Assignment 1

1 Softmax

1. **Show that softmax(x) = softmax(x + c).**

This exercise is not merely academic. This property allows us to substract the maximum element in a vector from all other elements in that vector, and then apply softmax to the result, without affecting the softmax values. This is important to ensure numerical stability. Without normalization, the elements e^{x_i} can be very large values that lead to numerical overflow. With normalization, all elements are mapped to values between 0 and 1.

$$\begin{aligned} softmax(x+c)_j &= \frac{e^{x_j+c}}{\sum_i e^{x_i+c}} \\ &= \frac{e^{x_j}e^{c}}{\sum_i e^{x_i}e^{c}} \\ &= \frac{e^{x_j}}{\sum_i e^{x_i}} \\ &= sofmax(x)_j \end{aligned}$$

2. **Implement the softmax function in q1_softmax.py.**

We first define a helper function, `softmax_vector(v)`, that calculates the element-wise softmax values of the vector `v`. This function uses the property we proved above.

```
def softmax_vector(v):
  m = np.max(v)
  return np.exp(v - m)/np.sum(np.exp(v - m))
```

There are two cases: when `x` is a matrix and when `x` is a vector. If `x` is a matrix, we apply `softmax_vector` row-wise, using the function `np.apply_along_axis`. If `x` is a vector, we just apply `softmax_vector` directly.

```
if len(x.shape) > 1:
  # Matrix
  x = np.apply_along_axis(softmax_vector, 1, x)
else:
  # Vector
  x = softmax_vector(x)
```

2 Neural Network Basics

1. **Derive the gradients of the sigmoid function and show that it can be rewritten as a function of the function value**

The only "trick" here is the addition and subtraction of e^{-x} in line 3.

$$\begin{aligned}
\sigma\prime(x) &= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{e^{-x}}{1+e^{-x}} \cdot \frac{1}{1+e^{-x}} \\
&= \frac{e^{-x}}{1+e^{-x}} \cdot \frac{1+e^{-x}-e^{-x}}{1+e^{-x}} \\
&= \sigma(x)(1-\sigma(x))
\end{aligned}$$

2. **Derive the gradient with regard to the inputs of a softmax function when cross entropy loss is used for evaluation**

This gradient is the first step in the back-propagation of our neural network. Every other gradient we calculate during backprop will contain this gradient as a factor. As the hint points out, y is a one-hot vector, so the only term j that contributes to the gradient is where $y_j = 1$. All other terms are zero. This simplifies the expressions to

$$\begin{aligned}
CE(y, \hat{y}) &= -\sum_i y_i \log(\hat{y}_i) \\
&= \log(softmax(\theta)_j) \\
&= \log(\frac{e^{\theta_j}}{\sum_k e^{\theta_k}}) \\
&= -\theta_j + \log(\sum_k e^{\theta_k})
\end{aligned}$$

The gradient with respect to θ_j is then

$$\begin{aligned}
\frac{\partial CE(y, \hat{y})}{\partial \theta_j} &= -1 + \frac{1}{\sum_k e^{\theta_k}} \cdot e^{\theta_j} \\
&= -1 + softmax(\theta)_j \\
&= \hat{y}_j - y_j
\end{aligned}$$

For $k \neq j$, the only difference is that the term -1 is instead 0, so the gradient is just $softmax(\theta)_k$. Putting all gradients together in vector notation, we get $\frac{\partial CE(y,\hat{y})}{\partial \theta} = \hat{y} - y$.

3. **Derive the gradients with respect to the inputs x to an one-hidden-layer neural network**

Book keeping is really important here, both conceptually and computationly. Conceptually, it will help to break out the applications of the chain rule into manageable chunks. Computationly, it shows what computations can be stored and re-used later. So we'll start by writing down all the intermediate variables we'll need. Going from the output layer to the input layer, we have:

$$
\begin{aligned}
\hat{y} &= softmax(\theta) \\
\theta &= hW_2 + b_2 \\
h &= \sigma(\phi) \\
\phi &= xW_1 + b_1
\end{aligned}
$$

Picking up where we left off in question 2.2, we get

$$
\begin{aligned}
\frac{\partial J}{\partial \theta} &= \hat{y} - y \\
\frac{\partial J}{\partial h} &= \frac{\partial J}{\partial \theta}\frac{\partial \theta}{\partial h} = (\hat{y} - y)W_2^T \\
\frac{\partial J}{\partial \phi} &= \frac{\partial J}{\partial h}\frac{\partial h}{\partial \phi} = (\hat{y} - y)W_2^T \circ \sigma'(\phi) \\
\frac{\partial J}{\partial x} &= \frac{\partial J}{\partial \phi}\frac{\partial \phi}{\partial x} = (\hat{y} - y)W_2^T \circ \sigma'(\phi)W_1^T
\end{aligned}
$$

Notice how some values computed during forward propagation, such as ϕ and $\hat{y}$, are re-used during backprop. Also note that the derivative of a function that is applied element-wise during forward propagation is multiplied element-wise during backprop, as indicated by the Hadamard product ($\circ$).

