Solution to Assignment 1, Problem 1(a)

Gregory King

Softmax Prove that softmax (denoted softmax($\mathbf{x}$)) is invariant to a constant offset in the input, that is, for any input vector $\mathbf{x}$ and any constant c,

$$\text{softmax}(\mathbf{x}) = \text{softmax}(\mathbf{x} + c), \tag{1}$$

where $(\mathbf{x} + c)$ means adding the constant c to every dimension of $\mathbf{x}$.

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

Starting from the definition of softmax,

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1} e^{x_j}} \tag{2}$$

it follows, the linear shift, $\mathbf{x} + c$, yields the following (for the i element):

$$\begin{aligned}
\text{softmax}(\mathbf{x} + c)_i &= \frac{e^{(x_i + c)}}{\sum_{j=1} e^{(x_j + c)}} && (3)\\
&= \frac{e^c}{e^c} \frac{e^{(x_i)}}{\sum_{j=1} e^{(x_j)}} && (4)\\
&= \text{softmax}(\mathbf{x})_i && (5)
\end{aligned}$$

since this is true for an arbitary i, it follows immediately that the linear shift has no affect on the softmax probabilities.

Solution to Assignment 1, Problem 1(b)

Gregory King

Given an input matrix of `N`-rows and `d`-columns, compute the softmax prediction for each row. 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. A non-vectorized implementation will not receive full credit!

```python
import numpy as np

def softmax(x):
    c = np.max(x, axis=x.ndim - 1, keepdims=True)
    #for numerical stability
    y = np.sum(np.exp(x - c), axis=x.ndim - 1, keepdims=True)
    x = np.exp(x - c)/y
    return x
```

Solution to Assignment 1, Problem 2(a)

Gregory King

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.

Denote the sigmoid function as $\sigma(z)$,

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{6}$$

It follows, using the chain rule (and noting that $e^{-z} = 1/\sigma(z) - 1$),

$$\begin{aligned}
\sigma'(z) &= \frac{-1}{(1 + e^{-z})^2} \times (-e^{-z}) \\
&= \sigma^2(z) \left(\frac{1}{\sigma(z)} - 1 \right) \\
&= \sigma(z) - \sigma^2(z) \\
&= \sigma(z)(1 - \sigma(z))
\end{aligned}$$

Solution to Assignment 1, Problem 2(b)

Gregory King

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 θ, when the prediction is made by $\hat{y} = \text{softmax}(\theta)$. Remember the cross entropy function is

$$\text{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log \hat{y}_i \tag{7}$$

where y is the one-hot label vector, and $\hat{y}$ is the predicted probability vector for all classes.

Hint: you might want to consider the fact many elements of y are zeros, and assume that only the k-th dimension of y is one.

Starting with, cross entropy,

$$\text{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log \hat{y}_i \tag{8}$$

We can derive the gradient of the cross-entropy function, using back propagation,

$$\frac{\partial(\text{CE})}{\partial \hat{y}_i} = -\frac{y_j}{\hat{y}_i} \tag{9}$$

Thus,

$$\frac{\partial(\text{CE})}{\partial \theta_k} = \frac{\partial(\text{CE})}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial \theta_k} \tag{10}$$

$$= -\frac{y_j}{\hat{y}_i} \frac{\partial \hat{y}_i}{\partial \theta_k} \tag{11}$$

Calculating the partial derivative of $\hat{y}_i$ (where $i = k$)

$$\frac{\partial \hat{y}_i}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} \left(\frac{e^{\theta_i}}{\sum_{j=1} e^{\theta_j}} \right) \tag{12}$$

$$= \frac{e^{\theta_i}}{\sum_{j=1} e^{\theta_j}} - \left(\frac{e^{\theta_i}}{\sum_{j=1} e^{\theta_j}} \right)^2 \tag{13}$$

$$= \hat{y}_i \cdot (1 - \hat{y}_i) \tag{14}$$

and (where $i \neq k$),

$$\frac{\partial \hat{y}_i}{\partial \theta_k} = \frac{\partial}{\partial \theta_k} \left(\frac{e^{\theta_i}}{\sum_{j=1} e^{\theta_j}} \right) \tag{15}$$

