Neural Networks hw1

Dylan Satow

October 2025

1 Written Problems

1.1 Hard-Coding Neural Networks

The goal is to create a neural network that outputs 1 if the input sequence (x_1, x_2, x_3, x_4) is a superincreasing sequence, and 0 otherwise. A sequence is superincreasing if every element is greater than the sum of its predecessors. For a 4-element sequence, this translates to the following three conditions being true simultaneously:

1. $x_2 > x_1$
2. $x_3 > x_1 + x_2$
3. $x_4 > x_1 + x_2 + x_3$

We can design the network to check each of these conditions in one of the three hidden units, and then use the output unit to perform a logical AND on the results from the hidden units.

Activation Functions For both the hidden layer (ϕ_1) and the output layer (ϕ_2), we will use the hard threshold activation function, defined in the assignment as:

$$\phi(z) = I(z \geq 0) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Hidden Layer Each hidden unit h_k will compute $h_k = \phi_1(\mathbf{w}_k^{(1)} \cdot \mathbf{x} + b_k^{(1)})$. We want $h_k = 1$ if the k^{th} condition is met, and $h_k = 0$ otherwise.

- For h_1 (checks $x_2 > x_1 \iff x_2 - x_1 > 0$): We need the input to the activation, z_1, to be ≥ 0 if and only if this condition holds. We can set $z_1 = x_2 - x_1$. This gives the weights $\mathbf{w}_1^{(1)} = [-1, 1, 0, 0]$ and bias $b_1^{(1)} = 0$.
- For h_2 (checks $x_3 > x_1 + x_2 \iff x_3 - x_1 - x_2 > 0$): We set $z_2 = x_3 - x_1 - x_2$. This gives the weights $\mathbf{w}_2^{(1)} = [-1, -1, 1, 0]$ and bias $b_2^{(1)} = 0$.

- For h_3 (checks $x_4 > x_1 + x_2 + x_3 \iff x_4 - x_1 - x_2 - x_3 > 0$): We set $z_3 = x_4 - x_1 - x_2 - x_3$. This gives the weights $\mathbf{w}_3^{(1)} = [-1, -1, -1, 1]$ and bias $b_3^{(1)} = 0$.

Output Layer The output unit y computes $y = \phi_2(\mathbf{w}^{(2)} \cdot \mathbf{h} + b^{(2)})$. We need $y = 1$ if and only if $h_1 = 1$, $h_2 = 1$, and $h_3 = 1$. This is a logical AND operation. The input to the output activation is $z_{out} = w_1^{(2)} h_1 + w_2^{(2)} h_2 + w_3^{(2)} h_3 + b^{(2)}$. Let's set the weights $\mathbf{w}^{(2)} = [1, 1, 1]$. Then $z_{out} = h_1 + h_2 + h_3 + b^{(2)}$.

- If all hidden units are 1, their sum is 3. We need $z_{out} = 3 + b^{(2)} \geq 0$.
- If any hidden unit is 0, the maximum sum is 2. We need $z_{out} = 2 + b^{(2)} < 0$.

From these two conditions, we need $b^{(2)} \geq -3$ and $b^{(2)} < -2$. A value such as $b^{(2)} = -2.5$ would work. For an integer solution, if we choose $b^{(2)} = -3$, the first condition becomes $3 - 3 = 0 \geq 0$ (output 1) and the second becomes $2 - 3 = -1 < 0$ (output 0). This works perfectly.

Final Parameters

- **Activation Functions**: $\phi_1(z) = \phi_2(z) = I(z \geq 0)$.
- **Hidden Layer Weights**:

$$W^{(1)} = \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

- **Hidden Layer Biases**:

$$\mathbf{b}^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- **Output Layer Weights**:

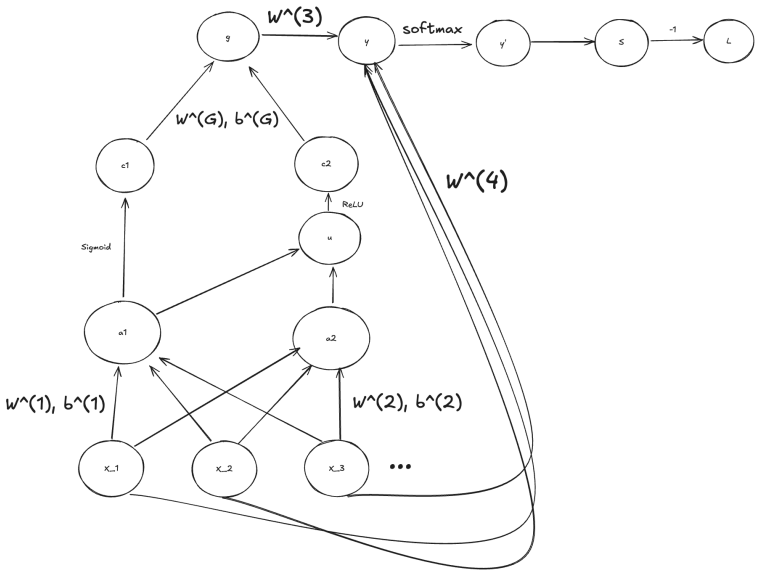
$$\mathbf{w}^{(2)} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

- **Output Layer Bias**:

$$b^{(2)} = -3$$

1.2 Backpropagation

Computation Graph



Backward Pass

We derive the backpropagation equations by applying the chain rule, starting from the output J and moving backward through the graph. We compute the gradient of the cost J with respect to each variable. Let $\delta_v = \frac{\partial J}{\partial v}$ denote the gradient of the loss with respect to a variable v.

1. **Loss**: $J = -S$
$$\frac{\partial J}{\partial S} = -1$$

2. **Score**: $S = \sum_k t_k \log(y'_k)$ (assuming t is one-hot, so $I(t_k = 1)$ is t_k) The gradient of the cross-entropy loss with respect to the softmax input y is a standard result:
$$\delta_y = \frac{\partial J}{\partial y} = y' - t$$

3. **Output Logits**: $y = W^{(3)}g + W^{(4)}x$. From here, we can compute gradients for $g, x, W^{(3)}, W^{(4)}$.
$$\delta_g = \frac{\partial J}{\partial g} = (W^{(3)})^T \delta_y$$
$$\delta_x^{(\text{from y})} = (W^{(4)})^T \delta_y$$

The gradients for the weights are:

$$\delta_{W^{(3)}} = \delta_y g^T$$
$$\delta_{W^{(4)}} = \delta_y x^T$$

4. **Hidden Layer g**: $g = W^{(g)}[c_1; c_2] + b^{(g)}$. Let $c_{concat} = [c_1; c_2]$.

$$\delta_{c_{concat}} = \frac{\partial J}{\partial c_{concat}} = (W^{(g)})^T \delta_g$$

This gradient is then split for c_1 and c_2:

$$\delta_{c_1} = (\delta_{c_{concat}})_{1:d_1} \quad \text{and} \quad \delta_{c_2} = (\delta_{c_{concat}})_{d_1+1:end}$$

(where d_1 is the dimension of c_1). The gradients for the weights are:

$$\delta_{W^{(g)}} = \delta_g c_{concat}^T$$
$$\delta_{b^{(g)}} = \delta_g$$

5. **ReLU Activation**: $c_2 = \text{ReLU}(u)$

$$\delta_u = \frac{\partial J}{\partial u} = \delta_{c_2} \odot I(u > 0) \quad (\odot \text{ is element-wise product})$$

6. **Sum**: $u = a_1 + a_2$

$$\delta_{a_1}^{(\text{from u})} = \delta_u \quad \text{and} \quad \delta_{a_2} = \delta_u$$

7. **Sigmoid Activation**: $c_1 = \sigma(a_1)$

$$\delta_{a_1}^{(\text{from c1})} = \delta_{c_1} \odot (\sigma(a_1)(1 - \sigma(a_1))) = \delta_{c_1} \odot (c_1(1 - c_1))$$

8. **Total gradients for** a_1, a_2:

$$\delta_{a_1} = \delta_{a_1}^{(\text{from c1})} + \delta_{a_1}^{(\text{from u})} = (\delta_{c_1} \odot (c_1(1 - c_1))) + \delta_u$$
$$\delta_{a_2} = \delta_u$$

9. **Linear Layers** a_1, a_2: For $a_1 = W^{(1)}x + b^{(1)}$:

$$\delta_x^{(\text{from a1})} = (W^{(1)})^T \delta_{a_1}$$
$$\delta_{W^{(1)}} = \delta_{a_1} x^T$$
$$\delta_{b^{(1)}} = \delta_{a_1}$$

For $a_2 = W^{(2)}x + b^{(2)}$:

$$\delta_x^{(\text{from a2})} = (W^{(2)})^T \delta_{a_2}$$
$$\delta_{W^{(2)}} = \delta_{a_2} x^T$$
$$\delta_{b^{(2)}} = \delta_{a_2}$$

10. **Total Gradient for x**: The final gradient for x is the sum from all paths:

$$\delta_x = \frac{\partial J}{\partial x} = \delta_x^{(\text{from y})} + \delta_x^{(\text{from a1})} + \delta_x^{(\text{from a2})}$$
$$\frac{\partial J}{\partial x} = (W^{(4)})^T (y' - t) + (W^{(1)})^T \delta_{a_1} + (W^{(2)})^T \delta_{a_2}$$

1.3 Automatic Differentiation

Compute Hessian

The function is given by $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{v}\mathbf{v}^T\mathbf{x}$. Let's define a matrix $A = \mathbf{v}\mathbf{v}^T$. The function becomes a standard quadratic form: $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x}$.

