Practical-5 CBOW

Imports and Initial Setup

```python
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib as mpl
import matplotlib.pylab as pylab
import numpy as np
%matplotlib inline
```

Explanation:
- **matplotlib** and **seaborn** are for visualization. **matplotlib.pyplot** provides plotting functions, and **seaborn** offers advanced styling for plots.
- **%matplotlib inline** enables inline plotting in Jupyter notebooks (if you're using one).
- **numpy (np)** is for numerical operations, especially useful for handling arrays and matrices, which are crucial in machine learning.

Data Preparation

```python
import re
```

Explanation:
- **re**: This library handles regular expressions, which are useful for text preprocessing, like removing unwanted characters.

```python
sentences = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells."""
```

Explanation:

- This variable **sentences** contains a block of text used for training the CBOW model. It’s meant to simulate a small corpus or set of sentences, which the model will learn word relationships from.

```python
# Clean Data

# remove special characters
sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences)

# remove 1 letter words
sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()

# lower all characters
sentences = sentences.lower()
```

Explanation:
- **Data Cleaning**:
  - **Remove special characters**: Replaces any non-alphanumeric characters with spaces to clean up the text.
  - **Remove single characters**: Gets rid of isolated single characters, as they may not carry useful information in this context.
  - **Convert to lowercase**: Makes the text lowercase to ensure that words like "Process" and "process" are treated the same.

Vocabulary Preparation

```python
words = sentences.split()
vocab = set(words)
```

Explanation:
- **Tokenization**:
  - **words**: Splits the cleaned text into a list of individual words.
  - **vocab**: Creates a set of unique words, which will be used to build the vocabulary. This set will be used to assign each word a unique index for training.

```python
```

vocab_size = len(vocab)
embed_dim = 10
context_size = 2
```

#### Explanation:
- **Define Model Parameters**:
  - **vocab_size**: Total number of unique words in our vocabulary, needed for building word embeddings.
  - **embed_dim**: Dimensionality of the word embeddings. Each word will be represented as a vector of 10 values.
  - **context_size**: Number of words to look at on each side of a target word. Here, we look 2 words before and 2 words after.

---

### Vocabulary Indexing

```python
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for i, word in enumerate(vocab)}
```

Explanation:
- **Create Word-Index Mappings**:
  - **word_to_ix**: Maps each word in the vocabulary to a unique integer index.
  - **ix_to_word**: Reverse mapping to retrieve the word given its index. This is useful for converting predictions back into words.

Data Preparation for CBOW

```python
# Data bags
# data - [(context), target]
data = []
for i in range(2, len(words) - 2):
    context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]
    target = words[i]
    data.append((context, target))
print(data[:5])
```

Explanation:
- **Generate Training Pairs**:
  - **Loop through each word** with a range starting from index 2 and ending
```

two indices before the last word.
  - **Context words**: For each target word, define its context by selecting two words before and two words after it.
  - **Target word**: The word we want to predict based on its context.
  - **Append (context, target) pairs** to `data`. Each pair will be used for training the CBOW model.
  - **print(data[:5])**: Prints the first five pairs for verification.

Initialize Embeddings

```python
embeddings =  np.random.random_sample((vocab_size, embed_dim))
```

Explanation:
- **Initialize Random Embeddings**:
  - **embeddings**: Creates an array of random numbers with dimensions `vocab_size x embed_dim`. Each row represents a word's embedding vector.
  - This is the initial random embedding matrix, which will be adjusted during training to capture word meanings.

Linear Model

```python
# Linear Model
def linear(m, theta):
    w = theta
    return m.dot(w)
```

Explanation:
- **Define Linear Transformation**:
  - **linear function**: Takes an input matrix `m` and applies a transformation using the weights matrix `theta` (also called `w` here).
  - **dot product**: Multiplies input matrix `m` with weights `w` to obtain transformed output, which will later be passed to an activation function.

Log Softmax and Cross Entropy Loss

```python
# Log softmax + NLLloss = Cross Entropy
```

def log_softmax(x):
    e_x = np.exp(x - np.max(x))
    return np.log(e_x / e_x.sum())
```

#### Explanation:
- **Log Softmax**:
  - **log_softmax** function computes the log of softmax values for an input `x`.
  - **Softmax**: Transforms the output values into probabilities, with each value representing the likelihood of a word being the correct target.
  - **Log**: Converts probabilities into log-probabilities, which are easier to use with the cross-entropy loss.

---

```python
def NLLLoss(logs, targets):
    out = logs[range(len(targets)), targets]
    return -out.sum()/len(out)
```

Explanation:
- **Negative Log-Likelihood Loss (NLLLoss)**:
  - **logs**: Log-probabilities from `log_softmax`.
  - **targets**: Indices of actual target words.
  - **Select target log-probabilities**: Extracts the log-probabilities of correct target words.
  - **Mean Negative Log-Sum**: Takes the average of the negative log-sum, giving the model a measure of its error. This is what the model tries to minimize during training.

