Continuous Bag of Words (CBOW)

is a natural language processing (NLP) technique used to train word embeddings, which are dense vector representations of words in a way that captures their semantic meaning. CBOW is one of the two architectures used in Word2Vec, a popular word embedding model developed by Tomas Mikolov and his team at Google.

The main idea behind CBOW is to predict a target word based on the context of the surrounding words. Unlike another Word2Vec architecture called Skip-gram, which predicts the context words given a target word, CBOW does the opposite. It aims to learn the word embeddings by predicting a target word using its neighboring words.

Here's how CBOW works:

1. Data Preparation:
To train a CBOW model, you start with a large corpus of text. You slide a fixed-size window (typically a small number of words) over the text, and at each position, you collect the context words within that window. These context words are used to predict the target word in the center of the window.

2. Word Embeddings:
Each word in the vocabulary is assigned a unique vector representation (embedding). These embeddings are initialized with random values.

3. **Neural Network Architecture:**
CBOW uses a neural network to predict the target word based on the context words. The architecture typically consists of an input layer, a hidden layer, and an output layer. The input layer represents the context words, and the output layer represents the target word. The hidden layer is where the transformation from context to target information occurs.

4. **Training Objective:**
The objective of the training process is to minimize the difference between the predicted word (in the output layer) and the actual target word for a given context. This is usually done using techniques like the softmax function and cross-entropy loss.

5. **Backpropagation and Optimization:**
The model adjusts the word embeddings in the hidden layer and the weights in the neural network through backpropagation and gradient descent. The goal is to minimize the prediction error.

6. **Word Embedding Learning:**

As the model is trained, the word embeddings in the hidden layer gradually adapt to capture the semantic relationships between words in the training data. The embeddings for words with similar meanings become more similar in vector space.

7. **Word Vector Usage:**
Once training is complete, you can use the learned word embeddings to perform various NLP tasks. These embeddings are effective at capturing semantic similarities, and you can use them to find similar words, analyze word relationships, or as input features for downstream NLP applications like sentiment analysis, machine translation, or text classification.

CBOW is known for its simplicity and efficiency in learning word embeddings, especially for words with frequent occurrences in the training data. It's a valuable tool in NLP because it helps capture semantic meaning in a distributed representation, which can be used to improve the performance of various language understanding tasks.

Implementation:

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

1. **import matplotlib.pyplot as plt**: This line imports the **matplotlib** library, which is a widely used library for creating visualizations in Python. By importing **matplotlib.pyplot** as **plt**, you can use **plt** as a shorthand for functions and methods provided by **matplotlib.pyplot**.
2. **import seaborn as sns**: This line imports the **seaborn** library, which is built on top of **matplotlib** and provides a higher-level interface for creating informative and attractive statistical graphics. It's particularly useful for creating aesthetically pleasing data visualizations.
3. **import matplotlib as mpl**: This line imports the **matplotlib** library with the alias **mpl**. It allows you to access **matplotlib** functions and attributes using **mpl** as a prefix, which can be helpful if there are naming conflicts with other libraries or code.
4. **import matplotlib.pylab as pylab**: This line imports the **pylab** module from **matplotlib**. The **pylab** module is a convenience module that provides a more MATLAB-like interface for **matplotlib**. However, it's not recommended to use **pylab** because it can lead to naming conflicts and confusion in your code.
5. **import numpy as np**: This line imports the **numpy** library, which is a fundamental library for numerical and scientific computing in Python. It provides support for arrays and

matrices, which are essential for data manipulation and calculations in data visualization and data analysis.

6. **%matplotlib inline**: This is a magic command in Jupyter Notebook or Jupyter Lab environments. When you run this command, it sets the backend for **matplotlib** to display plots inline within the notebook, meaning that any plots generated using **matplotlib** will be shown directly in the notebook cells rather than in separate windows or files. This is particularly useful for interactive data analysis and visualization in Jupyter environments.

```
#Data Prepration
import re
```

The **import re** statement in Python is used to import the "re" module, which stands for "regular expressions." Regular expressions are a powerful tool for pattern matching and text manipulation. The "re" module provides functions and methods for working with regular expressions in Python.

