Transformers

Bahar Niknam

Overview

Introduction to Sequence-to-Sequence models (Seq2Seq)

1. Applications of Seq2Seq Models
2. Working with Sequences
3. language models
4. Key Components of a Seq2Seq Model
5. RNN based model

Attention Mechanisms and Transformers

1. Queries, Keys, and Values
2. Scaled Dot Product Attention
3. Additive attention
4. The Bahdanau Attention Mechanism for RNN-based.
5. Multi-Head Attention
6. Self-Attention and Positional Encoding

Residual Connections

The Transformer Architecture

1. Encoder stack
2. Decoder stack

Introduction to Sequence-to-Sequence models (Seq2Seq)

When introducing linear and logistic regression and multilayer perceptron, we assume that each feature vector xi consisted of a fixed number of components $x1$, . . . , xd, where each numerical feature xj corresponded to a particular attribute. These datasets are sometimes called tabular, because they can be arranged in tables, where each example i gets its own row, and each attribute gets its own column. Crucially, with tabular data, we seldom assume any particular structure over the columns.

we moved on to image data, where inputs consist of the raw pixel values at each coordinate in an image. Image data hardly fitted the bill of a tabular dataset. There, we needed to call upon convolutional neural networks (CNNs) to handle the hierarchical structure and invariances. However, our data were still of fixed length. Every Fashion-MNIST image is represented as a 28×28 grid of pixel values. Moreover, our goal was to develop a model that looked at just one image and then outputted a single prediction. But what should we do when faced with a sequence of images, as in a video, or when tasked with producing a sequentially structured prediction, as in the case of image captioning?

A great many learning tasks require dealing with sequential data. Image captioning, speech synthesis, and music generation all require that models produce outputs consisting of sequences. In other domains, such as time series prediction, video analysis, and musical information retrieval, a model must learn from inputs that are sequences. These demands often arise simultaneously: tasks such as translating passages of text from one natural language to another, engaging in dialogue, or controlling a robot, demand that models both ingest and output sequentially structured data.

One key insight paved the way for a revolution in sequence modeling. While the inputs and targets for many fundamental tasks in machine learning cannot easily be represented as fixed-length vectors, they can often nevertheless be represented as varying-length sequences of fixed-length vectors. For example, documents can be represented as sequences of words; medical records can often be represented as sequences of events (encounters, medications, procedures, lab tests, diagnoses); videos can be represented as varying-length sequences of still images.

Applications of Seq2Seq Models

Machine Translation: Translating text from one language to another.

Text Summarization: Producing a concise summary of a text document.

Speech Recognition: Converting audio signals (sequence of features) into text sequences.

Dialogue Systems: Generating responses in conversational AI.

Image Captioning: Describing images with natural language (when combined with CNNs for image feature extraction).

Working with Sequences

Text is among the most common forms of sequence data encountered in deep learning.

we will need some basic tools for converting raw text into sequences of the appropriate form. Typical preprocessing pipelines execute the following steps:

1. Load text as strings into memory.
2. Split the strings into tokens (e.g., words or characters).
3. Build a vocabulary dictionary to associate each vocabulary element with a numerical index.
4. Convert the text into sequences of numerical indices.

language models

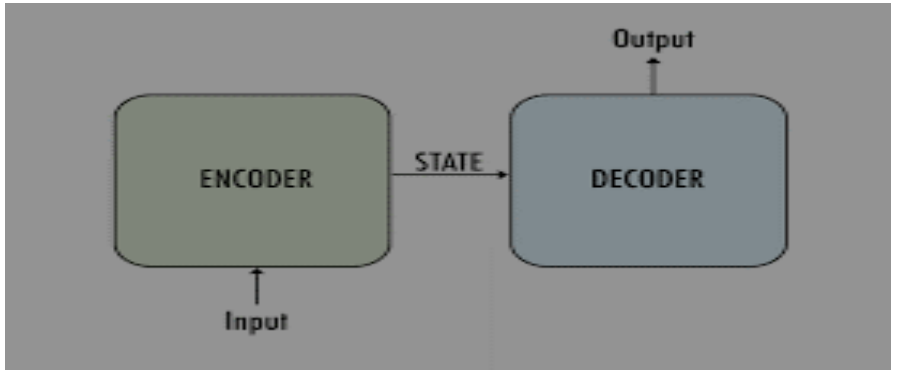
Sometimes, especially when working with language, we wish to estimate the joint probability of an entire sequence. This is a common task when working with sequences composed of discrete tokens, such as words. Generally, these estimated functions are called sequence models and for natural language data, they are called language models. The field of sequence modeling has been driven so much by natural language processing, that we often describe sequence models as “language models”, even when dealing with non-language data. Language models prove useful for all sorts of reasons. Sometimes we want to evaluate the likelihood of sentences. For example, we might wish to compare the naturalness of two candidate outputs generated by a machine translation system or by a speech recognition system. But language modeling gives us not only the capacity to evaluate likelihood, but the ability to sample sequences, and even to optimize for the most likely sequences.

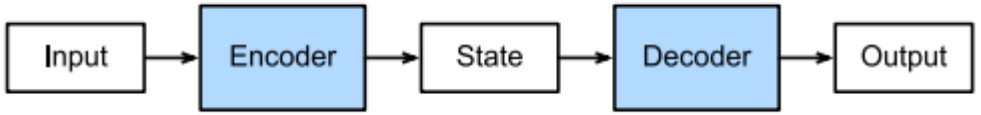
Language models are incredibly useful. For instance, an ideal language model should generate natural text on its own, simply by drawing one token at a time $x_t \sim \boldsymbol{P(x_t \mid x_{t-1}, \ldots, x_1)}$. Quite unlike the monkey using a typewriter, all text emerging from such a model would pass as natural language, e.g., English text. Furthermore, it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments. Clearly we are still very far from designing such a system, since it would need to understand the text rather than just generate grammatically sensible content.

Language models estimate the joint probability of a text sequence. For long sequences, n-grams provide a convenient model by truncating the dependence. However, there is a lot of structure but not enough frequency to deal efficiently with infrequent word combinations via Laplace smoothing. Thus, we will focus on neural language modeling. To train language models, we can randomly sample pairs of input sequences and target sequences in minibatches. After training, we will use perplexity to measure the language model quality. Language models can be scaled up with increased data size, model size, and amount in training compute. Large language models can perform desired tasks by predicting output text given input text instructions.

Key Components of a Seq2Seq Model

In general **sequence-to-sequence problems** like machine translation, **inputs and outputs are of varying lengths** that are unaligned. The **standard approach to handling this sort of data is to design an encoder–decoder architecture**. consisting of two major components: an encoder that takes a variable-length sequence as input, and a **decoder that acts as a conditional language model**, taking in the encoded input and the **leftwards** context of the target sequence and predicting the subsequent token in the target sequence.





1. Encoder:

Purpose: Encodes the input sequence into a fixed-length vector representation, often referred to as a "context vector" or "thought vector."

Architecture: Usually implemented with Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), Gated Recurrent Units (GRUs), or Transformers.

Process: The encoder processes the input sequence step-by-step.

Each input token is transformed into a hidden state that captures its context.

The final hidden state (or a combination of hidden states) summarizes the entire input sequence.

2. Decoder:

Purpose: Generates the output sequence based on the encoded representation.

Architecture: Similar to the encoder, it uses RNNs, LSTMs, GRUs, or Transformers.

Process: Takes the context vector (from the encoder) and generates one output token at a time in an autoregressive manner.

At each step, the decoder's input includes the previous token and its own hidden state.

RNN based model

In so-called sequence-to-sequence problems such as machine translation where inputs and outputs each consist of variable-length unaligned sequences, we generally rely on encoder–decoder architectures. we will demonstrate the application of an encoder–decoder architecture, where **both the encoder and decoder are implemented as RNNs, to the task of machine translation (Cho et al.,2014, Sutskever et al., 2014).**

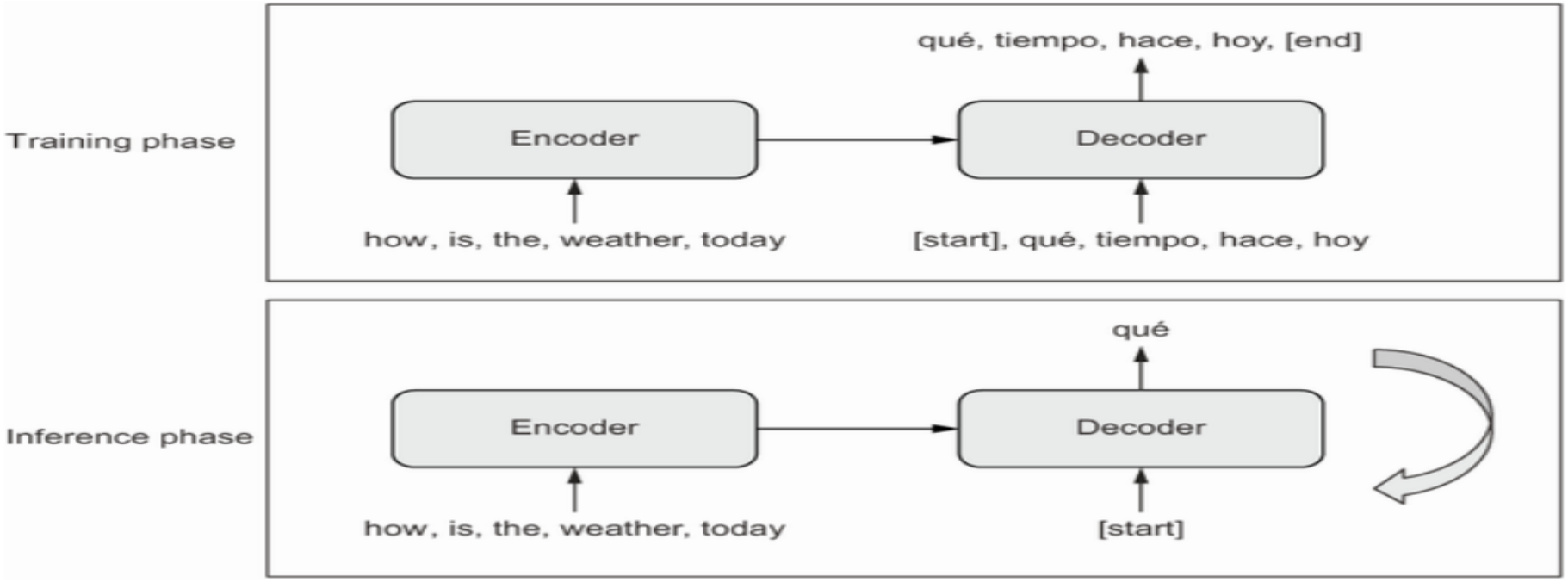
the encoder RNN will take a variable-length sequence as input and transform it in to a fixed-shape hidden state.

we will introduce attention mechanisms, which allow us to access encoded inputs without having to compress the entire input into a single fixed-length representation.

Then to generate the output sequence, one token at a time, **the decoder model, consisting of a separate RNN, will predict each successive target token given both the input sequence and the preceding tokens in the output.** During training, the decoder will typically be conditioned upon the preceding tokens in the official "ground truth" label. However, at test time, we will want to condition each output of the decoder on the tokens already predicted.

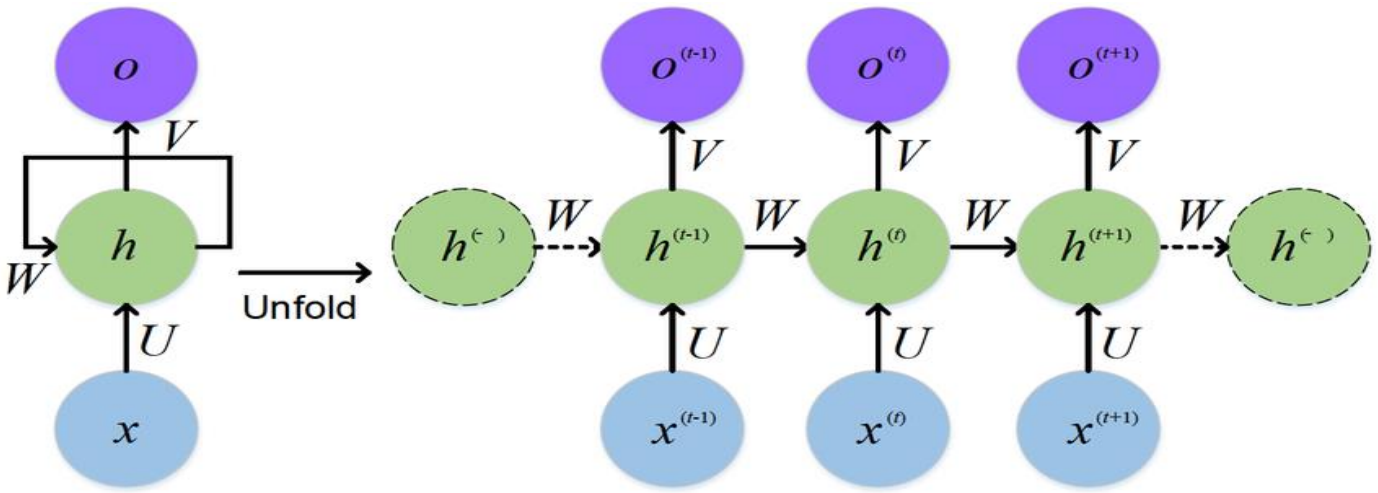
Note that if we **ignore the encoder**, the decoder in a sequence-to-sequence architecture be haves just like a **normal language model.** illustrates how to use two RNNs for sequence-to-sequence learning in machine translation.

