OPTION #1: Build a TensorFlow Demo

Scott Miner

Colorado State University – Global Campus

Abstract

This paper investigates the creation of a Word2Vec word embedding model using Wikipedia data and TensorFlow 2.0+. Word embeddings, which represent words as dense, lower-dimensional vectors, are crucial for various natural language processing tasks, including semantic parsing, sentiment analysis, part-of-speech tagging, and named-entity recognition. The paper focuses on the skip-gram negative-sampling model of the Word2Vec algorithm, a scalable and efficient method for generating high-quality word embeddings. Using TensorFlow 2.0+ and a Wikipedia dataset, the model is trained through unsupervised learning techniques, demonstrating the potential applications in the author's chatbot development project. The paper concludes with suggestions for enhancing the model and its dataset, highlighting the significance of word embeddings in natural language processing tasks.

```
Command Prompt - python
Microsoft Windows [Version 10.0.19042.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Scott>python
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr  6 2021, 13:40:21) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> print('Num GPUs Available: ', len(tf.config.list_physical_devices('GPU')))
Num GPUs Available:  1
>>> tf.test.is_built_with_cuda()
True
>>> tf.config.list_physical_devices('GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
>>>
>>> _
```

Figure 1. Python output showing GPU support for TensorFlow installed

```
#!/usr/bin/env python
# coding: utf-8

# # Word2Vec (Word Embedding)
#
# Implement Word2Vec algorithm to compute vector representations of words, with TensorFlow 2.0. This example is using a small chunk of Wikipedia articles to train from.
#
# More info: [Mikolov, Tomas et al. "Efficient Estimation of Word Representations in Vector Space.", 2013](https://arxiv.org/pdf/1301.3781.pdf)
#
# - Author: Aymeric Damien
# - Project: https://github.com/aymericdamien/TensorFlow-Examples/

# # Import

from __future__ import division, print_function, absolute_import

import collections
import os
import random
import urllib.request
import zipfile

import numpy as np
import tensorflow as tf

# # Set Parameters

# Training Parameters.
learning_rate = 0.1
batch_size = 128
num_steps = 3000000
display_step = 10000
eval_step = 200000

# Evaluation Parameters.
eval_words = [b'five', b'of', b'going', b'hardware', b'american', b'britain']

# Word2Vec Parameters.
embedding_size = 200 # Dimension of the embedding vector.
max_vocabulary_size = 50000 # Total number of different words in the vocabulary.
min_occurrence = 10 # Remove all words that does not appears at least n times.
skip_window = 3 # How many words to consider left and right.
num_skips = 2 # How many times to reuse an input to generate a label.
num_sampled = 64 # Number of negative examples to sample.

# # Download File
# Download a small chunk of Wikipedia articles collection.
url = 'http://mattmahoney.net/dc/text8.zip'
data_path = 'text8.zip'
if not os.path.exists(data_path):
    print("Downloading the dataset... (It may take some time)")
```

Figure 2. Code screenshot (part 1)

```
    filename, _ = urllib.request.urlretrieve(url, data_path)
    print("Done!")
# Unzip the dataset file. Text has already been processed.
with zipfile.ZipFile(data_path) as f:
    text_words = f.read(f.namelist()[0]).lower().split()

# Build the dictionary and replace rare words with UNK token.
count = [('UNK', -1)]
count.extend(collections.Counter(text_words).most_common(max_vocabulary_size - 1))

# Remove samples with less than 'min_occurrence' occurrences.
for i in range(len(count) - 1, -1, -1):
    if count[i][1] < min_occurrence:
        count.pop(i)
    else:
        # The collection is ordered, so stop when 'min_occurrence' is reached.
        break
# Compute the vocabulary size.
vocabulary_size = len(count)

# Assign an id to each word.
word2id = dict()
for i, (word, _)in enumerate(count):
    word2id[word] = i

data = list()
unk_count = 0
for word in text_words:
    # Retrieve a word id, or assign it index 0 ('UNK') if not in dictionary.
    index = word2id.get(word, 0)
    if index == 0:
        unk_count += 1
    data.append(index)
count[0] = ('UNK', unk_count)
id2word = dict(zip(word2id.values(), word2id.keys()))

print("Words count:", len(text_words))
print("Unique words:", len(set(text_words)))
print("Vocabulary size:", vocabulary_size)
print("Most common words:", count[:10])

data_index = 0
# Generate training batch for the skip-gram model.
def next_batch(batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    # get window size (words left and right + current one).
    span = 2 * skip_window + 1
```

