NLP – Assignment 3

Rajneesh Pandey, 106119100

Word Embeddings

Data Preparation

To get started, import and initialize all the libraries we will need.

Data preparation

In the data preparation phase, starting with a corpus of text, we will:

- Clean and tokenize the corpus.
- Extract the pairs of context words and centre word that will make up the training data set for the CBOW model. The context words are the features that will be fed into the model, and the centre words are the target values that the model will learn to predict.
- Create simple vector representations of the context words (features) and centre words (targets) that have be used by the neural network of the CBOW model.

Cleaning and tokenization

To demonstrate the cleaning and tokenization process, consider a corpus that contains emojis and various punctuation signs.

Finally, get rid of numbers and punctuation other than periods, and convert all the remaining tokens to lowercase.

Sliding window of words

Now that we have transformed the corpus into a list of clean tokens, we have slid a window of words across this list. For each window we have extract a centre word and the context words.

The first argument of this function is a list of words (or tokens). The second argument, C, is the context half-size. Recall that for a given centre word, the context words are made of C words to the left and C words to the right of the centre word.

Here we have used this function to extract context words and centre words from a list of tokens. These context and centre words will make up the training set that we will use to train the CBOW model.

The first example of the training set is made of:

- the context words "i", "am", “because” "i",
- and the centre word to be predicted: "happy".

Transforming words into vectors for the training set

To finish preparing the training set, we need to transform the context words and centre words into vectors.

Mapping words to indices and indices to words

The centre words will be represented as one-hot vectors, and the vectors that represent context words are also based on one-hot vectors.

To create one-hot word vectors, we have start by mapping each unique word to a unique integer (or index). We have provided a helper function, get_dict, that creates a Python dictionary that maps words to integers and back.

We have use this dictionary to get the index of a word.

Finally, get the length of either of these dictionaries to get the size of the vocabulary corpus, in other words the number of different words making up the corpus.

We have now group all these steps in a convenient function, which takes as parameters: a word to be encoded, a dictionary that maps words to indices, and the size of the vocabulary.

Getting context word vectors

To create the vectors that represent context words, we will calculate the average of the one-hot vectors representing the individual words.

Let's start with a list of context words.

Using Python's list comprehension construct and the word_to_one_hot_vector function that we created in the previous section, we have created a list of one-hot vectors representing each of the context words.

Now create the context_words_to_vector function that takes in a list of context words, a word-to-index dictionary, and a vocabulary size, and outputs the vector representation of the context words.

Building the training set

We have now combined the functions that we created in the previous sections, to build a training set for the CBOW model, starting from the following tokenized corpus.

The continuous bag-of-words model

The CBOW model is based on a neural network, the architecture of which looks like the figure below, as we'll recall from the lecture.

Activation functions

Let's start by implementing the activation functions, ReLU and softmax.

ReLU

ReLU is used to calculate the values of the hidden layer, in the following formulas:

$$\begin{aligned} \mathbf{z_1} &= \mathbf{W_1}\mathbf{x} + \mathbf{b_1} \\ \mathbf{h} &= \mathrm{ReLU}(\mathbf{z_1}) \end{aligned}$$

Notice that using numpy's `random.rand` function returns a numpy array filled with values taken from a uniform distribution over [0, 1). Numpy allows vectorization so each value is multiplied by 10 and then subtracted 5.

Softmax

The second activation function that we need is softmax. This function is used to calculate the values of the output layer of the neural network, using the following formulas:

$$\begin{aligned} \mathbf{z_2} &= \mathbf{W_2}\mathbf{h} + \mathbf{b_2} \\ \mathbf{\hat{y}} &= \mathrm{softmax}(\mathbf{z_2}) \end{aligned}$$

To calculate softmax of a vector z, the i-th component of the resulting vector is given by:

$$\textrm{softmax}(\textbf{z})_i = \frac{e^{z_i}}{\sum\limits_{j=1}^{V} e^{z_j}}$$

This is for one element. We have use numpy's vectorized operations to calculate the values of all the elements of the softmax(z) vector in one go.

Dimensions: 1-D arrays vs 2-D column vectors

Before moving on to implement forward propagation, backpropagation, and gradient descent in the next lecture notebook, let's have a look at the dimensions of the vectors we've been handling until now.

Create a vector of length V filled with zeros.

To perform matrix multiplication in the next steps, we actually need wer column vectors to be represented as a matrix with one column. In numpy, this matrix is represented as a 2-dimensional array.

The easiest way to convert a 1D vector to a 2D column matrix is to set its .shape property to the number of rows and one column, as shown in the next cell.