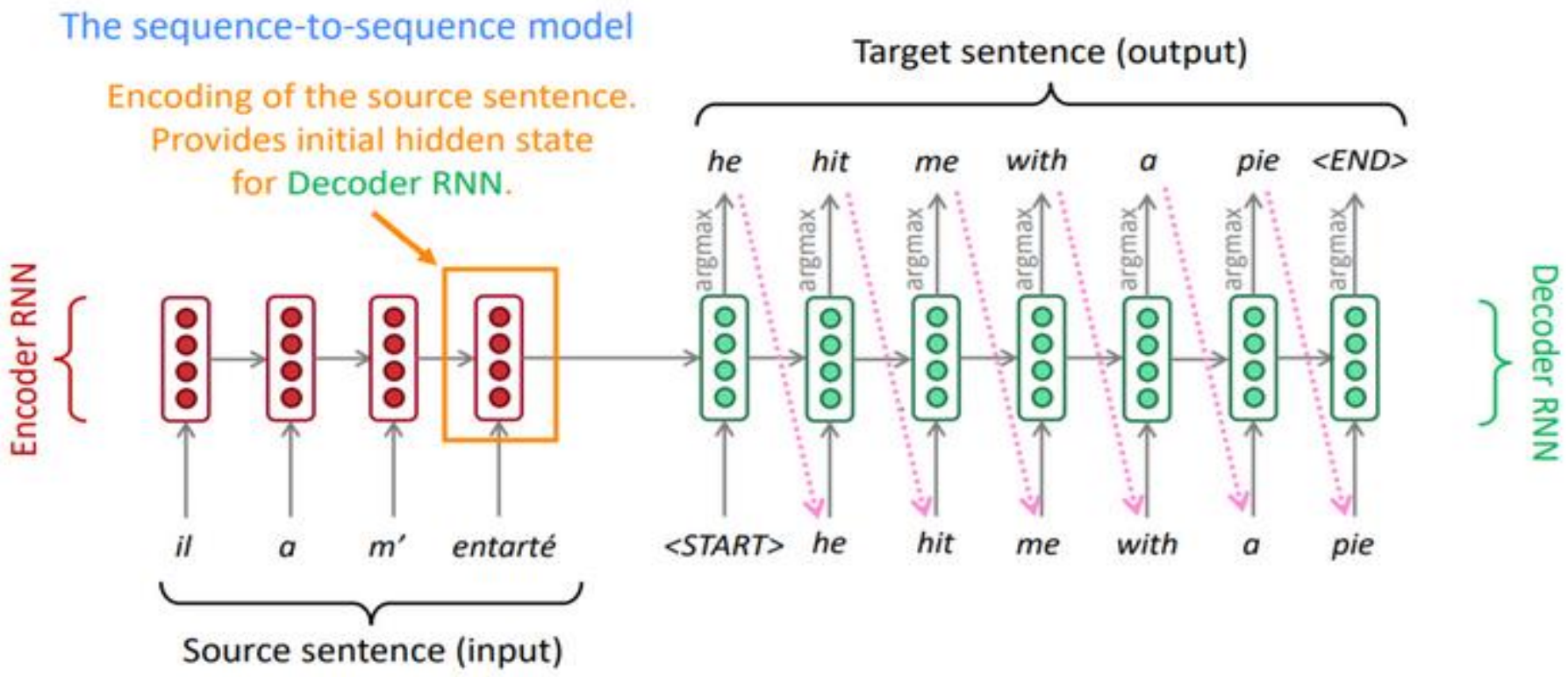
Figure 11.12 Sequence-to-sequence learning: the source sequence is processed by the encoder and is then sent to the decoder. The decoder looks at the target sequence so far and predicts the target sequence offset by one step in the future. During inference, we generate one target token at a time and feed it back into the decoder.



Sequence-to-sequence learning

with an RNN encoder and an RNN decoder.




The sequence-to-sequence model
Encoding of the source sentence.
Provides initial hidden state
for Decoder RNN.
Target sentence (output)
Encoder RNN
Decoder RNN
argmax
he
hit
me
with
a
pie
<END>
il
a
m'
entarté
<START>
Source sentence (input)

Encoder-Decoder

Here we discuss how an **RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length.** This comes up in many applications, such as speech recognition, machine translation and question answering, where the input and output sequences in the training set are generally not of the same length (although their lengths might be related).

We often call **the input to the RNN the "context."** We want to produce a representation of this context, C. **The context C might be a vector or sequence of vectors that summarize the input sequence X= $(x^{(1)}, \ldots, x^{(n_x)})$.**

The encoder-decoder or sequence-to-sequence architecture:

- The idea is very simple:

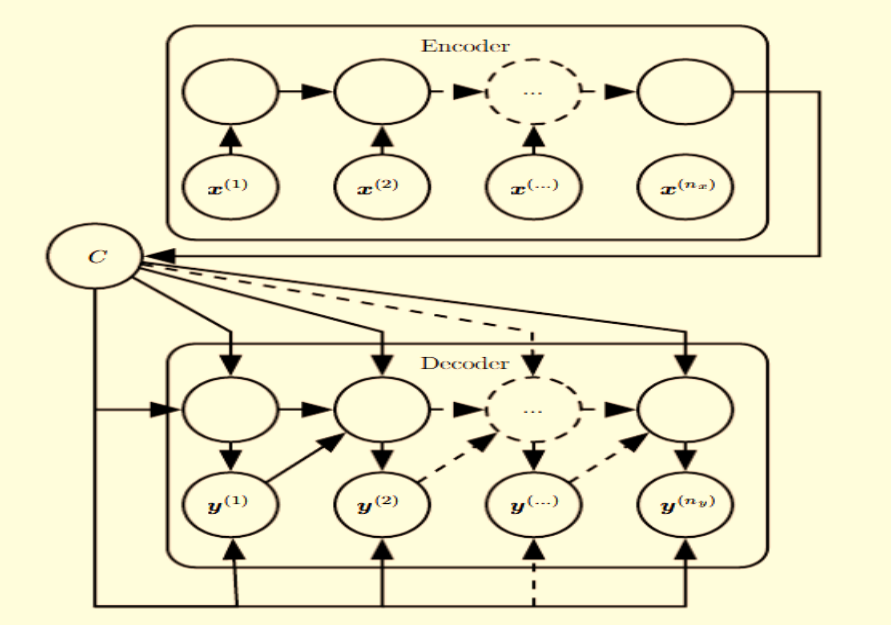
1) **An Encoder Or Reader Or Input RNN** processes the input sequence. The encoder emits the context C, usually as a simple function of its final hidden state.

(2) **A Decoder Or writer Or output RNN** is conditioned on that fixed-length vector

to generate the output sequence Y= $(y^{(1)}, \ldots, y^{(n_y)})$.

The innovation of this kind of architecture is that the **lengths n_x and n_y can vary from each other.**

- **Example of an encoder-decoder or sequence-to-sequence RNN architecture**, for learning to generate an output sequence Y= $(y^{(1)}, \ldots, y^{(n_y)})$. given an input sequence X= $(x^{(1)}, \ldots, x^{(n_x)})$

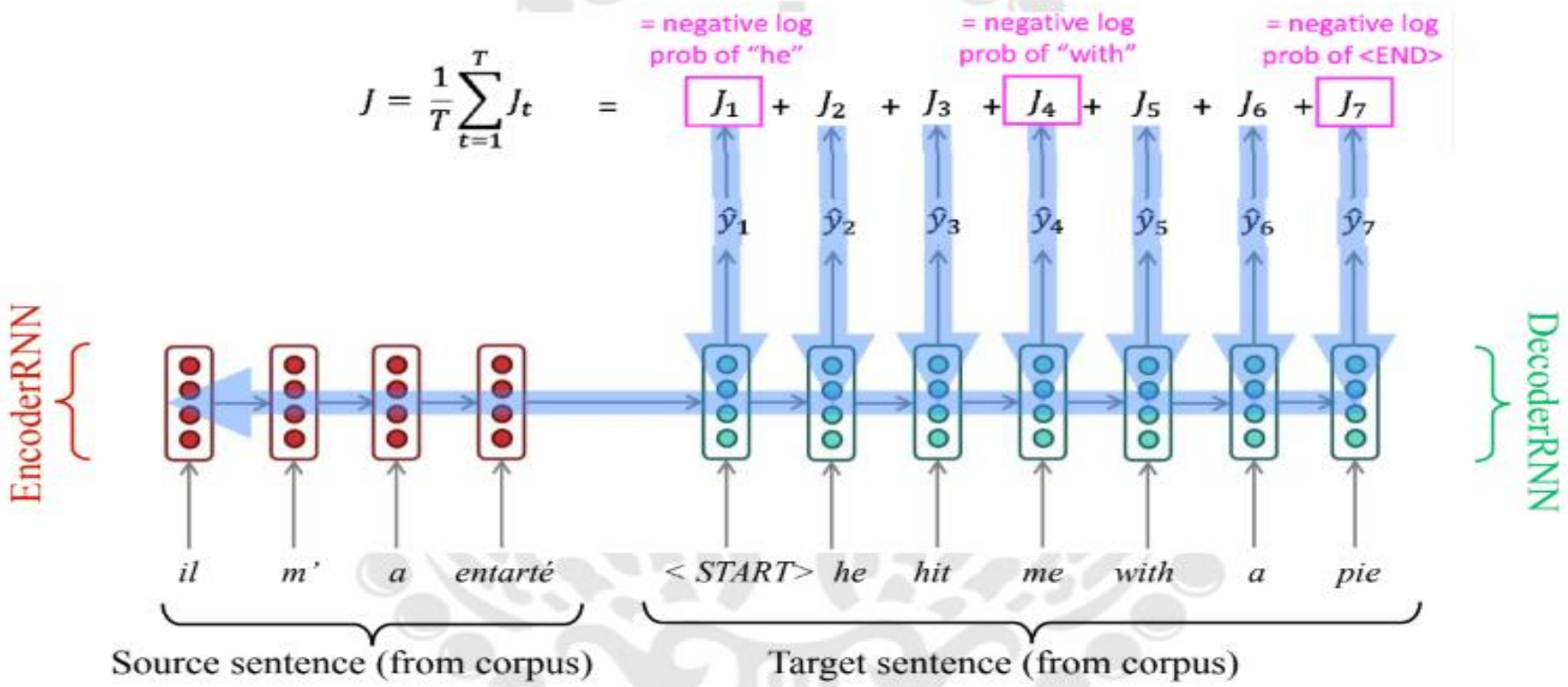


Training (Previous Step Output)

In a sequence-to-sequence architecture, the two RNNs are trained jointly to maximize the average of log P($y^{(1)}, \ldots, y^{(n_y)}$ | $x^{(1)}, \ldots, x^{(n_x)}$) over all the pairs of x and y sequences in the training set .

The last state h_{n_x} of the encoder RNN is typically used as a representation C of the input sequence that is provided as input to the decoder RNN.

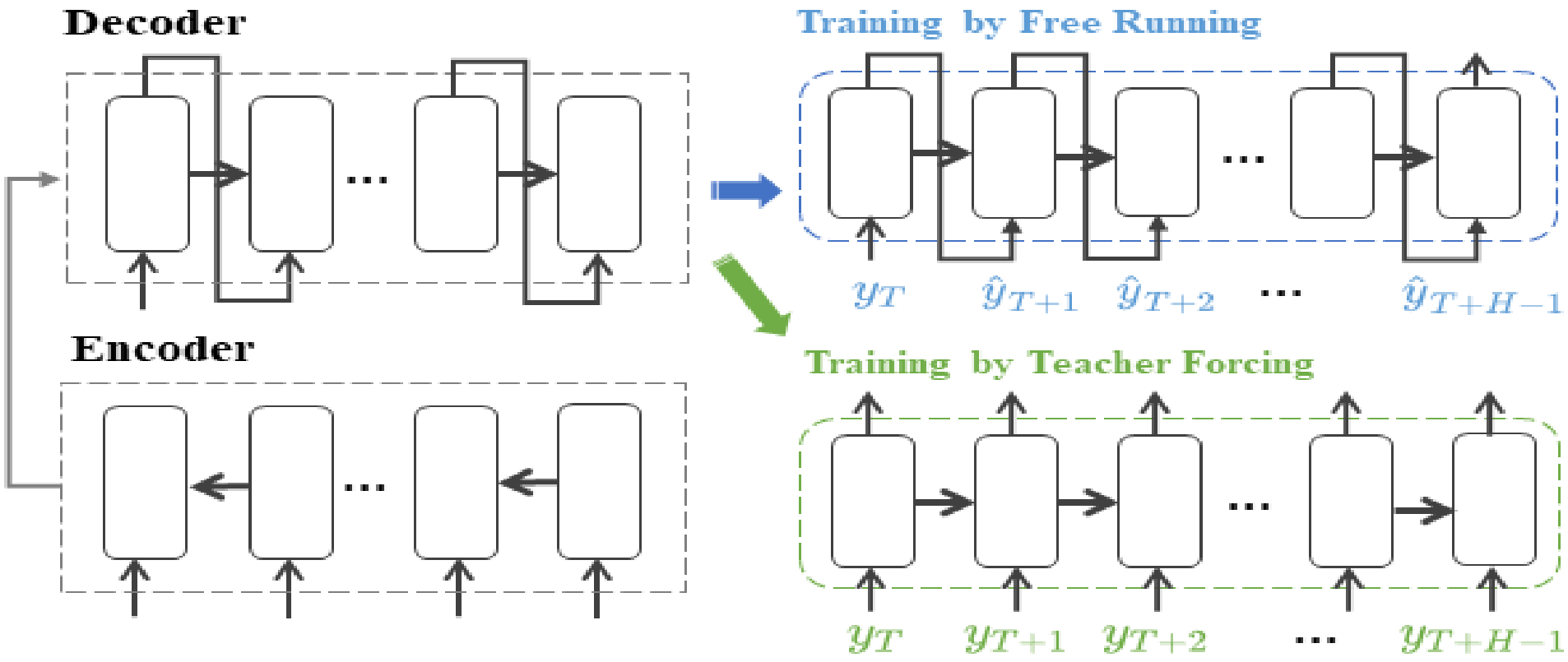
There is no constraint that the encoder must have the same size of hidden layer as the decoder.


= negative log prob of "he"
= negative log prob of "with"
= negative log prob of <END>
$J = \frac{1}{T}\sum_{t=1}^{T} J_t = J_1 + J_2 + J_3 + J_4 + J_5 + J_6 + J_7$
$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$ $\hat{y}_5$ $\hat{y}_6$ $\hat{y}_7$
EncoderRNN
DecoderRNN
il m' a entarté
< START> he hit me with a pie
Source sentence (from corpus)
Target sentence (from corpus)

Training (Teacher Forcing)

- **Teacher Forcing is a training technique used in sequence-to-sequence models** (Seq2Seq, like machine translation) and especially in Encoder-Decoder architectures. This technique is designed to help the model **learn faster and more stably.**
- In Teacher Forcing, at each prediction step, instead of using the model's predicted output from the previous step as the input for the next step, **the actual word from the training data (Ground Truth) is used as input for the next step.**

- Step 1: Initial Prediction in the Decoder In the first step of the Decoder: (the **final hidden state of the Encoder) is used as the initial state in the Decoder**. The **first input to the Decoder is usually a special token like <START>**.**Using this input, the Decoder predicts the first word.**
- Step 2: **Using Ground Truth Instead of the Model's Output Instead of using (the model's prediction from the previous step) as the input for the next step, the actual word (Ground Truth) from the training data is fed to the model.**
- This process continues until all the words in the target sequence are predicted.


Decoder
Encoder
Training by Free Running
y_T
$\hat{y}_{T+1}$
$\hat{y}_{T+2}$
$\hat{y}_{T+H-1}$
Training by Teacher Forcing
y_T
y_{T+1}
y_{T+2}
y_{T+H-1}

- **Advantages:**

1. **Faster Learning**:
    1. Since the model uses the correct data in each step, it learns faster.
2. **Prevention of Error Accumulation**:
    1. By using Ground Truth, errors from previous steps do not propagate to later steps.
3. **Stable Training**:
    1. The model remains stable during the early training stages when predictions are usually poor.

- **Disadvantages:**

1. **Exposure Bias**:
    1. During inference (real usage), the model cannot access the Ground Truth and must rely on its own predictions. The difference between the training inputs (Ground Truth) and the inference inputs (model predictions) can cause a performance gap.
2. **Lack of Error Tolerance**:
    1. The model does not learn how to handle its own mistakes because it is always fed the correct input during training.

The Challenge of Long-Term Dependencies

The basic problem is that **gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies** arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones. Many other sources provide a deeper treatment.

The vanishing and exploding gradient problem for RNNs was independently discovered by separate researchers.