Let's take a moment to appreciate the oddness of this problem. After all, in the backprop algorithm we don't calculate the gradient w.r.t. the input data, x, which we consider to be fixed. Rather, the gradient is calculated w.r.t. the *parameters* W_1, b_1, W_2, b_2. Put another way, it's the parameter values that we adjust, via backprop and gradient descent, in order to minimize the loss function. Indeed, when we implement backprop below, we do so in terms of the parameter gradients. The work we've done in this problem is not used. Sad!

4. **How many parameters are there in this neural network?**

The number of parameters is given by the number of elements in the matrices W_1 and W_2, and in the vectors b_1 and b_2. Assume the input has dimension D_x, the output dimension D_y, and their are H hidden units. The matrix W_1 maps the inputs to the hidden units, so its dimensions must by $D_x \times H$. The vector b_1 is added to xW_1 row-wise, so it must have D_x elements. Similar reasoning shows that W_2 has dimensions $H \times D_y$ and b_2 has D_y elements. Total parameters are therefore

$$
D_x \cdot H + H + H \cdot D_y + D_y = H \cdot (D_x + D_y + 1) + D_y
$$

5. **Implement the sigmoid activation function**

This is very straigthforward. Given the vector `x`, the sigmoid is calcuated by

```
s = 1./(1 + np.exp(-x))
```

Given a function value `s` of the sigmoid, the gradient is given by

```
ds = s*(1 - s)
```

6. **Implement a gradient checker**

Our gradient checker approximates the gradient of a real-valued function $f(x)$ at a particular vector-value x. Let ϵ be a small positive number, say, 0.001. Let h_i be a vector with the same dimensions as x, with ϵ in the ith position, and zeros everywhere else. Then the partial derivative of f w.r.t. to x_i is estimated by

$$\frac{\partial f}{\partial x_i} = \frac{f(x + h_i) - f(x - h_i)}{2\epsilon}$$

This estimate becomes exact as $\epsilon \to 0$. The code below implements this. You can ignore the calls to `random.setstate`; they are unrelated to the gradient calculation.

```
xh = x
xh[ix] = xh[ix] + h
random.setstate(rndstate)
fxh1, gradh = f(xh)
xh[ix] = xh[ix] - 2*h
random.setstate(rndstate)
fxh2, gradh = f(xh)
numgrad = (fxh1 - fxh2)/(2*h)
```

7. **Implement the forward and backward passes for a neural network**

The forward-pass is straight-forward. We just reverse the steps from question 2.3, starting with the input x, and ending with the output $\hat{y}$. In code, with `x` replaced by `data`, this becomes:

```
### forward propagation
# layer 1
phi = np.dot(data, W1) + b1
h = sigmoid(phi)
# layer 2
theta = np.dot(h, W2) + b2
y_hat = softmax(theta)
# cross-entropy cost
cost = -np.sum(labels * np.log(y_hat))
```

The backward-pass requires us to calculate the gradients w.r.t. the parameters W_1, W_2, b_1, b_2. We use the same set of variables that we defined in question 2.3, beginning with our old friend, the gradient w.r.t. θ.

$$
\begin{aligned}
\frac{\partial J}{\partial \theta} &= \hat{y} - y \\
\frac{\partial J}{\partial W_2} = \frac{\partial J}{\partial \theta}\frac{\partial \theta}{\partial W_2} &= h^T(\hat{y} - y) \\
\frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial \theta}\frac{\partial \theta}{\partial b_2} &= \hat{y} - y \\
\frac{\partial J}{\partial h} = \frac{\partial J}{\partial \theta}\frac{\partial \theta}{\partial h} &= (\hat{y} - y){W_2}^T \\
\frac{\partial J}{\partial \phi} = \frac{\partial J}{\partial h}\frac{\partial h}{\partial \phi} &= (\hat{y} - y){W_2}^T \circ \sigma'(\phi) \\
\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial \phi}\frac{\phi}{W_1} &= x^T(\hat{y} - y){W_2}^T \circ \sigma'(\phi) \\
\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial \phi}\frac{\phi}{b_1} &= (\hat{y} - y){W_2}^T \circ \sigma'(\phi)
\end{aligned}
$$

