CAP 6619
Deep Learning
Assignment 5

Sai Supraja Chinthapalli
Z23760554

1.Explain why temporal data is not I.I.D. (independent and identical distribution) using example of text data.

Temporal text data, unlike what the I.I.D. assumption requires, is not independent and identically distributed. The reason is because:

<u>I.I.D. Breakdown</u>

Independent: I.I.D. assumes data points are independent, meaning knowing one data point tells you nothing about another. However, in temporal text data, the meaning of a sentence or word can depend on what came before it. For example, in the sentence "It was raining today, so I brought an umbrella," knowing it's raining today makes it more likely the next sentence talks about an umbrella.

Identically Distributed: I.I.D. also assumes all data points come from the same probability distribution. Text data often changes over time. For instance, writing styles may evolve, slang terms become popular, or news events influence vocabulary choices. These factors would cause the distribution of words used to differ across different time periods in the data.

Text Example:

Analyzing news articles: An article from 2010 might be more likely to mention "dial-up internet" compared to an article from 2023, which might focus on "high-speed internet." This shows how the distribution of words changes with time.

Furthermore, news articles about a developing event, like an election, will likely show dependence. An article early in the race might use words like "candidate" and "primary," while later articles might use "winner" and "administration." The meaning and likelihood of certain words depend on the context built by previous articles.

Conclusion:

Temporal text data exhibits dependencies between data points due to the evolving nature of language and the sequential flow of information. This violates the core assumptions of independence and identical distribution in I.I.D.

2.Given the same input data, RNN model can generate different outputs at each time step. Explain why this is possible using the output formula of RNN.

RNNs (Recurrent Neural Networks) can produce different outputs at each time step for the same input data due to their internal hidden state and the concept of backpropagation through time (BPTT).The output formula contributes as follows:

The Formula:

While there are various RNN architectures, a basic RNN's output at a specific time step (t) can be expressed as:

h(t) = f(W_h * h(t-1) + W_x * x(t) + b)

- h(t): The hidden state vector at time step t (often considered the output)
- f: An activation function (e.g., tanh, ReLU)
- W_h: Weight matrix for hidden layer connections
- h(t-1): The hidden state vector from the previous time step (t-1)
- W_x: Weight matrix for input connections
- x(t): The input vector at time step t
- b: Bias vector

Understanding the Difference:

1. **Hidden State Influence:** The hidden state (h(t-1)) acts as a form of memory. It carries information about previous inputs and influences the current output (h(t)). This information can vary slightly during training due to weight updates, leading to subtly different outputs even with the same input sequence (x(t)).
2. **Activation Function:** The activation function (f) introduces non-linearity. Small variations in the weighted sum (W_h * h(t-1) + W_x * x(t) + b) can be amplified differently by the activation function, leading to slightly different outputs at each step.
3. **Backpropagation Through Time (BPTT):** During training, BPTT unfolds the RNN across time steps. Errors are propagated backwards, adjusting the weights based on the entire sequence. These adjustments can be slightly different for each sequence due to weight initialization and the specific input data, subtly influencing future outputs for the same input sequence.

Even though the input data (x(t)) remains the same, the interplay between the hidden state, activation function, and weight updates from BPTT can cause the RNN to generate slightly different outputs at each time step.

The architecture of the RNN (e.g., number of layers, hidden units) and the choice of activation function can also impact the variability in outputs. In practice, the differences might be very small, but they can accumulate over longer sequences and become more noticeable.

Overall, the RNN's output formula allows for a dynamic interplay between past information, non-linear activation, and weight adjustments, leading to the possibility of different outputs even with the same input data.

3.Can we use bidirectional LSTM in the chatbot application? Explain the reason in detail.

Bidirectional LSTMs (BLSTMs) are a great fit for chatbot applications, and the reason is:

Understanding Chatbot Needs:

Chatbots need to understand the context of a conversation to respond appropriately. This context can be spread across multiple user sentences. For example, if a user says "I'm looking for a good Italian restaurant," and then follows up with "What about delivery options?", the chatbot's response to the second question should consider the first one.

Limitations of Standard LSTMs:

Standard LSTMs process information sequentially, meaning they can only access information from previous time steps. This limits their ability to capture context spread across multiple user inputs.

BLSTM to the Rescue:

BLSTMs addresses this limitation by combining two LSTMs:

1. **Forward LSTM:** Processes the input sequence in the normal forward direction, capturing dependencies from the beginning of the sentence/conversation.
2. **Backward LSTM:** Processes the same sequence in reverse order, capturing dependencies from the end.