$$= - \left(\frac{e^{\theta_i} e^{\theta_k}}{\sum_{j=1} e^{\theta_j}} \right) \tag{16}$$

$$= - \hat{y}_i \hat{y}_k \tag{17}$$

Combining Equations 9, 14, 17, yields

$$\frac{\partial (\text{CE})}{\partial \theta_k} = \begin{cases} -y_j(1 - \hat{y}_k) & \text{for } i = k \\ y_j \hat{y}_k & \text{for } i \neq k \end{cases} \tag{18}$$

Requiring y_j to be non-zero, imposes that the auxiliary condition, $k = j$ and $y_j = 1$, hence it follows immediately,

$$\frac{\partial (\text{CE})}{\partial \theta_j} = \begin{cases} (\hat{y}_j - 1) & \text{for } i = j \\ \hat{y}_j & \text{for } i \neq j \end{cases} \tag{19}$$

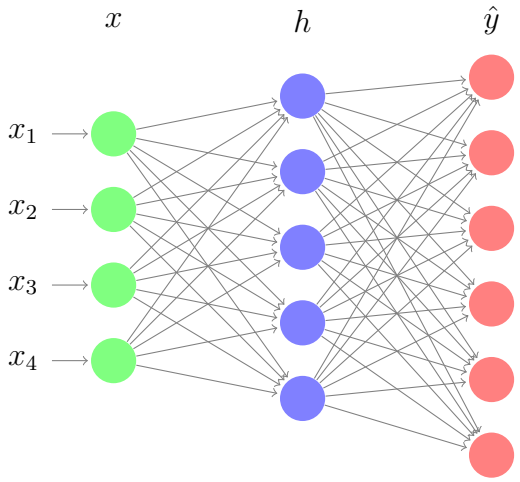
Which is equivalent to (with the substitution y_j for 1 in the first case of Equation 19),

$$\frac{\partial (\text{CE})}{\partial \boldsymbol{\theta}} = \hat{\mathbf{y}} - \mathbf{y} \tag{20}$$

Solution to Assignment 1, Problem 2(c)

Gregory King

Derive the gradients with respect to the inputs x to an one-hidden-layer neural network (that is, find $\partial J/\partial \mathbf{x}$ where J 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 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:

$$\mathbf{h} = \text{sigmoid}(\mathbf{x}\mathrm{W_1} + \mathbf{b_1}) \tag{21}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{h}\mathrm{W_2} + \mathbf{b_2}) \tag{22}$$

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

In order to simplify the notation used to solve the problem, define the following terms:

$$\mathbf{x}^{(2)} \equiv \mathbf{h} \tag{23}$$

$$\mathbf{z}_i \equiv \mathbf{x}^{(\mathbf{i})}\mathrm{W_i} + \mathbf{b_i} \tag{24}$$

Now, to calculate $\partial J/\partial \mathbf{x^1}$, one can use the back propagation algorithm. Starting with the results from Question 2(b):

$$\frac{\partial J}{\partial \mathbf{z}_2} = \hat{\mathbf{y}} - \mathbf{y} \tag{25}$$

and

$$\frac{\partial \mathbf{z}_i}{\partial \mathbf{x}^{(i)}} = \mathrm{W}_\mathbf{i}^\top \tag{26}$$

Sigmoid (σ) derivative can be found in Question 2(a), but we define:

$$\frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{z}_1} \equiv \sigma'(z_1) \tag{27}$$

Combining these, and using $\cdot$ to denote element-wise product:

$$\frac{\partial J}{\partial z_i} = (\hat{\mathbf{y}} - \mathbf{y})\mathrm{W}_\mathbf{2}^\top \cdot \sigma'(\mathbf{z_1}) \tag{28}$$

Finally, using the results from Equation 26 (but for the first layer):

$$\frac{\partial J}{\partial \mathbf{x}^{(1)}} = (\hat{\mathbf{y}} - \mathbf{y})\mathrm{W}_\mathbf{2}^\top \cdot \sigma'(\mathbf{z_1}) \cdot \mathrm{W}_\mathbf{1}^\top \tag{29}$$

Solution to Assignment 1, Problem 2(d)

Gregory King

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?

