# Attention & encoding

ground-work for Transformers

## Next token prediction

## Self-supervision: autoregression

- Pretend that we don’t know the ending.
- Again, ground truth for free.

Input:
*The city of Warsaw is the capital and largest city of*

Expected output:
*Poland*

## Autoregressive models - using full sequence

- *The -> city*
- *The city -> of*
- *The city of -> Warsaw*
- *The city of Warsaw -> is*
- *The city of Warsaw is -> the*
- *The city of Warsaw is the -> capital*
- *The city of Warsaw is the capital -> and*
- *The city of Warsaw is the capital and -> largest*
- *The city of Warsaw is the capital and largest -> city*
- *The city of Warsaw is the capital and largest city -> of*
- *The city of Warsaw is the capital and largest city of -> Poland*

## Generating the next token

- A model generates conditional probability distribution over the next token, e.g.,
  P(*“Poland”* | *“The city of Warsaw is the capital of”).*

- Loss function: cross-entropy

Note that for tokens $w_1, w_2, \ldots, w_n$

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, \ldots, w_{i-1})$$

Define perplexity as $\frac{\sum_{i=1}^{n} \log(P(w_i | w_1, \ldots, w_{i-1}))}{n}$

Note that for tokens $w_1, w_2, \ldots, w_n$

$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, \ldots, w_{i-1})$

Define perplexity as $\frac{\sum_{i=1}^{n} \log(P(w_i | w_1, \ldots, w_{i-1}))}{n}$

- P(*The* | *<START>*)
- P(*city* | *The*)
- P(*of* | *The city*)
- P(*Warsaw* | *The city of*)
- P(*is* | *The city of Warsaw*)
- P(the | *The city of Warsaw is*)
- P(capital | *The city of Warsaw is the*)

## From text to vectors

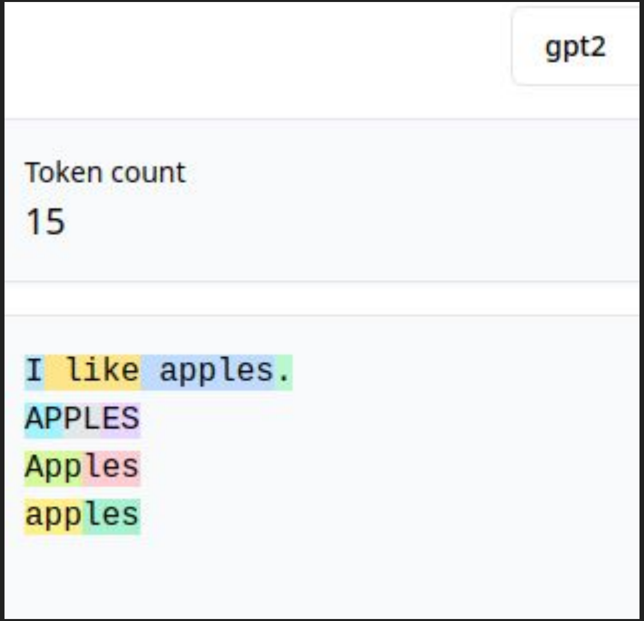
tokenization + embedding + unembedding

## Tokenization

- Extremes:
  - Characters as tokens.
  - Words as tokens.

- Typically something in between
  - tens of thousands of tokens.

- https://tiktokenizer.vercel.app/

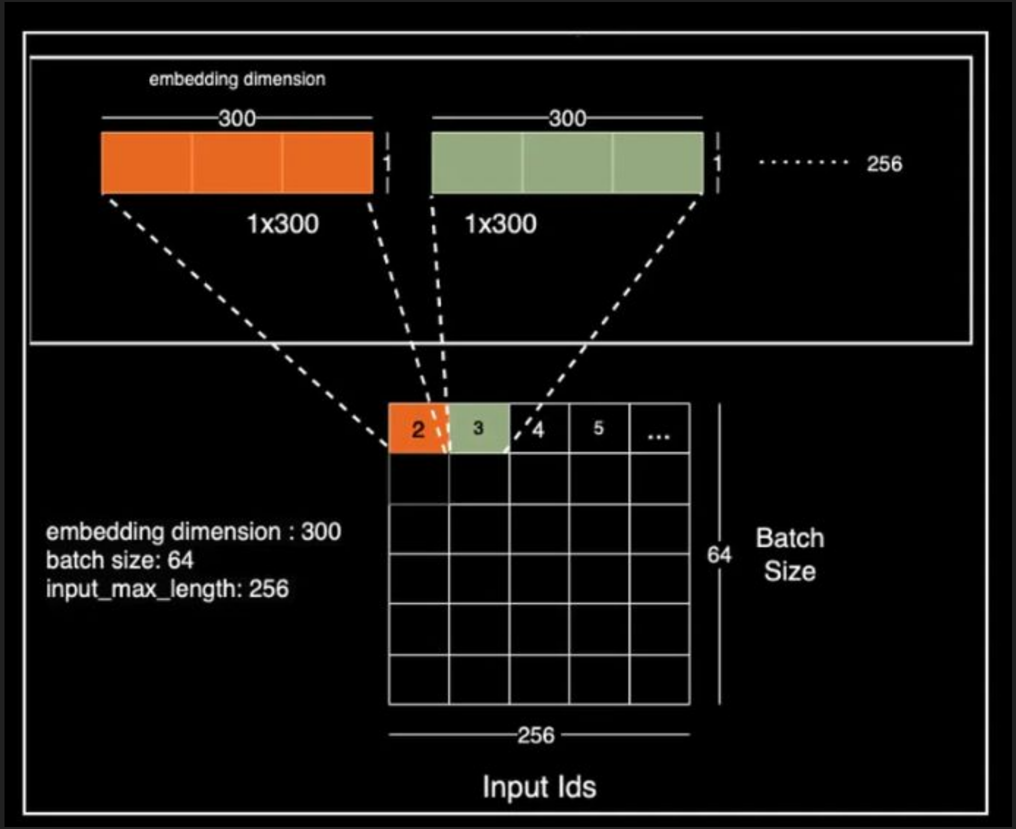


Token count
15

```
I like apples.
APPLES
Apples
apples
```

# Embedding layer

- Embedding is a matrix vocab_size x $d_{model}$
- A token with index $w_i$ is mapped to `embedding[`$w_i$`]`
- Alternatively `one_hot(w_i) * embedding`