```python
def log_softmax_crossentropy_with_logits(logits, target):
    out = np.zeros_like(logits)
    out[np.arange(len(logits)), target] = 1
    softmax = np.exp(logits) / np.exp(logits).sum(axis=-1, keepdims=True)
    return (- out + softmax) / logits.shape[0]
```

Explanation:
- **Combined Log Softmax and Cross Entropy**:
  - **One-hot encoding** for `target`: Converts the target indices into a format where only the correct class is 1, and others are 0.
  - **Softmax** computes probabilities, similar to `log_softmax`, but without
```

taking the log.
  - **Error Computation**: Returns the gradient of the cross-entropy loss with respect to logits for backpropagation.

Forward Pass Function

```python
def forward(context_idxs, theta):
    m = embeddings[context_idxs].reshape(1, -1)
    n = linear(m, theta)
    o = log_softmax(n)
    return m, n, o
```

Explanation:
- **Forward Pass**:
  - **context_idxs**: The indices of context words.
  - **m**: Retrieves and flattens embeddings for the context words.
  - **n**: Applies the linear transformation to get logits.
  - **o**: Applies log_softmax to get log-probabilities, which are used to calculate the loss.

Backward Pass (Gradient Calculation)

```python
def backward(preds, theta, target_idxs):
    m, n, o = preds
    dlog = log_softmax_crossentropy_with_logits(n, target_idxs)
    dw = m.T.dot(dlog)
    return dw
```

Explanation:
- **Backward Pass**:
  - **preds**: Output from the forward pass.
  - **dlog**: Computes gradients with respect to the logits.
  - **dw**: Calculates gradient of the weights `theta` by multiplying context embedding matrix `m.T` with `dlog`.

Optimization Step

```python
def optimize(theta, grad, lr=0.03):
    theta -= grad * lr
    return theta
```

Explanation:
- **Update Weights**:
  - **theta**: Weight matrix for the model.
  - **grad**: Gradient of the loss with respect to `theta`.
  - **lr**: Learning rate, controlling the update size.
  - **theta -= grad * lr**

Training Loop: Generate Training Data and Optimize Weights

```python
# Training
#Generate training data
theta = np.random.uniform(-1, 1, (2 * context_size * embed_dim, vocab_size))
```

Explanation:
- **Initialize the `theta` matrix**:
  - **theta**: A randomly initialized weight matrix that will be updated during training.
  - **Dimensions**: `(2 * context_size * embed_dim, vocab_size)` means this matrix has rows equal to the number of input features (determined by `context_size` and `embed_dim`) and columns equal to the vocabulary size. This structure helps the model map from input embeddings to output probabilities.

```python
epoch_losses = {}

for epoch in range(80):

    losses = []

    for context, target in data:
        context_idxs = np.array([word_to_ix[w] for w in context])
        preds = forward(context_idxs, theta)
```

```
        target_idxs = np.array([word_to_ix[target]])
        loss = NLLLoss(preds[-1], target_idxs)

        losses.append(loss)

        grad = backward(preds, theta, target_idxs)
        theta = optimize(theta, grad, lr=0.03)

    epoch_losses[epoch] = losses
```

Explanation:
- **Training Loop**:
  - **epoch_losses**: Dictionary to store the loss for each epoch.
  - **epochs (80)**: The model goes through the entire dataset 80 times to learn patterns in the data.

- **Inner Loop (per context-target pair)**:
  - **context_idxs**: Converts context words into their respective indices.
  - **forward**: Computes the predicted output (`preds`) for the context words using the `theta` matrix.
  - **target_idxs**: Converts the target word into its index.
  - **loss**: Calculates the error between the predicted output and the actual target using Negative Log-Likelihood Loss (`NLLLoss`).
  - **backward**: Computes gradients of the loss with respect to `theta`.
  - **optimize**: Updates `theta` by applying gradients, allowing the model to improve its predictions.

Loss Analysis: Plotting Loss per Epoch

```python
# Analyze
# Plot loss/epoch
ix = np.arange(0,80)
fig = plt.figure()
fig.suptitle('Epoch/Losses', fontsize=20)
plt.plot(ix,[epoch_losses[i][0] for i in ix])
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Losses', fontsize=12)
```

Explanation:
- **Loss Plotting**:
  - **ix**: Array of epoch numbers from 0 to 80.

  - **plt.plot**: Plots the loss per epoch. This shows how the model’s error decreases as it trains, indicating it is learning.
  - **xlabel and ylabel**: Labeling the axes to show “Epochs” on the x-axis and “Losses” on the y-axis for easy interpretation.

