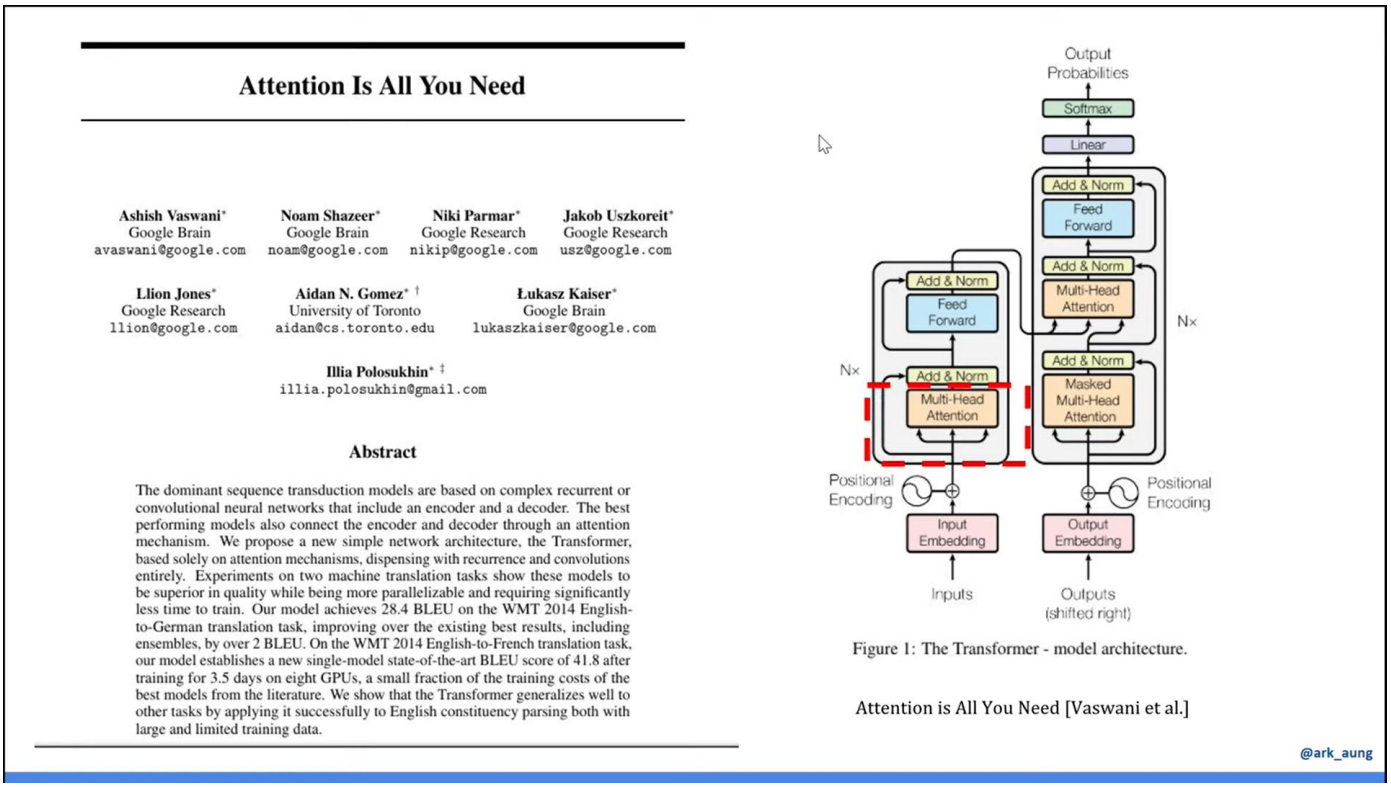
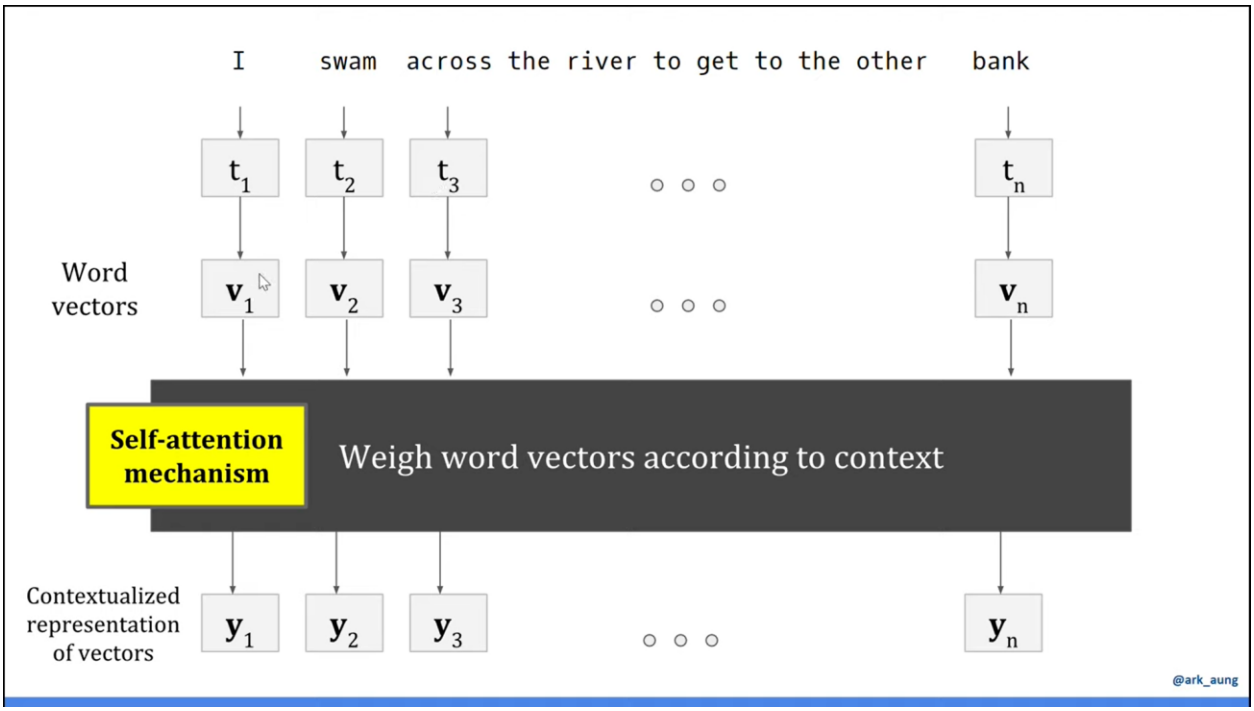


Self attention mechanism add the contextual information in the sentence.

This contextual information added the weight to each words accroding to the context of sentence. Due to this similar word/token/vector have similar direction. NN weigh the words according to their context

It show the word vector that have same semantic meaning are very close to each other.



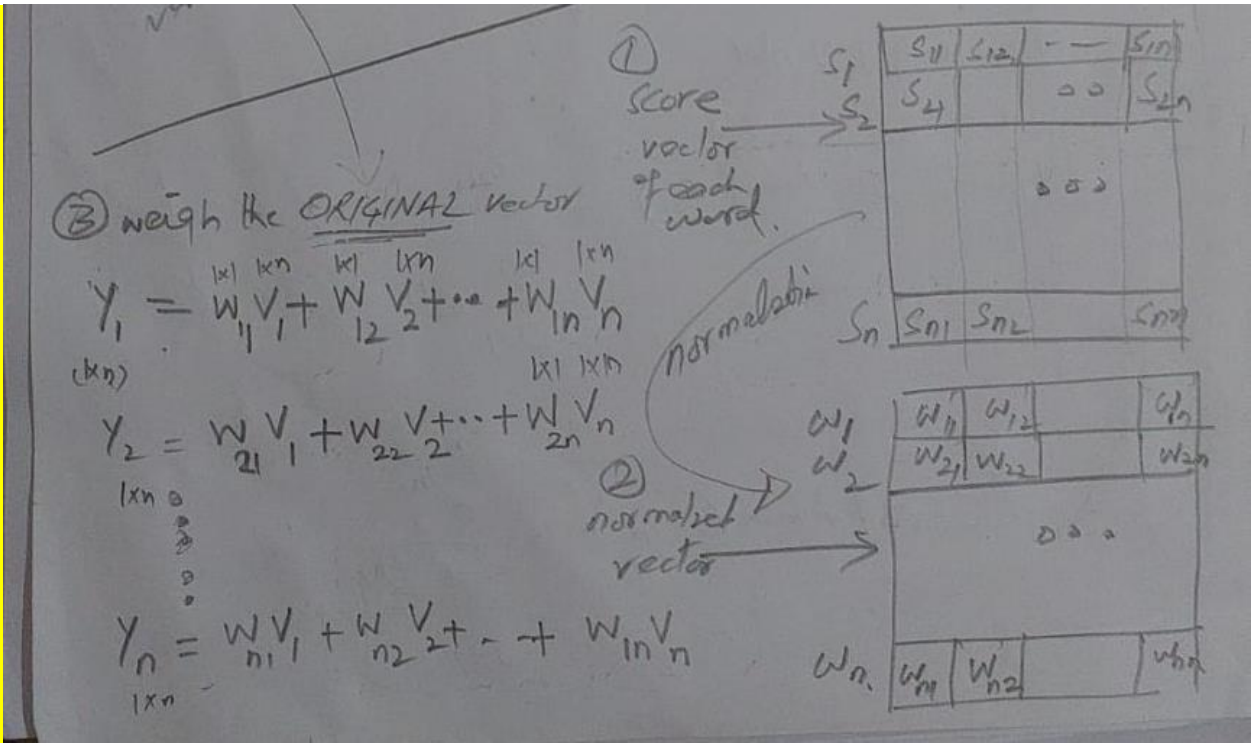
After weighing the word vectors accoriding to their context, we go the contextual representation of the words.

Here after weighing the word (applying the self attention mechanism), bank, swam, and river words have similary/ closer values.

Three steps:

1. Dot product between vectors to get the sclar values
2. Normalzied the values to get the weights
3. Weigh the vectors according the weight and get the contextualized reprsenteation of every vectors.

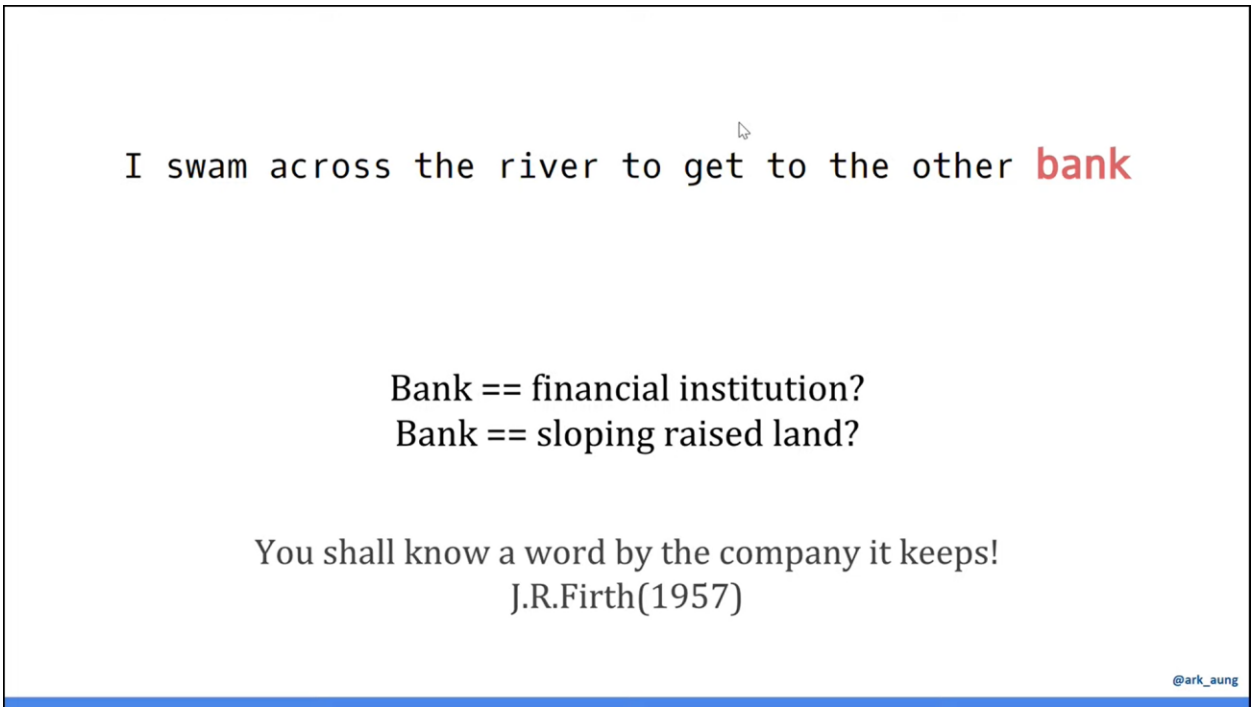
4.



I swam across the river to get to the other **bank**

Bank == financial institution?
Bank == sloping raised land?

@ark_aung



Q. What is the meaning of work bank?

A. You shall know a word by the company it keep. .JR Firth(1957)

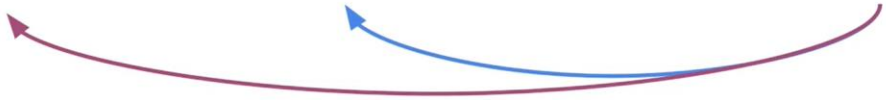
Therefore, bank is not about the financial intitute as if we see their neighborhood words.

If the sentence is

"I deposited my paycheck at the bank yesterday."

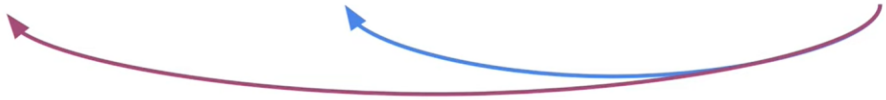
In this sentence, "bank" refers to a financial institution where money is stored, managed, and deposited.

I swam across the river to get to the other **bank**



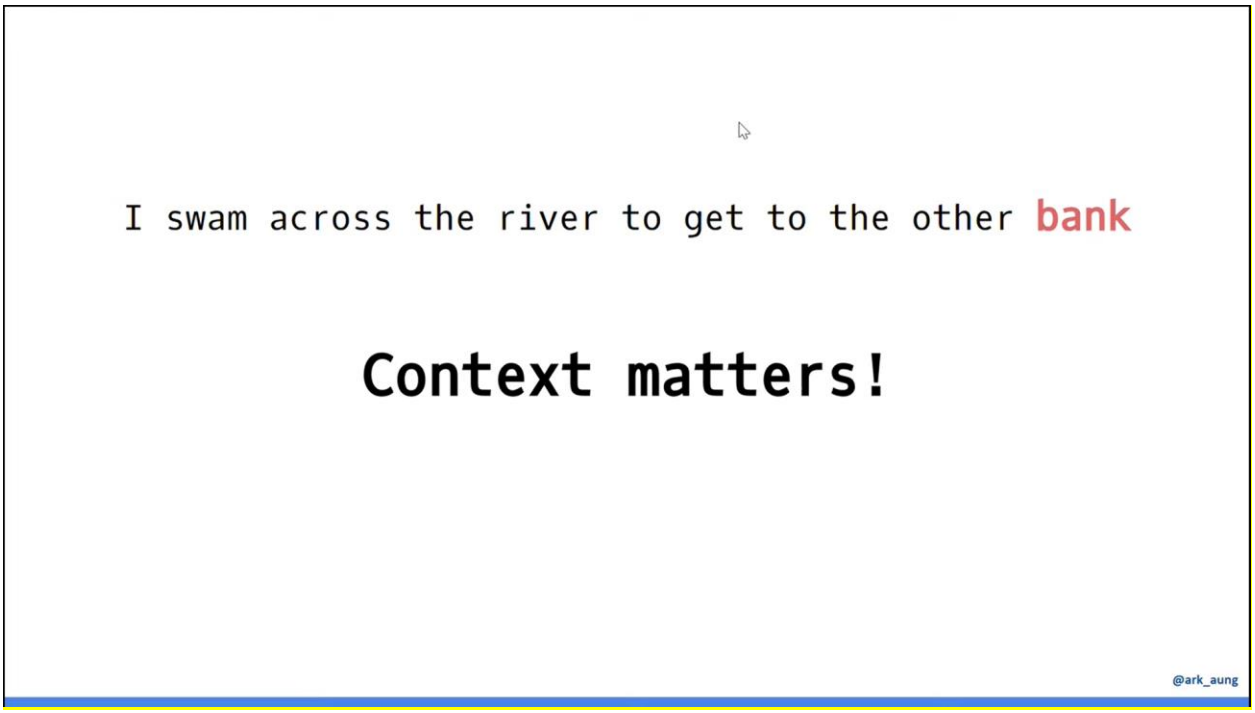
Bank == financial institution?
Bank == sloping raised land?

I swam across the river to get to the other **bank**

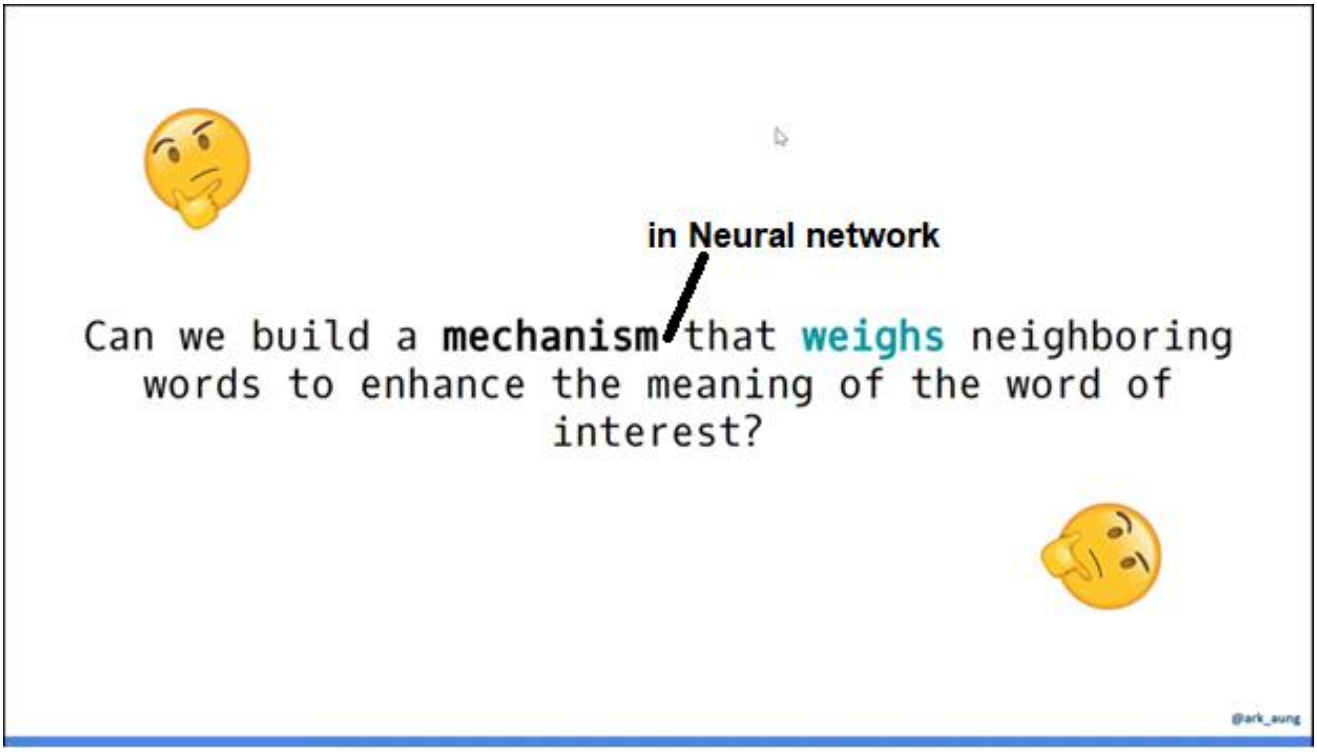


~~Bank == financial institution~~
Bank == sloping raised land

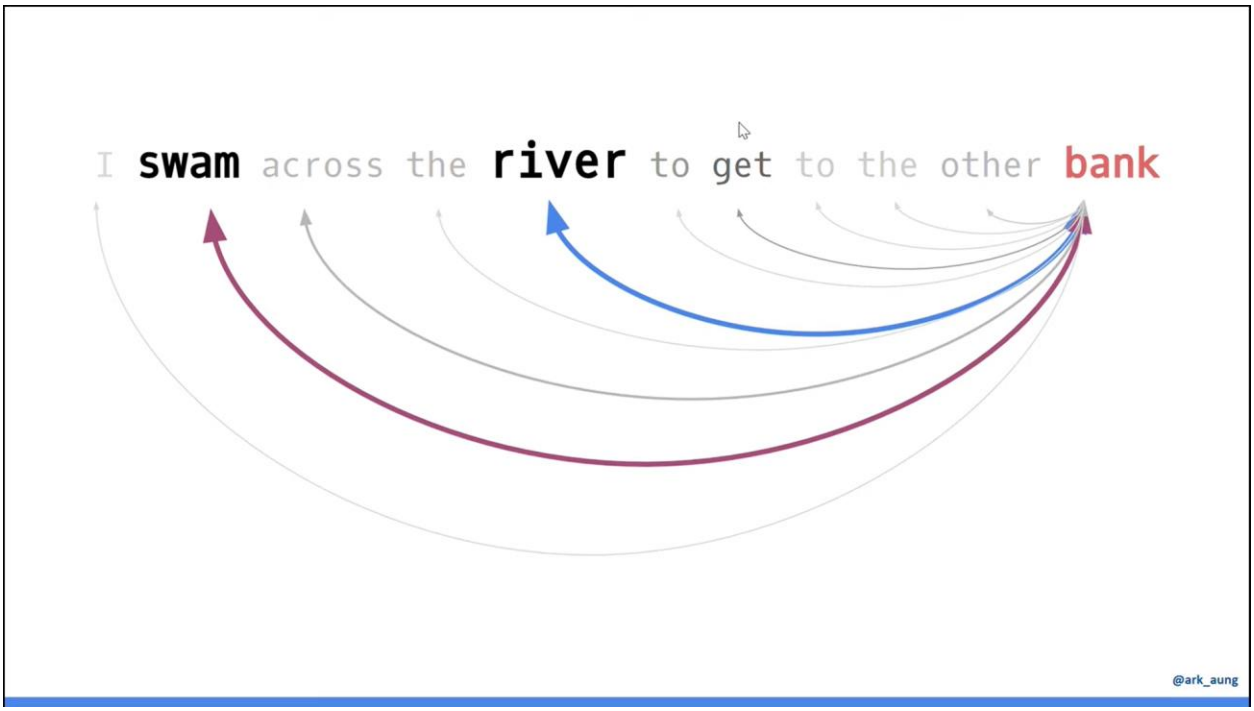
Things changes correpsonding to the contax of the sentences



Contaxt is imprtant to define the meaning of word

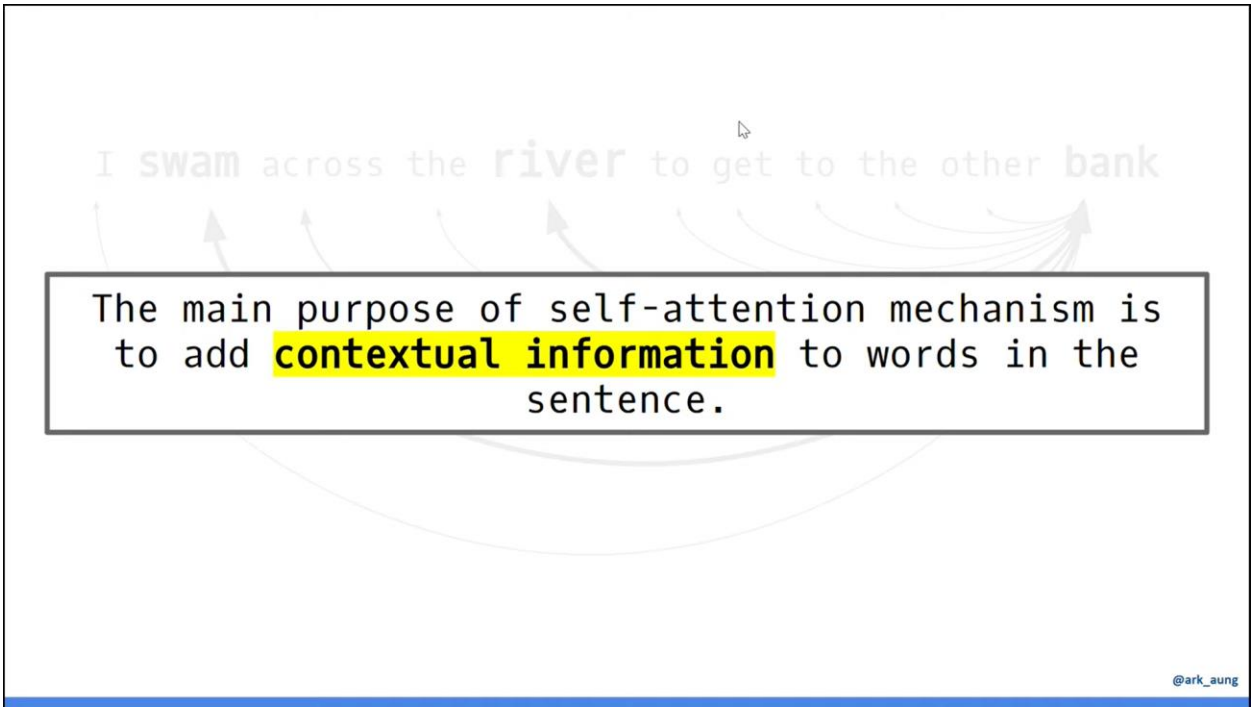


Can we build the mechasim in the NN that weigh the words according to their context

