Programming Assignment 1: Learning Distributed Word Representations

Version: 1.0
Version Release Date: 2021-01-22
Due Date: Thursday, Feb. 4, at 11:59pm
Based on an assignment by George Dahl

Submission: You must submit two files through MarkUs[1]: (1) a PDF file containing your writeup, titled `a1-writeup.pdf`, which will be an export of the iPython notebook, and (2) your code file `a1-code.ipynb`. There will be sections in the notebook for you to write your responses. Your writeup must be typed. There will be sections in the notebook for you to write your responses. Make sure that the relevant outputs (e.g. `print_gradients()` outputs, plots, etc.) are included and clearly visible.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

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.

The teaching assistants for this assignment are Harris Chan and Summer Tao. Send your email with subject "*[CSC413] PA1 ...*" to csc413-2021-01-tas@cs.toronto.edu or post on Piazza with the tag `pa1`.

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.

[1]`https://markus.teach.cs.toronto.edu/csc413-2021-01/main`
[2]`https://csc413-uoft.github.io/2021/assets/misc/syllabus.pdf`

Starter code and data

The starter code is at `https://colab.research.google.com/github/csc413-uoft/2021/blob/master/assets/assignments/a1-code.ipynb`.

The starter helper function will download the specific the dataset from `http://www.cs.toronto.edu/~jba/a1_data.tar.gz`. Look at the file `raw_sentences.txt`. It 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).

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 within IPython:

```
import pickle
data = pickle.load(open('data.pk', 'rb'))
```

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 $372,500 \times 4$ 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.

Now look at the notebook ipynb file `a1-code.ipynb`, which contains the starter code for the assignment. Even though you only have to modify a few specific locations in the code, you may want to read through this code before starting the assignment.

1 Linear Embedding – GLoVe (2pts)

In this section we will be implementing a simplified version of GLoVe [Jeffrey Pennington and Manning]. Given a corpus with V distinct words, we define the co-occurrence matrix $X \in V \times V$ with entries X_{ij} representing the frequency of the i-th word and j-th word in the corpus appearing in the same *context* - in our case the adjacent words. The co-occurrence matrix can be *symmetric* (*i.e.*, $X_{ij} = X_{ji}$) if the order of the words do not matter, or *asymmetric* (*i.e.*, $X_{ij} \neq X_{ji}$) if we wish to distinguish the counts for when i-th word appears before j-th word. GLoVe aims to find a d-dimensional embedding of the words that preserves properties of the co-occurrence matrix by representing the i-th word with two d-dimensional vectors $\mathbf{w}_i, \tilde{\mathbf{w}}_i \in \mathbb{R}^d$, as well as two scalar biases

$b_i, \tilde{b}_i \in \mathbb{R}$. This objective can be written as [3]:

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

When the bias terms are omitted and we tie the two embedding vectors $\mathbf{w}_i = \tilde{\mathbf{w}}_i$, then GLoVe corresponds to finding a rank-d symmetric factorization of the co-occurrence matrix.

1.1 GLoVE Parameter Count [0pt]

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

1.2 Expression for gradient $\frac{\partial L}{\partial \mathbf{w}_i}$ [1pt]

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

1.3 Implement the gradient update of GLoVE [1pt]

Implement the gradient update of GLoVe in `a1-code.ipynb`. Look for the `## YOUR CODE HERE ##` comment for where to complete the code.

1.4 Effects of embedding dimension [0pt]

Train the both the symmetric and asymmetric GLoVe model with varying dimensionality d. Comment on the results:

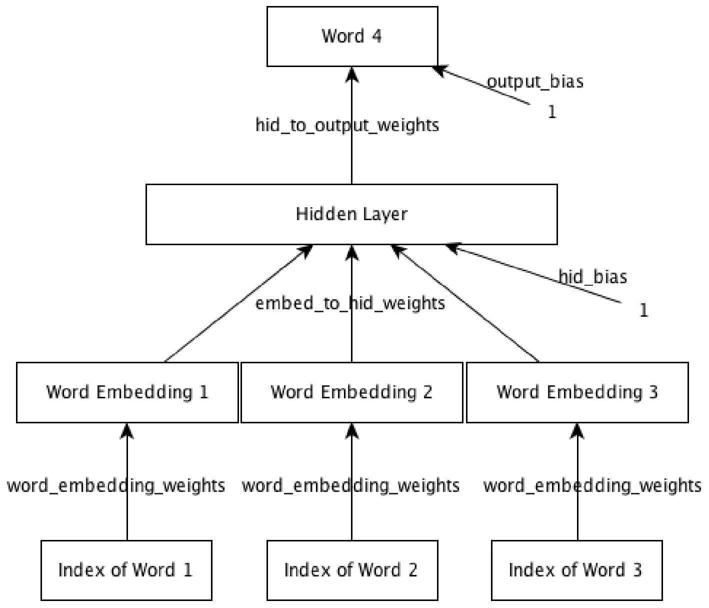
1. Which d leads to optimal validation performance for the asymmetric and symmetric models?
2. Why does / doesn't larger d always lead to better validation error?
3. Which model is performing better, and why?

