Supplementary Slides

Md Tamjidul Hoque

… Word Embeddings

“You shall know a word by the company it keeps.”

- *J. R. Firth*

See: https://projector.tensorflow.org/

Word Embedding
Designing a loss function for learning word embeddings

- The vocabulary for even a simple real-world task can easily exceed 10,000 words.
- Therefore, we cannot develop word vectors by hand for large text corpora
- and need to devise a way to automatically find good word embeddings using some machine learning algorithms (for example, neural networks) to perform this laborious task efficiently.
- Also, to use any sort of machine learning algorithm for any sort of task, we need to define a loss, so completing the task becomes minimizing the loss.
- Let us define the loss for finding good word embedding vectors.

… Designing a loss function for learning word embeddings

- First, let us recall the equation we discussed at the beginning of this section:

$$P(w_{i-m}, \dots, w_{i-1}, w_{i+1}, \dots, w_{i+m}) = \prod_{j \neq i, j=i-m}^{i+m} P(w_j | w_i)$$

- With this equation in mind, we can define a loss/cost function for the machine learning model (e.g., neural network):

$$J(\theta) = -\frac{1}{N} \prod_{i=1}^{N} \prod_{j \neq i, j=i-m, j \neq 0}^{i+m} P(w_j | w_i)$$

- Remember, $J(\theta)$ is a loss (that is, cost), not a reward.

- Also, we want to maximize $P(w_j | w_i)$.

- Thus, we need a minus sign in front of the expression to convert it into a cost function.

… Designing a loss function for learning word embeddings

- Now, instead of working with the product operator, let us convert this to log space.
- Converting the equation to log space will introduce consistency and numerical stability.
- This gives us the following equation:

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq i, j=i-m, j\neq 0}^{i+m} \log P(w_j|w_i)$$

- This formulation of the cost function is known as the ***negative log-likelihood***.

… Designing a **loss function** for learning word embeddings

- Now, how do we calculate $P(w_j|w_i)$?
  - We use two vectors per word *w* as:
    - v_w where *w* is the center word and
    - u_w where *w* is the context word.
  - Then for a center word w_c and a context word w_o:

$$P(w_o|w_c) = \frac{exp(u_{w_o}^T\, v_{w_c})}{\sum_{w=1}^{N} exp(u_w^T\, v_{w_c})}$$

    - Here the dot product $(u_{w_o}^T\, v_{w_c})$ compares similarity of w_c and w_o. Since, $u_{w_o}^T\, v_{w_c} =$ u. v $= \sum_{i=1}^{D} u_i\, v_i$., where D is the size of the coding dimension.
    - $\sum_{w=1}^{N} exp(u_w^T\, v_{w_c})$ part normalize over the entire vocabulary to provide probability distribution.

Now, it is time to learn about the existing algorithms that use this cost function to find good word embeddings.

The skip-gram algorithm

- The skip-gram algorithm (developed in 2013), is an algorithm that exploits the context of the words of written text to learn good word embeddings.

- First, we need to design a mechanism to extract a dataset that can be fed to our learning model:

  - Such a dataset should be a set of tuples of the format (input, output).

  - The data preparation process should do the following:

    - Capture the surrounding words of a given word

    - Perform in an unsupervised manner

- The skip-gram model uses the following approach to design such a dataset:

… The skip-gram algorithm: dataset

1. For a given word w_i, a context window size ***m*** is assumed.

   - By context window size, we mean the number of words considered as context on a single side.

   - Therefore, for w_i, the context window (including the target word w_i) will be of size ***2m+1*** and will look like this:

     - $[w_{i-m}, \ldots, w_{i-1}, w_i, w_{i+1}, \ldots, w_{i+m}]$.

2. Next, input-output tuples are formed as:

   - $[\ldots, (w_i, w_{i-m}), \ldots,(w_i, w_{i-1}), (w_i, w_{i+1}), \ldots, (w_i, w_{i+m}), \ldots];$

   - here, (***m+1***) ≤ ***i*** ≤ (***N−m***) and ***N*** is the number of words in the text to get a practical insight.

   - Let us assume the following sentence and context window size (***m***) of 1: *The dog barked at the mailman*

… The skip-gram algorithm: dataset

- Let us assume the following sentence and context window size (***m***) of 1: *The dog barked at the mailman*

- For this example, the dataset would be as follows:

  - [(*dog*, *The*), (*dog*, *barked*), (*barked*, *dog*), (*barked*, *at*), ..., (*the*, *at*), (*the*, *mailman*)]

… The skip-gram algorithm: Training Model

- Once the data is in the (input, output) format, we can use a neural network to learn the word embeddings.

