CS 224N: Assignment #1

Due date: 1/25 11:59 PM PST (You are allowed to use three (3) late days maximum for this assignment)

These questions require thought, but do not require long answers. Please be as concise as possible.

We encourage students to discuss in groups for assignments. **However, each student must finish the problem set and programming assignment individually, and must turn in her/his assignment.** We ask that you abide by the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work is done by yourself. If you have discussed the problems with others, please include a statement saying who you discussed problems with. Failure to follow these instructions will be reported to the Office of Community Standards.

Please review any additional instructions posted on the assignment page at http://cs224n.stanford.edu/assignment1. When you are ready to submit, please follow the instructions on the course website. **Make sure you test your code using the provided commands and do not edit outside of the marked areas. Code that does not run on corn or incorporates additional libraries will receive no credit.**

1 Softmax (10 points)

(a) (5 points) Prove that softmax is invariant to constant offsets in the input, that is, for any input vector $\boldsymbol{x}$ and any constant c,

$$\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} + c)$$

where $\boldsymbol{x} + c$ means adding the constant c to every dimension of $\boldsymbol{x}$. Remember that

$$softmax(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}} \tag{1}$$

Note: In practice, we make use of this property and choose $c = -\max_i x_i$ when computing softmax probabilities for numerical stability (i.e., subtracting its maximum element from all elements of $\boldsymbol{x}$).

(b) (5 points) Given an input matrix of N rows and D columns, compute the softmax prediction for each row using the optimization in part (a). Write your implementation in `q1_softmax.py`. You may test by executing `python q1_softmax.py`.

Note: The provided tests are not exhaustive. Later parts of the assignment will reference this code so it is important to have a correct implementation. Your implementation should also be efficient and vectorized whenever possible (i.e., use numpy matrix operations rather than for loops). A non-vectorized implementation will not receive full credit!

2 Neural Network Basics (30 points)

(a) (3 points) Derive the gradients of the sigmoid function and show that it can be rewritten as a function of the function value (i.e., in some expression where only $\sigma(x)$, but not x, is present). Assume that the input x is a scalar for this question. Recall, the sigmoid function is

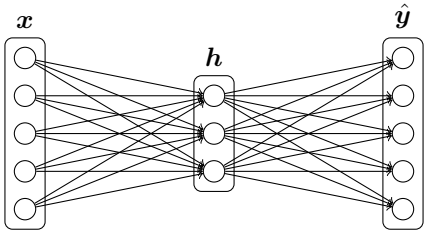
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

(b) (3 points) Derive the gradient with regard to the inputs of a softmax function when cross entropy loss is used for evaluation, i.e., find the gradients with respect to the softmax input vector $\boldsymbol{\theta}$, when the prediction is made by $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{\theta})$. Remember the cross entropy function is

$$CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_i y_i \log(\hat{y}_i) \tag{3}$$

where $\boldsymbol{y}$ is the one-hot label vector, and $\hat{\boldsymbol{y}}$ is the predicted probability vector for all classes. (*Hint: you might want to consider the fact many elements of* $\boldsymbol{y}$ *are zeros, and assume that only the* k*-th dimension of* $\boldsymbol{y}$ *is one.*)

(c) (6 points) Derive the gradients with respect to the *inputs* $\boldsymbol{x}$ to an one-hidden-layer neural network (that is, find $\frac{\partial J}{\partial \boldsymbol{x}}$ where $J = CE(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is the cost function for the neural network). The neural network employs sigmoid activation function for the hidden layer, and softmax for the output layer. Assume the one-hot label vector is $\boldsymbol{y}$, and cross entropy cost is used. (Feel free to use $\sigma'(x)$ as the shorthand for sigmoid gradient, and feel free to define any variables whenever you see fit.)



Recall that the forward propagation is as follows

$$\boldsymbol{h} = \text{sigmoid}(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1) \qquad\qquad \hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{h}\boldsymbol{W}_2 + \boldsymbol{b}_2)$$

Note that here we're assuming that the input vector (thus the hidden variables and output probabilities) is a row vector to be consistent with the programming assignment. When we apply the sigmoid function to a vector, we are applying it to each of the elements of that vector. $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ $(i = 1, 2)$ are the weights and biases, respectively, of the two layers.

(d) (2 points) How many parameters are there in this neural network, assuming the input is D_x-dimensional, the output is D_y-dimensional, and there are H hidden units?