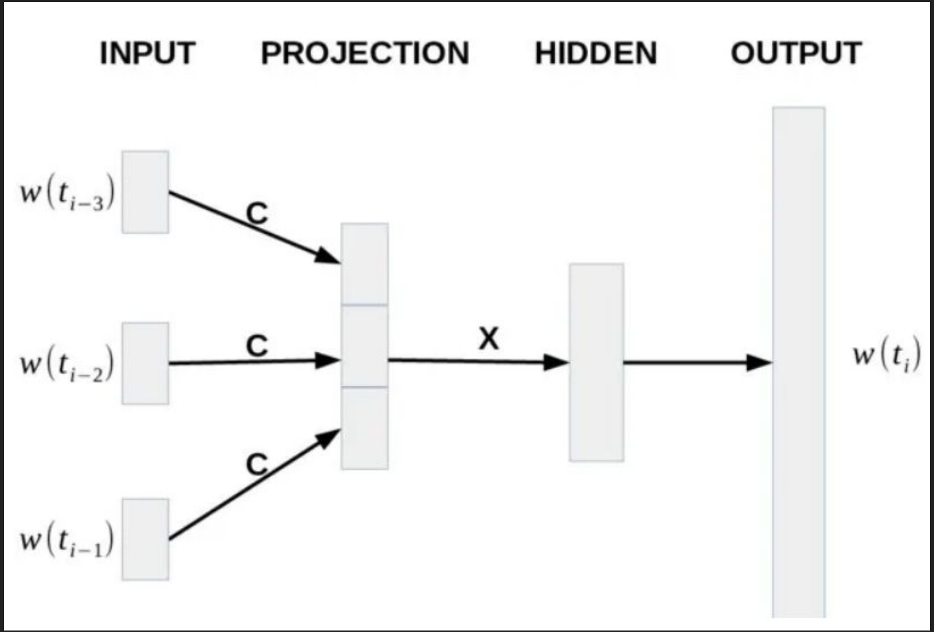
source

## Unembedding layer

- Linear projection from $d_{model}$ to vocab_size
- Matrix of size $d_{model}$ x vocab_size
- To get a distribution over tokens we typically apply softmax

# 2003 - MLP for next token prediction

- Word Embeddings
- Feedforward:
  - Input: concatenate n previous word vectors
- Softmax Output over all possible tokens.



## Toy example - zero layer transformer

- Input token -> embedding -> unembedding -> predicted_token

- The model is a product of:
  - Embedding matrix $W_E$ (vocab x $d_{model}$)
  - Unembedding matrix  $W_U$ ($d_{model}$ x vocab)

- What is the $W_E W_U$ that minimizes train loss?

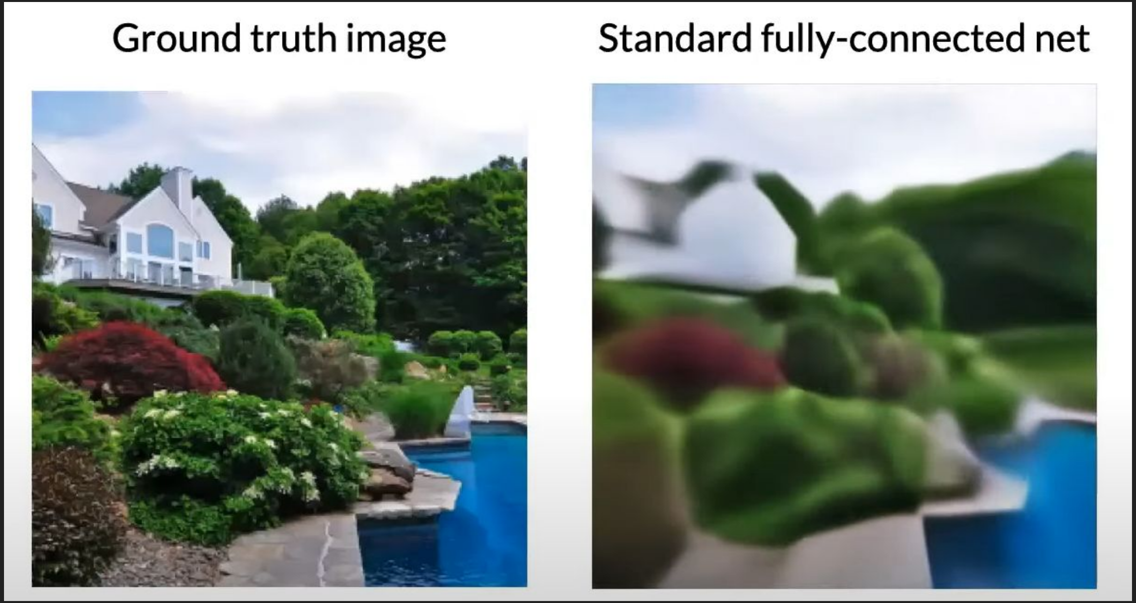
## Positional encoding

## Toy problem: memorize a 2D image

- Consider a fixed RGB image.
- We want to train a model model (neural network) for this specific image.
- Input: pixel coordinates (x, y)
  Expected output: pixel values (RGB)
- Old exam task.

## Toy problem: memorize a 2D image

Surprise: the regular MLP is unable to learn the image.