- First, let us identify the variables we need to learn the word embeddings.

- To store the word embeddings, we need a N × D matrix, where N is the vocabulary size and D is the dimensionality of the word embeddings (that is, the number of elements in the vector that represents a single word).

- D is a user-defined hyperparameter.

- The higher the D is, the more expressive the word embeddings learned will be.

- This matrix will be referred to as the *embedding space* or the *embedding layer*.

… The skip-gram algorithm: Training Model

- Next, we have a softmax layer with weights of size D × N, a bias of size N.

- Each word will be represented as a *one-hot encoded vector of size N* with one element being 1 and all the others being 0.
  - Therefore, an input word and the corresponding output words would each be of size N.
  - Let us refer to the i^{th} input as x_i, the corresponding embedding of x_i as z_i, and the corresponding output as y_i.

- At this point, we have the necessary variables defined.
  - Next, for each input x_i, we will look up the embedding vectors from the embedding layer corresponding to the input.

… The skip-gram algorithm: Training Model

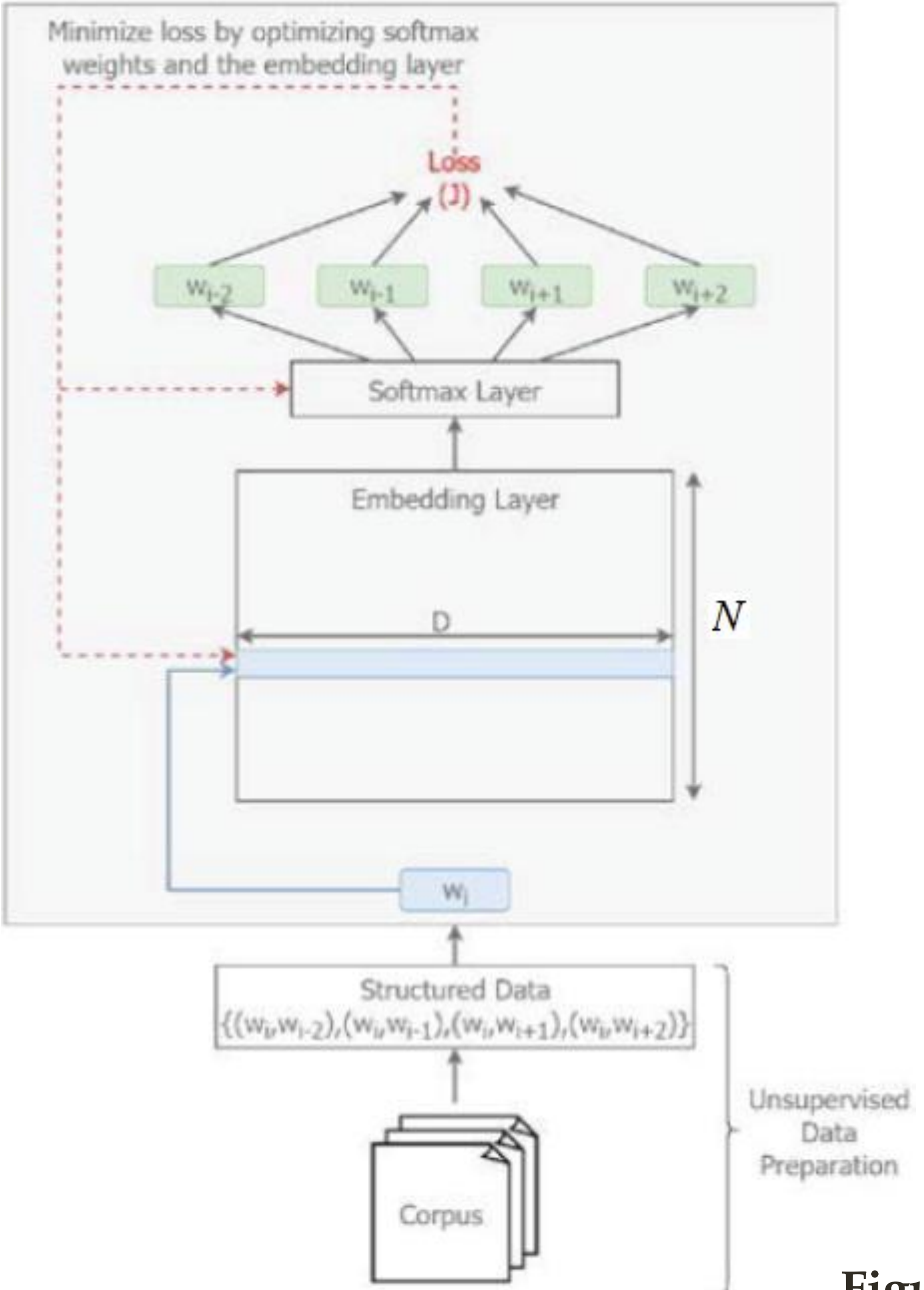

Figure 7: The conceptual skip-gram model.

