HW1 Maitar Asher, ma4205

1 Written Problems

1.1 Hard-Coding Networks: Verify Element in List [10 pts]

The reading on multilayer perceptrons located at https://www.cs.columbia.edu/~zemel/Class/Nndl/files/Grosse-Multilayer-Perceptrons.pdf may be useful for this question.

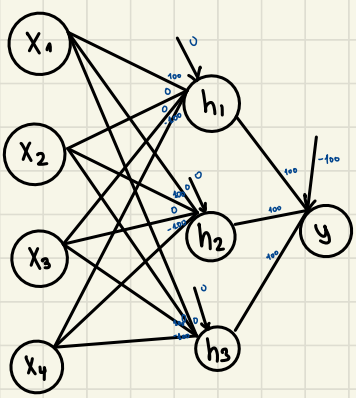
In this problem, you need to find a set of weights and biases for a multilayer perceptron that determines if an input is in a list of length 3. You receive an input list with three elements x_1, x_2, x_3, and fourth input x_4 where $x_i \in \mathbb{Z}$. If $x_4 = x_j$ for $j = 1, 2$ or 3, the network should output 1, and 0 otherwise. You may assume all elements in the input list are distinct for simplicity. You will use the following architecture: All of the hidden units and the output unit use an indicator activation function:

$$\phi(z) = \mathbb{I}(z \in [-1, 1]) = \begin{cases} 1 & \text{if } z \in [-1, 1] \\ 0 & \text{otherwise} \end{cases} \rightarrow -1, 0, 1$$

Please give a set of weights and biases for the network which correctly implements this function. Your answer should include:

- A 3×4 weight matrix $\mathbf{W}^{(1)}$ for the hidden layer
- A 3-dimensional vector of biases $\mathbf{b}^{(1)}$ for the hidden layer
- A 3-dimensional weight vector $\mathbf{w}^{(2)}$ for the output layer
- A scalar bias $b^{(2)}$ for the output layer

You do not need to show your work.



$$W^{(1)}_{3\times4} = \begin{bmatrix} 100 & 0 & 0 & -100 \\ 0 & 100 & 0 & -100 \\ 0 & 0 & 100 & -100 \end{bmatrix} \qquad b^{(1)}_{3\times1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$W^{(2)}_{3\times1} = \begin{bmatrix} 100 \\ 100 \\ 100 \end{bmatrix} \qquad b^{(2)} = -100$$

tests

* $X = \begin{bmatrix} 100 \\ 0 \\ 99 \\ -100 \end{bmatrix}$ $\quad W^{(1)}X + b^{(1)} = \begin{bmatrix} 20K \\ 10K \\ 19.9K \end{bmatrix} \xrightarrow{\phi(\cdot)} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \rightsquigarrow \phi(0-100) = \phi(-100) = 0$

* $X = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 1 \end{bmatrix}$ $\quad W^{(1)}X + b^{(1)} = \begin{bmatrix} 100 \\ -100 \\ 0 \end{bmatrix} \xrightarrow{\phi(\cdot)} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \rightsquigarrow \phi(100-100) = \phi(0) = 1$

1.2 Backpropagation

The reading on backpropagation located at `https://www.cs.columbia.edu/~zemel/Class/Nndl/files/Grosse-Backpropagation.pdf` may be useful for this question.

1.2.1

lets say # of hidden units is U
# of features (input) is d
# of classes is n

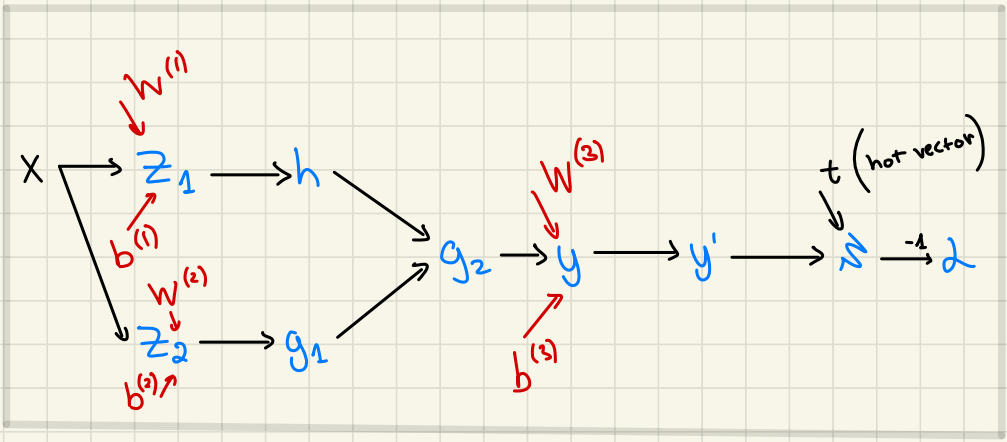
Consider a neural network defined with the following procedure:

$$\mathbf{z}_1 = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \quad (u \times d)(d \times 1) + (u \times 1) \rightarrow u \times 1$$
$$\mathbf{h} = \text{ReLU}(\mathbf{z}_1) \rightarrow u \times 1$$
$$\mathbf{z}_2 = \mathbf{W}^{(2)}\mathbf{x} + \mathbf{b}^{(2)} \rightarrow u \times 1$$
$$\mathbf{g}_1 = \sigma(\mathbf{z}_2) \rightarrow u \times 1$$
$$\mathbf{g}_2 = \mathbf{h} \odot \mathbf{g}_1 \rightarrow (\text{Hadamard product, component wise multiplication}) \quad u \times 1 \odot u \times 1$$
$$\mathbf{y} = \mathbf{W}^{(3)}\mathbf{g}_2 + \mathbf{b}^{(3)} \quad (n \times u)(u \times 1) + (n \times 1) \rightarrow n \times 1$$
$$\mathbf{y}' = \text{softmax}(\mathbf{y}) \rightarrow n \times 1$$
$$\mathcal{S} = \sum_{k=1}^{N} \mathbb{I}(t_k = 1) \log(\mathbf{y}'_k)$$
$$\mathcal{J} = -\mathcal{S}$$

for input $\mathbf{x}$ with class label t where $\text{ReLU}(\mathbf{z}) = \max(\mathbf{z}, 0)$ denotes the ReLU activation function, $\sigma(\mathbf{z}) = \frac{1}{1+e^{-\mathbf{z}}}$ denotes the Sigmoid activation function, both applied elementwise, and $\text{softmax}(\mathbf{y}) = \frac{\exp(\mathbf{y})}{\sum_{i=1}^{N}\exp(\mathbf{y}_i)}$. Here, $\odot$ denotes element-wise multiplication.

Computation Graph [5 pts]: Draw the computation graph relating $\mathbf{x}$, t, $\mathbf{z}_1$, $\mathbf{h}$, $\mathbf{z}_2$, $\mathbf{g}_1$, $\mathbf{g}_2$, $\mathbf{y}$, $\mathbf{y}'$, $\mathcal{S}$ and $\mathcal{J}$.

Backward Pass [5 pts]: Derive the backprop equations for computing $\bar{\mathbf{x}} = \frac{\partial \mathcal{J}}{\partial \mathbf{x}}$, one variable at a time, similar to the vectorized backward pass derived in Lec 2.



$\bar{L} = 1$ $\quad \bar{S} = -1$

$\bar{y'} = \bar{S}\left(t \oslash y'\right)$ // $(n \times 1)$ $\quad \left(\bar{y}'_k = \bar{S} \cdot (t_k) \cdot \left(\frac{1}{\ln(e)} \cdot \frac{1}{y'_k} \right) = \bar{S}(t_k)\left(\frac{1}{y'_k}\right) \right)$ $\quad t_k \in \{0,1,2,3\}$

↳ (this is element wise division)

$\bar{y} = \bar{y}' \odot (\text{Softmax}'(y))$ *

$\overline{W^{(3)}} = \bar{y}\left(g_2^T\right)$ // $(n \times 1)(1 \times u) = (n \times 1)$ $\quad \bar{b} = \bar{y}$ // $(n \times 1)$

$\overline{g_2} = W^{(3)T}\,\bar{y}$ // $(u \times n)(n \times 1) = (u \times 1)$

$\overline{g_1} = \overline{g_2} \odot h$ $\quad \bar{h} = \overline{g_2} \odot g_1$ // $(u \times 1)$

$\overline{z_1} = \bar{h} \odot (\text{Relu}'(z_1))$ $\quad \overline{z_2} = \overline{g_1} \odot \sigma'(z_2)$ // $(u \times 1)$

$\overline{W^{(1)}} = \overline{z_1}\, x^T$ // $(u \times 1)(1 \times d) = (u \times d)$ $\quad \overline{b^{(1)}} = \overline{z_1}$ // $(u \times 1)$

$\overline{W^{(2)}} = \overline{z_2}\, x^T$ $\quad \overline{b^{(2)}} = \overline{z_2}$

1.2.2 Automatic Differentiation

Consider the function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ where $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{v}\mathbf{v}^\top \mathbf{x}$, and $\mathbf{v} \in \mathbb{R}^{n\times 1}$ and $\mathbf{x} \in \mathbb{R}^{n\times 1}$. Here, we will explore the relative costs of evaluating Jacobians and vector-Jacobian products. Specifically, we will study vector-Hessian products, which is a special case of vector-Jacobian products, where the Jacobian is of the gradient of our function. We denote the gradient of $\mathcal{L}$ with respect to $\mathbf{x}$ as $\mathbf{g} \in \mathbb{R}^{1\times n}$ and the Hessian of $\mathcal{L}$ w.r.t. $\mathbf{x}$ with $\mathbf{H} \in \mathbb{R}^{n\times n}$. The Hessian of $\mathcal{L}$ w.r.t. $\mathbf{x}$ is defined as:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial x_1^2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_n} \\ \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_2^2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_n^2} \end{pmatrix}$$

Compute Hessian [5 pts]: Compute $\mathbf{H}$ for $n = 3$ and $\mathbf{v}^\top = [4, 2, 3]$ at $\mathbf{x}^\top = [1, 2, 3]$. In other words, write down the numbers in:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial x_1^2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_3} \\ \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_2^2} & \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_3} \\ \frac{\partial^2 \mathcal{L}}{\partial x_3 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_3 \partial x_2} & \frac{\partial^2 \mathcal{L}}{\partial x_3^2} \end{pmatrix}$$

Scalar derivative			Vector derivative		
$f(x)$	$\to$	$\frac{df}{dx}$	$f(\mathbf{x})$	$\to$	$\frac{df}{d\mathbf{x}}$
bx	$\to$	b	$\mathbf{x}^T\mathbf{B}$	$\to$	$\mathbf{B}$
bx	$\to$	b	$\mathbf{x}^T\mathbf{b}$	$\to$	$\mathbf{b}$
x^2	$\to$	$2x$	$\mathbf{x}^T\mathbf{x}$	$\to$	$\mathbf{2x}$
bx^2	$\to$	$2bx$	$\mathbf{x}^T\mathbf{B}\mathbf{x}$	$\to$	$\mathbf{2Bx}$

$f \to \mathbb{R}^u \to \mathbb{R}^v$

$[v \times u][u \times 1] = [v \times 1]$

$L(x) = \frac{1}{2} x^T(vv^T)x = \frac{1}{2} x^T A x$

$vv^T = A$ $(n \times n)$

$\frac{\partial(L(x))}{\partial x} = \frac{1}{2} \cdot 2Ax = Ax = \begin{pmatrix} a_1^T x \\ a_2^T x \\ a_3^T x \end{pmatrix}$

Let a_1^T, a_2^T, a_3^T be the rows of matrix A

$$A = vv^T = \begin{pmatrix} 4 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 4 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 16 & 8 & 12 \\ 8 & 4 & 6 \\ 12 & 6 & 9 \end{pmatrix}$$

$$Ax = \begin{pmatrix} 16 & 8 & 12 \\ 8 & 4 & 6 \\ 12 & 6 & 9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 16 \cdot x_1 + 8 \cdot x_2 + 12 \cdot x_3 \\ 8 \cdot x_1 + 4 \cdot x_2 + 6 \cdot x_3 \\ 12 \cdot x_1 + 6 \cdot x_2 + 9 \cdot x_3 \end{pmatrix}$$

$$H = \begin{pmatrix} 16 & 8 & 12 \\ 8 & 4 & 6 \\ 12 & 6 & 9 \end{pmatrix}$$

Computation Cost [5 pts]: What is the number of scalar multiplications and memory cost to compute the Hessian $\mathbf{H}$ in terms of n? You may use big $\mathcal{O}$ notation.

Scalar multiplications → $O(n^2)$

Memory Cost (Space) → $O(n^2)$

1.3 Vector-Hessian Products [10 pts]

Compute $\mathbf{z} = \mathbf{H}\mathbf{y} = \mathbf{v}\mathbf{v}^\top\mathbf{y}$ where $n = 3$, $\mathbf{v}^\top = [1, 2, 3]$, $\mathbf{y}^\top = [1, 1, 1]$ using two algorithms: reverse-mode and forward-mode autodiff. In backpropagation (also known as reverse-mode autodiff), you will compute $\mathbf{M} = \mathbf{v}^\top\mathbf{y}$ first, then compute $\mathbf{v}\mathbf{M}$. Whereas, in forward-mode, you will compute $\mathbf{H} = \mathbf{v}\mathbf{v}^\top$ then compute $\mathbf{H}\mathbf{y}$. Write down the numerical values of $\mathbf{z}^T = [z_1, z_2, z_3]$ for the given $\mathbf{v}$ and $\mathbf{y}$. What is the number of scalar multiplications and memory cost in evaluating $\mathbf{z}$ with backpropagation (reverse-mode) in terms of n? What about forward-mode?

