Lab 4: Skip-gram with Negative Sampling

Introduction

In this laboratory session, we embark on an exploration into the domain of word embeddings, with a particular focus on the Skip-gram model and Negative Sampling technique. This approach is pivotal in the field of natural language processing (NLP) as it provides a method for representing words in dense vector spaces, capturing semantic relationships and syntactic patterns.

You'll engage with the Skip-gram model with Negative Sampling, a cornerstone architecture introduced by Mikolov et al.[1] that aims to predict context words given a target word.

Throughout this lab, you will:

- Understand and implement the Skip-gram model architecture.
- Explore the concept and application of Negative Sampling to improve training efficiency.
- Train a Skip-gram model with Negative Sampling on a corpus of text data.
- Analyze and visualize the learned word embeddings to uncover semantic and syntactic relationships between words.

Learning Outcomes

Upon completion of this laboratory exercise, participants will have acquired the ability to:

- Convert textual data into numerical formats that are amenable to machine learning models, with a specific emphasis on the Skip-gram model for generating word embeddings.
- Grasp and implement the Skip-gram model architecture along with the technique of Negative Sampling, aimed at efficient training of word representations.
- Assess and critique the quality of word embeddings produced by the Skip-gram model, utilizing visualization techniques to explore semantic and syntactic relationships between words.

Deliverables

For the successful completion of this laboratory session, you are required to submit the following files:

1. `data_processing.py` - This script should encompass functionalities for loading the text data, preprocessing it to a suitable format for training, and generating batches of word pairs for the Skip-gram model using the custom batching function.

[1] Mikolov, Tomas, et al. 'Distributed representations of words and phrases and their compositionality.' Advances in neural information processing systems 26 (2013)

2. `skipgram.py` - This file is expected to define the SkipGramNeg class, encapsulating the architecture of the Skip-gram model with Negative Sampling. It should include methods for initializing the model, forward propagation to generate input and output vectors, and negative sampling for efficient training.

3. `train.py` - Within this script, you will implement the `train_skipgram` function, responsible for orchestrating the training process of the Skip-gram model, including optimization, loss calculation, and periodic evaluation of the model's performance and embedding quality.

4. `model.py` - Trained model with your chosen optimization. This would be used to represent the embeddings learned by your model.

5. `analysis.txt` - Text file including a comment about the learned relationships by your model. It shall not have more than 10 lines.

These files should be compressed into a single `.zip` archive and named with your student identifier. The `.zip` file must then be uploaded to the designated Moodle assignment task before the deadline.

Exercises

✔ Exercise 1: My name is Guy Incognito![1]

Your task is to implement the `load_and_preprocess_data` function, which takes a file path as input and returns a list of preprocessed and tokenized words from the text contained within the file.

1. Begin by opening the file located at the provided infile path.

2. Preprocess and tokenize the text of the file. We have provided you a customized `tokenize` function, but you are allowed to use one of your own.

3. Finally, return the list of tokens generated by the tokenize function.

✔ Exercise 2: Déjà vu

In this exercise, you are to develop the `create_lookup_tables` function that creates two essential dictionaries for NLP tasks: one mapping words to integers (`vocab_to_int`) and the other mapping integers back to words (`int_to_vocab`). Here's your brief:

1. Calculate the frequency of each word in the input list words.

2. Sort the vocabulary by frequency, from the most to the least common.

3. Generate `vocab_to_int` and `int_to_vocab` dictionaries for word-integer and integer-word mappings, respectively, using enumeration on the sorted vocabulary.

[1] The Simpsons - Guy Incognito

Exercise 3: Give Me the Short Version

Words that show up often such as "the", "of", and "for" don't provide much context to the nearby words. If we discard some of them, we can remove some of the noise from our data and in return get faster training and better representations. For example, while the Skip-gram model benefits from observing the co-occurrences of "France" and "Paris", it benefits much less from observing the frequent co-occurrences of "France" and "the", as nearly every word co-occurs frequently within a sentence with "the". This process is called **subsampling** by Mikolov. For each word w_i in the training set, we'll discard it with probability given by

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

where t is a threshold parameter and $f(w_i)$ is the frequency of word w_i in the total dataset. t is often set around 10^{-5}. We choose this subsampling formula because it aggressively subsamples words whose frequency is greater than t while preserving the ranking of the frequencies.

