Project 3: PCA, Softmax Regression and NN

Due May 16 by 11:59pm **Points** 100 **Submitting** a file upload
File Types py, txt, and pdf **Available** Apr 23 at 2pm - May 19 at 1am 25 days

This assignment was locked May 19 at 1am.

Project 3: PCA, Softmax Regression and NN

In this project, we will explore dimensionality reduction (PCA), softmax regression and neural networks. The project files are available for download in the p3 folder on the course Files page.

Files to turn in:

```
dr.py           Implementation of PCA
softmax.py      Implementation of softmax regression
nn.py           Implementation of fully connected neural network
partners.txt    Lists the full names of all members in your team.
writeup.pdf     Answers all the written questions in this assignment
Files you created for Extra-Credits
```

You will be using helper functions in the following .py files and datasets:

```
util.py         Helper functions for PCA
datasets.py     Helper functions for PCA
utils.py        Helper functions for Softmax Regression and NN
digits          Toy dataset for PCA
data/*          Training and dev data for SR and NN
```

Please **do not** change the file names listed above. Submit partners.txt even if you do the project alone. **Only one member** in a team (even if it is a cross-section team) is supposed to submit the project.

AutogradingPlease **do not** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's output -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work. Please **do not** import (potentially unsafe) system-related modules such as sys, exec, eval... Otherwise the autograder will assign a zero point without grading.

Part 1 PCA *[30%]*

Files you might want to look at for PCA:

```
datasets.py     Some simple toy data sets
digits         Digits data
util.py         Utility functions, plotting, etc.
```

(https://github.com/hal3/ciml/tree/master/projects/p3#pca-30)

Our first tasks are to implement PCA. If implemented correctly, these should be 5-line functions (plus the supporting code I've provided): just be sure to use numpy's eigenvalue computation code. Implement PCA in the function `pca` in `dr.py`.

The pseudo-code in Algorithm 37 in CIML demonstrates the role of covariance matrix in PCA. However, the implementation of covariance matrix in practice requires much more concerns. One of them is to decide whether we require an unbiased estimation of the covariance matrix, i.e. normalize D by (N-1) instead of N (biased). Even the popular packages, such as matlab and sklearn, differ in the implementation. To make things easy, we'll require the submitted code to implement an unbiased version. More references: **PCA tutorial (http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf)** , **Sample mean and covariance (https://en.wikipedia.org/wiki/Sample_mean_and_covariance)** . Using sklearn.decomposition.PCA or numpy.cov is prohibited. Read the comments in dr.py carefully.

Qpca1 ***(15%)*** **Implement PCA**

Our first test of PCA will be on Gaussian data with a known covariance matrix. First, let's generate some data and see how it looks, and see what the *sample covariance* is:

```
>>> from numpy import *
>>> from matplotlib.pyplot import *
>>> import util
>>> Si = util.sqrtm(array([[3,2],[2,4]]))
>>> x = dot(random.randn(1000,2), Si)
>>> plot(x[:,0], x[:,1], 'b.')
>>> show(False)
>>> dot(x.T,x) / real(x.shape[0]-1) # The sample covariance matrix. Random generated data caus
e result to vary
array([[ 3.01879339,  2.07256783],
       [ 2.07256783,  4.15089407]])
```

(Note: The reason we have to do a matrix square-root on the covariance is because Gaussians are transformed by standard deviations, not by covariances.)

Note that the sample covariance of the data is almost exactly the true covariance of the data. If you run this with 100,000 data points (instead of 1000), you should get something even closer to `[[3,2],[2,4]]`.

Now, let's run PCA on this data. We basically know what should happen, but let's make sure it happens anyway (still, given the random nature, the numbers won't be exactly the same).

```
>>> import dr
>>> (P,Z,evals) = dr.pca(x, 2)
>>> Z
array([[-0.60270316, -0.79796548],
       [-0.79796548,  0.60270316]])
>>> evals
array([ 5.72199341,  1.45051781])
```

This tells us that the largest eigenvalue corresponds to the direction `[-0.603, -0.798]` and the second largest corresponds to the direction `[-0.798, 0.603]`. We can project the data onto the first eigenvalue and plot it in red, and the second eigenvalue in green. (Unfortunately we have to do some ugly reshaping to get dimensions to match up.)

```
>>> x0 = dot(dot(x, Z[:,0]).reshape(1000,1), Z[:,0].reshape(1,2))
>>> x1 = dot(dot(x, Z[:,1]).reshape(1000,1), Z[:,1].reshape(1,2))
>>> plot(x[:,0], x[:,1], 'b.', x0[:,0], x0[:,1], 'r.', x1[:,0], x1[:,1], 'g.')
>>> show(False)
```