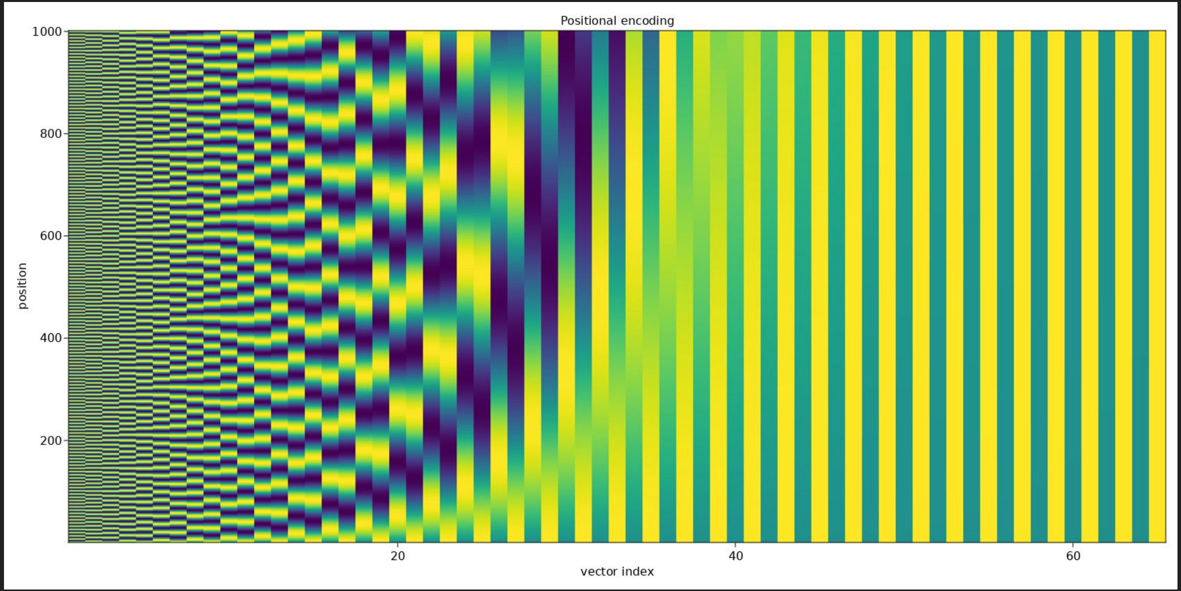


source

# Positional encoding

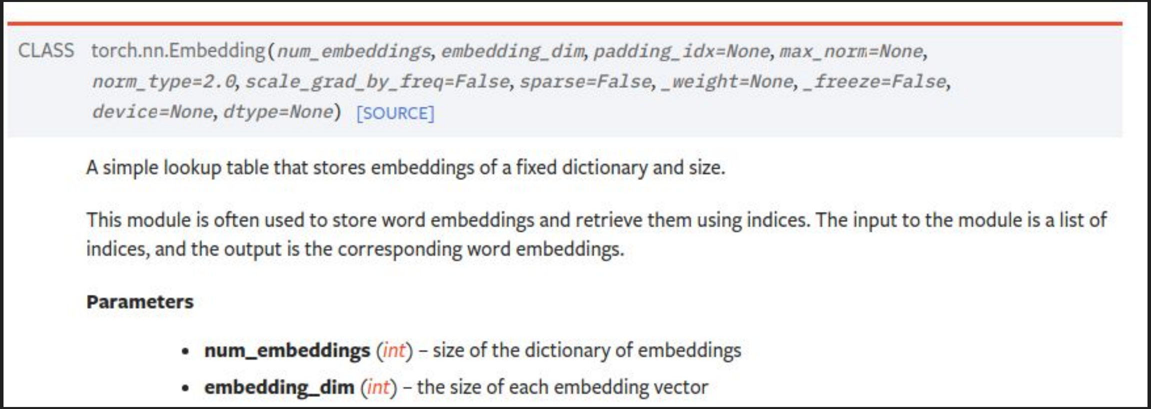
- Instead of raw integer x, use a special vector, where:
  - d - dimensionality of the encoding
  - N is bigger than max position (e.g., N=1000)

$$p_x = \begin{bmatrix} \sin(t/1000^{2\cdot 1/d}) \\ \cos(t/1000^{2\cdot 1/d}) \\ \sin(t/1000^{2\cdot 2/d}) \\ \cos(t/1000^{2\cdot 2/d}) \\ \vdots \\ \sin(t/1000^{2\cdot \frac{d}{2}/d}) \\ \cos(t/1000^{2\cdot \frac{d}{2}/d}) \end{bmatrix}$$



# Learned encoding

- For each possible value of t use an embedding vector (initially random).



CLASS torch.nn.Embedding(*num_embeddings*, *embedding_dim*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*, *_weight=None*, *_freeze=False*, *device=None*, *dtype=None*) [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

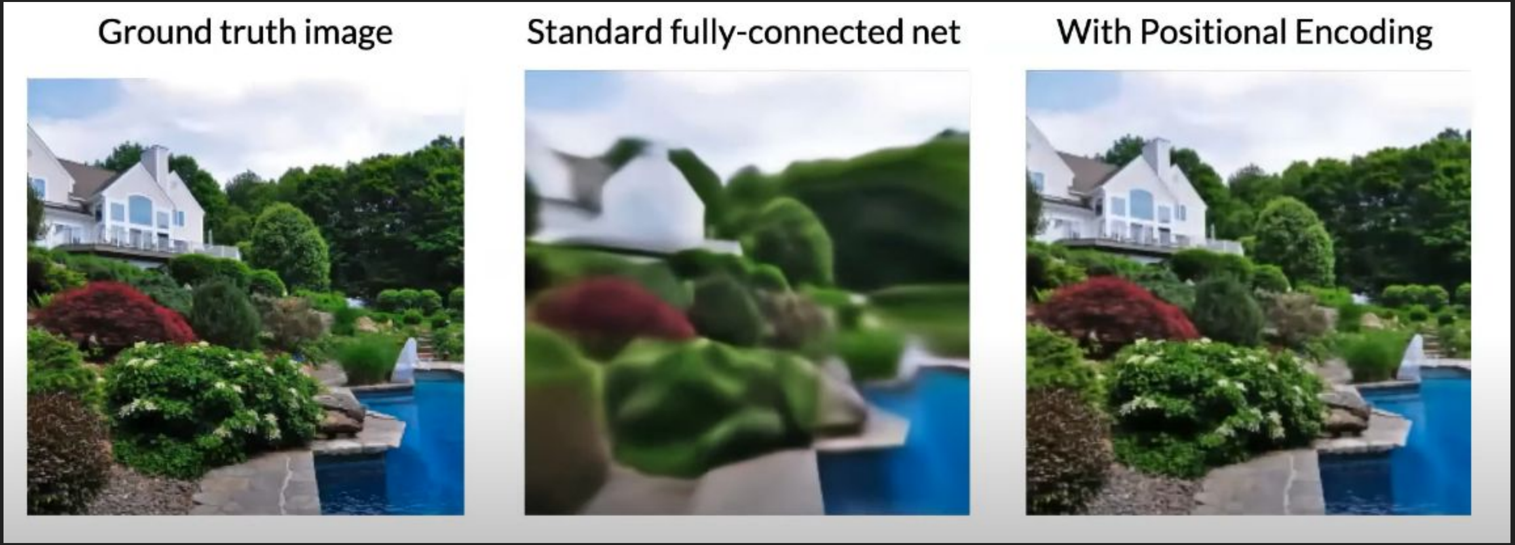
This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

**Parameters**

- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector

## Toy problem: memorize a 2D image

With positional or learned encoding it works!