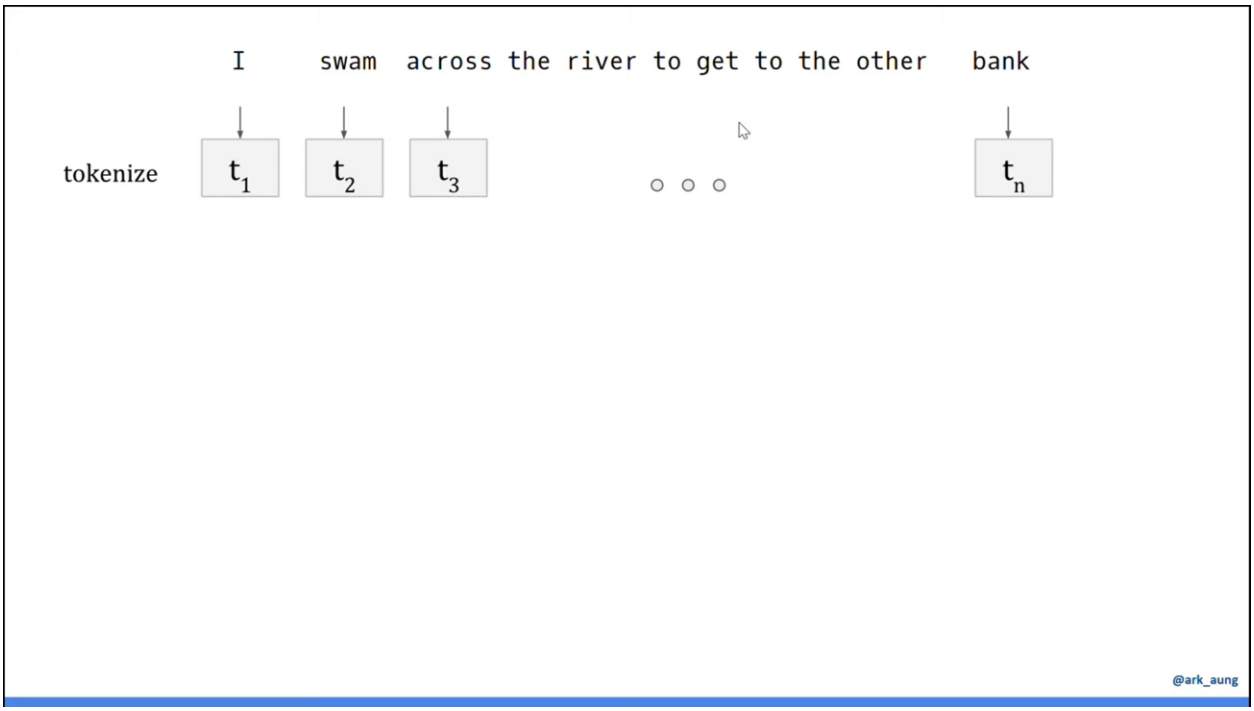


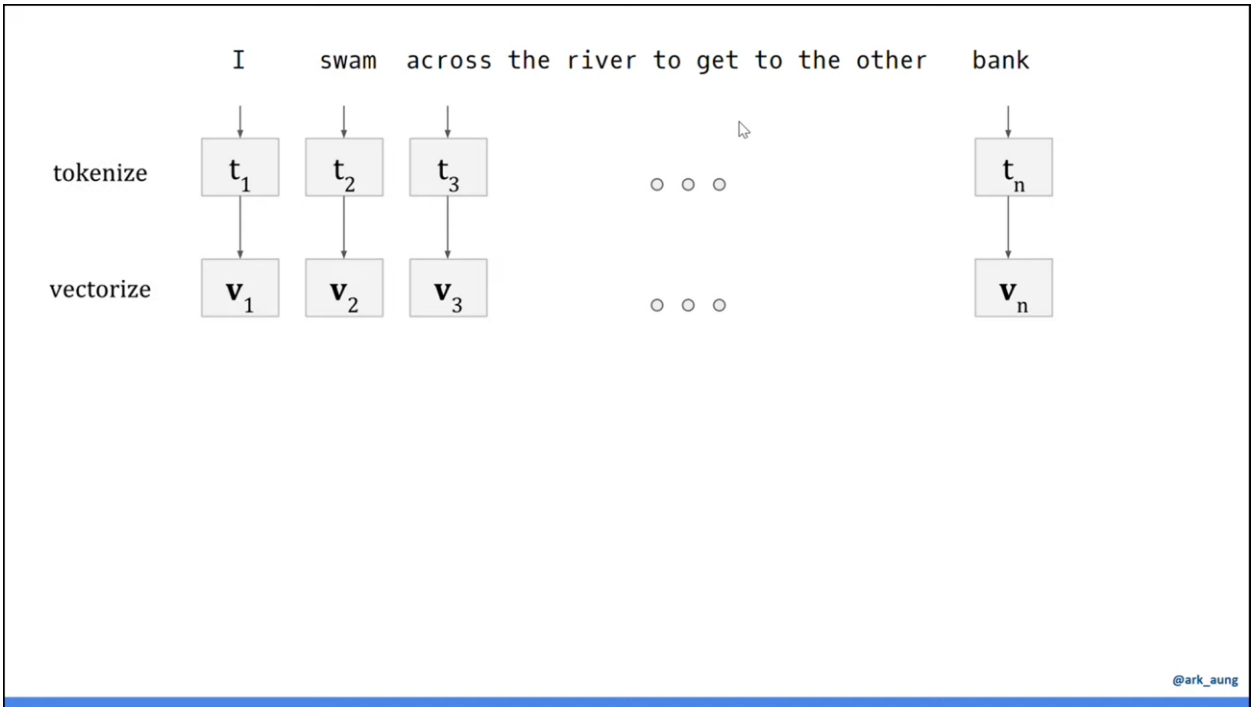
** If we weigh the words according to their context, then the words/vectors like bank, river and swam are very close to each other

Swam and river are seems to be more closely related words to the bank, compare to the other words.

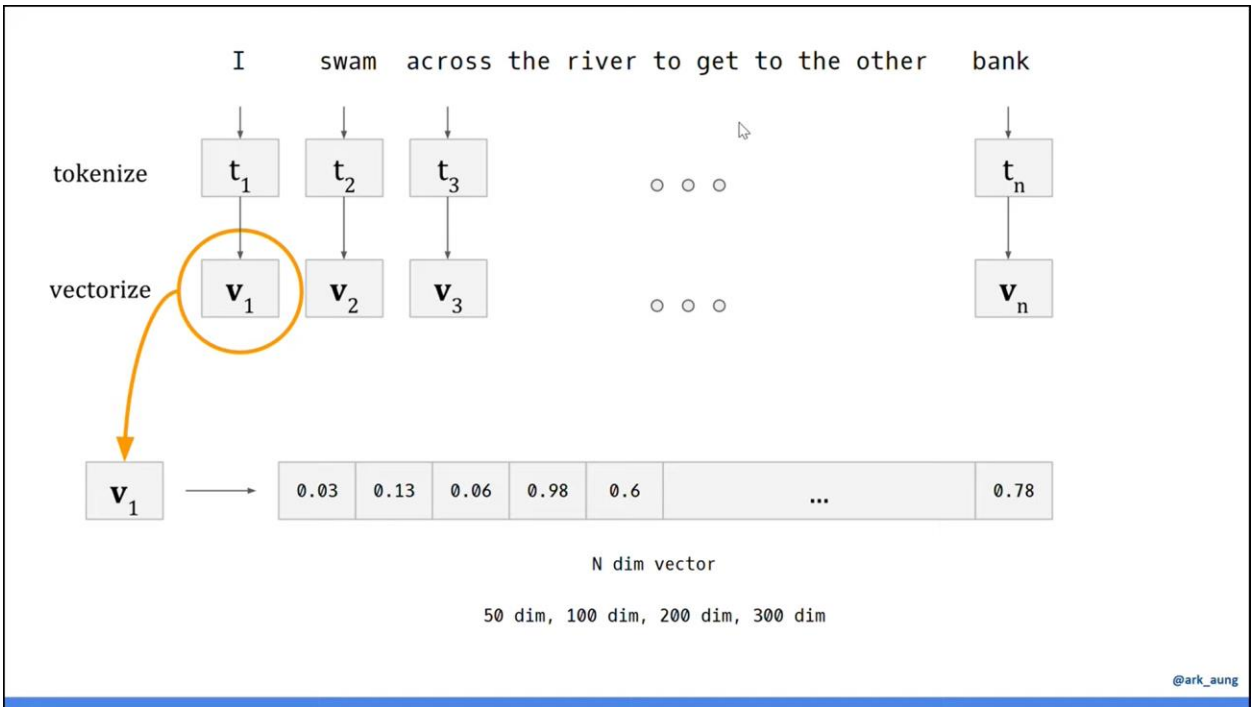


let see more tengible example

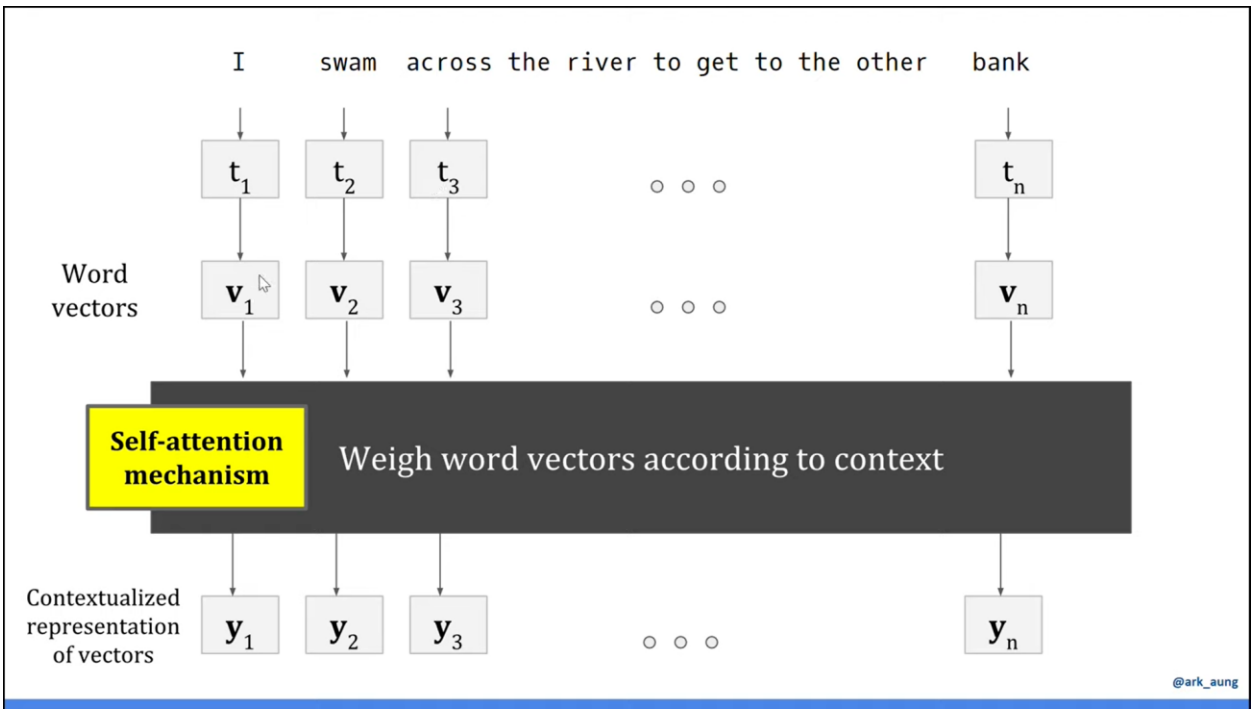




Word to vector conversion : as the NN doesn not understand the words, so need to convert thme into vectors. NN can understand the vectors.



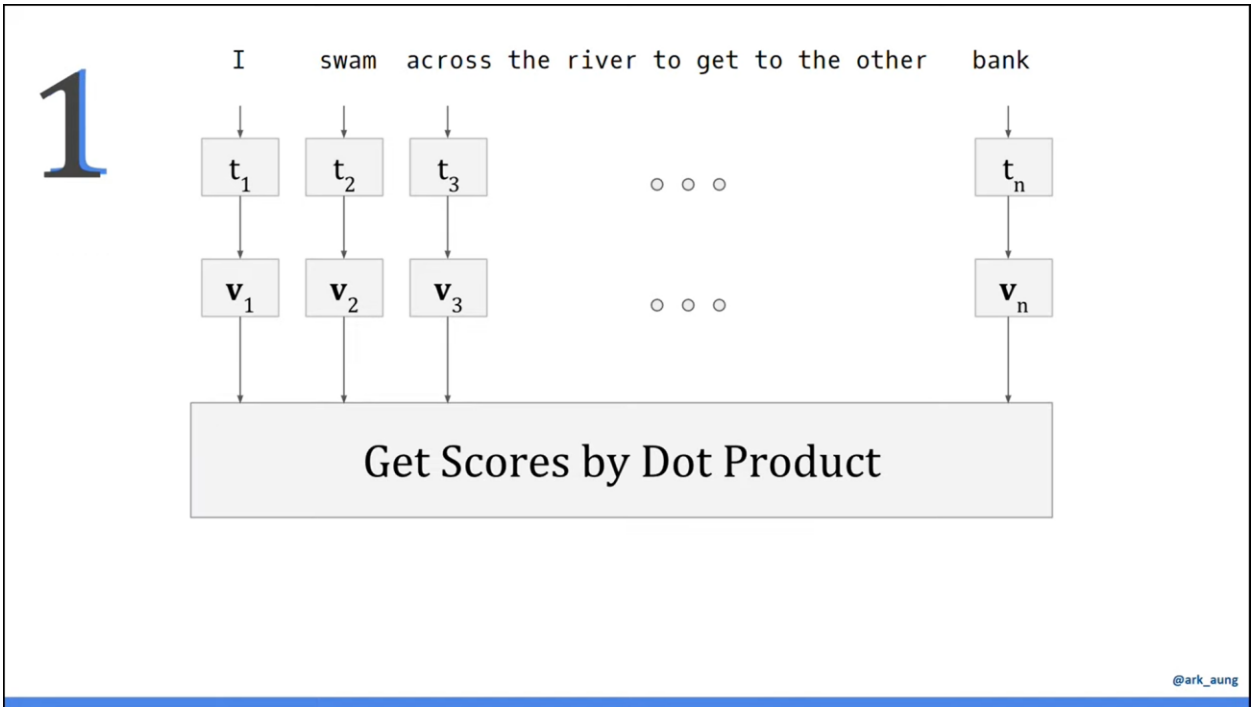
Word with same semantic meaning have the same vector values as shown in the following slides

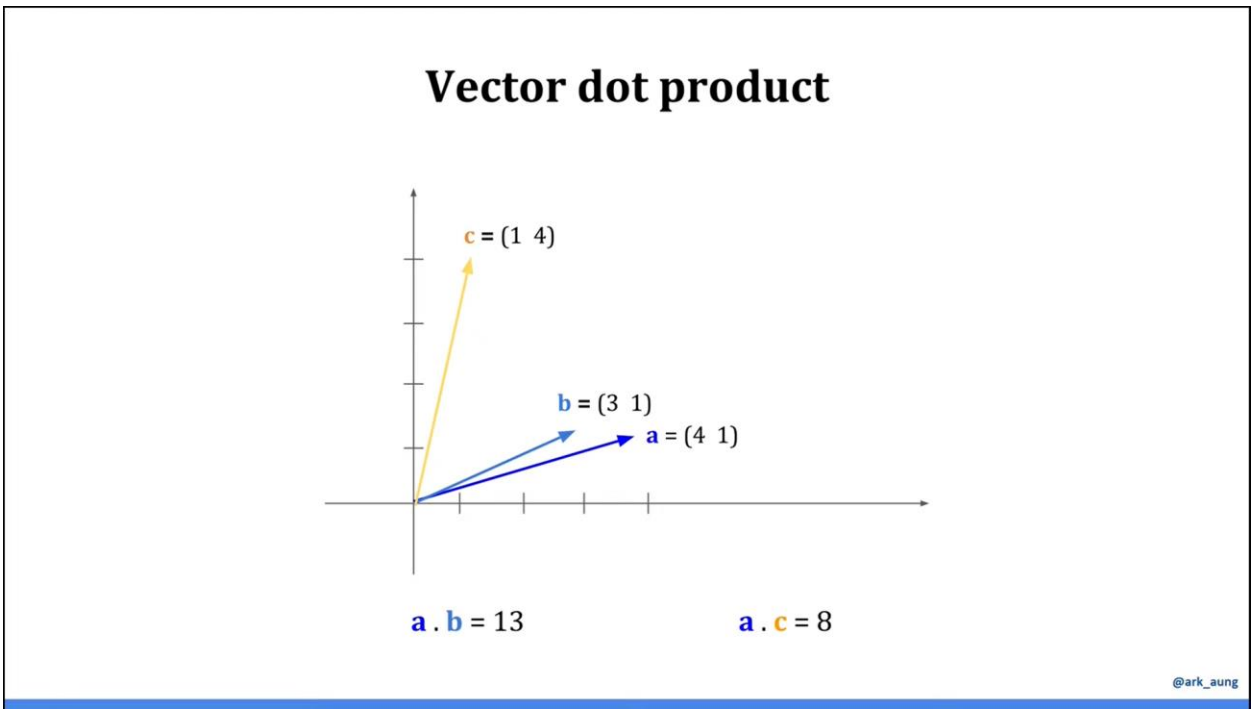


After weighing the word vectors accoriding to their context, we go the contextual representation of the words.

Here after weighing the word (applying the self attention mechanism), bank, swam, and river words have similary/ closer values.

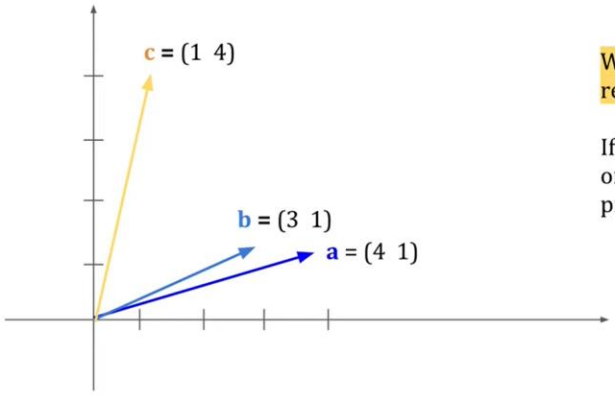
Folloing is the strategy we adapt for getting the contextual representation of vectors





Here the words are in the form of vectors. Dot produce between perpendicular vector is zero if there are not close to each other.

Vector dot product



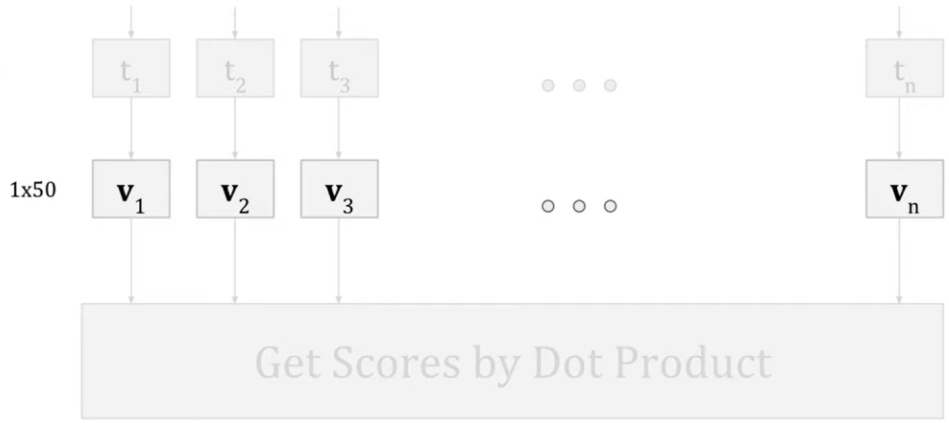
What I want you to remember:

If two vectors are close to one another, their dot product is going to be big

a . **b** = 13 **a** . **c** = 8

@ark_aung

1

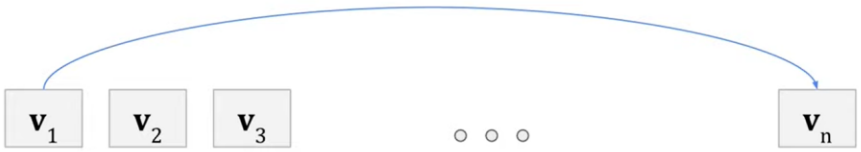


@ark_aung

$$s_{11} = \mathbf{v}_1 \mathbf{v}_1^T$$

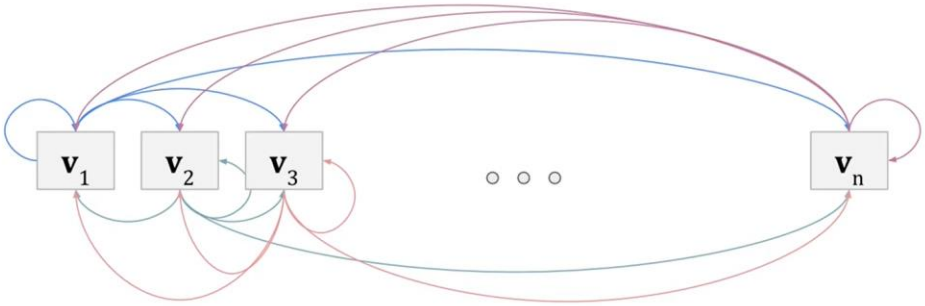
1x50 50x1

$$s_{12} = \mathbf{v}_1 \mathbf{v}_2^T$$



$$s_{1n} = \mathbf{v}_1 \mathbf{v}_n^T$$

@ark_aung



$$s_{11} = \mathbf{v}_1 \mathbf{v}_1^T$$
$$s_{12} = \mathbf{v}_1 \mathbf{v}_2^T$$
$$s_{13} = \mathbf{v}_1 \mathbf{v}_3^T$$

0
0
0

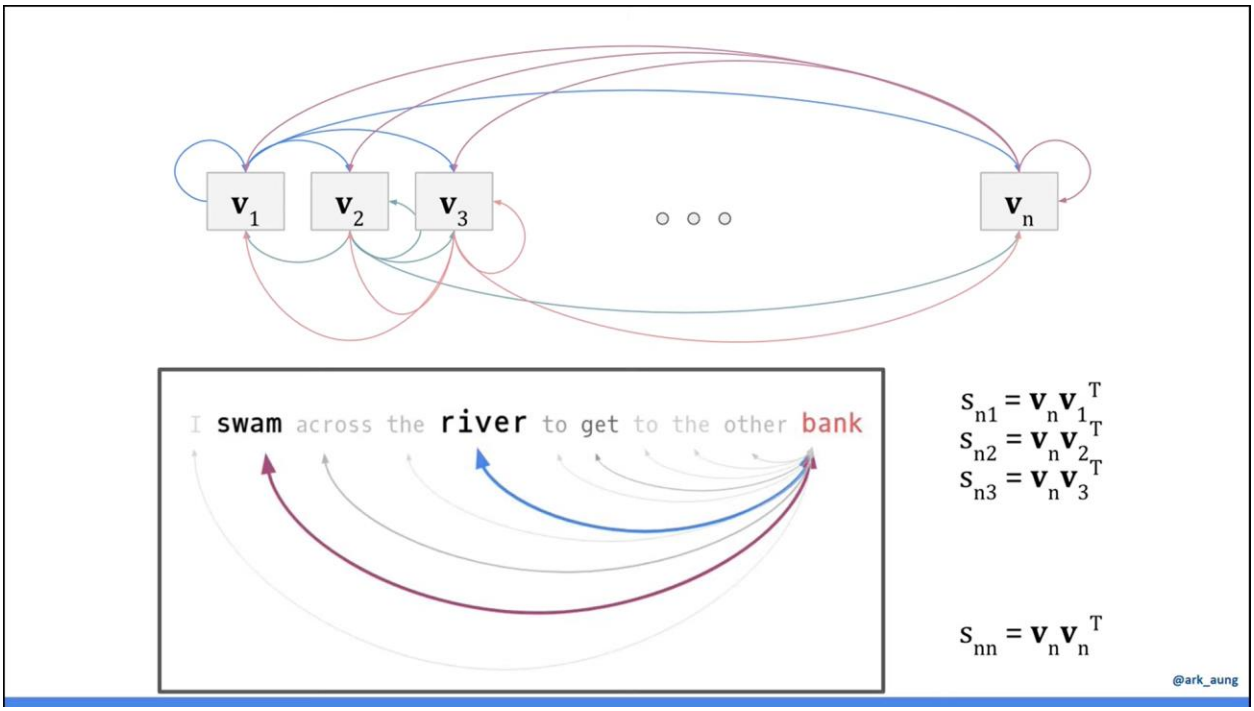
$$s_{1n} = \mathbf{v}_1 \mathbf{v}_n^T$$

$$s_{n1} = \mathbf{v}_n \mathbf{v}_1^T$$
$$s_{n2} = \mathbf{v}_n \mathbf{v}_2^T$$
$$s_{n3} = \mathbf{v}_n \mathbf{v}_3^T$$

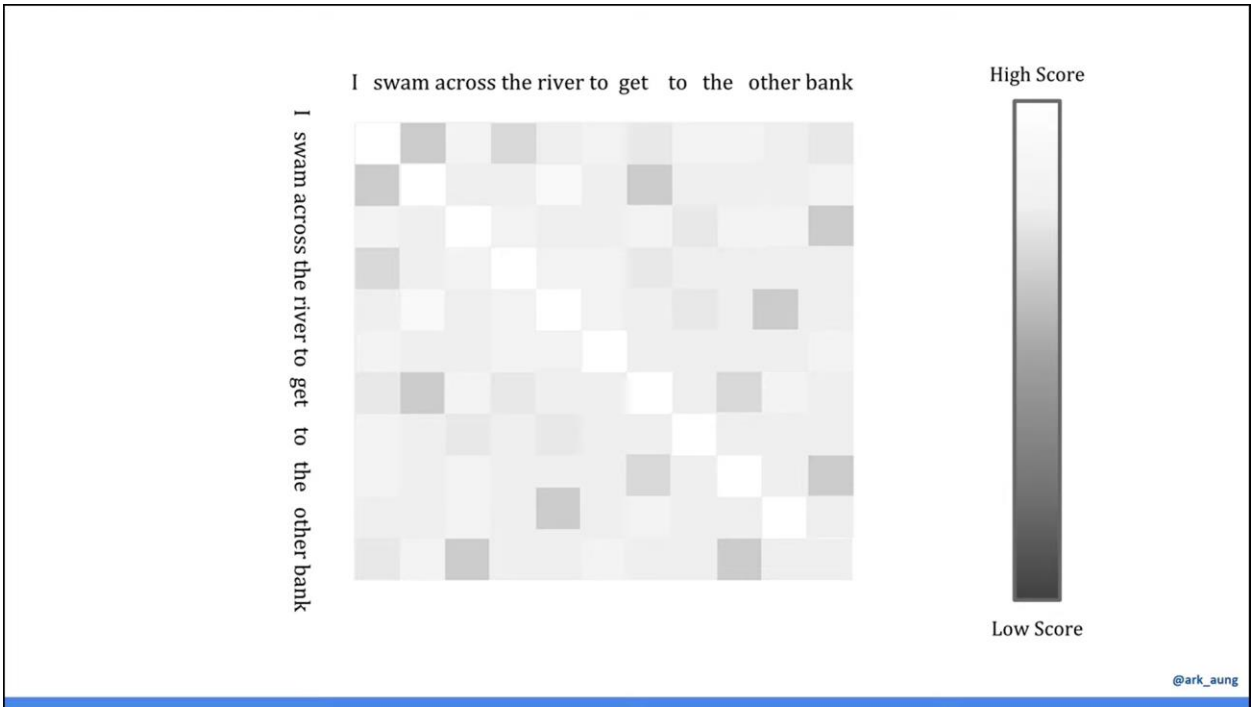
0
0
0

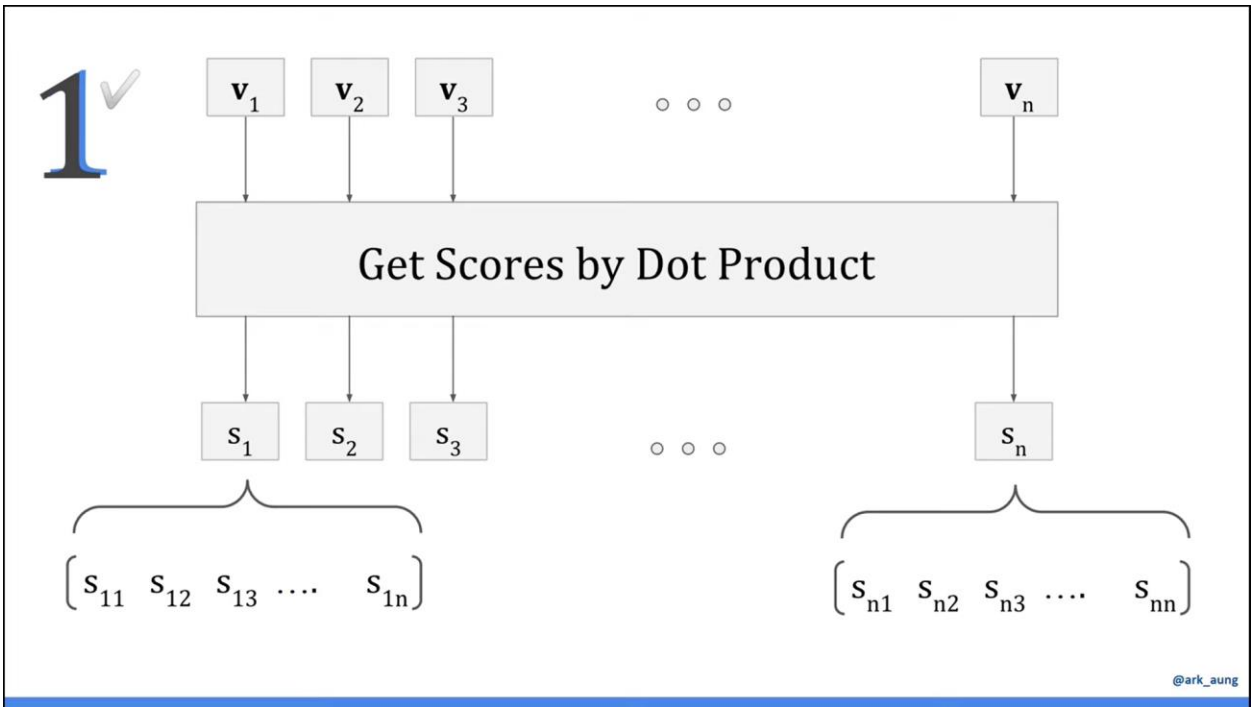
$$s_{nn} = \mathbf{v}_n \mathbf{v}_n^T$$