Figure 3. Code screenshot (part 2)

```
    buffer = collections.deque(maxlen=span)
    if data_index + span > len(data):
        data_index = 0
    buffer.extend(data[data_index:data_index + span])
    data_index += span
    for i in range(batch_size // num_skips):
        context_words = [w for w in range(span) if w != skip_window]
        words_to_use = random.sample(context_words, num_skips)
        for j, context_word in enumerate(words_to_use):
            batch[i * num_skips + j] = buffer[skip_window]
            labels[i * num_skips + j, 0] = buffer[context_word]
        if data_index == len(data):
            buffer.extend(data[0:span])
            data_index = span
        else:
            buffer.append(data[data_index])
            data_index += 1
    # Backtrack a little bit to avoid skipping words in the end of a batch.
    data_index = (data_index + len(data) - span) % len(data)
    return batch, labels

# Ensure the following ops & var are assigned on CPU
# (some ops are not compatible on GPU).
with tf.device('/cpu:0'):
    # Create the embedding variable (each row represent a word embedding vector).
    embedding = tf.Variable(tf.random.normal([vocabulary_size, embedding_size]))
    # Construct the variables for the NCE loss.
    nce_weights = tf.Variable(tf.random.normal([vocabulary_size, embedding_size]))
    nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

def get_embedding(x):
    with tf.device('/cpu:0'):
        # Lookup the corresponding embedding vectors for each sample in X.
        x_embed = tf.nn.embedding_lookup(embedding, x)
        return x_embed

def nce_loss(x_embed, y):
    with tf.device('/cpu:0'):
        # Compute the average NCE loss for the batch.
        y = tf.cast(y, tf.int64)
        loss = tf.reduce_mean(
            tf.nn.nce_loss(weights=nce_weights,
                           biases=nce_biases,
                           labels=y,
                           inputs=x_embed,
                           num_sampled=num_sampled,
                           num_classes=vocabulary_size))
        return loss

# Evaluation.
def evaluate(x_embed):
```

Figure 4. Code screenshot (part 3)

```

# Evaluation.
def evaluate(x_embed):
    with tf.device('/cpu:0'):
        # Compute the cosine similarity between input data embedding and every embedding vectors
        x_embed = tf.cast(x_embed, tf.float32)
        x_embed_norm = x_embed / tf.sqrt(tf.reduce_sum(tf.square(x_embed)))
        embedding_norm = embedding / tf.sqrt(tf.reduce_sum(tf.square(embedding), 1, keepdims=True), tf.float32)
        cosine_sim_op = tf.matmul(x_embed_norm, embedding_norm, transpose_b=True)
        return cosine_sim_op

# Define the optimizer.
optimizer = tf.optimizers.SGD(learning_rate)

# Optimization process.
def run_optimization(x, y):
    with tf.device('/cpu:0'):
        # Wrap computation inside a GradientTape for automatic differentiation.
        with tf.GradientTape() as g:
            emb = get_embedding(x)
            loss = nce_loss(emb, y)

        # Compute gradients.
        gradients = g.gradient(loss, [embedding, nce_weights, nce_biases])

        # Update W and b following gradients.
        optimizer.apply_gradients(zip(gradients, [embedding, nce_weights, nce_biases]))

# Words for testing.
x_test = np.array([word2id[w] for w in eval_words])

# Run training for the given number of steps.
for step in range(1, num_steps + 1):
    batch_x, batch_y = next_batch(batch_size, num_skips, skip_window)
    run_optimization(batch_x, batch_y)

    if step % display_step == 0 or step == 1:
        loss = nce_loss(get_embedding(batch_x), batch_y)
        print("step: %i, loss: %f" % (step, loss))

    # Evaluation.
    if step % eval_step == 0 or step == 1:
        print("Evaluation...")
        sim = evaluate(get_embedding(x_test)).numpy()
        for i in range(len(eval_words)):
            top_k = 8  # number of nearest neighbors.
            nearest = (-sim[i, :]).argsort()[1:top_k + 1]
            log_str = '"%s" nearest neighbors:' % eval_words[i]
            for k in range(top_k):
                log_str = '%s %s,' % (log_str, id2word[nearest[k]])
            print(log_str)
```

Figure 5. Code screenshot (part 5)