source

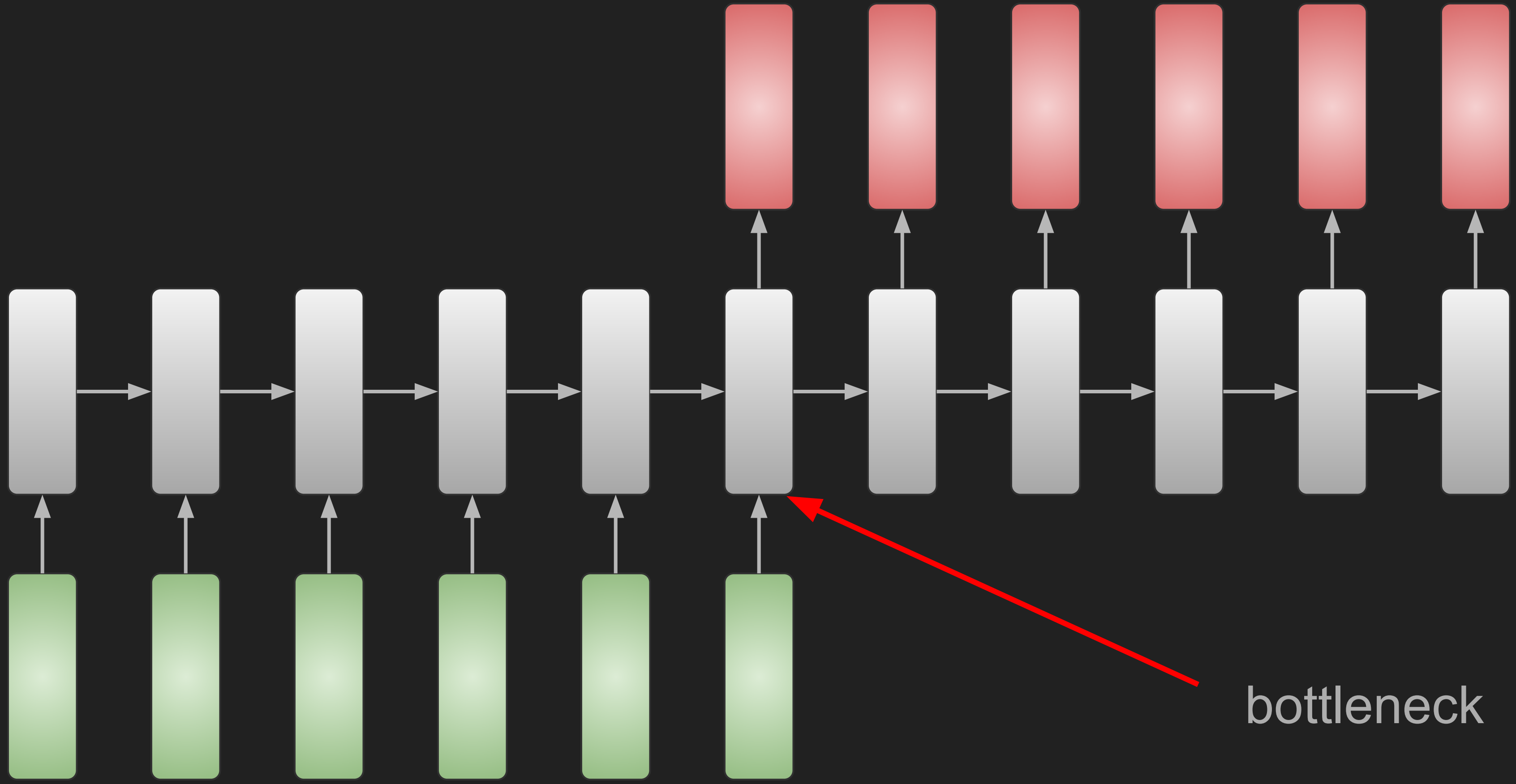
## Bottleneck and attention

## Bottleneck in RNNs

Consider the following setting:

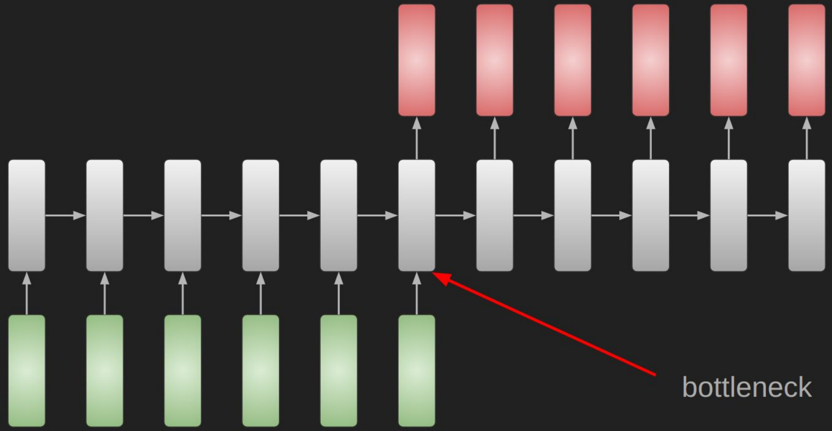
- The network reads a paragraph of an article, then reads a question and has to produce an answer.

- The network reads a sentence to be translated, then has to produce the answer.


bottleneck

## Attention

- Attention - circumvents the problem.
- The model is allowed to look back far (thousands of tokens) anytime.
- More on this next week.



## Soft attention

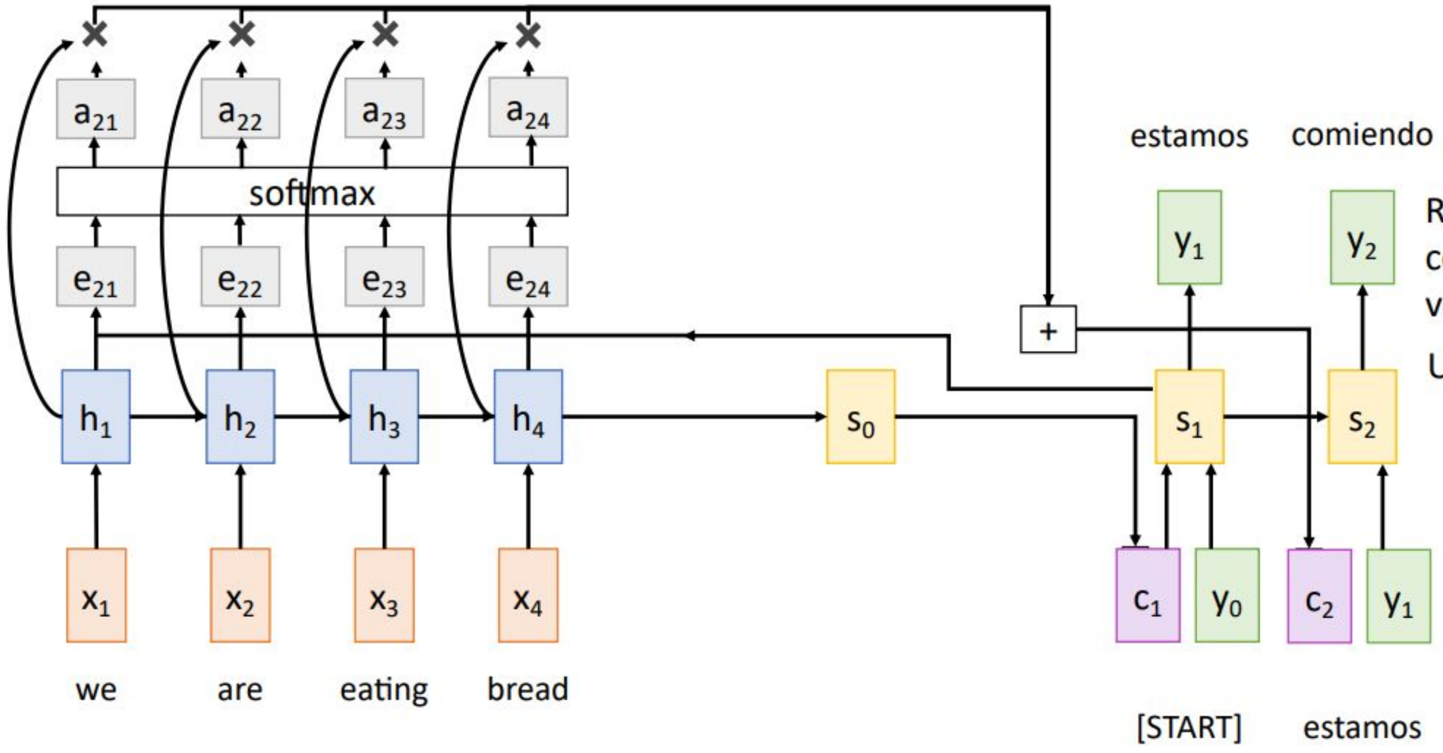
Given the current state s, and a set of vectors $\{h_1,...,h_n\}$:

1. For each i in {1,...,n} compute
   - $e_i := \langle s, h_i \rangle$.
2. Apply softmax on $\{e_1,...,e_n\}$, to obtain a distribution $\{a_i\}$.
3. Compute a linear combination of vectors $\{h_i\}$ weighted by $\{a_i\}$
   - $x = \text{sum}_i\ a_i * h_i$

We say that $a_i$ are attention weights.

<use board>

# Sequence-to-Sequence with RNNs **and Attention**



Repeat: Use $s_1$ to compute new context vector $c_2$

Use $c_2$ to compute $s_2$, $y_2$

Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

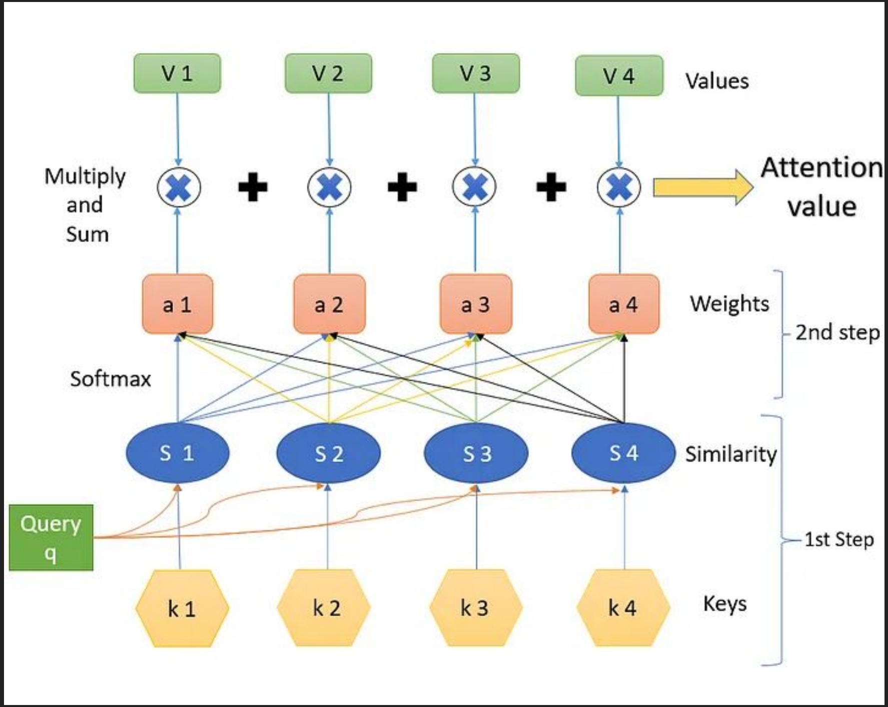
## Soft attention generalized

In the shown example we used the same vectors $\{h_i\}$ to compute the distances, as well as in the weighted sum (final result). We can generalize this by using different vectors for keys $k_i$, values $v_i$ and query q.

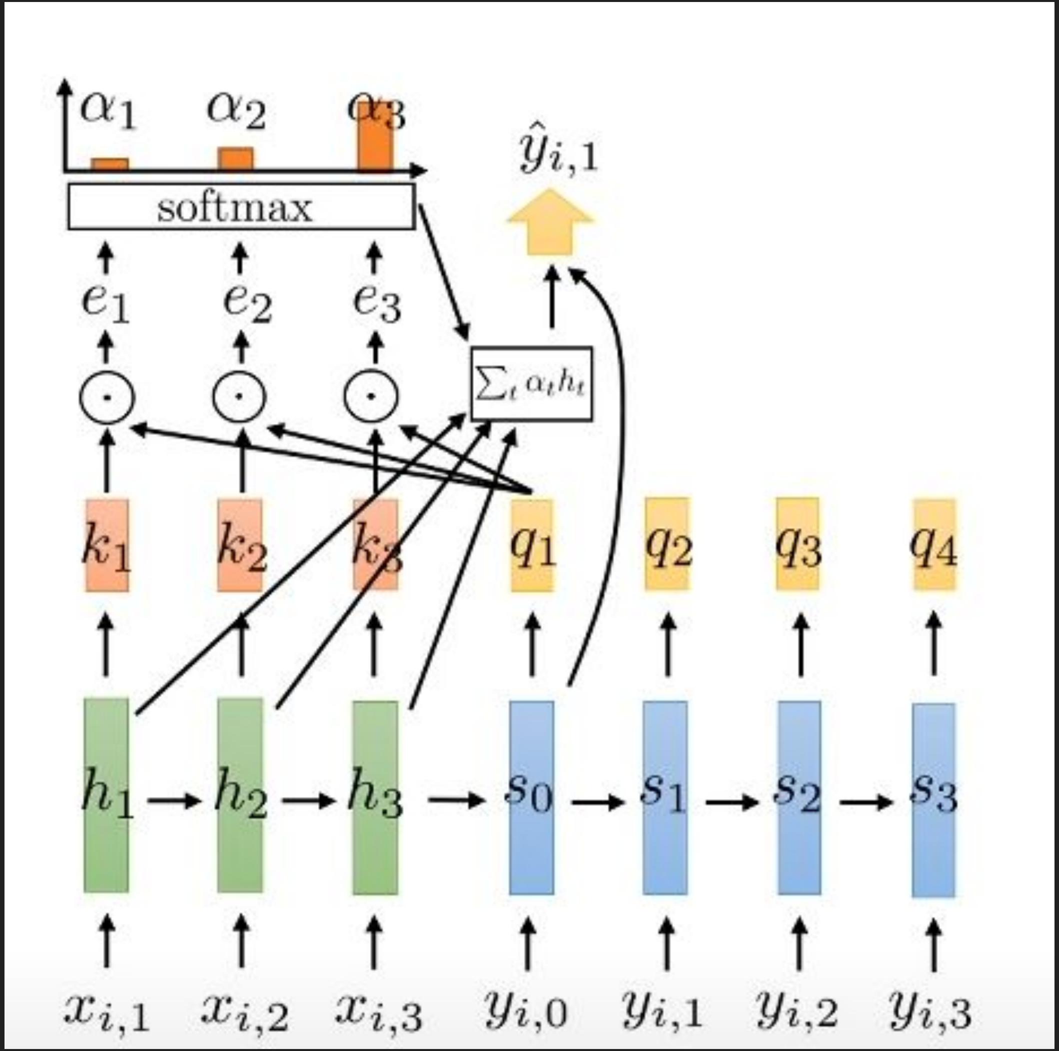
1. For each i in {1,...,n} compute
   - `e`$_i$ `:= <q,k`$_i$`>`.
2. Apply softmax on $\{e_1,\ldots,e_n\}$, to obtain a distribution $\{a_i\}$.
3. Compute a linear combination of vectors $\{v_i\}$ weighted by $\{a_i\}$
   - `x = sum`$_i$ `a`$_i$ `* v`$_i$

