# Programming Assignment 1: Learning Distributed Word Representations

**Due Date:** Feb. 7, 2018
**TA:** Jing Yao Li (csc321staff@cs.toronto.edu)
Based on an assignment by George Dahl

**Submission:** You must submit two files through MarkUs[1]: a PDF file containing your writeup, titled `a1-writeup.pdf`, and your code file `language_model.py`. Your writeup must be typeset using LaTeX.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

## Introduction

In this assignment we will make neural networks learn about words. One way to do this is to train a network that takes a sequence of words as input and learns to predict the word that comes next.

This assignment will ask you to implement the backpropagation computations for a neural language model and then run some experiments to analyze the learned representation. The amount of code you have to write is very short — only about 5 lines — but each line will require you to think very carefully. You will need to derive the updates mathematically, and then implement them using matrix and vector operations in NumPy.

## Part 0: Getting set up

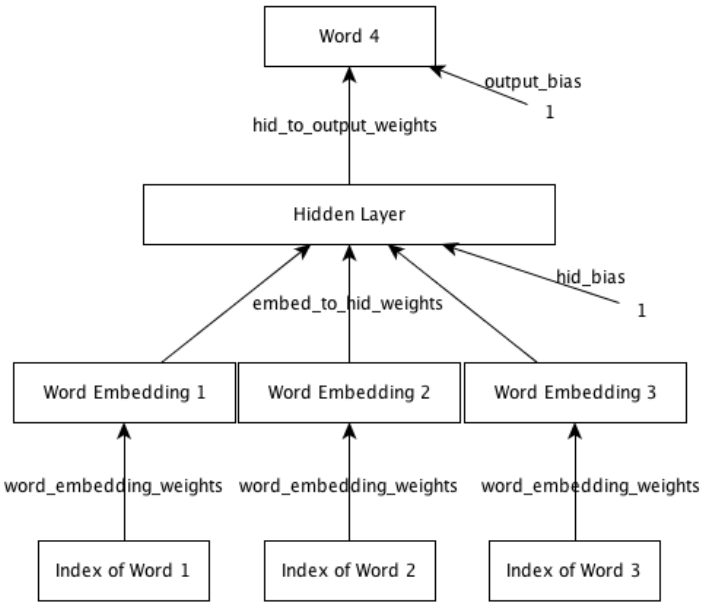
The first step is to install Python and the required libraries. The directions are on the course web site (`http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/`). That page also links to some Python and NumPy tutorials, which you may wish to read before doing this assignment.

[1] `https://markus.teach.cs.toronto.edu/csc321-2018-01`
[2] `http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/syllabus.pdf`

## Part 1: Network architecture (2pts)

In this assignment, we will train a neural language model like the one we covered in lecture. It receives as input 3 consecutive words, and its aim is to predict a distribution over the next word (the *target* word). We train the model using the cross-entropy criterion, which is equivalent to maximizing the probability it assigns to the targets in the training set. Hopefully it will also learn to make sensible predictions for sequences it hasn't seen before.

The model architecture is as follows:




The network consists of an input layer, embedding layer, hidden layer and output layer. The input consists of a sequence of 3 consecutive words, given as integer valued indices. (*I.e.*, the 250 words in our dictionary are arbitrarily assigned integer values from 0 to 249.) The embedding layer maps each word to its corresponding vector representation. This layer has $3 \times D$ units, where $D$ is the embedding dimension, and it essentially functions as a lookup table. We share the *same* lookup table between all 3 positions, *i.e.* we don't learn a separate word embedding for each context position. The embedding layer is connected to the hidden layer, which uses a logistic nonlinearity. The hidden layer in turn is connected to the output layer. The output layer is a softmax over the 250 words.

As a warm-up, please answer the following questions, each worth 1 point out of 10.

1. As above, assume we have 250 words in the dictionary and use the previous 3 words as inputs. Suppose we use a 16-dimensional word embedding and a hidden layer with 128 units. The

trainable parameters of the model consist of 3 weight matrices and 2 sets of biases. What is the total number of trainable parameters in the model? Which part of the model has the largest number of trainable parameters?

2. Another method for predicting the next word is an *n-gram model*, which was mentioned in Lecture 7. If we wanted to use an n-gram model with the same context length as our network, we'd need to store the counts of all possible 4-grams. If we stored all the counts explicitly, how many entries would this table have?

## Starter code and data

Download and extract the archive from the course web page `http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/a1-code.zip`.

Look at the file `raw_sentences.txt`. It contains the sentences that we will be using for this assignment. These sentences are fairly simple ones and cover a vocabulary of only 250 words.