The gradient of $\mathcal{L}$ with respect to $\mathbf{x}$ is given by $\mathbf{g} = \nabla_{\mathbf{x}}\mathcal{L} = \frac{1}{2}(A+A^T)\mathbf{x}$. The matrix $A = \mathbf{v}\mathbf{v}^T$ is symmetric, since $A^T = (\mathbf{v}\mathbf{v}^T)^T = (\mathbf{v}^T)^T\mathbf{v}^T = \mathbf{v}\mathbf{v}^T = A$. Therefore, the gradient simplifies to $\mathbf{g} = \frac{1}{2}(A + A)\mathbf{x} = A\mathbf{x}$.

The Hessian, H, is the Jacobian of the gradient $\mathbf{g}$ with respect to $\mathbf{x}$:

$$H = \frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \frac{\partial (A\mathbf{x})}{\partial \mathbf{x}} = A$$

So, the Hessian is simply $H = \mathbf{v}\mathbf{v}^T$. Note that the Hessian is constant and does not depend on the value of $\mathbf{x}$.

For $n = 3$ and $\mathbf{v} = [3, 1, 4]^T$, we compute the outer product:

$$H = \mathbf{v}\mathbf{v}^T = \begin{bmatrix} 3 \\ 1 \\ 4 \end{bmatrix} \begin{bmatrix} 3 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 3 \cdot 3 & 3 \cdot 1 & 3 \cdot 4 \\ 1 \cdot 3 & 1 \cdot 1 & 1 \cdot 4 \\ 4 \cdot 3 & 4 \cdot 1 & 4 \cdot 4 \end{bmatrix}$$

The resulting Hessian matrix is:

$$H = \begin{bmatrix} 9 & 3 & 12 \\ 3 & 1 & 4 \\ 12 & 4 & 16 \end{bmatrix}$$

Computation Cost

Scalar Multiplications: To compute the Hessian $H = \mathbf{v}\mathbf{v}^T$, we need to compute each of its n^2 elements. Each element H_{ij} is the product of two scalars: $v_i \cdot v_j$. This requires one multiplication. Since there are n^2 elements in the $n \times n$ matrix, the total number of scalar multiplications is n^2. The computational cost is $O(n^2)$.

Memory Cost: The memory cost is determined by the size of the final matrix we need to store. The Hessian H is an $n \times n$ matrix. Storing this matrix requires space for n^2 scalar values. The memory cost is $O(n^2)$.

1.3.1 Vector-Hessian Products

Numerical Computation We are asked to compute $\mathbf{z} = H\mathbf{y}$ for $H = \mathbf{v}\mathbf{v}^T$, with $\mathbf{v} = [4, 1, 2]^T$ and $\mathbf{y} = [2, 2, 1]^T$.

Reverse-Mode: In this approach, we group operations as $\mathbf{z} = \mathbf{v}(\mathbf{v}^T\mathbf{y})$.

1. First, compute the scalar $M = \mathbf{v}^T\mathbf{y}$:

$$M = \begin{bmatrix}4 & 1 & 2\end{bmatrix}\begin{bmatrix}2\\2\\1\end{bmatrix} = (4)(2) + (1)(2) + (2)(1) = 8 + 2 + 2 = 12$$

2. Then, compute the final vector $\mathbf{z} = \mathbf{v}M$:

$$\mathbf{z} = \begin{bmatrix}4\\1\\2\end{bmatrix}(12) = \begin{bmatrix}48\\12\\24\end{bmatrix}$$

Forward-Mode: In this approach, we group operations as $\mathbf{z} = (\mathbf{v}\mathbf{v}^T)\mathbf{y}$.

1. First, compute the full Hessian matrix $H = \mathbf{v}\mathbf{v}^T$:

$$H = \begin{bmatrix}4\\1\\2\end{bmatrix}\begin{bmatrix}4 & 1 & 2\end{bmatrix} = \begin{bmatrix}16 & 4 & 8\\4 & 1 & 2\\8 & 2 & 4\end{bmatrix}$$

2. Then, compute the final vector $\mathbf{z} = H\mathbf{y}$:

$$\mathbf{z} = \begin{bmatrix}16 & 4 & 8\\4 & 1 & 2\\8 & 2 & 4\end{bmatrix}\begin{bmatrix}2\\2\\1\end{bmatrix} = \begin{bmatrix}16(2) + 4(2) + 8(1)\\4(2) + 1(2) + 2(1)\\8(2) + 2(2) + 4(1)\end{bmatrix} = \begin{bmatrix}32 + 8 + 8\\8 + 2 + 2\\16 + 4 + 4\end{bmatrix} = \begin{bmatrix}48\\12\\24\end{bmatrix}$$

Both methods yield the same result: $\mathbf{z}^T = [48, 12, 24]$.

Computational Cost Analysis We analyze the cost of computing $Z = H\mathbf{y}_1\mathbf{y}_2^T$, interpreted as $Z = (\mathbf{v}\mathbf{v}^T)(\mathbf{y}_1\mathbf{y}_2^T)$. The resulting matrix Z has dimensions $n \times m$. The relative efficiency of each mode depends on the shape of Z—that is, whether it is "tall" $(n > m)$ or "wide" $(m > n)$.

Forward-Mode Cost: This approach computes the full $n \times n$ Hessian matrix H first: $Z = (\mathbf{v}\mathbf{v}^T)(\mathbf{y}_1\mathbf{y}_2^T)$.

1. Compute $H = \mathbf{v}\mathbf{v}^T$: n^2 multiplications.
2. Compute $Y = \mathbf{y}_1\mathbf{y}_2^T$: nm multiplications.
3. Compute $Z = HY$: n^2m multiplications.

The total multiplications are $n^2 + nm + n^2m = n^2(1 + m) + nm$. This approach is burdened by the $O(n^2)$ cost of creating and using the explicit Hessian, which is expensive if n is large.

Reverse-Mode Cost: This approach avoids forming the full Hessian by using associativity: $Z = \mathbf{v}(\mathbf{v}^T(\mathbf{y}_1\mathbf{y}_2^T))$.

1. Compute $Y = \mathbf{y}_1\mathbf{y}_2^T$: nm multiplications.
2. Compute the row vector $\mathbf{r} = \mathbf{v}^T Y$: nm multiplications.
3. Compute the final matrix $Z = \mathbf{v}\mathbf{r}$: nm multiplications.

The total multiplications are $3nm$. This approach cleverly avoids the $O(n^2)$ costs.

Comparison: Forward-mode is preferable when $n^2(1+m) + nm < 3nm$, which simplifies to $n < m(2-n)$. This inequality reveals how the shape of the $n \times m$ output matrix Z dictates the best strategy:

- If $n \geq 2$, the term $(2-n)$ is zero or negative, making the inequality impossible to satisfy for positive n, m. This covers all cases where Z is square or "tall" ($n \geq m$). For these shapes, **reverse-mode** is always superior because it avoids the large $O(n^2)$ cost of building the Hessian.
- If $n = 1$, the inequality becomes $1 < m$. In this scenario, the Hessian is a trivial 1×1 scalar. **Forward-mode** becomes more efficient when $m > 1$. This corresponds to producing a very "wide" matrix Z (e.g., 1×100), where the negligible cost of the 1×1 Hessian makes the forward approach more economical.

2 Programming Problems

2.1 GLoVE Word Representations

2.1.1 GLoVE Parameter Count (2.1.1)

The problem states that for each word in a vocabulary of size V, the GLoVE model learns two embedding vectors and two scalar biases. For each word, the parameters are:

- A d-dimensional embedding vector $\mathbf{w}_i$, contributing d parameters.
- A second d-dimensional embedding vector $\tilde{\mathbf{w}}_i$, contributing d parameters.
- A scalar bias b_i, contributing 1 parameter.
- A second scalar bias $\tilde{b}_i$, contributing 1 parameter.

The total number of parameters for a single word is $d + d + 1 + 1 = 2d + 2$. Since there are V words in the vocabulary, the total number of parameters in the model is $V \times (2d + 2)$.

2.1.2 Expression for Gradient (2.1.2)

The simplified GLoVE loss function is given by:

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

To find the gradients, we differentiate this loss function with respect to a specific parameter vector $\mathbf{w}_k$ and a specific bias b_k.

Gradient with respect to $\mathbf{w}_i$: The parameter vector $\mathbf{w}_i$ only appears in the loss function terms where the first summation index is equal to i. Therefore, we only need to sum over the second index, j.

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

Using the chain rule, where $\frac{d}{dx} u^2 = 2u \frac{du}{dx}$:

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

The derivative of the inner term with respect to $\mathbf{w}_i$ is $\tilde{\mathbf{w}}_j$. This gives the final expression:

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

Gradient with respect to b_i: Similarly, the bias b_i only appears in terms where the first index is i.

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

Applying the chain rule:

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

The derivative of the inner term with respect to the scalar b_i is 1. This gives the final expression:

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

2.1.3 Implementing Gradient

See notebook

2.1.4 Effect of Embedding Dimension (2.1.4)

The following plots show the training and validation loss for the symmetric and asymmetric GLoVE models across different embedding dimensions.