Reverse mode

$$M = v^T y = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = 6$$

- Scalar multiplications: $O(n)$
  - n for dot product +
  - n for scalar M multiplied by n elments
- Memory cost (space): $O(n)$

$$vM = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} (6) = \begin{pmatrix} 6 \\ 12 \\ 18 \end{pmatrix}$$

Forward mode

$$H = vv^T = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix}$$

- Scalar multiplications: $O(n^2)$
- Memory cost (space): $O(n^2)$

$$Hy = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 12 \\ 18 \end{pmatrix}$$

1.3.1 Trade-off of Reverse- and Forward-mode Autodiff [10 pts]

Consider computing $\mathbf{Z} = \mathbf{H}\mathbf{y}_1\mathbf{y}_2^\top$ where $\mathbf{v} \in \mathbb{R}^{n\times 1}, \mathbf{y}_1 \in \mathbb{R}^{n\times 1}$ and $\mathbf{y}_2 \in \mathbb{R}^{m\times 1}$. What are number of scalar multiplications and memory cost in evaluating $\mathbf{Z}$ with reverse-mode in terms of n and m? What about forward-mode? When is forward-mode a better choice?

$\mathbf{H}$ → $n\times n$

Reverse mode

- Scalar multiplications: $O(nm)$
- Memory Cost (space): $O(nm)$

$(y_1 y_2^T) \rightarrow \begin{pmatrix} \\ n\times 1 \\ \\ \end{pmatrix} \begin{pmatrix} 1\times m \end{pmatrix} \rightarrow nm$ multiplications

$v^T(y_1 y_2^T) \rightarrow (1\times n)\begin{pmatrix} \\ n\times m \\ \\ \end{pmatrix} \rightarrow \underset{\text{cols}}{m}\cdot\overset{\text{for each dot product}}{(n)} \rightarrow nm$ multiplications

$v(v^T y_1 y_2^T) \rightarrow \begin{pmatrix} \\ n\times 1 \\ \\ \end{pmatrix}^{(1\times m)} \rightarrow nm$ multiplications

Forward mode

- Scalar multiplications: $O(n^2 + nm)$
- Memory Cost (space): $O(n^2 + nm)$

$H \rightarrow \begin{pmatrix} \\ n\times 1 \\ \\ \end{pmatrix}^{(1\times n)} \rightarrow n^2$ multiplications

$(H)y_1 \rightarrow (n\times n)(n\times 1) \rightarrow n$ dot products (each dot n elements)
total n^2 multiplications

$(Hy_1)(y_2^T) \rightarrow \begin{pmatrix} \\ n\times 1 \\ \\ \end{pmatrix}^{(1\times m)} \rightarrow nm$ multiplications

Thus forward mode is a better choice only when $m > n$ (when m is a lot larger than n)

Programming Assignment 1: Learning Distributed Word Representations

Due Date: October 4, 2023 by 2pm

Submission: You must submit two files:

1. [ ] A PDF file containing your writeup, which will be the PDF export of this notebook (i.e., by printing this notebook webpage as PDF). Note that after you generate the PDF file of the Notebook, you will need to **concatenate it with the PDF file of your solutions to written problems**. Make sure that the relevant outputs (e.g. `print_gradients()` outputs, plots, etc.) are included and clearly visible. The **concatenated PDF file** needs to be submitted to **GradeScope**.
2. [ ] The `hw1_code_<YOUR_UNI>.ipynb` iPython Notebook, where `<YOUR_UNI>` is your uni. This file needs to be submitted to the **CourseWorks assignment page**.

The programming assignments are individual work.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

Introduction

In this assignment we will learn about word embeddings and make neural networks learn about words. We could try to match statistics about the words, or we could train a network that takes a sequence of words as input and learns to predict the word that comes next.

This assignment will ask you to implement a linear embedding and then the backpropagation computations for a neural language model and then run some experiments to analyze the learned representation. The amount of code you have to write is very short but each line will require you to think very carefully. You will need to derive the updates mathematically, and then implement them using matrix and vector operations in NumPy.

Name: Maitar Asher

UNI: ma4265

Starter code and data

First, perform the required imports for your code:

In [3]:
```
import collections
import pickle
import numpy as np
import os
from tqdm import tqdm
import pylab
from six.moves.urllib.request import urlretrieve
import tarfile
```

```
import sys

TINY = 1e-30
EPS = 1e-4
nax = np.newaxis
```

If you're using colaboratory, this following script creates a folder - here we used 'A1' - in order to download and store the data. If you're not using colaboratory, then set the path to wherever you want the contents to be stored at locally.

You can also manually download and unzip the data from [http://www.cs.columbia.edu/~zemel/Class/Nndl/files/a1_data.tar.gz] and put them in the same folder as where you store this notebook.

Feel free to use a different way to access the files *data.pk* , *partially_trained.pk*, and *raw_sentences.txt*.

The file *raw_sentences.txt* contains the sentences that we will be using for this assignment. These sentences are fairly simple ones and cover a vocabulary of only 250 words (+ 1 special `[MASK]` token word).

In [4]:
```
################################################################################
# Setup working directory
################################################################################
# Change this to a local path if running locally
#%mkdir -p /content/A1/
#%cd /content/A1

################################################################################
# Helper functions for loading data
################################################################################
# adapted from
# https://github.com/fchollet/keras/blob/master/keras/datasets/cifar10.py

def get_file(fname,
             origin,
             untar=False,
             extract=False,
             archive_format='auto',
             cache_dir='data'):
    datadir = os.path.join(cache_dir)
    if not os.path.exists(datadir):
        os.makedirs(datadir)

    if untar:
        untar_fpath = os.path.join(datadir, fname)
        fpath = untar_fpath + '.tar.gz'
    else:
        fpath = os.path.join(datadir, fname)

    print('File path: %s' % fpath)
    if not os.path.exists(fpath):
        print('Downloading data from', origin)

        error_msg = 'URL fetch failure on {}: {} -- {}'
        try:
            try:
                urlretrieve(origin, fpath)
            except URLError as e:
                raise Exception(error_msg.format(origin, e.errno, e.reason))
            except HTTPError as e:
                raise Exception(error_msg.format(origin, e.code, e.msg))
        except (Exception, KeyboardInterrupt) as e:
            if os.path.exists(fpath):
```

```
                os.remove(fpath)
            raise

    if untar:
        if not os.path.exists(untar_fpath):
            print('Extracting file.')
            with tarfile.open(fpath) as archive:
                archive.extractall(datadir)
        return untar_fpath

    if extract:
        _extract_archive(fpath, datadir, archive_format)

    return fpath
```

In [5]:
```
# Download the dataset and partially pre-trained model
get_file(fname='a1_data',
                        origin='http://www.cs.columbia.edu/~zemel/Class/Nndl/files/a
                        untar=True)
drive_location = 'data'
PARTIALLY_TRAINED_MODEL = drive_location + '/' + 'partially_trained.pk'
data_location = drive_location + '/' + 'data.pk'
```

```
File path: data/a1_data.tar.gz
Downloading data from http://www.cs.columbia.edu/~zemel/Class/Nndl/files/a1_data.tar.
gz
Extracting file.
```

We have already extracted the 4-grams from this dataset and divided them into training, validation, and test sets. To inspect this data, run the following:

In [6]:
```
data = pickle.load(open(data_location, 'rb'))
print(data['vocab'][0]) # First word in vocab is [MASK]
print(data['vocab'][1])
print(len(data['vocab'])) # Number of words in vocab
print(data['vocab']) # All the words in vocab
print(data['train_inputs'][:10]) # 10 example training instances
for idx in data['train_inputs'][372498]:
    print(idx, data['vocab'][idx])
```

```
[MASK]
all
251
['[MASK]', 'all', 'set', 'just', 'show', 'being', 'money', 'over', 'both', 'years',
'four', 'through', 'during', 'go', 'still', 'children', 'before', 'police', 'office',
'million', 'also', 'less', 'had', ',', 'including', 'should', 'to', 'only', 'going',
'under', 'has', 'might', 'do', 'them', 'good', 'around', 'get', 'very', 'big', 'dr.',
'game', 'every', 'know', 'they', 'not', 'world', 'now', 'him', 'school', 'several',
'like', 'did', 'university', 'companies', 'these', 'she', 'team', 'found', 'where',
'right', 'says', 'people', 'house', 'national', 'some', 'back', 'see', 'street', 'ar
e', 'year', 'home', 'best', 'out', 'even', 'what', 'said', 'for', 'federal', 'since',
'its', 'may', 'state', 'does', 'john', 'between', 'new', ';', 'three', 'public', '?',
'be', 'we', 'after', 'business', 'never', 'use', 'here', 'york', 'members', 'percen
t', 'put', 'group', 'come', 'by', '$', 'on', 'about', 'last', 'her', 'of', 'could',
'days', 'against', 'times', 'women', 'place', 'think', 'first', 'among', 'own', 'fami
ly', 'into', 'each', 'one', 'down', 'because', 'long', 'another', 'such', 'old', 'nex
t', 'your', 'market', 'second', 'city', 'little', 'from', 'would', 'few', 'west', 'th
ere', 'political', 'two', 'been', '.', 'their', 'much', 'music', 'too', 'way', 'whit
e', ':', 'was', 'war', 'today', 'more', 'ago', 'life', 'that', 'season', 'company',
'-', 'but', 'part', 'court', 'former', 'general', 'with', 'than', 'those', 'he', 'm
e', 'high', 'made', 'this', 'work', 'up', 'us', 'until', 'will', 'ms.', 'while', 'off
icials', 'can', 'were', 'country', 'my', 'called', 'and', 'program', 'have', 'then',
'is', 'it', 'an', 'states', 'case', 'say', 'his', 'at', 'want', 'in', 'any', 'as', 'i
f', 'united', 'end', 'no', ')', 'make', 'government', 'when', 'american', 'same', 'ho
w', 'mr.', 'other', 'take', 'which', 'department', '--', 'you', 'many', 'nt', 'day',
'week', 'play', 'used', "'s", 'though', 'our', 'who', 'yesterday', 'director', 'mos
t', 'president', 'law', 'man', 'a', 'night', 'off', 'center', 'i', 'well', 'or', 'wit
hout', 'so', 'time', 'five', 'the', 'left']
[[ 28  26  90 144]
 [184  44 249 117]
 [183  32  76 122]
 [117 247 201 186]
 [223 190 249   6]
 [ 42  74  26  32]
 [242  32 223  32]
 [223  32 158 144]
 [ 74  32 221  32]
 [ 42 192  91  68]]
28 going
26 to
36 get
193 it
```

Now `data` is a Python dict which contains the vocabulary, as well as the inputs and targets for all three splits of the data. `data['vocab']` is a list of the 251 words in the dictionary; `data['vocab'][0]` is the word with index 0, and so on. `data['train_inputs']` is a 372,500 x 4 matrix where each row gives the indices of the 4 consecutive context words for one of the 372,500 training cases. The validation and test sets are handled analogously.

Even though you only have to modify two specific locations in the code, you may want to read through this code before starting the assignment.

Part 1: GLoVE Word Representations (16pts)

In this part of the assignment, you will implement a simplified version of the GLoVE embedding (please see the handout for detailed description of the algorithm) with the loss defined as

$$L(\{\mathbf{w}_i, \tilde{\mathbf{w}}_i, b_i, \tilde{b}_i\}_{i=1}^V) = \sum_{i,j=1}^V (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

.

Note that each word is represented by two d-dimensional embedding vectors $\mathbf{w}_i, \tilde{\mathbf{w}}_i$ and two scalar biases $b_i, \tilde{b}_i$.

Answer the following questions:

1.1. GLoVE Parameter Count [1pt]

Given the vocabulary size V and embedding dimensionality d, how many parameters does the GLoVE model have? Note that each word in the vocabulary is associated with 2 embedding vectors and 2 biases.

1.1 **Answer**: 2(Vd + V) = 2V(d+1)

(Vd for the matrix R and V biases) multiply by 2 because we have 2 embeddings for each word

1.2. Expression for gradient $\frac{\partial L}{\partial \mathbf{w}_i}$ and $\frac{\partial L}{\partial \mathbf{b}_i}$ [6pts]

Write the expression for $\frac{\partial L}{\partial \mathbf{w}_i}$ and $\frac{\partial L}{\partial \mathbf{b}_i}$, the gradient of the loss function L with respect to parameter vector $\mathbf{w}_i$ and $\mathbf{b}_i$. The gradient should be a function of $\mathbf{w}, \tilde{\mathbf{w}}, b, \tilde{b}, X$ with appropriate subscripts (if any).