Now, back to digits data. Let's look at some "eigendigits." (These numbers should be exact match)

```
>>> import datasets
>>> (X,Y) = datasets.loadDigits()
>>> (P,Z,evals) = dr.pca(X, 784)
>>> evals
array([ 0.05471459,  0.04324574,  0.03918324,  0.03075898,  0.02972407, .....
```

Eventually, the eigenvalues drop to zero (some may be negative due to floating point errors).

Qpca2 (10%): Plot the normalized eigenvalues (include the plot in your writeup). How many eigenvectors do you have to include before you've accounted for 90% of the variance? 95%? (Hint: see function `cumsum`.)

Now, let's plot the top 50 eigenvectors:

```
>>> util.drawDigits(Z.T[:50,:], arange(50))
>>> show(False)
```

Qpca3 (5%): Do these look like digits? Should they? Why or why not? (Include the plot in your write-up.) (Make sure you have got rid of the imaginary part in pca.)

Part II Softmax Regression *[45%]*

Files to edit/turn in for this part

```
softmax.py
writeup.pdf
```

(https://github.com/hal3/ciml/tree/master/projects/p3#pca-30)

The goal of this part of the project is to implement Softmax Regression in order to classify the MNIST digit dataset. Softmax Regression is essentially a two-layer neural network where the output layer applies the Softmax cost function, a multiclass generalization of the logistic cost function.

In logistic regression, we have a hypothesis function of the form

$P[y=1] = \frac{1}{1+e^{-\vec{w}\cdot\vec{x}}}$

where $\vec{w}$ is our weight vector. Like the hyperbolic tangent function, the logistic function is also a sigmoid function with the characteristic 's'-like shape, though it has a range of (0, 1) instead of (-1, 1). Note that this is
technically not a classifier since it returns probabilities instead of a predicted class, but it's easy to turn it into a classifier by simply choosing the class with the highest probability.

Since logistic regression is used for binary classification, it is easy to see that:

$$\begin{aligned} P[y=1] &= \frac{1}{1+e^{-\vec{w}\cdot\vec{x}}} \\ &= \frac{e^{\vec{w}\cdot\vec{x}}}{e^{\vec{w}\cdot\vec{x}}+1} \\ &= \frac{e^{\vec{w}\cdot\vec{x}}}{e^{\vec{w}\cdot\vec{x}}+e^{\vec{0}\cdot\vec{x}}} \end{aligned}$$

Similarly,

$$P[y=0] = 1 - \frac{1}{1+e^{-\vec{w}\cdot\vec{x}}}$$
$$= \frac{e^{\vec{w}\cdot\vec{x}}+1}{e^{\vec{w}\cdot\vec{x}}+1} - \frac{e^{\vec{w}\cdot\vec{x}}}{e^{\vec{w}\cdot\vec{x}}+1}$$
$$= \frac{e^{\vec{0}\cdot\vec{x}}}{e^{\vec{w}\cdot\vec{x}}+e^{\vec{0}\cdot\vec{x}}}$$

From this form it appears that we can assign the vector $\overrightarrow{w_1} = \vec{w}$ as the weight vector for class 1 and $\overrightarrow{w_0} = \vec{0}$ as the weight vector for class 0. Our probability formulas are now unified into one equation:

$$P[y=i] = \frac{e^{\overrightarrow{w_i}\cdot\vec{x}}}{\sum_j e^{\overrightarrow{w_j}\cdot\vec{x}}}$$

This immediately motivates generalization to classification with more than 2 classes. By assigning a separate weight vector $\overrightarrow{w_i}$ to each class, for each example $\vec{x}$ we can predict the probability that it is class i, and again we can classify by choosing the most probable class. A more compact way of representing the values $\overrightarrow{w_i}\cdot\vec{x}$ is $W\vec{x}$ where each row i of W is $\overrightarrow{w_i}$. We can also represent a dataset $\{\overrightarrow{x_i}\}$ with a matrix X where each column is
a single example.

