EMBEDDED LEARNING

SESSION 4: PREDICTION OF NEXT WORD USING LSTM

28/01/2025

By:
Priyanka Saxena,
Assistant Professor, CSE, KMIT.

IMPORTING LIBRARIES

```
# Import necessary libraries
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
```

Create an example dataset

First, you are creating a dictionary named `data`. This dictionary contains a key `"text"` and a list of movie review strings as its value.

```
# Step 1: Create an example dataset
data = {
    "text": [
        "The movie was fantastic and very engaging",
        "I hated the acting and the storyline",
        "It was boring and lacked depth",
        "Amazing performance by the actors and great direction",
        "Not worth watching at all",
        "One of the best movies I have ever seen"
    ]
}
```

Create a DataFrame from the dataset

```
# Create a DataFrame from the dataset
df = pd.DataFrame(data)
```

TOKENIZATION & LOWER CASING

```
# Split the text into words and lowercase each word
def tokenize(text):
    return [word.lower() for word in text.split()]
```

tokenize function: This function takes a string `text` as input.

text.split(): This splits the string into a list of words based on spaces.

word.lower(): This converts each word in the list to lowercase.

[word.lower() for word in text.split()]: This is a list comprehension that processes each word in the text, converting it to lowercase, and returns a list of lowercase words.

```
df["tokens"] = df["text"].apply(tokenize)
print(df)
```

```
                                                text  \
0            The movie was fantastic and very engaging
1                  I hated the acting and the storyline
2                    It was boring and lacked depth
3  Amazing performance by the actors and the di...
4                        Not worth watching at all
5                    One of the best movies I have seen

                                              tokens
0  [the, movie, was, fantastic, and, very, engaging]
1       [i, hated, the, acting, and, the, storyline]
2                 [it, was, boring, and, lacked, depth]
3  [amazing, performance, by, the, actors, and, g...
4                      [not, worth, watching, at, all]
5  [one, of, the, best, movies, i, have, ever, seen]
```

After running this code the Data frame will have an additional column “tokens”. Which contains the list of words , all in lower cases.

Create word-to-vector mappings manually

```
def create_vocab_and_vectors(tokens_list, vector_size=50):
```

Function Definition:

- `def create_vocab_and_vectors(tokens_list, vector_size=50):` defines a

Initialize Vocabulary:

- `vocab = {}` initializes an empty dictionary to store the vocabulary and their corresponding vectors.

```
            # Assign a random vector of the given size to each unique word
            vocab[token] = np.random.rand(vector_size)
    return vocab
```

Kmit

Iterate Through Tokens:

- `for tokens in tokens_list:` iterates over each list of tokens in the tokens list.
- `for token in tokens:` iterates over each token in the current list of tokens.

```
    for tokens in tokens_list:
        for token in tokens:
            if token not in vocab:
                # Assign a random vector of the given size to each unique word
                vocab[token] = np.random.rand(vector_size)
    return vocab
```

Assign Random Vectors:

- `if token not in vocab:` checks if the token is not already in the vocabulary.
- `vocab[token] = np.random.rand(vector_size)` assigns a random vector of the specified size (default 50) to each unique token using NumPy's `random.rand` function.

Return Vocabulary:

- `return vocab` returns the dictionary containing the vocabulary and their corresponding random vectors.

```
unique word
return vocab
```

Create vocabulary and word embeddings

```
# Create vocabulary and word embeddings
vocab = create_vocab_and_vectors(df['tokens'])
vector_size = 50  # Embedding size
```

- **Create Vocabulary:** `vocab = create_vocab_and_vectors(df['tokens'])` calls the `create_vocab_and_vectors` function with the tokenized text column from the DataFrame to create the vocabulary and their corresponding embeddings.
- **Embedding Size:** `vector_size = 50` specifies the size of the embedding vectors, which is set to 50 in this case.

Create vocabulary and word embeddings

```
# Create vocabulary and word embeddings
vocab = create_vocab_and_vectors(df['tokens'])
vector_size = 50  # Embedding size
```