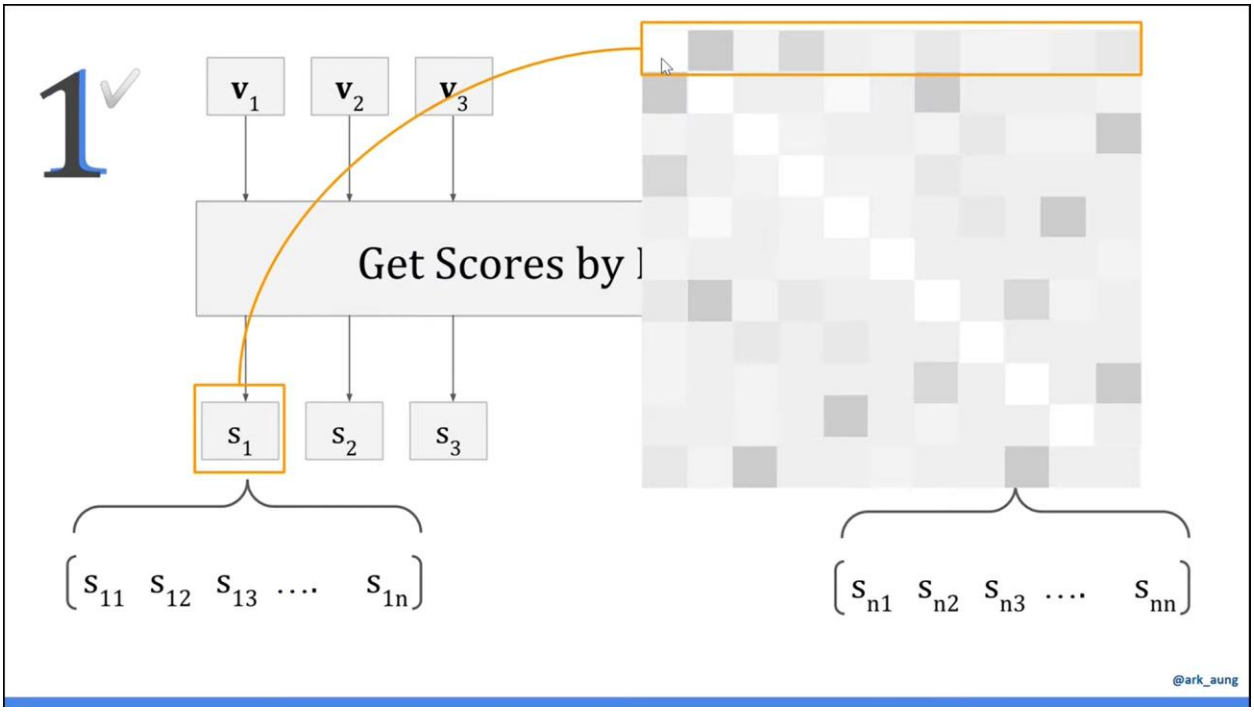
@ark_aung

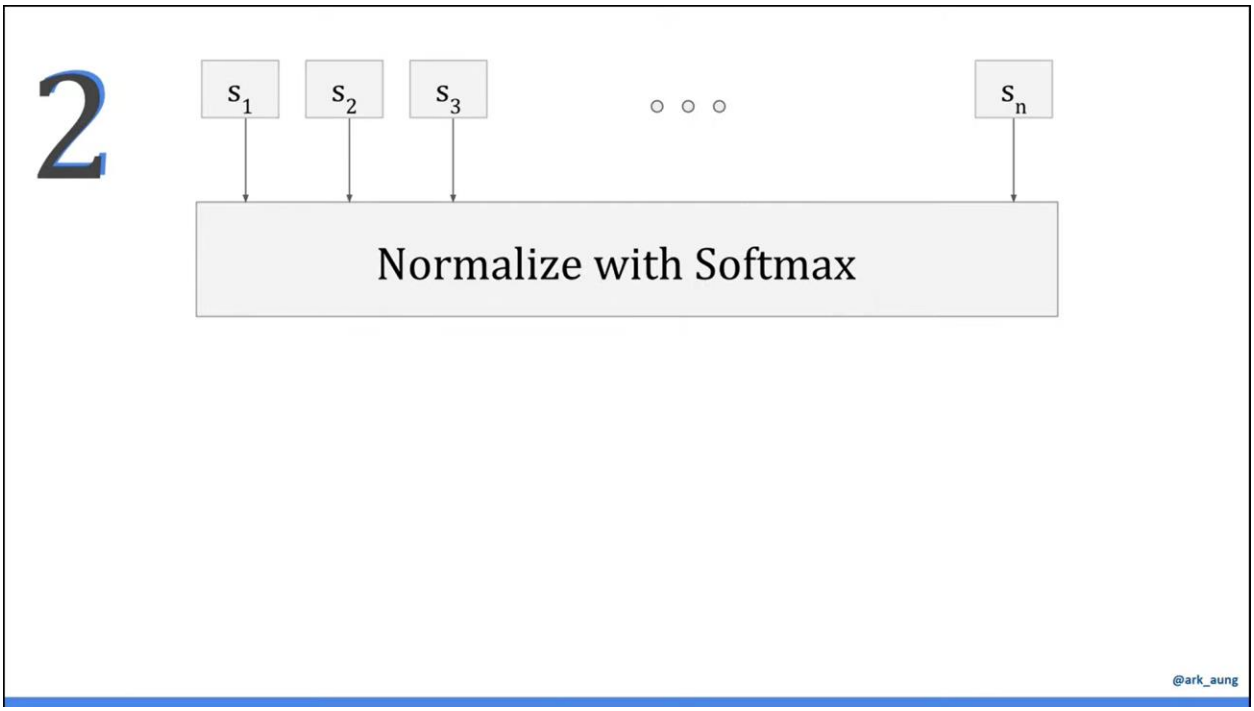


Graphically we can see the higher values are at diagonal




1
v_1 v_2 v_3 ○ ○ ○ v_n
Get Scores by Dot Product
s_1 s_2 s_3 ○ ○ ○ s_n
s_{11} s_{12} s_{13} …. s_{1n}
s_{n1} s_{n2} s_{n3} …. s_{nn}
@ark_aung


1
v_1 v_2 v_3
Get Scores by
s_1 s_2 s_3
s_{11} s_{12} s_{13} …. s_{1n}
s_{n1} s_{n2} s_{n3} …. s_{nn}
@ark_aung



Normalized because range can be very large between the dot product values, so we need to normalized it using softmax

The softmax normalization ensures that:

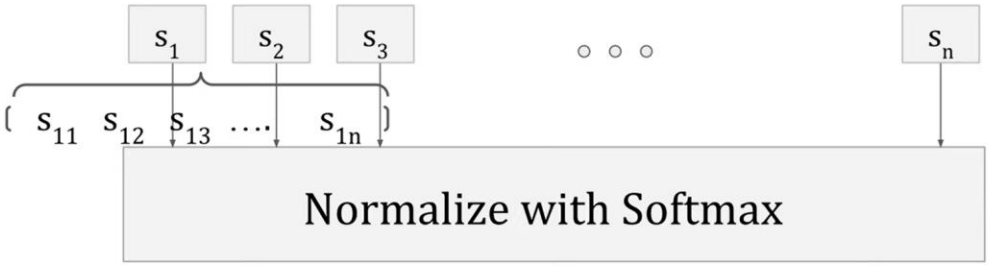
- **Scores are interpretable as probabilities.**

1. Convert Scores into Probabilities:

The raw similarity scores (dot products between query and key vectors) can be arbitrary real numbers. These raw values don't directly represent how much focus should be placed on each token. The softmax function converts these scores into a probability distribution, where all attention scores sum to 1.

- **This makes it easy to interpret the results: higher probability means more attention is given to that token.**

For example, if we have raw similarity scores like [3,1,−2], applying softmax converts them into probabilities like [0.84,0.13,0.03], where the first token gets the most attention.



$w_{11} = \text{softmax}(s_{11})$
$w_{12} = \text{softmax}(s_{12})$
$w_{13} = \text{softmax}(s_{13})$

$$\sigma(s_i) \triangleq \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}$$

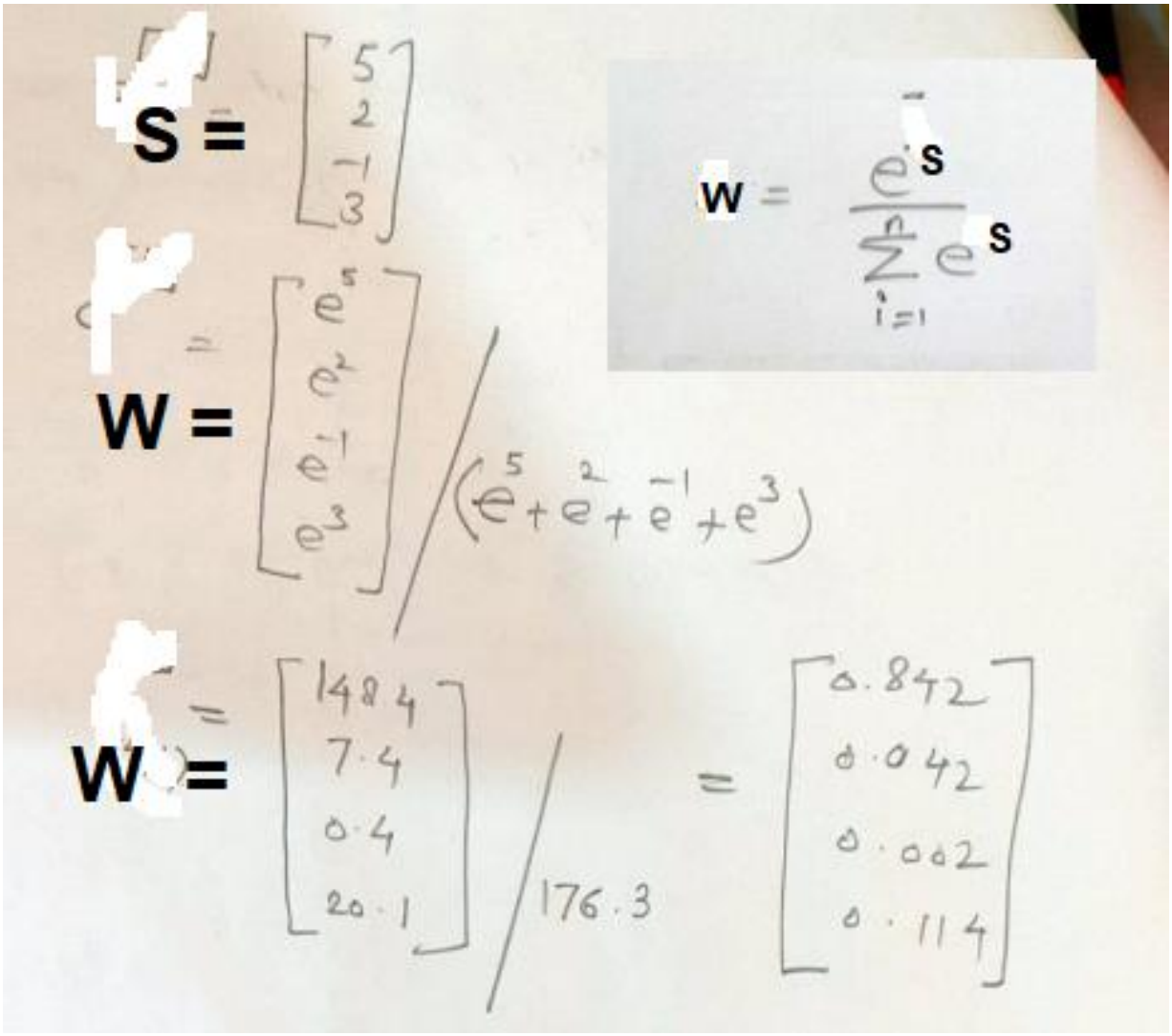
$w_{n1} = \text{softmax}(s_{11})$
$w_{n2} = \text{softmax}(s_{n2})$
$w_{n3} = \text{softmax}(s_{n3})$

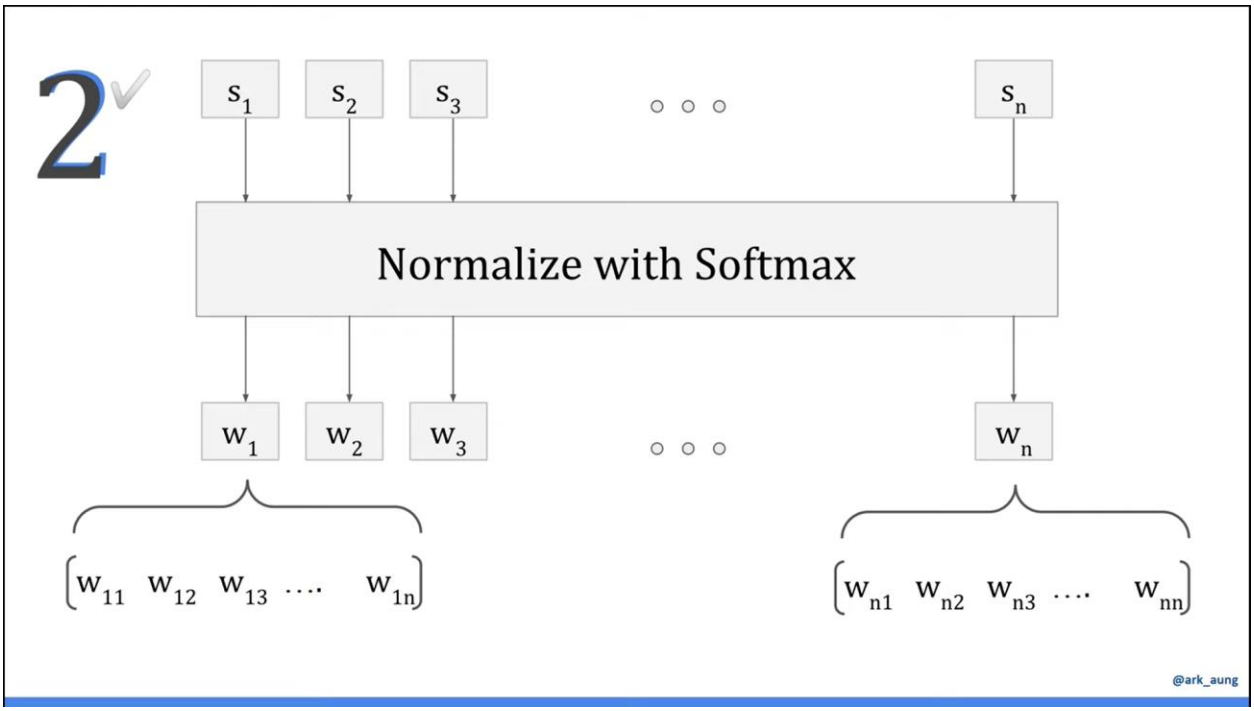
$w_{1n} = \text{softmax}(s_{1n})$

$$\sum_{i=1}^{n} w_{1i} = 1$$

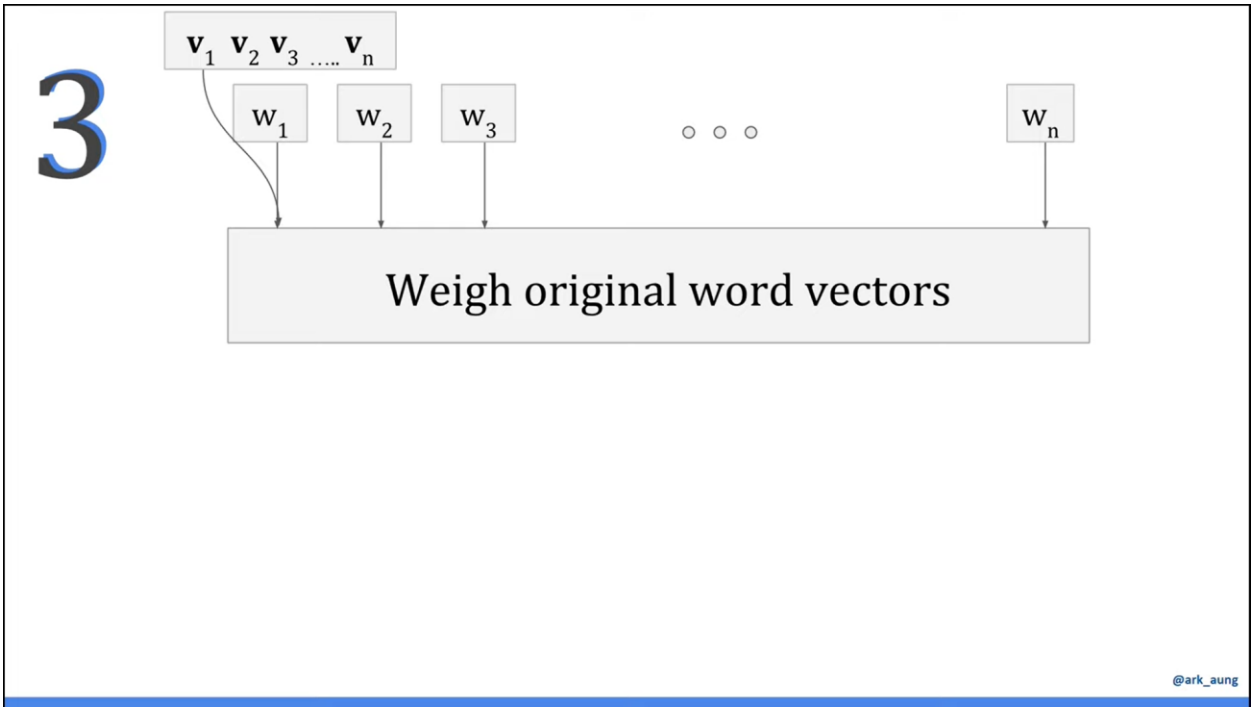
$w_{nn} = \text{softmax}(s_{nn})$

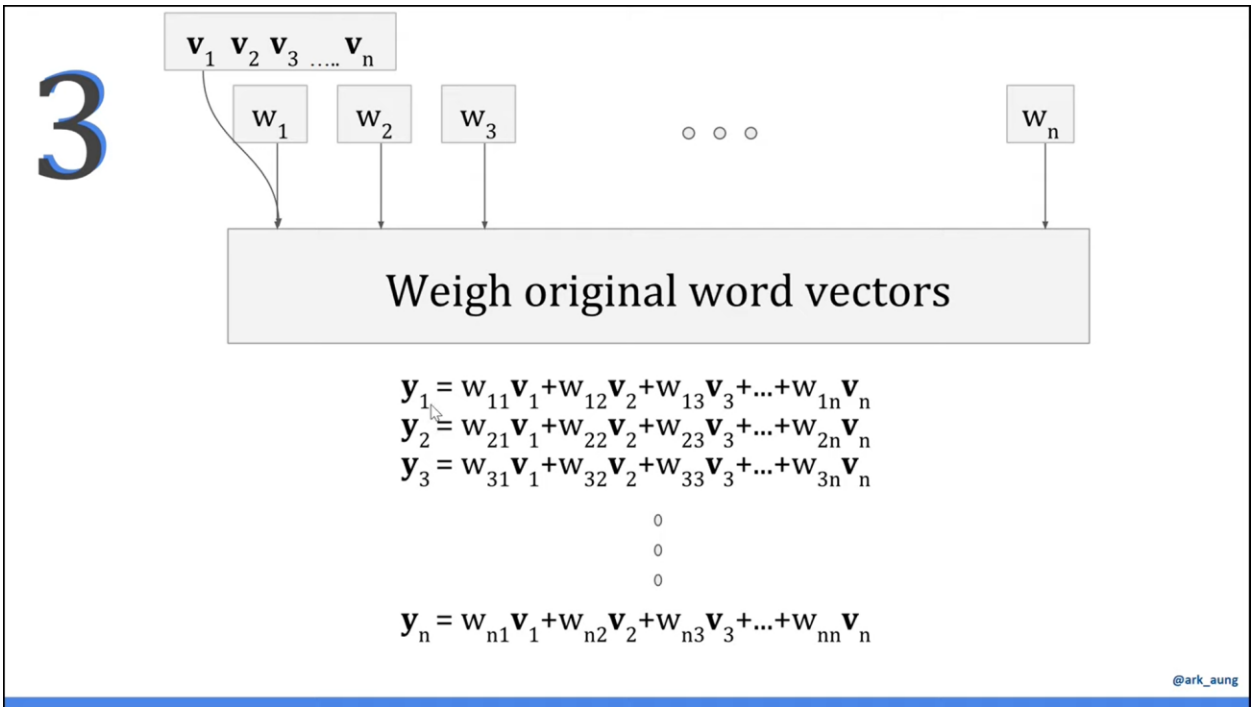
@ark_aung


S = [5, 2, -1, 3]
W = e^S / Σ_{i=1}^{n} e^S
W = [e^5, e^2, e^-1, e^3] / (e^5 + e^2 + e^-1 + e^3)
W = [148.4, 7.4, 0.4, 20.1] / 176.3 = [0.842, 0.042, 0.002, 0.114]



We get weights from the dot produce scroes





W's are weights and v are the vectors. Y1 is the contextual representation of vecto v1.

Y1 is got by gving w11 weight to itself (v1)

w12 v2 is the influence of v2 on the y1

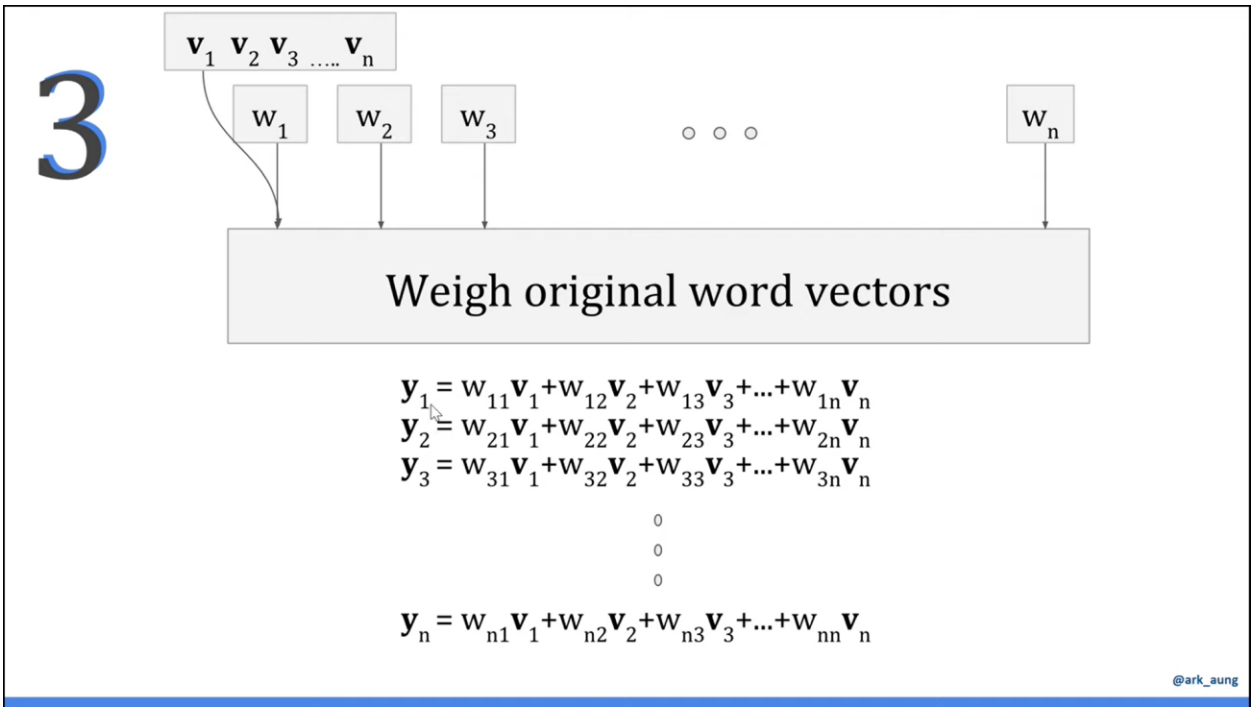
w13 v3 is the influence of v3 on the y1

....