Prediction Function

```python
# Predict function
def predict(words):
    context_idxs = np.array([word_to_ix[w] for w in words])
    preds = forward(context_idxs, theta)
    word = ix_to_word[np.argmax(preds[-1])]
    return word
```

Explanation:
- **Predict Function**:
  - **context_idxs**: Converts the input words (context) into indices using `word_to_ix`.
  - **forward**: Generates predictions using the current `theta` matrix.
  - **np.argmax**: Selects the word with the highest probability as the predicted target word.
  - **ix_to_word**: Converts the index back to the actual word using `ix_to_word`.
  - This function allows us to test the model by inputting a few context words and seeing what target word it predicts.

Example Prediction

```python
# (['we', 'are', 'to', 'study'], 'about')
predict(['we', 'are', 'to', 'study'])
```

Explanation:
- **Example Use of `predict`**:
  - This line predicts the target word for the context words `['we', 'are', 'to', 'study']`.
  - **Expected Output**: The model will output the word it finds most probable to be associated with this context, helping us check if the model learned meaningful relationships.

Model Accuracy Function

```python
# Accuracy
def accuracy():
    wrong = 0

    for context, target in data:
        if(predict(context) != target):
            wrong += 1

    return (1 - (wrong / len(data)))
```

Explanation:
- **Accuracy Calculation**:
  - **Loop through all context-target pairs in `data`**.
  - **Prediction Check**: If the predicted target word doesn’t match the actual target, increment the `wrong` counter.
  - **Accuracy Formula**: `(1 - (wrong / len(data)))` gives the percentage of correct predictions. Higher accuracy means the model has learned well.

Final Prediction Example

```python
predict(['processes', 'manipulate', 'things', 'study'])
```

Explanation:
- **Another Example Prediction**:
  - This line tests the model by inputting the context `['processes', 'manipulate', 'things', 'study']`.
  - It checks whether the model can predict a meaningful target word based on this new context, allowing us to evaluate the learned associations.

——————————————————————————————————————

Here are key questions for Assignment 5, which focuses on implementing the Continuous Bag of Words (CBOW) model for natural language processing (NLP):

Important Questions and Answers

1. **What is Natural Language Processing (NLP)?**
   - NLP is a field of AI that enables computers to understand, interpret, and generate human language.

2. **What is Word Embedding in NLP?**
   - Word embeddings are vector representations of words that capture semantic relationships, allowing similar words to have similar vector representations.

3. **What is the Word2Vec technique?**
   - Word2Vec is a model that transforms words into vector space, using CBOW or Skip-gram to learn word relationships based on context.

4. **Explain the architecture of the Continuous Bag of Words (CBOW) model.**
   - In CBOW, the model predicts a target word from a given context of surrounding words, learning embeddings for each word based on this context.

5. **What is the input and output of the CBOW model?**
   - **Input**: Context words (surrounding words in a sentence).
   - **Output**: Target word (the word predicted based on the context).

6. **What is the purpose of a Tokenizer in NLP?**
   - A tokenizer splits text into smaller units (tokens), like words or sentences, enabling easier processing for NLP tasks.

7. **Explain the window size parameter in the CBOW model.**
   - Window size determines the number of context words used on each side of the target word, affecting how much surrounding context the model considers.

8. **What are Embedding and Lambda layers in Keras?**
   - **Embedding Layer**: Transforms integer-encoded words into dense vectors.
   - **Lambda Layer**: Allows custom operations to be applied within the model, often used for function-based transformations.

9. **What is the purpose of the yield() function in Python?**
   - `yield()` is used in generators to return intermediate values without exiting the function, allowing iteration over large data without excessive memory use.

10. **Describe the process of data preparation for training a CBOW model.**
    - Data preparation involves tokenizing text, creating word pairs of context and target words, and encoding them into numerical format for the model.

11. **Why is the Sequential model commonly used in CBOW implementations?**

- Sequential models are simple to set up and suitable for layer-wise stacking, making them ideal for straightforward architectures like CBOW.

12. **What is Gensim, and how is it used in NLP?**
    - Gensim is an NLP library in Python for topic modeling and word embedding. It provides tools to train models like Word2Vec, CBOW, and Skip-gram.

13. **How does the CBOW model represent word similarity?**
    - The CBOW model learns word embeddings such that similar words have similar vector representations, allowing the model to predict related words accurately.

14. **Explain the concept of context and target words in CBOW.**
    - **Context Words**: Words surrounding a target word in a sentence.
    - **Target Word**: The word being predicted based on its surrounding context.

15. **What is the purpose of vectorization in CBOW?**
    - Vectorization converts words into numerical representations (embeddings), enabling the model to perform mathematical operations and learn relationships.

Would you like additional questions or more details on any of these answers?