Importing the "re" Module: Once you have imported the "re" module, you can use its functions and methods to perform operations related to pattern matching and text processing.

1. **Common Uses**: The "re" module is commonly used for the following tasks:
    - **Pattern Matching**: You can define patterns using regular expressions and search for, match, or extract specific text patterns within a given string.
    - **Text Parsing**: Regular expressions can be used to parse structured text data, such as log files, CSV files, and more.
    - **Text Cleaning**: You can use regular expressions to clean and modify text, such as removing unwanted characters, formatting text, or extracting specific information.

```
# 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()
```

Contains a series of operations using the `re` module for text processing and cleaning.

1. `sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences)`

- This line uses the `re.sub()` function from the `re` module to perform a substitution operation on the `sentences` variable.
- The regular expression pattern `[^A-Za-z0-9]+` matches one or more characters that are not letters (uppercase or lowercase) or digits (0-9). It essentially matches any character that is not alphanumeric.
- The `re.sub()` function replaces all the matched substrings with a single space.
- The result is that it removes any non-alphanumeric characters from the `sentences` variable and replaces them with spaces. This is often done to clean and normalize text data, making it more suitable for text processing or analysis.

2. `sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()`
    - This line is another use of the `re.sub()` function to perform a substitution operation on the `sentences` variable.
    - The regular expression pattern `(?:^| )\w(?:$| )` is a bit more complex:
        - `(?:^| )` matches the start of the string or a space.
        - `\w` matches a single word character (letters, digits, or underscore).
        - `(?:$| )` matches the end of the string or a space.
    - Essentially, this pattern is designed to match single word characters surrounded by spaces or at the start or end of the string.
    - The `re.sub()` function replaces all the matched substrings with a single space.
    - The `strip()` method is then applied to remove any leading or trailing spaces that might have been introduced by the previous step.
    - The result is that it removes isolated single word characters (e.g., "a" or "I") and replaces them with spaces, while ensuring there are no leading or trailing spaces.
3. `sentences = sentences.lower()`
    - This line converts all the text in the `sentences` variable to lowercase.
    - It's a common operation in text processing to ensure that the text is case-insensitive, making it easier to work with and compare strings. This step is often taken to standardize the text.

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

The code you provided involves splitting a sentence into words and then creating a vocabulary set from those words.

words = sentences.split()

- In this line, the **.split()** method is applied to the **sentences** variable.
- The **.split()** method is a built-in string method in Python that splits a string into a list of substrings based on whitespace (spaces and/or tabs) by default.
- In this case, it splits the **sentences** string into a list of words, where each word is an element in the resulting list.
- For example, if **sentences** is "This is a sample sentence," then **words** would be a list containing **['This', 'is', 'a', 'sample', 'sentence']**.

2. **vocab = set(words)**
   - In this line, the **set()** function is used to convert the list of words (**words**) into a set called **vocab**.
   - A set is a built-in data structure in Python that stores an unordered collection of unique elements. In this context, it is used to create a unique set of words from the list.
   - This step effectively removes duplicate words from the list, as sets only store unique elements. It's a common technique to generate a vocabulary, which is a collection of unique words present in a text.
   - The **vocab** set will contain a unique set of words from the original sentence, and any duplicate words will be automatically removed.

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

The code you provided assigns values to three variables: **vocab_size**, **embed_dim**, and **context_size**.

1. **vocab_size = len(vocab)**
   - **vocab** appears to be a set containing a collection of unique words from a text, as explained in the previous code snippet.
   - The **len()** function is used to determine the number of elements in the set **vocab**, which corresponds to the number of unique words.
   - This number is assigned to the variable **vocab_size**.
   - **vocab_size** represents the size of the vocabulary, i.e., the total number of unique words in the text. It's a crucial parameter in many natural language processing tasks.
2. **embed_dim = 10**
   - In this line, the variable **embed_dim** is assigned the value **10**.
   - **embed_dim** stands for "embedding dimension" and represents the number of dimensions in the vector space where words will be represented as vectors.