Training the CBOW model

Forward propagation

Let's dive into the neural network itself, which is shown below with all the dimensions and formulas we'll need.

Set N equal to 3. Remember that N is a hyperparameter of the CBOW model that represents the size of the word embedding vectors, as well as the size of the hidden layer.

Also set V equal to 5, which is the size of the vocabulary we have used so far.

Initialization of the weights and biases

Before we start training the neural network, we need to initialize the weight matrices and bias vectors with random values.

we will implement a function to do this were self using numpy.random.rand.

get_training_examples, which uses the yield keyword, is known as a generator. When run, it builds an iterator, which is a special type of object that... we have iterate on (using a for loop for instance), to retrieve the successive values that the function generates.

In this case get_training_examples yields training examples, and iterating on training_examples will return the successive training examples.

next is another special keyword, which gets the next available value from an iterator. Here, we'll get the very first value, which is the first training example. If we run this cell again, we'll get the next value, and so on until the iterator runs out of values to return.

In this notebook next is used because we will only be performing one iteration of training. In this week's assignment with the full training over several iterations we'll use regular for loops with the iterator that supplies the training examples.

The vector representing the context words, which will be fed into the neural network, is:

Now convert these vectors into matrices (or 2D arrays) to be able to perform matrix multiplication on the right types of objects.

Values of the hidden layer

Now that we have initialized all the variables that we need for forward propagation, we have calculate the values of the hidden layer using the following formulas:

(1)z1=W1x+b1(2)h=ReLU(z1)
First, we have calculate the value of z1.

```
# Compute z1 (values of first hidden layer before applying the ReLU function)
z1 = np.dot(W1, x) + b1
```

np.dot is numpy's function for matrix multiplication.

As expected we get an N by 1 matrix, or column vector with N elements, where N is equal to the embedding size, which is 3 in this example.

Values of the output layer

Here are the formulas we need to calculate the values of the output layer, represented by the vector

$$\begin{aligned}\mathbf{z_1} &= \mathbf{W_1}\mathbf{x} + \mathbf{b_1} \\ \mathbf{h} &= \mathrm{ReLU}(\mathbf{z_1})\end{aligned}$$

As we've performed the calculations with random matrices and vectors (apart from the input vector), the output of the neural network is essentially random at this point. The learning process will adjust the weights and biases to match the actual targets better.

The neural network predicted the word "happy": the largest element of y^ is the third one, and the third word of the vocabulary is "happy".

Here's how we could implement this in Python:

```
print(Ind2word[np.argmax(y_hat)])
```

Cross-entropy loss

Now that we have the network's prediction, we have calculate the cross-entropy loss to determine how accurate the prediction was compared to the actual target.

Remember that we are working on a single training example, not on a batch of examples, which is why we are using *loss* and not *cost*, which is the generalized form of loss.

The formula for cross-entropy loss is:

$$J = -\sum_{k=1}^{V} y_k \log \hat{y}_k$$

loss = np.sum(-np.log(y_hat)*y)

Now use this function to calculate the loss with the actual values of y and y^.

Backpropagation

The formulas that we will implement for backpropagation are the following:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{W_1}} &= \mathrm{ReLU}\left(\mathbf{W_2^\top} (\mathbf{\hat{y}} - \mathbf{y})\right)\mathbf{x}^\top \\
\frac{\partial J}{\partial \mathbf{W_2}} &= (\mathbf{\hat{y}} - \mathbf{y})\mathbf{h^\top} \\
\frac{\partial J}{\partial \mathbf{b_1}} &= \mathrm{ReLU}\left(\mathbf{W_2^\top} (\mathbf{\hat{y}} - \mathbf{y})\right) \\
\frac{\partial J}{\partial \mathbf{b_2}} &= \mathbf{\hat{y}} - \mathbf{y}
\end{aligned}$$

Let's start with an easy one.

Calculate the partial derivative of the loss function with respect to b2, and store the result in grad_b2.

$$\frac{\partial J}{\partial \mathbf{b_2}} = \mathbf{\hat{y}} - \mathbf{y}$$

Next, calculate the partial derivative of the loss function with respect to W2, and store the result in grad_W2.

$$\frac{\partial J}{\partial \mathbf{W_2}} = (\mathbf{\hat{y}} - \mathbf{y})\mathbf{h^\top}$$

Now calculate the partial derivative with respect to b1 and store the result in grad_b1.

$$\frac{\partial J}{\partial \mathbf{b_1}} = \mathrm{ReLU}\left(\mathbf{W_2^\top}(\mathbf{\hat{y}} - \mathbf{y})\right)$$

Finally, calculate the partial derivative of the loss with respect to W1, and store it in grad_W1.

$$\frac{\partial J}{\partial \mathbf{W_1}} = \mathrm{ReLU}\left(\mathbf{W_2^\top}(\mathbf{\hat{y}} - \mathbf{y})\right)\mathbf{x}^\top$$

Before moving on to gradient descent, double-check that all the matrices have the expected dimensions.