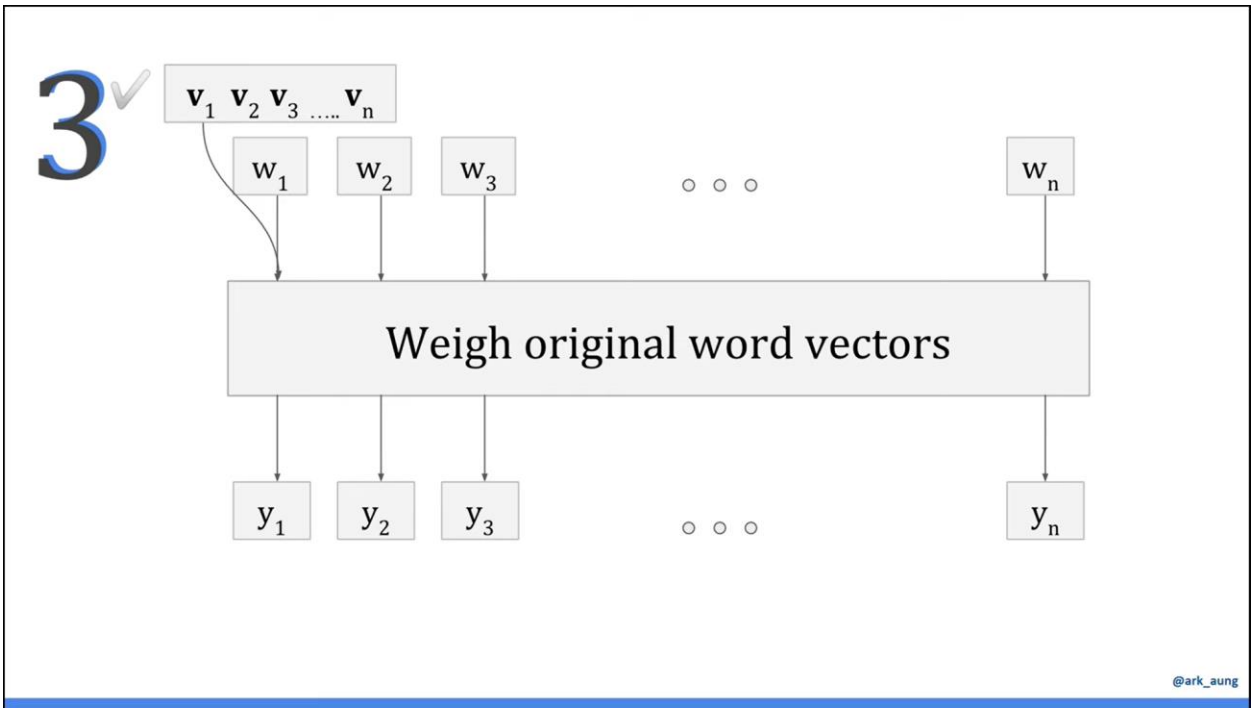
Now y1 is the contextual resprsentation of v1. That is it is no longer pointing towards the same direction as before. It is pointing towards the different direction, which is influenced by its neighbours

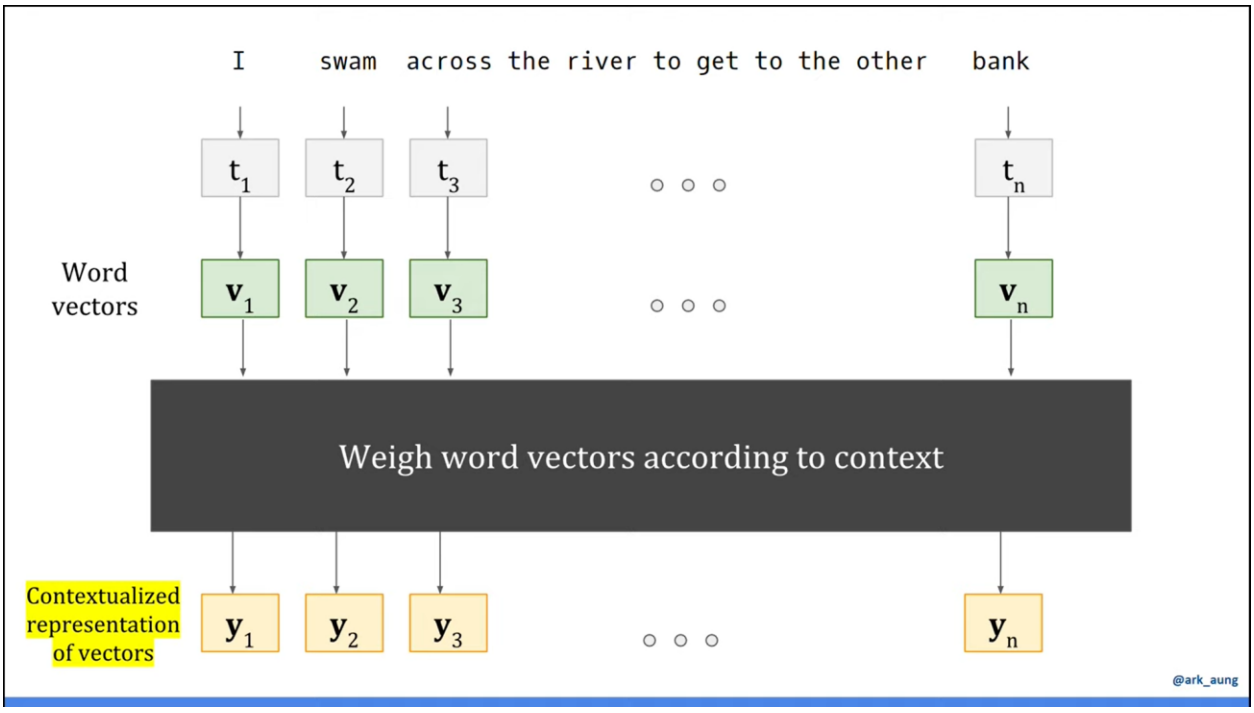
V1 is the orignal vector, while y1 is the enhanced version v1/ augumented version of v1 by weighing its sorrounding to iself by following process:

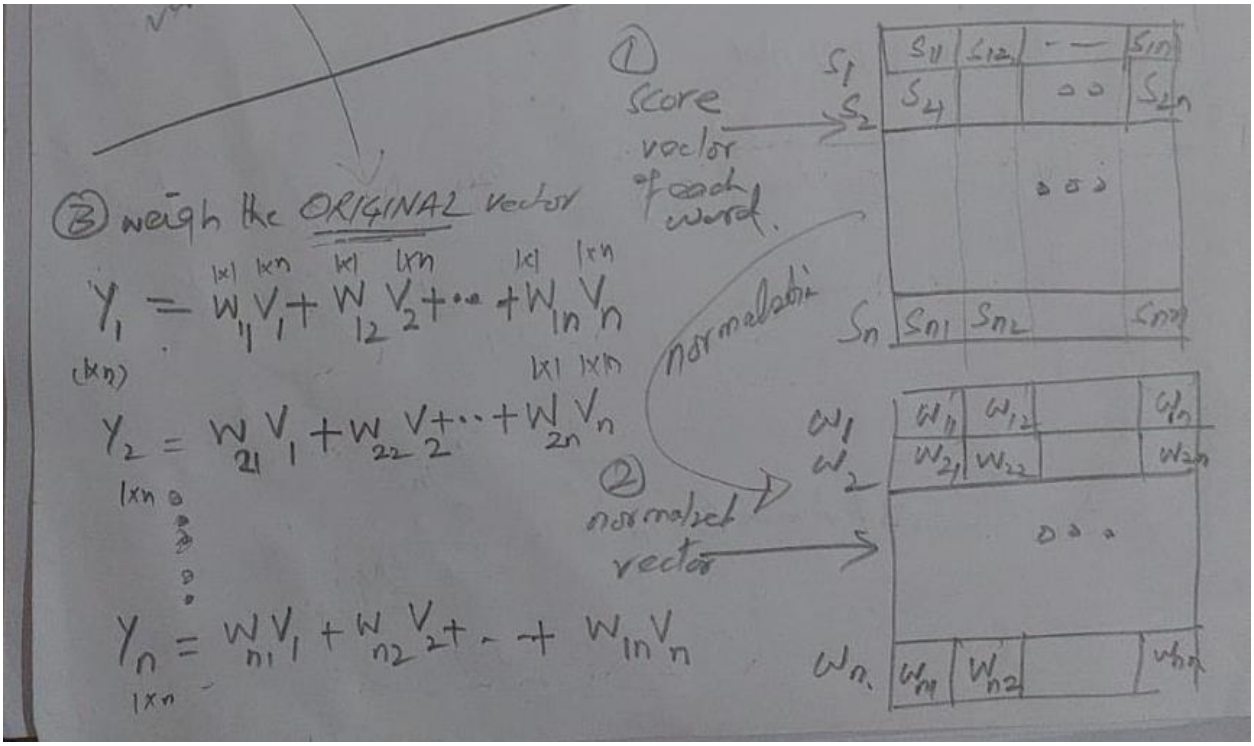
1. **Get score by produce**
2. **Normalze**
3. **Weigh the original vectors.**



Contextual Representation: The result of the weighted sum of the value vectors produces a new representation for the query word that is informed by the context of other words in the sequence. This representation captures not just the identity of the query word but also relevant information from other words, effectively enriching its context.

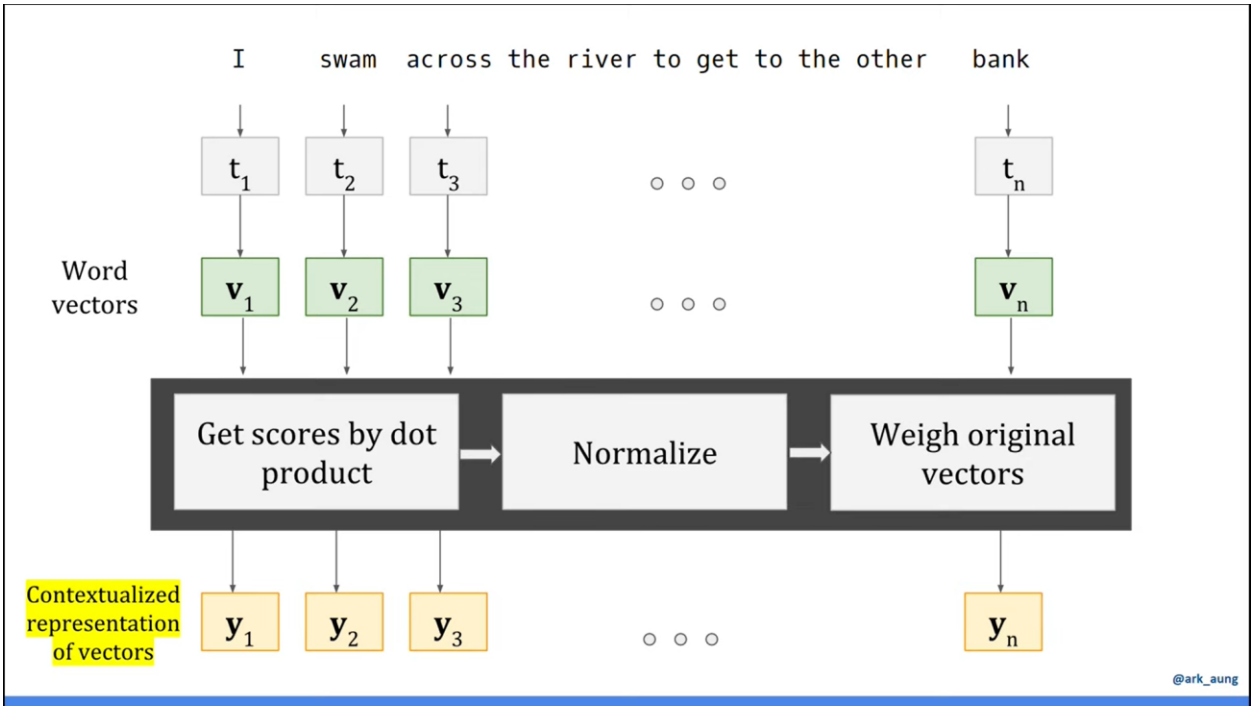



I swam across the river to get to the other bank
t1 t2 t3 ... tn
Word vectors
v1 v2 v3 ... vn
Weigh word vectors according to context
Contextualized representation of vectors
y1 y2 y3 ... yn
@ark_aung


① score vector of each word.
② normalized vector
③ weigh the ORIGINAL vector
normalize
Y1 = W11 V1 + W12 V2 + ... + W1n Vn
Y2 = W21 V1 + W22 V2 + ... + W2n Vn
Yn = Wn1 V1 + Wn2 V2 + ... + Wnn Vn

Lecture#2

1. **recap**



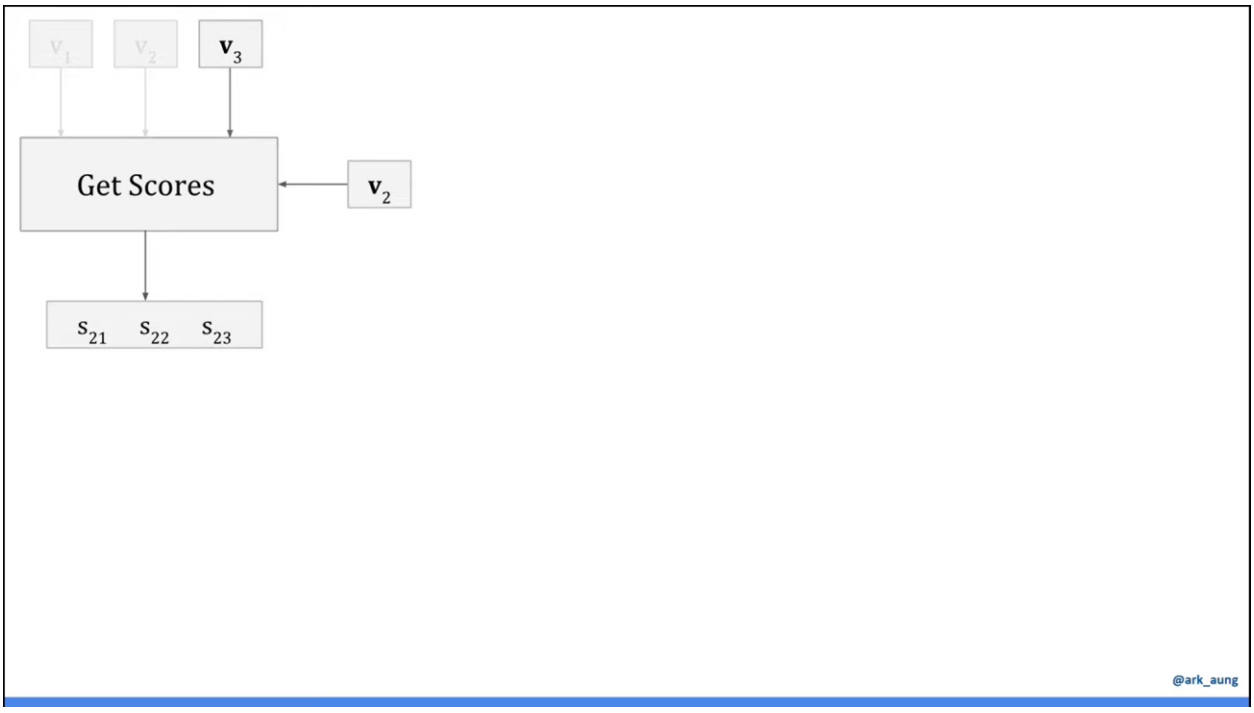
2. **Database terms:**

- **Query/question= which word is close to v2 or am I close to v1 or am I close to v2 etc**
- **Key: v1, v2, v3 are database value that needs to be compare**
- **Values: v1, v2, v3 are the values that used to get the contextulazed reprsention to v2**

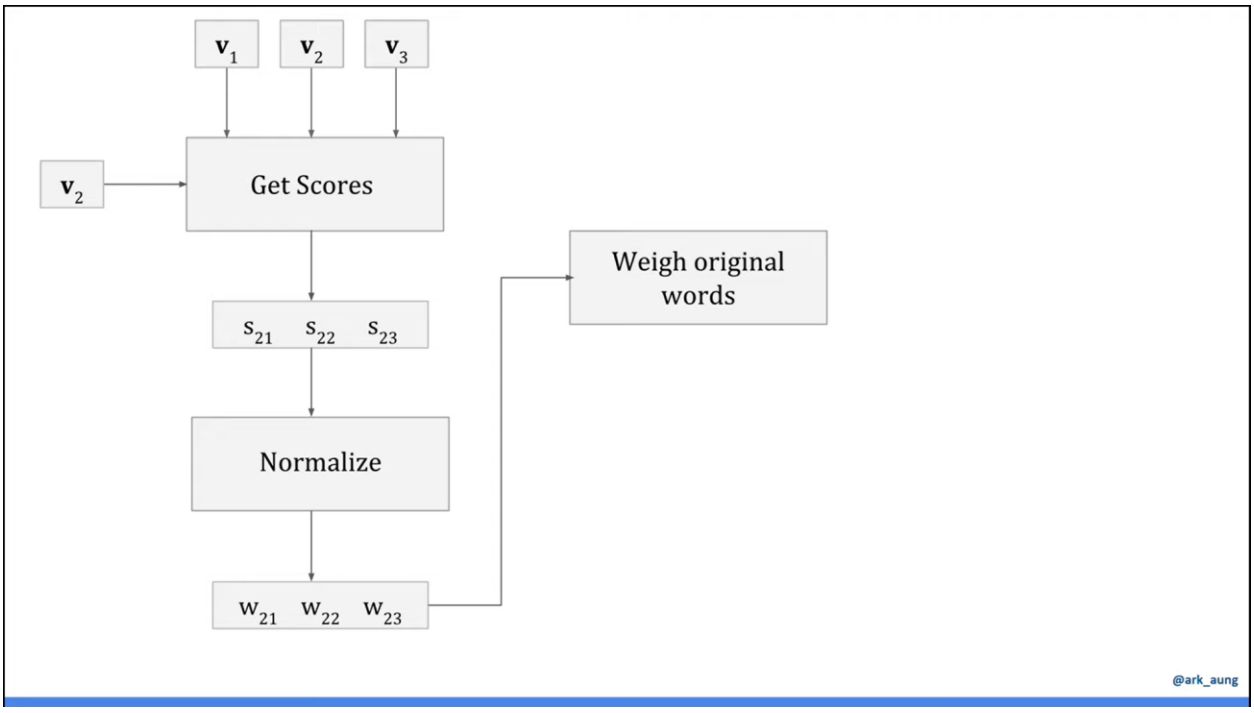
3. **Exmple: Lets see more concete exmaple of three words and want to find the contextulzed the repsepentation of v2.**

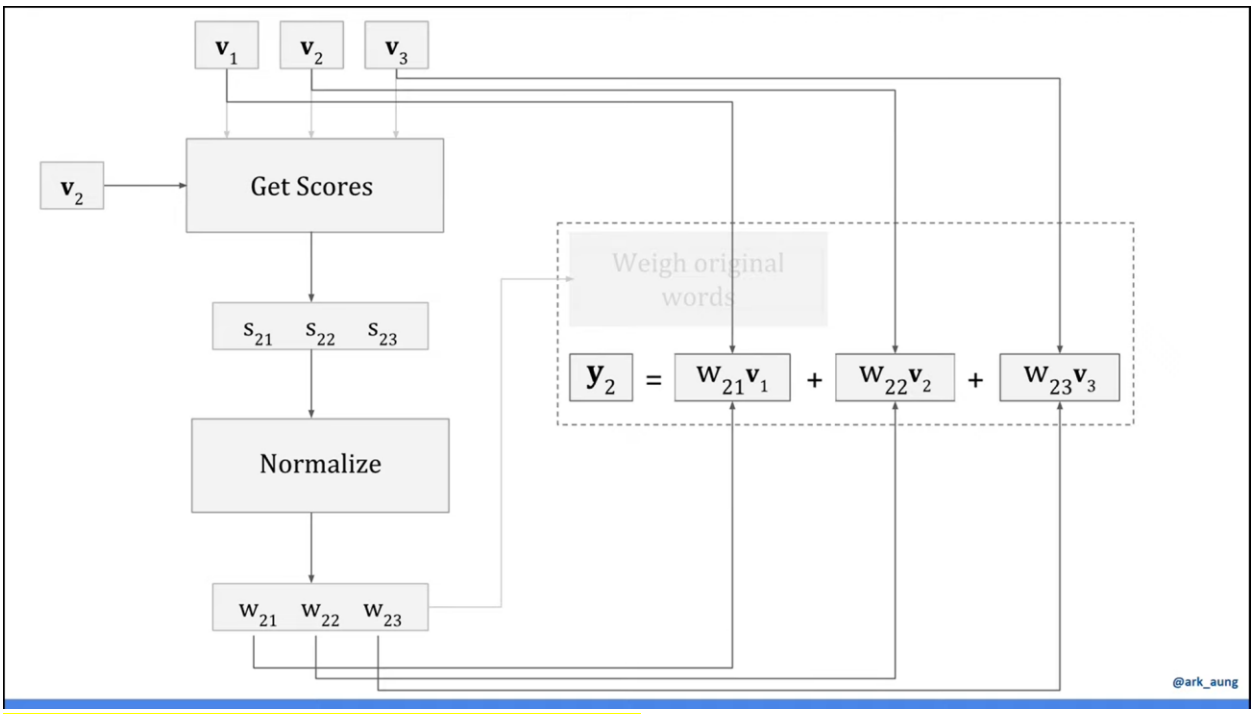

$\mathbf{v}_1$
$\mathbf{v}_2$
$\mathbf{v}_3$
Get Scores
$\mathbf{v}_2$
s_{21}
@ark_aung


$\mathbf{v}_1$
$\mathbf{v}_2$
$\mathbf{v}_3$
Get Scores
$\mathbf{v}_2$
s_{21} s_{22}
@ark_aung


v_1
v_2
v_3
Get Scores
v_2
s_{21} s_{22} s_{23}
@ark_aung

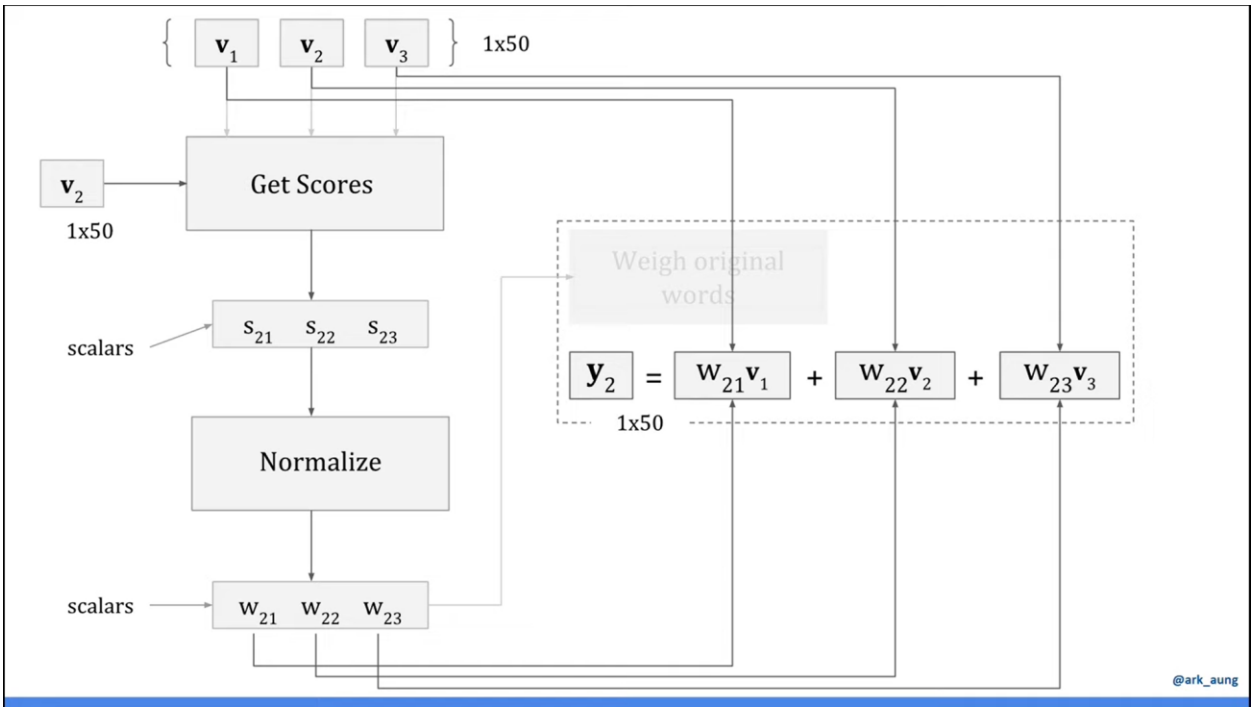

v_1
v_2
v_3
Get Scores
v_2
s_{21} s_{22} s_{23}
Normalize
w_{21} w_{22} w_{23}
@ark_aung

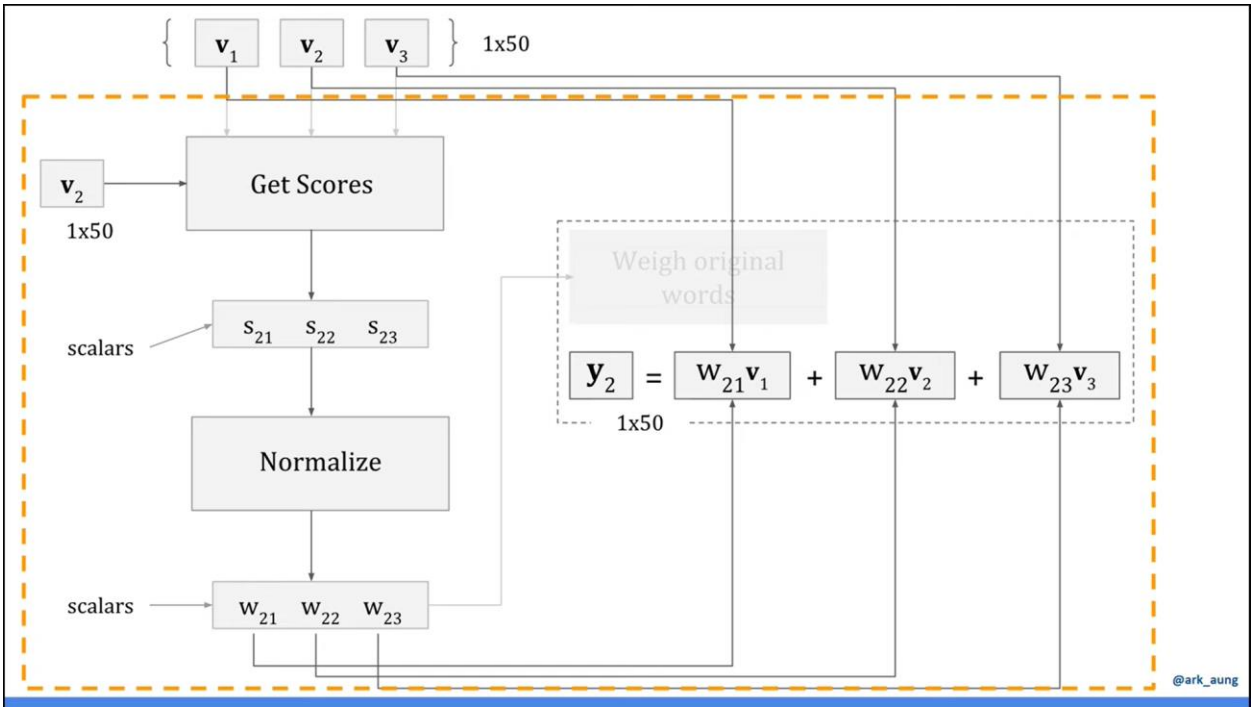




y2 is contextulized repsrentation of orignal vector v2

4. **dimension**





5. **Orang box is he self attensionor attention mechianism**

Which word is closer to v2, With respect to datasebase, it is query/question , and keys are other words that need to check with v2 to check either they are similar to v2 or not, values…… we need to inject the weights to preserve the diemension

6. **Where are the learning????**

for exmple of translation where the model output the contectualized vectors and now the decoder generate the french sentense using the contextualized information but if the prediction is not good then loss will be higher…. So there is need to update that could also be corresponding to query, key, and value vectors….. so that we able to produce the good results.

