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

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.](#) suggest subsampling of frequent words as a helpful practice to improve embedding quality.

The `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` 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 of the word frequencies for sampling.

Key point: The `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_size)

  # 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_sampler(
          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_candidates], 0)
      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.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import zipfile

# Unzip the archive
local_zip = '/content/drive/MyDrive/Colab Notebooks/Text.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall()

zip_ref.close()
```

```
import os

base_dir = 'Text'

print("Contents of base directory:")
print(os.listdir(base_dir))

print("\nContents of AI directory:")
print(os.listdir(f'{base_dir}/AI'))

print("\nContents of NLP directory:")
```

```
print(os.listdir(f'{base_dir}/NLP'))
```

```
Contents of base directory:
['AI', 'Stats', 'NLP', 'CV']

Contents of AI directory:
['Artificial intelligence and education in China.pdf', '10.2478_rem-2020-0003.pdf', 's00146-020-01033-8.pdf', 'sustainability-13-07941.pdf', 'article_222

Contents of NLP directory:
['Advances_in_Natural_Language_Processing.pdf', 'doc4.pdf', 'Using natural language processing technology for qualitative data analysis.pdf', 'doc10.pdf'
```

```
main_dir_files = os.listdir(base_dir)
main_dir_files
```

```
['AI', 'Stats', 'NLP', 'CV']
```

```
!pip install PyPDF2
```

```
Requirement already satisfied: PyPDF2 in /usr/local/lib/python3.10/dist-packages (3.0.1)
```

```
import PyPDF2

for i in main_dir_files:
  sub_dir_files = os.listdir(base_dir+'/'+i)
  k=1
  for j in sub_dir_files:
    #create file object variable
    #opening method will be rb
    pdffileobj=open(base_dir+'/'+i+'/'+j,'rb')

    #create reader variable that will read the pdffileobj
    reader = PyPDF2.PdfReader(pdffileobj)

    #This will store the number of pages of this pdf file
    x = len(reader.pages)
    text = ''
    for pages in range(x):
      page = reader.pages[pages]
      text += page.extract_text()

    #filename = 'file'+str(i)+str(k)
    filename = open(base_dir+"/"+str(i)+"/"+str(k)+".txt", "a")
    filename.writelines(text)
```

```
    filename.close()
    k+=1
```

```
/usr/local/lib/python3.10/dist-packages/PyPDF2/_cmap.py:142: PdfReadWarning: Advanced encoding /StandardEncoding not implemented yet
  warnings.warn(
```

```
train_ds = tf.keras.utils.text_dataset_from_directory(
  base_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  batch_size=10)
```

```
Found 117 files belonging to 4 classes.
Using 94 files for training.
```

```
val_ds = tf.keras.utils.text_dataset_from_directory(
  base_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  batch_size=10)
```

```
Found 117 files belonging to 4 classes.
Using 23 files for validation.
```

```
text_ds = tf.keras.utils.text_dataset_from_directory(
  base_dir,
  seed=123)
```

```
Found 117 files belonging to 4 classes.
```

```
doc_len = len(list(text_ds.as_numpy_iterator())[0][0])
```

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 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` on the text dataset to create vocabulary.

```
class_len = len(list(text_ds.as_numpy_iterator())[0])
```

```
# 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_size)

  # 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
```

```
          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_sampler(
            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_candidates], 0)
        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


  for i in main_dir_files:
    sub_dir_files = os.listdir(base_dir+'/'+i)
    for j in sub_dir_files:
      text_ds = tf.data.TextLineDataset(base_dir+'/'+i+'/'+j).filter(lambda x: tf.cast(tf.strings.length(x), bool))
      vectorize_layer.adapt(text_ds.batch(1024))

      #inverse_vocab = vectorize_layer.get_vocabulary()
      #print(inverse_vocab[:20])

      # Vectorize the data in text_ds.
      text_vector_ds = text_ds.batch(batch_size=1024).prefetch(buffer_size=AUTOTUNE).map(vectorize_layer).unbatch()