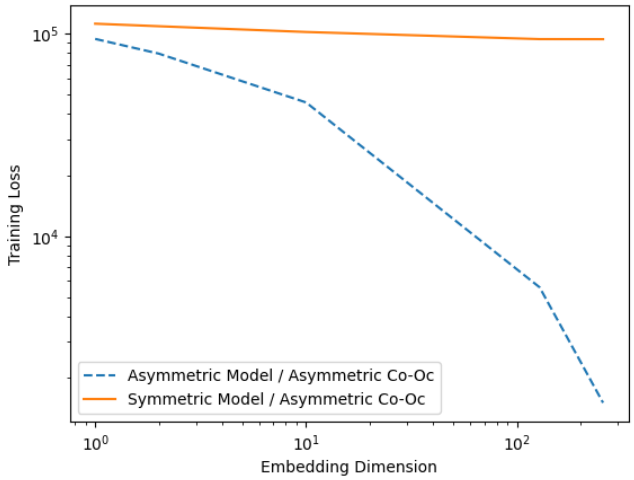


Figure 1: Training Loss vs. Embedding Dimension

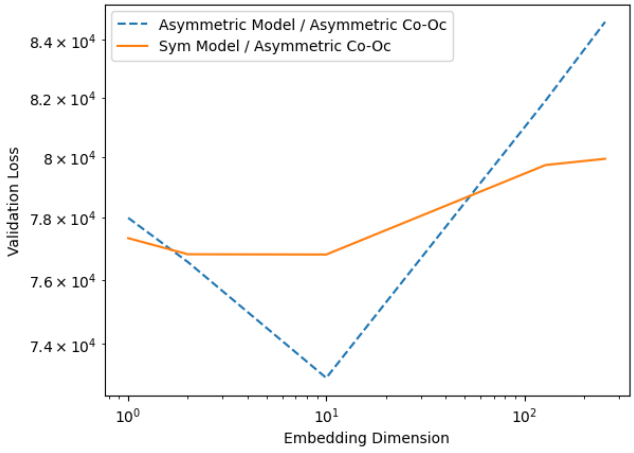


Figure 2: Validation Loss vs. Embedding Dimension

1. Optimal d: Based on the validation loss graph:

- For the **asymmetric model** (dashed blue line), the validation loss is minimized at $d = 10$.
- For the **symmetric model** (solid orange line), the validation loss is minimized and roughly constant for d between 2 and 10.

2. Effect of d on validation error: A larger embedding dimension d does not always lead to better validation error. This phenomenon is explained by the bias-variance tradeoff.

- When d is too small, the model has high bias and low variance. It is too simple to capture the underlying patterns in the data, which leads to underfitting.
- When d is too large, the model has low bias and high variance. It becomes overly complex and starts to fit the noise in the training data instead of the signal. This overfitting reduces its ability to generalize to the unseen validation set, causing the validation error to increase.

The validation loss graph clearly shows this U-shaped curve, where the error first decreases and then increases as d grows.

3. Which model is better, and why? The **asymmetric model performs better** because it achieves a lower overall validation loss (at its optimal dimension $d = 10$) compared to the symmetric model.

The reason is that the asymmetric model has more parameters ($2V(d+1)$ vs. $V(d+2)$ for the symmetric case where $W = \tilde{W}$), making it more expressive. The underlying word co-occurrence data is often not symmetric (e.g., the probability of "York" following "New" is different from "New" following "York"). The asymmetric model has the flexibility to capture these non-symmetric relationships, whereas the symmetric model is constrained and cannot.

2.2 Training the Model

2.2.1 Gradient with respect to output layer

see notebook

2.2.2 Gradient with respect to parameters

see notebook

2.2.3 Printing 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] 0.01159689251148945
param_gradient.word_embedding_weights[22, 4] -0.022267062381729714
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] 0.01604516132110911
param_gradient.embed_to_hid_weights[30, 9] -0.4312854367997418
param_gradient.embed_to_hid_weights[35, 21] 0.06679896665436336

param_gradient.hid_bias[10] 0.023428803123345162
param_gradient.hid_bias[20] -0.02437045237887423

param_gradient.output_bias[0] 0.000970106146902794
param_gradient.output_bias[1] 0.1686894627476322
param_gradient.output_bias[2] 0.0051664774143909235
param_gradient.output_bias[3] 0.1509622647181436
```

2.2.4 Run Model

See gradient printing above

2.3 Arithmetic and Analysis

For the trained model, there was an interesting cluster of the pronouns, with his, my, our, your, and their all being very close together. Another interseting one was big, little, and few all grouped together.

Comparing Glove t-sne to the trained model, the points and clusters in glove seem much more evenly space out, whereas the trained model has a little more clumping. This is interesting, and shows the difference in the training process for the two methods.

For the 2d representation vs the tsne representation, there was much more clustering together in the 2d, whereas the tsne was much more spread out with its clusters. For the normal 2d, there was one really big cluster with a large portion of the words residing in it.

Note, latex graph formatting is being weird here and its making grpahs show up at the end of the notebook. They are on the last 4 pages for this question

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.

Starter code and data

First, perform the required imports for your code:

In [51]:
```
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 [52]:
```
################################################################################
# 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
```

```
mkdir: /content: Read-only file system
[Errno 2] No such file or directory: '/content/A1'
/Users/dylan/home/columbia/neural_networks
```

In [53]:
```
# Download the dataset and partially pre-trained model
get_file(fname='a1_data',
                        origin='http://www.cs.columbia.edu/~zemel/Class/Nnd
                        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
Extracting file.
/var/folders/j_/4tqbj6gx5m3_qv7vcpbs2f540000gn/T/ipykernel_65859/343202416
8.py:51: DeprecationWarning: Python 3.14 will, by default, filter extracted
tar archives and reject files or modify their metadata. Use the filter argum
ent to control this behavior.
  archive.extractall(datadir)
```

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 [54]:
```
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
```

```
[MASK]
all
251
['[MASK]', np.str_('all'), np.str_('set'), np.str_('just'), np.str_('show'),
np.str_('being'), np.str_('money'), np.str_('over'), np.str_('both'), np.str
_('years'), np.str_('four'), np.str_('through'), np.str_('during'), np.str_
('go'), np.str_('still'), np.str_('children'), np.str_('before'), np.str_('p
olice'), np.str_('office'), np.str_('million'), np.str_('also'), np.str_('le
ss'), np.str_('had'), np.str_(','), np.str_('including'), np.str_('should'),
np.str_('to'), np.str_('only'), np.str_('going'), np.str_('under'), np.str_
('has'), np.str_('might'), np.str_('do'), np.str_('them'), np.str_('good'),
np.str_('around'), np.str_('get'), np.str_('very'), np.str_('big'), np.str_
('dr.'), np.str_('game'), np.str_('every'), np.str_('know'), np.str_('the
y'), np.str_('not'), np.str_('world'), np.str_('now'), np.str_('him'), np.st
r_('school'), np.str_('several'), np.str_('like'), np.str_('did'), np.str_
('university'), np.str_('companies'), np.str_('these'), np.str_('she'), np.s
tr_('team'), np.str_('found'), np.str_('where'), np.str_('right'), np.str_
('says'), np.str_('people'), np.str_('house'), np.str_('national'), np.str_
('some'), np.str_('back'), np.str_('see'), np.str_('street'), np.str_('ar
e'), np.str_('year'), np.str_('home'), np.str_('best'), np.str_('out'), np.s
tr_('even'), np.str_('what'), np.str_('said'), np.str_('for'), np.str_('fede
ral'), np.str_('since'), np.str_('its'), np.str_('may'), np.str_('state'), n
p.str_('does'), np.str_('john'), np.str_('between'), np.str_('new'), np.str_
(';'), np.str_('three'), np.str_('public'), np.str_('?'), np.str_('be'), n
p.str_('we'), np.str_('after'), np.str_('business'), np.str_('never'), np.st
r_('use'), np.str_('here'), np.str_('york'), np.str_('members'), np.str_('pe
rcent'), np.str_('put'), np.str_('group'), np.str_('come'), np.str_('by'), n
p.str_('$'), np.str_('on'), np.str_('about'), np.str_('last'), np.str_('he
r'), np.str_('of'), np.str_('could'), np.str_('days'), np.str_('against'), n
p.str_('times'), np.str_('women'), np.str_('place'), np.str_('think'), np.st
r_('first'), np.str_('among'), np.str_('own'), np.str_('family'), np.str_('i
nto'), np.str_('each'), np.str_('one'), np.str_('down'), np.str_('because'),
np.str_('long'), np.str_('another'), np.str_('such'), np.str_('old'), np.str
_('next'), np.str_('your'), np.str_('market'), np.str_('second'), np.str_('c
ity'), np.str_('little'), np.str_('from'), np.str_('would'), np.str_('few'),
np.str_('west'), np.str_('there'), np.str_('political'), np.str_('two'), n
p.str_('been'), np.str_('.'), np.str_('their'), np.str_('much'), np.str_('mu
sic'), np.str_('too'), np.str_('way'), np.str_('white'), np.str_(':'), np.st
r_('was'), np.str_('war'), np.str_('today'), np.str_('more'), np.str_('ag
o'), np.str_('life'), np.str_('that'), np.str_('season'), np.str_('compan
y'), np.str_('-'), np.str_('but'), np.str_('part'), np.str_('court'), np.str
_('former'), np.str_('general'), np.str_('with'), np.str_('than'), np.str_
('those'), np.str_('he'), np.str_('me'), np.str_('high'), np.str_('made'), n
p.str_('this'), np.str_('work'), np.str_('up'), np.str_('us'), np.str_('unti
l'), np.str_('will'), np.str_('ms.'), np.str_('while'), np.str_('official
s'), np.str_('can'), np.str_('were'), np.str_('country'), np.str_('my'), n
p.str_('called'), np.str_('and'), np.str_('program'), np.str_('have'), np.st
r_('then'), np.str_('is'), np.str_('it'), np.str_('an'), np.str_('states'),
np.str_('case'), np.str_('say'), np.str_('his'), np.str_('at'), np.str_('wan
t'), np.str_('in'), np.str_('any'), np.str_('as'), np.str_('if'), np.str_('u
nited'), np.str_('end'), np.str_('no'), np.str_(')'), np.str_('make'), np.st
r_('government'), np.str_('when'), np.str_('american'), np.str_('same'), n
p.str_('how'), np.str_('mr.'), np.str_('other'), np.str_('take'), np.str_('w
hich'), np.str_('department'), np.str_('--'), np.str_('you'), np.str_('man
y'), np.str_('nt'), np.str_('day'), np.str_('week'), np.str_('play'), np.str
_('used'), np.str_("'s"), np.str_('though'), np.str_('our'), np.str_('who'),
```

```
np.str_('yesterday'), np.str_('director'), np.str_('most'), np.str_('preside
nt'), np.str_('law'), np.str_('man'), np.str_('a'), np.str_('night'), np.str
_('off'), np.str_('center'), np.str_('i'), np.str_('well'), np.str_('or'), n
p.str_('without'), np.str_('so'), np.str_('time'), np.str_('five'), np.str_
('the'), np.str_('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]]
```

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**: Each of the V words in the vocabulary is associated with two d-dimensional embedding vectors (w_i and w_tilde_i) and two scalar biases (b_i and b_tilde_i).

