1. Linear Embedding - GLoVe

1.1. GLoVE Parameter Count

Since the dimension of $\boldsymbol{w}_i$ and $\boldsymbol{w}_i$ are d and the dimension of the biases are 1, there are $2d+2$ parameters to train for each vocabulary. Thus, we can get that the GLoVe model have $2V(d+1)$ trainable parameters.

1.2. Expression for gradient $\frac{\partial L}{\partial \boldsymbol{w}_i}$

According to the given loss function, we can get $\frac{\partial L}{\partial \boldsymbol{w}_i}$ should be

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{w}_i} &= \frac{\partial}{\partial \boldsymbol{w}_i} \sum_{i,j=1}^{V} (\boldsymbol{w}_i^T \tilde{\boldsymbol{w}}_i + b_i + \tilde{b}_j - \log X_{ij})^2 \\ &= 2 \sum_{i,j=1}^{V} (\boldsymbol{w}_i^T \tilde{\boldsymbol{w}}_i + b_i + \tilde{b}_j - \log X_{ij}) \tilde{\boldsymbol{w}}_i\end{aligned}$$

1.3. Implement the gradient update of GLoVE

The code for this question is shown below.

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

  if not W_tilde is None and not b_tilde is None:
  ###########################   YOUR CODE HERE   ############################
    loss = (W @ W_tilde.T + b @ np.ones([1,n]) + np.ones([n,1]) @ b_tilde.T - (log_co_occurence))
    grad_W = 2 * (loss @ W_tilde)
    grad_W_tilde = 2 * (loss.T @ W)
    grad_b = 2 * (np.ones([1,n]) @ loss).T
    grad_b_tilde = 2 * (np.ones([1,n]) @ loss).T
  ##########################################################################
  else:
    loss = (W @ W.T + b @ np.ones([1,n]) + np.ones([n,1])@b.T - 0.5*(log_co_occurence + log_co_occurence.T))
    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
```

1.4. Effects of embedding dimension

2. Network architecture

2.1. Number of parameters in neural network model

Let's find the number of the trainable parameters of the word embedding weights first. Since there are V words in the dictionary and the dimension of the word embedding layer is $N \times D$, we can get that there are $V \times D$ trainable parameters.

For weights between the word embedding layer and the hidden layer, since there are H units in the hidden layer, the dimension of the matrix that connects two layers should be $ND \times H$, which is the number of trainable parameters.

Thus, for the biases of the hidden layer, there should be $H \times 1$ trainable parameters.

Similarly, since the output layer consists of V words, there are $V \times H$ trainable parameters for

weights between the hidden layer and the output layer.
Thus, for the biases of the output layer, there should be $V \times 1$ trainable parameters.
Since V is much larger than other variables, we only need to consider the part that depends on V.
Thus, we can get that weights between the hidden layer and the output layer, *hid_to_output_weights*, has the largest number of trainable parameters since $H > D$.

2.2. Number of parameters in n-gram mode
For each gram, we can choose any word from V words. Thus, there are V^N number of combinations for the previous N words. For the prediction, since the output layer is a softmax over the V words, there are V words. Thus, there are V^{N+1} entries in the n-gram model scale with N.

2.3. Comparing neural network and n-gram model scaling

3. Training the Neural Network

3.1. Implement gradient with respect to output layer inputs
The code for this question is shown below.

```
def compute_loss_derivative(self, output_activations, expanded_target_batch, target_mask):
        """Compute the derivative of the multiple target position cross-entropy loss function \n"

            For example:

         [y_{0} ....  y_{V-1}] [y_{V}, ..., y_{2*V-1}] [y_{2*V} ... y_{i,3*V-1}] [y_{3*V} ... y_{i,4*V-1}]

         Where for colum j + n*V,

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

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

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

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

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

        The arguments are as follows:

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

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

        ###########################   YOUR CODE HERE ##############################
        # Loss
        loss = output_activations - expanded_target_batch
        expanded_mask = np.repeat(target_mask, self.vocab_size, axis=1).reshape(loss.shape)
        return np.multiply(expanded_mask, loss)
        # ############################################################################

```

3.2. Implement gradient with respect to parameters

The code for this question is shown below.

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

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

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

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

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

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

        # Embedding layer
        word_embedding_weights_grad = np.zeros((self.vocab_size, self.embedding_dim))
        for w in range(self.context_len):
            word_embedding_weights_grad += np.dot(self.indicator_matrix(input_batch[:, w:w+1], mask_zero_index=False).T,
                                                  embed_deriv[:, w * self.embedding_dim:(w + 1) * self.embedding_dim])
```

```python

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