1.2 **Answer**:

$$\frac{\partial L}{\partial \mathbf{w}_i} = 2\sum_{j=1}^{V}\left((\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})\tilde{\mathbf{w}}_j\right)$$

$$\frac{\partial L}{\partial \mathbf{b}_i} = 2\sum_{j=1}^{V}\left((\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})\right)$$

1.3. Implement the gradient update of GLoVE. [6pts]

See `YOUR CODE HERE` Comment below for where to complete the code

We have provided a few functions for training the embedding:

- `calculate_log_co_occurence` computes the log co-occurrence matrix of a given corpus
- `train_GLoVE` runs momentum gradient descent to optimize the embedding
- `loss_GLoVE` :
  - INPUT - $V \times d$ matrix `W` (collection of V embedding vectors, each d-dimensional); $V \times d$ matrix `W_tilde` ; $V \times 1$ vector `b` (collection of V bias terms); $V \times 1$ vector `b_tilde` ; $V \times V$ log co-occurrence matrix.
  - OUTPUT - loss of the GLoVE objective
- `grad_GLoVE` : **TO BE IMPLEMENTED.**
  - INPUT:
    - $V \times d$ matrix `W` (collection of V embedding vectors, each d-dimensional), embedding for first word;
    - $V \times d$ matrix `W_tilde` , embedding for second word;
    - $V \times 1$ vector `b` (collection of V bias terms);
    - $V \times 1$ vector `b_tilde` , bias for second word;

    - $V \times V$ log co-occurrence matrix.
  - OUTPUT:
    - $V \times d$ matrix `grad_W` containing the gradient of the loss function w.r.t. `W`;
    - $V \times d$ matrix `grad_W_tilde` containing the gradient of the loss function w.r.t. `W_tilde`;
    - $V \times 1$ vector `grad_b` which is the gradient of the loss function w.r.t. `b`.
    - $V \times 1$ vector `grad_b_tilde` which is the gradient of the loss function w.r.t. `b_tilde`.

Run the code to compute the co-occurence matrix. Make sure to add a 1 to the occurences, so there are no 0's in the matrix when we take the elementwise log of the matrix.

In [7]:
```
vocab_size = len(data['vocab']) # Number of vocabs

def calculate_log_co_occurence(word_data, symmetric=False):
  "Compute the log-co-occurence matrix for our data."
  log_co_occurence = np.zeros((vocab_size, vocab_size))
  for input in word_data:
    # Note: the co-occurence matrix may not be symmetric
    log_co_occurence[input[0], input[1]] += 1
    log_co_occurence[input[1], input[2]] += 1
    log_co_occurence[input[2], input[3]] += 1
    # If we want symmetric co-occurence can also increment for these.
    if symmetric:
      log_co_occurence[input[1], input[0]] += 1
      log_co_occurence[input[2], input[1]] += 1
      log_co_occurence[input[3], input[2]] += 1
  delta_smoothing = 0.5  # A hyperparameter.  You can play with this if you want.
  log_co_occurence += delta_smoothing  # Add delta so log doesn't break on 0's.
  log_co_occurence = np.log(log_co_occurence)
  return log_co_occurence
```

In [8]:
```
asym_log_co_occurence_train = calculate_log_co_occurence(data['train_inputs'], symmet
asym_log_co_occurence_valid = calculate_log_co_occurence(data['valid_inputs'], symmet
```

- [ ] **TO BE IMPLEMENTED**: Calculate the gradient of the loss function w.r.t. the parameters W, $\tilde{W}$, $\mathbf{b}$, and $\mathbf{b}$. You should vectorize the computation, i.e. not loop over every word. The reading of the matrix cookbook from https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf might be useful.

In [9]:
```
def loss_GLoVE(W, W_tilde, b, b_tilde, log_co_occurence):
  "Compute the GLoVE loss."
  n,_ = log_co_occurence.shape
  if W_tilde is None and b_tilde is None:
    return np.sum((W @ W.T + b @ np.ones([1,n]) + np.ones([n,1])@b.T - log_co_occuren
  else:
    return np.sum((W @ W_tilde.T + b @ np.ones([1,n]) + np.ones([n,1])@b_tilde.T - lo

def grad_GLoVE(W, W_tilde, b, b_tilde, log_co_occurence):
  "Return the gradient of GLoVE objective w.r.t W and b."
  "INPUT: W - Vxd; W_tilde - Vxd; b - Vx1; b_tilde - Vx1; log_co_occurence: VxV"
  "OUTPUT: grad_W - Vxd; grad_W_tilde - Vxd, grad_b - Vx1, grad_b_tilde - Vx1"
  n,_ = log_co_occurence.shape

  if not W_tilde is None and not b_tilde is None:
  ############################   YOUR CODE HERE  ##############################
    #pass
    loss = (W @ W_tilde.T + b @ np.ones([1,n]) + np.ones([n,1]) @ b_tilde.T - (log_co
    #grad_W = 2 *(W_tilde.T @ loss).T #***
    grad_W = 2 *(loss @ W_tilde) #***
    grad_W_tilde = 2 * (W.T @ loss).T
```

```
    #grad_b = 2 * (np.ones([1,n]) @ loss).T #***
    grad_b = 2 * (loss @ np.ones([1,n]).T)
    grad_b_tilde = 2 * (np.ones([1,n]) @ loss).T
    #grad_b_tilde = 2 * (loss @ np.ones([n,1]))
  ##########################################################################
  else:
    loss = (W @ W.T + b @ np.ones([1,n]) + np.ones([n,1])@b.T - 0.5*(log_co_occurence
    grad_W = 4 *(W.T @ loss).T
    grad_W_tilde = None
    grad_b = 4 * (np.ones([1,n]) @ loss).T
    grad_b_tilde = None

  return grad_W, grad_W_tilde, grad_b, grad_b_tilde

def train_GLoVE(W, W_tilde, b, b_tilde, log_co_occurence_train, log_co_occurence_vali
  "Traing W and b according to GLoVE objective."
  n,_ = log_co_occurence_train.shape
  learning_rate = 0.1 / n  # A hyperparameter.  You can play with this if you want.
  for epoch in range(n_epochs):
    grad_W, grad_W_tilde, grad_b, grad_b_tilde = grad_GLoVE(W, W_tilde, b, b_tilde, l
    W = W - learning_rate * grad_W
    b = b - learning_rate * grad_b
    if not grad_W_tilde is None and not grad_b_tilde is None:
      W_tilde = W_tilde - learning_rate * grad_W_tilde
      b_tilde = b_tilde - learning_rate * grad_b_tilde
    train_loss, valid_loss = loss_GLoVE(W, W_tilde, b, b_tilde, log_co_occurence_trai
    if do_print:
      print(f"Train Loss: {train_loss}, valid loss: {valid_loss}, grad_norm: {np.sum(
  return W, W_tilde, b, b_tilde, train_loss, valid_loss
```

1.4. Effect of embedding dimension d [3pts]

Train the both the symmetric and asymmetric GLoVe model with varying dimensionality d by running the cell below. Comment on:

1. Which d leads to optimal validation performance for the asymmetric and symmetric models?
2. Why does / doesn't larger d always lead to better validation error?
3. Which model is performing better, and why?

1.4 Answer:

1. Best d (the point in which validation lose goes up -- avoiding overfitting):
   - asymmetric models: 10
   - symmetric models: 10
2. Increasing the dimensionality (d) doesn't necessarily result in improved validation error. This is because when d becomes larger and approaches the size of the vocabulary (v) or even larger, the model tends to capture more specific and from the data, leading to overfitting.
3. The asymmetric models perform better. I suppose it is because the log occurrences matrices of the symmetric models supposedly captures more information about the data by looking at the context words both from left to right and right to left but grammer wise in English it makes more sense to capture the relationship between words by only looking from left to right.

Train the GLoVE model for a range of embedding dimensions

In [10]:
```
np.random.seed(1)
n_epochs = 500  # A hyperparameter.  You can play with this if you want.
embedding_dims = np.array([1, 2, 10, 128, 256])  # Play with this
#embedding_dims = np.array([1, 10, 70, 128])  # Play with this
# Store the final losses for graphing
asymModel_asymCoOc_final_train_losses, asymModel_asymCoOc_final_val_losses = [], []
```

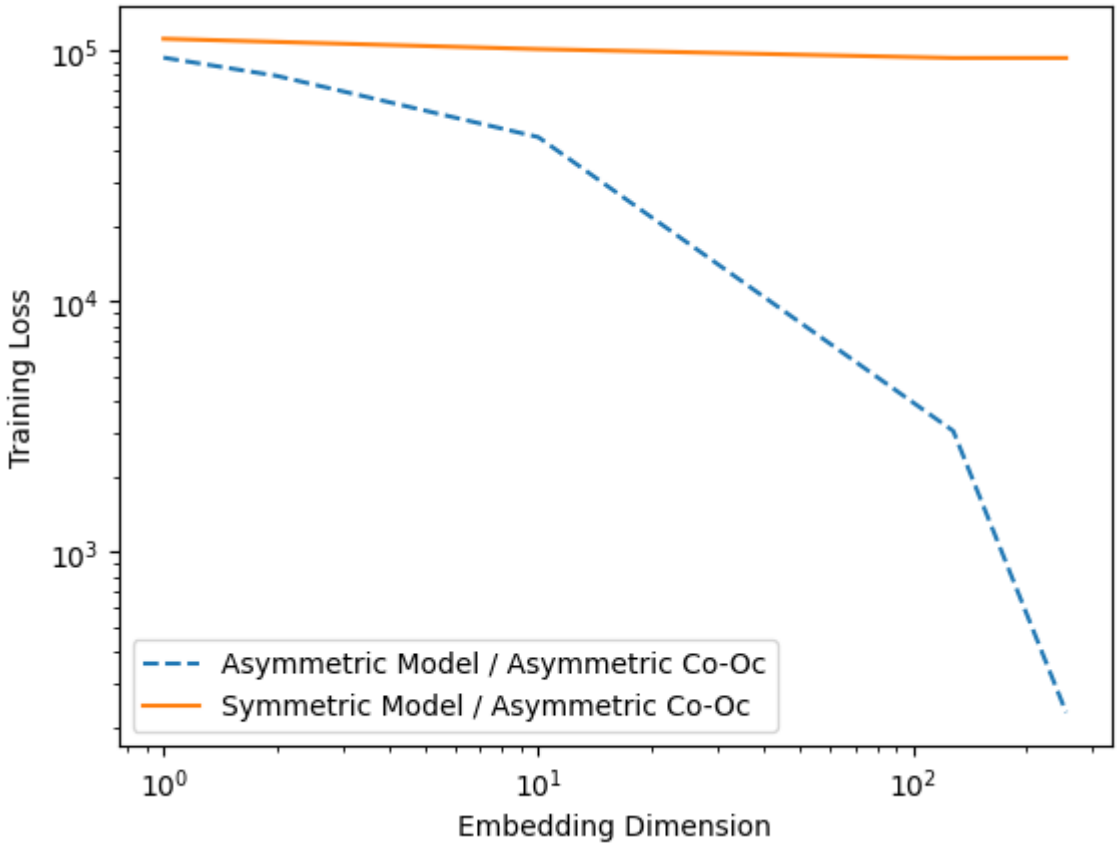
```
symModel_asymCoOc_final_train_losses, symModel_asymCoOc_final_val_losses = [], []
Asym_W_final_2d, Asym_b_final_2d, Asym_W_tilde_final_2d, Asym_b_tilde_final_2d = None
W_final_2d, b_final_2d = None, None
do_print = False  # If you want to see diagnostic information during training

for embedding_dim in tqdm(embedding_dims):
  init_variance = 0.1  # A hyperparameter.  You can play with this if you want.
  W = init_variance * np.random.normal(size=(vocab_size, embedding_dim))
  W_tilde = init_variance * np.random.normal(size=(vocab_size, embedding_dim))
  b = init_variance * np.random.normal(size=(vocab_size, 1))
  b_tilde = init_variance * np.random.normal(size=(vocab_size, 1))
  if do_print:
    print(f"Training for embedding dimension: {embedding_dim}")

  # Train Asym model on Asym Co-Oc matrix
  Asym_W_final, Asym_W_tilde_final, Asym_b_final, Asym_b_tilde_final, train_loss, val
  if embedding_dim == 2:
    # Save a parameter copy if we are training 2d embedding for visualization later
    Asym_W_final_2d = Asym_W_final
    Asym_W_tilde_final_2d = Asym_W_tilde_final
    Asym_b_final_2d = Asym_b_final
    Asym_b_tilde_final_2d = Asym_b_tilde_final
  asymModel_asymCoOc_final_train_losses += [train_loss]
  asymModel_asymCoOc_final_val_losses += [valid_loss]
  if do_print:
    print(f"Final validation loss: {valid_loss}")

  # Train Sym model on Asym Co-Oc matrix
  W_final, W_tilde_final, b_final, b_tilde_final, train_loss, valid_loss = train_GLoV
  if embedding_dim == 2:
    # Save a parameter copy if we are training 2d embedding for visualization later
    W_final_2d = W_final
    b_final_2d = b_final
  symModel_asymCoOc_final_train_losses += [train_loss]
  symModel_asymCoOc_final_val_losses += [valid_loss]
  if do_print:
    print(f"Final validation loss: {valid_loss}")
```

```
100%|██████████| 5/5 [00:28<00:00,  5.71s/it]
```

Plot the training and validation losses against the embedding dimension.

