```python
# modules.skipgram.py
# https://github.com/QuantCS109/TrumpTweets/blob/master/modules/skipgram.py
# This is a .py file used in the notebooks

import numpy as np
import random
import torch
from torch import nn
from collections import Counter
from sklearn.manifold import TSNE
import matplotlib.cm as cm
import matplotlib.pyplot as plt

def subsampling(threshold, int_words):
    word_counts = Counter(int_words)
    total_count = len(int_words)
    freqs = {word: count/total_count for word, count in word_counts.items()}
    p_drop = {word: 1 - np.sqrt(threshold/freqs[word]) for word in word_counts}
    train_words = [word for word in int_words if random.random() < (1 - p_drop[word])]
    return freqs, train_words

def get_target(words, idx, window_size=5):
    ''' Get a list of words in a window around an index. '''

    r = np.random.randint(1, window_size + 1)
    start = idx - r if (idx - r) > 0 else 0
    stop = idx + r
    target_words = words[start:idx] + words[idx + 1:stop + 1]

    return list(target_words)


def get_batches(words, batch_size, window_size=5):
    ''' Create a generator of word batches as a tuple (inputs, targets) '''

    n_batches = len(words) // batch_size

    # only full batches
    words = words[:n_batches * batch_size]

    for idx in range(0, len(words), batch_size):
        x, y = [], []
        batch = words[idx:idx + batch_size]
        for ii in range(len(batch)):
            batch_x = batch[ii]
            batch_y = get_target(batch, ii, window_size)
            y.extend(batch_y)
            x.extend([batch_x] * len(batch_y))
        yield x, y


def cosine_similarity(embedding, valid_size=16, valid_window=100, device='cpu'):
    """ Returns the cosine similarity of validation words with words in the embedding matrix.
        Here, embedding should be a PyTorch embedding module.
    """

    embed_vectors = embedding.weight

    magnitudes = embed_vectors.pow(2).sum(dim=1).sqrt().unsqueeze(0)

    # pick N words from our ranges (0,window) and (1000,1000+window). lower id implies more frequent
    valid_examples = np.array(random.sample(range(valid_window), valid_size // 2))
    valid_examples = np.append(valid_examples,
                               random.sample(range(1000, 1000 + valid_window), valid_size // 2))
    valid_examples = torch.LongTensor(valid_examples).to(device)

    valid_vectors = embedding(valid_examples)
    similarities = torch.mm(valid_vectors, embed_vectors.t()) / magnitudes

    return valid_examples, similarities

def cosine_similarity_tweet(tweet_embedding, embedding,  device='cpu'):
    """ Returns the cosine similarity of validation words with words in the embedding matrix.
        Here, embedding should be a PyTorch embedding module.
    """

    embed_vectors = embedding.weight

    magnitudes = embed_vectors.pow(2).sum(dim=1).sqrt().unsqueeze(0)

    # pick N words from our ranges (0,window) and (1000,1000+window). lower id implies more frequent
    valid_examples = np.array(random.sample(range(valid_window), valid_size // 2))
    valid_examples = np.append(valid_examples,
                               random.sample(range(1000, 1000 + valid_window), valid_size // 2))
    valid_examples = torch.LongTensor(valid_examples).to(device)

    valid_vectors = embedding(valid_examples)
    similarities = torch.mm(valid_vectors, embed_vectors.t()) / magnitudes

    return valid_examples, similarities


class SkipGramNeg(nn.Module):
    def __init__(self, n_vocab, n_embed, noise_dist=None):
        super().__init__()

        self.n_vocab = n_vocab
        self.n_embed = n_embed
        self.noise_dist = noise_dist

        # define embedding layers for input and output words
        self.in_embed = nn.Embedding(n_vocab, n_embed)
        self.out_embed = nn.Embedding(n_vocab, n_embed)

        # Initialize embedding tables with uniform distribution
        # I believe this helps with convergence
        self.in_embed.weight.data.uniform_(-1, 1)
        self.out_embed.weight.data.uniform_(-1, 1)

    def forward_input(self, input_words):
        input_vectors = self.in_embed(input_words)
        return input_vectors

    def forward_output(self, output_words):
        output_vectors = self.out_embed(output_words)
        return output_vectors

    def forward_noise(self, batch_size, n_samples):
        """ Generate noise vectors with shape (batch_size, n_samples, n_embed)"""
        if self.noise_dist is None:
            # Sample words uniformly
