WORD EMBEDDINGS: DENSE WORD REPRESENTATIONS

Session 6

David Buchaca Prats
2023

ONE-HOT REPRESENTATION

- We have seen we can learn a Vocabulary mapping that maps strings to integers and we can use it to construct "one hot encoded" words.
- If we denote by oh the function that creates the one hot encoding of a word, vectors have the following form:
  - $\mathrm{oh}(cat) = (0,\ldots,0,\overset{cat}{1},0,\ldots,0)$
  - $\mathrm{oh}(dog) = (0,\ldots,0,\overset{cat}{0},\overset{dog}{1},0,\ldots,0)$
- In general let us denote by o_j^V the one hot vector induced by a vocabulary with V terms that activates position j. That is $o_j^V = (0,\ldots,0,\overset{j}{1},0,\ldots,0) \in \mathbb{R}^V$
  - For example: $o_{750}^V = (0,\ldots,0,\overset{750}{1},0,\ldots,0) \in \mathbb{R}^V$

ONE-HOT REPRESENTATION PROBLEMS

- The distance between 2 words (one hot encoded vectors) is always 1, as long as two words are different.
- Is it sensible that d(cat, dog) = d(cat, table)? Not really
- We want to learn a function "em" (embedding) that maps words to continuous real value numbers such that $\|\mathrm{em}(\mathit{cat}) - \mathrm{em}(\mathit{dog})\|^2 < \|\mathrm{em}(\mathit{cat}) - \mathrm{em}(\mathit{table})\|^2$
- $\mathrm{oh}(\mathit{cat}) = (\overset{a}{0}, \ldots, 0, \overset{cat}{1}, 0, \ldots, \overset{zoo}{0}, \overset{<unk>}{0})$ $\quad$ $\mathrm{em}(\mathit{cat}) = (0.76, 0.54, \ldots, 0.1, 0.1, \ldots, 0.1)$
- $\mathrm{oh}(\mathit{dog}) = (\overset{a}{0}, \ldots, 0, \overset{dog}{1}, 0, \ldots, \overset{zoo}{0}, \overset{<unk>}{0})$ $\quad$ $\mathrm{em}(\mathit{dog}) = (0.70, 0.4, \ldots, 0.1, 0.1, \ldots, 0.1)$
- $\mathrm{oh}(\mathit{table}) = (\overset{a}{0}, \ldots, 0, \overset{table}{1}, 0, \ldots, \overset{zoo}{0}, \overset{<unk>}{0})$ $\quad$ $\mathrm{em}(\mathit{table}) = (0.1, 0.1, \ldots, 0.1, 0.1, \ldots, 0.5, 0.7)$
- $\mathrm{oh}(\mathit{chair}) = (\overset{a}{0}, \ldots, 0, \overset{chair}{1}, 0, \ldots, \overset{zoo}{0}, \overset{<unk>}{0})$ $\quad$ $\mathrm{em}(\mathit{chair}) = (0.1, 0.1, \ldots, 0.1, 0.1, \ldots, 0.6, 0.8)$

TRAINING WORD EMBEDDINGS: PREDICTING NEARBY WORDS

- In order to get a dense representation for words, techniques such as Word2vec or Glove learn a mapping that "embeds" words to dense embeddings 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.
  - To do so, the corpus is usually processed to generate pairs of input/output words.
  - This process essentially converts a Corpus where each sentence might have different length to a tabular dataset. Then, learning is performed in the tabular dataset, where the input and output dimensions of the model (which is a neural network) equal the vocabulary size.
  - Note that learning here is "self-supervised". There are actually no labels in the dataset but we create them from any corpus.

BASIC WORD EMBEDDING METHODS

- We will focus on Word2Vec (Google, 2013)
  - This paper presents two learning tasks to learn word embeddings:
    - CBOW: Continuous bag-of-words
      - In this task the input is a bunch of words in a sentence, output is one word of the sentence that is 'masked'.
    - Skip-Gram: Continuous skip-gram
      - In this task the input is a word in a sentence, the output is a bunch of words that are next to the input word
- Other relevant techniques are
  - Global Vectors or GloVe (Stanford, 2014)
  - FastText (Facebook, 2016)

WORD2VEC OVERVIEW

- Word2vec should not be seen as a single algorithm but more as software package implementing different ideas.
- The objective of the package is to allow learning word embeddings.
- The package has two distinct models:
  - CBOW
  - Skip-Gram
- The package implements different ideas for fast learning with big vocabularies:
  - Negative Sampling: Allows fast normalization of the softmax summing over
  - Hierarchical Softmax: Allows faster evaluation with O(log n) time instead of O(n)
- The package also implements relevant preprocessing of the text, including
  - 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, CONTEXT SIZE, WINDOW SIZE

- Let **center word** be the "target" word that we consider in every example on the learning task that we will cast to learn embeddings from a Corpus.
  - "I want a new pair of shoes"
    center word is "pair" and context words are ["a", "new", "of", "shoes"]
- Let **context size** be an integer defining the amount of words considered to be in the context of the center word. This integer is the number of words in the left/right for the context word.
  - "I want a new pair of shoes"
    Has a context of 2 words, that is, C = 2.
- Let "window size" be C*2 +1 be the sliding window size used to generate training data from a Corpus.
  - "I want a new pair of shoes"
  - The window size has 5 elements ["a", "new", "pair", "of", "shoes"]

