CSC 578 Neural Networks and Deep Learning
Fall 2018/19

Homework #1: Intro to Neural Networks

Overview

Your task in this assignment is to make some modifications to the NNDL book code "network.py" and write a small application code. The objective of the assignment is to strengthen you understanding of the concepts and mathematics of neural networks, and to expose to an example of how neural networks are implemented.

The amount of code you write for this assignment is not much. However, reading and understanding the code written by somebody else requires you to not only learn the details of the code but also to deal with the technical intricacies of the language and libraries used. You may need to consult developer community sites such as Stackoverflow. But this assignment is not meant to be just a coding exercise; you are supposed to learn how the concept of neural networks is implemented in this code.

Deliverables:

You submit all of these. More instructions are found at the end of this page.

- Code files
- A video
- A documentation file **(*)**

There is also a chance for **Extra Credit**. It is a visualization task.

Specifics

The overall picture of your task is to:

- (1) make required modifications in the network definition code "NN578_network.py"; and
- (2) create a new file named "578hw1.py" (which imports NN578_network) and write the required application code.

The book code, along with a couple of files modified for this course, is available at this **Git code repository**. Visit there and download the repository in a zip file on your local computer. Or you can also use the **IBM Cognitive Class Labs**. This is a free development site, equipped with various Machine Learning tools including **Python 3, Jupyter Notebook and Tensorflow**. Also this video provides an easy intro to CCLabs (although the video is made for a different course at a different institution).

Your code must be compatible with **Python 3** (REQUIRED; 3.5 or above). Also for this assignment, you shouldn't need special libraries besides standard ones (e.g. random) or numpy (and possibly pandas if you so insist on using it).

NOTE: You may use **Jupyter Notebook** for the application code if you wish (in which case the file name will be "578hw1.ipynb"). If you do, you must **re-start the kernel** of the application code **every time** you make changes in the network code.

IMPORTANT DISCLAIMER:

- You must add your **name**, **course/section number** and the **assignment name** in a comment section at the top of BOTH FILES . Files without those information will be returned **ungraded**.
- If your code **fails to run** at any point for any reason whatsoever, it will be will be returned **ungraded** as well.

---

(1) Modifications to be made in the network code ("NN578_network.py")

1. Edit the function **evaluate()** (which is called after an epoch is complete) so that, in addition to accuracy, it computes the Mean Squared Error (MSE), Cross-entropy and log-likelihood.

   MSE is described in NNDL 1 (Eq. (6)). Cross-entropy is in NNDL3 (Eq. (57)). Log likelihood is in NNDL3 (Eq. (80)), but note that the formula is MISSING **1/n** in the beginning, which divides the sum by the number of instances in the dataset to give the average.

   Then the function should return those five results (correctcount, accuracy, MSE, Cross-entropy and log-likelihood) in a list.

   - As a hint, for MSE and Cross-entropy, you can look at the two function classes (QuadraticCost and CrossEntropyCost) in "578NN_network2.py".
   - Note that the function feedforward() returns the output layer's activation in an **Numpy ndarray of (n,1)** (e.g. [[0.57201899], [0.63172043], [0.26575742]], whose shape is (3,1)). So you probably want to transform/flatten it when you compare with the target output (y) (e.g. [0., 1., 0.], whose shape is (3,)).

   Note that NO PRINTING takes place in evaluate(). It only _returns_ five values.

2. Edit the function **SGD()** so that it does the following.
   a. Call evaluate() for **training_data**, at the end of every epoch, and print the returned results in the form below. It should also call evaluate() for test_data as well if it is passed in. See below for the formatting example. Note that test_data was not passed in, you omit the second line in the output for an epoch.

```
[Epoch 0] Training: MSE=aaaa, CrossEntropy=xxxx, LogLikelihood=yyyy, Correct: zzz/nn
          Test:     MSE=bbbb, CrossEntropy=xxxx, LogLikelihood=yyyy, Correct: zzz/nn
[Epoch 1] Training: MSE=aaaa, CrossEntropy=xxxx, LogLikelihood=yyyy, Correct: zzz/nn
          Test:     MSE=bbbb, CrossEntropy=xxxx, LogLikelihood=yyyy, Correct: zzz/nn
...
```

   Note that you call evaluate() only and **exactly ONCE** for a dataset. You compute measures above using the results returned from evaluate().

   b. Collect the performance results returned from evaluate() for all epochs, for training_data and test_data into individual lists separately, and return the two lists in a list (to the caller of the function). Note that, if test_data was not provided, the collected list for test_data will be an empty list.

   c. Add **Early Stopping**, which terminates the epoch loop prematurely if the classification accuracy for the training data became 100%. You stop the loop at the end of the epoch loop, after evaluate(test_data) is called and its results are printed.

