word2vec

Run in Google (https://colab.research.google.com/github/tensorflow/text/blob/master/docs/tutorials/word2 Colab

word2vec is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. Embeddings learned through word2vec have proven to be successful on a variety of downstream natural language processing tasks.

Note: This tutorial is based on Efficient estimation of word representations in vector space (https://arxiv.org/pdf/1301.3781.pdf) and Distributed representations of words and phrases and their compositionality (https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf) . It is not an exact implementation of the papers. Rather, it is intended to illustrate the key ideas.

These papers proposed two methods for learning representations of words:

- **Continuous bag-of-words model**: predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.
- **Continuous skip-gram model**: predicts words within a certain range before and after the current word in the same sentence. A worked example of this is given below.

You'll use the skip-gram approach in this tutorial. First, you'll explore skip-grams and other concepts using a single sentence for illustration. Next, you'll train your own word2vec model on a small dataset. This tutorial also contains code to export the trained embeddings and visualize them in the TensorFlow Embedding Projector (http://projector.tensorflow.org/).

Skip-gram and negative sampling

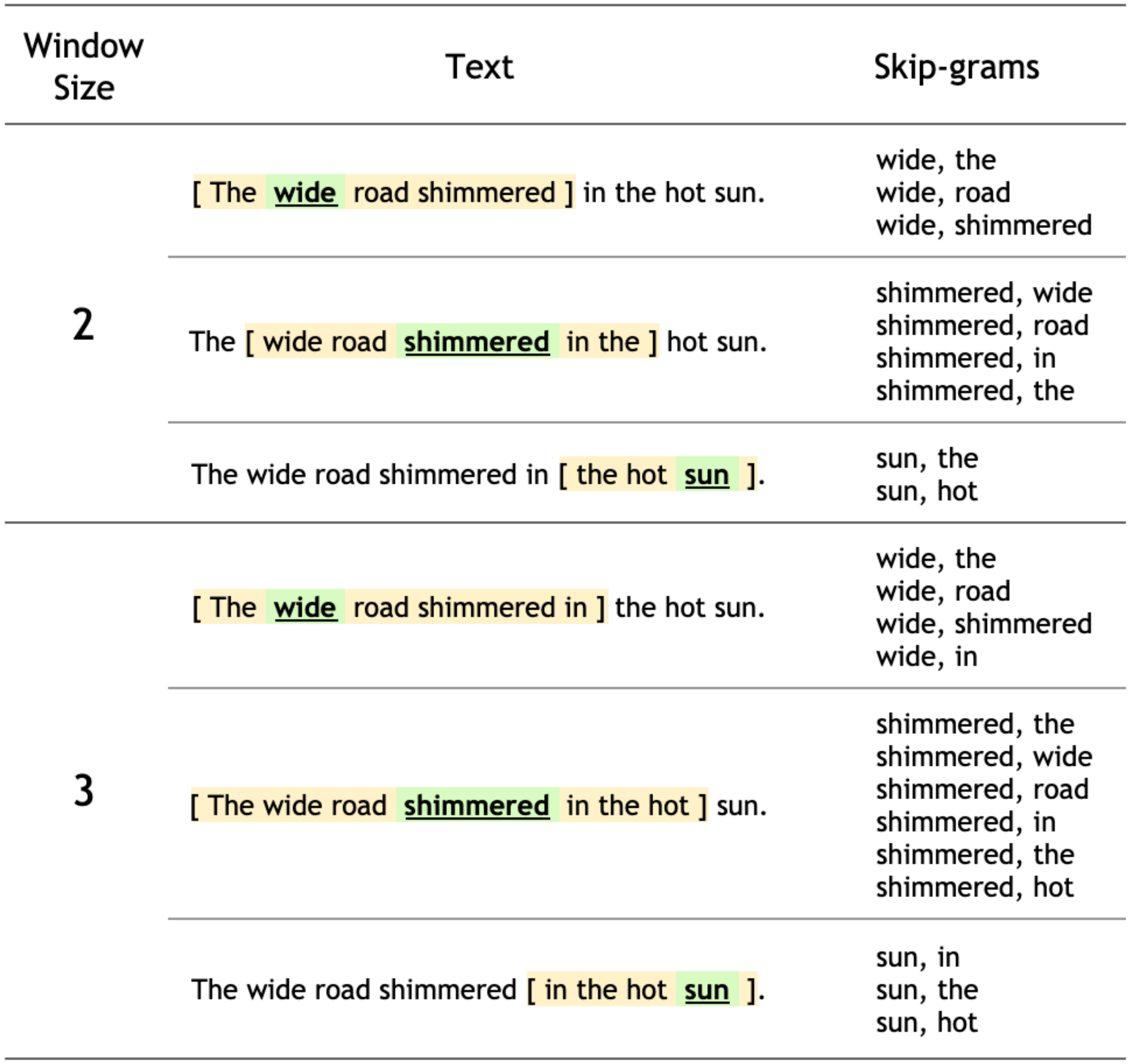
While a bag-of-words model predicts a word given the neighboring context, a skip-gram model predicts the context (or neighbors) of a word, given the word itself. The model is trained on skip-grams, which are n-grams that allow tokens to be skipped (see the diagram below for an example). The context of a word can be represented through a set of skip-gram pairs of `(target_word, context_word)` where `context_word` appears in the neighboring context of `target_word`.