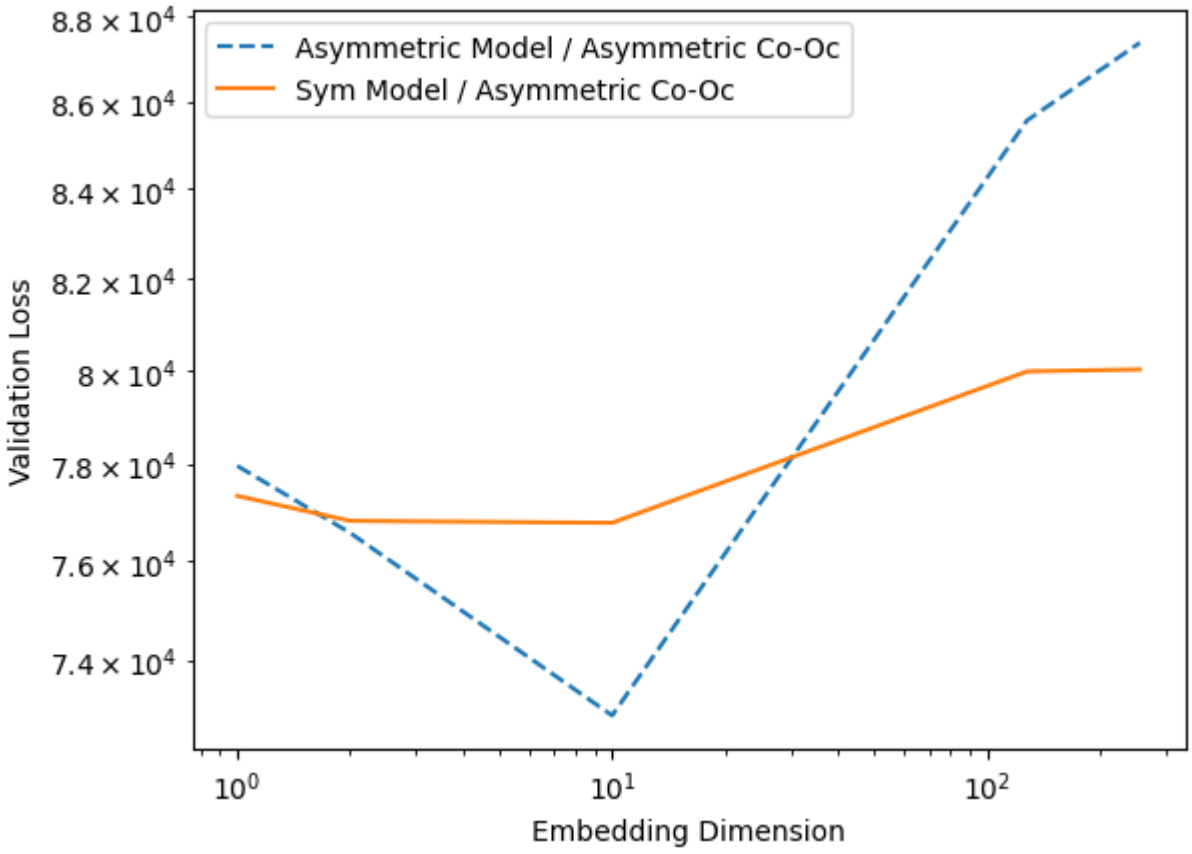
In [11]:
```
pylab.loglog(embedding_dims, asymModel_asymCoOc_final_train_losses, label="Asymmetric
pylab.loglog(embedding_dims, symModel_asymCoOc_final_train_losses , label="Symmetric
pylab.xlabel("Embedding Dimension")
pylab.ylabel("Training Loss")
pylab.legend()
```

Out[11]:
```
<matplotlib.legend.Legend at 0x79800c6cfe80>
```



In [12]:
```
pylab.loglog(embedding_dims, asymModel_asymCoOc_final_val_losses, label="Asymmetric M
pylab.loglog(embedding_dims, symModel_asymCoOc_final_val_losses , label="Sym Model / 
pylab.xlabel("Embedding Dimension")
pylab.ylabel("Validation Loss")
pylab.legend(loc="upper left")
```

Out[12]:
```
<matplotlib.legend.Legend at 0x79800c3dab00>
```



Part 2: Training the model (26pts)

We will modify the architecture slightly from the previous section, inspired by BERT \citep{devlin2018bert}. Instead of having only one output, the architecture will now take in $N = 4$ context words, and also output predictions for $N = 4$ words. See Figure 2 diagram in the handout for the diagram of this architecture.

During training, we randomly sample one of the N context words to replace with a `[MASK]` token. The goal is for the network to predict the word that was masked, at the corresponding output word position. In practice, this `[MASK]` token is assigned the index 0 in our dictionary. The weights $W^{(2)}$ = `hid_to_output_weights` now has the shape $NV \times H$, as the output layer has NV neurons, where the first V output units are for predicting the first word, then the next V are for predicting the second word, and so on. We call this as *concatenating* output uniits across all word positions, i.e. the $(j + nV)$-th column is for the word j in vocabulary for the n-th output word position. Note here that the softmax is applied in chunks of V as well, to give a valid probability distribution over the V words. Only the output word positions that were masked in the input are included in the cross entropy loss calculation:

There are three classes defined in this part: `Params`, `Activations`, `Model`. You will make changes to `Model`, but it may help to read through `Params` and `Activations` first.

$$C = -\sum_{i}^{B}\sum_{n}^{N}\sum_{j}^{V} m_n^{(i)}(t_{n,j}^{(i)} \log y_{n,j}^{(i)}),$$

Where $y_{n,j}^{(i)}$ denotes the output probability prediction from the neural network for the i-th training example for the word j in the n-th output word, and $t_{n,j}^{(i)}$ is 1 if for the i-th training example, the word j is the n-th word in context. Finally, $m_n^{(i)} \in \{0, 1\}$ is a mask that is set to 1 if we are predicting the n-th word position for the i-th example (because we had masked that word in the input), and 0 otherwise.

There are three classes defined in this part: `Params`, `Activations`, `Model`. You will make changes to `Model`, but it may help to read through `Params` and `Activations` first.

In [13]:
```
class Params(object):
    """A class representing the trainable parameters of the model. This class has fiv

            word_embedding_weights, a matrix of size V x D, where V is the number of w
                    and D is the embedding dimension.
            embed_to_hid_weights, a matrix of size H x ND, where H is the number of hi
                    columns represent connections from the embedding of the first cont
                    for the second context word, and so on. There are N context words.
            hid_bias, a vector of length H
            hid_to_output_weights, a matrix of size NV x H
            output_bias, a vector of length NV"""

    def __init__(self, word_embedding_weights, embed_to_hid_weights, hid_to_output_we
                 hid_bias, output_bias):
        self.word_embedding_weights = word_embedding_weights
        self.embed_to_hid_weights = embed_to_hid_weights
        self.hid_to_output_weights = hid_to_output_weights
        self.hid_bias = hid_bias
        self.output_bias = output_bias

    def copy(self):
        return self.__class__(self.word_embedding_weights.copy(), self.embed_to_hid_w
                              self.hid_to_output_weights.copy(), self.hid_bias.copy()

    @classmethod
    def zeros(cls, vocab_size, context_len, embedding_dim, num_hid):
        """A constructor which initializes all weights and biases to 0."""
```

```
        word_embedding_weights = np.zeros((vocab_size, embedding_dim))
        embed_to_hid_weights = np.zeros((num_hid, context_len * embedding_dim))
        hid_to_output_weights = np.zeros((vocab_size * context_len, num_hid))
        hid_bias = np.zeros(num_hid)
        output_bias = np.zeros(vocab_size * context_len)
        return cls(word_embedding_weights, embed_to_hid_weights, hid_to_output_weight
                   hid_bias, output_bias)

    @classmethod
    def random_init(cls, init_wt, vocab_size, context_len, embedding_dim, num_hid):
        """A constructor which initializes weights to small random values and biases
        word_embedding_weights = np.random.normal(0., init_wt, size=(vocab_size, embe
        embed_to_hid_weights = np.random.normal(0., init_wt, size=(num_hid, context_l
        hid_to_output_weights = np.random.normal(0., init_wt, size=(vocab_size * cont
        hid_bias = np.zeros(num_hid)
        output_bias = np.zeros(vocab_size * context_len)
        return cls(word_embedding_weights, embed_to_hid_weights, hid_to_output_weight
                   hid_bias, output_bias)

    ###### The functions below are Python's somewhat oddball way of overloading opera
    ###### we can do arithmetic on Params instances. You don't need to understand thi

    def __mul__(self, a):
        return self.__class__(a * self.word_embedding_weights,
                              a * self.embed_to_hid_weights,
                              a * self.hid_to_output_weights,
                              a * self.hid_bias,
                              a * self.output_bias)

    def __rmul__(self, a):
        return self * a

    def __add__(self, other):
        return self.__class__(self.word_embedding_weights + other.word_embedding_weig
                              self.embed_to_hid_weights + other.embed_to_hid_weights,
                              self.hid_to_output_weights + other.hid_to_output_weight
                              self.hid_bias + other.hid_bias,
                              self.output_bias + other.output_bias)

    def __sub__(self, other):
        return self + -1. * other
```

In [14]:

```
class Activations(object):
    """A class representing the activations of the units in the network. This class h

        embedding_layer, a matrix of B x ND matrix (where B is the batch size, D is t
                and N is the number of input context words), representing the activat
                layer on all the cases in a batch. The first D columns represent the
                first context word, and so on.
        hidden_layer, a B x H matrix representing the hidden layer activations for a
        output_layer, a B x V matrix representing the output layer activations for a

    def __init__(self, embedding_layer, hidden_layer, output_layer):
        self.embedding_layer = embedding_layer
        self.hidden_layer = hidden_layer
        self.output_layer = output_layer

def get_batches(inputs, batch_size, shuffle=True):
    """Divide a dataset (usually the training set) into mini-batches of a given size.
    'generator', i.e. something you can use in a for loop. You don't need to understa
    works to do the assignment."""

    if inputs.shape[0] % batch_size != 0:
        raise RuntimeError('The number of data points must be a multiple of the batch
    num_batches = inputs.shape[0] // batch_size

    if shuffle:
        idxs = np.random.permutation(inputs.shape[0])
```

```
        inputs = inputs[idxs, :]

    for m in range(num_batches):
        yield inputs[m * batch_size:(m + 1) * batch_size, :]
```

In this part of the assignment, you implement a method which computes the gradient using backpropagation. To start you out, the *Model* class contains several important methods used in training:

- `compute_activations` computes the activations of all units on a given input batch
- `compute_loss` computes the total cross-entropy loss on a mini-batch
- `evaluate` computes the average cross-entropy loss for a given set of inputs and targets

You will need to complete the implementation of two additional methods which are needed for training, and print the outputs of the gradients.

2.1 Implement gradient with respect to output layer inputs [8pts]

- [ ] **TO BE IMPLEMENTED**: `compute_loss_derivative` computes the derivative of the loss function with respect to the output layer inputs.

In other words, if C is the cost function, and the softmax computation for the j-th word in vocabulary for the n-th output word position is:

$$y_{n,j} = \frac{e^{z_{n,j}}}{\sum_l e^{z_{n,l}}}$$

This function should compute a $B \times NV$ matrix where the entries correspond to the partial derivatives $\partial C / \partial z_j^n$. Recall that the output units are concatenated across all positions, i.e. the $(j + nV)$-th column is for the word j in vocabulary for the n-th output word position.

2.2 Implement gradient with respect to parameters [8pts]

- [ ] **TO BE IMPLEMENTED**: `back_propagate` is the function which computes the gradient of the loss with respect to model parameters using backpropagation. It uses the derivatives computed by *compute_loss_derivative*. Some parts are already filled in for you, but you need to compute the matrices of derivatives for `embed_to_hid_weights`, `hid_bias`, `hid_to_output_weights`, and `output_bias`. These matrices have the same sizes as the parameter matrices (see previous section).

In order to implement backpropagation efficiently, you need to express the computations in terms of matrix operations, rather than *for* loops. You should first work through the derivatives on pencil and paper. First, apply the chain rule to compute the derivatives with respect to individual units, weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You should be able to express all of the required computations using only matrix multiplication, matrix transpose, and elementwise operations --- no *for* loops! If you want inspiration, read through the code for *Model.compute_activations* and try to understand how the matrix operations correspond to the computations performed by all the units in the network.

To make your life easier, we have provided the routine `checking.check_gradients`, which checks your gradients using finite differences. You should make sure this check passes before continuing with the assignment.

