University of Arizona		Homework#5	Intro to NNs
CSC 585 Algorithms for NLP		Due date:	November 7, 2023
Fall 2023		**Name:**	
Instructor: Eduardo Blanco			

Instructions

- Bring a hard copy to class with your answers to the questions in Questions 1 and the report for Question 2.
- Submit your implementation for Questions 1 and 2 on GitHub Classroom. You will find the invite link for the repository in d2l. Follow these steps:

```
# Accept the invite link for the assignment available at d2l.
# It will create a repo named 04-dep-parsing-GITHUB_USER
#   Please use a github user name that is your name (or something close to it).
# Then set up the environment
> git clone YOUR_REPO
> cd 04-dep-parsing-GITHUB_USER
> python3 -m venv .env
> source .env/bin/activate
> pip install -r requirements.txt
```

No test cases for now on GitHub, but you are told the expected accuracies. You are encouraged to discuss in Piazza, but you cannot share code in Piazza. Anything you can write in English is fine (feature descriptions, papers you used for inspiration, and so on). Just don't share Python code (or pseudo-code).

Question 1 [20pt]
Implement the gradient descent algorithm that we discussed in class to train a sigmoid unit (one neuron with sigmoid activation). Your implementation only needs to handle binary classification tasks (each instance will have class 0 or 1). In addition, you may assume that all attributes have binary values (either 0 or 1).

- A train (`train.data`) and test file (`test.data`) are provided (`data/` folder).
- Use the starter code provided.
- Your program must take 4 arguments as input: (1) training file, (2) test file, (3) learning rate and (4) number of epochs. Example:

```
$ python perceptron.py data/train.dat data/test.dat 0.01 10
```

- A few notes:
  - Initialize all the weights to 0.
  - Use 0.5 as the classification threshold (i.e., classify the instance as 1 if the unit outputs a value that is greater or equal than 0.5, otherwise classify the instance as 0).
  - Remember that the weight update rule is defined as follows: $w_j = w_j + \eta \times \text{Err} \times \sigma(in) \times (1 - \sigma(in)) \times x_j$, where η is the learning rate, Err is the error with one instance, σ is the sigmoid function, in is the input to the activation (i.e., the output of the dot product of weights and attributes), and x_j is the jth input.

  - A working implementation will get 80% accuracy (with some (not any) learning rate and number of epochs).
  - You can try many hyperparameters using a grid search (with the command line):

```
for lr in 0.005 0.01 0.025 0.05 0.1
do
  for e in 1 2 3 4 5 10 15 25 50 100
  do
    echo "$lr $e"
    python3 ./neuron.py  ../data/train.dat ../data/test.dat $lr $e
    echo " "
  done
done
```

- Answer the following questions in your report:
  1. Can a single neuron memorize `train.dat`? Why?
  2. Can a single neuron memorize `test.dat`? Why?
  3. You could calculate Err as the difference between (a) the target (gold label) and (b) the output of your classifier or the output of the sigmoid. Note that the output of the classifier is discrete (0 or 1) and the output of the sigmoid ranges from 0 to 1. Which one is better? Answer these questions in your report:
     (a) empirically (e.g., "I get accuracies X and Y with the first and second option respectively, so the first option is better because it yields better accuracy" and
     (b) explain with your own words why one option is better.

     Mentioning accuracy is not acceptable as a (non-empirical) explanation (Question 3b).
  4. Real datasets often come with useless attributes. For example, a "national id number" should be useless for predicting income (assuming that these numbers are assigned randomly). What weights do you expect to learn for useless attributes?

Question 2 [80pt]
Improve your implementation of the arc-standard algorithm so that you get better accuracy. The key is to move from a "majority baseline" oracle to an oracle that is learned from data. A perfectly fine option is to use word embeddings and one neuron with sigmoid activation, but you can get fancier. A few notes:

- You can use any feature (except the gold!) that comes with the dataset and any other feature you can think of as long as you can implement an extractor.
- Here are a few options for embeddings:
  - word2vec: `https://code.google.com/archive/p/word2vec/`
  - GloVe: `https://nlp.stanford.edu/projects/glove/`

  You can use any static word embeddings you can download, but do cite which ones you use.
- You can (should?) use some manually extracted features. We saw in class feature templates; useful features include word forms and part-of-speech tags within a window (from the stack and buffer). Including information about the partial parse is also useful.

- You can get inspiration from any paper you can find, but do cite your sources. You are asked to replicate at least part of this paper:

  > Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 740–750, Doha, Qatar. Association for Computational Linguistics.

- **The only hard requirement is that you must use the arc-standard parsing algorithm.** You can use the simplest neural network (one neuron) or something more complicated: a multilayer perceptron or any other neural classifier. Using APIs and toolkits is fine (keras, pytorch, etc.) as long as the only thing you learn is an oracle to decide the next transition given a configuration (as part of the arc-standard parsing algorithm).

You will be graded as follows (and this time we will be strict):

- Report (10 points). Write a report explaining what you tried, what worked and what did not work (3 pages is plenty (including tables and plots); more is fine but not needed). I am not interested in an explanation of the arc-standard algorithm. I am interested in how you learned the oracle (features, architecture, etc.), what you thought would work but did not work, and where you got stuck (and how you got unstuck).
- Evaluation with the public dataset:
  - submissions within 2 accuracy points of the best submission: 45 points
  - submissions within 5 accuracy points of the best submission: 40 points
  - submissions within 10 accuracy points of the best submission: 35 points
  - submissions within 15 accuracy points of the best submission: 30 points
  - submissions within 20 accuracy points of the best submission: 25 points
- Evaluation with the private dataset. I will chose a different slice of the English chunk of universal dependencies; whatever you hard code is unlikely to generalize:
  - submissions within 2 accuracy points of the best submission: 25 points
  - submissions within 5 accuracy points of the best submission: 20 points
  - submissions within 10 accuracy points of the best submission: 15 points
  - submissions within 15 accuracy points of the best submission: 10 points