            noise_dist = torch.ones(self.n_vocab)
        else:
            noise_dist = self.noise_dist

        # Sample words from our noise distribution
        noise_words = torch.multinomial(noise_dist,
                                        batch_size * n_samples,
                                        replacement=True)

        device = "cuda" if self.out_embed.weight.is_cuda else "cpu"
        noise_words = noise_words.to(device)

        noise_vectors = self.out_embed(noise_words).view(batch_size, n_samples, self.n_embed)

        return noise_vectors


class NegativeSamplingLoss(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, input_vectors, output_vectors, noise_vectors):
        batch_size, embed_size = input_vectors.shape

        # Input vectors should be a batch of column vectors
        input_vectors = input_vectors.view(batch_size, embed_size, 1)

        # Output vectors should be a batch of row vectors
        output_vectors = output_vectors.view(batch_size, 1, embed_size)

        # bmm = batch matrix multiplication
        # correct log-sigmoid loss
        out_loss = torch.bmm(output_vectors, input_vectors).sigmoid().log()
        out_loss = out_loss.squeeze()

        # incorrect log-sigmoid loss
        noise_loss = torch.bmm(noise_vectors.neg(), input_vectors).sigmoid().log()
        noise_loss = noise_loss.squeeze().sum(1)  # sum the losses over the sample of noise vectors

        # negate and sum correct and noisy log-sigmoid losses
        # return average batch loss
        return -(out_loss + noise_loss).mean()


def cosine_similarity_sample(embedding, val, device='cpu'):

    embed_vectors = embedding.weight

    # magnitude of embedding vectors, |b|
    magnitudes = embed_vectors.pow(2).sum(dim=1).sqrt().unsqueeze(0)

    valid_examples = torch.LongTensor(val).to(device)

    valid_vectors = embedding(valid_examples)
    similarities = torch.mm(valid_vectors, embed_vectors.t()) / magnitudes

    return valid_examples, similarities

def word_similarities(word, num, model,  vocab_to_int, int_to_vocab):
    word_int = vocab_to_int[word]
    valid_examples, valid_similarities = cosine_similarity_sample(model.in_embed, [word_int])
    closest_idxs = valid_similarities.topk(num)
    closest_words = [int_to_vocab[int(a)] for a in closest_idxs.indices[0]]
    return closest_words

# Code adapted from
# https://towardsdatascience.com/google-news-and-leo-tolstoy-visualizing-word2vec-word-embeddings-with-t-sne-11558d8bd4d
def get_clusters(keys, num, model, embeddings, vocab_to_int, int_to_vocab):
    embedding_clusters = []
```

```python
    word_clusters = []
    for word in keys:
        embed_sub = []
        words = []
        for similar_word in word_similarities(word, num, model, vocab_to_int, int_to_vocab):
            words.append(similar_word)
            embed_sub.append(embeddings[vocab_to_int[similar_word],:])
        embedding_clusters.append(embed_sub)
        word_clusters.append(words)
    return embedding_clusters, word_clusters

def tsne_plot_similar_words(title, labels, embedding_clusters, word_clusters, a, filename=None):
    plt.figure(figsize=(16, 9))
    colors = cm.rainbow(np.linspace(0, 1, len(labels)))
    for label, embeddings, words, color in zip(labels, embedding_clusters, word_clusters, colors):
        x = embeddings[:, 0]
        y = embeddings[:, 1]
        plt.scatter(x, y, c=color, alpha=a, label=label)
        for i, word in enumerate(words):
            plt.annotate(word, alpha=0.5, xy=(x[i], y[i]), xytext=(5, 2),
                         textcoords='offset points', ha='right', va='bottom', size=10)
    plt.legend(loc=4)
    plt.title(title)
    plt.grid(True)
    if filename:
        plt.savefig(filename, format='png', dpi=150, bbox_inches='tight')
    plt.show()

def plot_similar_words(keys, model, vocab_to_int, int_to_vocab, num=20, file=None):
    embeddings = model.in_embed.weight.to('cpu').data.numpy()
    embedding_clusters, word_clusters = get_clusters(keys, num, model, embeddings, vocab_to_int, int_to_vocab)
    embedding_clusters = np.array(embedding_clusters)
    n, m, k = embedding_clusters.shape
    tsne_model_en_2d = TSNE(perplexity=15, n_components=2, init='pca', n_iter=3500, random_state=32)
    embeddings_en_2d = np.array(tsne_model_en_2d.fit_transform(embedding_clusters.reshape(n * m, k))).reshape(n, m, 2)
    tsne_plot_similar_words('Similar words from Trump', keys, embeddings_en_2d, word_clusters, 0.7, file)
```