Deep Learning Methods for NLP

Machine Learning for Natural Language Processing

Pejman Rasti

esaip La Salle
ÉCOLE D'INGÉNIEURS

Today Lecture Outline

- Deep Learning Framework
- The Multi-Layer Perceptron
- Recurrent Neural Network
- Self-Attention Mechanism and the Transformer Architecture

Motivations

So far, we have seen, **techniques to represent tokens with vectors**

Given a certain representations of tokens:

➔ **How can we model a sequence of tokens to perform a specific task?**

In the past 10 years, a “new” class of machine learning techniques has become very popular and successful in NLP: **Deep Learning**

In this session, we introduce Deep Learning with a focus on the methods used in NLP

Framework

We want to model $(X_1, .., X_T)$ i.e. find the correct label Y

$$dnn_\theta : \quad \mathbb{R}^{d,T} \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

- Output space is $\mathbb{R}^p$ for **Regression** tasks

- Output space is $[|0, K|]^p$ for **Classification** tasks

Framework

We want to model $(X_1, .., X_T)$ i.e. find the correct label Y

$$dnn_\theta : \quad \mathbb{R}^{d,T} \quad \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

Questions: when we do Deep Learning…

- **How do we define** dnn_θ **?**
- **How do we train** dnn_θ **with data ?**

Framework

Given a sequence of vectors $(X_1, .., X_T)$ we want to predict *Y*

$$dnn_\theta : \quad \mathbb{R}^{d,T} \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

Most Deep Learning Models (all the ones we will use in this course):

- are **parametric**
- defined as a **composition of "simple" functions (linear & non-linear)**
- are trained in an **end-to-end** fashion with **backpropagation**

NB: In Deep Learning, **the parametrization** is called **the Architecture**

Different Types of Architecture

How can we define our predictive function f ? dnn_θ ?

➔ Multi-Layer Perceptron
➔ Recurrent Layers
➔ Attention Layers
➔ Self-Attention Layers (in a Transformer Architecture)

Different Types of Architecture

How can we define our predictive function f ? dnn_θ ?

➔ Multi-Layer Perceptron
➔ Recurrent Layers
➔ Attention Layers
➔ Self-Attention Layers (in a Transformer Architecture)

How do we **train our model**? (i.e. estimate the parameters of the model)

➔ **Stochastic Gradient Descent** also called **backpropagation** in this context

The MultiLayer Perceptron (MLP)

aka "the Most simple Deep Learning Architecture"

The **MLP** works **on unidimensional data** (e.g. dimension *d*)

We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$))

$$dnn_\theta : \quad \mathbb{R}^d \to \mathbb{R}^2$$
$$X \mapsto \hat{Y}$$

The MultiLayer Perceptron (MLP)

aka "the Most simple Deep Learning Architecture"

The **MLP** works on unidimensional data (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$))

$$dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta \times d}, b_1 \in \mathbb{R}^{\delta}, W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^{\delta}$

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta \times d}$, $b_1 \in \mathbb{R}^{\delta}$, $W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^\delta$

➔ This model is a ***2-layer* MLP** model

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta \times d}$, $b_1 \in \mathbb{R}^{\delta}$, $W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^{\delta}$

➔ This model is a ***2-layer* MLP** model
➔ With **1 *hidden layer*** of δ dimension

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta\times d}$, $b_1 \in \mathbb{R}^{\delta}$, $W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^\delta$

- ➔ This model is a ***2-layer* MLP** model
- ➔ With **1 *hidden layer*** of dimension δ
- ➔ Taking as input a vector of **dimension *d*** to output a vector of **dimension 2**

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta\times d}$, $b_1 \in \mathbb{R}^{\delta}$, $W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^\delta$

➔ This model is a ***2-layer* MLP** model
➔ With **1 *hidden layer*** of dimensic δ
➔ Taking as input a vector of **dimension *d*** to output a vector of **dimension 2**
➔ Such a model is also referred to as a ***Feed-Forward* Neural Network (FNN)**

The MultiLayer Perceptron: Diagram View

Neurons

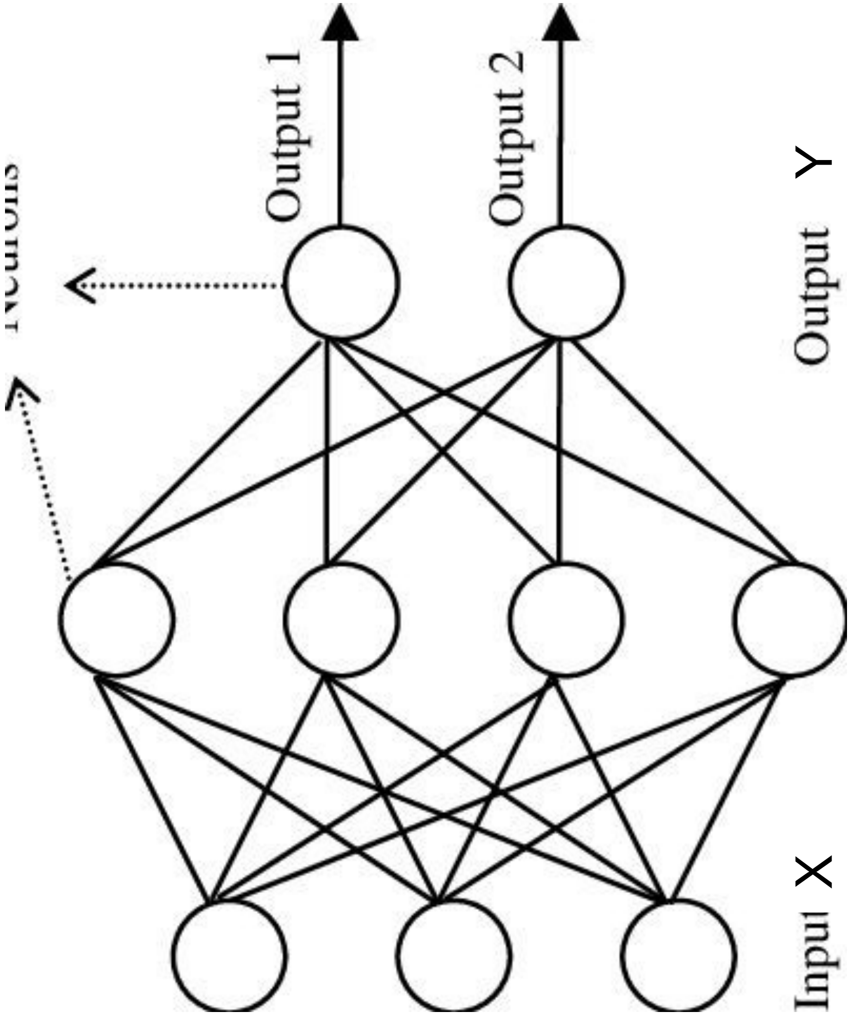
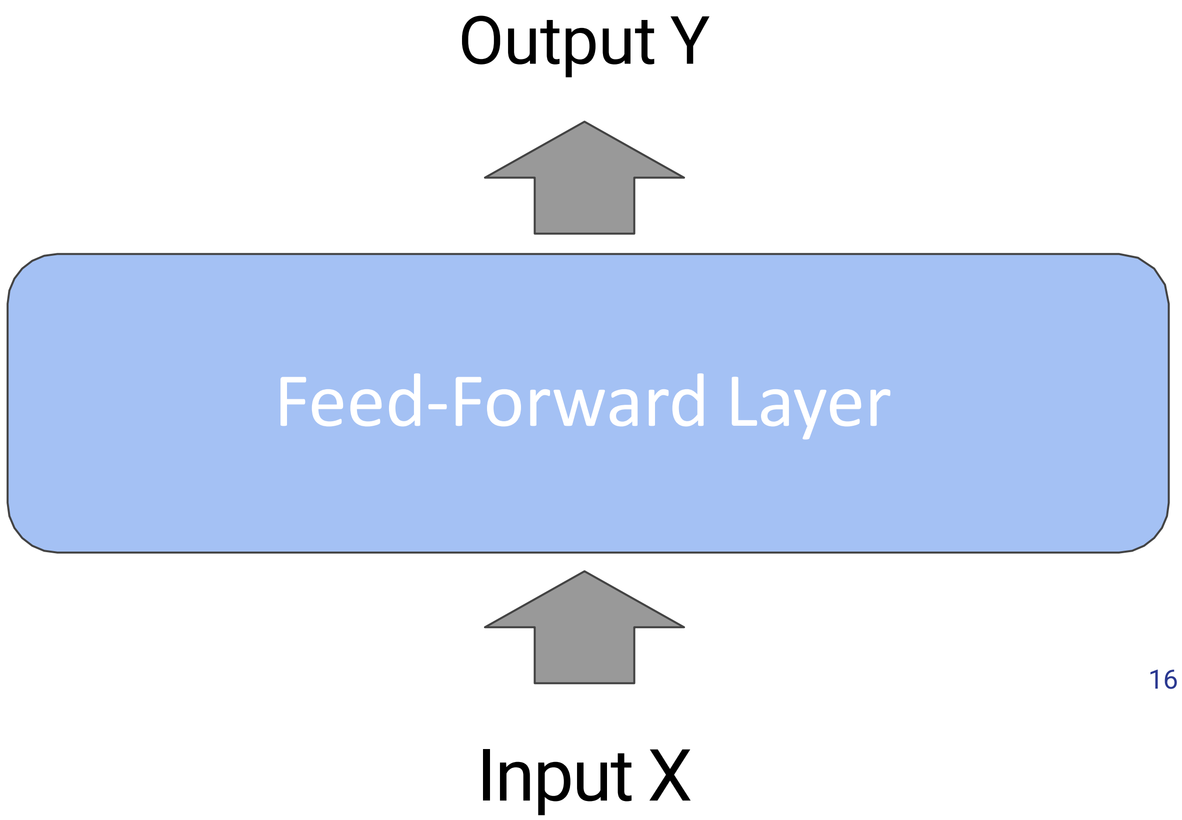


Figure from (R. Rezvani et. al. 2012)

In Deep Learning, it is usual to represent equations **with diagrams**

The MultiLayer Perceptron: Diagram View



In Deep Learning, it is usual to represent equations **with diagrams**

The MultiLayer Perceptron:

We have defined a 2-layers MLP model

We can define in the same way a **3-layers**, **4-layers**, **L-layers** MLP

$$\boxed{dnn_{(W_i\ b_i, i\in[|1,L|])}(X) = W_L\varphi_{L-1}(\ldots\varphi_2 \circ W_2\varphi_1(W_1X + b_1) + b_2)\ldots) + b_L}$$