Word embeddings are dense vector representations of words that capture their semantic meaning.
- A common practice in NLP is to choose an embedding dimension that is relatively small, often between 50 to 300 dimensions, but for this specific code, it's set to 10, meaning that each word will be represented as a 10-dimensional vector.

3. **context_size = 2**
   - The variable **context_size** is assigned the value **2**.
   - In NLP, the context size refers to the number of words considered on each side of a target word when creating word embeddings.
   - A **context_size** of 2 implies that when constructing word embeddings, the algorithm will consider two words to the left and two words to the right of the target word. This means that the context for each word consists of a total of 5 words (2 to the left, the target word itself, and 2 to the right).
   - The choice of **context_size** is essential in word embedding algorithms like Word2Vec, as it affects the information captured about the relationships between words.

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

The code you provided is used to create two dictionaries: **word_to_ix** and **ix_to_word**. These dictionaries are typically used in natural language processing (NLP) tasks, particularly when working with word embeddings or text processing.

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

- This line creates the **word_to_ix** dictionary, which maps words (from the **vocab** set) to their corresponding integer indices.
- It uses a dictionary comprehension, a concise way to create dictionaries in Python.
- The **enumerate()** function is used to iterate over the elements in the **vocab** set, providing both the index **i** and the word **word** for each element.
- For each word in the **vocab** set, the dictionary comprehension assigns the word as the key and the index as the value in the **word_to_ix** dictionary.
- This dictionary allows you to look up the index of a word in the vocabulary quickly.

2. **ix_to_word = {i: word for i, word in enumerate(vocab)}**
   - This line creates the **ix_to_word** dictionary, which is the inverse of **word_to_ix**. It maps integer indices to their corresponding words.

- Like the previous line, it uses a dictionary comprehension with **enumerate()** to iterate over the elements in the **vocab** set, providing both the index **i** and the word **word** for each element.
- For each word in the **vocab** set, the dictionary comprehension assigns the index as the key and the word as the value in the **ix_to_word** dictionary.
- This dictionary allows you to look up the word associated with a given index in the vocabulary.

```
# 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])
```

The code you provided is used to prepare data for a word embedding or language modeling task. It creates a dataset `data` by extracting context-target pairs from a list of words (`words`).

`data = []`: This line initializes an empty list named `data` to store the context-target pairs extracted from the text.

1. `for i in range(2, len(words) - 2):`: This line sets up a `for` loop that iterates through the range of indices from 2 to `len(words) - 2`. The loop variable `i` represents the index of the target word in the list of `words`.
    - `range(2, len(words) - 2)` ensures that you skip the first two words (to the left of the target word) and the last two words (to the right of the target word) because they don't have enough context words.
2. `context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]`: Inside the loop, this line creates a `context` list that contains the four words surrounding the target word.
    - `words[i - 2]`, `words[i - 1]` represent the two words to the left of the target word.
    - `words[i + 1]`, `words[i + 2]` represent the two words to the right of the target word.
    - The `context` list holds these four context words.

3. `target = words[i]`: This line assigns the word at the current index `i` to the `target` variable. This is the word you want to predict based on its surrounding context.
4. `data.append((context, target))`: Inside the loop, this line appends a tuple containing the `context` list and the `target` word to the `data` list. This effectively creates a context-target pair and adds it to the dataset.
5. `print(data[:5])`: After the loop completes, this line prints the first five context-target pairs in the `data` list. These pairs are the result of processing the text.

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

The code you provided creates a matrix of word embeddings using NumPy, with random initial values.

