images in an unsupervised fashion. In particular, you will implement the cost function and compute its gradient using backpropagation. First, download the following Python file and put it in the same directory as your Jupyter Notebook for this problem. The file provides useful helper functions for manipulating and visualizing the data.

<https://www.dropbox.com/s/e7opv0iybk4gr7a/utils.py?dl=1>.

Next, we load the dataset, which consists of 10,000 patches extracted from greyscale images of mountains, rivers and trees.

<https://www.dropbox.com/s/c9crd7fjk1523wh/images.npy?dl=1>.

Each patch is 8 pixels by 8 pixels, and the 64 pixel values are flattened into a vector whose entries are real numbers between zero and one. The vectors are normalized so that they all have the same mean and variance. The shape of the data matrix X is 10000×64 .

```
%matplotlib inline
import numpy as np
from numpy.linalg import norm
import matplotlib.pyplot as plt
from scipy.optimize import fmin_l_bfgs_b as minimize

from utils import normalize, tile_raster_images, sigmoid
from utils import ravelParameters, unravelParameters
from utils import initializeParameters
from utils import computeNumericalGradient

nV = 8*8      # number of visible units
nH = 25       # number of hidden units
dW = 0.0001   # weight decay term
sW = 3        # sparsity penalty term

npy = 'images.npy'
X = normalize(np.load(npy))
plt.imshow(tile_raster_images(X=X,
                              img_shape=(8,8),tile_shape=(5,5),
                              tile_spacing=(1,1)),cmap='gray')
plt.show()
```

- (a) We implement the function which computes the cost and the gradient of the sparse autoencoder. This function will be passed to an optimization engine, together with the `theta` vector that contains the current state of all the model parameters. The first step of the function is therefore to unpack the `theta` vector into W_1, W_2, b_1, b_2 . Some of the other steps are provided in the template below.

```
def sparseAutoencoderCost(theta,nV,nH,dW,sW,X):

    W1,W2,b1,b2 = unravelParameters(theta,nH,nV)
    n = X.shape[0]

    z2 =
    a2 = sigmoid(z2)
    z3 =
    a3 = sigmoid(z3)

    eps = a3-X
    loss =
    decay =

    # Compute sparsity terms and total cost
    rho = 0.01
    a2mean = np.mean(a2,axis=0).reshape(nH,1)
    kl = np.sum(rho*np.log(rho/a2mean)+\
                (1-rho)*np.log((1-rho)/(1-a2mean)))
    dkl = -rho/a2mean+(1-rho)/(1-a2mean)
    cost = loss+dW*decay+sW*kl

    d3 =
    d2 = (sW*dkl.T+np.dot(d3,W2.T))*a2*(1-a2)
    W1grad =
    W2grad =
    b1grad =
    b2grad =

    grad = ravelParameters(W1grad,W2grad,b1grad,b2grad)
    print(' .',end="")
    return cost,grad
```

Run the following code to check that your function computes.

```
theta = initializeParameters(nH,nV)
cost,grad = sparseAutoencoderCost(theta,nV,nH,dW,sW,X)
```

The shapes of the scalars, vectors and matrices are given below for your convenience. Here, the number of data points is n , and the shape of a scalar is indicated by $()$.

```
X.shape = (n, nV)

W1.shape = W1grad.shape = (nV, nH)
W2.shape = W2grad.shape = (nH, nV)
b1.shape = b1grad.shape = (nH,)
b2.shape = b2grad.shape = (nV,)

z2.shape = a2.shape = d2.shape = (n, nH)
z3.shape = a3.shape = d3.shape = (n, nV)

eps.shape = (n, nV)
loss.shape = ()
decay.shape = ()
```

Recall that given a data matrix $X \in \mathbb{R}^{n \times d}$, the cost and gradient are as follows.

