ELEC-E5150 — Exercise 3 part 2: Neural Language Models

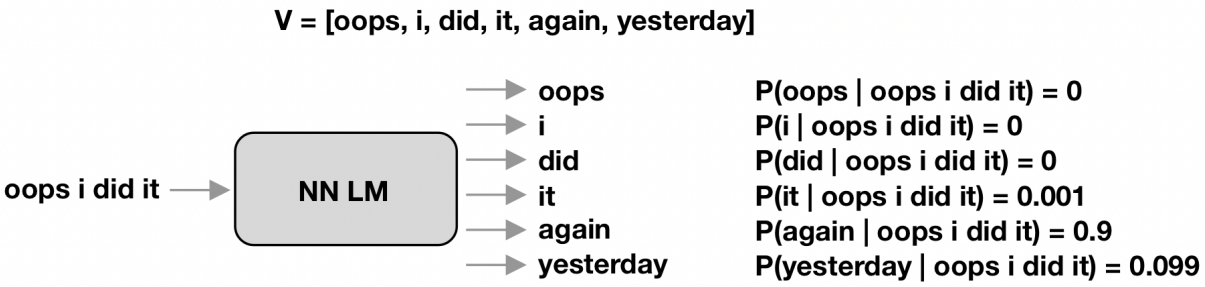
TASK: Generate an ending for a sequence of words using two pre-trained neural language models: an FFNN and an RNN.

The goal of this task is for you to gain the understanding of how neural language models work. You will learn the difference between Feed Forward and Recurrent neural language models.

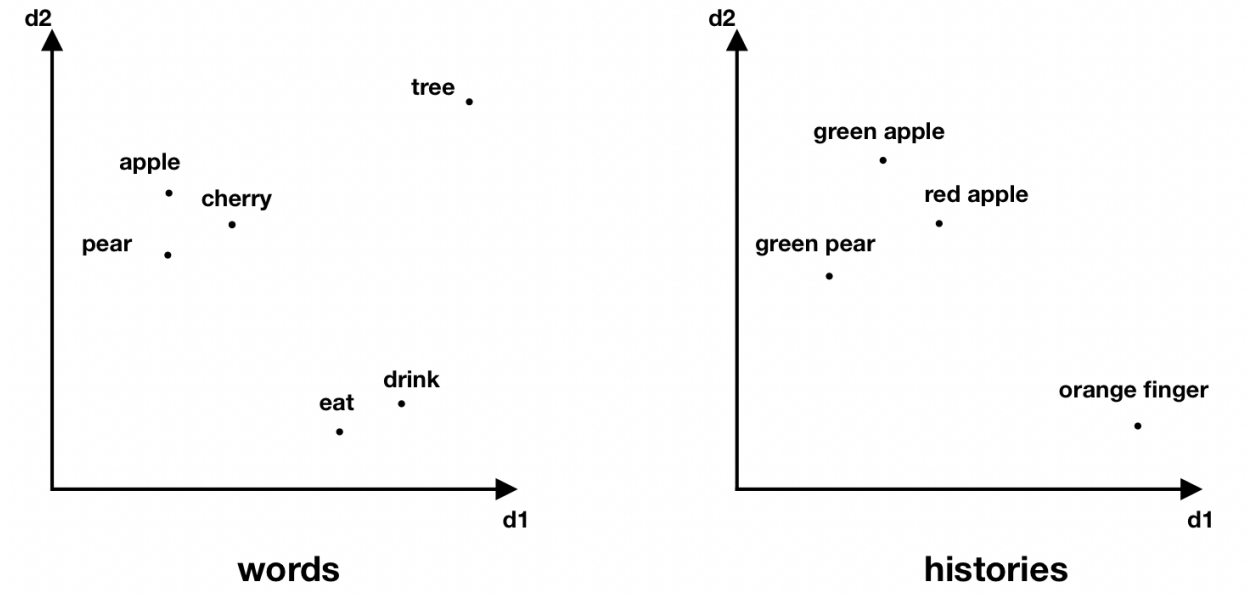
Neural Language Models (LMs)

We can think of neural language models simply as a tool for predicting upcoming words.

In this sense, language modelling is a multi-class classification problem, where the number of classes is the number of words in our vocabulary V. A neural language model is trying to predict a word given its history.



A language model is learning useful representations of words' histories to maximize the probability of a word given its history. Representations here are just vectors. A model is trying to put similar words and histories closer together in a vector space.



In this task, we'll walk through two types of neural LMs to see how they construct the representations of word histories. You will be given parameters (weights) for a Feed Forward Neural Network (FFNN) and a Recurrent Neural Network (RNN) LMs. Using these parameters, you'll need to generate endings for a beginning of a word sequence. For example, given **oops i did**, you'll need to first generate **it**, and then, using **oops i did it** you'll need to generate **again**.