1. `import numpy as np`: This code assumes that you have imported the NumPy library and given it the alias `np`. NumPy is a popular library for numerical and scientific computing in Python, and it's commonly used for working with arrays and matrices.
2. `embeddings = np.random.random_sample((vocab_size, embed_dim))`:
   - `embeddings` is a variable that will store the word embeddings, typically represented as a matrix.
   - `np.random.random_sample` is a NumPy function used to generate random numbers from a uniform distribution between 0.0 and 1.0.
   - `(vocab_size, embed_dim)` is a tuple that specifies the shape of the matrix. `vocab_size` represents the number of unique words in your vocabulary (as explained in a previous code snippet), and `embed_dim` represents the desired dimensionality of the word embeddings.
   - The function creates a matrix with `vocab_size` rows (each row corresponds to a unique word) and `embed_dim` columns (each column represents a dimension in the embedding space).
   - The resulting `embeddings` matrix is populated with random values. These initial values are often used as a starting point when training word embeddings using techniques like Word2Vec or GloVe. During training, these random values will be adjusted based on the context in which the words appear, and they will gradually capture the semantic meaning of words.

```
def linear(m, theta):
    w = theta
    return m.dot(w)
```

The code you provided defines a Python function named `linear` that performs a linear transformation of a vector `m` using a weight vector `theta`. Here's an explanation of the code:

1. **`def linear(m, theta):`**: This line defines a function named `linear` that takes two arguments, `m` and `theta`.
    - `m` is assumed to be a NumPy array or a vector that you want to transform.
    - `theta` is also expected to be a NumPy array or vector representing the weights for the linear transformation.
2. **`w = theta`**: This line assigns the `theta` array to a new variable `w`. Essentially, it's creating an alias for the `theta` array with the name `w`.
3. **`return m.dot(w)`**: This line performs the linear transformation.
    - `m.dot(w)` computes the dot product between the input vector `m` and the weight vector `w`. The dot product is a mathematical operation that calculates the sum of the products of corresponding elements in the two vectors.
    - The result of this operation is returned as the output of the `linear` function.

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())
```

The code you provided defines a Python function named `log_softmax` that computes the logarithm of the softmax function for an input vector `x`. Here's an explanation of the code:

1. **`def log_softmax(x):`**: This line defines a function named `log_softmax` that takes a single argument, `x`, which is expected to be a NumPy array or vector.

2. `e_x = np.exp(x - np.max(x))`: This line calculates the softmax values for the input vector `x`. Let's break it down step by step:
    - `np.max(x)` finds the maximum value in the input vector `x`. This is done to avoid potential issues with numeric stability when exponentiating large values. Subtracting the maximum value from `x` ensures that the largest exponentiated value will be zero or negative.
    - `x - np.max(x)` subtracts the maximum value from each element in the vector `x`.
    - `np.exp(x - np.max(x))` applies the exponential function (element-wise) to the result of the subtraction. This is done to compute the exponentials of the values in the vector `x` after centering it around the maximum value. The `e_x` vector now contains these exponentials.
3. `return np.log(e_x / e_x.sum())`: This line computes the log of the softmax values and returns the result. Here's how it works:
    - `e_x.sum()` calculates the sum of all values in the `e_x` vector, which is the sum of exponentials of the centered input vector `x`.
    - `e_x / e_x.sum()` divides each element in the `e_x` vector by the sum, which normalizes the values to create probabilities. The resulting vector contains the probabilities of each element in the input vector `x`.
    - `np.log(e_x / e_x.sum())` computes the natural logarithm of the probabilities. This is the log-softmax operation.
    - The result is a new vector where each element represents the log of the softmax probability for the corresponding element in the input vector `x`.

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

The provided code defines a Python function named **`NLLLoss`** that computes the negative log-likelihood loss. This type of loss function is commonly used in machine learning, particularly in classification tasks, to measure the difference between predicted probability distributions (in the form of logarithms) and the actual target values.

1. `def NLLLoss(logs, targets):`: This line defines a function named `NLLLoss` that takes two arguments, `logs` and `targets`.
    - `logs` is expected to be a matrix of log-probabilities or log-scores. Each row of this matrix corresponds to a data point, and each column represents the log-probabilities of different classes or categories.

    - `targets` is expected to be a vector containing the target class labels for each data point.
2. **`out = logs[range(len(targets)), targets]`**: This line calculates the log-probabilities for the target classes. Here's how it works:
    - `range(len(targets))` generates a list of integers from 0 to one less than the length of the `targets` vector. This effectively creates a range of indices that corresponds to the rows of the `logs` matrix.
    - `logs[range(len(targets)), targets]` is a NumPy operation that selects specific elements from the `logs` matrix. It retrieves the log-probabilities from `logs` corresponding to the target classes specified in the `targets` vector. This creates a new vector, `out`, containing the log-probabilities of the target classes for each data point.
3. **`return -out.sum() / len(out)`**: This line calculates and returns the negative log-likelihood loss.
    - `-out.sum()` calculates the negative sum of the log-probabilities. This is because negative log-likelihood is commonly used as a loss function in machine learning, and the negative sign ensures that higher probabilities result in lower loss values.
    - `len(out)` gives the number of data points, representing the total number of target log-probabilities summed together.