The outputs from both LSTMs are then concatenated or fed through another layer to create the final output.

Benefits of BLSTMs in Chatbots:

- **Improved Context Understanding:** By considering information from both directions, BLSTMs can better grasp the overall context of a conversation, leading to more relevant and coherent responses.
- **Long-Term Dependency Learning:** BLSTMs can capture long-term dependencies between words or phrases scattered across the conversation, even if they are far apart. This is crucial for complex conversations or those requiring referencing previous statements.
- **Disambiguation:** BLSTMs can help disambiguate the meaning of words based on the surrounding context. For example, "book" can refer to reserving something or a physical book. A BLSTM can use the surrounding words to determine the intended meaning.

Overall, BLSTMs provide a significant advantage in chatbot applications by enabling them to:

- Understand the complete context of a conversation.
- Capture long-term dependencies.
- Disambiguate word meanings.

These capabilities lead to more natural, engaging, and informative chatbot interactions.

- BLSTMs are computationally more expensive compared to standard LSTMs, but their benefits often outweigh the cost.
- BLSTMs are often combined with attention mechanisms in chatbots to further focus on the most relevant parts of the conversation history.

By leveraging BLSTMs, chatbot developers can create more intelligent and interactive chatbots that can effectively understand and respond to user queries.

4.In p. 10 of RNN slides, compute the error value (using cross-entropy) in the first time step (when input is “h”).

The error value (using cross-entropy) in the first time step for the training sequence "hello" when the input is "h." is given as:

- The RNN uses a softmax output layer for character prediction.
- The target label for the first time step is the next character in the sequence, which is "e."
- The vocabulary includes all characters in "hello" and potentially others.

General Approach:

1. **One-Hot Encoding:**
   - Encode "h" as a one-hot vector with a 1 at the corresponding index (depending on the vocabulary size) and 0s elsewhere.
2. **Forward Pass (Conceptual):**
   - The one-hot encoded vector for "h" is multiplied by the input weight matrix (W_x).
   - If the architecture uses a hidden state (h(t-1)) for the first time step, it's multiplied by the hidden weight matrix (W_h) and added to the previous result.
   - An optional bias vector (b) might be added.
   - The combined result goes through a non-linear activation function (e.g., tanh or ReLU).
   - The output of the activation function is then fed into the softmax layer.
3. **Softmax Output (Hypothetical):**
   - The softmax layer normalizes the activation values into a probability distribution across all characters in the vocabulary, summing to 1.
   - The element in the softmax output vector corresponding to "e" represents the predicted probability of "e" appearing after "h."
4. **Cross-Entropy Error Calculation:**
   - Using the formula cross_entropy = - (y_true * log(y_pred)):
     - y_true is the element in the one-hot encoded vector for "e" (1 at its corresponding index).
     - y_pred is the predicted probability of "e" from the softmax output vector (unknown in this case).

    - The resulting value would be negative (due to the log term) and closer to 0 if the predicted probability for "e" is high (meaning the model is confident about predicting "e" after "h").

So the output is:

- Assume the softmax output is: [0.2, 0.7, 0.05, 0.02, 0.01].
- "e" is likely at index 1 (depending on vocabulary order), so y_pred is 0.7.
- The cross-entropy error (assuming a one-hot encoded vector with 1 at index 1 for "e") would be:
- cross_entropy = - (1 * log(0.7)) ≈ -0.357

Therefore the cross entropy is -0.357

5.In LSTM, the activation functions of gates (forget, input, and output) are sigmoid functions. Explain what will happen if we use ReLU instead.

If we use ReLU (Rectified Linear Unit) instead of sigmoid functions for the activation functions in LSTM gates (forget, input, and output) then the reasons are as follows:

Potential Benefits:

- **Faster Training:** ReLU is computationally simpler than sigmoid, leading to faster training times.
- **Reduced Vanishing Gradients:** Sigmoid functions can suffer from vanishing gradients in deep networks. ReLU, in theory, can help alleviate this issue because gradients are non-zero for positive values.

Potential Drawbacks:

- **Dead Neurons:** ReLU neurons output zero for negative inputs. In LSTMs, this can lead to "dead neurons" where the gate never allows information to pass through. This can cripple the LSTM's ability to learn long-term dependencies.
- **Unstable Gradients:** While ReLU avoids vanishing gradients, it can introduce exploding gradients for large positive values. This can make training unstable.