Consider the following sentence of eight words:

The wide road shimmered in the hot sun.

The context words for each of the 8 words of this sentence are defined by a window size. The window size determines the span of words on either side of a `target_word` that can be considered a `context word`. Below is a table of skip-grams for target words based on different window sizes.

Note: For this tutorial, a window size of `n` implies n words on each side with a total window span of 2*n+1 words across a word.



Window Size	Text	Skip-grams
2	[The **wide** road shimmered] in the hot sun.	wide, the wide, road wide, shimmered
	The [wide road **shimmered** in the] hot sun.	shimmered, wide shimmered, road shimmered, in shimmered, the
	The wide road shimmered in [the hot **sun**].	sun, the sun, hot
3	[The **wide** road shimmered in] the hot sun.	wide, the wide, road wide, shimmered wide, in
	[The wide road **shimmered** in the hot] sun.	shimmered, the shimmered, wide shimmered, road shimmered, in shimmered, the shimmered, hot
	The wide road shimmered [in the hot **sun**].	sun, in sun, the sun, hot

The training objective of the skip-gram model is to maximize the probability of predicting context words given the target word. For a sequence of words w_1, w_2, ... w_T, the objective can be written as the average log probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t)$$

where `c` is the size of the training context. The basic skip-gram formulation defines this probability using the softmax function.

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_{w}}^\top v_{w_I}\right)}$$

where *v* and *v*$^{'}$ are target and context vector representations of words and *W* is vocabulary size.

Computing the denominator of this formulation involves performing a full softmax over the entire vocabulary words, which are often large (10^5-10^7) terms.

The noise contrastive estimation (https://www.tensorflow.org/api_docs/python/tf/nn/nce_loss) (NCE) loss function is an efficient approximation for a full softmax. With an objective to learn word embeddings instead of modeling the word distribution, the NCE loss can be simplified (https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf)
to use negative sampling.

The simplified negative sampling objective for a target word is to distinguish the context word from `num_ns` negative samples drawn from noise distribution $P_n(w)$ of words. More precisely, an efficient approximation of full softmax over the vocabulary is, for a skip-gram pair, to pose the loss for a target word as a classification problem between the context word and `num_ns` negative samples.

A negative sample is defined as a `(target_word, context_word)` pair such that the `context_word` does not appear in the `window_size` neighborhood of the `target_word`. For the example sentence, these are a few potential negative samples (when `window_size` is `2`).

```
(hot, shimmered)
(wide, hot)
(wide, sun)
```

In the next section, you'll generate skip-grams and negative samples for a single sentence. You'll also learn about subsampling techniques and train a classification model for positive and negative training examples later in the tutorial.

Setup

```
import io
import re
import string
import tqdm

import numpy as np

import tensorflow as tf
from tensorflow.keras import layers
```

```
# Load the TensorBoard notebook extension
%load_ext tensorboard
```

```
SEED = 42
AUTOTUNE = tf.data.AUTOTUNE
```

Vectorize an example sentence

Consider the following sentence:

The wide road shimmered in the hot sun.

Tokenize the sentence:

```
sentence = "The wide road shimmered in the hot sun"
tokens = list(sentence.lower().split())
```

```
print(len(tokens))
```

```
8
```

Create a vocabulary to save mappings from tokens to integer indices:

```
vocab, index = {}, 1  # start indexing from 1
vocab['<pad>'] = 0  # add a padding token
for token in tokens:
  if token not in vocab:
    vocab[token] = index
    index += 1
vocab_size = len(vocab)
print(vocab)
```

```
{'<pad>': 0, 'the': 1, 'wide': 2, 'road': 3, 'shimmered': 4, 'in': 5, 'hot': 6,
```

Create an inverse vocabulary to save mappings from integer indices to tokens:

```
inverse_vocab = {index: token for token, index in vocab.items()}
print(inverse_vocab)
```

```
{0: '<pad>', 1: 'the', 2: 'wide', 3: 'road', 4: 'shimmered', 5: 'in', 6: 'hot',
```

Vectorize your sentence:

```
example_sequence = [vocab[word] for word in tokens]
print(example_sequence)
```

```
[1, 2, 3, 4, 5, 1, 6, 7]
```

Generate skip-grams from one sentence

The **`tf.keras.preprocessing.sequence`** (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence) module provides useful functions that simplify data preparation for word2vec. You can use the **`tf.keras.preprocessing.sequence.skipgrams`** (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/skipgrams) to generate skip-gram pairs from the `example_sequence` with a given `window_size` from tokens in the range `[0, vocab_size)`.