Self attention mechanism learn to focus on important parts of a sentence though training.

Why Learning is Necessary

- **Without learning, the attention mechanism would not adapt to different contexts, resulting in poor performance.**
- **With learning, the model dynamically adjusts the attention weights based on the meaning and context of the sentence, ensuring the model attends to the most relevant parts for the task at hand (e.g., translating "bank" correctly as a financial institution).**

Learning allows the model to capture context-specific relationships, ensuring that it performs well across various tasks and contexts.

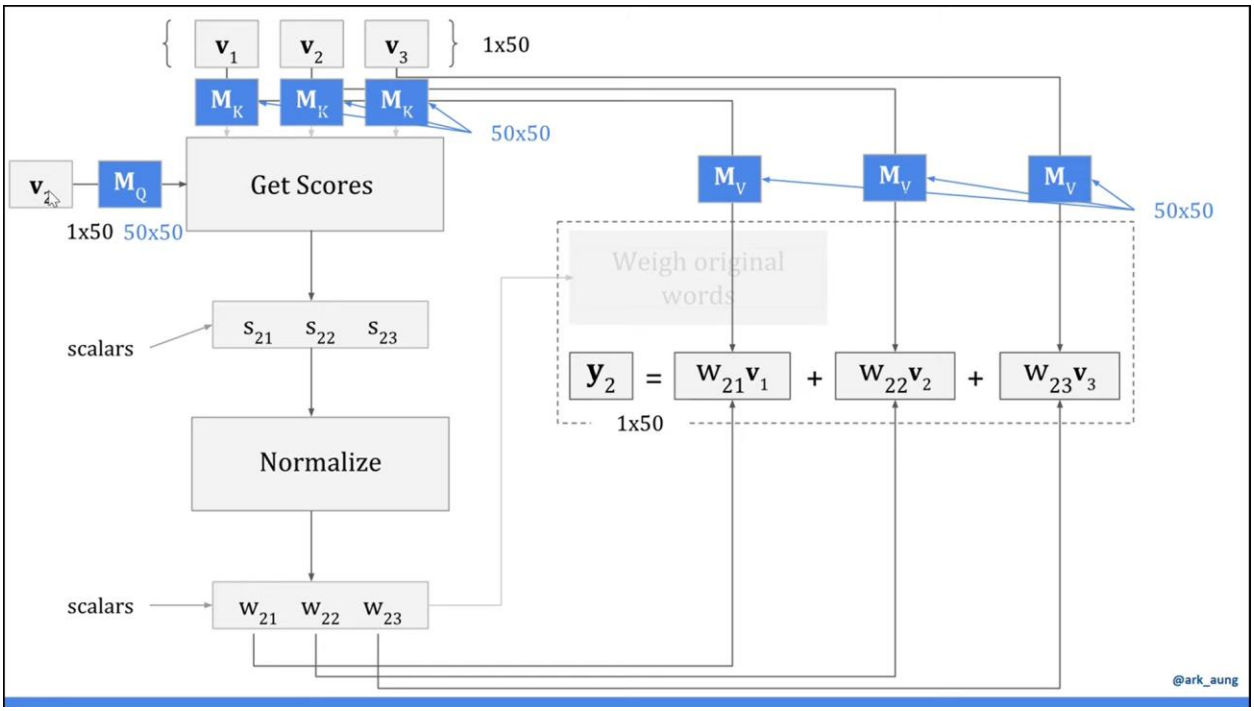
For example, in the sentence "I deposited money at the bank", after training, the model learns to:

- **Pay attention to "money" when processing "bank" to understand that this is a financial institution.**
- **Ignore irrelevant words like "at" and "the" for understanding the meaning of "bank."**

Step 5: Training and Backpropagation

The learning process occurs through training. When the model translates sentences, it compares its output to the correct translation (or output). A loss function is used to measure the difference between the model's prediction and the actual correct answer.

- **The model adjusts the query, key, and value weights through backpropagation, updating the weights to improve the attention scores and overall performance on the task.**
- **Over time, the model learns how much attention to pay to various words based on context. It learns to focus on the right words to improve accuracy.**



These weights are learn during BP, while the diemension of outputs are same …. These weights are the dense layers. We have 3 linear layers one for query, one is for value and one for key

V1 is 1x50 dimensional vector if there is single linear layer having 50 neurans than no of paramenters are 50x50. And matrix size is 50x50

Can you draw dense having 10 neurons and show forward and backward pass.

7. **Toy example**

Example Contextualized representation of a vector:

Let's go through a simple, **toy example** to demonstrate how the **self-attention mechanism** works in the context of a Transformer model.

Example Sentence:

We'll use the short sentence: **"The cat eats fish."**

Objective:

Let's understand how each word in this sentence attends to other words using **self-attention**.

Step 1: Assign Embeddings

For simplicity, let's assume we have the following **embedding vectors** for each word (these are made-up numbers to keep the math simple):

- **"The" → [1, 0]**
- **"cat" → [0, 1]**
- **"eats" → [1, 1]**
- **"fish" → [0, 0]**

Step 2: Create Query, Key, and Value Matrices

Let's also assume that we have the following **learnable weight matrices** for **query (W_Q)**, **key (W_K)**, and **value (W_V)**:

$$W_Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W_K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad W_V = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Step 3: Calculate Query, Key, and Value Vectors for Each Word

We calculate the query, key, and value vectors for each word using the weight matrices:

For the word "The":

- Query: $Q_{\text{The}} = W_Q \cdot \text{Embedding}_{\text{The}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- Key: $K_{\text{The}} = W_K \cdot \text{Embedding}_{\text{The}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- Value: $V_{\text{The}} = W_V \cdot \text{Embedding}_{\text{The}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$

For the word "cat":

- Query: $Q_{\text{cat}} = W_Q \cdot \text{Embedding}_{\text{cat}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- Key: $K_{\text{cat}} = W_K \cdot \text{Embedding}_{\text{cat}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- Value: $V_{\text{cat}} = W_V \cdot \text{Embedding}_{\text{cat}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$

For the word "eats":

- Query: $Q_{\text{eats}} = W_Q \cdot \text{Embedding}_{\text{eats}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Key: $K_{\text{eats}} = W_K \cdot \text{Embedding}_{\text{eats}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Value: $V_{\text{eats}} = W_V \cdot \text{Embedding}_{\text{eats}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \end{bmatrix}$

For the word "fish":

- Query: $Q_{\text{fish}} = W_Q \cdot \text{Embedding}_{\text{fish}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- Key: $K_{\text{fish}} = W_K \cdot \text{Embedding}_{\text{fish}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- Value: $V_{\text{fish}} = W_V \cdot \text{Embedding}_{\text{fish}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Step 4: Calculate Attention Scores (Dot Products between Query and Key)

We calculate the **attention score** for each pair of words by taking the dot product between the **query** of one word and the **key** of another.

Let's focus on the word "eats" and compute how much it attends to other words:

1. **Attention score between "eats" and "The":**

$$\text{Attention score} = Q_{\text{eats}} \cdot K_{\text{The}} = [1, 1] \cdot [0, 1] = 0 + 1 = 1$$

2. **Attention score between "eats" and "cat":**

$$\text{Attention score} = Q_{\text{eats}} \cdot K_{\text{cat}} = [1, 1] \cdot [1, 0] = 1 + 0 = 1$$

3. **Attention score between "eats" and "eats":**

$$\text{Attention score} = Q_{\text{eats}} \cdot K_{\text{eats}} = [1, 1] \cdot [1, 1] = 1 + 1 = 2$$

4. **Attention score between "eats" and "fish":**

$$\text{Attention score} = Q_{\text{eats}} \cdot K_{\text{fish}} = [1, 1] \cdot [0, 0] = 0 + 0 = 0$$

Step 5: Apply Softmax to Get Attention Weights

The attention weights are calculated by applying the **softmax** function to the attention scores:

$$\text{Softmax}(1, 1, 2, 0)$$

1. **Exponentiate each score:**
   - $e^1 = 2.718$, $e^1 = 2.718$, $e^2 = 7.389$, $e^0 = 1$
2. **Sum the exponentiated scores:**

$$2.718 + 2.718 + 7.389 + 1 = 13.825$$

3. **Calculate the attention weights:**
   - For "The": $\frac{2.718}{13.825} \approx 0.196$
   - For "cat": $\frac{2.718}{13.825} \approx 0.196$

.....

- We computed the **attention weights** with respect to "eats" as follows:
  - Attention weight for "The": 0.196
  - Attention weight for "cat": 0.196
  - Attention weight for "eats": 0.534
  - Attention weight for "fish": 0.072

These weights represent **how much the word "eats" pays attention** to each of the other words, including itself.

Step 6: Calculate the Weighted Sum of Value Vectors

Recap of Value Vectors:

- Value vector for "The": $[1, 3]$
- Value vector for "cat": $[2, 4]$
- Value vector for "eats": $[3, 7]$
- Value vector for "fish": $[0, 0]$

Contextual Vector Calculation for "eats":

The **contextual vector** is the weighted sum of all the value vectors using the attention weights. This is calculated as follows:

$$\text{Contextual Vector} = (0.196 \cdot [1, 3]) + (0.196 \cdot [2, 4]) + (0.534 \cdot [3, 7]) + (0.072 \cdot [0, 0])$$

Step 6: Calculate the Weighted Sum of Value Vectors

Recap of Value Vectors:

- Value vector for "The": $[1, 3]$
- Value vector for "cat": $[2, 4]$
- Value vector for "eats": $[3, 7]$
- Value vector for "fish": $[0, 0]$

Contextual Vector Calculation for "eats":

The **contextual vector** is the weighted sum of all the value vectors using the attention weights. This is calculated as follows:

$$\text{Contextual Vector} = (0.196 \cdot [1, 3]) + (0.196 \cdot [2, 4]) + (0.534 \cdot [3, 7]) + (0.072 \cdot [0, 0])$$

Step-by-Step Calculation:

1. **Multiply each value vector by its corresponding attention weight:**
   - For "The": $0.196 \times [1, 3] = [0.196, 0.588]$
   - For "cat": $0.196 \times [2, 4] = [0.392, 0.784]$
   - For "eats": $0.534 \times [3, 7] = [1.602, 3.738]$
   - For "fish": $0.072 \times [0, 0] = [0, 0]$
2. **Add these weighted value vectors together:**

$$\text{Contextual Vector} = [0.196, 0.588] + [0.392, 0.784] + [1.602, 3.738] + [0, 0]$$

3. **Perform the addition:**

$$\begin{aligned}\text{Contextual Vector} &= [0.196 + 0.392 + 1.602 + 0, \quad 0.588 + 0.784 + 3.738 + 0] \\ &= [2.19, \quad 5.11]\end{aligned}$$

What Does This Mean?

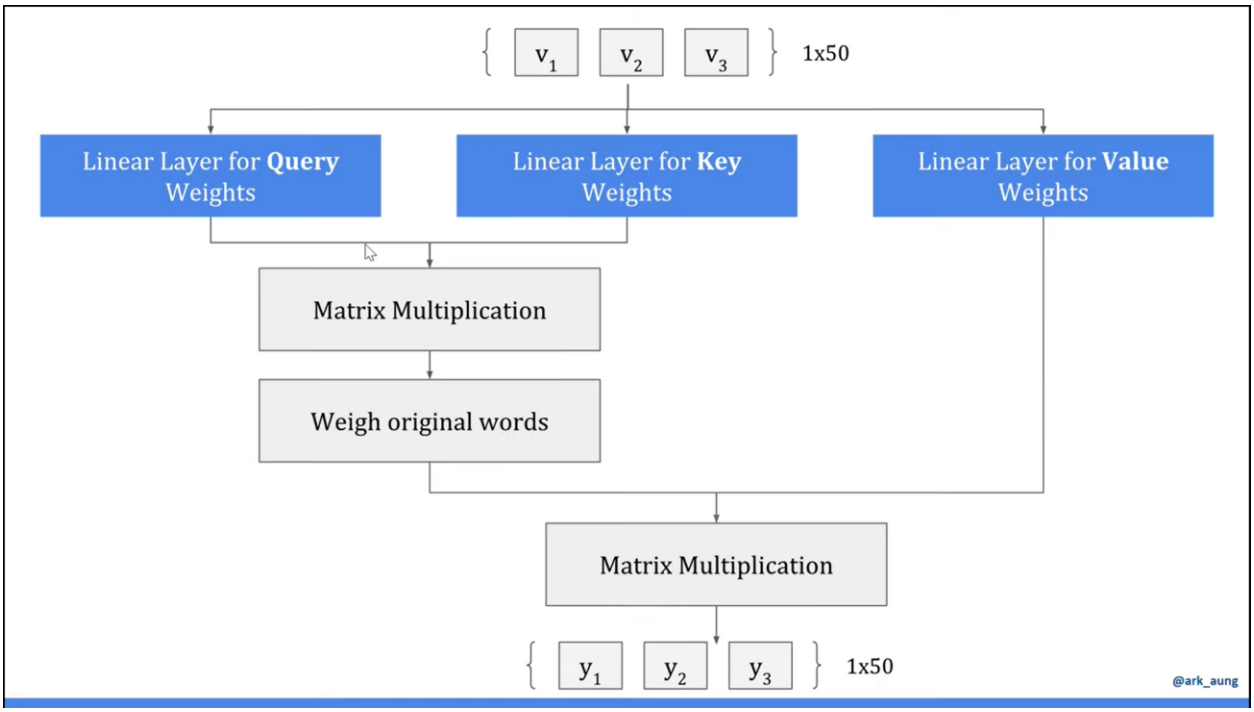
The **contextual vector** for the word **"eats"** now incorporates information from all the words in the sentence based on how much attention "eats" paid to each of them.

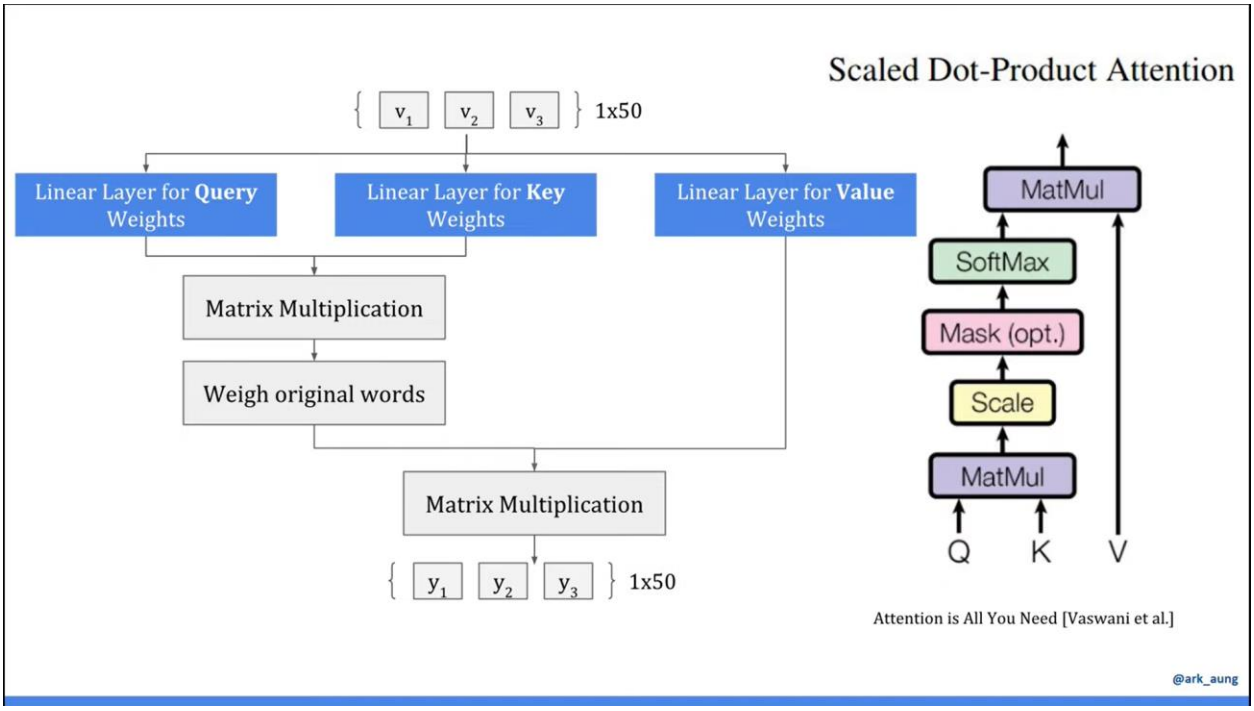
This new vector is a **context-aware representation** of the word "eats" that reflects its relationship with the words "The," "cat," and "fish."

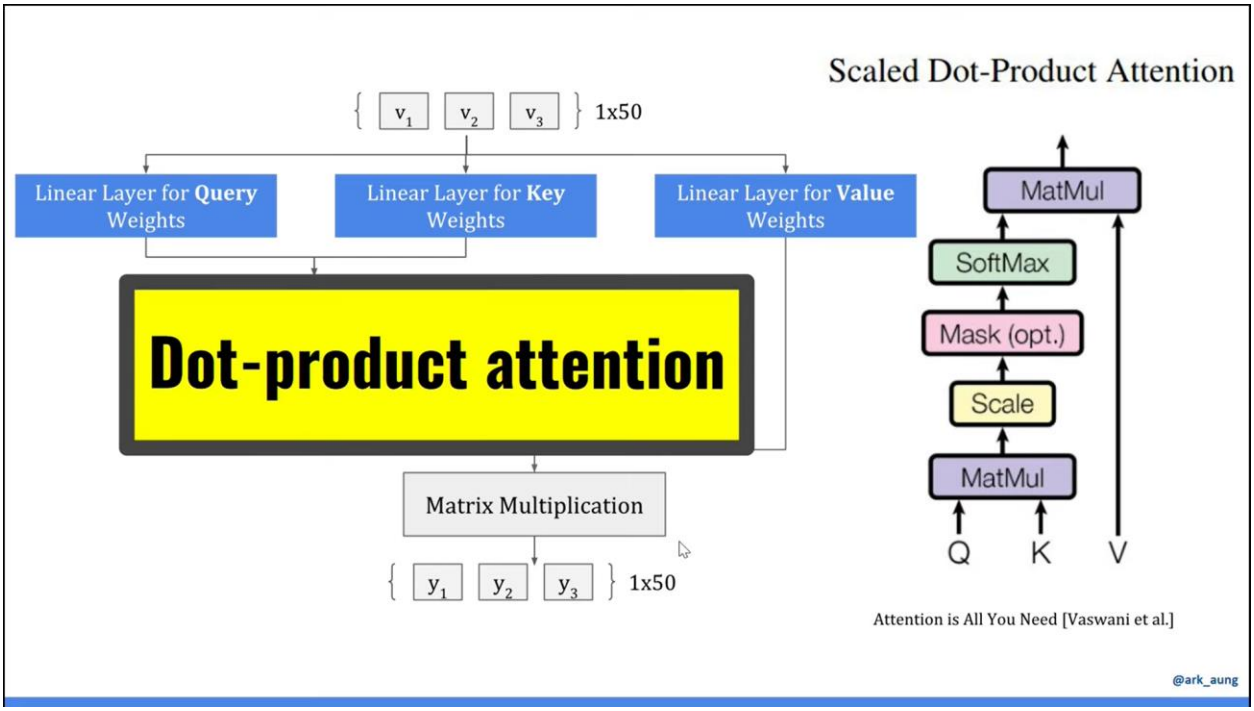
Recap of the Process:

1. **Calculate Query, Key, and Value vectors** for each word.
2. **Compute attention scores** (dot product of Query and Key).
3. **Apply softmax** to obtain attention weights.
4. **Use the attention weights to create a weighted sum** of the Value vectors to produce the final contextual vector.

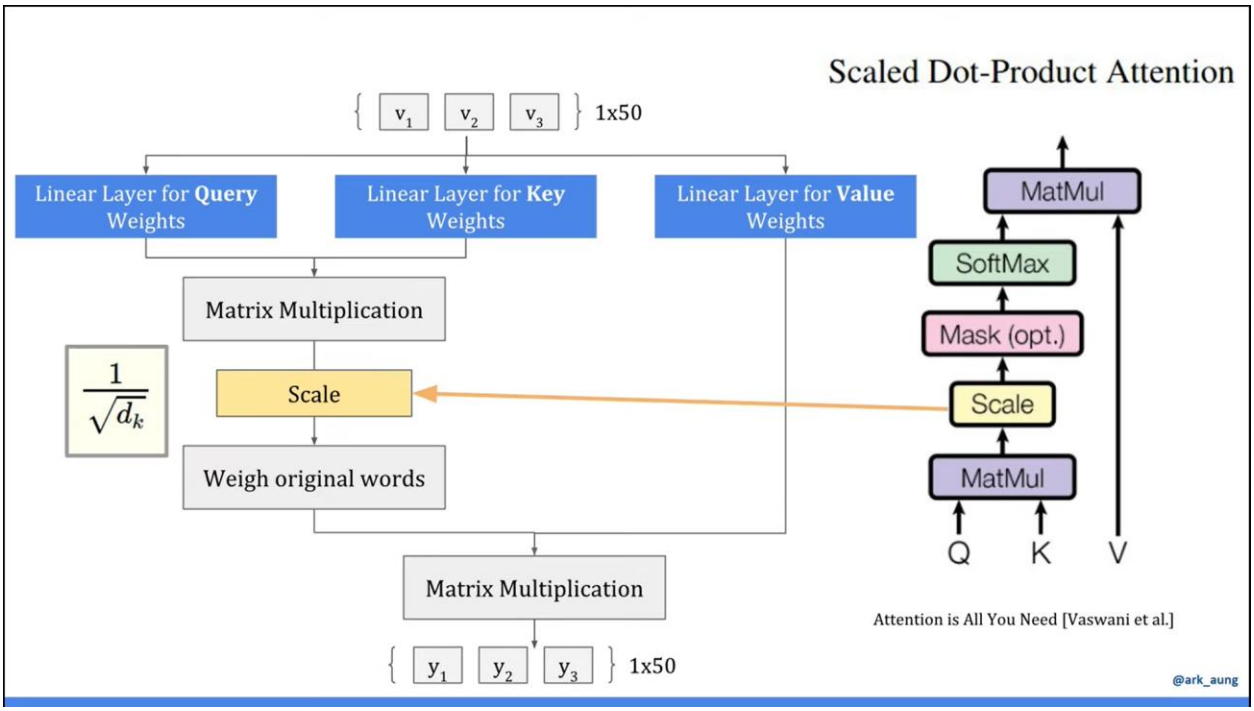
This toy example demonstrates how **self-attention** works in a simple context. By following this process for each word in the sentence, the model can create **context-aware representations** for all words, which improves its understanding of the relationships between them.







Here is the scaled version:



Intution behind the scaling:

Assume we have query vector v1= [1 1 1]

Key vectors: v1= [1 1 1], v2= [1 0 0], v3= [0 1 0], v4= [0 0 1]

S11 = v1 v1.T, S12 = v1 v2.T, S13 = v1 v3.T, S14 = v1 v4.T

S1 = [S11 S12 S13 S14] = [3 1 1 1]

Without scaling:

W1 = softmax(S1) = [0.711 0.096 0.096 0.096]

- **Without scaling, the softmax outputs are skewed more towards the largest value, giving 71.1% attention to the first token/vector.**

During gradient computation, we have:

dW1/dS1 = W1 (1-w1) = [W11 (1-w11) W12 (1-w12) W13 (1-w13) W14 (1-w14)]=

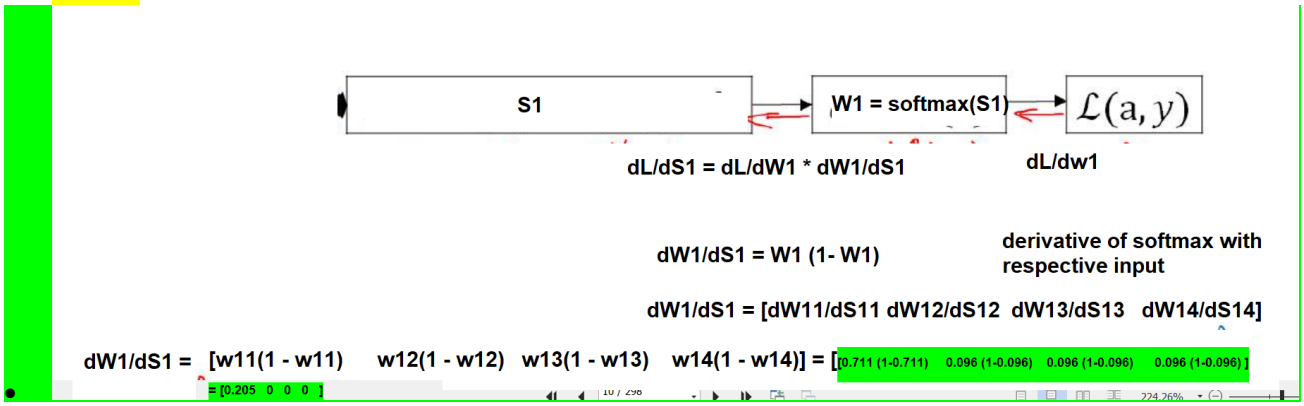
= [0.711 (1-0.711) 0.096 (1-0.096) 0.096 (1-0.096) 0.096 (1-0.096)]

= [0.711 (1-0.711) 0.096 (1-0.096) 0.096 (1-0.096) 0.096 (1-0.096)]

= [0.205 0.087 0.087 0.087 0.087]

= [0.205 0 0 0]

- In this case, the gradient for the last three token/vectors is negligible, and only first tokens contribute to the model's learning, leading to lesser meaningful updates, shown below:



With Scaling (scaling factor as dimension of vector is 4, so sqrt(4) = 2:

S1 = [S11 S12 S13 S14] = [3 1 1 1]

S1' = S1 / sqrt(4) = [3 1 1 1]/ 2 = [1.5 0.5 0.5 0.5]

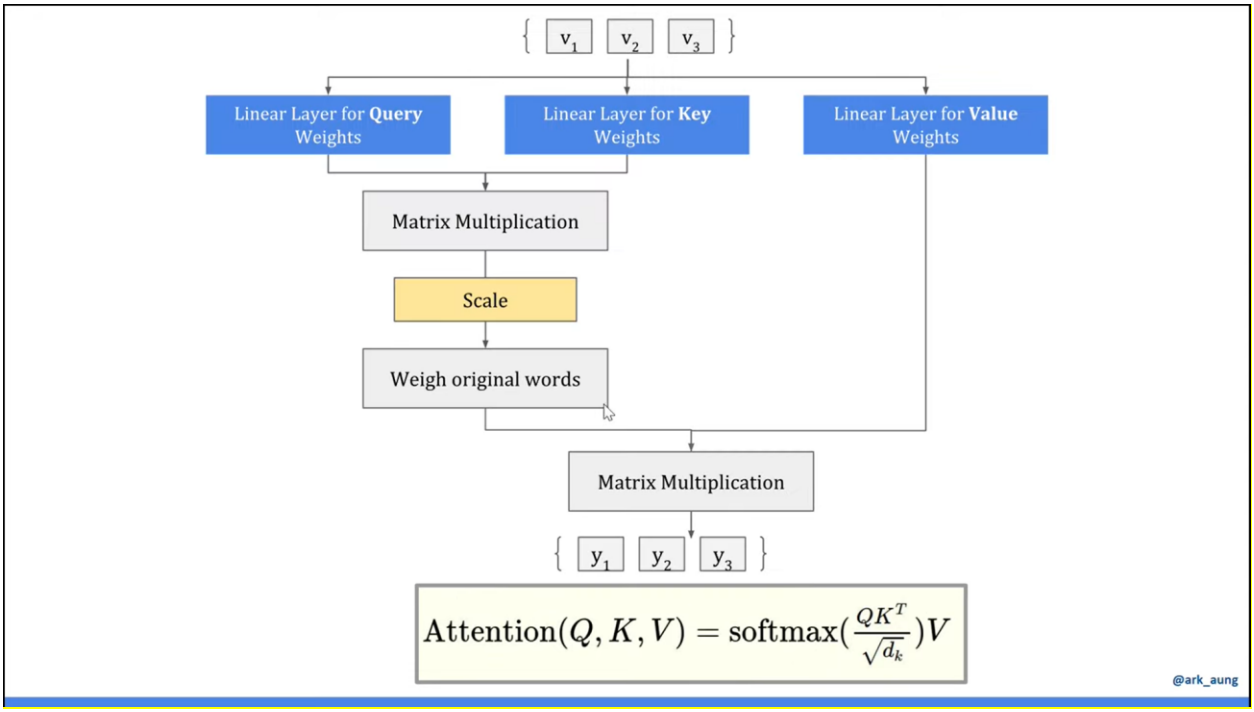
W1 = softmax(S1) = [0.47 0.17 0.17 0.17]