Additional Considerations:

- **Variants of ReLU:** Leaky ReLU, which allows a small positive gradient for negative inputs, can be a good compromise to address the dead neuron problem.
- **Not a Guaranteed Improvement:** While ReLU offers potential benefits, it doesn't always outperform sigmoid in LSTMs. The optimal choice depends on the specific task and dataset.

Overall, Using ReLU in LSTM gates can be a double-edged sword. While it offers advantages in training speed and potentially vanishing gradients, it introduces risks of dead neurons and unstable gradients.

6. What are the problems with using vocabulary indexing in text processing. Explain it using the example given in the slides.

From the example in the slides it perfectly illustrates the problems with vocabulary indexing in text processing:

1. Loss of Semantics: Vocabulary indexing treats words as isolated entities, assigning them unique indices. This approach ignores the richness of natural language, where meaning arises from the interplay of words, their order, and the overall structure of the sentence.

- In the sentence "It is a beautiful day," the order and combination of words convey the meaning of a pleasant day. Vocabulary indexing simply converts each word to its corresponding number, losing the relationship between "beautiful" and "day."

2. Out-of-Vocabulary (OOV) Words: New words or variations (slang, technical terms) that aren't included in the predefined vocabulary can't be processed. This becomes a hurdle when dealing with large and diverse datasets.

- Imagine the sentence "The weather is warmer today." Vocabulary indexing wouldn't be able to represent "weather" or "warmer" because they weren't included in the defined vocabulary. This becomes a limitation when dealing with large datasets or unexpected terms.

3. Polysemy: Many words have multiple meanings depending on context. Vocabulary indexing assigns a single index, failing to capture these nuances.

- Words like "is" have various meanings depending on context. In this example, "is" acts as a linking verb. Vocabulary indexing assigns a single index (0) to "is," regardless of its role in the sentence.

4. Limited Representation of Sentence Structure: Vocabulary indexing only captures the order of individual words, neglecting grammatical structure. Tasks like sentiment analysis or machine translation often rely on understanding sentence structure, which vocabulary indexing struggles with.

- The sequence (7, 0, 2, 10, 8) doesn't capture any information about the grammatical structure of the sentence (subject-verb-object). This makes it difficult for tasks like sentiment analysis or machine translation that rely on understanding sentence structure.

In essence, vocabulary indexing offers a simplistic view of text, neglecting the richness of natural language.

Some alternative approaches that address these limitations are:

- **Word Embeddings:** These represent words as vectors in a high-dimensional space. The positions in this space capture semantic relationships between words. Words with similar meanings will have closer vectors.
- **Part-of-Speech (POS) Tagging:** This assigns grammatical tags (noun, verb, adjective) to each word, providing information about the sentence structure.
- **N-grams:** Sequences of n consecutive words (bigrams, trigrams) can be used to capture some context beyond single words.

These techniques offer a more nuanced representation of text, allowing for more sophisticated text processing tasks.

7. In TF-IDF,
(i) when a 100-word document contains the term "cat" 12 times, compute TF value of 'cat'.

In TF-IDF (Term Frequency-Inverse Document Frequency), TF refers to Term Frequency, which calculates how often a particular term appears in a specific document.

The TF value of 'cat' in the given scenario is given as follows:

1. **Number of occurrences:** We know the term "cat" appears 12 times in the document.
2. **Document length:** The document length is 100 words.
3. **TF calculation:** TF (cat) = (Number of times "cat" appears) / (Total words in the document)

   TF (cat) = 12 words / 100 words

   TF (cat) = 0.12

Therefore, the TF value of 'cat' in this 100-word document is 0.12. This indicates that "cat" appears in 12% of the words in the document.

(ii) The size of the corpus is 10 million documents. If we assume there are 0.3 million documents that contain the term "cat". Compute IDF (use the log form)

Given :

- Total number of documents in the corpus (N): 10,000,000
- Number of documents containing the term "cat" (df(t)): 300,000

We need to compute the IDF (Inverse Document Frequency) using the formula:

IDF(t) = log(N / df(t))

where:

- t - term we're calculating IDF for ("cat" in this case)

By solving this we get:

1. **Plug in the values:** IDF(cat) = log(10,000,000 / 300,000)
2. **Evaluate the logarithm (assuming base-10):** IDF(cat) = log10(100/3)

Therefore, the IDF value of "cat" in this corpus is log10(100/3).

