Lect 4:Word2vec

Negative sampling

Predictive models

- **Predictive models** directly try to predict a word from its neighbors in terms of learned small, dense embedding vectors (considered parameters of the model).

- **Neural-network-inspired models:**
  - **word2vec** (Mikolov et al., 2013)
  - **FastText** (Bojanowski et al., 2016)

Continuous bag-of-words (CBOW) vs Skip-Gram Models

Skip-gram:
works well with small amount of the training data, represents well even rare words or phrases

CBOW:
several times faster to train than the skip-gram, slightly better accuracy for the frequent words

Skip-Gram model is a better choice most of the time due to its ability to predict infrequent words, but this comes at the price of increased computational cost.

If training time is a big concern, and you have large enough data to overcome the issue of predicting infrequent words, CBOW model may be a more viable choice.

Vanilla Skip-Gram

W_output (old)

-0.560	0.340	0.160
-0.910	-0.440	1.560
-1.210	-0.130	-1.320
1.670	-0.150	-1.030
1.720	-1.460	0.730
0.000	1.390	-0.120
-0.060	1.520	-0.790
0.800	1.850	-1.670
-1.370	1.320	-0.480
0.670	1.990	-1.850
-1.520	-1.740	-1.860

(11X3)

− Learning R. 0.05 X

grad_W_output

0.064	0.071	-0.014
0.098	0.015	0.063
0.069	0.089	0.045
0.014	0.085	0.079
-0.021	0.067	0.071
-0.098	-0.088	0.091
-0.072	-0.078	-0.089
0.046	-0.079	-0.053
-0.049	-0.087	0.025
-0.060	0.092	0.042
0.074	0.050	0.070

(11X3)

= W_output (new)

-0.563	**0.336**	**0.161**
-0.915	**-0.441**	**1.557**
-1.213	**-0.134**	**-1.322**
1.669	**-0.154**	**-1.034**
1.721	**-1.463**	**0.726**
0.005	**1.394**	**-0.125**
-0.056	**1.524**	**-0.786**
0.798	**1.854**	**-1.667**
-1.368	**1.324**	**-0.481**
0.673	**1.985**	**-1.852**
-1.524	**-1.743**	**-1.864**

(11X3)

Negative Sampling

W_output (old)

-0.560	0.340	0.160
-0.910	-0.440	1.560
-1.210	-0.130	-1.320
1.670	-0.150	-1.030
1.720	-1.460	0.730
0.000	1.390	-0.120
-0.060	1.520	-0.790
0.800	1.850	-1.670
-1.370	1.320	-0.480
0.670	1.990	-1.850
-1.520	-1.740	-1.860

(11X3)

− Learning R. 0.05 X

grad_W_output

Not computed!

Positive sample, w_o	0.031	0.030	0.041
Negative sample, k=1	-0.090	0.031	-0.065
Negative sample, k=2	0.056	0.098	-0.061
Negative sample, k=3	0.069	0.084	-0.044

(11X3)

= W_output (new)

-0.560	0.340	0.160
-0.910	-0.440	1.560
-1.210	-0.130	-1.320
1.670	-0.150	-1.030
1.720	-1.460	0.730
0.000	1.390	-0.120
-0.060	1.520	-0.790
0.798	**1.849**	**-1.672**
-1.366	**1.318**	**-0.477**
0.667	**1.985**	**-1.847**
-1.523	**-1.744**	**-1.858**

(11X3)

Problems with Skip-Gram

For Example:

Training corpus has 10,000 unique vocabs
(V = 10000) [size of I/P and O/P Vectors are 10,000]

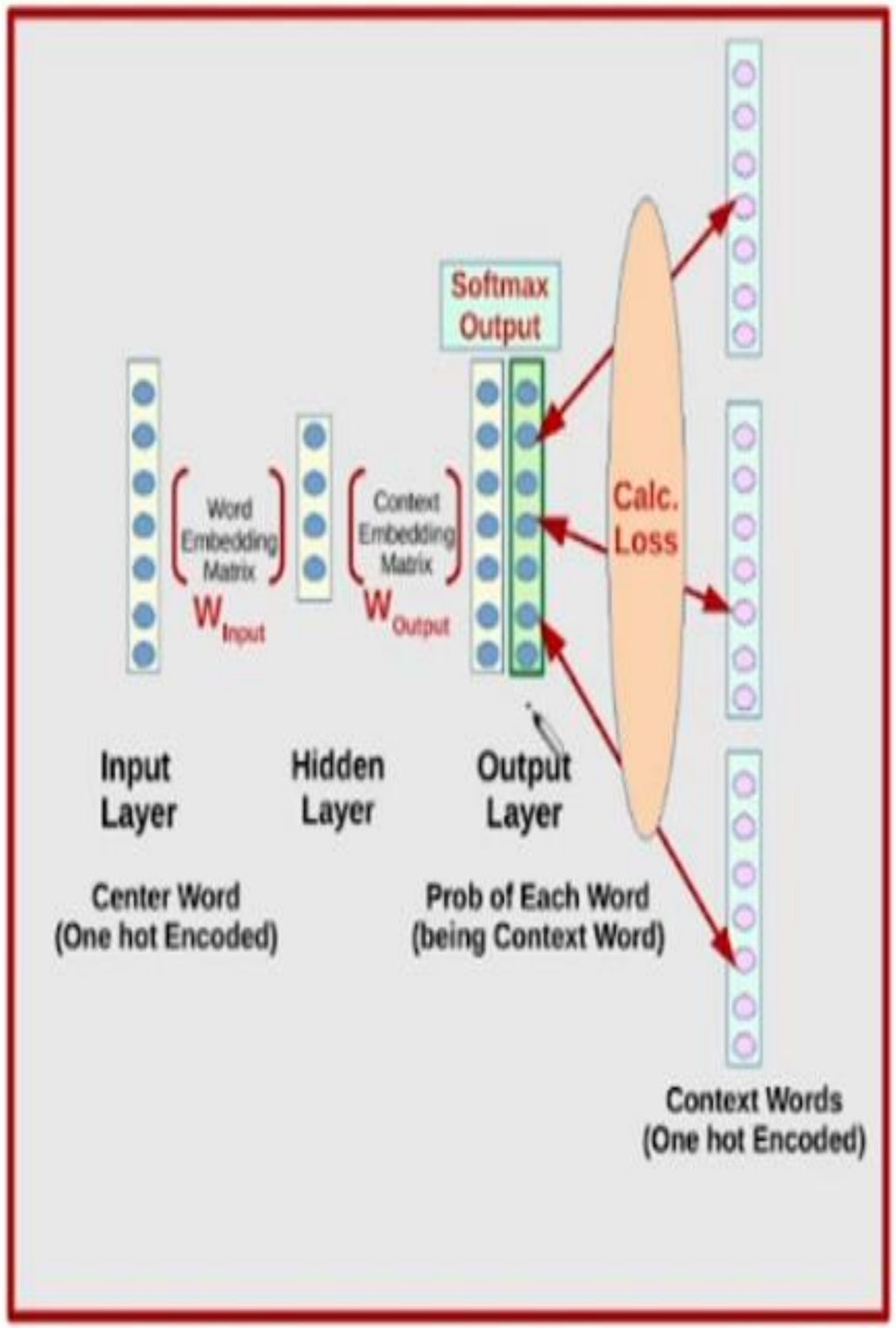
Hidden layer is 300-dimensional
(N = 300) [Embedding Size].