# Soft attention generalized

1. For each i in {1,...,n} compute
   - $e_i := \langle q, k_i \rangle$.
2. Apply softmax on $\{e_1,\ldots,e_n\}$, to obtain a distribution $\{a_i\}$.
3. Compute a linear combination of vectors $\{v_i\}$ weighted by $\{a_i\}$
   - $x = \text{sum}_i\ a_i * v_i$



source

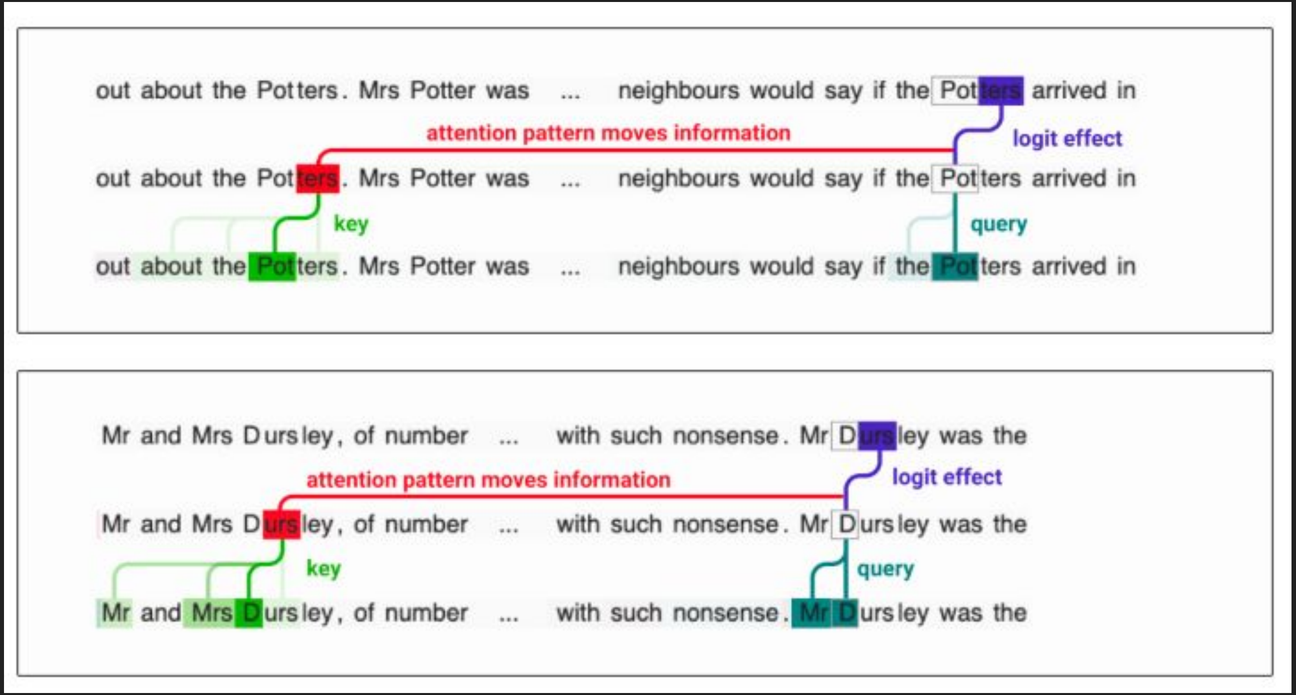

$\alpha_1$
$\alpha_2$
$\alpha_3$
softmax
$\hat{y}_{i,1}$
$e_1$
$e_2$
$e_3$
$\sum_t \alpha_t h_t$
$k_1$
$k_2$
$k_3$
$q_1$
$q_2$
$q_3$
$q_4$
$h_1$
$h_2$
$h_3$
$s_0$
$s_1$
$s_2$
$s_3$
$x_{i,1}$
$x_{i,2}$
$x_{i,3}$
$y_{i,0}$
$y_{i,1}$
$y_{i,2}$
$y_{i,3}$

source

# Induction head in 2-layer attention

- A very interesting phenomenon in 2+-layer attention
- Induction head
  - [A][B] …. [A] → [B]
  - E.g. Michael Jordan … Michael -> Jordan
  - Works for any A and B

