README: NLP Assignment 1

-Shubham Patel (112007216)

In this assignment, goal is to train a model over the corpus from below mentioned URL -

```
http://mattmahoney.net/dc/
```

It supports two loss functions, Cross Entropy and NCE loss.

Files :

- **word2vec_basic.py** : Contains main code and generate_batch functions, also all hyper-parameters are hard coded here.
- **loss-func.py** : Contains cross_entropy_loss and nce_loss functions
- **word_analogy.py** : Contains code for word analogy task of finding least and most similar pair. Model is hard coded in this file, make sure to toggle it between cross entropy and NCE before running
- **word2vec_cross_entropy.model** : Model file generated post training post hyper-parameter tuning, with cross_entropy_loss function
- **word2vec_nce.model** : Model file generated post training post hyper-parameter tuning, with NCE_loss function
- **word_analogy_test_predictions_cross_entropy.txt** : Prediction file, obtained by running word_analogy.py on file word_analogy_test.txt, with word2vec_cross_entropy.model
- **word_analogy_test_predictions_nce** : Prediction file, obtained by running word_analogy.py on file word_analogy_test.txt, with word2vec_nce.model
- **Report.pdf** : Contains details on hyper-parameter tuning, analysis, 20 similar words to {first, american, would} and summarization of NCE loss method.

To train the model with Cross Entropy loss function :

> python word2vec_basic.py

To train the model with NCE loss function :

> python word2vec_basic.py nce

To run word_analogy.py :

> python word_analogy.py

Default hyperparameters :

batch_size	skip_window	num_skips	max_num_steps	embedding_size
128	4	8	200000	128

Configuration of submitted models :

Model	batch_size	skip_window	num_skips	max_num_steps
word2vec_nce.model	128	2	4	200000
word2vec_cross_entropy.model	32	4	8	200000

TASK 1: Implement GENERATE_BATCH function

A deque of size = window size (2* **skip_window** +1)is used to implement it. For iterating inside a window to generate batches and labels, below mentioned two approaches were used :

1. Locking the center element (index: **skip_window**) then picking a random unseen element from rest of the window (say, element at index X) and mark it as seen : Adding this skip_window,X pair in batch and label array. Following the same process till **num_skips** iteration.
2. Locking the center element (index: **skip_window**) then picking an element from left and an element from right at a distance of **dist_from_center** : Adding skip_window, right element and skip_window, left element in batches, labels respectively. Follow the same process till **num_skips** addition are made from a window.

However, on accuracy testing over both implementation of both approaches, it was found out that approach 2 is better, thus as final version, approach 2 is applied.

TASK 2: Implement CROSS_ENTROPY_loss function

Cross entropy is implement as per the formula mentioned below :

> **v_c , u_o = inputs, true_w** from function parameters

$$CrossEntropy = -log\left[\frac{(\exp(u_o^T v_c))}{\sum_x\ (\exp(u_x^T v_c))}\right]$$

$$= log(\sum_x\ (\exp(u_x^T v_c))) - log(\exp(u_o^T v_c))$$

$$A = log(\sum_x\ (\exp(u_x^T v_c)))$$

$$B = log(\exp(u_o^T v_c))$$

Computing A :

```
uotvc = tf.matmul(uot, inputs)
exp_uotvc = tf.exp(uotvc)
A = tf.log(exp_uotvc + 0.00000001)
```

Computing B :

```
sigma_exp_uotvc = tf.reduce_sum(exp_uotvc, 1)
B = tf.log(sigma_exp_uotvc + 0.00000001)
```

TASK 3: Implement NCE _loss function

NCE loss is implemented as per the formula mentioned below :

$$J(\theta, Batch) = \sum_{(w_o,w_c)\in Batch} -\left[\log Pr(D=1, w_o|w_c) + \sum_{x\in V^k} \log(1 - Pr(D=1, w_x|w_c))\right]$$

$$\text{where,}$$
$$Pr(D=1, w_o|w_c) = \sigma\left(s(w_o, w_c) - \log\left[kPr(w_o)\right]\right)$$
$$Pr(D=1, w_x|w_c) = \sigma\left(s(w_x, w_c) - \log\left[kPr(w_x)\right]\right)$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\text{and}$$
$$s(w_o, w_c) = (u_c^T u_o) + b_o$$

Here, we do not need to do the outer summation for entire batch, that is handled outside the function in *word2vec_basic.py* file.