The `vocab` dictionary now contains each unique word from the tokenized text as keys and their corresponding random vectors as values.

```
print(vocab)
```

```
{'the': array([0.38744321, 0.8916052 , 0.73167899, 0.54742179, 0.43081383,
       0.78248937, 0.89516262, 0.87534074, 0.76851172, 0.07263612,
       0.07478732, 0.86357795, 0.38115607, 0.9675381 , 0.92081921,
       0.26048363, 0.94358368, 0.12329314, 0.7712002 , 0.57128162,
       0.10319807, 0.98963232, 0.10342938, 0.30967625, 0.59590938,
       0.87577642, 0.37442955, 0.18888726, 0.39949943, 0.69824547,
       0.89737771, 0.54550179, 0.82162166, 0.41584817, 0.49153553,
       0.01037213, 0.6921251 , 0.32644233, 0.07371048, 0.59023089,
       0.64564328, 0.03695809, 0.189198  , 0.84692931, 0.46415689,
       0.72411348, 0.97117402, 0.2874406 , 0.2417614 , 0.84319522]), 'movie': array([0.58398734, 0.13170456, 0.77987123, 0.84089   , 0.04157031,
       0.13865874, 0.22886793, 0.8792954 , 0.4755909 , 0.74436017,
       0.07761087, 0.41084256, 0.56916204, 0.37096508, 0.19006213,
       0.0179009 , 0.76413168, 0.30742548, 0.67151398, 0.39996839,
       0.16601247, 0.53093101, 0.89713327, 0.4982774 , 0.32376246,
       0.38985699, 0.30480152, 0.21470051, 0.51975183, 0.9243313 ,
       0.00860521, 0.44649258, 0.07269069, 0.23644515, 0.42535753,
       0.98871004, 0.26429927, 0.94150525, 0.08149767, 0.85244813,
       0.7125823 , 0.17787011, 0.99831109, 0.18236983, 0.33226898,
       0.45305397, 0.94907712, 0.83192389, 0.71794518, 0.849265  ]), 'was': array([0.27019659, 0.8230212 , 0.32321761, 0.53044306, 0.5177153 ,
       0.81737606, 0.16981987, 0.66064419, 0.15310685, 0.79375485,
       0.87444412, 0.86623506, 0.31580771, 0.67720332, 0.66030564,
       0.87507215, 0.75086498, 0.16366295, 0.28237115, 0.10484656,
       0.42665788, 0.3605103 , 0.87611387, 0.88515801, 0.98711625,
       0.98792058, 0.56200789, 0.37241493, 0.44990552, 0.63811508,
       0.52136186, 0.47048701, 0.53803991, 0.809388  , 0.31541634,
       0.68997428, 0.12993946, 0.08581267, 0.41141571, 0.880777  ,
       0.22823402, 0.59414431, 0.37864815, 0.61583449, 0.77768928,
```

Prepare dataset for word prediction

```
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
    # Target word
    target = tokens[i + context_size]
    X.append([vocab[word] for word in context])  # Convert words to vectors
    y.append(list(vocab.keys()).index(target))  # Use the word's index in the vocabulary
return np.array(X), np.array(y)
```

Function Definition:

- `def prepare_sequences(tokens, vocab, context_size=3):` defines a function that takes a list of tokens, a vocabulary dictionary (`vocab`), and an optional context size (`context_size`) with a default value of 3.

```python
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
    X, y = [], []
    for i in range(len(tokens) - context_size):
        # Context words
        context = tokens[i:i + context_size]
        # Target word
        target = tokens[i + context_size]
        X.append([vocab[word] for word in context])  # Convert words to vectors
        y.append(list(vocab.keys()).index(target))  # Use the word's index in the vocabulary
    return np.array(X), np.array(y)
```

- “context_size” is a parameter called “ Context window”
- It refers to the number of surrounding words used to predict a target word in a word prediction model.
- Essentially, it determines how many previous words you consider as context to predict the next word.

```
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
```

For `context_size = 3`:

Suppose you have a sentence with the following tokens (after tokenizing):

```
["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
```

If you're trying to predict the word "fox", your context would be the previous 3 words (before "fox").

- **Context for "fox"**: `["The", "quick", "brown"]` (3 words before "fox")
- **Target word**: `"fox"` (word we are predicting)

Similarly, if you're trying to predict the next word after "jumps", the context will be:

- **Context for "jumps"**: `["fox", "brown", "quick"]`
- **Target word**: `"jumps"`

```
...e vocabulary
```

```
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
```

You slide this context window across the entire sentence to create multiple input-output pairs for training the model. For each word in the sentence, you generate:

- A **context** (the previous `context_size` words), and
- A **target** (the word to predict, which comes right after the context).