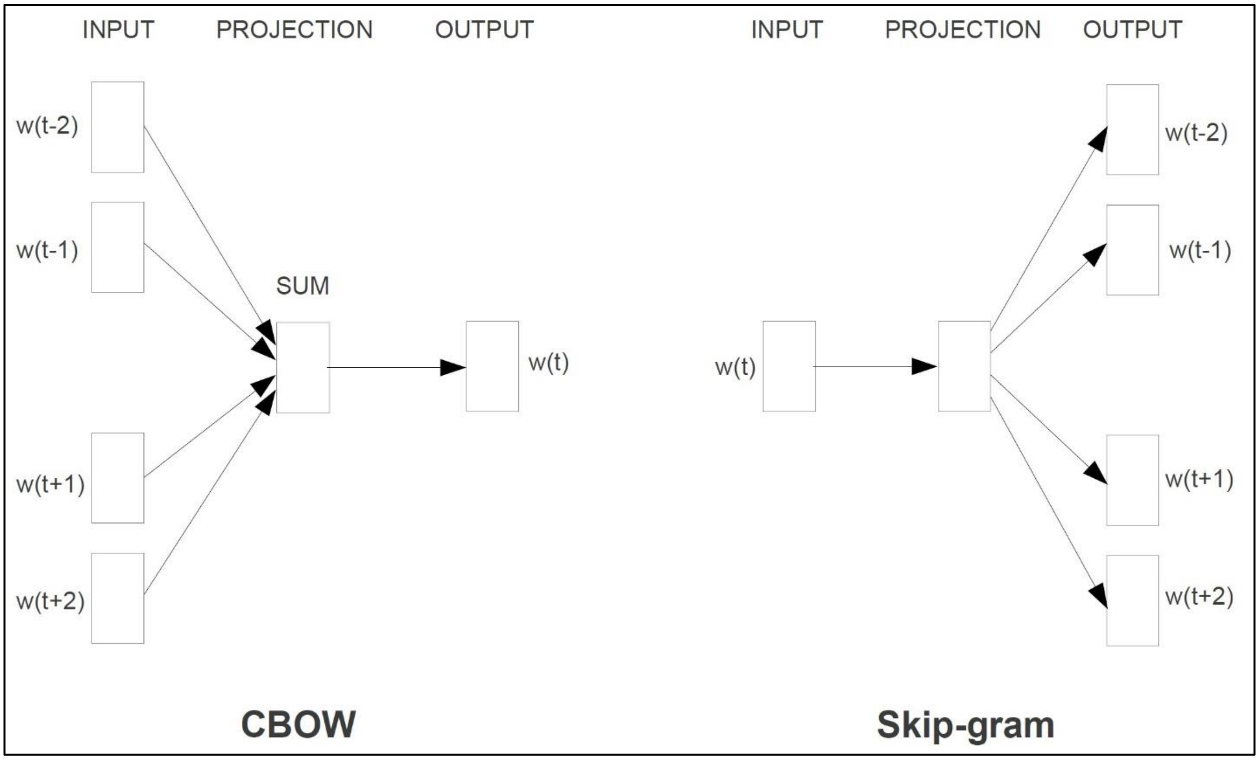
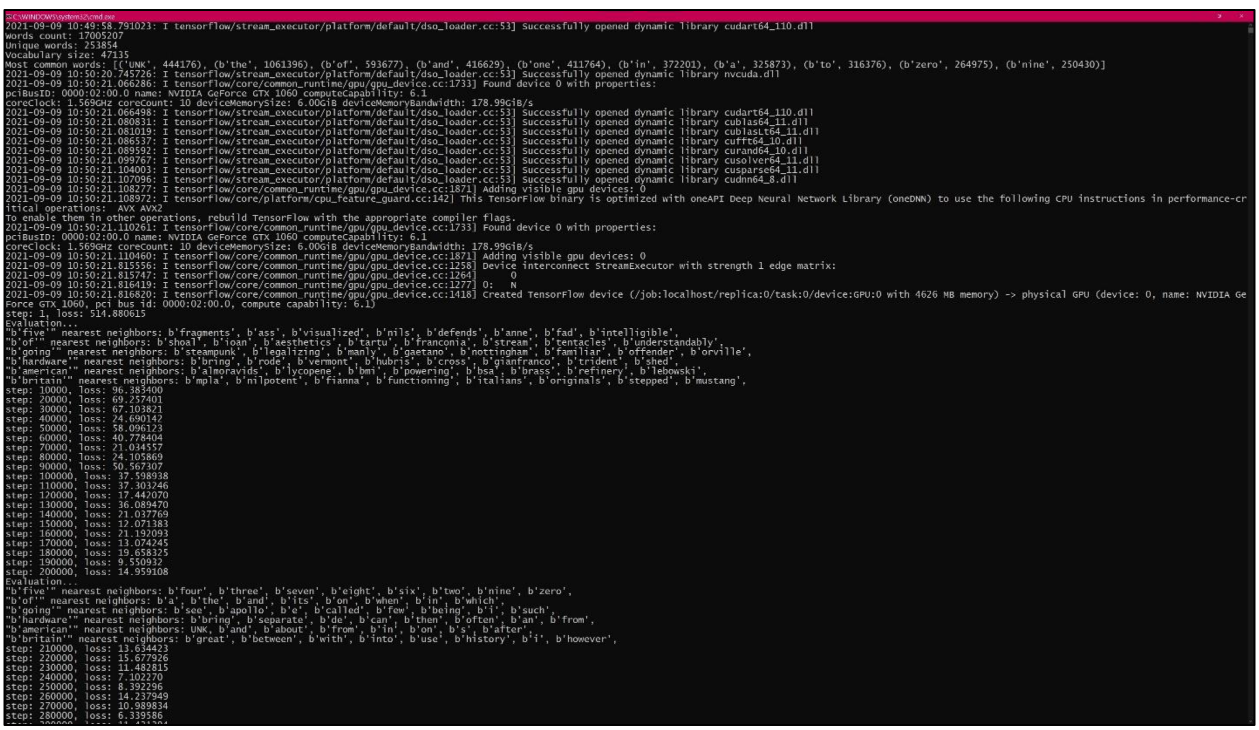
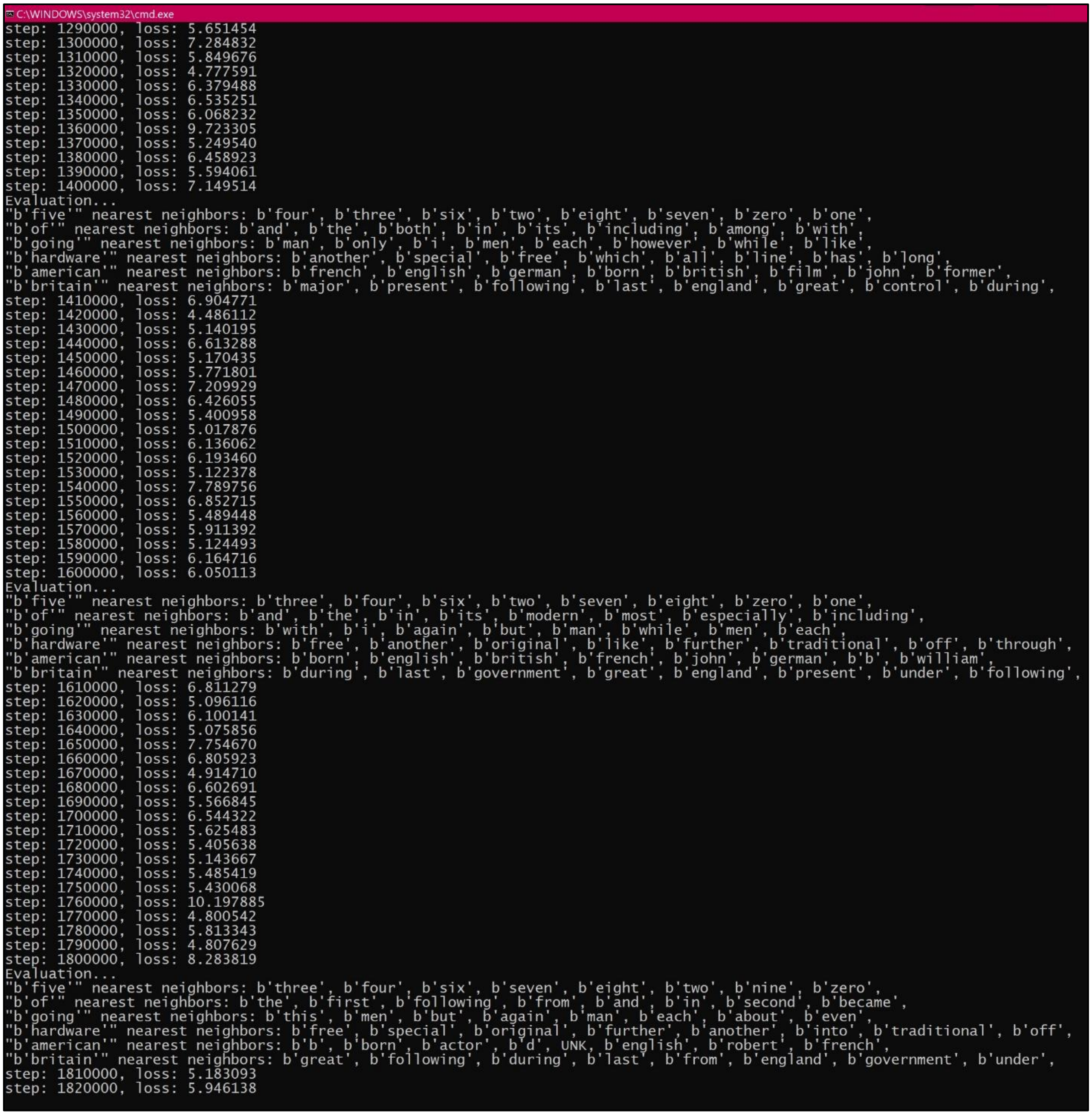


