McMaster University, Physics 3G03 March 4 - March 18 2021
http://avenue.mcmaster.ca/

Homework 4 : PCA, Autoencoders, K-Means Clustering and fcNN

You should submit source a python notebook along with a pdf of the notebook for all exercises.

Due: Thursday March 18 2021, 11:30pm on avenue, with a grace period till the following Monday at 11:30pm

☛**Reading:** Parts of the first lectures are based on *The Hundred Page Machine Learning Book* by A. Burkov. Found at http://themlbook.com. Since then we have gone through Chapters 2,3,4,5,6,7,8 and parts of App A,B in *Machine Learning Refined*. An early version of *Machine Learning Refined* is available at https://github.com/jermwatt/machine_learning_refined Lately we have been looking at fully connected networks (fcNN) and have started on convolutions.

Exercise 1 *Final Project*

Describe in 1 paragraph (or longer) what your final project will be about. At the very minimum you can chose a dataset from http://archive.ics.uci.edu/ml/index.php or another repository and model and analyze it using some of the techniques used in the course. You're welcome to go beyond that and look at techniques such as natural language processing, style transfer in image processing, etc.. As long as it is something that is clearly aligned with neural networks and machine learning it should be fine. The final project will count for 20% of the mark for the course. The point is to perform some independent work on a subject that you find interesting. A write up is needed but we will limit the page count to something reasonable. More details later.

While it is natural to reuse code in many settings it will not be acceptable to simply submit a python notebook you have found on the internet. You should quote your sources and show your independent usage.

It's ok if you decide later on to switch to something else but you should start planning and thinking about your final project now.

Exercise 2 *MNIST Hand-Written Digits Classified using Python*

In this exercise we return to the hand written digits in the MNIST dataset that we classified in class using Keras+Tensorflow. However, here we shall try to classify them *directly* in python *without* using Keras+Tensoflow. However, it is convenient to import the dataset using Keras+Tensorflow so let's do that before switching completely to python.

```
import tensorflow as tf
from autograd import numpy as np
import pandas as pd
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Note the special version of numpy. In order to use our usual python perceptron routines let us transform the data:

```
train_labels.shape=(1,60000)
test_labels.shape=(1,10000)
train_images=(train_images.reshape(60000,784)).T
test_images=(test_images.reshape(10000,784)).T
```

We are now ready to use our usual multiclass perceptron:

```
# compute C linear combinations of input point, one per classifier
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# multiclass perceptron
def multiclass_perceptron(w,x,y,iter):
    # get subset of points
    x_p = x[:,iter]
    y_p = y[:,iter]

    # pre-compute predictions on all points
    all_evals = model(x_p,w)

    # compute maximum across data points
    a =  np.max(all_evals,axis = 0)

    # compute cost in compact form using numpy broadcasting
    b = all_evals[y_p.astype(int).flatten(),np.arange(np.size(y_p))]
    cost = np.sum(a - b)

    # return average
    return cost/float(np.size(y_p))
```

2.1 Implement (in python) a version of mini-batch gradient descent that takes a batch-size as input `def gradient_descent(g,w,x_train,y_train,alpha,max_its,batch_size):` For a given batch-size it should determine how many number of batches (`num_batch`) is needed for a full ephoch, then construct a batch by sequentially going through the samples in `num_batch` steps each time using a `batch_size` group of samples. Decide what to do if the `batch_size` is not a divisor of the number of samples. For each of the `max_its` number of epochs perform `num_batch` gradient steps. You should record the cost-history at each epoch. You can implement all this by modifying previous versions of gradient descent.

If you're ambitious, you can make things run faster by flattening `g`. As explained at the end of appendix B.10 in Watt et al you can use `g_flat, unflatten, w = flatten_func(g, w)`. You then have to be careful by using `g_flat` instead of `g`.

2.2 Use your mini-batch gradient descent with a number of epochs, `max_its=5`, and a random starting vector `w = 0.1*np.random.randn(N+1,C)` where `N,C` have their usual definition. Use a learning rate of $\alpha = 0.001$. Make 2 separate runs, first with a batchsize of 200 then with a batchsize that is equal to the number of samples (so called full batch). Plot the cost histories versus the number of epochs. Which batchsize performs the best ?

2.3 Use the best set of weights you obtained to make a histogram of the percentage of misclassified samples as a function of the digit $(0 \dots 9)$. Is there a digit that is more often misclassified ?

Exercise 3 *Autoencoder*

In this exercise we shall try to reproduce the results shown in Fig. 8.5 from example 8.3. We shall use the dataset `'2d_span_data_centered.csv'`.

3.1 Implement the model `def model(x,C):` and cost `def autoencoder(C):` for the autoencoder. Then use the usual gradient descent to determine a *single* spanning vector. You can use the following parameters for your starting point: `g = autoencoder; alpha_choice = 10**(-4); max_its = 1000; C = 0.1*np.random.randn(2,1); C = np.array([[-3.5],[3.5]])`. After optimization make the following 3 plots (Fig 8.5). The original data in the 2D plane along with the optimized spanning vector, the encoded data (in one dimension), and finally the decoded data in the 2D plane.

Exercise 4 *PCA*

In this exercise we shall try to reproduce the results shown in Fig. 8.7 from example 8.4. We shall use the dataset `'2d_span_data.csv'` along with the code on page 218-219.

4.1 Import the data set and use `def center(X)` to first center the data then `def compute_pcs(X,lam)` to compute the principal components. Finally, reproduce Fig 8.7.

Exercise 5 *K-Means Clustering*

In this exercise you shall implement K-Means clustering directly in python. We shall use a data set with 3 'blobs' of data points, `'blobs.dat'`.

5.1 Implement 2 functions `def update_assignments(data,centroids)` and `def update_centroids(data,old_centroids,assignments)` that updates the assignments of the data points to the centroids and the position of the centroids, respectively. Then pick 3 points and initialize 3 centroids with the coordinates of the 3 points. Use the initial positions of the centroids to perform 5 full sweeps through all the points, each time updating the assignments and then the position of the centroids.

5.2 By varying the number of centroids K from 1 throught 10, produce a scree plot for the data. Is there a clear indication of the correct value for K ?

Exercise 6 *Fraud Detection*

In this exercise we will study fraud detection of fake bank notes using the data set available at `https://archive.ics.uci.edu/ml/datasets/banknote+authentication`. The goal of this exercise is to use Keras+Tensorflow to first perform logistic regression with a single neuron, analyze the results and then use a fcNN (again in Keras+Tensorflow) to improve the results. In the data set the $P = 1372$ banknotes have been classified as fake or real. The notes have been scanned and a wavelet transform has been applied to each image. The resulting attributes of the dataset are the following:

1. variance of Wavelet Transformed image (continuous)
2. skewness of Wavelet Transformed image (continuous)
3. curtosis of Wavelet Transformed image (continuous)
4. entropy of image (continuous)
5. class (integer) (1 for fake 0 for real)

We can import the data directly from the archive site using:

```
import numpy as np
from urllib.request import urlopen
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00267/data_banknote_authentica
raw_data = urlopen(url)
dataset = np.loadtxt(raw_data, delimiter=",")
print(dataset.shape)
```

Here we want to plot the data using a simple 2D plot so we decide to only work with two features x_1: skewness of wavelet transformed image and x_2: entropy of wavelet transformed image. So, we only select these two attributes along with the classification.

```
X=dataset[:,[1,3]]
Y=dataset[:,4]
print(X.shape)
print(Y.shape)
```

6.1 Use matplotlib to make a 2D plot with the individual data points shown as symbols with different colors for the fake and real data points. Can you visually separate the two classes by a straight line ?

6.2 Use Keras+Tensorflow to perform a logistic regression on the data using a single neuron with sigmoid activation. This can be done using:

```
Dense(1, batch_input_shape=(None, 2),activation='sigmoid')
```

Note the use of `batch_input_shape=(None, 2)` the `None` is used later for the actual batch size, so this means that we're expecting a yet to be determined number of rows each with 2 columns corresponding to $x1$ and $x2$. Compile the model using stochastic gradient descent for the optimizer with a learning rate of 0.15 and 'binary_crossentropy' for the loss. Train the model over 400 epochs with a batchsize of 128. Use `history.history['accuracy']` and `history.history['loss']` to plot the accuracy and loss history. You can access the history by using `history = model.fit(..)`. Is the logistic regression working well ?

6.3 Now we want to visualize how things are working. Create linear arrays of 50 points along the $x1$ and $x2$ axis' using:

```
x1list = np.linspace(np.min(X[:,0])-2, np.max(X[:,0])+2, 50)
x2list = np.linspace(np.min(X[:,1])-2, np.max(X[:,1])+2, 50)
```

This defines a grid in the 2D plane. Use `model.predict` to evaluate the model's prediction on the grid. Note, that if you want to check a single point you need to have the shape correct. For instance you can use:

```
model.predict(np.reshape(np.array([0.0,0.1]),(1,2)))
```

You need this since the individual points are exptected to have shape (1,2) - a row-vector not a column vector. Furthermore, since you're using a sigmoid the model should predict numbers between 0.0 and 1.0. Then plot the results as a contour map using the matplotlib function `plt.contourf`. Then, as you did above, plot the actual data points on top of the contour map using color coding for the fake and real data points. Comment on your observations.

6.4 We now want to improve on the model by building a network with a hidden layer. First insert a dense layer of 8 neurons with sigmoid activation. Follow this layer with an output layer of 2 neurons with softmax activation. In this case the output can be interpreted as a probabilities for fake and real. Furthemore we convert to *one-hot* encoding for the labels so that we transform y=0 to (1,0) and y=1 to (0,1). This can be done using a keras utility in a simple way.

```
Y_c=to_categorical(Y,2)
```

Once this is done we can use `'categorical_crossentropy'` for the loss, the same learning rate of 0.15 along with stochastic gradient descent for compiling the model. We have to use `'categorical_crossentropy'` and not `'sparse_categorical_crossentropy'` since we have one-hot encoded y. Build, compile and finally fit this model using 400 epochs and a batch size of 128. As above, plot the accuracy and loss versus the epochs. Coment on your results.

6.5 In this case if we use `model.predict` we should get two numbers corresponding to probabilities for fake and real. As before, use the probability for *real* to make a 2D contour plot of the predictions of the model with the actual data points super imposed. Compare your results to what was obtained with a single neuron and comment on your observations.