With scaling, the dot product might result in a more balanced softmax output like

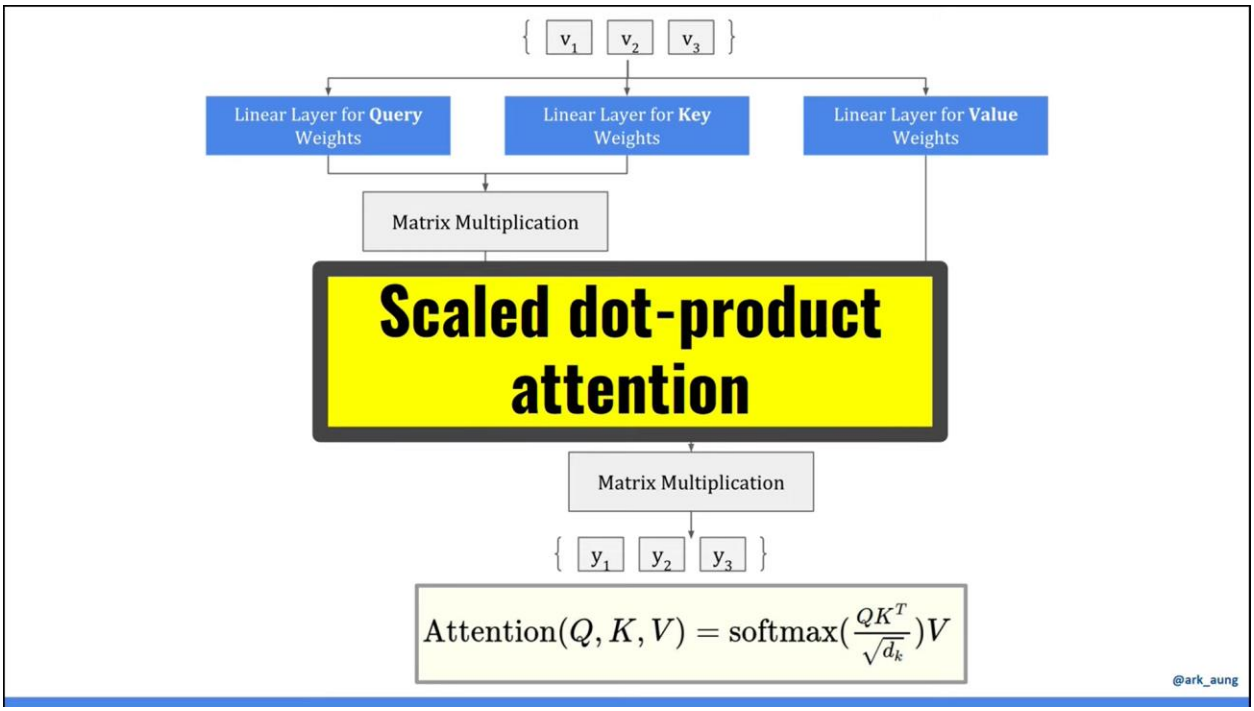
[0.47 0.17 0.17 0.17]. In this case, the gradient for the last three token is no longer negligible, and all tokens contribute to the model's learning, leading to more meaningful updates.

Effect of Scaling:

- **Without scaling, the softmax outputs are skewed more towards the largest value, giving 71.1% attention to the first key.**
- **With scaling, the attention is more balanced, distributing as 47%, 17%, and 17%, and 17%, which prevents overly sharp peaks in attention distribution**



This is similar to the diagram in paper


{ v1 v2 v3 }
Linear Layer for Query Weights
Linear Layer for Key Weights
Linear Layer for Value Weights
Matrix Multiplication
Scaled dot-product attention
Matrix Multiplication
{ y1 y2 y3 }
$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
@ark_aung

Lecture # 03

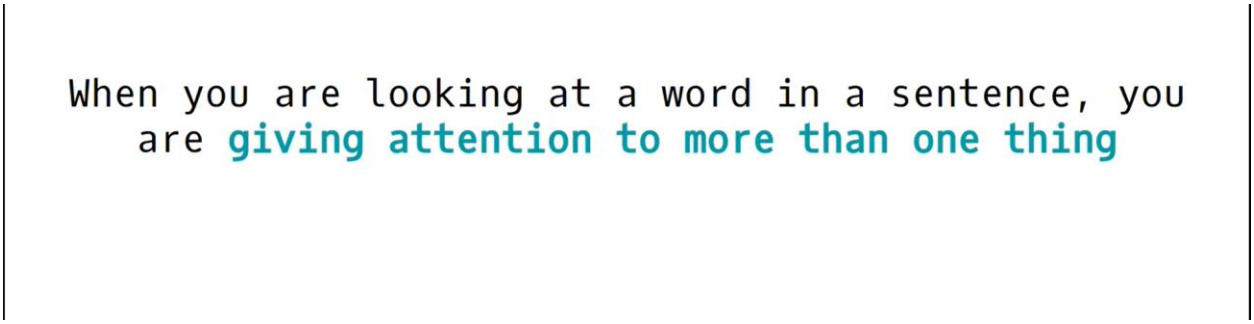
Two new idea in transformer compare to other NN: positional encoding + multi head attention

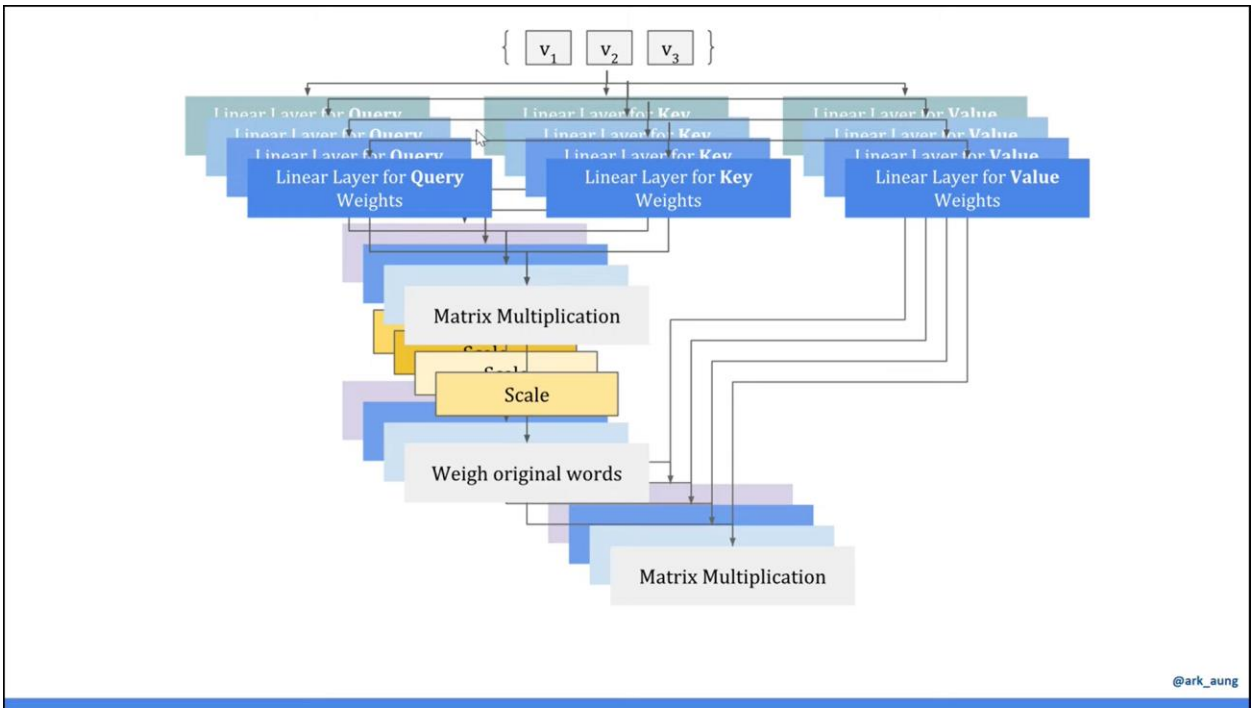
Question, can we use the same architecture of transformer

1. Translation
2. Next word generation.
3. How the chat gpt work…. Is is work like next word generation
4. Is the translation is just like the generation of a next work???

1. **Multihead attention**
2. **WordtoVec encoding**
3. **Positional encoding,**
4. **Encoder layers**
5. **Deconder layers**
6. **Cross attention:**
7. **Mask multihead layers etc.**

1. **Multihead attention**





1. **Analogous to Convolutional Neural Networks (CNNs)**

Multiple Feature Maps: The concept is similar to using multiple kernels in CNNs, where each kernel learns different features from the input. In transformers, multiple heads allow for learning various "feature maps" of word relationships, enhancing the model's ability to understand context

2. **Reasons for Using Multiple Attention Heads:** It expands the model's ability to focus on different positions

A. Diverse Relationships: Each attention head can focus on different aspects of the input, enabling the model to learn various relationships among words. This is particularly important in natural language processing, where words can relate in complex ways (e.g., negation, moderation)

B. Parallel Processing: By running multiple attention heads in parallel, the transformer can process information more efficiently and capture richer interpretations of the input data

E. Capture Different Aspects of Relationships

- Each attention head computes **self-attention** independently.

- By doing this, different heads can **focus on different parts of the input sequence** or capture different types of relationships between words (or tokens). This means that:

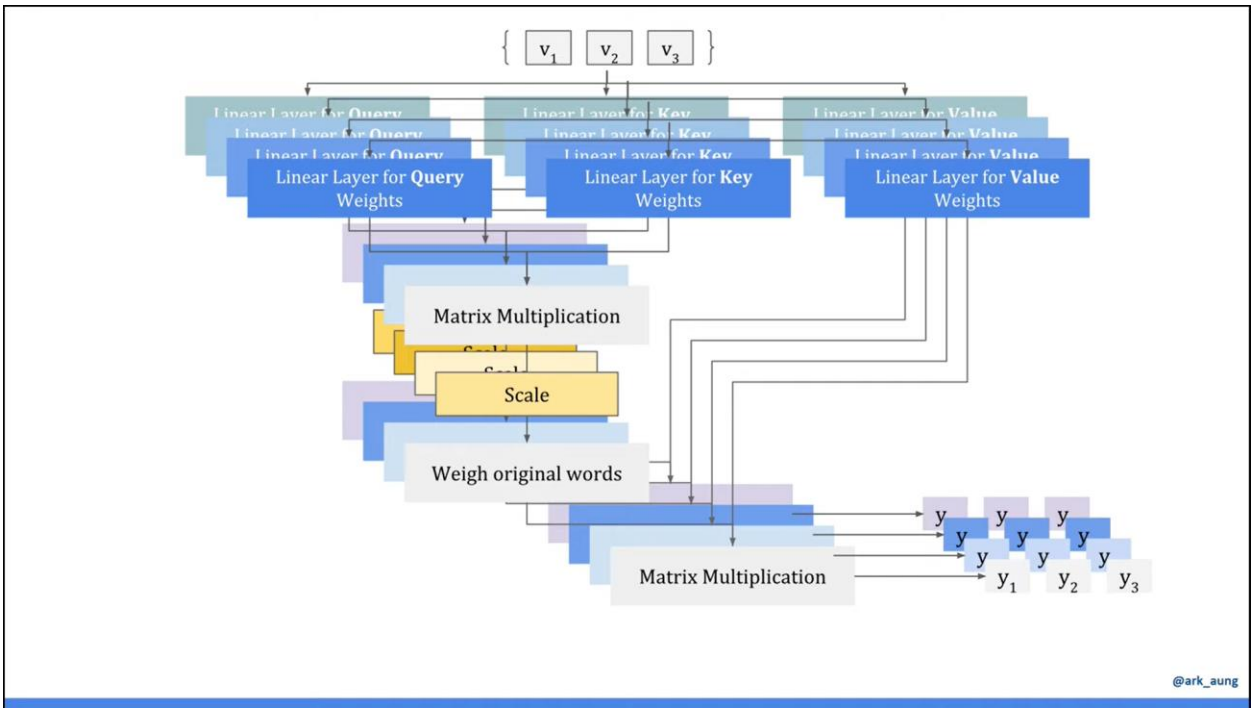
    - One head might focus on **short-range dependencies** (words close to each other).
    - Another head might capture **long-range dependencies** (words far apart).
    - A different head might focus on syntactic information, while another might focus on semantic information.

Summary:

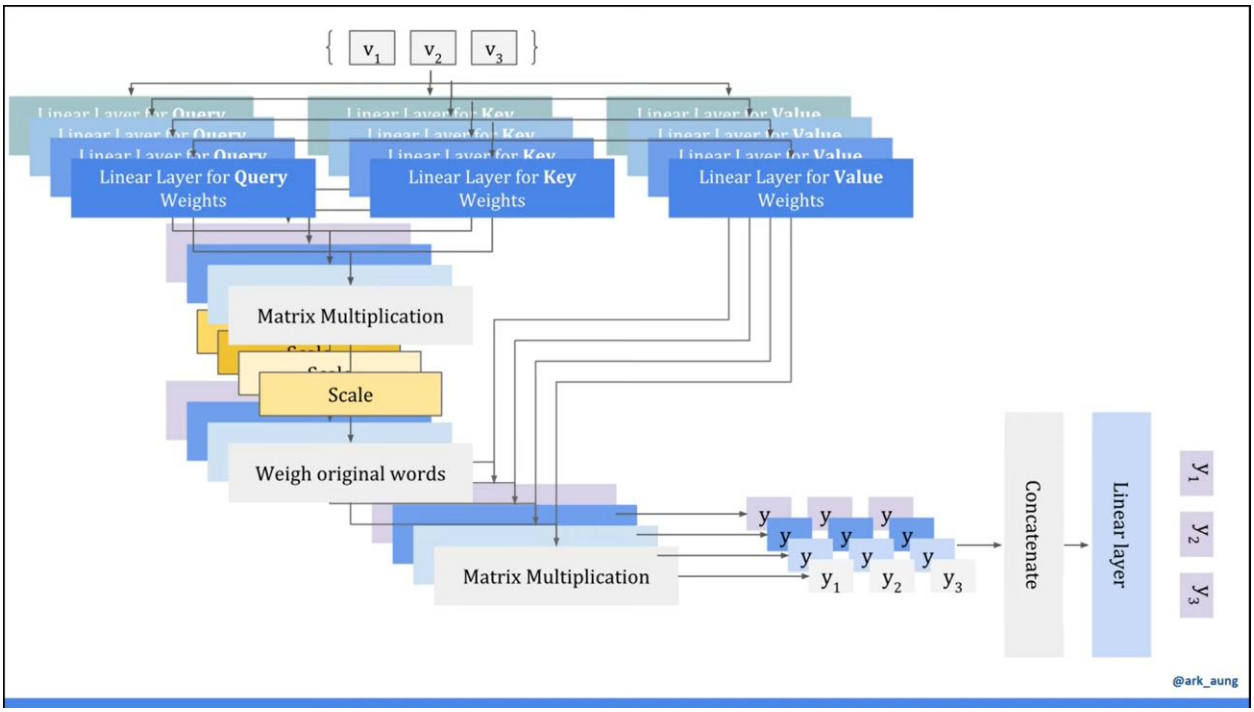
- The **training data** and **task-specific requirements** guide each head to learn different attention patterns based on the feedback from the loss function.
- Different heads in **multi-head attention** learn to focus on **short-range vs. long-range dependencies**, as well as **syntactic vs. semantic relationships**, because each head has **independent weight matrices** that are learned through training.
- Some heads specialize in **grammatical structure**, while others focus on **meaning**.
- This diversity of attention patterns makes the model more powerful and capable of capturing complex relationships in the input sequence.

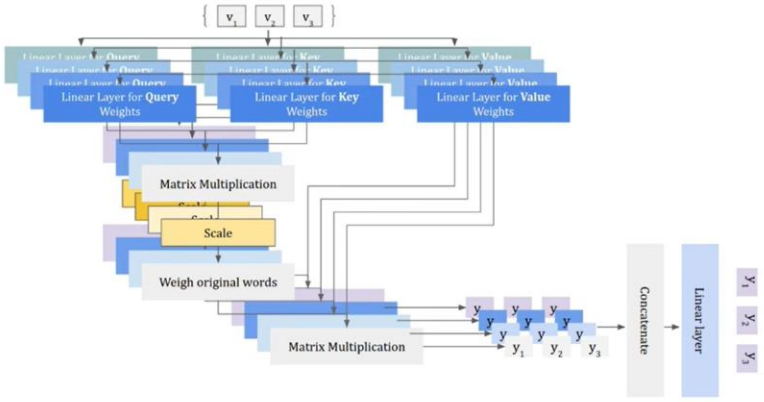
By allowing different attention heads to specialize in capturing different types of relationships, the Transformer model can better understand the complexities of natural language, improving performance on tasks like translation, summarization, and classification.

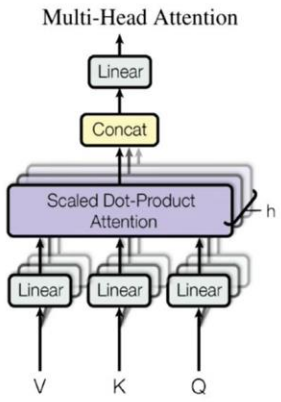
,



We concatenate ally contexutal vectors to genrate the single vectors







Attention is All You Need [Vaswani et al.]

@ark_aung

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

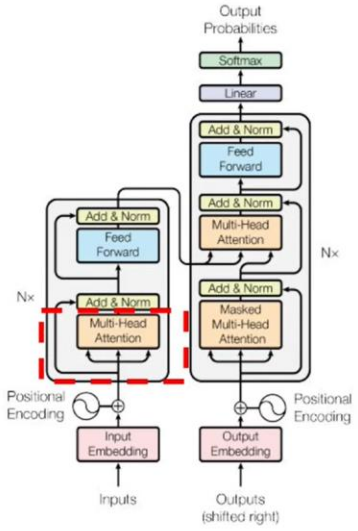


Figure 1: The Transformer - model architecture.

Attention is All You Need [Vaswani et al.]

@ark_aung

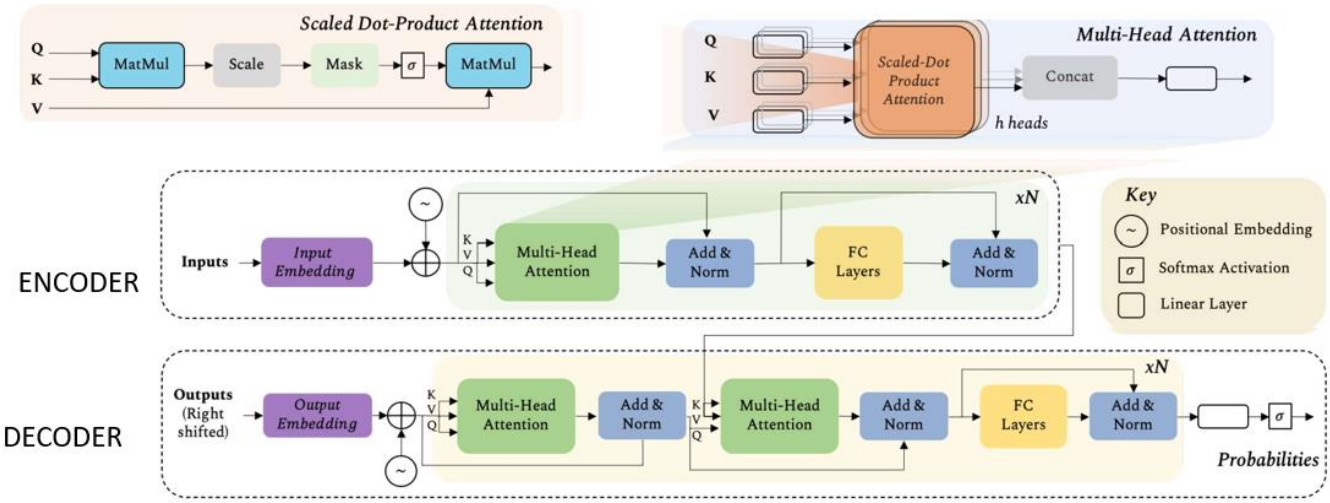
UCF Center for Research in Computer Vision

Transformer

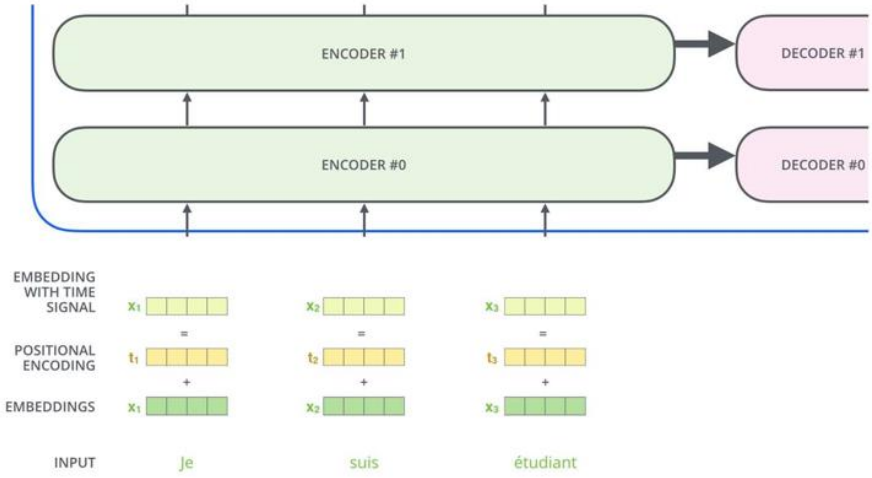
- It consists of Encoder and Decoder Blocks
- Main components of each block:
  - Self-Attention
  - Layer Normalization
  - Feed Forward Network

Tranformer use self attention mechanism to caputre the a weighted sum of all the tokens in a sequence at each layer.

Transformer



2. Word and postional embeddings



a. Word Embeddings

Word encoding in transformers refers to the process of converting textual data into numerical representations that can be processed by the model. This involves several key components:

- Definition: Word embeddings are dense vector representations of words that capture semantic meanings. In transformers, each word or token is mapped to a vector in a high-dimensional space, where similar meanings are represented by vectors that are close together.

b. **Positional Encoding**:

  - Since the Transformer model has no inherent sense of the order of words due to its lack of recurrence or convolution (RNN/LSTM), it requires a way to incorporate positional information into the input embeddings. Positional encodings are added to the input embeddings to allow the model to understand the order of words in a sequence.
  - SINCE transformer process all token simultaneously, so it need some additional mechanism to encode the postional information

- Purpose: Since transformers do not utilize recurrence or convolution, they need a way to incorporate the order of words in a sequence. Positional encoding adds information about the position of each word in the input sequence, ensuring that the model understands the sequence's structure.