First, let's see what NN LMs are supposed to be doing in general, and then we'll look at the models you'll be using for the task.

GENERAL Neural LM

step 1

Let's try predicting a word given a previous word. For neural networks, we need to represent all our words as numbers or (even better) as vectors of numbers. We can first assign every word in our vocabulary an id. Then, we can represent each word as a one-hot vector (a vector with 1 at the id position of a word and zeros everywhere else). We can interpret these vectors as a probability distribution for a word: we are 100% sure this word is itself.

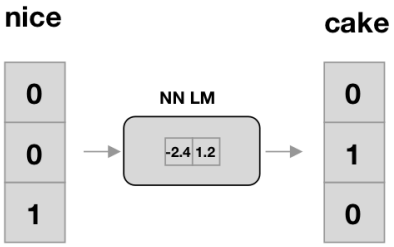
word	id	vector
a	0	[1,0,0]
cake	1	[0,1,0]
nice	2	[0,0,1]

	id	0	1	2	3	4
oops	0	1	0	0	0	0
i	1	0	1	0	0	0
did	2	0	0	1	0	0
it	3	0	0	0	1	0
again	4	0	0	0	0	1

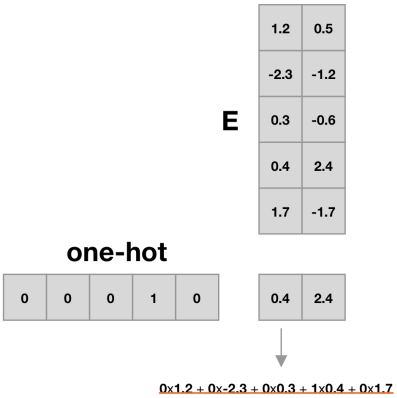
step 2

a	1.2	-0.3
cake	1.4	1.6
nice	-2.4	1.2

A neural network maps this one-hot representation into some useful distributed vector space representation and then maps it back to word space of one-hot vectors, **but now we hope to get a vector for the next word!** We don't really get a one-hot vector out of our Neural LM, we get a vector with probabilities for all the words in our vocabulary to follow the history we gave. The transformations in the example above are happening through matrix multiplication.



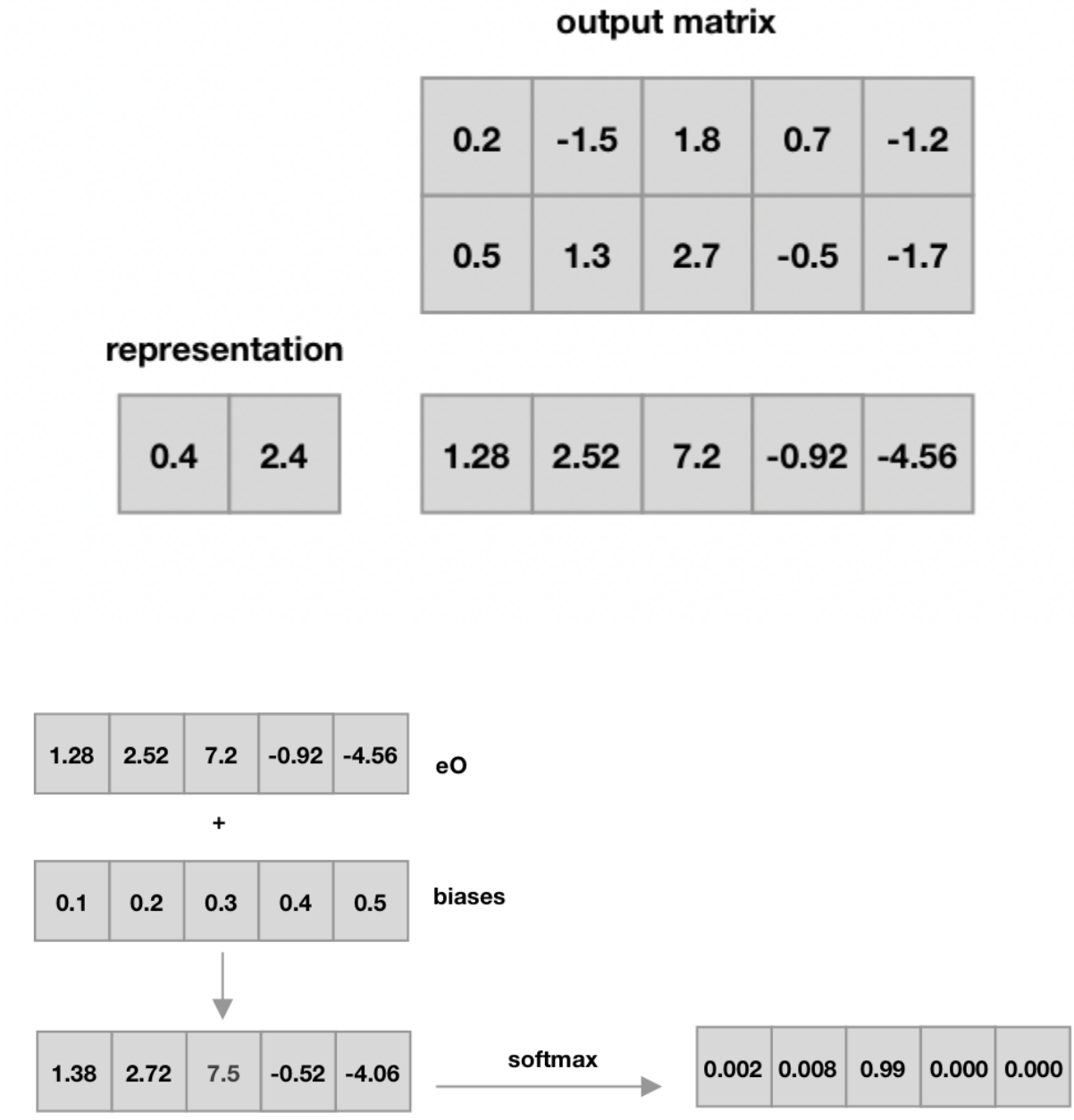
First, we take our one-hot representation and multiply it by embedding matrix $e = v_{hot}E$ and we'll get a row for our word. The embedding matrix E is the parameters (weights) that a neural network learns. They are the dimensions to map similar words close to each other.