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

```
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}")
```

```
529
100%|██████████| 529/529 [00:00<00:00, 1089.02it/s]


targets.shape: (1690,)
contexts.shape: (1690, 5)
labels.shape: (1690, 5)
2900
100%|██████████| 2900/2900 [00:04<00:00, 702.59it/s]


targets.shape: (9015,)
contexts.shape: (9015, 5)
labels.shape: (9015, 5)
22320
100%|██████████| 22320/22320 [00:02<00:00, 8823.20it/s]


targets.shape: (6790,)
contexts.shape: (6790, 5)
labels.shape: (6790, 5)
626
100%|██████████| 626/626 [00:00<00:00, 837.41it/s]


targets.shape: (2501,)
contexts.shape: (2501, 5)
labels.shape: (2501, 5)
647
100%|██████████| 647/647 [00:00<00:00, 1058.52it/s]
```

```
targets.shape: (2161,)
contexts.shape: (2161, 5)
labels.shape: (2161, 5)
3273
100%|██████████| 3273/3273 [00:01<00:00, 1895.01it/s]


targets.shape: (4136,)
contexts.shape: (4136, 5)
labels.shape: (4136, 5)
4158
100%|██████████| 4158/4158 [00:01<00:00, 4025.57it/s]


targets.shape: (3856,)
contexts.shape: (3856, 5)
labels.shape: (3856, 5)
11360
100%|██████████| 11360/11360 [00:04<00:00, 2278.11it/s]


targets.shape: (18385,)
contexts.shape: (18385, 5)
labels.shape: (18385, 5)
467
100%|██████████| 467/467 [00:00<00:00, 982.34it/s]
```

Configure the dataset for performance

To perform efficient batching for the potentially large number of training examples, use the `tf.data.Dataset` API. After this step, you would have a `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=None), TensorSpec(shape=(1024, 5), dtype=tf.int64, name=None)), TensorSpec(s
```

Apply `Dataset.cache` and `Dataset.prefetch` to improve performance:

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

```
<_PrefetchDataset element_spec=((TensorSpec(shape=(1024,), dtype=tf.int64, name=None), TensorSpec(shape=(1024, 5), dtype=tf.int64, name=None)), TensorSpe
```

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 to define your word2vec model with the following layers:

- `target_embedding`: A `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` 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` layer that computes the dot product of target and context embeddings from a training pair.
- `flatten`: A `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_true)
```

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` optimizer.

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

```
              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
4/4 [==============================] - 1s 15ms/step - loss: 1.6083 - accuracy: 0.2478
Epoch 2/20
4/4 [==============================] - 0s 12ms/step - loss: 1.5989 - accuracy: 0.5977
Epoch 3/20
4/4 [==============================] - 0s 16ms/step - loss: 1.5890 - accuracy: 0.8386
Epoch 4/20
4/4 [==============================] - 0s 12ms/step - loss: 1.5773 - accuracy: 0.9453
Epoch 5/20
4/4 [==============================] - 0s 12ms/step - loss: 1.5630 - accuracy: 0.9795
Epoch 6/20
4/4 [==============================] - 0s 12ms/step - loss: 1.5454 - accuracy: 0.9871
Epoch 7/20
4/4 [==============================] - 0s 11ms/step - loss: 1.5241 - accuracy: 0.9883
Epoch 8/20
4/4 [==============================] - 0s 13ms/step - loss: 1.4984 - accuracy: 0.9871
Epoch 9/20
4/4 [==============================] - 0s 13ms/step - loss: 1.4681 - accuracy: 0.9861
Epoch 10/20
4/4 [==============================] - 0s 12ms/step - loss: 1.4326 - accuracy: 0.9861
Epoch 11/20
4/4 [==============================] - 0s 12ms/step - loss: 1.3920 - accuracy: 0.9849
Epoch 12/20
4/4 [==============================] - 0s 11ms/step - loss: 1.3460 - accuracy: 0.9839
Epoch 13/20
4/4 [==============================] - 0s 11ms/step - loss: 1.2949 - accuracy: 0.9832
Epoch 14/20
4/4 [==============================] - 0s 13ms/step - loss: 1.2391 - accuracy: 0.9810
Epoch 15/20
4/4 [==============================] - 0s 12ms/step - loss: 1.1792 - accuracy: 0.9792
Epoch 16/20
4/4 [==============================] - 0s 11ms/step - loss: 1.1160 - accuracy: 0.9783
Epoch 17/20
4/4 [==============================] - 0s 11ms/step - loss: 1.0506 - accuracy: 0.9771
Epoch 18/20
```

```
4/4 [==============================] - 0s 13ms/step - loss: 0.9842 - accuracy: 0.9766
Epoch 19/20
4/4 [==============================] - 0s 15ms/step - loss: 0.9180 - accuracy: 0.9753
Epoch 20/20
4/4 [==============================] - 0s 11ms/step - loss: 0.8533 - accuracy: 0.9751
<keras.src.callbacks.History at 0x7a83b2106350>
```

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` and `Layer.get_weights`. The `TextVectorization.get_vocabulary` function provides the vocabulary to build a metadata file with one token per line.

Create and save the vectors and metadata files:

Download the `vectors.tsv` and `metadata.tsv` to analyze the obtained embeddings in the Embedding Projector:

```
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.
- To learn more about advanced text processing, read the Transformer model for language understanding tutorial.