W_1 must have dimensions: $D_x \times H$. The bias ($\mathbf{b_1}$) for the first layer must have dimensions H. Adding these two together, yields $(D_x + 1) \times H$. Proceeding to the second layer, there must be $H \times D_y$ parameters associated with the weight matrix W_2. The bias ($\mathbf{b_2}$) for the second layer must have dimensions D_y elements. This yields,

$$(D_x + 1) \times H + D_y \times (H + 1) \tag{30}$$

weights, for each input vector of dimensions D_x.

Solution to Assignment 1, Problem 2(e)

Gregory King

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

```python
import numpy as np

def sigmoid(x):
    """
    Compute the sigmoid function for the input here.
    """
    x = 1 / (1 + np.exp(-x))
    return x

def sigmoid_grad(f):
    """
    Compute the gradient for the sigmoid function here. Note that
    for this implementation, the input f should be the sigmoid
    function value of your original input x.
    """
    f *= (1-f)
    return f
```

Solution to Assignment 1, Problem 2(f)

Gregory King

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`

```
import numpy as np
import random

def gradcheck_naive(f, x):
    """
    Gradient check for a function f
    - f should be a function that takes a single argument and
      outputs the cost and its gradients
    - x is the point (numpy array) to check the gradient at
    """
    rndstate = random.getstate()
    random.setstate(rndstate)
    fx, grad = f(x) # Evaluate function value at original point
    h = 1e-4

    # Iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        ix = it.multi_index
        ### YOUR CODE HERE:
        old_xix = x[ix]
        x[ix] = old_xix + h
        random.setstate(rndstate)
        fp = f(x)[0]
        x[ix] = old_xix - h
        random.setstate(rndstate)
        fm = f(x)[0]
        x[ix] = old_xix

        numgrad = (fp - fm)/(2* h)
        ### END YOUR CODE

        # Compare gradients
        reldiff = abs(numgrad - grad[ix]) / max(1, abs(numgrad), abs(grad[ix]))
        if reldiff > 1e-5:
```

```
            print("Gradient check failed.")
            print("First gradient error found at index %s" % str(ix))
            print("Your gradient: %f \t Numerical gradient: %f" % (grad[ix], nu
            return

        it.iternext() # Step to next dimension

    print("Gradient check passed!")
```

Solution to Assignment 1, Problem 2(g)

Gregory King

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 `q2_neural.py`.

```
import numpy as np

from q1_softmax import softmax
from q2_sigmoid import sigmoid, sigmoid_grad

def forward_backward_prop(data, labels, params, dimensions):
    """
    Forward and backward propagation for a two-layer sigmoidal network

    Compute the forward propagation and for the cross entropy cost,
    and backward propagation for the gradients for all parameters.
    """

    ### Unpack network parameters (do not modify)
    ofs = 0
    Dx, H, Dy = (dimensions[0], dimensions[1], dimensions[2])
    N = data.shape[0]

    W1 = np.reshape(params[ofs:ofs+ Dx * H], (Dx, H))
    ofs += Dx * H
    b1 = np.reshape(params[ofs:ofs + H], (1, H))
    ofs += H
    W2 = np.reshape(params[ofs:ofs + H * Dy], (H, Dy))
    ofs += H * Dy
    b2 = np.reshape(params[ofs:ofs + Dy], (1, Dy))

    ### YOUR CODE HERE: forward propagation
    layer1   = np.dot(data,W1) + b1
    layer1_a = sigmoid(layer1)
    layer2   = np.dot(layer1_a, W2) + b2
    # need to calculate the softmax loss
    probs = softmax(layer2)
    cost  = -np.sum(np.log(probs[np.arange(N), np.argmax(labels, axis=1)])) / N
    dx    = probs.copy()
    dx[np.arange(N), np.argmax(labels, axis=1)] -= 1
```

```
    dx /= N
    # dx is the gradient of the loss w.r.t. y_{est}
    ### END YOUR CODE

    ### YOUR CODE HERE: backward propagation
    #There is no regularization :/
    # dx -> sigmoid -> W2 * layer1_a + b -> sigmoid -> W1 * data + b1 -> ..
    dlayer2   = np.zeros_like(dx)
    gradW2    = np.zeros_like(W2)
    gradW1    = np.zeros_like(W1)
    gradb2    = np.zeros_like(b2)
    gradb1    = np.zeros_like(b1)

    gradW2    = np.dot(layer1_a.T, dx)
    gradb2    = np.sum(dx, axis=0)
    dlayer2   = np.dot(dx, W2.T)
    dlayer1   = sigmoid_grad(layer1_a) * dlayer2
    gradW1    = np.dot(data.T, dlayer1)
    gradb1    = np.sum(dlayer1, axis=0)
    ### END YOUR CODE

    ### Stack gradients (do not modify)
    grad = np.concatenate((gradW1.flatten(), gradb1.flatten(),
        gradW2.flatten(), gradb2.flatten()))

    return cost, grad
```