In Python code, with `y` replaced by `labels` and `x` by `data`, this becomes

```
### backward propagation
y_delta = y_hat - labels
# layer 2
gradW2 = np.dot(h.T, y_delta)
gradb2 = y_delta
gradh = np.dot(y_delta, W2.T)
grad_phi = gradh * sigmoid_grad(h)
# layer 1
gradW1 = np.dot(data.T, grad_phi)
gradb1 = grad_phi
```

3 word2vec

1. **Derive the gradients with respect to v_c in the skipgram model with cross-entropy loss**

Before calculating yet another gradient, let's pause to consider two techniques for checking the correctness of our gradient calculations. Neither technique guarantees a correct gradient, but they can rule out certain incorrect gradients.

First, check dimensions. The gradient w.r.t. a vector or matrix has the same dimensions as that vector or matrix. For example, the gradient w.r.t. an $M \times N$ matrix is another $M \times N$ matrix.

Second, check that the gradient updates model parameters in the right direction during gradient descent. In general there are two cases to consider. If the the model output, $\hat{y}$, is lower than the target value, y, then the gradient update should increase the value of $\hat{y}$. If $\hat{y}$ is too high, the update should decrease it.

For example, in Problem 2.1, we calculated the gradient w.r.t. θ_j as $\hat{y}_j - y_j$. If y_j is zero, we want $\hat{y}_j$ to be as close to zero as possible, to minimize the loss function. Because the gradient is positive, and gradient descent updates parameters in the direction *opposite* their gradients, we can see that θ_j will be smaller after the update. This will, in turn, decrease the value of $\hat{y}_j$, which is what we want.

In Problem 2.2 above, we calculated this gradient w.r.t. the generic parameter vector θ. We need to write θ in terms of the center word vector, v_c, and the matrix of outer word vectors U. Let v_c be a row vector of length D. Then U is a matrix of dimension $D \times W$, where W is the size of our vocabulary. Then we can write θ as

$$\theta = v_c U$$

By the chain rule, we have for v_c

$$\frac{\partial J}{\partial v_c} = \frac{\partial J}{\partial \theta}\frac{\partial \theta}{\partial v_c} = (\hat{y} - y)U^T.$$

2. **As in the previous part, derive gradients for the "output" word vectors u_w**

For U we get

$$\frac{\partial J}{\partial U} = \frac{\partial J}{\partial U}\frac{\partial \theta}{\partial U} = {v_c}^T(\hat{y} - y).$$

3. **Repeat assuming we are using the negative sampling loss**

We have

$$\begin{aligned}
\frac{\partial J}{\partial v_c} &= \frac{\partial J}{\partial \sigma}\frac{\partial \sigma}{\partial v_c} \\
&= -(1 - \sigma({u_o}^T v_c))\frac{\partial \sigma}{\partial v_c} - \sum_k (1 - \sigma(-{u_k}^T v_c))\frac{\partial \sigma}{\partial v_c} \\
&= -(1 - \sigma({u_o}^T v_c)){u_o}^T + \sum_k (1 - \sigma(-{u_k}^T v_c)){u_k}^T
\end{aligned}$$

By design, u_o is not amoung the vectors u_k, so the summation goes away when calculating $\frac{\partial J}{\partial u_o}$., leaving only the term on the left. We get

$$\frac{\partial J}{\partial u_o} = \frac{\partial J}{\partial \sigma}\frac{\partial \sigma}{\partial u_o} = -(1 - \sigma({u_o}^T v_c)){v_c}^T$$

For the u_k vectors, we get

$$\frac{\partial J}{\partial u_k} = \frac{\partial J}{\partial \sigma}\frac{\partial \sigma}{\partial u_k} = (1 - \sigma(-{u_k}^T v_c)){v_c}^T$$

Negative sampling loss samples K words from the vocabulary of size W, whereas the softmax-CE loss sums over the entire vocabulary. The speedup is proportation to $\frac{K}{W}$.

4. **Derive gradients for all of the word vectors for skip-gram and CBOW given the previous parts**

In previous parts, we calculated gradients for single v_c, u_o word vector pairs. Here we combine these to compute the gradient for all context words. Let U be the matrix of output vectors as defined in Part 3.1. For skipgram, the loss function is just a linear combination of the loss functions for each word pair, so the gradient is just the sum of individual gradients,

$$\nabla_{v_c} J_{skip-gram} = \sum_{j \neq i} \nabla_{v_c} F(w_{c+j}, v_c)$$

.

