5th:

CBOW: Continuous Bag of Words, CBOW, which stands for Continuous Bag of Words, is a type of neural network model used in natural language processing (NLP) and specifically in the context of word embeddings. Word embeddings are dense vector representations of words that capture semantic relationships between words based on their context.

In the case of CBOW, the model is designed to predict a target word based on its surrounding context.

The architecture of CBOW is considered simpler compared to other language models like Skip-gram.

Here's a basic overview of how CBOW works:

1. **Input:** CBOW takes a context of surrounding words as input. The context is represented as a set of one-hot encoded vectors, where each vector corresponds to a word in the context.
2. **Projection Layer:** The one-hot encoded vectors are then multiplied by an input weight matrix, resulting in a dense vector representation for each word in the context.
3. **Aggregation:** The dense vectors for the context words are usually averaged or summed to create a single vector that represents the context.
4. **Output:** This context vector is then used to predict the target word. The output layer is typically a softmax layer that assigns probabilities to each word in the vocabulary, and the target word is the one with the highest probability.

```
with open('5.text.txt', 'r') as file:
    sentences = file.read()
```

Code:

`re` module for regular expressions.

- sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences):
  In simpler terms, it removes special characters (anything that is not a letter or digit) from the `sentences` string and replaces them with a single space.
  It looks for one-letter words (single word characters `\w` surrounded by spaces or appearing at the beginning or end of the string) and replaces them with a single space.

  This is done to ensure uniformity in the text, making it case-insensitive for further processing.

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

In simpler terms, it breaks the `sentences` string into a list of words. Each word becomes an element in the list assigned to the variable `words`.

- In Python, a set is an unordered collection of unique elements.
- By converting the list of words to a set, duplicate words are removed, and you're left with a set of unique words from the original `sentences` string.

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

`embed_dim` stands for embedding dimension. In the context of natural language processing and word embeddings, it often represents the size of the vector space in which words are embedded. A

A context size of 2 means that the model considers two words to the left and two words to the right of the target word.

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

In summary, these lines of code create two dictionaries: `word_to_ix` and `ix_to_word`. These dictionaries establish a bidirectional mapping between words and their respective indices in the vocabulary. This kind of mapping is essential for converting words to indices (for model input) and indices back to words (for interpreting model output).

```
# 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])
```

- `data` **Initialization:**
  - `data = []`: Initializes an empty list to store training examples.
- **Loop Over Words:**
  - `for i in range(2, len(words) - 2):`: Iterates over the indices of words in the `words` list, excluding the first two and last two words. This is to ensure there is enough context for the specified context window.
- **Forming Context-Target Pairs:**
  - `context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]`: Forms a context window of size 4 around the current word at index `i`.
  - `target = words[i]`: Sets the target word as the word at the current index `i`.
- **Appending to Data:**
  - `data.append((context, target))`: Appends a tuple `(context, target)` to the `data` list, representing a training example.
- **Printing First 5 Examples:**
  - `print(data[:5])`: Prints the first 5 examples in the dataset.

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

- This line creates a NumPy array named `embeddings`.
- The array is initialized with random samples from a uniform distribution over the interval [0.0, 1.0).
- The shape of the array is determined by `(vocab_size, embed_dim)`. This means each unique word in the vocabulary (`vocab`) is associated with a vector of length `embed_dim`.
- The purpose of these embeddings is to represent words in a continuous vector space. These initial random values will be updated during the training of a neural network

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

This function essentially represents a linear transformation, where `m` is the input data and `theta` is the weight vector applied to the input data. The result is the linear combination of the input features with corresponding weights.

The function returns the result of the dot product of the input matrix `m` and the weight vector `theta`.

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

`x`: A NumPy array representing the input values.

The function begins by calculating the exponentials of the input array `x` after subtracting the maximum value in `x` to improve numerical stability. This is a common practice to prevent large exponentials that could lead to numerical overflow.

`NLLLoss` (Negative Log Likelihood Loss):

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

The negative log likelihood loss is computed by taking the sum of the extracted log probabilities and negating it. The result is then divided by the number of instances to get the average loss.

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]
```

- **Parameters:**
  - `logits`: A NumPy array containing the raw scores (logits) for each class. It's assumed to be a 2D array, where each row corresponds to a set of logits for a particular instance, and each column corresponds to a class.
  - `target`: A NumPy array containing the target class labels for each instance.

Explanation:

- The function starts by creating a one-hot encoded array (`out`) based on the target labels. This is done using NumPy's advanced indexing. The element at position `[i, target[i]]` is set to 1 for each instance `i`.
  - `out[np.arange(len(logits)), target] = 1`: Creates a one-hot encoded array for the target labels.
- The softmax function is then calculated by exponentiating the logits and normalizing the result across classes.
  - `softmax = np.exp(logits) / np.exp(logits).sum(axis=-1, keepdims=True)`: Computes the softmax probabilities for each class.
- The gradient of the log softmax cross-entropy loss is computed and returned. The formula involves subtracting the one-hot encoded array from the softmax probabilities and then dividing by the number of instances.

- `return (-out + softmax) / logits.shape[0]`: Computes the gradient of the loss and returns the average over all instances.

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