We have already extracted the 4-grams from this dataset and divided them into training, validation, and test sets. To inspect this data, run the following within IPython:

```
import cPickle
data = cPickle.load(open('data.pk', 'rb'))
```

Now `data` is a Python dict which contains the vocabulary, as well as the inputs and targets for all three splits of the data. `data['vocab']` is a list of the 250 words in the dictionary; `data['vocab'][0]` is the word with index 0, and so on. `data['train_inputs']` is a $372,500 \times 3$ matrix where each row gives the indices of the 3 context words for one of the $372,500$ training cases. `data['train_targets']` is a vector giving the index of the target word for each training case. The validation and test sets are handled analogously.

Now look at the file `language_model.py`, which contains the starter code for the assignment. Even though you only have to modify two specific locations in the code, you may want to read through this code before starting the assignment. There are three classes defined in this file:

- `Params`, a class which represents the trainable parameters of the network, *i.e.* all of the edges in the above figure. This class has five fields:
  - `word_embedding_weights`, a matrix of size $N_V \times D$, where $N_V$ is the number of words in the vocabulary and $D$ is the embedding dimension.
  - `embed_to_hid_weights`, a matrix of size $N_H \times 3D$, where $N_H$ is the number of hidden units. The first $D$ columns represent connections from the embedding of the first context word, the next $D$ columns for the second context word, and so on.
  - `hid_bias`, a vector of length $N_H$
  - `hid_to_output_weights`, a matrix of size $N_V \times N_H$

  - `output_bias`, a vector of length $N_V$

- `Activations`, a class which represents the activations of all the networks's units (*i.e.* the boxes in the above figure) on a batch of data. This has three fields:
  - `embedding_layer`, a matrix of $B \times 3D$ matrix (where $B$ is the batch size), representing the activations for the embedding layer on all the cases in a batch. The first $D$ columns represent the embeddings for the first context word, and so on.
  - `hidden_layer`, a $B \times N_H$ matrix representing the hidden layer activations for a batch
  - `output_layer`, a $B \times N_V$ matrix representing the output layer activations for a batch
- `Model`, a class for the language model itself, which includes a lot of routines related to training and making predictions. These methods will be discussed in later sections.

## Part 2: Training the model (4pts)

In the first part of the assignment, you implement a method which computes the gradient using backpropagation. To start you out, the `Model` class contains several important methods used in training:

- `compute_activations` computes the activations of all units on a given input batch
- `compute_loss` computes the total cross-entropy loss on a mini-batch
- `evaluate` computes the average cross-entropy loss for a given set of inputs and targets

You will need to complete the implementation of two additional methods which are needed for training:

- `compute_loss_derivative` computes the derivative of the loss function with respect to the output layer inputs. In other words, if $C$ is the cost function, and the softmax computation is

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}},$$

  this function should compute a $B \times N_V$ matrix where the entries correspond to the partial derivatives $\partial C / \partial z_i$.

- `back_propagate` is the function which computes the gradient of the loss with respect to model parameters using backpropagation. It uses the derivatives computed by `compute_loss_derivative`. Some parts are already filled in for you, but you need to compute the matrices of derivatives for `embed_to_hid_weights`, `hid_bias`, `hid_to_output_weights`, and `output_bias`. These matrices have the same sizes as the parameter matrices (see previous section).

In order to implement backpropagation efficiently, you need to express the computations in terms of matrix operations, rather than `for` loops. You should first work through the derivatives on pencil and paper. First, apply the chain rule to compute the derivatives with respect to individual units, weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You should be able to express all of the required computations using only matrix multiplication, matrix transpose, and elementwise operations — no `for` loops! If you want inspiration, read through the code for `Model.compute_activations` and try to understand how the matrix operations correspond to the computations performed by all the units in the network.

To make your life easier, we have provided the routine `checking.check_gradients`, which checks your gradients using finite differences. You should make sure this check passes before continuing with the assignment.

Once you've implemented the gradient computation, you'll need to train the model. The function `train` in `language_model.py` implements the main training procedure. It takes two arguments:

- `embedding_dim`: The number of dimensions in the distributed representation.
- `num_hid`: The number of hidden units

For example, execute the following:

```
model = language_model.train(16, 128)
```

As the model trains, the script prints out some numbers that tell you how well the training is going. It shows:

- The cross entropy on the last 100 mini-batches of the training set. This is shown after every 100 mini-batches.
- The cross entropy on the entire validation set every 1000 mini-batches of training.