Number of Weights in the "Context Embedding Matrix" is 300*10,000 = 3,000,000 Weights

Softmax is calculated for Each Word in the Vocab (10,000 times)

[It is a Multi class classification Problems]

Number of classes = V = 10,000 Classes



Softmax is computationally very expensive

There is an issue with the vanilla Skip-Gram — softmax is computationally very expensive, as it requires scanning through the entire output embedding matrix (W_{output}) to compute the probability distribution of all V words, where V can be millions or more.

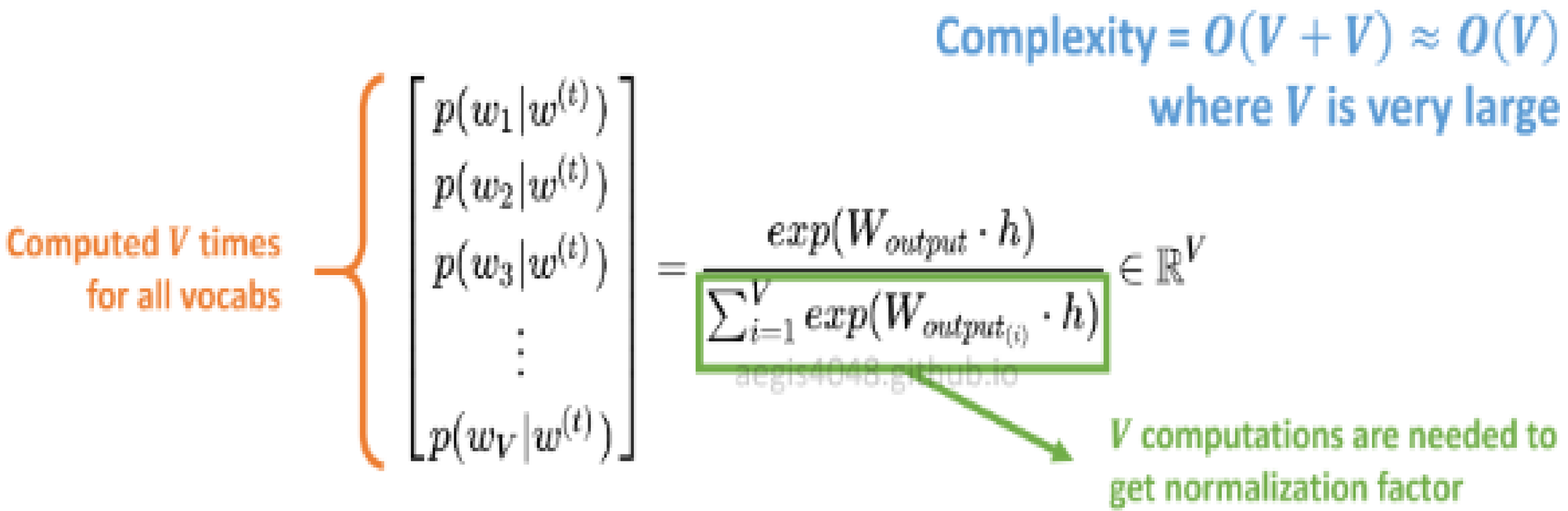


Figure 3: Algorithm complexity of vanilla Skip-Gram

Furtheremore, the normalization factor in the denominator also requires V iterations. In mathematical context, the normalization factor needs to be computed for each probability p($w_{context}|w_{center}$), making the alogrithm complexity = $O(V \times V)$. However, when implemented on code, the normalization factor is computed only once and cached as a Python variable, making the alogrithm complexity = $O(V + V) \approx O(V)$. This is possible because normalization factor is the same for all words.

Due to this computational inefficiency, **softmax is not used in most implementaions of Skip-Gram**. Instead we use an alternative called *negative sampling* with `sigmoid function`, which rephrases the problem into a set of independent binary classification task of algorithm complexity = $O(K + 1)$, where K typically has a range of $[5, 20]$.

Negative Sampling implementation of Skip-Gram

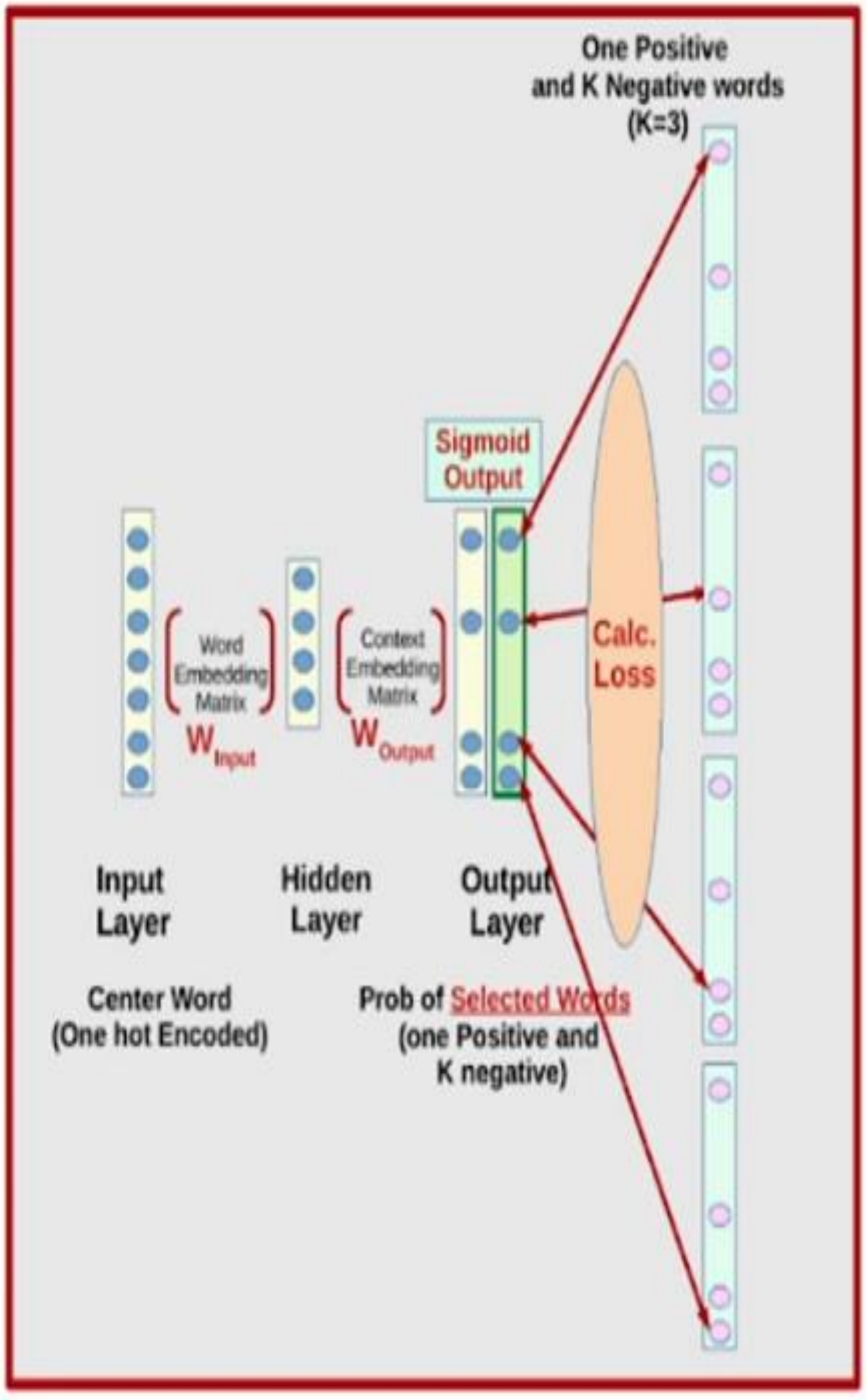
Negative Sampling:

For each training sample define:

- **Center Word (W_c)**
- **ONE Word in the Context of W_c**
  (Positive Context Sample C_{pos})
- **K Number of Words not in the context of W_c**
  (Negative Context Samples).

