Word Embeddings: Dense Word Representations

Natural Language Processing

Daniel Ortiz Martínez, David Buchaca Prats

One-Hot Representation

- We can learn a vocabulary mapping that maps strings to integers and we can use it to construct "one hot encoded" words

- We denote by o the function creating the one hot encoding of a word:

$$o(\text{cat}) = (0, ..., 0, \overset{\text{cat}}{1}, \overset{\text{dog}}{0}, ..., 0)$$

$$o(\text{dog}) = (0, ..., 0, 0, \overset{\text{cat}}{0}, \overset{\text{dog}}{1}, ..., 0)$$

- In general let us denote by o_j^V the one hot vector induced by a vocabulary V that activates position j:

$$o_j^V = (0, ..., 0, \overset{j}{1}, 0, ..., 0) \in \mathbb{R}^V$$

One-Hot Representation Problems

- The distance between 2 different words (one hot encoded vectors) is always 1
- Does it make sense that $d(\text{cat}, \text{dog}) = d(\text{cat}, \text{table})$? Not really
- We want to learn a function e (embedding) that maps words to continuous real value numbers such that:

$$\|e(\text{cat}) - e(\text{dog})\|^2 < \|e(\text{cat}) - e(\text{table})\|^2$$

$$o(\text{cat}) = (0, ..., 0, \overset{\text{cat}}{1}, \overset{\text{dog}}{0}, \overset{\text{table}}{0}, \overset{\text{chair}}{0}, ..., 0) \qquad e(\text{cat}) = (0.76, 0.54, ..., 0.10, , ..., 0.10, 10)$$

$$o(\text{dog}) = (0, ..., 0, \overset{\text{cat}}{0}, \overset{\text{dog}}{1}, \overset{\text{table}}{0}, \overset{\text{chair}}{0}, ..., 0) \qquad e(\text{dog}) = (0.70, 0.40, ..., 0.10, , ..., 0.10, 0.10)$$

$$o(\text{table}) = (0, ..., 0, \overset{\text{cat}}{0}, \overset{\text{dog}}{0}, \overset{\text{table}}{1}, \overset{\text{chair}}{0}, ..., 0) \qquad e(\text{table}) = (0.10, 0.10, ..., 0.10, , ..., 0.50, 0.60)$$

$$o(\text{chair}) = (0, ..., 0, \overset{\text{cat}}{0}, \overset{\text{dog}}{0}, \overset{\text{table}}{0}, \overset{\text{chair}}{1}, ..., 0) \qquad e(\text{table}) = (0.10, 0.10, ..., 0.10, , ..., 0.60, 0.80)$$

Training Word Embeddings: Predicting Nearby Words

- To get a dense representation for words, techniques such as Word2vec (Mikolov et al. 2013) or Glove (Pennington et al. 2014) learn a mapping that *embeds* words to dense vectors of a pre-fixed dimension, which we call *embedding dimension*
- Given a corpus, word embedding techniques set a learning task based on predicting nearby words of a *center* or *pivot* word
  - The data is usually processed to generate pairs of input/output words
  - Original dataset with sentences of different lengths is converted to a tabular dataset
  - Learning is performed in the tabular dataset, where the input and output dimensions of the neural model equal the vocabulary size
  - There are actually no labels in the dataset (learning is *self-supervised*)

Basic Word Embeddings Methods

- We will focus on Word2Vec
- This paper presents two learning tasks to learn word embeddings:
    - Continuous bag-of-words (**CBOW**): input is a bunch of words in a sentence, output is one word of the sentence that is *masked*
    - Continuous skip-gram (**Skip-gram**): input is a word in a sentence, the output is a bunch of words that are next to the input word (also known as the context words)
- Other relevant techniques are
    - Global Vectors or GloVe
    - FastText (Joulin et al. 2016)

Word2Vec Overview

- Word2vec should not be seen as a single algorithm but more as software package to learn word embeddings
- The package has two distinct models:
  - CBOW
  - Skip-Gram
- Word2Vec implements ideas for fast learning with big vocabularies:
  - **Negative Sampling**: Allows fast normalization of the softmax
  - **Hierarchical Softmax**: Allows faster evaluation with O(log n) time instead of O(n)