TRANSFORMING A CORPUS TO A TABULAR DATASET

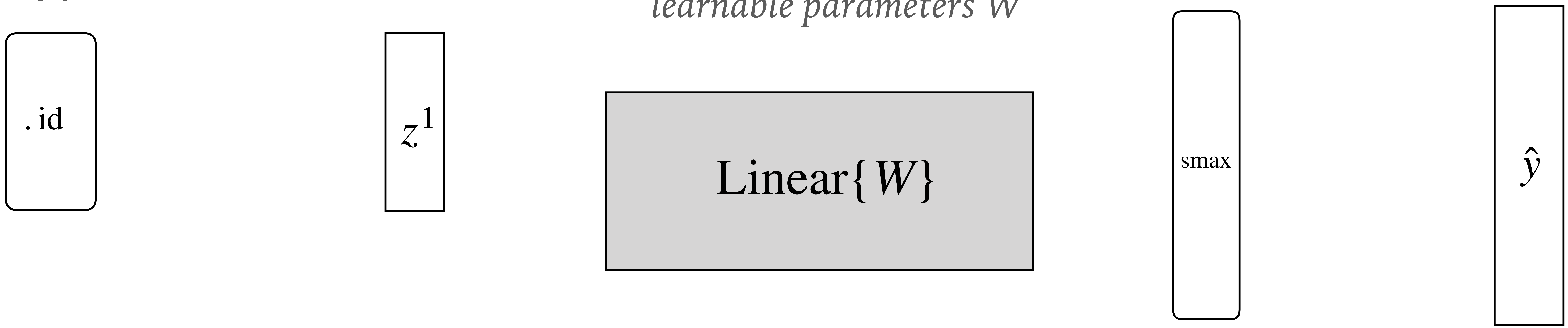
- In order to prepare data to learn the embeddings we will do the following:
  - For each sentence in the corpus, pass a sliding window over the sentence and generate a bunch of training examples for "the tabular dataset".
- Example: Consider a sentence "I want a new pair of shoes", C=2

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: NEURAL NETWORK ARCHITECTURE

- Consider the following notation:

Identity function | *Preactivation values* | *Linear layer with learnable parameters W* | *Softmax layer* | *Output predictions*



- Recall that the softmax is defined as follows:

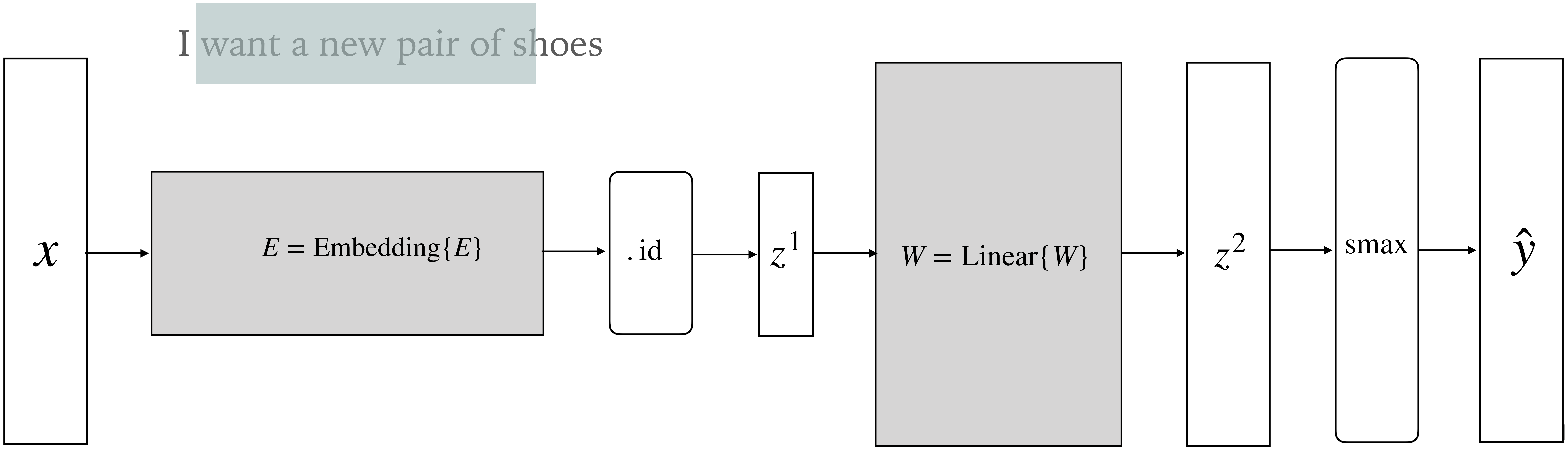
$$smax(z) = \left(\frac{e^{z_1}}{\sum_{j=1}^{V} e^{z_j}}, \; \dots \;, \frac{e^{z_V}}{\sum_{j=1}^{V} e^{z_j}} \right)$$