This exercise tasks you with defining the `subsample_words` function.

1. The function calculates the frequency of each word in a provided list, using the `vocab_to_int` dictionary for word-to-integer conversion.
2. It then determines a subsampling probability $P(w_i)$ for each word.
3. Go through word-integers and discard the integer i with probablility $P(w_i)$ shown above. Note that $P(w_i)$ is therefore the probability that the word w_i is discarded. Assign the subsampled words to a list.
4. The function returns two outputs: a list of subsampled words (the ones that were not discarded), represented as integers, and a dictionary of word frequencies.

Exercise 4: You are the Average of Your Closest Friends

Now that our data is subsampled, we need to get it into the proper form to pass it into our network. With the skip-gram architecture, for each word in the text, we want to define a surrounding context and grab all the words in a window around that word, with size C. We could use the same window size C all the time, and generate all (center word, context word) pairs by sliding that fixed-sized window through the text. However, often the window size is changed throughout the text, using a sampling strategy. From [Mikolov et al.](https://arxiv.org/pdf/1301.3781.pdf):

> Since the more distant words are usually less related to the current word than those close to it, we give less weight to the distant words by sampling less from those words in our training examples... If we choose $C = 5$, for each training word we will select randomly a number R in range $[1 : C]$, and then use R words from history and R words from the future of the current word as correct labels.

Implement a function `get_target` that receives a list of words, an index, and a window size, then returns a list of words in the window around the index. In Exercise 4, you'll implement the `get_target` function. The function requires a list of words (`words`), the index of the target word (`idx`), and the

maximum window size (`window_size`). Then, it randomly chooses the window size for each target word to generate training examples. Finally, returns a list of context words around the target word, excluding the target word itself.

Make sure to use the algorithm described above, where you chose a random number of words to from the window. For example, say, we have an input and we're interested in the idx=2 token, 741:

$$[5233, 58, 741, 10571, 27349, 0, 15067, 58112, 3580, 58, 10712]$$

Suppose we have set $C = 5$. We will randomly select a window size from 1 to 5. Suppose we randomly choose $R = 2$. Then `get_target` should return a list of four values:

$$[5233, 58, 10571, 27349]$$

Exercise 5: Group and Conquer

In Exercise 5, you're tasked with developing the `get_batches` function.This function prepares data by organizing input-target pairs, where each pair consists of a center word and its surrounding context words, determined by a variable window size.

- The core aim is to create a generator function that efficiently yields batches of word pairs for the Skip-gram training process. Each batch should adhere to a specified size (`batch_size`), ensuring consistency in training iterations.
- You must handle words encoded as integers, taking a list (`words`) as input alongside the `batch_size` and `window_size`. The window size dictates how far from the target word context words can be selected.
- For each target word in a batch, you'll identify context words within the dynamically chosen window size around it, forming pairs that serve as the training data for predicting the context given a target word.
- The function iterates through the entire list of words, segmenting it into batches and for each batch, generating the specified input-target pairs. This process ensures that every word in a batch is considered as a target word, with its corresponding context words identified based on the window size.

It's crucial to manage the edge cases, such as the beginning and end of the word list, where the window might extend beyond the available indices. Proper handling ensures that all words have a fair chance of being included as both input and context words.

Hint #1:

You have already seen the Pytorch's way to retrieve data when training a model using Pytorch's `Dataset` and `DataLoader`. Here we have planned the function to use the basic Python's `yield` function, but feel free to modify the functions to use full Pytorch (the test for the `get_batches` function would not be taken into account for your grade).

Exercise 6: The Cosine Chronicles, Word Embedding Edition

Here, you'll create a function that will help us observe our model as it learns. We're going to choose a few words. Then, we'll print out the closest words to them using the cosine similarity. You're tasked with implementing the `cosine_similarity` function. This function is used to evaluate the quality of word embeddings generated by your Skip-gram model. By computing the cosine similarity between selected words and all other words in the embedding space, you can intuitively assess which words are deemed similar by your model.