Note: `negative_samples` is set to `0` here, as batching negative samples generated by this function requires a bit of code. You will use another function to perform negative sampling in the next section.

```
window_size = 2
positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
      example_sequence,
      vocabulary_size=vocab_size,
      window_size=window_size,
      negative_samples=0)
print(len(positive_skip_grams))
```

```
26
```

Print a few positive skip-grams:

```
for target, context in positive_skip_grams[:5]:
  print(f"({target}, {context}): ({inverse_vocab[target]}, {inverse_vocab[contex
(6, 1): (hot, the)
(3, 5): (road, in)
(1, 6): (the, hot)
(4, 3): (shimmered, road)
(5, 6): (in, hot)
```

Negative sampling for one skip-gram

The `skipgrams` function returns all positive skip-gram pairs by sliding over a given window span. To produce additional skip-gram pairs that would serve as negative samples for training, you need to sample random words from the vocabulary. Use the **`tf.random.log_uniform_candidate_sampler`** (https://www.tensorflow.org/api_docs/python/tf/random/log_uniform_candidate_sampler) function to sample `num_ns` number of negative samples for a given target word in a window. You can call the function on one skip-grams's target word and pass the context word as true class to exclude it from being sampled.

Key Point: `num_ns` (the number of negative samples per a positive context word) in the `[5, 20]` range is shown to work (https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf) best for smaller datasets, while `num_ns` in the `[2, 5]` range suffices for larger datasets.

```
# Get target and context words for one positive skip-gram.
target_word, context_word = positive_skip_grams[0]

# Set the number of negative samples per positive context.
num_ns = 4

context_class = tf.reshape(tf.constant(context_word, dtype="int64"), (1, 1))
negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sampler(
    true_classes=context_class,  # class that should be sampled as 'positive'
    num_true=1,  # each positive skip-gram has 1 positive context class
    num_sampled=num_ns,  # number of negative context words to sample
    unique=True,  # all the negative samples should be unique
    range_max=vocab_size,  # pick index of the samples from [0, vocab_size]
    seed=SEED,  # seed for reproducibility
    name="negative_sampling"  # name of this operation
)
print(negative_sampling_candidates)
print([inverse_vocab[index.numpy()] for index in negative_sampling_candidates])
```

```
tf.Tensor([2 1 4 3], shape=(4,), dtype=int64)
['wide', 'the', 'shimmered', 'road']
```

Construct one training example

For a given positive (`target_word`, `context_word`) skip-gram, you now also have `num_ns` negative sampled context words that do not appear in the window size neighborhood of `target_word`. Batch the `1` positive `context_word` and `num_ns` negative context words into one tensor. This produces a set of positive skip-grams (labeled as `1`) and negative samples (labeled as `0`) for each target word.

```
# Reduce a dimension so you can use concatenation (in the next step).
squeezed_context_class = tf.squeeze(context_class, 1)

# Concatenate a positive context word with negative sampled words.
context = tf.concat([squeezed_context_class, negative_sampling_candidates], 0)

# Label the first context word as `1` (positive) followed by `num_ns` `0`s (nega
label = tf.constant([1] + [0]*num_ns, dtype="int64")
target = target_word
```

Check out the context and the corresponding labels for the target word from the skip-gram example above:

```
print(f"target_index    : {target}")
print(f"target_word     : {inverse_vocab[target_word]}")
print(f"context_indices : {context}")
print(f"context_words   : {[inverse_vocab[c.numpy()] for c in context]}")
print(f"label           : {label}")
```

```
target_index    : 6
target_word     : hot
context_indices : [1 2 1 4 3]
context_words   : ['the', 'wide', 'the', 'shimmered', 'road']
label           : [1 0 0 0 0]
```