CBOW: NEURAL NETWORK ARCHITECTURE

- Consider the input of the model to be

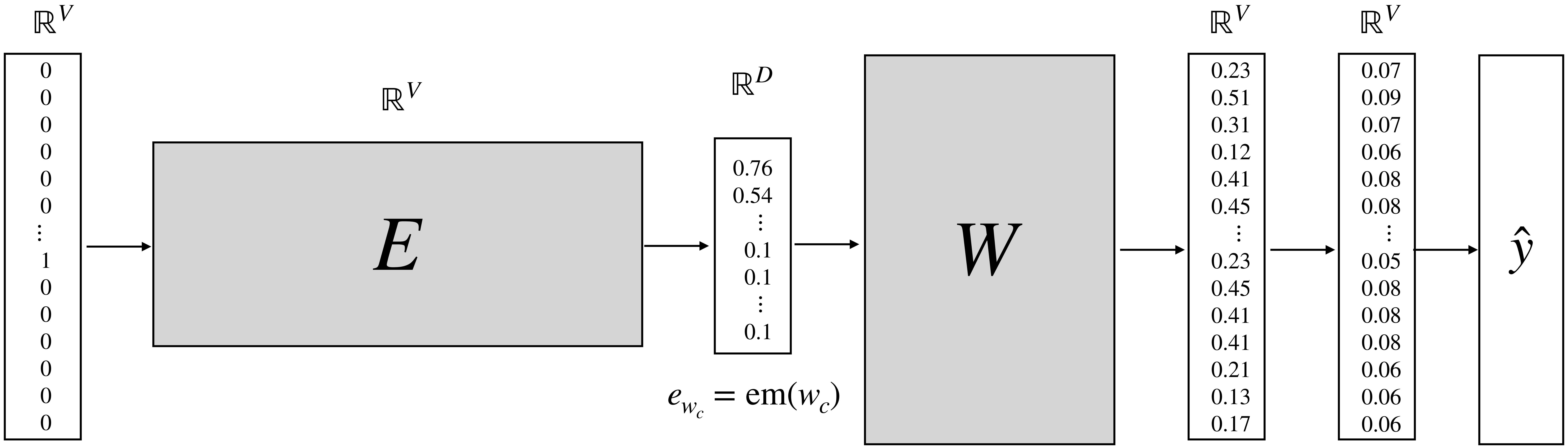
$$x = \mathrm{oh}(w_{t-2}) + \mathrm{oh}(w_{t-1}) + \mathrm{oh}(w_{t+1}) + \mathrm{oh}(w_{t+2}) = [0,0,\ldots,\overset{w_{t-2}}{1},\ldots,\overset{w_{t-1}}{1},\ldots,\overset{w_{t+1}}{1}\ldots,\overset{w_{t+2}}{1}\ldots,0,0]$$

  - That is, x is a vector of size "vocabulary" with C*2 ones at the corresponding indices of the vocabulary of the words present in the sliding window.



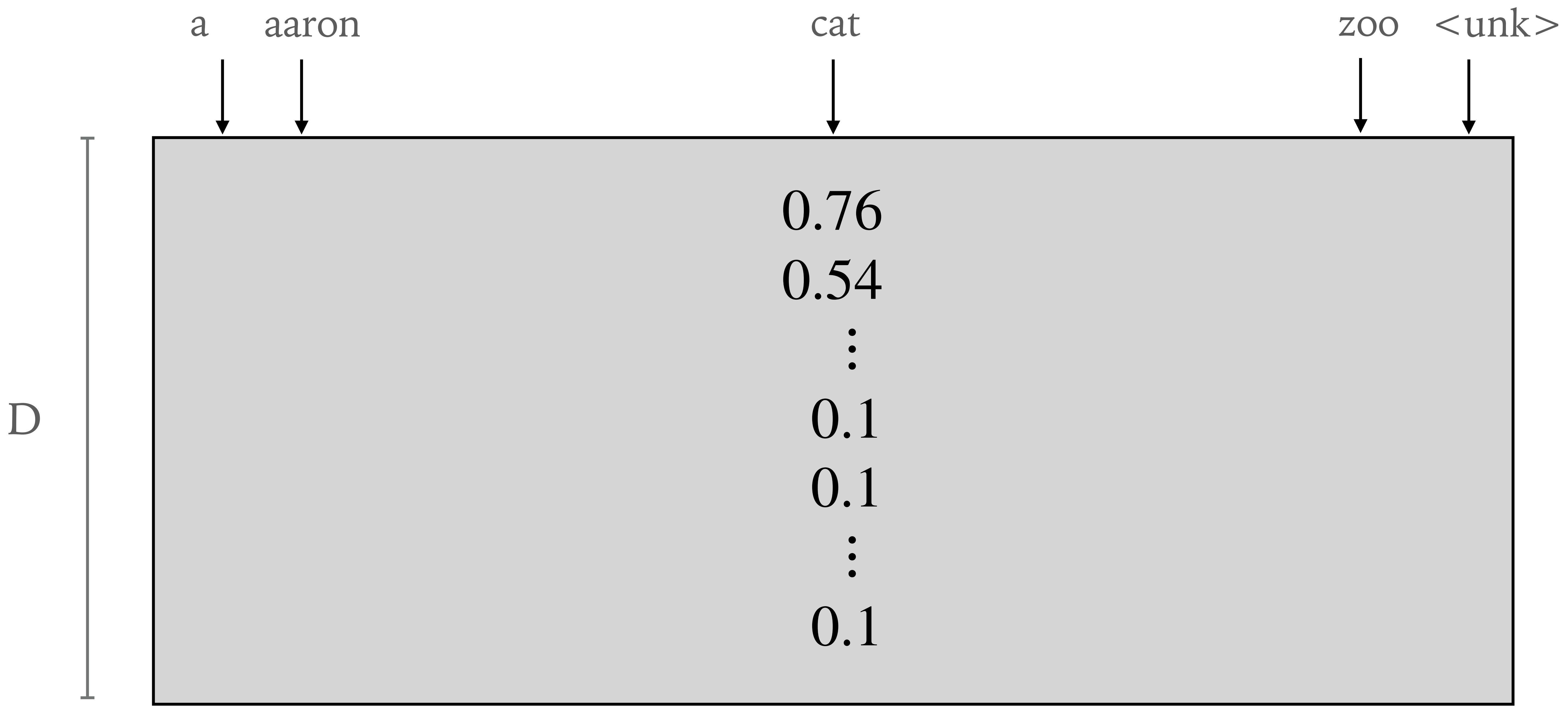
CBOW WORD EMBEDDINGS PLACEMENT IN THE MODEL

