SIMON D. LEVY

Computer Science Department, Washington and Lee University

CSCI 315 Assignment #2

Due on Github 11:59PM Friday 3 February

Goal

The goal of this assignment is to build your first neural network, a perceptron, to process a real data set: handprinted digits. In the process of doing this, you will also gain more familiarity with NumPy. Although you'll see perceptron implemenations in the textbook we're using, **please write all your code from scratch**. As Richard Feynman said: *What I cannot build, I do not understand*!

Part 0: Numpy warmup

If you're already familiar with NumPy, you can probably skip ahead to Part 1. Otherwise, fire up IDLE3 or your favorite Python3 interpreter, and do this:

```
>>> import numpy as np
```

Now we can just use the abbreviation `np` whenever we want to use something from NumPy. For example, we can generate a 3×4 matrix (array) of zeros like this:

```
>>> np.zeros((3,4))
```

Note that the `zeros` function takes a tuple (pair) of values in parentheses, not two separate inputs. To generate a matrix of random values between 0 and 1, try this:

```
>>> np.random.random((3,4))
```

Repeating this command, you should see new random values each time. The power of NumPy comes from its ability to act upon an entire matrix at a time. For example:

```
>>> 2* np.random.random((3,4)) - 1
```

Will give you random numbers between -1 and +1 instead of between 0 and +1.
You can also run a single function on all elements of the matrix; for example:

```
>>> np.tanh(np.random.random((3,4)))
```

Finally, NumPy can do linear algebra for you, multiplying a matrix by another matrix or by a vector (row or column of numbers) using a special operation called `dot` (short for *dot product*). Try this:

```
>>> w = np.random.random((4,3))
>>> I = np.random.random((1,4))
>>> np.dot(I, w)
```

You should see a 1×3 matrix of random numbers. Why? Because `I` has one row and four columns, and `w` has four rows and three columns, their product (dot) has one row and three columns.Now that you've got a handle on NumPy, it's time to build a perceptron.

Part 1: Build your perceptron

Write a module `perceptron.py` based on slides #3 and #4 of the lecture notes. Your module should of course start by importing NumPy:

```
import numpy as np
```

Your module should then define a class `Perceptron`. As usual when creating a Python class, you should first define an `__init__` method and a `__str__` method. The `__init__` method should allow you to specify the number of inputs *n* and the number of outputs *m*, which will be instance variables for the class. Once you've got that part of `__init__` written, you can write the `__str__` method. This method should return a string with some useful info about the perceptron; for example, *A perceptron with 5 inputs and 3 outputs*.What other instance variables do we need to represent the state of our perceptron? Weights, of course! Here's where we start using NumPy. Looking at the Perceptron Learning Algorithm in slide #6, you see that it initializes the weights as an $(n+1) \times m$ matrix of small random numbers above and below 0. Looking over your NumPy warmup above, it should be pretty obvious how to do this: instead of multiplying by 2 and subtracting 1, figure out appropriate values to get the `w` values between some small interval, like [-.05,+.05]. You can debug your perceptron by printing out the weights at the end of your `__init__`, but make sure to get rid of that debugging output once you've got the init working. Or, you could have your `__str__`method return a a string representation of the weights, although this will become impractical for larger perceptrons.

Before we can train our perceptron to do anything interesting, we need a way of testing it in a given input. So add a method `test`. In a single line of code, this method should take an input vector (numpy array) `I`, append a 1 to it for the bias term (use `np.append`), perform the `np.dot` operation on it and the perceptron's weights, compare the resulting vector with 0, and return the result.

Now that you've got your `test` method written, how should you test the test? As usual with software testing, it's best to make sure it works as expected on a few simple, extreme cases. The simplest possible perceptron would have a single input and a single output ($m = n = 1$). This will give you just two weights. By examining the weights, you should be able to tell ahead of time what the `test` method will output on the two extreme values of the input: an input of 0 and an input of 1. Keep creating and testing this tiny perceptron until you get one that distinguishes an input of 0 from an input of 1. Once you've got that working, try a perceptron with two inputs instead of one. You can use `np.zeros` and `np.ones` to create an input vector of all 0s or 1s. Do the same validation as you did with the one-input perceptron, looking at the weights to make sure the output makes sense.

Repeat with a perceptron of one input and two outputs, then finally two inputs and two outputs. If you can verify the results with these simple cases, there's unlikely to be anything wrong with your `test` method.