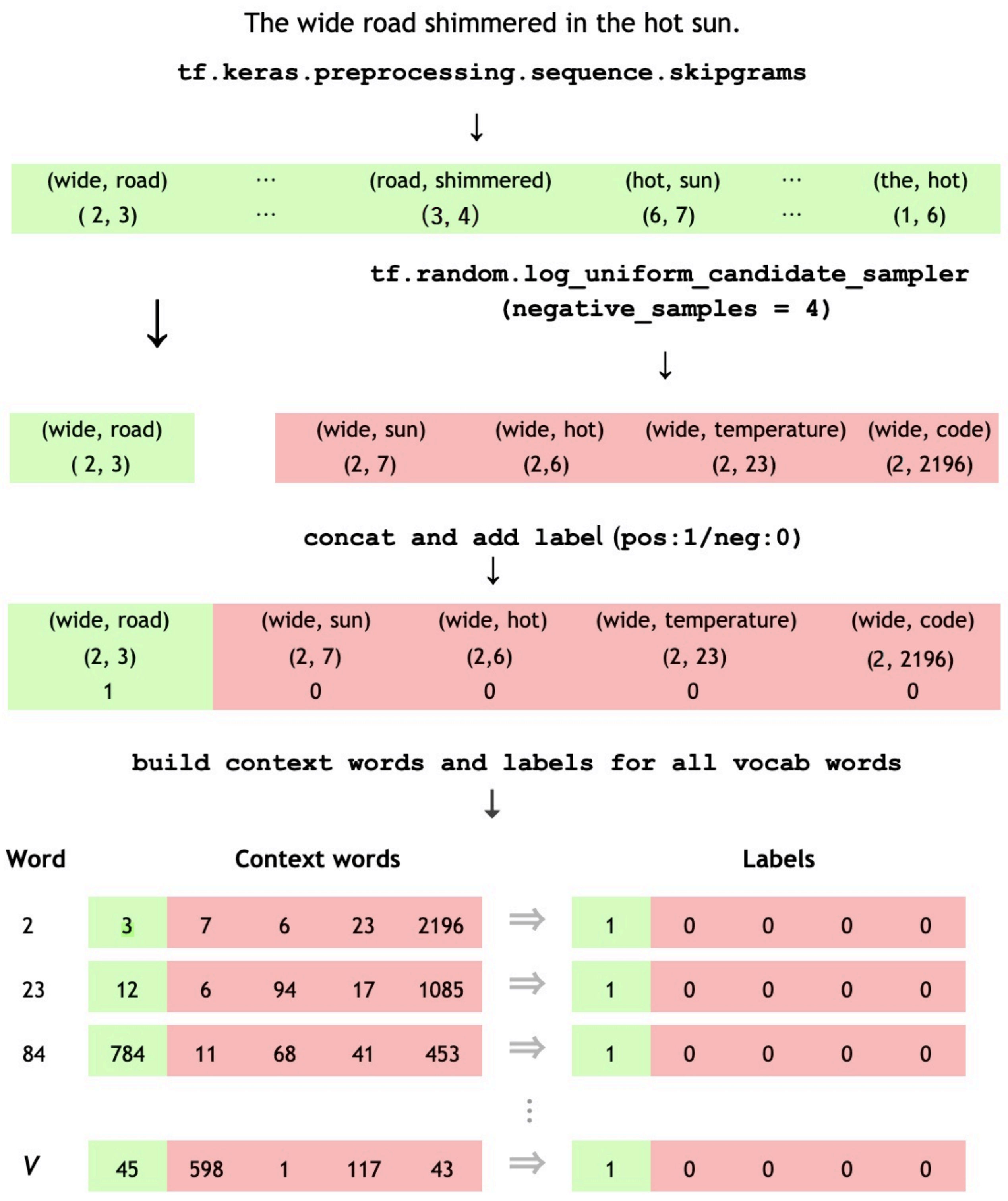
A tuple of **`(target, context, label)`** tensors constitutes one training example for training your skip-gram negative sampling word2vec model. Notice that the target is of shape **`(1,)`** while the context and label are of shape **`(1+num_ns,)`**

```
print("target  :", target)
print("context :", context)
print("label   :", label)
```

```
target  : 6
context : tf.Tensor([1 2 1 4 3], shape=(5,), dtype=int64)
label   : tf.Tensor([1 0 0 0 0], shape=(5,), dtype=int64)
```

Summary

This diagram summarizes the procedure of generating a training example from a sentence:


The wide road shimmered in the hot sun.
tf.keras.preprocessing.sequence.skipgrams
(wide, road) (2, 3)
(road, shimmered) (3, 4)
(hot, sun) (6, 7)
(the, hot) (1, 6)
tf.random.log_uniform_candidate_sampler (negative_samples = 4)
(wide, sun) (2, 7)
(wide, hot) (2,6)
(wide, temperature) (2, 23)
(wide, code) (2, 2196)
concat and add label (pos:1/neg:0)
build context words and labels for all vocab words
Word
Context words
Labels
2 3 7 6 23 2196 ⇒ 1 0 0 0 0
23 12 6 94 17 1085 ⇒ 1 0 0 0 0
84 784 11 68 41 453 ⇒ 1 0 0 0 0
V 45 598 1 117 43 ⇒ 1 0 0 0 0

Notice that the words `temperature` and `code` are not part of the input sentence. They belong to the vocabulary like certain other indices used in the diagram above.