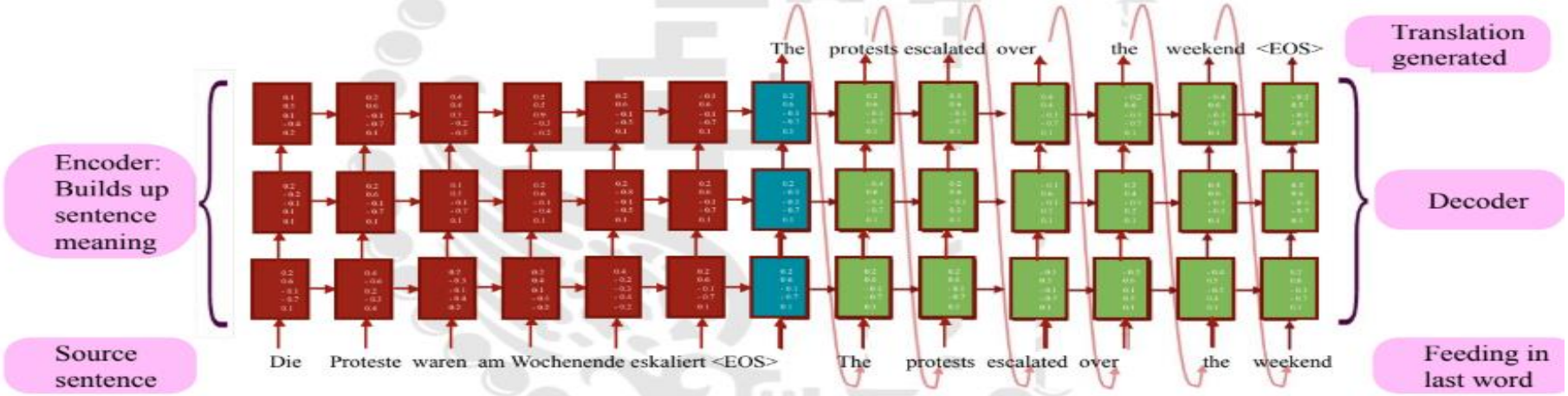
One may hope that the problem can be avoided simply by staying in a region of parameter space where the gradients do not vanish or explode. Unfortunately, in order to store memories in a way that is robust to small perturbations, the RNN must enter a region of parameter space where gradients vanish. **Specifically, whenever the model is able to represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction.**

This means not that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. **In practice, the experiments show that as we increase the span of the dependencies that need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful training of a traditional RNN via SGD rapidly reaching 0 for sequences of only length 10 or 20.**

Multi -layer deep encoder-decoder machine translation net

[Sutskever et al. 2014; Luong et al. 2015]

The hidden states from RNN layer i are the inputs to RNN layer $i+1$



Conditioning = Bottleneck

Attention Mechanisms and Transformers

The **core idea behind the Transformer model is the attention mechanism**, an innovation that was originally envisioned as an enhancement for encoder–decoder RNNs applied to sequence-to-sequence applications, such as machine translations (**Bahdanau** et al., 2014). You might recall that in the **first sequence-to-sequence models for machine translation (Sutskever et al., 2014), the entire input was compressed by the encoder into a single fixed length vector to be fed into the decoder. The intuition behind attention is that rather than compressing the input, it might be better for the decoder to revisit the input sequence at every step.** Moreover, rather than always seeing the same representation of the input, one might imagine that the decoder should selectively focus on particular parts of the input sequence at particular decoding steps.

Bahdanau's attention mechanism provided a simple means by which **the decoder could dynamically attend to different parts of the input at each decoding step**. The high-level idea is that the encoder could produce a representation of length equal to the original input sequence. Then, at decoding time, the decoder can (via some control mechanism) receive as input a context vector consisting of a weighted sum of the representations on the input at each time step. Intuitively, the weights determine the extent to which each step's context "focuses" on each input token, and the key is to make this process for assigning the weights differentiable so that it can be learned along with all of the other neural network parameters.

Attention adding seq2seq: RNN based

Let's continue looking at attention models at this high level of abstraction. An attention model differs from a classic sequence-to-sequence model in two main ways:

- First, **the encoder passes a lot more data to the decoder. Instead of passing the last hidden state of the encoding stage, the encoder passes all the hidden states to the decoder.**
- Second, an attention decoder does an extra step before producing its output. In order to focus on the parts of the input that are relevant to this decoding time step, **the decoder does the following**:

1. Look at the set of encoder hidden states it received – **each encoder hidden state is most associated with a certain word in the input sentence**

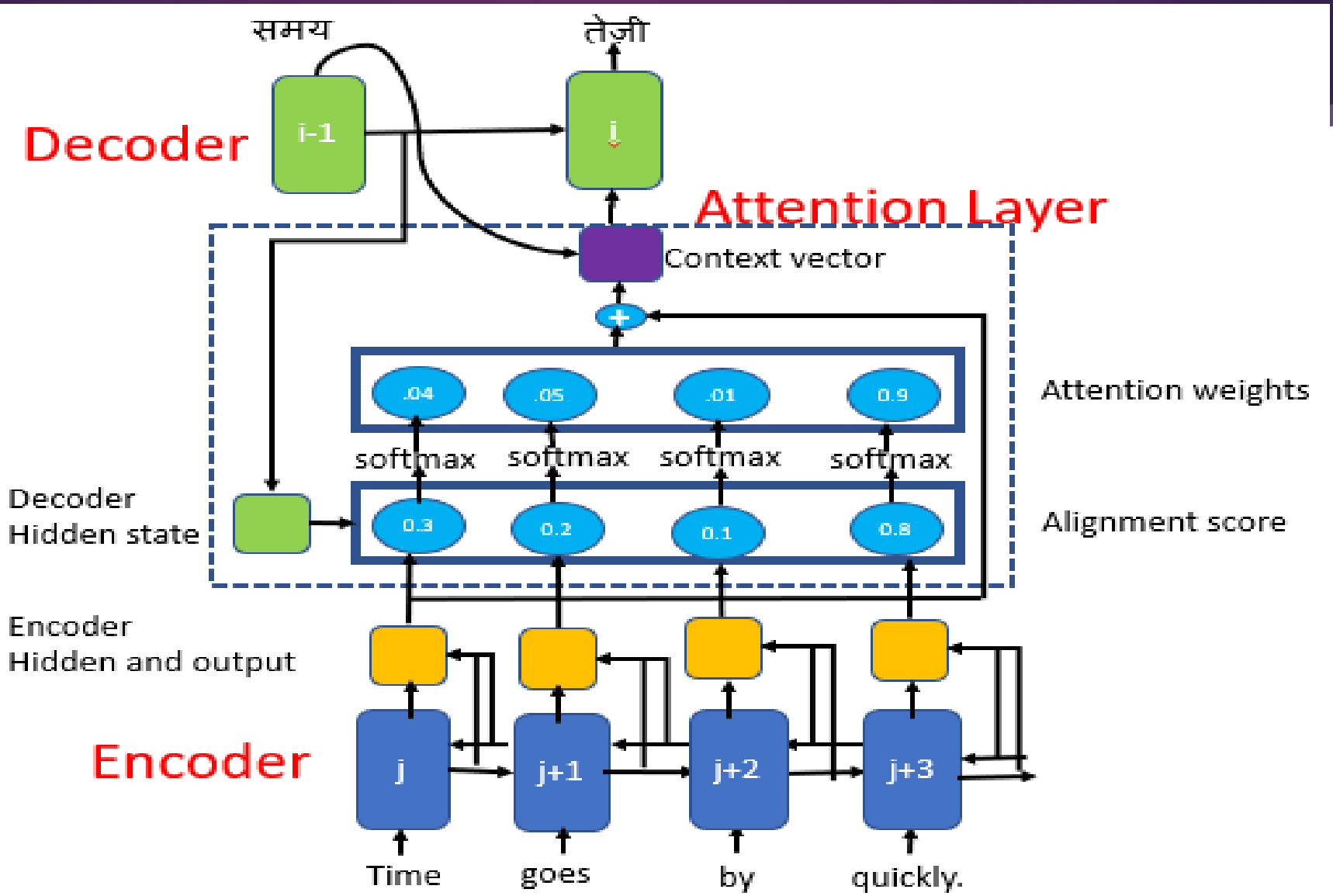
2. **Give each hidden state a score** (let's ignore how the scoring is done for now)

3. **Multiply each hidden state by its soft maxed score**, thus **amplifying hidden states with high scores, and drowning out hidden states with low scores**

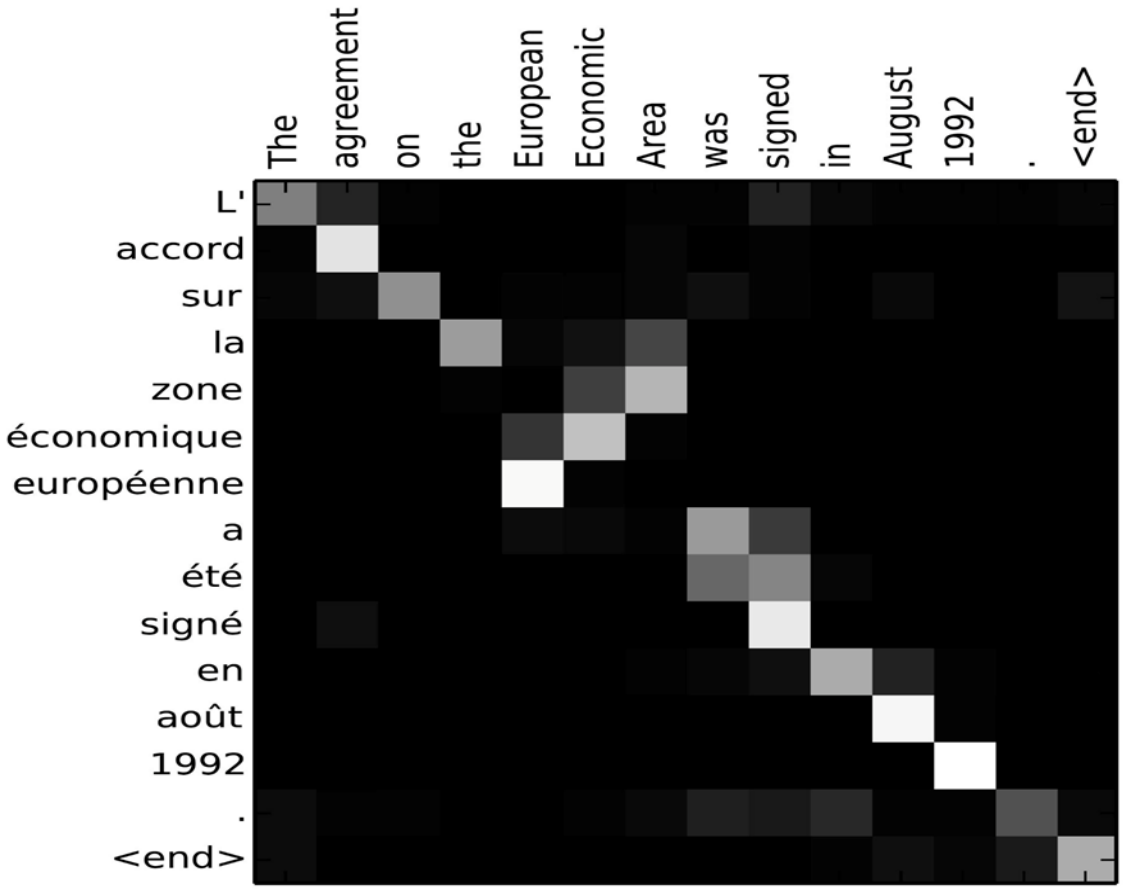
This scoring exercise is done at each time step on the decoder side.

attention process works:

1. The attention decoder RNN takes in the embedding of the <Start> token, and an initial decoder hidden state.
2. The RNN processes its inputs, producing an output and a new hidden state vector (h4). The output is discarded.
3. Attention Step: We use the encoder hidden states and the h4 vector to calculate a context vector (C4) for this time step.
4. We concatenate h4 and C4 into one vector.
5. We pass this vector through a feedforward neural network (one trained jointly with the model).
6. The output of the feedforward neural networks indicates the output word of this time step.
7. Repeat for the next time steps


समय
तेज़ी
Decoder
i-1
i
Attention Layer
Context vector
.04
.05
.01
0.9
Attention weights
softmax
softmax
softmax
softmax
Decoder
Hidden state
0.3
0.2
0.1
0.8
Alignment score
Encoder
Hidden and output
Encoder
j
j+1
j+2
j+3
Time
goes
by
quickly.

You can see how the model paid attention correctly when outputting "European Economic **Area**". In French, the order of these words is reversed ("**zone** économique européenne ") as compared to English.



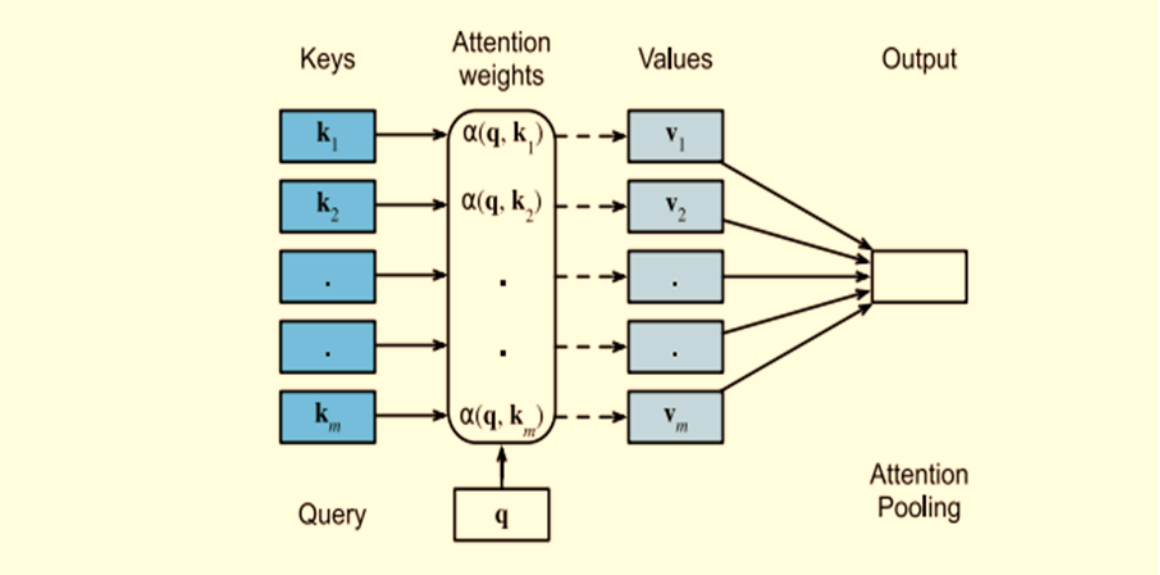
Initially, **the idea was a remarkably successful enhancement to the recurrent neural networks that already dominated machine translation applications. The models performed better than the original encoder–decoder sequence-to-sequence architectures.** Furthermore, researchers noted that some nice qualitative insights sometimes emerged from inspecting the pattern of attention weights. In translation tasks, attention models often assigned high attention weights to cross-lingual synonyms when generating the corresponding words in the target language. For example, when **translating the sentence "my feet hurt" to "j'ai mal au pieds",** the **neural network might assign high attention weights to the representation of "feet" when generating the corresponding French word "pieds".** These insights spurred claims that attention models confer **"interpretability"** although what precisely the attention weights mean—i.e., how, if at all, they should be interpreted remains a hazy research topic.