In this assignment, we are going to use row vectors. This means, v_{hot} is a vector of 1xV dimensionality, where V is the number of words in our vocabulary.

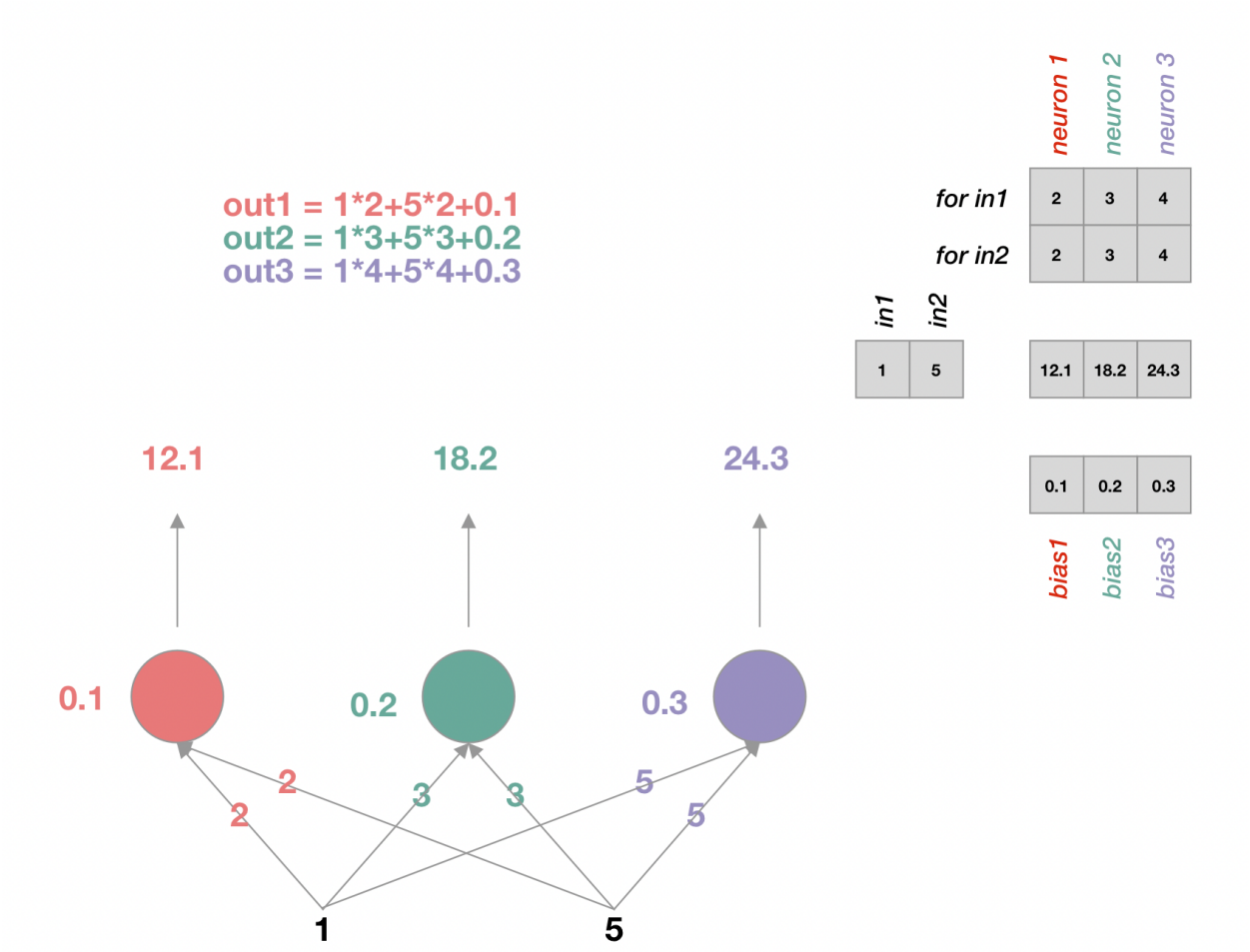
step 3

Then, we take this hidden word representation and transform it back into one-hot dimensionality by multiplying it by output matrix with dxV dimensions and adding bias. Here d is the number of dimensions our representation has. To transform this vector into probabilities we then apply softmax to it $softmax(eO + b_o)$.



Note: To just predict the most probable ending for a given history, you don't really need softmax. You can just choose the largest value in the $eO + b_o$ vector (make sure you understand why). But for the sake of this assignement you need to apply it.

To understand the connection between matrix multiplication and the structure of a neural network you can take a look at the diagram below. It shows a vector with 2 dimensions (possibly representing a word) going into a layer with three neurons. Each neuron has 2 weights (for each dimension in the input) and a bias associated with it, and those weights are stored in the matrices like E and O!