The total number of parameters is the sum of the sizes of all these parameters:

- Parameters for all w_i vectors: V * d
- Parameters for all w_tilde_i vectors: V * d
- Parameters for all b_i biases: V * 1
- Parameters for all b_tilde_i biases: V * 1

Total parameters = V*d* + *V*d + V + V = 2Vd + 2V = 2V(d+1).

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**: The loss function is given by: $L = \sum_{k,j=1}^{V} (\mathbf{w}_k^\top \tilde{\mathbf{w}}_j + b_k + \tilde{b}_j - \log X_{kj})^2$

To find the gradient with respect to $\mathbf{w}_i$, we only need to consider the terms in the sum where $k = i$: $\frac{\partial L}{\partial \mathbf{w}_i} = \frac{\partial}{\partial \mathbf{w}_i} \sum_{j=1}^{V} (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2$ Using the chain rule:
$\frac{\partial L}{\partial \mathbf{w}_i} = \sum_{j=1}^{V} 2(\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij}) \frac{\partial}{\partial \mathbf{w}_i} (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j)$
$\frac{\partial L}{\partial \mathbf{w}_i} = 2 \sum_{j=1}^{V} (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij}) \tilde{\mathbf{w}}_j$

Similarly, for the gradient with respect to the bias term b_i:
$\frac{\partial L}{\partial b_i} = \frac{\partial}{\partial b_i} \sum_{j=1}^{V} (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2$
$\frac{\partial L}{\partial b_i} = \sum_{j=1}^{V} 2(\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij}) \frac{\partial}{\partial b_i} (b_i)$
$\frac{\partial L}{\partial b_i} = 2 \sum_{j=1}^{V} (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})$

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 [55]:
```
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
  log_co_occurence += delta_smoothing  # Add delta so log doesn't break on 0
  log_co_occurence = np.log(log_co_occurence)
  return log_co_occurence
```

In [56]:
```
asym_log_co_occurence_train = calculate_log_co_occurence(data['train_inputs'
asym_log_co_occurence_valid = calculate_log_co_occurence(data['valid_inputs'
```

- ☐ **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 [57]:
```
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_c
  else:
    return np.sum((W @ W_tilde.T + b @ np.ones([1,n]) + np.ones([n,1])@b_til

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:
  "OUTPUT: grad_W - Vxd; grad_W_tilde - Vxd, grad_b - Vx1, grad_b_tilde - Vx
  n,_ = log_co_occurence.shape

  if not W_tilde is None and not b_tilde is None:
  ############################   YOUR CODE HERE  ############################
    loss = (W @ W_tilde.T + b @ np.ones([1,n]) + np.ones([n,1])@b_tilde.T -
    grad_W = 2 * loss @ W_tilde
    grad_W_tilde = 2 * loss.T @ W
    grad_b = 2 * loss @ np.ones((n, 1))
    grad_b_tilde = 2 * loss.T @ np.ones((n, 1))
  ##########################################################################
  else:
    loss = (W @ W.T + b @ np.ones([1,n]) + np.ones([n,1])@b.T - 0.5*(log_co_
    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_occur
  "Traing W and b according to GLoVE objective."
  n,_ = log_co_occurence_train.shape
  learning_rate = 0.05 / n  # A hyperparameter.  You can play with this if y
  for epoch in range(n_epochs):
```

```
    grad_W, grad_W_tilde, grad_b, grad_b_tilde = grad_GLoVE(W, W_tilde, b, b
    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_occur
    if do_print:
      print(f"Train Loss: {train_loss}, valid loss: {valid_loss}, grad_norm:
  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. **Optimal `d`:**
   - For the **asymmetric model** (dashed blue line), the validation loss is minimized at **d=10**.
   - For the **symmetric model** (solid orange line), the validation loss is minimized for **d between 2 and 10**, where it remains roughly constant.
2. **Effect of `d` on validation error:** A larger `d` does not always lead to better validation error. This is due to the bias-variance tradeoff.
   - If `d` is too small, the model is too simple (high bias) and may underfit, failing to capture the structure of the data.
   - If `d` is too large, the model becomes too complex (high variance) and may overfit the training data, learning noise rather than the underlying signal. This hurts its ability to generalize to unseen data, leading to a higher validation error. The graphs show this pattern, where the validation error first decreases and then increases as `d` grows.
3. **Which model is better?** The **asymmetric model performs better**, as it achieves a lower minimum validation loss (at d=10) than the symmetric model. The asymmetric model has more parameters and is more expressive. The co-occurrence data is inherently asymmetric (e.g., the probability of "York" following "New" is not the same as "New" following "York"). The asymmetric model can capture this, while the symmetric model is constrained (`W = W_tilde`), which limits its ability to model these asymmetric relationships.

Train the GLoVE model for a range of embedding dimensions

In [58]:
```
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
# 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_
W_final_2d, b_final_2d = None, None
do_print = False  # If you want to see diagnostic information during trainin