In [15]:
```
class Model(object):
    """A class representing the language model itself. This class contains various me
    the model and visualizing the learned representations. It has two fields:

        params, a Params instance which contains the model parameters
        vocab, a list containing all the words in the dictionary; vocab[0] is the wor
               0, and so on."""

    def __init__(self, params, vocab):
        self.params = params
        self.vocab = vocab

        self.vocab_size = len(vocab)
        self.embedding_dim = self.params.word_embedding_weights.shape[1]
        self.embedding_layer_dim = self.params.embed_to_hid_weights.shape[1]
        self.context_len = self.embedding_layer_dim // self.embedding_dim
        self.num_hid = self.params.embed_to_hid_weights.shape[0]
```

```
    def copy(self):
        return self.__class__(self.params.copy(), self.vocab[:])

    @classmethod
    def random_init(cls, init_wt, vocab, context_len, embedding_dim, num_hid):
        """Constructor which randomly initializes the weights to Gaussians with stand
        and initializes the biases to all zeros."""
        params = Params.random_init(init_wt, len(vocab), context_len, embedding_dim,
        return Model(params, vocab)

    def indicator_matrix(self, targets, mask_zero_index=True):
        """Construct a matrix where the (k + j*V)th entry of row i is 1 if the j-th t
         for example i is k, and all other entries are 0.

         Note: if the j-th target word index is 0, this corresponds to the [MASK] tok
               and we set the entry to be 0.
        """
        batch_size, context_len = targets.shape
        expanded_targets = np.zeros((batch_size, context_len * len(self.vocab)))
        targets_offset = np.repeat((np.arange(context_len) * len(self.vocab))[np.newa
        targets += targets_offset

        for c in range(context_len):
          expanded_targets[np.arange(batch_size), targets[:,c]] = 1.
          if mask_zero_index:
            # Note: Set the targets with index 0, V, 2V to be zero since it correspon
            expanded_targets[np.arange(batch_size), targets_offset[:,c]] = 0.
        return expanded_targets

    def compute_loss_derivative(self, output_activations, expanded_target_batch, targ
        """Compute the derivative of the multiple target position cross-entropy loss

            For example:

         [y_{0} ....  y_{V-1}] [y_{V}, ..., y_{2*V-1}] [y_{2*V} ... y_{i,3*V-1}] [y_{

         Where for colum j + n*V,

            y_{j + n*V} = e^{z_{j + n*V}} / \sum_{m=0}^{V-1} e^{z_{m + n*V}}, for n=0

        This function should return a dC / dz matrix of size [batch_size x (vocab_siz
        where each row i in dC / dz has columns 0 to V-1 containing the gradient the
        context word from i-th training example, then columns vocab_size to 2*vocab_s
        output context word of the i-th training example, etc.

        C is the loss function summed acrossed all examples as well:

            C = -\sum_{i,j,n} mask_{i,n} (t_{i, j + n*V} log y_{i, j + n*V}), for j=0

        where mask_{i,n} = 1 if the i-th training example has n-th context word as th
        otherwise mask_{i,n} = 0.

        The arguments are as follows:

            output_activations - A [batch_size x (context_len * vocab_size)] tensor,
                for the activations of the output layer, i.e. the y_j's.
            expanded_target_batch - A [batch_size (context_len * vocab_size)] tensor,
                where expanded_target_batch[i,n*V:(n+1)*V] is the indicator vector fo
                the n-th context target word position, i.e. the (i, j + n*V) entry is
                i'th example, the context word at position n is j, and 0 otherwise.
            target_mask - A [batch_size x context_len x 1] tensor, where target_mask[
                if for the i'th example the n-th context word is a target position, o

        Outputs:
            loss_derivative - A [batch_size x (context_len * vocab_size)] matrix,
                where loss_derivative[i,0:vocab_size] contains the gradient
                dC / dz_0 for the i-th training example gradient for 1st output
                context word, and loss_derivative[i,vocab_size:2*vocab_size] for
```

```
            the 2nd output context word of the i-th training example, etc.
        """

        ############################   YOUR CODE HERE  ##############################
        """
        target_mask_reshaped shape is [batch_size x (context_len * vocab_size)]
        repeat each elem in each row vocab_size times
        """
        target_mask_reshaped = np.tile(target_mask, (1, 1, self.vocab_size)).reshape(

        """
        print(expanded_target_batch[1,0:self.vocab_size])
        print("expanded_target_batch\n", expanded_target_batch)
        print("output_activations\n", output_activations)
        print("target_mask_reshaped:\n",target_mask_reshaped)
        print("output\n", target_mask_reshaped * expanded_target_batch * np.log(outpu

        Cost for every contex word: C = (m_n)*((sum over all j' words in voacc)(t_n,j

        Derivative for 1 sample dC / dz_n,j = (m_n)*(y_n,j - t_n,j)

        """
        return (target_mask_reshaped * (output_activations - expanded_target_batch))
        ##########################################################################

    def compute_loss(self, output_activations, expanded_target_batch):
        """Compute the total loss over a mini-batch. expanded_target_batch is the mat
        by calling indicator_matrix on the targets for the batch."""
        return -np.sum(expanded_target_batch * np.log(output_activations + TINY))

    def compute_activations(self, inputs):
        """Compute the activations on a batch given the inputs. Returns an Activation
        You should try to read and understand this function, since this will give you
        how to implement back_propagate."""

        batch_size = inputs.shape[0]
        if inputs.shape[1] != self.context_len:
            raise RuntimeError('Dimension of the input vectors should be {}, but is i
                self.context_len, inputs.shape[1]))

        # Embedding layer
        # Look up the input word indies in the word_embedding_weights matrix
        embedding_layer_state = np.zeros((batch_size, self.embedding_layer_dim))
        for i in range(self.context_len):
            embedding_layer_state[:, i * self.embedding_dim:(i + 1) * self.embedding_
                self.params.word_embedding_weights[inputs[:, i], :]

        # Hidden layer
        inputs_to_hid = np.dot(embedding_layer_state, self.params.embed_to_hid_weight
                       self.params.hid_bias
        # Apply logistic activation function
        hidden_layer_state = 1. / (1. + np.exp(-inputs_to_hid))

        # Output layer
        inputs_to_softmax = np.dot(hidden_layer_state, self.params.hid_to_output_weig
                            self.params.output_bias

        # Subtract maximum.
        # Remember that adding or subtracting the same constant from each input to a
        # softmax unit does not affect the outputs. So subtract the maximum to
        # make all inputs <= 0. This prevents overflows when computing their exponent
        inputs_to_softmax -= inputs_to_softmax.max(1).reshape((-1, 1))

        # Take softmax along each V chunks in the output layer
        output_layer_state = np.exp(inputs_to_softmax)
        output_layer_state_shape = output_layer_state.shape
        output_layer_state = output_layer_state.reshape((-1, self.context_len, len(se
        output_layer_state /= output_layer_state.sum(axis=-1, keepdims=True) # Softma
        output_layer_state = output_layer_state.reshape(output_layer_state_shape) # F
```

```
        return Activations(embedding_layer_state, hidden_layer_state, output_layer_st

    def back_propagate(self, input_batch, activations, loss_derivative):
        """Compute the gradient of the loss function with respect to the trainable pa
        of the model. The arguments are as follows:

             input_batch - the indices of the context words
             activations - an Activations class representing the output of Model.comp
             loss_derivative - the matrix of derivatives computed by compute_loss_der

        Part of this function is already completed, but you need to fill in the deriv
        computations for hid_to_output_weights_grad, output_bias_grad, embed_to_hid_w
        and hid_bias_grad. See the documentation for the Params class for a descripti
        these matrices represent."""

        # The matrix with values dC / dz_j, where dz_j is the input to the jth hidden
        # i.e. h_j = 1 / (1 + e^{-z_j})
        hid_deriv = np.dot(loss_derivative, self.params.hid_to_output_weights) \
                    * activations.hidden_layer * (1. - activations.hidden_layer)


        ############################   YOUR CODE HERE  ##############################
        hid_to_output_weights_grad = np.dot(loss_derivative.T, activations.hidden_lay
        output_bias_grad =  np.sum(loss_derivative, axis = 0)
        embed_to_hid_weights_grad = np.dot(hid_deriv.T, activations.embedding_layer)
        hid_bias_grad = np.sum(hid_deriv, axis = 0)
        ############################################################################

        # The matrix of derivatives for the embedding layer
        embed_deriv = np.dot(hid_deriv, self.params.embed_to_hid_weights)

        # Embedding layer
        word_embedding_weights_grad = np.zeros((self.vocab_size, self.embedding_dim))
        for w in range(self.context_len):
            word_embedding_weights_grad += np.dot(self.indicator_matrix(input_batch[:
                                                  embed_deriv[:, w * self.embedding_d

        return Params(word_embedding_weights_grad, embed_to_hid_weights_grad, hid_to_
                      hid_bias_grad, output_bias_grad)

    def sample_input_mask(self, batch_size):
        """Samples a binary mask for the inputs of size batch_size x context_len
        For each row, at most one element will be 1.
        """
        mask_idx = np.random.randint(self.context_len, size=(batch_size,))
        #mask = np.zeros((batch_size, self.context_len), dtype=np.int)# Convert to on
        """
        I got AttributeError: module 'numpy' has no attribute 'int'
        so changed it
        """
        mask = np.zeros((batch_size, self.context_len), dtype=np.int_)
        mask[np.arange(batch_size), mask_idx] = 1
        return mask

    def evaluate(self, inputs, batch_size=100):
        """Compute the average cross-entropy over a dataset.

            inputs: matrix of shape D x N"""

        ndata = inputs.shape[0]

        total = 0.
        for input_batch in get_batches(inputs, batch_size):
            mask = self.sample_input_mask(batch_size)
            input_batch_masked = input_batch * (1 - mask)
            activations = self.compute_activations(input_batch_masked)
            target_batch_masked = input_batch * mask
            expanded_target_batch = self.indicator_matrix(target_batch_masked)
```

```
            cross_entropy = -np.sum(expanded_target_batch * np.log(activations.output
            total += cross_entropy

        return total / float(ndata)

    def display_nearest_words(self, word, k=10):
        """List the k words nearest to a given word, along with their distances."""

        if word not in self.vocab:
            print('Word "{}" not in vocabulary.'.format(word))
            return

        # Compute distance to every other word.
        idx = self.vocab.index(word)
        word_rep = self.params.word_embedding_weights[idx, :]
        diff = self.params.word_embedding_weights - word_rep.reshape((1, -1))
        distance = np.sqrt(np.sum(diff ** 2, axis=1))

        # Sort by distance.
        order = np.argsort(distance)
        order = order[1:1 + k]  # The nearest word is the query word itself, skip tha
        for i in order:
            print('{}: {}'.format(self.vocab[i], distance[i]))

    def word_distance(self, word1, word2):
        """Compute the distance between the vector representations of two words."""

        if word1 not in self.vocab:
            raise RuntimeError('Word "{}" not in vocabulary.'.format(word1))
        if word2 not in self.vocab:
            raise RuntimeError('Word "{}" not in vocabulary.'.format(word2))

        idx1, idx2 = self.vocab.index(word1), self.vocab.index(word2)
        word_rep1 = self.params.word_embedding_weights[idx1, :]
        word_rep2 = self.params.word_embedding_weights[idx2, :]
        diff = word_rep1 - word_rep2
        return np.sqrt(np.sum(diff ** 2))
```

2.3 Print the gradients [8pts]

To make your life easier, we have provided the routine `check_gradients`, which checks your gradients using finite differences. You should make sure this check passes before continuing with the assignment. Once `check_gradients()` passes, call `print_gradients()` and include its output in your write-up.