Word2Vec Overview

- The package also implements relevant preprocessing of the text:
    - **Dynamic context windows**: words that are near to the target (or center) word are more important than other words that are far away from the target (or center) word
    - **Subsampling**: Used to counter the imbalance between the rare and frequent words

Definitions for CBOW

Center word

I want a new pair of shoes

Context words

Context size $= C = 2$

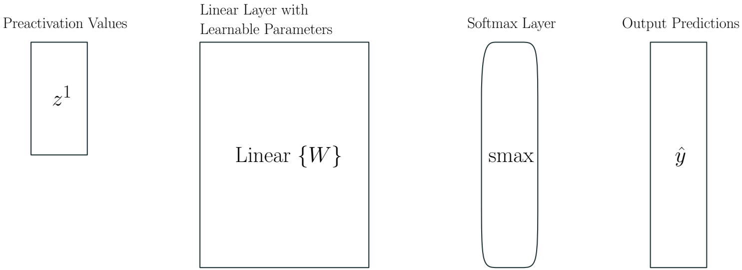
Window size $= 2 \cdot C + 1 = 5$

Transforming a Corpus to a Tabular Dataset

- In order to learn the embeddings, a sliding window is passed over each sentence, generating a set of training examples for the tabular dataset
- Example: Consider the sentence "I want a new pair of shoes", C=2

Sliding Window	Window	Input	Output
I want a new pair of shoes	[I, want, a, new, pair]	[I, want, new, pair]	a
I want a new pair of shoes	[want, a, new, pair, of]	[want, a, pair, of]	new
I want a new pair of shoes	[a, new, pair, of, shoes]	[a, new, of, shoes]	pair

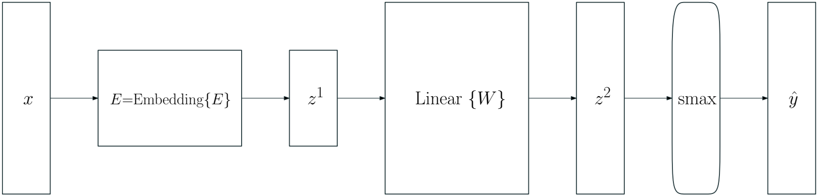
CBOW: Notation



$$\text{smax} = \left(\frac{e^{z_1}}{\Sigma_{j=1}^{V} e^{z_j}}, \ldots, \frac{e^{z_V}}{\Sigma_{j=1}^{V} e^{z_j}} \right)$$

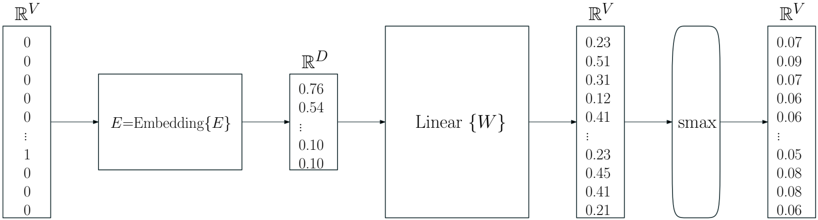
- Consider the following input to the CBOW model:

$$x = o(w_{t-2}) + o(w_{t-1}) + o(w_{t+1}) + o(w_{t+2}) = [0, 0, ..., \overset{w_{t-2}}{1}, ..., \overset{w_{t-1}}{1}, ..., \overset{w_{t+1}}{1}, ..., \overset{w_{t+2}}{1}, 0, 0]$$



CBOW Word Embeddings Placement in the Model

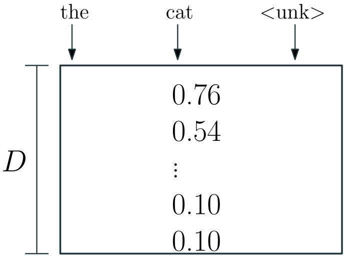
- The weight matrix E with one colum per each vocabulary word
- Column j of the matrix, $E[:, j] := e_j$, contains a dense vector of shape D, which can be used as a word embedding for word in position j
- D is the dimensionality of the word embedding and a hyperparameter of the algorithm