- Use a PyTorch Embedding module, which contains the word vectors learned by the Skip-gram model.
- Select a subset of words (`valid_size`) from within a specified range (`valid_window`) to serve as your basis for comparison. These words are randomly chosen to represent a diverse set of points in the embedding space, as evaluating this function for all the word embeddings would result in an excessive computational cost.
- Calculate the cosine similarity between the embeddings of the selected words and all other words in the vocabulary. The cosine similarity metric will help you understand the degree of similarity, with a value close to 1 indicating high similarity and a value close to -1 indicating high dissimilarity.

Remember, cosine similarity is given by:

$$\text{cosine\_similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

where:

- A and B are vectors.
- $A \cdot B$ denotes the dot product of vectors.
- $\|A\|$ and $\|B\|$ denote the Euclidean norms of vectors A and B, respectively

Exercise 7: The Negative Skipper: Navigating SkipGramNeg

The `SkipGramNeg` class implements a variant of the Skip-gram model that utilizes negative sampling.

Exercise 7.1: Setting the Stage

The `__init__` method initializes the SkipGramNeg model with the specified vocabulary size (n_vocab), embedding dimension (n_embed), and an optional noise distribution (noise_dist) for negative sampling. It establishes two embedding layers: `in_embed` for input (center) words and `out_embed` for output (context) words, both initialized to uniform distributions. The layer `in_embed` stores the center-word vectors $\mathbf{v}_w$ for each word w, and `out_embed` stores the context-word vector $\mathbf{u}_w$ for w.

Have a look at the documentation of PyTorch's Embedding layer to understand how it works. Also, here's a brief notebook illustrating how it works.

And if you need extra intuition, here's some runnable code:

```python
import torch
import torch.nn as nn

# Define the size of the vocabulary and the dimensionality of the embeddings
vocab_size = 100
embedding_dim = 10

# Normally, you'd create an instance of the Embedding layer as follows
embedding_layer = nn.Embedding(vocab_size, embedding_dim)
# The embedding layer would have 100 word vectors, each vector of size 10

# In this example, we'll create the embedding layer with
# the vector for the word with index 0 as [1, 2, ..., 10],
# the vector for the word with index 1 as [11, 12, ..., 20], etc
embedding_values = torch.arange(1, vocab_size * embedding_dim + 1).reshape(vocab_size,
↪  embedding_dim)
embedding_layer = nn.Embedding.from_pretrained(embedding_values)

# Define an input tensor with batch size 3 and sequence length 5
# Each row represents a list of word indices
input_tensor = torch.tensor([[1, 4, 6, 2, 9],
                             [2, 3, 5, 8, 7],
                             [9, 6, 1, 4, 3]])

# Pass the input tensor through the embedding layer
# It will return the corresponding word vectors
embedded_output = embedding_layer(input_tensor)

# The output will be a tensor of shape (batch_size, sequence_length, embedding_dim)
print("Output shape:", embedded_output.shape)
print("Actual Output:")
print(embedded_output)
```

```
Output shape: torch.Size([3, 5, 10])
Actual Output:
tensor([[[ 11,  12,  13,  14,  15,  16,  17,  18,  19,  20],
         [ 41,  42,  43,  44,  45,  46,  47,  48,  49,  50],
         [ 61,  62,  63,  64,  65,  66,  67,  68,  69,  70],
         [ 21,  22,  23,  24,  25,  26,  27,  28,  29,  30],
         [ 91,  92,  93,  94,  95,  96,  97,  98,  99, 100]],

        [[ 21,  22,  23,  24,  25,  26,  27,  28,  29,  30],
         [ 31,  32,  33,  34,  35,  36,  37,  38,  39,  40],
         [ 51,  52,  53,  54,  55,  56,  57,  58,  59,  60],
         [ 81,  82,  83,  84,  85,  86,  87,  88,  89,  90],
         [ 71,  72,  73,  74,  75,  76,  77,  78,  79,  80]],
```

```
[[ 91,  92,  93,  94,  95,  96,  97,  98,  99, 100],
 [ 61,  62,  63,  64,  65,  66,  67,  68,  69,  70],
 [ 11,  12,  13,  14,  15,  16,  17,  18,  19,  20],
 [ 41,  42,  43,  44,  45,  46,  47,  48,  49,  50],
 [ 31,  32,  33,  34,  35,  36,  37,  38,  39,  40]]])
```

Exercise 7.2: Into the Embedding Abyss

The `forward_input` function is responsible for retrieving the embedding vectors for a given batch of input center words. By passing integer-encoded words through the `in_embed` layer, it grabs the embedding vectors for corresponding center words. For example, if "heart" is encoded with the integer as 958, when you pass this numbers as input to `in_embed`, you get out the 958th row of the embedding matrix. This process is called an *embedding lookup*.