In [16]:

```
def relative_error(a, b):
    return np.abs(a - b) / (np.abs(a) + np.abs(b))


def check_output_derivatives(model, input_batch, target_batch):
    def softmax(z):
        z = z.copy()
        z -= z.max(-1, keepdims=True)
        y = np.exp(z)
        y /= y.sum(-1, keepdims=True)
        return y

    batch_size = input_batch.shape[0]
    z = np.random.normal(size=(batch_size, model.context_len, model.vocab_size))
    y = softmax(z).reshape((batch_size, model.context_len * model.vocab_size))
    z = z.reshape((batch_size, model.context_len * model.vocab_size))

    expanded_target_batch = model.indicator_matrix(target_batch)
    target_mask = expanded_target_batch.reshape(-1, model.context_len, len(model.voca
    loss_derivative = model.compute_loss_derivative(y, expanded_target_batch, target_
```

```
    if loss_derivative is None:
        print('Loss derivative not implemented yet.')
        return False

    if loss_derivative.shape != (batch_size, model.vocab_size * model.context_len):
        print('Loss derivative should be size {} but is actually {}.'.format(
            (batch_size, model.vocab_size), loss_derivative.shape))
        return False

    def obj(z):
        z = z.reshape((-1, model.context_len, model.vocab_size))
        y = softmax(z).reshape((batch_size, model.context_len * model.vocab_size))
        return model.compute_loss(y, expanded_target_batch)

    for count in range(1000):
        i, j = np.random.randint(0, loss_derivative.shape[0]), np.random.randint(0, l

        z_plus = z.copy()
        z_plus[i, j] += EPS
        obj_plus = obj(z_plus)

        z_minus = z.copy()
        z_minus[i, j] -= EPS
        obj_minus = obj(z_minus)

        empirical = (obj_plus - obj_minus) / (2. * EPS)
        rel = relative_error(empirical, loss_derivative[i, j])
        if rel > 1e-4:
            print('The loss derivative has a relative error of {}, which is too large
            return False

    print('The loss derivative looks OK.')
    return True


def check_param_gradient(model, param_name, input_batch, target_batch):
    activations = model.compute_activations(input_batch)
    expanded_target_batch = model.indicator_matrix(target_batch)
    target_mask = expanded_target_batch.reshape(-1, model.context_len, len(model.voca
    loss_derivative = model.compute_loss_derivative(activations.output_layer, expande
    param_gradient = model.back_propagate(input_batch, activations, loss_derivative)

    def obj(model):
        activations = model.compute_activations(input_batch)
        return model.compute_loss(activations.output_layer, expanded_target_batch)

    dims = getattr(model.params, param_name).shape
    is_matrix = (len(dims) == 2)

    if getattr(param_gradient, param_name).shape != dims:
        print('The gradient for {} should be size {} but is actually {}.'.format(
            param_name, dims, getattr(param_gradient, param_name).shape))
        return

    for count in range(1000):
        if is_matrix:
            slc = np.random.randint(0, dims[0]), np.random.randint(0, dims[1])
        else:
            slc = np.random.randint(dims[0])

        model_plus = model.copy()
        getattr(model_plus.params, param_name)[slc] += EPS
        obj_plus = obj(model_plus)

        model_minus = model.copy()
        getattr(model_minus.params, param_name)[slc] -= EPS
        obj_minus = obj(model_minus)
```

```
        empirical = (obj_plus - obj_minus) / (2. * EPS)
        exact = getattr(param_gradient, param_name)[slc]
        rel = relative_error(empirical, exact)
        if rel > 3e-4:
            import pdb; pdb.set_trace()
            print('The loss derivative has a relative error of {}, which is too large
            return False

    print('The gradient for {} looks OK.'.format(param_name))


def load_partially_trained_model():
    obj = pickle.load(open(PARTIALLY_TRAINED_MODEL, 'rb'))
    params = Params(obj['word_embedding_weights'], obj['embed_to_hid_weights'],
                                    obj['hid_to_output_weights'], obj['hid_bias'],
                                    obj['output_bias'])
    vocab = obj['vocab']
    return Model(params, vocab)


def check_gradients():
    """Check the computed gradients using finite differences."""
    np.random.seed(0)

    np.seterr(all='ignore')  # suppress a warning which is harmless

    model = load_partially_trained_model()
    data_obj = pickle.load(open(data_location, 'rb'))
    train_inputs = data_obj['train_inputs']
    input_batch = train_inputs[:100, :]
    mask = model.sample_input_mask(input_batch.shape[0])
    input_batch_masked = input_batch * (1 - mask)
    target_batch_masked = input_batch * mask

    if not check_output_derivatives(model, input_batch_masked, target_batch_masked):
        return

    for param_name in ['word_embedding_weights', 'embed_to_hid_weights', 'hid_to_outp
                       'hid_bias', 'output_bias']:
        input_batch_masked = input_batch * (1 - mask)
        target_batch_masked = input_batch * mask
        check_param_gradient(model, param_name, input_batch_masked, target_batch_mask


def print_gradients():
    """Print out certain derivatives for grading."""
    np.random.seed(0)

    model = load_partially_trained_model()
    data_obj = pickle.load(open(data_location, 'rb'))
    train_inputs = data_obj['train_inputs']
    input_batch = train_inputs[:100, :]

    mask = model.sample_input_mask(input_batch.shape[0])
    input_batch_masked = input_batch * (1 - mask)
    activations = model.compute_activations(input_batch_masked)
    target_batch_masked = input_batch * mask
    expanded_target_batch = model.indicator_matrix(target_batch_masked)
    target_mask = expanded_target_batch.reshape(-1, model.context_len, len(model.voca
    loss_derivative = model.compute_loss_derivative(activations.output_layer, expande
    param_gradient = model.back_propagate(input_batch, activations, loss_derivative)

    print('loss_derivative[2, 5]', loss_derivative[2, 5])
    print('loss_derivative[2, 121]', loss_derivative[2, 121])
    print('loss_derivative[5, 33]', loss_derivative[5, 33])
    print('loss_derivative[5, 31]', loss_derivative[5, 31])
    print()
    print('param_gradient.word_embedding_weights[27, 2]', param_gradient.word_embeddi
    print('param_gradient.word_embedding_weights[43, 3]', param_gradient.word_embeddi
```

```
    print('param_gradient.word_embedding_weights[22, 4]', param_gradient.word_embeddi
    print('param_gradient.word_embedding_weights[2, 5]', param_gradient.word_embeddin
    print()
    print('param_gradient.embed_to_hid_weights[10, 2]', param_gradient.embed_to_hid_w
    print('param_gradient.embed_to_hid_weights[15, 3]', param_gradient.embed_to_hid_w
    print('param_gradient.embed_to_hid_weights[30, 9]', param_gradient.embed_to_hid_w
    print('param_gradient.embed_to_hid_weights[35, 21]', param_gradient.embed_to_hid_
    print()
    print('param_gradient.hid_bias[10]', param_gradient.hid_bias[10])
    print('param_gradient.hid_bias[20]', param_gradient.hid_bias[20])
    print()
    print('param_gradient.output_bias[0]', param_gradient.output_bias[0])
    print('param_gradient.output_bias[1]', param_gradient.output_bias[1])
    print('param_gradient.output_bias[2]', param_gradient.output_bias[2])
    print('param_gradient.output_bias[3]', param_gradient.output_bias[3])
```

In [17]:
```
# Run this to check if your implement gradients matches the finite difference within
# Note: this may take a few minutes to go through all the checks
check_gradients()
```

```
The loss derivative looks OK.
The gradient for word_embedding_weights looks OK.
The gradient for embed_to_hid_weights looks OK.
The gradient for hid_to_output_weights looks OK.
The gradient for hid_bias looks OK.
The gradient for output_bias looks OK.
```

In [18]:
```
# Run this to print out the gradients
print_gradients()
```

```
loss_derivative[2, 5] 0.0
loss_derivative[2, 121] 0.0
loss_derivative[5, 33] 0.0
loss_derivative[5, 31] 0.0

param_gradient.word_embedding_weights[27, 2] 0.0
param_gradient.word_embedding_weights[43, 3] 0.011596892511489458
param_gradient.word_embedding_weights[22, 4] -0.0222670623817297
param_gradient.word_embedding_weights[2, 5] 0.0

param_gradient.embed_to_hid_weights[10, 2] 0.3793257091930164
param_gradient.embed_to_hid_weights[15, 3] 0.01604516132110917
param_gradient.embed_to_hid_weights[30, 9] -0.4312854367997419
param_gradient.embed_to_hid_weights[35, 21] 0.06679896665436337

param_gradient.hid_bias[10] 0.023428803123345148
param_gradient.hid_bias[20] -0.024370452378874197

param_gradient.output_bias[0] 0.000970106146902794
param_gradient.output_bias[1] 0.16868946274763222
param_gradient.output_bias[2] 0.0051664774143909235
param_gradient.output_bias[3] 0.15096226471814364
```

2.4 Run model training [2pts]

Once you've implemented the gradient computation, you'll need to train the model. The function *train* implements the main training procedure. It takes two arguments:

- `embedding_dim` : The number of dimensions in the distributed representation.
- `num_hid` : The number of hidden units

As the model trains, the script prints out some numbers that tell you how well the training is going. It shows:

- The cross entropy on the last 100 mini-batches of the training set. This is shown after every 100 mini-batches.
- The cross entropy on the entire validation set every 1000 mini-batches of training.

At the end of training, this function shows the cross entropies on the training, validation and test sets. It will return a *Model* instance.

In [19]:
```
_train_inputs = None
_train_targets = None
_vocab = None

DEFAULT_TRAINING_CONFIG = {'batch_size': 100,  # the size of a mini-batch
                           'learning_rate': 0.1,  # the learning rate
                           'momentum': 0.9,  # the decay parameter for the momentum v
                           'epochs': 50,  # the maximum number of epochs to run
                           'init_wt': 0.01,  # the standard deviation of the initial
                           'context_len': 4,  # the number of context words used
                           'show_training_CE_after': 100,  # measure training error a
                           'show_validation_CE_after': 1000,  # measure validation er
                           }


def find_occurrences(word1, word2, word3):
    """Lists all the words that followed a given tri-gram in the training set and the
    times each one followed it."""

    # cache the data so we don't keep reloading
    global _train_inputs, _train_targets, _vocab
    if _train_inputs is None:
        data_obj = pickle.load(open(data_location, 'rb'))
        _vocab = data_obj['vocab']
        _train_inputs, _train_targets = data_obj['train_inputs'], data_obj['train_tar

    if word1 not in _vocab:
        raise RuntimeError('Word "{}" not in vocabulary.'.format(word1))
    if word2 not in _vocab:
        raise RuntimeError('Word "{}" not in vocabulary.'.format(word2))
    if word3 not in _vocab:
        raise RuntimeError('Word "{}" not in vocabulary.'.format(word3))

    idx1, idx2, idx3 = _vocab.index(word1), _vocab.index(word2), _vocab.index(word3)
    idxs = np.array([idx1, idx2, idx3])

    matches = np.all(_train_inputs == idxs.reshape((1, -1)), 1)

    if np.any(matches):
        counts = collections.defaultdict(int)
        for m in np.where(matches)[0]:
            counts[_vocab[_train_targets[m]]] += 1

        word_counts = sorted(list(counts.items()), key=lambda t: t[1], reverse=True)
        print('The tri-gram "{} {} {}" was followed by the following words in the tra
            word1, word2, word3))
        for word, count in word_counts:
            if count > 1:
                print('    {} ({} times)'.format(word, count))
            else:
                print('    {} (1 time)'.format(word))
    else:
        print('The tri-gram "{} {} {}" did not occur in the training set.'.format(wor


def train(embedding_dim, num_hid, config=DEFAULT_TRAINING_CONFIG):
    """This is the main training routine for the language model. It takes two paramet

        embedding_dim, the dimension of the embedding space
```

```
    num_hid, the number of hidden units."""
# For reproducibility
np.random.seed(123)

# Load the data
data_obj = pickle.load(open(data_location, 'rb'))
vocab = data_obj['vocab']
train_inputs = data_obj['train_inputs']
valid_inputs = data_obj['valid_inputs']
test_inputs = data_obj['test_inputs']

# Randomly initialize the trainable parameters
model = Model.random_init(config['init_wt'], vocab, config['context_len'], embedd

# Variables used for early stopping
best_valid_CE = np.infty
end_training = False

# Initialize the momentum vector to all zeros
delta = Params.zeros(len(vocab), config['context_len'], embedding_dim, num_hid)

this_chunk_CE = 0.
batch_count = 0
for epoch in range(1, config['epochs'] + 1):
    if end_training:
        break

    print()
    print('Epoch', epoch)

    for m, (input_batch) in enumerate(get_batches(train_inputs, config['batch_siz
        batch_count += 1

        # For each example (row in input_batch), select one word to mask out
        mask = model.sample_input_mask(config['batch_size'])
        input_batch_masked = input_batch * (1 - mask) # We only zero out one word
        target_batch_masked = input_batch * mask # We want to predict the masked

        # Forward propagate
        activations = model.compute_activations(input_batch_masked)

        # Compute loss derivative
        expanded_target_batch = model.indicator_matrix(target_batch_masked)
        loss_derivative = model.compute_loss_derivative(activations.output_layer,
        loss_derivative /= config['batch_size']

        # Measure loss function
        cross_entropy = model.compute_loss(activations.output_layer, expanded_tar
        this_chunk_CE += cross_entropy
        if batch_count % config['show_training_CE_after'] == 0:
            print('Batch {} Train CE {:1.3f}'.format(
                batch_count, this_chunk_CE / config['show_training_CE_after']))
            this_chunk_CE = 0.

        # Backpropagate
        loss_gradient = model.back_propagate(input_batch, activations, loss_deriv

        # Update the momentum vector and model parameters
        delta = config['momentum'] * delta + loss_gradient
        model.params -= config['learning_rate'] * delta

        # Validate
        if batch_count % config['show_validation_CE_after'] == 0:
            print('Running validation...')
            cross_entropy = model.evaluate(valid_inputs)
            print('Validation cross-entropy: {:1.3f}'.format(cross_entropy))

            if cross_entropy > best_valid_CE:
                print('Validation error increasing!  Training stopped.')
```

```
                    end_training = True
                    break

                best_valid_CE = cross_entropy

    print()
    train_CE = model.evaluate(train_inputs)
    print('Final training cross-entropy: {:1.3f}'.format(train_CE))
    valid_CE = model.evaluate(valid_inputs)
    print('Final validation cross-entropy: {:1.3f}'.format(valid_CE))
    test_CE = model.evaluate(test_inputs)
    print('Final test cross-entropy: {:1.3f}'.format(test_CE))

    return model
```

Run the training.

