Lecture 15:

Language Models and Transformer

Markus Hohle

University California, Berkeley

Machine Learning Algorithms

MSSE 277B, 3 Units

Course Map

Lecture 1: Course Overview and Introduction to Machine Learning

classic ML tools & algorithms

Lecture 2: Bayesian Methods in Machine Learning

Lecture 3: Dimensionality Reduction: Principal Component Analysis

Lecture 4: Linear and Non-linear Regression and Classification

Lecture 5: Unsupervised Learning: K-Means, GMM, Trees

Lecture 6: Adaptive Learning and Gradient Descent Optimization Algorithms

ANNs/AI/Deep Learning

Lecture 7: Introduction to Artificial Neural Networks - The Perceptron

Lecture 8: Introduction to Artificial Neural Networks - Building Multiple Dense Layers

Lecture 9: Convolutional Neural Networks (CNNs) - Part I

Lecture 10: CNNs - Part II

Lecture 11: Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs)

Lecture 12: Combining LSTMs and CNNs

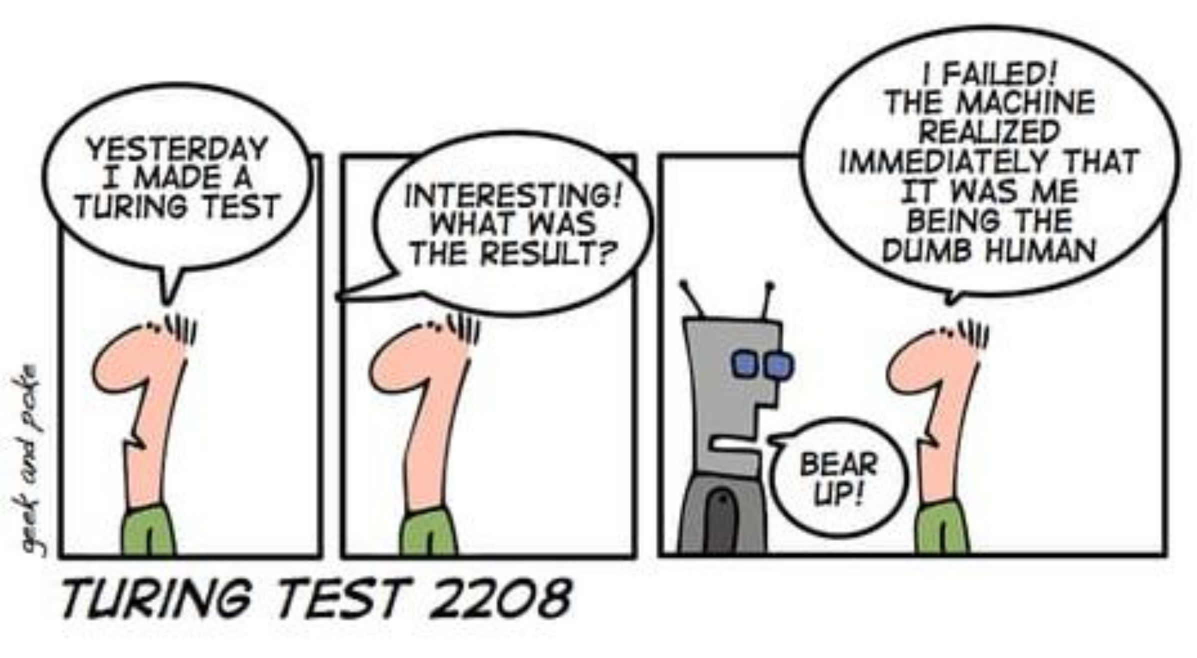
Lecture 13: Running Models on GPUs and Parallel Processing

Lecture 14: Project Presentations

Lecture 15: Transformer

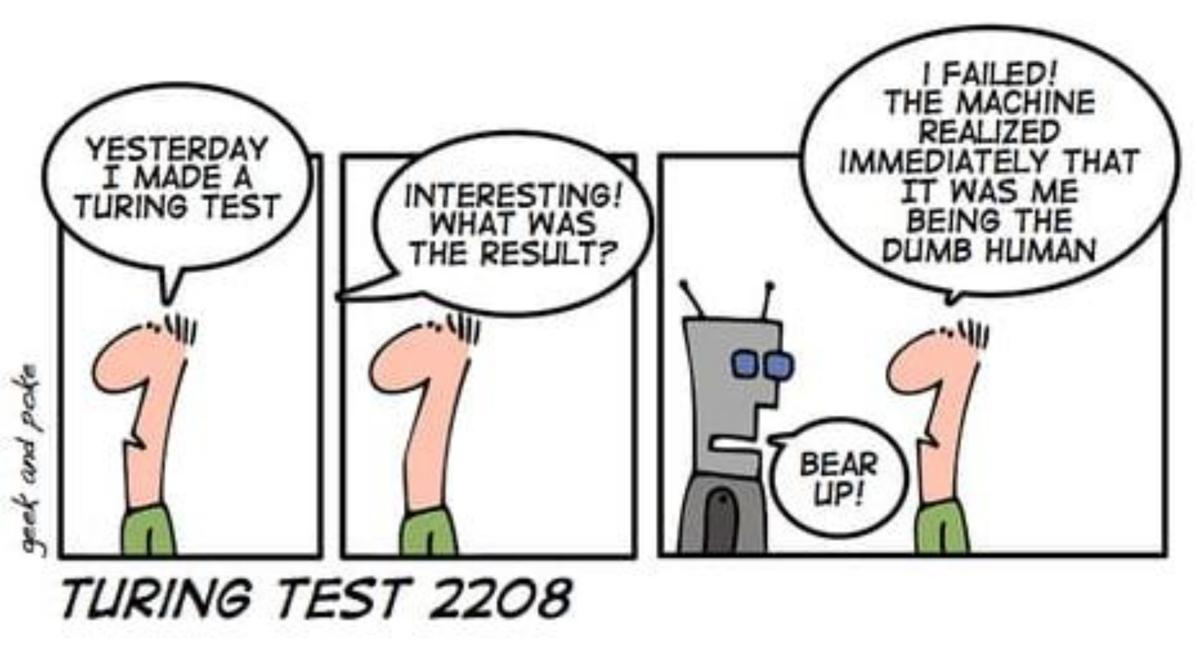
Lecture 16: GNN

Outline



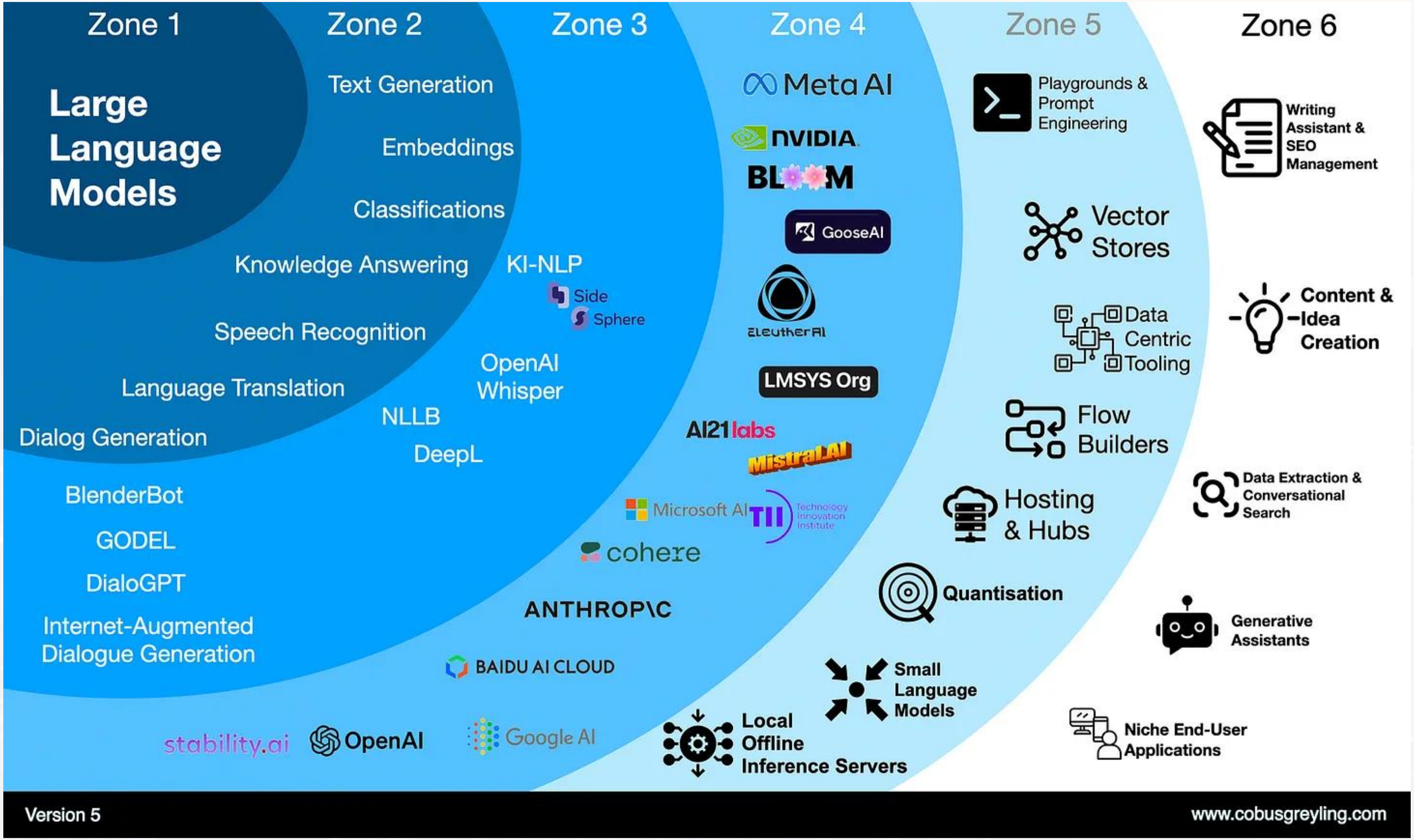
- **Introduction**
- **Bigram and MAP**
- **Positional Encoding**
- **Word Embedding**
- **Attention**
- **Transformer Architecture**

Outline



- **Introduction**
- Bigram and MAP
- Positional Encoding
- Word Embedding
- Attention
- Transformer Architecture

LLM & Transformer: Introduction



corpus: (large, representative) data set containing sequences of a language

token: individual, independent entity of a language

alphabet/vocabulary: set of tokens

token	**size of *alphabet***
- letters in a word	- 10^2
- words in a sentence	- $10^4 \ldots 10^6$
(**upper/lower case, cases, gender, tenses, conjugations**)	
- amino acids in a protein sequence	- 21
- nucleotides in a DNA/RNA sequence	- 4
- motifs in a DNA/RNA sequence	- 10^4

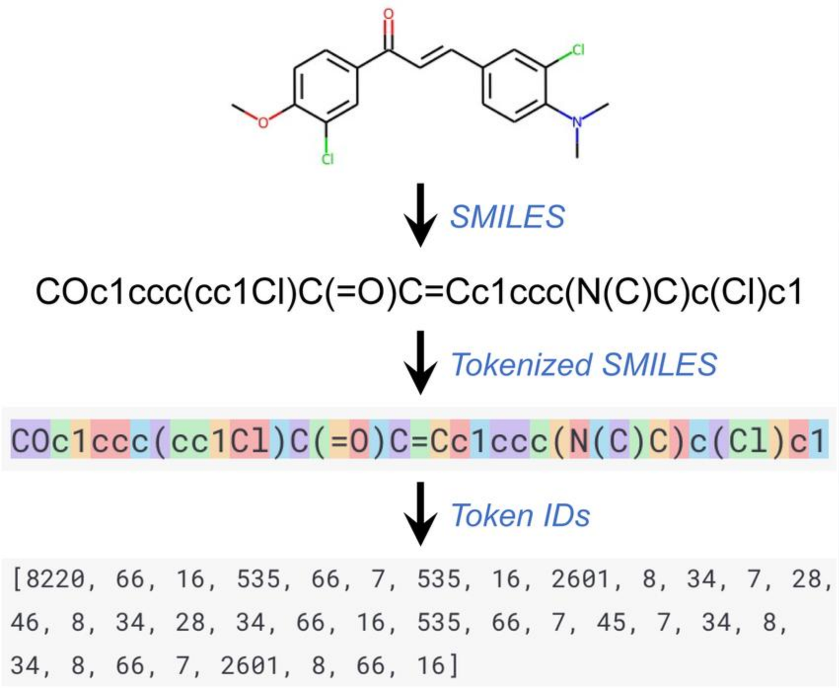
LLM & Transformer:

corpus: (large, representative) data set containing sequences of a language

token: individual, independent entity of a language

alphabet/vocabulary: set of tokens

tokenization



DOI:10.1039/D3SC04610A

token:
- single atom vs…
- …functional group

LLM & Transformer:

corpus: (large, representative) data set containing sequences of a language

token: individual, independent entity of a language

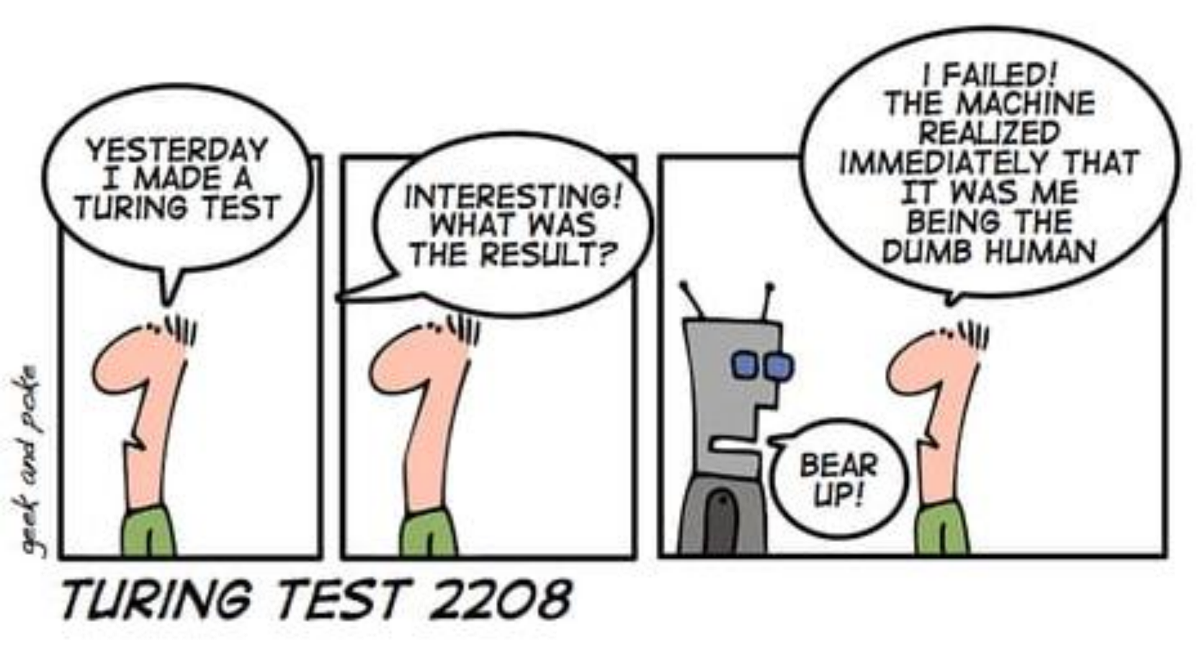
alphabet/vocabulary: set of tokens

note: language models don't know grammar as we do, but they don't need to anyway…

three things make context (details: see later**):**

- **word embedding** (relation between similar/different token)
- **positional encoding** (location of token in a sequence)
- **attention** (relation between token within a sequence)

Outline



- Introduction
- **Bigram and MAP**
- Positional Encoding
- Word Embedding
- Attention
- Transformer Architecture

LLM & Transformer:

$X_1\ X_2\ X_3\ X_4\ X_5\ \dots.X_n$ sequence of *n* **token** *X*

actually:

$$P(X_1\ X_2\ X_3\ X_4\ X_5\ \dots.X_n) = P(X_n|X_{n-1}\dots X_1)P(X_{n-1}|X_{n-2}\dots X_1)\dots P(X_1)$$