Figure 6. The CBOW and SG models. Adapted from "Efficient Estimation of Word Representations in Vector Space," by Mikolov *et al.*, 2013, *ArXiv:1301.3781 [Cs]*, p. 5.



```
2021-09-09 10:49:58.791023: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cudart64_110.dll
Words count: 17005207
Unique words: 253854
Vocabulary size: 47135
Most common words: [('UNK', 444176), (b'the', 1061396), (b'of', 593677), (b'and', 416629), (b'one', 411764), (b'in', 372201), (b'a', 325873), (b'to', 316376), (b'zero', 264975), (b'nine', 250430)]
2021-09-09 10:50:20.745726: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library nvcuda.dll
2021-09-09 10:50:21.066286: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1733] Found device 0 with properties:
pciBusID: 0000:02:00.0 name: NVIDIA GeForce GTX 1060 computeCapability: 6.1
coreClock: 1.569GHz coreCount: 10 deviceMemorySize: 6.00GiB deviceMemoryBandwidth: 178.99GiB/s
2021-09-09 10:50:21.066498: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cudart64_110.dll
2021-09-09 10:50:21.080831: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cublas64_11.dll
2021-09-09 10:50:21.081019: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cublasLt64_11.dll
2021-09-09 10:50:21.086537: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cufft64_10.dll
2021-09-09 10:50:21.089592: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library curand64_10.dll
2021-09-09 10:50:21.099767: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cusolver64_11.dll
2021-09-09 10:50:21.104003: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cusparse64_11.dll
2021-09-09 10:50:21.107096: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library cudnn64_8.dll
2021-09-09 10:50:21.108277: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1871] Adding visible gpu devices: 0
2021-09-09 10:50:21.108972: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-cr
itical operations:  AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2021-09-09 10:50:21.110261: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1733] Found device 0 with properties:
pciBusID: 0000:02:00.0 name: NVIDIA GeForce GTX 1060 computeCapability: 6.1
coreClock: 1.569GHz coreCount: 10 deviceMemorySize: 6.00GiB deviceMemoryBandwidth: 178.99GiB/s
2021-09-09 10:50:21.110460: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1871] Adding visible gpu devices: 0
2021-09-09 10:50:21.815556: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1258] Device interconnect StreamExecutor with strength 1 edge matrix:
2021-09-09 10:50:21.815747: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1264]      0
2021-09-09 10:50:21.816419: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 0:   N
2021-09-09 10:50:21.816820: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1418] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 4626 MB memory) -> physical GPU (device: 0, name: NVIDIA Ge
Force GTX 1060, pci bus id: 0000:02:00.0, compute capability: 6.1)
step: 1, loss: 514.880615
Evaluation...
"b'five'" nearest neighbors: b'fragments', b'ass', b'visualized', b'nils', b'defends', b'anne', b'fad', b'intelligible',
"b'of'" nearest neighbors: b'shoal', b'ioan', b'aesthetics', b'tartu', b'franconia', b'stream', b'tentacles', b'understandably',
"b'going'" nearest neighbors: b'steampunk', b'legalizing', b'manly', b'gaetano', b'nottingham', b'familiar', b'offender', b'orville',
"b'hardware'" nearest neighbors: b'bring', b'rode', b'vermont', b'hubris', b'cross', b'gianfranco', b'trident', b'shed',
"b'american'" nearest neighbors: b'almoravids', b'lycopene', b'bmi', b'powering', b'bsa', b'brass', b'refinery', b'lebowski',
"b'britain'" nearest neighbors: b'mpla', b'nilpotent', b'fianna', b'functioning', b'italians', b'originals', b'stepped', b'mustang',
step: 10000, loss: 96.383400
step: 20000, loss: 69.257401
step: 30000, loss: 67.103821
step: 40000, loss: 24.690142
step: 50000, loss: 58.096123
step: 60000, loss: 40.778404
step: 70000, loss: 21.034557
step: 80000, loss: 24.105869
step: 90000, loss: 50.567307
step: 100000, loss: 37.598938
step: 110000, loss: 37.303246
step: 120000, loss: 17.442070
step: 130000, loss: 36.089470
step: 140000, loss: 21.037769
step: 150000, loss: 12.071383
step: 160000, loss: 21.192093
step: 170000, loss: 13.074245
step: 180000, loss: 19.658325
step: 190000, loss: 9.550932
step: 200000, loss: 14.959108
Evaluation...
"b'five'" nearest neighbors: b'four', b'three', b'seven', b'eight', b'six', b'two', b'nine', b'zero',
"b'of'" nearest neighbors: b'a', b'the', b'and', b'its', b'on', b'when', b'in', b'which',
"b'going'" nearest neighbors: b'see', b'apollo', b'e', b'called', b'few', b'being', b'i', b'such',
"b'hardware'" nearest neighbors: b'bring', b'separate', b'de', b'can', b'then', b'often', b'an', b'from',
"b'american'" nearest neighbors: UNK, b'and', b'about', b'from', b'in', b'on', b's', b'after',
"b'britain'" nearest neighbors: b'great', b'between', b'with', b'into', b'use', b'history', b'i', b'however',
step: 210000, loss: 13.634423
step: 220000, loss: 15.677926
step: 230000, loss: 11.482815
step: 240000, loss: 7.102270
step: 250000, loss: 8.392296
step: 260000, loss: 14.237949
step: 270000, loss: 10.989834
step: 280000, loss: 6.339586
```