This result indicates that "cat" is a relatively common term in the corpus, as its IDF value is lower compared to terms that appear in fewer documents.

(iii) Compute TF-IDF of 'cat'

As we have both TF and IDF values for "cat," the TF-IDF value can be given as :

TF-IDF(cat) = TF(cat) * IDF(cat)

We obtained the TF value in the previous step: TF(cat) = 0.12

We calculated the IDF value IDF(cat) = log10(100/3)

Therefore, TF-IDF(cat) = 0.12 * log10(100/3)

8. In CBOW,
(i) Can we use a vocabulary index method (instead of a one-hot vector) as an input to CBOW

No, directly using a vocabulary index method (like assigning a unique integer to each word) as input to CBOW is not acceptable because of the following reasons:

1. **Loss of Information:** Vocabulary indexing treats words as isolated entities identified by their position (index) in the vocabulary. This approach discards valuable information about the word itself and its relationship with other words.
2. **Limited Distance Representation:** Neural networks rely on vector representations to capture semantic relationships between words. A simple index doesn't provide this capability. One-hot vectors, while sparse, offer a better representation by activating the corresponding dimension for the word, allowing the network to learn distances and similarities between words.
3. **Functionality Issues:** The CBOW model operates by predicting the target word based on the surrounding context words. It needs to perform mathematical operations on the input vectors (e.g., averaging, concatenation) to capture the context. These operations wouldn't be meaningful with simple indices.

In essence, vocabulary indexing alone doesn't provide the necessary information for the neural network in CBOW to learn effectively**.**

One-hot vectors, although less efficient in terms of memory usage, offer a better trade-off by preserving some information about word identity while allowing the neural network to learn relationships between words.

(ii) The activation function of the output layer should be softmax. Explain the reason.

In the output layer of a CBOW model, the activation function should indeed be softmax.

1. CBOW as a Multi-class Classification Problem:

In CBOW, the model predicts the target word based on its surrounding context words. This prediction can be framed as a multi-class classification problem. The model needs to output a probability distribution over all the words in the vocabulary, indicating how likely each word is to be the target word given the context.

2. Softmax and Probability Distribution:

The softmax function is specifically designed to generate a probability distribution from a vector of real numbers. It takes a vector of values (one for each word in the vocabulary) as input and transforms them into a probability distribution where the sum of all the outputs adds up to 1.

Example:

Imagine a CBOW model predicting the next word after "the" and "sky" (context words). The softmax function would take the activation values for all words in the vocabulary (e.g., "blue", "cloud", "rain", etc.) and convert them into probabilities. The highest probability would correspond to the word the model predicts is most likely to follow "the sky."

3. Interpretation of Outputs:

Softmax activation allows the CBOW model's outputs to be interpreted as actual probabilities. This is crucial for evaluating the model's performance and understanding its predictions. For instance, a probability of 0.8 for "blue" and 0.1 each for "cloud" and "rain" suggests the model strongly predicts "blue" as the next word.

Therefore the softmax function ensures the CBOW model's output layer produces a valid probability distribution for word prediction, enabling meaningful interpretation of the model's results.

(iii) After training is done in CBOW, how do we extract the word embedding vector for a certain word.

In a trained CBOW model, the word embedding vectors are typically stored within the model's weight matrices. We can extract the word embedding vector for a specific word as follows:

1. Identifying the Embedding Layer:

The exact method for extracting word embeddings depends on the deep learning framework used to train the CBOW model (e.g., TensorFlow, PyTorch). However, most frameworks have a designated layer responsible for holding the word embeddings. This layer is often named something like "embedding_layer" or "word_embedding."

2. Accessing the Weight Matrix:

Within the identified embedding layer, there's a weight matrix that stores the actual word embedding vectors. This matrix has dimensions (vocabulary_size, embedding_dimension). Each row in this matrix represents the embedding vector for a specific word in the vocabulary.

3. Mapping Word to Index:

Each word has a corresponding unique index in that vocabulary. We'll need to map the word we're interested in (e.g., "apple") to its corresponding index in the vocabulary.

4. Extracting the Embedding Vector:

Once you have the word's index and the weight matrix from the embedding layer, you can simply access the corresponding row in the weight matrix. This row represents the word embedding vector for the specific word you requested.