for embedding_dim in tqdm(embedding_dims):
  init_variance = 0.1  # A hyperparameter.  You can play with this if you wa
  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
```

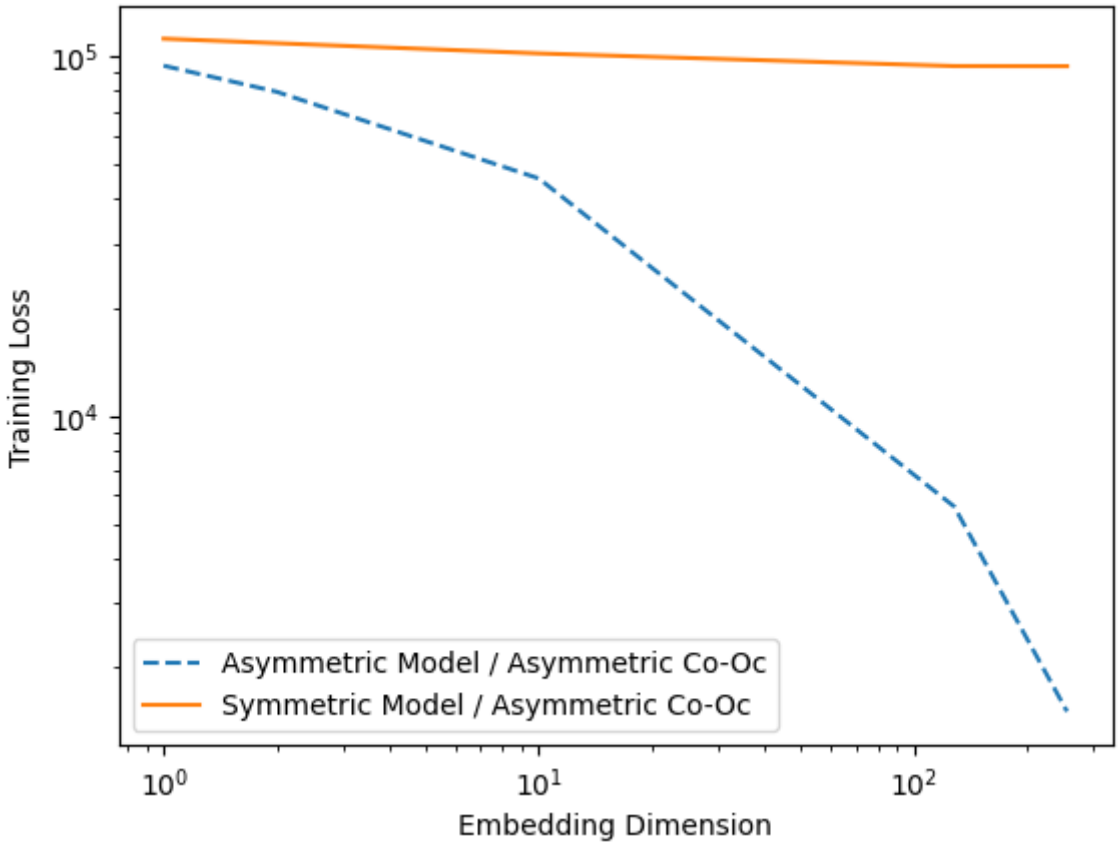
```
  Asym_W_final, Asym_W_tilde_final, Asym_b_final, Asym_b_tilde_final, train_
  if embedding_dim == 2:
    # Save a parameter copy if we are training 2d embedding for visualizatio
    Asym_W_final_2d = Asym_W_final
    Asym_W_tilde_final_2d = Asym_W_tilde_final
    Asym_b_final_2d = Asym_b_final
    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 = t
  if embedding_dim == 2:
    # Save a parameter copy if we are training 2d embedding for visualizatio
    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 [00:04<00:00,  1.10it/s]
```

Plot the training and validation losses against the embedding dimension.

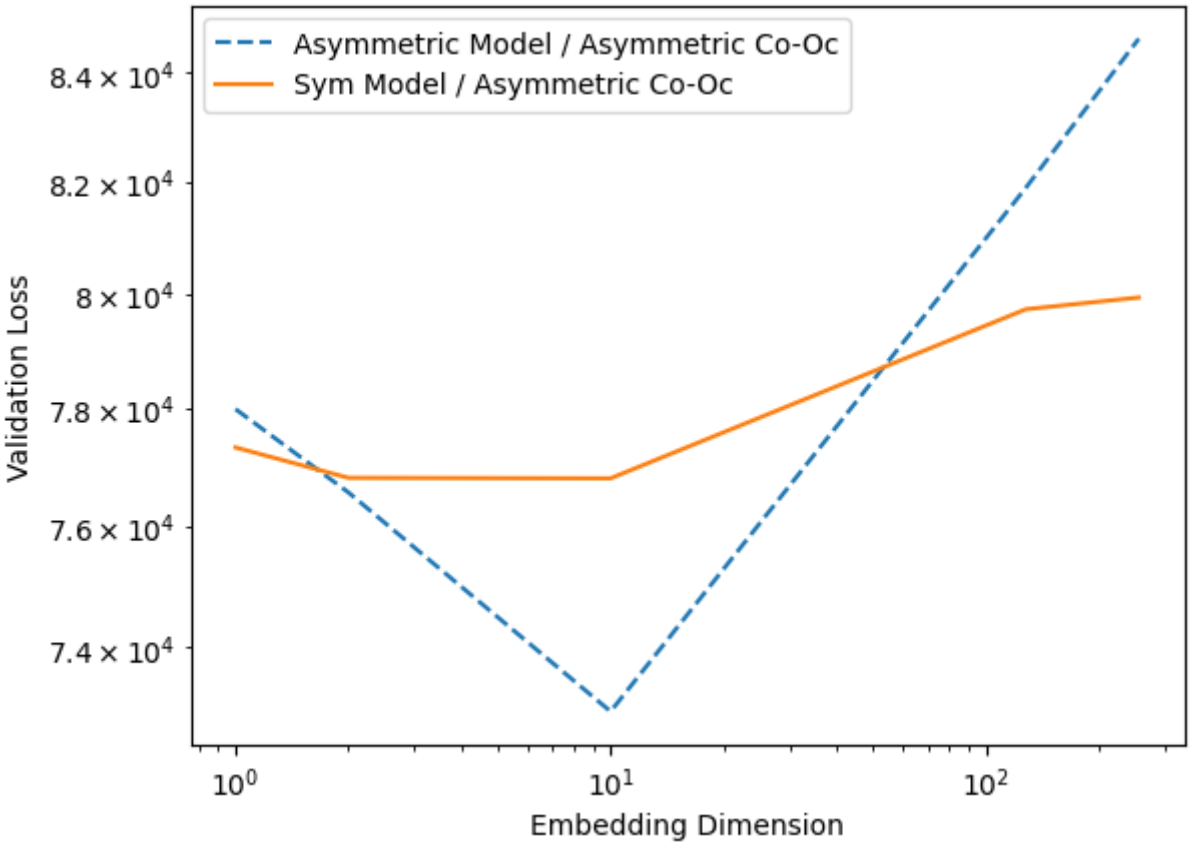
In [59]:
```
pylab.loglog(embedding_dims, asymModel_asymCoOc_final_train_losses, label="A
pylab.loglog(embedding_dims, symModel_asymCoOc_final_train_losses , label="S
pylab.xlabel("Embedding Dimension")
pylab.ylabel("Training Loss")
pylab.legend()
```

Out[59]:
```
<matplotlib.legend.Legend at 0x12319bb60>
```



In [60]:
```
pylab.loglog(embedding_dims, asymModel_asymCoOc_final_val_losses, label="Asy
pylab.loglog(embedding_dims, symModel_asymCoOc_final_val_losses , label="Sym
pylab.xlabel("Embedding Dimension")
pylab.ylabel("Validation Loss")
pylab.legend(loc="upper left")
```

Out[60]: `<matplotlib.legend.Legend at 0x1232bf470>`



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 [61]:
```
class Params(object):
    """A class representing the trainable parameters of the model. This clas

          word_embedding_weights, a matrix of size V x D, where V is the nu
                  and D is the embedding dimension.
          embed_to_hid_weights, a matrix of size H x ND, where H is the num
                  columns represent connections from the embedding of the f
                  for the second context word, and so on. There are N conte
          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_
                 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
                              self.hid_to_output_weights.copy(), self.hid_bi

    @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_di
        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_outp
                   hid_bias, output_bias)

    @classmethod
    def random_init(cls, init_wt, vocab_size, context_len, embedding_dim, nu
```

```
        """A constructor which initializes weights to small random values an
        word_embedding_weights = np.random.normal(0., init_wt, size=(vocab_s
        embed_to_hid_weights = np.random.normal(0., init_wt, size=(num_hid,
        hid_to_output_weights = np.random.normal(0., init_wt, size=(vocab_si
        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_outp
                   hid_bias, output_bias)

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

    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_embed
                              self.embed_to_hid_weights + other.embed_to_hid
                              self.hid_to_output_weights + other.hid_to_outp
                              self.hid_bias + other.hid_bias,
                              self.output_bias + other.output_bias)

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

In [62]:

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

        embedding_layer, a matrix of B x ND matrix (where B is the batch siz
                and N is the number of input context words), representing th
                layer on all the cases in a batch. The first D columns repre
                first context word, and so on.
        hidden_layer, a B x H matrix representing the hidden layer activatio
        output_layer, a B x V matrix representing the output layer activatio

    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 gi
    'generator', i.e. something you can use in a for loop. You don't need to
    works to do the assignment."""

    if inputs.shape[0] % batch_size != 0:
        raise RuntimeError('The number of data points must be a multiple of
    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 [63]:
```
class Model(object):
    """A class representing the language model itself. This class contains v
    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] i
               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 w
        and initializes the biases to all zeros."""
        params = Params.random_init(init_wt, len(vocab), context_len, embedd
        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 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 [
               and we set the entry to be 0.
        """
```

```
        batch_size, context_len = targets.shape
        expanded_targets = np.zeros((batch_size, context_len * len(self.voca
        targets_offset = np.repeat((np.arange(context_len) * len(self.vocab)
        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
                expanded_targets[np.arange(batch_size), targets_offset[:,c]]
        return expanded_targets

    def compute_loss_derivative(self, output_activations, expanded_target_ba
        """Compute the derivative of the multiple target position cross-entr

            For example:

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

         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}}

        This function should return a dC / dz matrix of size [batch_size x (
        where each row i in dC / dz has columns 0 to V-1 containing the grad
        context word from i-th training example, then columns vocab_size to
        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})

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

        The arguments are as follows:

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

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

        ############################   YOUR CODE HERE  ######################
        batch_size = output_activations.shape[0]
```

```
        # Reshape mask from (B, N, 1) to (B, N) to be able to repeat it
        reshaped_mask = np.repeat(
            target_mask.reshape(batch_size, self.context_len), self.vocab_si
        )
        # The derivative is (y - t) for the masked words, and 0 otherwise
        loss_derivative = reshaped_mask * (output_activations - expanded_tar
        return loss_derivative
        ###########################################################################

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

    def compute_activations(self, inputs):
        """Compute the activations on a batch given the inputs. Returns an A
        You should try to read and understand this function, since this will
        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 {},
                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_d
        for i in range(self.context_len):
            embedding_layer_state[:, i * self.embedding_dim:(i + 1) * self.e
                self.params.word_embedding_weights[inputs[:, i], :]

        # Hidden layer
        inputs_to_hid = np.dot(embedding_layer_state, self.params.embed_to_h
                               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_ou
                                   self.params.output_bias

        # Subtract maximum.
        # Remember that adding or subtracting the same constant from each in
        # softmax unit does not affect the outputs. So subtract the maximum
        # make all inputs <= 0. This prevents overflows when computing their
        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_le
        output_layer_state /= output_layer_state.sum(axis=-1, keepdims=True)
        output_layer_state = output_layer_state.reshape(output_layer_state_s

        return Activations(embedding_layer_state, hidden_layer_state, output
```

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

             input_batch - the indices of the context words
             activations - an Activations class representing the output of M
             loss_derivative - the matrix of derivatives computed by compute

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

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

        ############################   YOUR CODE HERE  ######################
        hid_to_output_weights_grad = np.dot(loss_derivative.T, activations.h
        output_bias_grad = np.sum(loss_derivative, axis=0)
        embed_to_hid_weights_grad = np.dot(hid_deriv.T, activations.embeddin
        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.embedd
        for w in range(self.context_len):
            word_embedding_weights_grad += np.dot(self.indicator_matrix(inpu
                                                  embed_deriv[:, w * self.em

        return Params(word_embedding_weights_grad, embed_to_hid_weights_grad
                      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
        For each row, at most one element will be 1.
        """
        mask_idx = np.random.randint(self.context_len, size=(batch_size,))

        # changed mask for numpy version (TA said it was okay on ED)
        mask = np.zeros((batch_size, self.context_len), dtype=int)# Convert
        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_maske
            cross_entropy = -np.sum(expanded_target_batch * np.log(activatio
            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 distan

        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,
        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 wo