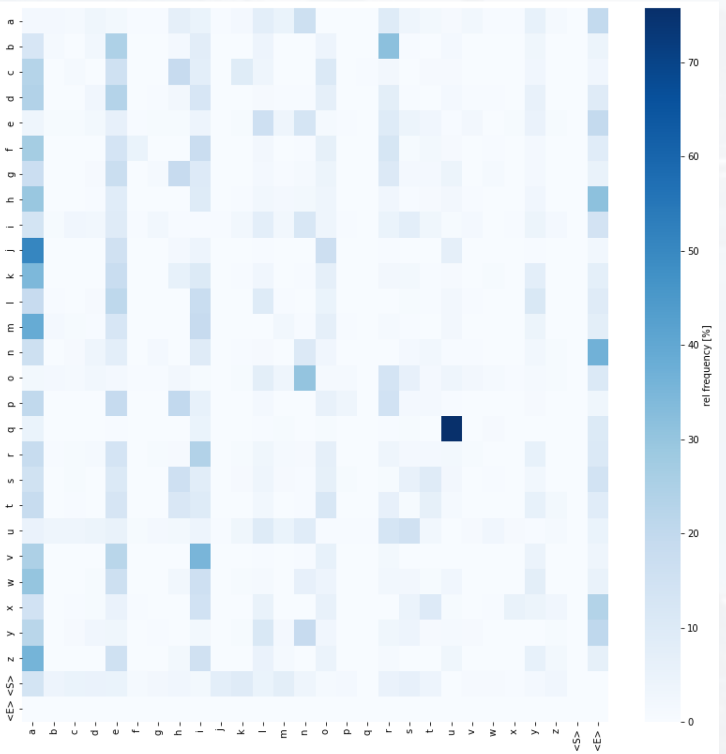
bigram (1st order Markov Chain, see e.g. first WhatsApp versions):

$$P(X_1\ X_2\ X_3\ X_4\ X_5\ \dots.X_n) = P(X_n|X_{n-1})\underbrace{P(X_{n-1}|X_{n-2})}\dots P(X_1)$$

P(i|j): that token ***i*** is generated after token ***j***

→ N x N transition matrix from frequencies

→ **"bigram" = "two words"**



frequency matrix of letters in common names

bigram (1st order Markov Chain):

let's build our own bigram model: **generate new names** based on a corpus of names

```
In [15]: words[0:12]
Out[15]:
['emma',
 'olivia',
 'ava',
 'isabella',
 'sophia',
 'charlotte',
 'mia',
 'amelia',
 'harper',
 'evelyn',
 'abigail',
 'emily']
```

see **Andrej Karpathy's GitHub** repository

$$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots . X_n) = \boxed{P(X_n|X_{n-1})P(X_{n-1}|X_{n-2})} \ldots \boxed{P(X_1)}$$

we only need to count **how often** a letter is followed by another

we also need to indicate when a name has **started** and **ended**

`['<S>'] + ['olivia'] + ['<E>']`

→ **alphabet:** 26 letters + the two special "letters"

let's create a dictionary first (will help for counting):

LLM & Transformer: Bigram & MAP

<u>bigram</u> <u>(1st order Markov Chain):</u>

let's build our own bigram model: **generate new names** based on a corpus of names

1) dictionary first (will help for counting)

2) count how often letter ***i*** is followed by letter ***j***

   → bigram matrix ***N***

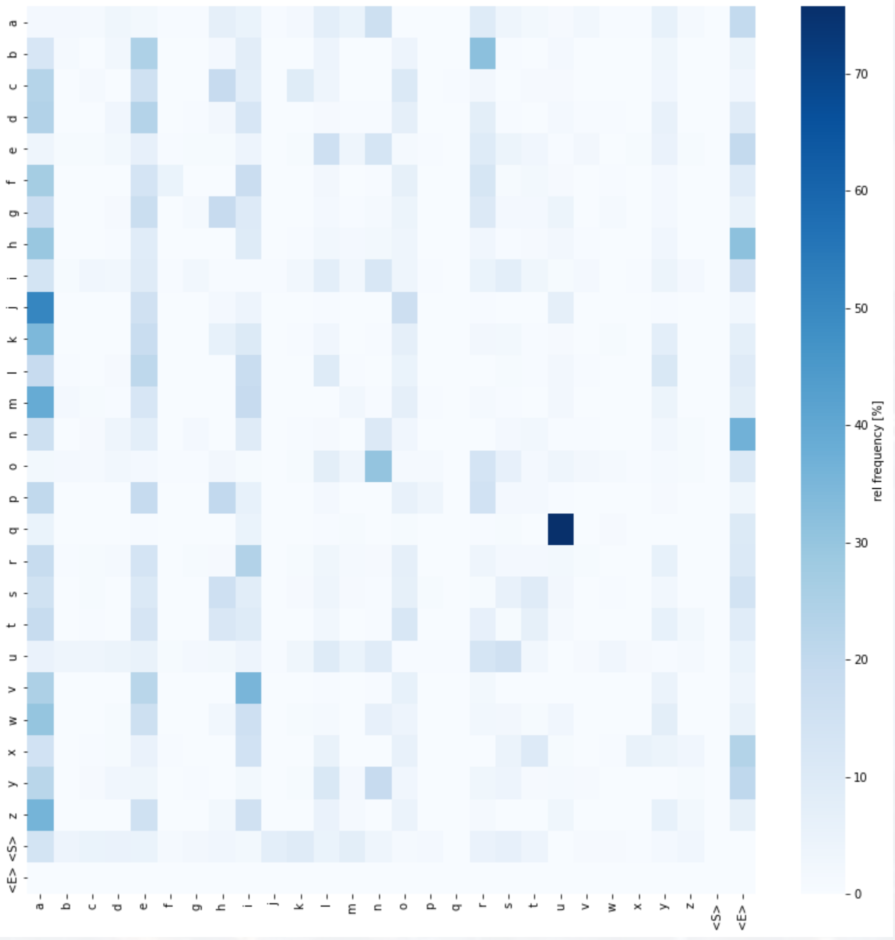
3) normalize ***N*** accordingly

4) begin with a start token

5) draw a letter randomly based on ***N***, using

```
np.argmax(np.random.multinomial(1,p))
```

6) if next token is stop token → stop



bigram (1st order Markov Chain):

let's build our own bigram model: **generate new names** based on a corpus of names

check out **`Bigram.ipynb`**

`B.SampleNames(15)` vs totally random **`B.SampleNames(15, False)`**

some names are gibberish

some names sound real

some names **are** real

```
In [295]: B.SampleNames(15)
keesa
ann
ja
jon
nma
malynojana
sall
daha
drvah
lzaxi
tyunusthun
jorrwro
ja
asoow
s
```

```
In [296]: B.SampleNames(15,False)

mtkgy
yufexhviovmorhqvikbbbjxebpxwurejaqlzzuwuanxmmomhr<S>uhb
xlmusadjfdzxadaotd
ik<S>vdtydvxevtaselkykcfbamceprtvl
zyr<S>inzoerobzwuovx
eg<S>pbdvikf<S>tomcnkfsjay<S>rikatnaykizszcivpds<S>zj
kh<S>y<S>ualzugqgakakeubjbasc
bblupnibtqmyl<S>vyobf
kybs
rznjgpmlo
tnhoxuckkjjzbwmj<S>vshkycicf<S>kowskphy
rxodh
jvswmzw
jzpcfnpcg
```

LLM & Transformer: Bigram & MAP

Note, there is no conceptual difference between applying our model to *letters in a word* vs *words in a sentence*

<u>caveats:</u>

- the bigram model derives $P(X_n)$ from **observed** frequencies
  → essentially **MLE** (problematic if a letter hasn't appeared in the sequence yet
  → $P(X_n)$ assumed to be zero!)

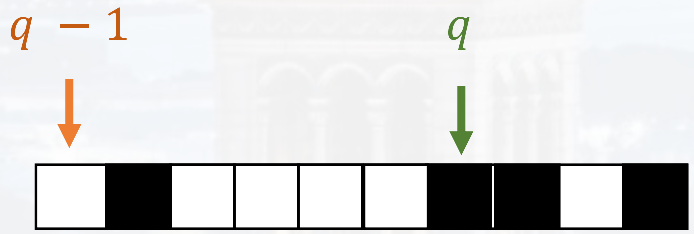
```
Nsam = N/np.sum(N+0.0001, axis = 1, keepdims = True)
S_bi += np.sum(-N[:,i]*np.log(N[:,i]+1e-16))
```

- can we implement something that is closer to:

$$P(X_1\, X_2\, X_3\, X_4\, X_5 \dots . X_n) = P(X_n|X_{n-1} \dots X_1)P(X_{n-1}|X_{n-2} \dots X_1) \dots P(X_1) \quad ?$$

binomial process

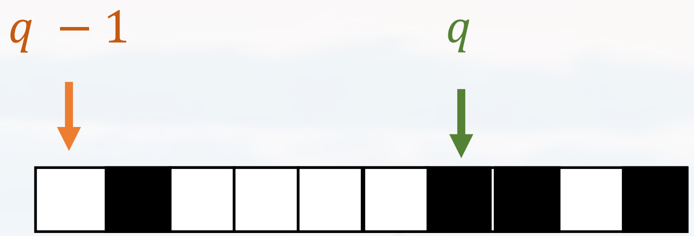
$$P(k|n,q) = \binom{n}{k} q^k (1-q)^{n-k}$$



$$q = ?$$

$$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots . X_n) = P(X_n|X_{n-1} \ldots X_1)P(X_{n-1}|X_{n-2} \ldots X_1) \ldots P(X_1)$$

Bayesian **P**arameter **E**stimation*)



$$P(k|n,q) = \binom{n}{k} q^k (1-q)^{n-k} \qquad q = ?$$

likelihood function **(here: binomial)**

$$P(q|data\ set) = \frac{\boxed{P(data\ set|q)}\boxed{P(q)}}{\boxed{P(data\ set)}}$$

prior **(~$P(X_n|X_{n-1} \ldots X_1)$)**

evidence **(const wrt q)**

$$= \frac{1}{\int_0^1 P(q|data\ set)dq}(1-q)^{n-k}q^k$$

$q = const$
before 1st data point
(max entropy!)

$$= \frac{q^{k+\alpha-1}(1-q)^{n-k+\beta-1}}{\int_0^1 q^{k+\alpha-1}(1-q)^{n-k+\beta-1}\,dq}$$

q = conjugate prior
after nth data point

***) see lecture 2**

$$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots .\, X_n) = P(X_n|X_{n-1} \ldots X_1)P(X_{n-1}|X_{n-2} \ldots X_1) \ldots P(X_1)$$

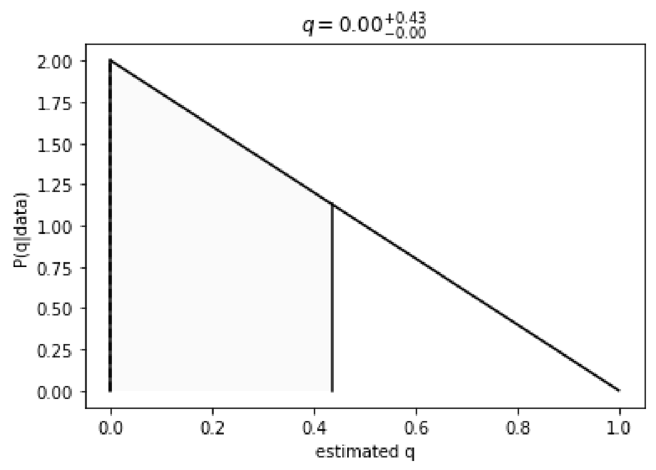
$$P(k|n,q) = \binom{n}{k} q^k (1-q)^{n-k} \qquad q = ?$$

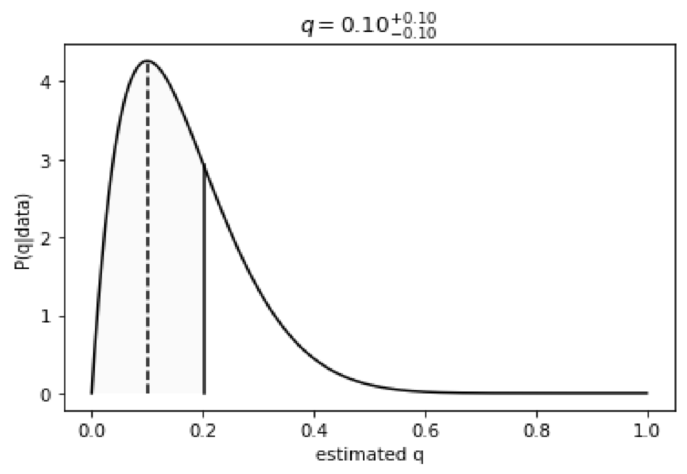
Bayesian
Parameter
Estimation*)

n = 1 n = 2 n = 3 n = 4 n = 5 … n = 9 n = 10

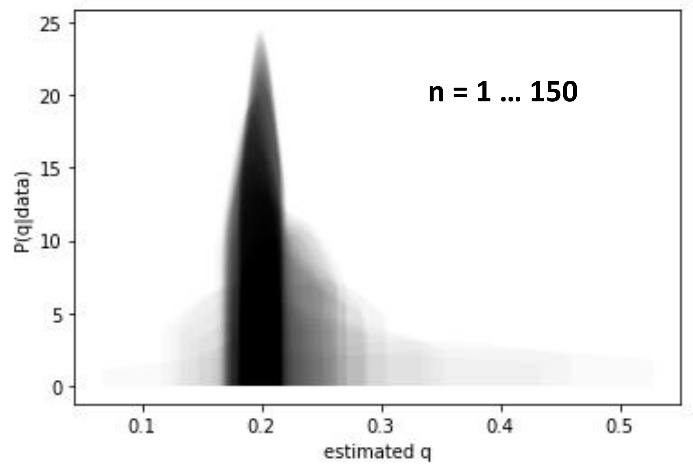
system is learning i. e. reducing entropy

$P(X_1)$





$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots .\, X_n)$



*) see lecture 2

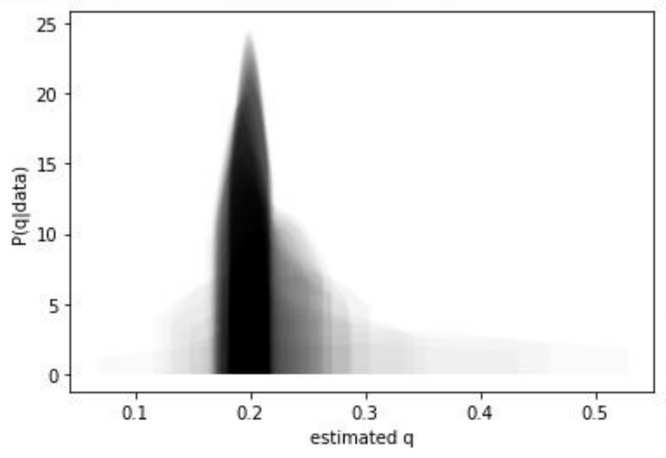
$$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots .\, X_n) = P(X_n|X_{n-1} \ldots X_1)P(X_{n-1}|X_{n-2} \ldots X_1) \ldots P(X_1)$$

Bayesian **P**arameter **E**stimation*)

$$P(k|n,q) = \binom{n}{k} q^k (1-q)^{n-k}$$

$$P(q|data\ set) = \frac{q^{k+\alpha-1}(1-q)^{n-k+\beta-1}}{\boxed{\int_0^1 q^{k+\alpha-1}(1-q)^{n-k+\beta-1}\, dq}}$$

Beta function



more general, we want to learn the probability $P_j(a)$ of letter a at position *j* $\quad q \rightarrow P_j(a)$

→ multinomial problem

→ conjugate prior is the **Dirichlet distribution**

$$P(sequence) \sim \prod_j \boxed{\prod_a P(a)_j^{\alpha(a)-1}}$$

equivalent to what was $P(q|data\ set)$ earlier

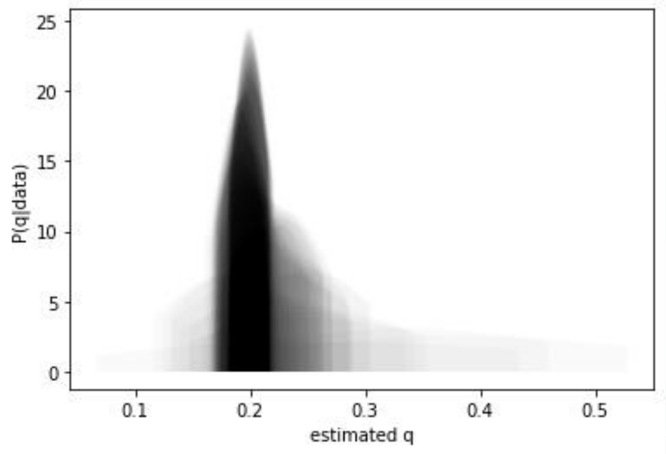
***) see lecture 2**

LLM & Transformer:

$$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots .\, X_n) = P(X_n|X_{n-1} \ldots X_1)P(X_{n-1}|X_{n-2} \ldots X_1) \ldots P(X_1)$$