So, for the sentence above, with a `context_size` of 3, you would get pairs like:

- **Input**: `["The", "quick", "brown"]` → **Target**: `"fox"`
- **Input**: `["quick", "brown", "fox"]` → **Target**: `"jumps"`
- **Input**: `["brown", "fox", "jumps"]` → **Target**: `"over"`
- ... and so on.

```python
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
    X.append([vocab[word] for word in context])  # Convert words to vectors
    y.append(list(vocab.keys()).index(target))  # Use the word's index in the vocabulary
return np.array(X), np.array(y)
```

The choice of context size can affect the performance of your word prediction model:

- A **larger context** might capture more information but could be computationally expensive.
- A **smaller context** might be more focused but may miss broader sentence-level relationships.

```
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
```

- It creates input-output pairs where the context is the words around a target word.
- For each sequence of words in `tokens`, the context is formed by taking `context_size` words before the target word.
- The function returns two lists, `X` and `y`, where:
  - `X` contains the context words converted to their corresponding vector representations (from the `vocab` dictionary).
  - `y` contains the index of the target word in the vocabulary.

ocabulary

```
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
    X, y = [], []
```

`X, y` creation:

- After defining `prepare_sequences`, you iterate over all rows in `df['tokens']` to generate the context-target pairs for each sentence.
- The resulting sequences are appended to the `X` and `y` lists, which are then converted into numpy arrays for efficient processing during training.

```
vectors
ex in the vocabulary
    return np.array(X), np.array(y)
```

```
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
```

Steps in the loop:

- For each word in the `tokens` list (except for the last `context_size` words, because there's no target word after them), the function:
  - Selects a **context** window of size `context_size` (i.e., the words before the current word).
  - The **target** word is the word that immediately follows this context.
- The context words are converted into vectors using the `vocab` dictionary, where each word is mapped to its corresponding index.
- The target word is converted into its index using the vocabulary dictionary (`vocab`).

e vocabulary

Extract Context and Target Words:

- `context = tokens[i:i + context_size]` extracts a list of `context_size` tokens starting at index `i`.
- `target = tokens[i + context_size]` selects the token immediately following the context as the target word.

```
        # Context words
        context = tokens[i:i + context_size]
        # Target word
        target = tokens[i + context_size]
        X.append([vocab[word] for word in context])  # Convert words to vectors
        y.append(list(vocab.keys()).index(target))  # Use the word's index in the vocabulary
    return np.array(X), np.array(y)
```

```
# Step 4: Prepare dataset for word prediction
def
```

Convert to Vectors and Indices:

- `X.append([vocab[word] for word in context])` converts the context words to their corresponding vectors using the vocabulary and appends the list of vectors to `X`.
- `y.append(list(vocab.keys()).index(target))` finds the index of the target word in the vocabulary and appends it to `y`.

```
        X.append([vocab[word] for word in context])  # Convert words to vectors
        y.append(list(vocab.keys()).index(target))  # Use the word's index in the vocabulary
    return np.array(X), np.array(y)
```

```python
# Step 4: Prepare dataset for word prediction
def prepare_sequences(tokens, vocab, context_size=3):
    X, y = [], []
    for i in range(len(tokens) - context_size):
        # Context words
        context = tokens[i:i + context_size]
        # Target word
        rds to vectors
        d's index in the vocabulary
    return np.array(X), np.array(y)
```

Return Arrays:

- `return np.array(X), np.array(y)` converts the lists `X` and `y` to NumPy arrays and returns them.

Let's apply the function that we created....

```
# Create sequences for all rows in the dataset
context_size = 3
X, y = [], []
for tokens in df['
    X_seq, y_seq =
    X.extend(X_seq
    y.extend(y_seq

X, y = np.array(X), np.array(y)
```

Process Each Token List:

- Iterate through each list of tokens in the DataFrame column `df['tokens']`.
- For each list of tokens, call `prepare_sequences` to get the context (`X_seq`) and target (`y_seq`) sequences.
- Extend the main `X` and `y` lists with the sequences obtained for each row.

When we call the `prepare_sequences` function for each list of tokens, it returns two lists: `X_seq` (context sequences) and `y_seq` (target sequences). These lists contain multiple sequences of context words and their corresponding target words.

- If we use `append`, it would add the entire list `X_seq` as a single element in `X`, and `y_seq` as a single element in `y`. This would result in nested lists, which is not what we want for further processing.
- When we use `extend`, it adds each element of `X_seq` and `y_seq` to the respective `X` and `y` lists individually. This way, we maintain a flat structure for `X` and `y`, which is necessary for them to be converted to NumPy arrays and used in the word prediction model.

```python
# Example with append
list1 = [1, 2, 3]
list2 = [4, 5]
list1.append(list2)
# list1 is now: [1, 2, 3, [4, 5]]

# Example with extend
list1 = [1, 2, 3]
list2 = [4, 5]
list1.extend(list2)
# list1 is now: [1, 2, 3, 4, 5]
```

```
# Create sequences for all rows in the dataset
context_size = 3
X, y = [], []
for tokens in df['tokens']:
    X_seq, y_seq = prepare_sequences(tokens, vocab, context_size)
    X.extend(X_seq)
    y.extend(y_seq)

X, y = np.array(X), np.array(y)
```

Return:

- It returns the prepared sequences as two numpy arrays:
  - `x` : The context words represented as vectors.
  - `y` : The index of the target word.