Figure 7. Program output (part 1)



```
C:\WINDOWS\system32\cmd.exe
step: 1290000, loss: 5.651454
step: 1300000, loss: 7.284832
step: 1310000, loss: 5.849676
step: 1320000, loss: 4.777591
step: 1330000, loss: 6.379488
step: 1340000, loss: 6.535251
step: 1350000, loss: 6.068232
step: 1360000, loss: 9.723305
step: 1370000, loss: 5.249540
step: 1380000, loss: 6.458923
step: 1390000, loss: 5.594061
step: 1400000, loss: 7.149514
Evaluation...
"b'five'" nearest neighbors: b'four', b'three', b'six', b'two', b'eight', b'seven', b'zero', b'one',
"b'of'" nearest neighbors: b'and', b'the', b'both', b'in', b'its', b'including', b'among', b'with',
"b'going'" nearest neighbors: b'man', b'only', b'i', b'men', b'each', b'however', b'while', b'like',
"b'hardware'" nearest neighbors: b'another', b'special', b'free', b'which', b'all', b'line', b'has', b'long',
"b'american'" nearest neighbors: b'french', b'english', b'german', b'born', b'british', b'film', b'john', b'former',
"b'britain'" nearest neighbors: b'major', b'present', b'following', b'last', b'england', b'great', b'control', b'during',
step: 1410000, loss: 6.904771
step: 1420000, loss: 4.486112
step: 1430000, loss: 5.140195
step: 1440000, loss: 6.613288
step: 1450000, loss: 5.170435
step: 1460000, loss: 5.771801
step: 1470000, loss: 7.209929
step: 1480000, loss: 6.426055
step: 1490000, loss: 5.400958
step: 1500000, loss: 5.017876
step: 1510000, loss: 6.136062
step: 1520000, loss: 6.193460
step: 1530000, loss: 5.122378
step: 1540000, loss: 7.789756
step: 1550000, loss: 6.852715
step: 1560000, loss: 5.489448
step: 1570000, loss: 5.911392
step: 1580000, loss: 5.124493
step: 1590000, loss: 6.164716
step: 1600000, loss: 6.050113
Evaluation...
"b'five'" nearest neighbors: b'three', b'four', b'six', b'two', b'seven', b'eight', b'zero', b'one',
"b'of'" nearest neighbors: b'and', b'the', b'in', b'its', b'modern', b'most', b'especially', b'including',
"b'going'" nearest neighbors: b'with', b'i', b'again', b'but', b'man', b'while', b'men', b'each',
"b'hardware'" nearest neighbors: b'free', b'another', b'original', b'like', b'further', b'traditional', b'off', b'through',
"b'american'" nearest neighbors: b'born', b'english', b'british', b'french', b'john', b'german', b'b', b'william',
"b'britain'" nearest neighbors: b'during', b'last', b'government', b'great', b'england', b'present', b'under', b'following',
step: 1610000, loss: 6.811279
step: 1620000, loss: 5.096116
step: 1630000, loss: 6.100141
step: 1640000, loss: 5.075856
step: 1650000, loss: 7.754670
step: 1660000, loss: 6.805923
step: 1670000, loss: 4.914710
step: 1680000, loss: 6.602691
step: 1690000, loss: 5.566845
step: 1700000, loss: 6.544322
step: 1710000, loss: 5.625483
step: 1720000, loss: 5.405638
step: 1730000, loss: 5.143667
step: 1740000, loss: 5.485419
step: 1750000, loss: 5.430068
step: 1760000, loss: 10.197885
step: 1770000, loss: 4.800542
step: 1780000, loss: 5.813343
step: 1790000, loss: 4.807629
step: 1800000, loss: 8.283819
Evaluation...
"b'five'" nearest neighbors: b'three', b'four', b'six', b'seven', b'eight', b'two', b'nine', b'zero',
"b'of'" nearest neighbors: b'the', b'first', b'following', b'from', b'and', b'in', b'second', b'became',
"b'going'" nearest neighbors: b'this', b'men', b'but', b'again', b'man', b'each', b'about', b'even',
"b'hardware'" nearest neighbors: b'free', b'special', b'original', b'further', b'another', b'into', b'traditional', b'off',
"b'american'" nearest neighbors: b'b', b'born', b'actor', b'd', UNK, b'english', b'robert', b'french',
"b'britain'" nearest neighbors: b'great', b'following', b'during', b'last', b'from', b'england', b'government', b'under',
step: 1810000, loss: 5.183093
step: 1820000, loss: 5.946138
```