Notes:

- K is hyper parameter
- Negative Samples are selected Randomly from words (not in the context of W_c)
- Training sample= Pair of (W_c and C_{pos}) and K Negative Samples
- Construct multiple training samples for each word based on number of Context Words



Negative Sampling implementation of Skip-Gram

For Example:

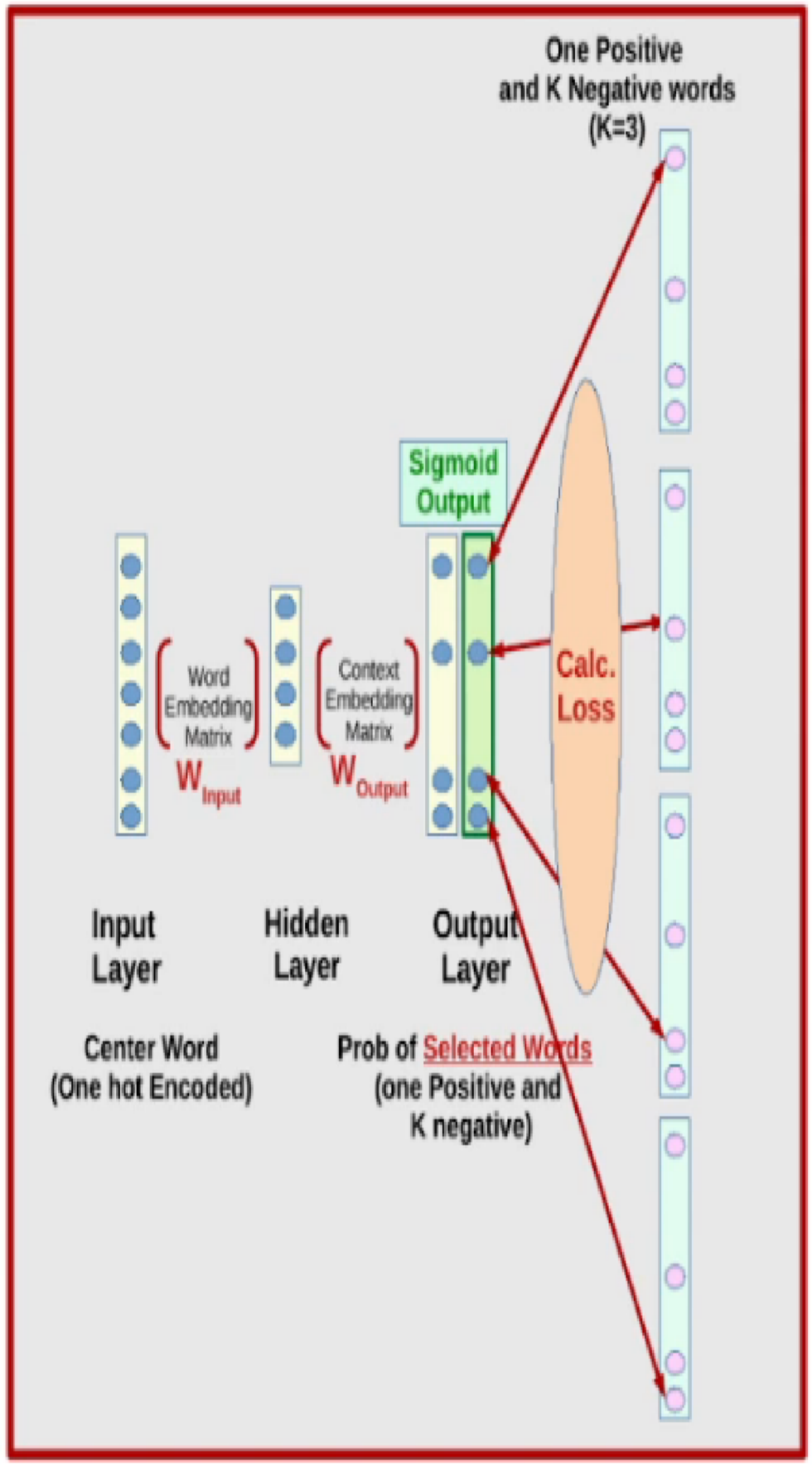
Training corpus has 10,000 unique vocabs
(V = 10000) [size of I/P and O/P Vectors are 10000]

Hidden layer is 300-dimensional
(N = 300) [Embedding Size].

Number of Weights in the "Context Embedding Matrix" is 300*10,000 = 3,000,000 Weights

Only Weights for selected K+1 Words are updated (300 * [K+1]) (instead of (300 * 10,000)

[It is a Binary class classification Problems]
Word is Positive (Context Word) or Negative
Use Sigmoid (instead of Softmax)

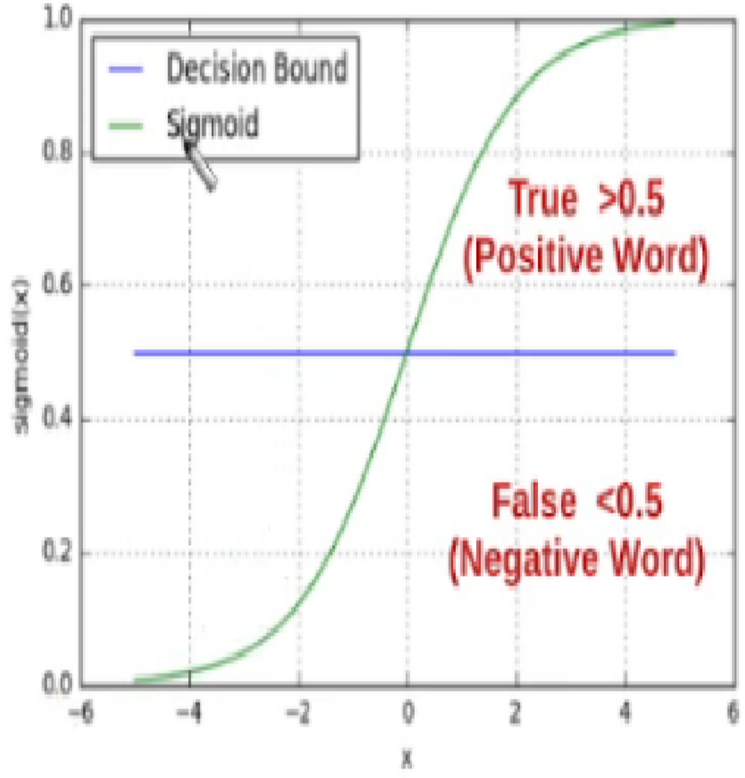


Why Using Sigmoid instead of Softmax

Negative sampling converts multi-classification task into binary-classification task.

The new objective is to predict, for any given word-context pair (w,c), whether the word (c) is in the context window of the the center word (w) or not.

In other words:
If the Training Sample is One positive word (W_{pos}) and 3 Negative words: W_{neg1}, W_{neg2} , W_{neg3} , then



Construct 4 pairs
$\{ W_c , W_{pos} \}, \{ W_c , W_{neg1} \} , \{ W_c , W_{neg2} \} , \{ W_c , W_{neg3} \}$

For each pair check if the candidate word is Positive (in context on W_c) or Negative

It is a binary classification problem

How does negative sampling work?