At the end of training, this function shows the cross entropies on the training, validation and test sets. It will return a `Model` instance.

To convince us that you have correctly implemented the gradient computations, please include the following with your assignment submission:

- A printout of your code for the two methods you had to implement: `Model.compute_loss_derivative` and `Model.back_propagate`. *Please do not include any other code.*
- The output of the function `checking.print_gradients`. This prints out part of the gradients for a partially trained network which we have provided, and we will check them against the correct outputs. *Important:* make sure to give the output of `checking.print_gradients`, not `checking.check_gradients`.

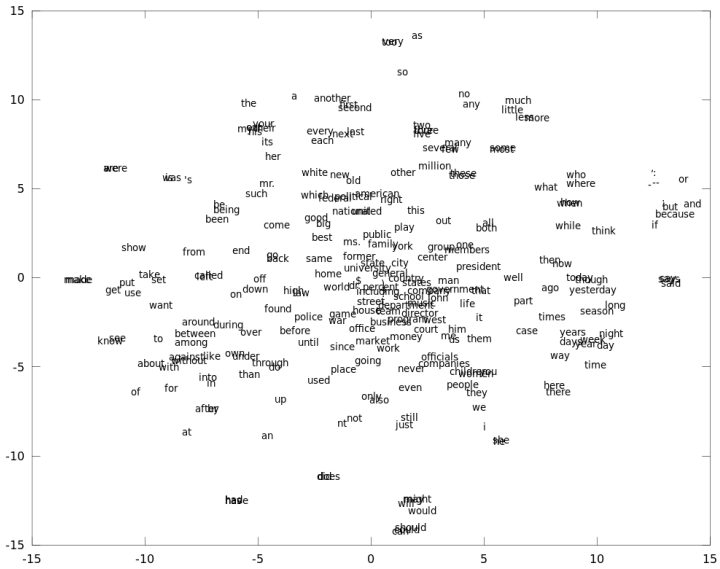
This is worth 4 points: 1 for the loss derivatives, 1 for the bias gradients, and 2 for the weight gradients. Since we gave you a gradient checker, you have no excuse for not getting full points on this part.

## Part 3: Analysis (4pts)

In this part, you will analyze the representation learned by the network. You should first train a model with a 16-dimensional embedding and 128 hidden units, as discussed in the previous section; you'll use this trained model for the remainder of this section. **Important:** if you've made any fixes to your gradient code, you must reload the `language_model` module and then re-run the training procedure. Python does not reload modules automatically, and you don't want to accidentally analyze an old version of your model.

These methods of the `Model` class can be used for analyzing the model after the training is done.

- `tsne_plot` creates a 2-dimensional embedding of the distributed representation space using an algorithm called t-SNE. (You don't need to know what this is for the assignment, but we may cover it later in the course.) Nearby points in this 2-D space are meant to correspond to nearby points in the 16-D space. From the learned model, you can create pictures that look like this:




- `display_nearest_words` lists the words whose embedding vectors are nearest to the given word
- `word_distance` computes the distance between the embeddings of two words

- `predict_next_word` shows the possible next words the model considers most likely, along with their probabilities

Using these methods, please answer the following questions, each of which is worth 1 point.

1. Pick three words from the vocabulary that go well together (for example, `'government of united'`, `'city of new'`, `'life in the'`, `'he is the'` etc.). Use the model to predict the next word. Does the model give sensible predictions? Try to find an example where it makes a plausible prediction even though the 4-gram wasn't present in the dataset (`raw_sentences.txt`). To help you out, the function `language_model.find_occurrences` lists the words that appear after a given 3-gram in the training set.

2. Plot the 2-dimensional visualization using the method `Model.tsne_plot`. Look at the plot and find a few clusters of related words. What do the words in each cluster have in common? (You don't need to include the plot with your submission.)

3. Are the words `'new'` and `'york'` close together in the learned representation? Why or why not?

4. Which pair of words is closer together in the learned representation: `('government', 'political')`, or `('government', 'university')`? Why do you think this is?

## What you have to submit

For reference, here is everything you need to hand in. See the top of this handout for submission directions.

- A PDF file titled `a1-writeup.pdf` containing the following:
  - Both questions from Part 1
  - The output of `checking.print_gradients()`
  - Answers to all four questions from Part 3
  - Optional: Which part of this assignment did you find the most valuable? The most difficult and/or frustrating?
- Your code file `language_model.py`

This assignment is graded out of 10 points: 2 for Part 1, 4 for Part 2, and 4 for Part 3.