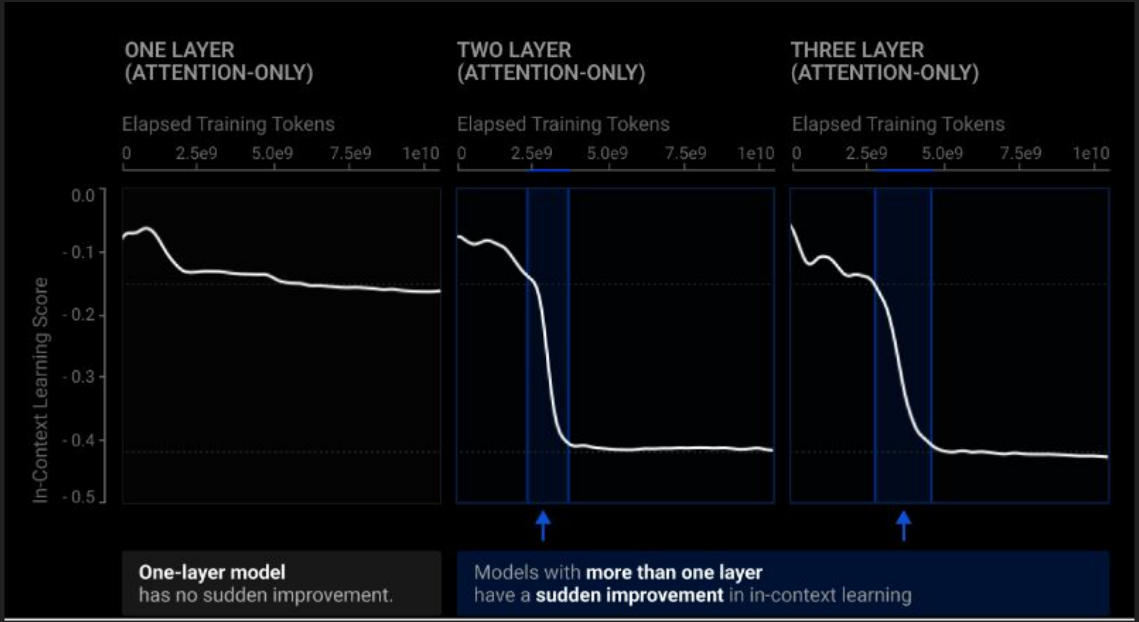
# Induction head


out about the Potters. Mrs Potter was ... neighbours would say if the Potters arrived in
attention pattern moves information
logit effect
out about the Potters. Mrs Potter was ... neighbours would say if the Potters arrived in
key
query
out about the Potters. Mrs Potter was ... neighbours would say if the Potters arrived in
Mr and Mrs Dursley, of number ... with such nonsense. Mr Dursley was the
attention pattern moves information
logit effect
Mr and Mrs Dursley, of number ... with such nonsense. Mr Dursley was the
key
query
Mr and Mrs Dursley, of number ... with such nonsense. Mr Dursley was the

source

## Induction head

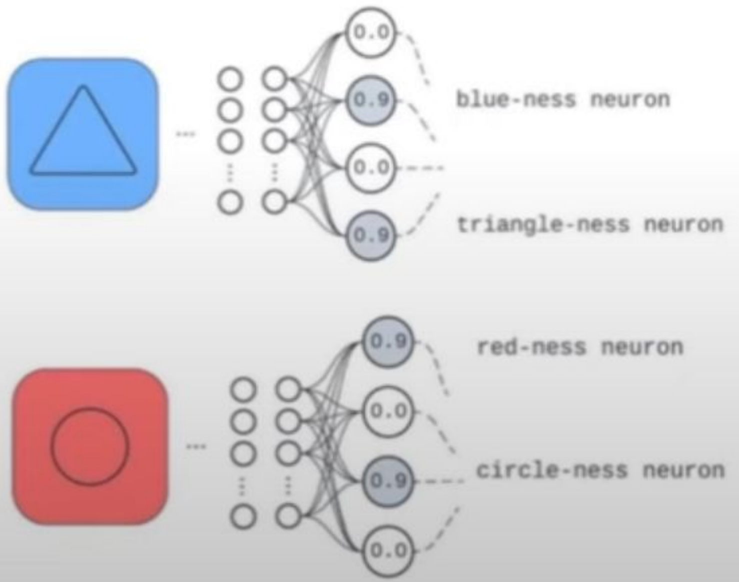
- A very interesting phenomenon in 2+-layer attention
- Not present in  1-layer attention

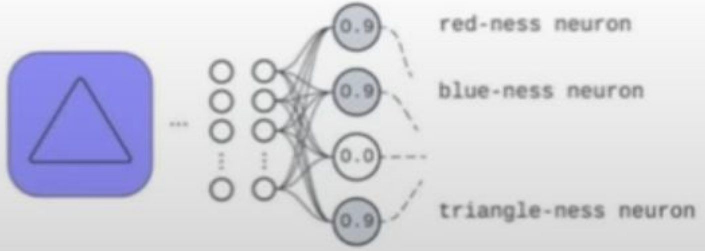


## Attention vs RNN comparison

- Quadratic dependence on the context length for attention.
- Multi-step process for RNN, highly parallel for attention.
- Attention does not have the bottleneck problem.

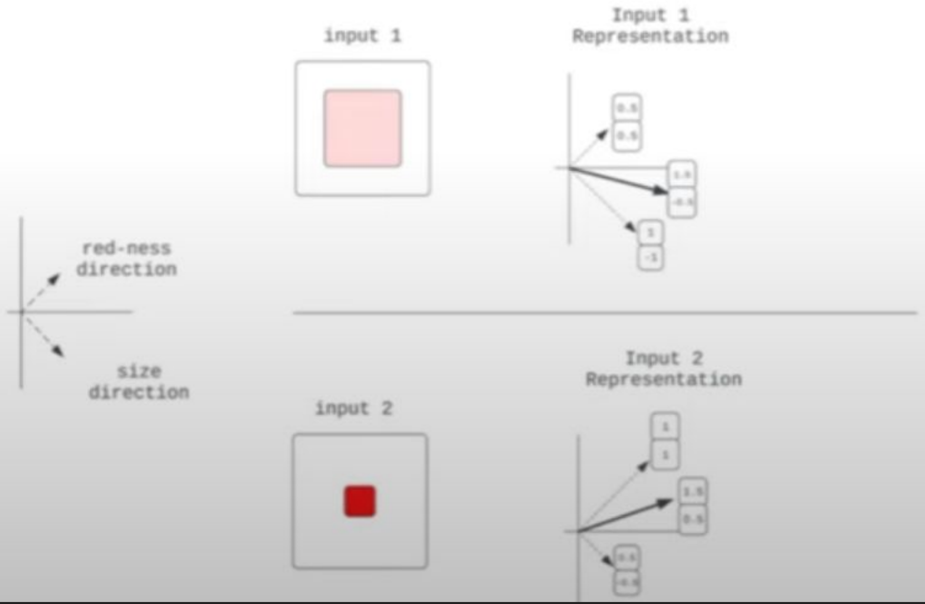
## Decomposability





## Linearity

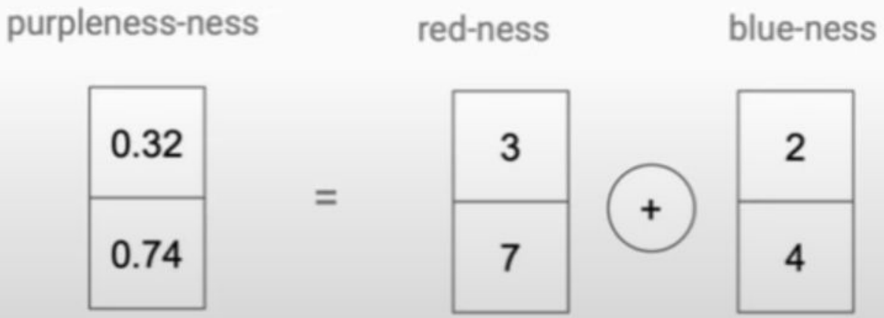
These discrete quality vectors are composed by a Sum to give the observed representation.