Here's a breakdown of the steps:

- **Trained CBOW Model:** We assume you have a trained CBOW model using a specific deep learning framework.
- **Identify Embedding Layer:** Locate the layer that holds the word embeddings (e.g., "embedding_layer").
- **Access Weight Matrix:** Obtain the weight matrix from the embedding layer. This matrix has dimensions (vocabulary_size, embedding_dimension).
- **Map Word to Index:** Find the index of the word you want the embedding for (e.g., index of "apple").
- **Extract Embedding Vector:** Access the row in the weight matrix corresponding to the word's index. This row is the word embedding vector for the requested word.

- The extracted embedding vector will be a 1D array of numbers (length = embedding_dimension) representing the word's embedding in the high-dimensional space learned during training.

9. In Skip-Gram, show that F=exp(X) satisfies the following:

$$F\left((w_i - w_j)\tilde{w}_k\right) = \frac{F(w_i\tilde{w}_k)}{F\left(w_j\tilde{w}_k\right)}$$

Absolutely, F(x) = exp(x) satisfies the Skip-Gram property:

Equation:

$$F\left((w_i - w_j)\tilde{w}_k\right) = \frac{F(w_i\tilde{w}_k)}{F\left(w_j\tilde{w}_k\right)}$$

Here,

wi, wj, and wk represent word vectors.

“- “denotes vector subtraction.

element-wise vector dot product is sometimes denoted by · or *

Steps to Show the Property Holds:

1. **Property of Exponentiation:** We'll leverage the same property of exponentiation as before:

a^(b + c) = a^b * a^c

This applies for any real numbers a, b, and c.

2. **Applying the Property:** Let's break down the equation using the property:

$F((w_i - w_j) \circ w_k) = \exp((w_i - w_j) \circ w_k)$ // Because F(x) = exp(x)

$F(w_i \circ w_k) / F(w_j \circ w_k) = \exp(w_i \circ w_k) / \exp(w_j \circ w_k)$

3. **Simplifying the Equation:**

Now, we can simplify the equation further because exponentiation distributes over element-wise multiplication of vectors:

$\exp((w_i \circ w_k) - (w_j \circ w_k)) = \exp(w_i \circ w_k) / \exp(w_j \circ w_k)$

4. **Matching Left and Right Sides:**

The left side becomes:

exp((wi - wj) ∘ wk)

This is because (wi ∘ wk) - (wj ∘ wk) results in element-wise subtraction of corresponding elements from wi ∘ wk and wj ∘ wk, which is the same as (wi - wj) ∘ wk due to the distributive property of vector subtraction over element-wise multiplication.

The right side remains:

exp(wi ∘ wk) / exp(wj ∘ wk)

Therefore, we've shown that for F(x) = exp(x), the equation holds true:

exp((wi - wj) ∘ wk) = exp(wi ∘ wk) / exp(wj ∘ wk)

In essence, the property of exponentiation allows us to manipulate the equation with F(x) = exp(x) and demonstrate that both sides become equivalent.

This equation captures the core principle of Skip-Gram: the dot product of the difference between two word vectors (wi - wj) and a third word vector (wk) should be proportional to the ratio of the dot products between the first word vector (wi) and wk and the second word vector (wj) and wk.

10. If we remove $f(X_{ij})$ from the following formula, what problems do we have?

$$\sum_{i,j\in D} f(X_{ij})\left(w_i^T\tilde{w}_j - \log X_{ij} + \tilde{b}_j + b_i\right)^2$$

If we remove f(X_{ij})it would lead to several issues:

1. **Loss of Weighting:** The term f(Xij) likely serves as a weighting factor. It could be scaling the contribution of each term based on the value of Xij. Removing f(Xij) would mean treating all terms equally, regardless of the value of Xij. This might not be appropriate, especially if some Xij values are more significant or relevant than others.

2. **Bias:** The function f(Xij) might be correcting for bias or adjusting the contribution of each term based on some characteristics of Xij. Removing it could introduce bias into the model.

3. Impact on Model Behavior: Depending on the nature of f(Xij), its removal could drastically change the behavior of the model. For instance, if f(Xij) is a nonlinear transformation, removing it would make the model linear, which could be inappropriate for certain datasets.

4. Loss of Contextual Information: f(Xij) likely contains contextual information about Xij that is important for the model. Removing it could lead to a loss of this contextual information, potentially resulting in a less accurate or less informative model.