```

3.3. Print the gradients The output for print_gradients() is shown below.

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

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

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

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

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

```

3.4. Run model training

4. Arithmetics and Analysis

4.1. t-SNE

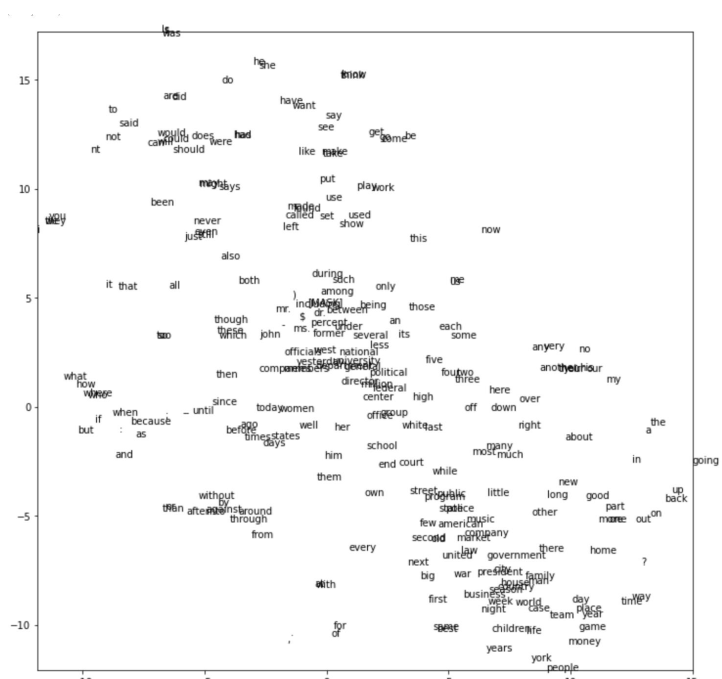

Figure 1: The tsne plot representation using the trained weights.

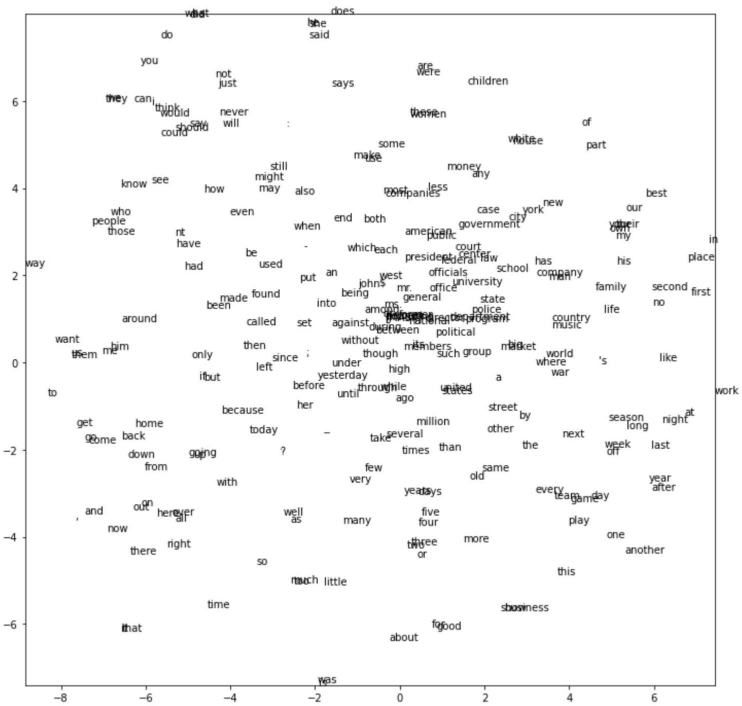

Figure 2: The tsne plot GLoVE representation.

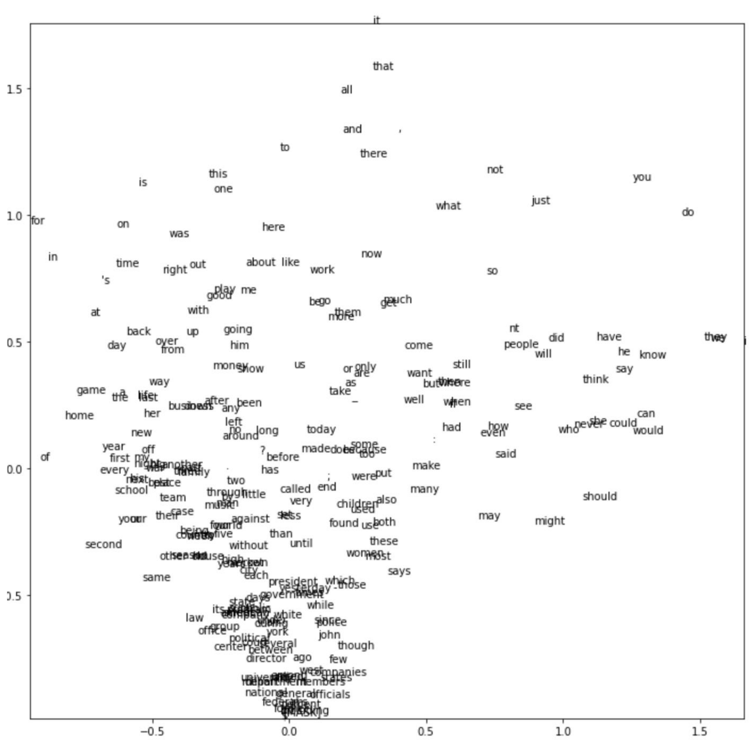

Figure 3: The 2d GLoVe representation.

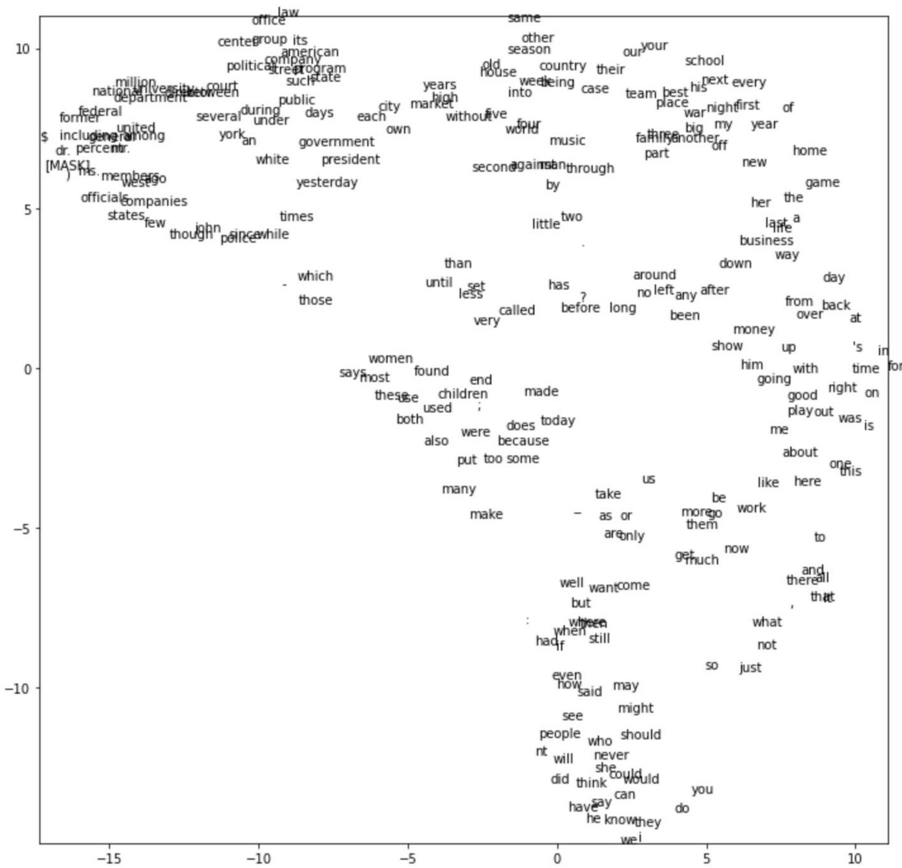

Figure 4: The tsne plot GLoVE representation.

According to Figure 1 (tsne_plot_representation), we can see that words with the similar function in sentences or the structure in terms of the speech gather together. For example, on the top left of the plot, there is a cluster of auxiliary verbs including “would”, “could”, and “should”, and a cluster of question words such as “what”, “how”, and “who” on the middle left of the plot.
For Figure 1 and 2 (tsne_plot_GLoVE_representation), we can see that words in Figure 1 are distributed like a linear on the main diagonal, while words in Figure 2 diverge circularly around the biggest cluster on the lower middle. In addition, we found that positions of different clusters are different in two plots.
Comparing with Figure 1 and 2, we can see that words in Figure 3 (plot_2d_GLoVe_representation) seem to be more clustered and are distributed like a fan shape. Many nouns gather together on the bottom left cluster, which is the largest cluster and words are fan out upward.

4.2. Word Analogy Arithmetic

4.2.1. Specific example

The results are shown below.