Simple example to understand what we wrote:

```
tokens = ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
vocab = {
    "the": 0,
    "quick": 1,
    "brown": 2,
    "fox": 3,
    "jumps": 4,
    "over": 5,
    "lazy": 6,
    "dog": 7
}
context_size = 3
```

Iteration Over the Tokens:

For `tokens = ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]`, and `context_size = 3`, here's how the function works:

- For `i = 0`: The context is `["the", "quick", "brown"]` and the target is `"fox"`.
  - `X[0] = [vocab["the"], vocab["quick"], vocab["brown"]] = [0, 1, 2]`
  - `y[0] = vocab["fox"] = 3`
- For `i = 1`: The context is `["quick", "brown", "fox"]` and the target is `"jumps"`.
  - `X[1] = [vocab["quick"], vocab["brown"], vocab["fox"]] = [1, 2, 3]`
  - `y[1] = vocab["jumps"] = 4`
- For `i = 2`: The context is `["brown", "fox", "jumps"]` and the target is `"over"`.
  - `X[2] = [vocab["brown"], vocab["fox"], vocab["jumps"]] = [2, 3, 4]`
  - `y[2] = vocab["over"] = 5`
- Continue this process for each word until `i = len(tokens) - context_size`.

This would be the result:

```
X = [
    [0, 1, 2],  # context for "fox"
    [1, 2, 3],  # context for "jumps"
    [2, 3, 4],  # context for "over"
    [3, 4, 5],  # context for "the"
    [4, 5, 6],  # context for "lazy"
    [5, 6, 7]   # context for "dog"
]

y = [3, 4, 5, 6, 7, 0]  # target words: ["fox", "jumps", "over", "the", "lazy", "dog"]
```

Result in our actual code that we wrote:

```
print(X)
```

```
[[[0.38744321 0.8916052  0.73167899 ... 0.2874406  0.2417614  0.84319522]
  [0.58398734 0.13170456 0.77987123 ... 0.83192389 0.71794518 0.849265  ]
  [0.27019659 0.8230212  0.32321761 ... 0.55934773 0.45816191 0.05528316]]

 [[0.58398734 0.13170456 0.77987123 ... 0.83192389 0.71794518 0.849265  ]
  [0.27019659 0.8230212  0.32321761 ... 0.55934773 0.45816191 0.05528316]
  [0.90852709 0.51320504 0.0877475  ... 0.66407392 0.98703229 0.18741184]]

 [[0.27019659 0.8230212  0.32321761 ... 0.55934773 0.45816191 0.05528316]
  [0.90852709 0.51320504 0.0877475  ... 0.66407392 0.98703229 0.18741184]
  [0.98479322 0.90880119 0.86176678 ... 0.52348119 0.0283327  0.05441004]]

 ...

 [[0.37424785 0.220555   0.50302398 ... 0.32883818 0.80114232 0.54776564]
  [0.60468989 0.0262119  0.4997348  ... 0.13859337 0.48241074 0.21721113]
  [0.56816289 0.44098638 0.96420841 ... 0.03211344 0.75771514 0.165077  ]]

 [[0.60468989 0.0262119  0.4997348  ... 0.13859337 0.48241074 0.21721113]
  [0.56816289 0.44098638 0.96420841 ... 0.03211344 0.75771514 0.165077  ]
  [0.91242135 0.64459492 0.63194834 ... 0.01726223 0.25234499 0.08415864]]

 [[0.56816289 0.44098638 0.96420841 ... 0.03211344 0.75771514 0.165077  ]
  [0.91242135 0.64459492 0.63194834 ... 0.01726223 0.25234499 0.08415864]
  [0.71577398 0.90226848 0.45414043 ... 0.699232   0.77734966 0.43205818]]]
```

Result in our actual code that we wrote:

```
print(y)
```

```
[ 3  4  5  6  9  4  0 10  4 13 14  0 18  4 19 20 24 25 28 29  7 30 31 32]
```

Next step is to convert data into tensors

```
# Step 5: Convert data to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)
```

Why Convert to Tensors?

- **GPU Acceleration**: PyTorch tensors support computations on GPUs, which can significantly speed up training and inference.
- **Autograd**: PyTorch tensors come with built-in support for automatic differentiation, which is essential for training neural networks.
- **Compatibility**: Most PyTorch operations and models are designed to work with tensors, so converting data to this format ensures compatibility with the PyTorch ecosystem.

```
# Step 5: Convert data to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)
```

- **Input Features (`X`)**: Use `float32` for efficient and precise floating-point arithmetic required in neural network computations.
- **Target Labels (`y`)**: Use `long` (64-bit integer) for precise integer representation of categorical target labels, compatible with loss functions used in training.