- This model has a weight matrix E has as many columns as words in the vocabulary.
- Column j of the matrix, $E[\,\cdot\,, j] := e_j$, contains a dense vector of shape D, this can be used as a word embedding for word assigned to position j.
- D is the dimensionality of the word embedding and a hyperparameter of the algorithm.



CBOW WORD EMBEDDING

- Note that if we multiply a one hot vector times E, that is $E \cdot o_j^V$ we get $E[: ,j]$
- Consider $vocab : A^* \longrightarrow \mathbb{N}$ maps strings (from Kleene closure of an alphabet A) to positions. Then $\mathrm{em}(cat) := E \cdot o^V_{vocab(cat)} = E[: , vocab(cat)]$



CBOW FORWARD PASS: FIRST LAYER

- The first Embedding Layer $E : \mathbb{R}^V \longrightarrow \mathbb{R}^D$ maps vectors of size "vocabulary" to D dimensional embeddings. This layer takes as input vectors of the form:

$$x = \mathrm{oh}(w_{t-2}) + \mathrm{oh}(w_{t-1}) + \mathrm{oh}(w_{t+1}) + \mathrm{oh}(w_{t+2}) = [0,0,\ldots, \overset{w_{t-2}}{1}, \ldots, \overset{w_{t-1}}{1}, \ldots, \overset{w_{t+1}}{1} \ldots, \overset{w_{t+2}}{1} \ldots,0,0]$$