CBOW Word Embedding

- If we multiply a one hot vector times E, that is $E \cdot o_j$ we get $E[:, j]$
- Vocabulary $A^* \longrightarrow \mathbb{N}$ maps strings (from Kleene closure of an alphabet A) to V positions

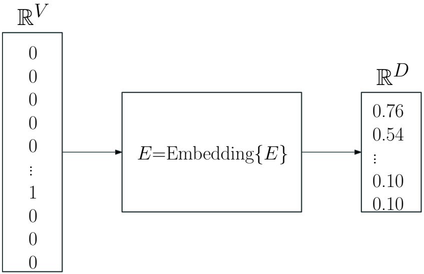
$$e(\mathrm{cat}) := E \cdot o^V_{\mathrm{vocab(cat)}} = E[:, \mathrm{vocab(cat)}]$$



CBOW Forward Pass: First Layer

- The first embedding Layer $E : \mathbb{R}^{\mathbb{V}} \longrightarrow \mathbb{R}$ maps vectors of size V to D dimensional vectors
  - **Input**: $x = o(w_{t-2}) + o(w_{t-1}) + o(w_{t+1}) + o(w_{t+2})$
  - **Output**: $E(x) = \frac{1}{2C}\left(E \cdot o(w_{t-2}) + E \cdot o(w_{t-1}) + E \cdot o(w_{t-1} + E \cdot o(w_{t+2}))\right)$
- Note that the previous expression can be written as[a]

$$E(x) = \frac{1}{2C}\left(E[:, \mathrm{vocab}(w_{t-2}) + E[:, \mathrm{vocab}(w_{t-1}) + E[:, \mathrm{vocab}(w_{t+1}) + E[:, \mathrm{vocab}(w_{t+2})]\right)$$



[a] Matrix operations are no longer used, the relevant columns of E are retrieved instead.

Embedding Layer versus Linear Layer

```python
import timeit
import torch

# Initialize variables
num_embeddings = 10000
embedding_dim = 200
E = torch.nn.Embedding(num_embeddings, embedding_dim)

# Prepare input for the embedding layer
vocab = {'a':0, 'house':1, 'i':2}
x = torch.tensor([vocab['house'], vocab['i']])

# Prepare input for the dense layer
x_onehot = torch.zeros(num_embeddings)
x_onehot[vocab['house']] = 1
x_onehot[vocab['i']] = 1

# Create linear layer equivalent to embedding layer
E_trans_weight = E.weight.transpose(0, 1)
E_linear = torch.nn.Linear(num_embeddings, embedding_dim, bias=False)
E_linear.weight = torch.nn.Parameter(E_trans_weight)

# Calculate time
num_loop = 100000
time_embed = timeit.timeit("E.forward(x).sum(axis=0)", number=num_loop, globals=globals())
print("Embedding layer time:", time_embed/num_loop)
time_linear = timeit.timeit("E_linear.forward(x_onehot)", number=num_loop, globals=globals())
print("Linear layer time:", time_linear/num_loop)
print("Ratio:", time_linear/time_embed)

# Check the outputs are the same
torch.testing.assert_close(E_linear.forward(x_onehot), E.forward(x).sum(axis=0))
```

Embedding Layer versus Linear Layer: Output

```
Embedding layer time: 7.889608349942136e-06
Linear layer time: 7.09692469699803e-05
Ratio: 8.995281365329218
```

- Embedding layer is almost one order of magnitude faster
- The embedding layer works as a lookup table while the linear layer needs to compute matrix operations