Figure 8. Program output (part 2)

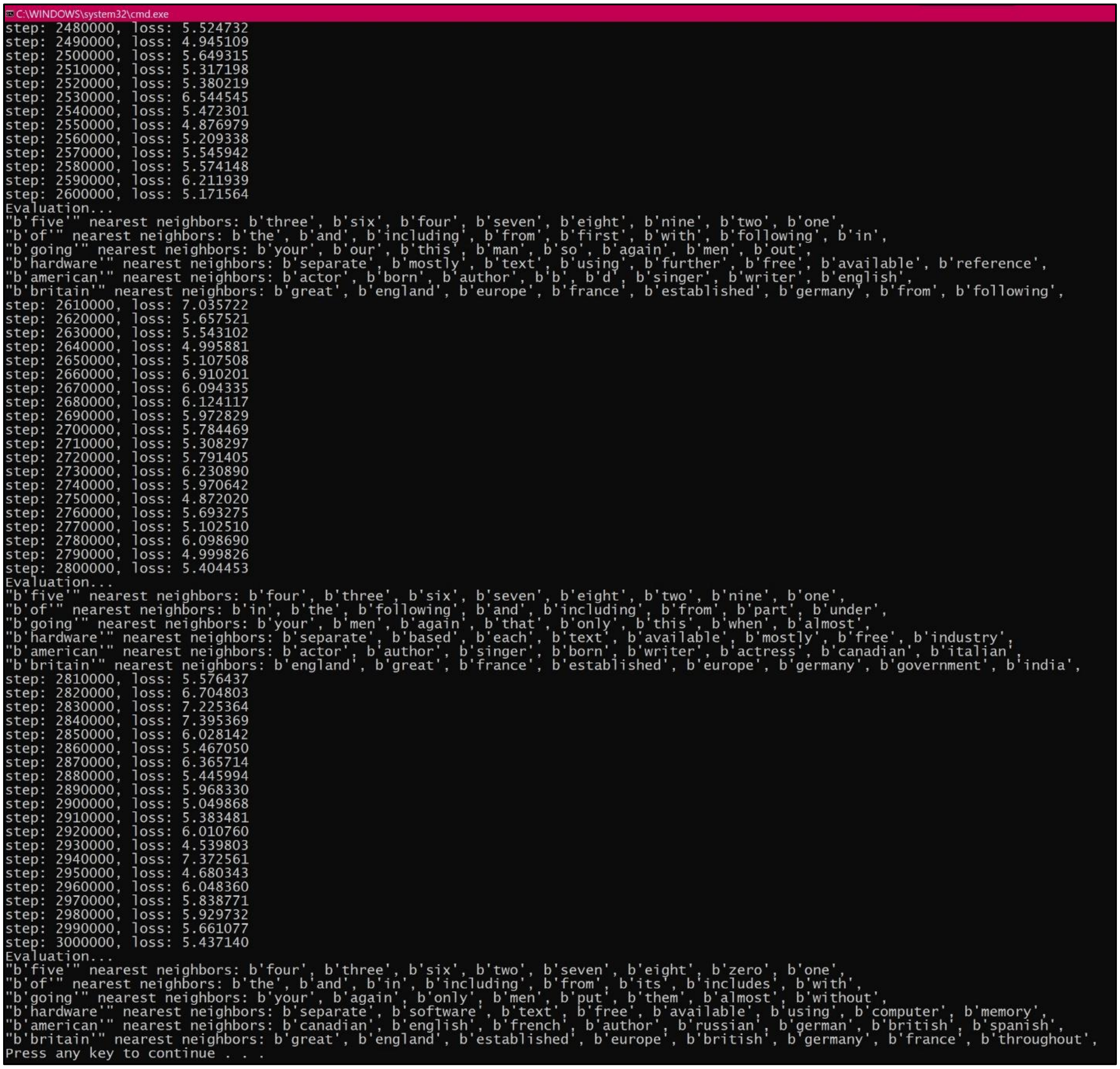


Figure 9. Program output (part 3)