[3]We have simplified the objective by omitting the weighting function. For the complete algorithm please see [Jeffrey Pennington and Manning]

2 Network architecture (2pts)

In this assignment, we will train a neural language model like the one we covered in lecture and as in Bengio et al. [2003]. It receives as input 3 consecutive words, and its aim is to predict a distribution over the next word (the *target* word). We train the model using the cross-entropy criterion, which is equivalent to maximizing the probability it assigns to the targets in the training set. Hopefully it will also learn to make sensible predictions for sequences it hasn't seen before.

The model architecture is as follows:



The network consists of an input layer, embedding layer, hidden layer and output layer. The input consists of a sequence of 3 consecutive words, given as integer valued indices. (*e.g.*, the 250 words in our dictionary are arbitrarily assigned integer values from 0 to 249.) The embedding layer maps each word to its corresponding vector representation. The layer after embedding layer has $3 \times D$ units, where D is the embedding dimension of a single word. The embedding layer is connected to the hidden layer, which uses a logistic nonlinearity. The hidden layer in turn is connected to the output layer. The output layer is a softmax over the V words, where V is the number of words in the dictionary.

2.1 Number of parameters in neural network model [1pt]

As above, assume in general that we have V words in the dictionary and use the previous N words as inputs. Suppose we use a D-dimensional word embedding and a hidden layer with H hidden units. The trainable parameters of the model consist of 3 weight matrices and 2 sets of biases. What is the total number of trainable parameters in the model, as a function of V, N, D, H? Which part of the model has the largest number of trainable parameters?

2.2 Number of parameters in n-gram mode [1pt]

Another method for predicting the next words is an *n-gram model*, which was mentioned in Lecture 3. If we wanted to use an n-gram model with the same context length N as our network, we'd need to store the counts of all possible $(N + 1)$-grams. If we stored all the counts explicitly, how many entries would this table have?

2.3 Comparing neural network and n-gram model scaling [0pt]

How do the parameters in the neural network model scale with the number of context words N versus how the number of entries in the n-gram model scale with N?

3 Training the Neural Network (3pts)

We will modify the architecture slightly from the previous section, inspired by BERT [Devlin et al., 2018]. 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 1 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[4]. Only the output word positions that were masked in the input are included in the cross entropy loss calculation:

$$C = -\sum_{i}^{B}\sum_{n}^{N}\sum_{j}^{V} m_n^{(i)}(t_{n,j}^{(i)} \log y_{n,j}^{(i)}), \tag{1}$$

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,

[4]For simplicity we also include the `[MASK]` token as one of the possible prediction even though we know the target should not be this token

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.