… The skip-gram algorithm: Training Model

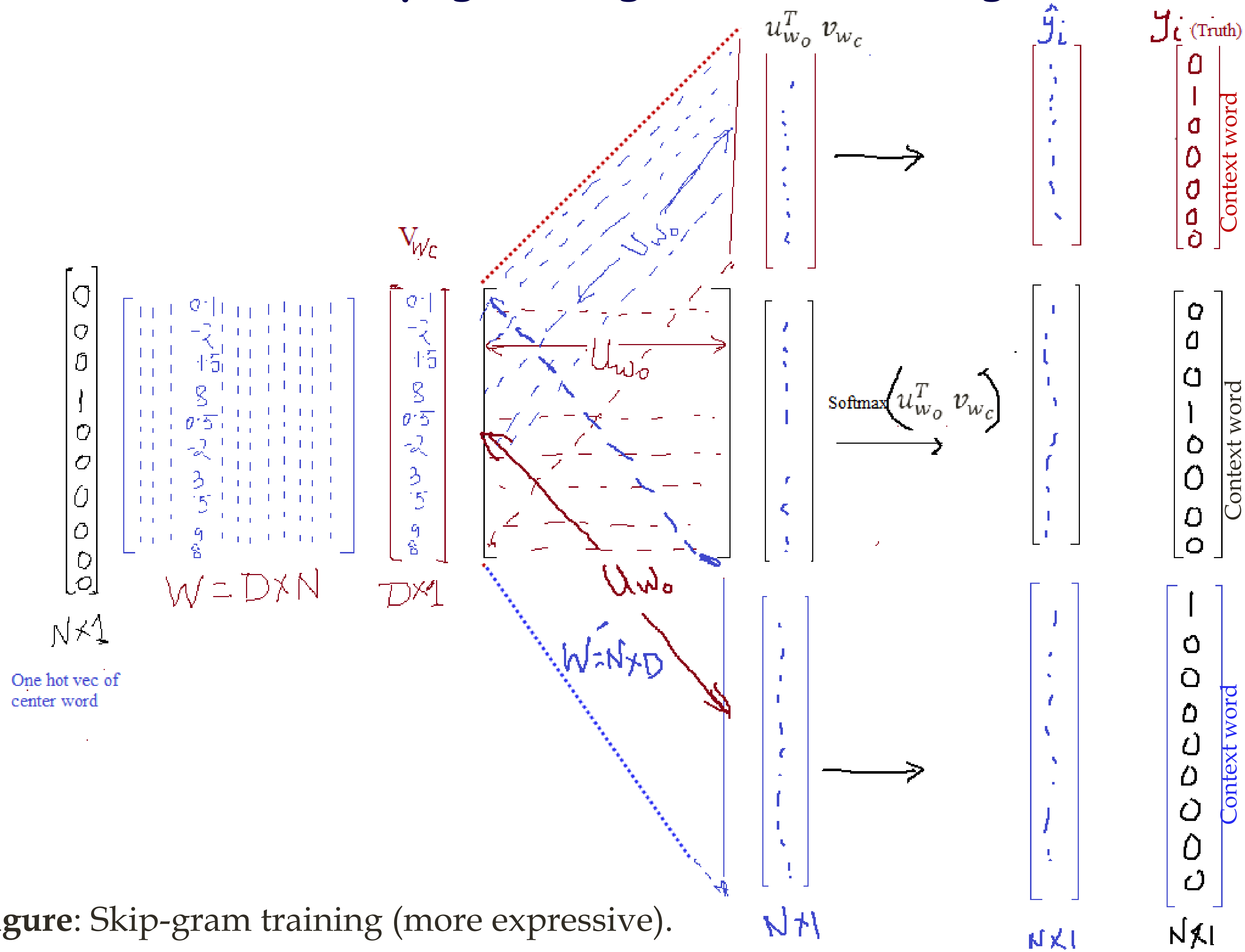

Figure: Skip-gram training (more expressive).