However, attention mechanisms soon emerged as more significant concerns, beyond their usefulness as an enhancement for encoder–decoder recurrent neural networks and their putative usefulness for picking out salient inputs. **Vaswani et al. (2017) proposed the Transformer architecture for machine translation, dispensing with recurrent connections altogether, and instead relying on cleverly arranged attention mechanisms to capture all relationships among input and output tokens. The architecture performed remarkably well, and by 2018 the Transformer began showing up in the majority of state-of-the-art natural language processing systems.** Moreover, at the same time, the dominant practice in natural language processing became to pretrain large-scale models on enormous generic background corpora to optimize some self-supervised pretraining objective, and then to fine-tune these models using the available downstream data. The gap between Transformers and traditional architectures grew especially wide when applied in this pretraining paradigm, and thus the ascendance of Transformers coincided with the ascendence of such large-scale pretrained models, now sometimes called foundation models (Bommasani et al., 2021).

Queries, Keys, and Values

For now, simply consider the following: denote by D def = $\{(k_1, v_1), \ldots (k_m, v_m)\}$ a database of m tuples of keys and values. Moreover, denote by q a query. Then we can define the attention over D

$$\mathrm{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\mathrm{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i,$$



where $\alpha(q, k_i) \in R$ $(i = 1, \ldots, m)$ are **scalar attention weights**. The operation itself is typically referred to as **attention pooling**. The name attention derives from the fact that the operation pays particular attention to the terms for which the weight α is significant. As such, the attention over D generates a **linear combination of values contained in the database**.

- The **weights α(q, k_i) are nonnegative.** In this case the output of the attention mechanism is contained in the convex cone spanned by the values v_i.
- The weights α(q, k_i) form a convex combination,

$$\sum \alpha(q, k_i) = 1 \text{ and } \alpha(q, k_i) \geq 0$$

for all i. This is the **most common setting in deep learning**.

- Exactly one of the weights α(q, k_i) is 1, while all others are 0. This is asking to a traditional database query.
- All weights are equal, i.e., α(q, k_i) =1/m for all i. This amounts to averaging across the entire database, also called **average pooling** in deep learning.

Query (Q): **A vector that represents the element you want to compute attention for.**
It acts as a "question" asking, "What should I focus on?"

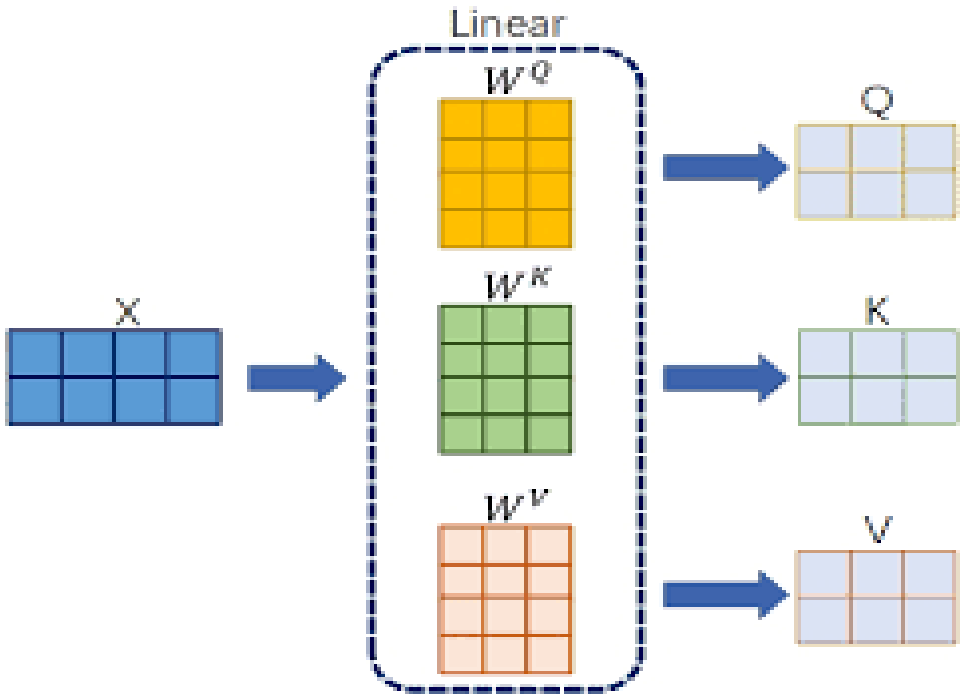
- **Key (K)**: **A vector that represents the elements you are comparing against.**
Each key serves as metadata about the corresponding value.
- **Value (V)**: **A vector that contains the actual information or representation you want to retrieve or weigh based on the attention scores.**
- Each token in a sequence has its **Query**, **Key**, and **Value** computed as **linear transformations** of the token's embedding:

$$Q = XW_Q \ , \quad K = XW_K \ , \quad V = XW_V$$

where:

- **X is the input embedding or the output from the previous layer.**
- W_Q, W_K, W_V are learned weight matrices.

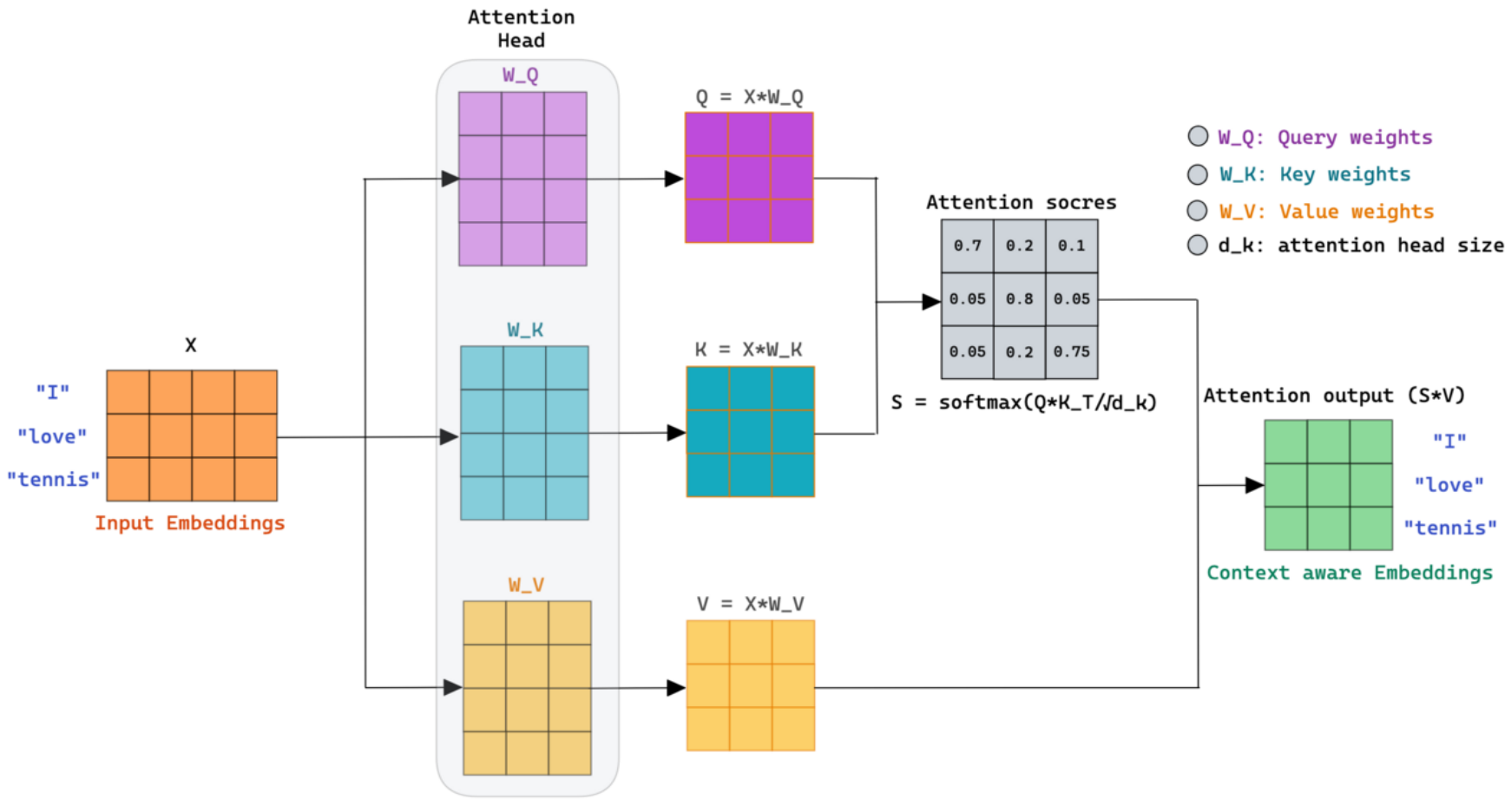
For a sentence like **"The cat sat on the mat,"** the attention mechanism: Uses queries to ask, "What words should 'cat' focus on?“ Compares the **query for "cat"** to **the keys for all tokens in the sentence**. **Weighs the values (word representations)** accordingly, allowing "cat" **to focus more on related words like "sat" and "mat."**


Linear
W^Q
W^K
W^V
X
Q
K
V

Scaled Dot Product Attention

In general, it requires that both the query and the key have the **same vector length**, say d, even though this can be addressed easily by replacing q^Tk with q^TMk where M is a matrix suitably chosen for translating between both spaces. For now assume that the **dimensions match**. In practice, we often think of minibatches for efficiency, such as computing attention for n queries and m key-value pairs, where queries and keys are of length d and values are of length v. The scaled dot product attention of queries Q $\in R^{n.d}$, keys K $\in R^{m.d}$, and values V $\in R^{m.v}$ thus can be written as:

$$\mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n\times v}.$$



Additive Attention

When queries q and keys k are vectors of different dimension, we can either use a matrix to address the mismatch via q^TMk , or we can use **additive attention** as the scoring function.

Another benefit is that, as its name indicates, the attention is additive. This can lead to some minor computational savings. Given a **query q ∈ R^q and a key k ∈ R^k**, the additive attention scoring function (Bahdanau et al., 2014) is given by

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R},$$

where **$Wq \in R^{d.q}$, $W_k \in R^{d.k}$, and $W_v \in R^d$ are the learnable parameters**. This term is then **fed into a softmax to ensure both nonnegativity and normalization.**

Recap

we introduced the two key attention scoring functions: dot product and additive attention. They are effective tools for aggregating across sequences of variable length.

In particular, **the dot product attention is the mainstay of modern Transformer architectures.**

When queries and keys are vectors of different lengths, we can use the additive attention scoring function instead. Optimizing these layers is one of the key areas of advance in recent years. For instance, NVIDIA's Transformer Library157 and Megatron (Shoeybi et al.,2019) crucially rely on efficient variants of the attention mechanism.

The Bahdanau Attention Mechanism for RNN-based model

Conventionally, in an RNN all relevant information about a source sequence is translated into some internal fixed-dimensional state representation by the encoder. It is this very state that is used by the decoder as the complete and exclusive source of information for generating the translated sequence. In other words, the sequence-to-sequence mechanism treats the intermediate state as a sufficient statistic of whatever string might have served as input.

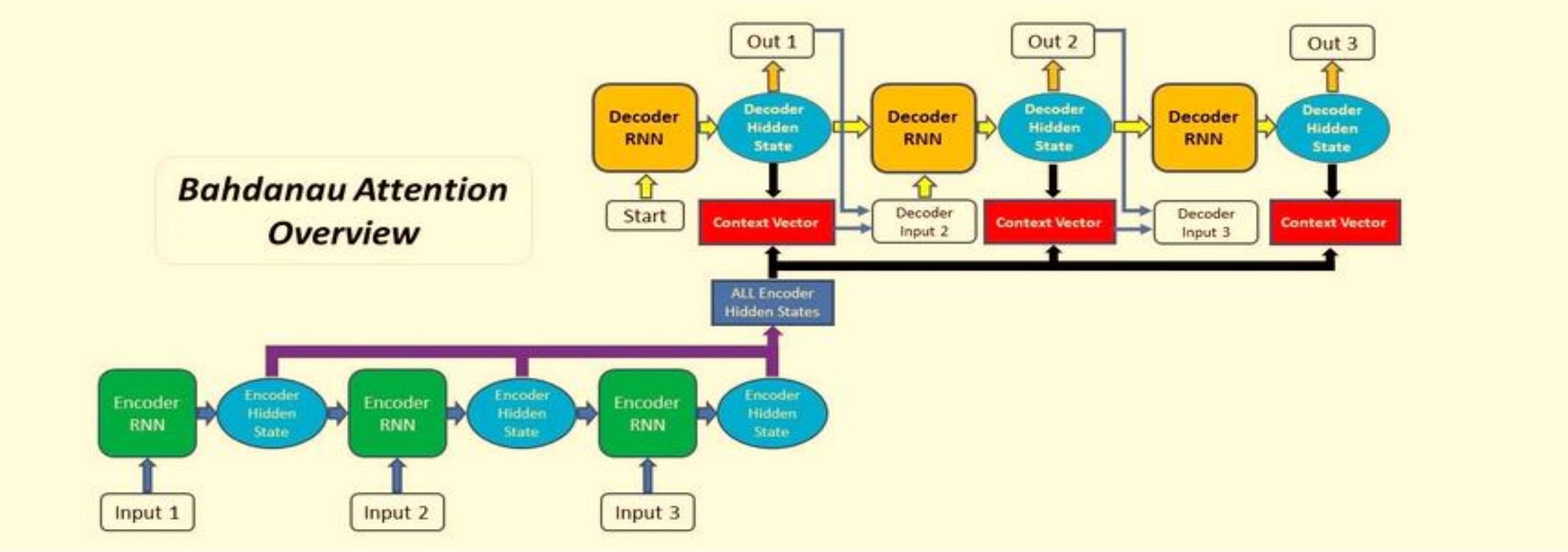
While this is quite reasonable for short sequences, it is clear that it is infeasible for long ones, such as a book chapter or even just a very long sentence. After all, before too long there will simply not be enough “space” in the intermediate representation to store all that is important in the source sequence. Consequently the decoder will fail to translate long and complex sentences.

We need to implement the RNN decoder in the Seq2SeqAttentionDecoder class.

The state of the decoder is initialized with **(i) the hidden states of the last layer of the encoder at all time steps, used as keys and values for attention**; **(ii) the hidden state of the encoder at all layers at the final time step, which serves to initialize the hidden state of the decoder; and (iii) the valid length of the encoder, to exclude the padding tokens in attention pooling. At each decoding time step, the hidden state of the final layer of the decoder, obtained at the previous time step, is used as the query of the attention mechanism.**

Both the output of the attention mechanism and the input embedding are concatenated to serve as the input of the RNN decoder.