Dirichlet distribution

$$P(sequence) \sim \prod_j \prod_a P(a)_j^{\alpha(a)-1}$$



note: $\sum_{over\ all\ a} P(a)_j = 1$ → N – dim simplex

note: - we don't need to extract $P(a)$ from the maximum of the pdf given by the BPE posterior

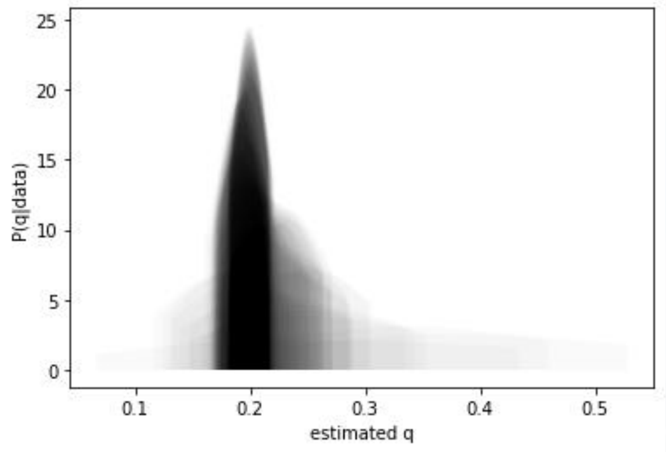
- we can directly derive the maximum of $P(a)$ from $P(sequence)$ given the constrain $\sum_{over\ all\ a} P(a)_j = 1$ **(Lagrangian multipliers)**

- **Maximum a-posteriori (MAP)** approach → see XXmotif (Siebert & Soeding, 2016)

$$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots .\, X_n) = P(X_n|X_{n-1} \ldots X_1)P(X_{n-1}|X_{n-2} \ldots X_1) \ldots P(X_1)$$

Dirichlet distribution

$$P(sequence) \sim \prod_j \prod_a P(a)_j^{\alpha(a)-1}$$



note: $\sum_{over\ all\ a} P(a)_j = 1$ → N – dim simplex

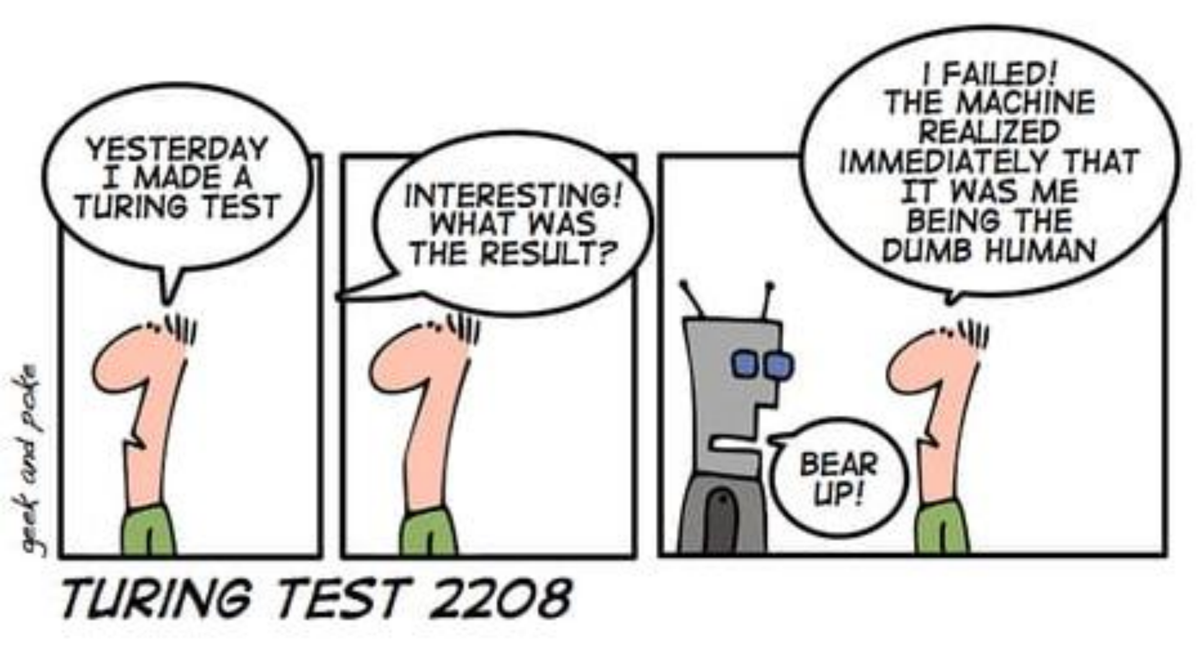
Maximum a-posteriori (MAP)

- XXmotif (Siebert & Soeding, 2016) significantly outperformed PWMs

- it struggled however with related motifs which where **physically located far apart** from each other

→ solution see later: attention
→ older solutions: LSTMs

Outline



- Introduction
- Bigram and MAP
- **Positional Encoding**
- Word Embedding
- Attention
- Transformer Architecture

three things make context:

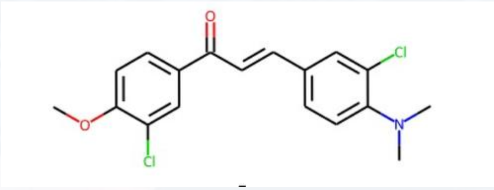
- **positional encoding** (location of token in a sequence)
- **word embedding** (relation between similar/different token)
- **attention** (relation between token within a sequence)

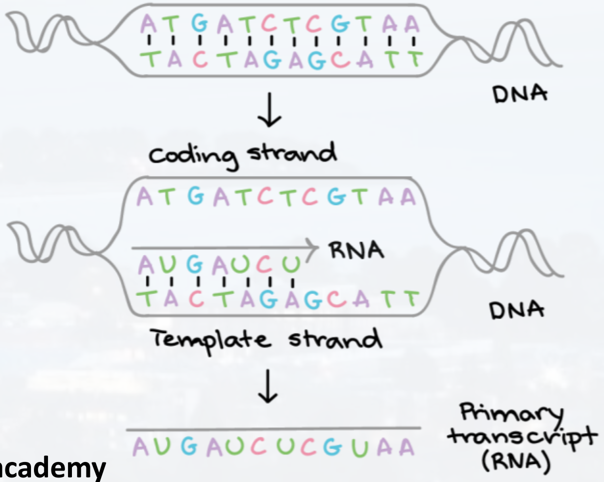
“The cat jumped on the roof.”

order matters!:

1st: article
2nd: noun/subject
3rd: verb
4th : noun/object (in English)

→ **positional encoding**





Khan academy

goal: find a positional encoding that is

- reasonably simple
- independent from the length of the sequence
- somehow normalized

<u>one idea: n-bit binary encoding</u>

position	code
	76543210 ← 8bit i.e. eight dimensions
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
10	00001010
11	00001011
12	00001100
13	00001101
14	00001110
15	00001111
16	00010000

depending on dimensions (bit)

→ different frequencies

bit	**frequency**
0	1/2
1	1/4
2	1/8
…	

Does that look familiar?
→ *like* Fourier Series

even dimensions: $E(p, 2k) = \sin\left(\frac{p}{10{,}000^{\frac{2k}{d}}}\right)$

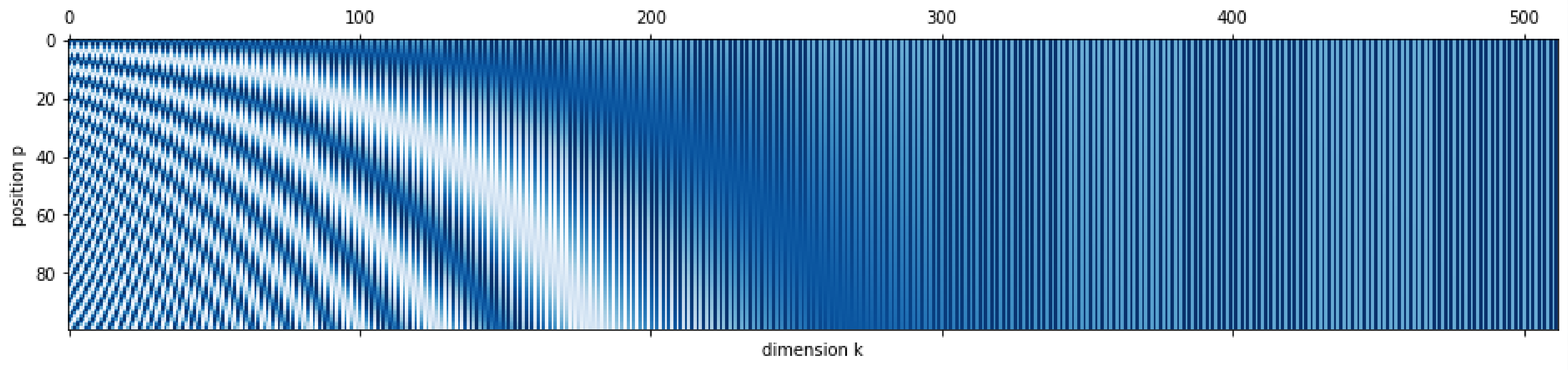
odd dimensions: $E(p, 2k+1) = \cos\left(\frac{p}{10{,}000^{2k/d}}\right)$

p: position in sequence
k: dimension index
d: number of dimensions
10,000: an arbitrary number

Vaswani et al., 2017

run **`PlotPositionEncoding.py`**

more info [here]



even dimensions: $E(p, 2k) = \sin\left(\frac{p}{10{,}000^{\frac{2k}{d}}}\right)$

odd dimensions: $E(p, 2k+1) = \cos\left(\frac{p}{10{,}000^{2k/d}}\right)$

p: position in sequence
k: dimension index
d: number of dimensions
10,000: an arbitrary number

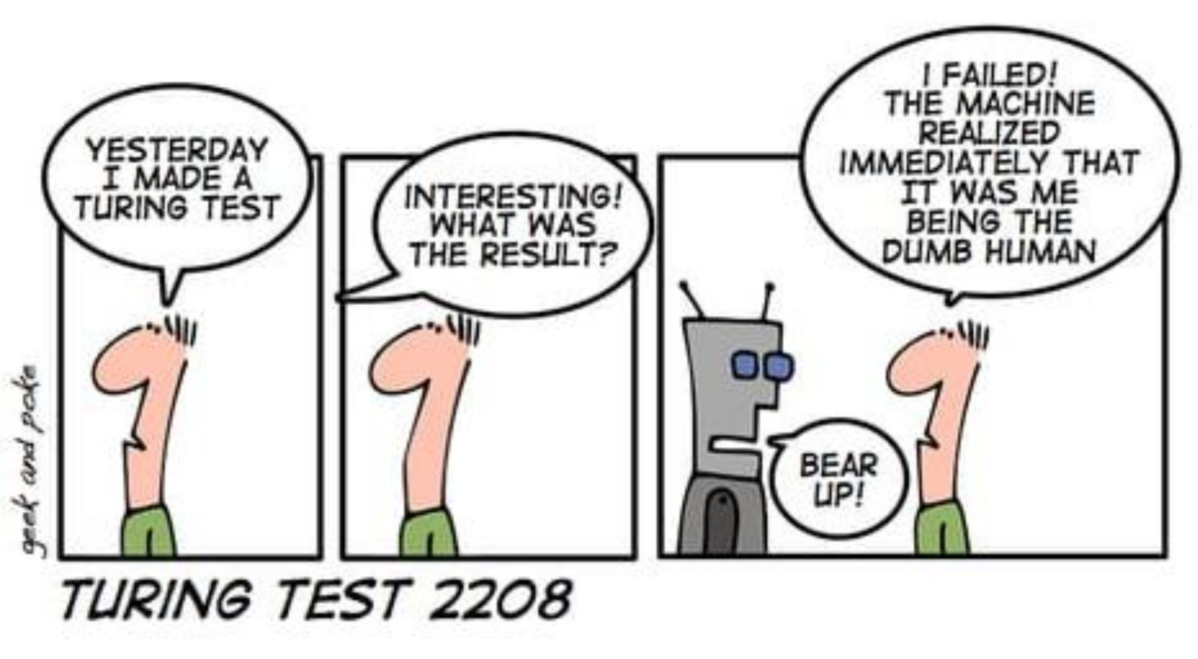
Vaswani et al., 2017

run **`PlotPositionEncoding.py`**

note:

- easier to handle numerically vs discrete encoding
- $\cos(x + 2\pi k) = \cos(x)$ and $\sin(x + 2\pi k) = \sin(x)$
  → absolute position not relevant but ***relative*** position

Outline



- Introduction
- Bigram and MAP
- Positional Encoding
- **Word Embedding**
- Attention
- Transformer Architecture

three things make context:

- **positional encoding** (location of token in a sequence)
- **word embedding** (relation between similar/different token)
- **attention** (relation between token within a sequence)

problem: turning token (words/letters) into numbers

single letters:

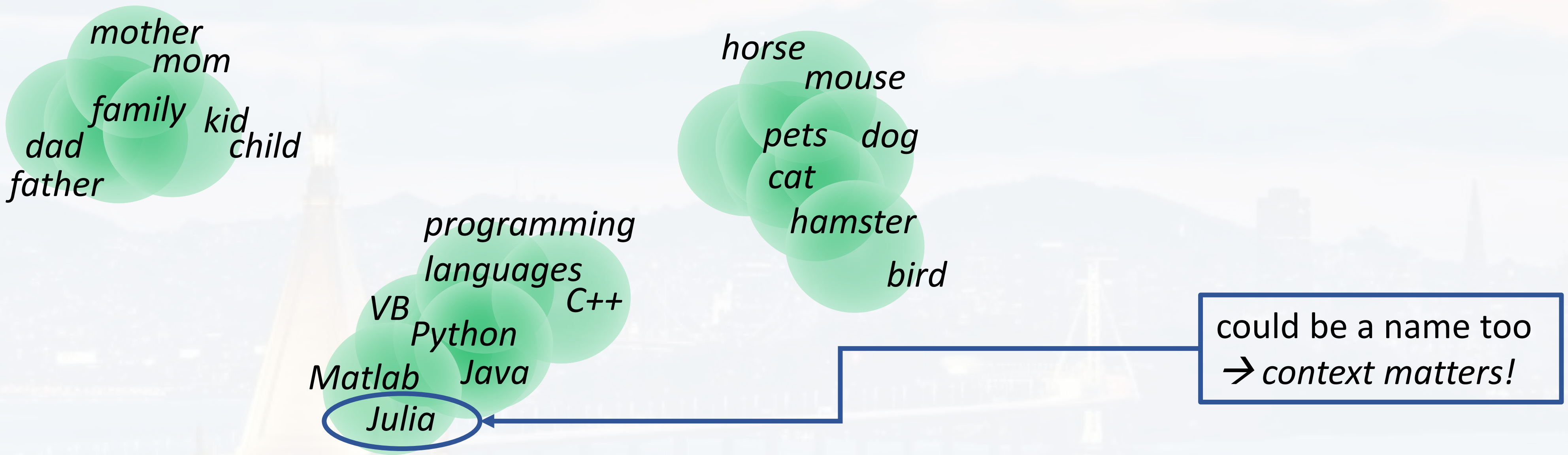
ACGT - one – hot works perfectly (four different token)
abcd… - lower/upper case, special characters (50 different token), one – hot is fine too

words:

actual words - $10^4 \ldots 10^6$ (upper/lower case, cases, gender, tenses, conjugations)
→ one – hot doesn't work (matrices would be too large)
→ some words have a **similar meaning,** should be **close** in parameter space (**cluster**)