W_l and b_l are trainable parameters. $W_l \in \mathbb{R}^{\delta_{l-1}\times\delta_l}$, $b_l \in \mathbb{R}^{\delta_l}$, with $\delta_l \in \mathbb{N}^*, \forall\, l \in [|1, L|]$

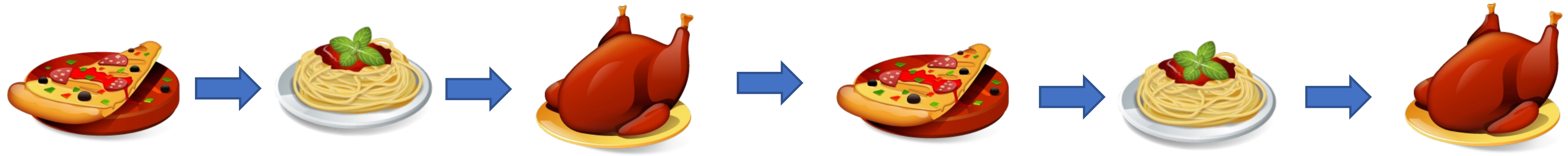
φ_l fixed non-linear functions, $\varphi_l : \mathbb{R}^{\delta_{l-1}} \to \mathbb{R}^{\delta_l}, \forall\, l \in [|1, L-1|]$

Recurrent Neural Network

Example

Cooking Schedule

Monday Tuesday Wednesday Thursday Friday Saturday

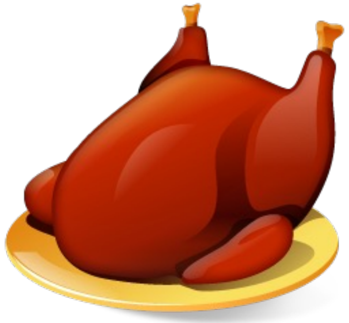


Example

Vectors

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

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

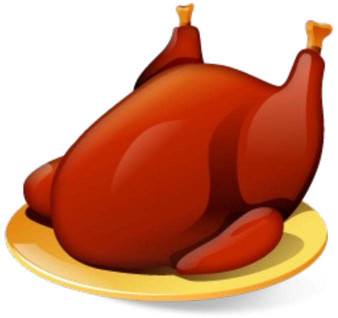


$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Example

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

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



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

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

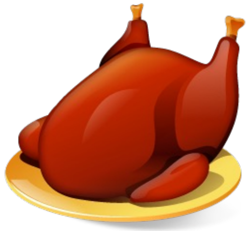
Example

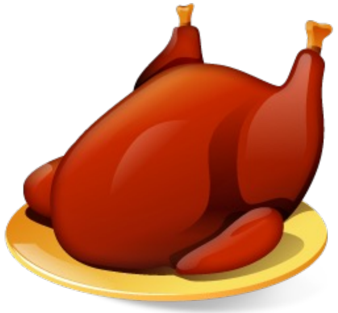
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

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

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

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





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

Example

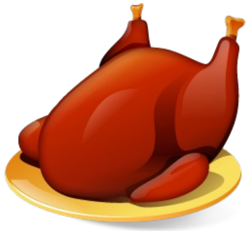
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

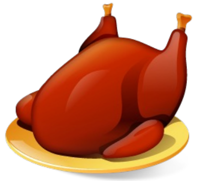
$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

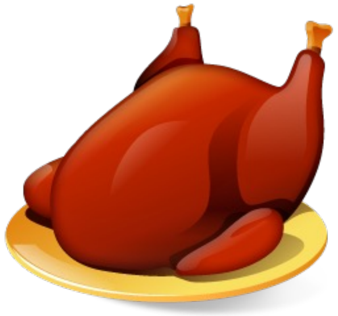
$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

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

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

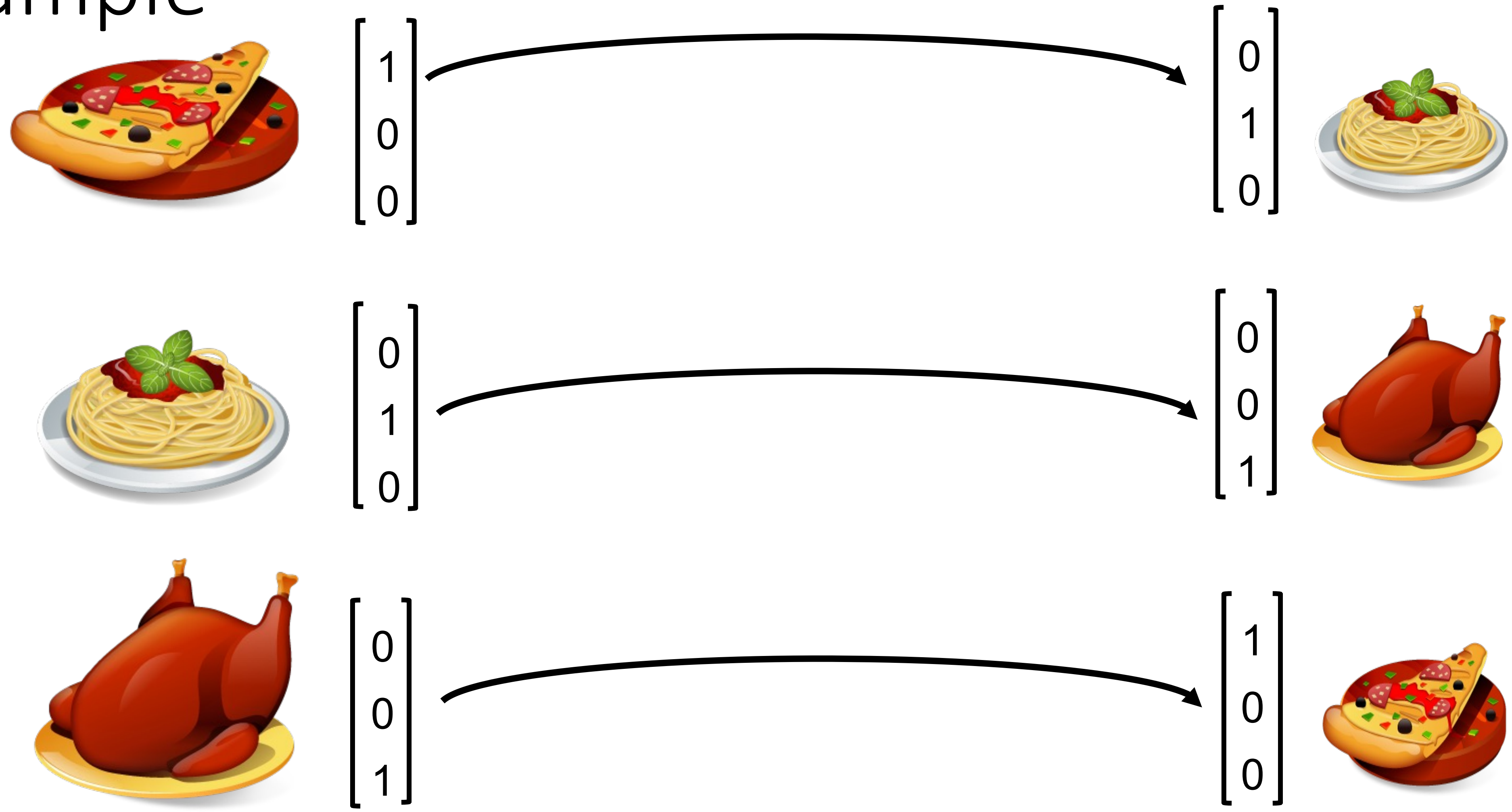






$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

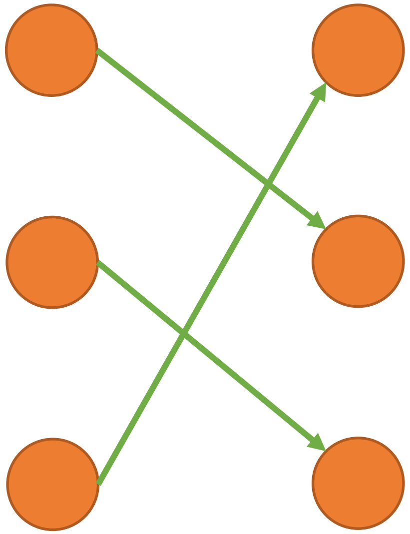
Example



Example

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

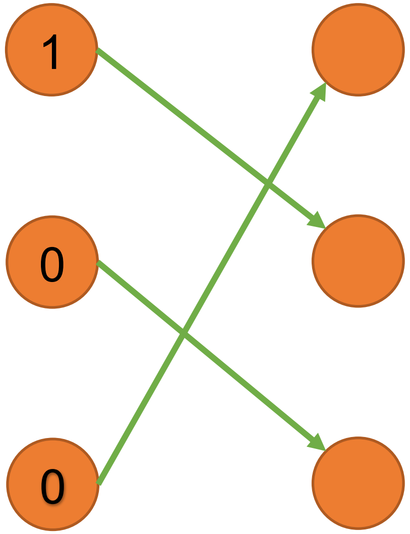
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$



Example

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

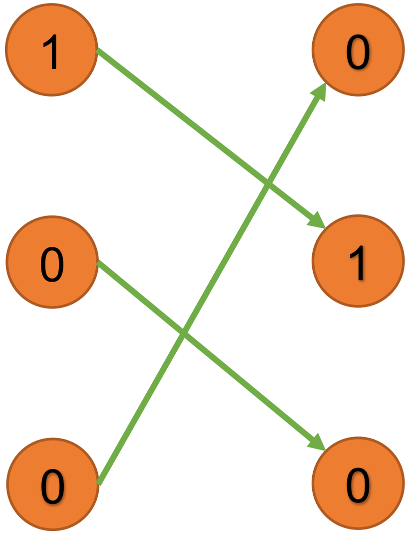
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$



Example

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

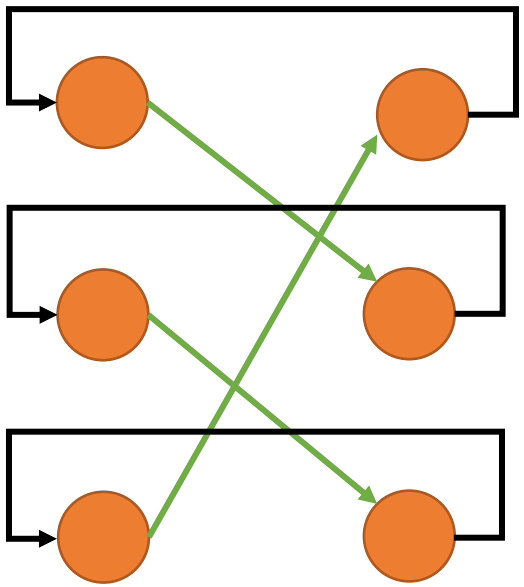
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$