Compile all steps into one function

Skip-gram sampling table

A large dataset means larger vocabulary with higher number of more frequent words such as stopwords. Training examples obtained from sampling commonly occurring words (such as `the`, `is`, `on`) don't add much useful information for the model to learn from. Mikolov et al. (https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf)
suggest subsampling of frequent words as a helpful practice to improve embedding quality.

The `tf.keras.preprocessing.sequence.skipgrams` (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/skipgrams) function accepts a sampling table argument to encode probabilities of sampling any token. You can use the `tf.keras.preprocessing.sequence.make_sampling_table` (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/make_sampling_table) to generate a word-frequency rank based probabilistic sampling table and pass it to the `skipgrams` function. Inspect the sampling probabilities for a `vocab_size` of 10.

```
sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(size=10)
print(sampling_table)
```

```
[0.00315225 0.00315225 0.00547597 0.00741556 0.00912817 0.01068435
 0.01212381 0.01347162 0.01474487 0.0159558 ]
```

`sampling_table[i]` denotes the probability of sampling the i-th most common word in a dataset. The function assumes a Zipf's distribution (https://en.wikipedia.org/wiki/Zipf%27s_law) of the word frequencies for sampling.

Key Point: The `tf.random.log_uniform_candidate_sampler` (https://www.tensorflow.org/api_docs/python/tf/random/log_uniform_candidate_sampler) already assumes that the vocabulary frequency follows a log-uniform (Zipf's) distribution. Using these distribution weighted sampling also helps approximate the Noise Contrastive Estimation (NCE) loss with simpler loss functions for training a negative sampling objective.

Generate training data

Compile all the steps described above into a function that can be called on a list of vectorized sentences obtained from any text dataset. Notice that the sampling table is built before sampling skip-gram word pairs. You will use this function in the later sections.

```
# Generates skip-gram pairs with negative sampling for a list of sequences
# (int-encoded sentences) based on window size, number of negative samples
# and vocabulary size.
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
  # Elements of each training example are appended to these lists.
  targets, contexts, labels = [], [], []

  # Build the sampling table for `vocab_size` tokens.
  sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(vocab_siz

  # Iterate over all sequences (sentences) in the dataset.
  for sequence in tqdm.tqdm(sequences):

    # Generate positive skip-gram pairs for a sequence (sentence).
    positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
          sequence,
          vocabulary_size=vocab_size,
          sampling_table=sampling_table,
          window_size=window_size,
          negative_samples=0)

    # Iterate over each positive skip-gram pair to produce training examples
    # with a positive context word and negative samples.
    for target_word, context_word in positive_skip_grams:
      context_class = tf.expand_dims(
          tf.constant([context_word], dtype="int64"), 1)
      negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sampl
          true_classes=context_class,
          num_true=1,
          num_sampled=num_ns,
          unique=True,
          range_max=vocab_size,
          seed=seed,
          name="negative_sampling")

      # Build context and label vectors (for one target word)
      context = tf.concat([tf.squeeze(context_class,1), negative_sampling_candid
      label = tf.constant([1] + [0]*num_ns, dtype="int64")
      # Append each element from the training example to global lists.
      targets.append(target_word)
      contexts.append(context)
      labels.append(label)

  return targets, contexts, labels
```

Prepare training data for word2vec

With an understanding of how to work with one sentence for a skip-gram negative sampling based word2vec model, you can proceed to generate training examples from a larger list of sentences!

Download text corpus

You will use a text file of Shakespeare's writing for this tutorial. Change the following line to run this code on your own data.

```
path_to_file = tf.keras.utils.get_file('shakespeare.txt', 'https://storage.googl
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/dat
1115394/1115394 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Read the text from the file and print the first few lines:

```
with open(path_to_file) as f:
  lines = f.read().splitlines()
for line in lines[:20]:
  print(line)
```

```
First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
```

Use the non empty lines to construct a **`tf.data.TextLineDataset`** (https://www.tensorflow.org/api_docs/python/tf/data/TextLineDataset) object for the next steps:

```
text_ds = tf.data.TextLineDataset(path_to_file).filter(lambda x: tf.cast(tf.stri
```

Vectorize sentences from the corpus

You can use the **`TextVectorization`** layer to vectorize sentences from the corpus. Learn more about using this layer in this Text classification (https://www.tensorflow.org/tutorials/keras/text_classification) tutorial. Notice from the first few sentences above that the text needs to be in one case and punctuation needs to be removed. To do this, define a **`custom_standardization function`** that can be used in the TextVectorization layer.