When predicting a token, if not all the input tokens are relevant, the RNN encoder–decoder with the Bahdanau attention mechanism selectively aggregates different parts of the input sequence. This is achieved by treating the state (context variable) as an output of additive attention pooling. **In the RNN encoder–decoder, the Bahdanau attention mechanism treats the decoder hidden state at the previous time step as the query, and the encoder hidden states at all the time steps as both the keys and values.**


Bahdanau Attention Overview
Out 1
Out 2
Out 3
Decoder RNN
Decoder Hidden State
Decoder RNN
Decoder Hidden State
Decoder RNN
Decoder Hidden State
Start
Context Vector
Decoder Input 2
Context Vector
Decoder Input 3
Context Vector
ALL Encoder Hidden States
Encoder RNN
Encoder Hidden State
Encoder RNN
Encoder Hidden State
Encoder RNN
Encoder Hidden State
Input 1
Input 2
Input 3

Multi-Head Attention

In practice, given the same set of queries, keys, and values we may want our model to combine knowledge from **different behaviors** of the same attention mechanism, such as **capturing dependencies of various ranges (e.g., shorter-range vs. longer-range) within a sequence.** Thus, **it may be beneficial to allow our attention mechanism to jointly use different representation subspaces of queries, keys, and values.**

To this end, instead of performing a single attention pooling, **queries, keys, and values can be transformed with h independently learned linear projections.** Then these h projected queries, keys, and values are fed into attention pooling in **parallel**. In the end, h attention pooling outputs are **concatenated** and transformed with another learned **linear projection to produce the final output**. This design is called multi-head attention, where each of the h attention pooling outputs is a head (Vaswani et al., 2017).

- explains the formula of **Multi-Head Attention**. For **each head h_i**, the input values are:

  - $W_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$: Learnable matrix for transforming **queries**.
  - $W_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$: Learnable matrix for transforming **keys**.
  - $W_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$: Learnable matrix for transforming **values**.

  Here:

  - d_q, d_k, and d_v are the lengths of the input vectors (queries, keys, and values).
  - p_q, p_k, and p_v are the dimensions of the transformed vectors after applying the learnable matrices.

- How does Multi-Head Attention work?

Input Transformation:

- For each head h_i, the queries, keys, and values are transformed using $W_i^{(q)} \in R^{p_q d_q}$, $W_i^{(k)} \in R^{p_k d_k}$, and $W_i^{(v)} \in R^{p_v d_v}$
- Then, an attention mechanism like **Additive Attention** or **Scaled Dot Product Attention** is used to compute weights and combine the inputs.

Combining Heads:

- The outputs from all heads (h heads) are **concatenated**.
- **This concatenated output is then passed through a linear transformation with another learnable parameter matrix (W_0) to produce the final output.**

- **Overall Equation for the Final Output:**

Output= W_0.Concat($Head_1$, $Head_2$,..., $Head_h$)

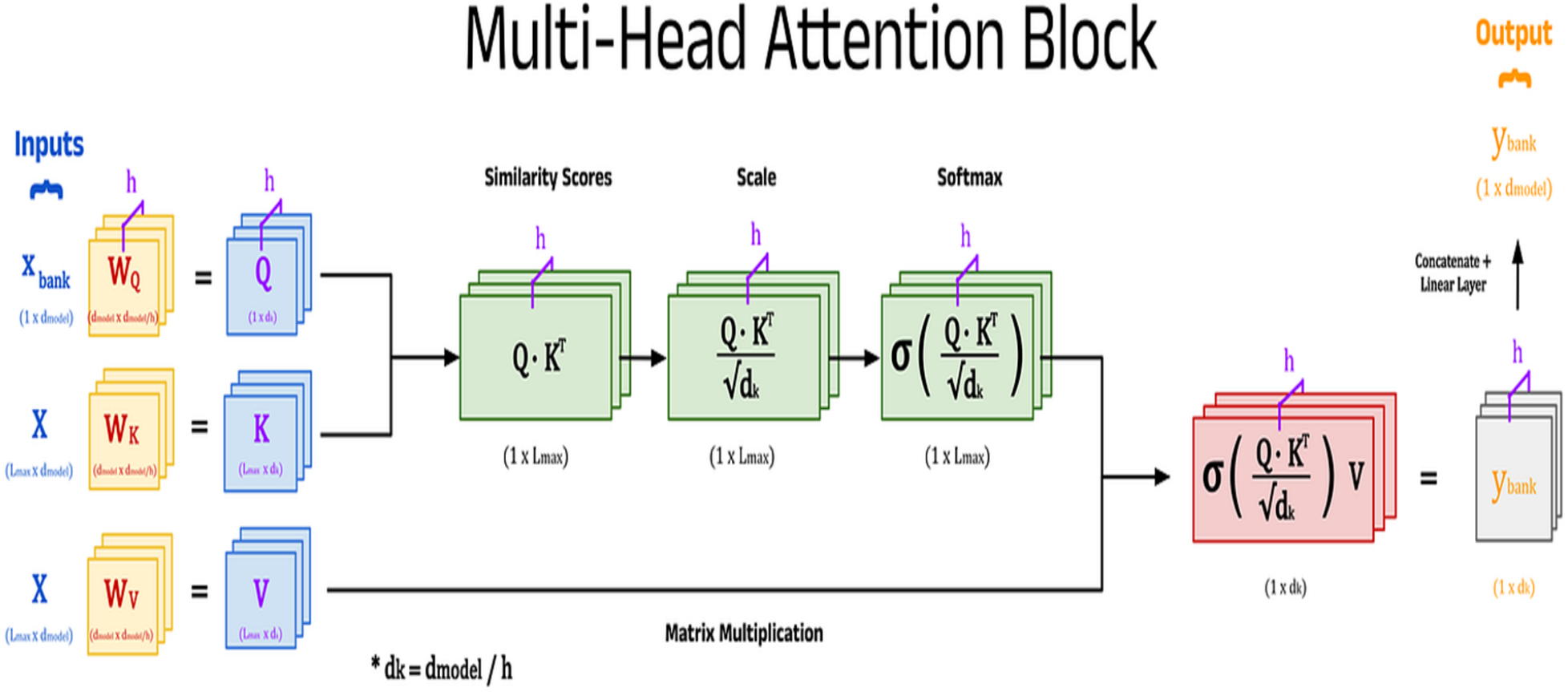
Where:

W_0 **:** Final learnable matrix for transforming the concatenated output.

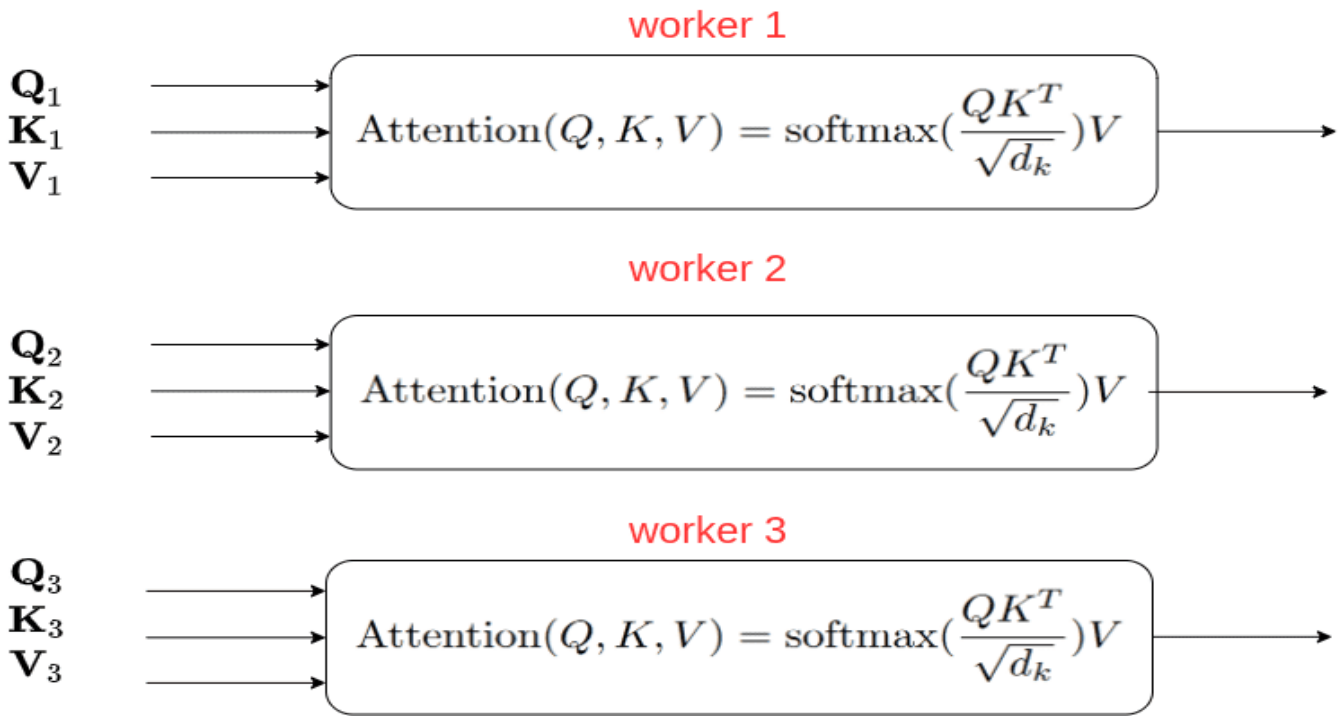
$Head_i$ **:** Output of the i-th head using the attention mechanism f.

$\mathbf{Head_i = f(Q\ W_i^{(q)}, K\ W_i^{(k)}, V\ W_i^{(v)})}$

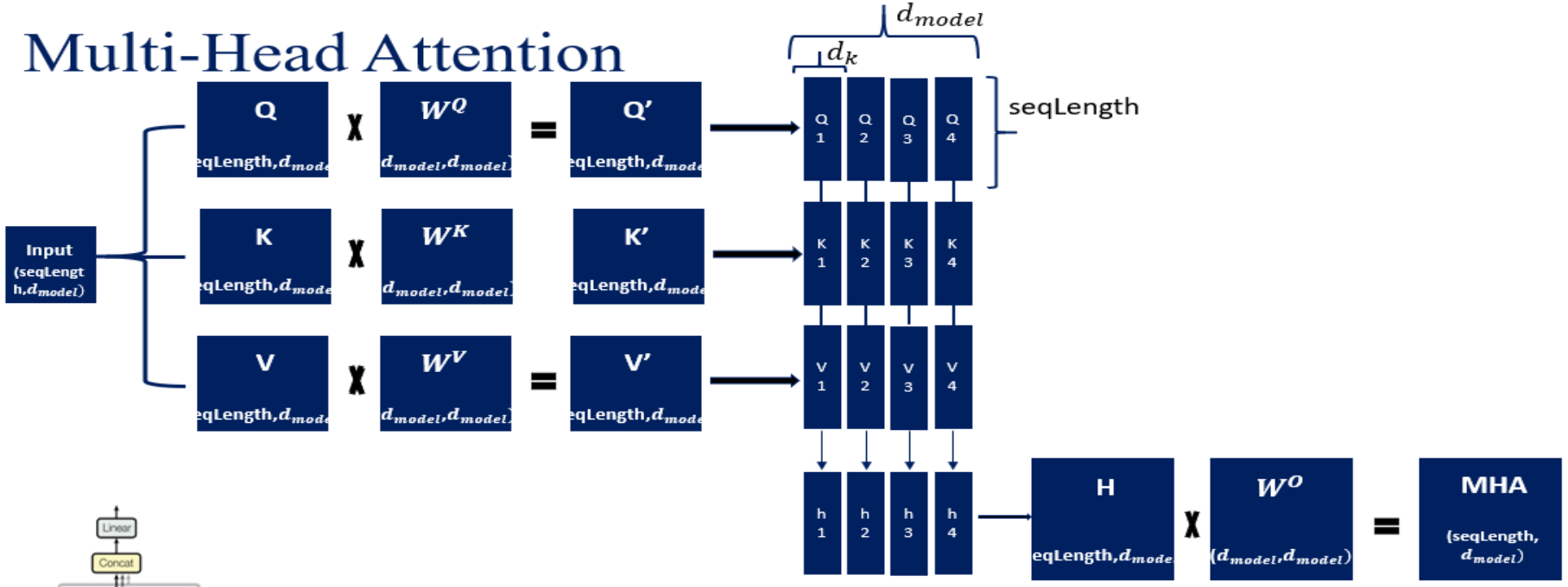
Multi-Head Attention Block

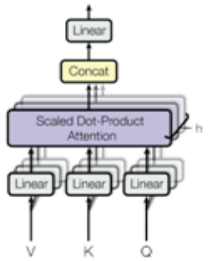


Each attention head can be implemented in parallel



Multi-Head Attention





Self-Attention and Positional Encoding

Now with attention mechanisms in mind, **imagine feeding a sequence of tokens into an attention mechanism such that at every step, each token has <u>its own query, keys, and values.</u>** Here, when computing the value of a token's representation at the next layer, the token can attend (via its query vector) to any other's token (matching based on their key vectors). Using the full set of query-key compatibility scores, we can compute, for each token, a representation by building the appropriate weighted sum over the other tokens. Because every token is attending to each other token (unlike the case where decoder steps attend to encoder steps), such architectures are typically described as self-attention models (Lin et al., 2017, Vaswani et al., 2017), and elsewhere described as intra-attention model (Cheng et al., 2016, Parikhet al., 2016, Paulus et al., 2017). we will discuss sequence encoding using self-attention, including using additional information for the sequence order.

- **Initial Assumptions:**

1. **Input:**
   1. Assume we have n tokens represented by their embedding vectors:

   $X=[x_1, x_2,..., x_n] \in R^{n.d}$

      1. n: Number of tokens (sequence length).
      2. d: Dimensionality of each embedding vector.

