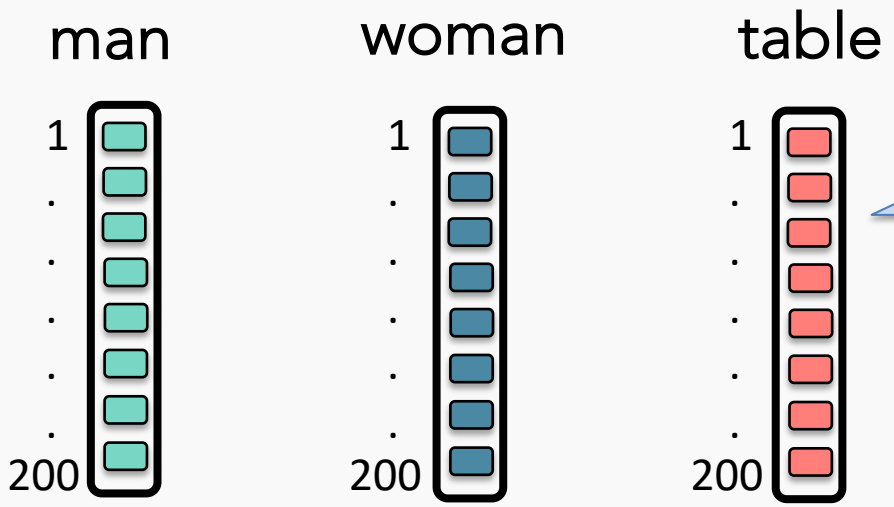
# Word Embeddings

Pavlos Protopapas

# RECAP: Word Embeddings

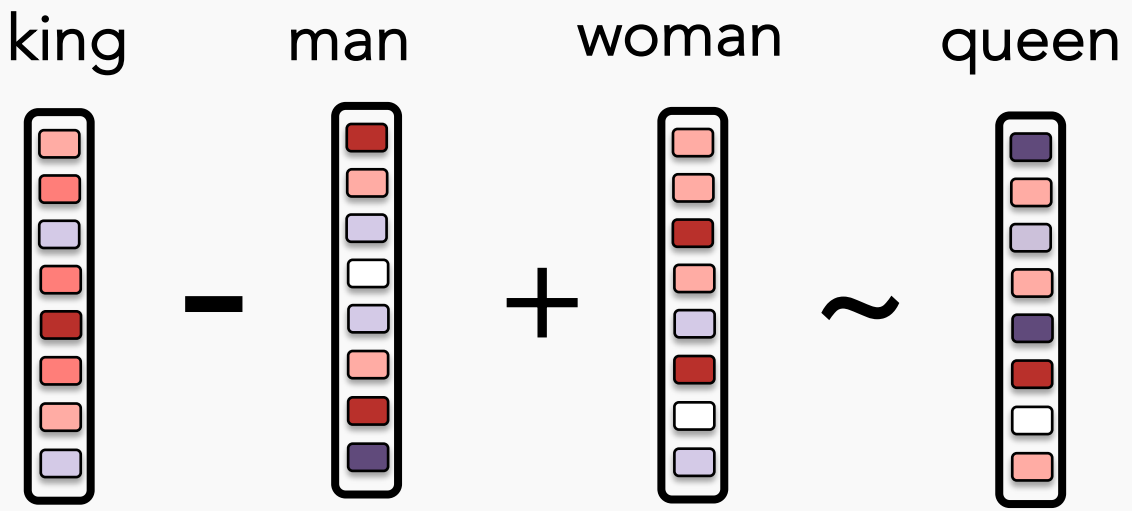
Each word is represented by a word embedding (e.g., vector of length 200)



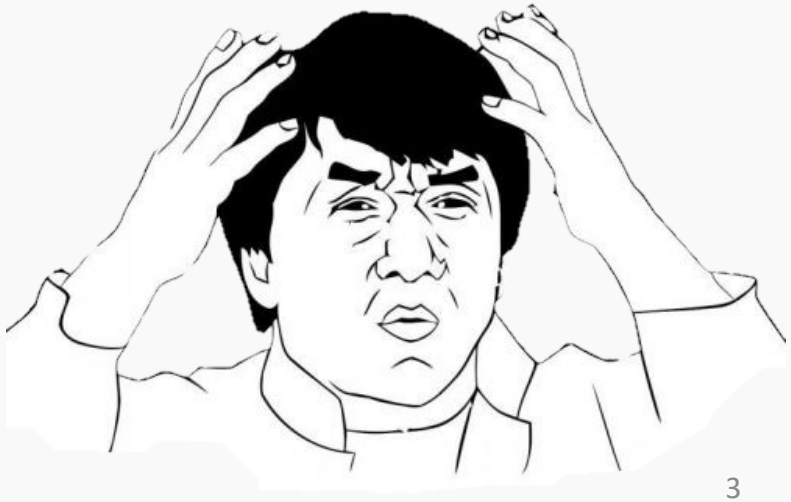

- Each rectangle is a *floating-point* scalar
- Words that are more *semantically similar* to one another will have **embeddings** that are also proportionally similar
- We can *use pre-existing* word embeddings that have been trained on gigantic corpora

# Recap: Word Embeddings

These word embeddings are so rich that you get nice properties:




**HOW?!?**

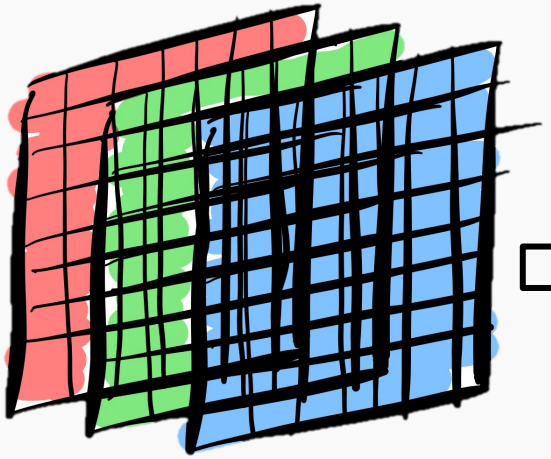


# Computer Vision vs Language models

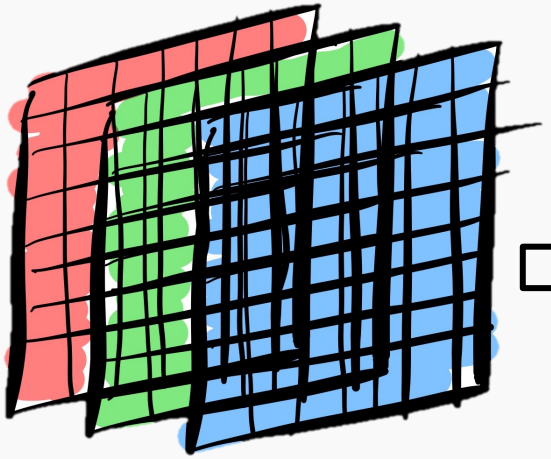
**CAT**

**DOG**

# Computer Vision vs Language models



```
array([[[ 63,  95, 208, 231, 190, 186, 196, 208,  91, 110,  31,  66,
         163,  75,  96,  84,  86, 106,   0,  57,  90, 147,  31, 148,
          63,  10,  61, 231,  80, 127, 174, 240,  45,  56, 153, 193,
         238,  85,  38, 118,  94, 145, 144, 192,  82,  48, 145,  17,
         176, 150,  43, 111,  55, 104, 249,  42,  25, 145,  88, 102,
         115,  12,  91, 238, 235, 198, 141,  96, 118, 167, 222, 123,
          43, 235,  10, 205,  68, 249, 208,  71, 112,  94, 146,   1,
          16, 134,   7,  94,  28, 148,  77,  58, 122, 232,  59, 194,
          16,   7, 204, 120, 174, 192,  72, 197,  95,  18, 188, 243,
          83, 201, 137, 250, 237, 149, 222, 150,  96,   2, 191,  15,
         164, 167, 147,  70, 193,  30,  33, 108,  65,  77, 192, 237,
         189,  47, 166, 116, 141, 107,  39, 158, 241,  33, 170,  37,
         102, 173,  82, 181, 229, 189, 161,  18, 251, 187,  95,  28,
          45, 132,  66, 201, 166, 243,  18, 135,   9, 171, 110,  74,
          99, 207, 185, 228, 243,  17, 236,  60,  61, 142,  12,  54,
         203,  33,  30,  97, 202, 231,  34, 137, 162, 172, 100, 233,
         116,  58, 172,  35,  91, 100,   2,   6, 221,  86, 176,  15,
          15, 117,  57,  54, 108, 158, 145, 155,  89, 116,  73,  28,
         169,  33, 175, 120, 129, 187,  18,  82],
        [ 36,  94,  73, 107, 127, 205, 232, 139, 189, 180,  94, 192,
          64, 218, 138,  38, 249, 243,  36,  31,  41, 116,  81, 208,
         106, 176, 108,   2,  38,  86, 152, 185,  43, 145, 217, 109,
          93, 144, 250,  56,  83,  52, 248, 115, 129, 236, 102, 117,
         211, 200,   9,  11,  49, 120, 251, 101, 177,  39,  43,  25,
         217, 108, 121, 217, 225, 201, 124,  10,  80, 102, 161, 100,
```

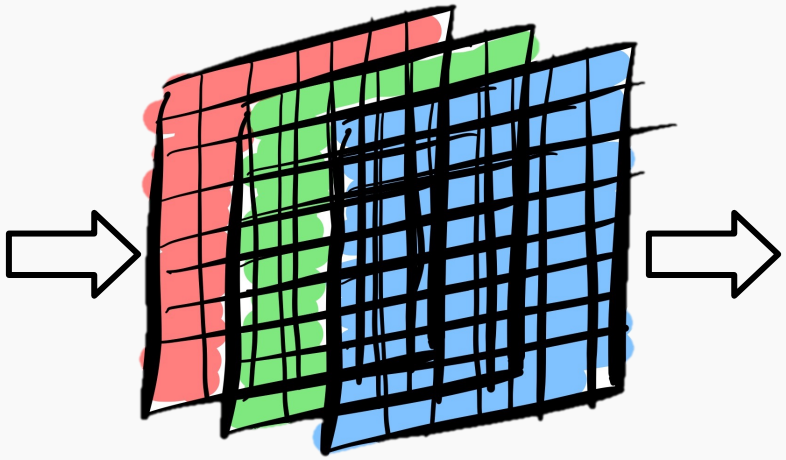


```
array([[[154, 187,  54,  33,  40,  57,  67, 196,  48,  77, 223,  36,
          70,  36, 173, 229, 131,  53,  15,  68,  39, 241, 210,   3,
           7, 135,  11,  35,  63,  19, 183,  65, 214, 194, 156, 177,
           5, 238, 241, 112, 183,  30, 234,  50, 201,  67,   1, 174,
           2, 190, 161, 210,  90,  44,  66,   1, 145, 176,  22, 232,
          57, 133, 234,  52, 195, 104,  18, 180,  65,  67, 244,  72,
         188, 104, 206, 186, 224,  95, 247, 145, 212, 126,  40,  54,
         170, 251, 131, 157,   9, 132, 221, 194, 205,  52,  97, 106,
         145,  64,  12, 244,  57, 110,   5, 126, 106, 121, 145,  31,
         185,  10,   5,  25,   8,  99,  17,  24, 228, 169, 112, 154,
          50, 207, 215, 211,  92,  25, 230,  40,  70, 213, 226, 218,
          72, 161,  32,  90,   7, 250,  79, 221, 127, 187, 175, 228,
           4,  56, 122, 206, 130, 104, 231, 155,  60, 144, 106, 242,
          21, 237, 217, 120, 137, 191,  87, 218, 207,  22,  11,   0,
          38,  28,  19,  21, 253, 205,  28, 181,  40, 236, 204, 233,
         244, 167,  10, 191,  28,   3,  53, 102, 157, 148, 143, 105,
         210, 195,  83, 101, 235, 204, 166,  58, 241, 170,  26, 162,
         110, 162,  43, 200,   5, 213,  80, 130, 237, 193, 241, 103,
          68,  17, 115,  15,  81, 171,  34,  33],
        [172, 181,  55, 176,  99, 100,   8, 105, 252,  17, 212, 233,
         214, 174,  91, 212,   5,  13,  18,  21, 183,  99,  78,  44,
         100, 175, 241, 135,  37, 135, 191, 215, 129, 138, 221,  25,
          90, 184,  69,  95,  58, 133, 217, 188,  65, 160, 180, 198,
          70,   1, 104, 136,  39, 197,  68, 220, 239,  26, 104,  96,
          79,  44,  97, 131, 214,  19, 240, 247,  36,  23, 205,  85,
```

# Computer Vision vs Language models

**IMAGE OF A CAT**

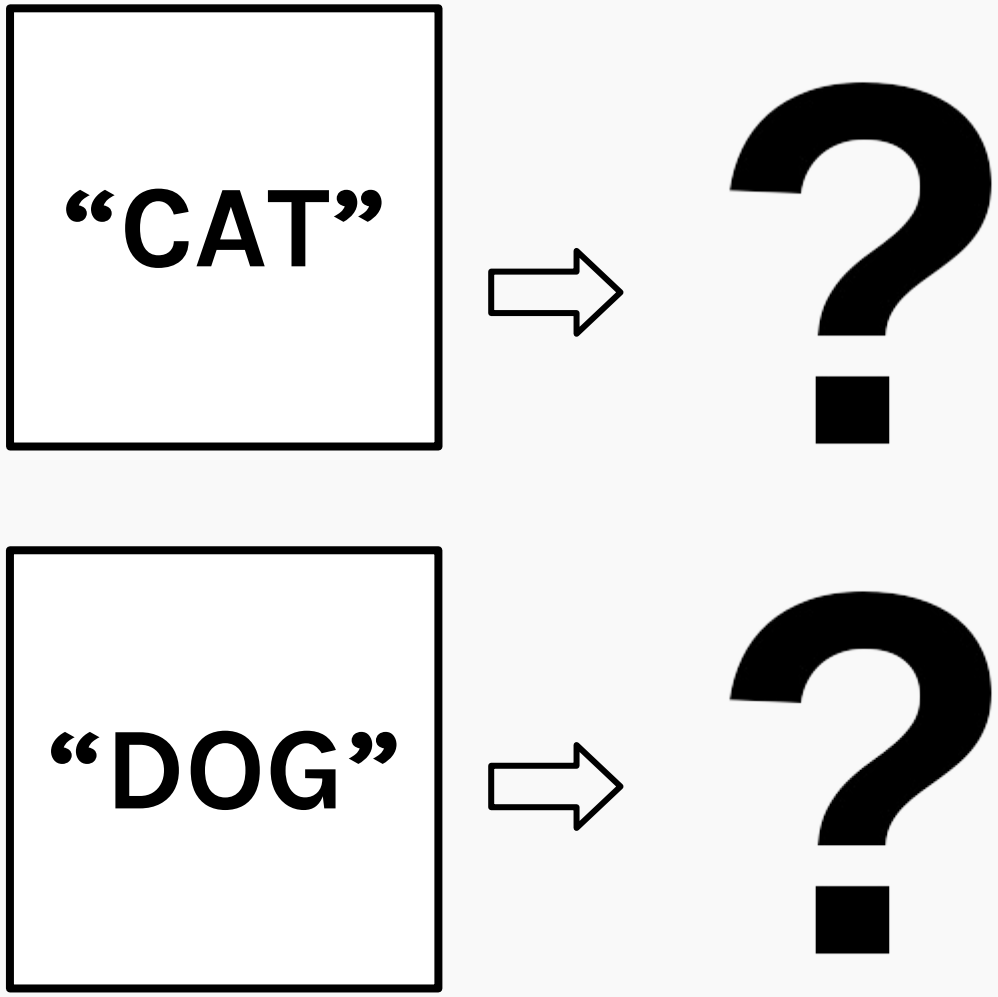
**RGB CHANNELS**



**3-D TENSOR**

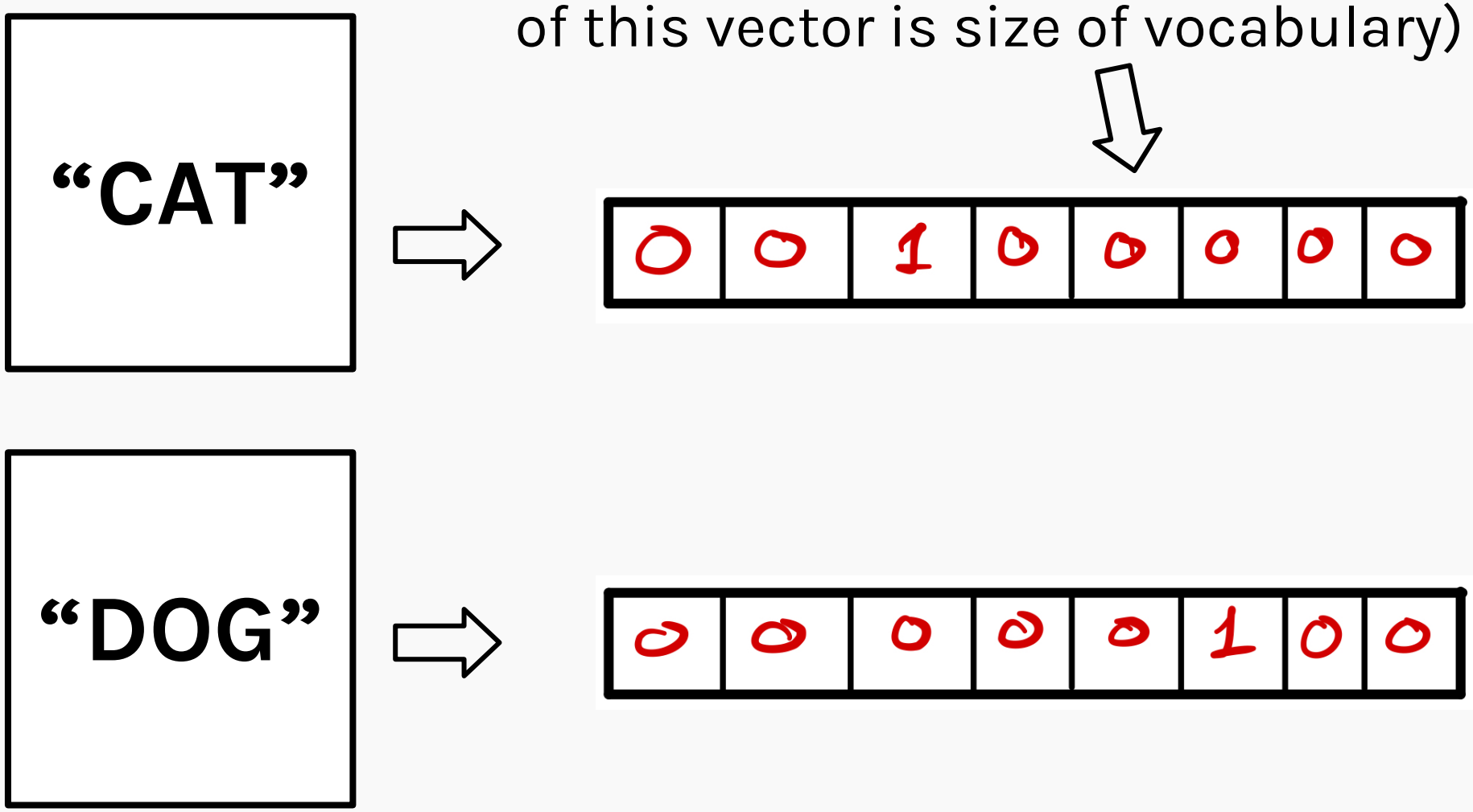
```
array([[[ 63,  95, 208, 231, 190, 186, 196, 208,  91, 110,  31,  66,
          163,  75,  96,  84,  86, 106,   0,  57,  90, 147,  31, 148,
           63,  10,  61, 231,  80, 127, 174, 240,  45,  56, 153, 193,
          238,  85,  38, 118,  94, 145, 144, 192,  82,  48, 145,  17,
          176, 150,  43, 111,  55, 104, 249,  42,  25, 145,  88, 102,
          115,  12,  91, 238, 235, 198, 141,  96, 118, 167, 222, 123,
           43, 235,  10, 205,  68, 249, 208,  71, 112,  94, 146,   1,
           16, 134,   7,  94,  28, 148,  77,  58, 122, 232,  59, 194,
           16,   7, 204, 120, 174, 192,  72, 197,  95,  18, 188, 243,
           83, 201, 137, 250, 237, 149, 222, 150,  96,   2, 191,  15,
          164, 167, 147,  70, 193,  30,  33, 108,  65,  77, 192, 237,
          189,  47, 166, 116, 141, 107,  39, 158, 241,  33, 170,  37,
          102, 173,  82, 181, 229, 189, 161,  18, 251, 187,  95,  28,
           45, 132,  66, 201, 166, 243,  18, 135,   9, 171, 110,  74,
           99, 207, 185, 228, 243,  17, 236,  60,  61, 142,  12,  54,
          203,  33,  30,  97, 202, 231,  34, 137, 162, 172, 100, 233,
          116,  58, 172,  35,  91, 100,   2,   6, 221,  86, 176,  15,
           15, 117,  57,  54, 108, 158, 145, 155,  89, 116,  73,  28,
          169,  33, 175, 120, 129, 187,  18,  82],
         [ 36,  94,  73, 107, 127, 205, 232, 139, 189, 180,  94, 192,
           64, 218, 138,  38, 249, 243,  36,  31,  41, 116,  81, 208,
          106, 176, 108,   2,  38,  86, 152, 185,  43, 145, 217, 109,
           93, 144, 250,  56,  83,  52, 248, 115, 129, 236, 102, 117,
          211, 200,   9,  11,  49, 120, 251, 101, 177,  39,  43,  25,
          217, 108, 121, 217, 225, 201, 124,  10,  80, 102, 161, 100,
```

# Computer Vision vs Language models



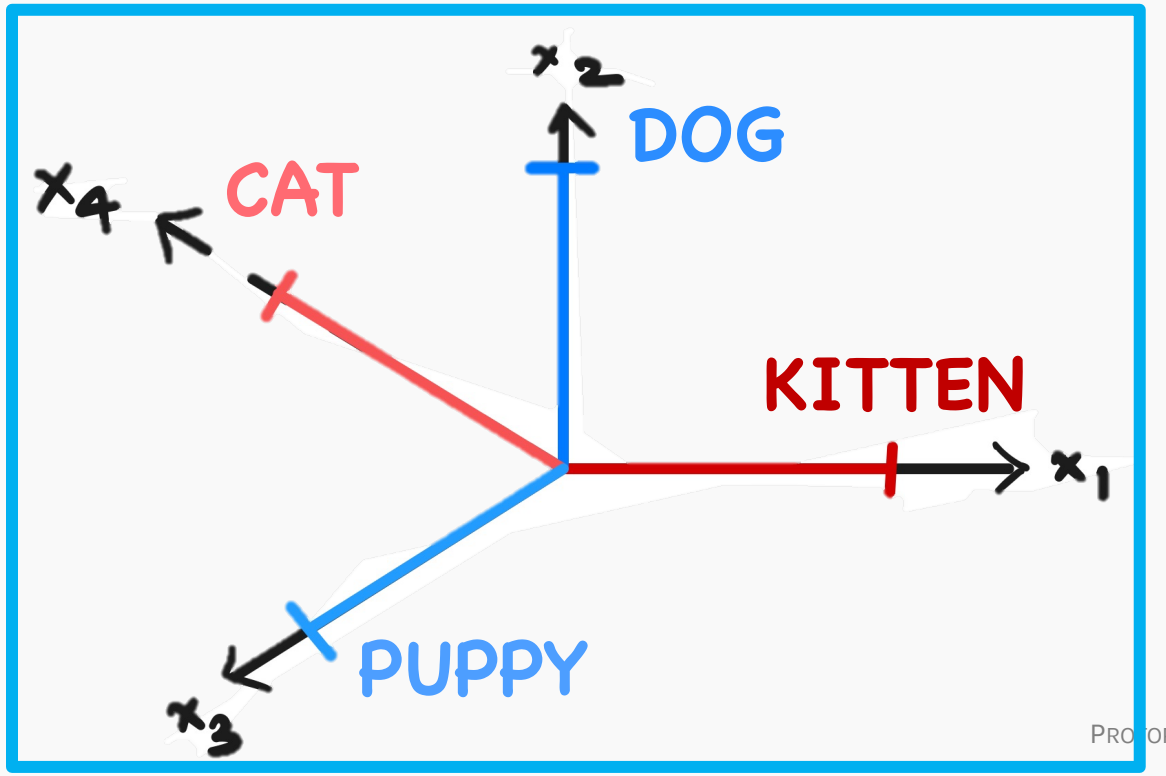

# Words to numbers – One Hot Encoding

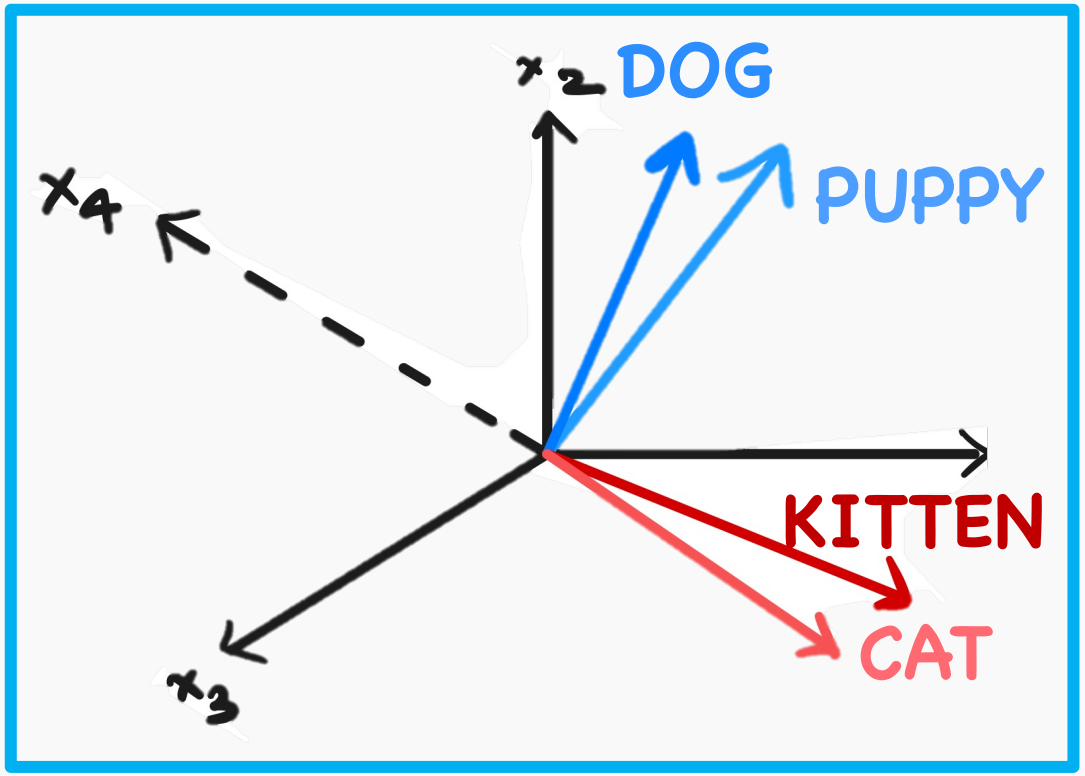
One-hot encoding of the word “cat” (length of this vector is size of vocabulary)



# One Hot Encoding Issues

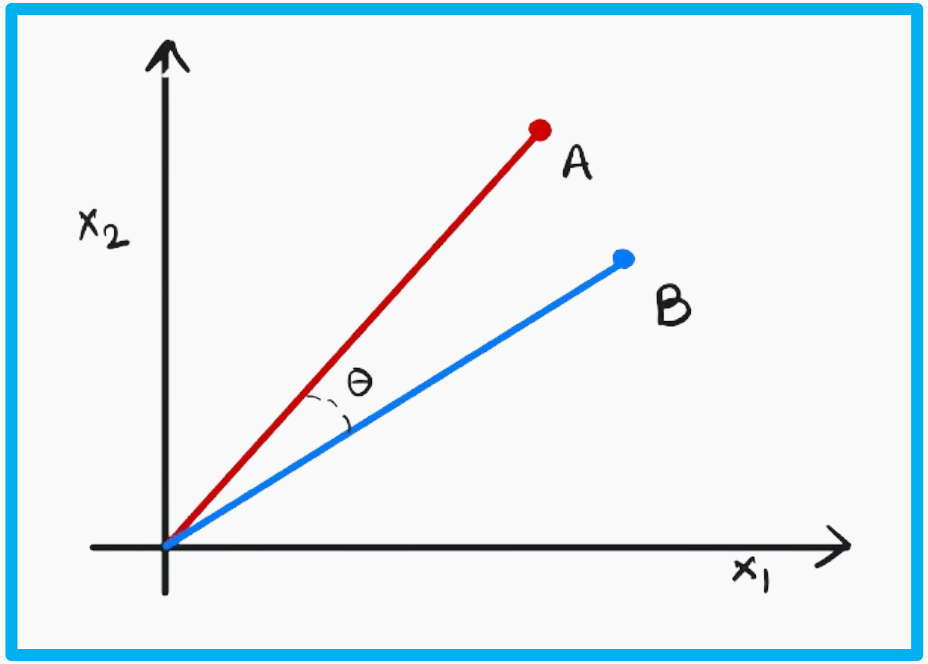
- The vocabulary $\boldsymbol{V}$ of a corpus (large swath of text) can have 10,000 words.
- One-hot encoding of such a corpus is huge.
- Moreover, similarities between words cannot  be established.




# SIMILARITY

What is “Cosine Similarity”?




$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

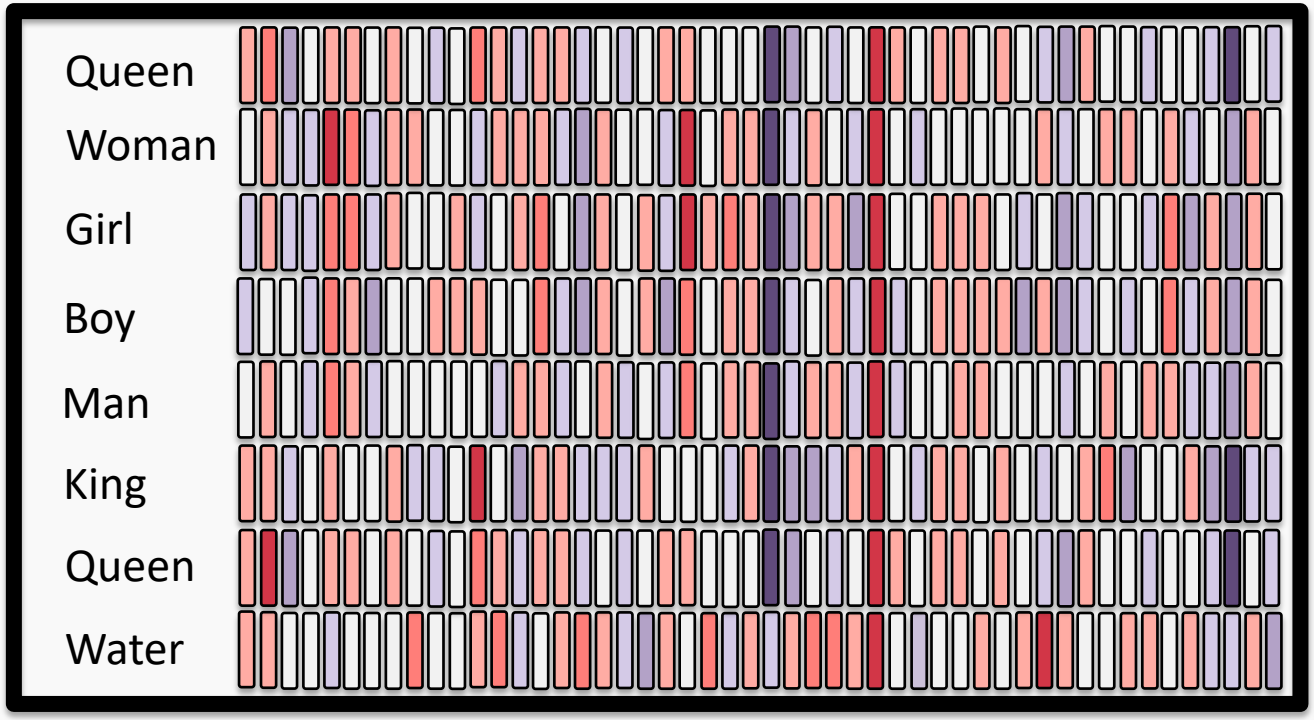
Where $A_i$ & $B_i$ are components of vector $A$ & $B$ respectively

# Word Embeddings

- The vector representation of a word is called an **Embedding**.
- Two words will be **"similar"** if their vector representations are *close* to each other.
- An **Embedding Matrix** is simply a collection of embedding values for all words in the vocabulary.

# Obligatory example

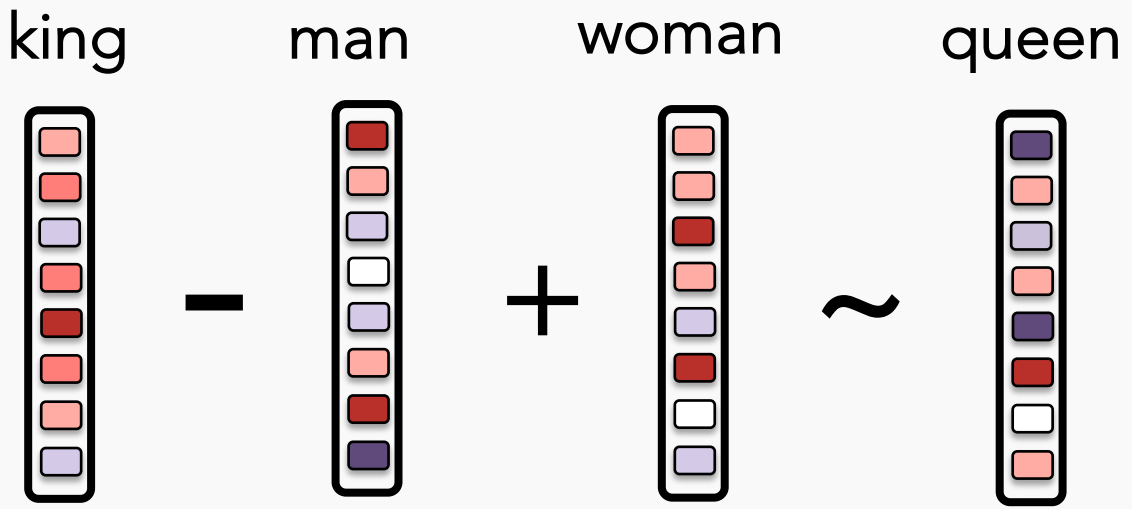
**Embedding Matrix**




Vector Representations for a few words

(Color gradient indicates values from embedding)

Since these **words** are now mapped to **numbers** in $R^n$, we can operate on them

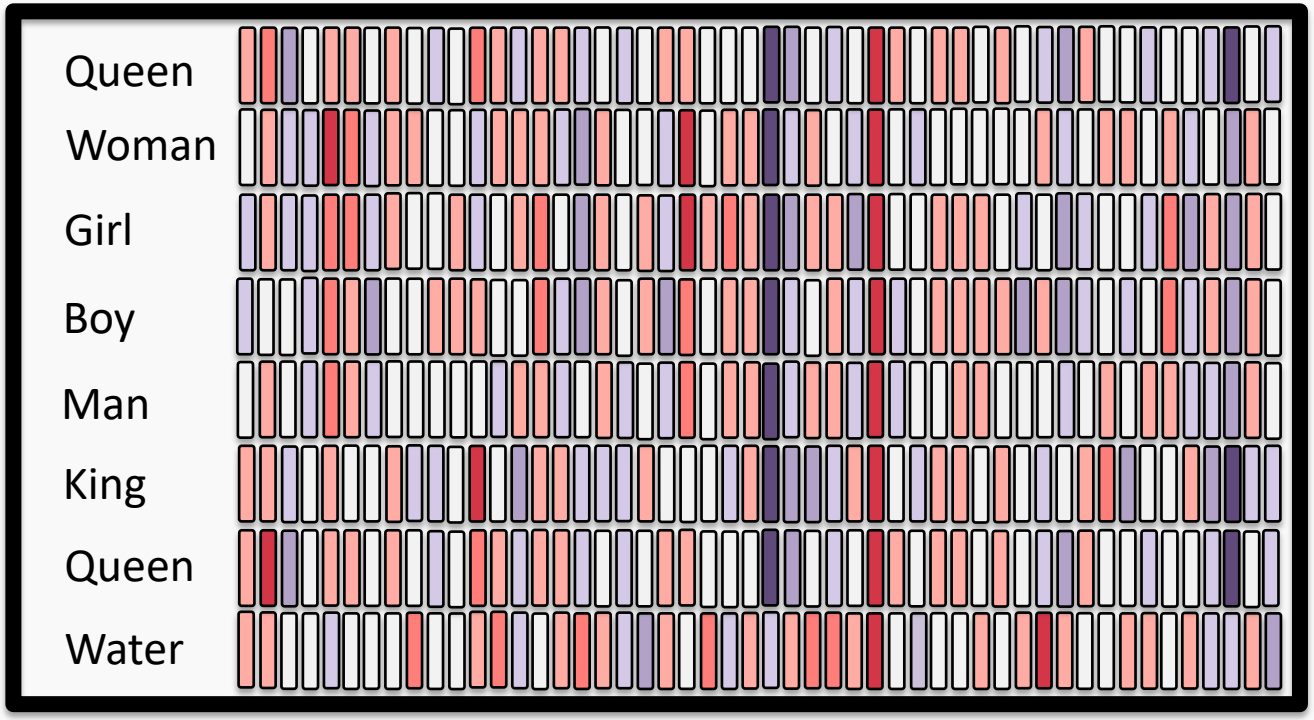


# What we want

**Embeddings Wishlist?**

- We want the words of our vocabulary to be represented by a low-dimensional vector space.
- We also want these vector representations to have some semantic meaning, i.e vector representations of similar words must be close to each other.

# So how do we get such a rich word “embedding?”




💡 **IDEA:** We could use a language model!

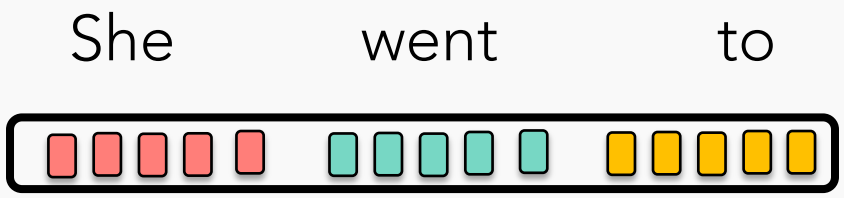
# **RECAP**: Language Modelling: **neural networks**

Language modeling is about predicting the next word using the previous words

$$P(x_{t+1}|x_t, x_{t-1}, \dots, x_1)$$

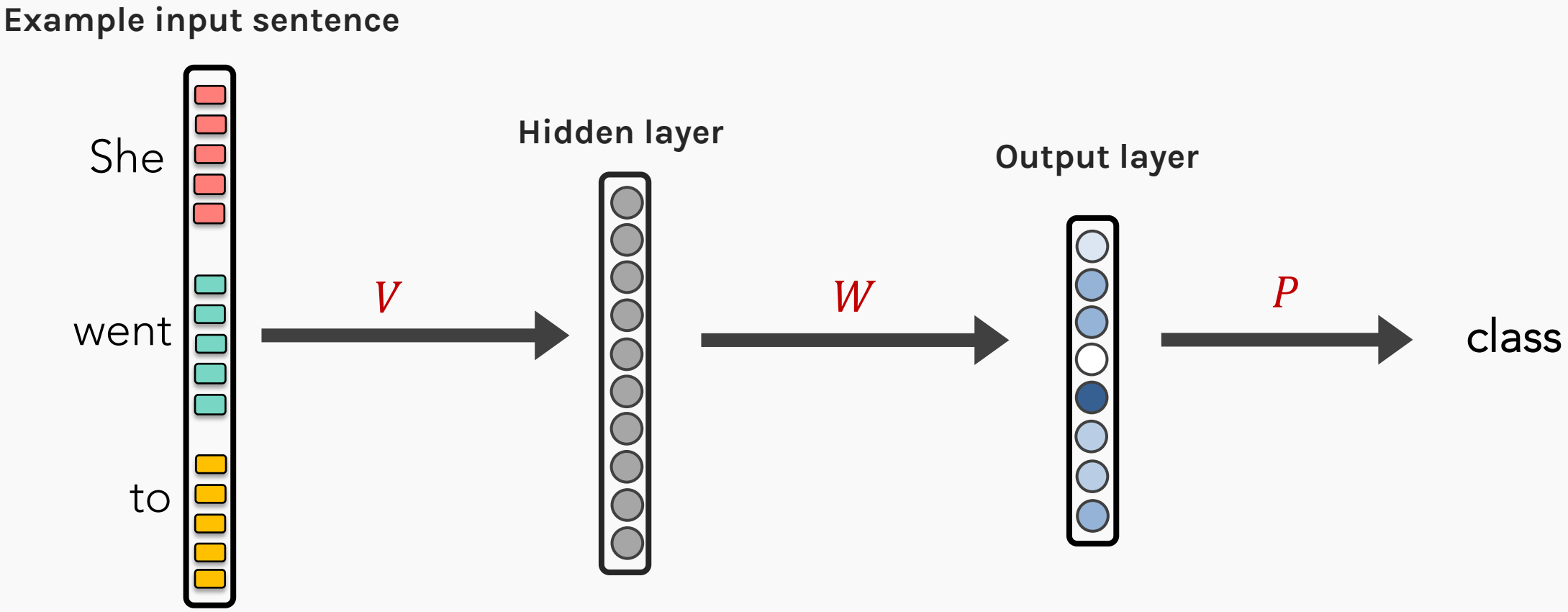
next word — previous words

**Example input sentence**




# **RECAP**: Language Modelling: **Feed-forward Neural Net**

**General Idea:** using *windows* of words, predict the next word



# Word Embeddings Training

- Text is a semantic sequence of words i.e., words used in a sentence are not random.
- We assume that If we build a neural network for language models and train them sufficiently well, we could get an embedding of words which can have a semantic relationship.
- We expect that two words that are similar will be mapped closely in the embedding space.

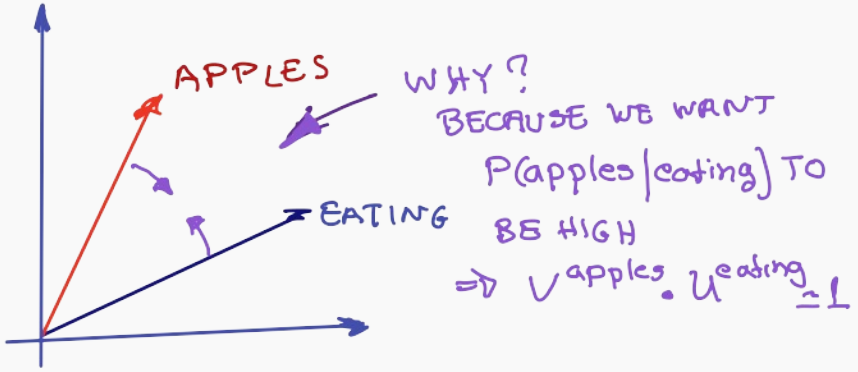
**Example**:

SENTENCE #1: Pavlos ate an apple before the lecture.

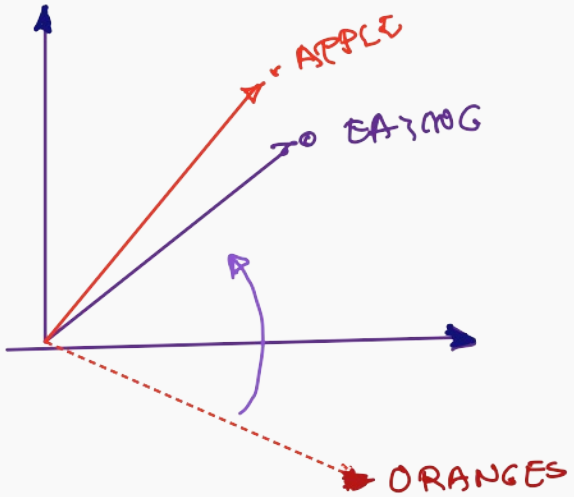
SENTENCE #2: Eleni ate an orange before the session.

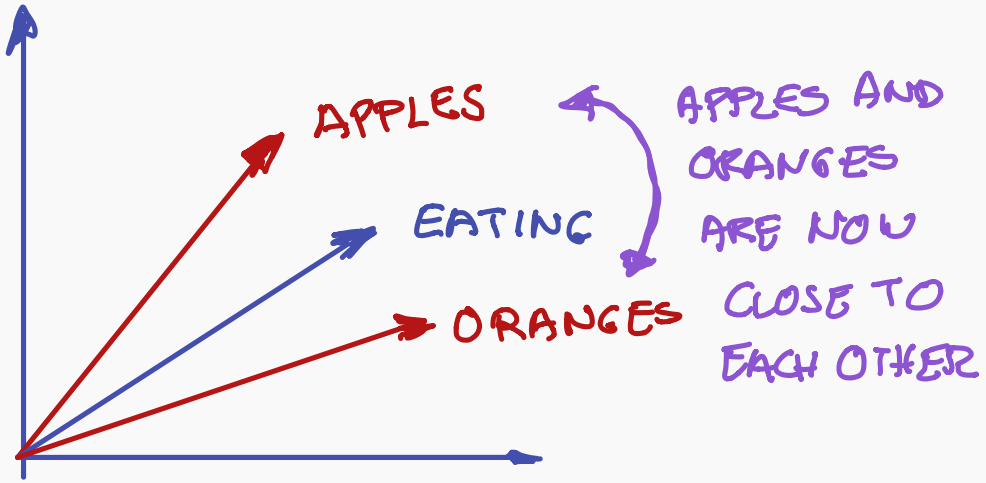
Both apple & orange are surrounded by similar words.

I like eating apples before dinner. I also like eating oranges after dinner.




I like eating apples before dinner. I also like eating oranges after dinner.



# Word Embeddings Training

- Text is a semantic sequence of words i.e., words used in a sentence are not random.
- We assume that If we build a neural network for language models and train them sufficiently well, we could get an embedding of words which can have a semantic relationship.
- We expect that two words that are similar will be mapped closely in the embedding space.

**Example**:

SENTENCE #1: Pavlos ate an apple before the lecture.

SENTENCE #2: Eleni ate an orange before the session.

Both apple & orange are surrounded by similar words.

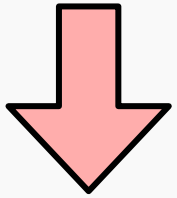
# Word Embeddings Training

Training Set

Model/Neural Network

Loss Function

# Word Embeddings Training



Training Set

Model/Neural Network

Loss Function

# Training set

Training Set

- To build a language model training set, we need to select a sequence of some words as input and use the next immediate word as the output label
- We can use a sliding window to create several such training examples
- There are other approaches to building a language model training set, but more on that later

# Example sentence: Guess the next word

The dog was chased by a ___

# How do we set up a training set?

The dog was chased by a cat as it was crossing the street …

Sliding window across running text

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |

Dataset

| input 1 | input 2 | output |
|---|---|---|
| was | chased | by |
| | | |
| | | |
| | | |
| | | |

# How do we set up a training set?

The dog was chased by a cat as he was crossing the street …

Sliding window across running text

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |

Dataset

| input 1 | input 2 | output |
|---|---|---|
| was | chased | by |
| chased | by | a |
| | | |
| | | |
| | | |

# Continuous Bags of Words (CBOW)

The dog was chased by a cat as he was crossing the street …

Sliding window across running text

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |
| dog | was | chased | by | a | cat | as | … |

Dataset

| input 1 | input 2 | output |
|---|---|---|
| was | chased | by |
| chased | by | a |
| by | a | cat |
| | | |
| | | |

**NOTE**: This approach of building training samples is called **Continuous Bags of Words** (CBOW)

# New sentence: Guess the word in the blank

The dog was chased by a ___

# New sentence: Guess the word in the blank

If we go from left to right,

the most likely word is **cat**

The dog was chased by a ___

# New sentence: Guess the word in the blank

However, if we see the complete sentence, the most likely word now is **RED or WHITE or BLACK**

The dog was chased by a ___ cat

# New sentence: Guess the word in the blank

**Why not look both ways?**

The dog was chased by a ___ cat

# New sentence: Guess the word in the blank

**What about inverting the problem:**
**predict context words instead of middle words**

The dog was chased by a ___ cat

This leads to the **Skip-Gram** architecture

# SKIP-GRAM: Predict Surrounding Words

Choose a window size (here 4) and construct a dataset by sliding a window across.

dog was chased by a red cat as he was crossing the street …

| dog | was | chased | by | a | red | cat | as | he | was | … |
|---|---|---|---|---|---|---|---|---|---|---|

| input word | target word |
|---|---|
| a | chased |
| a | by |
| a | red |
| a | cat |

# SKIP-GRAM: Predict Surrounding Words

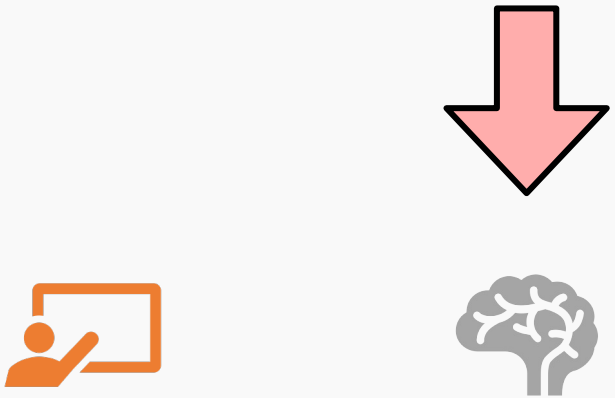
Choose a window size (here 4) and construct a dataset by sliding a window across.

dog was chased by a red cat as he was crossing the street …

| dog | was | chased | by | a | red | cat | as | he | was | … |
|---|---|---|---|---|---|---|---|---|---|---|

| input word | target word |
|---|---|
| a | chased |
| a | by |
| a | red |
| a | cat |
| red | by |
| red | a |
| red | cat |
| red | as |

# Word Embeddings Model

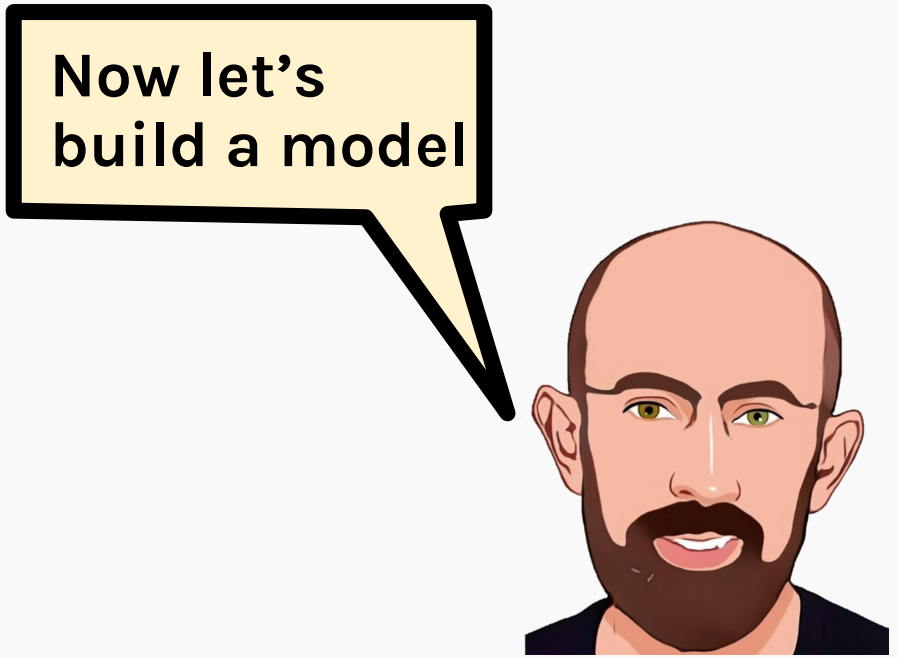


Training Set



Model/Neural Network

Loss Function



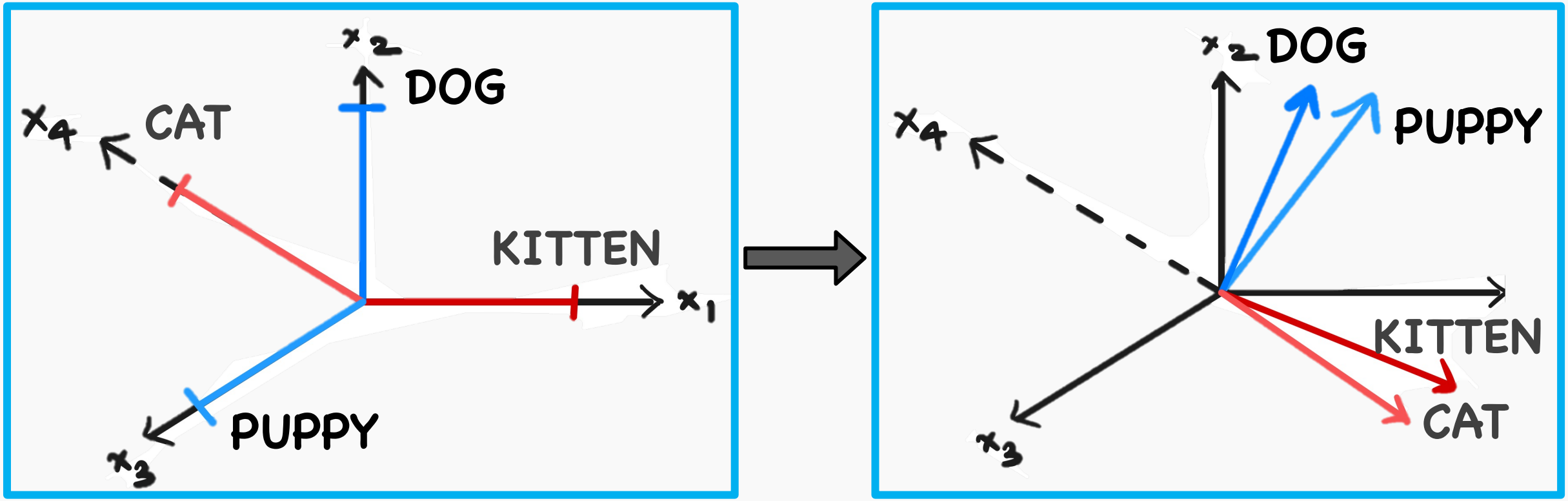
# Model

Model/Neural Network

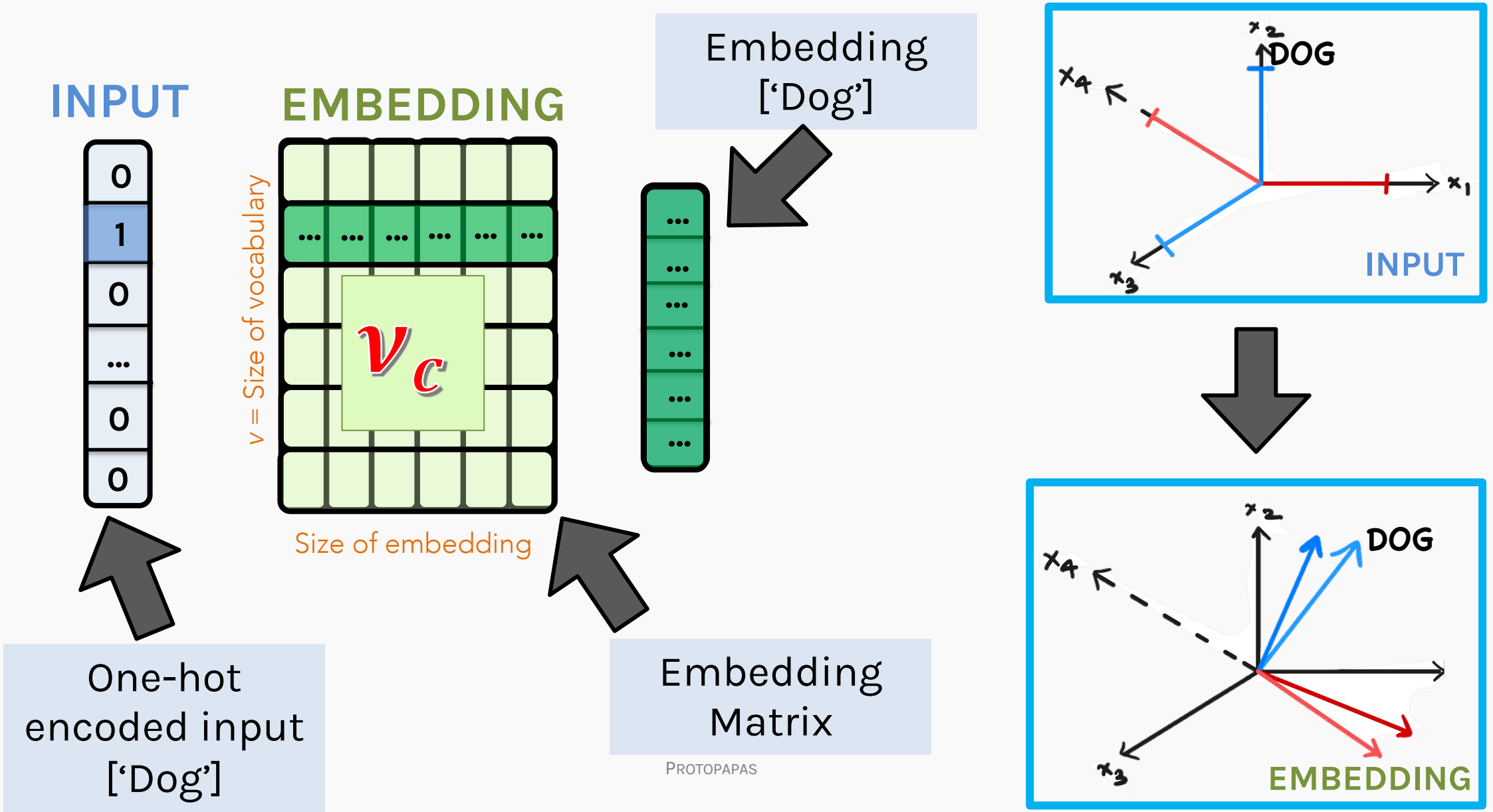
- To build a language model we need a network that takes a one-hot encoded input, connected to a low dimensional hidden state of size N, and an output with the same size as the input.
- We can then map the output (logits) to probabilities by using the softmax function
- In principle, the hidden state will be the embedding of the word

# Going from One-Hot encoded to Embedding

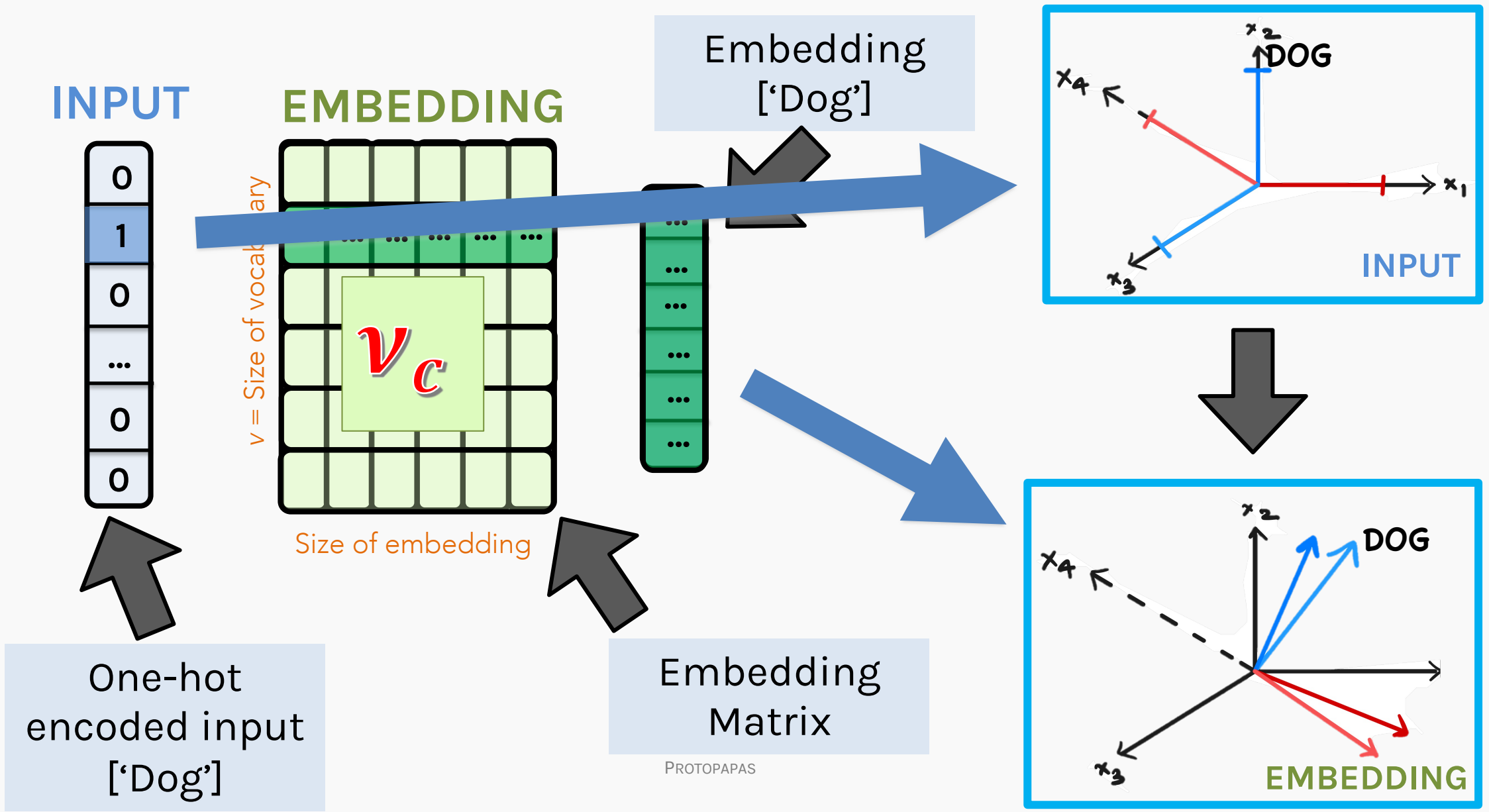
How do we go from one-hot encoding to embedding space?

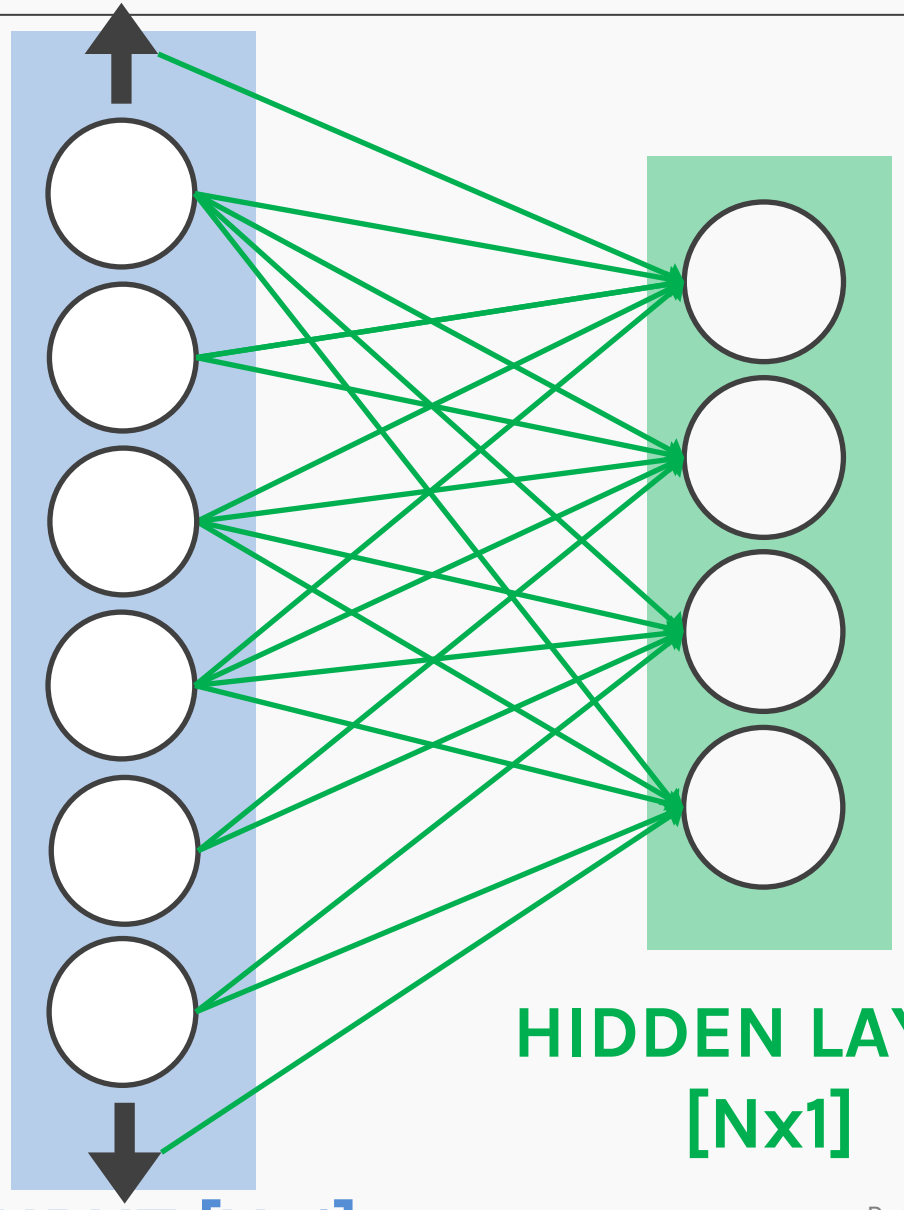


# Going from One-Hot encoded to Embedding



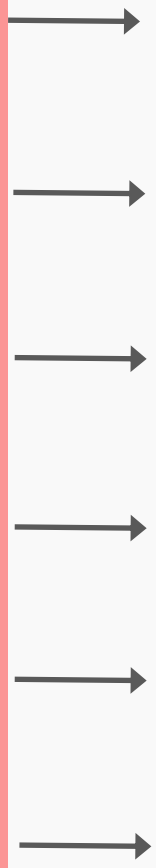
# Going from One-Hot encoded to Embedding





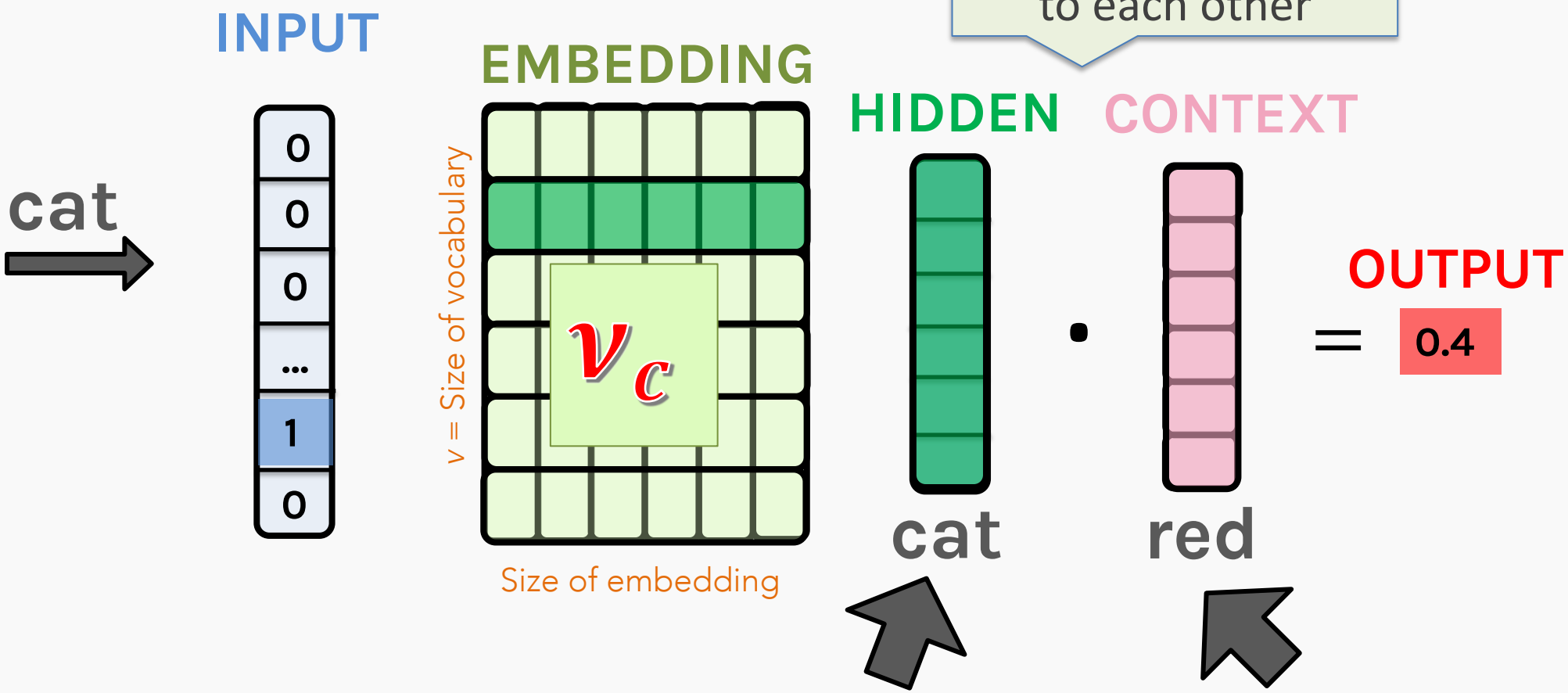

## WISHLIST

We want to go from hidden state of center word, $w_c$, to probabilities $P(w_o|\, w_c)$ for each word, $w_o$, in the vocabulary



# Skipgram Language model

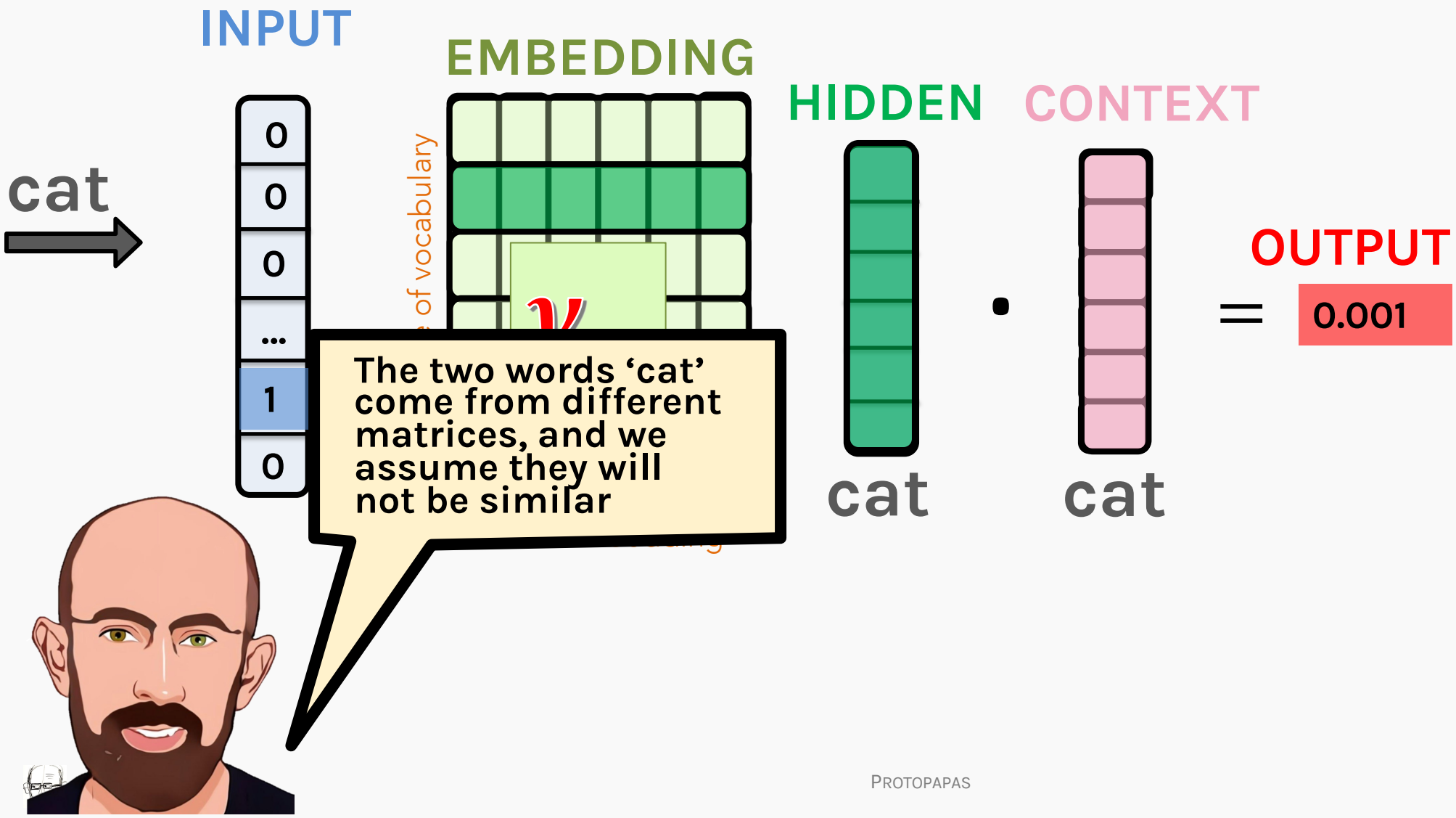
Cosine similarity describes how close the two vectors are to each other



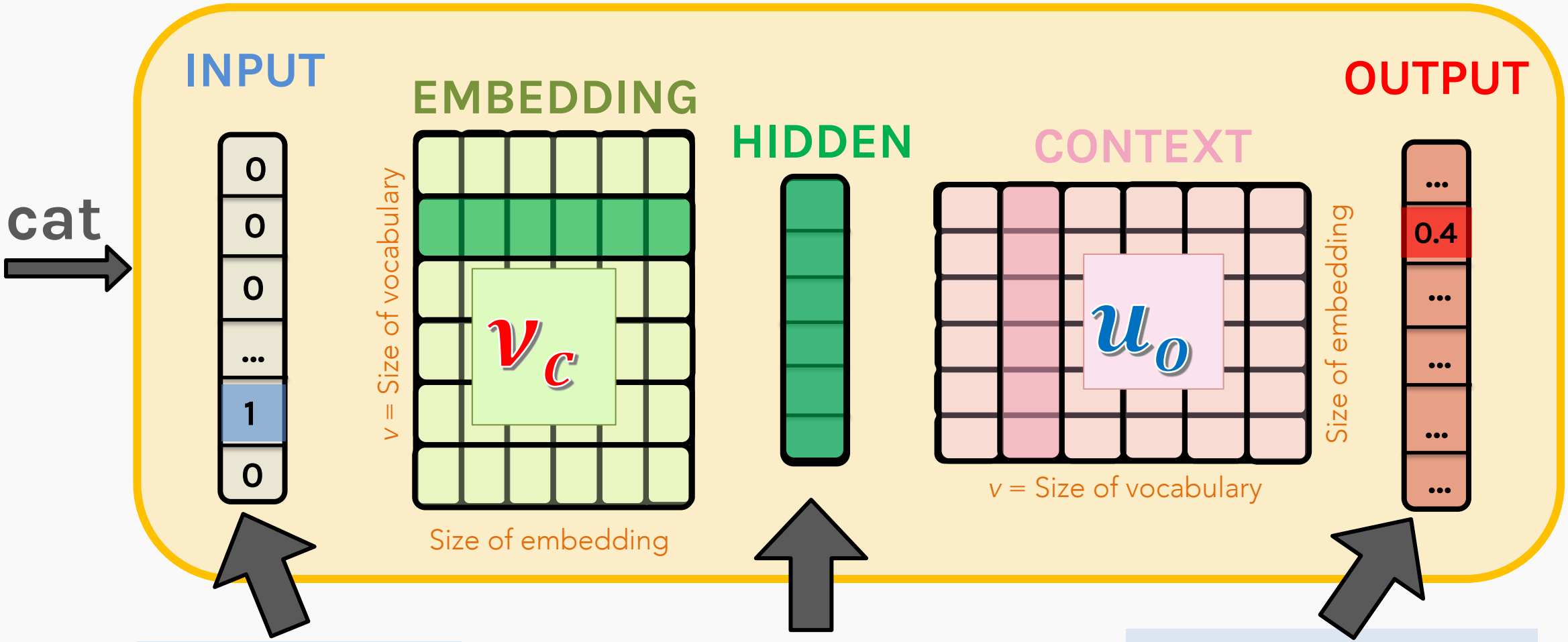

Vector for ['cat'] from embedding matrix

Vector for ['RED'] from a 'different' context matrix

# Skipgram Language model



# Skipgram Language model

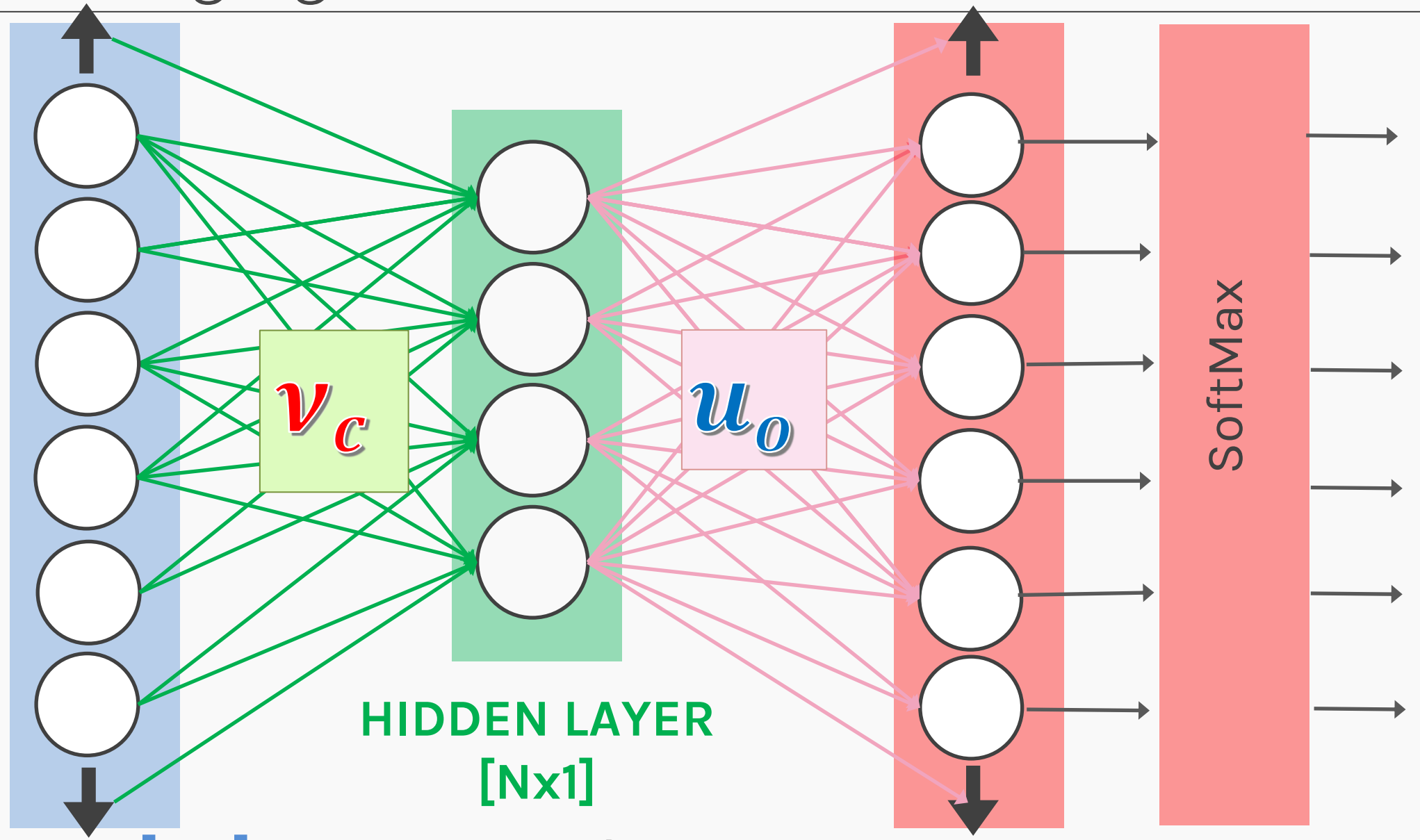



One-hot encoded input [Vx1]

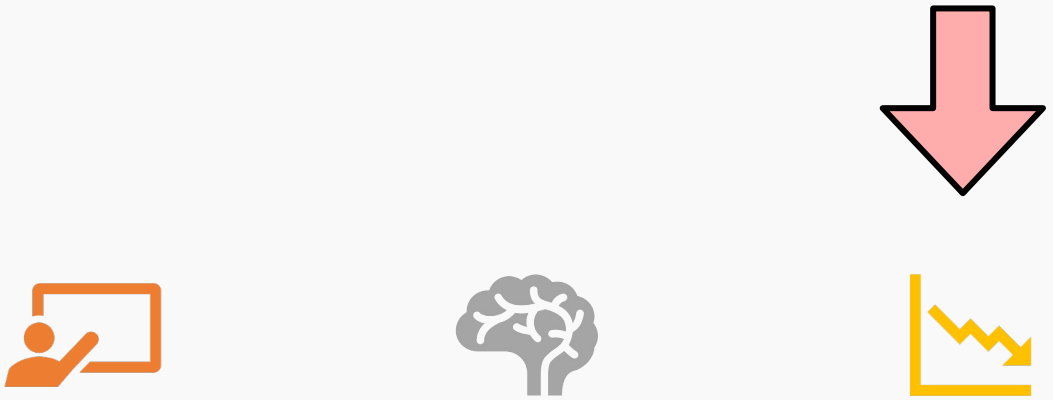
Hidden layer [Nx1]

Output layer $p(w_{oi}|w_c)$ [VX1]

# Skipgram language model – Neural net edition




# Loss Function



Training Set

Model/Neural Network

Loss Function

# Training set

Loss Function

- Once we have the output probabilities, we can select the context words for each input, and multiply the probabilities to construct the likelihood
- We need to maximize this probability (likelihood)
- The loss of choice is the Negative Log Likelihood, which must be minimized – this is equivalent to maximize the likelihood of the context words given central words

# SKIP-GRAM: Details

We assume that Naive Bayes style, the joint probability of all **context** words ($w_o$) in a window conditioned on the central word ($w_c$), is the product of the individual conditional probabilities:

$$P(\{w_o\}|w_c) = \prod_{i \in window} \mathbb{P}(w_{oi}|w_c)$$

$\{w_o\}$ = words in the window of the central word: context words

$w_c$ = central word

product of the probabilities of each word in the window, given the central word

# Loss function for gradient descent

Then, assuming a text sequence of length $T$ and window size $m$, the likelihood function is:

$$\prod_{t=1}^{T} \prod_{-m \le j \le m, j \neq 0} \mathbb{P}\left(w^{(t+j)} \middle| w^{t}\right)$$

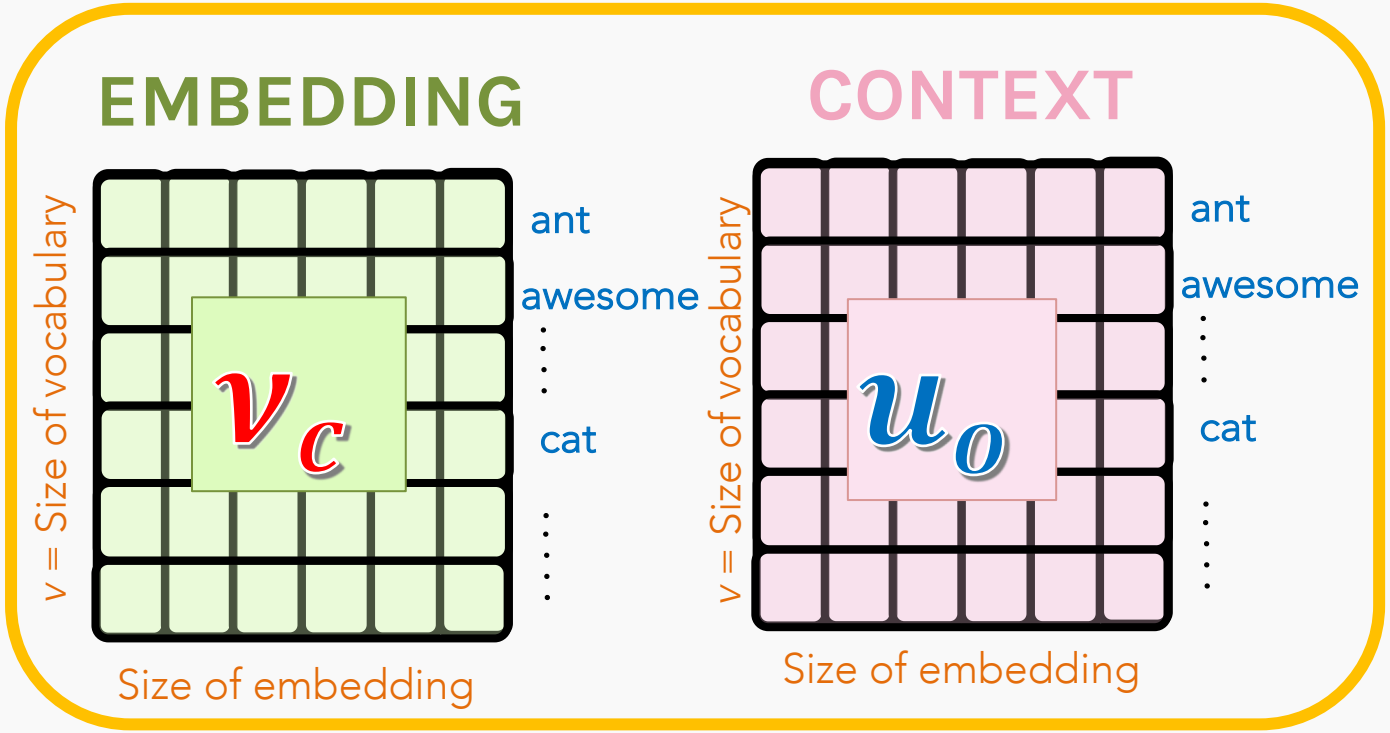
| input word | target word |
|---|---|
| a | chased |
| a | by |
| a | red |
| a | cat |
| red | by |
| red | a |
| red | cat |
| red | as |

We want to maximize this likelihood hence we will minimize the Negative Log likelihood and use it as our loss function

$$\mathcal{L} = -\sum_{t=1}^{T} \sum_{-m \le j \le m, j \neq 0} \log \mathbb{P}\left(w^{(t+j)} \mid w^{(t)}\right)$$

# Alternatively…

## Look-up table approach



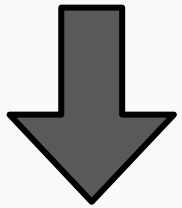

Now assume that each word is represented as 2 embeddings, an **input** embedding, $v_c$, (c is for central) when we talk about the central word and a context embedding ($u_o$) when we talk about the surrounding window (o is for output).

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(u_o^\top v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c)},$$

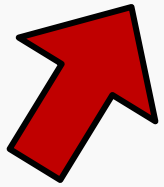
The probability of an output word, given a central word, is assumed to be given by a softmax of the dot product of the embeddings.

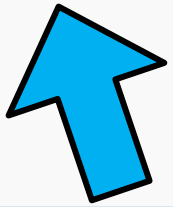
# Loss function for gradient descent

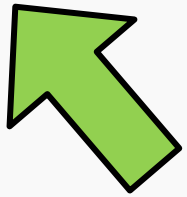
$$\mathcal{L} = -\sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log \mathbb{P}\left(w^{(t+j)} \mid w^{(t)}\right)$$



$$\mathcal{L} = -\sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log \frac{\exp(u_o^\intercal v_c)}{\sum_{i \in V} \exp(u_i^\intercal v_c)}$$







Sum over all the central words in the training

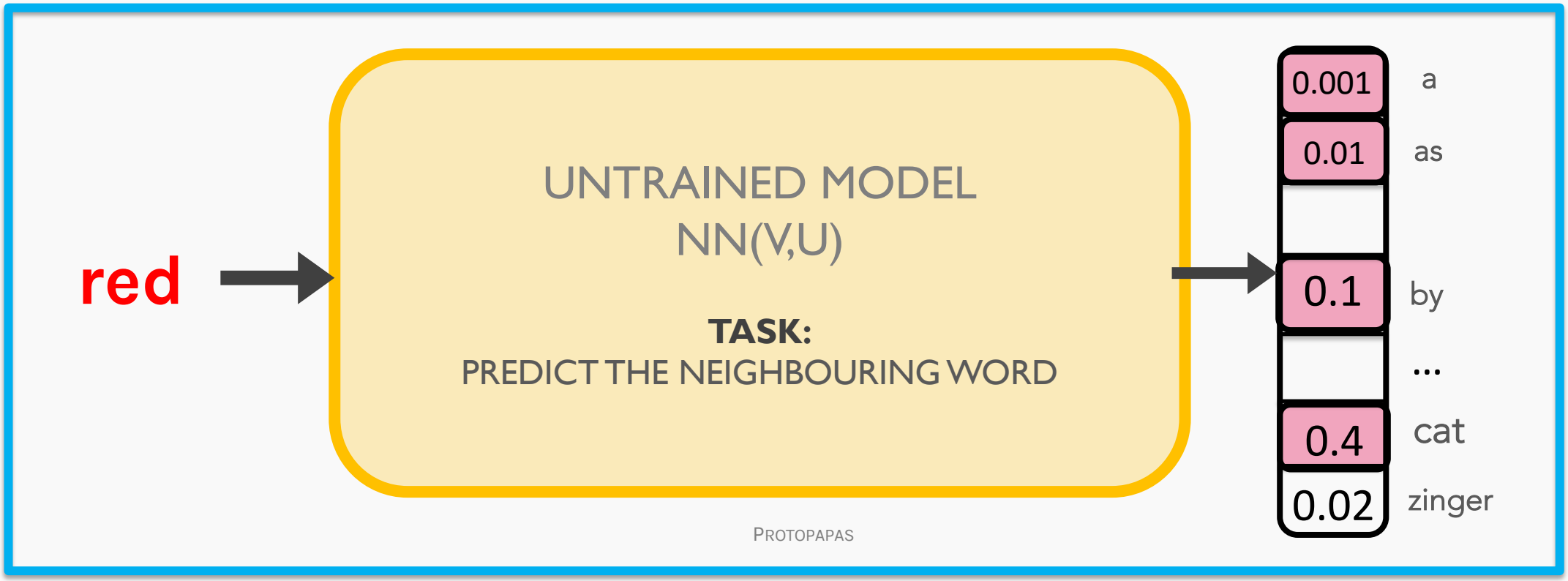
Sum over all the words in the window

Softmax over the dot product with every possible word in the vocabulary

# Putting it all together...

1. Look up embeddings
2. Calculate predictions
3. Project to outward vocabulary

With random initial weights , we make a prediction for surrounding words, and calculate the NLL for the prediction. We then backpropagate the NLL's gradients to find new weights and repeat

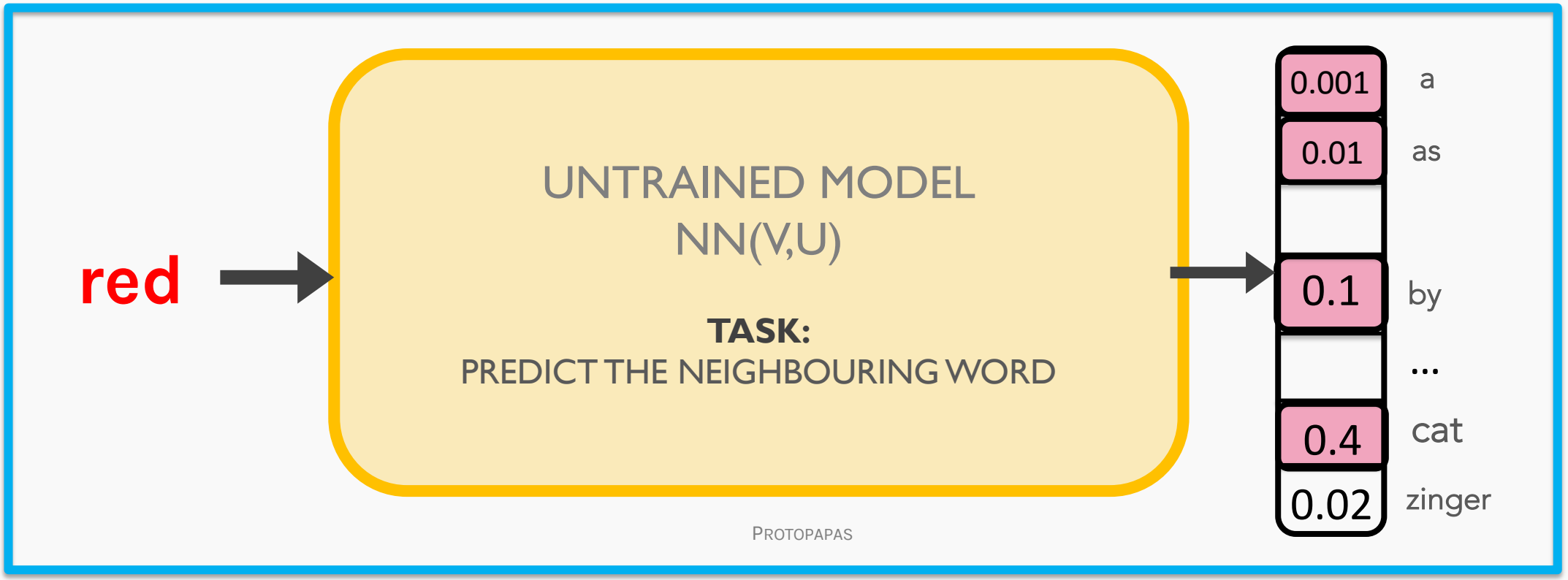


# Putting it all together…

1. Look up embeddings
2. Calculate predictions
3. Project to outward vocabulary

What weights are you talking about?

andom initial weights , we make a
pr iction for surrounding words, and
calcu ate the NLL for the prediction. We then
backpropagate the NLL's gradients to find
new weights and repeat



DONE!?



# Problems with implementation

- In the forward mode, the calculation of softmax requires a sum over the entire vocabulary

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)},$$

# Problems with implementation

- In the forward mode, the calculation of softmax requires a sum over the entire vocabulary

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)},$$

# Problems with implementation

- In the backward mode, the gradients need this sum too. For example:

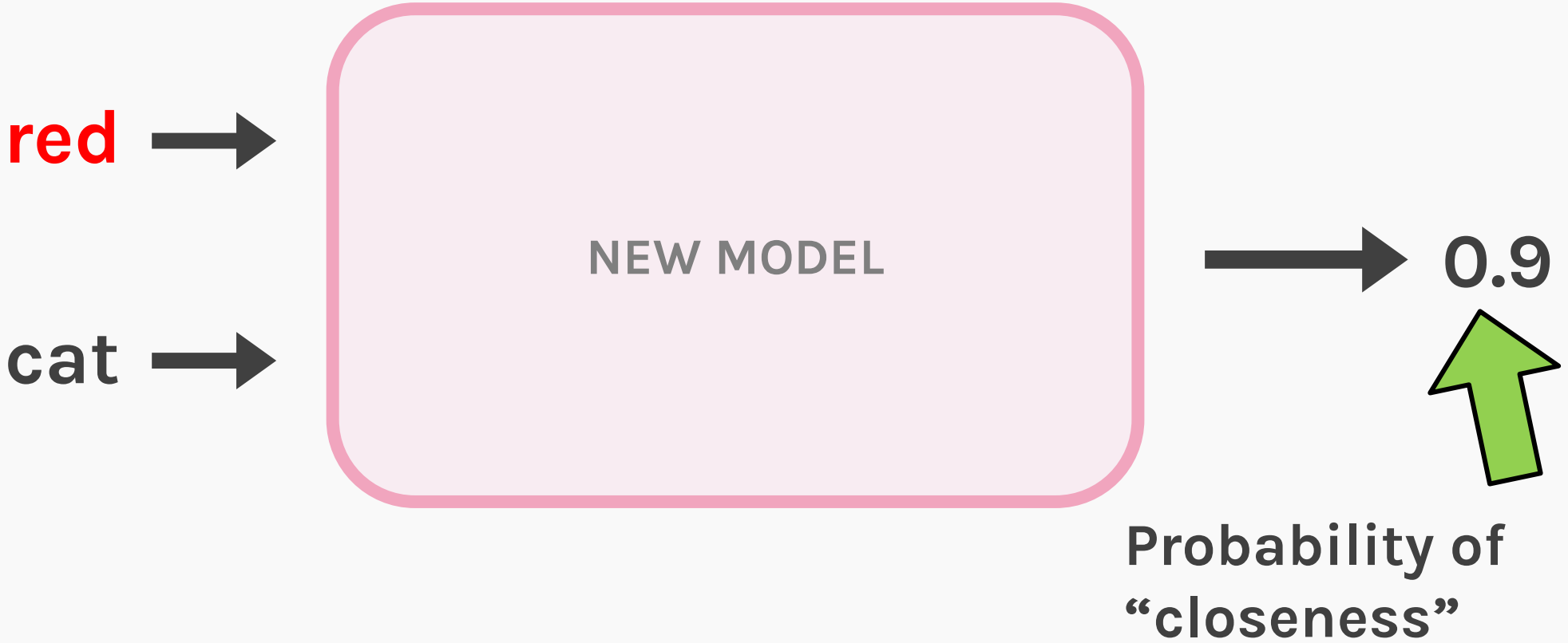
$$\frac{\partial \log P(w_o \mid w_c)}{\partial \boldsymbol{v}_c} = \boldsymbol{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) \boldsymbol{u}_j.$$

For large vocabularies, this is very expensive!

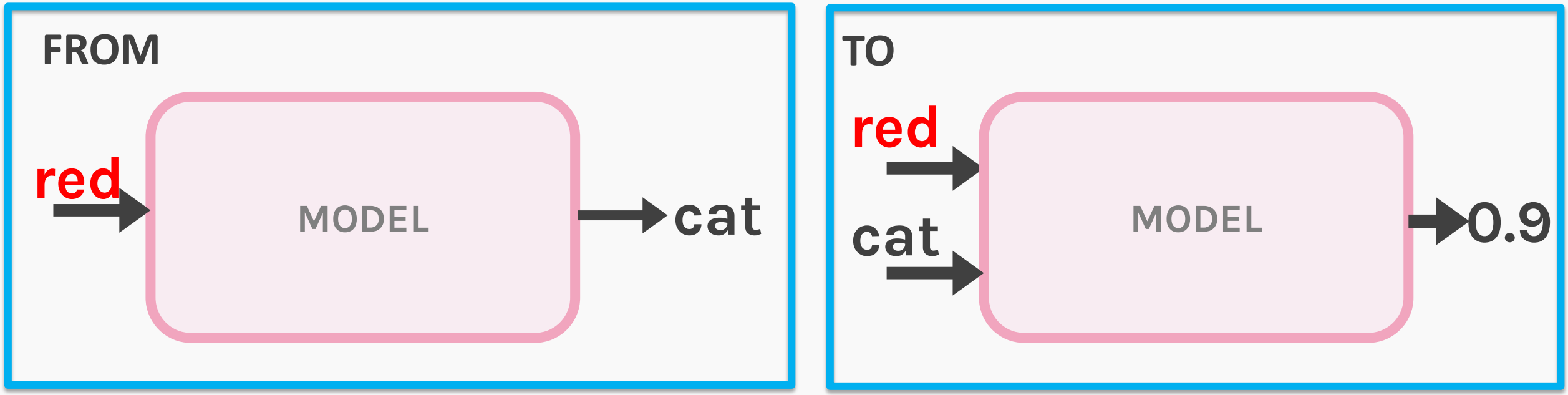
# Changing Tasks

# Changing Tasks

TO:



# Changing Tasks




Changing from predicting neighbors to "*are we neighbors?*" changes model from neural net to logistic regression.

# Changing Tasks (cont)

We'll now choose $P(D = 1|w_c, w_o) = \sigma(u_o^T v_c)$ and will maximize the likelihood:

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P\left(D = 1 \mid w^{(t)}, w^{(t+j)}\right)$$

But the response variable in the dataset changes to all 1's, and a trivial classifier always returning 1 will give the best score.

Not good (this is equivalent to all embeddings being equal and infinite)!

# Changing Tasks (cont)

We'll now choose $P(D = 1|w_c, w_o) = \sigma(u_o^T v_c)$ and will maximize the likelihood:

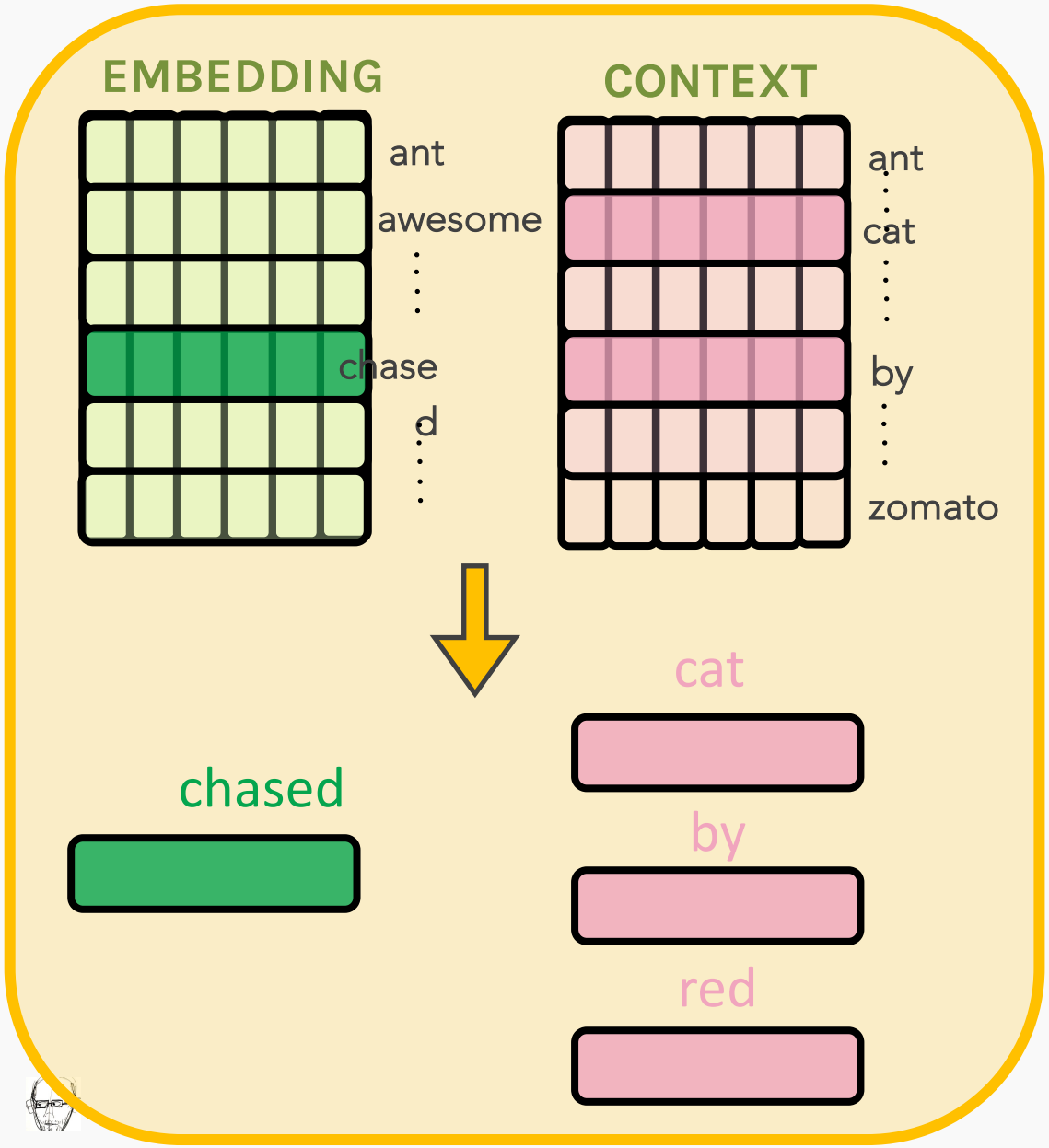
$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P\left(D = 1 \mid w^{(t)}, w^{(t+j)}\right)$$

Why infinite?
$\sigma(x) = 1$
$x \to \infty$

But the response variable in the dataset changes to all 1's, and a trivial classifier always returning 1 will give the best score.

Not good (this is equivalent to all embeddings being equal and infinite)!

# Training the model




- The positive sampling probabilities are simply sigmoids.
- We now compute the loss, and repeat over training examples in our batch, and backpropagate to obtain gradients and change the embeddings and weights some, for each batch, in each epoch

| Input word1 | Input word2 | Target | Input1 · Input2 | sigmoid() |
|---|---|---|---|---|
| chased | cat | 1 | -1.11 | 0.25 |
| chased | by | 1 | 0.2 | 0.55 |
| chased | red | 1 | 0.74 | 0.68 |

# Negative Sampling (change)

we need to introduce negative samples to our dataset – samples of words that are not neighbours. Our model needs to return 0 for those samples

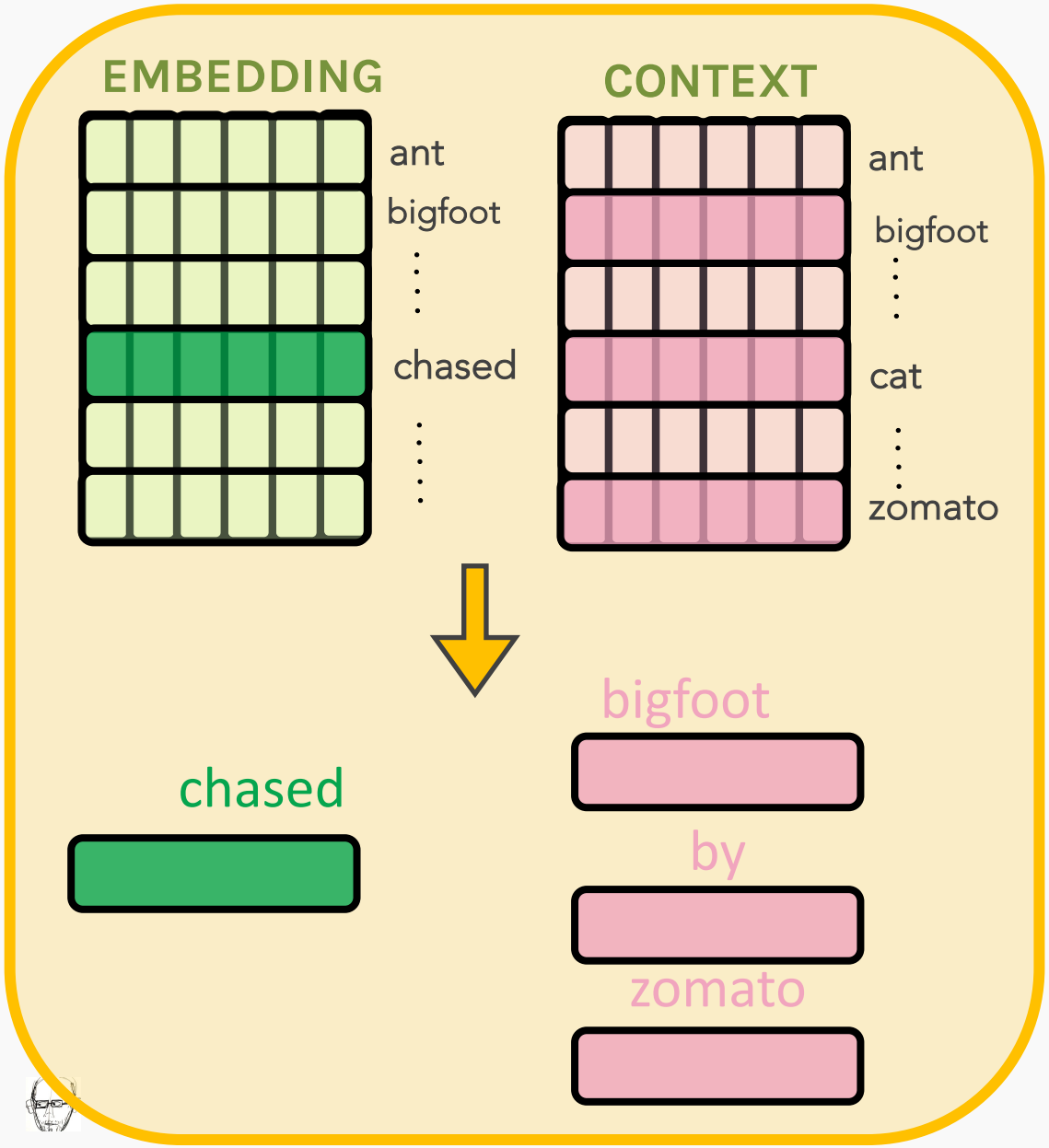
Pick randomly from our vocabulary (random sampling) and label them with **0**.

| input word | target word |
|---|---|
| a | chased |
| a | by |
| a | red |
| a | cat |
| red | by |
| red | a |
| red | cat |
| red | as |
| | |

| input word1 | Input word2 | target |
|---|---|---|
| a | chased | 1 |
| a | by | 1 |
| a | red | 1 |
| a | cat | 1 |
| red | by | 1 |
| red | a | 1 |
| red | cat | 1 |
| red | as | 1 |
| | | |

| input word | Output word | target |
|---|---|---|
| a | chased | 1 |
| a | bigfoot | 0 |
| a | zomato | 0 |
| a | by | 1 |
| .. | .. | .. |
| .. | .. | .. |
| a | red | 1 |
| | | |
| | | |

# Training the model




- The negative sampling probabilities are now sigmoids subtracted from 1, whereas the positives are simply sigmoids.
- We now compute the loss, and repeat over training examples in our batch.
- And backpropagate to obtain gradients and change the embeddings and weights some, for each batch, in each epoch

| Input word | Output word | Target | Input · Output | Sigmoid() |
|---|---|---|---|---|
| chased | bigfoot | 0 | -1.11 | 0.25 |
| chased | by | 1 | 0.2 | 0.55 |
| chased | zomato | 0 | 0.74 | 0.68 |

# The result

- We discard the Context matrix and **save the embedding matrix**.
- We can use the embedding matrix for our next task (perhaps a sentiment classifier).
- We could have trained embeddings along with that particular task to make the embeddings sentiment specific. There is always a tension between domain/task specific embeddings and generic ones.
- This tension is usually resolved in favor of using generic embeddings since task specific datasets seem to be smaller.
- We can still unfreeze pre-trained embedding layers to modify them for domain specific tasks via transfer learning.

# Usage of word2vec

- The pre-trained word2vec and other embeddings (such as GloVe) are used everywhere in NLP today.
- The ideas have been used elsewhere as well. AirBnB and Anghami model sequences of listings and songs using word2vec like techniques.
- Alibaba and Facebook use word2vec and graph embeddings for recommendations and social network analysis.

# What's next

**Lecture 5:** Using RNNs as a language: ELMo

**Lecture 6:** Seq2Seq + Attention

**Lecture 7:** Self attention models, Transformer models

**Lecture 8:** BERT

# THANK YOU