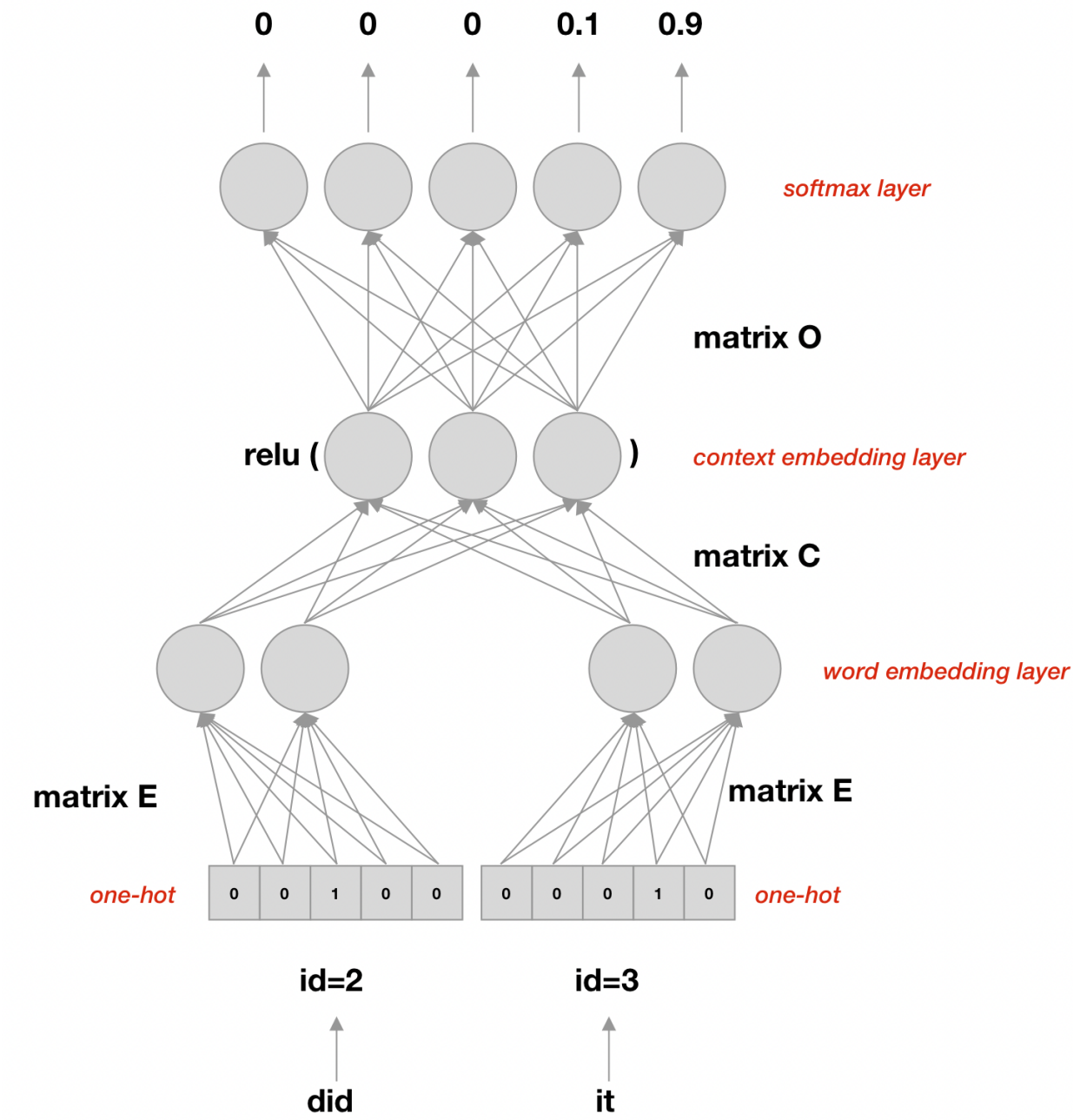


FFNN LM

Our feedforward LM is very similar to the general case, only it transforms our texts not just into hidden *word* representations, but also into hidden *context* representations (it can make a vector for red apple vs apple). We embed not just previous words but previous word pairs (or it can be triplets and longer). You can view this model as a kind of n-gram extension. The embeddings for two previous layers are concatenated and then transformed into a context representation h.

$$