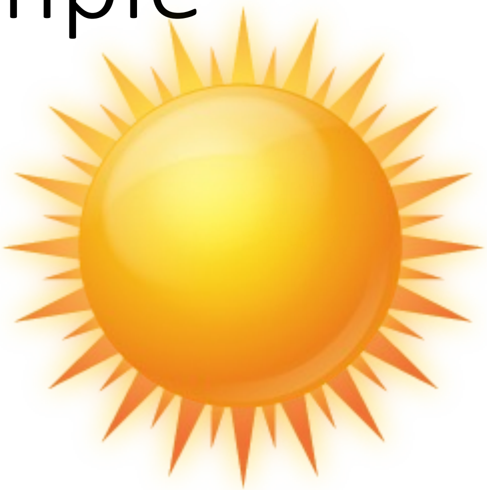
$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Example

Simple (Recurrent) neural network



Example



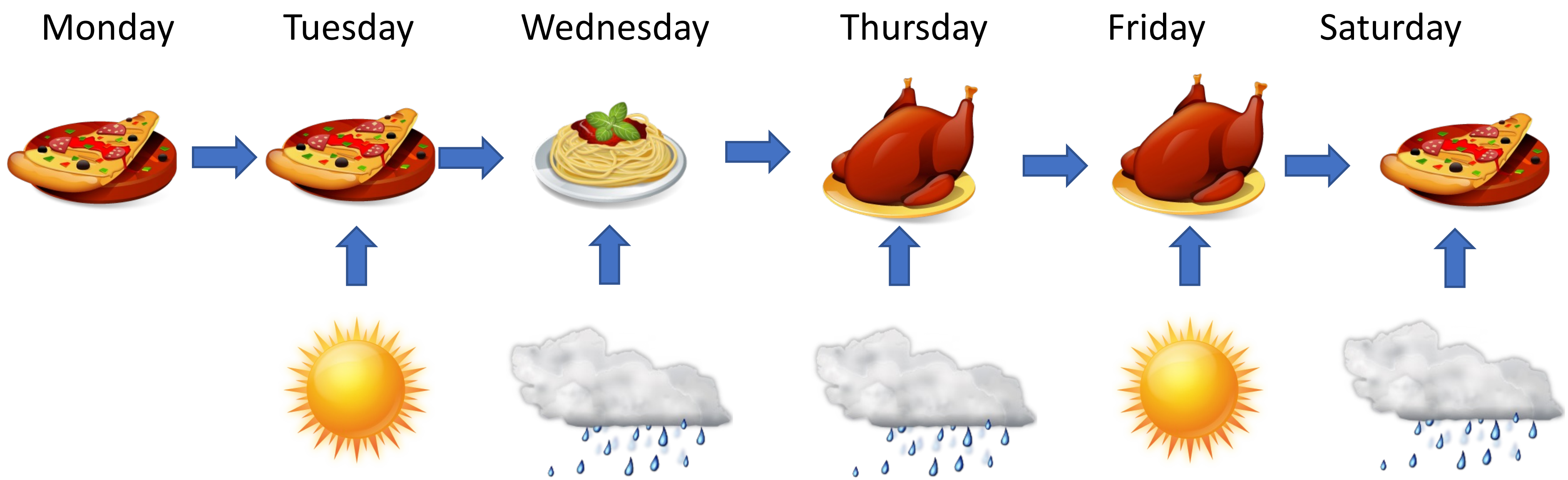
Weather

Same as yesterday

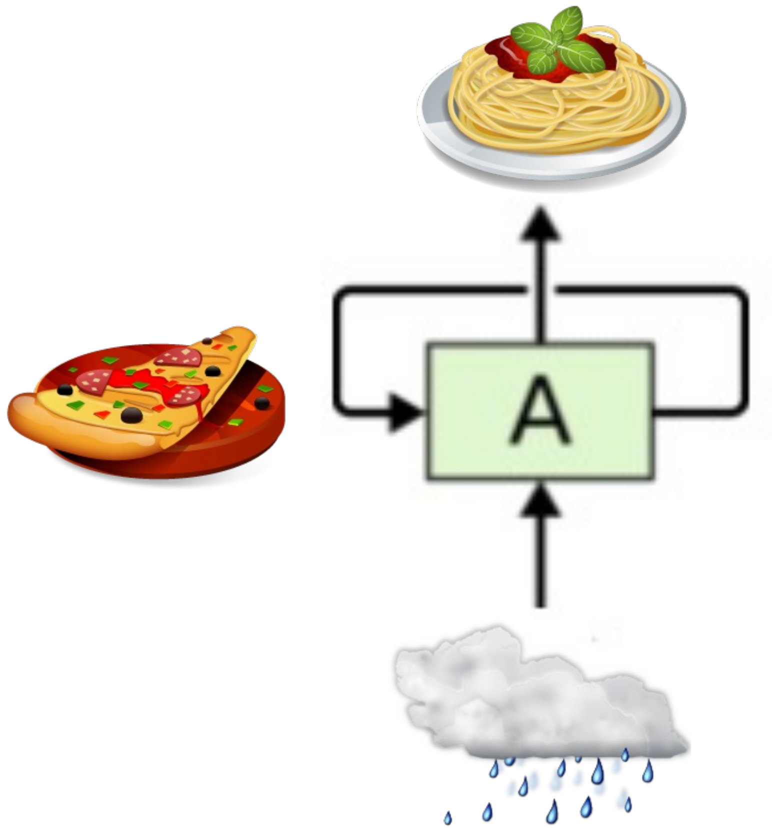
Next dish

Example

Cooking Schedule



Example

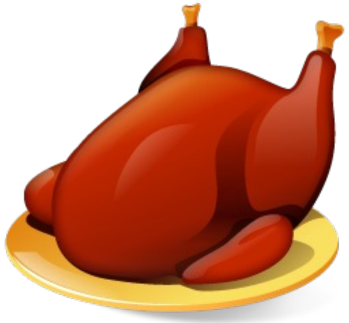


Example

Vectors

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

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



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

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

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

Example

More Complicated RNN

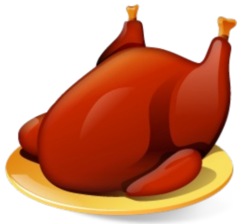
$$\left[\begin{array}{c} 1\ 0\ 0 \\ 0\ 1\ 0 \\ 0\ 0\ 1 \\ \hline 0\ 0\ 1 \\ 1\ 0\ 0 \\ 0\ 1\ 0 \end{array}\right] \quad + \quad \left[\begin{array}{c} 1\ 0 \\ 1\ 0 \\ 1\ 0 \\ \hline 0\ 1 \\ 0\ 1 \\ 0\ 1 \end{array}\right] \quad \Rightarrow$$

Food Add Weather Merge

Example

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

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



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

$$\left[\begin{array}{ccc}1&0&0\\0&1&0\\0&0&1\\\hline 0&0&1\\1&0&0\\0&1&0\end{array}\right]\begin{bmatrix}1\\0\\0\end{bmatrix}=\left[\begin{array}{c}1\\0\\0\\\hline 0\\1\\0\end{array}\right]$$

Food

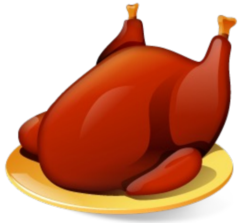
Same

Next Day

Example

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

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

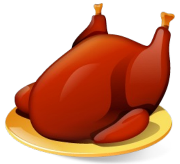


$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \left[\begin{array}{c} 0 \\ 1 \\ 0 \\ \hline 0 \\ 0 \\ 1 \end{array}\right]$$

Food

Same

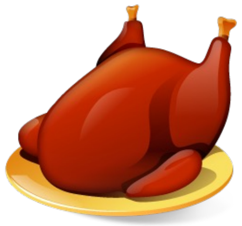


Next Day

Example

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

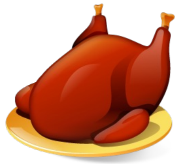
$$\begin{bmatrix}0\\1\\0\end{bmatrix}$$



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

$$\left[\begin{array}{ccc}1&0&0\\0&1&0\\0&0&1\\\hline 0&0&1\\1&0&0\\0&1&0\end{array}\right]\begin{bmatrix}0\\0\\1\end{bmatrix}=\left[\begin{array}{c}0\\0\\1\\\hline 1\\0\\0\end{array}\right]$$

Food



Same

Next Day

Example

More Complicated RNN

$$\left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] \quad + \quad \left[\begin{array}{cc} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ \hline 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{array}\right] \quad \Rightarrow$$

Food Add Weather Merge

Example

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

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

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

Weather

Same

Next Day

Example

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

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

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

Weather

Same

Next Day

Example

More Complicated RNN

$$
\left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] + \left[\begin{array}{cc} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ \hline 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{array}\right]
$$

Food Add Weather Merge

Example

Add

$$\left[\begin{array}{c}1\\0\\0\\\hline 0\\1\\0\end{array}\right] + \left[\begin{array}{c}0\\0\\0\\\hline 1\\1\\1\end{array}\right] = \left[\begin{array}{c}1\\0\\0\\\hline 1\\2\\1\end{array}\right]$$

 Same

 Next Day

 Same

 Next Day

Example

More Complicated RNN

$$\left[\begin{array}{c} 1\ 0\ 0 \\ 0\ 1\ 0 \\ 0\ 0\ 1 \\ \hline 0\ 0\ 1 \\ 1\ 0\ 0 \\ 0\ 1\ 0 \end{array}\right] \quad + \quad \left[\begin{array}{c} 1\ 0 \\ 1\ 0 \\ 1\ 0 \\ \hline 0\ 1 \\ 0\ 1 \\ 0\ 1 \end{array}\right]$$