Qsr1 ***(10%)***
(1) Show that the probabilities sum to 1.
(2) What are the dimensions of W? X? WX?

We can also train on this model with an appropriate loss function. The Softmax loss function is given by

$$L(W) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y_i = k\right\} \log \frac{e^{\overrightarrow{w_k}\cdot\overrightarrow{x_i}}}{\sum_{j=1}^{K} e^{\overrightarrow{w_j}\cdot\overrightarrow{x_i}}}\right]$$

where m is the number of examples, k is the number of classes, and $1\left\{y_i = k\right\}$is an indicator variable that equals 1 when the statement inside the brackets is true, and 0 otherwise. The gradient (which you will not derive) is given by:

$$\nabla_{\overrightarrow{w_k}} L(W) = -\sum_{i=1}^{m}\left[\overrightarrow{x_i}\left(1\{y_i = k\} - P[y_i = k]\right)\right]$$

Note that the indicator and the probabilities can be represented as matrices, which makes the code for the loss and the gradient very simple. (See **here (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/)** for more details)

softmax.py contains a mostly-complete implementation of Softmax Regression. A code stub also has been provided in run_softmax.py. Once you correctly implement the incomplete portions of softmax.py, you will be able to run run_softmax.py in order to classify the MNIST digits.

Qsr2 ***(15%)***

(1) Complete the implementation of the cost function.
(2) Complete the implementation of the predict function.

Check your implementation by running:

```
>>> python run_softmax.py
```

The output should be:

```
RUNNING THE L-BFGS-B CODE

* * *

Machine precision = 2.220D-16
N = 7840 M = 10

At X0 0 variables are exactly at the bounds

At iterate 0 f= 2.30259D+00 |proj g|= 6.37317D-02

At iterate 1 f= 1.52910D+00 |proj g|= 6.91122D-02

At iterate 2 f= 7.72038D-01 |proj g|= 4.43378D-02

...

At iterate 401 f= 2.19686D-01 |proj g|= 2.52336D-04

At iterate 402 f= 2.19665D-01 |proj g|= 2.04576D-04

* * *

Tit = total number of iterations
Tnf = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip = number of BFGS updates skipped
Nact = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F = final function value
```

```
* * *

N Tit Tnf Tnint Skip Nact Projg F
7840 402 431 1 0 0 2.046D-04 2.197D-01
F = 0.21966482316858085

STOP: TOTAL NO. of ITERATIONS EXCEEDS LIMIT

Cauchy time 0.000E+00 seconds.
Subspace minimization time 0.000E+00 seconds.
Line search time 0.000E+00 seconds.

Total User time 0.000E+00 seconds.

Accuracy: 93.99%
```

Qsr3 ***(10%)***

In the cost function, we see the line

```
W_X = W_X - np.max(W_X)
```

This means that each entry is reduced by the largest entry in the matrix.
(1) Show that this does not affect the predicted probabilities.
(2) Why might this be an optimization over using W_X? Justify your answer.

Qsr4 ***(10%)***

Use the learningCurve function in runClassifier.py to plot the accuracy of the classifier as a function of the number of examples seen. Include the plot in your write-up. Do you observe any overfitting or underfitting? Discuss and expain what you observe.

Part III NN [25% and Extra 15%]

Files to edit/turn in.

```
nn.py
writeup.pdf
files created for the Q2 extra credits
```

In this part of the project, we'll implement a fully-connected neural network in general for the MNIST dataset. For **Qnn1** you will complete nn.py. For **Qnn2**, create your own files.

A code stub also has been provided in run_nn.py. Once you correctly implement the incomplete portions of nn.py, you will be able to run run_nn.py in order to classify the MNIST digits.

The dataset is included under ./data/. You will be using the following helper functions in "utils.py". In the following description, let K, N, d denote the number of classes, number of samples and number of features.

- Selected functions in "utils.py"

```
loadMNIST(image_file, label_file) #returns data matrix X with shape (d, N) and labels with sha
pe (N,)
onehot(labels) # encodes labels into one hot style with shape (K,N)
acc(pred_label, Y) # calculate the accuracy of prediction given ground truth Y, where pred_lab
el is with shape (N,), Y is with shape (N,K).
data_loader(X, Y=None, batch_size=64, shuffle=False) # Iterator that yield X,Y with shape(d, b
atch_size) and (K, batch_size).
```

Qnn1 (20% for Qnn1.1, 1.2, 1.3 and 5% for Qnn 1.4) Implement the NN

The scaffold has been built for you. Initialize the model and print the architecture with the following:

```
>>> from nn import NN, Relu, Linear, SquaredLoss
>>> from utils import data_loader, acc, save_plot, loadMNIST, onehot
>>> model = NN(Relu(), SquaredLoss(), hidden_layers=[128,128])
>>> model.print_model()
```

Two activation functions (Relu, Linear) and self.predict(X) have been implemented for you.