Define the LSTM model for word prediction

```
# Step 6: Define the LSTM model for word prediction
class WordPredictionLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, vocab_size):
                                                                    ue)
        _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
        out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
        return out
```

Class Definition: `WordPredictionLSTM`

This class defines an LSTM-based neural network for word prediction.

`__init__` **Method**

The `__init__` method initializes the layers and parameters of the model.

```python
# Step 6: Define the LSTM model for word prediction
class WordPredictionLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, vocab_size):
        _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
        out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
        return out
```

- `input_size` : The size of the input features (typically the size of word embeddings).
- `hidden_size` : The size of the hidden state in the LSTM.
- `vocab_size` : The size of the vocabulary (number of unique words).

- Defines an LSTM layer that takes `input_size` as input and produces `hidden_size` as output.
- `batch_first=True` means that the input and output tensors are of shape `(batch_size, seq_length, features)`.

```
# Step

class W

    def

        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
        out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
        return out
```

```python
# Step 6: Define the LSTM model for word prediction
class WordPredictionLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, vocab_size):
        super(WordPredictionLSTM, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)
```

- Defines a fully connected (linear) layer that maps `hidden_size` to `vocab_size`. This layer will convert the hidden state from the LSTM to the predicted word indices.

```python
        return out
```

```python
# Step 6: Define the LSTM model for word prediction
class WordPredictionLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, vocab_size):
        super(WordPredictionLSTM, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
        out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
        return out
```

```
def forward(self, x):
    _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
    out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
    return out
```

`forward(self, x)`:

The input `x` represents a batch of sequences of word embeddings (or indices, depending on how you've prepared your data). The shape of `x` is typically `(batch_size, sequence_length, input_size)`, where:

- `batch_size` is the number of sequences in the batch.
- `sequence_length` is the number of tokens (words) in each sequence.
- `input_size` is the size of each word embedding (for example, 100-dimensional embeddings).

```
def forward(self, x):
    _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
    out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
    return out
```

- **LSTM Layer**: This line passes the input `x` (which is the batch of word embeddings) through the LSTM.
- The LSTM returns two outputs:
  - The first output (not captured here) is the output of the LSTM for each time step in the sequence, which has the shape `(batch_size, sequence_length, hidden_size)`.
  - The second output is a tuple containing the hidden state (`hidden`) and the cell state (not used here).

Kmit

```
def forward(self, x):
    _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
```

- `hidden` contains the hidden states of the LSTM at each time step for the **last layer** of the LSTM, and its shape is `(num_layers, batch_size, hidden_size)`. Since you're using a single-layer LSTM (`num_layers=1`), `hidden` will have the shape `(1, batch_size, hidden_size)`.

The underscore (`_`) is used to ignore the output that we're not interested in (the LSTM output for all time steps and the cell state).

```
def forward(self, x):
    _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
    out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
    return out
```

- `hidden[-1]` selects the last hidden state of the LSTM for each sequence in the batch. The reason we're using `hidden[-1]` is that this corresponds to the final time step of the sequence, which is typically considered to have captured the most relevant information about the entire sequence in sequence models like LSTM.
- `hidden[-1]` has the shape `(batch_size, hidden_size)`.

```
def forward(self, x):
    _, (hidden, _) = self.lstm(x)  # Get the hidden state from LSTM
    out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
    return out
```

- The selected final hidden state (`hidden[-1]`) is passed through a fully connected (linear) layer (`self.fc`). This layer maps the hidden state, which has a size of `hidden_size` , to a vector of size `vocab_size` . Each element of the output vector corresponds to the predicted logit (raw score) for a word in the vocabulary.
- The output `out` has the shape `(batch_size, vocab_size)` , where `vocab_size` is the size of the vocabulary (i.e., the number of unique words in your dataset).

```python
# Step 6: Define the LSTM model for word prediction
class WordPredictionLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, vocab_size):
        super(WordPredictionLSTM, self).__init__()
        self.lstm = nn.LSTM(inp
        self.fc = nn.Linear(hid

    def forward(self, x):
        _, (hidden, _) = self.l
        out = self.fc(hidden[-1])  # Pass hidden state through a fully connected layer
        return out
```

LSTM processes the input sequence, extracting hidden states at each time step.

Last hidden state is taken as the representation of the entire sequence.

Fully connected layer maps the last hidden state to the vocabulary space (output logits).