```
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]
```

The provided code defines a Python function named `log_softmax_crossentropy_with_logits` that calculates the loss for a classification problem using the log-softmax and cross-entropy loss. This type of loss function is commonly used in machine learning for training and evaluating classification models.

1. **`def log_softmax_crossentropy_with_logits(logits, target):`**: This line defines the function `log_softmax_crossentropy_with_logits`, which takes two arguments: `logits` and `target`.

- `logits` represents the model's raw output scores for each class. It's a matrix where each row corresponds to a data point, and each column represents the unnormalized scores for different classes.
- `target` is a vector containing the true class labels for each data point.

2. `out = np.zeros_like(logits)`: This line initializes a matrix `out` with the same shape as `logits`. The purpose of this matrix is to represent one-hot encoding for the target classes. Each row of `out` will have a 1 in the column corresponding to the true class label and 0s in all other columns.
    - `np.zeros_like()` creates an array of zeros with the same shape as `logits`. It ensures that `out` has the same dimensions as `logits`.
3. `out[np.arange(len(logits)), target] = 1`: This line assigns 1 to the correct class label column for each data point in the `out` matrix. It performs one-hot encoding.
    - `np.arange(len(logits))` generates an array of integers from 0 to the number of rows in `logits`. This represents the indices of the rows.
    - `target` contains the true class labels, so `out[np.arange(len(logits)), target]` sets the values at these specific row-column indices to 1.
4. `softmax = np.exp(logits) / np.exp(logits).sum(axis=-1, keepdims=True)`: This line calculates the softmax probabilities for the model's output scores.
    - `np.exp(logits)` computes the exponentials of the elements in the `logits` matrix. This gives us the unnormalized probabilities.
    - `np.exp(logits).sum(axis=-1, keepdims=True)` calculates the sum of exponentials along the last axis (axis=-1), effectively summing across the classes for each data point. The `keepdims=True` argument retains the original shape of the result.
5. `return (-out + softmax) / logits.shape[0]`: This line calculates and returns the cross-entropy loss with respect to the log-softmax probabilities.
    - `(-out + softmax)` calculates the element-wise difference between the one-hot encoded `out` matrix and the softmax probabilities.
    - `logits.shape[0]` gives the number of data points (i.e., the number of rows in `logits`). This is used to normalize the loss, making it the average loss over all data points.

Forward function

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

    return m, n, o
```

return m, n, o

The provided code defines a Python function named `forward` that computes forward passes in a neural network or model, given context indices (`context_idxs`) and a weight vector `theta`. This function is typically used in models for natural language processing or deep learning.