Solution to Assignment 1, Problem 3(a)

Gregory King

Assume you are given a predicted word vector $\boldsymbol{v}_c$ corresponding to the center word c for `skipgram`, and word prediction is made with the `softmax` function found in `word2vec` models

$$\hat{\boldsymbol{y}}_{\boldsymbol{o}} = p(\ \boldsymbol{o} \mid \boldsymbol{c}\ ) = \frac{\exp\left(\boldsymbol{u}_o^\top \boldsymbol{v}_c\right)}{\sum_{j=1}^{|W|} \exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_c\right)} \tag{31}$$

where w denotes the w-th word and $\boldsymbol{u}_w$ ($w = 1, ..., |\mathrm{W}|$) are the 'output' word vectors for all words in the vocabulary (v'_w in the lecture notes). Assume the 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 the softmax prediction for every word, $\boldsymbol{y}$ as the expected word vector, and the loss function

$$J_{\text{softmax-CE}}(\boldsymbol{o}, \boldsymbol{v}_c, \boldsymbol{U}) = CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{32}$$

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

Applying cross-entropy cost to the above softmax probability defined above:

$$J = -\log p = -\boldsymbol{u}_o^\top \boldsymbol{v}_c + \log \sum_{j=1}^{|V|} \exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_c\right) \tag{33}$$

Let $z_j = \boldsymbol{u}_j^\top \boldsymbol{v}_c$, and δ_j^i be the indicator function (Kronecker delta), then

$$\frac{\partial J}{\partial z_k} = -\delta_k^i + \frac{\exp\left(\boldsymbol{u}_i^\top \boldsymbol{v}_c\right)}{\sum_{j=1}^{|V|} \exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_c\right)} \tag{34}$$

Now, using the chain rule, we can calculate,

$$\frac{\partial J}{\partial \boldsymbol{v}_c} = \frac{\partial J}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{v}_c} \tag{35}$$

$$= \sum_{j=1}^{|V|} \boldsymbol{u}_j^\top \left(\frac{e^{z_j}}{\sum_{k=1}^{|V|} e^{z_k}} - 1 \right) \tag{36}$$

$$= \sum_{k=1}^{|V|} \boldsymbol{P}(\boldsymbol{u}_j | \boldsymbol{v}_c) \boldsymbol{u}_j - \boldsymbol{u}_j \tag{37}$$

Solution to Assignment 1, Problem 3(b)

Gregory King

As in the previous problem, derive gradients for the 'output' word vectors $\boldsymbol{u}_w$'s (including $\boldsymbol{u}_o$)

This follows immediately from the chain rule:

$$\frac{\partial J}{\partial \boldsymbol{u}_j} = \frac{\partial J}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{u}_j} \tag{38}$$

$$= \boldsymbol{v}_c \left(\frac{\exp\left(\boldsymbol{u}_0^\top \boldsymbol{v}_c\right)}{\sum_{j=1}^{|V|} \exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_c\right)} - \delta_j^0 \right) \tag{39}$$