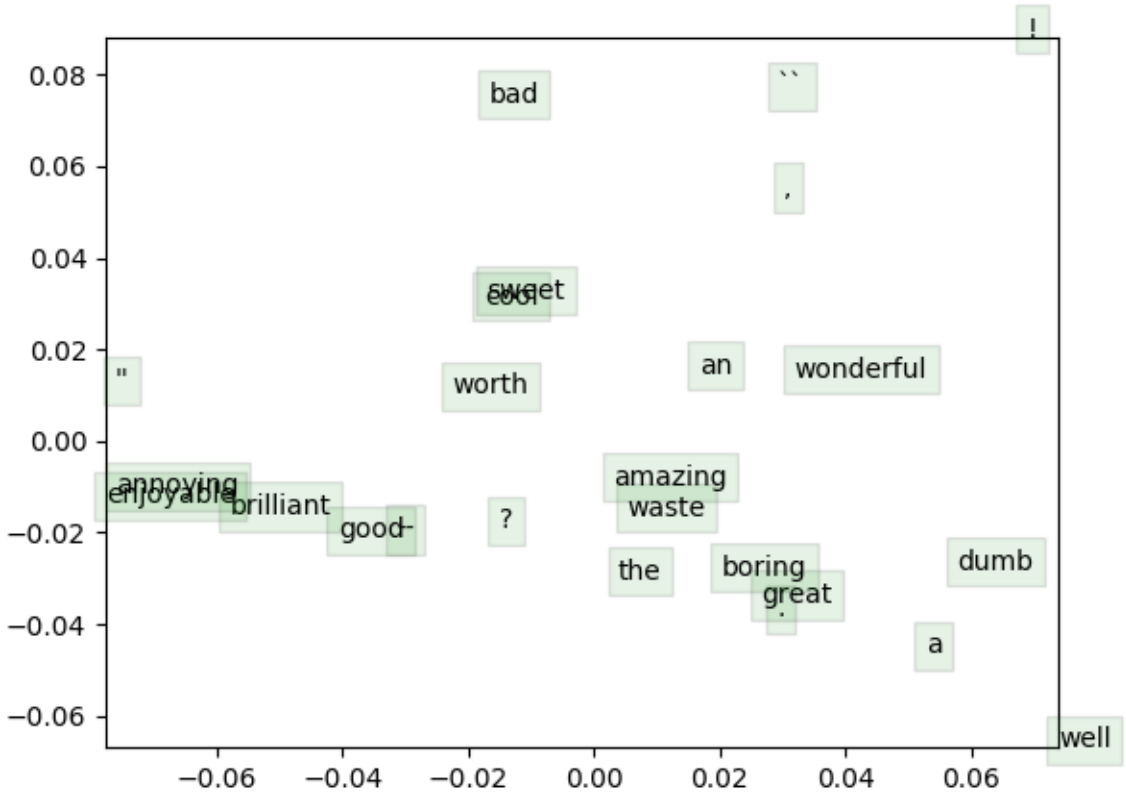
For the outer vectors, the equation is the same, with U replacing v_c.

5. **Implement the word2vec models and train your own word vectors with stochastic gradient descent**

We begin by normalizing the rows of a matrix to have unit length, with the code in `normalizeRows`. The costs and gradients derived in Parts 3.1 and 3.2 are then computed in `softmaxCostAndGradient`. Next we compute the gradient and cost associated with the negative sampling loss, as derivced in Part 3.4, in `getNegativeSamples`. Finally, we implement the cost and gradients of the skip-gram model, as derived in Question 3.4, in `skipgram`. Throughout, we've tried to keep the parameter names consistent with variable names in the corresponding derivations.

6. **Train word vectors with everything you just implemented!**

A plot of our trained word vectors is below. If anything, it appears that our vectors are co-locating antonyms, such as amazing/waste, boring/great, enjoyable/annoying.



4 Sentiment Analysis

1. **Implement a sentence featurizer.**

Our sentence features will simply be the average of the word vectors that make up the sentence. We calculate the features in `getSentenceFeatures` in `q4_sentiment.py`.

2. **Explain why we regularize our model**

Without requlariza tion, our models will overfit the training data. This will reduce test accuracy, which is the performance metric we really care about.

3. **Fill in the hyperparameter selection code**

This is a common pattern for returning the maximum value in a list. For details, see `chooseBestModel`.

4. **Train a model using your word vectors and pre-trained vectors**

We achieve 37% accuracy using pre-trained vectors, and only 28% accuracy using our vectors. The latter is only 12% better than random guessing.

5. **Plot the classification accuracy on the train and dev set with respect to the regularization value for the pretrained GloVe vectors**

Model complexity decreases as regularization increases. We can see that our model overfits without regularization, because the generalization error (orange line) increases as regularization increases. What is unexpected is that our training error also declines as model complexity declines.