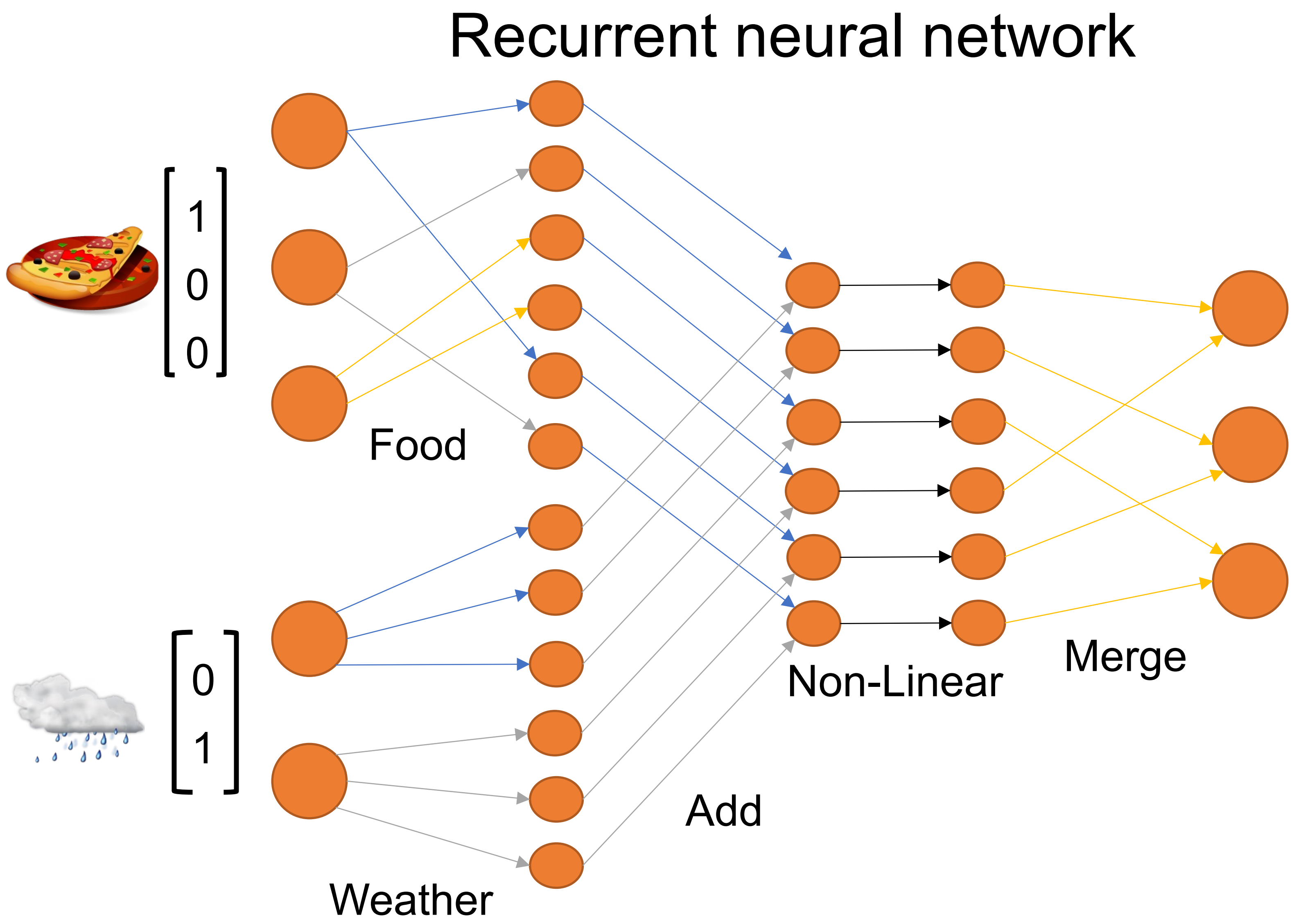
Food Add Weather Merge

Example

Merge

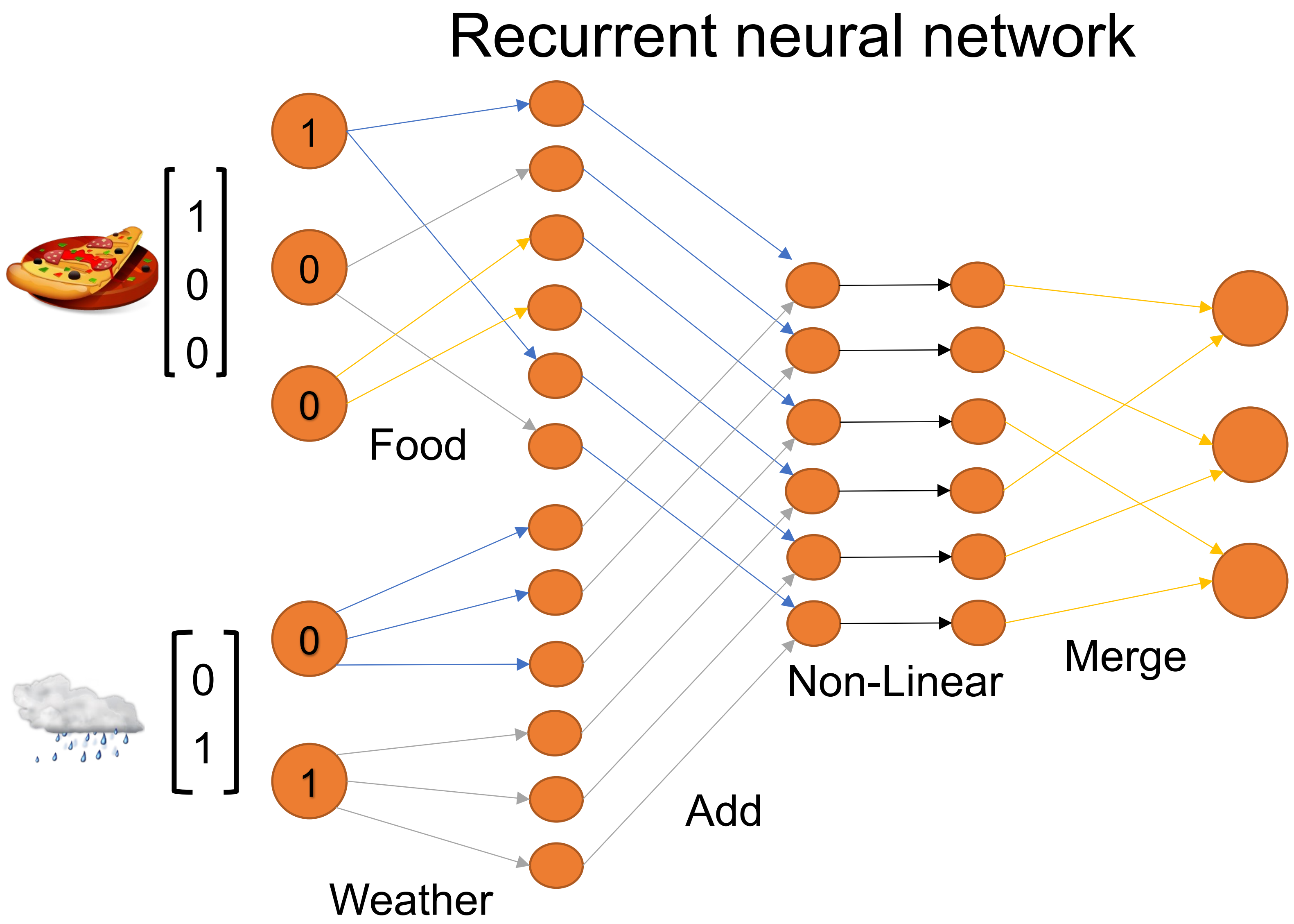
$$\left[\begin{array}{c}1\\0\\0\\ \hline 1\\2\\1\end{array}\right] \xrightarrow{\text{Non-Linear}} \left[\begin{array}{c}0\\0\\0\\ \hline 0\\1\\0\end{array}\right] \Rightarrow \begin{bmatrix}0+0\\0+1\\0+0\end{bmatrix} = \begin{bmatrix}0\\1\\0\end{bmatrix}$$

Example



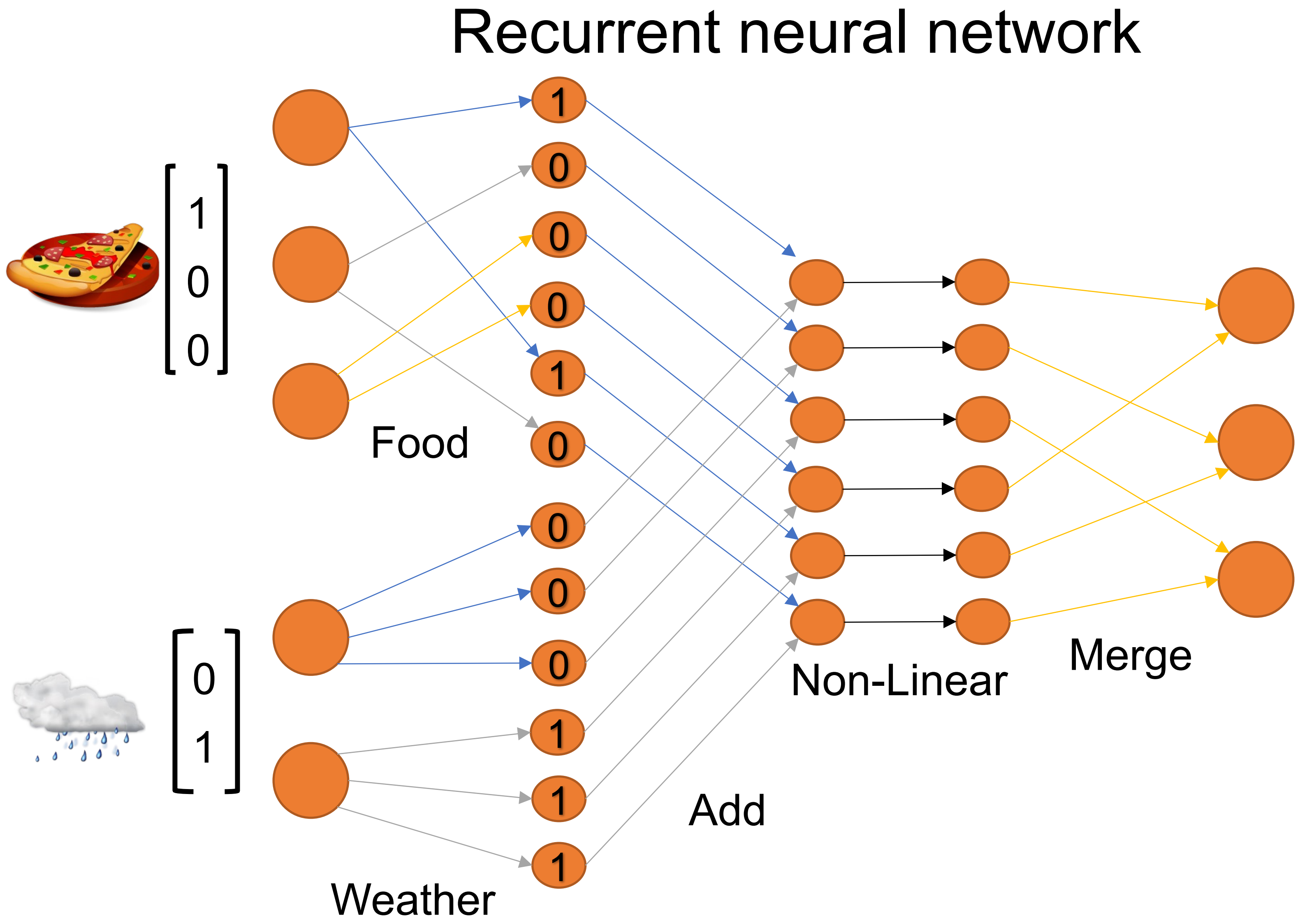
Recurrent neural network
1
0
0
Food
0
1
Weather
Add
Non-Linear
Merge