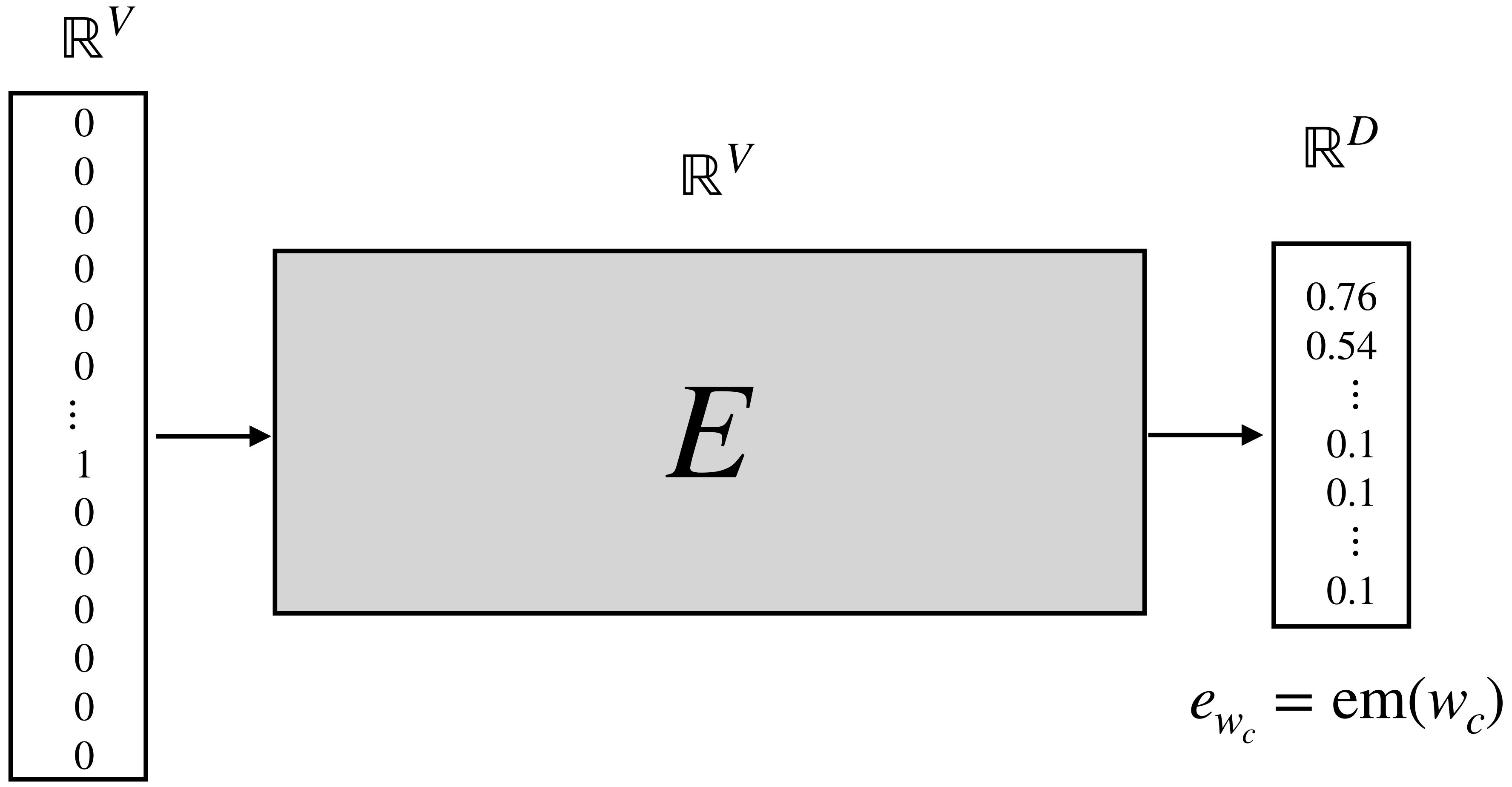
  And outputs

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

- Note that the previous expression can be written as:

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

  - This expression is not using a Matrix-vector anymore, it is simply getting the columns of E that are relevant



EMBEDDING LAYER VS LINEAR LAYER: SAME RESULTS, DIFFERENT EFFICIENCY

This code shows:

- *Equivalence of Embedding and Linear layer*
- *Forward pass speedup of Embedding vs Linear*

```
import torch
from torch import nn
num_embeddings = 10_000
embedding_dim = 200
E = 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(10_000)
x_onehot[vocab['house']] = 1
x_onehot[vocab['I']] = 1
```

```
# Create a linear layer that gets the same embeddings as the embedding layer
E_trans_weight = E.weight.transpose(0,1)
E_linear = nn.Linear(num_embeddings,embedding_dim,bias=False)
E_linear.weight = torch.nn.Parameter(E_trans_weight)
```

```
%timeit E.forward(x).sum(axis=0)
%timeit E_linear.forward(x_onehot)
```