3. Edit the function **backprop()** so that the local variable 'activations' is initially allocated with a structure which holds the activation value of ALL layers from the start, rather than the current code which starts with just the input layer (by "activations = [x]") and appends one layer at a time (by "activations.append(activation)").

   For example, if the network size was [4, 20, 3], you create an (Python) array/list consisting of Numpy ndarrays of shape (4,1), (20,3) and (3,1). Then during the forward-propagation, activation values of each layer are copied into the respective ndarray.

---

(2) Application code ("578hw1.py" or **"578hw1.ipynb" -- a start-up file is in the Git repository)**

The file will be a collection of code snippets that do various individual tasks (rather than a coherent program). You write code to do these tasks **in order**.

1. Check your implementation for Modification 1 (evaluate()), 2a (SGD printing) and 3 (activations in backprop()). Steps are:
   - **(*)** First add a line in the function **backprop()** that prints the values in the 'activations' array (after all values are assigned in).
   - Then in the application code,
     - load a saved network **"iris-423.dat"** (using the function load_network()).
     - define sample data instances (for training and testing; see below)
     - run the network for just one epoch, with eta = 1.0 (and mini_bath_size = 1)

   The code would be something like:

```
import NN578_network as network
import numpy as np
net4 = network.load_network("iris-423.dat")

# Create two data instances on the fly
inst1 = (np.array([5.7, 3, 4.2, 1.2]), np.array([0., 1., 0.]))
x1 = np.reshape(inst1[0], (4, 1))
y1 = np.reshape(inst1[1], (3, 1))
sample1 = [(x1, y1)]
inst2 = (np.array([4.8, 3.4, 1.6, 0.2]), np.array([1., 0., 0.]))
x2 = np.reshape(inst2[0], (4, 1))
y2 = np.reshape(inst2[1], (3, 1))
sample2 = [(x2, y2)]

# Call SGD with one instance for training and another for testing
net4.SGD(sample1, 1, 1, 1.0, sample2)
```

   You should get the values as below (or close to them, depending on the effect of randomized order of training instances).
   Note that the printed activation values here are from the feed-forward phase in backprop(), before weight updates in the backprop phrase. The activation values computed in the feedforward() function called at the end of each epoch in SGD() will be different.

```
[array([[5.7], [3. ], [4.2], [1.2]]),
 array([[9.99585174e-01], [1.65772381e-04]]),
 array([[0.64224084], [0.58352539], [0.28964339]])]
```

```
[Epoch 0] Training: MSE=0.26673129, CrossEntropy=1.61690061, LogLikelihood=2.3430627
          Test:     MSE=0.32440028, CrossEntropy=1.85581862, LogLikelihood=2.4017219
```

Lastly, comment out or remove the line you added in backprop() -- **(*)**, since it won't be needed after this.

2. Check your implementation for Modification 2b (SGD return value) and 2c (Early Stopping). Steps are:

   - First use a smaller dataset, **"xor.csv"**, to check if your early stopping is working correctly. You can use the code below to load the data.

```
import NN578_network as network
import numpy as np
ret = np.genfromtxt('../data/xor.csv', delimiter=',')
temp = np.array([(entry[:2],entry[2:]) for entry in ret])

temp_inputs = [np.reshape(x, (2, 1)) for x in temp[:,0]]
temp_results = [network.vectorize_target(2, y) for y in temp[:,1]] # convert one tar
xor_data = list(zip(temp_inputs, temp_results))
```

   Then you create a network of the size [2, 4, 2] (i.e., one hidden layer with 4 nodes), and train it using max_epochs = 30, mini_batch_size = 1 and eta = 2.2. You should be able to see the training terminates by reaching 100 % accuracy around epoch 18 (before the max 30). If it didn't terminate before 30 epochs, run the network several times... different initial weights may converge faster.