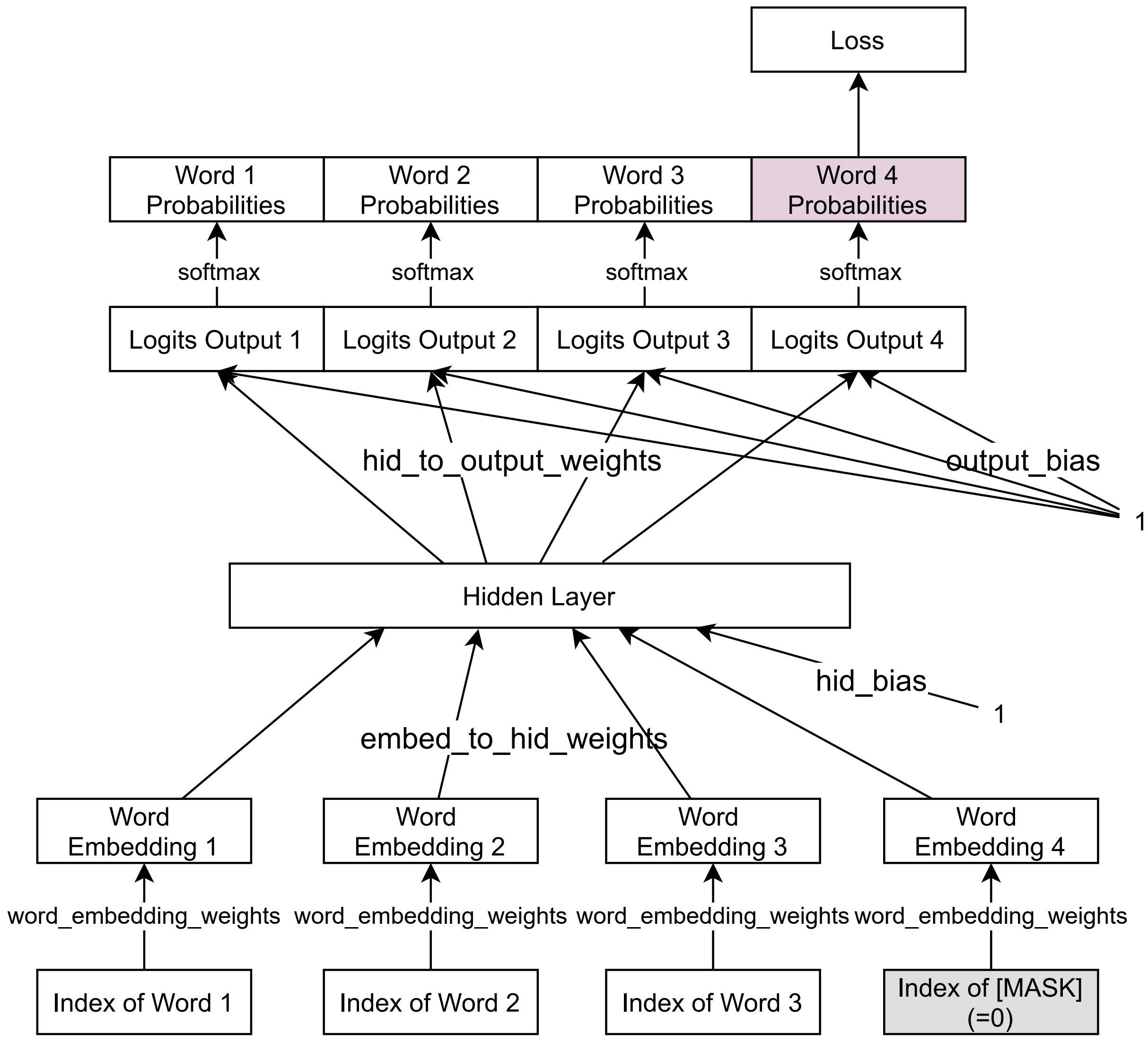


Figure 1: Modified architecture with N words output. During training, we mask out one of the input words by replacing it with a `[MASK]` token, and try to predict the masked out word in the corresponding position in the output. Only that output position is used in the cross entropy loss.

In this part of the assignment, you will 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.

3.1 Implement gradient with respect to output layer inputs [1pt]

`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:

$$y_{n,j} = \frac{e^{z_{n,j}}}{\sum_l e^{z_{n,l}}} \tag{2}$$

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.

3.2 Implement gradient with respect to parameters [1pt]

`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.

3.3 Print the gradients [1pt]

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.

3.4 Run model training [0 pt]

Once you've implemented the gradient computation, you'll need to train the model. The function `train` in `a1-code.ipynb` 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

For example, execute the following:

```
model = train(16, 128)
```

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.

4 Arithmetics and Analysis (2pts)

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.

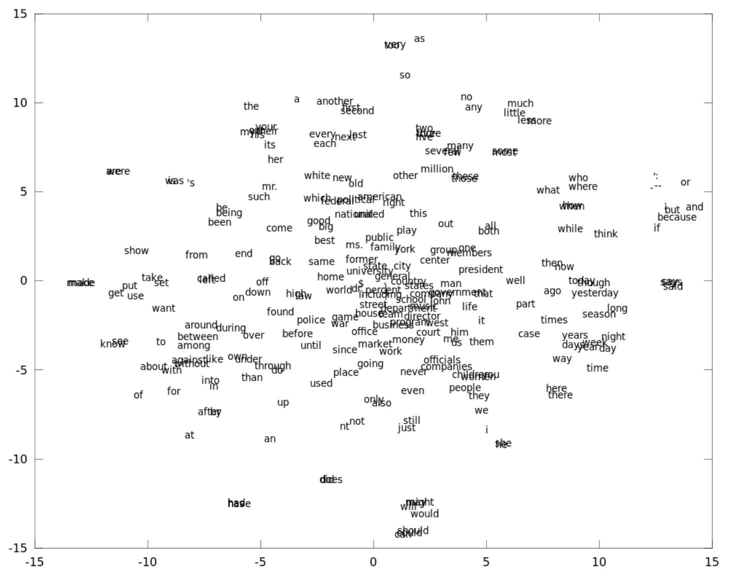
4.1 t-SNE [1pt]

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

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

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

- `tsne_plot_representation` creates a 2-dimensional embedding of the distributed representation space using an algorithm called t-SNE [Maaten and Hinton, 2008]. You don't need to know what this is for the assignment, but we may cover it later in the course. Nearby points in this 2-D space are meant to correspond to nearby points in the 16-D space. From the learned model, you can create pictures that look like this:



- `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

Using these methods, please answer the following question.