In [20]:
```
embedding_dim = 16
num_hid = 128
trained_model = train(embedding_dim, num_hid)
```

```
Epoch 1
Batch 100 Train CE 4.793
Batch 200 Train CE 4.645
Batch 300 Train CE 4.649
Batch 400 Train CE 4.629
Batch 500 Train CE 4.633
Batch 600 Train CE 4.648
Batch 700 Train CE 4.617
Batch 800 Train CE 4.607
Batch 900 Train CE 4.606
Batch 1000 Train CE 4.615
Running validation...
Validation cross-entropy: 4.615
Batch 1100 Train CE 4.615
Batch 1200 Train CE 4.624
Batch 1300 Train CE 4.608
Batch 1400 Train CE 4.595
Batch 1500 Train CE 4.611
Batch 1600 Train CE 4.598
Batch 1700 Train CE 4.577
Batch 1800 Train CE 4.578
Batch 1900 Train CE 4.568
Batch 2000 Train CE 4.589
Running validation...
Validation cross-entropy: 4.589
Batch 2100 Train CE 4.573
Batch 2200 Train CE 4.611
Batch 2300 Train CE 4.562
Batch 2400 Train CE 4.587
Batch 2500 Train CE 4.589
Batch 2600 Train CE 4.587
Batch 2700 Train CE 4.561
Batch 2800 Train CE 4.544
Batch 2900 Train CE 4.521
Batch 3000 Train CE 4.524
Running validation...
Validation cross-entropy: 4.496
Batch 3100 Train CE 4.504
Batch 3200 Train CE 4.449
Batch 3300 Train CE 4.384
Batch 3400 Train CE 4.352
Batch 3500 Train CE 4.324
Batch 3600 Train CE 4.261
Batch 3700 Train CE 4.267

Epoch 2
Batch 3800 Train CE 4.208
Batch 3900 Train CE 4.168
Batch 4000 Train CE 4.117
Running validation...
Validation cross-entropy: 4.112
Batch 4100 Train CE 4.105
Batch 4200 Train CE 4.049
Batch 4300 Train CE 4.008
Batch 4400 Train CE 3.986
Batch 4500 Train CE 3.924
Batch 4600 Train CE 3.897
Batch 4700 Train CE 3.857
Batch 4800 Train CE 3.790
Batch 4900 Train CE 3.796
Batch 5000 Train CE 3.773
Running validation...
Validation cross-entropy: 3.776
Batch 5100 Train CE 3.766
Batch 5200 Train CE 3.714
Batch 5300 Train CE 3.720
Batch 5400 Train CE 3.668
Batch 5500 Train CE 3.668
Batch 5600 Train CE 3.639
```

```
Batch 5700 Train CE 3.571
Batch 5800 Train CE 3.546
Batch 5900 Train CE 3.537
Batch 6000 Train CE 3.511
Running validation...
Validation cross-entropy: 3.531
Batch 6100 Train CE 3.494
Batch 6200 Train CE 3.495
Batch 6300 Train CE 3.477
Batch 6400 Train CE 3.455
Batch 6500 Train CE 3.435
Batch 6600 Train CE 3.446
Batch 6700 Train CE 3.411
Batch 6800 Train CE 3.376
Batch 6900 Train CE 3.419
Batch 7000 Train CE 3.375
Running validation...
Validation cross-entropy: 3.386
Batch 7100 Train CE 3.398
Batch 7200 Train CE 3.383
Batch 7300 Train CE 3.371
Batch 7400 Train CE 3.355

Epoch 3
Batch 7500 Train CE 3.320
Batch 7600 Train CE 3.315
Batch 7700 Train CE 3.342
Batch 7800 Train CE 3.293
Batch 7900 Train CE 3.285
Batch 8000 Train CE 3.296
Running validation...
Validation cross-entropy: 3.294
Batch 8100 Train CE 3.271
Batch 8200 Train CE 3.291
Batch 8300 Train CE 3.287
Batch 8400 Train CE 3.274
Batch 8500 Train CE 3.228
Batch 8600 Train CE 3.256
Batch 8700 Train CE 3.250
Batch 8800 Train CE 3.256
Batch 8900 Train CE 3.266
Batch 9000 Train CE 3.221
Running validation...
Validation cross-entropy: 3.233
Batch 9100 Train CE 3.247
Batch 9200 Train CE 3.229
Batch 9300 Train CE 3.224
Batch 9400 Train CE 3.217
Batch 9500 Train CE 3.208
Batch 9600 Train CE 3.201
Batch 9700 Train CE 3.199
Batch 9800 Train CE 3.225
Batch 9900 Train CE 3.187
Batch 10000 Train CE 3.176
Running validation...
Validation cross-entropy: 3.177
Batch 10100 Train CE 3.168
Batch 10200 Train CE 3.164
Batch 10300 Train CE 3.165
Batch 10400 Train CE 3.199
Batch 10500 Train CE 3.178
Batch 10600 Train CE 3.175
Batch 10700 Train CE 3.148
Batch 10800 Train CE 3.178
Batch 10900 Train CE 3.183
Batch 11000 Train CE 3.098
Running validation...
Validation cross-entropy: 3.147
Batch 11100 Train CE 3.159
```

```
Epoch 4
Batch 11200 Train CE 3.153
Batch 11300 Train CE 3.137
Batch 11400 Train CE 3.136
Batch 11500 Train CE 3.154
Batch 11600 Train CE 3.122
Batch 11700 Train CE 3.120
Batch 11800 Train CE 3.163
Batch 11900 Train CE 3.107
Batch 12000 Train CE 3.142
Running validation...
Validation cross-entropy: 3.114
Batch 12100 Train CE 3.134
Batch 12200 Train CE 3.131
Batch 12300 Train CE 3.122
Batch 12400 Train CE 3.101
Batch 12500 Train CE 3.071
Batch 12600 Train CE 3.136
Batch 12700 Train CE 3.117
Batch 12800 Train CE 3.126
Batch 12900 Train CE 3.079
Batch 13000 Train CE 3.100
Running validation...
Validation cross-entropy: 3.098
Batch 13100 Train CE 3.110
Batch 13200 Train CE 3.092
Batch 13300 Train CE 3.096
Batch 13400 Train CE 3.089
Batch 13500 Train CE 3.070
Batch 13600 Train CE 3.074
Batch 13700 Train CE 3.088
Batch 13800 Train CE 3.077
Batch 13900 Train CE 3.081
Batch 14000 Train CE 3.083
Running validation...
Validation cross-entropy: 3.079
Batch 14100 Train CE 3.083
Batch 14200 Train CE 3.105
Batch 14300 Train CE 3.144
Batch 14400 Train CE 3.091
Batch 14500 Train CE 3.081
Batch 14600 Train CE 3.134
Batch 14700 Train CE 3.098
Batch 14800 Train CE 3.077
Batch 14900 Train CE 3.075

Epoch 5
Batch 15000 Train CE 3.055
Running validation...
Validation cross-entropy: 3.068
Batch 15100 Train CE 3.093
Batch 15200 Train CE 3.064
Batch 15300 Train CE 3.087
Batch 15400 Train CE 3.089
Batch 15500 Train CE 3.041
Batch 15600 Train CE 3.083
Batch 15700 Train CE 3.072
Batch 15800 Train CE 3.067
Batch 15900 Train CE 3.070
Batch 16000 Train CE 3.075
Running validation...
Validation cross-entropy: 3.078
Validation error increasing!  Training stopped.

Final training cross-entropy: 3.063
Final validation cross-entropy: 3.073
Final test cross-entropy: 3.075
```

To convince us that you have correctly implemented the gradient computations, please include the following with your assignment submission:

- [ ] You will submit `hw1_code_<YOUR_UNI>.ipynb` through CourseWorks. You do not need to modify any of the code except the parts we asked you to implement.
- [ ] In your PDF file, include the output of the function `print_gradients`. This prints out part of the gradients for a partially trained network which we have provided, and we will check them against the correct outputs. **Important:** make sure to give the output of `print_gradients`, **not** `check_gradients`.

Since we gave you a gradient checker, you have no excuse for not getting full points on this part.

Part 3: Arithmetics and Analysis (8pts)

In this part, you will perform arithmetic calculations on the word embeddings learned from previous models and analyze the representation learned by the networks with t-SNE plots.

3.1 t-SNE

You will first train the models discussed in the previous sections; you'll use the trained models for the remainding of this section.

Important: if you've made any fixes to your gradient code, you must reload the a1-code module and then re-run the training procedure. Python does not reload modules automatically, and you don't want to accidentally analyze an old version of your model.

These methods of the Model class can be used for analyzing the model after the training is done:

- `tsne_plot_representation` creates a 2-dimensional embedding of the distributed representation space using an algorithm called t-SNE. (You don't need to know what this is for the assignment, but we may cover it later in the course.) Nearby points in this 2-D space are meant to correspond to nearby points in the 16-D space.
- `display_nearest_words` lists the words whose embedding vectors are nearest to the given word
- `word_distance` computes the distance between the embeddings of two words

Plot the 2-dimensional visualization for the trained model from part 3 using the method `tsne_plot_representation`. Look at the plot and find a few clusters of related words. What do the words in each cluster have in common? Plot the 2-dimensional visualization for the GloVe model from part 1 using the method `tsne_plot_GLoVe_representation`. How do the t-SNE embeddings for both models compare? Plot the 2-dimensional visualization using the method `plot_2d_GLoVe_representation`. How does this compare to the t-SNE embeddings? Please answer in 2 sentences for each question and show the plots in your submission.

3.1

1. Plot the 2-dimensional visualization for the trained model from part 3 using the method tsne_plot_representation. Look at the plot and find a few clusters of related words. What do the words in each cluster have in common? **Answer:** Some clusters of realted words I see (share similar usage/meaning/boht):

- {"Should", "Could", "Will", "Would", "Can"} -- similar usage, meaning
- {"She", "He"}
- {"Get", "Go", "Come"}
- {"Still", "Even", "Just"} -- similar usage

1. Plot the 2-dimensional visualization for the GloVe model from part 1 using the method tsne_plot_GLoVe_representation. How do the t-SNE embeddings for both models compare? **Answer:** The t-SNE embeddings for models seem to group words based on their meaning and usage For instance some realted groups of words I see when using the Glove model from part 1 are:

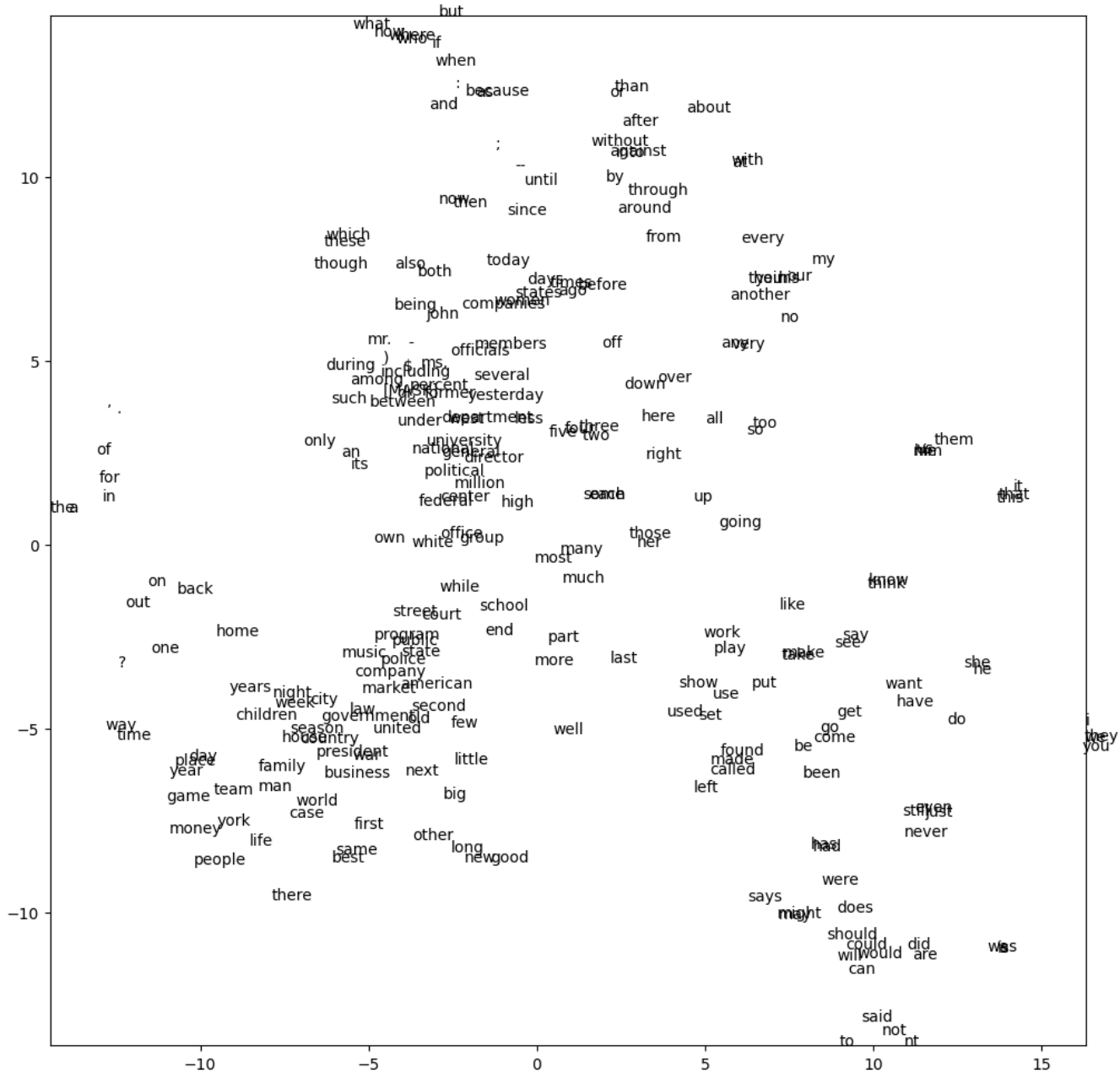
- {"Team", "Play", "Game"}
- {"How", "Where", "When"} -- similar usage, question words

1. Plot the 2-dimensional visualization using the method plot_2d_GLoVe_representation. How does this compare to the t-SNE embeddings? **Answer:** In comparison to the t-SNE embeddings, the 2-dimensional GLoVe representation appears more tightly clustered. This suggests that t-SNE embeddings have a better ability to capture distinct groups and semantic information. In the 2D GLoVe representation, many words are grouped closely together, making it more challenging to discern the relationships between them.