(e) (4 points) Fill in the implementation for the sigmoid activation function and its gradient in `q2_sigmoid.py`. Test your implementation using `python q2_sigmoid.py`. *Again, thoroughly test your code as the provided tests may not be exhaustive.*

(f) (4 points) To make debugging easier, we will now implement a gradient checker. Fill in the implementation for `gradcheck_naive` in `q2_gradcheck.py`. Test your code using `python q2_gradcheck.py`.

(g) (8 points) Now, implement the forward and backward passes for a neural network with one sigmoid hidden layer. Fill in your implementation in `q2_neural.py`. Sanity check your implementation with `python q2_neural.py`.

3 word2vec (40 points + 2 bonus)

(a) (3 points) Assume you are given a "predicted" word vector $\boldsymbol{v}_c$ corresponding to the center word c for Skip-Gram, and word prediction is made with the softmax function found in `word2vec` models

$$\hat{y}_o = p(o \mid c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{w=1}^{V} \exp(\boldsymbol{u}_w^\top \boldsymbol{v}_c)} \tag{4}$$

where u_w $(w = 1, \ldots, V)$ are the "output" word vectors for all words in the vocabulary. Assuming cross entropy cost is applied to this prediction and word o is the expected word (the o-th element of the one-hot label vector is one), derive the gradients with respect to $\boldsymbol{v}_c$.

Hint: It will be helpful to use notation from question 2. For instance, letting $\hat{\boldsymbol{y}}$ be the vector of softmax predictions for every word, $\boldsymbol{y}$ as the expected word vector, and the loss function

$$J_{softmax-CE}(o, \boldsymbol{v}_c, \boldsymbol{U}) = CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{5}$$

where $\boldsymbol{U} = [\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_V]$ is the matrix of all the output vectors. *Make sure you state the orientation of your vectors and matrices.*

(b) (3 points) As in the previous part, derive gradients for the "output" word vectors $\boldsymbol{u}_k$'s (including $\boldsymbol{u}_o$).

(c) (6 points) Repeat part (a) and (b) assuming we are using the negative sampling loss for the predicted vector $\boldsymbol{v}_c$, and the expected output word index is o. Assume that K negative samples (words) are drawn, and they are $1, 2, ..., K$ respectively for simplicity of notation ($o \notin \{1, \ldots, K\}$). Again, for a given word, o, denote its output vector as $\boldsymbol{u}_o$. The negative sampling loss function in this case is

$$J_{neg-sample}(o, \boldsymbol{v}_c, \boldsymbol{U}) = -\log(\sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c)) - \sum_{k=1}^{K} \log(\sigma(-\boldsymbol{u}_k^\top \boldsymbol{v}_c)) \tag{6}$$

where $\sigma(\cdot)$ is the sigmoid function.

After you've done this, describe with one sentence why this cost function is much more efficient to compute than the softmax-CE loss (you could provide a speed-up ratio, i.e., the runtime of the softmax-CE loss divided by the runtime of the negative sampling loss).

Note: the cost function here is the negative of what Mikolov et al had in their original paper, because we are doing a minimization instead of maximization in our code.

(d) (8 points) Suppose the center word is $c = w_t$ and the context words are $[w_{t-m}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+m}]$, where m is the context size. Derive gradients for all of the word vectors for Skip-Gram and CBOW given the previous parts.

Hint: feel free to use $F(o, \boldsymbol{v}_c)$ (where o is the expected word) as a placeholder for the $J_{softmax-CE}(o, \boldsymbol{v}_c, ...)$ or $J_{neg-sample}(o, \boldsymbol{v}_c, ...)$ cost functions in this part — you'll see that this is a useful abstraction for the coding part. That is, your solution may contain terms of the form $\frac{\partial F(\boldsymbol{o}, v_c)}{\partial ...}$.

Recall that for skip-gram, the cost for a context centered around c is

$$J_{\text{skip-gram}}(w_{t-m \ldots t+m}) = \sum_{-m \le j \le m, j \neq 0} F(w_{t+j}, \boldsymbol{v}_c) \tag{7}$$

where w_{t+j} refers to the word at the j-th index from the center.