```
# Now, create a custom standardization function to lowercase the text and
# remove punctuation.
def custom_standardization(input_data):
  lowercase = tf.strings.lower(input_data)
  return tf.strings.regex_replace(lowercase,
                                  '[%s]' % re.escape(string.punctuation), '')
```

```
# Define the vocabulary size and the number of words in a sequence.
```

```
vocab_size = 4096
sequence_length = 10

# Use the `TextVectorization` layer to normalize, split, and map strings to
# integers. Set the `output_sequence_length` length to pad all samples to the
# same length.
vectorize_layer = layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)
```

Call **`TextVectorization.adapt`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization#adapt) on the text dataset to create vocabulary.

```
vectorize_layer.adapt(text_ds.batch(1024))
```

```
2024-03-13 11:10:22.892874: W tensorflow/core/framework/local_rendezvous.cc:404]
```

Once the state of the layer has been adapted to represent the text corpus, the vocabulary can be accessed with **`TextVectorization.get_vocabulary`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization#get_vocabulary). This function returns a list of all vocabulary tokens sorted (descending) by their frequency.

```
# Save the created vocabulary for reference.
inverse_vocab = vectorize_layer.get_vocabulary()
print(inverse_vocab[:20])
```

```
['', '[UNK]', 'the', 'and', 'to', 'i', 'of', 'you', 'my', 'a', 'that', 'in', 'is
```

The `vectorize_layer` can now be used to generate vectors for each element in the `text_ds` (a **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset)). Apply **`Dataset.batch`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch), **`Dataset.prefetch`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#prefetch), **`Dataset.map`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map), and **`Dataset.unbatch`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#unbatch).

```
# Vectorize the data in text_ds.
text_vector_ds = text_ds.batch(1024).prefetch(AUTOTUNE).map(vectorize_layer).unb
```

Obtain sequences from the dataset

You now have a **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) of integer encoded sentences. To prepare the dataset for training a word2vec model, flatten the dataset into a list of sentence vector sequences. This step is required as you would iterate over each sentence in the dataset to produce positive and negative examples.

Note: Since the `generate_training_data()` defined earlier uses non-TensorFlow Python/NumPy functions, you could also use a **`tf.py_function`** (https://www.tensorflow.org/api_docs/python/tf/py_function) or **`tf.numpy_function`** (https://www.tensorflow.org/api_docs/python/tf/numpy_function) with **`tf.data.Dataset.map`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map).

```
sequences = list(text_vector_ds.as_numpy_iterator())
print(len(sequences))
```

```
32777
2024-03-13 11:10:26.564811: W tensorflow/core/framework/local_rendezvous.cc:404]
```

Inspect a few examples from `sequences`:

```
for seq in sequences[:5]:
  print(f"{seq} => {[inverse_vocab[i] for i in seq]}")
```

```
[ 89 270   0   0   0   0   0   0   0   0] => ['first', 'citizen', '', '', '', ''
[138  36 982 144 673 125  16 106   0   0] => ['before', 'we', 'proceed', 'any',
[34  0  0  0  0  0  0  0  0  0] => ['all', '', '', '', '', '', '', '', '', '']
[106 106   0   0   0   0   0   0   0   0] => ['speak', 'speak', '', '', '', '',
[ 89 270   0   0   0   0   0   0   0   0] => ['first', 'citizen', '', '', '', ''
```

Generate training examples from sequences

`sequences` is now a list of int encoded sentences. Just call the **`generate_training_data`** function defined earlier to generate training examples for the word2vec model. To recap, the function iterates over each word from each sequence to collect positive and negative context words. Length of target, contexts and labels should be the same, representing the total number of training examples.

```
targets, contexts, labels = generate_training_data(
    sequences=sequences,
    window_size=2,
    num_ns=4,
    vocab_size=vocab_size,
    seed=SEED)

targets = np.array(targets)
contexts = np.array(contexts)
labels = np.array(labels)

print('\n')
print(f"targets.shape: {targets.shape}")
print(f"contexts.shape: {contexts.shape}")
print(f"labels.shape: {labels.shape}")
```

```
100%|██████████| 32777/32777 [00:37<00:00, 865.32it/s]
targets.shape: (65462,)
contexts.shape: (65462, 5)
labels.shape: (65462, 5)
```

Configure the dataset for performance

To perform efficient batching for the potentially large number of training examples, use the **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) API. After this step, you would have a **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) object of `(target_word, context_word), (label)` elements to train your word2vec model!