- Mechanism: Positional encodings are generated using sine and cosine functions, creating unique vectors for each position in the input sequence. These positional vectors are added to the word embeddings, allowing the model to retain both meaning and order

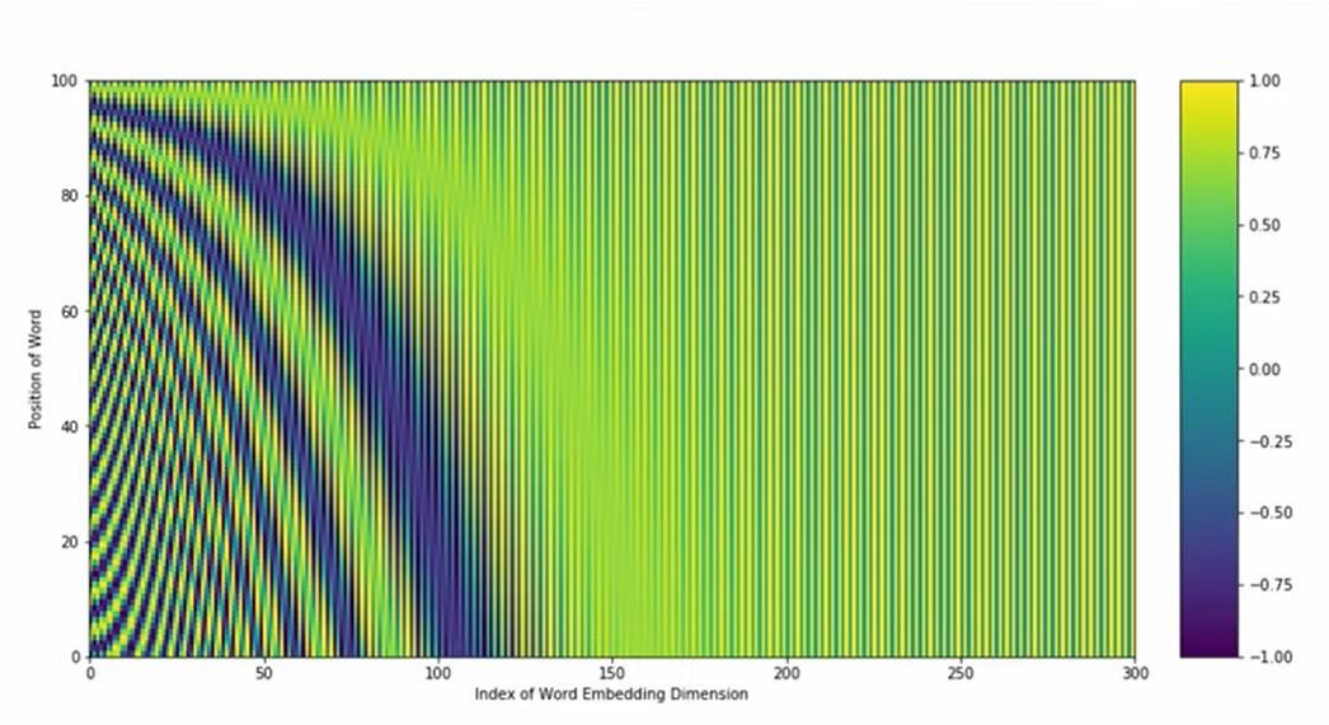
$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Sequence	Index of token, k	Positional Encoding Matrix with d=4, n=100 i=0	i=0	i=1	i=1
I	0	$P_{00}=\sin(0)=0$	$P_{01}=\cos(0)=1$	$P_{02}=\sin(0)=0$	$P_{03}=\cos(0)=1$
am	1	$P_{10}=\sin(1/1)=0.84$	$P_{11}=\cos(1/1)=0.54$	$P_{12}=\sin(1/10)=0.10$	$P_{13}=\cos(1/10)=1.0$
a	2	$P_{20}=\sin(2/1)=0.91$	$P_{21}=\cos(2/1)=-0.42$	$P_{22}=\sin(2/10)=0.20$	$P_{23}=\cos(2/10)=0.98$
Robot	3	$P_{30}=\sin(3/1)=0.14$	$P_{31}=\cos(3/1)=-0.99$	$P_{32}=\sin(3/10)=0.30$	$P_{33}=\cos(3/10)=0.96$

Positional Encoding Matrix for the sequence 'I am a robot'

- **n is a constant set to 100, which determines the wavelength scaling**

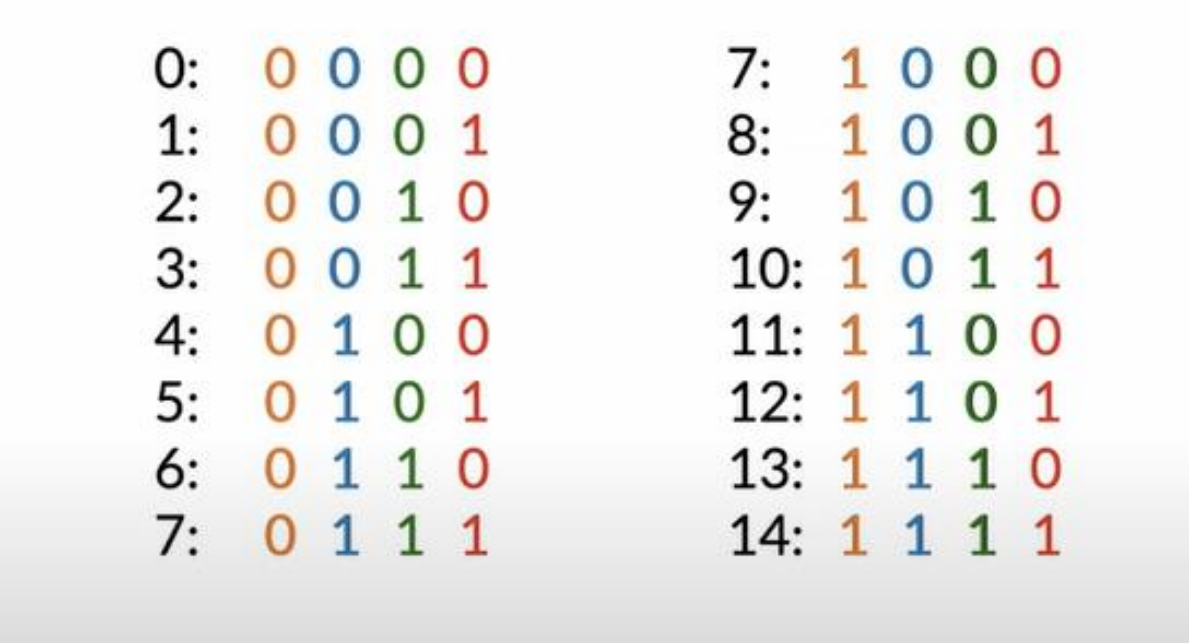


The sine and cosine functions with varying frequencies or wavelengths create distinctive patterns. The colors indicate values ranging from -1 to 1.

- The X-axis represents the length of the word embedding (512 dimensions, as used by the Transformer model).
- The Y-axis represents the position of a token within a sequence.

If a word is at the 20th position, the corresponding horizontal line showcases a unique combination of sine and cosine values for that position. This positional encoding is a distinctive set of values, ensuring that no other word shares the same encoding.

Positional encoding is just like the binary representation of decimal number, which is unique for all numbers.



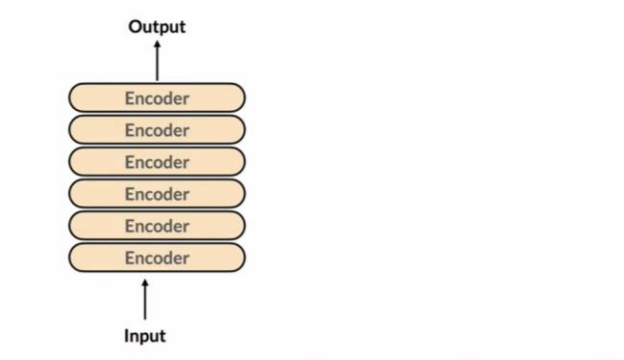
Another way of use binary reparsent. …but sin and cosine are beter

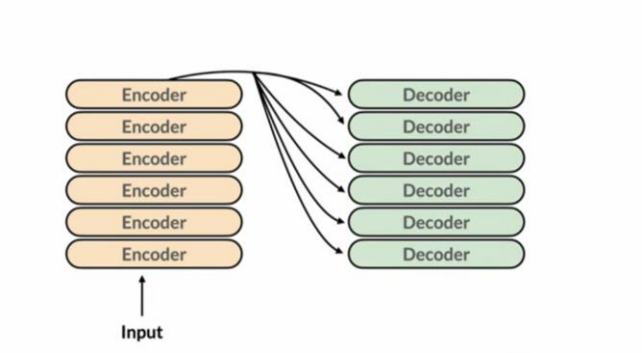
Position Encoding

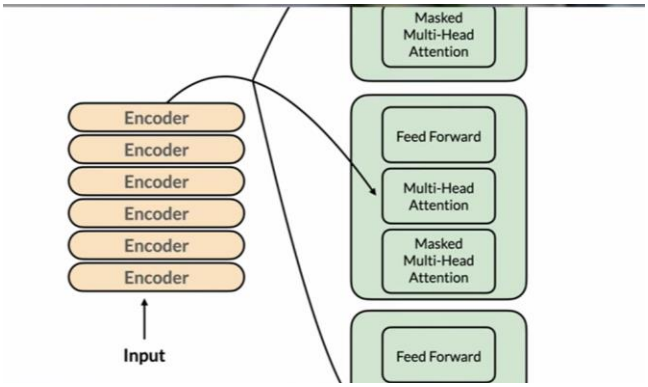
- Can also be learned
- Learn like other parameters

Summey: Encoding Process

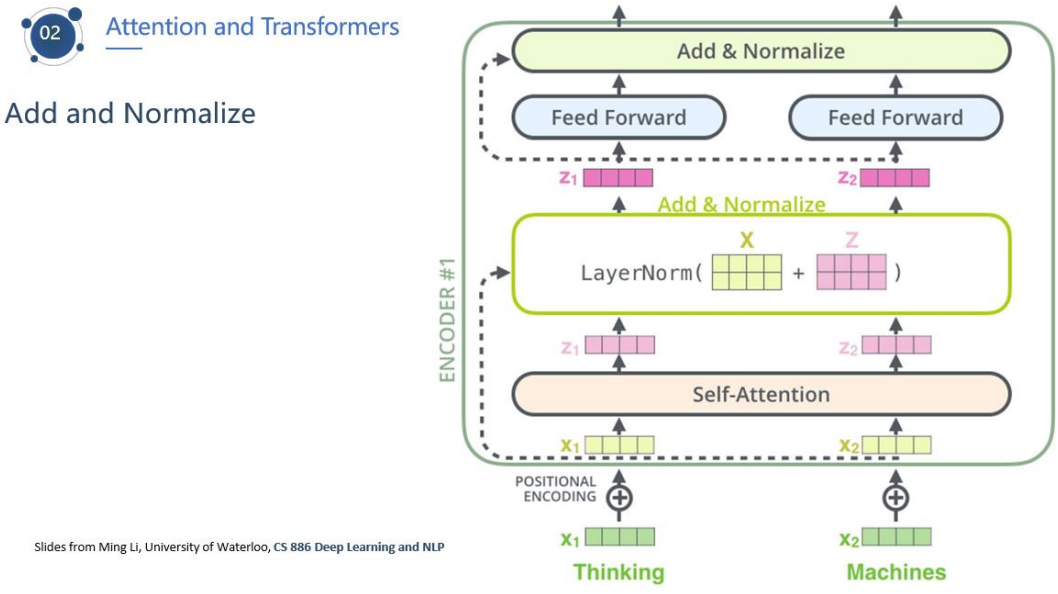
- **The encoding process typically involves:**
    - **Input Preparation: Text is tokenized into words or subwords.**
    - **Embedding Lookup: Each token is converted into its corresponding embedding vector.**
    - **Adding Positional Encoding: The positional encoding vector is added to each word embedding to form a final input representation for the transformer**







3. Add and normalize



Slides from Ming Li, University of Waterloo, CS 886 Deep Learning and NLP

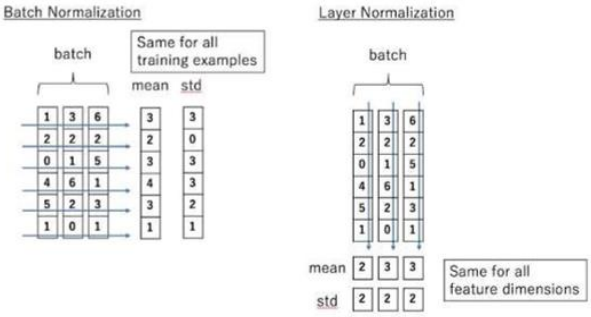
Add: Skip connection just like resnet

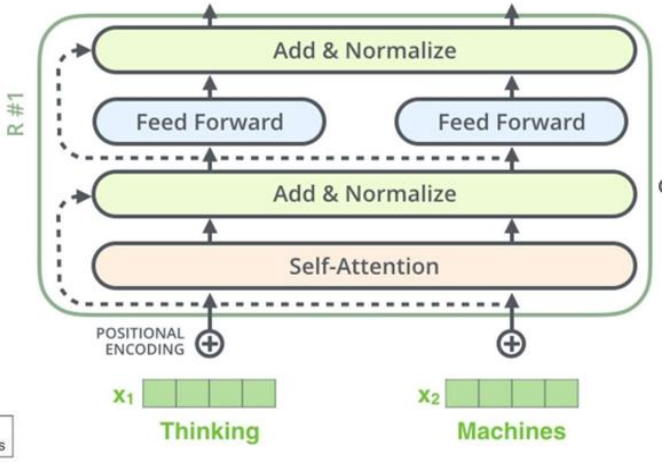
Normalize: layer normalization just like batch normalization

Attention and Transformers

Layer Normalization (Hinton)

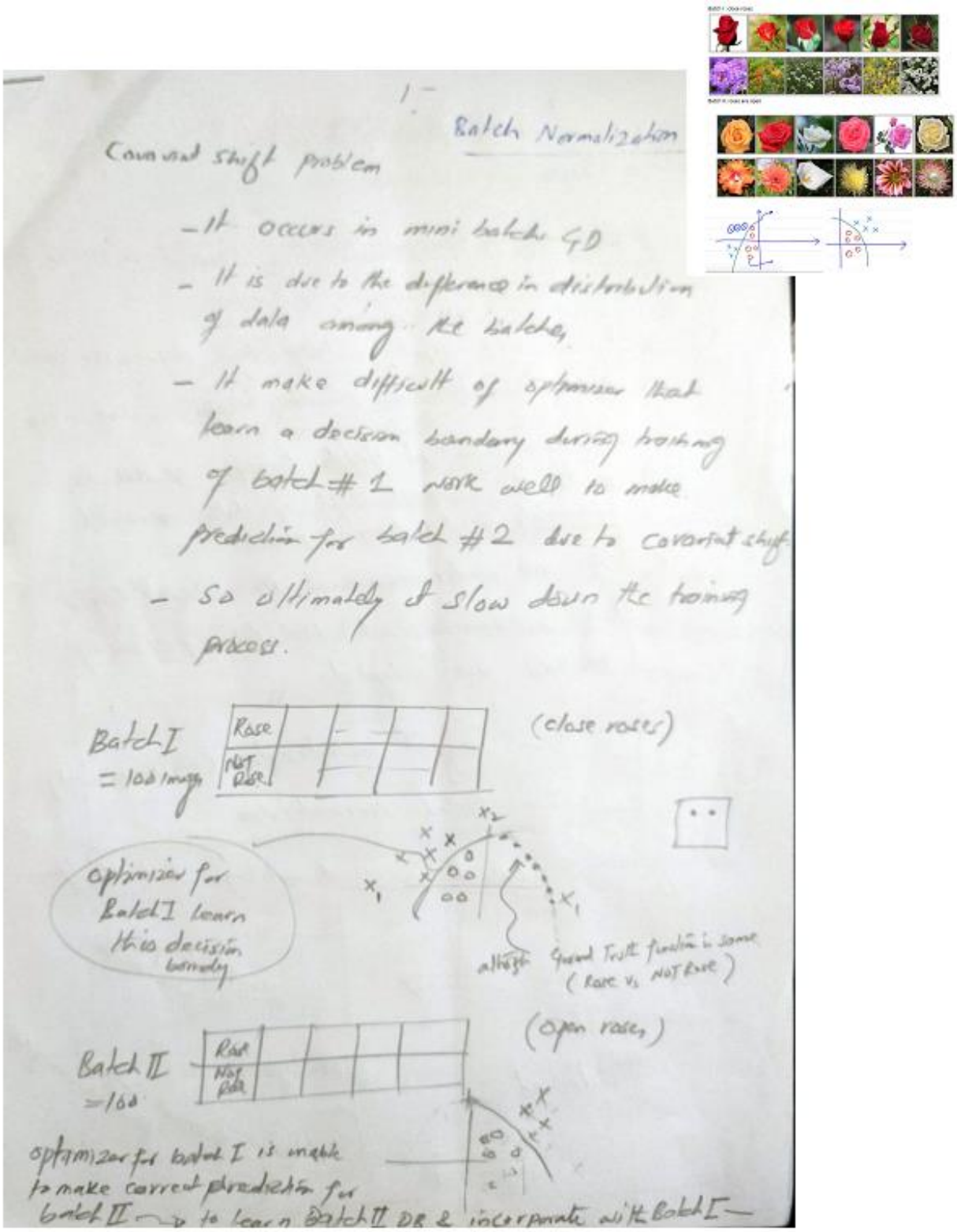
Layer normalization normalizes the inputs across the features.





Slides from Ming Li, Universityof Waterloo, CS 886 DeepLearning and NLP

$$\text{Normalized Activations} = \frac{\text{Activations} - \text{Mean}}{\text{Standard Deviation}}$$



Normalization: helps to overcome the covariance shift problem.

- covariate shift in neural networks refers to a change in the distribution of input data, which can lead to a decrease in model performance.
- Normalization refers to a change in the input data distribution that can occur when the input features have different statistics (e.g., mean and variance/close rose vs open rose) in different parts of the training data. Because the input distributions are more stable, the model learns more quickly. This often allows for the use of higher learning rates, which can significantly reduce training time.

The slide explains *Layer Normalization*, a technique introduced by Geoffrey Hinton, and provides a visual comparison with *Batch Normalization*. Here's a breakdown of the concepts presented:

1. Layer Normalization vs. Batch Normalization:

- **Batch Normalization:** This technique normalizes each input feature by computing the mean and standard deviation (std) across the batch. In essence, normalization occurs over the entire batch for each feature separately. This means that the mean and std are computed across all training examples for each feature dimension, **leading to a consistent scale within each feature across the batch.**
- **Layer Normalization:** Layer normalization normalizes the inputs across the feature dimension for each individual sample, making it independent of batch size
- In contrast, layer normalization normalizes across all feature dimensions for a single example, rather than across a batch. This means that the mean and std are calculated independently for each input example, not across the batch. This technique is beneficial in scenarios where batch statistics may not be stable or applicable, such as when working with sequential data like in Natural Language Processing (NLP) or with Transformers.

Key Differences Highlighted in the Slide:

- **Batch Normalization (Left Side Diagram):** Mean and standard deviation are computed column-wise (across all samples in a batch). The normalization is done independently for each feature.
- **Layer Normalization (Middle Diagram):** Mean and standard deviation are computed across all features of a given sample. This normalization is done per sample, which is why layer normalization is better suited for tasks like text or sequential data processing.

2. Transformer Architecture:

The slide provides a simplified view of a Transformer encoder, with the following key components:

- **Self-Attention:** The self-attention mechanism processes each input word or token (e.g., x1x_1x1 and x2x_2x2 in this example) by considering the relationships between all words/tokens in the input sequence. It captures dependencies regardless of their positions within the sequence.
- **Add & Normalize Layers:** This is where the Layer Normalization occurs. In the diagram, you can see two "Add & Normalize" boxes. In the first, normalization is applied after self-attention. In the

second, it's applied after the feed-forward network. This normalization ensures that the data is scaled appropriately at different stages of the network.

- **Feed-Forward Networks:** These networks apply non-linear transformations to the output of the self-attention mechanism. Each feed-forward block is followed by an "Add & Normalize" operation.
- **Positional Encoding:** Since Transformer models have no inherent understanding of word order, positional encodings are added to input embeddings to provide information about the positions of tokens in the sequence (labeled as 'Thinking' and 'Machines' in the diagram).

Conclusion:

The slide emphasizes how Layer Normalization is applied across features and how it fits into the broader context of Transformer architectures. It visually illustrates the use of normalization in handling dependencies between elements in sequential data (like in NLP) and the importance of normalization in stabilizing and improving deep learning models.

If you have specific questions about any part of the slide or want more details on any section, let me know!

Example:

1. Batch Normalization (BN):

Batch Normalization works by normalizing the activations of a layer across the entire batch. Let's consider a simple neural network example:

Example:

Imagine a fully connected neural network layer with 3 input features and a batch size of 4. Let's assume that the activations for a particular layer look like this:

batch size = 4
input features = 3

$$\text{Activations} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Here, each row represents an example, and each column represents a feature. We want to apply batch normalization to these activations.

Steps for Batch Normalization:

1. **Compute the Mean and Standard Deviation for each feature across the batch:**

$$\text{Mean} = \left[\tfrac{1+4+7+10}{4}, \quad \tfrac{2+5+8+11}{4}, \quad \tfrac{3+6+9+12}{4}\right] = \left[5.5, \quad 6.5, \quad 7.5\right]$$

$$\text{Standard Deviation} = \left[\text{std}(1,4,7,10), \quad \text{std}(2,5,8,11), \quad \text{std}(3,6,9,12)\right] \approx \left[3.5, \quad 3.5, \quad 3.5\right]$$

2. **Normalize each feature in the batch using the mean and standard deviation:**

$$\text{Normalized Activations} = \frac{\text{Activations} - \text{Mean}}{\text{Standard Deviation}}$$

Applying this to our activations:

$$\begin{bmatrix} \frac{1-5.5}{3.5}, & \frac{2-6.5}{3.5}, & \frac{3-7.5}{3.5} \\ \frac{4-5.5}{3.5}, & \frac{5-6.5}{3.5}, & \frac{6-7.5}{3.5} \\ \frac{7-5.5}{3.5}, & \frac{8-6.5}{3.5}, & \frac{9-7.5}{3.5} \\ \frac{10-5.5}{3.5}, & \frac{11-6.5}{3.5}, & \frac{12-7.5}{3.5} \end{bmatrix} = \begin{bmatrix} -1.29, & -1.29, & -1.29 \\ -0.43, & -0.43, & -0.43 \\ 0.43, & 0.43, & 0.43 \\ 1.29, & 1.29, & 1.29 \end{bmatrix}$$

3. **Apply learnable parameters (scale and shift):** After normalizing, we typically apply scaling and shifting parameters (γ and β), which are learnable during training.

2. Layer Normalization (LN):

Layer Normalization normalizes the activations across all features for each individual example in a batch.

Example:

Let's consider the same activation matrix as before:

$$\text{Activations} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Steps for Layer Normalization:

1. **Compute the Mean and Standard Deviation for each example (row-wise calculation):**
   - For the first example: $[1, 2, 3]$:

     $$\text{Mean} = \frac{1+2+3}{3} = 2, \quad \text{Standard Deviation} = \sqrt{\frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{3}} = 1$$

   - For the second example: $[4, 5, 6]$:

     $$\text{Mean} = \frac{4+5+6}{3} = 5, \quad \text{Standard Deviation} = 1$$

     Similarly, we find the mean and standard deviation for the remaining examples.

2. **Normalize each feature for each example using the example's mean and standard deviation:**
   - For the first example:

     $$\text{Normalized Activations} = \left[\frac{1-2}{1}, \frac{2-2}{1}, \frac{3-2}{1}\right] = [-1, 0, 1]$$

   - For the second example:

     $$\text{Normalized Activations} = \left[\frac{4-5}{1}, \frac{5-5}{1}, \frac{6-5}{1}\right] = [-1, 0, 1]$$

     Applying these to all examples:

     $$\text{Normalized Activations} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

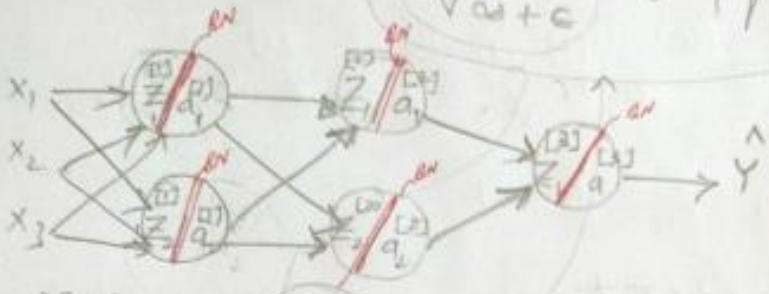
3. **Apply learnable parameters (scale and shift):** As with batch normalization, we typically apply scaling (γ) and shifting (β) parameters here too.

Summary of Key Differences:

- **Batch Normalization:** Normalizes across a batch for each feature. Primarily used in CNNs and dense layers.
- **Layer Normalization:** Normalizes across features for each example individually. Typically used in RNNs, LSTMs, and Transformers.

Adding Batch Norm to a NN

$$\tilde{Z} = \frac{Z - \mu}{\sqrt{\sigma^2 + \epsilon}} * \gamma + \beta \quad \rightarrow Z_{norm}$$



$X \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \text{Batch (BN) Norm} \rightarrow \tilde{Z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \rightarrow Z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \text{BN} \rightarrow \tilde{Z}^{[2]} \rightarrow a^{[2]} = g^{[2]}(\tilde{Z}^{[2]}) \rightarrow \cdots$

- Compute μ & σ
- Each example is subtracted from mean & scale by variance
- Rescale by γ & β

- Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \ldots W^{[l]}, b^{[l]}$

  $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]} \ldots \beta^{[l]}, \gamma^{[l]}$

  (additional parameters)

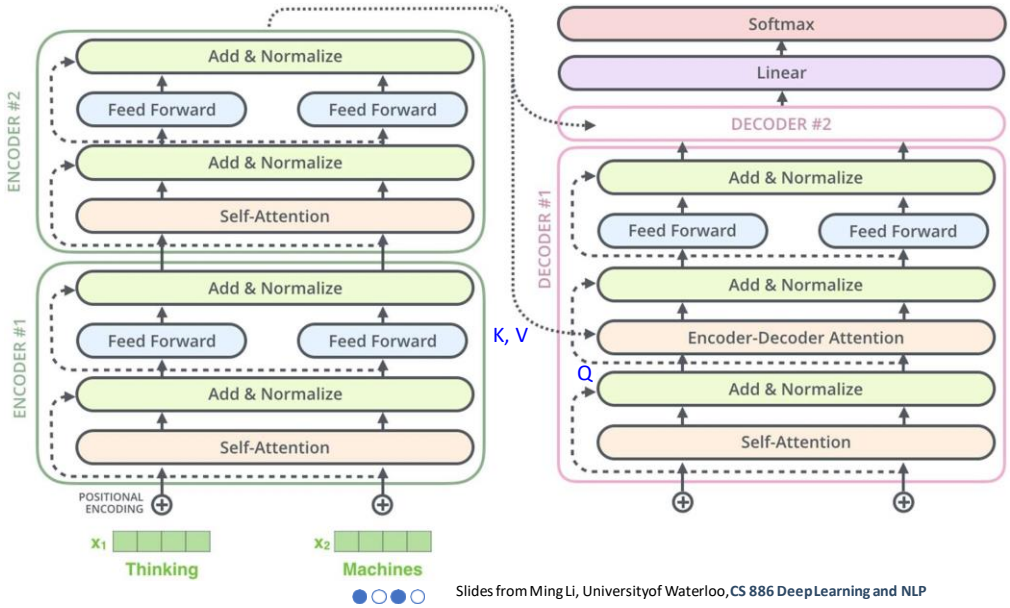
  (β in this context are different from BN)

- Compute $d\beta^{[l]}$ & $d\gamma^{[l]}$ during back propagation

$$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$$

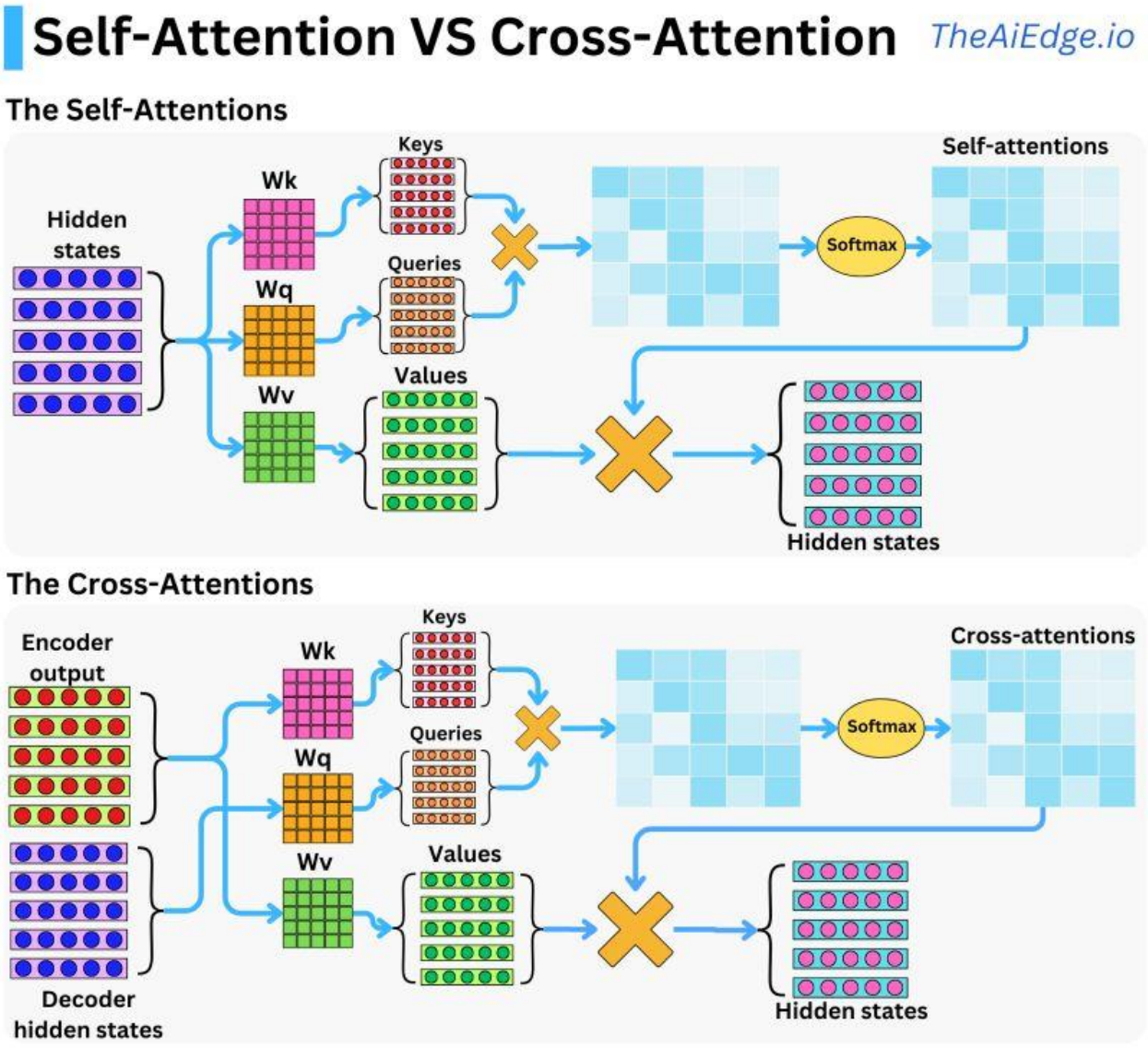
- Will will use Pytorch framework that do it for us (torch.nn.BatchNorm)