\begin{aligned}
e &= (v_{hot1}E, v_{hot2}E) \\
h &= relu(eC + b_c) \\
z &= hO + b_o \\
y &= softmax(z)
\end{aligned}
$$

This is how our FFNN architecture looks like.

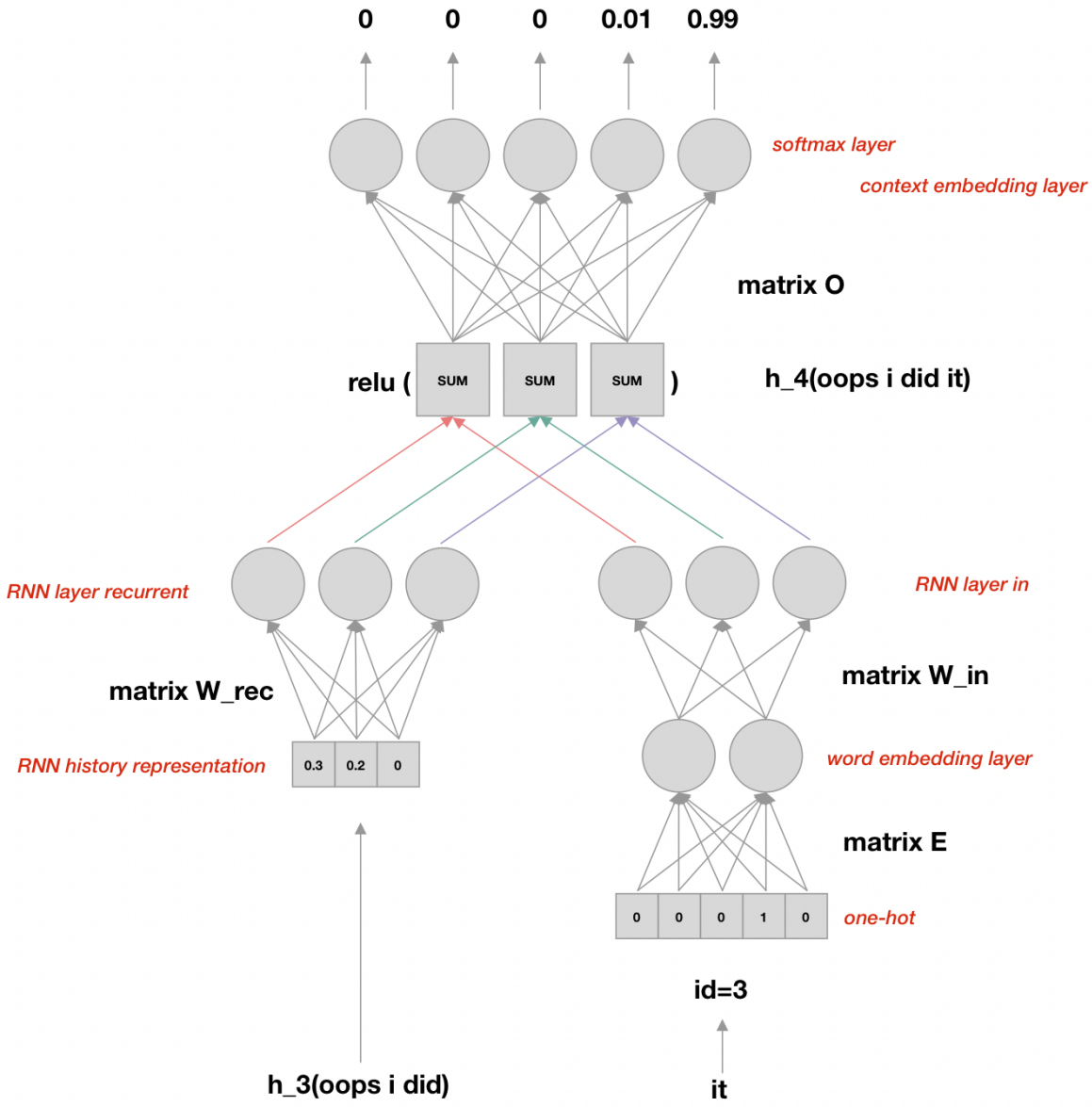


RNN LM

The principle for RNN is still very similar, it only differs in the way it represents the history. Instead of concatenating word representations, it sums representation for the current word with the representation for all the words before it. Below is the way we calculate a prediction for the second word in a sequence.