… The skip-gram algorithm: Training Model

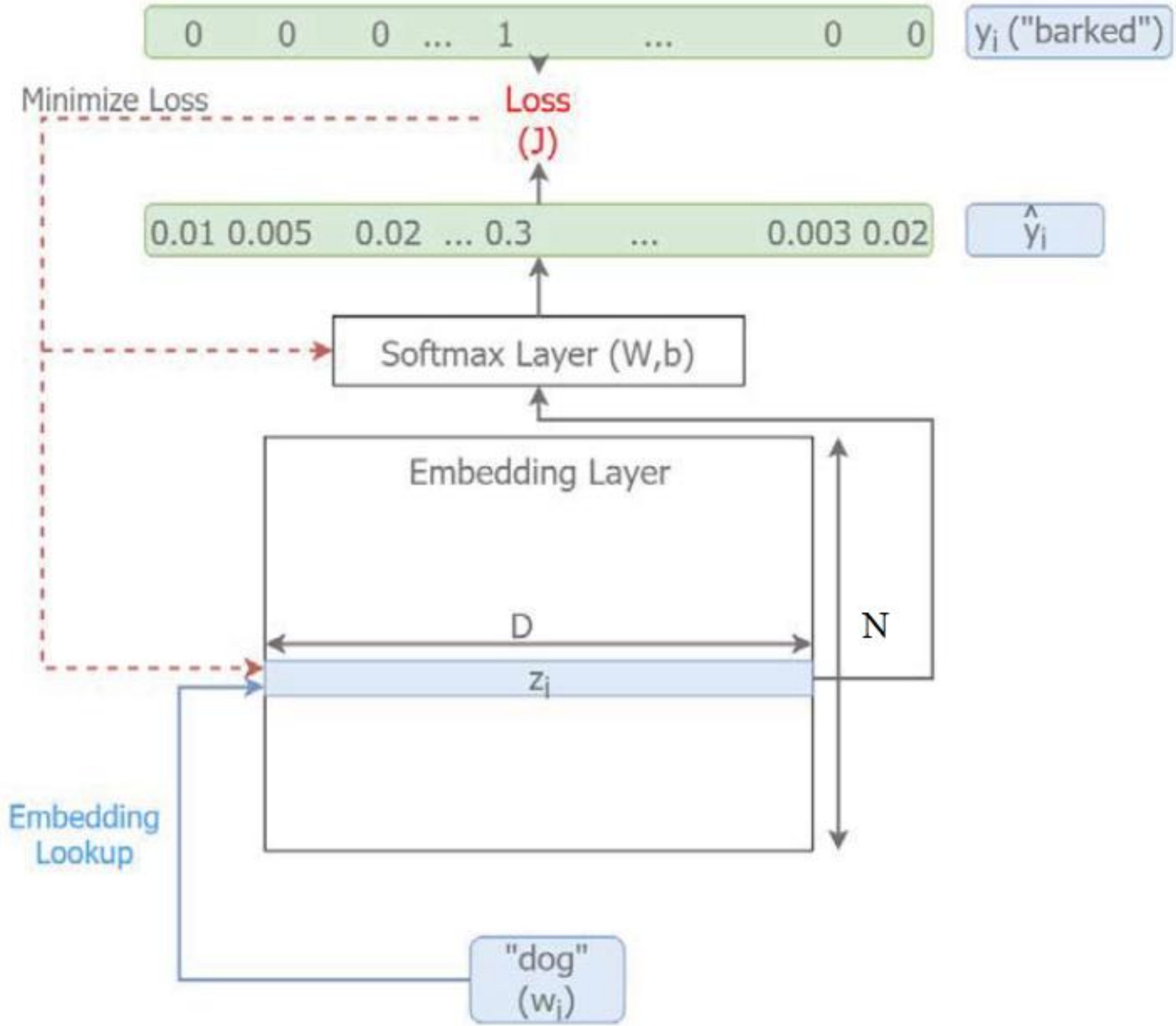

Figure 8: The implementation of the skip-gram model.

… The skip-gram algorithm: Training Model

- See exercise: (file) #7.1_word2vec_skip_gram_tensorflow.ipynb

The Continuous Bag-Of-Words (CBOW) Algorithm

- The CBOW model has a working similar to the skip-gram algorithm with one significant change in the problem formulation.
- In the skip-gram model, we predicted the context words from the target word.
- However, in the CBOW model, we will predict the target from contextual words.
- Let us compare what data looks like for skipgram and CBOW by taking the previous example sentence:

… The Continuous Bag-Of-Words (CBOW) Algorithm

The dog barked at the mailman.