Example



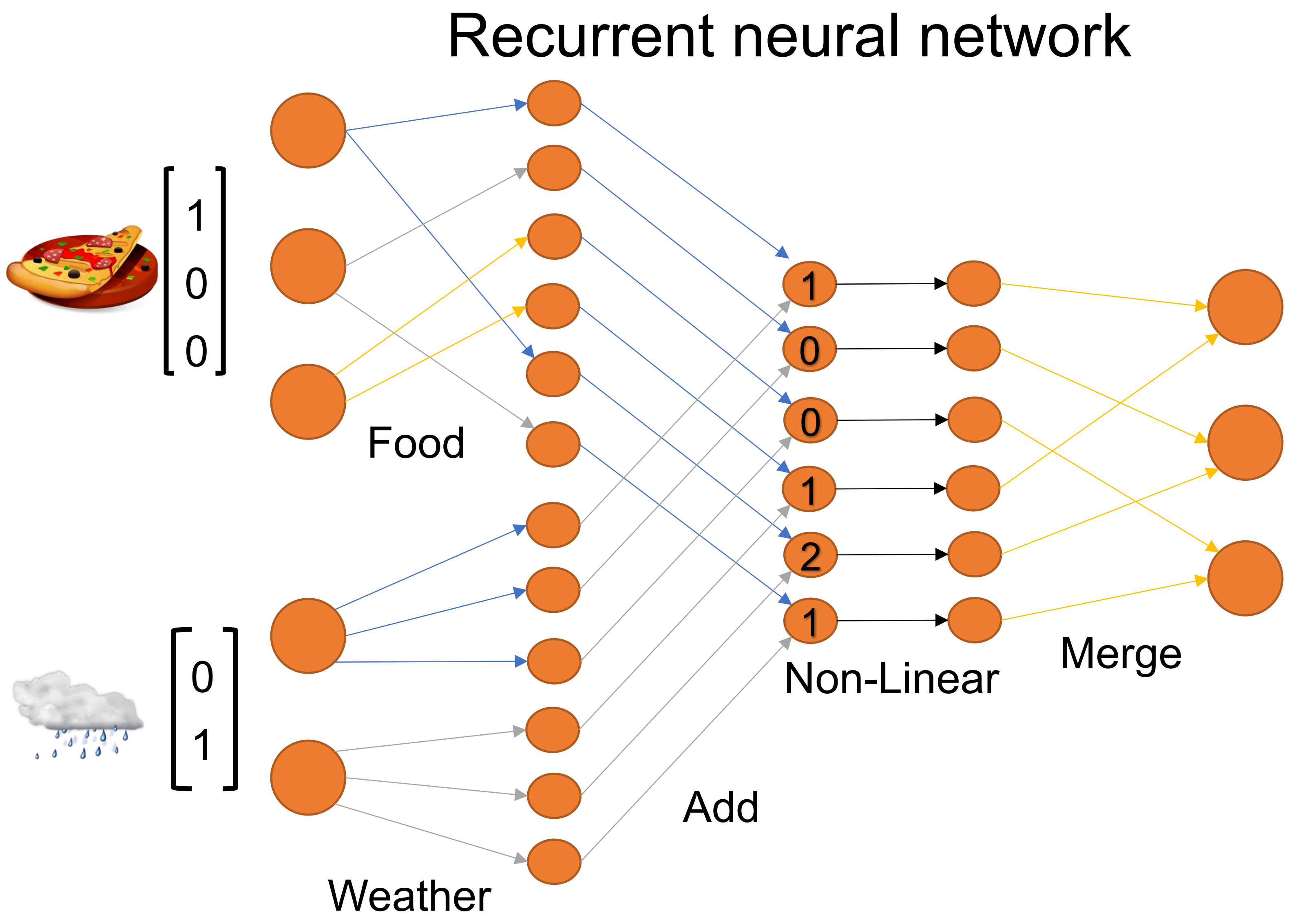
Recurrent neural network
1
0
0
Food
0
1
Weather
Add
Non-Linear
Merge

Example

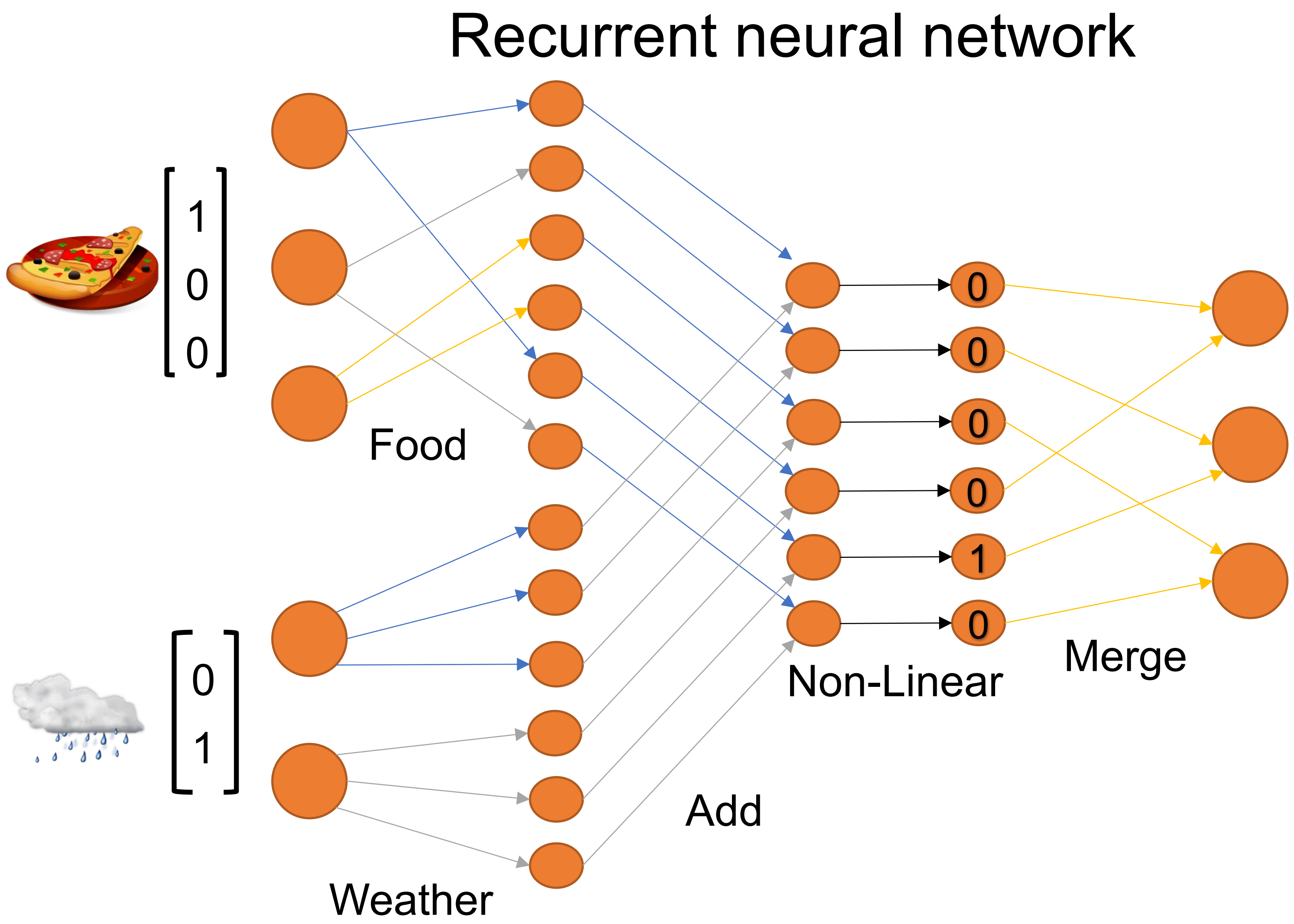


Recurrent neural network
1
0
0
Food
0
1
Weather
1
0
0
0
1
0
0
0
0
1
1
1
Add
Non-Linear
Merge