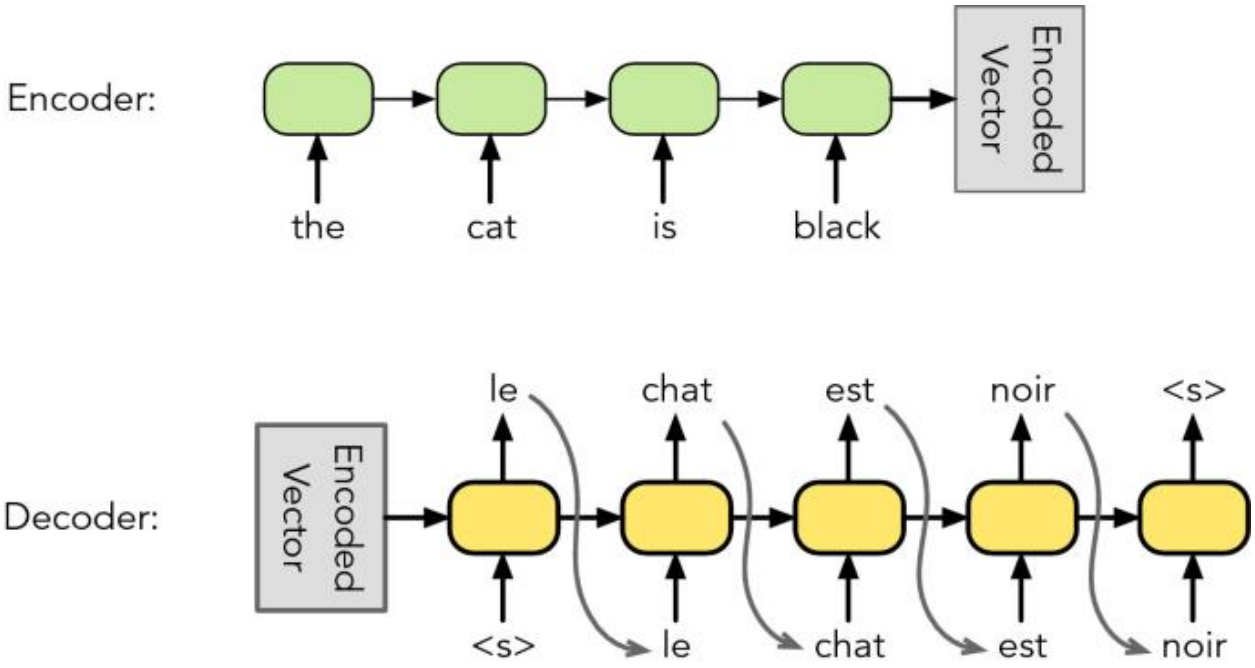
The complete transformer



Slides from Ming Li, Universityof Waterloo, CS 886 DeepLearning and NLP

Lecture 04:


Self-Attention VS Cross-Attention
TheAiEdge.io
The Self-Attentions
Hidden states
Wk
Wq
Wv
Keys
Queries
Values
Softmax
Self-attentions
Hidden states
The Cross-Attentions
Encoder output
Decoder hidden states
Wk
Wq
Wv
Keys
Queries
Values
Softmax
Cross-attentions
Hidden states



Toy Example

Let's consider a simple example where we translate "The cat sits on the mat" from English to French.

Step 1: Encoding

- The encoder processes the input sentence and generates hidden states for each word:
    - "The" → h_1
    - "cat" → h_2
    - "sits" → h_3
    - "on" → h_4
    - "the" → h_5
    - "mat" → h_6

Step 2: Decoding with Cross Attention [During cross-attention, this query vector interacts with all key vectors (K_1 to K_7). The key vector for "deposited" (K_2) receives a high attention weight because it closely matches the decoder's query vector, allowing the decoder to generate "déposé"]

- Suppose we are generating the first French word "Le" (the equivalent of "The"). The decoder creates a query vector based on its current state.
- As the decoder processes each token, it uses cross attention to focus on relevant parts of the encoder's output. For generating "Le," the decoder creates a query vector based on its current state (which starts with <SOS>). [This is often represented by a special token, such as <SOS> (start of sentence). This token is fed into the decoder to initiate the generation process]
- It computes **attention scores** using this query with all encoder hidden states:
    - Score for h_1 (The): high
    - Score for h_2 (cat): medium
    - Score for h_3 (sits): low
    - Score for h_4 (on): low
    - Score for h_5 (the): low
    - Score for h_6 (mat): low

Step 3: Weighted Sum

- After applying softmax to these scores, we get attention weights:
    - High weight on "The", lower weights on others.
- The context vector is computed as:

    Context = weight1·h1 + weight2·h2 +... + weight6·h6

Step 4: Generating Output

- The decoder uses this context vector to generate "Le". This process continues for subsequent words like "chat" (cat), where it will again use cross attention to focus on relevant parts of the encoder's output based on its current query.

- Iterative Process: The generated word "Le" is then fed back into the decoder as part of its input for generating subsequent words in the translation (e.g., "chat" for cat). This process continues until an end-of-sentence token is produced.

4. Cross-Attention Mechanism (Encoder-Decoder Attention) in Detail

This cross-attention layer is what enables the model to “look back” at the input sequence while generating the output sequence, ensuring the output words are contextually aligned with the input.

Summary of Key Points:

- **Query from Decoder**: Represents what the decoder is currently focusing on.
- **Keys and Values from Encoder**: Capture the processed information from the input sequence.
- **Cross-Attention Mechanism**: Matches the query from the decoder with the keys from the encoder to determine which input elements to focus on.
- **Context Vector**: A weighted combination of the values based on attention scores, guiding the decoder in generating output.

Let’s go through a simple translation example using the Transformer model and see how the **cross-attention mechanism** (encoder-decoder attention) works step-by-step.

Example:

- Imagine translating the
- English sentence "`She ate an apple`" into French as "`Elle a mangé une pomme`".

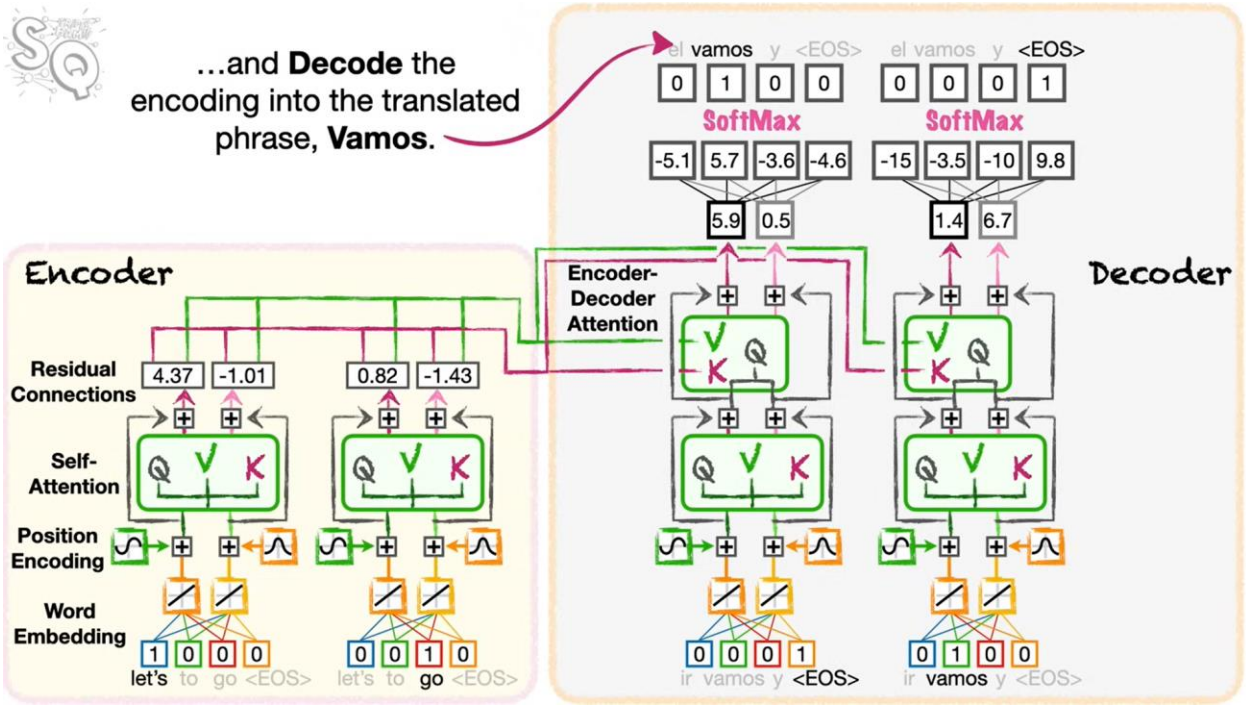
- The key goal of the cross-attention mechanism is to help the decoder pay attention to the most relevant parts of the encoded input sequence while generating each word in the translated output.

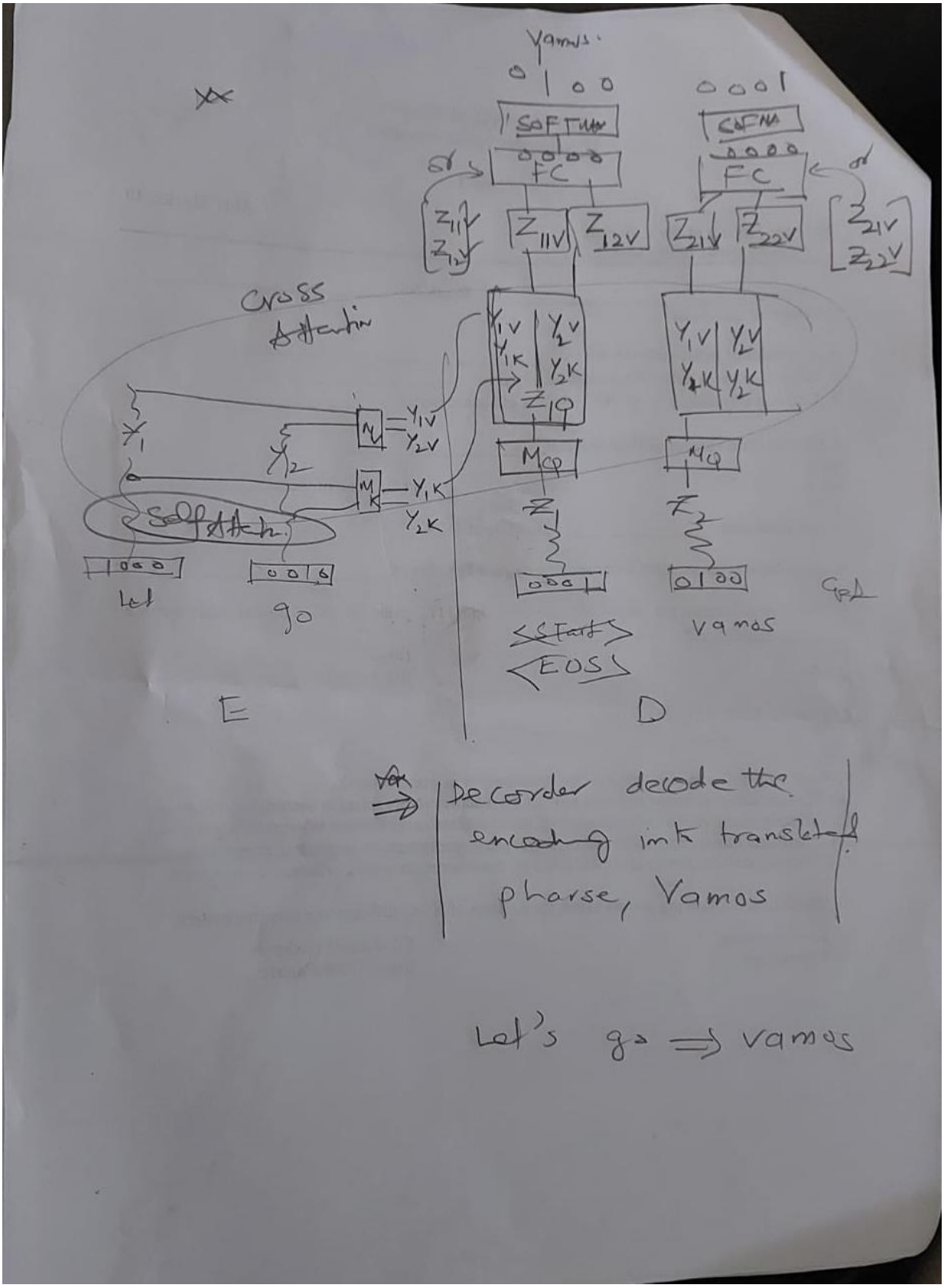
Let’s go through a simple translation example using the Transformer model and see how the **cross-attention mechanism** (encoder-decoder attention) works step-by-step.

Example:

Imagine translating the English sentence "She ate an apple" into French as "Elle a mangé une pomme".

The key goal of the cross-attention mechanism is to help the decoder pay attention to the most relevant parts of the encoded input sequence while generating each word in the translated output.

- 


Vamos:
0100
0001
SOFTMAX
SOFTMAX
FC
FC
or
or
Z_{11} Z_{12}
Z_{11V} Z_{12V}
Z_{21V} Z_{22V}
Z_{21} Z_{22}
Cross Attention
Y_1
Y_2
Y_{1V} Y_{2V}
Y_{1K} Y_{2K}
Z_{1Q}
M_Q
M_Q
Z_1
Z_2
Self Attention
0001
0010
0100
Let
go
<Start>
<EOS>
Vamos
E
D
Decorder decode the encoding into translate pharse, Vamos
Let's go ⇒ vamos

Next time:

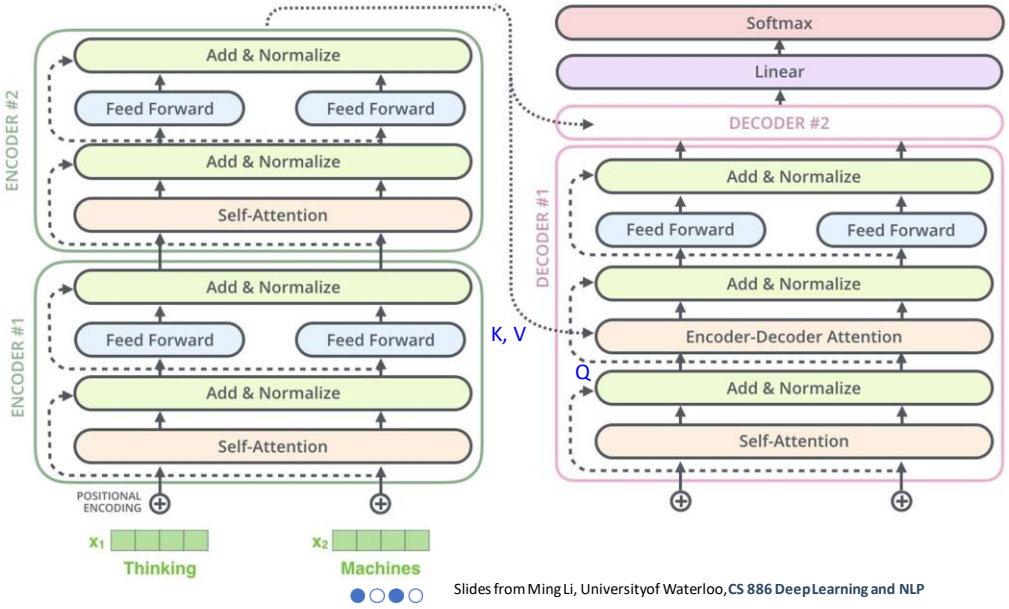
- **WordtoVec encoding**
- **Positional encoding,**
- **Encoder layers**
- **Deconder layers**
-

Lecture 4

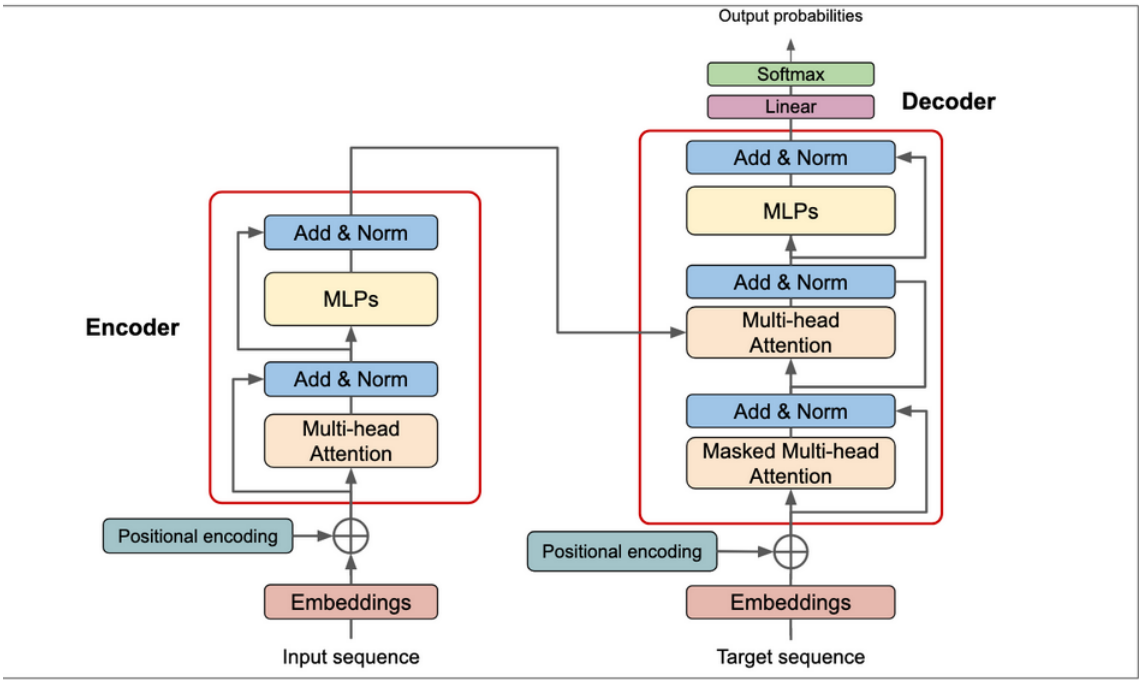
- **Cross attention**
- **Mask multihead layers**

02 Attention and Transformers

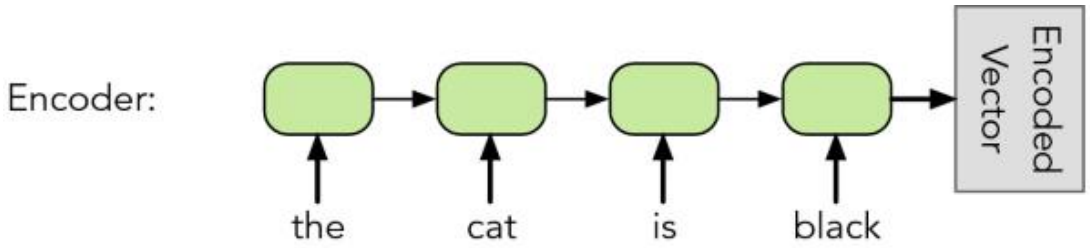
The complete transformer

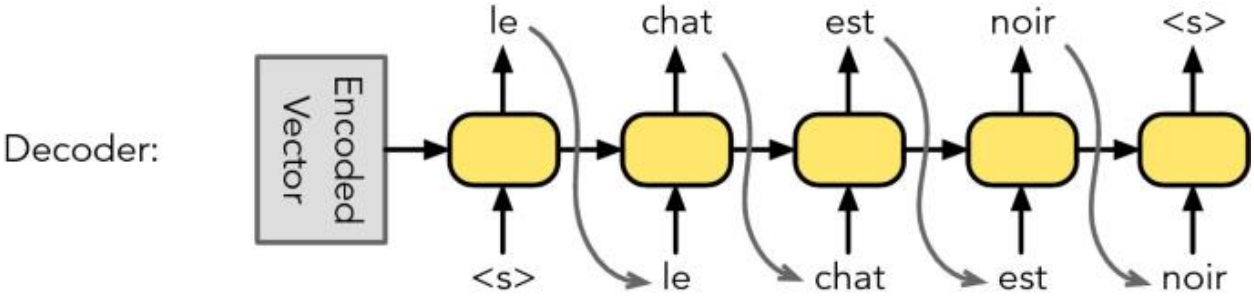


Slides from Ming Li, Universityof Waterloo, CS 886 Deep Learning and NLP



https://deeprevision.github.io/posts/001-transformer/





Deconder: Output in auto regressive way

What is Masked Self-Attention?

- **Masked self-attention is a mechanism used in the decoder of transformer models that allows the model to focus on previously generated words while generating the next word.**
- **It prevents the model from attending to future tokens, ensuring that the generation is autoregressive (i.e., it only considers words that have already been generated).**

Cross Attention

- **Cross Attention is a mechanism used in transformer models, particularly in the decoder, to allow the model to focus on relevant parts of the input sequence (provided by the encoder) while generating output.**

Context Vector:

  - **Represents the relationship between the current query (from the decoder) and the relevant information from the encoder's output.**
  - **Provides crucial context from the input sentence to aid in generating a coherent output.**

Hidden State: Tells the model what has been generated so far.

  - **Helps guide the model in predicting the next word based on previously generated words.**
  - **Hidden_state_current = Hidden_state_previous + Context_vector_current**

Example:

Translating "The cat eats fish" into French ("Le chat mange du poisson")

Suppose each word's embedding (from the encoder) is represented as a vector with three dimensions (for simplicity). We'll also assign the decoder's hidden state a similar 3-dimensional vector representation.

a. Encoder Output

Let's say the encoder generates embeddings for each word in the English sentence "The cat eats fish":

- `E_the = [0.5, 0.1, 0.3]`

- `E_cat = [0.6, 0.4, 0.2]`

- `E_eats = [0.7, 0.3, 0.5]`

- `E_fish = [0.2, 0.8, 0.6]`

[they are contextual vectors that contain, semantic, positional, and contextual information]

b. Decoder Hidden State Evolution (Example Values)

At each decoding step, the decoder will update its hidden state based on self-attention, cross-attention context provided by the encoder's embeddings

Step 1: Generating the First Word ("Le")

1. Initialization with `<start>` Token:

- The decoder starts with a hidden state initialized from the `<start>` token, say:

Hidden_state_0 = [0.1, 0.1, 0.1]

2. Cross-Attention Calculation:

- Attention weights are calculated for each encoder embedding to focus on the most relevant words for generating the first word. Suppose the weights for "The" and "cat" are higher:

Attention weights: [Weight_the = 0.6, Weight_cat = 0.4, Weight_eats = 0.0, Weight_fish = 0.0]

Attention weights: [0.6, 0.4, 0.0, 0.0] // <start> cross attention with encoder embedding

- Compute the context vector by applying these weights to the encoder embeddings:

Context_vector_1 = (0.6 * [0.5, 0.1, 0.3]) + (0.4 * [0.6, 0.4, 0.2]) = [0.54, 0.22, 0.26]

Context_vector_1 = [0.54, 0.22, 0.26]

3. Update Hidden State:

- The initial hidden state and context vector combine to form the new hidden state for predicting the first word. Suppose a simple addition operation (for illustration):

Hidden_state_1 = Hidden_state_0 + Context_vector_1 = [0.1, 0.1, 0.1] + [0.54, 0.22, 0.26] =

Hidden_state_1 = [0.64, 0.32, 0.36]

- This hidden state represents the context needed to generate the word "Le"

Hidden_state_1 gives the model the context needed to understand what word to generate next. The softmax function then helps convert this context into a probability for each possible word, allowing the model to choose the best word, which in this case is "Le"

Step 2: Generating the Second Word ("chat")

1. Masked Self-Attention on "Le":

- The hidden state now includes the word "Le," so the decoder applies masked self-attention to reference this.

2. Cross-Attention Calculation: (here we only take "Le" embeddings as query vector)

- New cross-attention weights focus more on `E_cat` for the next word:

Attention weights: [Weight_the = 0.1, Weight_cat = 0.9, Weight_eats = 0.0, Weight_fish = 0.0]

Attention weights: [0.1, 0.9, 0.0, 0.0]

- Compute the Context Vector:

Context_vector_2 = (0.1 * [0.5, 0.1, 0.3]) + (0.9 * [0.6, 0.4, 0.2]) = [0.59, 0.37, 0.21]

Context_vector_2 = [0.59, 0.37, 0.21]

3. Update Hidden State:

- Combine this with the previous hidden state:

Hidden_state_2 = Hidden_state_1 + Context_vector_2 = [0.64, 0.32, 0.36] + [0.59, 0.37, 0.21] = [1.23, 0.69, 0.57]

Hidden_state_2 = [1.23, 0.69, 0.57]

- This hidden state is now used to generate "chat."

Step 3: Generating the Third Word ("mange")

1. Masked Self-Attention on "Le chat":

- The hidden state includes both "Le" and "chat," so masked self-attention allows it to reference these without "looking ahead."

2. Cross-Attention Calculation:

- Attention now shifts to `E_eats`:

Attention weights[Weight_the = 0.0, Weight_cat = 0.0, Weight_eats = 1.0, Weight_fish = 0.0]

Attention weights[0.0, 0.0, 1.0, 0.0]

- Context vector:

Context_vector_3 = [0.7, 0.3, 0.5]

3. **Update Hidden State:**

- Combine with the previous hidden state:

Hidden_state_3 = Hidden_state_2 + Context_vector_3 = [1.23, 0.69, 0.57] + [0.7, 0.3, 0.5] = [1.93, 0.99, 1.07]

Hidden_state_3 = [1.93, 0.99, 1.07]

- This hidden state is used to predict "mange."

This process continues for each word in the sequence. Each hidden state is a combination of prior context and relevant encoder information, allowing the decoder to maintain a fluent and contextually accurate translation.

These hidden states, which evolve with each generation step, ensure that the translation flows naturally and aligns with the meaning of the original sentence.