1. Plot the 2-dimensional visualization for the trained model from part 3 using the method `tsne_plot_representation`. Look at the plot and find a few clusters of related words. What do the words in each cluster have in common? 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 sentence for each question and include the plots in your answer. [1pt]

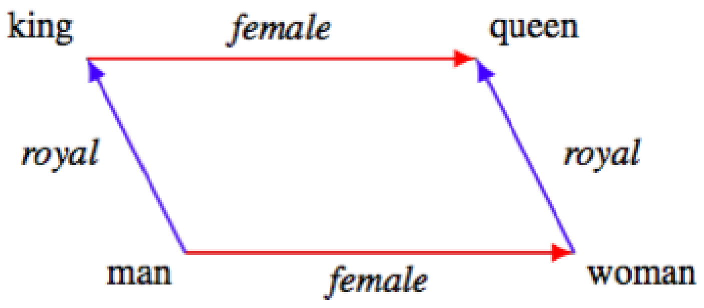


Figure 2: We approximately obtain the vector for *queen* by displacing the vector for *woman* by the difference of vector from *man* to *king*. Figure reproduced from [Ethayarajh et al., 2018]

4.2 Word Analogy Arithmetic [1pt]

A word analogy f is an invertible transformation that holds over a set of ordered pairs S iff $\forall(x, y) \in s, f(x) = y \wedge f^{-1}(y) = x$. When f is of the form $x \to x + r$, it is a linear word analogy.

Arithmetic operators can be applied to vectors generated by language models. There is a famous example: $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{women}} \approx \overrightarrow{\text{queen}}$. As shown in Figure 2, these linear word analogies form a parallelogram structure in the vector space [Ethayarajh et al., 2018].

In this section, we will explore a property of *linear word analogies*. A linear word analogy holds exactly over a set of ordered word pairs S iff $\| \overrightarrow{\text{x}} - \overrightarrow{\text{y}} \|^2$ is the same for every word pair, $\| \overrightarrow{\text{a}} - \overrightarrow{\text{x}} \|^2 = \| \overrightarrow{\text{b}} - \overrightarrow{\text{y}} \|^2$ for any two word pairs, and the vectors of all words in S are coplanar.

We will use the embeddings from the symmetric, asymmetrical GloVe model, and the neural network model from part 3 to perform arithmetics. The method to perform the arithmetic and retrieve the closest word embeddings is provided in the notebook using the method `find_word_analogy`. Please answer the following questions:

4.2.1 Specific example [1pt]

Perform arithmetic on words *he*, *him*, *her*, using: (1) symmetric, (2) averaging asymmetrical GloVe embedding, (3) concatenating asymmetrical GloVe embedding, and (4) neural network word embedding from part 3. That is, we are trying to find the closet word embedding vector to the vector $emb(he) - emb(him) + emb(her)$. For each sets of embeddings, you should list out: (1) what the closest word that is not one of those three words, and (2) the distance to that closest word. Is the closest word *she*? Compare the results with the tSNE plots.

4.2.2 Finding another Quadruplet [0pt]

Pick another quadruplet from the vocabulary which displays the parallelogram property (and also makes sense sementically) and repeat the above proceduces. Compare and comment on the results from arithmetic and tSNE plots.

What you have to submit

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

- A PDF file titled `a1-writeup.pdf` containing the following:
  - Part 1: Questions 1.1, 1.2, 1.3, 1.4. Completed code for `grad_GLoVE()` function.
  - Part 2: Questions 2.1, 2.2, 2.3.
  - Part 3: Completed code for `compute_loss_derivative()` (3.1), `back_propagate()` (3.2) functions, and the output of `print_gradients()` (3.3)
  - Part 4: Questions 4.1, 4.2.1, 4.2.2
- Your code file `a1-code.ipynb`

References

Richard Socher Jeffrey Pennington and Christopher D Manning. Glove: Global vectors for word representation. Citeseer.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

Kawin Ethayarajh, David Duvenaud, and Graeme Hirst. Towards understanding linear word analogies. *arXiv preprint arXiv:1810.04882*, 2018.