Initialize the Model

```
# Step 7: Initialize the model
input_size = vector_size  # Size of the word vector
hidden_size = 64  # Number of hidden units in LSTM
vocab_size = len(vocab)  # Vocabulary size
model = WordPredictionLSTM(input_size, hidden_size, vocab_size)
```

You're setting the necessary parameters for the LSTM model, such as the size of the word vectors, the hidden state size, and the vocabulary size.

After setting these parameters, you're initializing the `WordPredictionLSTM` model using these values, which prepares it for training.

Once the model is initialized, you would typically proceed with:

1. **Defining a loss function** (e.g., `CrossEntropyLoss`) to compare the predicted logits to the true target word.
2. **Choosing an optimizer** (e.g., `Adam`) to update the model's parameters during training.
3. **Feeding in batches of data** (input sequences and corresponding target words) to train the model.

Loss Function

The loss function measures how well the model's predictions match the actual target values. It quantifies the difference between the predicted output and the true output, and the goal during training is to minimize this loss.

```
# Step 8: Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
```

Why Cross-Entropy Loss:

- It is particularly suitable for classification tasks where the output is a probability distribution over several classes (words in this case). The aim is to predict the correct word out of the entire vocabulary.

Optimizer

The optimizer updates the model parameters based on the gradients of the loss function.

It controls how the model learns during training.

```
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

`optim.Adam` : Adam (short for Adaptive Moment Estimation) is an optimization algorithm that computes adaptive learning rates for each parameter.

Parameters:

- `model.parameters()` : This function returns an iterator over the model parameters (weights and biases) that need to be optimized.
- `lr=0.01` : Learning rate, which controls the step size during each iteration of parameter updates. A learning rate of 0.01 is relatively high and might be adjusted based on the performance during training.

Kmit

Why Adam Optimizer

- **Adaptive Learning Rates**: Adam adjusts the learning rates of individual parameters, making the optimization process more efficient and often leading to faster convergence.
- **Combines Advantages**: It incorporates both momentum (from SGD with momentum) and RMSProp, which helps in smoothing the optimization path and dealing with sparse gradients.
- **Default Choice**: Adam is a popular choice for training neural networks due to its robustness and efficiency.

Training loop

```python
# Step 9: Training loop
num_epochs = 30  # Number of epochs for training
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    optimizer.zero_grad()  # Reset gradients

    # Forward pass
    outputs = model(X_tensor)  # Get model predictions
    loss = criterion(outputs, y_tensor)  # Calculate loss

    # Backward pass and optimization
    loss.backward()  # Backpropagation
    optimizer.step()  # Update model parameters

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")
```

```python
# Step 9: Training loop
num_epochs = 30  # Number of epochs for training
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    optimizer.zero_grad()  # Reset gradients
```

`optimizer.zero_grad()` : This function clears the old gradients from the last step (otherwise they would accumulate). This ensures that the gradients are correctly computed for the current step.

```
# Forward pass
outputs = model(X_tensor)  # Get model predictions
loss = criterion(outputs, y_tensor)  # Calculate loss
```

- The model processes the input data `X_tensor` (which contains your training sequences) and performs the forward pass, producing **logits** for each word in the vocabulary (i.e., predictions for the next word).
- `outputs` will be a tensor of shape `(batch_size, vocab_size)` representing the predicted logits for each word in the vocabulary for each input sequence.

```
# Forward pass
outputs = model(X_tensor)  # Get model predictions
loss = criterion(outputs, y_tensor)  # Calculate loss
```

Loss function: The loss function (`criterion`) compares the model's predictions (`outputs`) with the true labels (`y_tensor`).

- `y_tensor` contains the indices of the true words (targets) for each sequence in the batch.
- Commonly, for word prediction tasks, **cross-entropy loss** is used, which is suitable for classification problems like this (predicting the next word).
- The loss will be a scalar value indicating how well the model's predictions match the true labels.

```
# Backward pass and optimization
loss.backward()  # Backpropagation
```

- This step computes the gradients of the loss with respect to each parameter in the model, i.e., it performs **backpropagation**.
- The gradients are used to adjust the weights in the model during optimization.

```
# Backward pass and optimization
loss.backward()  # Backpropagation
optimizer.step()  # Update model parameters
```

- After computing the gradients, the optimizer updates the model's parameters to minimize the loss.
- Common optimizers like **Adam** or **SGD** are used to adjust the parameters based on the gradients.