1. `def forward(context_idxs, theta):`: This line defines a function named `forward` that takes two arguments: `context_idxs` and `theta`.
    - `context_idxs` is expected to be a list or array of indices representing context words or tokens. These indices are often used to look up word embeddings for words in the context.
    - `theta` is a weight vector used in a linear transformation. It is expected to be a NumPy array.
2. `m = embeddings[context_idxs].reshape(1, -1)`: This line extracts word embeddings for the given context indices and prepares them for further processing.
    - `embeddings[context_idxs]` extracts the word embeddings from a pre-existing `embeddings` matrix (assumed to contain word embeddings). It selects the rows corresponding to the context indices specified in `context_idxs`.
    - `reshape(1, -1)` reshapes the extracted embeddings into a 1D array. This is often done to ensure the dimensions match what's expected by the subsequent linear transformation.
3. `n = linear(m, theta)`: This line computes the linear transformation of the context embeddings using the weight vector `theta`.
    - `m` contains the context embeddings obtained in the previous step.
    - `theta` is a weight vector applied in a linear transformation.
    - The `linear` function, which is assumed to be defined elsewhere in the code, calculates the dot product of `m` and `theta`, effectively performing the linear transformation. This step is often part of building neural network layers.
4. `o = log_softmax(n)`: This line calculates the log-softmax probabilities for the linear transformation results.
    - `n` contains the results of the linear transformation.

    - The `log_softmax` function, which is assumed to be defined elsewhere in the code, computes the log-softmax operation on `n`. Log-softmax is often used for modeling probability distributions over classes in classification tasks.
5. `return m, n, o`: This line returns the intermediate results of the forward pass:
    - `m` is the context embeddings.
    - `n` is the result of the linear transformation.
    - `o` is the log-softmax probabilities.

Backward function

```
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
```

The provided code defines a Python function named `backward` that computes gradients during the backward pass of a neural network, given certain predictions (`preds`), a weight vector (`theta`), and target indices (`target_idxs`). This function is commonly used in machine learning and deep learning models to compute gradients for training.

1. `def backward(preds, theta, target_idxs):`: This line defines a function named `backward` that takes three arguments: `preds`, `theta`, and `target_idxs`.
    - `preds` is expected to be a tuple containing three elements: `m`, `n`, and `o`. These elements represent intermediate results from a forward pass, where `m` is context embeddings, `n` is the linear transformation result, and `o` is the log-softmax probabilities.
    - `theta` is a weight vector, which is presumably used in the forward pass.
    - `target_idxs` contains the target indices, representing the true class labels for the data.
2. `m, n, o = preds`: This line unpacks the elements of the `preds` tuple into separate variables `m`, `n`, and `o`. This allows you to work with these intermediate results individually.
3. `dlog = log_softmax_crossentropy_with_logits(n, target_idxs)`: This line computes the gradients for the log-softmax loss with respect to the linear transformation results (`n`) and the target indices.

- The `log_softmax_crossentropy_with_logits` function, which is assumed to be defined elsewhere in the code, is used to calculate the gradients for the log-softmax loss. This function typically computes gradients based on the predicted probabilities (`n`) and the true class labels (`target_idxs`).

4. `dw = m.T.dot(dlog)`: This line computes the gradient with respect to the weight vector `theta`.
    - `m.T` is the transpose of the context embeddings `m`. Transposing the matrix is often required to match the dimensions for matrix multiplication.
    - `m.T.dot(dlog)` computes the dot product between the transposed context embeddings and the gradients of the log-softmax loss. This represents the gradient with respect to the weight vector `theta`.
5. `return dw`: This line returns the computed gradient, `dw`. The gradient can be used in optimization algorithms, such as gradient descent, to update the weight vector during training.

Optimize function

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

The provided code defines a Python function named `optimize` that performs a simple optimization step by updating a weight vector (`theta`) using gradients (`grad`) and a learning rate (`lr`). This function is commonly used in machine learning and deep learning for updating model parameters during training.