LLM & Transformer: Word Embedding

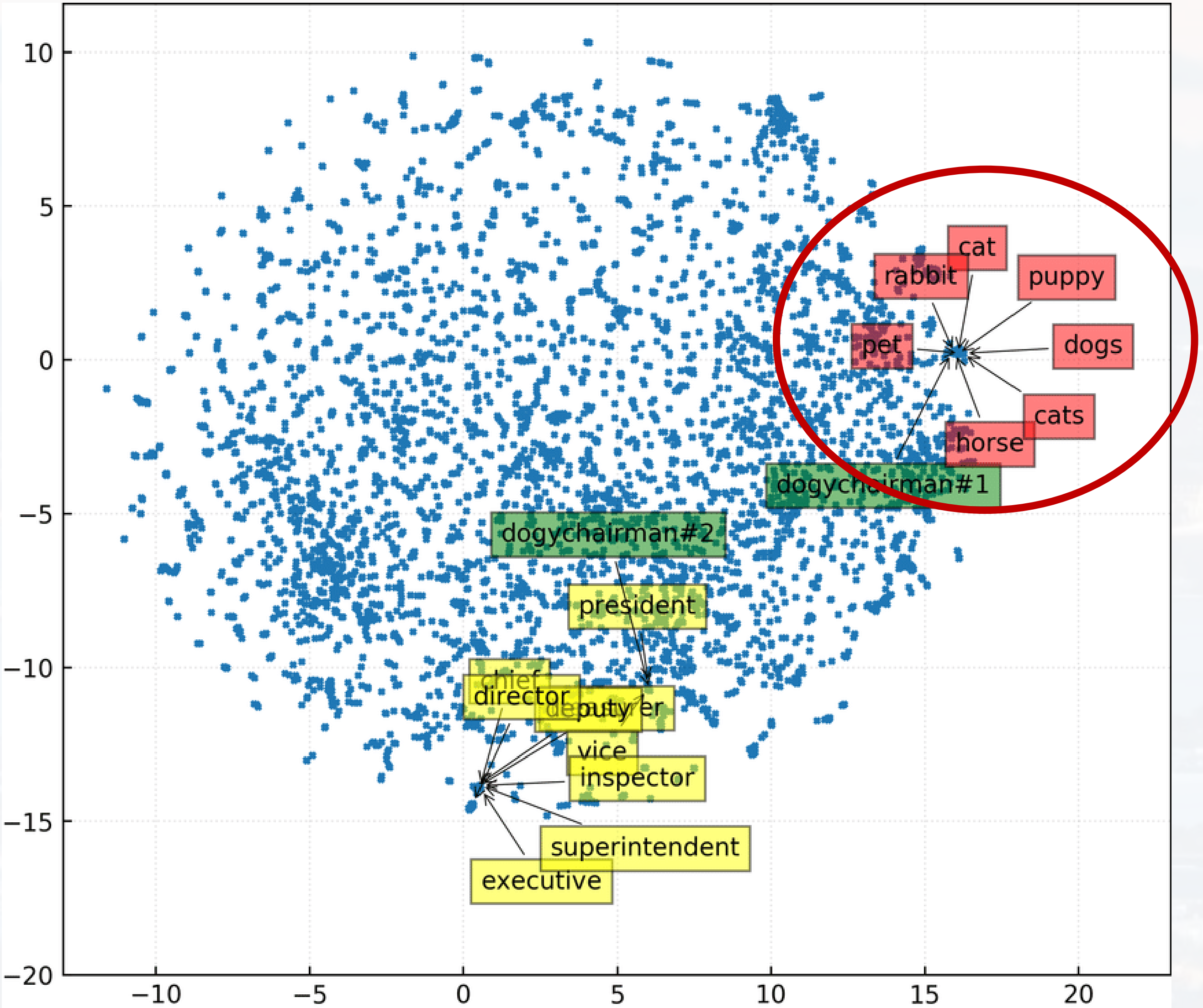
words with a **similar meaning** should form **cluster**



- embedding, instead of one – hot encoding
- from experience: **N = 30 – 300** dim vector for each token **(which is a lot less than 10^4 … 10^6)** is sufficient
- as a result: token with **similar meaning are close** in the vector space!

LLM & Transformer: Word Embedding

words with a **similar meaning** should form **cluster**



"Joint Learning of Sense and Word Embeddings"
M Alsuhaibani & D Bollegala

common training set: recorded speeches from the European Parliament:

…It seems absolutely disgraceful that we pass legislation and do not adhere to it ourselves. Mrs Lynne, you are quite right and I shall check whether this has actually not been done. I shall also refer the matter to the College of Quaestors, and I am certain that they will be keen to ensure that we comply with the regulations we ourselves vote on.
Madam President, Mrs Díez González and I had tabled questions on certain opinions of the Vice-President, Mrs de Palacio, which appeared in a Spanish newspaper.
The competent services have not included them in the agenda on the grounds that they had been answered in a previous part-session.
I would ask that they reconsider, since this is not the case….

words of similar meaning should appear in similar environment

→ *target token within a window*

Two common algorithms are ***C****ontinuous* ***B****ag* ***O****f* ***W****ords* and ***skip gram***

Continuous Bag Of Words

n:	number of unique token from corpus
m:	desired number of dimensions for embedding

The competent services have not included them in the agenda on the grounds that they…

target: *have* → one – hot encoding → target vector of length *n*

context window: *The competent services* *not included them* → one – hot encoding

shallow ANN:

- input layer of shape *n x m*
- hidden layer of shape *m x n*
- softmax

→ output vector of length *n*

learning weights such that both vectors **are identical**

→ ***weights[:,i]*** of the **hidden layer** make up the embedding vector of ***token i***

*Continuous **B**ag **O**f **W**ords*

n:	**number of unique token from corpus**
m:	**desired number of dimensions for embedding**

The competent services have not included them in the agenda on the grounds that they…

target: *not*

context window *competent services have included them in*

one – hot encoding

shallow ANN:

input layer of shape *n x m*

hidden layer of shape *m x n*

softmax

output vector of length *n*

one – hot encoding

target vector of length *n*

learning weights such that both vectors **are identical**

→ ***weights[:,i]*** of the **hidden layer** make up the embedding vector of ***token i***

***C**ontinuous **B**ag **O**f **W**ords*

n:	**number of unique token from corpus**
m:	**desired number of dimensions for embedding**

The competent services have not included them in the agenda on the grounds that they…

target: *included*

context window: *services have not* *them in the*

one – hot encoding

shallow ANN:

- input layer of shape *n x m*
- hidden layer of shape *m x n*
- softmax

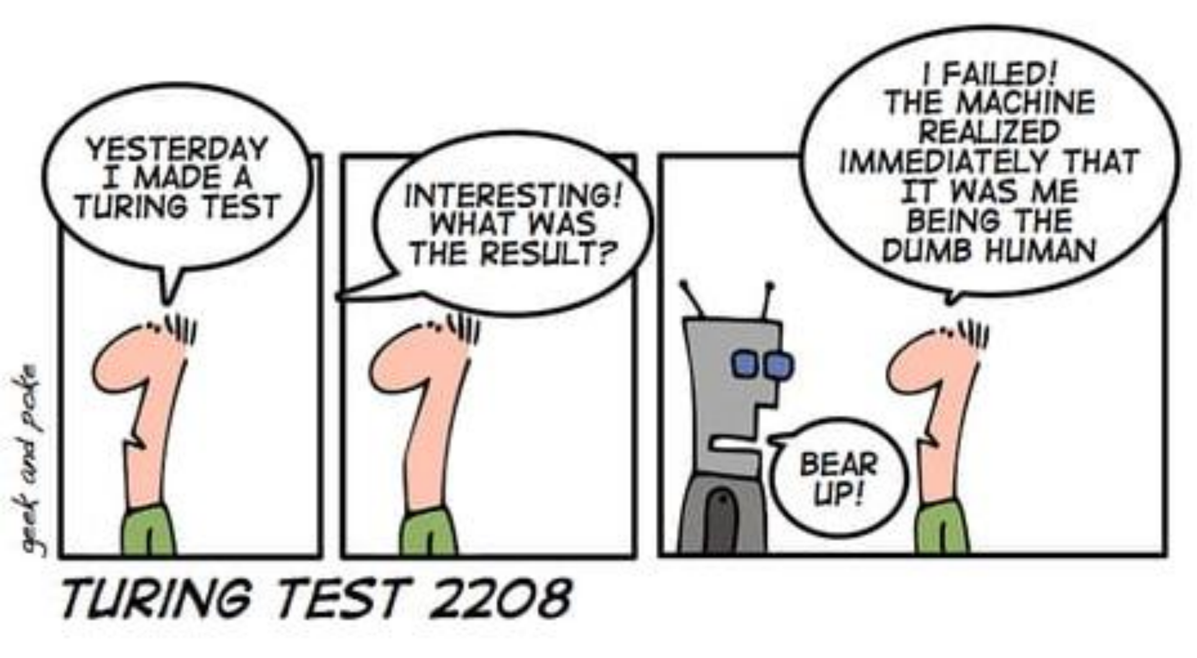
output vector of length *n*

… and so on….

one – hot encoding

target vector of length *n*

learning weights such that both vectors **are identical**
→ **weights*[:,i]*** of the **hidden layer** make up the embedding vector of ***token i***



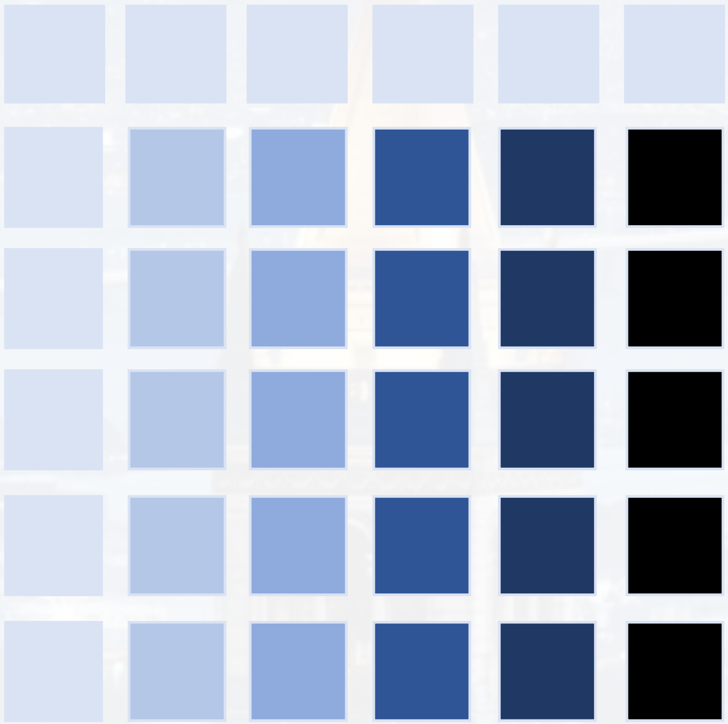
Outline

- Introduction
- Bigram and MAP
- Positional Encoding
- Word Embedding
- **Attention**
- Transformer Architecture

three things make context:

- **positional encoding** (location of token in a sequence)
- **word embedding** (relation between similar/different token)
- **attention** (relation between token within a sequence)

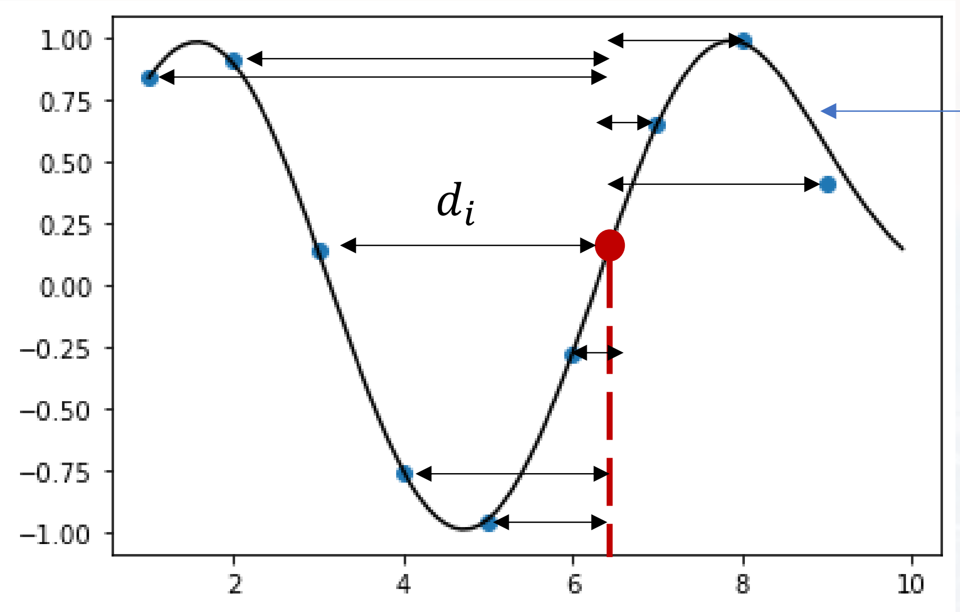
“The cat jumped on the roof.”



how the first token influences all other token

how the second token influences all other token

…. and so on



We want to interpolate between the blue dots
→ generating the black line
→ **no curve fitting!**

idea:
- select a point for which we want the interpolation for
- calculate distance d_i to every other point
- each data point should influence the value of the interpolated point
- the closer, the stronger the influence → weighted mean

$$y_{int} \sim \sum_{i=1}^{I} w_i \, y_i \qquad w_i \sim \frac{1}{\boxed{d_i}}$$

<u>calculating distance</u>


```
D = np.tile(V, (1, len(V))) -  np.tile(L.transpose(), (len(V), 1))
```

- each data point should influence the value of the interpolated point
- the closer, the stronger the influence → weighted mean

$$y_{int} \sim \sum_{i=1}^{I} w_i \, y_i$$

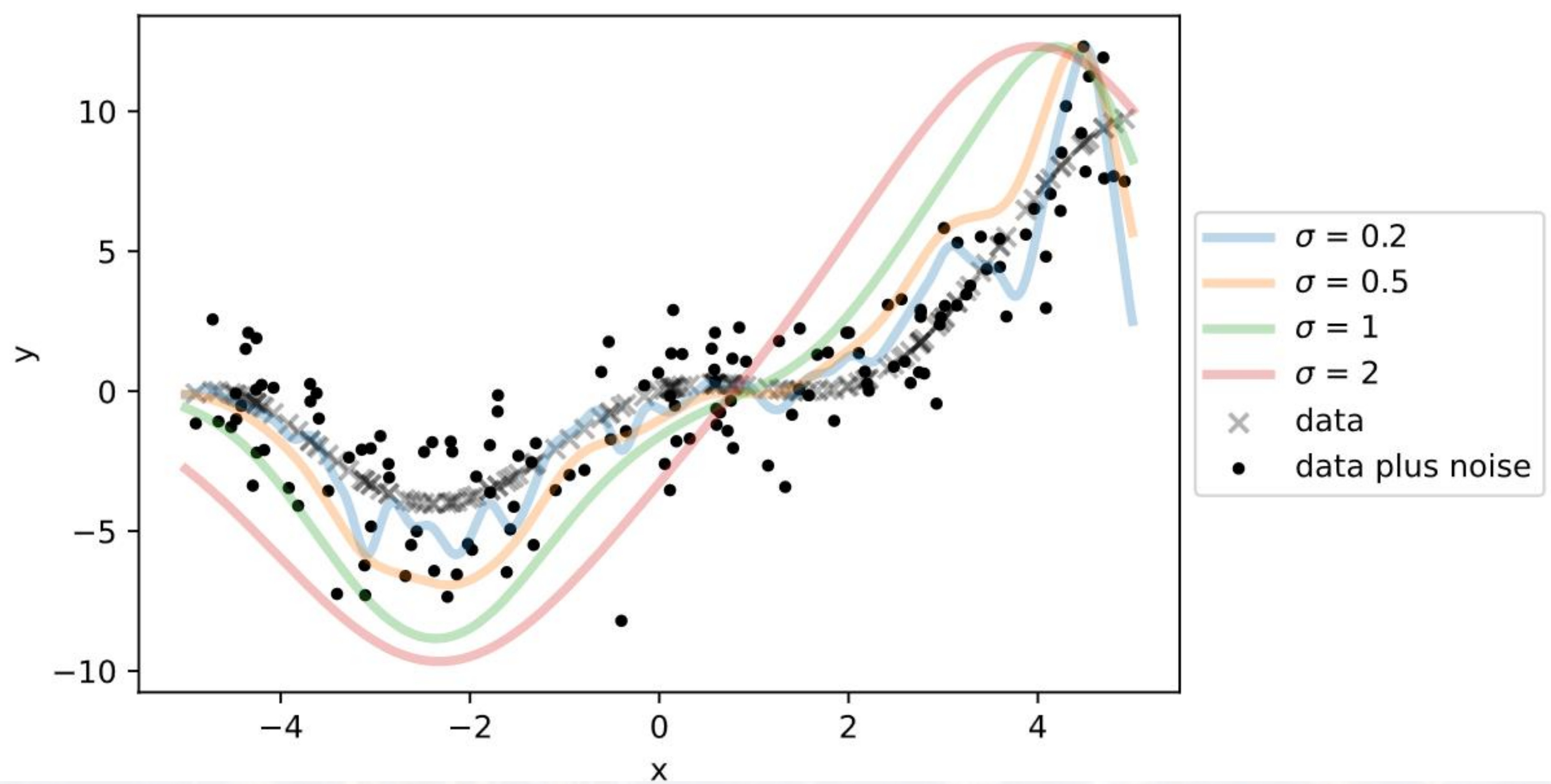
Gaussian kernel

```
W       = np.exp(-(D**2)/(sigma))
W       = W/np.sum(W + 1e-16, axis = 0)
yint    = np.dot(W.transpose(), y)
```

```
D = np.tile(V, (1, len(V))) -  np.tile(L.transpose(), (len(V), 1))
```