CBOW is slightly different. Instead of using $\boldsymbol{v}_c$ as the predicted vector, we use $\hat{\boldsymbol{v}}$ defined below. For (a simpler variant of) CBOW, we sum up the input word vectors in the context

$$\hat{\boldsymbol{v}} = \sum_{-m \le j \le m, j \neq 0} \boldsymbol{v}_{w_{t+j}} \tag{8}$$

then the CBOW cost is

$$J_{\text{CBOW}}(w_{c-m \ldots c+m}) = F(w_t, \hat{\boldsymbol{v}}) \tag{9}$$

Note: To be consistent with the $\hat{\boldsymbol{v}}$ notation such as for the code portion, for skip-gram $\hat{\boldsymbol{v}} = \boldsymbol{v}_c$.

(e) (12 points) In this part you will implement the word2vec models and train your own word vectors with stochastic gradient descent (SGD). First, write a helper function to normalize rows of a matrix in `q3_word2vec.py`. In the same file, fill in the implementation for the softmax and negative sampling cost and gradient functions. Then, fill in the implementation of the cost and gradient functions for the skip-gram model. When you are done, test your implementation by running `python q3_word2vec.py`. *Note: If you choose not to implement CBOW (part h), simply remove the NotImplementedError so that your tests will complete.*

(f) (4 points) Complete the implementation for your SGD optimizer in `q3_sgd.py`. Test your implementation by running `python q3_sgd.py`.

(g) (4 points) Show time! Now we are going to load some real data and train word vectors with everything you just implemented! We are going to use the Stanford Sentiment Treebank (SST) dataset to train word vectors, and later apply them to a simple sentiment analysis task. You will need to fetch the datasets first. To do this, run `sh get_datasets.sh`. There is no additional code to write for this part; just run `python q3_run.py`.

*Note: The training process may take a long time depending on the efficiency of your implementation **(an efficient implementation takes approximately an hour)**. Plan accordingly!*

When the script finishes, a visualization for your word vectors will appear. It will also be saved as `q3_word_vectors.png` in your project directory. **Include the plot in your homework write up.** Briefly explain in at most three sentences what you see in the plot.

(h) (Extra credit: 2 points) Implement the CBOW model in `q3_word2vec.py`. *Note: This part is optional but the gradient derivations for CBOW in part (d) are not!.*

4 Sentiment Analysis (20 points)

Now, with the word vectors you trained, we are going to perform a simple sentiment analysis. For each sentence in the Stanford Sentiment Treebank dataset, we are going to use the average of all the word vectors in that sentence as its feature, and try to predict the sentiment level of the said sentence. The sentiment level of the phrases are represented as real values in the original dataset, here we'll just use five classes:

"very negative (−−)", "negative (−)", "neutral", "positive (+)", "very positive (++)"

which are represented by 0 to 4 in the code, respectively. For this part, you will learn to train a softmax classifier, and perform train/dev validation to improve generalization.

(a) (2 points) Implement a sentence featurizer. A simple way of representing a sentence is taking the average of the vectors of the words in the sentence. Fill in the implementation in `q4_sentiment.py`.

(b) (1 points) Explain in at most two sentences why we want to introduce regularization when doing classification (in fact, most machine learning tasks).

(c) (2 points) Fill in the hyperparameter selection code in `q4_sentiment.py` to search for the "optimal" regularization parameter. You need to implement both `getRegularizationValues` and `chooseBestModel`. Attach your code for `chooseBestModel` to your written write-up. You should be able to attain at least 36.5% accuracy on the dev and test sets using the pretrained vectors in part (d).

(d) (3 points) Run `python q4_sentiment.py --yourvectors` to train a model using your word vectors from q3. Now, run `python q4_sentiment.py --pretrained` to train a model using pretrained GloVe vectors (on Wikipedia data). Compare and report the best train, dev, and test accuracies. Why do you think the pretrained vectors did better? Be specific and justify with 3 distinct reasons.

(e) (4 points) Plot the classification accuracy on the train and dev set with respect to the regularization value for the pretrained GloVe vectors, using a logarithmic scale on the x-axis. This should have been done automatically. **Include `q4_reg_acc.png` in your homework write up.** Briefly explain in at most three sentences what you see in the plot.

(f) (4 points) We will now analyze errors that the model makes (with pretrained GloVe vectors). When you ran `python q4_sentiment.py --pretrained`, two files should have been generated. Take a look at `q4_dev_conf.png` and **include it in your homework writeup**. Interpret the confusion matrix in at most three sentences.

(g) (4 points) Next, take a look at `q4_dev_pred.txt`. Choose 3 examples where your classifier made errors and briefly explain the error and what features the classifier would need to classify the example correctly (1 sentence per example). Try to pick examples with different reasons.