$$
\begin{aligned}
e_{w2} &= v_{hot2} E \\
h_2 &= relu(e_{w2} W_{in} + b_{in} + h_1 W_{rec} + b_{rec}) \\
z &= h_2 O + b_o \\
y &= softmax(z)
\end{aligned}
$$

This is how our RNN architecture looks like at the step of predicting *again* after seeing *oops i did it*.



TASK 2.1

Starting with a seed history, predict the next word. Use this predicted word as a part of the history for the next prediction. Keep predicting until you hit the end of sentence token. Report the word sequences you'll get, along with the softmax layer output that each word got.

You should return something like:

seed = oops i

did (0.9)

it(0.8)

again(0.9)

TASK 2.2 Explain how and why the predictions from FFNN LM and RNN LM differ. How both models differ from n-gram models?

General steps for both networks.

step 1

convert words in the seed sequence into their indices.

step 2

using the indices, extract embeddings from the embedding matrix (by multiplying their one-hot embeddings with the embedding matrix E)

FFNN:

step 3

concatenate the embeddings for seed words. take this concatenated vector and produce a hidden context representation.

$e = (v_{hot1}E, v_{hot2}E)$

$h = relu(eC + b_c)$

step 4

transform the hidden representation back to one-hot dimensionality and apply softmax to get probabilities over words:

step 5

choose the index of an element with the highest probability in vector output by softmax. This index will tell you which word was predicted

step 6

if the word that was predicted is not the end of sentence symbol '</s>', take this predited word as a second word in a history and predict again.

RNN:

step 3

put the first word embedding into the RNN. The first history vector h_0 will be zero.