```
BATCH_SIZE = 1024
BUFFER_SIZE = 10000
dataset = tf.data.Dataset.from_tensor_slices(((targets, contexts), labels))
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
print(dataset)
```

```
<_BatchDataset element_spec=((TensorSpec(shape=(1024,), dtype=tf.int64, name=Non
```

Apply **`Dataset.cache`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#cache) and **`Dataset.prefetch`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#prefetch) to improve performance:

```
dataset = dataset.cache().prefetch(buffer_size=AUTOTUNE)
print(dataset)
```

```
<_PrefetchDataset element_spec=((TensorSpec(shape=(1024,), dtype=tf.int64, name=
```

Model and training

The word2vec model can be implemented as a classifier to distinguish between true context words from skip-grams and false context words obtained through negative sampling. You can perform a dot product multiplication between the embeddings of target and context words to obtain predictions for labels and compute the loss function against true labels in the dataset.

Subclassed word2vec model

Use the Keras Subclassing API (https://www.tensorflow.org/guide/keras/custom_layers_and_models) to define your word2vec model with the following layers:

- `target_embedding`: A `tf.keras.layers.Embedding` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding) layer, which looks up the embedding of a word when it appears as a target word. The number of parameters in this layer are `(vocab_size * embedding_dim)`.
- `context_embedding`: Another `tf.keras.layers.Embedding` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding) layer, which looks up the embedding of a word when it appears as a context word. The number of parameters in this layer are the same as those in `target_embedding`, i.e. `(vocab_size * embedding_dim)`.
- `dots`: A `tf.keras.layers.Dot` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dot) layer that computes the dot product of target and context embeddings from a training pair.
- `flatten`: A `tf.keras.layers.Flatten` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten) layer to flatten the results of `dots` layer into logits.

With the subclassed model, you can define the `call()` function that accepts `(target, context)` pairs which can then be passed into their corresponding embedding layer. Reshape the `context_embedding` to perform a dot product with `target_embedding` and return the flattened result.

Key Point: The `target_embedding` and `context_embedding` layers can be shared as well. You could also use a concatenation of both embeddings as the final word2vec embedding.

```
class Word2Vec(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim):
    super(Word2Vec, self).__init__()
    self.target_embedding = layers.Embedding(vocab_size,
                                      embedding_dim,
                                      name="w2v_embedding")
    self.context_embedding = layers.Embedding(vocab_size,
                                       embedding_dim)

  def call(self, pair):
    target, context = pair
    # target: (batch, dummy?)  # The dummy axis doesn't exist in TF2.7+
    # context: (batch, context)
    if len(target.shape) == 2:
      target = tf.squeeze(target, axis=1)
    # target: (batch,)
    word_emb = self.target_embedding(target)
    # word_emb: (batch, embed)
    context_emb = self.context_embedding(context)
    # context_emb: (batch, context, embed)
    dots = tf.einsum('be,bce->bc', word_emb, context_emb)
    # dots: (batch, context)
    return dots
```

Define loss function and compile model

For simplicity, you can use `tf.keras.losses.CategoricalCrossEntropy` as an alternative to the negative sampling loss. If you would like to write your own custom loss function, you can also do so as follows:

```
def custom_loss(x_logit, y_true):
      return tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=y_tr
```

It's time to build your model! Instantiate your word2vec class with an embedding dimension of 128 (you could experiment with different values). Compile the model with the `tf.keras.optimizers.Adam` (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam) optimizer.

```
embedding_dim = 128
word2vec = Word2Vec(vocab_size, embedding_dim)
word2vec.compile(optimizer='adam',
                 loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
                 metrics=['accuracy'])
```

Also define a callback to log training statistics for TensorBoard:

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="logs")
```