        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))
```

```
<>:57: SyntaxWarning: invalid escape sequence '\s'
<>:57: SyntaxWarning: invalid escape sequence '\s'
/var/folders/j_/4tqbj6gx5m3_qv7vcpbs2f540000gn/T/ipykernel_65859/415188471
5.py:57: SyntaxWarning: invalid escape sequence '\s'
  y_{j + n*V} = e^{z_{j + n*V}} / \sum_{m=0}^{V-1} e^{z_{m + n*V}}, for n=
0,...,N-1
```

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 [64]:
```
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_si
    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(m
    loss_derivative = model.compute_loss_derivative(y, expanded_target_batch

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

    if loss_derivative.shape != (batch_size, model.vocab_size * model.contex
        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_
        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.ran

        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
            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(m
    loss_derivative = model.compute_loss_derivative(activations.output_layer
    param_gradient = model.back_propagate(input_batch, activations, loss_der

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

    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 {}.'.fo
            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
            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_bi
                                    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_
        return

    for param_name in ['word_embedding_weights', 'embed_to_hid_weights', 'hi
                       '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_b


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(m
    loss_derivative = model.compute_loss_derivative(activations.output_layer
    param_gradient = model.back_propagate(input_batch, activations, loss_der

    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.wor
    print('param_gradient.word_embedding_weights[43, 3]', param_gradient.wor
    print('param_gradient.word_embedding_weights[22, 4]', param_gradient.wor
    print('param_gradient.word_embedding_weights[2, 5]', param_gradient.word
    print()
    print('param_gradient.embed_to_hid_weights[10, 2]', param_gradient.embed
    print('param_gradient.embed_to_hid_weights[15, 3]', param_gradient.embed
```

```
    print('param_gradient.embed_to_hid_weights[30, 9]', param_gradient.embed
    print('param_gradient.embed_to_hid_weights[35, 21]', param_gradient.embe
    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 [65]:
```
# Run this to check if your implement gradients matches the finite differenc
# 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 [66]:
```
# 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] 0.01159689251148945
param_gradient.word_embedding_weights[22, 4] -0.022267062381729714
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] 0.01604516132110911
param_gradient.embed_to_hid_weights[30, 9] -0.4312854367997418
param_gradient.embed_to_hid_weights[35, 21] 0.06679896665436336

param_gradient.hid_bias[10] 0.023428803123345162
param_gradient.hid_bias[20] -0.02437045237887423

param_gradient.output_bias[0] 0.000970106146902794
param_gradient.output_bias[1] 0.1686894627476322
param_gradient.output_bias[2] 0.0051664774143909235
param_gradient.output_bias[3] 0.1509622647181436
```

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 [69]:
```
_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 m
                           'epochs': 50,  # the maximum number of epochs to
                           'init_wt': 0.01,  # the standard deviation of the
                           'context_len': 4,  # the number of context words
                           'show_training_CE_after': 100,  # measure trainin
                           'show_validation_CE_after': 1000,  # measure vali
                          }


def find_occurrences(word1, word2, word3):
    """Lists all the words that followed a given tri-gram in the training se
    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['

    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.inde
    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], rever
        print('The tri-gram "{} {} {}" was followed by the following words i
            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.'.f


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

        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'

    # Variables used for early stopping
    best_valid_CE = np.inf # changed for np version
    end_training = False

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

    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_count += 1

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

            # Forward propagate
```

```
            activations = model.compute_activations(input_batch_masked)

            # Compute loss derivative
            expanded_target_batch = model.indicator_matrix(target_batch_mask
            loss_derivative = model.compute_loss_derivative(activations.outp
            loss_derivative /= config['batch_size']

            # Measure loss function
            cross_entropy = model.compute_loss(activations.output_layer, exp
            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_af
                this_chunk_CE = 0.

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

            # 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_entro

                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 [70]:
```
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 3.248
Batch 9200 Train CE 3.229
Batch 9300 Train CE 3.224
Batch 9400 Train CE 3.216
Batch 9500 Train CE 3.206
Batch 9600 Train CE 3.201
Batch 9700 Train CE 3.195
Batch 9800 Train CE 3.234
Batch 9900 Train CE 3.183
Batch 10000 Train CE 3.179
Running validation...
Validation cross-entropy: 3.175
Batch 10100 Train CE 3.169
Batch 10200 Train CE 3.168
Batch 10300 Train CE 3.170
Batch 10400 Train CE 3.201
Batch 10500 Train CE 3.173
Batch 10600 Train CE 3.174
Batch 10700 Train CE 3.140
Batch 10800 Train CE 3.179
Batch 10900 Train CE 3.190
Batch 11000 Train CE 3.105
Running validation...
Validation cross-entropy: 3.147
Batch 11100 Train CE 3.168

Epoch 4
Batch 11200 Train CE 3.154
Batch 11300 Train CE 3.136
Batch 11400 Train CE 3.139
Batch 11500 Train CE 3.150
Batch 11600 Train CE 3.119
Batch 11700 Train CE 3.117
Batch 11800 Train CE 3.162
Batch 11900 Train CE 3.110
Batch 12000 Train CE 3.141
Running validation...
Validation cross-entropy: 3.119
Batch 12100 Train CE 3.132
Batch 12200 Train CE 3.134
Batch 12300 Train CE 3.126
Batch 12400 Train CE 3.105
Batch 12500 Train CE 3.075
Batch 12600 Train CE 3.142
Batch 12700 Train CE 3.126
Batch 12800 Train CE 3.135
Batch 12900 Train CE 3.093
Batch 13000 Train CE 3.116
Running validation...
Validation cross-entropy: 3.093
Batch 13100 Train CE 3.105
Batch 13200 Train CE 3.087
Batch 13300 Train CE 3.093
Batch 13400 Train CE 3.093
Batch 13500 Train CE 3.074
```

```
Batch 13600 Train CE 3.068
Batch 13700 Train CE 3.080
Batch 13800 Train CE 3.080
Batch 13900 Train CE 3.081
Batch 14000 Train CE 3.079
Running validation...
Validation cross-entropy: 3.078
Batch 14100 Train CE 3.083
Batch 14200 Train CE 3.105
Batch 14300 Train CE 3.141
Batch 14400 Train CE 3.083
Batch 14500 Train CE 3.074
Batch 14600 Train CE 3.142
Batch 14700 Train CE 3.094
Batch 14800 Train CE 3.066
Batch 14900 Train CE 3.083

Epoch 5
Batch 15000 Train CE 3.049
Running validation...
Validation cross-entropy: 3.055
Batch 15100 Train CE 3.096
Batch 15200 Train CE 3.070
Batch 15300 Train CE 3.090
Batch 15400 Train CE 3.099
Batch 15500 Train CE 3.058
Batch 15600 Train CE 3.074
Batch 15700 Train CE 3.070
Batch 15800 Train CE 3.082
Batch 15900 Train CE 3.076
Batch 16000 Train CE 3.080
Running validation...
Validation cross-entropy: 3.083
Validation error increasing!  Training stopped.

Final training cross-entropy: 3.063
Final validation cross-entropy: 3.072
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; youâ€™ll use the trained models for the remainding of this section.

Important: if youâ€™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 donâ€™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 donâ€™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.

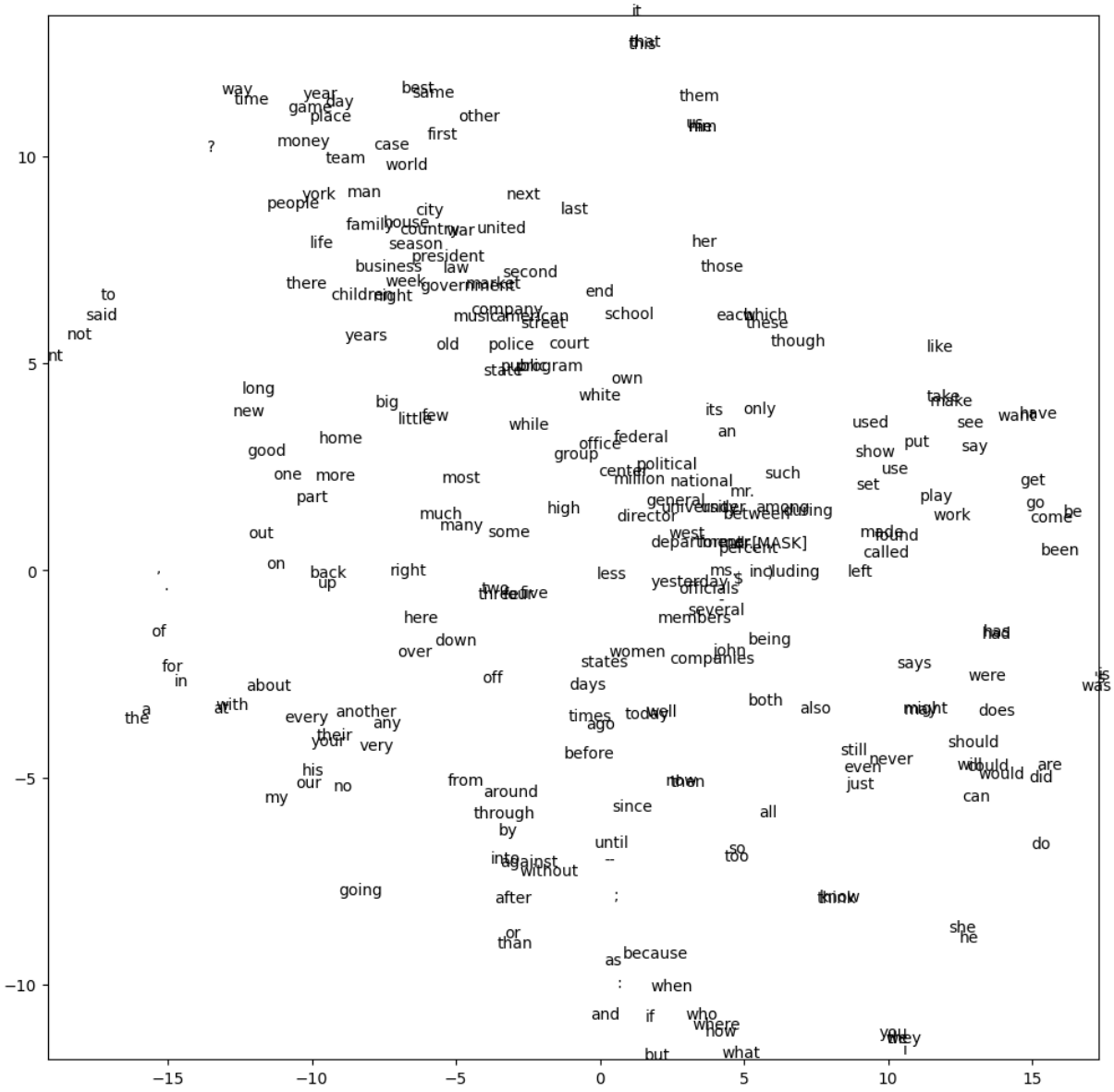
3.1 **Answer**: ****TODO: Write Part 3.1 answer here****

In [72]:
```
from sklearn.manifold import TSNE