Table of Contents

Abstract 2

Introduction to Word Representation 11

The Distributional Hypothesis 11

Word Embeddings and Their Applications 12

The word2vec Algorithm 12

Skip Grams and Continuous Bag-of-Words 12

Hierarchical Softmax and Negative Sampling 13

Dataset Description 13

Setting Parameters and Pre-processing Data 14

Data Preparation 14

Model Training 14

Model Evaluation and Results 15

Conclusion 15

References 17

OPTION #1: Build a TensorFlow Demo

This paper explores the process of building a *word2vec* word embedding (WE) model using Wikipedia data and TensorFlow (TF) 2.0+. TensorFlow, as defined by Abadi *et al.* (2016), is an interface for expressing machine learning algorithms. The installation of TensorFlow, including GPU support, was successful and without issues, as shown in Figure 1. The researcher followed a "TensorFlow-Examples" tutorial to build a WE model. After downloading and running the Jupyter Notebook file, the researcher examined the tutorial more closely using Visual Studio. Figures 2 – 5 display the demo code, while 7 – 9 illustrate the program's output. Some code statements required updates for compatibility with Python 3. This paper overviews word representations and WEs and delves into the demo's model and dataset details.

Introduction to Word Representation

Word representation lies at the core of natural language processing (NLP) (Levy & Goldberg, 2014). However, many contemporary NLP systems treat words as atomic units, lacking representations that capture the similarities between words (Mikolov *et al.*, 2013a). Consequently, these systems are often simple and robust but inadequate for numerous tasks and prone to poor generalization. For example, when employing symbolic representations where discrete symbols denote each word, it becomes impossible to discern the relationship between "coffee" and "water." Furthermore, although "water" represents a strong argument for the verb "drink," we cannot infer that "coffee" serves as an equally strong argument.

The Distributional Hypothesis

To address these limitations, researchers aim to develop word representations that convey semantic and syntactic similarities (Levy *et al.*, 2015). Harris (1954, as cited in Levy *et al.*, 2015) introduced the *distributional hypothesis*, which has since become the foundation for numerous paradigms designed to acquire such representations. According to this hypothesis, words that appear in similar contexts share similar meanings.

Word Embeddings and Their Applications

Word embeddings (WEs) represent words as dense, lower-dimensional vectors derived from neural network-inspired training methods and recent techniques, capturing both semantic and syntactic relationships between words (Levy & Goldberg, 2014; Rothe & Schütze, 2015). Although the dimensions of WEs are considered opaque, making it challenging to attribute specific meanings (Levy *et al.*, 2015), the geometric distances between these *d*-dimensional vectors accurately reflect word relationships (Almeida & Xexéo, 2019; Bamler & Mandt, 2017).

For example, Mikolov *et al.* (2013a) discovered that simple algebraic operations on WE vectors, such as vector("King") - vector("Man") + vector("Woman"), yield a vector closest to the word "Queen." Consequently, WEs prove valuable in various NLP tasks, including semantic parsing, sentiment analysis (Bamler & Mandt, 2017), part-of-speech tagging, and named-entity recognition (Wang *et al.*, 2019). These use cases illustrate how the researcher can utilize WEs in his chatbot development project.

The word2vec Algorithm

Word2vec, an algorithm introduced by Mikolov *et al.* (2013a), generates word embeddings (WEs) that scale efficiently with large datasets and deliver high-quality results (Kusner *et al.*, 2015).

Skip Grams and Continuous Bag-of-Words

According to "Stanford University," word2vec utilizes either skip-grams (SG) or continuous bag-of-words (CBOW) algorithms to create WEs, along with hierarchical softmax or negative sampling methods for calculating probability distributions. While CBOW predicts the current word based on context words, SG predicts surrounding words using the current word (Mikolov *et al.*, 2013a).

Hierarchical Softmax and Negative Sampling

This paper focuses on the SG with a negative sampling model, an unsupervised, state-of-the-art WE technique (Kusner *et al.*, 2015; Levy *et al.*, 2015). Levy *et al.* (2015) explain that the unsupervised SG with a negative sampling model associates each target word (w) with a vector (v_w) and each context word (c) with a vector (v_c). The model learns to maximize the dot product ($v_c \cdot v_w$) for “good” word-context pairs by treating each vector entry as a learnable parameter.

The negative sampling objective aims to maximize the log probability of observed word-context pairs in the data. To avoid a trivial solution of setting vc=vw, the objective includes word-context pairs with low probabilities. For instance, with training data “The quick brown fox jumps,” Jordan Boyd-Graber (2019) suggests corrupting the sample by replacing “brown” with a random word, such as “transparent.” The model aims to set vector values so that the dot product between focus and context words is high in the former case and low for the corrupted word-context pairs.

Optimizing with Stochastic Gradient Descent

Surprisingly, optimizing this negative sampling objective with stochastic-gradient descent (SGD) yields WEs with remarkable similarity for words in similar contexts (Levy *et al.*, 2015).

Dataset Description

The demo in this paper implements the *word2vec* algorithm to create word embeddings (WEs) from a Wikipedia data dump using TensorFlow 2.0+. Mahoney (2011) describes the *text8* dataset as a 100 MB cleaned-up version of a Wikipedia data dump from 2006. The lowercase dataset comprises English letters and spaces (Tomar, 2019). The demo reports dataset details, such as the number of words and unique words and the ten most frequently occurring words.