## Linearity

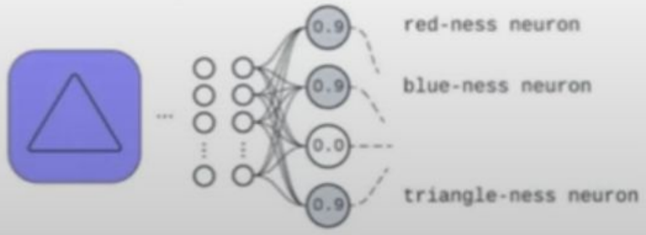
What could non-Linear composition look like?

```python
def compress_values(x1, x2, precision=1):
    z = 10 ** precision
    compressed_val = (floor(z * x1) + x2) / z
    return round(compressed_val, precision * 2)
```



## Demands of Linearity

But Linearity also has pretty stringent demands: As a compression scheme, it requires as many vector dimensions as the number of discrete qualities you want to encode.

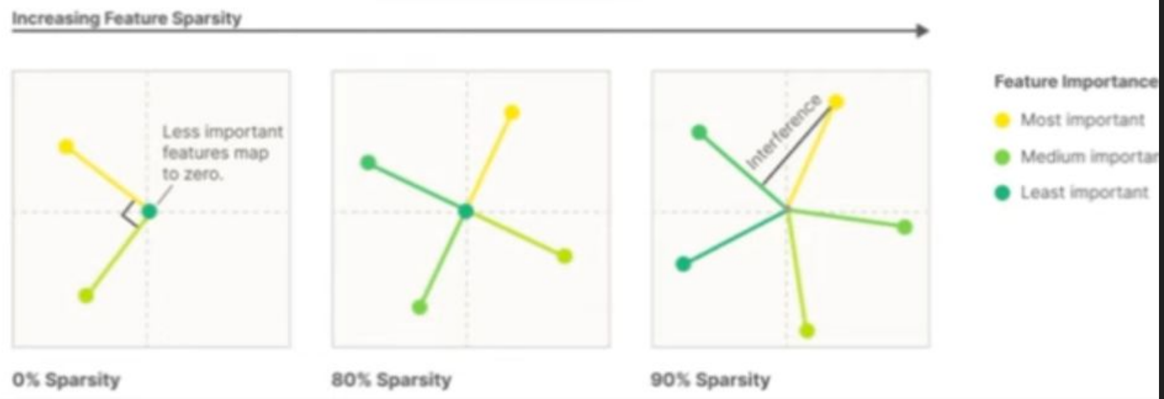


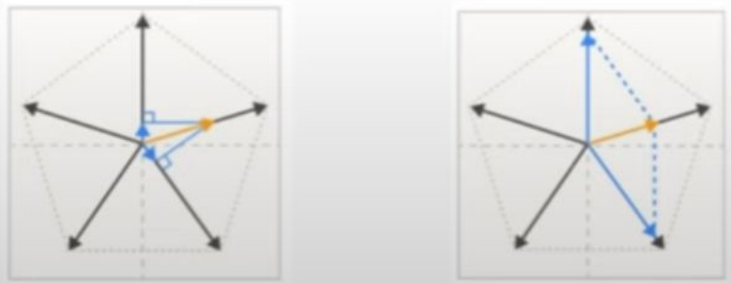
$$\overset{\text{Input Representation}}{\begin{bmatrix}0.9\\0.9\\0\\0.9\end{bmatrix}} = \overset{\text{red-ness}}{\begin{bmatrix}0.9\\0\\0\\0\end{bmatrix}} + \overset{\text{blue-ness}}{\begin{bmatrix}0\\0.9\\0\\0\end{bmatrix}} + \overset{\text{square-ness}}{\begin{bmatrix}0\\0\\0\\0\end{bmatrix}} + \overset{\text{triangle-ness}}{\begin{bmatrix}0\\0\\0\\0.9\end{bmatrix}}$$

# Sparsity

A key reason why this works is sparsity. Although language and other representation tasks have a very large number of helpful features that would be worth representing, they don't all show up in any given input at the same time.

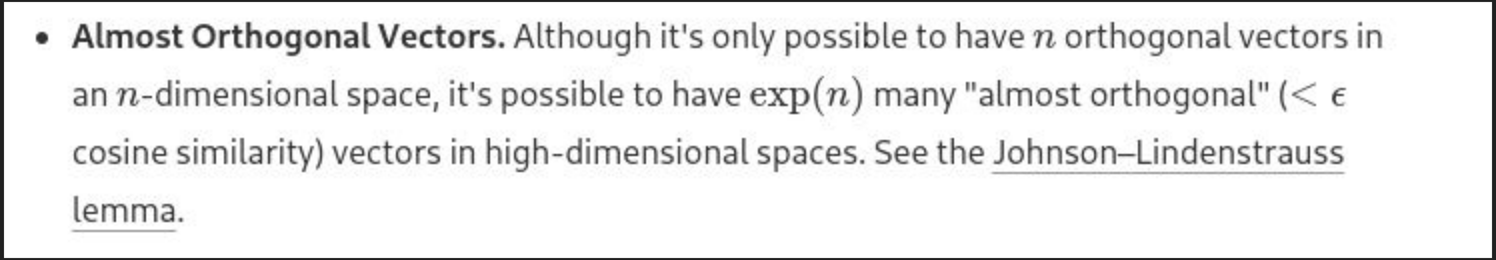
This means as sparsity increases, the interference costs of having more features than neurons drops off.





One sparsely activated Vector with little interference

Two activated Vectors being misinterpreted.



- **Almost Orthogonal Vectors.** Although it's only possible to have $n$ orthogonal vectors in an $n$-dimensional space, it's possible to have $\exp(n)$ many "almost orthogonal" ($< \epsilon$ cosine similarity) vectors in high-dimensional spaces. See the Johnson–Lindenstrauss lemma.

OBJ

## Feedback is a gift

https://tinyurl.com/dnn-2025-12-03