def tsne_plot_representation(model):
    """Plot a 2-D visualization of the learned representations using t-SNE."
```

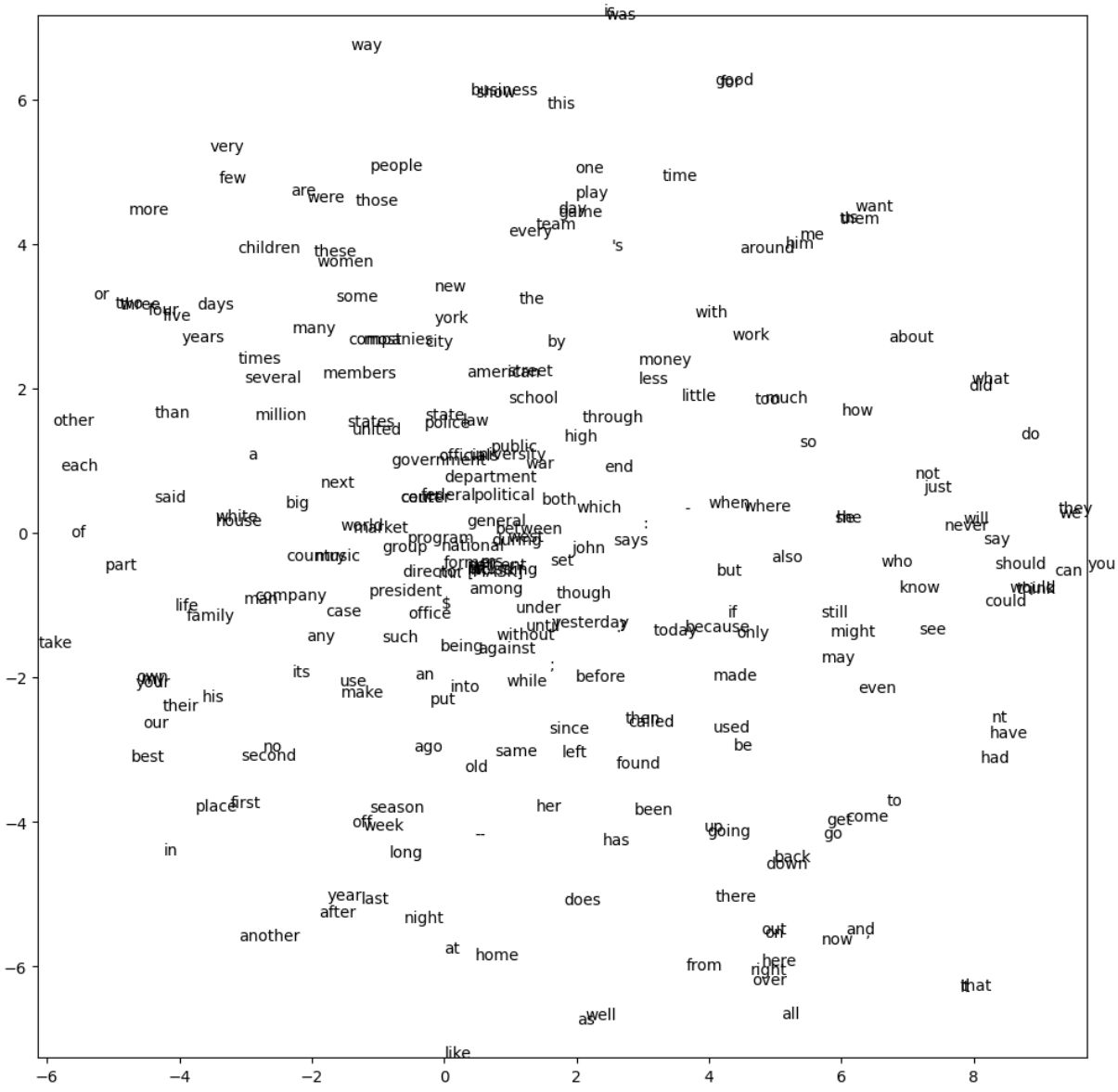
```
    print(model.params.word_embedding_weights.shape)
    mapped_X = TSNE(n_components=2).fit_transform(model.params.word_embeddin
    pylab.figure(figsize=(12,12))
    for i, w in enumerate(model.vocab):
        pylab.text(mapped_X[i, 0], mapped_X[i, 1], w)
    pylab.xlim(mapped_X[:, 0].min(), mapped_X[:, 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 [73]:
```
tsne_plot_representation(trained_model)
```