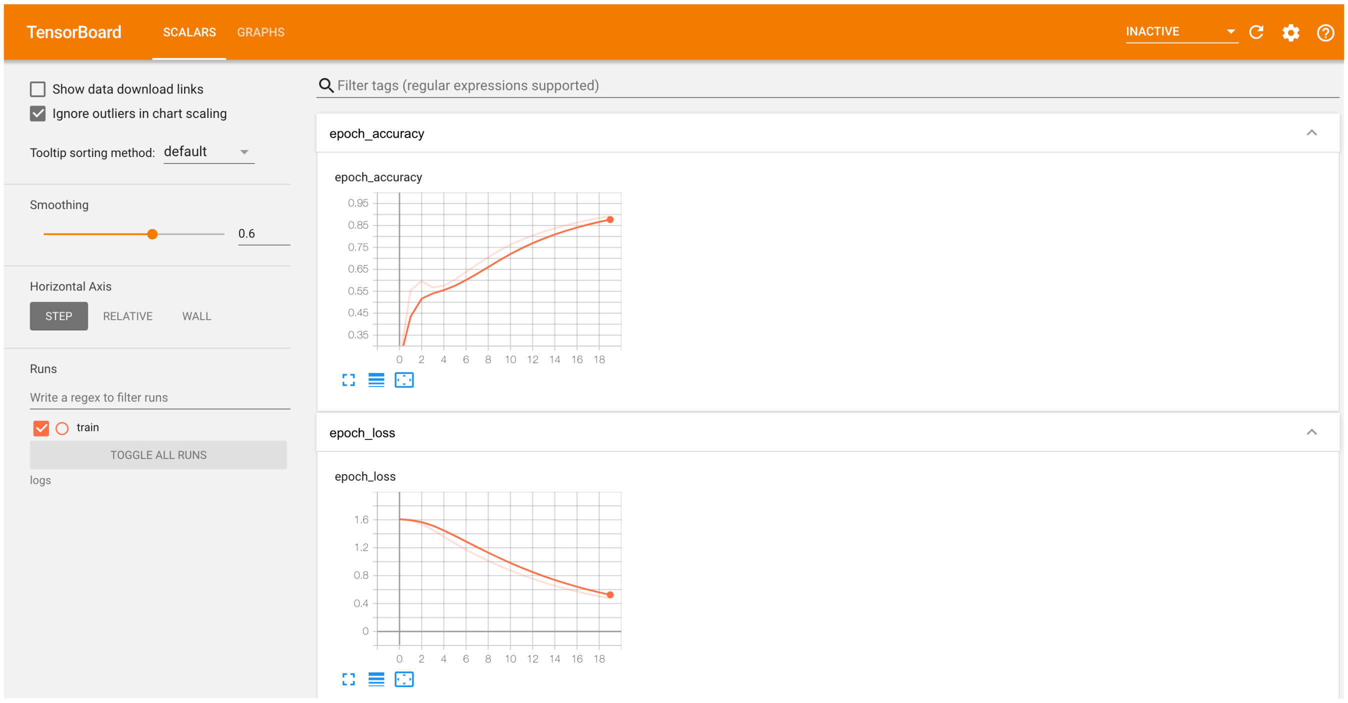
Train the model on the `dataset` for some number of epochs:

```
word2vec.fit(dataset, epochs=20, callbacks=[tensorboard_callback])
```

```
Epoch 1/20
WARNING: All log messages before absl::InitializeLog() is called are written to
I0000 00:00:1710328268.494065   10203 service.cc:145] XLA service 0x7f0a6c03e96
I0000 00:00:1710328268.494115   10203 service.cc:153]   StreamExecutor device (
I0000 00:00:1710328268.494119   10203 service.cc:153]   StreamExecutor device (
I0000 00:00:1710328268.494124   10203 service.cc:153]   StreamExecutor device (
I0000 00:00:1710328268.494127   10203 service.cc:153]   StreamExecutor device (
31/63 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.2072 - loss: 1.6092
I0000 00:00:1710328269.065899   10203 device_compiler.h:188] Compiled cluster u
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.2166 - loss: 1.6088
Epoch 2/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5957 - loss: 1.5886
Epoch 3/20
```

TensorBoard now shows the word2vec model's accuracy and loss:

```
#docs_infra: no_execute
%tensorboard --logdir logs
```



Embedding lookup and analysis

Obtain the weights from the model using **`Model.get_layer`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#get_layer) and **`Layer.get_weights`** (https://www.tensorflow.org/api_docs/python/tf/keras/Layer#get_weights). The **`TextVectorization.get_vocabulary`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization#get_vocabulary) function provides the vocabulary to build a metadata file with one token per line.

```
weights = word2vec.get_layer('w2v_embedding').get_weights()[0]
vocab = vectorize_layer.get_vocabulary()
```

Create and save the vectors and metadata files:

```
out_v = io.open('vectors.tsv', 'w', encoding='utf-8')
out_m = io.open('metadata.tsv', 'w', encoding='utf-8')

for index, word in enumerate(vocab):
  if index == 0:
    continue  # skip 0, it's padding.
  vec = weights[index]
  out_v.write('\t'.join([str(x) for x in vec]) + "\n")
  out_m.write(word + "\n")
out_v.close()
out_m.close()
```

Download the `vectors.tsv` and `metadata.tsv` to analyze the obtained embeddings in the Embedding Projector (https://projector.tensorflow.org/):

```
try:
  from google.colab import files
  files.download('vectors.tsv')
  files.download('metadata.tsv')
except Exception:
  pass
```

Next steps

This tutorial has shown you how to implement a skip-gram word2vec model with negative sampling from scratch and visualize the obtained word embeddings.

- To learn more about word vectors and their mathematical representations, refer to these notes (https://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes01-wordvecs1.pdf).
- To learn more about advanced text processing, read the Transformer model for language understanding (https://www.tensorflow.org/tutorials/text/transformer) tutorial.
- If you're interested in pre-trained embedding models, you may also be interested in Exploring the TF-Hub CORD-19 Swivel Embeddings (https://www.tensorflow.org/hub/tutorials/cord_19_embeddings_keras), or the Multilingual Universal Sentence Encoder (https://www.tensorflow.org/hub/tutorials/cross_lingual_similarity_with_tf_hub_multilingual_universal_encoder)
.
- You may also like to train the model on a new dataset (there are many available in TensorFlow Datasets (https://www.tensorflow.org/datasets)).