   - Next you verify the value returned from SGD(). If you are using Jupyter Notebook, after you execute the function in the previous step, you should see the output of the form below. Note the second list is for test data, which we didn't have.. so the performance results are all empty.

```
[[[2, 0.5, 0.3018070295642147, 1.6471739303033757, 0.6426855941426012],
  [2, 0.5, 0.273454307822242, 1.4936864725818466, 0.9832082595521336],
  ...],
 [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]]
```

   - Finally use a slightly larger dataset, **"iris.csv"** for an open experiment. In this dataset, each observation has 4 input values and 3 output values (corresponding to three output classes). You can load this data in the same way as above, except for the target (y) since the target in this dataset is already encoded in the vectorized format.

```
temp_results = [np.reshape(y, (3, 1)) for y in data[:,1]]
```

   Experiment with various network architecture configurations (e.g., # nodes in the hidden layer, # of hidden layers) and parameters (eta, mini_batch_size) to get the network to terminate with a 100 % accuracy before 30 epochs (max), if you can.
   Find **TWO SETS of configurations** (together with parameters) which are (relatively)significantly different. Then write your experiment, results and comments in the write-up **(*)**.

Extra Credit

You will receive some extra credit if you visualize the network learning.

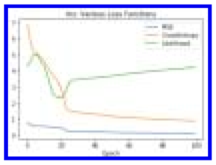
You can use any dataset you like, also plot any things you thought are interesting. Minimal requirements are:

1. Use the values returned from the function SGD() .. Modification 2b.
2. Give at least two plots to be considered for a full credit.
3. Write your comments and reflection in the write-up **(*)**.

Explore. Have fun :)

Examples:

- The plot below shows the comparison between various loss functions (computed in the evaluate() function). The iris dataset is randomly split into 70% training and 30% test, and the network of size [4, 20, 3] was run for the maximum of 100 epochs with eta = 0.125 and the mini batch size = 10. The values in the plot are for training data.

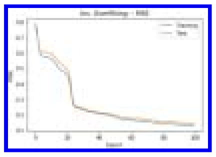
  As you see, the loss of MSE is pretty much decreasing monotonically as expected. That is because the gradient used in the code was based on MSE (the quadratic loss function).

  For CrossEntropy, the error values seem to be decreasing as well, which is a good thing. The decrease seems to taper off after 30 epochs or so, and it is coinciding with MSE (and that's a good sign too).

  As for Likelihood, after initial increase, values go down until 20 epochs or so, then go up. So ,this function does not seem to go with the gradient of the quadratic loss function.

  But this is just one run of the network training. Even with the same network architecture and parameter values, you know that initial weight values may produce quite different learning curves.

- The one below shows the comparison of MSE between training and test, to inspect overfitting.

  Although the values are consistently low for the training set, the difference between the two sets is not very much. The two curves are very close overall. So we can conclude that the overfitting was not severe for this run of the training.

Submission

1. Two code files. Be sure to add your **name**, **course/section number** and the **assignment name** at the top of BOTH FILES.

2. A **video.**
   - Create a publicly accessible video (e.g. Youtube) or a shared video.
   - Video capture of your desktop with voice over. (Recommended tools are Camtasia and Screen Recorder.)
   - A maximum of 5 minutes. NO MORE.
   - Content:
     - Walk through your code, and demonstrate that your code is working correctly.
     - Ensure to point out and explain how you implemented the requirements.
     - No submission of the video file. Just write the **link to the video in the documentation AND D2L's submission box (of HW #1 bin)**.

3. Documentation (*).
   - In docx or pdf.

- Minimum 0.75 page (i.e., one page where 3/4 of the page is filled).
- Be sure to add your **name**, **course/section number** and the **assignment name** at the top of the file.
- Write as much as you can. It is my policy that I consider terse answers insufficient, therefore won't give a full credit.
- Content should include:
  - IRIS open (early stopping) experiment.
  - Graphs and your comments for the extra credit part, if you did it.
  - Your reaction and reflection on this assignment overall (e.g. difficulty level, challenges you had).

DO NOT ZIP YOUR CODE OR WRITE UP. SUBMIT EACH FILE SEPERATELY.