Computing :

Part1:

$$log(P_r(D=1, w_x|w_c))$$

Steps:

$$s(w_o, w_c) = (u_c^T u_o) + b_o$$

uo: labels is used as a lookup over weights

uc : inputs

bo : labels is used as a lookup over biases

```
uo = tf.reshape(tf.nn.embedding_lookup(weights, labels), [-1, weights.shape[1]])
bo = tf.nn.embedding_lookup(biases, labels)
ucuo = tf.reduce_sum(tf.multiply(inputs, uo), 1)
ucuo = tf.reshape(ucuo, [ucuo.shape[0], 1])
soc = tf.add(ucuo, bo)
```

$$log(kP_r(w_o))$$

pwo : labels is used as a lookup over unigram_prob

```
pwo = (tf.nn.embedding_lookup([unigram_prob], labels))
k = len(sample)
logkpwo = tf.log(tf.scalar_mul(k, pwo) + 0.00000001)
```

$$P_r(D = 1, w_o|w_c) = \sigma[s(w_o, w_c) - log(kP_r(w_o))]$$

```
part1 = tf.subtract(soc, logkpwo)
part1 = tf.sigmoid(part1)
```

$$log(P_r(D = 1, w_x|w_c))$$

```
part1 = tf.log(part1 + 0.00000001)
```

Part 2:

$$\sum_x log(1 - P_r(D = 1, w_x|w_c))$$

Steps :

$$s(w_x, w_c) = (w_c^T w_x) + b_x$$

wx: sample is used as a lookup over weights

wc : inputs

bx : sample is used as a lookup over biases

```
wx = tf.nn.embedding_lookup(weights, sample)
wx = tf.reshape(wx, [sample.shape[0], -1])
bx = tf.nn.embedding_lookup(biases, sample)
bx = tf.reshape(bx, [bx.shape[0], 1])
wcwx = tf.matmul(inputs, tf.transpose(wx))
sxc = tf.add(wcwx, tf.transpose(bx))
```

$$log(kP_r(w_x))$$

pwx : labels is used as a lookup over unigram_prob

```
    pwx = tf.nn.embedding_lookup([unigram_prob], sample)
    pwx = tf.reshape(pwx, [pwx.shape[0], 1])
    logkpwx = tf.log(tf.scalar_mul(k, pwx) + 0.00000001)
```

$$P_r(D = 1, w_x|w_c) = \sigma[s(w_x, w_c) - log(kP_r(w_x))]$$

```
    pwc = tf.subtract(sxc, tf.transpose(logkpwx))
    pwc = tf.nn.sigmoid(pwc)
    part2 = tf.subtract(tf.ones([1, len(sample)]), pwc)
```

$$\sum_x log(1 - P_r(D = 1, w_x|w_c))$$

```
    part2 = tf.subtract(tf.ones([1, len(sample)]), pwc)
    part2 = tf.reduce_sum(tf.log(part2 + 0.00000001), 1)
```

Combining part1 and part2 :

$$-\left[log(P_r(D = 1, w_x|w_c)) + \sum_x log(1 - P_r(D = 1, w_x|w_c))\right]$$

```
    final_prob = tf.negative(tf.add(part1, part2))
```

TASK 4: Word Analogy

For each word provided in file :

dev - *word_analogy_dev.txt*

test - *word_analogy_test.txt*

Embedding/wordvec, as per the model used, is found out by code below :

```
v1 = embeddings[dictionary[word_id]]
```

For the first 3 pairs provided, we are taking wordvec of each word, in pairs and taking the difference and then averaging all the differences to get a averagevec :

```
    for j in range(0, 3):
        diff_set.append(
            np.subtract(embeddings[dictionary[given_set[i][j][0]]],
                        embeddings[dictionary[given_set[i][j][1]]]))

    average_vec = np.average(diff_set)
```

For the next 4 pairs in each line, we are again getting difference between both, then computing cosine difference of this difference with average_vec. Then similarity is calculated as : 1 - cosine_distance. Later, with this similarity we are finding least and most similar pair.

```
    for j in range(0, 4):
        diff_unknown = np.subtract(embeddings[dictionary[unknown_set[i][j][0]]],
                                   embeddings[dictionary[unknown_set[i][j][1]]])
        cosine_diff = distance.cosine(average_vec, diff_unknown)
        similarity = 1-cosine_diff
        if (similarity >= max_diff):
            max_diff = similarity
            index_max = j
        elif (similarity < min_diff):
            min_diff = similarity
            index_min = j
```

Finally result is stored in file - **word_analogy_dev_predictions_bymodel.txt**

Experimental Details

	batch_size	skip_window	num_skips	max_num_steps
baseline	128	4	8	1
default	128	4	8	200000
test1	64	4	8	200000
test2	32	4	8	200000
test3	128	8	16	200000
test4	128	2	4	200000
test5	128	8	8	200000
test6	128	4	8	300000
test7	128	4	8	100000