2. **Learnable Parameters:**
   1. Three learnable linear projection matrices to produce **Queries (Q), Keys (K), and Values (V):**

   $W^Q \in R^{d.d_k}$, $W^K \in R^{d.d_k}$, $W^v \in R^{d.d_v}$

- **1. Compute Queries, Keys, and Values:**
- **Each token embedding x_i** is transformed into three vectors: query (q_i), key (k_i), and value (v_i):
- Q= XW^Q ,K= XW^K ,V= XW^v
- **Each row of Q, K, and V corresponds to one token.**
- **2. Compute Compatibility Scores:**
- The compatibility score between token i and token j is computed using the dot product of their query and key vectors: Score(i, j)= $q_i \cdot k_i^T$
- **For the entire sequence, this can be computed as a matrix: S=$QK^T \in R^{n.n}$**
- **$S_{i,j}$: Indicates how much token i attends to token j.**

- **3. Normalize Scores with Softmax:**

- The scores $S_{i,j}$ are normalized to form a probability distribution using the **Softmax function** : **A=Softmax(S)∈ $R^{n.n}$**

- **4. Weighted Sum of Values:**

- **The new output representation for token i is computed as the weighted sum of the value vectors (V): $o_i = \sum_{j=1}^{n} A_{i,j}\, v_j$**

- **In matrix form: O = AV ∈ $\mathbf{R}^{n.d_v}$**

Step-by-step explanation of Self-Attention

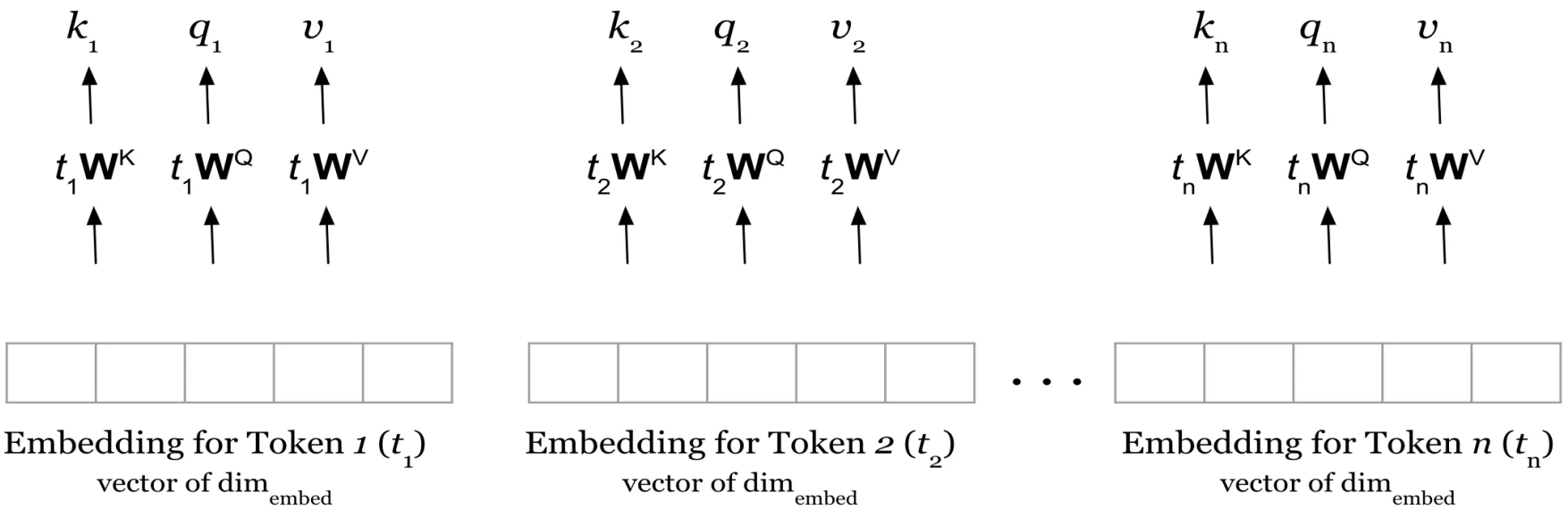
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

to compute Attention for all input tokens at once use Matrix operations (same as)
where, $Q = XW^Q$, $K = XW^K$, and $V = XW^V$

$$\boxed{Attention_{token_n} = \sum w_i v_i}$$

$$w_1 = k_1 \bullet q_n \qquad w_2 = k_2 \bullet q_n \qquad w_n = k_n \bullet q_n$$

© AIML.com Research



④ Attention for token n is computed as a weighted sum of values of all the input tokens

③ dot product of key vector (k_i) of every input token with the query vector of nth token (q_n) is used to dynamically compute weight (w_i) of each input token

② For each head, we initialize three matrices of *dim(embed, head_size)* called Key ($\mathbf{W}^K$), Query ($\mathbf{W}^Q$), and Value ($\mathbf{W}^V$)

Embedding for each of the input token is multiplied by $\mathbf{W}^K$, $\mathbf{W}^Q$, and $\mathbf{W}^V$ matrices to generate k, q, and v vectors

① Input is represented as matrix $\mathbf{X}$ of *dim(n, embed)*, where n is the num of input tokens

Comparing CNNs, RNNs, and Self-Attention

we will consider CNNs, RNNs, and self-attention. We will compare their **computational complexity, sequential operations**. Note that **sequential operations prevent parallel computation.**

Computational Complexity:

This criterion indicates the computational resources required to process a sequence:

Architecture	Computational Complexity for n tokens with d-dimensional vectors
RNN	$O(n \cdot d^2)$
CNN	$O(k \cdot n \cdot d^2)$, k is the filter size
Self-Attention	$O(n^2 \cdot d)$

- **RNN**: Processes each token sequentially, resulting in linear time with respect to the sequence length.
- **CNN**: Processes a range of tokens using local filters, with the filter size k affecting the complexity.
- **Self-Attention**: Computes attention for every pair of tokens, causing quadratic complexity in n, but allows for greater parallelization.

Sequential Operations:

Sequential operations prevent parallel computation.

Architecture	Number of Sequential Operations
RNN	$O(n)$
CNN	$O(\log(n))$ (layers cover the entire sequence)
Self-Attention	$O(1)$ (fully parallelizable)

Positional Encoding

Unlike **RNNs, which recurrently process tokens of a sequence one-by-one**, **self-attention** ditches **sequential operations** in favor of **parallel computation**. Note that **self-attention by itself does not preserve the order of the sequence**. What do we do if it really matters that the model knows in which order the input sequence arrived?

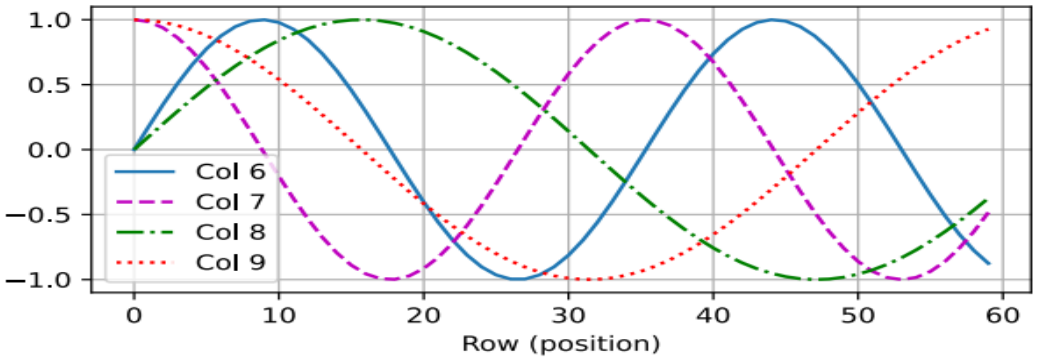
The dominant approach for preserving information about the order of tokens is to represent this to the model as an **additional input** associated with **each token.** These inputs are called **positional encodings**, and they can either be learned or fixed a priori. We now describe a simple scheme for fixed positional encodings based on **sine and cosine functions** (Vaswaniet al., 2017).

Suppose that the **input representation X $\in R^{n.d}$ contains the d-dimensional embeddings for n tokens of a sequence**. The positional encoding outputs X + P using a **positional embedding matrix P $\in R^{n.d}$** of the same shape, whose element on the ith row and the (2 j)th or the (2 j + 1)th column is

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right),$$

$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right).$$

In the **positional embedding matrix P**, **rows correspond to positions within a sequence** and **columns represent different positional encoding dimensions**. In the example below, we can see that the 6th and the 7th columns of the positional embedding matrix have a higher frequency than the 8th and the 9th columns. The offset between the 6th and the 7th (same for the 8th and the 9th) columns is due to the alternation of sine and cosine functions.



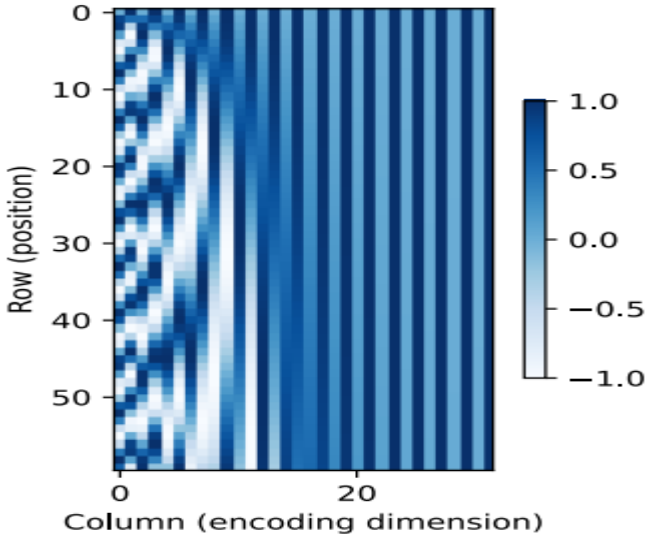
- For instance:

To see how the monotonically decreased frequency along the encoding dimension relates to absolute positional information, let's print out the binary representations of 0, 1, . . . , 7.

As we can see, the lowest bit, the second-lowest bit, and the third-lowest bit alternate on every number, every two numbers, and every four numbers, respectively.

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

In binary representations, a higher bit has a lower frequency than a lower bit. Similarly, as demonstrated in the heat map below, the positional encoding decreases frequencies along the encoding dimension by using trigonometric functions. Since the outputs are float numbers, such continuous representations are more space-efficient than binary representations.



Recap

In self-attention, the queries, keys, and values all come from the same place. Both CNNs and self-attention enjoy parallel computation and self-attention has the shortest maximum path length. However, the quadratic computational complexity with respect to the sequence length makes self-attention prohibitively slow for very long sequences. To use the sequence

order information, we can inject absolute or relative positional information by adding positional encoding to the input representations.

Residual Connections

- Residual connections are a **key concept in the design of deep neural networks**, addressing challenges that arise when increasing the number of layers in a model. This concept was first introduced by **Microsoft in a paper about ResNet (Residual Network)**, revolutionizing the field of deep learning.

The Problem with Deep Networks and the Need for Residual Connections

- **The Problem with Deep Networks and the Need for Residual Connections**

1. **Degradation in Performance of Deeper Networks:**

   It is expected that increasing the depth of a network should improve its accuracy. However, in practice, when the depth becomes very large, the phenomenon of **performance degradation** occurs. This means that even with more layers, the model's accuracy decreases. The root cause lies in difficulties with model convergence and effective learning.

2. **Difficulty in Learning Complex Mappings:**

   Each layer in a neural network attempts to learn a mapping H(x)H(x)H(x) that transforms input into the desired output. In very deep networks, learning these complex mappings becomes challenging.

The Core Idea of Residual Connections

- Residual connections mitigate these issues by introducing a simple shortcut. Instead of having the network directly learn H(x), it is broken down into two components: $H(x)=F(x)+x$

  F(x): The residual function, which is learned by deeper layers.

  x: The shortcut or initial input, which is added directly to the output.

- With this design:

- If the layers fail to learn F(x) (e.g., due to vanishing gradients), the residual connection ensures that the input x is passed to the next layer without loss.
- This guarantees that the network at least retains the identity mapping (input = output).

Key Impacts of Residual Connections

Direct Gradient Flow: In traditional methods, gradients struggle to propagate effectively to the initial layers in very deep networks, leading to the vanishing gradient problem. Residual connections provide a direct path for gradient flow, ensuring proper updates even in extremely deep networks.

Simplified Learning Process: Learning the residual function F(x) is much easier than learning the full mapping H(x), as the network only needs to focus on improving the residual.

Stabilizing the Network: Residual connections prevent unnecessary drastic changes in the parameters of layers, as the input information is directly passed to the next layer.

The ResNet Architecture

- **In ResNet:**
- Residual Blocks are the core design elements. **Each block consists of**:
  - Several layers (e.g., convolutional and normalization layers).
  - A shortcut path that adds the original input to the output.
- By stacking these blocks**, ResNet enabled the training of networks with over 150 layers, whereas conventional networks struggled even with 30–40 layers.**

Applications of Residual Connections in Other Architectures

- Residual connections are not limited to ResNet and are widely used in many modern architectures:

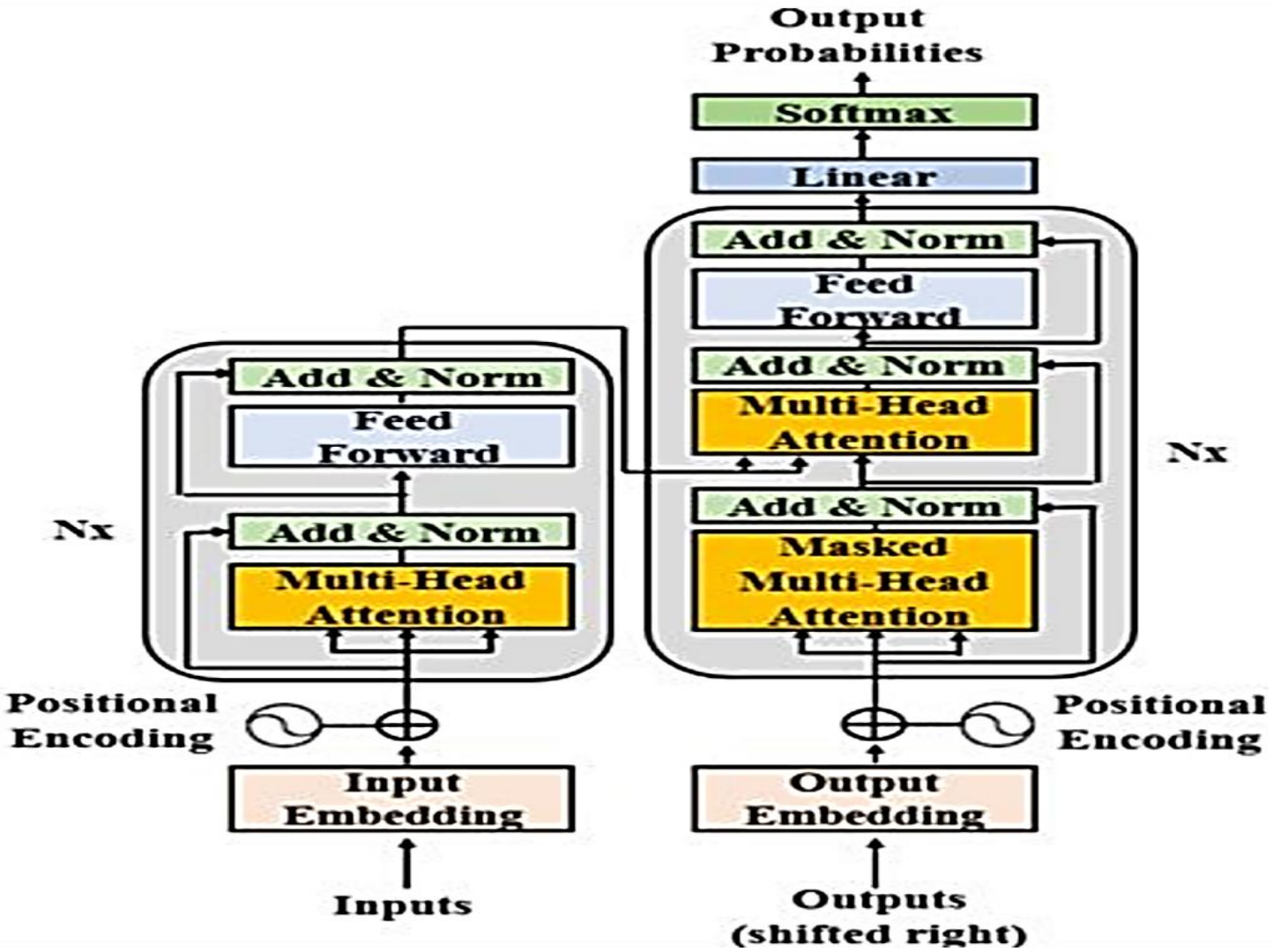
1. **Transformers:**
    1. In transformer models (e.g., BERT, GPT), residual connections are combined with attention layers to preserve the flow of information between layers.
2. **U-Net Architectures:**
    1. In image processing tasks (like image segmentation), residual connections help retain spatial features throughout the network.
3. **GANs and VAEs:**
    1. Residual connections in generative models improve stability and accuracy.