- **With negative sampling, word vectors are no longer learned by predicting context words of a center word**
- **Instead of using softmax to compute the V-dimensional probability distribution of observing an output word**

given an input word, p(wO|wI) the model uses sigmoid function to learn to differentiate the actual context words (*positive* (from randomly drawn words (*negative*)

Binomial Classification

(regression, logistic)	VS	(regression, zebra)
(regression, machine)		(regression, pimples)
(regression, sigmoid)		(regression, Gangnam-Style)
(regression, supervised)		(regression, toothpaste)
(regression, neural)		(regression, idiot)
Likely to observe		Unlikely to observe
$p(D = 1\|w, c_{pos}) \approx 1$		$p(D = 1\|w, c_{neg}) \approx 0$

The idea is that if the model can distinguish between the likely (positive) pairs vs unlikely (negative) pairs, good word vectors will be learned.

The idea is that **if the model can distinguish between the likely (positive) pairs vs unlikely (negative) pairs, good word vectors will be learned.**

The idea is that **if the model can distinguish between the likely (positive) pairs vs unlikely (negative) pairs, good word vectors will be learned.**

The probability of a word (c) appearing within the context of the center word (w) can be defined as the following :

$$p(D = 1|w, c; \theta) = \frac{1}{1 + exp(-\bar{c}_{output_{(j)}} \cdot w)} \in \mathbb{R}^1$$

where c is the word you want to know whether it came from the context window or the noise distribution. w is the input (center) word, and θ is the weight matrix passed into the model. Note that w is equivalent to the hidden layer (h). $\bar{c}_{output_{(j)}}$ is the word vector from the output weight matrix (W_{output}) of Figure 1.

- **This probability is computed K+1 times to obtain a probability distribution of a true context word and K negative samples :**

This probability is computed $K+1$ times to obtain a probability distribution of a true context word and K negative samples:

$$\begin{bmatrix} p(D=1|w, c_{pos}) \\ p(D=1|w, c_{neg,1}) \\ p(D=1|w, c_{neg,2}) \\ p(D=1|w, c_{neg,3}) \\ \vdots \\ p(D=1|w, c_{neg,K}) \end{bmatrix} = \frac{1}{1 + exp(-(\{\bar{c}_{pos}\} \cup \bar{W}_{neg}) \cdot h)} \in \mathbb{R}^{K+1} \quad (7)$$

Maximizing positive pairs and minimizing negative pairs

Center word: drilling

Current context word: engineer

	$K = 5$	$iter = 1$		$iter = 2$		$iter = 3$		$iter = 4$
engineer	$p(D=1\|w_{drilling}, c_{engineer})$	0.432		0.653		0.853		0.998
minimized	$p(D=1\|w_{drilling}, c_{minimized})$	0.312	Update θ	0.169	Update θ	0.042	Update θ	0.000
primary	$p(D=1\|w_{drilling}, c_{primary})$	0.565		0.221		0.083		0.001
concerns	$p(D=1\|w_{drilling}, c_{concerns})$	0.102		0.043		0.009		0.000
led	$p(D=1\|w_{drilling}, c_{led})$	0.853		0.532		0.115		0.001
page	$p(D=1\|w_{drilling}, c_{page})$	0.642		0.380		0.101		0.002

Derivation of cost function in negative sampling

The derivations written here are based on the work of word2vec Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method (Goldberg and Levy, 2014).

Consider a pair (w, c) of center word and its context. Did this pair come from the context window or the noise distribution? Let $p(D=1|w,c)$ be the probability that (w, c) is observed in the true corpus, and $p(D=0|w,c) = 1 - p(D=1|w,c)$ the probability that (w, c) is non-observed. There are parameters θ controlling the probability distribution: $p(D=1|w,c;\theta)$.

Negative sampling attempts to optimize the parameters θ by maximizing the probability of observing positive pairs (w, c_{pos}) while minimizing the probability of observing negative pairs (w, c_{neg}). For each positive pair, the model randomly draws K negative words $W_{neg} = \{c_{neg,j} | j = 1, \cdots, K\}$. Our objective function is then:

$$\underset{\theta}{\operatorname{argmax}}\ p(D=1|w,c_{pos};\theta) \prod_{c_{neg} \in W_{neg}} p(D=0|w,c_{neg};\theta) \tag{9}$$

$$= \underset{\theta}{\operatorname{argmax}}\ p(D=1|w,c_{pos};\theta) \prod_{c_{neg} \in W_{neg}} (1 - p(D=1|w,c_{neg};\theta)) \tag{10}$$

$$= \quad \underset{\theta}{\operatorname{argmax}} \; log\, p(D=1|w, c_{pos}; \theta) \prod_{c_{neg} \in W_{neg}} (1 - p(D=1|w, c_{neg}; \theta)) \tag{11}$$

Using the property of logs, the objective function can be simplified:

$$= \quad \underset{\theta}{\operatorname{argmax}} \; log\, p(D=1|w, c_{pos}; \theta) + log \prod_{c_{neg} \in W_{neg}} (1 - p(D=1|w, c_{neg}; \theta)) \tag{12}$$

$$= \quad \underset{\theta}{\operatorname{argmax}} \; log\, p(D=1|w, c_{pos}; \theta) + \sum_{c_{neg} \in W_{neg}} log\, (1 - p(D=1|w, c_{neg}; \theta)) \tag{13}$$

The binomial probability $p(D=1|w, c; \theta)$ can be replaced with eq-5:

$$= \quad \underset{\theta}{\operatorname{argmax}} \; log \frac{1}{1 + exp(-\bar{c}_{pos} \cdot \bar{w})} + \sum_{c_{neg} \in W_{neg}} log\, (1 - \frac{1}{1 + exp(-\bar{c}_{neg} \cdot \bar{w})}) \tag{14}$$

$$= \quad \underset{\theta}{\operatorname{argmax}} \; log \frac{1}{1 + exp(-\bar{c}_{pos} \cdot \bar{w})} + \sum_{c_{neg} \in W_{neg}} log \frac{1}{1 + exp(\bar{c}_{neg} \cdot \bar{w})} \tag{15}$$

Using the definition of sigmoid function $\sigma(x) = \frac{1}{1+exp(-x)}$:

$$= \quad \underset{\theta}{\operatorname{argmax}} \; log\, \sigma(\bar{c}_{pos} \cdot \bar{w}) + \sum_{c_{pos} \in W_{neg}} log\, \sigma(-\bar{c}_{neg} \cdot \bar{w}) \tag{16}$$

According to Mikolov, eq-16 replaces every $log\, p(w_O|w_I)$ in the vanilla Skip-Gram cost function defined in eq-1. Then, the cost function we want to minimize becomes:

$$J(\theta) = -\frac{1}{T}\sum_{i=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}(log\,\sigma(\bar{c}_{pos}\cdot\bar{w}) + \sum_{c_{neg}\in W_{neg}} log\,\sigma(-\bar{c}_{neg}\cdot\bar{w})) \tag{17}$$

However, when implemented in codes, batch gradient descent (making one update after iterating through the entire T corpus) is almost never used due to its high computational cost. Instead, we use stochastic gradient descent. Also, for negative sampling, gradients are calculated and weights are updated for each positive training pairs (w, c_{pos}). In the other words, one update for each word within the context window of a center word w. This is shown below. The new cost function is then:

$$J(\theta; w, c_{pos}) = -log\,\sigma(\bar{c}_{pos}\cdot\bar{w}) - \sum_{c_{neg}\in W_{neg}} log\,\sigma(-\bar{c}_{neg}\cdot\bar{w}) \tag{18}$$

Note that w is a word vector for an input word, and that it is equivalent to a hidden layer ($w = h$). For clarification:

$$J(\theta; w, c_{pos}) = -log\,\sigma(\bar{c}_{pos}\cdot h) - \sum_{c_{neg}\in W_{neg}} log\,\sigma(-\bar{c}_{neg}\cdot h) \tag{19}$$

Derivation of gradients

The goal of any machine learning model is to find the optimal values of a weight matrix (θ) to minimize prediction error. A general update equation for weight matrix looks like the following:

$$\theta^{(new)} = \theta^{(old)} - \eta \cdot \frac{\partial J}{\partial \theta} \tag{20}$$

θ is a parameter that needs to be optimized, and η is a learning rate. In negative sampling, we take the derivative to the cost function $J(\theta; w, c_{pos})$ defined in eq-19 with respect to θ. Note that the derivative of a sigmoid function is $\frac{\partial \sigma}{\partial x} = \sigma(x)(1-\sigma(x))$.

$$\frac{\partial J}{\partial \theta} = (\sigma(\bar{c}_{pos} \cdot h)) - 1)\frac{\partial \bar{c}_{pos} \cdot h}{\partial \theta} + \sum_{c_{neg} \in W_{neg}} \sigma(\bar{c}_{neg} \cdot h)\frac{\partial \bar{c}_{neg} \cdot h}{\partial \theta} \tag{21}$$

Since the parameter θ is a concatenation of the input and output weight matrix $[W_{input} \quad W_{output}]$, the cost function needs to be differentiated with respect to the both matrices — $[\frac{\partial J}{\partial W_{input}} \quad \frac{\partial J}{\partial W_{output}}]$.

2.3.1. Gradients with respect to output weight matrix $\frac{\partial J}{\partial W_{output}}$

With negative sampling, we do not update the entire output weight matrix W_{output}, but only a fraction of it. We update $K+1$ word vectors in the output weight matrix — $\bar{c}_{pos}$, $\bar{c}_{neg,1}$, ... $\bar{c}_{neg,K}$. We take partial derivatives to the cost function defined in eq-19 with respect to positive words and negative words. This can be done by replacing θ in eq-21 with $\bar{c}_{pos}$ and $\bar{c}_{neg}$ each.

$$\frac{\partial J}{\partial \bar{c}_{pos}} = (\sigma(\bar{c}_{pos} \cdot h) - 1) \cdot h \tag{22}$$

$$\frac{\partial J}{\partial \bar{c}_{neg}} = \sigma(\bar{c}_{neg} \cdot h) \cdot h \tag{23}$$

The update equations are then:

$$\bar{c}_{pos}^{(new)} = \bar{c}_{pos}^{(old)} - \eta \cdot (\sigma(\bar{c}_{pos} \cdot h) - 1) \cdot h \tag{24}$$

$$\bar{c}_{neg}^{(new)} = \bar{c}_{neg}^{(old)} - \eta \cdot \sigma(\bar{c}_{neg} \cdot h) \tag{25}$$

The gradients for positive and negative words can be merged for brevity:

$$\bar{c}_{j}^{(new)} = \bar{c}_{j}^{(old)} - \eta \cdot (\sigma(\bar{c}_{j} \cdot h) - t_j) \cdot h \tag{26}$$

where $t_j = 1$ for positive words ($c_j = c_{pos}$) and $t_j = 0$ for negative words ($c_j = c_{neg} \in W_{neg}$). $\bar{c}_j$ is the j-th word vector in the output word matrix ($\bar{c}_j \in W_{output}$). For each positive pairs (w, c_{pos}), eq-26 is applied to $K+1$ word vectors in W_{output} as shown in Figure 4.

Notes: Prediction error

In e-26, $\sigma(\bar{c}_j \cdot h) - t_j$ is called a *prediction error*. Recall that negative sampling attempts to maximize the the probability of observing positive pairs $p(c_{pos}|w) \rightarrow 1$ while minimizing the probability of observing negative pairs $p(c_{neg}|w) \rightarrow 0$. If good word vectors are learned, $\sigma(\bar{c}_{pos} \cdot h) \approx 1$ for positive pairs, and $\sigma(\bar{c}_{neg} \cdot h) \approx 0$ for negative pairs as shown in Figure 7.

The prediction error will gradually approach zero $\sigma(\bar{c}_{pos} \cdot h) - t_j \approx 0$, as the model iterates through the training samples (positive pairs) and optimizes the weights.

2.3.2. Gradients with respect to input weight matrix $\frac{\partial J}{\partial W_{input}}$

Just like vanilla Skip-Gram, only one word vector that corresponds to the input word w in W_{input} is updated with negative sampling. This is because the input layer is an one-hot-encoded vector (however, the equation for the gradient descent is different.) Therefore, taking the derivative for the input weight matrix is equivalent to taking the derivative to the hidden layer ($\frac{\partial J}{\partial W_{input}} = \frac{\partial J}{\partial h}$). We replace θ in eq-21 with h, and differentiate it:

$$\frac{\partial J}{\partial h} = (\sigma(\bar{c}_{pos} \cdot h) - 1) \cdot \bar{c}_{pos} + \sum_{c_{neg} \in W_{neg}} \sigma(\bar{c}_{neg} \cdot h) \cdot \bar{c}_{neg} \tag{27}$$

$$= \sum_{c_j \in \{c_{pos}\} \cup W_{neg}} (\sigma(\bar{c}_j \cdot h) - t_j) \cdot \bar{c}_j \tag{28}$$

Same as eq-26, $t_j = 1$ for positive words ($c_j = c_{pos}$) and $t_j = 0$ for negative words ($c_j = c_{neg} \in W_{neg}$). The update equation is then:

$$\bar{w}^{(new)} = \bar{w}^{(old)} - \eta \cdot \sum_{c_j \in \{c_{pos}\} \cup W_{neg}} (\sigma(\bar{c}_j \cdot h) - t_j) \cdot \bar{c}_j \tag{29}$$

Recall that w is a word vector in the input weight matrix ($\bar{w} \in W_{input}$) and $\bar{c}_j$ is a j-th word vector in the output weight matrix ($\bar{c}_j \in W_{output}$).

2.5. Numerical demonstration

Corpus = "Ned Stark is **the most honarable man**" + *all text in GoT*

Positive word pair: (`Ned` , `Stark`)

Recall that in negative sampling, one update is made for each of the positive training pairs. This means that C weight updates are made for each input (center) word, where C is the window size.

Our current positive word pair is (`Ned` , `Stark`). For the current positive pair, we `randomly draw` $K = 3$ negative words from the noise distribution: `pimples` , `zebra` , `idiot`

Forward Propagation: Computing hidden (projection) layer

Hidden layer (h) is *looked up* from W_{input} by multiplying the one-hot-encoded input vector with the input weight matrix W_{input}.

Current word-context pair = *(" Ned ", " Stark ")*
Current negative words = *"pimples", "zebra", "idiot"*

#	Token	x
0	honorable	0
1	is	0
2	man	0
3	most	0
4	**ned**	**1**
5	**stark**	**0**
6	the	0
	pimples	0
	zebra	0
Neg.	idiot	0
	coins	0
	donkey	0
	machine	0

×

W_{input}

-0.014	0.135	0.109
-0.665	0.669	0.309
0.702	0.621	-0.981
0.214	-0.813	-0.561
-0.018	0.404	-0.317
0.204	-0.007	-0.733
-0.652	-0.097	0.499
-0.706	0.745	0.002
-0.492	-0.709	0.133
-0.628	-0.073	0.502
0.456	0.767	-0.629
0.348	0.544	-0.740
-0.627	0.487	0.466

=

h

-0.018
0.404
-0.317

Figure 10: *Computing hidden (projection) layer*

Forward Propagation: Sigmoid output layer

Output layer is a probability distribution of positive and negative words ($c_{pos} \cup W_{neg}$), given a center word (w). It is computed with eq-7.

Recall that sigmoid function has $\sigma(x) = \frac{1}{1+exp(-x)}$.

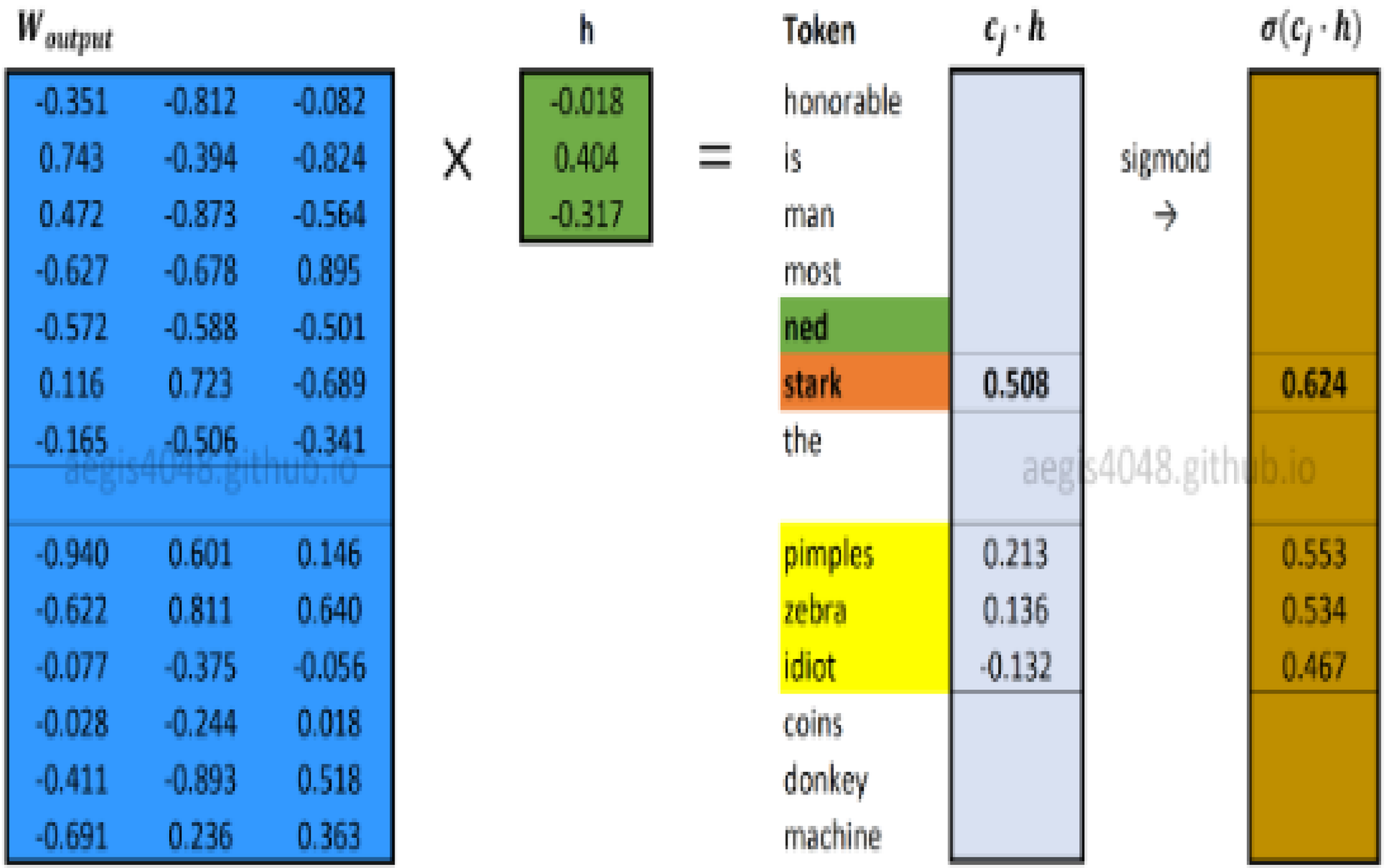


Figure 11: Sigmoid output layer

Backward Propagation: Prediction Error

The details about the prediction error is described above. Since our current positive word is `stark`, $t_j = 1$ for `stark` and $t_j = 0$ for other negative words (`pimples`, `zebra`, `idiot`).

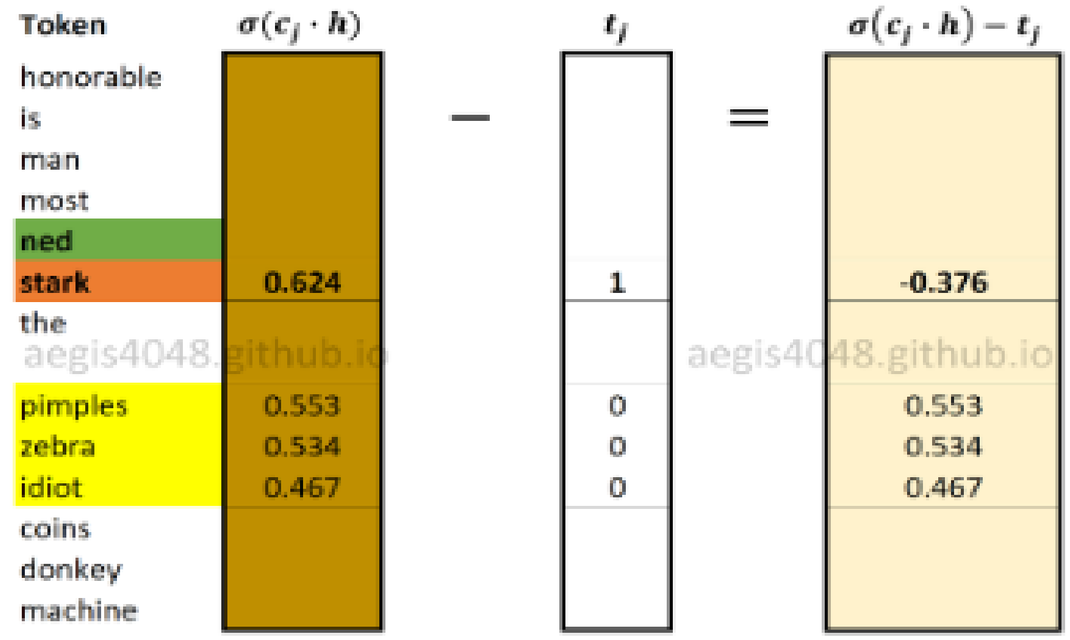


Figure 12: *Prediction errors of positive and negative words*

Backward Propagation: Computing ∇W_{input}

Gradients of input weight matrix ($\frac{\partial J}{\partial W_{input}}$) are computed using eq-28. Just like vanilla Skip-Gram, only the word vector in the input weight matrix W_{input} that corresponds to the input (center) word w is updated.

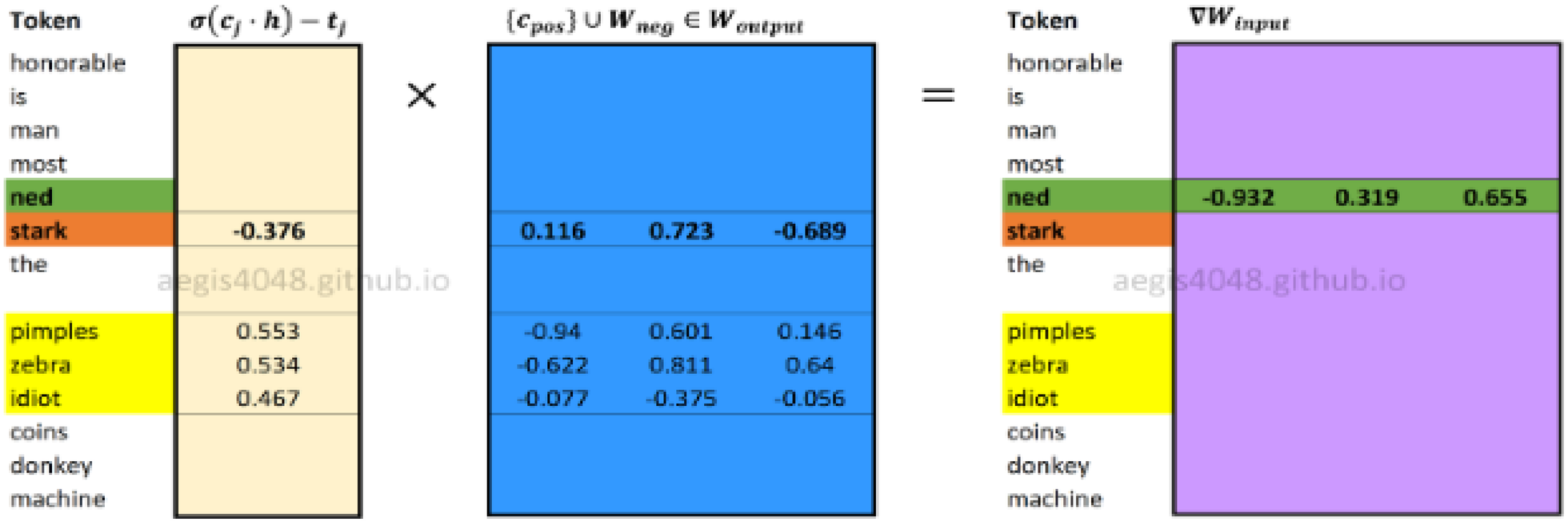


Figure 13: *Computing input weight matrix gradient* ∇W_{input}

Backward Propagation: Computing ∇W_{output}

With negative sampling, only a fraction of word vectors in the output weight matrix W_{output} is updated. Gradients for $K+1$ word vectors for positive and negative words in the W_{output} are computed using eq-22 and eq-23. Recall that K is the number of negative samples drawn from a noise distribution, and that $K=3$ in our example.

Token	$\sigma(c_j \cdot h) - t_j$
honorable	
is	
man	
most	
ned	
stark	**-0.376**
the	
pimples	0.553
zebra	0.534
idiot	0.467
coins	
donkey	
machine	

X

h
-0.018
0.404
-0.317

=

Token	∇W_{output}		
honorable			
is			
man			
most			
ned			
stark	0.007	-0.152	0.119
the			
pimples	-0.010	0.223	-0.175
zebra	-0.010	0.216	-0.169
idiot	-0.008	0.189	-0.148
coins			
donkey			
machine			

Figure 14: Computing output weight matrix gradient ∇W_{output}

Backward Propagation: Updating Weight matrices

Token	$W_{input}^{(old)}$			−	η	×	∇W_{input}			=	$W_{input}^{(new)}$		
honorable	-0.014	0.135	0.109		0.05						-0.014	0.135	0.109
is	-0.665	0.669	0.309								-0.665	0.669	0.309
man	0.702	0.621	-0.981								0.702	0.621	-0.981
most	0.214	-0.813	-0.561								0.214	-0.813	-0.561
ned	-0.018	0.404	-0.317				**-0.932**	**0.319**	**0.655**		**0.029**	**0.388**	**-0.350**
stark	0.204	-0.007	-0.733								0.204	-0.007	-0.733
the	-0.652	-0.097	0.499								-0.652	-0.097	0.499
pimples	-0.706	0.745	0.002								-0.706	0.745	0.002
zebra	-0.492	-0.709	0.133								-0.492	-0.709	0.133
idiot	-0.628	-0.073	0.502								-0.628	-0.073	0.502
coins	0.456	0.767	-0.629								0.456	0.767	-0.629
donkey	0.348	0.544	-0.740								0.348	0.544	-0.740
machine	-0.627	0.487	0.466								-0.627	0.487	0.466

aegis4048.github.io

Backward Propagation: Updating Weight matrices

Token	$W_{output}^{(old)}$			−	η	×	∇W_{output}			=	$W_{output}^{(new)}$		
honorable	-0.351	-0.812	-0.082		0.05						-0.351	-0.812	-0.082
is	0.743	-0.394	-0.824								0.743	-0.394	-0.824
man	0.472	-0.873	-0.564								0.472	-0.873	-0.564
most	-0.627	-0.678	0.895								-0.627	-0.678	0.895
ned	-0.572	-0.588	-0.501								-0.572	-0.588	-0.501
stark	0.116	0.723	-0.689				0.007	-0.152	0.119		**0.116**	**0.731**	**-0.695**
the	-0.165	-0.506	-0.341								-0.165	-0.506	-0.341
pimples	-0.94	0.601	0.146				-0.010	0.223	-0.175		**-0.940**	**0.590**	**0.155**
zebra	-0.622	0.811	0.64				-0.010	0.216	-0.169		**-0.622**	**0.800**	**0.648**
idiot	-0.077	-0.375	-0.056				-0.008	0.189	-0.148		**-0.077**	**-0.384**	**-0.049**
coins	-0.028	-0.244	0.018								-0.028	-0.244	0.018
donkey	-0.411	-0.893	0.518								-0.411	-0.893	0.518
machine	-0.691	0.236	0.363								-0.691	0.236	0.363

Figure 16: Updating W_{output}

Positive word pair: (`Ned` , `is`)

The center word `Ned` has two context words: `Stark` and `is` . This means that we have two positive pairs = two updates to make. Since we already update the matrices $[W_{input} \quad W_{output}]$ using (`Ned` , `Stark`), we will use (`Ned` , `is`) to update weight matrices this time.

In negative sampling, we draw new K negative words for each positive pairs. Assume that we randomly drew `coins` , `donkey` , and `machine` as our negative words this time.

Forward Propagation: Computing hidden (projection) layer

Current word-context pair = ***(" Ned ", " is ")***
Current negative words = ***"coins", "donkey", "machine"***

#	Token	x
0	honorable	0
1	**is**	**0**
2	man	0
3	most	0
4	**ned**	**1**
5	stark	0
6	the	0
	pimples	0
	zebra	0
Neg.	idiot	0
	coins	0
	donkey	0
	machine	0