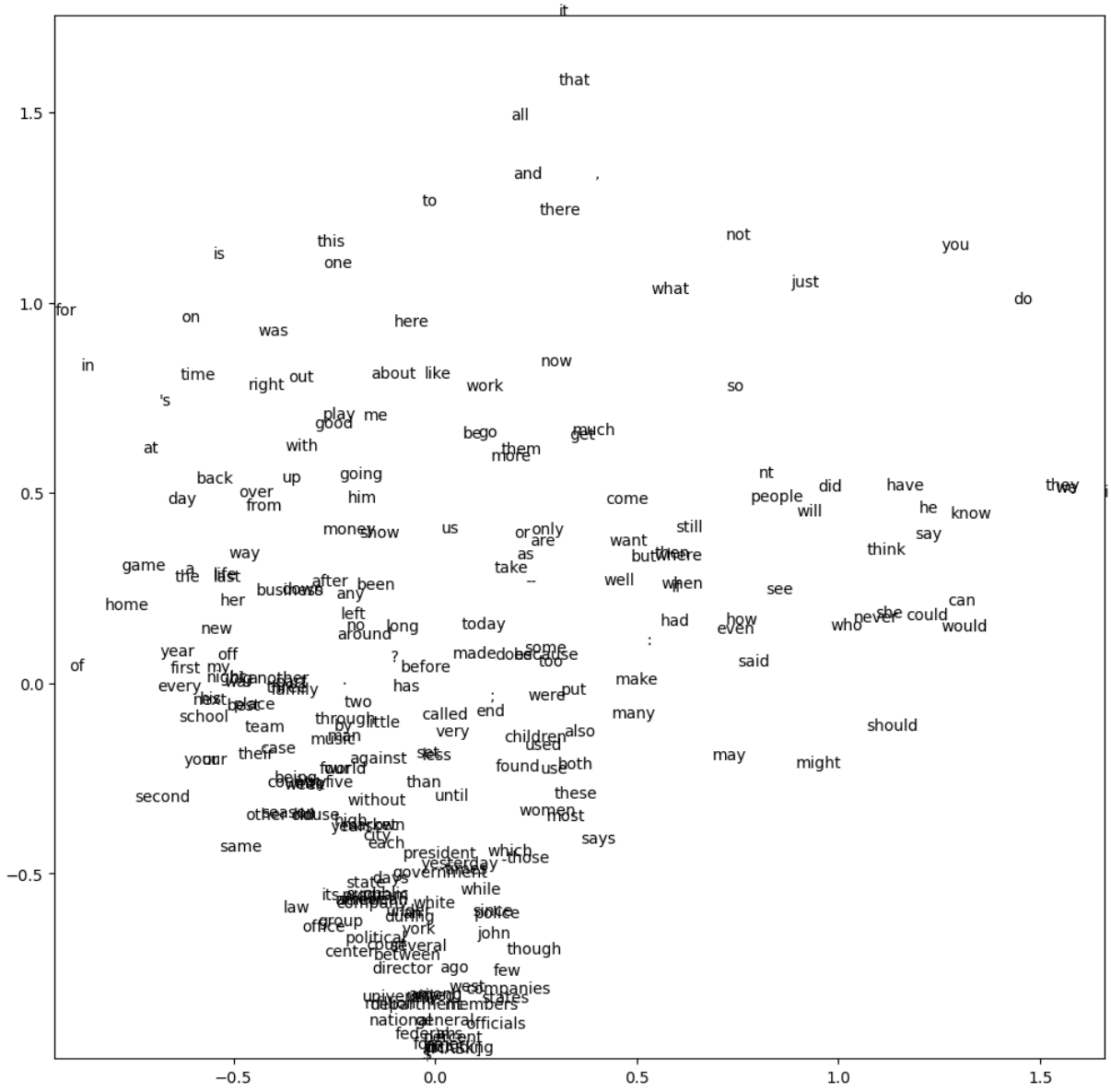
```
(251, 16)
```



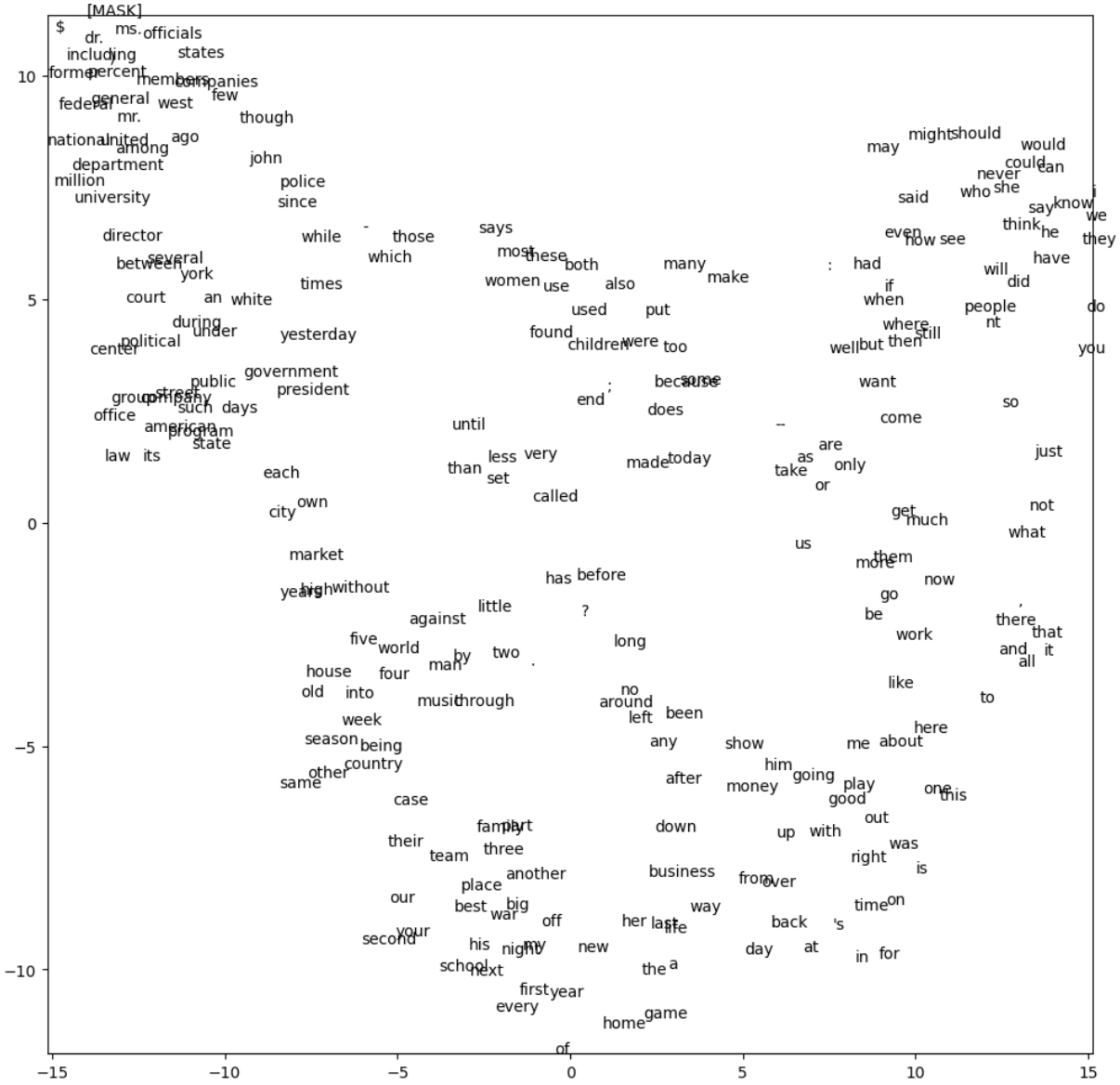
In [74]:
```
tsne_plot_GLoVE_representation(W_final, b_final)
```



In [75]:
```
plot_2d_GLoVE_representation(W_final_2d, b_final_2d)
```



In [76]:
```
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 [ ]:

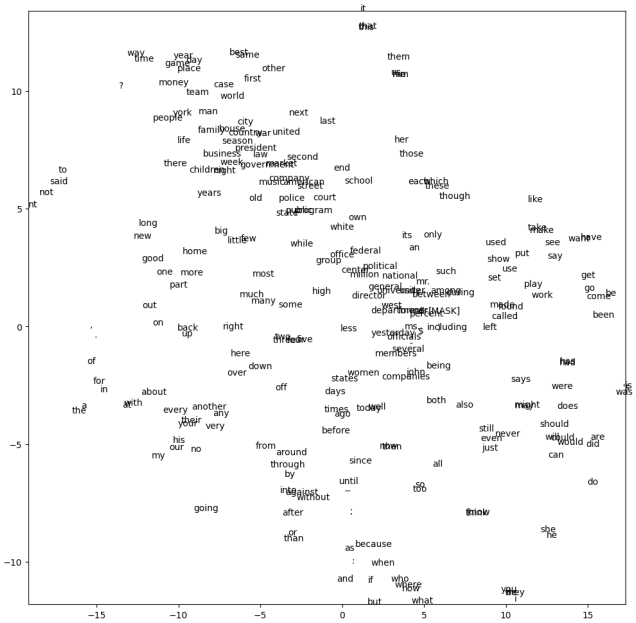


Figure 3: tsne for trained model

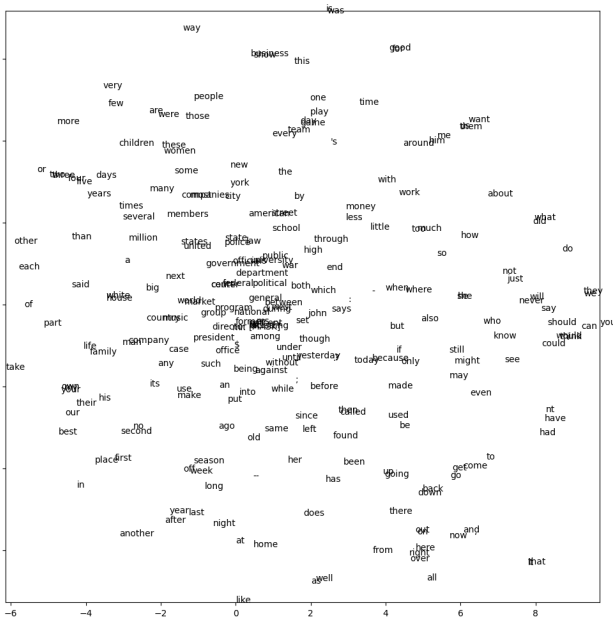


Figure 4: Training Loss vs. Embedding Dimension

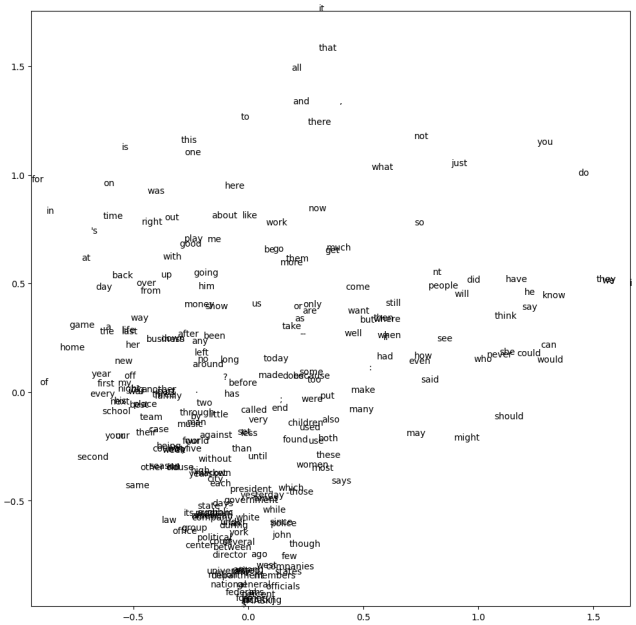


Figure 5: Glove 2d representation

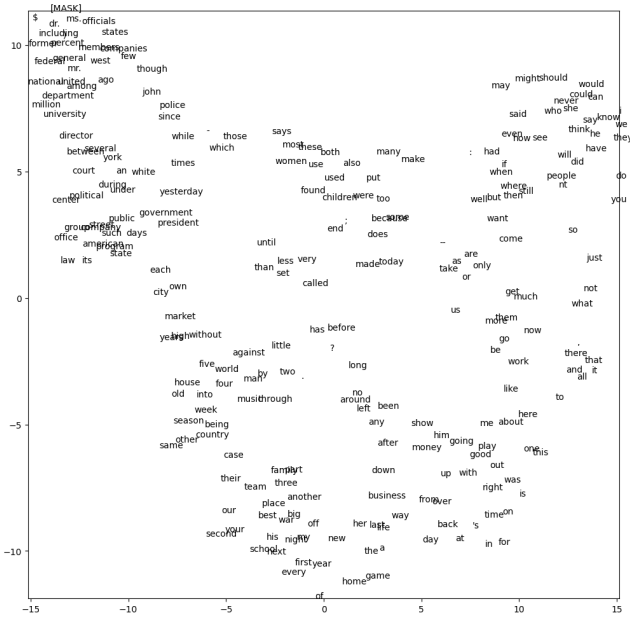


Figure 6: Glove Tsnen