The Transformer Architecture

In December 2017, the seminal Vaswani et al. **Attention Is All You Need article, written by Google Brain members and Google Research**, was published. The **Transformer was born**. The Transformer outperformed the existing state-of-the-art NLP models. The Transformer **trained faster than previous architectures** and obtained higher evaluation results. Transformers **have become a key component of NLP.**

The digital world would never have existed without NLP. Natural Language Processing would have remained primitive and inefficient without artificial intelligence. However, the use of Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) comes at a tremendous cost in terms of calculations and machine power.

Attention Is All You Need

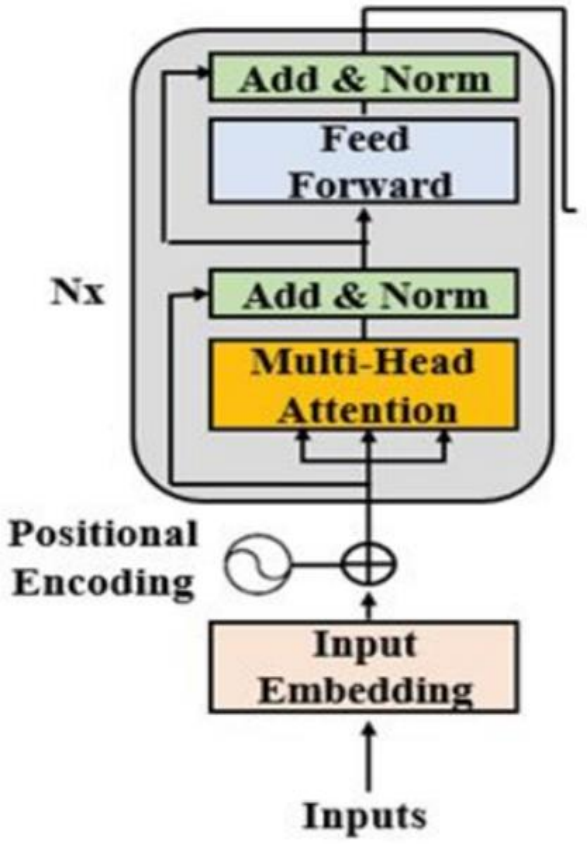


- The **original Transformer model** is a **stack of 6 layers**. The **output of layer l is the input of layer l+1 until the final prediction is reached.** There **is a 6- layer encoder stack on the left** and a **6-layer decoder stack on the right.**
- On the left, the inputs enter **the encoder** side of the Transformer through an **attention sub-layer** and **Feed-Forward Network (FFN) sub-layer**. On the right, the target outputs go into the **decoder side** of the Transformer through **two attention sub-layers** and an **FFN sub-layer.** We immediately notice that there is **no RNN, LSTM, or CNN**. Recurrence has been abandoned.

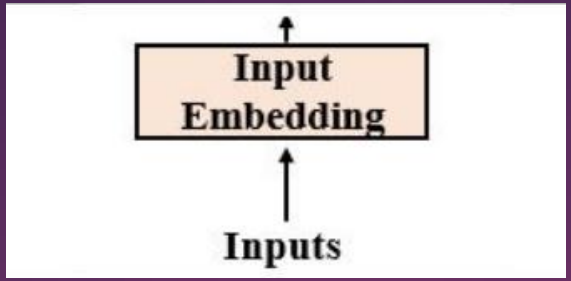
- **Attention has replaced recurrence**, which requires an increasing number of operations as the distance between two words increases. **The attention mechanism is a "word-to-word" operation.** The attention mechanism will find how **each word is related to all other words in a sequence**, including the word being analyzed itself.

The encoder stack

Each layer of the encoder stack has the following structure:



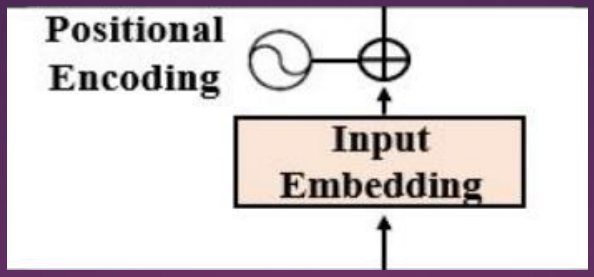
Input embedding



The input embedding sub-layer converts the **input tokens to vectors of dimension d_{model}= 512** using learned embeddings in the original Transformer model.

The embedding sub-layer works like other standard transduction models. A tokenizer will transform a sentence into tokens. Each tokenizer has its methods, but the results are similar.

Positional encoding



We enter this **positional encoding** function of the **Transformer** **with no idea of the position of a word in a sequence:**

We cannot **create independent positional vectors that would have a high cost on the training speed of the Transformer and make attention sub-layers very complex to work with.** The idea is to **add a positional encoding value to the input embedding** instead of having additional vectors to describe the position of a token in a sequence. We also know that the Transformer expects a fixed size d_{model} = 512 (or other constant value for the model) for each vector of the output of the positional encoding function.

In this sentence, we can see that **black and brown may be similar**, but **they are far apart:**

The black cat sat on the couch and the brown dog slept on the rug .

The word black is in position 2, pos=2

the word brown is in position 10, pos=10

Our problem is to find a way to add a value to the word embedding of each word so that it has that information. However, **we need to add a value to the d_{model} = 512 dimensions! For each word embedding vector, we need to find a way to provide information to i in the range(0,512) dimensions of the word embedding vector of black and brown .**

There are many ways to achieve this goal. The designers found a clever way to use a unit sphere to represent **positional encoding with sine and cosine values.**

Vaswani et al. (2017) provide **sine and cosine functions** so that we can generate **different frequencies for the positional encoding (PE)** for **each position and each dimension i of the d_{model} = 512 of the word embedding vector:**

$$PE_{(pos\ 2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos\ 2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

- If we go back to the sentence we are parsing in this section, we can see that black is in position pos=2 and brown is in position pos=10

  "The black cat sat on the couch and the brown dog slept on the rug ."

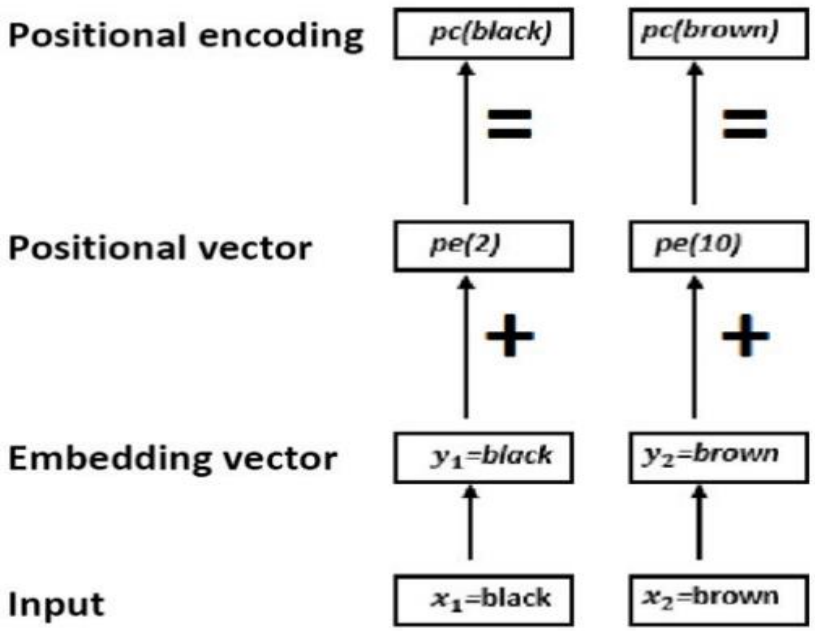
If we apply the **sine and cosine functions** literally for pos=2 and pos=10 we obtain a size=512 positional encoding vector:

The **encoding of the position** shows a **lower similarity value** than the **word embedding similarity.**

The **positional encoding has taken these words apart**. Bear in mind that word embeddings will vary with the corpus used to train them.

Adding positional encoding to the embedding vector

- The authors of the Transformer found a simple way by merely **adding the positional encoding vector to the word embedding vector:**



If we go back and take the word embedding of black , for example, and name it **y1=black**, we are ready to add it to the **positional vector pe(2)** we obtained with **positional encoding functions**. We will obtain the **positional encoding pc(black) of the input word black : pc(black)=y1+pe(2).**

if we apply it as shown, we might lose the information of the word embedding, which will be minimized by the positional encoding vector.

There are many possibilities to increase the value of y1 to make sure that the information of the word embedding layer can be used efficiently in the subsequent layers.

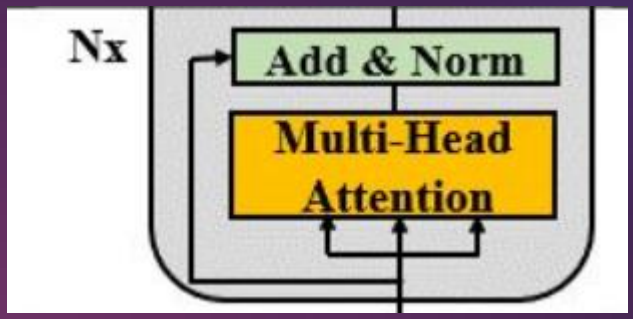
One of the many possibilities is to add an arbitrary value to y1, the word embedding of black :

y1*sqrt(d_{model})

that the initial word **similarity of their embeddings was very high, with a value of 0.99** . Then we saw **the positional encoding vector of positions 2 and 10 drew these two words apart with a lower similarity value of 0.86 .**

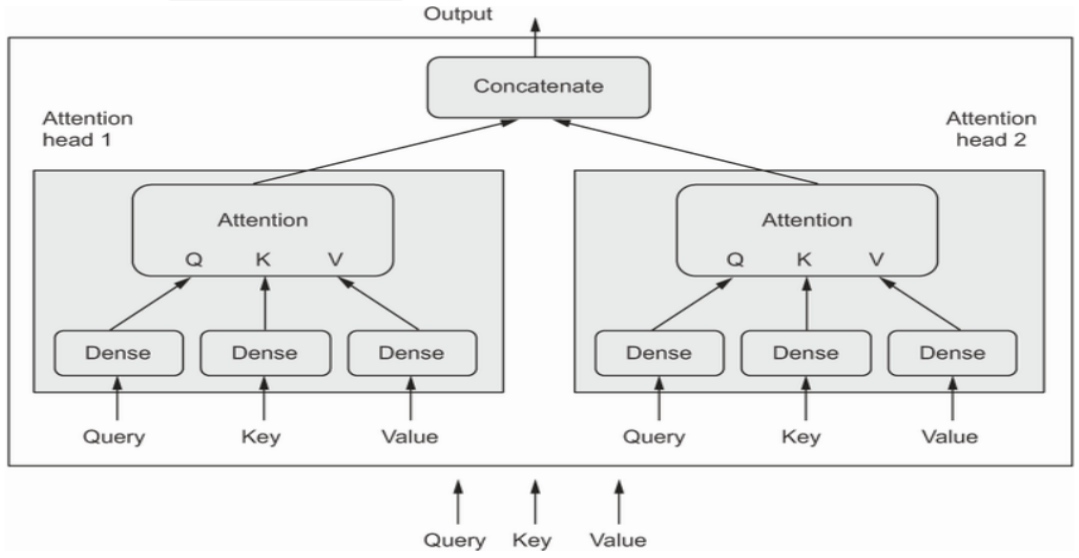
Finally, we added the word embedding vector of each word to its respective positional encoding vector. We saw that this brought the **cosine similarity of the two words to 0.96 .**The **positional encoding of each word now contains the initial word embedding information and the positional encoding values.**

Sub-layer 1: Multi-head attention



- The **multi-head attention sub-layer** contains **eight heads** and is followed by **post-layer normalization**, which will **add residual connections** to the output of the sub-layer and **normalize** it:

Figure 11.8 The `MultiHeadAttention` layer



The **input of the multi-attention sub-layer** of the first layer of the encoder stack is a vector that **contains the embedding and the positional encoding of each word**. The next layers of the stack do not start these operations over.

The dimension of the vector of each word x_n of an input sequence is d_{model} = 512:

pe(x_n)=[d1=9.09297407e-01, d2=9.09297407e-01,.., d512 = 1.00000000e+00]

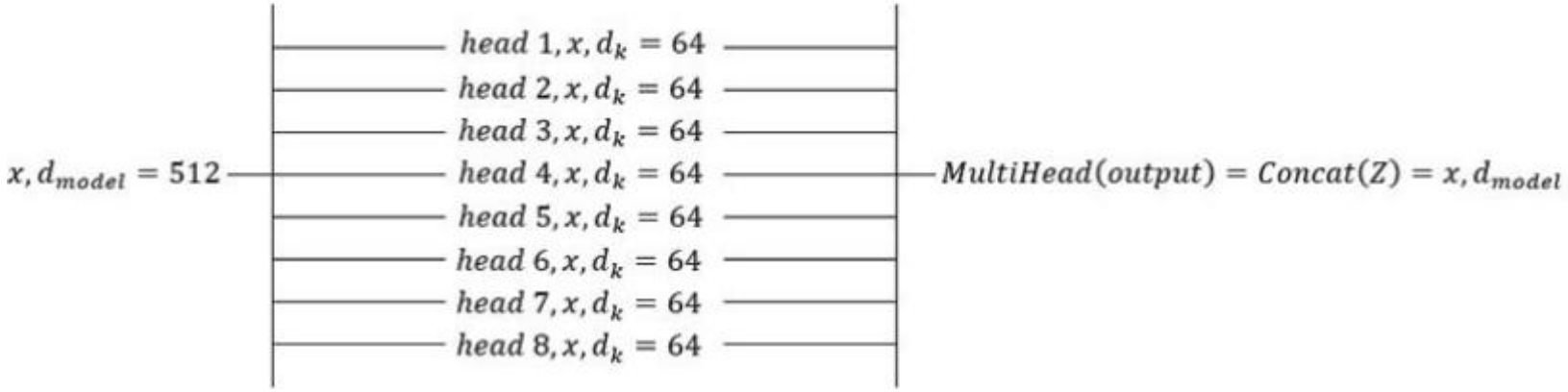
The representation of each word x_n has become a vector of d_{model} = 512 dimensions.

Each word is mapped to all the other words to determine how it fits in a sequence.

In the following sentence, we can see that "it" could be related to "cat" and "rug" in the sequence:

"The cat sat on the rug and it was dry-cleaned."

A better way is to **divide the d_{model} = 512 dimensions of each $pe(x_n)$ into 8 d_k = 64 dimensions.** We then can **run the 8 "heads" in parallel to speed up the training and obtain 8 different representation subspaces of how each word relates to another.** We then can run the 8 "heads" in parallel to speed up the training and obtain 8 different representation subspaces of how each word relates to another:



Each head

three primary matrixes: The input to each head is a sequence of vectors X, which can either be the embedded representations (embeddings) or the output of the previous layer. This input is projected into three different spaces to create the Query (Q), Key (K), and Value (V) matrices: Q= XW^Q ,K= XW^K ,V= XW^v

Calculating the Attention Scores:

To calculate the attention between tokens, the dot product of Q and K is computed: scores= QK^T

The scores are then scaled by dividing them by the square root of d_k , which stabilizes the gradients and improves numerical performance.