$$\begin{aligned}
z^{(2)} &= XW^{(1)} + \mathbf{1}_n b^{(1)\top} & a^{(2)} &= \text{sigmoid}(z^{(2)}) \\
z^{(3)} &= a^{(2)}W^{(2)} + \mathbf{1}_n b^{(2)\top} & a^{(3)} &= \text{sigmoid}(z^{(3)}) \\
\varepsilon &= a^{(3)} - X \\
\text{loss} &= \|\varepsilon\|^2 / 2n \\
\text{decay} &= (\|W^{(1)}\|^2 + \|W^{(2)}\|^2) / 2 \\
\text{cost} &= \text{loss} + \lambda \text{decay} + \beta \text{k1} \\
\delta^{(3)} &= \varepsilon * a^{(3)} * (1 - a^{(3)}) \\
\delta^{(2)} &= (\beta \mathbf{1}_n \text{dk1}^\top + \delta^{(3)} W^{(2)\top}) * a^{(2)} * (1 - a^{(2)}) \\
\Delta W^{(1)} &= X^\top \delta^{(2)} / n + \lambda W^{(1)} & \Delta b^{(1)} &= \delta^{(2)\top} \mathbf{1}_n / n \\
\Delta W^{(2)} &= a^{(2)\top} \delta^{(3)} / n + \lambda W^{(2)} & \Delta b^{(2)} &= \delta^{(3)\top} \mathbf{1}_n / n
\end{aligned}$$

where $*$ is the element-wise product, λ is the decay weight, β is the sparsity weight, and $\mathbf{1}_n$ is the column vector with n ones.

- (b) Compare your backprop gradient in `sparseAutoencoderCost` with the gradient computed numerically from the cost. The relative difference should be less than 10^{-9} .

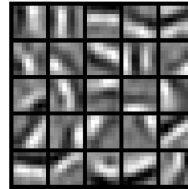
```
print('\nComparing numerical gradient with backprop gradient')
num_coords = 5
indices = np.random.choice(theta.size,num_coords,replace=False)
numgrad = computeNumericalGradient(lambda t:
    sparseAutoencoderCost(t,nV,nH,dW,sW,X)[0],theta,indices)
subnumgrad = numgrad[indices]
subgrad = grad[indices]
diff = norm(subnumgrad-subgrad)/norm(subnumgrad+subgrad)
print('\n',np.array([subnumgrad,subgrad]).T)
print('The relative difference is',diff)
```

- (c) Finally, run the following code to train the deep neural network and to visualize the features learnt by the autoencoder. The optimization may take several minutes.

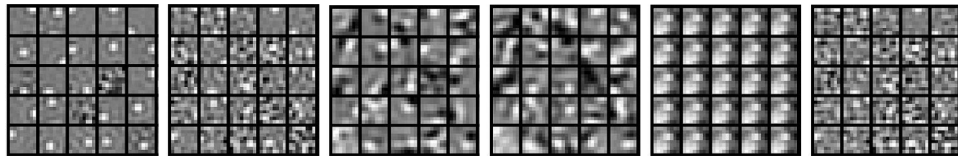
```
print('\nTraining neural network')
theta = initializeParameters(nH,nV)
opttheta,cost,messages = minimize(sparseAutoencoderCost,
    theta,fprime=None,maxiter=400,args=(nV,nH,dW,sW,X))

W1,W2,b1,b2 = unravelParameters(opttheta,nH,nV)
plt.imshow(tile_raster_images(X=W1.T,
    img_shape=(8,8),tile_shape=(5,5),
    tile_spacing=(1,1)),cmap='gray')
plt.show()
```

You should get the following visualization of the features, which represent edges.



If you get features that look like the ones below, there is some error in your code.



4. PROOF OF TWO EQUATIONS

- (a) In the slides of Lecture 8 (generative model), we discussed using MLE to calculate the probability of word w so that the probability that a document is generated is maximized. Please take a look at slides 13 and 14, and proof how θ_w is derived. Does it matter whether W denotes the collection of words in document \mathcal{S} or the collection of words in the dictionary?
- (b) In slide 15 of Lecture 8, we discussed how to estimate the parameters (μ and σ) so that the data samples are generated with a maximum probability under the spherical Gaussian assumption. Please demonstrate how μ and σ are derived based on MLE.