1. `def optimize(theta, grad, lr=0.03):`: This line defines a function named `optimize` that takes three arguments: `theta`, `grad`, and an optional learning rate `lr` (defaulting to 0.03 if not provided).
    - `theta` represents the weight vector, which is the set of model parameters that will be updated.
    - `grad` represents the gradients of the loss function with respect to `theta`. These gradients guide the optimization process by indicating how to adjust the model parameters.
    - `lr` is the learning rate, which determines the step size or the rate at which the weights are updated during optimization.
2. `theta -= grad * lr`: This line updates the weight vector `theta` based on the gradients and the learning rate.

- **`theta -= grad * lr`** subtracts the product of **`grad`** (gradients) and **`lr`** (learning rate) from the **`theta`** vector. This is a common operation in optimization algorithms, such as gradient descent.
- The learning rate controls the size of the update step. A smaller learning rate results in smaller updates and potentially slower convergence, while a larger learning rate can lead to faster updates but may risk overshooting the optimal solution.

3. **`return theta`**: This line returns the updated weight vector **`theta`** after the optimization step.

Training

#Genrate training data

```
theta = np.random.uniform(-1, 1, (2 * context_size * embed_dim,
vocab_size))
```

```
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
```

The provided code snippet appears to be part of a training loop for a neural network-based model. It involves initializing a weight matrix (**`theta`**) with random values and then iteratively training the model for multiple epochs.

1. **`theta = np.random.uniform(-1, 1, (2 * context_size * embed_dim, vocab_size))`**:
    - This line initializes a weight matrix **`theta`** with random values.
    - The **`np.random.uniform(-1, 1, (2 * context_size * embed_dim, vocab_size))`** function creates a NumPy array of random values.
    - The shape of the matrix is **`(2 * context_size * embed_dim, vocab_size)`**, which suggests that it's a weight matrix used for a neural network. The shape of the matrix corresponds to the number of weights (parameters) required for the model.
2. **`epoch_losses = {}`**:
    - This line initializes an empty dictionary **`epoch_losses`**. It is likely intended to store the losses for each epoch during training.
3. The following loop runs for a total of 80 epochs:

pythonCopy code
```python
for epoch in range(80):
```

4. **`losses = []`**:
    - At the start of each epoch, an empty list **`losses`** is created to store the losses calculated during each iteration of the training loop.
5. The code then seems to be iterating over your training data. Assuming the data is a list of context-target pairs:

pythonCopy code
```python
for context, target in data:
```

6. The **`context`** is represented as a list of words, and the code is converting these words to their corresponding indices (e.g., integers representing words in the vocabulary):

pythonCopy code
```python
context_idxs = np.array([word_to_ix[w] for w in context])
```

    - It appears that **`word_to_ix`** is a dictionary that maps words to their corresponding integer indices in the vocabulary. This line creates an array **`context_idxs`** containing the indices of the words in the **`context`**.
7. **`preds = forward(context_idxs, theta)`**:
    - The **`forward`** function is used to make predictions based on the context indices and the weight matrix **`theta`**. The result, **`preds`**, is likely a set of predicted probabilities or scores for each word in the vocabulary.
8. The **`target`** is also a word, which is then converted to its corresponding index:

pythonCopy code
```python
target_idxs = np.array([word_to_ix[target]])
```

    - This line creates an array **`target_idxs`** containing the index of the **`target`** word.
9. **`loss = NLLLoss(preds[-1], target_idxs)`**:

- The code calculates a loss using the `NLLLoss` function, which appears to be the negative log-likelihood loss (cross-entropy) between the predicted probabilities (in `preds[-1]`) and the target index.

10. The calculated loss is appended to the `losses` list for this epoch:

pythonCopy code

```
losses.append(loss)
```

11. The loop iterates through all the training data pairs, and `losses` accumulates the loss for each pair.
12. It's not shown in the code snippet, but you would typically update the model's weights (`theta`) using an optimization algorithm like gradient descent based on these calculated losses in a training loop.