Applying Soft max to Compute Attention Weights

The scaled scores are passed through the **Soft max** function to compute the attention weights:

Attention Weights=softmax(scores)

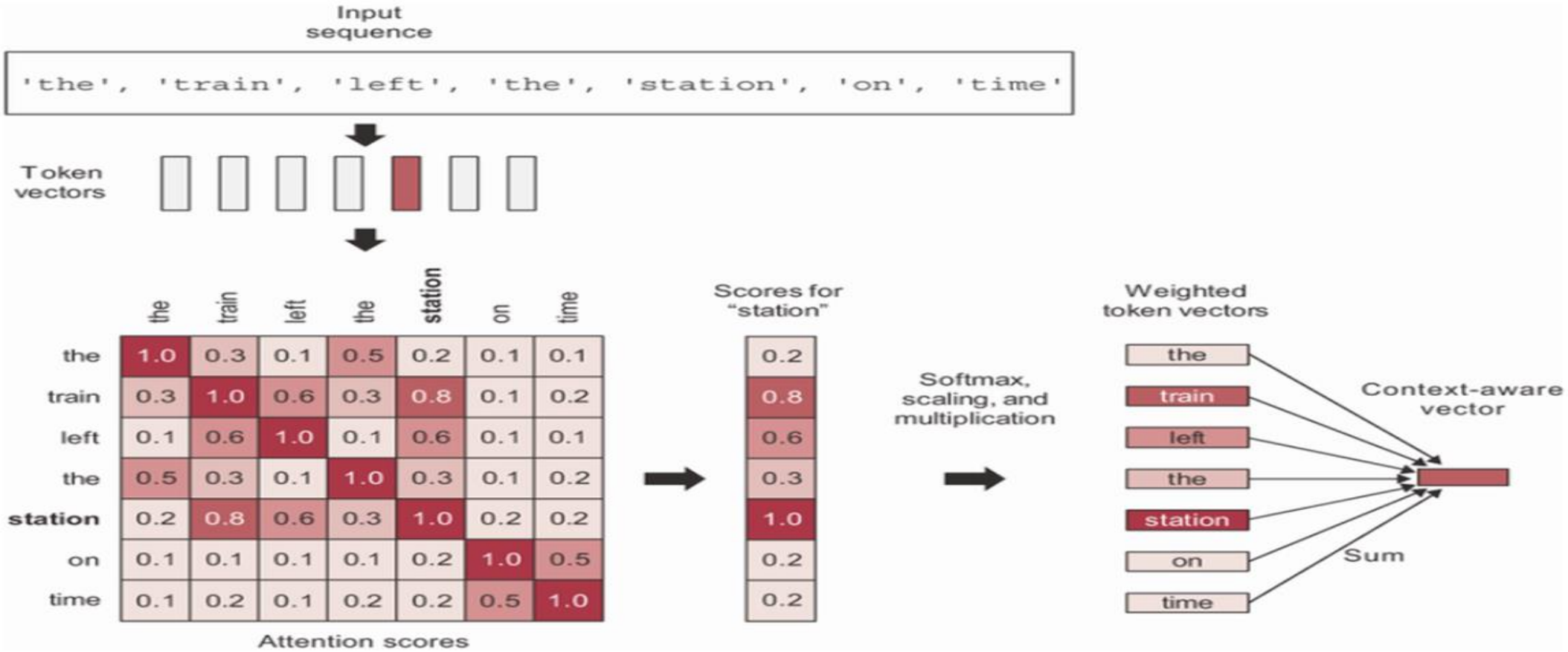
Each row of this matrix represents the attention weights for one token relative to all other tokens.

Weighted Combination of the Values (V)

The attention weights are then applied to the values V:

Output_{head}=Attention Weights·V

Figure 11.6 Self-attention: attention scores are computed between "station" and every other word in the sequence, and they are then used to weight a sum of word vectors that becomes the new "station" vector.



- **Final Steps in Multi-Head Attention**

In **Multi-Head Attention**, the above process is repeated in parallel for multiple heads. The outputs from all heads are combined as follows:

Concatenation of Head Outputs: The outputs from all heads

Concat(Output_1, Output_2, ..., Output_8)

Linear Transformation: The concatenated outputs are passed through another learnable weight matrix W_0 to transform them into the final output:

MultiHead(output) = Concat(Output_1, Output_2, ..., Output_8)W_0

W_0 is a learnable weight matrix with dimensions $\mathbf{d_{model}} \cdot \mathbf{d_{model}}$

Post-layer normalization

Each attention sub-layer and each feedforward sub-layer of the Transformer is followed by post-layer normalization (Post-LN)

The **Post-LN contains an add function and a layer normalization process.** The add function processes the residual connections that come from the input of the sub-layer. The goal of the residual connections is to make sure critical information is not lost. The Post-LN or layer normalization can thus be described as follows:

LayerNorm(x+Sublayer(x))

Sublayer(x) is the sub-layer itself. x is the information available at the input step of Sublayer(x).

The input of LayerNorm is a vector v resulting from x + Sublayer(x). d_{model} = 512 for every input and output of the Transformer, which standardizes all the processes. Many layer normalization methods exist, and variations exist from one model to another. The basic concept for v= x + Sublayer(x) can be defined by LayerNorm(v) :

$$\text{LayerNorm(v)}= \gamma \frac{v-\mu}{\sigma} + \beta$$

The variables are:

- μ is the mean of v of dimension d. As such:

$$\mu = \frac{1}{d}\sum_{k=1}^{d} v_k$$

- σ is the standard deviation v of dimension d. As such:

$$\sigma^2 = \frac{1}{d}\sum_{k=1}^{d} (v_{k-\mu})^2$$

- γ is a scaling parameter.
- β is a bias vector.

Sub-layer 2: Feedforward network

The **input of the FFN is the d_{model} = 512 output of the Post-LN of the previous sub-layer:**

The FFN sub-layer can be described as follows:

The FFNs in the encoder and decoder are **fully connected.**

The FFN is a **position-wise network. Each position is processed separately and in an identical way.**

The FFN **contains two layers and applies a "ReLU" activation function.**

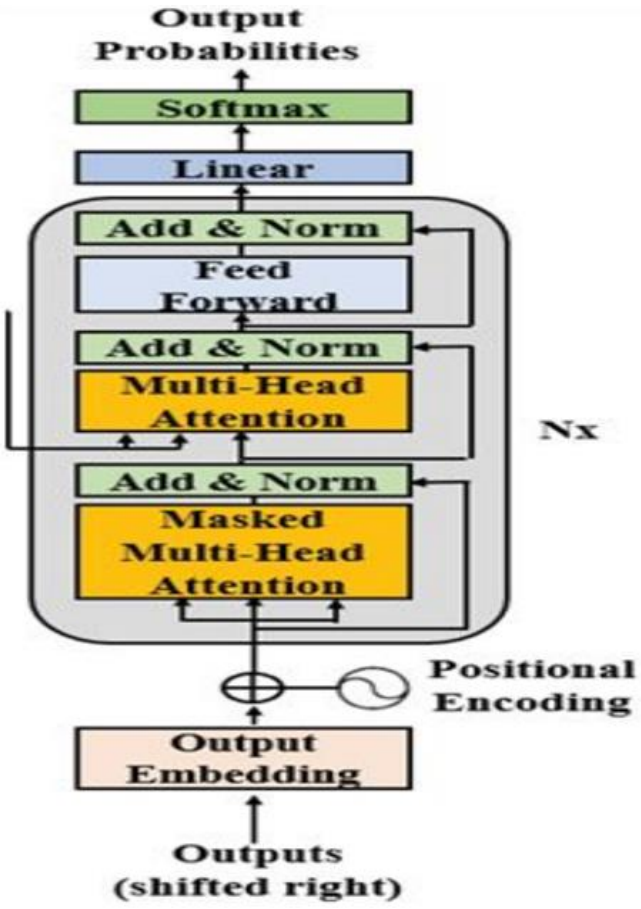
The input and output of the FFN layers is d_{model} = 512, but **the inner layer is larger with d_{ff} = 2048**

The **output of the FFN goes to the Post-LN**, as described in the previous section. Then the **output is sent to the next layer of the encoder stack** and the **multi-head attention layer of the decoder stack.**

Let's now explore the decoder stack.

The decoder stack

- The layers of the decoder of the Transformer model are stacks of layers like the encoder layers. Each layer of the decoder stack has the following structure:



The structure of the decoder layer remains the same as the **encoder for all the N=6 layers of the Transformer model.** Each layer contains **three sub-layers**: **a multi-headed masked attention mechanism**, a **multi-headed attention mechanism**, and a **fully connected position-wise feedforward network**.

A residual connection, Sublayer(x), surrounds each of the three main sublayers in the Transformer model like in the encoder stack:

LayerNormalization(x + Sublayer(x))

Output embedding and position encoding

The structure of the sub-layers of the decoder is mostly the same as the sublayers of the encoder. The **output embedding layer and position encoding function are the same as in the encoder stack.**

In the Transformer usage we are exploring through the model presented by Vaswani et al. (2017), the output is a **translation we need to learn**.

The output words go through the word embedding layer, and then the positional encoding function, like in the first layer of the encoder stack.

Let's see the specific properties of the multi-head attention layers of the decoder stack.

The attention layers

The Transformer is an auto-regressive model. It uses the **previous output sequences as an additional input**.

- Each of these two layers serves a specific purpose essential for generating the output sequence:

1. **Masked Multi-Head Attention (First Layer):**
   This layer models the internal dependencies of the output sequence. Its purpose is to ensure that, while generating each token, the model only attends to previous tokens in the output sequence and prevents using future tokens.
2. **Multi-Head Attention with Encoder (Second Layer):**
   This layer models the relationship between the input and output sequences. The model needs to attend to the encoder's output to make more accurate predictions for each token in the output sequence.

- **Masked Multi-Head Attention (Sub-layer 1):**
- **What is the role of this sub-layer?**
- The **Masked Multi-Head Attention** ensures that the model only looks at **previous positions and the current position in the sequence.**
- This is similar to completing a sentence while only knowing the preceding words and **guessing the next words.**

- **How is masking applied?**
- Masking creates a mask matrix that hides **future positions from the model**:
  - For example, if the model is predicting the **third token, the mask ensures that only tokens 1 and 2 are visible.**
  - The mask matrix can be represented as:

$$Mask[i,j] = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

- **How Does the Masking Matrix Work?**
- The masking matrix is a **matrix that shows which positions the model can attend to and which ones should be hidden.** Essentially, this matrix is applied during the attention calculations and ensures the model can only focus on allowed positions.

- **Steps:**

1. **Attention Scores Calculation:** **Before applying the mask, the model calculates the attention scores. These scores represent how much attention the model should give to each token in the sequence.**
2. **Applying the Mask:** **After calculating the attention scores, the masking matrix is applied:**
   1. If j>j (i.e., the position j is after position i, where i is the current position), the corresponding value in the attention score matrix is replaced by **negative infinity**
   2. If j≤i, the model can attend to that position, and the attention score is left unchanged.
3. **Soft max Application:** After applying the mask, the attention scores go through the **soft max function**, which normalizes them into a probability distribution.
   1. When an attention score is $-\infty$, **after applying the soft max function, this score will become zero.**
   2. This ensures that the model doesn't attend to positions it shouldn't, i.e., the future positions.

- **Multi-Head Attention with Encoder Output (Second Layer of Decoder)**
- **Purpose:** The second Multi-Head Attention layer in the decoder focuses on **attending to the encoder's output**. This is a **cross-attention mechanism** that helps the decoder make predictions based on the input sequence (provided by the encoder).
- **Input**:
  The input to this layer is the **query (Q)**, which comes from the output of the previous decoder layer (i.e., from the masked multi-head attention). The **key (K)** and **value (V)** come from the encoder's output, which contains the **encoded representations** of the input sequence (source sequence). In other words, while the first attention layer attends to the previous output sequence, this second layer attends to the input sequence, which has already been processed by the encoder.

- **No Masking:** **Unlike the first layer, no mask is applied in this attention layer** because there is no need to hide future tokens. Here, the model needs to attend to all positions in the input sequence, as the encoder has already processed the entire sequence. This allows the **decoder to access the relevant parts of the input sequence when generating each token in the output.**
- **Mechanism of Attention:** Just like the first layer, the multi-head attention mechanism works by computing the attention weights using queries (Q from the decoder) and keys (K from the encoder). The attention scores determine how much focus each token in the encoder output should have when generating a token in the decoder.

- Why No Masking? The absence of masking in this layer allows the decoder to freely attend to all positions of the input sequence. This is because the encoder has already processed the entire input sequence, so the decoder needs to focus on the most relevant parts of the input sequence to generate accurate predictions for each token in the output sequence.
- Objective: **This layer helps the decoder to establish relationships between the input and output sequences. For example, in machine translation, the model learns how each word or token in the output sequence relates to the corresponding word or token in the input sequence**. This is crucial for tasks like translation, where the output depends on the full context of the input sequence.

The FFN sub-layer, the Post-LN, and the linear layer

- **FFN Sub-layer in the Transformer:**
- Feed-Forward Networks (FFNs) in the Transformer architecture are used to create nonlinear transformations of the data. This sub-layer exists in both the encoder and decoder stacks and typically consists of two linear layers with a ReLU activation function between them.
- **FFN Structure:**
  - The first linear layer transforms the input into a higher-dimensional feature space (usually from $d_{model} to$).
  - Then a nonlinear activation function like **ReLU** is applied, helping the model learn more complex patterns.
  - After that, another linear layer is applied to map the output back to the original dimension of the model (d_{model} . d_{model}).

- **Generating the Output Sequence Step-by-Step:**
  - The **Transformer** is an **autoregressive** model, meaning it generates each output token based on previous tokens. It generates output sequentially, one token at a time.
  - This feature is particularly useful for tasks like **machine translation** or **language modeling**, where the model must make predictions step-by-step based on previous tokens.

- **Learning and Optimization Process:**
- During training, all the model parameters—such as the **linear layer parameters**, **FFN weights, and attention weights**—are continuously optimized. This process helps the model find the best parameters that improve its predictions and accuracy.
- **Backpropagation:** The model **uses backpropagation to update the weights**. In this process, the prediction error is propagated back to adjust the weights and parameters.
- **Weight Adjustment:** The learned parameters, like weights and biases, are updated during this process to improve the model's performance.

- **Linear Layer in the Transformer:**
- A **linear layer** in the **Transformer is used to produce the final output**. This layer transforms the **hidden state output into the desired output space.**
- **Transforming to Output Sequence:** The linear layer performs a **linear transformation** (a matrix multiplication), using its learned parameters (weights and biases) to generate a linear representation of the hidden state.

$$y = w*x + b$$

- **Learned Parameters (x and b):** These **parameters are learned during training to optimize the model's predictions.**

- **softmax function will convert into a probable element.**
- The decoder layer as the encoder layer will then go from layer l to layer l+1 up to the top layer of the N=6-layer transformer stack.

Reference

- Transformers for NLP (Denis Rothman)
- Deep Learning(Ian Goodfellow, Yoshua Bengio, and Aaron Courville)
- "Attention is All You Need" 2017 paper by Vaswani et al
- Deep Learning with Python, Second Edition(François Chollet)