Solution to Assignment 1, Problem 3(c)

Gregory King

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 is $\boldsymbol{o}(\boldsymbol{u}_o)$. Assume that K negative samples (words) are drawn, and they are $\boldsymbol{u}_1, ..., \boldsymbol{u}_k$, respectively for simplicity of notation ($k \in \{1, ..., K\}$ and $o \notin \{1, ..., K\}$). Again for a given word, $\boldsymbol{o}$, denote its output vector as $\boldsymbol{u}_o$. The negative sampling loss function in this case is,

$$J(\boldsymbol{u}_o, \boldsymbol{v}_c, \boldsymbol{U}) = -\log(\sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c)) - \sum_{k=1}^{K} \log(\sigma(-\boldsymbol{u}_k^\top \boldsymbol{v}_c)) \tag{40}$$

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.

Define $z_i \equiv \boldsymbol{w}_i^\top \hat{\boldsymbol{r}}$ and let δ_j^i be the Kronecker delta. If $i \notin \{1, ..., K\}$

$$\frac{\partial J}{\partial z_i} = -\frac{\partial}{\partial z_i}(\log(\sigma(z_i)) = -\frac{\sigma'(z_i)}{\sigma(z_i)} \tag{41}$$

$$= -\frac{\sigma(z_i)(1-\sigma(z_i))}{\sigma(z_i)} \tag{42}$$

$$= \sigma(z_i) - 1 \tag{43}$$

If $i \in \{1, ..., K\}$ then we need to take the derivative of the second term. It follows,

$$\frac{\partial J}{\partial z_i} = -\frac{\partial}{\partial z_i}\log(\sigma(-z_i)) = -\frac{\sigma'(-z_i)}{\sigma(-z_i)} \tag{44}$$

$$= \frac{\sigma(-z_i)(1-\sigma(-z_i))}{\sigma(-z_i)} \tag{45}$$

$$= 1 - \sigma(-z_i) = \sigma(z_i) \tag{46}$$

Which yields the following, where $j \in \boldsymbol{o}, 1, ..., K$,

$$\frac{\partial J}{\partial z_i} = \sigma(z_i) - \delta_0^i$$

$$\frac{\partial J}{\partial \boldsymbol{u}_j} = (\sigma(z_j) - \delta_0^j)\boldsymbol{v}_c \tag{47}$$

$$\frac{\partial J}{\partial \boldsymbol{v}_c} = (\sigma(z_0) - 1)\boldsymbol{u}_0 - \sum_{j=1}^{K}(\sigma(-z_j) - 1)\boldsymbol{u}_j \tag{48}$$

Solution to Assignment 1, Problem 3(d)

Gregory King

Derive the gradients for all of the word vectors for `skip-gram` and `CBOW` (optional) given a set of context words $[\texttt{word}_{c-\mathbf{m}}, ..., \texttt{word}_{c-\mathbf{1}}, \texttt{word}_c, \texttt{word}_{c+\mathbf{1}}, ..., \texttt{word}_{c+\mathbf{m}}]$, where $\mathbf{m}$ is the context size. You can denote the 'input' and 'output' word vectors for $\texttt{word}_k$ as $\boldsymbol{v}_k$ and $\boldsymbol{u}_k$ respectively for convenience.

Hint: feel free to use $F(\boldsymbol{o}, \boldsymbol{v}_c)$ (where $\boldsymbol{o}$ is the expected word) as a placeholder for $J_{\texttt{softmax-CE}}(\boldsymbol{o}, \boldsymbol{v}_c, ...)$ or $J_{\texttt{neg-sampling}}(\boldsymbol{o}, \boldsymbol{v}_c, ...)$ 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 $\frac{\partial F_i(\boldsymbol{o}, \boldsymbol{v}_c)}{\partial \lambda}$.