```
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")
```

```
Epoch 1/30, Loss: 3.4992
Epoch 2/30, Loss: 3.3169
Epoch 3/30, Loss: 3.1438
Epoch 4/30, Loss: 2.9934
Epoch 5/30, Loss: 2.8948
Epoch 6/30, Loss: 2.8075
Epoch 7/30, Loss: 2.7174
Epoch 8/30, Loss: 2.6248
Epoch 9/30, Loss: 2.5085
Epoch 10/30, Loss: 2.3784
Epoch 11/30, Loss: 2.2583
Epoch 12/30, Loss: 2.1176
Epoch 13/30, Loss: 1.9555
Epoch 14/30, Loss: 1.8029
Epoch 15/30, Loss: 1.6436
Epoch 16/30, Loss: 1.4912
Epoch 17/30, Loss: 1.3200
Epoch 18/30, Loss: 1.1661
Epoch 19/30, Loss: 1.0100
Epoch 20/30, Loss: 0.8708
Epoch 21/30, Loss: 0.7429
Epoch 22/30, Loss: 0.6342
Epoch 23/30, Loss: 0.5320
Epoch 24/30, Loss: 0.4449
Epoch 25/30, Loss: 0.3723
```

Prediction function

```python
# Step 10: Prediction function for next word
def predict_next_word(context, model, vocab, context_size=3):
    # Tokenize the input context
    tokens = tokenize(context)
    if len(tokens) < context_size:
        raise ValueError(f"Context must have at least {context_size} words")
    tokens = tokens[-context_size:]  # Use only the last `context_size` words

    # Convert tokens to vectors
    vectors = [vocab[word] for word in tokens]

    # Convert to PyTorch tensor and add batch dimension
    input_tensor = torch.tensor([vectors], dtype=torch.float32)  # Shape: (1, context_size, input_size)

    # Set the model to evaluation mode
    model.eval()

    # Make prediction
    with torch.no_grad():
        output = model(input_tensor)  # Get model predictions
        predicted_index = torch.argmax(output, dim=1).item()  # Get the index of the maximum value (class)

    # Convert index back to word
    predicted_word = list(vocab.keys())[predicted_index]
    return predicted_word
```

```python
# Step 10: Prediction function for next word
def predict_next_word(context, model, vocab, context_size=3):
    # Tokenize the input context
    tokens = tokenize(context)
    if len(tokens) < context_size:
        raise ValueError(f"Context must have at least {context_size} words")
    tokens = tokens[-context_size:]  # Use only the last `context_size` words
```

- `context` : The input context (a string of words) for which you want to predict the next word.
- `model` : The trained LSTM model used for prediction.
- `vocab` : The vocabulary dictionary mapping words to their respective indices.
- `context_size` : Number of words to consider as the context (default is 3).

```python
# Step 10: Prediction function for next word
def predict_next_word(context, model, vocab, context_size=3):
    # Tokenize the input context
    tokens = tokenize(context)
    if len(tokens) < context_size:
        raise ValueError(f"Context must have at least {context_size} words")
    tokens = tokens[-context_size:]  # Use only the last `context_size` words
```

- `tokenize(context)` : Converts the input context string into a list of tokens (words).
- **Check Context Length**: Raises an error if the number of tokens is less than the context size.
- **Extract Last Words**: Uses only the last `context_size` words from the tokens to ensure the context is the right length.

```
# Convert tokens to vectors
vectors = [vocab[word] for word in tokens]
```

- Converts each word in the `tokens` list to its corresponding index using the `vocab` dictionary.
- `vectors` : A list of indices representing the context words.

```
# Convert to PyTorch tensor and add batch dimension
input_tensor = torch.tensor([vectors], dtype=torch.float32)  # Shape: (1, context_size, input_size)
```

- The list of word vectors is converted into a PyTorch tensor. We add the extra batch dimension (hence the `[vectors]`), as PyTorch models typically expect the input to have a batch dimension, even if the batch size is just 1.
- The shape of this tensor will be `(1, context_size, input_size)`, where:
  - `1` is the batch size (1 sample),
  - `context_size` is the number of words used as context (e.g., 3),
  - `input_size` is the size of each word vector (e.g., 100, 300, etc.).

Evaluate the model

```
# Set the model to evaluation mode
model.eval()
```

Make predictions

```
# Make prediction
with torch.no_grad():
```

This tells PyTorch to **not track gradients** during the forward pass, which reduces memory usage and speeds up computations when making predictions, as no backpropagation is needed during inference.

```
# Make prediction
with torch.no_grad():
    output = model(input_tensor)  # Get model predictions
```

The input tensor is passed through the model, and the model generates the logits for each word in the vocabulary as predictions. The output will have shape `(1, vocab_size)` because the model produces a logit score for each word in the vocabulary.