Example

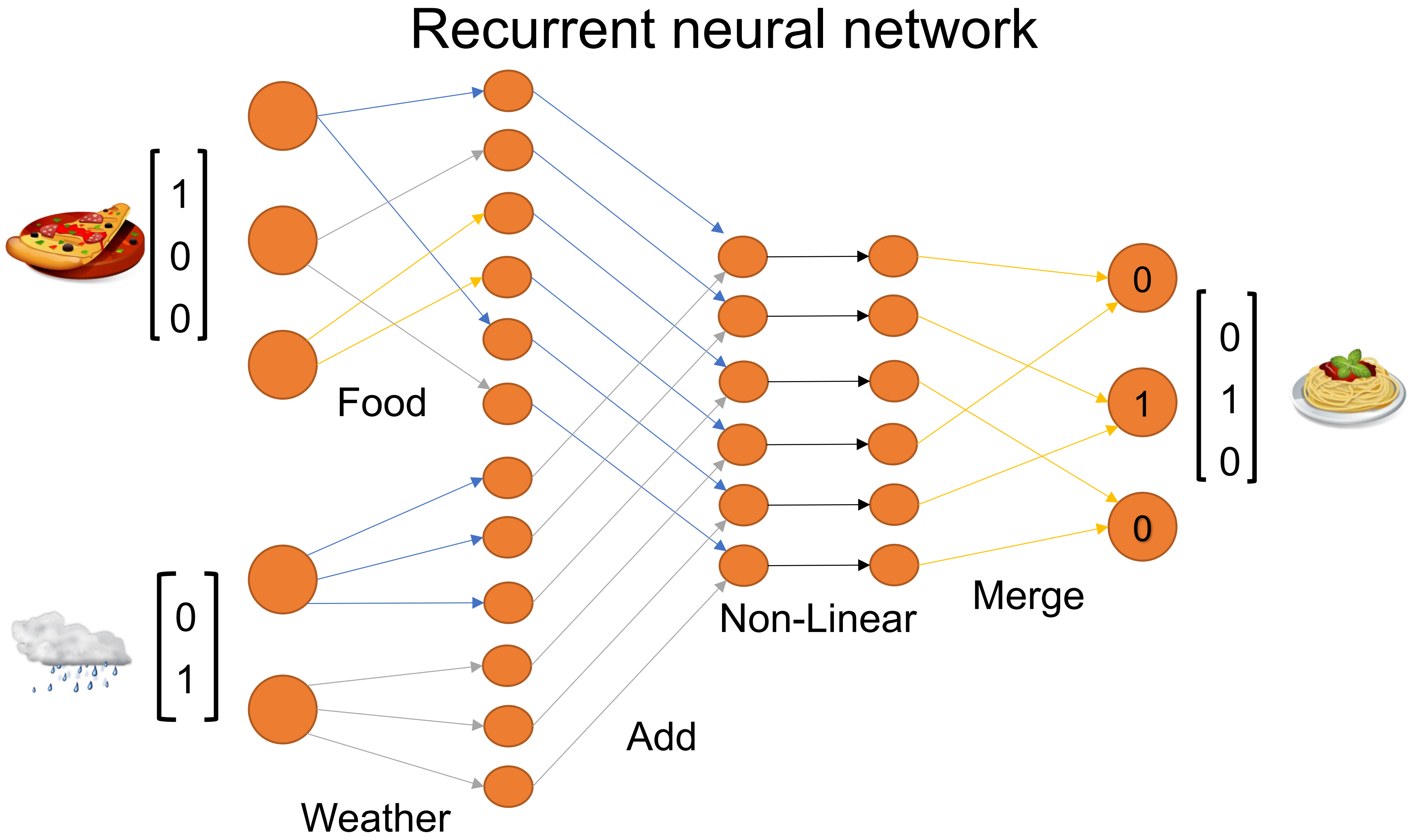

Recurrent neural network
1
0
0
Food
0
1
Weather
Add
1
0
0
1
2
1
Non-Linear
Merge

Example



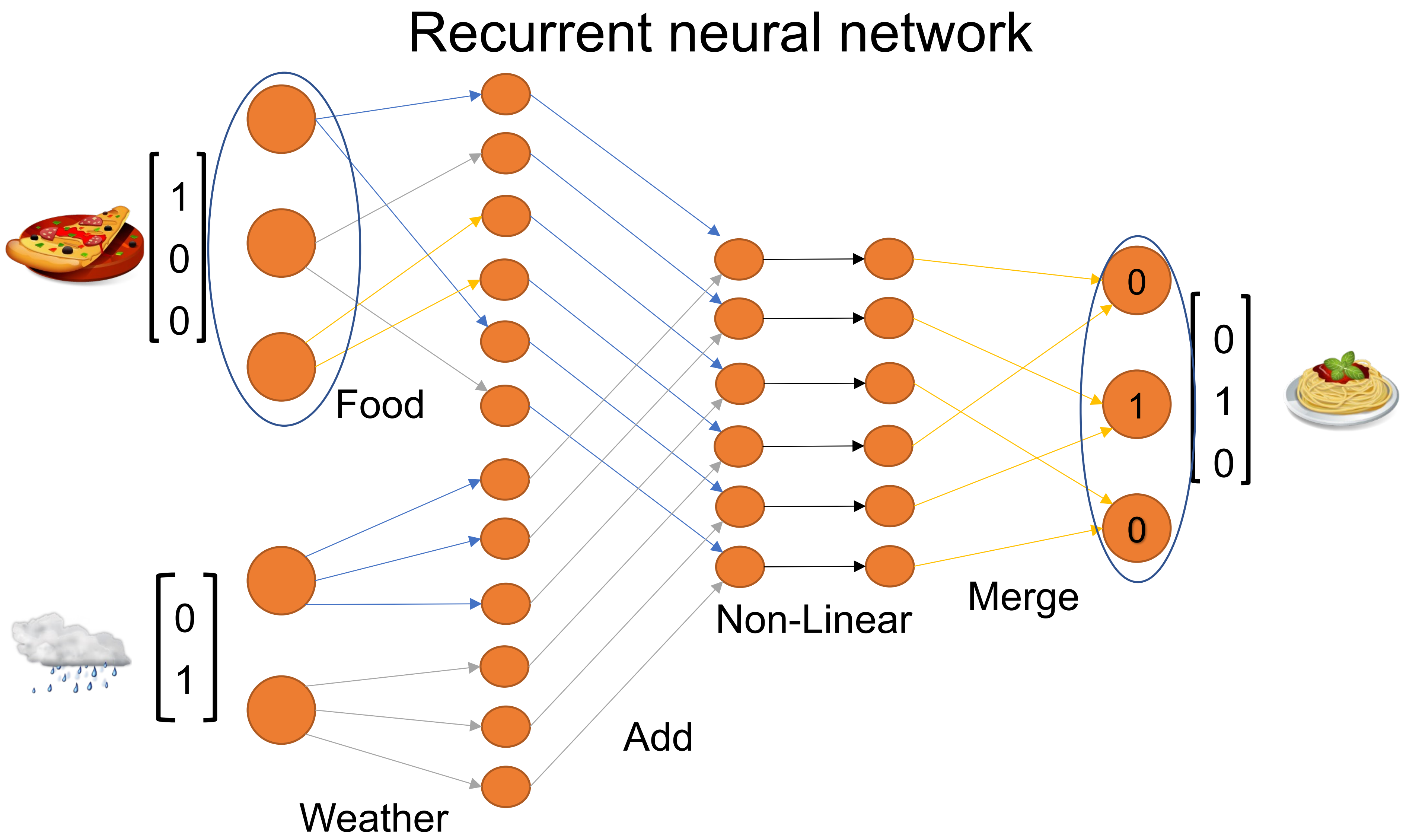
Recurrent neural network
1
0
0
Food
0
1
Weather
Add
0
0
0
0
1
0
Non-Linear
Merge

Example



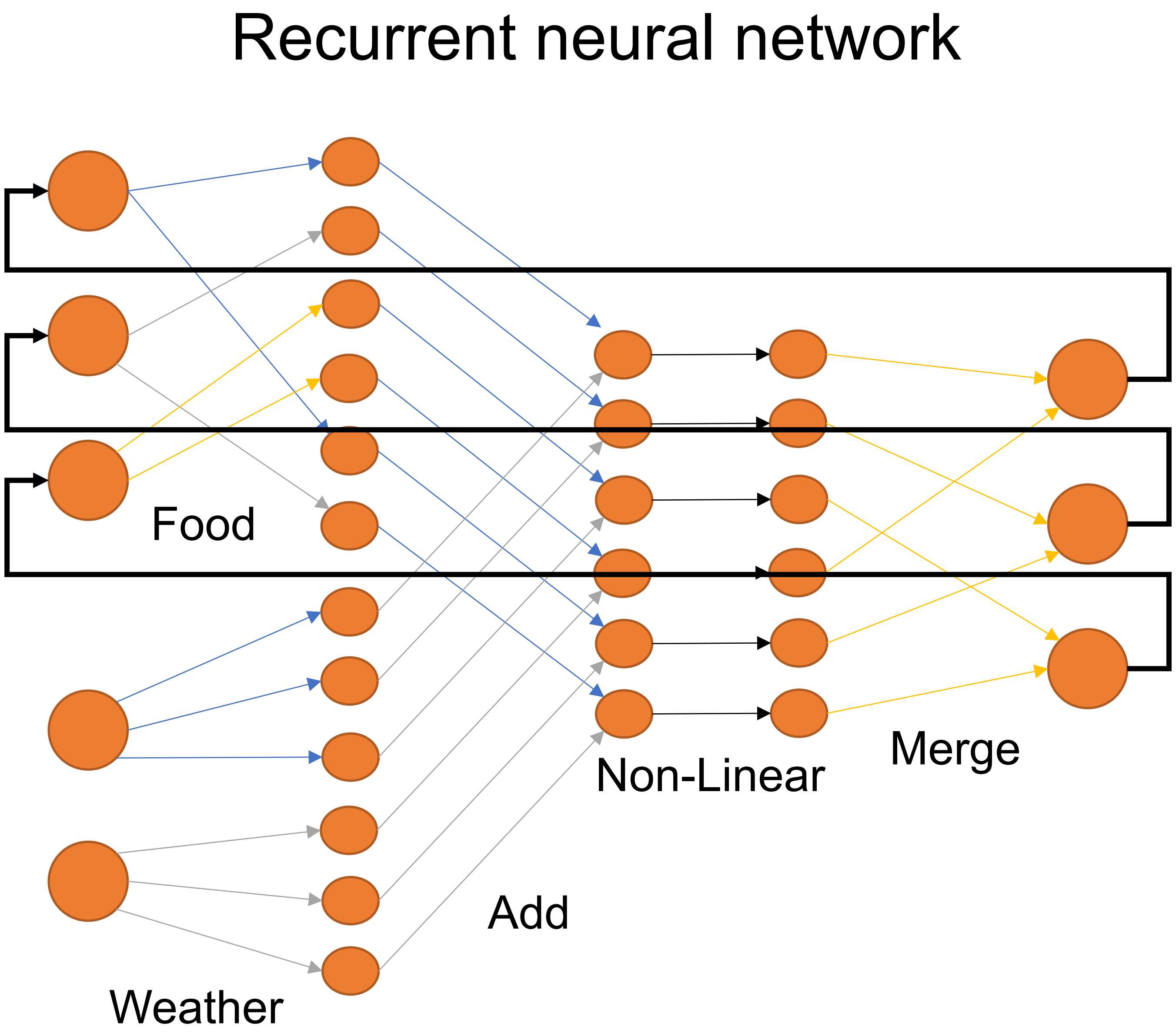
Recurrent neural network
1
0
0
Food
0
1
Weather
Add
Non-Linear
Merge
0
1
0
0
1
0