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

$$J_{\texttt{skip-gram}} = \sum_{-m \le j \le m, j \neq 0} F(\boldsymbol{w}_{c+j}, \boldsymbol{v}_c) \tag{49}$$

where $\boldsymbol{w}_{c+j}$ refers to the word at the j-th index from the center.

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

$$\hat{\boldsymbol{v}} = \sum_{-m \le j \le m, j \neq 0} \boldsymbol{v}_{c+j} \tag{50}$$

then the `CBOW` cost is

$$J_{\texttt{CBOW}}(\texttt{word}_{c-\mathbf{m}}, ..., \texttt{word}_{c+\mathbf{m}}) = F(\boldsymbol{w}_c, \hat{\boldsymbol{v}}) \tag{51}$$

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

The `skip-gram` model involves summing over context (and treating each expected word as being associated with a conditional probability of $\boldsymbol{v}_c$). This is,

$$J_{\texttt{skip-gram}} = \sum_{-m \le j \le m, j \neq 0} F(\boldsymbol{w}_{c+j}, \boldsymbol{v}_c) \tag{52}$$

The derivative of the loss has two terms, $\boldsymbol{w}_{c+j}$ and $\boldsymbol{v}_c$, which yields the following,

$$\frac{\partial J_{\texttt{skip-gram}}}{\partial \boldsymbol{w}_k} = \frac{\partial}{\partial \boldsymbol{w}_k} \sum_{-m \le j \le m, j \neq 0} F(\boldsymbol{w}_{c+j}, \boldsymbol{v}_c) \qquad = \sum_{-m \le j \le m, j \neq 0} \frac{\partial F}{\partial \boldsymbol{w}_{i+j}} \delta_k^{i+j} \tag{53}$$

and

$$\frac{\partial J_{\texttt{skip-gram}}}{\partial \boldsymbol{v}_c} = \sum_{-m \le j \le m, j \neq 0} \frac{\partial F}{\partial \boldsymbol{v}_c} \tag{54}$$

Whereas in the `CBOW` model, the summing over context is handled internally to the probability model (in other words, the context vector is represented by the sum of the word vectors and the conditional probability is with respect to this sum). This yields,

$$\frac{\partial J_{\texttt{CBOW}}}{\partial \boldsymbol{w}_i} = \frac{\partial F}{\partial \boldsymbol{w}_i} \tag{55}$$

and

$$\frac{\partial J_{\texttt{CBOW}}}{\partial \hat{\boldsymbol{r}}} \frac{\partial \hat{\boldsymbol{r}}}{\partial \boldsymbol{w}_k} = \frac{\partial F}{\partial \hat{\boldsymbol{r}}} \sum_{-m \le j \le m, j \neq 0} \frac{\partial \hat{\boldsymbol{r}}}{\partial \boldsymbol{w}_j} \delta_k^j \tag{56}$$

It is important to notice that the above is equivalent to doing a one-hot encoding of the `CBOW` vector (and applying the gradient to all components of the `CBOW` vector).

Solution to Assignment 1, Problem 3(e)

Gregory King

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 comment out the `raise NotImplementedError` so that your tests will complete.

```python
import numpy as np
import random

from q1_softmax import softmax
from q2_gradcheck import gradcheck_naive
from q2_sigmoid import sigmoid, sigmoid_grad

def normalizeRows(x):
    """ Row normalization function """
    # Implement a function that normalizes each row of a matrix to have unit le

    ### YOUR CODE HERE
    y = np.linalg.norm(x,axis=1,keepdims=True)
    x /= y
    ### END YOUR CODE
    return x
```

Solution to Assignment 1, Problem 3(f)