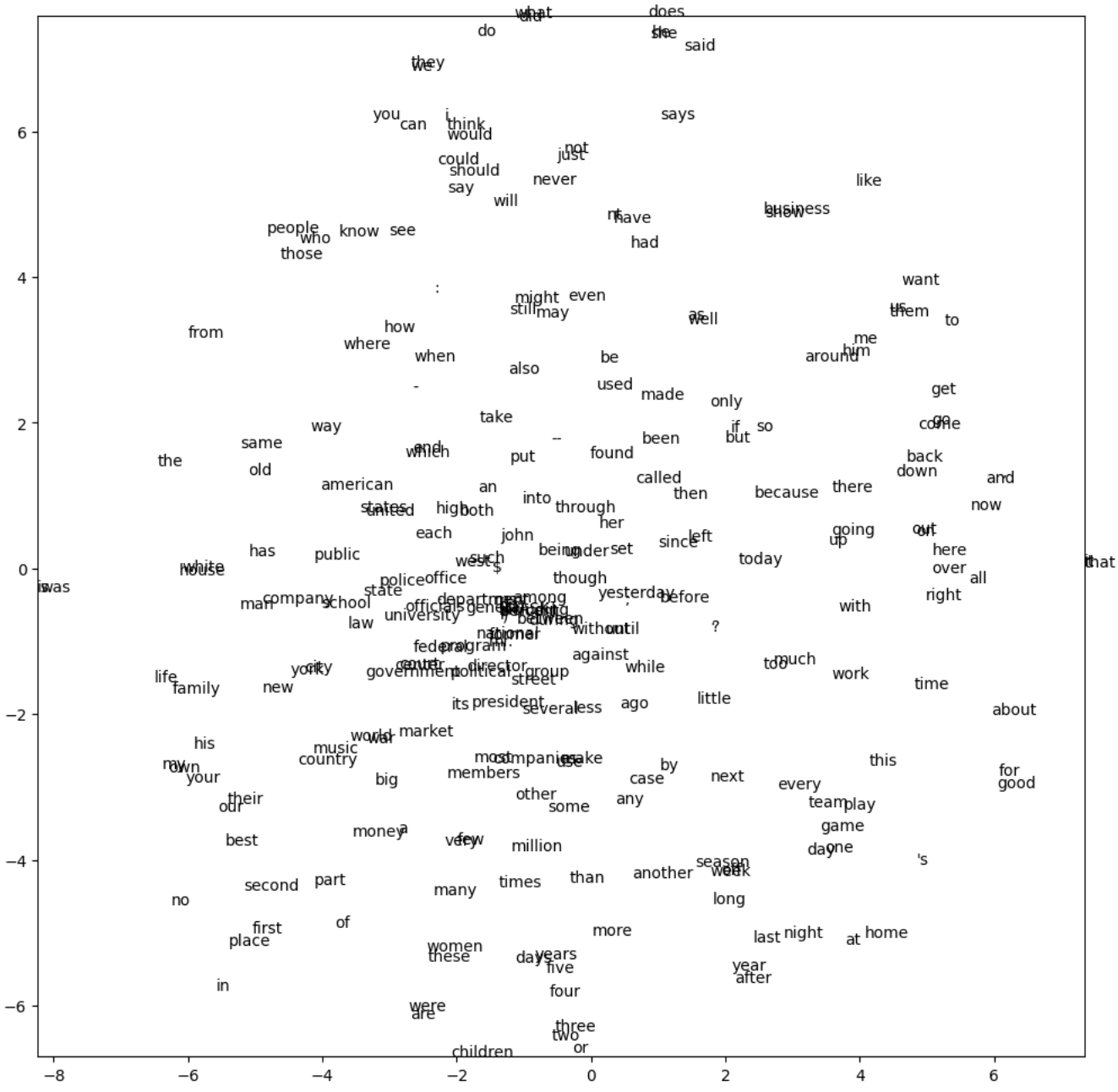
In [21]:
```
from sklearn.manifold import TSNE

def tsne_plot_representation(model):
    """Plot a 2-D visualization of the learned representations using t-SNE."""
    print(model.params.word_embedding_weights.shape)
    mapped_X = TSNE(n_components=2).fit_transform(model.params.word_embedding_weights
    pylab.figure(figsize=(12,12))
    for i, w in enumerate(model.vocab):
        pylab.text(mapped_X[i, 0], mapped_X[i, 1], w)
    pylab.xlim(mapped_X[:, 0].min(), mapped_X[:, 0].max())
    pylab.ylim(mapped_X[:, 1].min(), mapped_X[:, 1].max())
    pylab.show()

def tsne_plot_GLoVE_representation(W_final, b_final):
    """Plot a 2-D visualization of the learned representations using t-SNE."""
    mapped_X = TSNE(n_components=2).fit_transform(W_final)
    pylab.figure(figsize=(12,12))
    data_obj = pickle.load(open(data_location, 'rb'))
    for i, w in enumerate(data_obj['vocab']):
        pylab.text(mapped_X[i, 0], mapped_X[i, 1], w)
    pylab.xlim(mapped_X[:, 0].min(), mapped_X[:, 0].max())
    pylab.ylim(mapped_X[:, 1].min(), mapped_X[:, 1].max())
    pylab.show()

def plot_2d_GLoVE_representation(W_final, b_final):
    """Plot a 2-D visualization of the learned representations."""
    mapped_X = W_final
    pylab.figure(figsize=(12,12))
    data_obj = pickle.load(open(data_location, 'rb'))
    for i, w in enumerate(data_obj['vocab']):
        pylab.text(mapped_X[i, 0], mapped_X[i, 1], w)
    pylab.xlim(mapped_X[:, 0].min(), mapped_X[:, 0].max())
    pylab.ylim(mapped_X[:, 1].min(), mapped_X[:, 1].max())
    pylab.show()
```

In [22]:
```
tsne_plot_representation(trained_model)
```

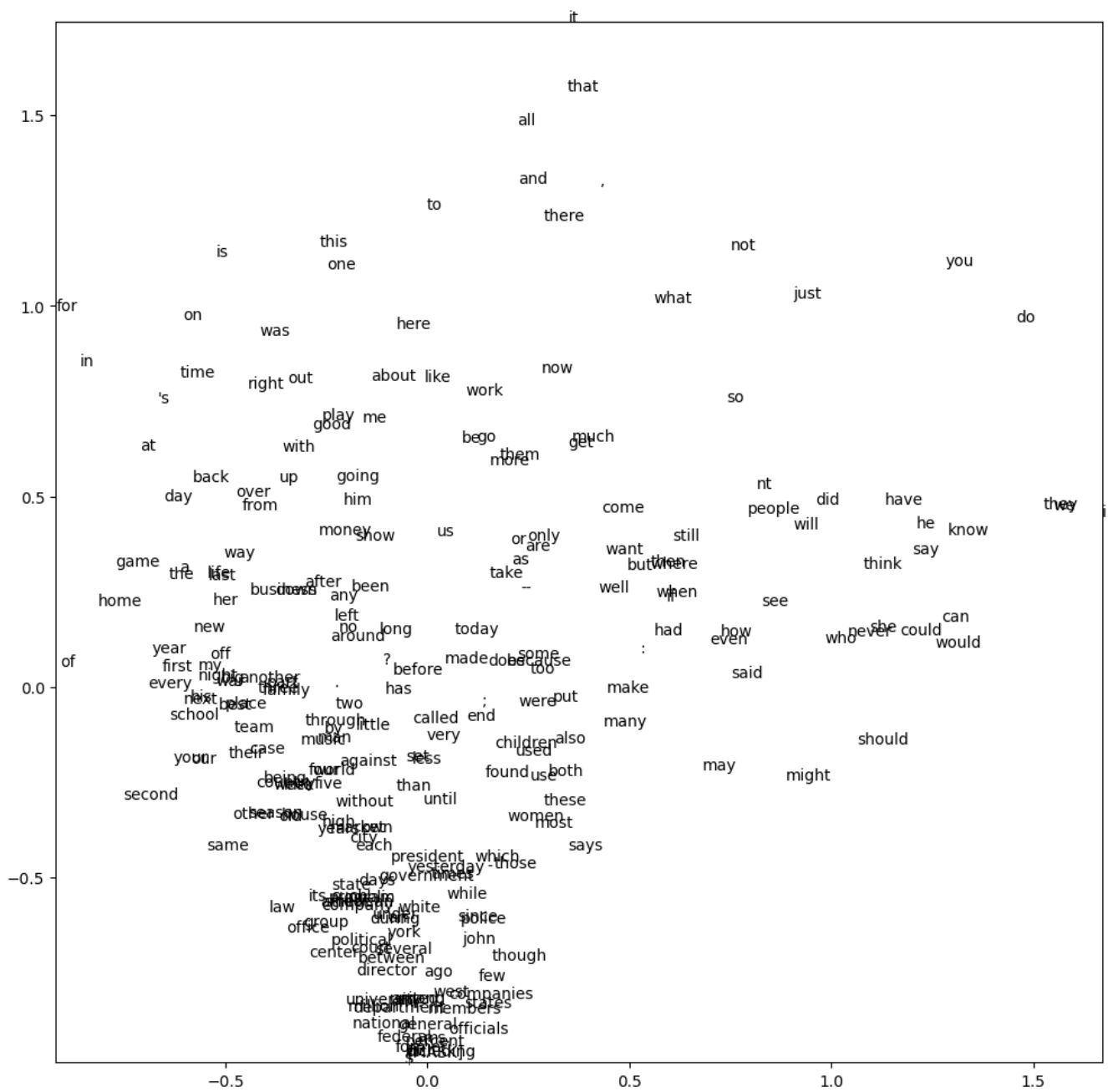
```
(251, 16)
```



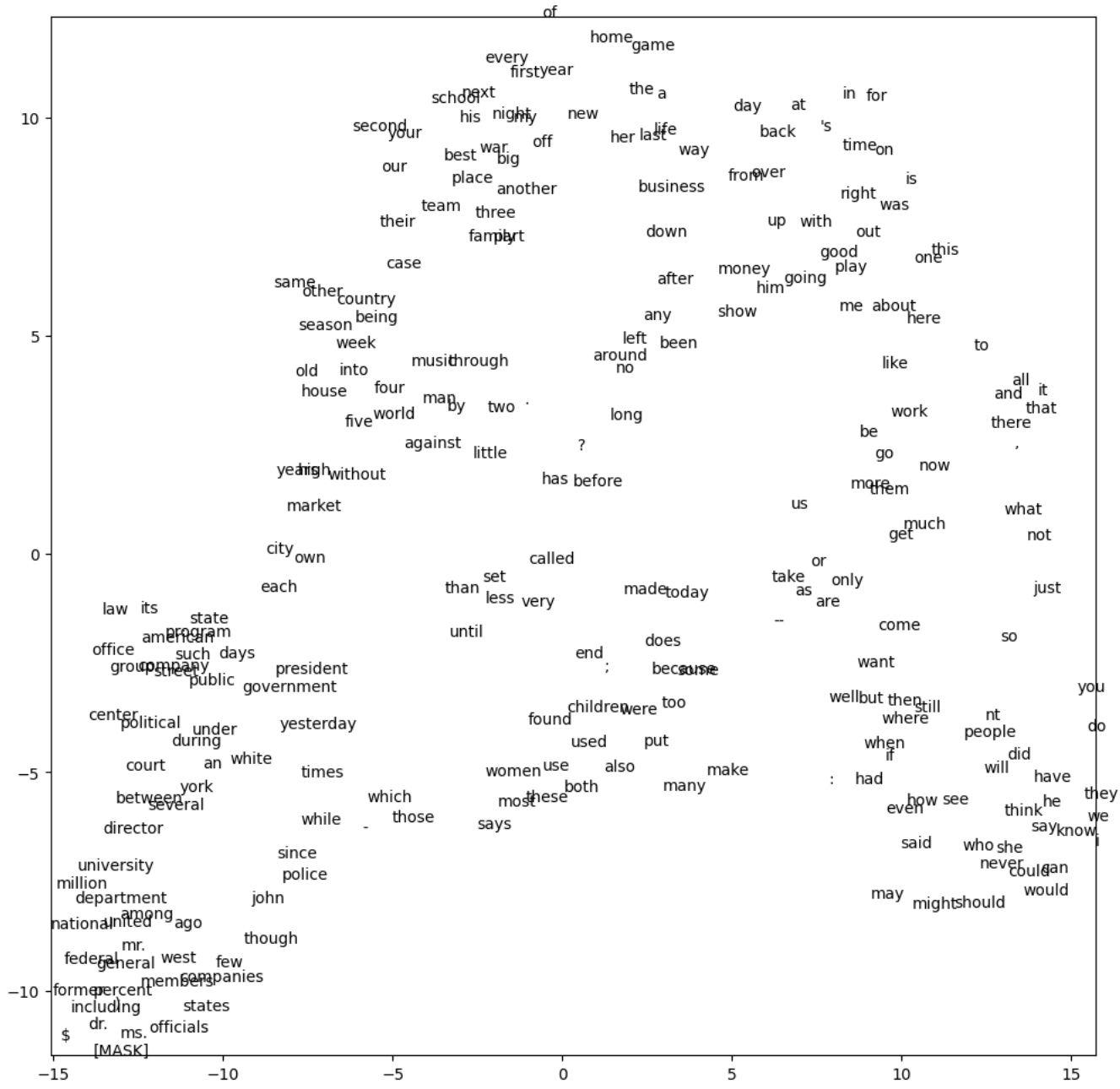
In [23]:
```
tsne_plot_GLoVE_representation(W_final, b_final)
```



In [24]:
```
plot_2d_GLoVE_representation(W_final_2d, b_final_2d)
```



In [25]:
```
tsne_plot_GLoVE_representation(W_final_2d, b_final_2d)
```



What you have to submit

For reference, here is everything you need to hand in. See the top of this handout for submission directions.

- Please make sure you finish the solve problems and submit your answers and codes according to the submission instruction:
  - [ ] **Part 1**: Questions 1.1, 1.2, 1.3, 1.4. Completed code for `grad_GLoVE` function.
  - [ ] **Part 2**: Completed code for `compute_loss_derivative()` (2.1), `back_propagate()` (2.2) functions, and the output of `print_gradients()` (2.3) and (2.4)
  - [ ] **Part 3**: Questions 3.1

In [ ]: