10-315
Introduction to ML

LLMs:
Word Embeddings & Attention

Instructor: Pat Virtue

Building up to Large Language Models

N-gram LMs

Word Embedding LMs

- Vector representation of vocab tokens
- Sampling next token
- Learning better vectors

Transformer LMs

- Increasing context size
- Attention
- Tranformer blocks

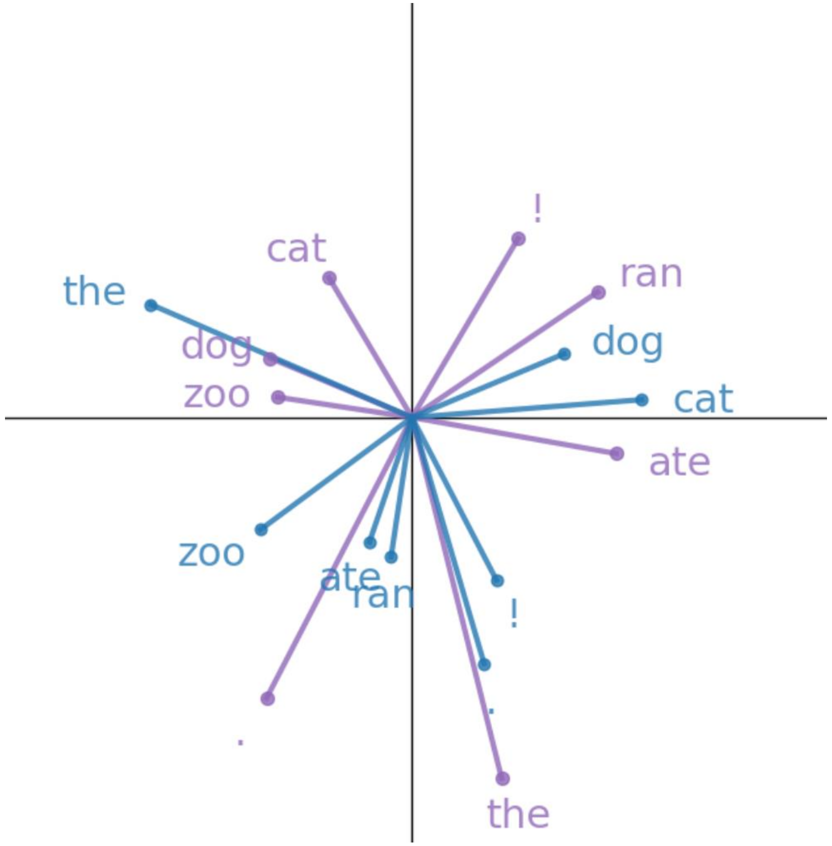
More Transformers

Word Embedding LMs

Word Embedding Language Models

Vector representation of vocab tokens

- Set of vectors for both previous and next token



Sampling next token

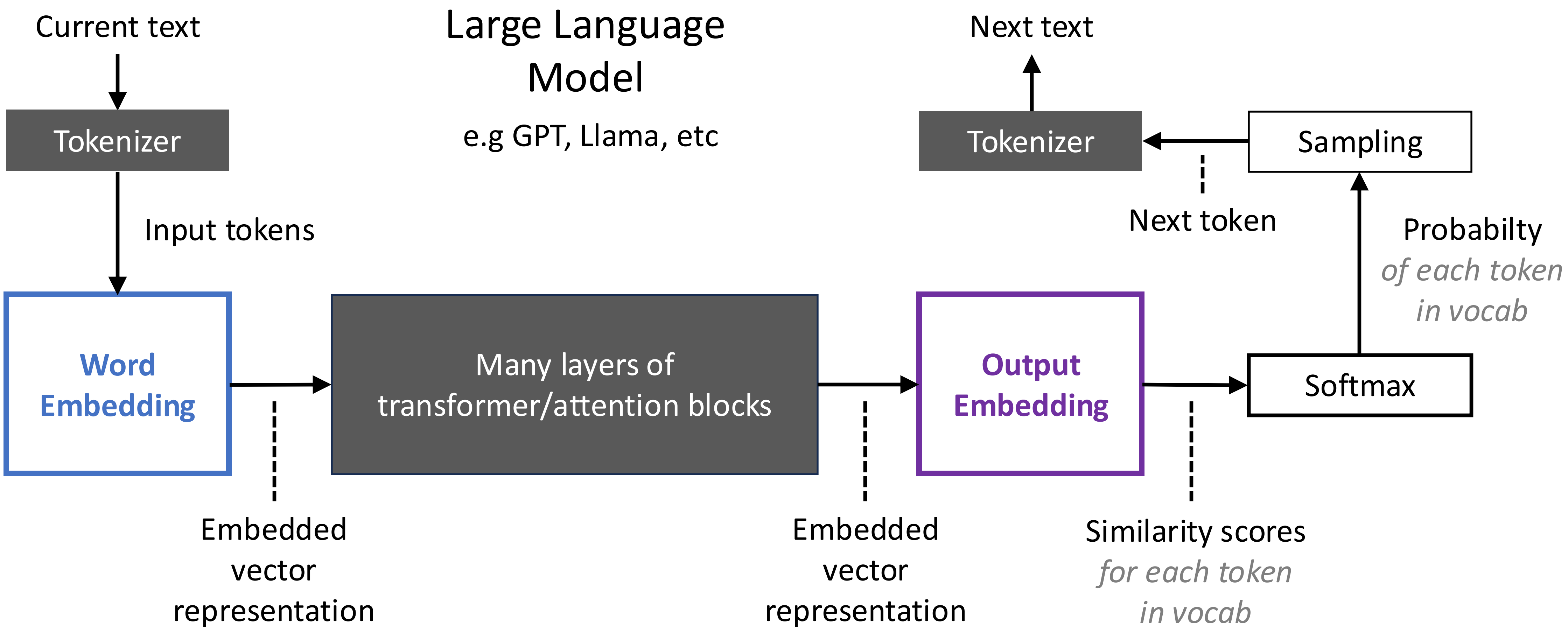
- Cosine similarity
- Softmax
- Sample from categorical distribution

Learning better vectors

- Cross-entropy loss
- SGD: looping through pairs of tokens in our corpus

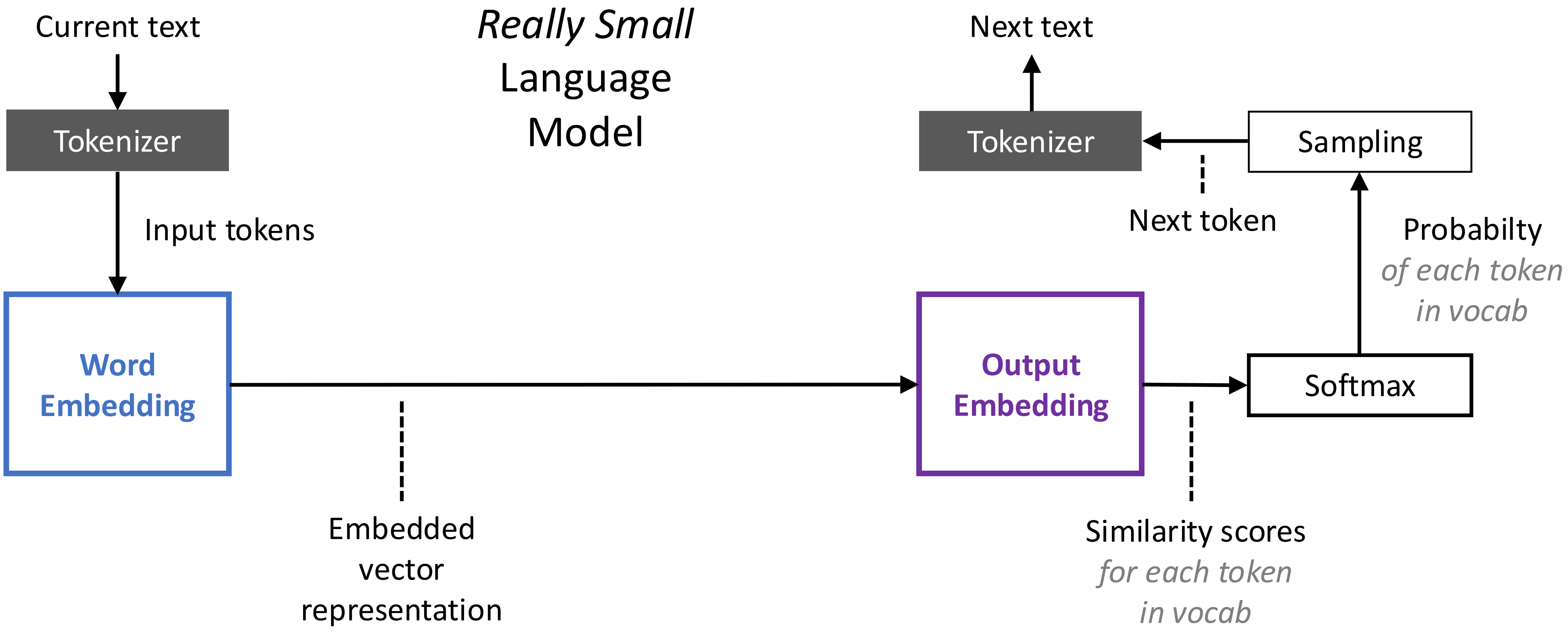
Word (Token) Embeddings

The beginning and the end of LLM networks



Simple Word Embedding LM

Building a language model with just word embedding layers ☺



Simple Word Embedding LM

Setup

Corpus:

```
The dog ran.
The dog ate.
The dog ran the zoo.
The cat ate the dog!
The cat ran the zoo.
```

→ Tokenizer →

(Simple) Tokenized Corpus:

```
[ 'the', 'dog', 'ran', '.', 'the',
  'dog', 'ate', '.', 'the', 'dog',
  'ran', 'the', 'zoo', '.', 'the',
  'cat', 'ate', 'the', 'dog', '!',
  'the', 'cat', 'ran', 'the',
  'zoo', '.' ]
```

Vocabulary:

```
[ '!',
  '.',
  'ate',
  'cat',
  'dog',
  'ran',
  'the',
  'zoo' ]
```

Simple Word Embedding LM

Vector representation for each token in vocabulary (initially random)

Two sets of vectors in $\mathbb{R}^M$ (we'll use $M = 2$ for better visualization)

V: to represent previous tokens

U: to represent next tokens

```
V:
!:     0.884,   0.196
.:     0.358,  -2.343
ate:  -1.085,   0.560
cat:   0.939,  -0.978
dog:   0.503,   0.406
ran:   0.323,  -0.493
the:  -0.792,  -0.842
zoo:  -1.280,   0.246
```

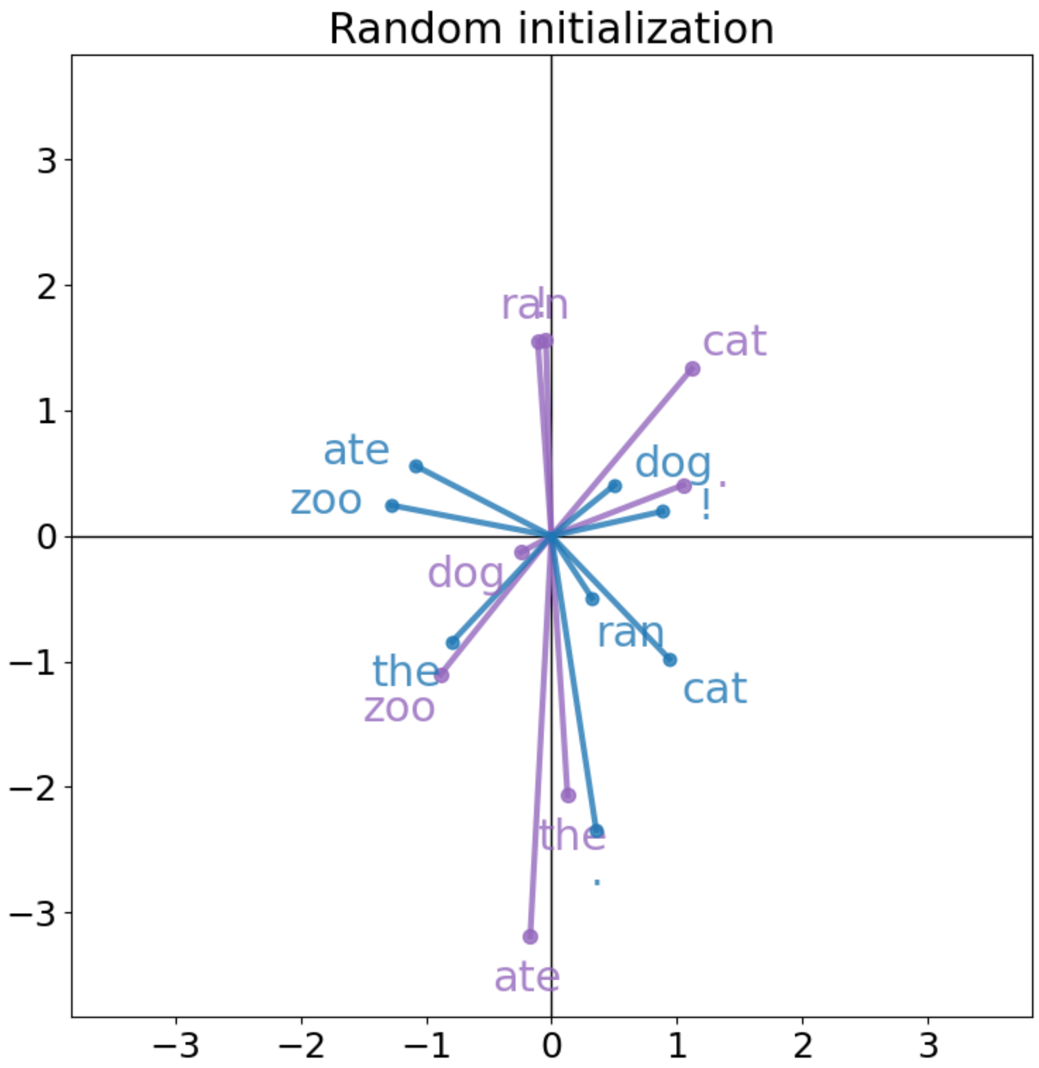
```
U:
!:    -0.044,   1.568
.:     1.051,   0.406
ate:  -0.169,  -3.190
cat:   1.120,   1.333
dog:  -0.243,  -0.130
ran:  -0.109,   1.556
the:   0.129,  -2.067
zoo:  -0.885,  -1.105
```

Simple Word Embedding LM

Vector representation for each token in vocabulary (initially random)

V: Previous

```
V:
!:      0.884,   0.196
.:      0.358,  -2.343
ate:   -1.085,   0.560
cat:    0.939,  -0.978
dog:    0.503,   0.406
ran:    0.323,  -0.493
the:   -0.792,  -0.842
zoo:   -1.280,   0.246
```

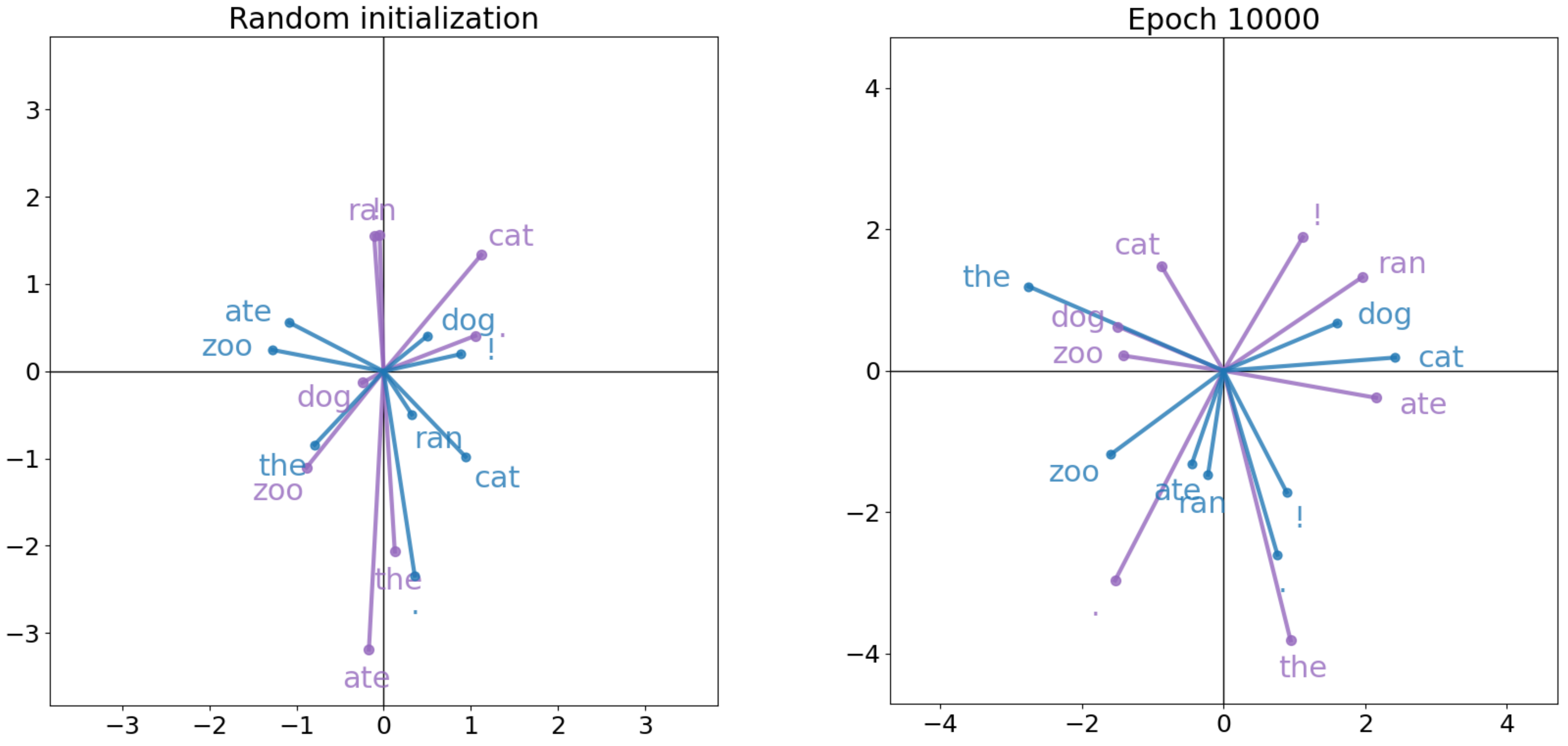


U: Next

```
U:
!:     -0.044,   1.568
.:      1.051,   0.406
ate:   -0.169,  -3.190
cat:    1.120,   1.333
dog:   -0.243,  -0.130
ran:   -0.109,   1.556
the:    0.129,  -2.067
zoo:   -0.885,  -1.105
```

Simple Word Embedding LM

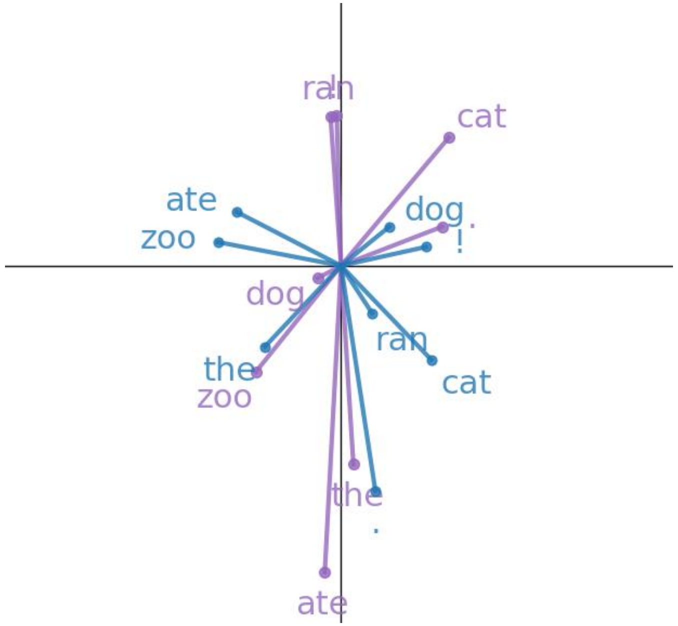
After training our LM, we'll learn more organized vectors (details later)



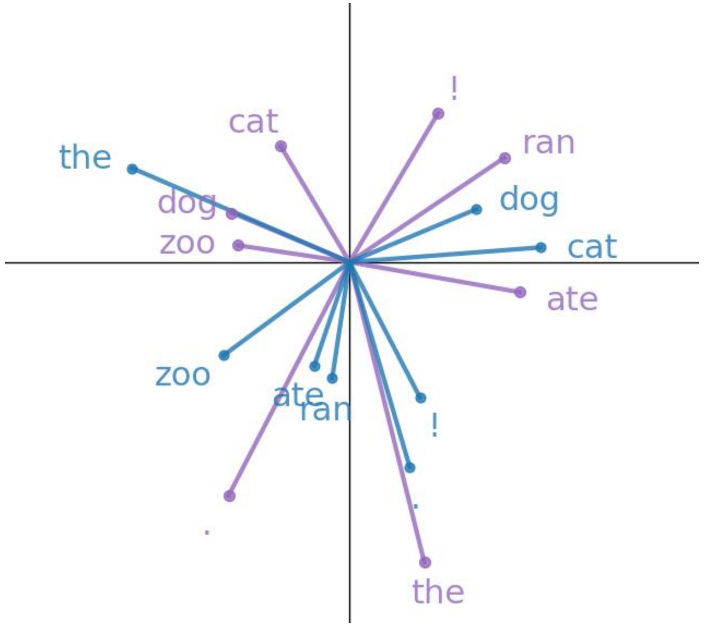
Simple Word Embedding LM

We can use either of these models to generate text, staring with "the dog"

Random vectors



Trained vectors



Generated tokens from random and trained models

the dog cat ate ran zoo zoo zoo dog dog cat cat ate ran . ate . ate ate ! the ate	**the dog** ate . the dog ! the zoo . the dog ran the zoo . the dog ate the dog !

Outline: Word Embedding LM

Vector representation of vocab tokens

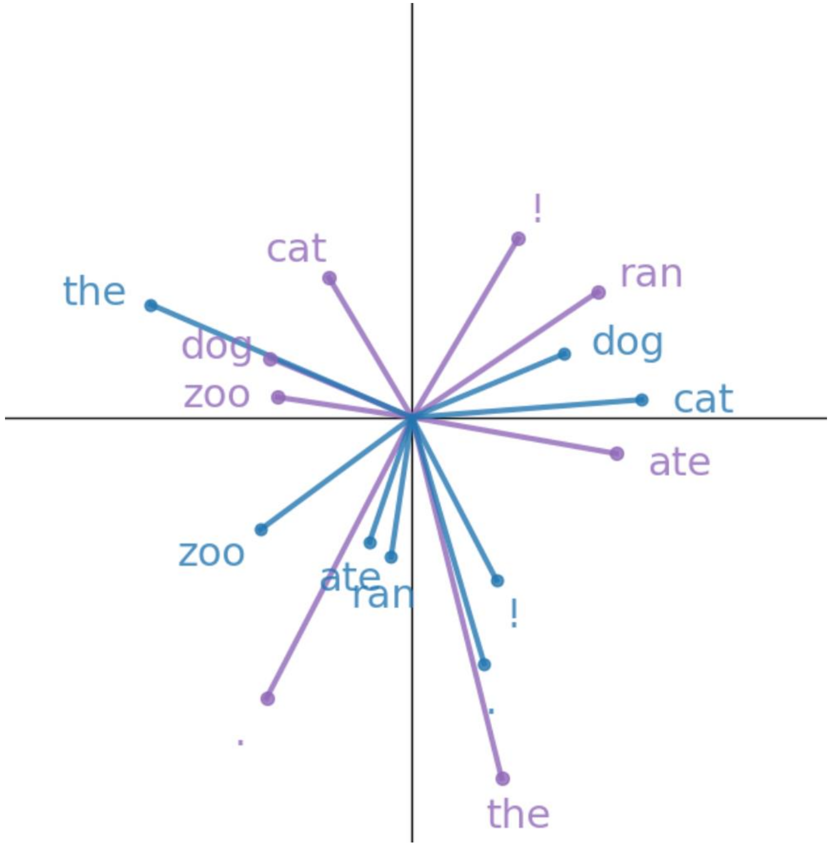
- Set of vectors for both previous and next token

Sampling next token

- Cosine similarity
- Softmax
- Sample from categorical distribution

Learning better vectors

- Cross-entropy loss
- SGD: looping through pairs of tokens in our corpus



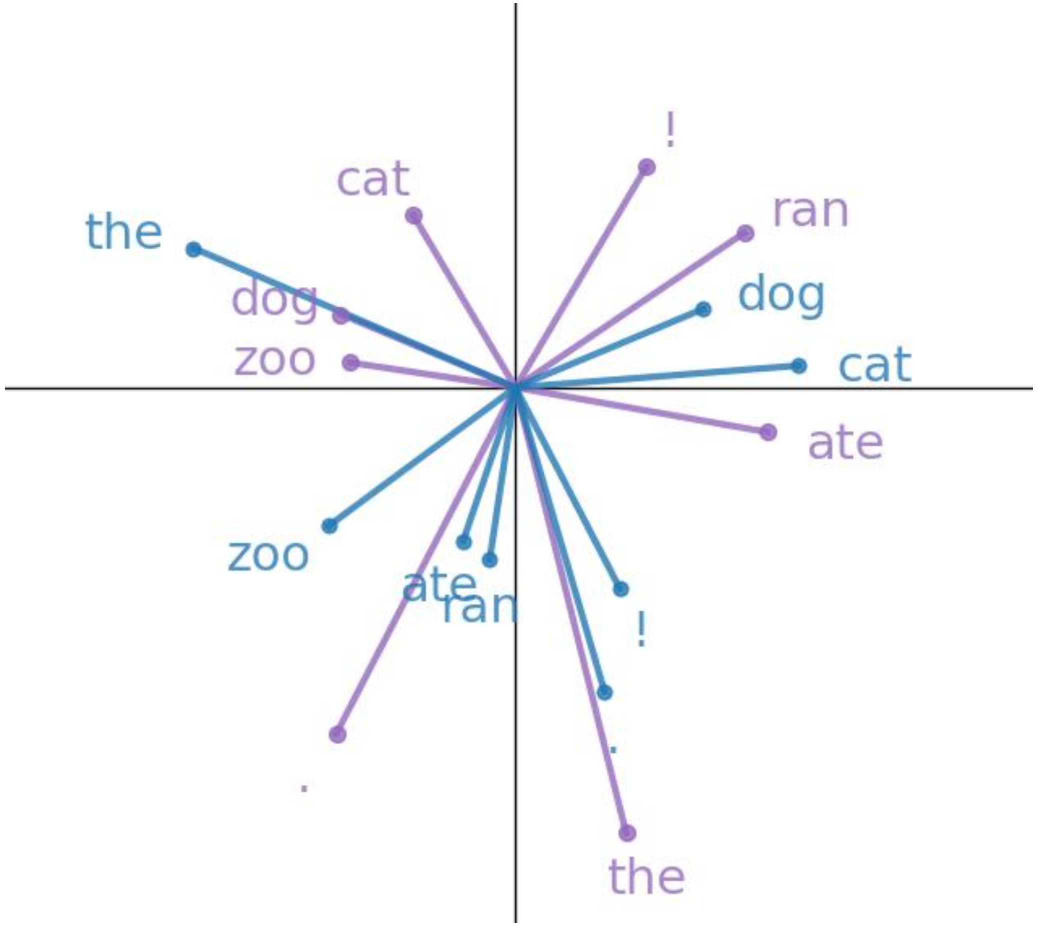
Sampling from Word Embeddings

Suppose we have a trained set of embedded vectors and we want to generate a the next token after the previous token 'the'.

V: previous tokens

U: next tokens

Which token should be our next token?



Sampling from Word Embeddings

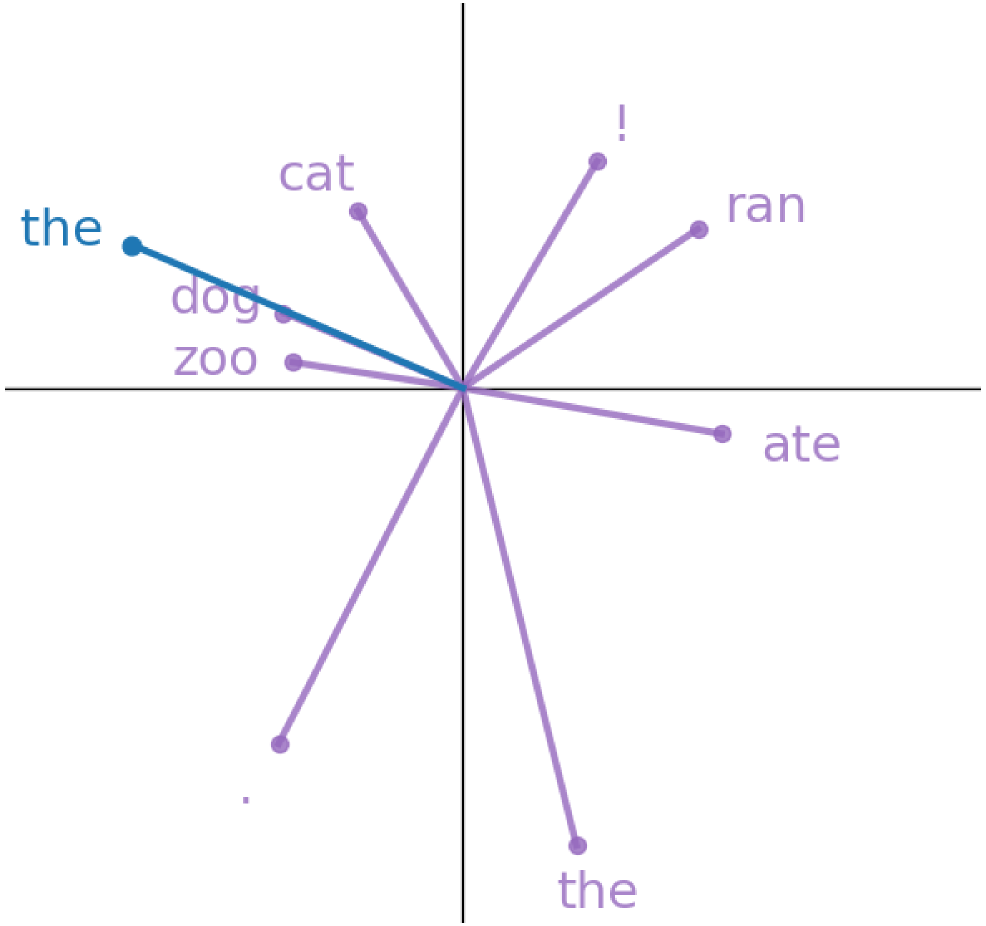
Suppose we have a trained set of embedded vectors and we want to generate a the next token after the previous token 'the'.

V: previous tokens

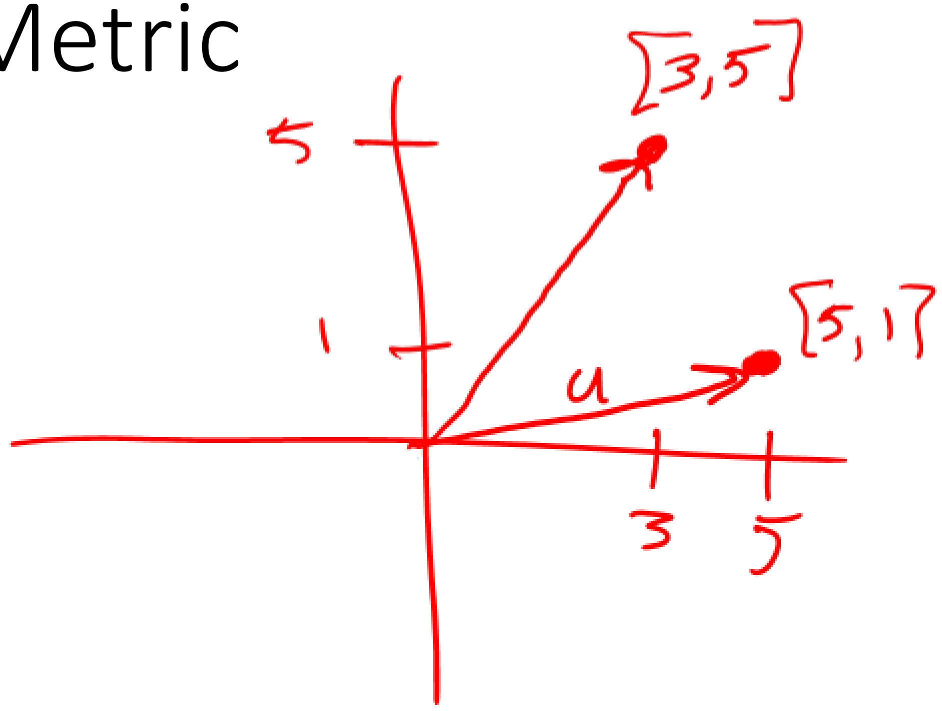
U: next tokens

Which token should be our next token?

1. Lookup the index i for the vocab token 'the'
2. Access the i-th row of V, $\mathbf{v}$
3. Compare $\mathbf{v}$ to all vectors in U



(Unnormalized) Cosine Similarity Metric



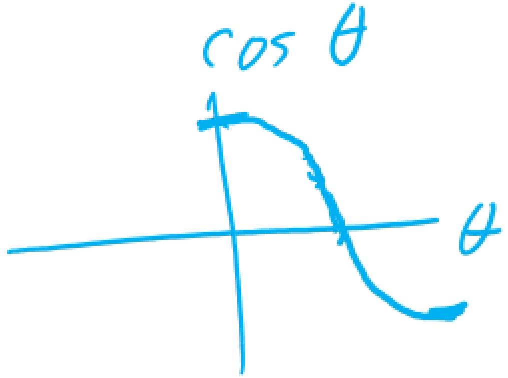
We've been using Euclidean distance

- $d(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|_2$

Cosine similarity

- Two vectors are similar if their dot product is positive and big
- $f(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T\mathbf{v}$
- (Why cosine?)
  - Two vectors are similar if the angle between them is small (small angle → large cos θ)
  - $f(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T\mathbf{v} = \|\vec{u}\|_2 \|\vec{v}\|_2 \cos\theta$

$$\cos\theta = \frac{\mathbf{u}^T\mathbf{v}}{\|\mathbf{u}\|_2 \|\mathbf{v}\|_2}$$

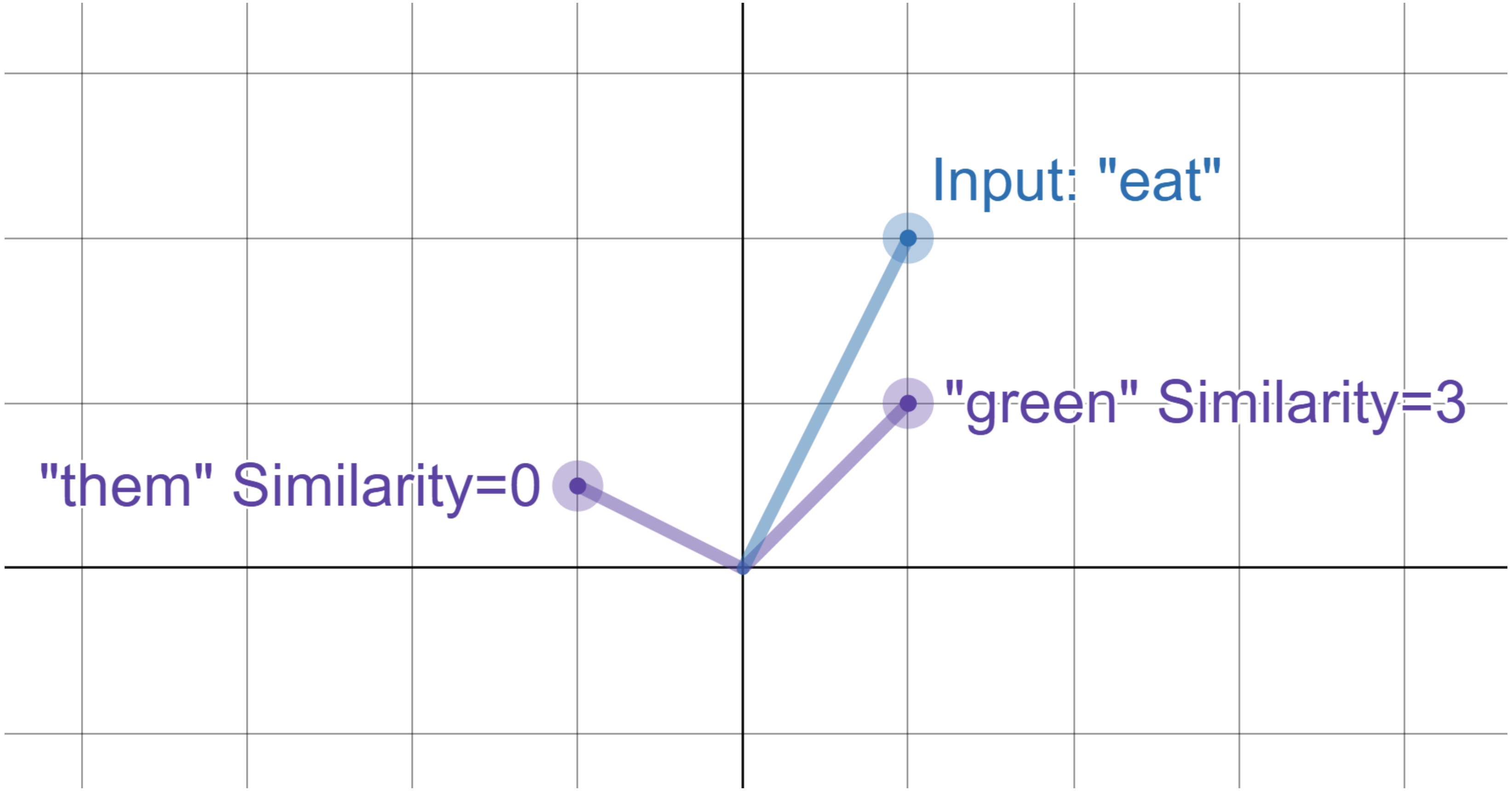


(Unnormalized) Cosine Similarity Metric

Cosine similarity Desmos demo

https://www.desmos.com/calculator/82m4zkjlkc

$f(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T \mathbf{v}$



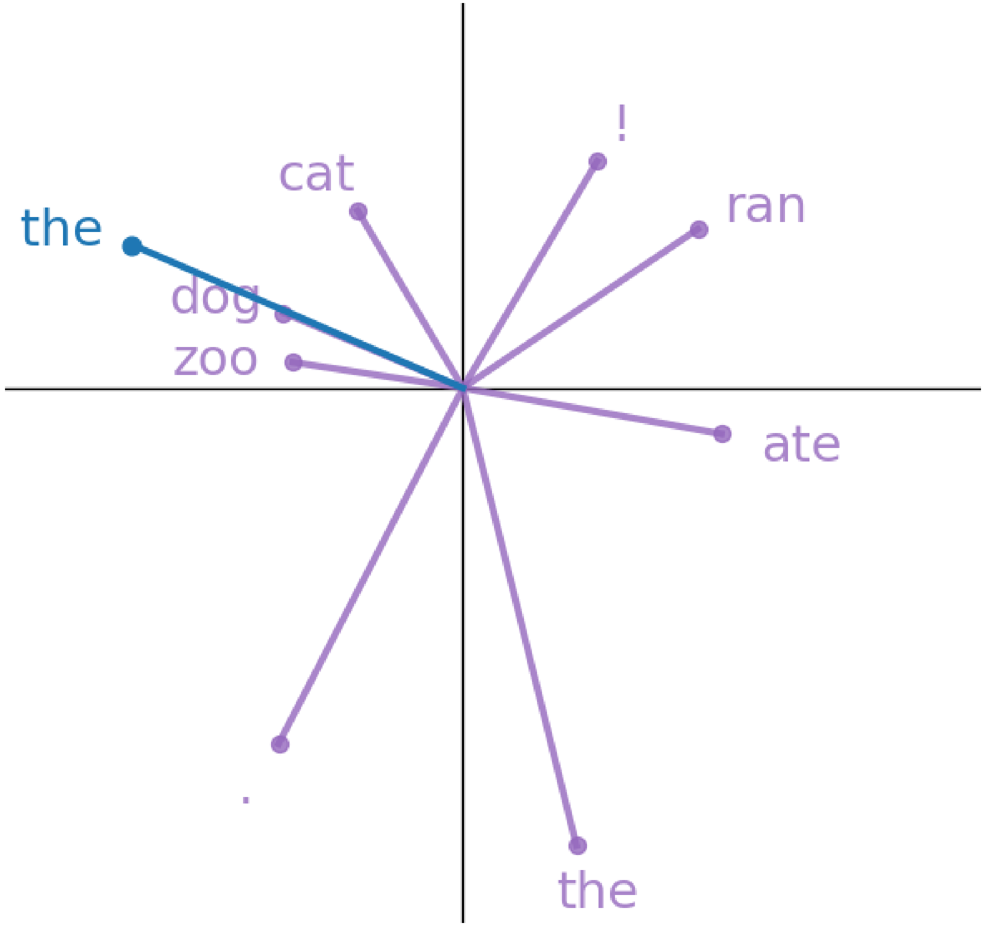
Sampling from Word Embeddings

Suppose we have a trained set of embedded vectors and we want to generate a the next token after the previous token 'the'.

V: previous tokens

U: next tokens

1. Compute similarity scores
   $\mathbf{s} = U\mathbf{v}$



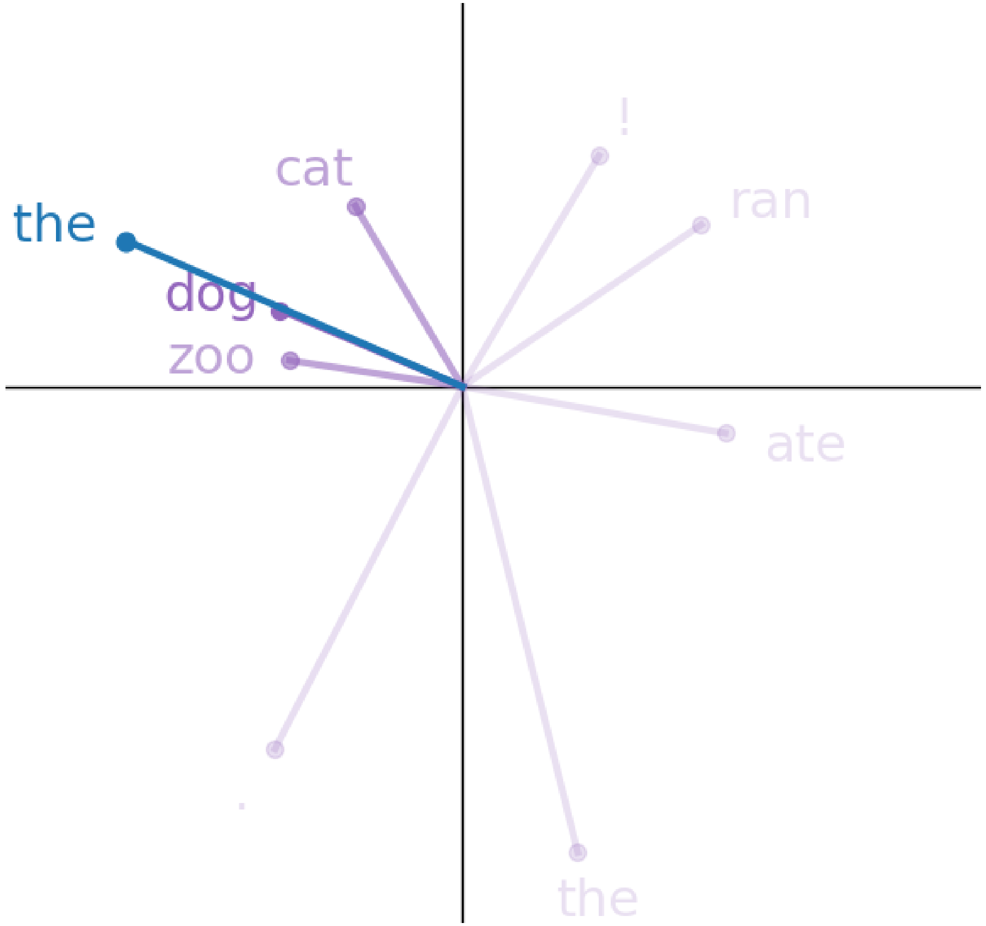
Sampling from Word Embeddings

Suppose we have a trained set of embedded vectors and we want to generate a the next token after the previous token 'the'.

V: previous tokens

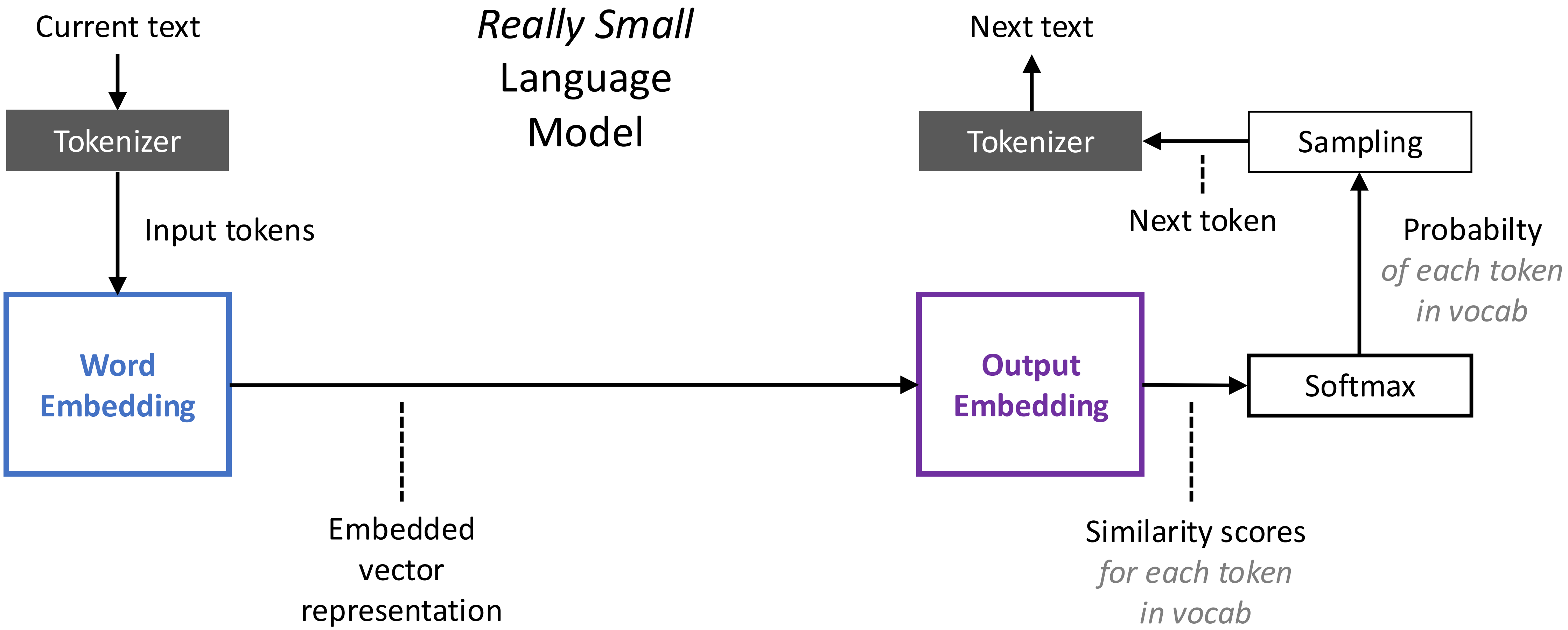
U: next tokens

1. Compute similarity scores
   $\mathbf{s} = U\mathbf{v}$
2. Convert to probabilities
   $\hat{\mathbf{y}} = g_{softmax}(\mathbf{s})$
3. Sample from Categorical distribution defined by $\hat{\mathbf{y}}$
4. LOOP: next token → prev



Simple Word Embedding LM

Building a language model with just word embedding layers ☺



PyTorch for Word Embedding LM

Two matrices of features vectors:

W_1: for context

W_2: for next token

Using PyTorch

```
WordEmbedLM(
  (encode): Linear(in_features=vocab_size, out_features=2)
  (decode): Linear(in_features=2, out_features=vocab_size)
)

F.softmax(model.forward(x_onehot))
```

PyTorch for Word Embedding LM

Two matrices of features vectors:

W_1: for context

W_2: for next token

Using PyTorch

torch.nn.Embedding(*num_embeddings, embedding_dim*)

```
WordEmbedLM(
  (encode): Linear(in_features=vocab_size, out_features=2)
  (decode): Linear(in_features=2, out_features=vocab_size)
)
```

x_index

```
F.softmax(model.forward(x_onehot))
```

Learning Better Vectors

Classic ML recipe:

1. Training data

2. Hypothesis function

$$\hat{\mathbf{y}} = g_{softmax}(U\mathbf{v})$$

3. Formulate objective

   Loss:

   Objective: $\frac{1}{N}\sum_i^N J^{(i)}(U,V) \qquad J^{(i)}(U,V)$

4. SGD to find parameters that optimize the objective

(Simple) Tokenized Corpus:

```
[ 'the', 'dog', 'ran', '.', 'the',
  'dog', 'ate', '.', 'the', 'dog',
  'ran', 'the', 'zoo', '.', 'the',
  'cat', 'ate', 'the', 'dog', '!',
  'the', 'cat', 'ran', 'the',
  'zoo', '.' ]
```

Tranformer LMs

Transformer Language Models

Increasing context size

- Uniform average of context vectors
- Position encoding

Attention

- Weighted average of context vectors
- Query Keys Values
- Expressive power of linear transforms

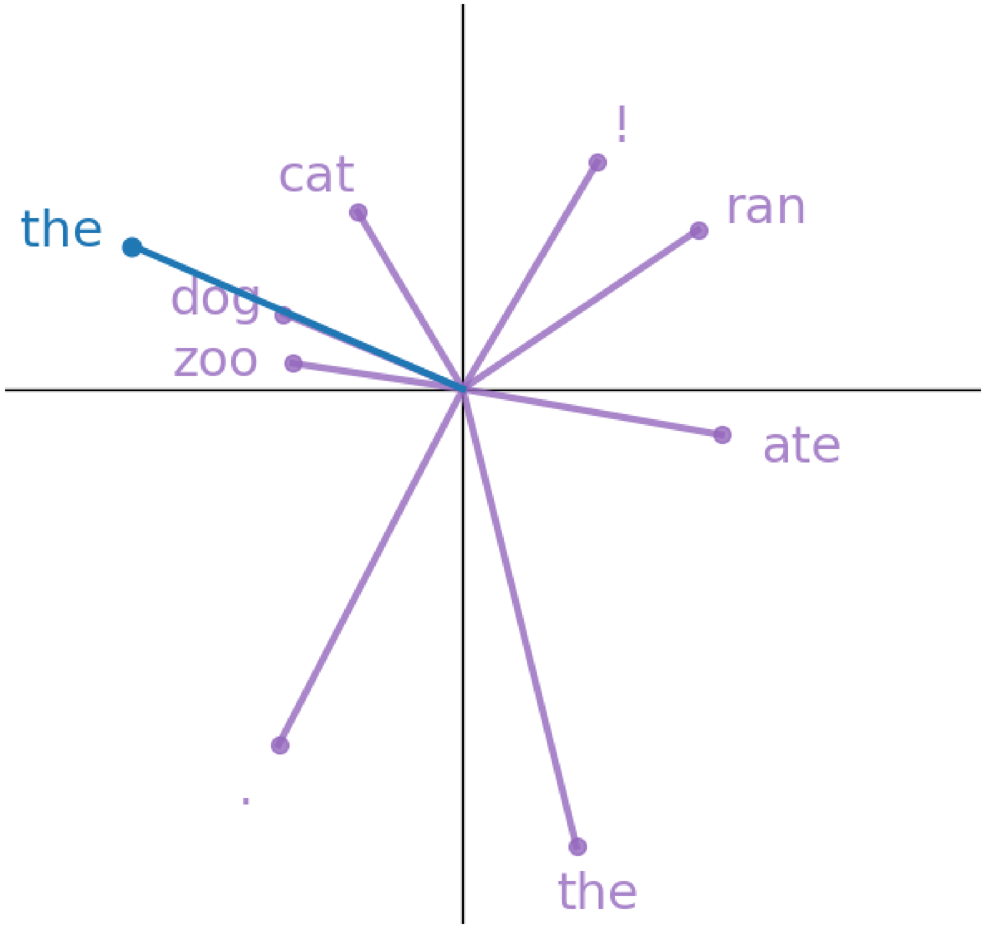
Transformer blocks

Increasing Context Size

What if we want to have more input tokens?

V: for context

U: for next token

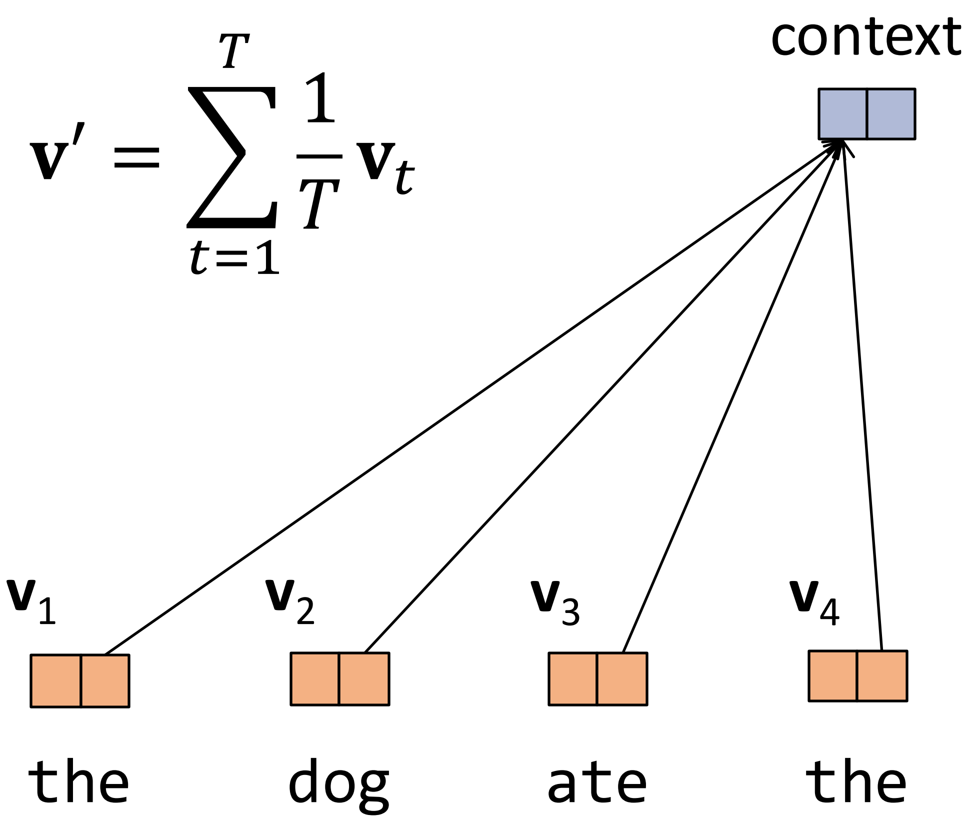


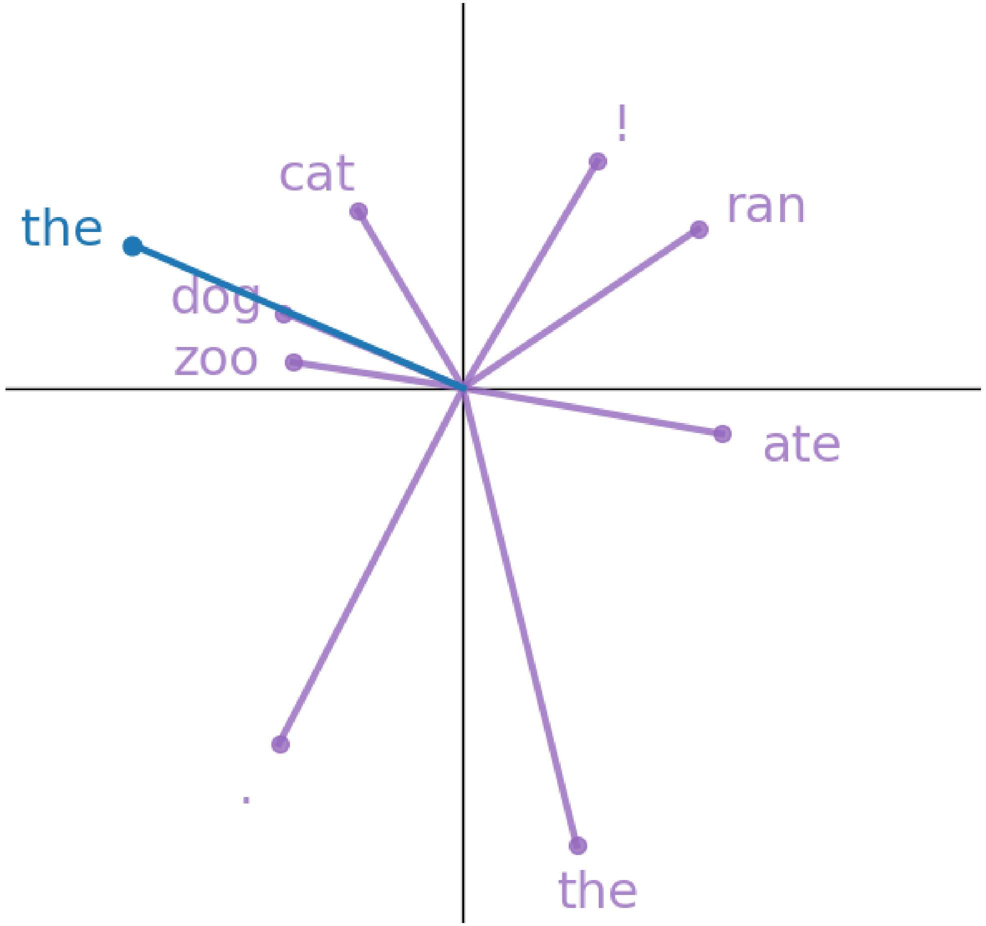
Increasing Context Size

What if we want to have more input tokens?

Uniform average over T input context tokens

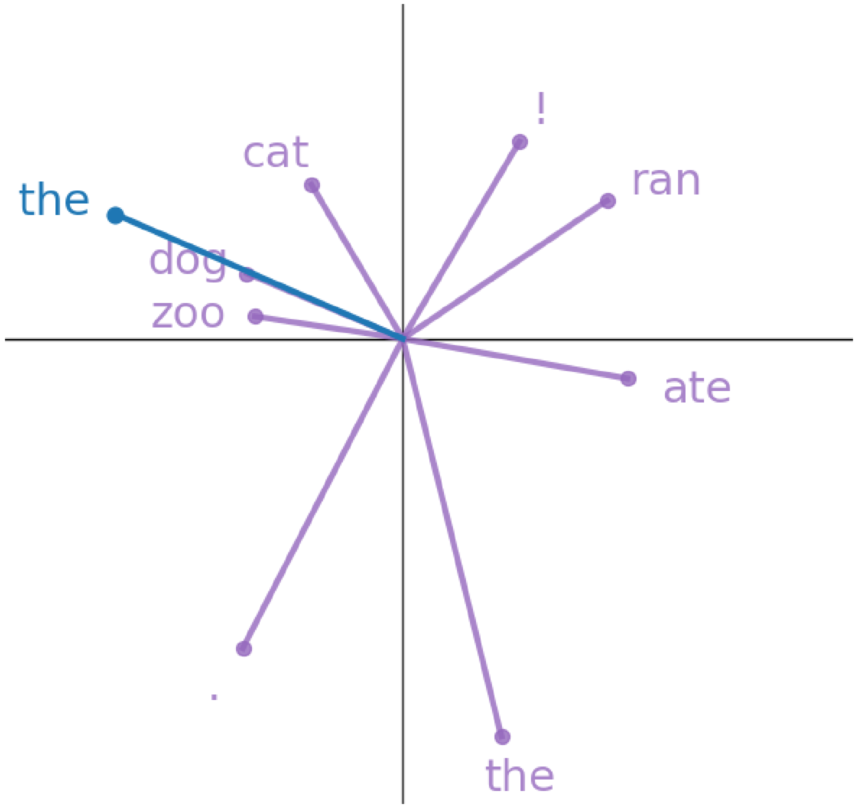
$$\mathbf{v}' = \sum_{t=1}^{T} \frac{1}{T} \mathbf{v}_t$$

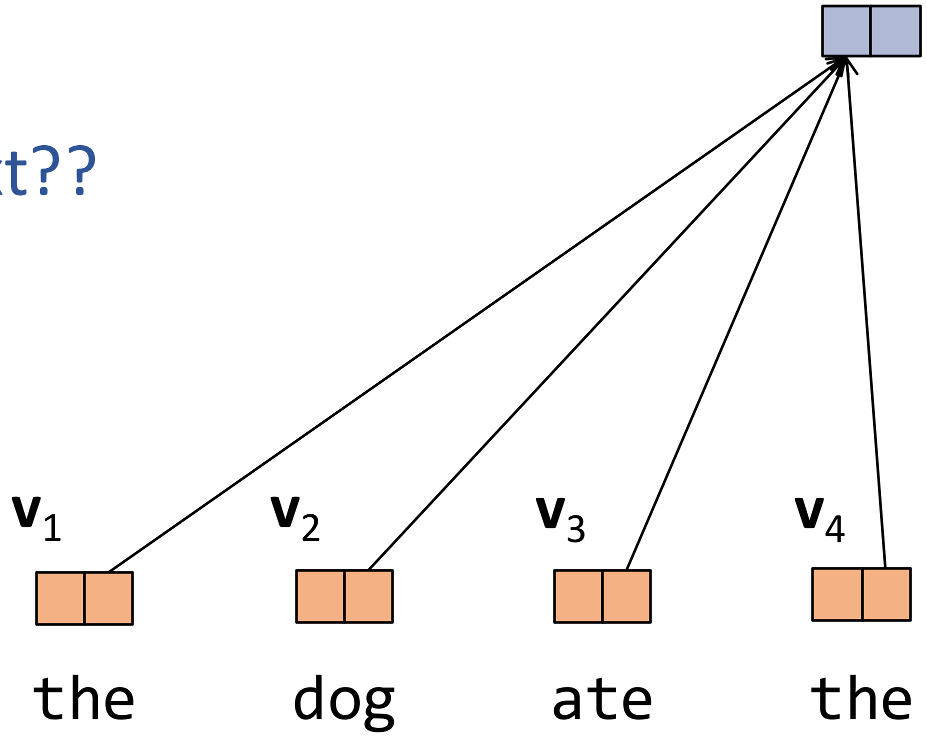




Position Encoding

What about position within the input context??





Position Encoding

What about position within the input context??

Rotary Position Encoding (RoPE) Desmos Demo

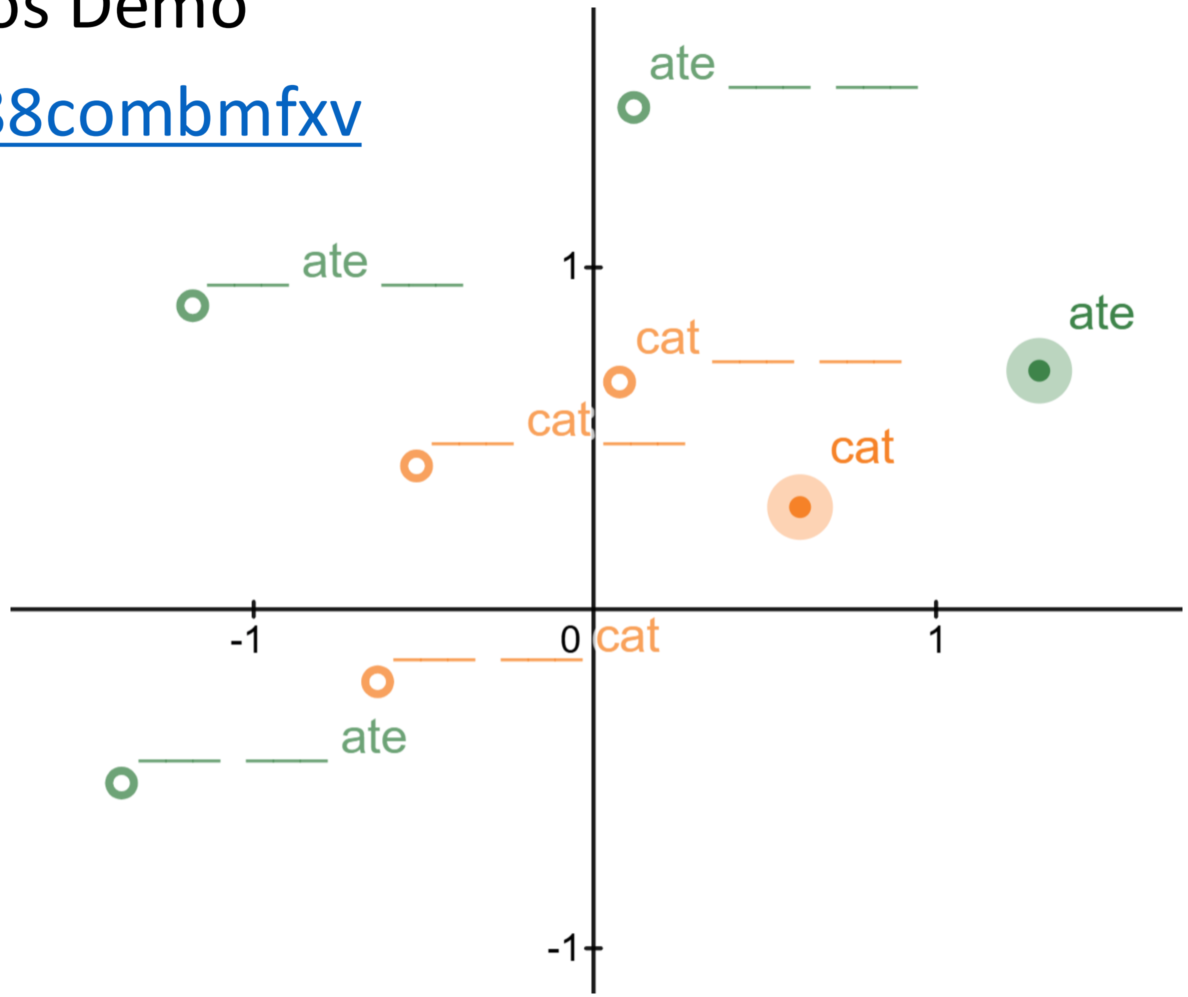
https://www.desmos.com/calculator/88combmfxv

2D version:

Given a fixed base rotation angle θ_1,

an embedded vector $\mathbf{x}$ at integer position, i_{pos}, will be rotated by angle $\theta = i_{pos}\theta_1$:

$$\mathbf{x}' = \text{Rotate}(\mathbf{x}, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \mathbf{x}$$



Attention

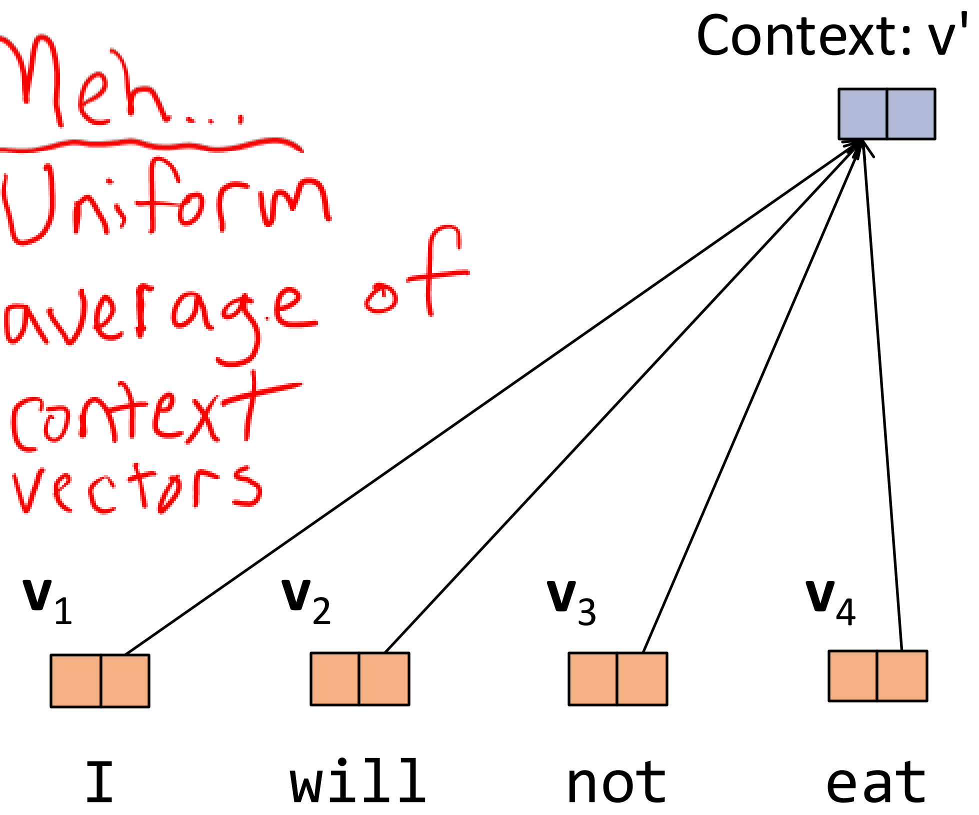
Learn to pay attention!

We can do better than uniform combination of input

$$\mathbf{v}' = \sum_{t=1}^{T} \frac{1}{T} \mathbf{v}_t \qquad \mathbf{v}' = \sum_{t=1}^{T} a_t \mathbf{v}_t$$

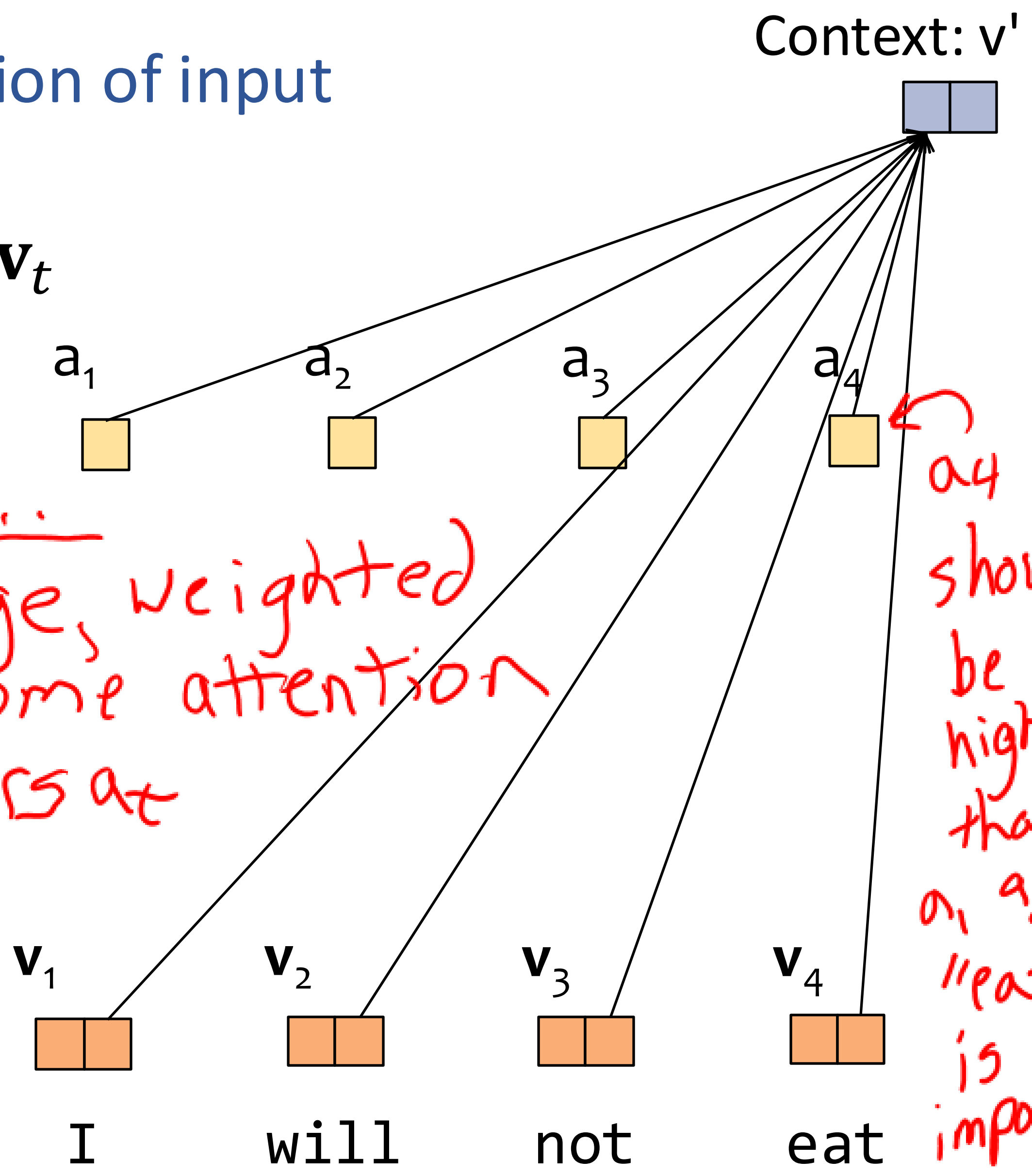
Meh...

Uniform average of context vectors



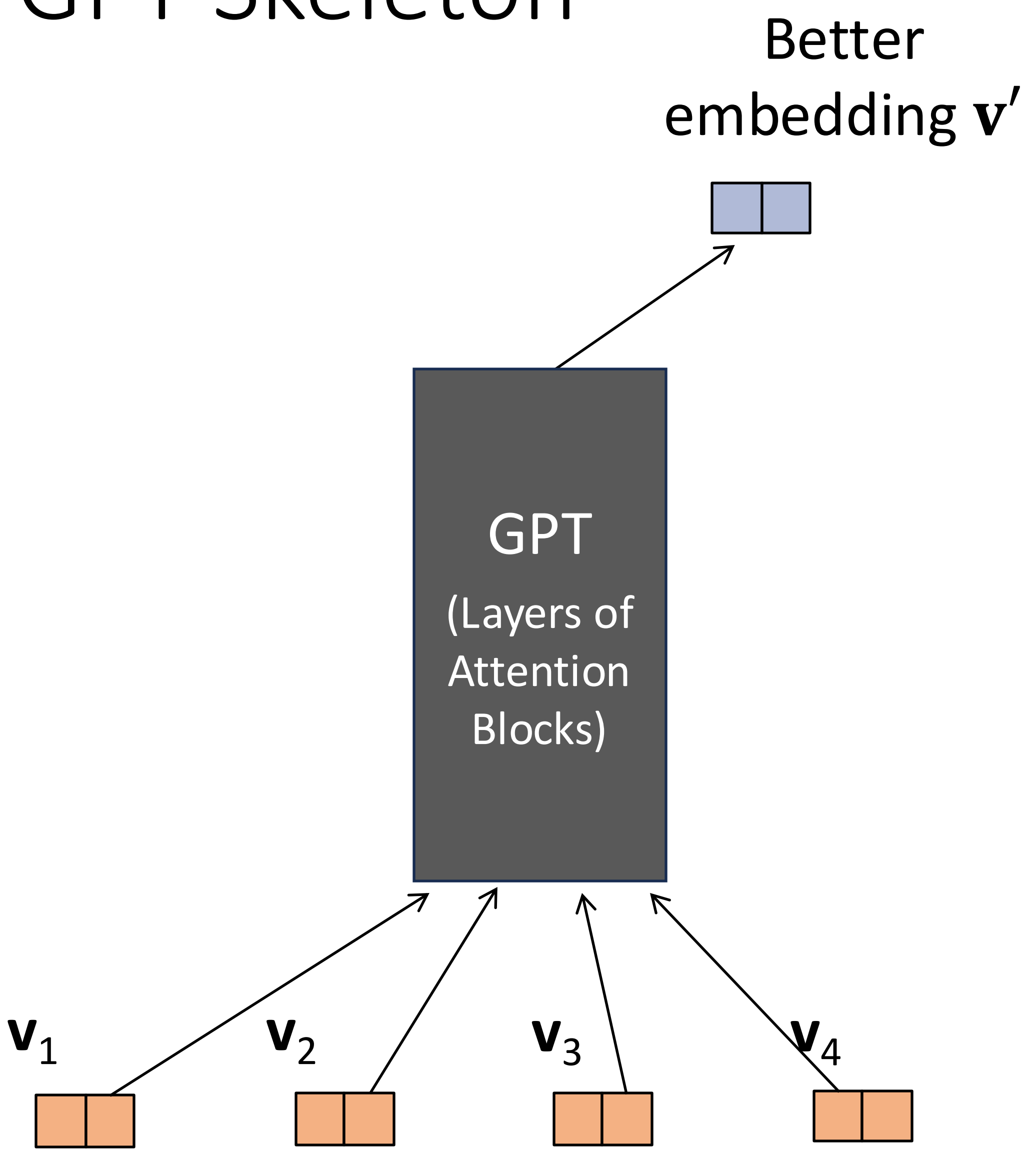
Want...

Average, weighted by some attention factors a_t

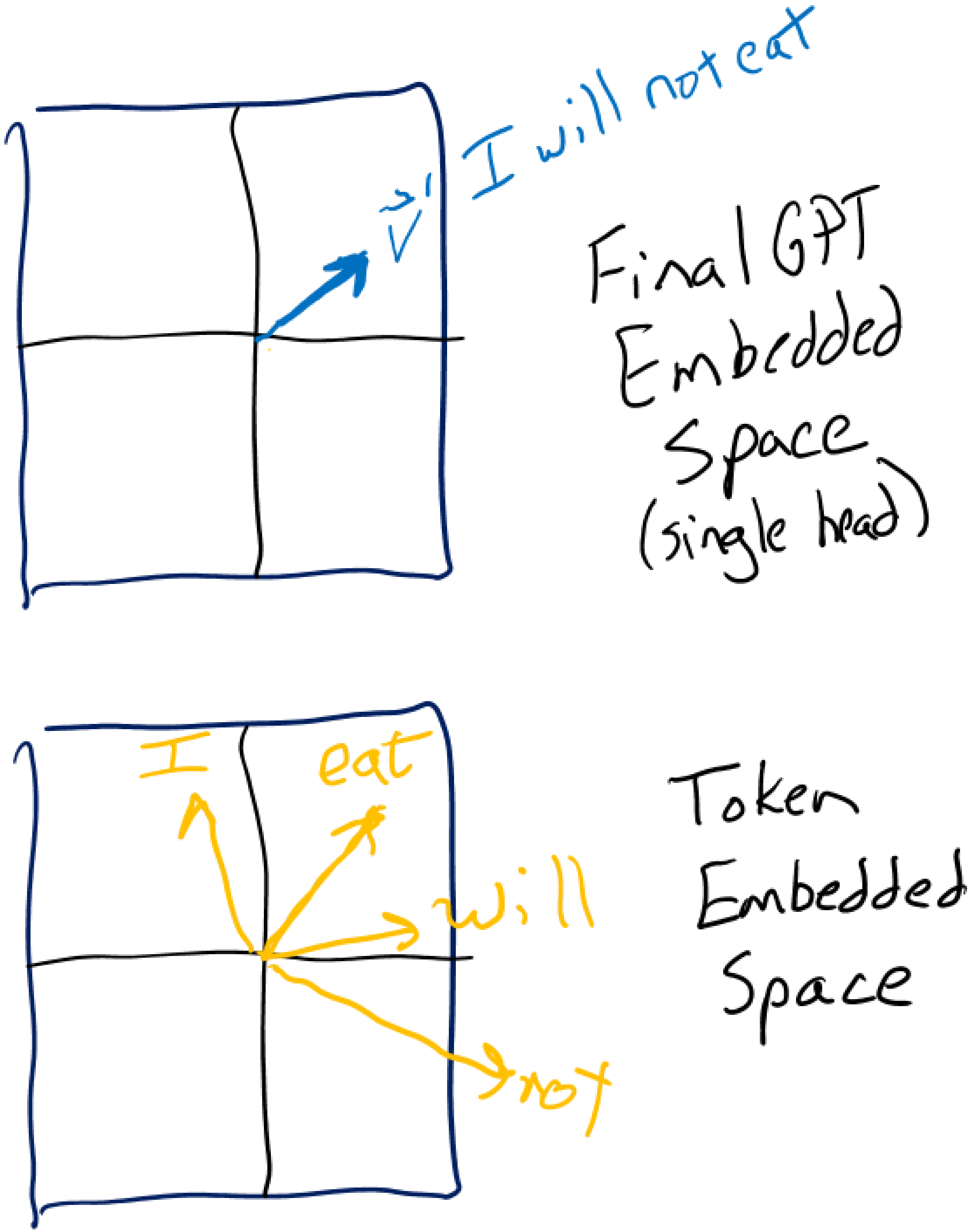


a_4 should be higher than a_1 a_2 a_3 "eat" is important

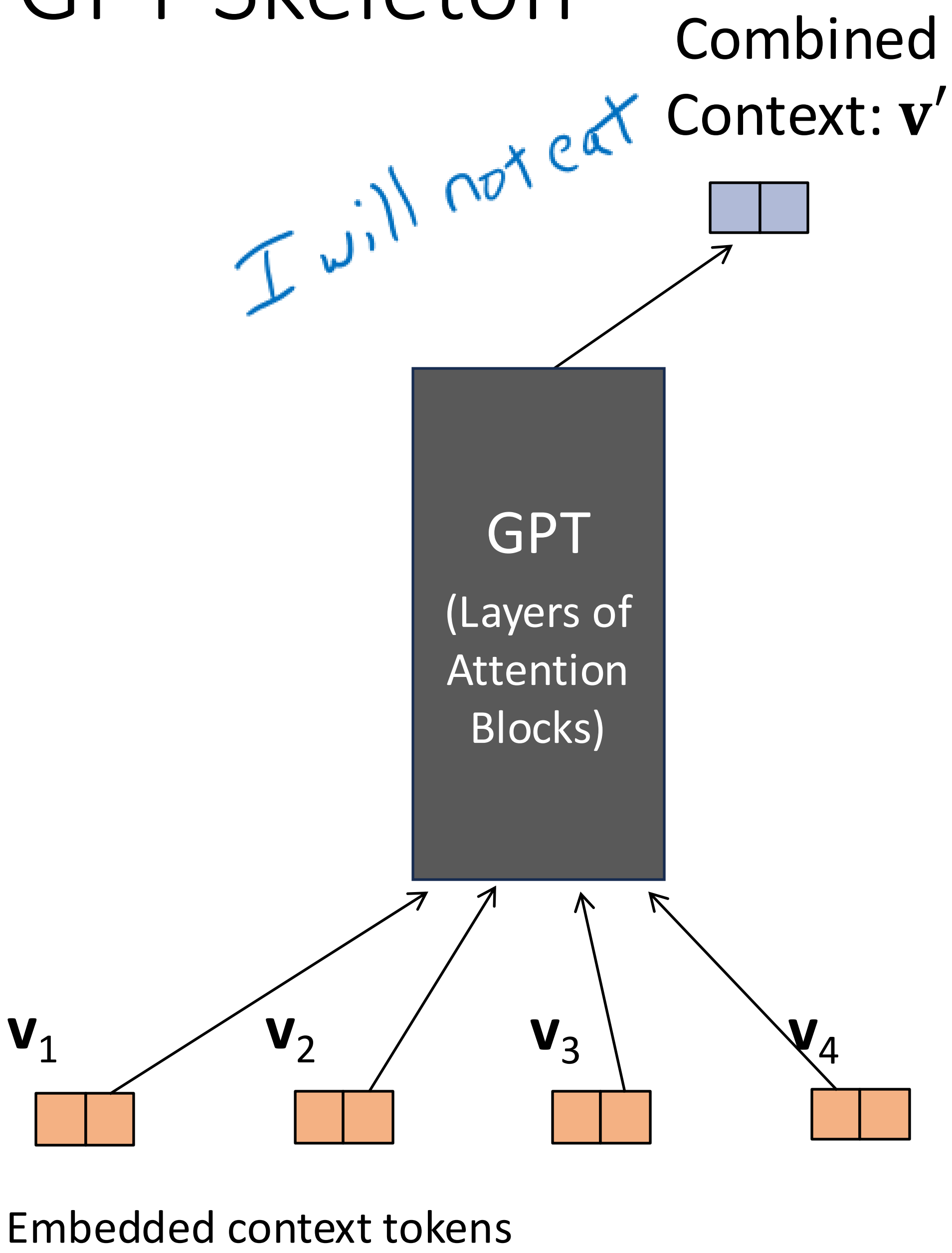
GPT Skeleton

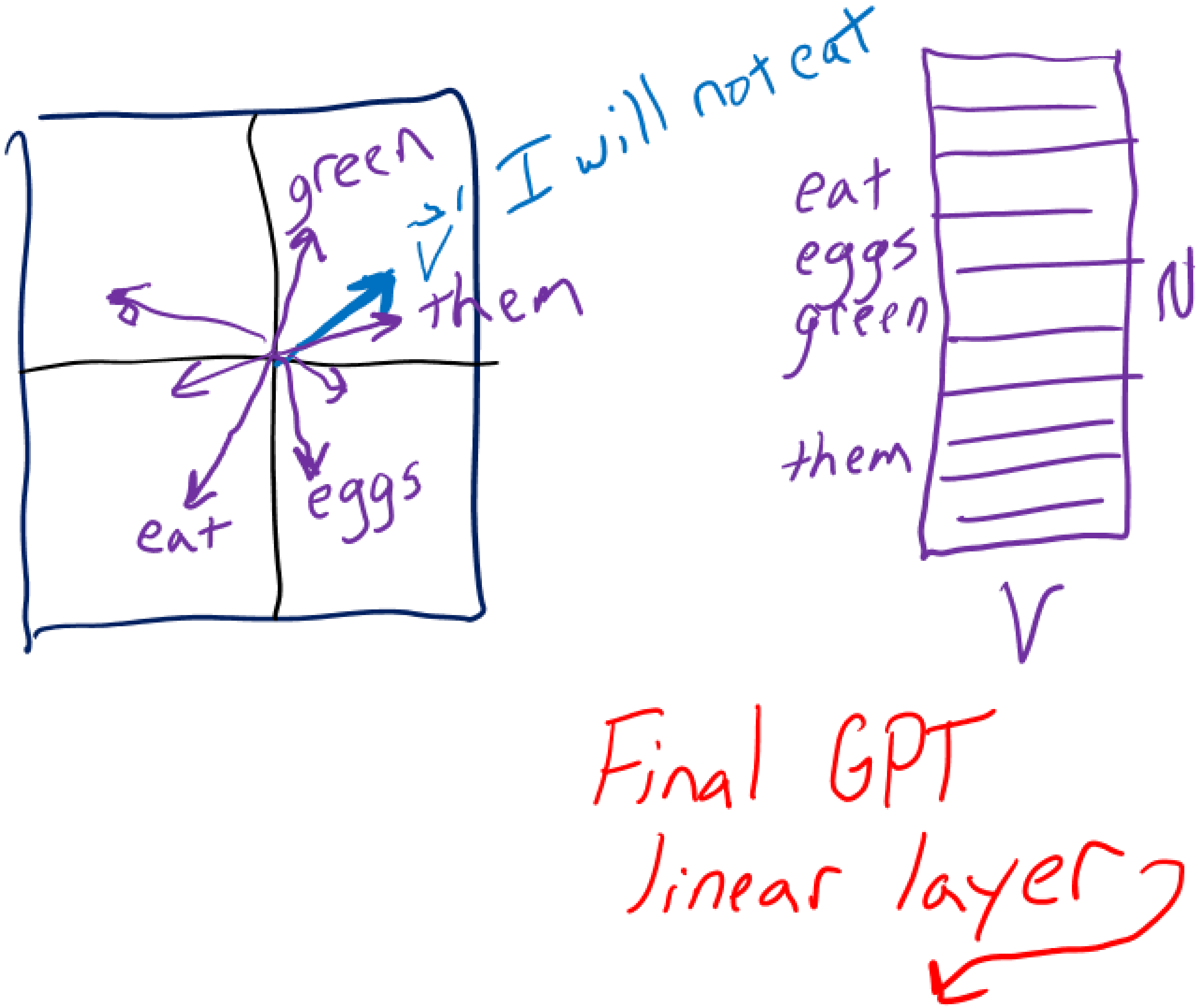


Embedded context tokens



GPT Skeleton





$$\hat{\mathbf{y}} = g_{softmax}(U\mathbf{v}')$$

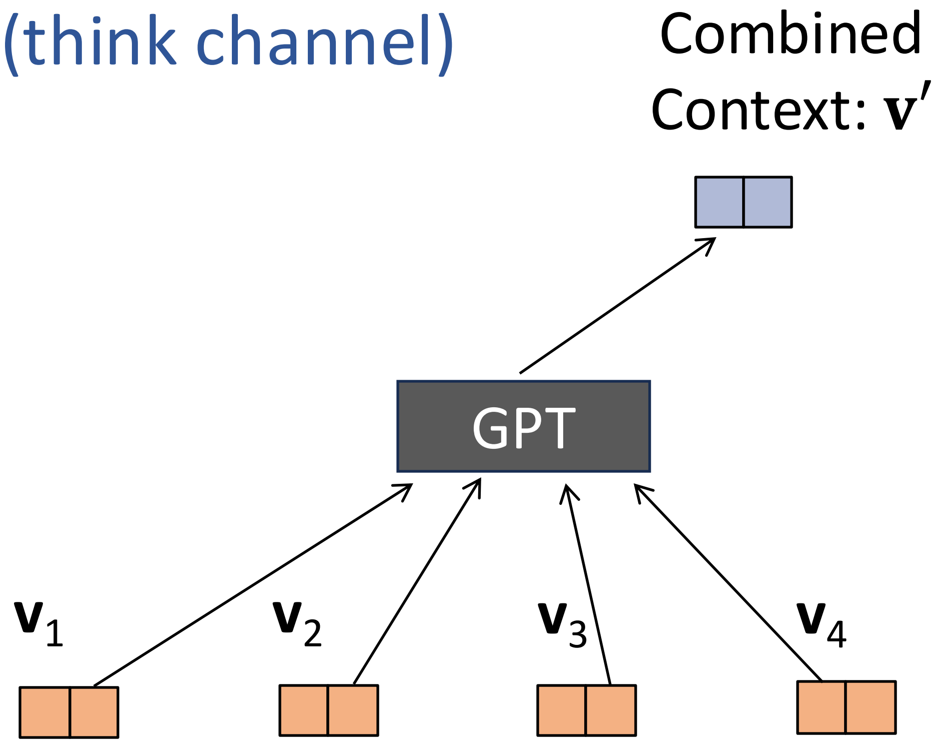
$$next\_idx = \underset{j}{\operatorname{argmax}}\, \hat{\mathbf{y}}$$

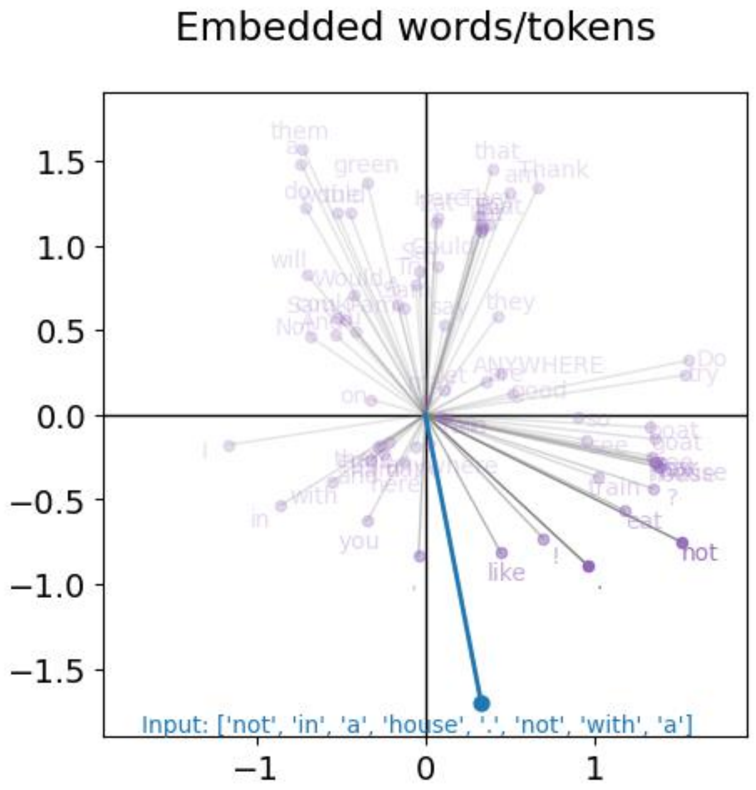
MinGPT Femto

2-D embedded space

1 attention layer

1 attention head (think channel)





$$\hat{\mathbf{y}} = g_{softmax}(U\mathbf{v}')$$

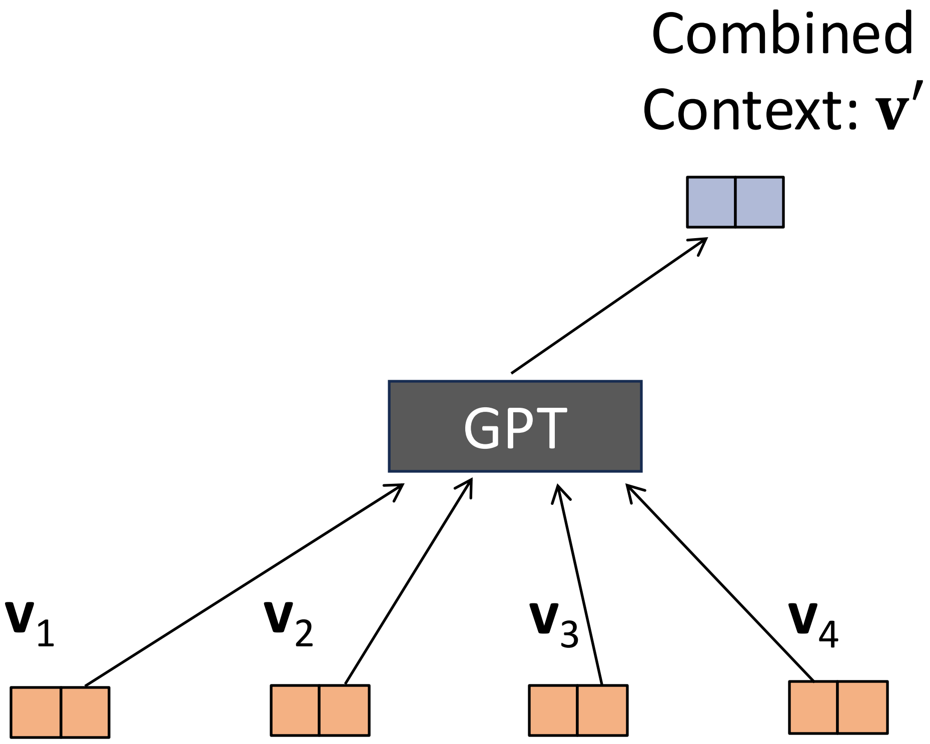
$$next\_idx = \underset{j}{\operatorname{argmax}}\, \hat{\mathbf{y}}$$

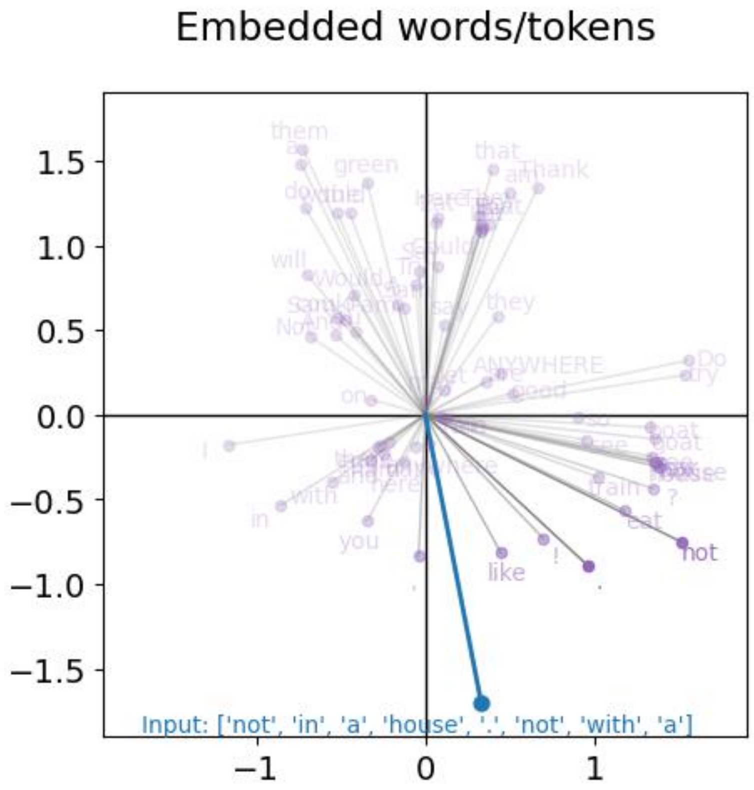
MinGPT Femto

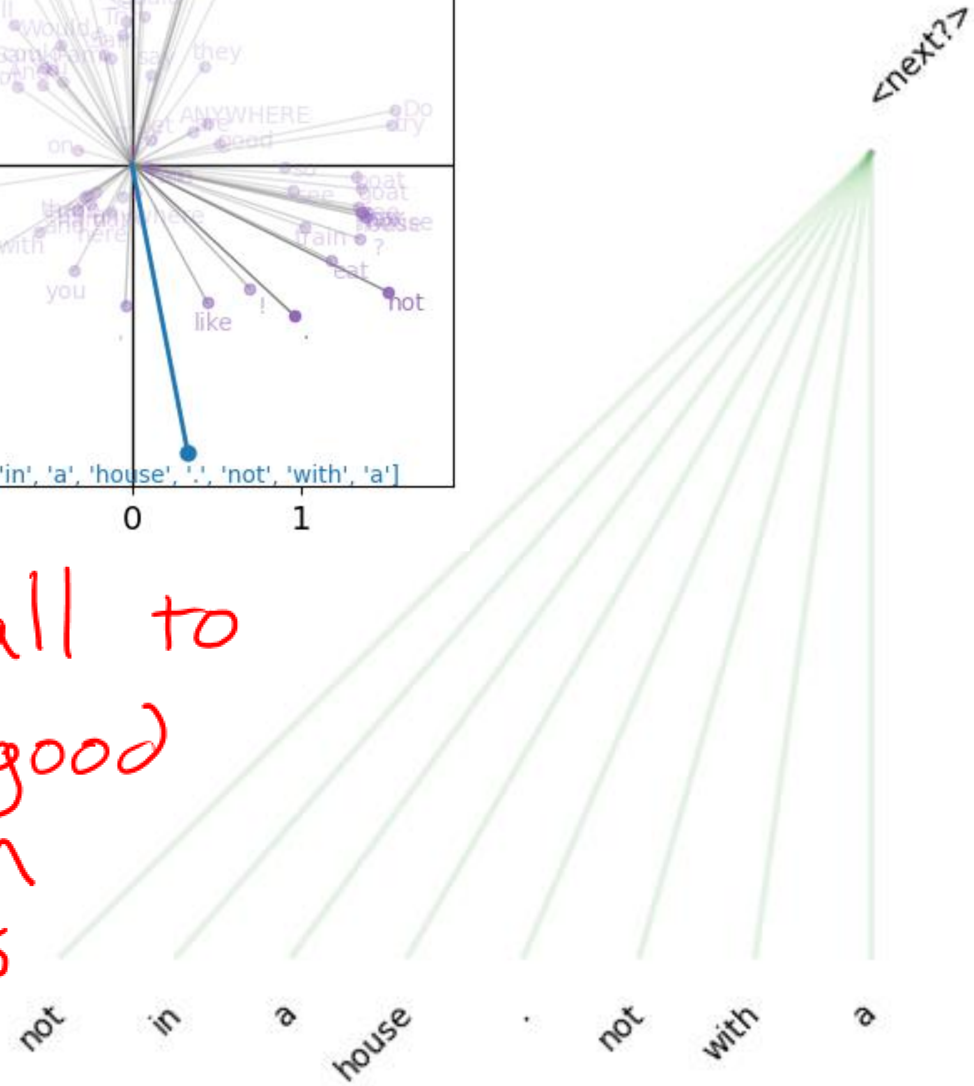
2-D embedded space

1 attention layer

1 attention head (think channel)







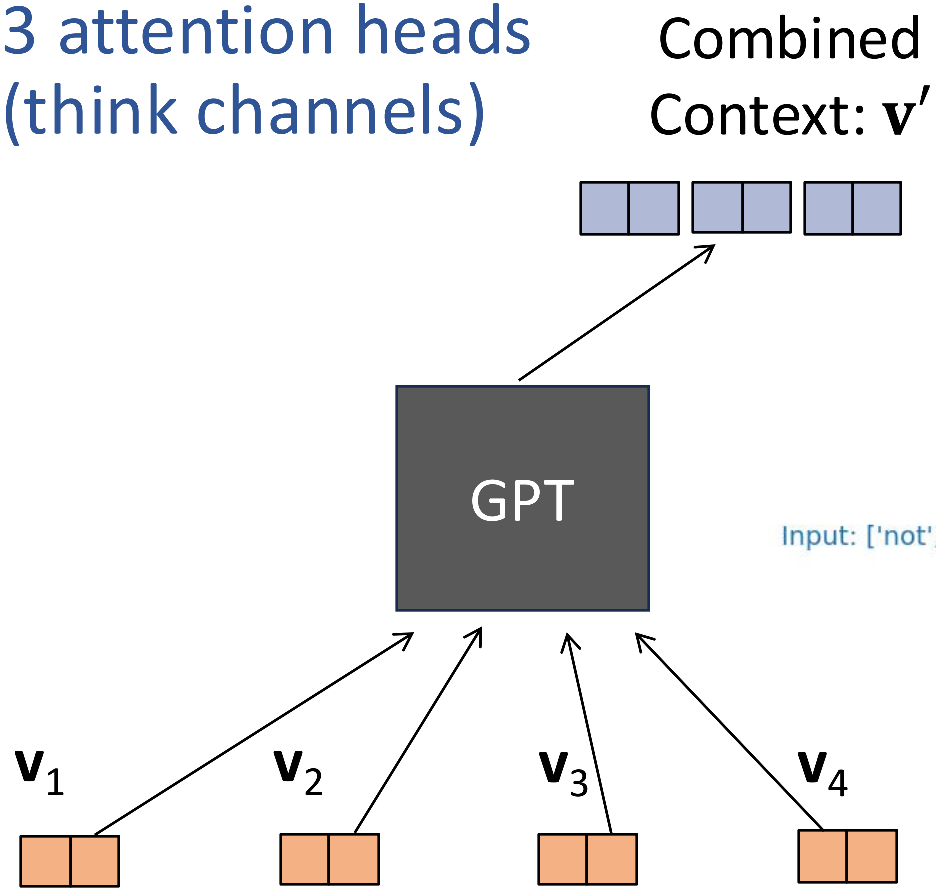
Too small to learn good attention weights

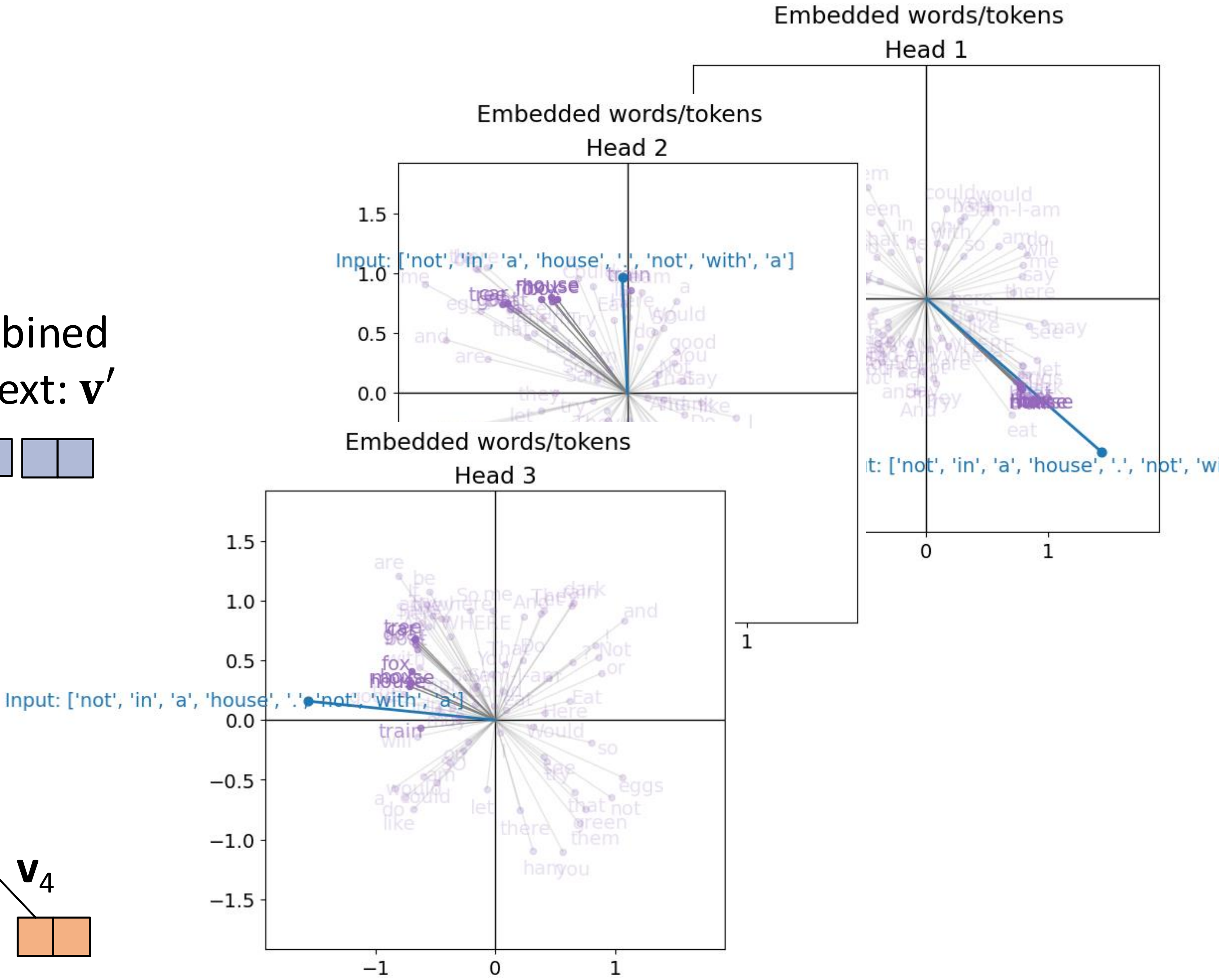
MinGPT Pico

2-D embedded space

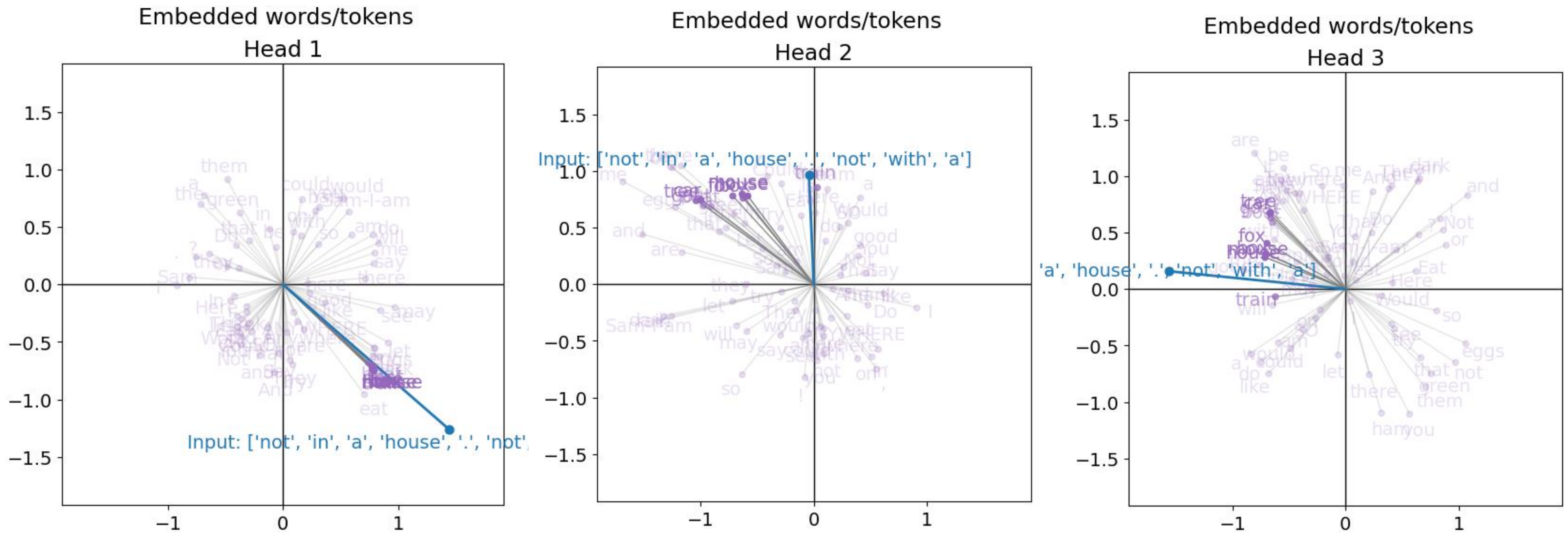
3 attention layer

3 attention heads (think channels)



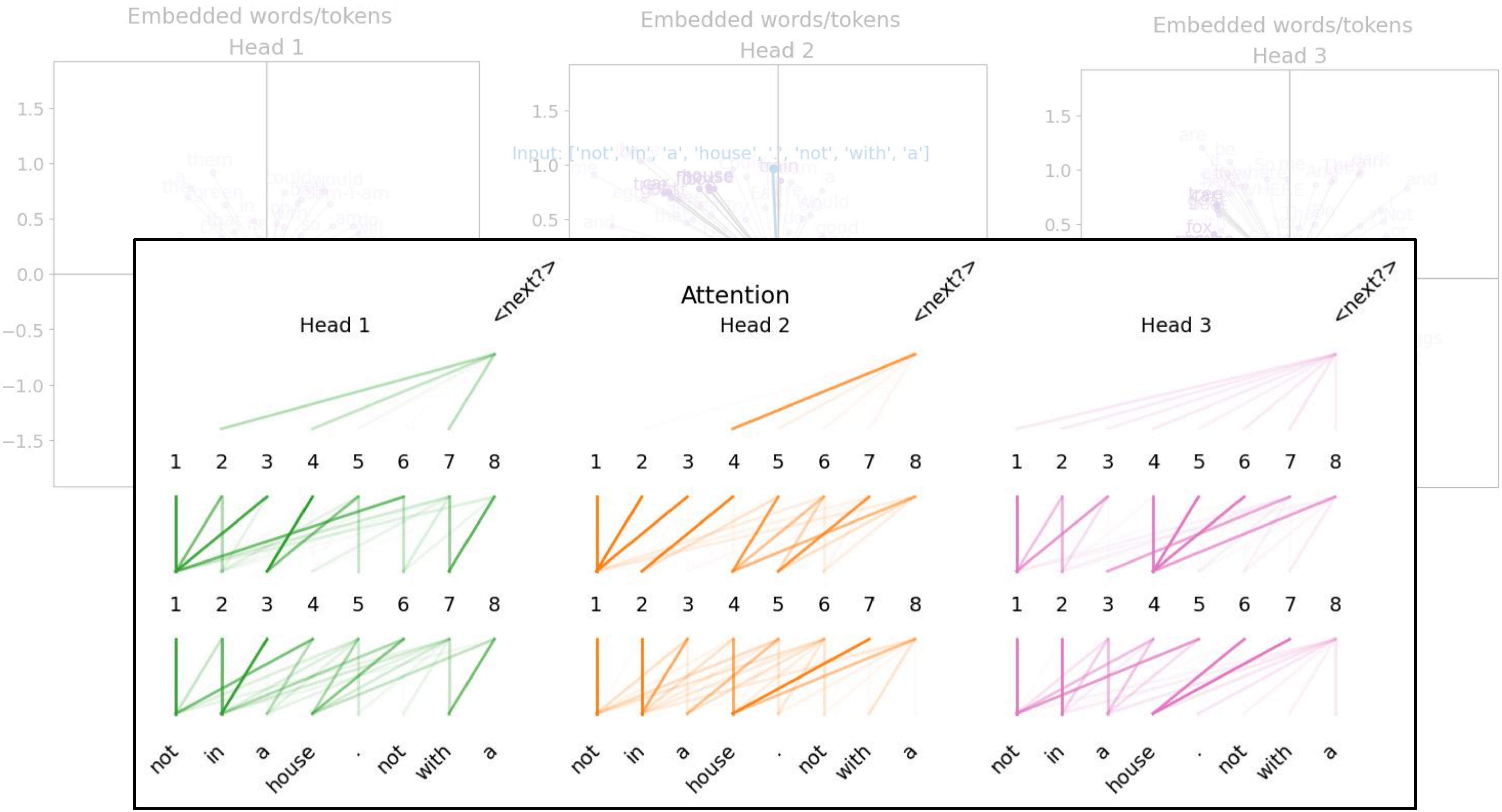


MinGPT Pico: Output embedded space - 3 heads



Three heads allows more room to learn diffent feature representations

MinGPT Pico: Attention Weights – 3 layers, 3 heads

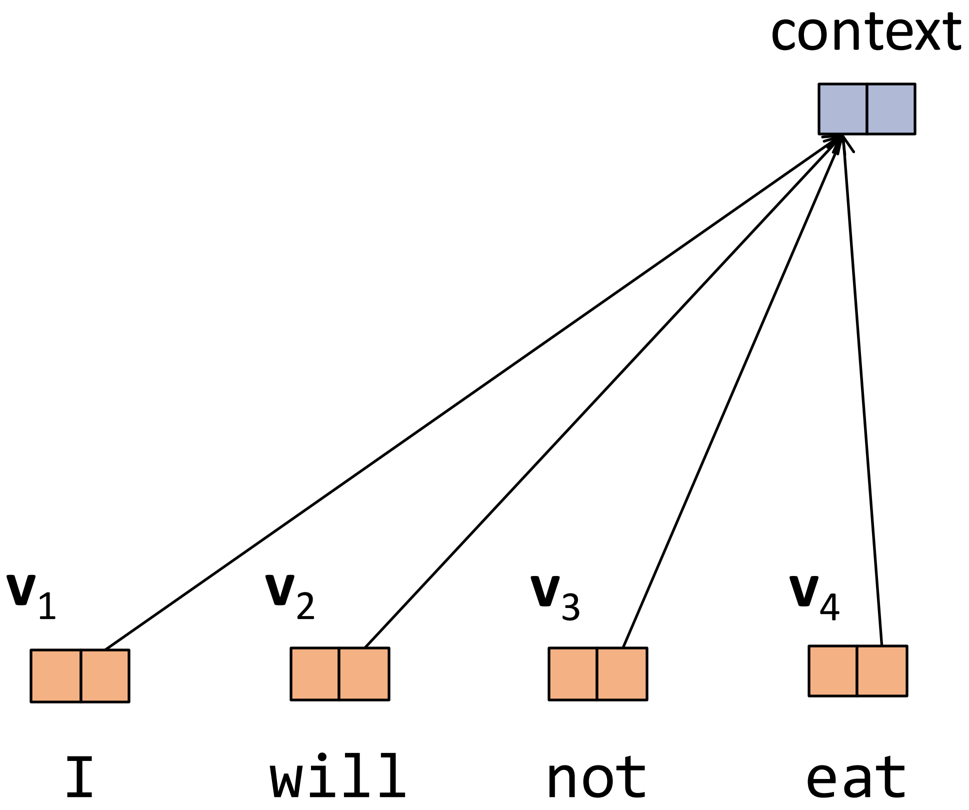


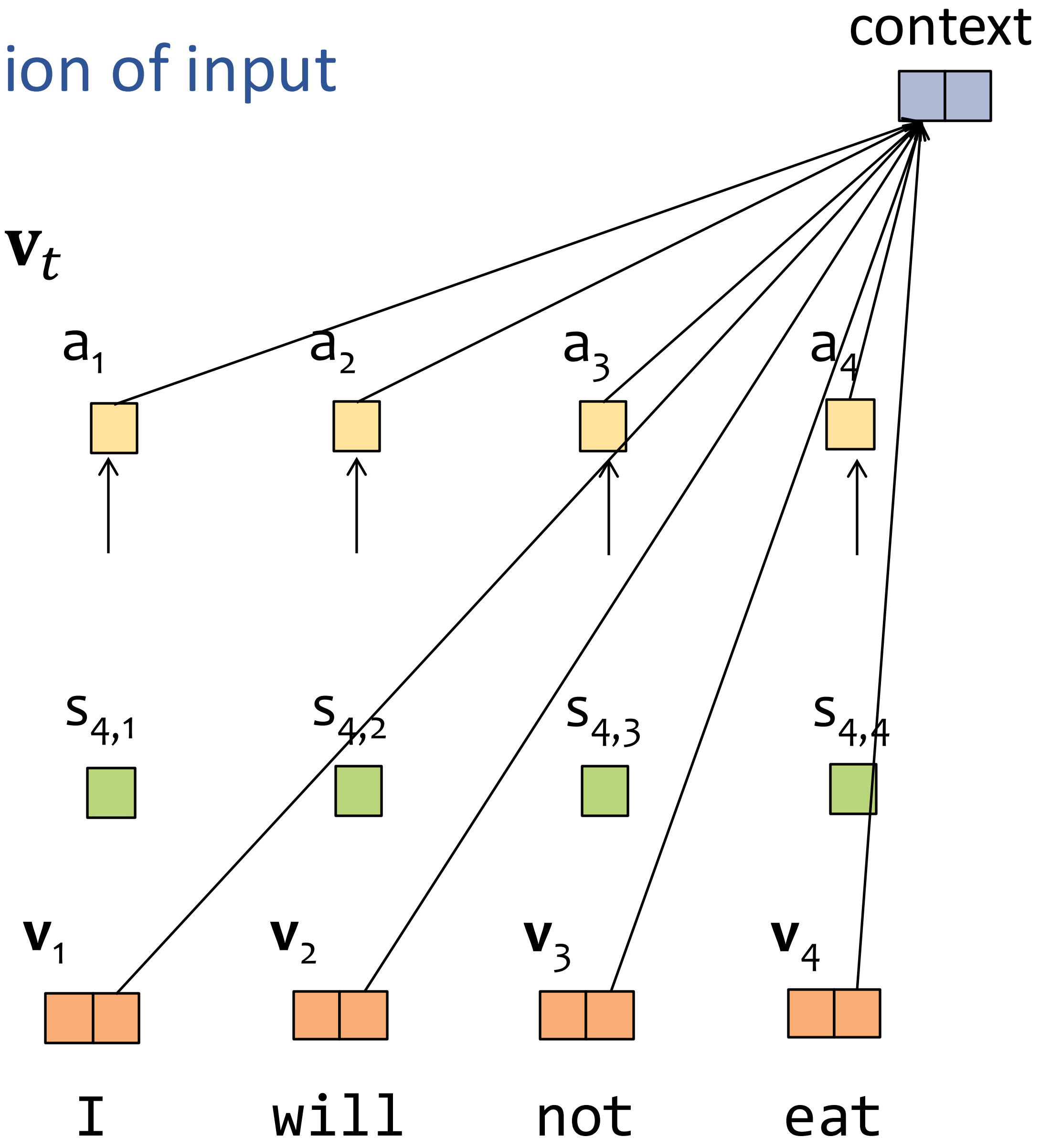
Learn to pay attention!

We can do better than uniform combination of input

$$\mathbf{v}' = \sum_{t=1}^{T} \frac{1}{T} \mathbf{v}_t$$

$$\mathbf{v}' = \sum_{t=1}^{T} a_t \mathbf{v}_t$$





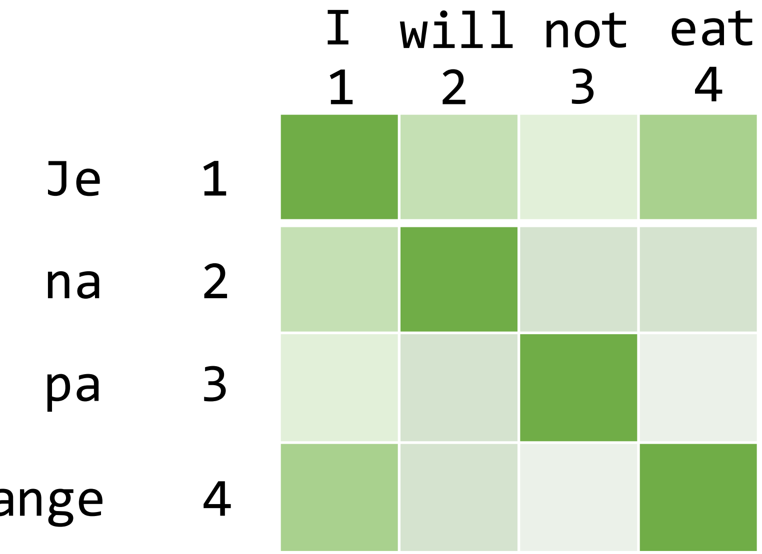
Learn to pay attention!

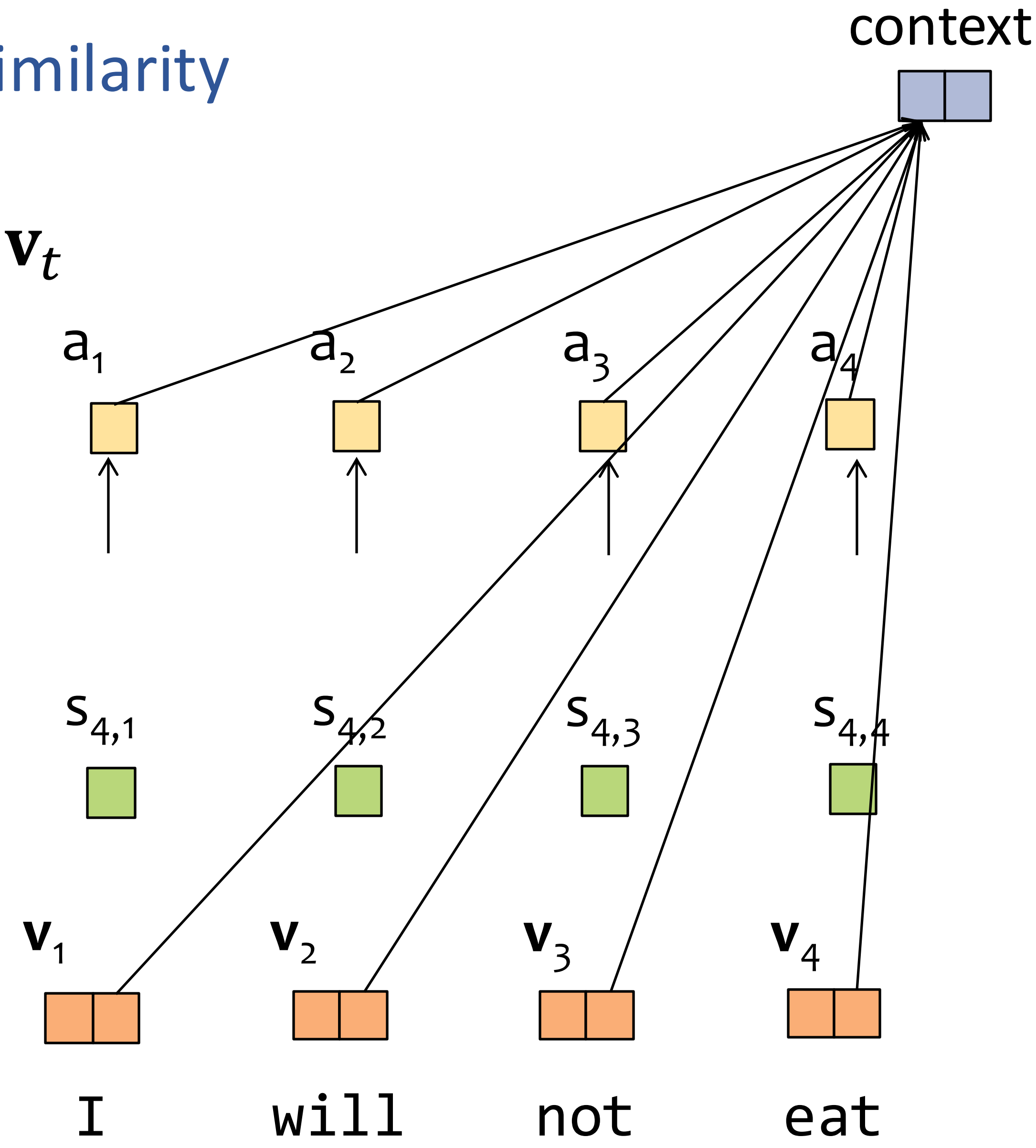
If only we had a way to measure vector similarity

$$\mathbf{v}' = \sum_{t=1}^{T} a_t \mathbf{v}_t$$

Cosine similarity matrix!

$$S = VV^T$$





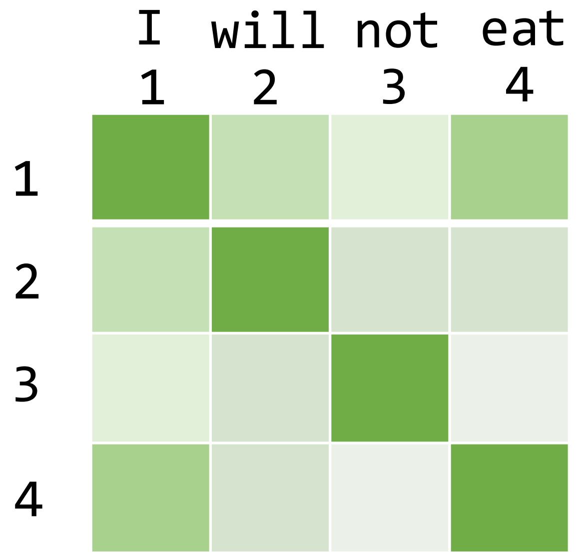
Learn to pay attention!

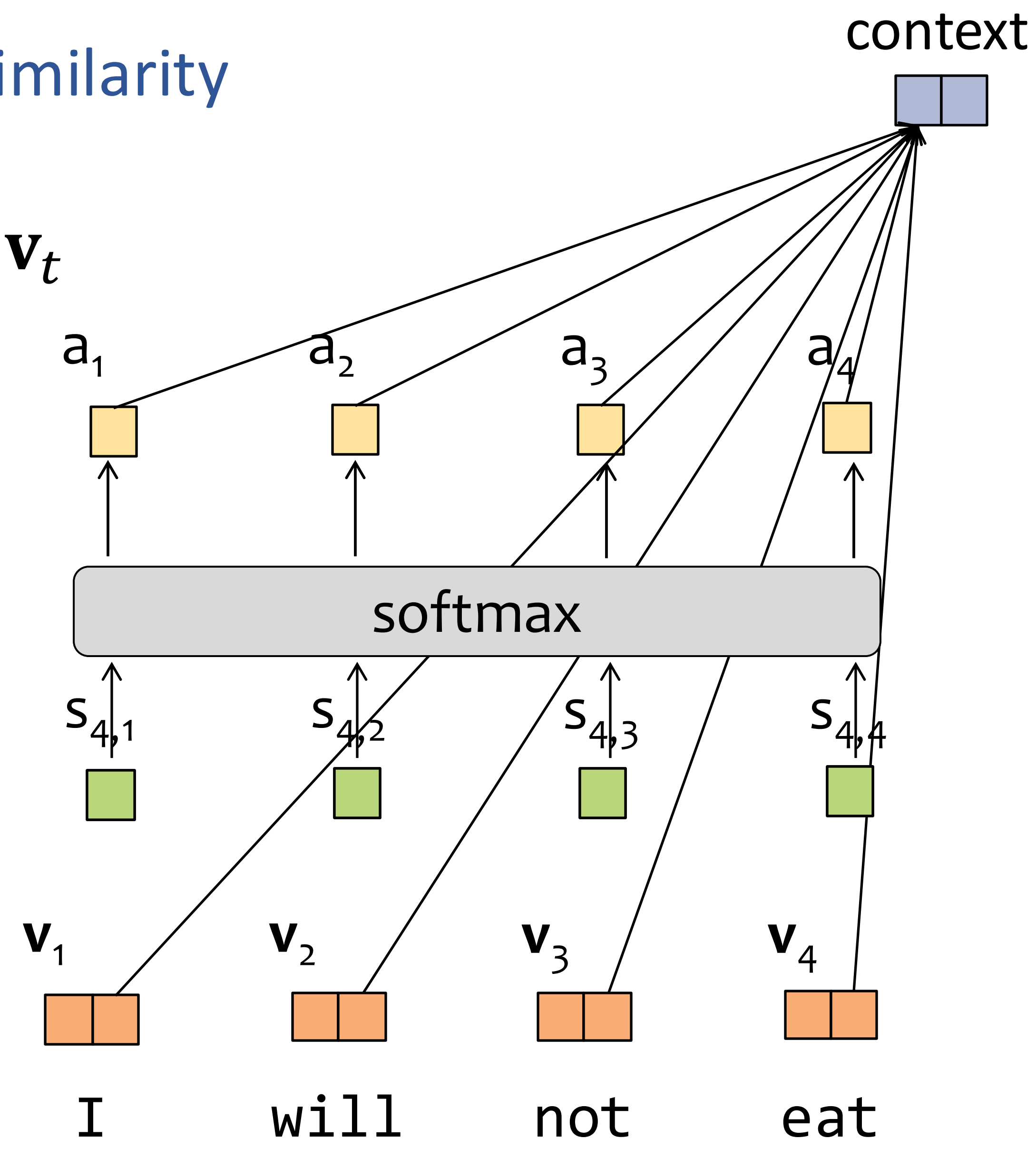
If only we had a way to measure vector similarity

$$\mathbf{v}' = \sum_{t=1}^{T} a_t \mathbf{v}_t$$

Cosine similarity matrix!

$$S = VV^T$$

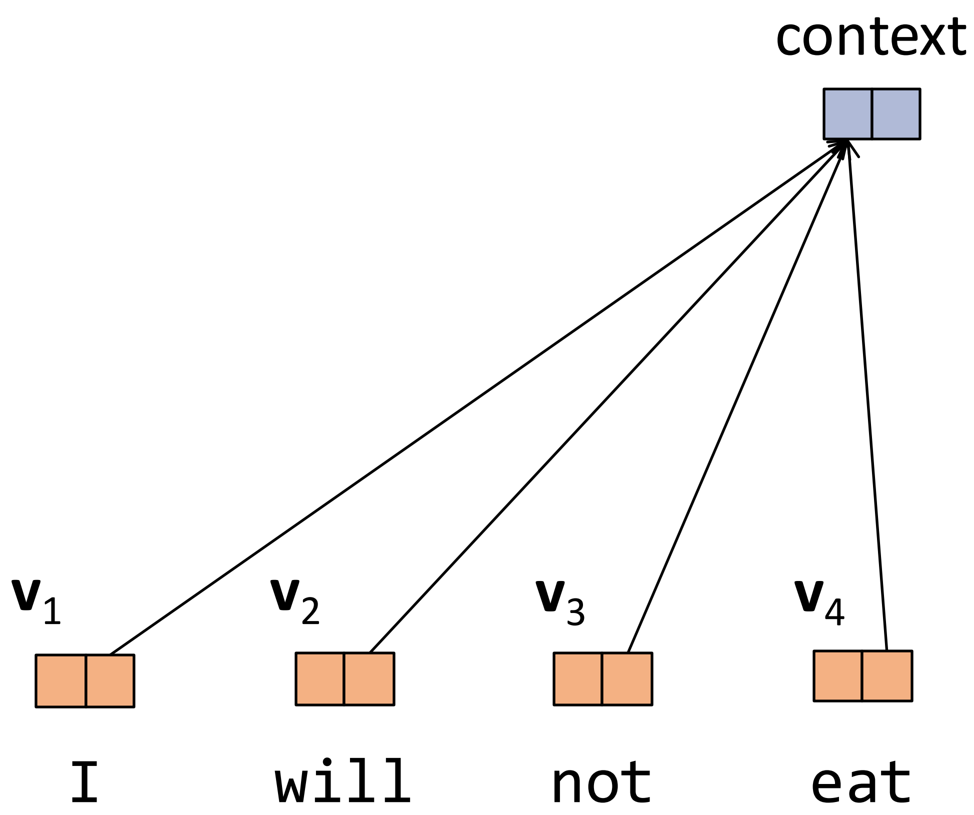


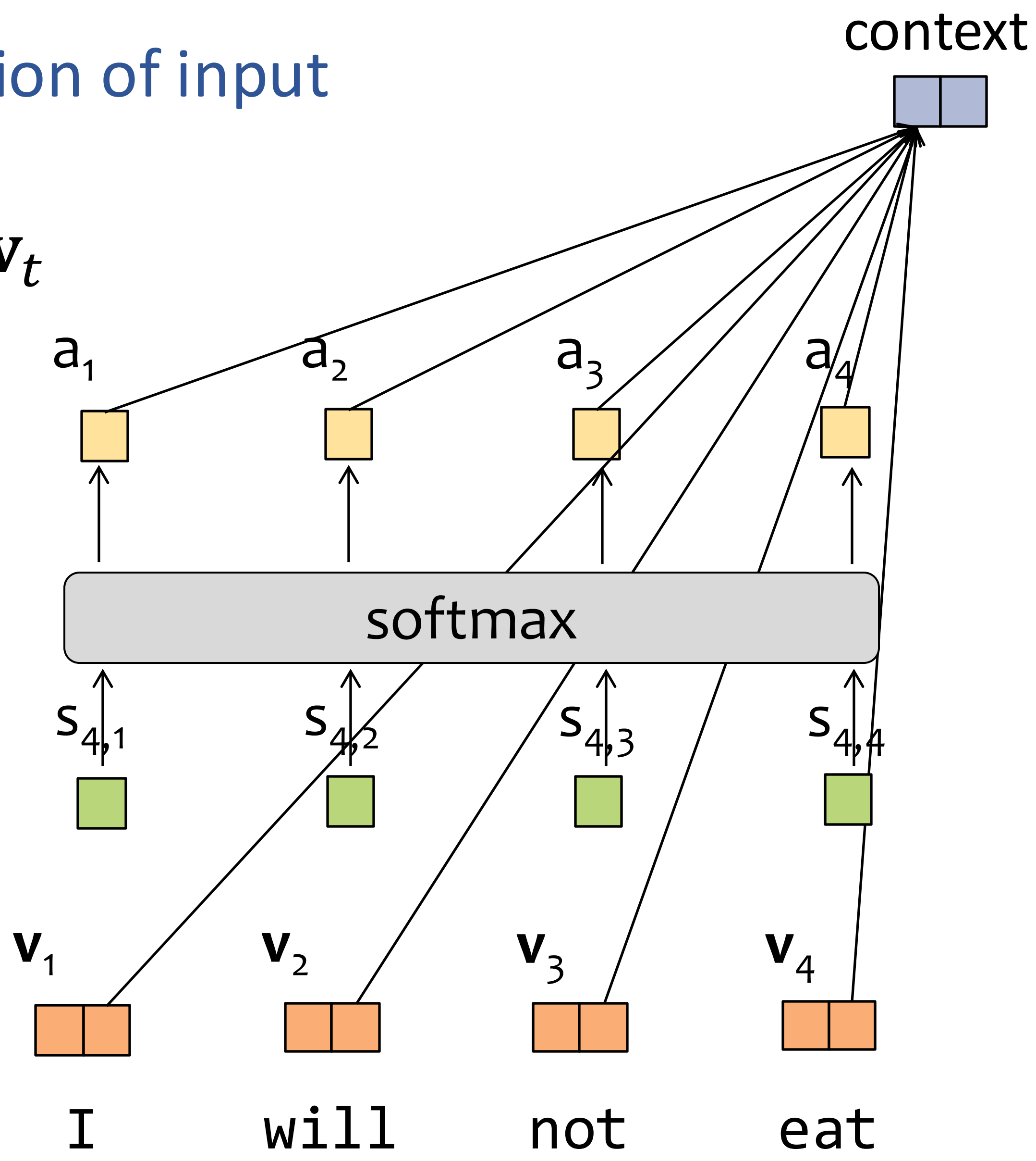


Learn to pay attention!

We can do better than uniform combination of input

$$\mathbf{v} = \sum_{t=1}^{T} \frac{1}{T} \mathbf{v}_t \qquad \mathbf{v} = \sum_{t=1}^{T} a_t \mathbf{v}_t$$





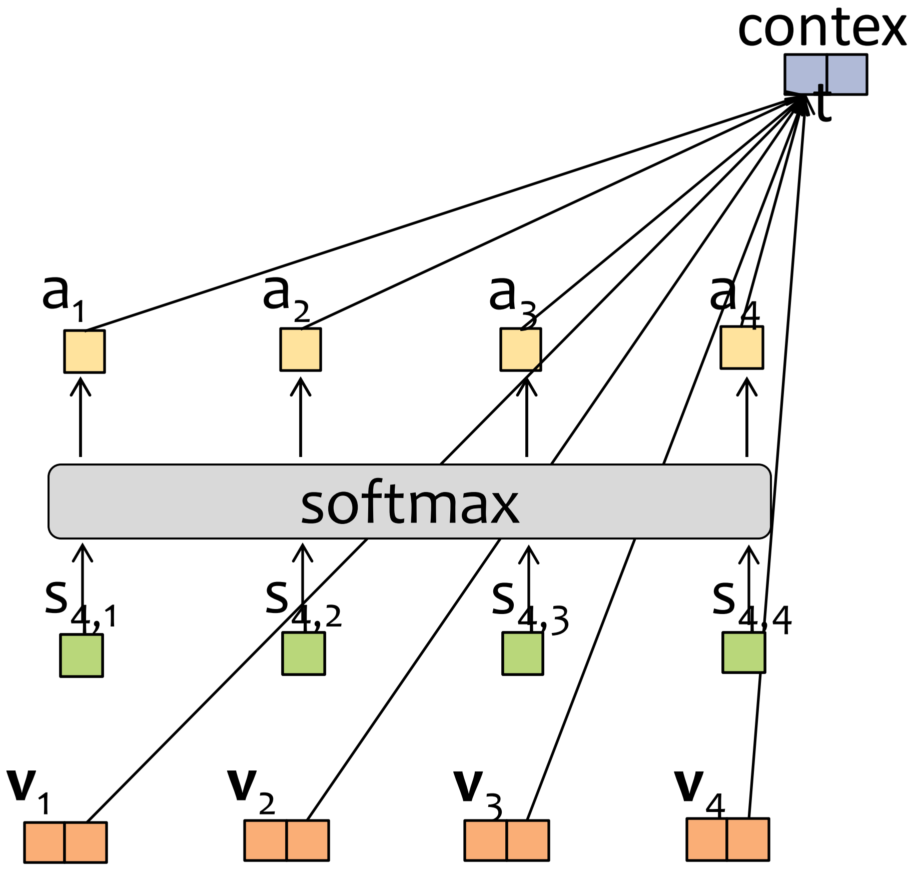
Learn to pay attention!

But...there is an issue with just doing VV^T ☹

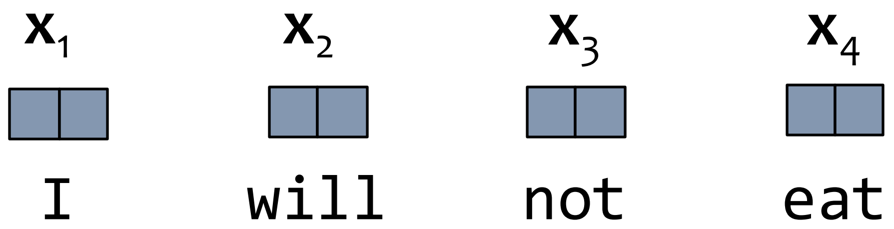
We're really just comparing input to input

→ Symmetric with strong diagonal ☹

$$S = VV^T$$



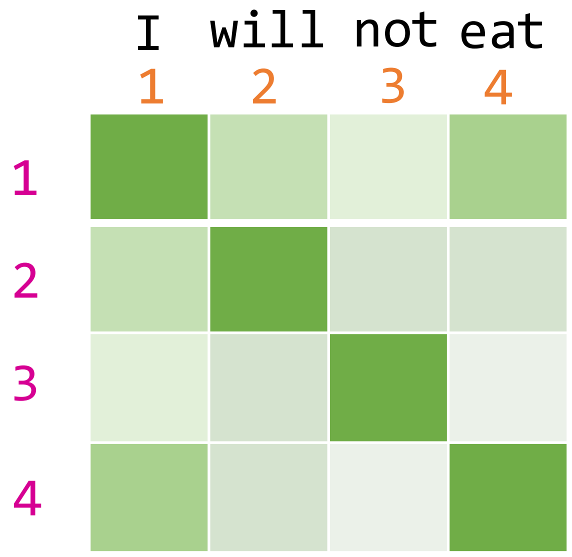
$$V = XW_V$$



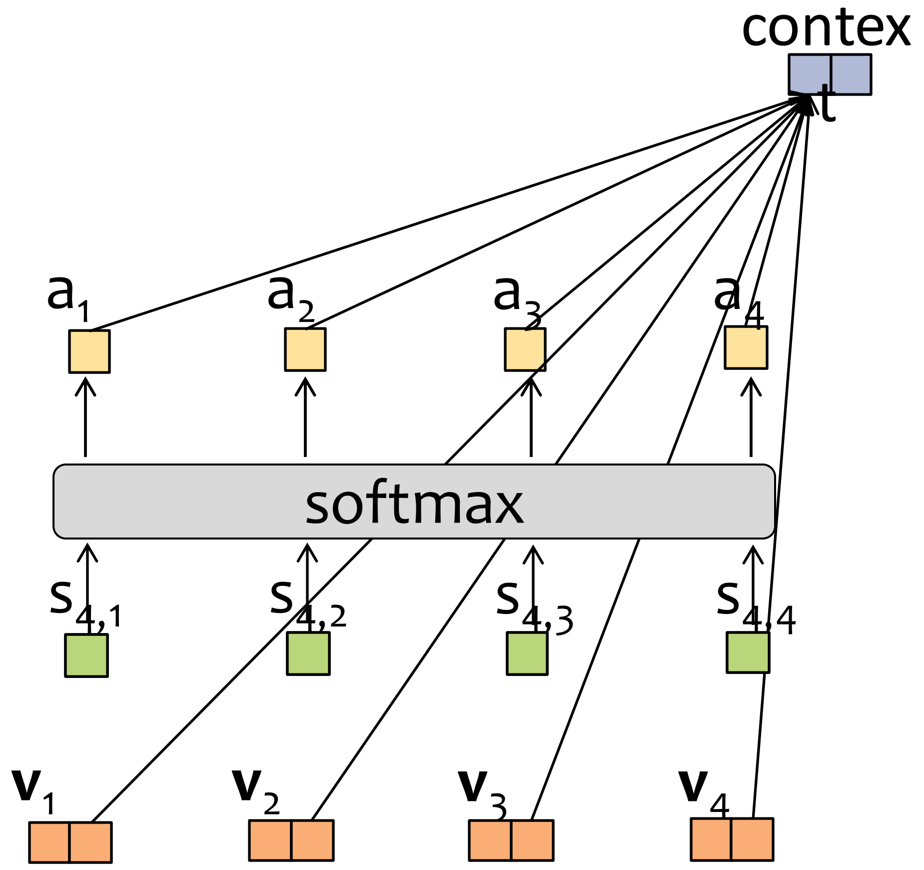
Learn to pay attention!

Instead learn a query vectors $\mathbf{q}_t$ to represent the output

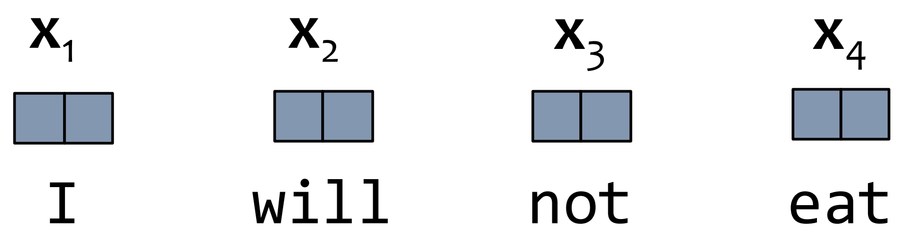
$$S = QV^T$$



$$Q = XW_Q$$



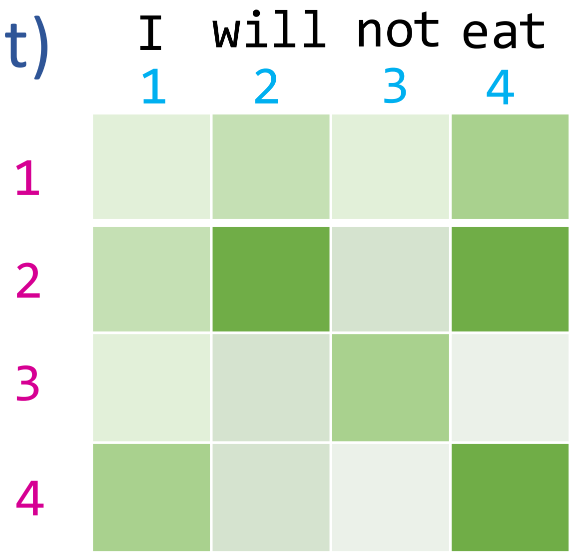
$$V = XW_V$$



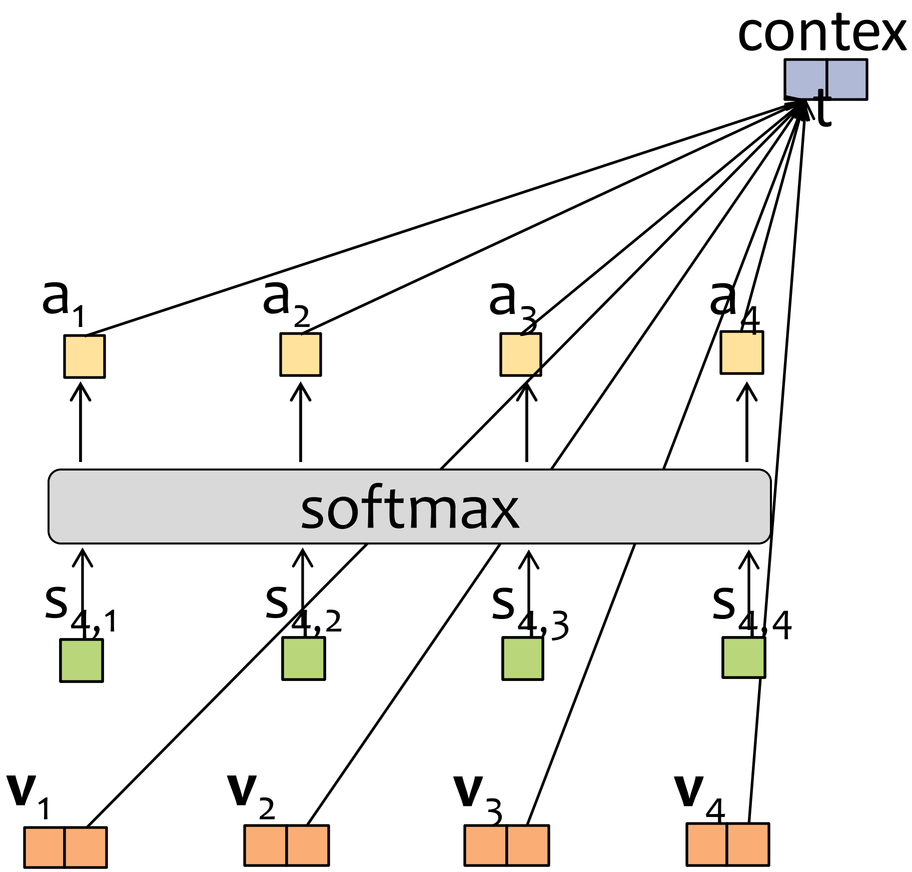
Learn to pay attention!

Instead learn a query vectors $\mathbf{q}_t$ to represent the output

(And also $\mathbf{k}_t$ for the input)



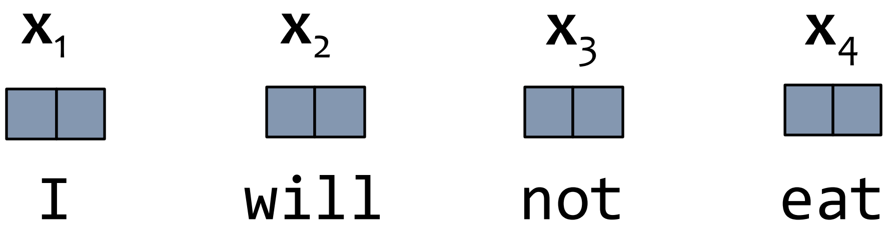
$$S = QK^T/\sqrt{d_k}$$



$$Q = XW_Q$$

$$K = XW_K$$

$$V = XW_V$$



Learn to pay attention!

Instead learn a query vectors $\mathbf{q}_t$ to represent the output

(And also $\mathbf{k}_t$ for the input)

Attention:

Query, Key, Value

$$Q = XW_Q$$

$$K = XW_K$$

$$V = XW_V$$

$$S = QK^T / \sqrt{d_k}$$