CBOW Forward Pass: First Layer Example

- We have defined $E : \mathbb{R}^V \longrightarrow \mathbb{R}^D$ to be

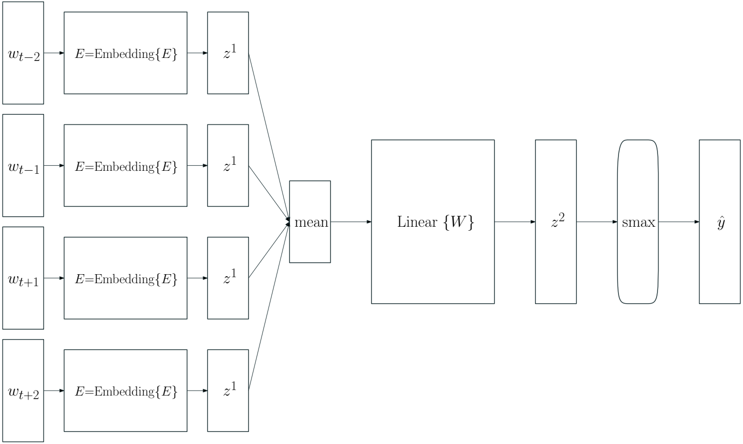
$$E(x) = \frac{1}{2C}\left(E \cdot o(w_{t-2}) + E \cdot o(w_{t-1}) + E \cdot o(w_{t-1} + E \cdot o(w_{t+2}))\right)$$

- Consider the following example:
  - Sentence and sliding window: " I love books because I love learning"
  - vocab = {am:0, because:1, happy:2, I:3, love:4, learning:5}
- The input for E in this example would be:

	I	love	because	I
am	0	0	0	0
because	0	0	1	0
happy	0	0	0	0
I	1	0	0	1
love	0	1	0	0
learning	0	0	0	0

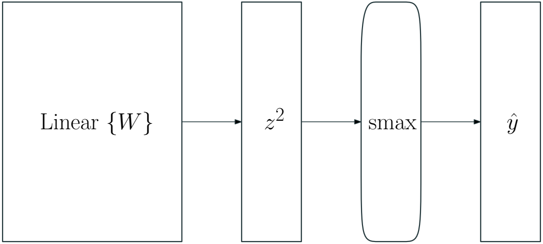
CBOW Forward Pass: Alternative View of First Layer

- This diagram emphasizes that the model is not exactly a standard MLP. In the forward pass examples are "forwarded" with the same weight matrix E and the results are aggregated with a mean



CBOW Forward Pass: Second Layer

- The second layer takes the mean vector over the activated columns of E considered in the training example and passes the signal over a linear layer $W : \mathbb{R}^D \longrightarrow \mathbb{R}^V$
- Then the output of the linear layer, z_2, passes over a softmax

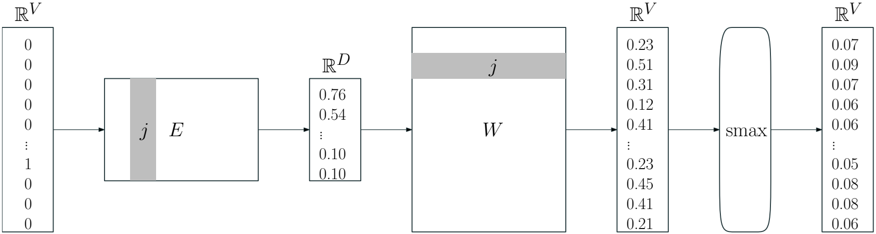


CBOW Overview

- Initialization step:
  1. Iterate over all the corpus to find the words in the vocabulary
  2. Build a vocab mapping that assigns a different integer to every word
- For a given sentence, select all possible windows. For each window:
  1. Compute the embedding layer activation of the input $z^1 = E(x)$
  2. Generate output scores $z^2 = W \cdot x$
  3. Turn scores into probabilities by calculating $\hat{y} = \mathrm{smax}(z)$
  4. Compute the gradient of the cross-entropy loss
  5. Update the weights using gradient descent

CBOW Word Embedding Extraction

- After learning we have two matrices E and W
- E contains word embeddings as columns
- W contains word embeddings as rows
- We can extract the word embeddings as a mean over the two matrices
- Alternatively, we can just keep the column in the embedding matrix



Improving Training Efficiency with Negative Sampling

- The second layer takes the mean vector over the activated columns of E considered in the training example and passes the signal over a Linear Layer
- Then the output of the linear layer z^2 passes over a Softmax.
- Softmax step is very costly due to the computation of $\sum_{j=1}^{V} e^{z_j}$
- To alleviate this problem the training objective function is modified

Improving Training Efficiency with Negative Sampling

- Basic assumption: for each training sample, only a small percentage of weights will need to be updated
- Updates will be focused on the words of the context window and some random words outside the window, which constitute *negative samples*
  - Maximize probability to predict center word from context words
  - Minimize probability to predict center word from negative samples
- Negative samples are generated by a noise distribution