Setting Parameters and Pre-processing Data

Before processing, the program imports necessary libraries and sets various training, model, and evaluation parameters. It is designed to return the eight nearest neighbors (NNs) of six test words, with embedding vector dimensions set to 200, a maximum vocabulary size of 50,000, and a minimum word occurrence threshold of 10. Hyperparameter tuning could potentially improve the model's performance.

Data Preparation

The program downloads the *text8.zip* file, processes it, and creates a dictionary object containing the frequency counts for the 50,000 most frequently occurring words. Infrequent words are removed, reducing the vocabulary size to 47,135. The program counts "unknown" words, adds word indices to the data list, and creates two dictionary objects for converting words between string and numerical representations.

Afterward, the program outputs the dataset information mentioned earlier and ensures that specific functions are computed on the CPU rather than the GPU, as not all operations are GPU-compatible. The program creates an embedding variable with randomly generated values, where each row represents a WE vector. It also generates weight and bias variables for calculating the Noise Contrastive Estimation (NCE) loss and defines the SGD optimizer, setting the learning rate parameter to 0.1 (Mikolov et al., 2013b). Additionally, the program creates a test dataset that converts each testing word to its corresponding index.

Model Training

Next, the program trains the model for a specified number of steps, set at 3,000,000 in the demo. Training begins with creating feature (*x*) and target (*y*) variables for the data using the vocabulary. The program employs a context window of size seven, incrementally moving through each word in the vocabulary. Explanatory and target variables are generated by selecting the center word from each context window as the model's input and randomly choosing two context words from the same window as ground truth target variables. This results in an unsupervised learning model.

During training, the model applies the SGD optimization process to the data, converting each word in the focus and context vectors into their distributed representations. It then computes the average NCE loss for each batch using the weight and bias vectors and randomly sampling 64 negative classes.

Model Evaluation and Results

Subsequently, the model computes gradients for each batch and updates its weights, biases, and embeddings based on these gradients. The program reports the model's loss after every 10,000th step. It evaluates the skip-gram model at every 200,000th step by converting the six test words to their corresponding embeddings and calculating the cosine similarity between each test set embedding and all other embeddings. These cosine similarities are ranked in descending order, and the eight nearest neighbors (NNs) for each test word are output.

Comparing the first evaluation output to the three millionth step (Figures 7 – 9) reveals the program's increased accuracy. For example, the NNs for "Britain" at the last training step, which include "England," "Europe," "British," "Germany," and "France," are more semantically and syntactically related to the target word than the NNs output after step 1.

Conclusion

In summary, this paper explored the concepts of word representations and word embeddings (WEs), delving into the word2vec algorithm and its underlying principles. Utilizing TensorFlow 2.0+ and a Wikipedia dataset, the paper demonstrated the process of constructing WEs through the unsupervised learning techniques of the skip-gram negative-sampling model. The paper also highlighted the potential applications of these techniques in the author's chatbot development project and suggested avenues for enhancing the model and its dataset.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv:1603.04467 [Cs]*. http://arxiv.org/abs/1603.04467

Almeida, F., & Xexéo, G. (2019). Word embeddings: A survey. *ArXiv Preprint ArXiv:1901.09069*.

Bamler, R., & Mandt, S. (2017). Dynamic Word Embeddings. *International Conference on Machine Learning*, 380–389. https://proceedings.mlr.press/v70/bamler17a.html

Jordan Boyd-Graber. (2019, February 17). *Understanding Word2Vec*. https://www.youtube.com/watch?v=QyrUentbkvw

Kusner, M., Sun, Y., Kolkin, N., & Weinberger, K. (2015). From word embeddings to document distances. *International Conference on Machine Learning*, 957–966.

Levy, O., & Goldberg, Y. (2014). Dependency-Based Word Embeddings. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 302–308. https://doi.org/10.3115/v1/P14-2050

Levy, O., Goldberg, Y., & Dagan, I. (2015). Improving Distributional Similarity with Lessons Learned from Word Embeddings. *Transactions of the Association for Computational Linguistics*, *3*, 211–225. https://doi.org/10.1162/tacl_a_00134

Mahoney, M. (2011, September 1). *About the Test Data*. http://mattmahoney.net/dc/textdata.html

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013a). Efficient Estimation of Word Representations in Vector Space. *ArXiv:1301.3781 [Cs]*. http://arxiv.org/abs/1301.3781

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013b). Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems*, *26*. https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

Rothe, S., & Schütze, H. (2015). AutoExtend: Extending Word Embeddings to Embeddings for Synsets and Lexemes. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 1793–1803. https://doi.org/10.3115/v1/P15-1173

Stanford University School of Engineering. (2017, April 3). *Lecture 2 | Word Vector Representations: Word2vec*. https://www.youtube.com/watch?v=ERibwqs9p38

TensorFlow-Examples/tensorflow_v2 at master · aymericdamien/TensorFlow-Examples. (n.d.). GitHub. Retrieved September 7, 2021, from https://github.com/aymericdamien/TensorFlow-Examples

Tomar, D. (2019, September 1). Using benchmark datasets: Character-level Language Modeling. *Medium*. https://dhananjaytomar.medium.com/using-benchmark-datasets-character-level-language-modeling-ef16afa21101

Wang, B., Wang, A., Chen, F., Wang, Y., & Kuo, C.-C. J. (2019). Evaluating Word Embedding Models: Methods and Experimental Results. *APSIPA Transactions on Signal and Information Processing*, *8*, e19. https://doi.org/10.1017/ATSIP.2019.12