```
# Make prediction
with torch.no_grad():
    output = model(input_tensor)  # Get model predictions
    predicted_index = torch.argmax(output, dim=1).item()  # Get the index of the maximum value (class)
```

- `torch.argmax(output, dim=1)` finds the index of the word with the highest logit (i.e., the word the model predicts to be the next word).
- `.item()` converts the index from a tensor to a Python integer.

```
# Convert index back to word
predicted_word = list(vocab.keys())[predicted_index]
return predicted_word
```

- This converts the predicted index back to the corresponding word by accessing the `vocab` dictionary (which maps words to their vector representations).
- `vocab.keys()` returns the list of words in the vocabulary, and `predicted_index` is used to fetch the correct word.

Interactive testing function:

```
# Interactive Testing Function
def interactive_predict(model, vocab, context_size=3):
    print("\nInteractive Word Prediction")
    print("Enter a context sentence to predict the next word.")
    print("Type 'exit' to quit.\n")

    while True:
        context = input("Enter context: ")
        if context.lower() == 'exit':
            print("Exiting interactive testing. Goodbye!")
            break

        try:
            next_word = predict_next_word(context, model, vocab, context_size)
            print(f"Predicted next word: \"{next_word}\"")
        except ValueError as e:
            print(f"Error: {e}. Ensure the context has at least {context_size} words.")
```

```python
# Interactive Testing Function
def interactive_predict(model, vocab, context_size=3):
    print("\nInteractive Word Prediction")
    print("Enter a context sentence to predict the next word.")
    print("Type 'exit' to quit.\n")

    while True:
        context = input("Enter context: ")
        if context.lower() == 'exit':
            print("Exiting interactive testing. Goodbye!")
            break
```

```
    try:
        next_word = predict_next_word(context, model, vocab, context_size)
        print(f"Predicted next word: \"{next_word}\"")
    except ValueError as e:
        print(f"Error: {e}. Ensure the context has at least {context_size} words.")
```

- The `predict_next_word` function is called to predict the next word based on the given context. If the context has at least `context_size` words, the model predicts the next word and prints it.

```
    try:
        next_word = predict_next_word(context, model, vocab, context_size)
        print(f"Predicted next word: \"{next_word}\"")
    except ValueError as e:
        print(f"Error: {e}. Ensure the context has at least {context_size} words.")
```

- If there is a `ValueError` (which could be raised if the context has fewer words than `context_size`), an error message is displayed.
- This ensures that the user knows the required context size to make a valid prediction.

Run interactive testing:

```
# Run interactive testing
interactive_predict(model, vocab)
```

```
Interactive Word Prediction
Enter a context sentence to predict the next word.
Type 'exit' to quit.

Enter context:
```

```
# Run interactive testing
interactive_predict(model, vocab)
```

```
Interactive Word Prediction
Enter a context sentence to predict the next word.
Type 'exit' to quit.

Enter context: the movie
Error: Context must have at least 3 words. Ensure the context has at least 3 words.
Enter context:
```

```
# Run interactive testing
interactive_predict(model, vocab)
```

```
Interactive Word Prediction
Enter a context sentence to predict the next word.
Type 'exit' to quit.

Enter context: the movie
Error: Context must have at least 3 words. Ensure the context has at
Enter context: the movie was
<ipython-input-14-affcdf6c7266>:13: UserWarning: Creating a tensor fr
  input_tensor = torch.tensor([vectors], dtype=torch.float32)  # Shap
Predicted next word: "fantastic"
Enter context:
```

```
# Run interactive testing
interactive_predict(model, vocab)
```

```
Interactive Word Prediction
Enter a context sentence to predict the next word.
Type 'exit' to quit.

Enter context: the movie
Error: Context must have at least 3 words. Ensure th
Enter context: the movie was
<ipython-input-14-affcdf6c7266>:13: UserWarning: Cre
  input_tensor = torch.tensor([vectors], dtype=torch
Predicted next word: "fantastic"
Enter context: the movie was fantastic and very
Predicted next word: "engaging"
Enter context:
```

```
# Run interactive testing
interactive_predict(model, vocab)
```

```
Interactive Word Prediction
Enter a context sentence to predict the next word.
Type 'exit' to quit.

Enter context: the movie
Error: Context must have at least 3 words. Ensure the context has at least 3 words.
Enter context: the movie was
<ipython-input-14-affcdf6c7266>:13: UserWarning: Creating a tensor from a list of nu
  input_tensor = torch.tensor([vectors], dtype=torch.float32)  # Shape: (1, context_
Predicted next word: "fantastic"
Enter context: the movie was fantastic and very
Predicted next word: "engaging"
Enter context: exit
Exiting interactive testing. Goodbye!
```

Its Time for quiz!!!