60x faster!!

```
11 µs ± 105 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
660 µs ± 55.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

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

CBOW FORWARD PASS: FIRST LAYER EXAMPLE

- We have defined $E : \mathbb{R}^V \longrightarrow \mathbb{R}^D$ to be
  $$E(x) = \frac{1}{2C}\big(E[:, oh(w_{t-1})] + E[:, oh(w_{t-2})] + E[:, oh(w_{t+1})] + E[:, oh(w_{t+2})]\big)$$
  - Consider the example sentence with sliding window shaded
    "I love books because I love learning"
  - Consider vocab={am:0, because:1, happy:2, I:3, love:4, learning:5}
  - Then the input for E in this example would be

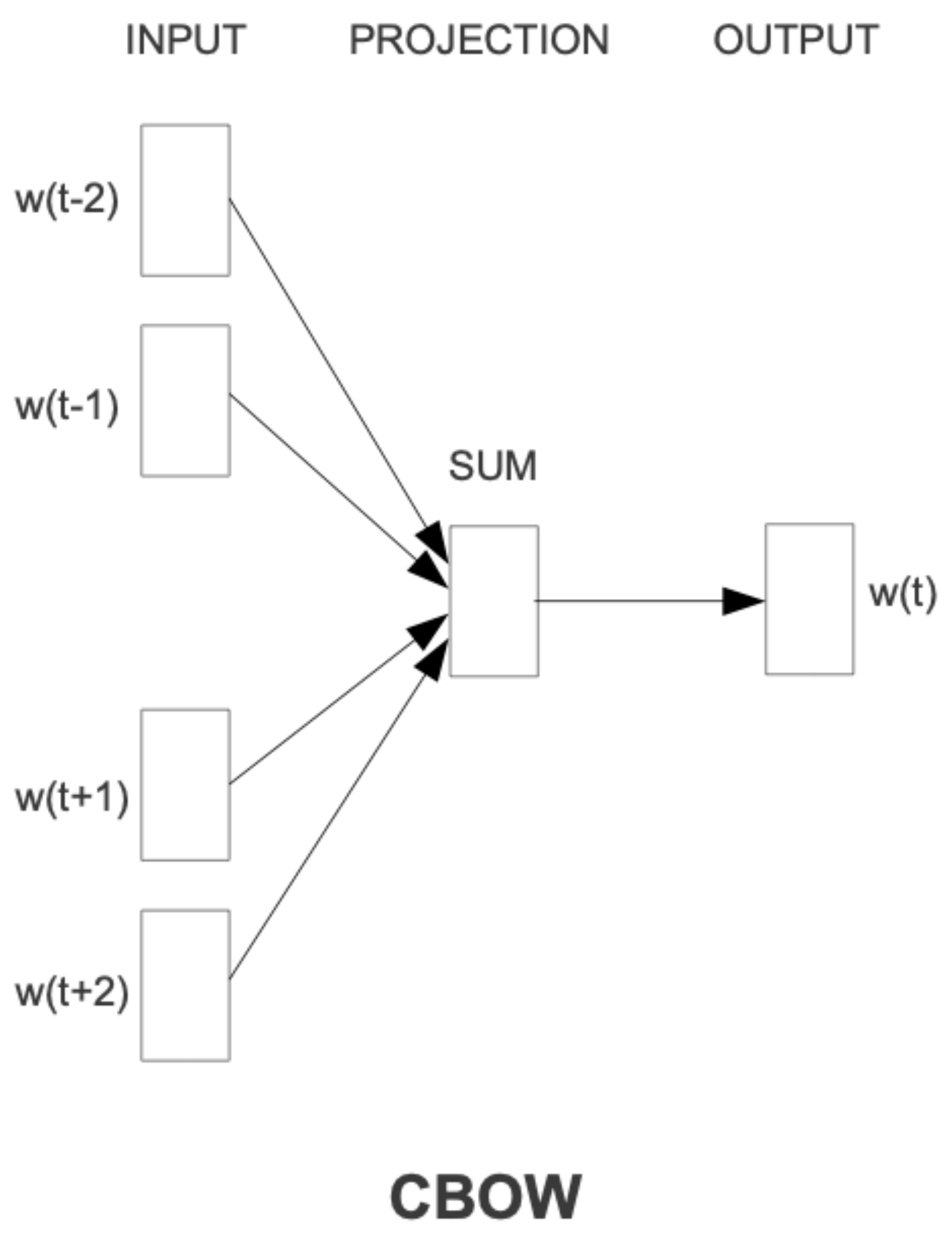
	I	love	because	I
am	0	0	0	0
because	0	0	1	0
happy	0	0	0	0
I	1	0	0	1
love	0	1	0	0
learning	0	0	0	0

$$\longrightarrow$$

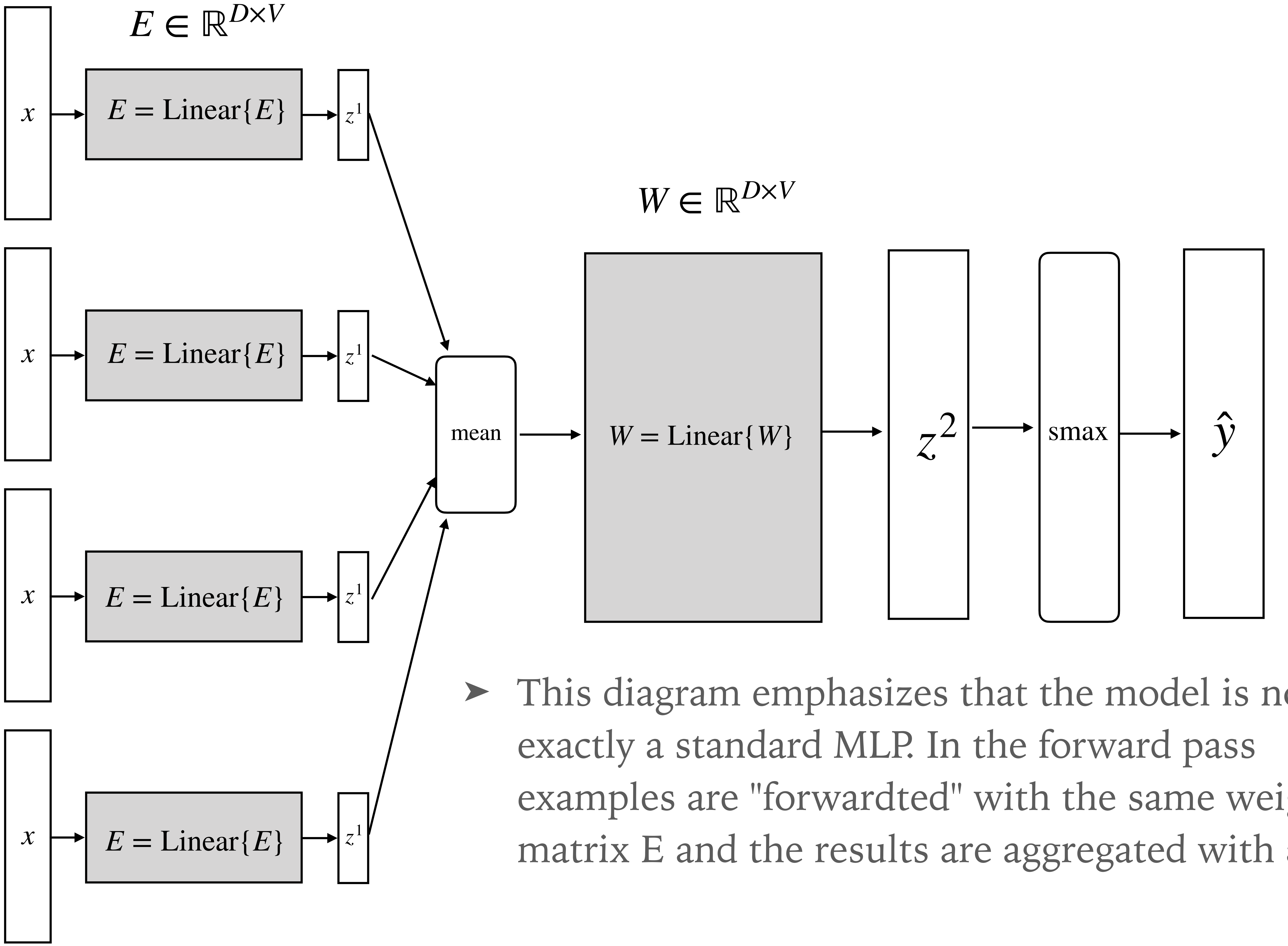
$$x = \frac{1}{4}\big(oh(I) + oh(love) + oh(because) + oh(I)\big)$$

0
0.25
0
0.5
0.25
0

CBOW FORWARD PASS: FIRST LAYER, ANOTHER VIEW



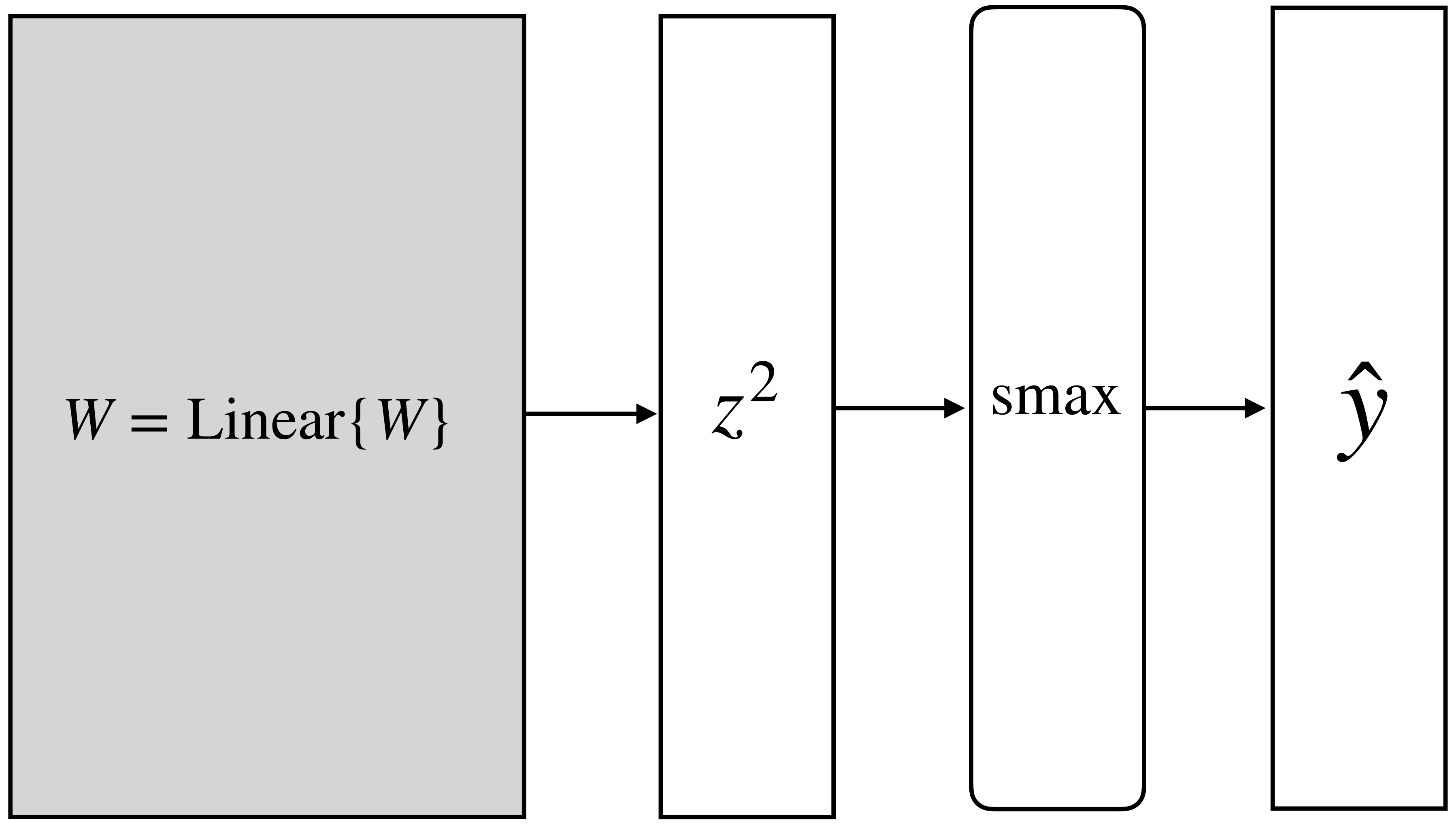
- Original diagram word2vec paper



- This diagram emphasizes that the model is not exactly a standard MLP. In the forward pass examples are "forwardted" 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.