Exercise 7.3: Seeking Contextual Harmony

Similarly, the `forward_output` function fetches the embedding vectors for the context words using the `out_embed` layer. These vectors serve as positive samples in the training process, against which the model learns to distinguish the noise vectors generated for negative sampling.

Exercise 7.4: Embracing the Negative

The `forward_noise` method generates negative samples for each batch, which are also called noise words. Depending on whether a noise distribution is provided, it either samples uniformly across the vocabulary or according to the specified distribution. The selected noise words are then converted into embeddings, which the model learns to differentiate from the actual context words.

Exercise 8: You are Lost Again!

In this exercise, you will develop `NegativeSamplingLoss` class. It shall be a PyTorch module for calculating the loss used in training word embeddings with the Skip-gram model, particularly when employing negative sampling. This loss function is critical for distinguishing between actual context words and randomly selected noise words, thus refining the quality of the embeddings.

The `forward` method is where the computation of negative sampling loss occurs. It accepts three tensors as input: `input_vectors`, `output_vectors` (representing positive samples), and `noise_vectors` (representing negative samples). The function calculates the loss by applying the log-sigmoid function to the dot products of these vectors, adhering to the negative sampling strategy. Specifically, it:

- Reshapes `input_vectors` and `output_vectors` to facilitate batch matrix multiplication (`torch.bmm`).
- Calculates the loss for positive samples by taking the log-sigmoid of the dot product between input and output vectors.
- Computes the loss for negative samples in a similar manner, using the negated noise vectors.
- Aggregates the losses, summing across the negative samples and combining with the positive sample loss to produce the batch's average loss.

Exercise 9: Time to Go to the Gym

The `train_skipgram` function orchestrates the training process of the SkipGramNeg modelThis function is a comprehensive routine that manages data batching, model updates, and evaluation within each training epoch.

The specific steps are:

1. Initialization: Prepares the `NegativeSamplingLoss` criterion and the Adam optimizer with the specified learning rate.
2. Data Loading: Utilizes the `get_batches` function to generate batches of input-target pairs based on the `window_size`.
3. Batch Processing:
   (a) For each batch, converts word indices to tensors and moves them to the specified device.
   (b) Extracts input and output vectors using the model's forward methods and generates noise vectors for negative sampling.
   (c) Computes the loss using the `NegativeSamplingLoss` criterion.
4. Optimization: Performs backpropagation and updates the model parameters.
5. Evaluation: At specified intervals (`print_every`), the function evaluates the model by computing cosine similarities between selected validation words and all other words in the embedding space, demonstrating how the model perceives word similarities.

Exercise 10: I Want to Tell You my Secret... I See Word Embeddings

After training the model, you can check the embeddings it has learned using the `plot_embeddings` function. We have prepared this function for you, but you should use it (for example, in the `main()` function) to analyze the relationship the model has found between the word tensors. We do not expect you to analyze the whole picture, but we expect you to find at least one interesting relationship and comment about it.

Test your work!

After implementing each function, it's crucial to validate your work. We have prepared test functions for each required function, except the `train_skipgram` function. These tests will help ensure that your implementations meet the specified requirements. If a test function returns `None`, it indicates that the corresponding function might not be completed and the test is skipped. It is essential to pass all tests to ensure the functionality and accuracy of your code.

Additionally, a `main.py` file is provided. Previous to running this file, you must download the `text8` dataset from here, and extract the contents of the .zip file to the `./data` folder. Then, the `main.py` script runs the necessary functions to load the text data present in the `text8` file, preprocess it, create a new model, train it, and visualize the word embeddings learned by the model. It would save the model when the training step is finished, so you do not need to train it again if you want to check the embeddings again.