Example



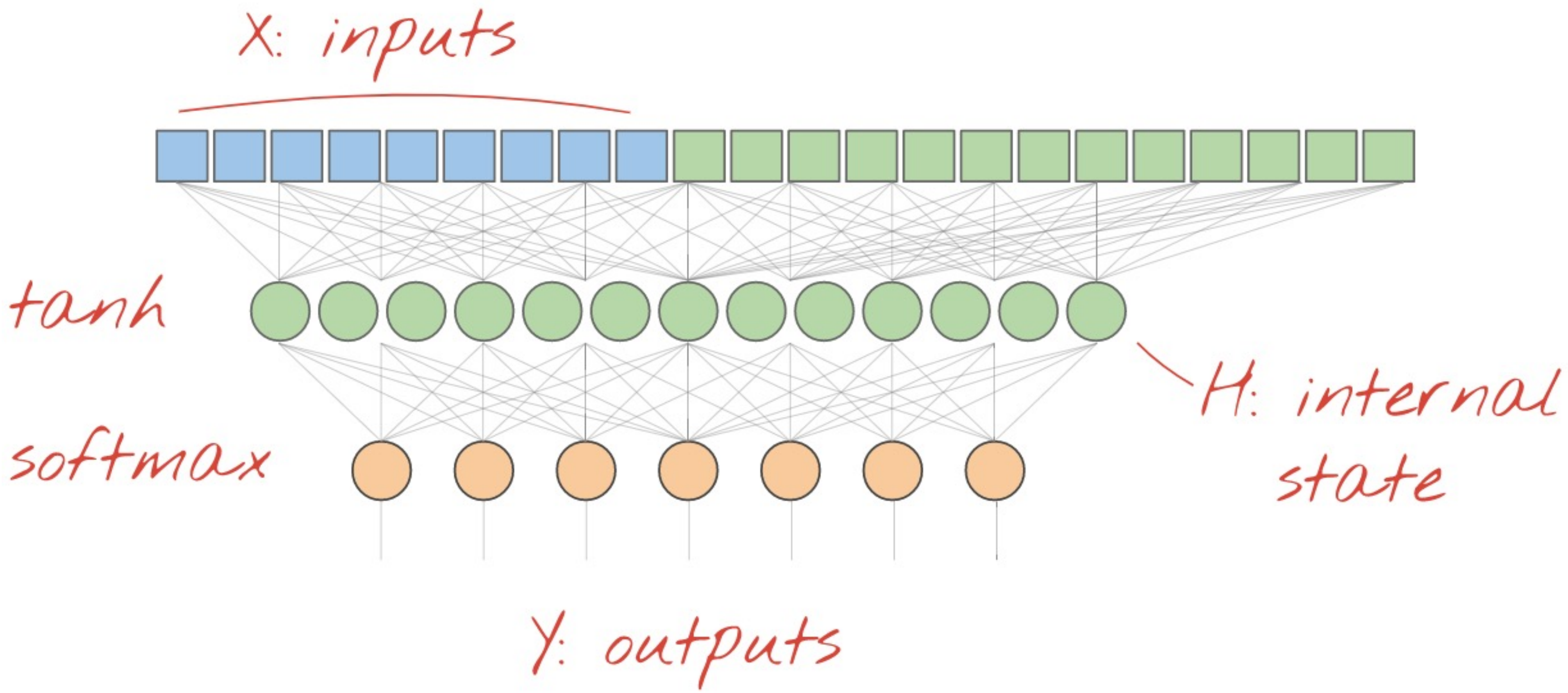
Recurrent neural network
1
0
0
Food
0
1
Weather
Add
Non-Linear
Merge
0
1
0
0
1
0

Example


Recurrent neural network
Food
Weather
Add
Non-Linear
Merge

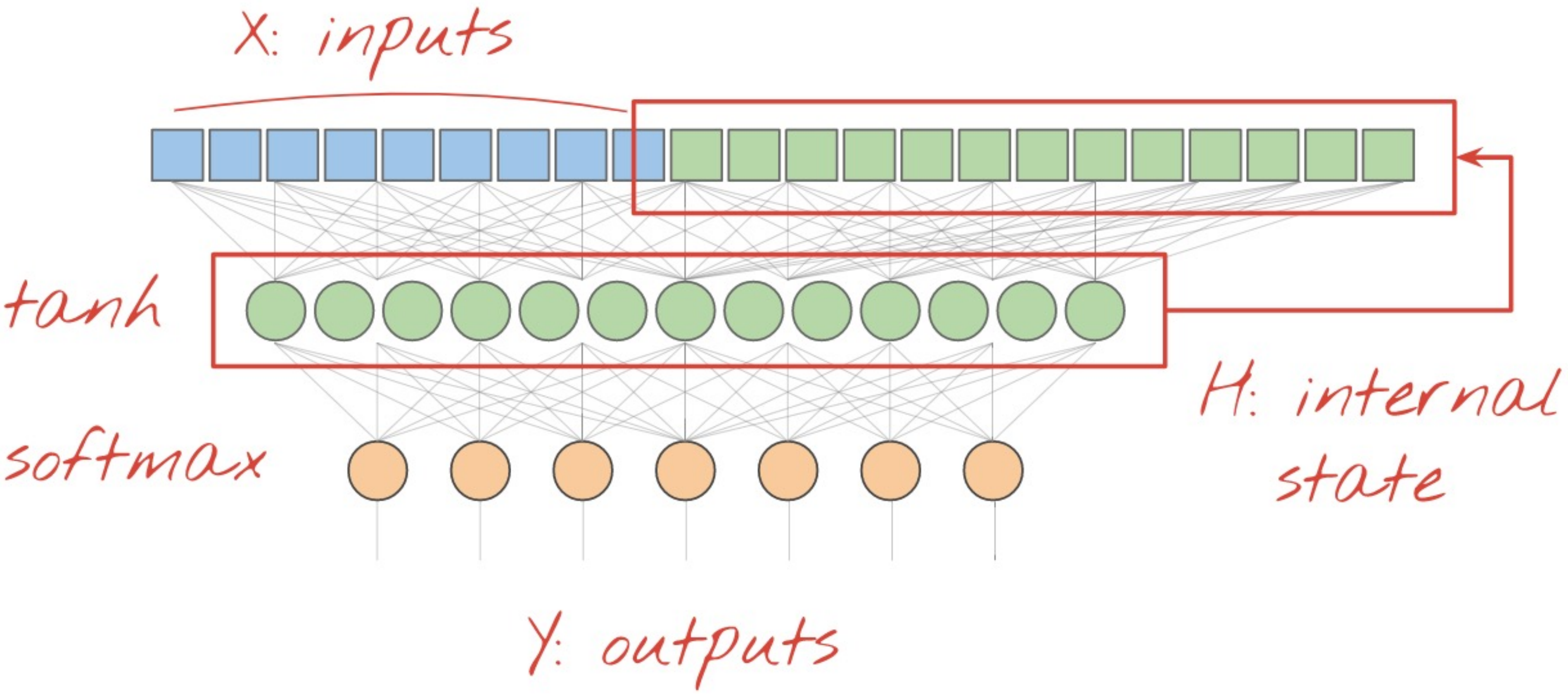
RNN

Recurrent neural network



RNN

Recurrent neural network



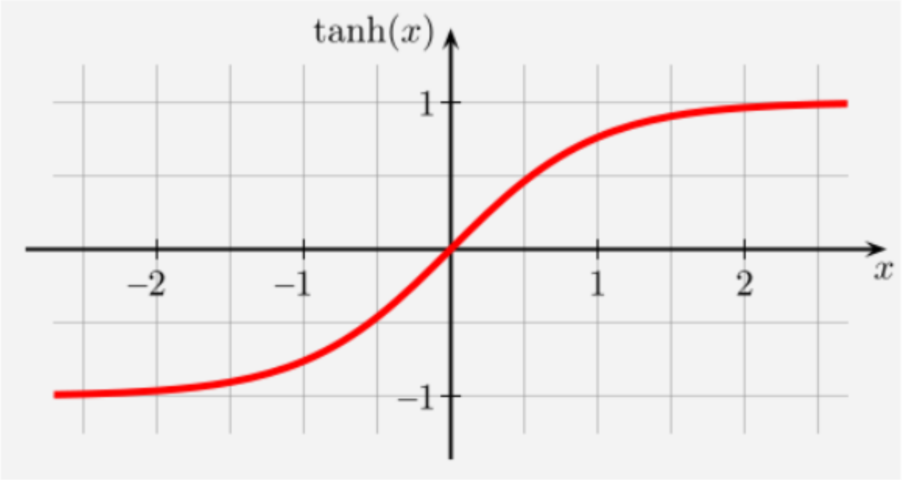
RNN

concatenation

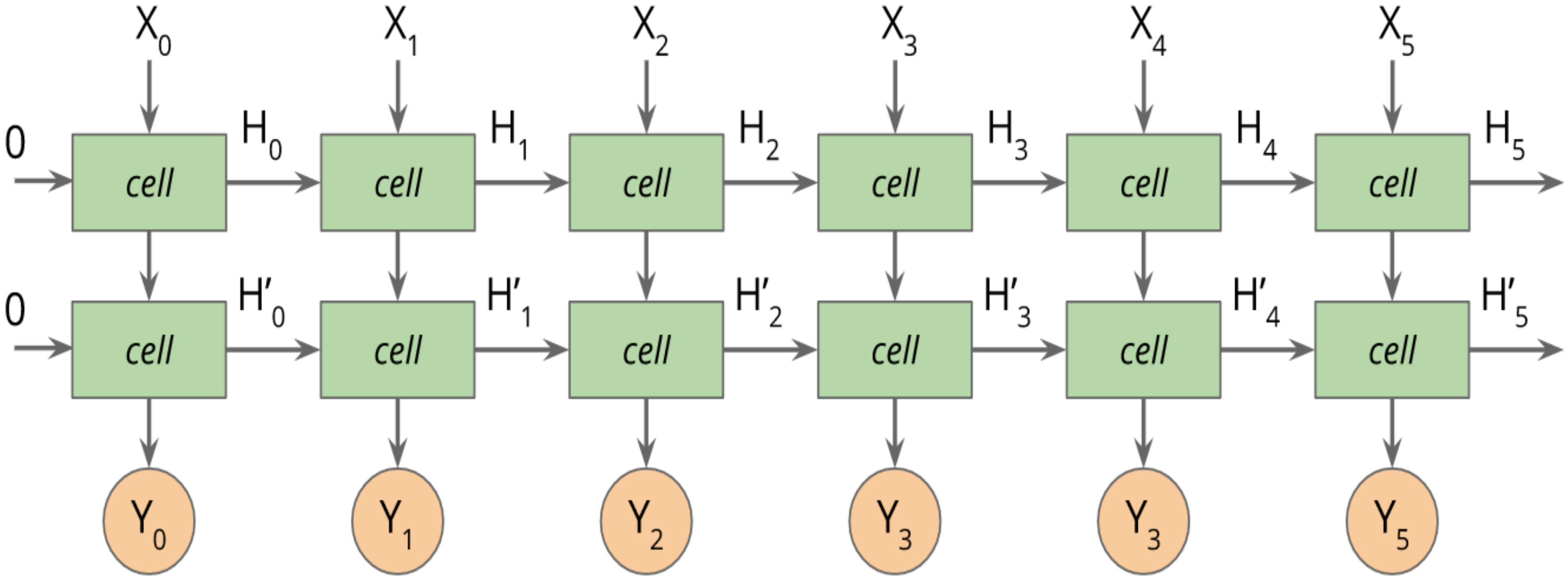
$$X = X_t \mid H_{t-1}$$

$$H_t = \tanh(X.W_H + b_H)$$

$$Y_t = \text{softmax}(H_t.W + b)$$



Deep RNN

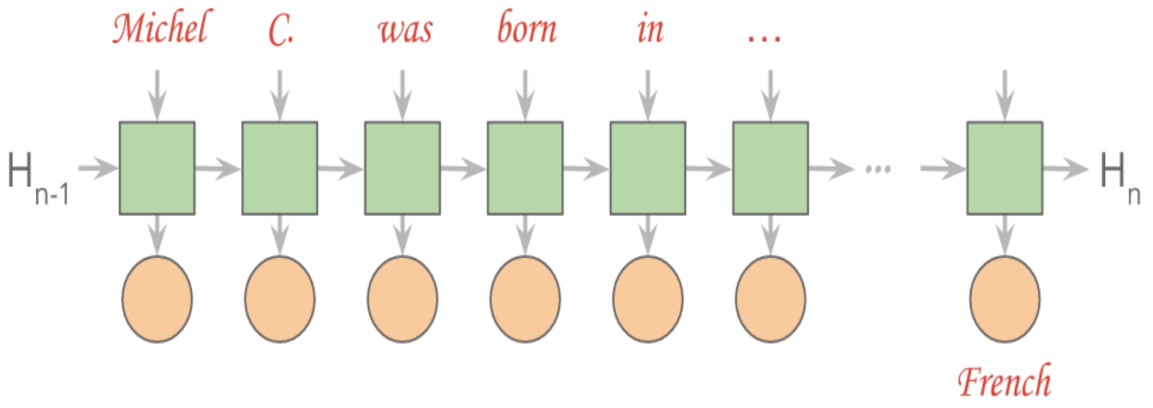


X0
X1
X2
X3
X4
X5
0
H0
H1
H2
H3
H4
H5
cell
cell
cell
cell
cell
cell
0
H'0
H'1
H'2
H'3
H'4
H'5
cell
cell
cell
cell
cell
cell
Y0
Y1
Y2
Y3
Y3
Y5

Long Term Dependency

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ?????