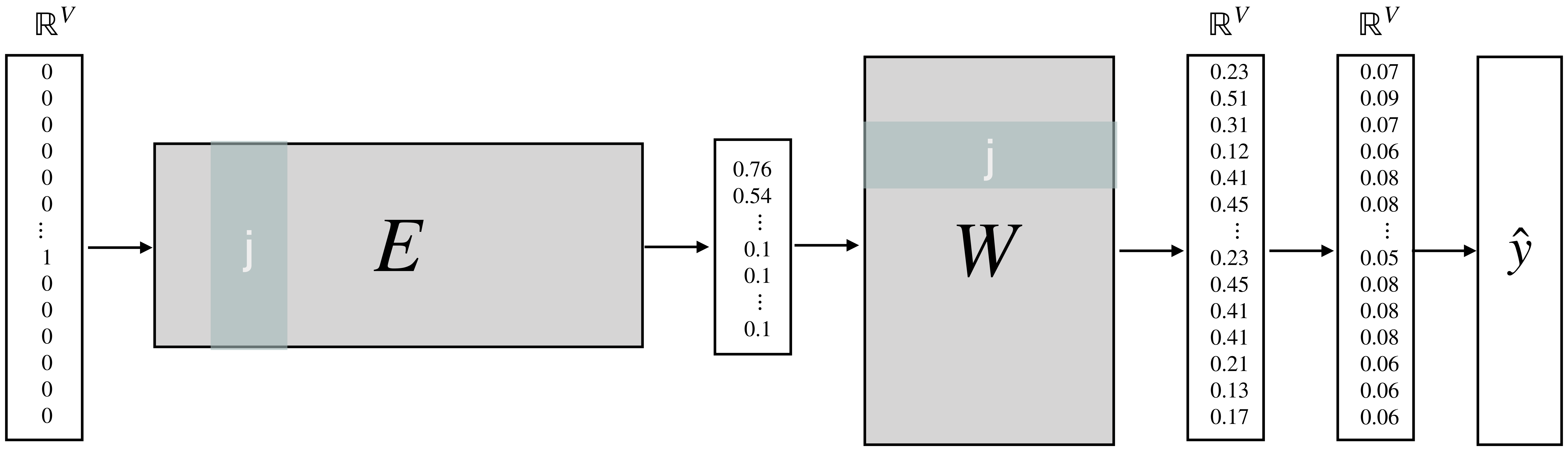


CBOW OVERVIEW

- Initialization step:
  - Iterate over all the corpus to find the words in the vocabulary
  - Build a vocab mapping that assigns a different integer to every word
- For a given sentence, select all possible windows. For each window do 1) to 4)
  - 1) Compute the embedding layer activation of the input $z^1 = E(x)$
  - 2) Generate output scores $z^2 = Wx$
  - 3) Turn scores into probabilities using a Softmax $\hat{y} = \text{smax}(z^2)$
  - 4) Compute the gradient of the cross-entropy loss and 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 'final word embeddings' as a mean over the two matrices.
    - The word embedding for word associated to position j is $\frac{1}{2}\left(E[:,j] + W[j,:]\right)$



FORWARD PASS: IMPROVING EFFICIENCY IN THE SECOND LAYER 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 $W : \mathbb{R}^D \longrightarrow \mathbb{R}^V$.
- Then the output of the linear layer z_2 passes over a Softmax.
- Since the Softmax requires normalizing over the vocabulary we could change a Softmax with V logistic regressions.

  - Doing so there is no need to compute $\sum_{j=1}^{V} e^{z_j^2}$

  - Now we can select positions at random to represent them 'negative' terms and update the logistic regressions of those positions.

  - The number of logistic regressions to be updated, k, is a hyperparameter of the negative sampling method.

WORD2VEC IN GENSIM

```python
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.

```python
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_vectors,axis=0)
```

WORD EMBEDDINGS CAN BE COMBINED WITH SPARSE REPRESENTATIONS

- One can stack sparse representations with dense representations with the goal 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	Count Vectorizer + Word2vec Average
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).
- (ICLR 2017): A simple but though-to-beat baseline for sentence embeddings
- (NAACL-2019): Vector of Locally-Aggregated Word Embeddings (VLAWE): A Novel Document-level Representation
- (AAAI 2020): P-SIF Document Embeddings Using Partition Averaging
- (ICPR 2020): Efficient Sentence Embedding via Semantic Subspace Analysis
- (ICNLSP 2021): Static Fuzzy Bag-ofWords: a Lightweight and Fast Sentence Embedding Algorithm