5. Violation of Modeling Assumptions: If f(Xij) is necessary for meeting certain modeling assumptions (such as linearity, normality, etc.), removing it could violate these assumptions, leading to biased or unreliable results.

6. Reduced Flexibility: Removing f(Xij) might reduce the flexibility of the model. Models often include such functions to capture nonlinear relationships or complex interactions.

Removing it would make the model simpler but could also make it less capable of capturing the complexities of the underlying data.

In summary, removing f(Xij) from the formula would likely lead to a less flexible, potentially biased model that fails to appropriately weight or adjust for the importance of different Xij values.

11. 11. Autoencoder can reduce input dimensions.
(i) In Sparse autoencoder, the latent space dimension is larger than the input dimension. Explain how a sparse autoencoder can reduce the input dimension.

A standard characteristic of sparse autoencoders is that the latent space dimension (code dimension) is typically **smaller** than the input dimension. This allows for dimensionality reduction while capturing essential features of the data.

A sparse autoencoder achieves dimensionality reduction even when the latent space appears larger:

1. Bottleneck Layer: Sparse autoencoders often introduce a bottleneck layer in the hidden architecture. This layer has a smaller number of units compared to both the input and output layers.

Example:

- Input dimension: 1000 features
- Bottleneck layer: 256 units
- Output dimension (reconstruction): 1000 features

2. Information Bottleneck and Sparsity:

- The bottleneck layer acts as an information bottleneck. It forces the encoder to compress the input data into a lower-dimensional representation (256 units) while capturing the most important features for reconstruction.
- Sparsity constraints are often applied during training. This encourages the network to activate only a few neurons in the bottleneck layer at a time, promoting the selection of the most informative features.

3. Reconstruction:

- The decoder aims to reconstruct the original input data (1000 dimensions) using the compressed representation from the bottleneck layer (256 units).
- While the output dimension appears to be the same as the input, the crucial aspect is that the network successfully learned a lower-dimensional representation that captures the essence of the data.

In essence, the sparse autoencoder achieves dimensionality reduction by:

- Forcing information compression through the bottleneck layer.
- Encouraging sparsity to select the most informative features.
- Reconstructing the input data from the compressed representation, demonstrating the effectiveness of the dimensionality reduction.

Therefore, even though the latent space might have more units than the input in some cases (depending on the architecture), the key lies in the network's ability to learn a compressed representation that captures the most important features for reconstruction in a lower-dimensional space.

(ii) What happens if a sparse autoencoder uses L2(Ridge) regularization? Explain it.

The L2 (Ridge) regularization affects a sparse autoencoder in the following ways:

Impact of L2 Regularization:

L2 regularization adds a penalty term to the loss function of the autoencoder. This penalty term is proportional to the sum of the squares of the weights in the network. The goal is to:

- **Reduce model complexity:** By penalizing large weights, L2 regularization discourages the network from fitting the training data too closely, potentially reducing overfitting.
- **Improve generalization:** A less complex model with smaller weights might generalize better to unseen data.

Impact on Sparsity in Sparse Autoencoders:

While L2 regularization can be beneficial for a sparse autoencoder, it might have a conflicting effect with sparsity:

- **Sparsity encourages:** Sparse autoencoders often use techniques like dropout or penalty terms that encourage only a few neurons in each layer to be active at a time. This promotes the selection of the most informative features.
- **L2 regularization discourages:** L2 regularization penalizes all weights, including those that might be important for achieving sparsity. This can potentially weaken the effect of the sparsity-inducing techniques.

Finding the Balance:

Therefore, when using L2 regularization in a sparse autoencoder, it's important to find the right balance. Too much regularization can hinder the network's ability to achieve the desired level of sparsity, potentially reducing the effectiveness of dimensionality reduction and feature selection.

Alternative Regularization Techniques:

For sparse autoencoders, some alternative regularization techniques might be more suitable:

- **L1 regularization (LASSO):** L1 regularization penalizes the absolute values of the weights, which can directly drive some weights to zero, promoting sparsity more effectively.
- **Elastic Net:** This combines L1 and L2 regularization, offering a compromise between reducing weights to zero (L1) and shrinking them towards zero (L2).

Overall, While L2 regularization can be helpful for reducing model complexity and improving generalization, it might have a negative impact on sparsity in sparse autoencoders. It's essential to consider alternative regularization techniques or carefully tune the L2 regularization hyperparameter to achieve the desired balance between sparsity and model complexity.