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
```

The provided code defines a Python function named `predict` that takes a list of words as input and uses a trained model to predict a single word based on the given context words. Here's a step-by-step explanation of the code:

1. `def predict(words):`: This line defines a function named `predict` that takes a single argument, `words`, which is expected to be a list of words.
2. `context_idxs = np.array([word_to_ix[w] for w in words])`:
   - This line creates an array `context_idxs` by mapping each word in the `words` list to its corresponding integer index using the `word_to_ix` dictionary.
   - `word_to_ix` is assumed to be a dictionary that maps words to their integer indices. This mapping is typically created during the preprocessing step, where each unique word is assigned a unique index.
3. `preds = forward(context_idxs, theta)`:
   - The `forward` function is used to make predictions based on the context indices `context_idxs` and the weight matrix `theta`.
   - `preds` likely represents the model's predictions for the next word, given the provided context words. The output is typically a set of probabilities or scores for each word in the vocabulary.
4. `word = ix_to_word[np.argmax(preds[-1])]`:

- This line selects the word with the highest probability or score from the `preds[-1]` array, which is the output of the model's prediction.
- `np.argmax(preds[-1])` returns the index of the word with the highest probability.
- `ix_to_word` is assumed to be a dictionary that maps integer indices back to their corresponding words.

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

The `predict` function, when called with the list of context words `['we', 'are', 'to', 'study']`, uses a trained model to predict the next word based on this context. Here's a step-by-step explanation of the prediction process for this particular input:

1. `context_idxs = np.array([word_to_ix[w] for w in words])`:
   - The input list of words `['we', 'are', 'to', 'study']` is mapped to their corresponding integer indices using the `word_to_ix` dictionary. This creates an array of context indices. For example, if 'we' has an index of 3, 'are' has an index of 5, 'to' has an index of 10, and 'study' has an index of 15, the resulting `context_idxs` might be `[3, 5, 10, 15]`.
2. `preds = forward(context_idxs, theta)`:
   - The `forward` function is used to make predictions based on the context indices and the weight matrix `theta`. It calculates the model's prediction for the next word based on the provided context.
3. `np.argmax(preds[-1])`:
   - This part selects the word with the highest predicted probability from the `preds[-1]` array. The `np.argmax` function finds the index of the element with the maximum value in the array. For example, if the highest probability corresponds to an index of 7, this indicates that the predicted word is the one associated with index 7.
4. `word = ix_to_word[np.argmax(preds[-1])]`:
   - The index obtained in the previous step is mapped back to the corresponding word using the `ix_to_word` dictionary. This retrieves the actual word associated with the highest predicted probability.
5. Finally, the function returns the predicted word:
   - In this case, the function would return the word that the model predicts to come after the context "we are to study."

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

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

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

The `accuracy` function is designed to calculate the accuracy of a language model, particularly its word prediction performance. Here's a step-by-step explanation of the function:

1. `def accuracy():`: This line defines a function named `accuracy` without any arguments.
2. `wrong = 0`: It initializes a variable `wrong` to zero. This variable is used to keep track of the number of incorrect predictions made by the language model.
3. The function iterates over a dataset named `data`, which appears to contain context and target word pairs:

   pythonCopy code

   ```
   for in
   ```

   - `context` represents the context words, and `target` is the actual target word that the model should predict.
4. For each context-target pair, the code calls the `predict` function to obtain the model's prediction for the next word based on the context:

   pythonCopy code

   ```
   if
   ```

   - If the predicted word does not match the actual target word, it means the model made an incorrect prediction. In this case, `wrong` is incremented by 1.
5. After processing all the context-target pairs in the dataset, the function calculates the accuracy:

   pythonCopy code

   ```
   return 1 len
   ```

   - The accuracy is calculated as `(1 - (wrong / len(data)))`. It represents the proportion of correct predictions made by the model. The `wrong` count is divided by the total number of context-target pairs in the dataset (`len(data)`), and this fraction is subtracted from 1 to find the accuracy.