Finally, it's time to write the `train` method. This method should take two parameters: a NumPy matrix of input patterns (one pattern per row) and a NumPy matrix of target patterns (one per row), with the same number of rows as the input patterns. (Don't worry about raising an exception if they have a different number of rows; you can that later if you like, after you've got the basics working.) You can use `np.hstack` for augmenting your input matrix with a final column of 1's. Your `train` method should loop over the rows in the matrices, running the Perceptron Learning Algorithm from slide #6 of the lecture notes. For (Ij∗w)(Ij∗w) you can use `np.dot` as in the previous assignment. For ITj∗DjIjT∗Dj you can use `np.outer`. You can use 1000 iterations as a default, allowing your user to specify a different number by using a Python named argument for the third input to `train`. As with your `test` method, debug this method by using very simple functions (input/target sets). The Boolean functions AND and OR are the classic training examples for a perceptron, so try them (train/test one perceptron for OR, and another for AND). As with the previous assignment, I should be able to open your code in IDLE, hit F5, and see a nice clean printout showing me the trained perceptron's behavior on each of the four inputs for these two Boolean functions.

It should be easy to get perfect results with these two simple functions (for reasons we will discuss in class). So you might want to add a third, optional parameter to your `train` method, `niter=1000`, for the number of iterations of the outer loop. Experiment with how small a number it takes for the weights to *converge* (stop giving a better result). For this part, I often got convergence / success in under five iterations.

Part 2: A realistic data set

Our realistic data set consists of handprinted digits, originally provided by Yann Le Cun. Each digit is described by a 14×14 pixel array. Each pixel has a grey level with value ranging from 0 to 1. The data is split between two files, a training set that contains the examples used for training your neural network, and a test set that contains examples you'll use to evaluate the trained network. Both training and test sets are organized the same way. Each file begins with 250 examples of the digit "0", followed by 250 examples of the digit "1", and so forth up to the digit "9". There are thus 2500 examples in the training set and another 2500 examples in the test set.

Each digit begins with a label on a line by itself, e.g., "train4-17", where the "4" indicates the target digit, and the "17" indicates the example number. The next 14 lines contain 14 real values specifying pixel intensities for pixels in the corresponding row. Finally, there is a line with 10 integer values indicating the target. The vector "1 0 0 0 0 0 0 0 0 0" indicates the target 0; the vector "0 0 0 0 0 0 0 0 0 1" indicates the target 9.
Write code to read in a file — either train or test — and build a data structure containing the input-output examples. Although the digit pixels lie on a 14×14 array, the input to your network will be a 196-element vector. The arrangement into rows is to help our eyes see the patterns. You might also write code to visualize individual examples.

Note: we will use the same data set in the next assignment when we implement back propagation, so the utility functions you write here will be reused. It's worth writing some nice code to step through examples and visualize the patterns.

Part 3: Perceptron as classifier

Train a perceptron to discriminate 2 from not-2. That is, lump together all the examples of the digits {0, 1, 3, 4, 5, 6, 7, 8, 9}. You will have 250 examples of the digit 2 in your training file and 2250 examples of not-2. Assess performance on the test set and report false-positive and false-negative rates. The false-positive rate is the rate at which the classifier says "yes" when it should say "no" (i.e., the proportion of not-2's which are classified as 2's). The false-negative rate is rate at which the classifier says "no" when it should say "yes" (i.e., the proportion of 2's which are classified as not-2's).

Note: The perceptron algorithm is an "on line" algorithm: you adjust the weights after each example is presented. Next assignment, we're going to change your code to implement back propagation. Back propagation can be run in an "on line" mode, or "batch". To anticipate next assignment's work, you might want to set up your code this assignment to process minibatches of between 1 and 2500 examples. You will compute the summed weight update for all examples in the minibatch, and then update the weights at the end of the minibatch. (A minibatch with 1 example corresponds to the on-line algorithm; a minibatch with 2500 examples corresponds to the batch algorithm.)

Remember an important property of the perceptron algorithm: it is guaranteed to converge only if there is a setting of the weights that will classify the training set perfectly. (The learning rule corrects errors. When all examples are classified correctly, the weights stop changing.) With a noisy data set like this one, the algorithm will not find an exact

solution. Also remember that the perceptron algorithm is not performing gradient descent. Instead, it will jitter around the solution continually changing the weights from one iteration to the next. The weight changes will have a small effect on performance, so you'll see training set performance jitter a bit as well.

Part 4 (Extra Credit)

Train a perceptron to discriminate 8 from 0. You will have 500 training examples and 500 test examples.

Part 5 (Extra Credit)

Train a perceptron with 10 outputs to classify the digits 0-9 into distinct classes. Using the test set, construct a confusion matrix. The 10×10 confusion matrix will specify the frequency by which each input digit is assigned to each output class. The diagonal of this matrix will indicate correct classifications.

* Based on https://www.cs.colorado.edu/~mozer/Teaching/syllabi/DeepLearning2015/assignments/assignment2.html