```
return m, n, o
```

- **Parameters:**
  - `context_idxs`: A NumPy array representing the indices of context words.
  - `theta`: A parameter (possibly the weight vector) used in the linear transformation.

Explanation:

1. **Embedding Layer:**
   - `m = embeddings[context_idxs].reshape(1, -1)`: This line extracts the embeddings for the given context indices. It assumes that `embeddings` is a matrix where each row corresponds to the embedding vector of a word in the vocabulary. The result (`m`) is reshaped into a 1D array.
2. **Linear Transformation:**
   - `n = linear(m, theta)`: This line performs a linear transformation using the function `linear`. It applies the weight vector `theta` to the context embeddings (`m`).
3. **Log Softmax Activation:**
   - `o = log_softmax(n)`: This line applies the log softmax activation function to the linear output `n`.
4. **Return Statement:**
   - The function returns the intermediate results `m`, `n`, and the final output `o`.

```
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
```

Parameters:

- `preds`: A tuple containing the intermediate results from the forward pass, where `preds[0]` is `m`, `preds[1]` is `n`, and `preds[2]` is `o`.
- `theta`: A parameter vector (possibly the weight vector) used in the linear transformation.
- `target_idxs`: A NumPy array representing the target indices.

In summary, the `backward` function takes the intermediate results from the forward pass (`preds`), the parameter vector (`theta`), and the target indices (`target_idxs`). It computes the gradient of the loss with respect to the parameters using backpropagation. This gradient can be used in an optimization algorithm (e.g., gradient descent) to update the parameters during the training of a neural network.

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

I

Parameters:

- `theta`: The current model parameters (e.g., weight vector) that need to be updated.
- `grad`: The gradient of the loss with respect to the parameters.

- `lr`: Learning rate, a hyperparameter that determines the step size during optimization.

n summary, the `optimize` function performs a parameter update using the gradient descent algorithm. The learning rate (`lr`) determines the size of the step taken in the direction opposite to the gradient. This step is typically part of an iterative optimization process used to train machine learning models.

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

Explanation:

- `context_size` is likely the size of the context window.
- `embed_dim` is the dimensionality of the word embeddings.
- `vocab_size` is the size of the vocabulary.
- `2 * context_size * embed_dim` is the size of the parameter vector `theta` for the linear transformation in the model. It's common in neural network architectures to have a weight matrix for each input feature.

```
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
```

- **Variable Name:** `epoch_losses`
  - This dictionary will store the training losses for each epoch.
- **Training Loop:**
  - The outer loop runs for 80 epochs.
- **Inner Loop (Mini-batch):**
  - It iterates over the training data, where each `data` item is a tuple containing `context` and `target`.
  - `context_idxs` is a NumPy array containing the indices of the context words obtained from the `word_to_ix` dictionary.
  - `preds` is the result of the forward pass using the current `theta`.
  - `target_idxs` is a NumPy array containing the index of the target word.
  - `loss` is calculated using the negative log-likelihood loss.
  - The loss is appended to the `losses` list.
- **Backpropagation and Optimization:**
  - The gradients are calculated using the backward function.
  - The model parameters (`theta`) are updated using the `optimize` function and the computed gradient.
- **Epoch-wise Losses:**
  - The `losses` list for each epoch is stored in the `epoch_losses` dictionary.

```
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)
```

`plt.plot(ix, [epoch_losses[i][0] for i in ix])`**:**

- Plots a line chart where the x-axis represents the epochs (`ix`) and the y-axis represents the losses for the first item (index 0) in the `epoch_losses` dictionary for each epoch.
- This code visualizes how the loss changes over epochs during the training process. The x-axis represents the epochs, and the y-axis represents the corresponding loss values for the first item in the `epoch_losses` dictionary.

```
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
```

- **Convert Words to Indices:**
  - `context_idxs = np.array([word_to_ix[w] for w in words])`: Converts the input words to their corresponding indices using the `word_to_ix` dictionary.
- **Forward Pass:**
  - `preds = forward(context_idxs, theta)`: Performs a forward pass through the trained model using the context indices and the learned parameters (`theta`).
- **Prediction:**
  - `word = ix_to_word[np.argmax(preds[-1])]`: Finds the index of the word with the highest probability in the output predictions (`preds[-1]`). Then, it uses `ix_to_word` to get the actual word associated with that index.
- **Return Prediction:**
  - Returns the predicted word.

```
def accuracy():
    wrong = 0

    for context, target in data:
        if(predict(context) != target):
```

```
            wrong += 1

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

- **Initialization:**
  - `wrong = 0`: Initializes a counter for the number of incorrect predictions.
- **Iterating Over Data:**
  - The function iterates through each `(context, target)` pair in the provided `data`.
  - `context` is a list of words, and `target` is the actual target word.
- **Prediction and Comparison:**
  - `if predict(context) != target:`: Calls the `predict` function for the given context and compares the predicted word with the actual target word. If they are not equal, it increments the `wrong` counter.
- **Accuracy Calculation:**
  - `return 1 - (wrong / len(data))`: Calculates the accuracy by subtracting the fraction of incorrect predictions from 1. This gives the ratio of correct predictions to the total number of predictions.