- For **skip-gram**, data tuples - (input word, output word) - might look like this:
    - (*dog*, *the*), (*dog*, *barked*), (*barked*, *dog*) and so on.
- For **CBOW**, data tuples would look like the following:
    - ([*the*, *barked*], *dog*), ([*dog*, *at*], *barked*), and so on.
- Consequently, the input of the CBOW has a dimensionality of 2 × ***m*** × D, where ***m*** is the context window size and D is the dimensionality of the embeddings.

… The Continuous Bag-Of-Words (CBOW) Algorithm

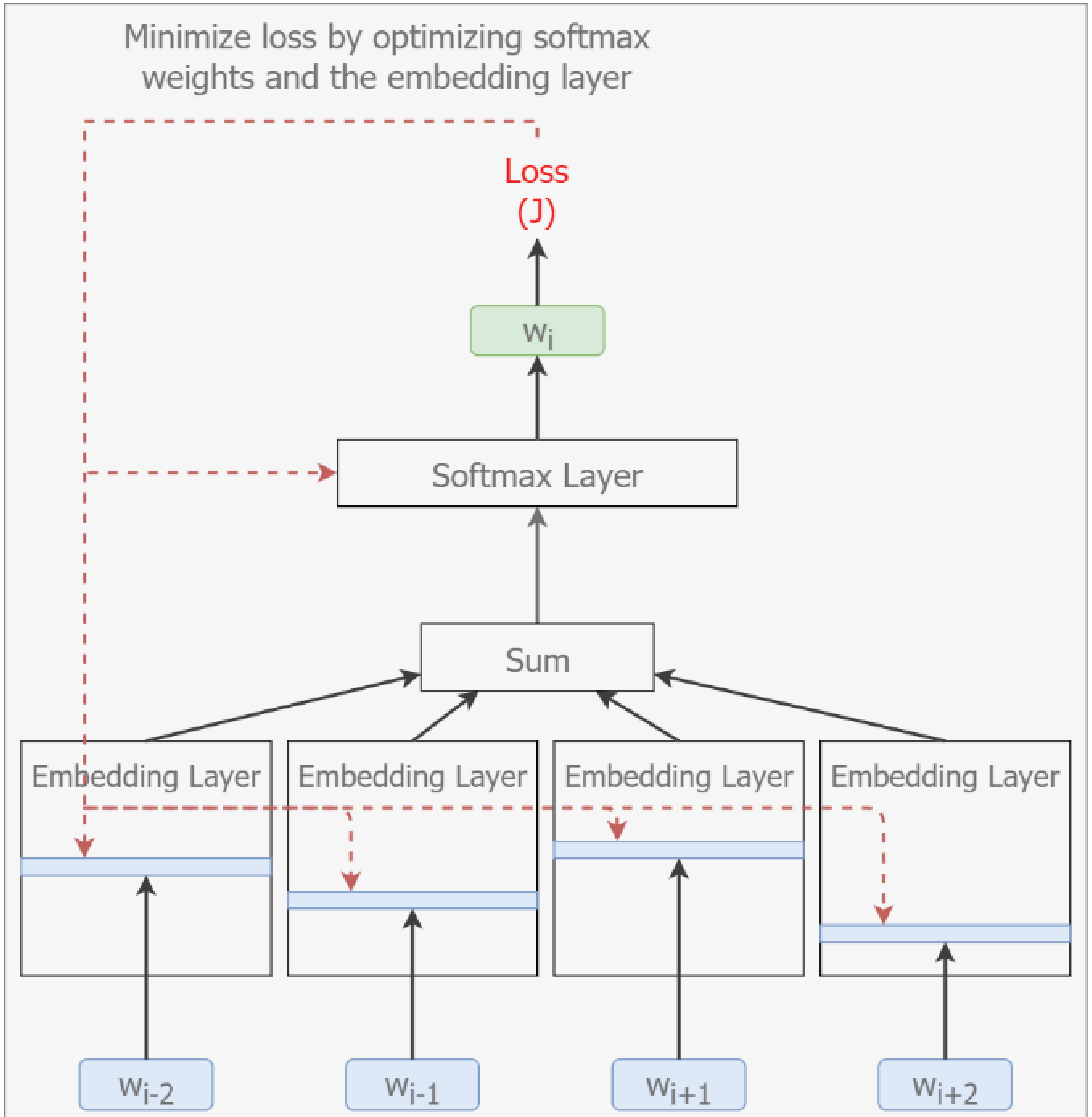

Figure 13: The CBOW model.

… The Continuous Bag-Of-Words (CBOW) Algorithm

- See exercise: (file) #7.2_word2vec_CBOW_tensorflow.ipynb