Qnn1.1 Implement squared loss cost functions **(TODO 0 & TODO 1)**
Assume $\bar{Y}$ is the output of the last layer before loss calculation (without activation), which is a K-by-N matrix. Y is the one hot encoded ground truth of the same shape. Implement the following loss function and its gradient (You need to calculate and implement the gradient of the loss function yourself) (Notice that the loss functions are normalized by batch_size N):
$L\left(Y,\bar{Y}\right) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|\bar{Y}_i - Y_i\|^2 = \frac{1}{2N}\|\bar{Y} - Y\|_{fro}^2$, where Y_i is the i-th column of Y.

Typically we would use cross entropy loss, the formula of which is provided for your reference (but you are only required to implement for squared loss):

$L_{CE}(Y,\bar{Y}) = \frac{1}{N}\sum_{i=1}^{N} NLL(Y_i,\bar{Y}_i)$, where

$$NLL(y,\bar{y}) = -\log(\sum_{j=1}^{K} y_j \frac{\exp(\bar{y}_j)}{\sum_{k=1}^{K}\exp(\bar{y}_k)}).$$

Qnn1.2. Compute the gradients **(TODO 2 & TODO 3)**
Implement the forward pass (TODO 2) and back propagation (TODO 3) for gradient calculation. Use "activation.activate" and "activation.backprop_grad" in your code so that your gradient

computation works for different choices of activation functions.
Do the following to see if the loss goes down.

```
>>> x_train, label_train = loadMNIST('data/train-images.idx3-ubyte', 'data/train-labels.idx1-u
byte')
>>> x_test, label_test = loadMNIST('data/t10k-images.idx3-ubyte', 'data/t10k-labels.idx1-ubyt
e')
>>> y_train = onehot(label_train)
>>> y_test = onehot(label_test)

>>> model = NN(Relu(), SquaredLoss(), hidden_layers=[128, 128], input_d=784, output_d=10)
>>> model.print_model()
>>> training_data, dev_data = {"X":x_train, "Y":y_train}, {"X":x_test, "Y":y_test}
>>> from run_nn import train_1pass
>>> model, plot_dict = train_1pass(model, training_data, dev_data, learning_rate=1e-2, batch_s
ize=64)
```

Qnn1.3 Run in epochs

An epoch is a full pass of the training data. Run run_nn.py.

```
>>> python3 run_nn.py
```

Report your final accuracy on the dev set. You can either use the default setting or tune the architecture (number of layers, size of layers and loss function) and hyperparameters (lr, batch_size, max_epoch).

Qnn1.4 **(No implementation needed for this question).** When initializing the weight matrix, in some cases it may be appropriate to initialize the entries as small random numbers rather than all zeros. Give one reason why this may be a good idea.

Qnn2 (Extra-Credit 15%) Try something new.

Choose one of the following directions (outside research may be required) for further exploration (Feel free to copy nn.py, utils.py and run_nn.py as a starting point. **Make sure** that your code for Qnn2 is separated from your code for Qnn1):

(1) Do dimension reduction with PCA. Try with different dimensions. Can you observe the trade-off in time and acc? Plot training time v.s. dimension, testing time v.s dimension and acc v.s. dimension. Visualize the principal components.

(2) Improve your results with ensemble methods. Describe your implementation. Can you observe improved performance compared with that of Q4? Why? (http://ciml.info/dl/v0_99/ciml-v0_99-ch13.pdf)

(3) Implement a new optimizer (By implementing a different self.update for the NN class). Compare with the original SGD optimizer. You can read about the optimizers in (http://ruder.io/deep-learning-optimization-2017/). Does this new method take less number of samples to converge? Does this new method take less time to converge?

Qnn2.1 Explain what you did and what you found. Comment the code so that it is easy to follow. Support your results with plots and numbers. Provide the implementation so we can replicate your results.