Word2Vec in Gensim

```
import gensim.models.word2vec as w2v

num_features = 300
num_epochs = 10

# Minimum word count threshold
min_word_count = 0

# Number of threads to run in parallel
num_workers = multiprocessing.cpu_count()

# Context window length
context_size = 5

# Downsample setting for frequent words
# 0 - 1e-5 is good for this
downsampling = 1e-3
seed = 1

# Optional training algorithm: 1 for skip-gram, otherwise CBOW
sg = 0

word2vec = w2v.Word2Vec(
    sg=sg,
    seed=seed,
    workers=num_workers,
    vector_size=num_features,
    min_count=min_word_count,
    window=context_size,
    sample=downsampling)
```

Sentence Representations from Word Embeddings

- A naive way to generate a fixed size vector for a sentence is to get for each word in the sentence the embedding and average those vectors

```
def sentence_to_wordlist(raw):
    clean = re.sub("^a-zA-Z", " ", raw)
    clean = clean.lower()
    words = clean.split()
    return words

def doc_to_vec(sentence, word2vec):
    word_list = sentence_to_wordlist(sentence)
    word_vectors = []
    for w in word_list:
        word_vectors.append(word2vec.wv.get_vector(w))

    return np.mean(word_vectores, axis=0)
```

Combining Word Embeddings with Sparse Representations

- Sparse representations can be combined with dense representations to improve results
- The following table shows accuracy of a perceptron on the 20 newsgroup dataset with different input features:

20 newsgroup dataset	**Word2Vec Average**	**Count Vectorizer**	**Both**
Train	0.814	0.999	0.999
Test	0.726	0.752	0.768

From Word Vectors to Sentence Vectors

- There are many works that leverage word level embeddings to generate sentence level embeddings, usually by computing a weighted average of the embeddings of the words in a sentence (or doing this in chunks and concatenating the results)
    - (Arora et al. 2017)
    - (Ionescu and Butnaru 2019)
    - (Gupta et al. 2020)
    - (Wang et al. 2020)
    - (Muffo et al. 2021)

References i

- Arora, Sanjeev, Yingyu Liang, and Tengyu Ma (Apr. 2017). "A simple but tough-to-beat baseline for sentence embeddings". English (US). In: 5th International Conference on Learning Representations, ICLR 2017.
- Gupta, Vivek, Ankit Saw, Pegah Nokhiz, Praneeth Netrapalli, Piyush Rai, and Partha P. Talukdar (2020). "P-SIF: Document Embeddings Using Partition Averaging". In: *CoRR* abs/2005.09069. URL: `https://arxiv.org/abs/2005.09069`.

References ii

Ionescu, Radu Tudor and Andrei Butnaru (June 2019). "Vector of Locally-Aggregated Word Embeddings (VLAWE): A Novel Document-level Representation". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 363–369. DOI: 10.18653/v1/N19-1033. URL: https://aclanthology.org/N19-1033.

Joulin, Armand, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov (2016). *Bag of Tricks for Efficient Text Classification*. arXiv: 1607.01759 [cs.CL].

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean (2013). "Efficient Estimation of Word Representations in Vector Space". In: *CoRR* abs/1301.3781. URL: `http://dblp.uni-trier.de/db/journals/corr/corr1301.html#abs-1301-3781`.

Muffo, Matteo, Roberto Tedesco, Licia Sbattella, and Vincenzo Scotti (1111 2021). "Static Fuzzy Bag-of-Words: a Lightweight and Fast Sentence Embedding Algorithm". In: *Proceedings of the 4th International Conference on Natural Language and Speech Processing (ICNLSP 2021)*. Trento, Italy: Association for Computational Linguistics, pp. 73–82. URL: `https://aclanthology.org/2021.icnlsp-1.9`.

Pennington, Jeffrey, Richard Socher, and Christopher Manning (Oct. 2014). "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162.

Wang, Bin, Fenxiao Chen, Yuncheng Wang, and C.-C. Jay Kuo (2020). "Efficient Sentence Embedding via Semantic Subspace Analysis". In: *CoRR* abs/2002.09620. URL: https://arxiv.org/abs/2002.09620.