Gaussian kernel

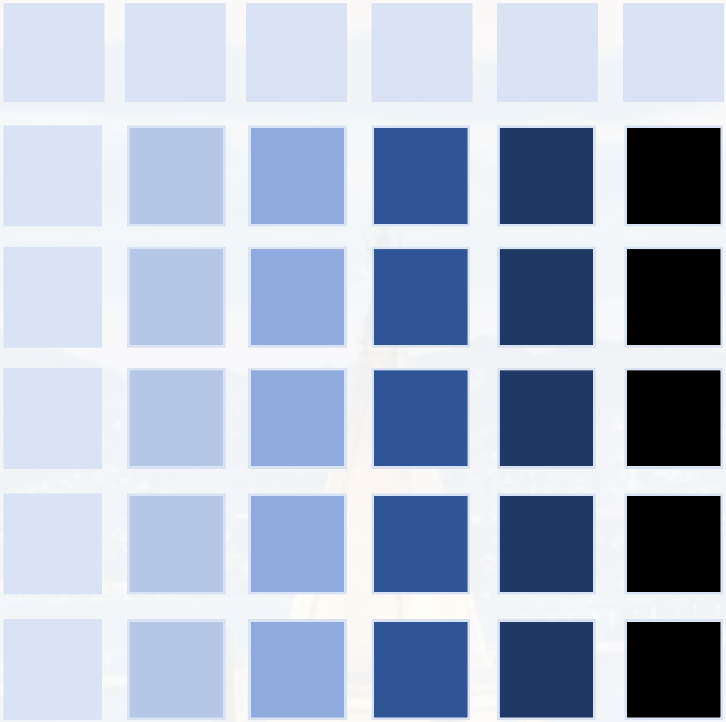
```
W       = np.exp(-(D**2)/(sigma))
W       = W/np.sum(W + 1e-16, axis = 0)
yint    = np.dot(W.transpose(), y)
```

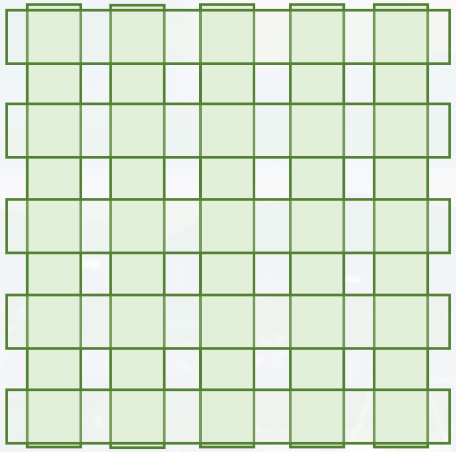


check out:

`SmoothGaussKernel.py`
`SmoothExamples.py`

“The cat jumped on the roof.”





Gaussian kernel

```
W     = np.exp(-(D**2)/(sigma))
W     = W/np.sum(W + 1e-16, axis = 0)
yint  = np.dot(W.transpose(), y)
```

actual attention:

these weights are learnable,

no kernel assumed!

LLM & Transformer: Attention

self attention imagine you want to built & train a movie GenAI **that creates movies based on queries**.

training data

“Thrilling horror science fiction movie, plays in space in a distant future”

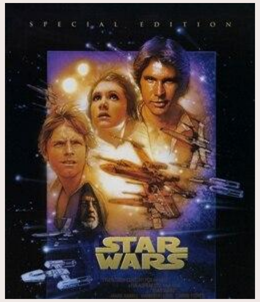
$$X = X_1, X_2, \ldots . X_N$$

each token is a vector X_n of length E

Alien

“Entertaining science fiction movie, plays in space in a distant past”

Star Wars



“Boring love story, plays on a ship in the past”

Titanic

key

value

LLM & Transformer: Attention

self attention imagine you want to built & train a movie GenAI **that creates movies based on queries**.

training data

each token is a vector X_n of length E $X = X_1, X_2, \ldots . X_N$

key

Alien

value

“Boring love story, plays in space.”

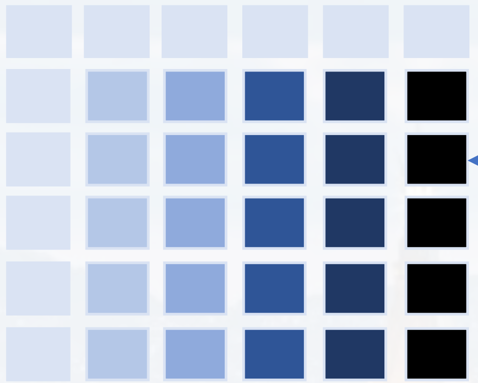
query

LLM & Transformer: Attention

self attention

key **value** **query**

"Boring love story, plays on a ship in the past"



each token is a vector X_n of length E $X = X_1, X_2, \ldots . X_N$

output:

$$Y_n = \sum_{m=1}^{N} w_{nm} X_m$$

- weights should be trainable
- weight = 0, X_m has no influence on output
- weights should be positive so that neg weights don't counteract positive weights
- normalization: $\boxed{\sum_{m=1}^{N} w_{nm} = 1}$

for returning the best suggestion:

comparing key vector $\boldsymbol{X_n}$ to query vector $\boldsymbol{X_m}$ via dot product

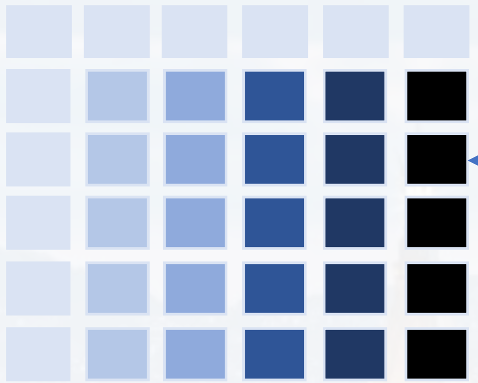
$$\boxed{w_{nm} = \frac{\exp(\boldsymbol{X_n} \circ \boldsymbol{X_m})}{\sum_{\mu=1}^{N} \exp(\boldsymbol{X_n} \circ \boldsymbol{X_\mu})}}$$ softmax

LLM & Transformer: Attention

self attention **key** **value** **query**

each token is a vector X_n of length E $X = X_1, X_2, \ldots . X_N$

“Boring love story, plays on a ship in the past”



weights w_{nm}

output:

$$Y_n = \sum_{m=1}^{N} w_{nm} X_m$$

- weights should be trainable
- weight = 0, X_m has no influence on output
- weights should be positive so that neg weights don’t counteract positive weights
- normalization: $\sum_{m=1}^{N} w_{nm} = 1$

for returning the best suggestion:

comparing key vector $\boldsymbol{X_n}$ to query vector $\boldsymbol{X_m}$ via dot product

$$\boxed{w_{nm} = \frac{\exp(\boldsymbol{X_n}\mathbf{W(K)} \circ \boldsymbol{X_m}\mathbf{W(Q)})}{\sum_{\mu=1}^{N} \exp(\boldsymbol{X_n}\mathbf{W(K)} \circ \boldsymbol{X_\mu}\mathbf{W(Q)})}}$$

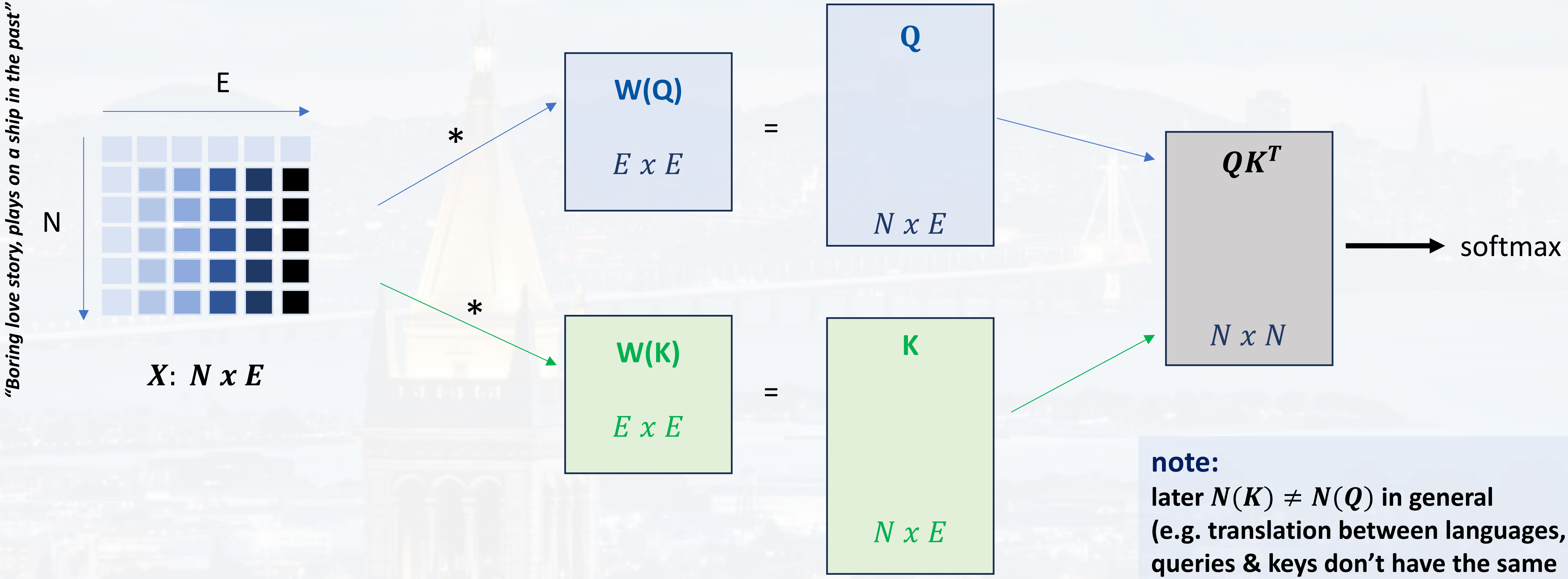
softmax

self attention **key** **value** **query**

$$w_{nm} = \frac{\exp(X_n \mathrm{W(K)} \circ X_m \mathrm{W(Q)})}{\sum_{\mu=1}^{N} \exp(X_n \mathrm{W(K)} \circ X_\mu \mathrm{W(Q)})}$$

N: number of token
E: number of embedding dimensions

output: $$Y_n = \sum_{m=1}^{N} w_{nm} X_m$$



note:
later $N(K) \neq N(Q)$ in general
(e.g. translation between languages, queries & keys don't have the same number of tokens in general etc)

self attention **key** **value** **query**

$$\boxed{w_{nm} = \frac{\exp(X_n W(K) \circ X_m W(Q))}{\sum_{\mu=1}^{N} \exp(X_n W(K) \circ X_\mu W(Q))}}$$

N: number of token
E: number of embedding dimensions

output:

$$Y_n = \sum_{m=1}^{N} w_{nm}\, X_m = \sum_{m=1}^{N} \frac{\exp(X_n W(K) \circ X_m W(Q))}{\sum_{\mu=1}^{N} \exp(X_n W(K) \circ X_\mu W(Q))} X_m$$

$$= \sum_{m=1}^{N} softmax\left(\boldsymbol{Q}_n \boldsymbol{K}_m{}^{\boldsymbol{T}}\right) X_m$$

$$\rightarrow \sum_{m=1}^{N} softmax\left(\boldsymbol{Q}_n \boldsymbol{K}_m{}^{\boldsymbol{T}}\right) X_m \mathbf{W(V)} \qquad \rightarrow \mathbf{Y} = softmax\left(\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}\right) \mathbf{V} \quad \textbf{value}$$

summarizing the characteristics of the movie by using the movie itself

The output would be a movie, generated by weighted contributions of those movies (values), where the keys match well with the query

self attention **key** **value** **query**

$$w_{nm} = \frac{\exp(X_n W(K) \circ X_m W(Q))}{\sum_{\mu=1}^{N} \exp(X_n W(K) \circ X_\mu W(Q))}$$

N: number of token
E: number of embedding dimensions

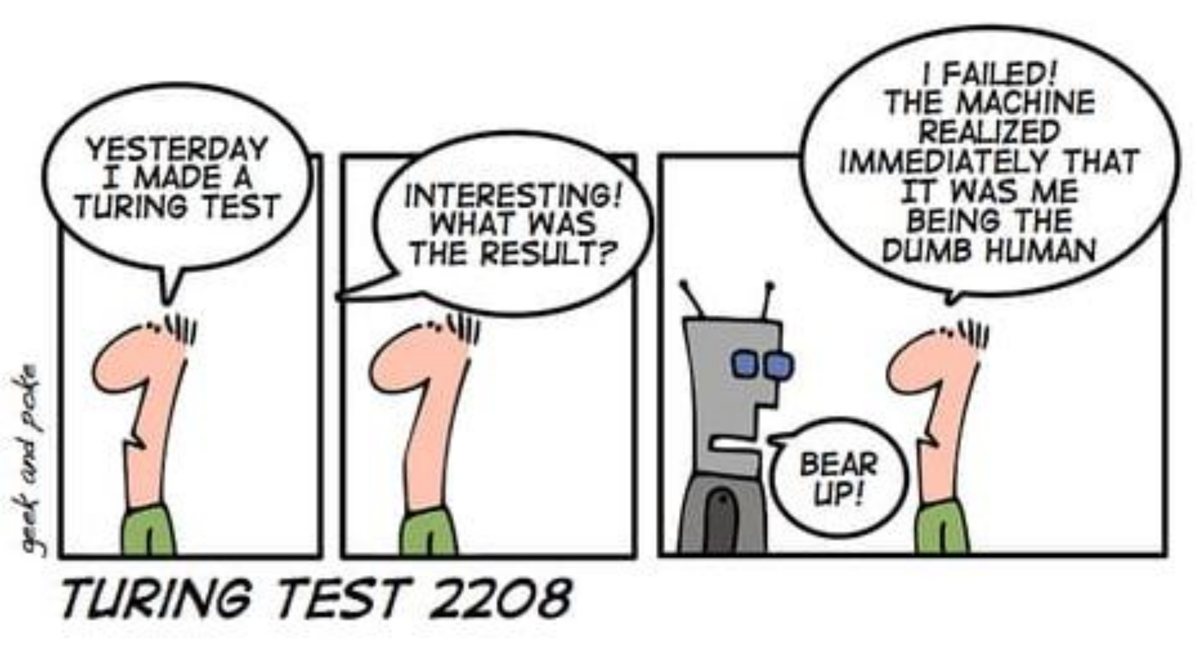
output: $$Y_n = \sum_{m=1}^{N} w_{nm} X_m$$

$$\rightarrow \mathrm{Y} = \boldsymbol{softmax}(\boldsymbol{QK^T})\mathbf{V}$$

note:

- dot product scales with E, therefore normalization: $softmax\left(\frac{\boldsymbol{QK^T}}{\sqrt{E}}\right)$
- cross attention:
  - eg. key a phrase in language A, query in language B
  - **encoder/decoder** structure, see next slides
- we want to recognize many underlying pattern → **multiple attention** layers in parallel → transformer