$$e_{w1} = v_{hot1} E$$
$$h_1 = relu(e_{w1} W_{in} + b_{in} + + b_{rec})$$

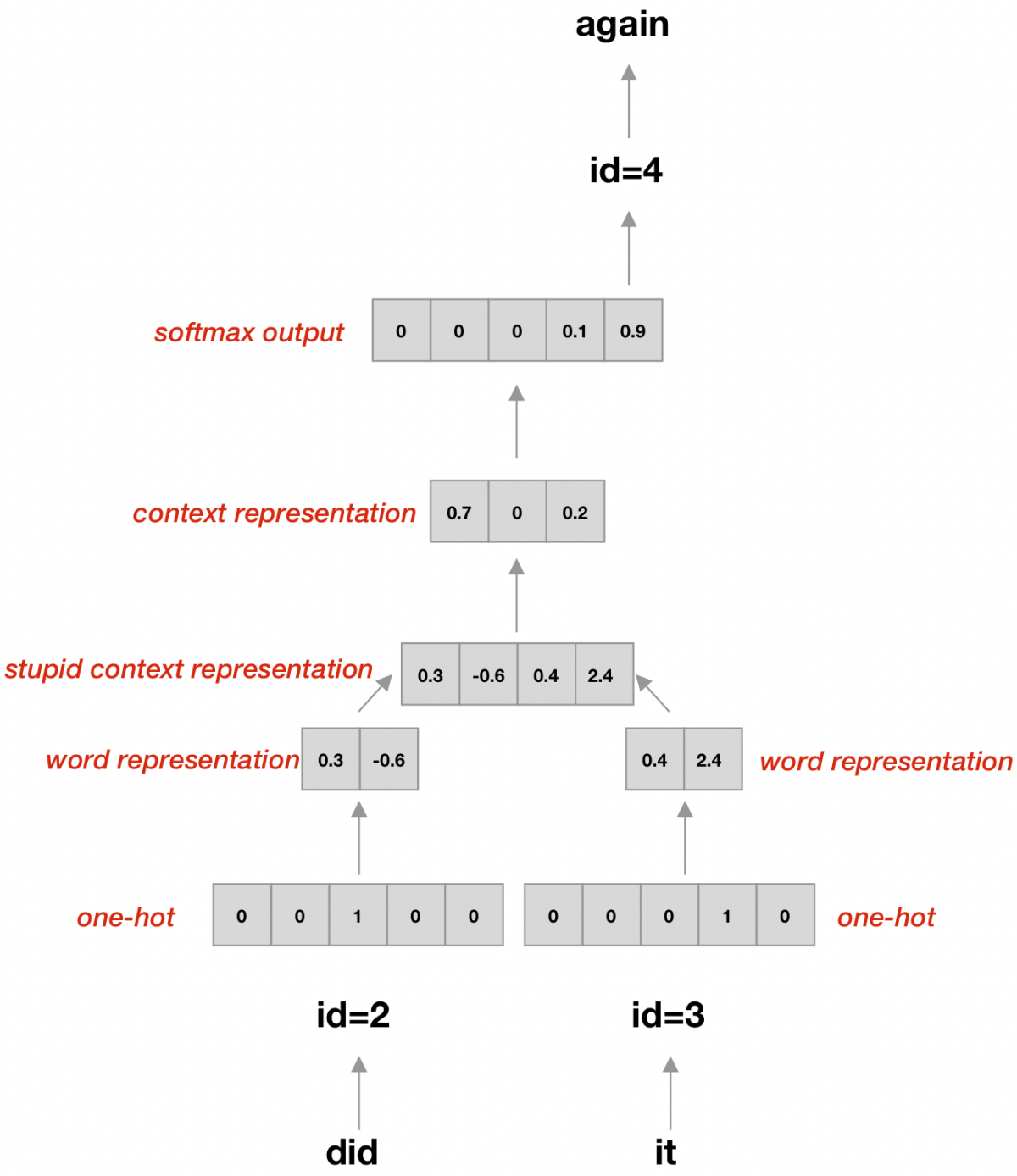
step 4

put the second word embedding into RNN, and then the third one. Append the first prediction word you make to the history and keep on predicting.

$$z = h_3 O + b_o$$
$$y = softmax(z)$$

flow of the FFNN

Below you can see how vectors go through the network.