Gregory King

Complete the implementation for your `SGD` optimizer in `q3_sgd.py`. Test your implementation by running `python q3_sgd.py`.

```
import numpy as np

def sgd(f, x0, step, iterations,
            postprocessing = None,
            useSaved = False,
            PRINT_EVERY=10):
    """ Stochastic Gradient Descent """
    # Implement the stochastic gradient descent method in this
    # function.

    # Inputs:
    # - f: the function to optimize, it should take a single
    #     argument and yield two outputs, a cost and the gradient
    #     with respect to the arguments
    # - x0: the initial point to start SGD from
    # - step: the step size for SGD
    # - iterations: total iterations to run SGD for
    # - postprocessing: postprocessing function for the parameters
    #     if necessary. In the case of word2vec we will need to
    #     normalize the word vectors to have unit length.
    # - PRINT_EVERY: specifies every how many iterations to output

    # Output:
    # - x: the parameter value after SGD finishes

    # Anneal learning rate every several iterations
    ANNEAL_EVERY = 20000

    if useSaved:
        start_iter, oldx, state = load_saved_params()
        if start_iter > 0:
            x0 = oldx;
            step *= 0.5 ** (start_iter / ANNEAL_EVERY)

        if state:
```

```
            random.setstate(state)
    else:
        start_iter = 0

    x = x0

    if not postprocessing:
        postprocessing = lambda x: x

    expcost = None

    for iter in range(start_iter + 1, iterations + 1):
        ### Don't forget to apply the postprocessing after every iteration!
        ### You might want to print the progress every few iterations.

        cost = None
        ### YOUR CODE HERE
        x = postprocessing(x)
        cost, grad = f(x)
        x -= step * grad
        ### END YOUR CODE

        if iter % PRINT_EVERY == 0:
            if not expcost:
                expcost = cost
            else:
                expcost = .95 * expcost + .05 * cost
            print("iter %d: %f" % (iter, expcost))

        if iter % SAVE_PARAMS_EVERY == 0 and useSaved:
            save_params(iter, x)

        if iter % ANNEAL_EVERY == 0:
            step *= 0.5

    return x
```

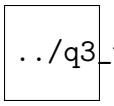


Figure 1: Plot of the first two components of 10 dimensional $\mathcal{R}^n$ word embedding for the **SST**. The words `a`, `the`, '.', ',' appear to have a distinct location away from the adjectives.

Solution to Assignment 1, Problem 3(g)

Gregory King

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 sentiment analysis task. 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 you 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.

Solution to Assignment 1, Problem 3(h)

Gregory King

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

Solution to Assignment 1, Problem 4

Gregory King

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 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 regressor with SGD, and perform train/dev validation to imporve generalization of your regressor.

Solution to Assignment 1, Problem 4(a)

Gregory King

Implement a sentence featurizer and softmax regression. Fill in the implementation in `q4_softmaxreg.py`. Sanity check your implementation with `python q4_softmaxreg.py`.

Solution to Assignment 1, Problem 4(b)

Gregory King

Explain in fewer than three sentences why we want to introduce regularization when doing classification (in fact, most machine learning tasks).

Regularization helps prevent over-fitting. By limiting the parameter space it forces the model to have a higher chance of learning features (rather than memorizing the training data).

Solution to Assignment 1, Problem 4(c)

Gregory King

Fill in the hyperparameter selection code in `q4_sentiment.py` to search for the 'optimal' regularization parameter. **What values did you select? Report your train, dev, and test accuracies. Justify your hyperparameter search methodology in at most one sentence.**.

Note: you should be able to attain at least 30% accuracy on dev.

Reg	Train	Dev
0.00e+00	28.68	25.89
1.00e-06	28.68	25.89
2.14e-06	28.68	25.89
4.57e-06	28.65	25.89
9.77e-06	28.64	25.79
2.09e-05	28.63	25.7
4.47e-05	28.65	25.89
9.55e-05	28.5	26.07
2.04e-04	28.04	26.16
4.37e-04	27.43	25.43
9.33e-04	27.22	25.43
2.00e-03	27.22	25.52
4.27e-03	27.24	25.52
9.12e-03	27.25	25.52
1.95e-02	27.25	25.52
4.17e-02	27.25	25.52
8.91e-02	27.25	25.52
1.91e-01	27.25	25.52
4.07e-01	27.25	25.52
8.71e-01	27.25	25.52
1.86e+00	27.25	25.52
3.98e+00	27.25	25.52

Solution to Assignment 1, Problem 4(d)

Gregory King

Plot the classification accuracy on the train and dev set with respect to the regularization value, 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.