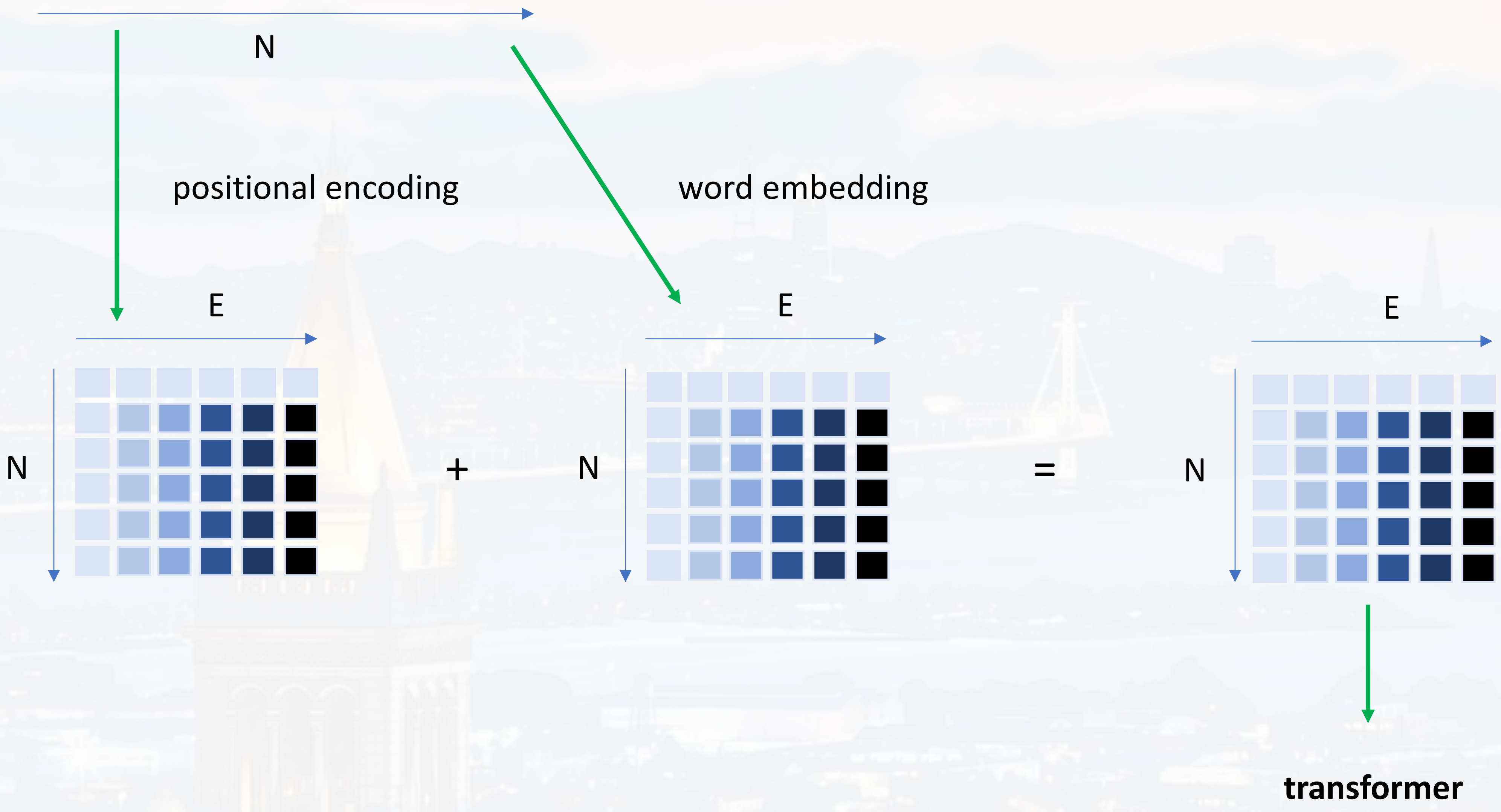
Outline



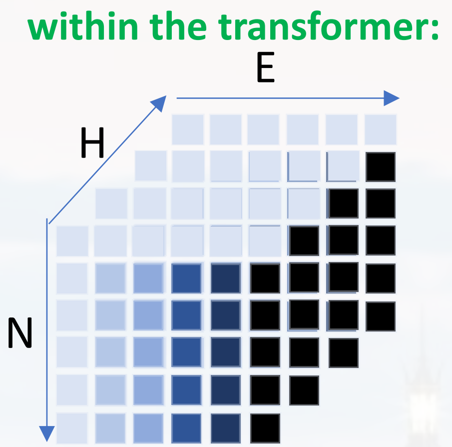
- Introduction
- Bigram and MAP
- Positional Encoding
- Word Embedding
- Attention
- **Transformer Architecture**

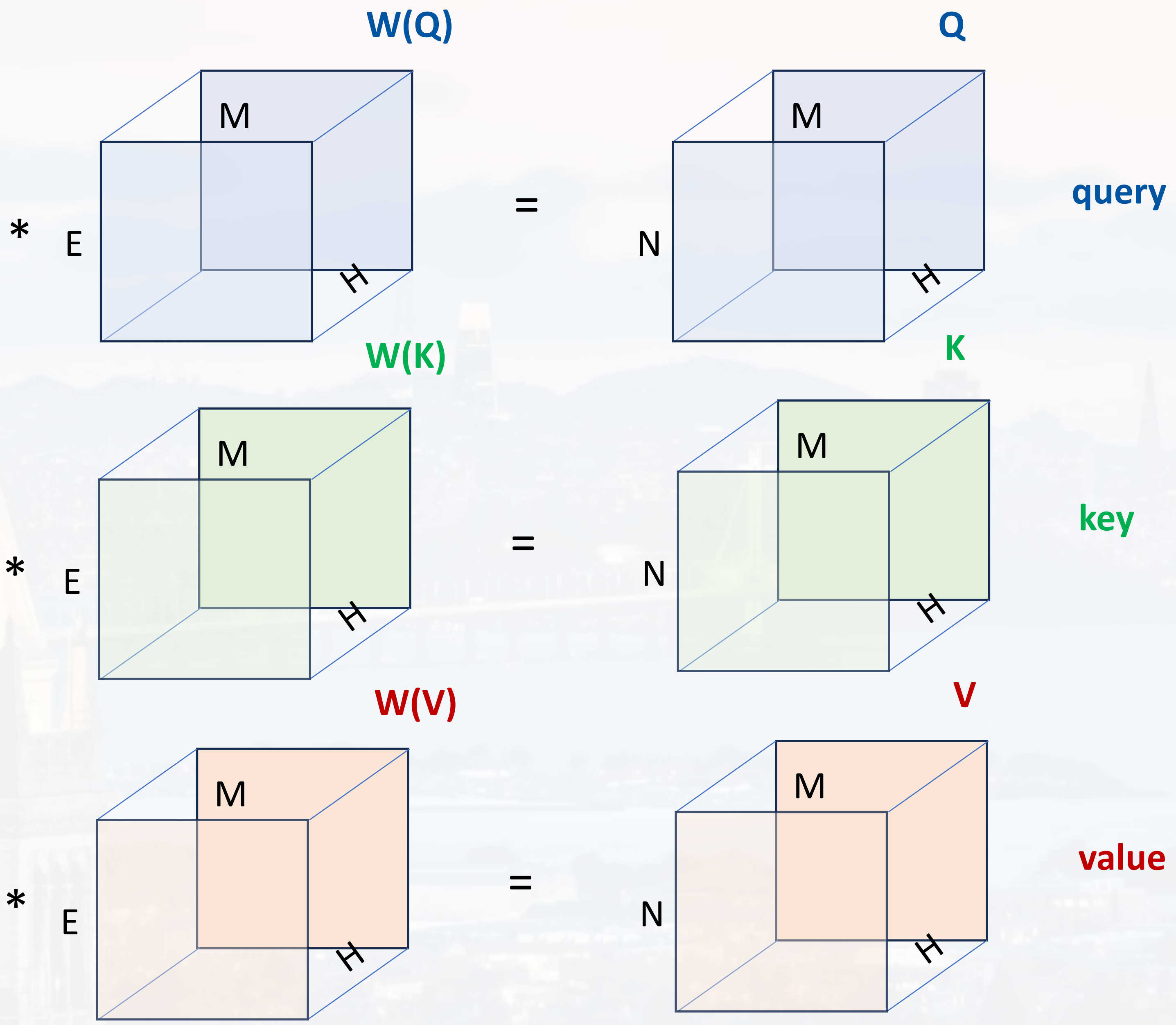
N:	**number of token**
E:	**number of embedding dimensions**

“The cat jumped on the roof.”



attention:





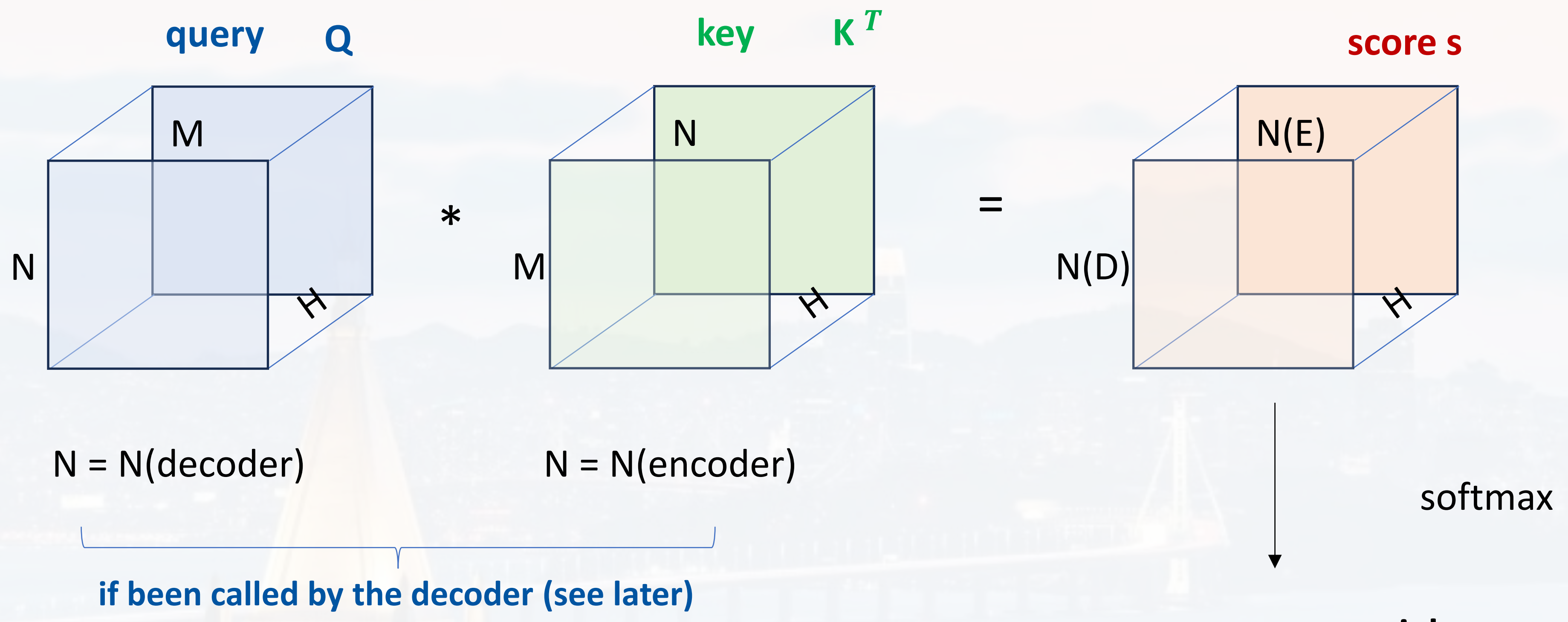
W(Q), W(K), W(V): learnable

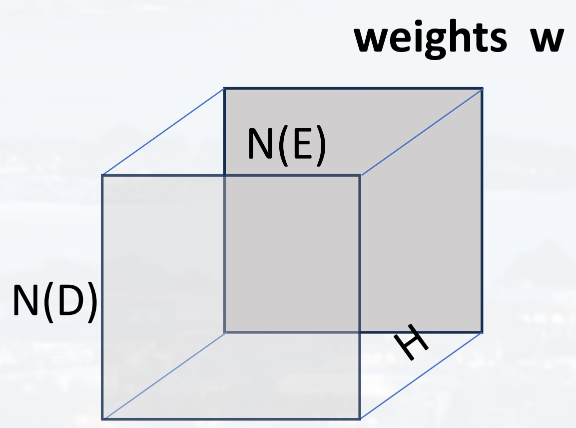
N:	**number of token**
E:	**number of embedding dimensions**
H:	**number of heads (= 8)**
M:	**head size (= 64)**

LLM & Transformer: Transformer Architecture

within the transformer:

attention:





W(Q), W(K), W(V): learnable

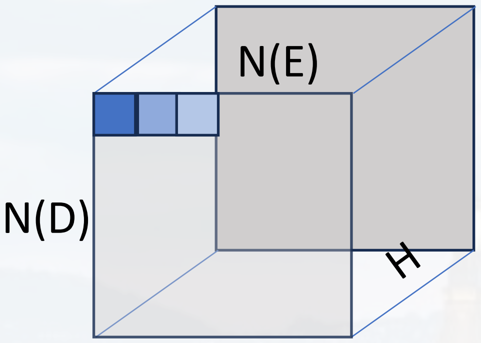
N:	**number of token**
E:	**number of embedding dimensions**
H:	**number of heads (= 8)**
M:	**head size (= 64)**

within the transformer:

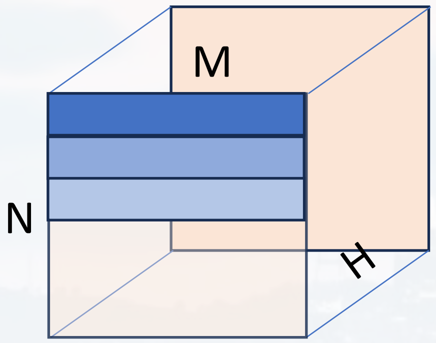
N:	**number of token**
E:	**number of embedding dimensions**
H:	**number of heads (= 8)**
M:	**head size (= 64)**

attention:

weights w

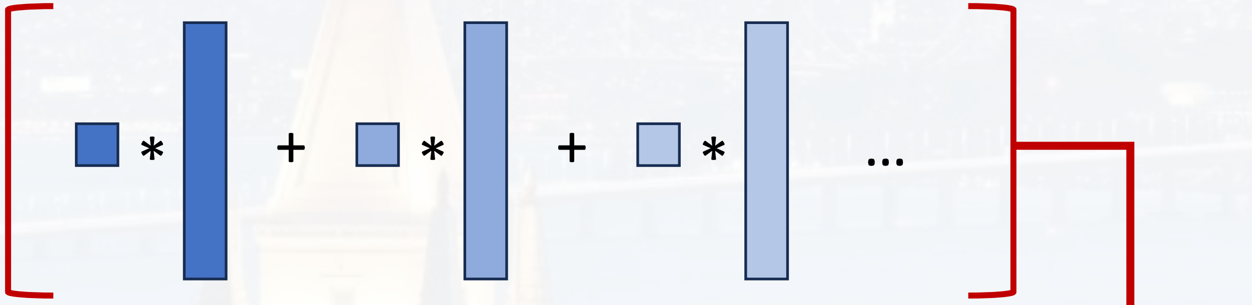


value V

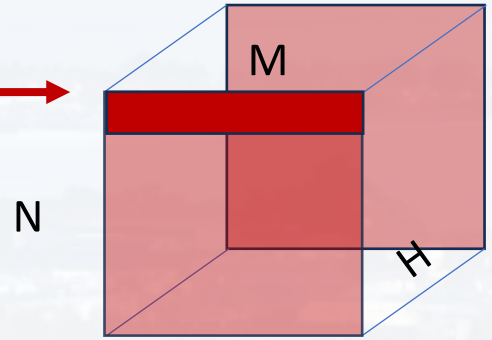


N(E) = N(encoder)
N(D) = N(decoder) , see later

$$Y_n = \sum_{m=1}^{N} w_{nm} X_m$$



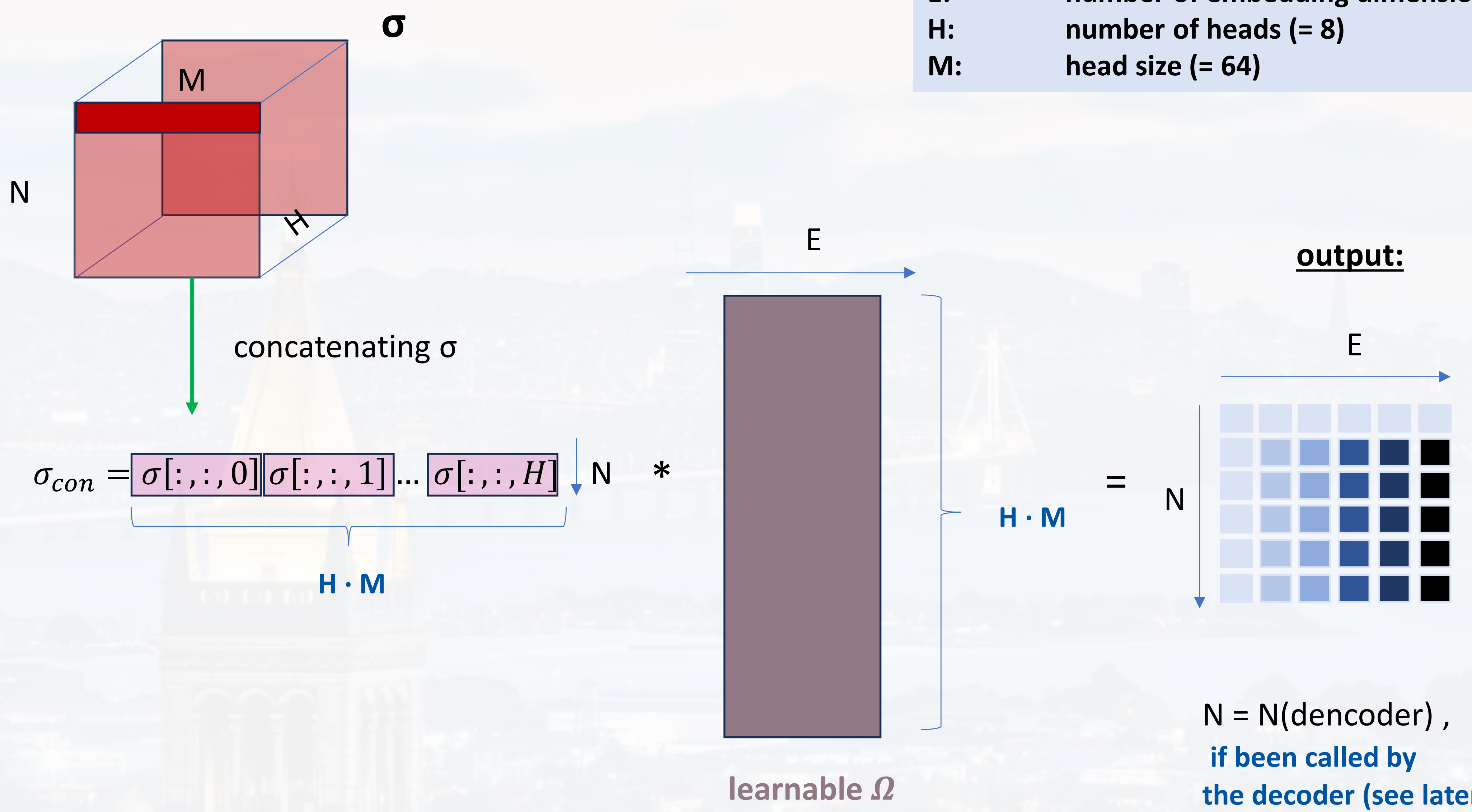
σ



$$\sigma[i,:,k] = \sum_j w[i,j,k] * v[j,:,k]$$

within the transformer:

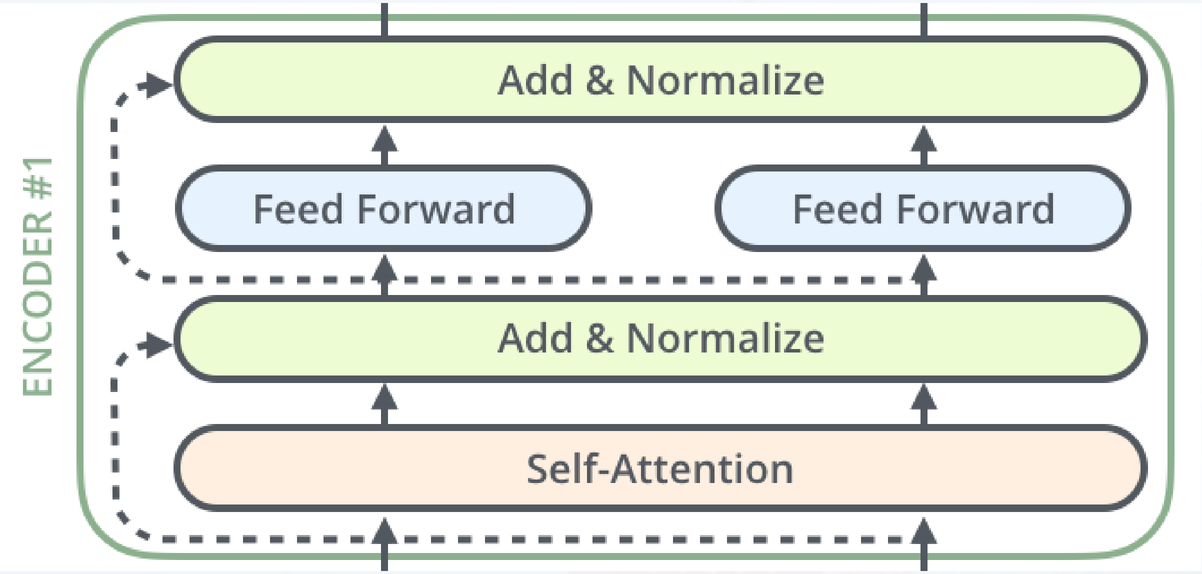
N:	**number of token**
E:	**number of embedding dimensions**
H:	**number of heads (= 8)**
M:	**head size (= 64)**

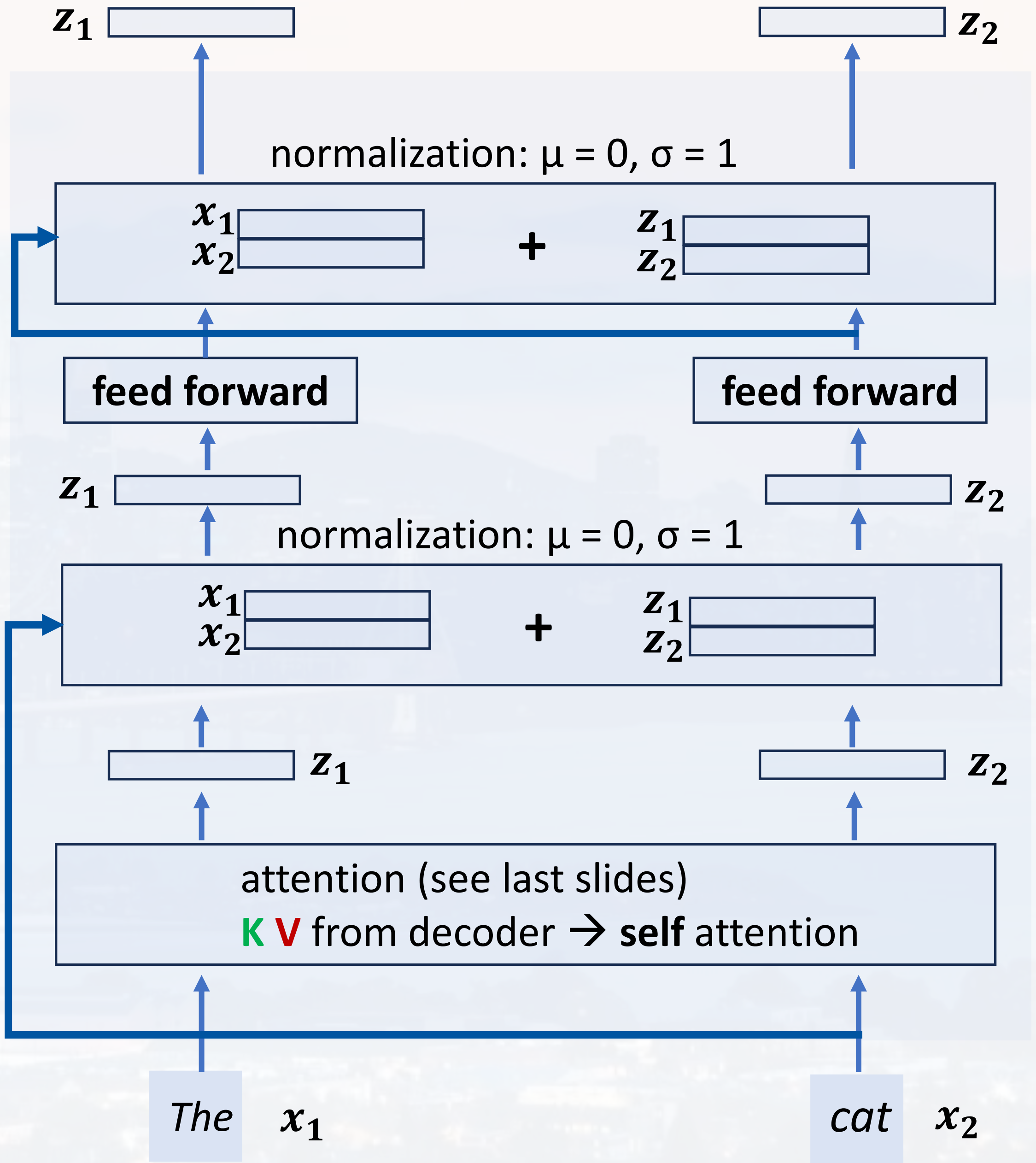


N = N(dencoder) ,
if been called by the decoder (see later)

that was attention → now: encoder

https://jalammar.github.io/illustrated-transformer/



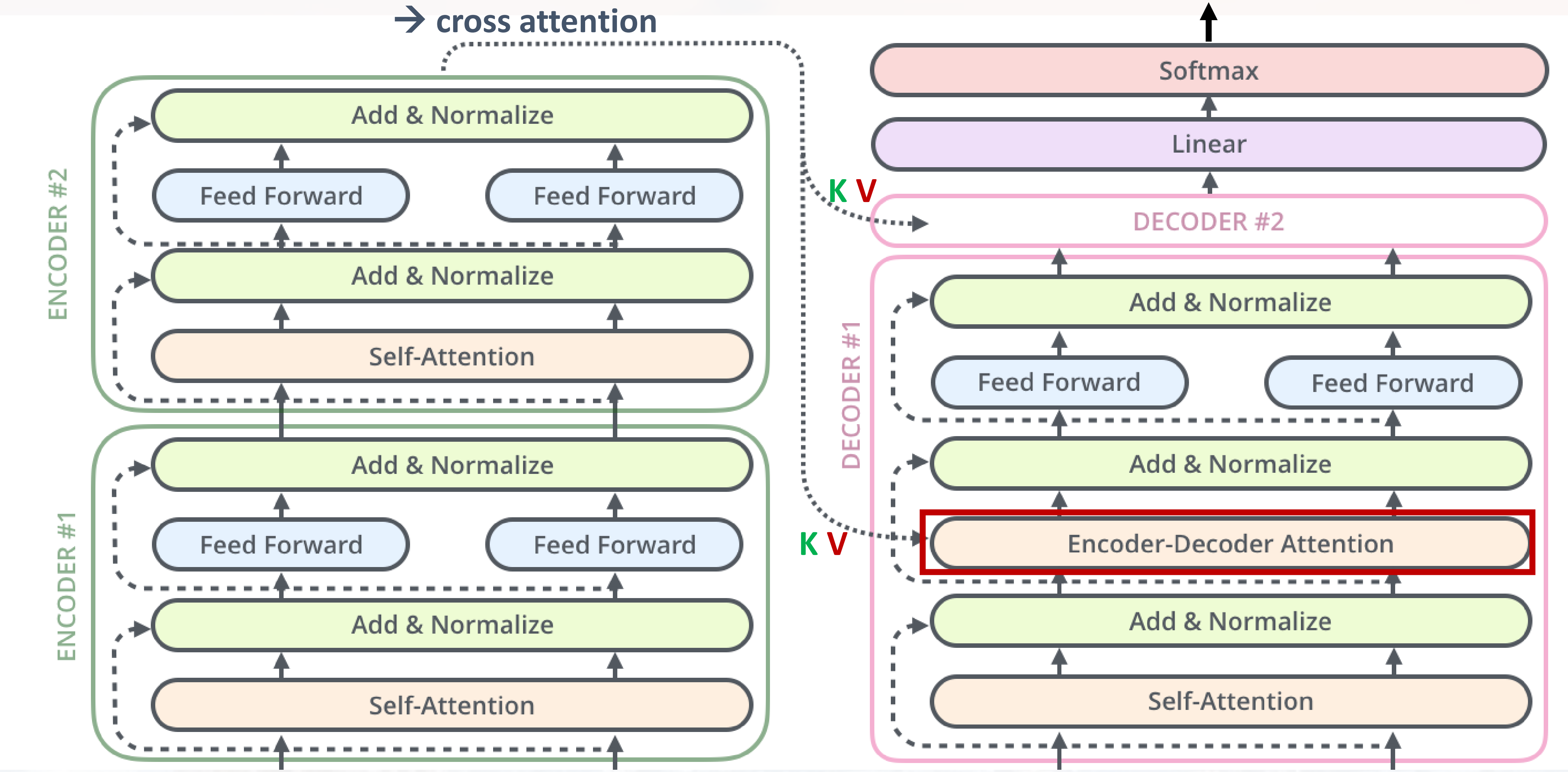


positional encoding + word embedding

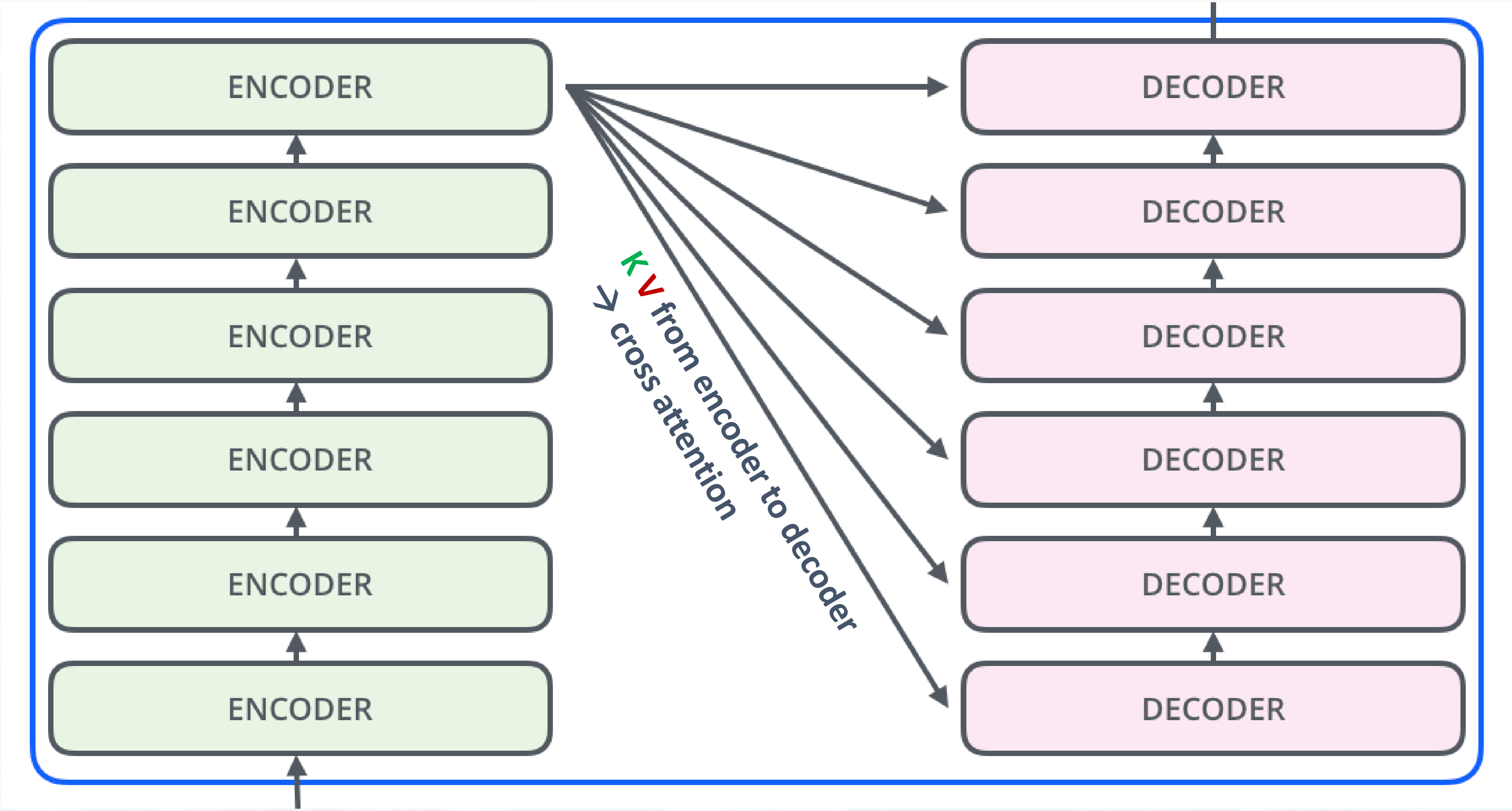
LLM & Transformer: Transformer Architecture

generated token: **argmax** from softmax
(length = vocab size)