```
## GloVe embeddings
The top 10 closest words to emb(he) - emb(him) + emb(her) are:
he: 1.4213098857979793
she: 1.48167433432594
said: 2.1025960106397767
then: 2.2720425987761406
does: 2.301964867719902
says: 2.318047293286045
who: 2.328984314854128
where: 2.334702431567161
did: 2.353623598835888
```

```
should: 2.4126428205989865

# Concatenation of W_final_asym, W_tilde_final_asym
The top 10 closest words to emb(he) - emb(him) + emb(her) are:
he: 2.046826000951795
she: 2.3455038844018743
i: 3.0624522787351487
we: 3.2848647174761094
they: 3.390910580609287
you: 4.568945007203308
john: 4.805241000654006
program: 5.084420284826234
president: 5.104152796566877
never: 5.111163924178705

# Averaging asymmetric GLoVE vectors
The top 10 closest words to emb(he) - emb(him) + emb(her) are:
he: 1.0154702232698416
she: 1.0744126028585648
should: 1.6078139338035942
could: 1.6799073061855805
i: 1.693953840244398
would: 1.700020962168106
did: 1.766206937557185
can: 1.7744377144463797
might: 1.7765376616413824
will: 1.7931360498829227

## Neural Netework Word Embeddings
The top 10 closest words to emb(he) - emb(him) + emb(her) are:
he: 2.4284684644619032
she: 17.4415802699889
have: 25.921497697983263
they: 25.981587972296392
want: 26.437644546989542
we: 27.128094534488834
i: 27.215833550319473
but: 28.03028938337095
about: 28.163403568035555
this: 28.531350495330678

```

According to the outputs, we can see that the closest word that is not “he”, “him”, or “her” is “she” for all 4 different arithmetic. The corresponding distances are shown above. According to four plots, they all show the parallelogram property of the quadruplets approximately. Figure 5, 6, 7, and 8 show how they present in the corresponding plot.

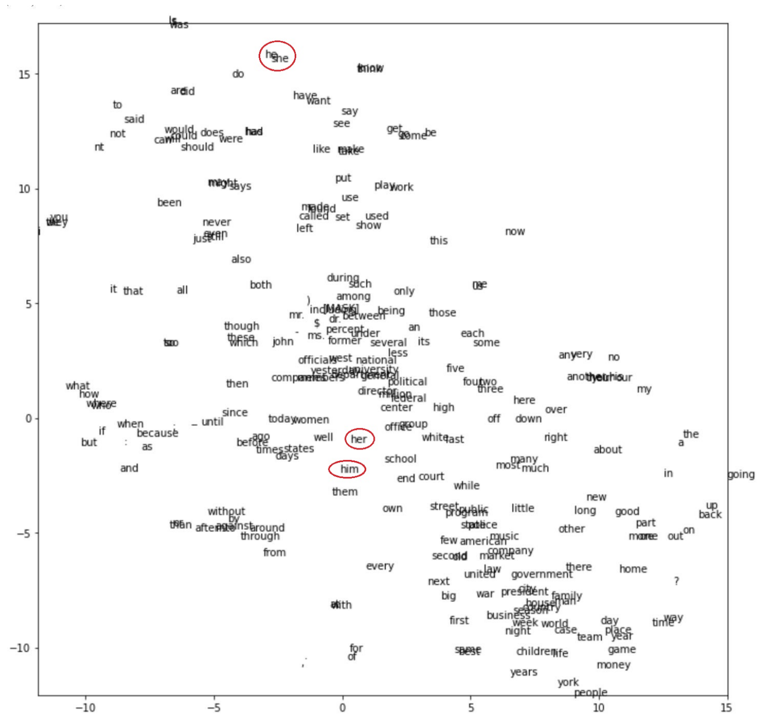

Figure 5: The tsne plot representation using the trained weights.

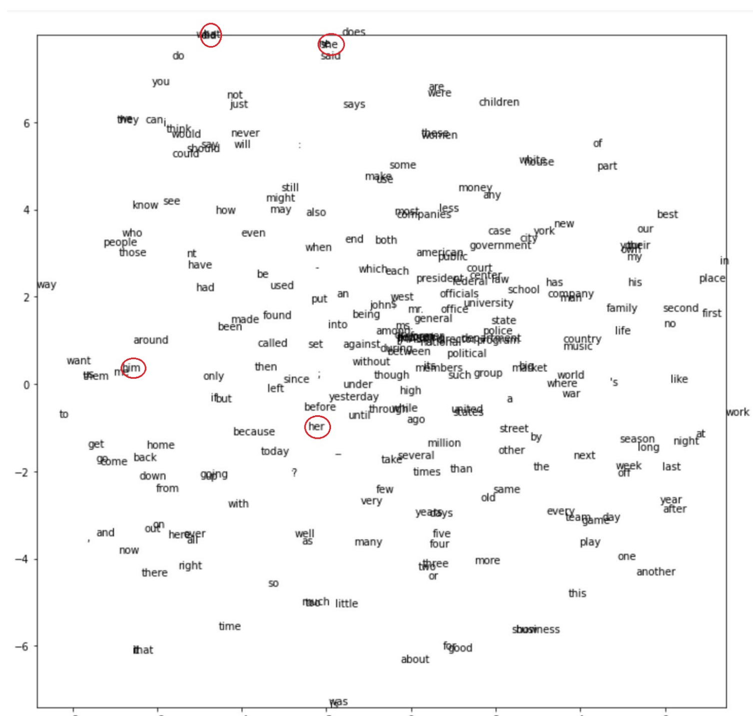

Figure 6: The tsne plot GLoVE representation.

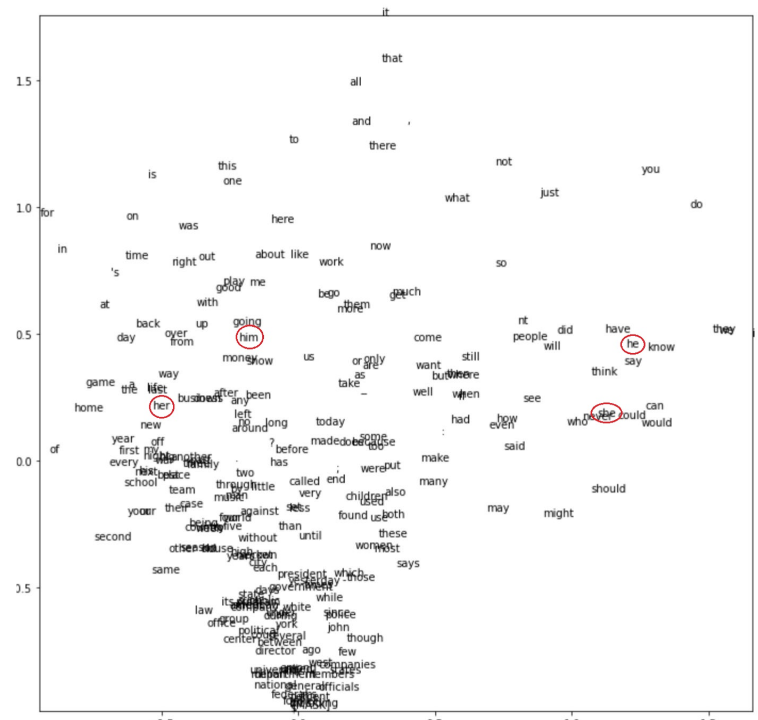

Figure 7: The 2d GLoVe representation.

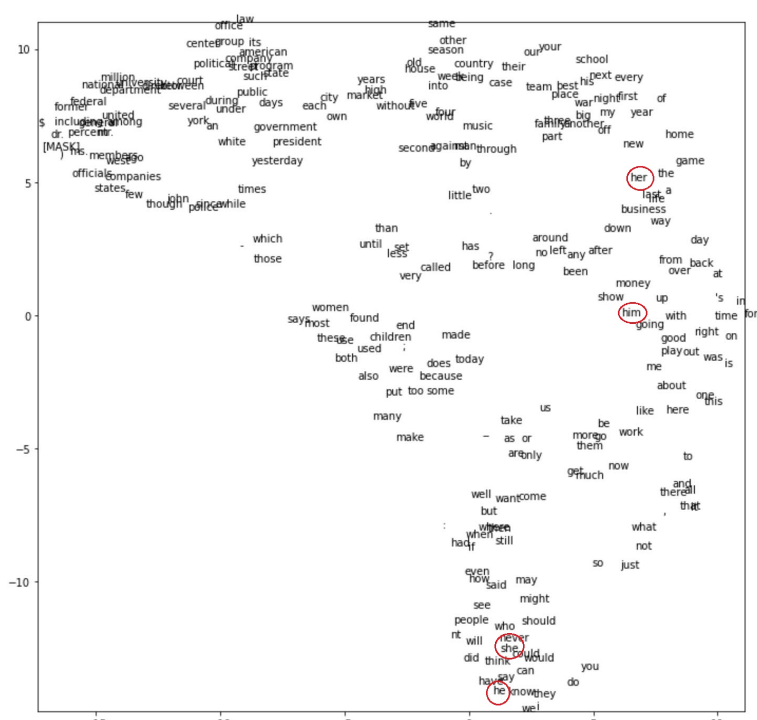

Figure 8: The tsne plot GLoVE representation.