Gradient descent

During the gradient descent phase, we will update the weights and biases by subtracting α times the gradient from the original matrices and vectors, using the following formulas.

$$\begin{aligned}
\mathbf{W_1} &:= \mathbf{W_1} - \alpha \frac{\partial J}{\partial \mathbf{W_1}} \\
\mathbf{W_2} &:= \mathbf{W_2} - \alpha \frac{\partial J}{\partial \mathbf{W_2}} \\
\mathbf{b_1} &:= \mathbf{b_1} - \alpha \frac{\partial J}{\partial \mathbf{b_1}} \\
\mathbf{b_2} &:= \mathbf{b_2} - \alpha \frac{\partial J}{\partial \mathbf{b_2}}
\end{aligned}$$

First, let set a value for α.

The difference is very subtle (hint: take a closer look at the last row), which is why it takes a fair amount of iterations to train the neural network until it reaches optimal weights and biases starting from random values.

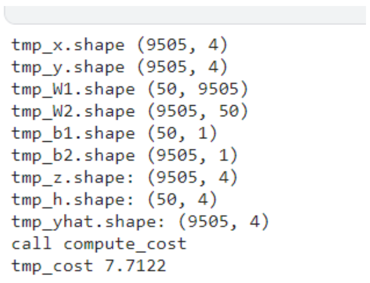
Now calculate the new values of W2 (to be stored in W2_new), b1 (in b1_new), and b2 (in b2_new).

$$\begin{aligned}
\mathbf{W_2} &:= \mathbf{W_2} - \alpha \frac{\partial J}{\partial \mathbf{W_2}} \\
\mathbf{b_1} &:= \mathbf{b_1} - \alpha \frac{\partial J}{\partial \mathbf{b_1}} \\
\mathbf{b_2} &:= \mathbf{b_2} - \alpha \frac{\partial J}{\partial \mathbf{b_2}}
\end{aligned}$$

Results

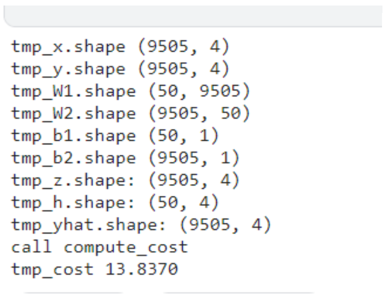
After performing the task on C=1,2,3

For C=1



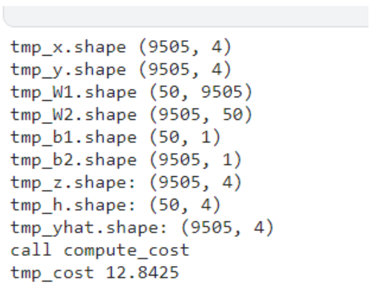
```
tmp_x.shape (9505, 4)
tmp_y.shape (9505, 4)
tmp_W1.shape (50, 9505)
tmp_W2.shape (9505, 50)
tmp_b1.shape (50, 1)
tmp_b2.shape (9505, 1)
tmp_z.shape: (9505, 4)
tmp_h.shape: (50, 4)
tmp_yhat.shape: (9505, 4)
call compute_cost
tmp_cost 7.7122
```

For C=2



```
tmp_x.shape (9505, 4)
tmp_y.shape (9505, 4)
tmp_W1.shape (50, 9505)
tmp_W2.shape (9505, 50)
tmp_b1.shape (50, 1)
tmp_b2.shape (9505, 1)
tmp_z.shape: (9505, 4)
tmp_h.shape: (50, 4)
tmp_yhat.shape: (9505, 4)
call compute_cost
tmp_cost 13.8370
```

For C=3



```
tmp_x.shape (9505, 4)
tmp_y.shape (9505, 4)
tmp_W1.shape (50, 9505)
tmp_W2.shape (9505, 50)
tmp_b1.shape (50, 1)
tmp_b2.shape (9505, 1)
tmp_z.shape: (9505, 4)
tmp_h.shape: (50, 4)
tmp_yhat.shape: (9505, 4)
call compute_cost
tmp_cost 12.8425
```

Cost comparison:

C1<C3<C2