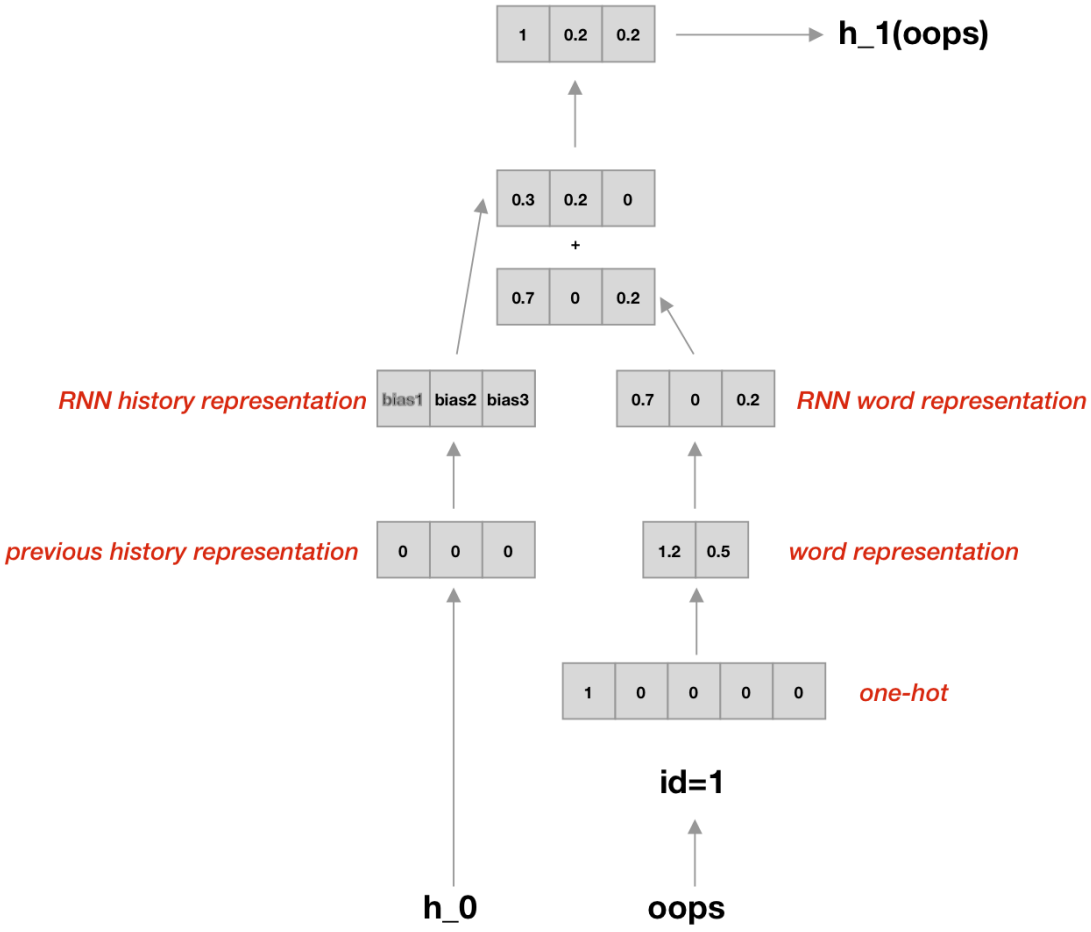


flow of the RNN

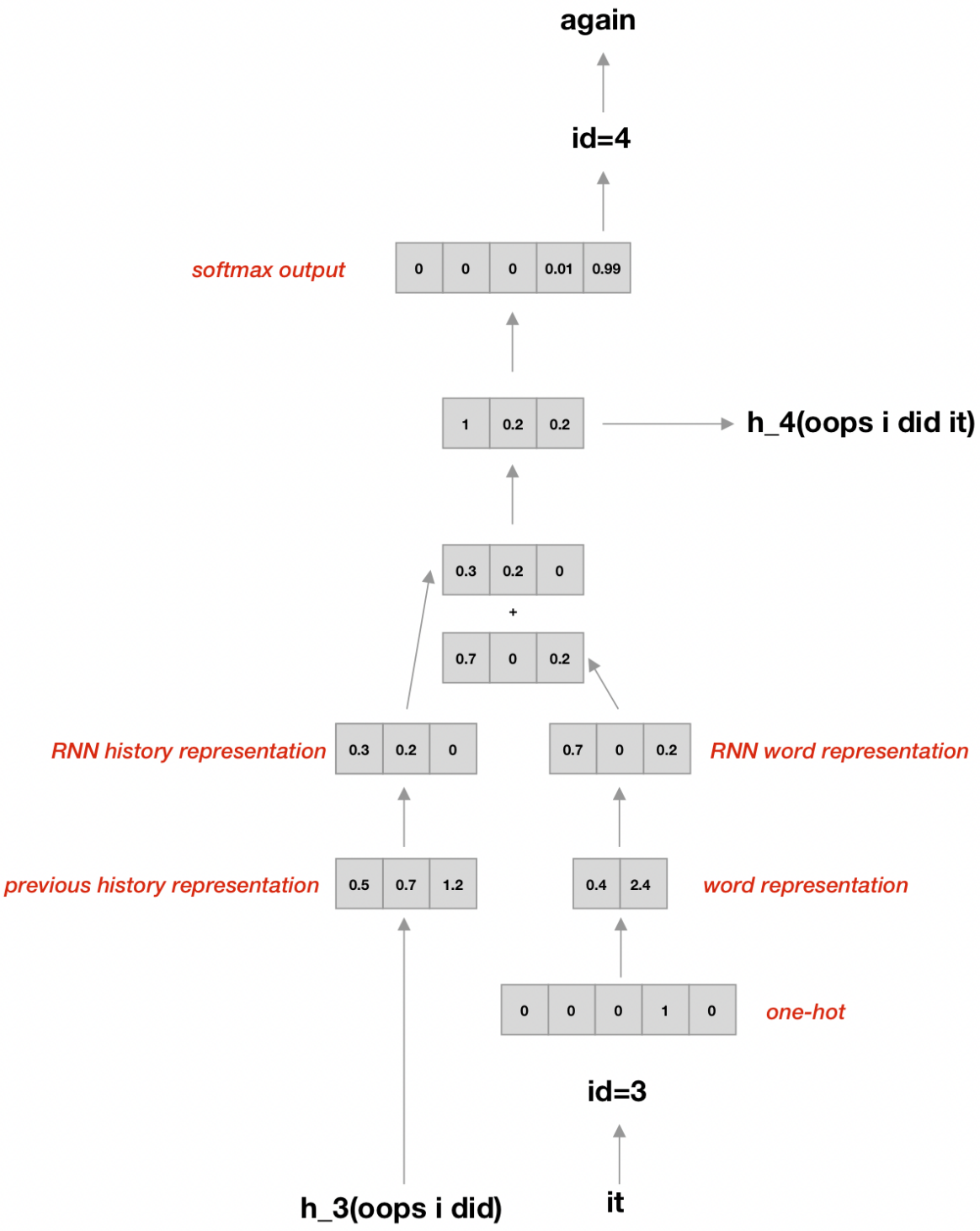
the flow of the RNN has two stages:

1. the first history word in
2. the rest of the words in history in

Here is how the information flows for the very first word



And here is how the information flows for the rest of the history.


again
id=4
softmax output
0
0
0
0.01
0.99
1
0.2
0.2
h_4(oops i did it)
0.3
0.2
0
+
0.7
0
0.2
RNN history representation
0.3
0.2
0
0.7
0
0.2
RNN word representation
previous history representation
0.5
0.7
1.2
0.4
2.4
word representation
0
0
0
1
0
one-hot
id=3
h_3(oops i did)
it