$\times$

W_{input}

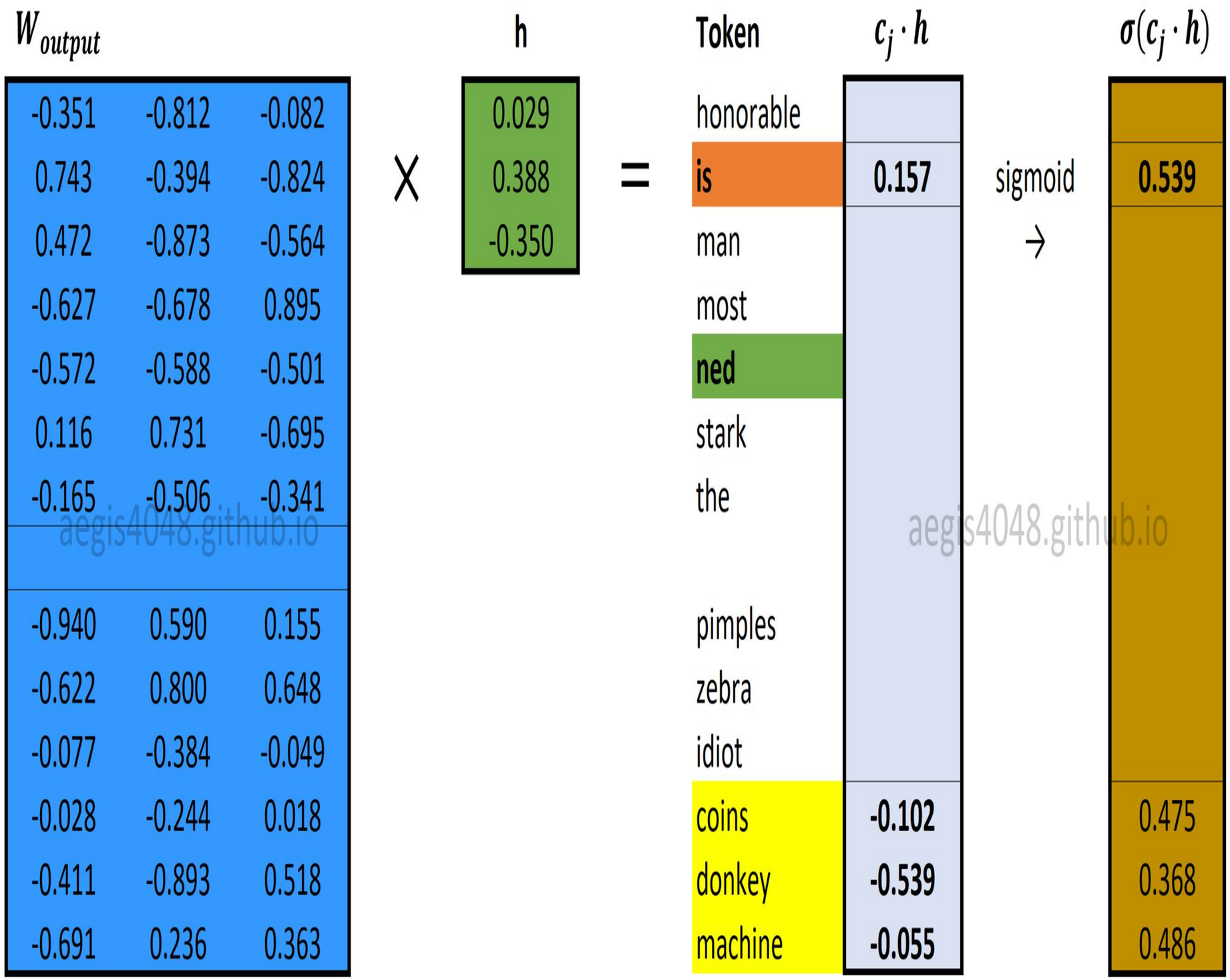
-0.014	0.135	0.109
-0.665	0.669	0.309
0.702	0.621	-0.981
0.214	-0.813	-0.561
0.029	0.388	-0.350
0.204	-0.007	-0.733
-0.652	-0.097	0.499
-0.706	0.745	0.002
-0.492	-0.709	0.133
-0.628	-0.073	0.502
0.456	0.767	-0.629
0.348	0.544	-0.740
-0.627	0.487	0.466

$=$

h
0.029
0.388
-0.350

Figure 17: *Computing hidden (projection) layer*

Sigmoid output layer



Backward Propagation: Prediction Error

Our current positive word is `stark` : $t_j = 1$ for `stark` and $t_j = 0$ for other negative words (`coins` , `donkey` , and `machine`).

Token	$\sigma(c_j \cdot h)$		t_j		$\sigma(c_j \cdot h) - t_j$
honorable					
is	0.539	−	1	=	-0.461
man					
most					
ned					
stark					
the					
pimples					
zebra					
idiot					
coins	0.475		0		0.475
donkey	0.368		0		0.368
machine	0.486		0		0.486

Figure 19: Prediction errors of positive and negative words

Backward Propagation: Computing ∇W_{input}

Token	$\sigma(c_j \cdot h) - t_j$		$\{c_{pos}\} \cup W_{neg} \in W_{output}$				Token	∇W_{input}		
honorable							honorable			
is	-0.461	×	0.743	-0.394	-0.824	=	is			
man							man			
most							most			
ned							ned	-0.843	-0.148	0.756
stark							stark			
the							the			
pimples							pimples			
zebra							zebra			
idiot							idiot			
coins	0.475		-0.028	-0.244	0.018		coins			
donkey	0.368		-0.411	-0.893	0.518		donkey			
machine	0.486		-0.691	0.236	0.363		machine			

Figure 20: Computing input weight matrix gradient ∇W_{input}

Backward Propagation: Computing ∇W_{output}

Token	$\sigma(c_j \cdot h) - t_j$
honorable	
is	-0.461
man	
most	
ned	
stark	
the	
pimples	
zebra	
idiot	
coins	0.475
donkey	0.368
machine	0.486

X

h
0.029
0.388
-0.350

=

Token	∇W_{output}		
honorable			
is	-0.013	-0.179	0.161
man			
most			
ned			
stark			
the			
pimples			
zebra			
idiot			
coins	0.014	0.184	-0.166
donkey	0.011	0.143	-0.129
machine	0.014	0.189	-0.170

Figure 21: Computing output weight matrix gradient ∇W_{output}

Token	$W_{input}^{(old)}$				η		∇W_{input}			=	$W_{input}^{(new)}$		
honorable	-0.014	0.135	0.109	−	0.05	X				=	-0.014	0.135	0.109
is	-0.665	0.669	0.309								-0.665	0.669	0.309
man	0.702	0.621	-0.981								0.702	0.621	-0.981
most	0.214	-0.813	-0.561								0.214	-0.813	-0.561
ned	**0.029**	**0.388**	**-0.350**				**-0.843**	**-0.148**	**0.756**		**0.071**	**0.395**	**-0.388**
stark	0.204	-0.007	-0.733								0.204	-0.007	-0.733
the	-0.652	-0.097	0.499								-0.652	-0.097	0.499
pimples	-0.706	0.745	0.002								-0.706	0.745	0.002
zebra	-0.492	-0.709	0.133								-0.492	-0.709	0.133
idiot	-0.628	-0.073	0.502								-0.628	-0.073	0.502
coins	0.456	0.767	-0.629								0.456	0.767	-0.629
donkey	0.348	0.544	-0.740								0.348	0.544	-0.740
machine	-0.627	0.487	0.466								-0.627	0.487	0.466

aegis4048.github.io

Token	$W_{output}^{(old)}$				η		∇W_{ouput}				$W_{output}^{(new)}$		
honorable	-0.351	-0.812	-0.082	−	0.05	X				=	-0.351	-0.812	-0.082
is	0.743	-0.394	-0.824				-0.013	-0.179	0.161		**0.744**	**-0.385**	**-0.832**
man	0.472	-0.873	-0.564								0.472	-0.873	-0.564
most	-0.627	-0.678	0.895								-0.627	-0.678	0.895
ned	-0.572	-0.588	-0.501								-0.572	-0.588	-0.501
stark	0.116	0.731	-0.695								0.116	0.731	-0.695
the	-0.165	-0.506	-0.341								-0.165	-0.506	-0.341
pimples	-0.940	0.590	0.155								-0.940	0.590	0.155
zebra	-0.622	0.800	0.648								-0.622	0.800	0.648
idiot	-0.077	-0.384	-0.049								-0.077	-0.384	-0.049
coins	-0.028	-0.244	0.018				0.014	0.184	-0.166		**-0.029**	**-0.253**	**0.026**
donkey	-0.411	-0.893	0.518				0.011	0.143	-0.129		**-0.412**	**-0.900**	**0.524**
machine	-0.691	0.236	0.363				0.014	0.189	-0.170		**-0.692**	**0.227**	**0.372**

aegis4048.github.io