K V from encoder to decoder
→ cross attention

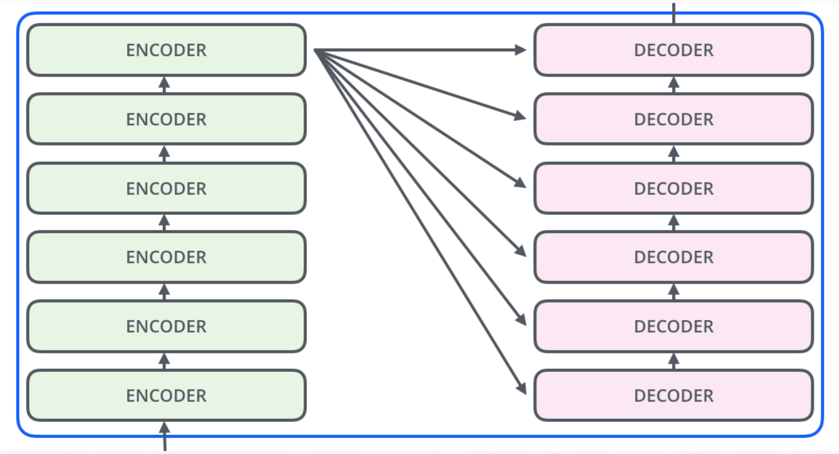


https://jalammar.github.io/illustrated-transformer/



https://jalammar.github.io/illustrated-transformer/

K V from encoder to decoder
→ cross attention

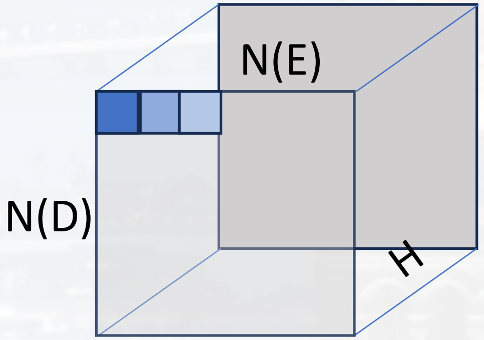


general: $N(E) \neq N(D)$

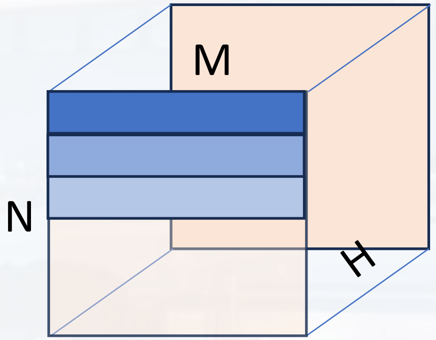
English:	Let there be light.	N(E) = 4
German:	Es werde Licht.	N(D) = 3
Latin:	Fiat lux.	N(D) = 2
Hebrew:	yehi ,or	N(D) = 2

attention:

weights w



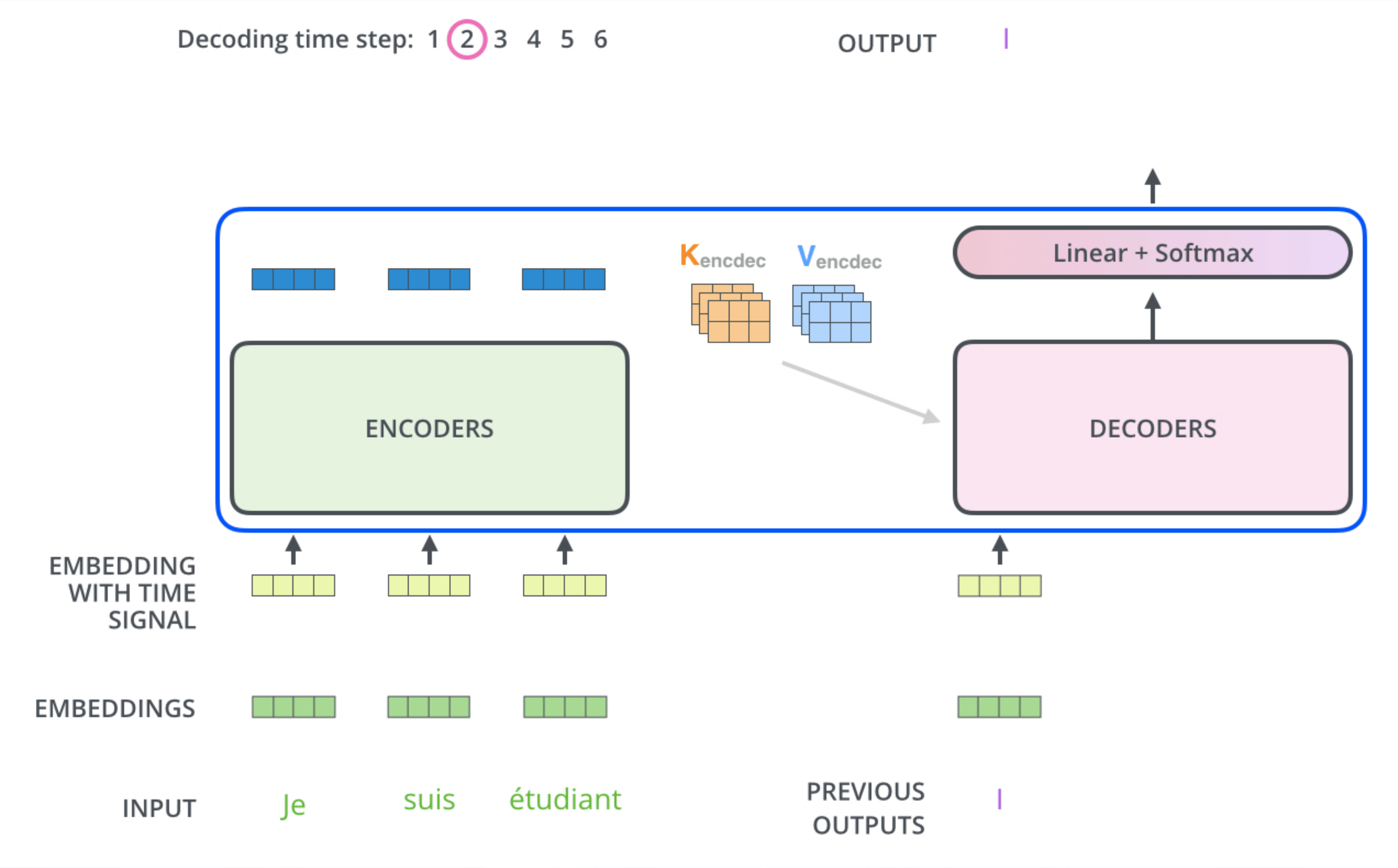
value V

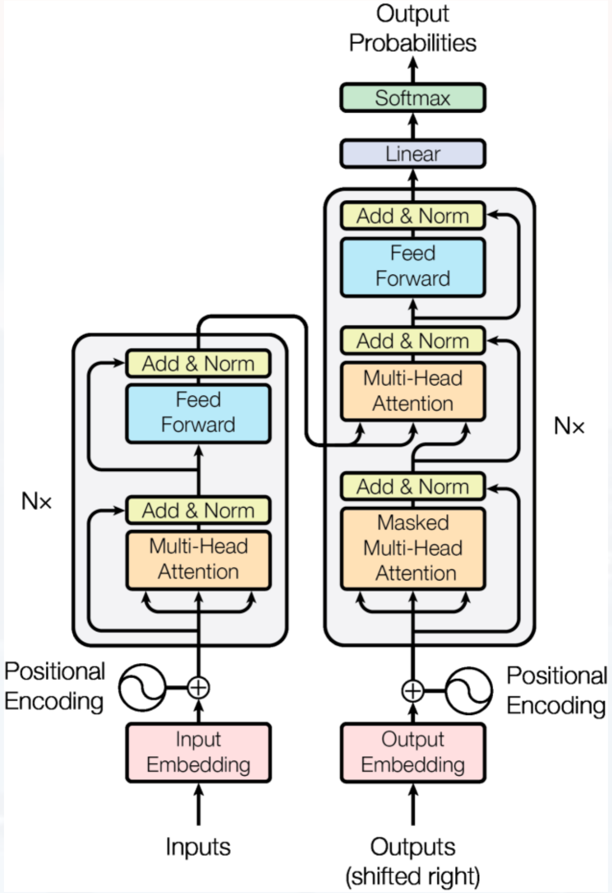


N = N(encoder) , **if been called by the decoder**

https://jalammar.github.io/illustrated-transformer/

LLM & Transformer: Transformer Architecture





https://jalammar.github.io/illustrated-transformer/

Attention Is All You Need (Vaswani et al, 2017) mask: $\mathcal{M}$

improvements:

masked attention $\mathrm{Y} = \boldsymbol{softmax}(\boldsymbol{Q}\boldsymbol{K}^T)\mathbf{V} \rightarrow \mathrm{Y} = \boldsymbol{softmax}(\boldsymbol{Q}\boldsymbol{K}^T + \boldsymbol{\mathcal{M}})\mathbf{V}$

causal masking: $w_{nn+k} = 0 \; \forall \; k > 0$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

token at position t should only consider **previous token** for predicting token at $t + 1$ (natural languages)

padding masking: batches of sequences might have different lengths,
→ shorter sequences are padded with special tokens.
→ model learns to ignore padding tokens
→ for inference

individual masking: we often know that some token can't appear after each other (natural languages)

improvements:

sampling strategies

vanilla: returning most **probable token**, from which we calculate the probabilities for the next token and return the most likely one etc

beam search: we store b (= **beam width**) sequences of length n and then return the **most likely sequence**

$$P(X_1\, X_2\, X_3\, X_4\, X_5 \ldots .\, X_n) = P(X_n | X_{n-1} \ldots X_1) P(X_{n-1} | X_{n-2} \ldots X_1) \ldots P(X_1)$$

top K-sampling: consider $\boldsymbol{K}$ **most probable token**
→ renormalize their probabilities
→ draw randomly from these K token

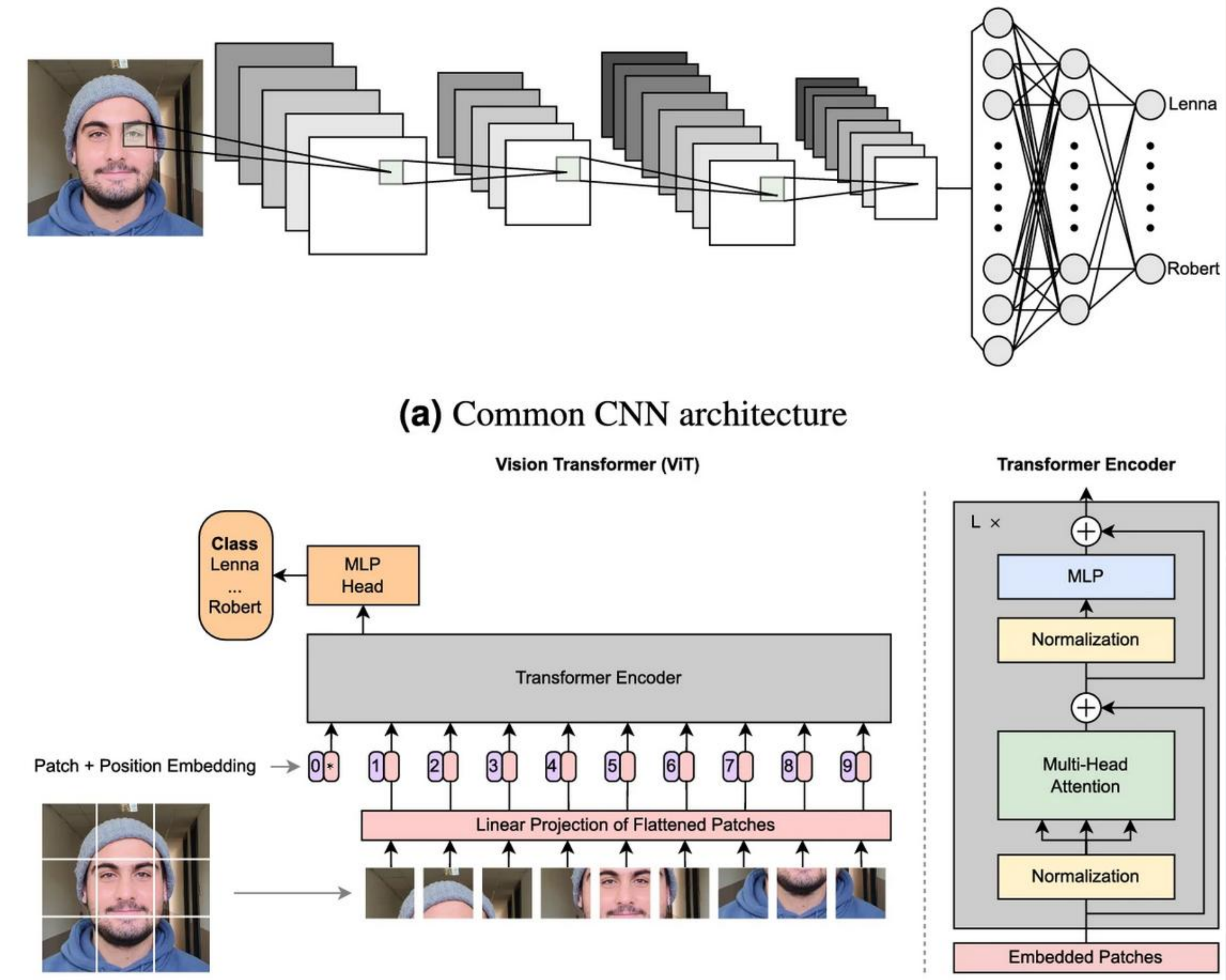
$$p_k = \frac{\exp(\pi_k / T)}{\sum_{k=1}^{K} \exp(\pi_k / T)}$$

$\boldsymbol{\pi_k}$: **probability**
$\boldsymbol{p_k}$: **renormalized probability**
$\boldsymbol{T}$: **"softening" parameter**

top P-sampling: like top K, but for sequences (see beam search)

improvements:

vision transformer



Rodrigo, M., Cuevas, C. & García, N. Comprehensive comparison between vision transformers and convolutional neural networks for face recognition tasks. *Sci Rep* 14, 21392 (2024).

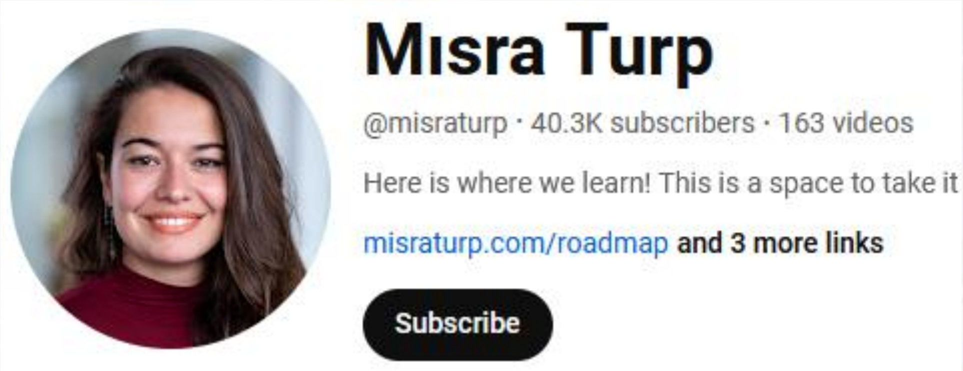
LLM & Transformer: Transformer Architecture

more about transformers:

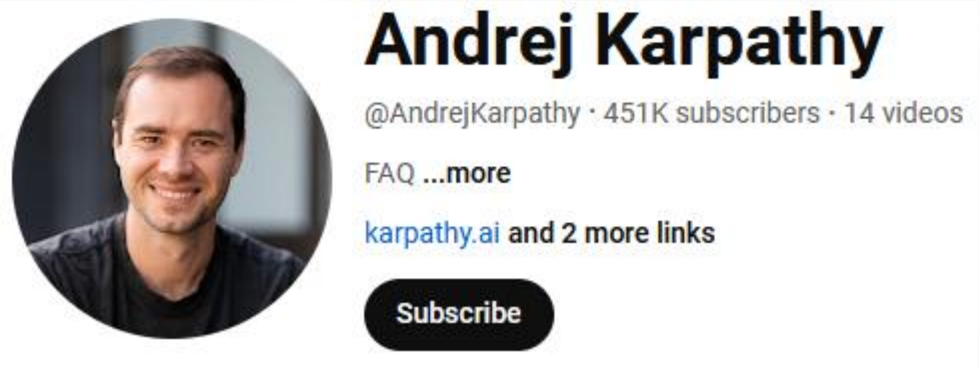
[Jay Alammar](#)

[Interactive Visualization](#)

[transformers intro](#)



[building GPT from scratch!](#)



Thank you very much for your