Short context → English, German, Russian, French ...

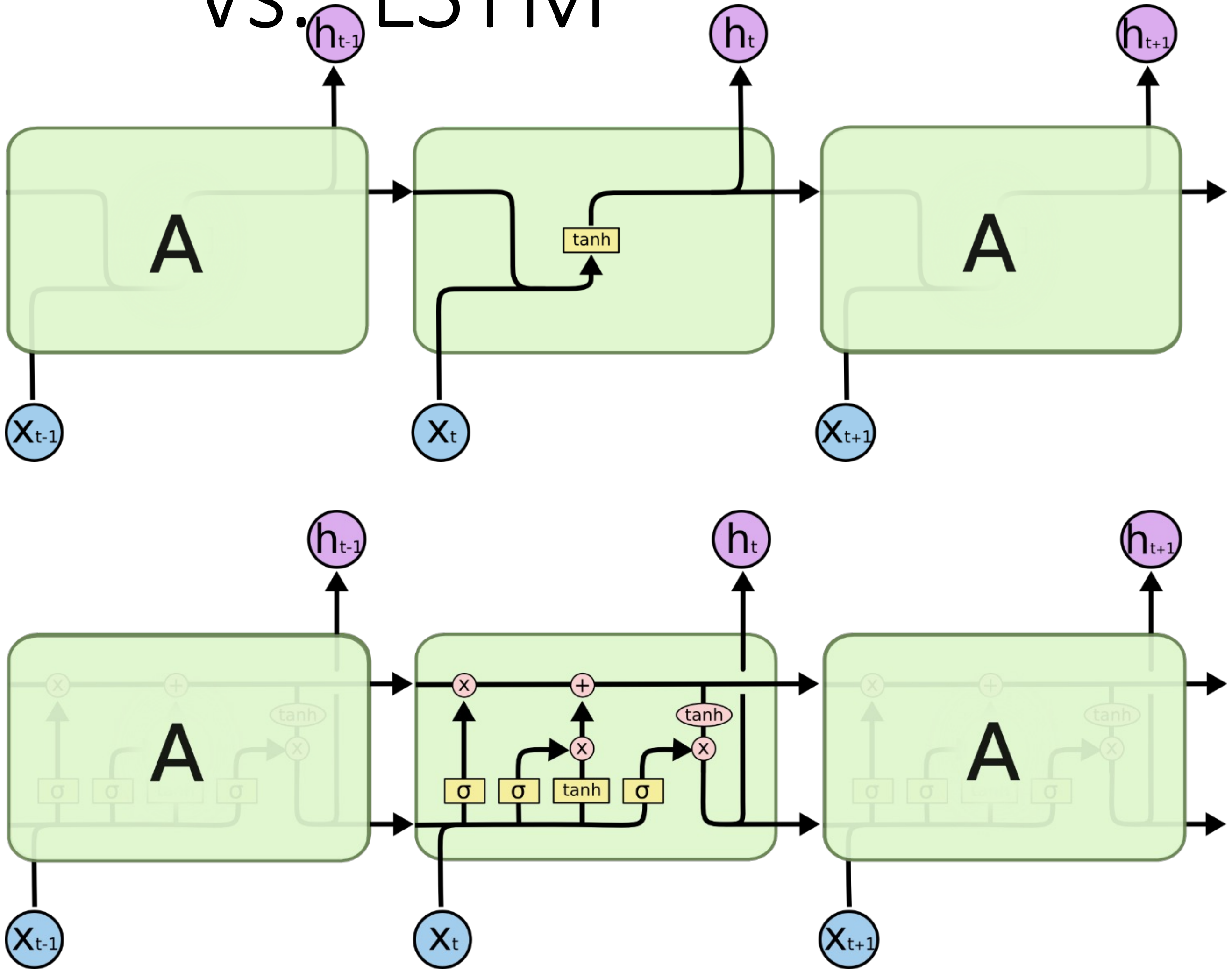


Long context → Problems...

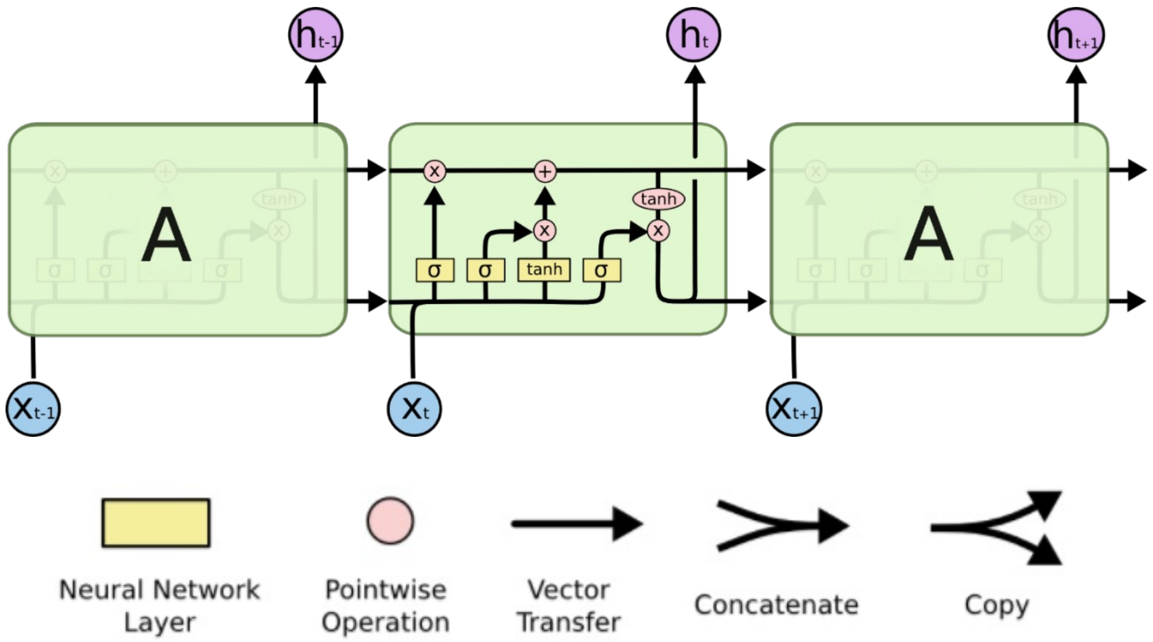
Simple RNN Vs. LSTM

RNN

LSTM

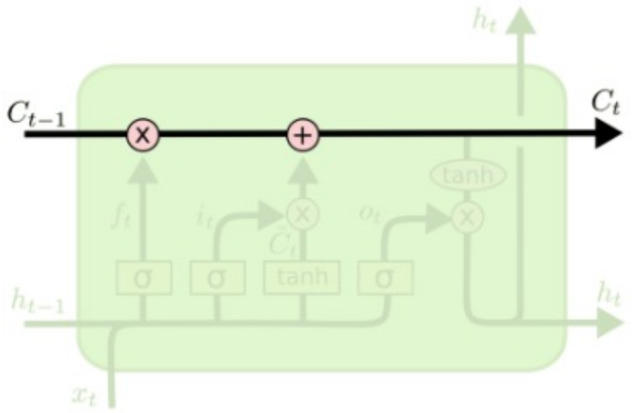


LSTM

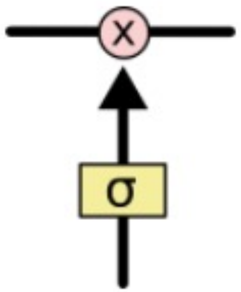


- Each line carries an entire vector, from the output of one node to the inputs of others
- The pink circles represent pointwise operations, like vector addition
- The yellow boxes are learned neural network layers.
- Lines merging denote concatenation
- Line forking denote its content being copied and the copies going to different locations

The Core Idea Behind LSTMs



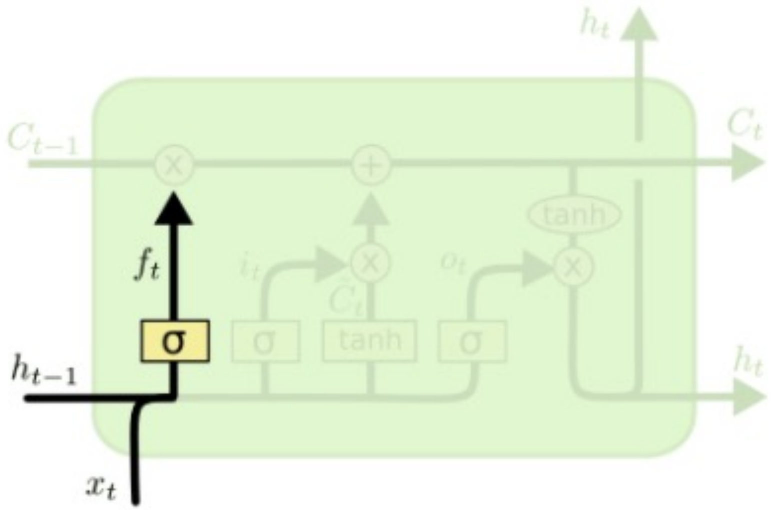
- The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions.



- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

- A value of zero means “let nothing through,” while a value of one means “let everything through!”

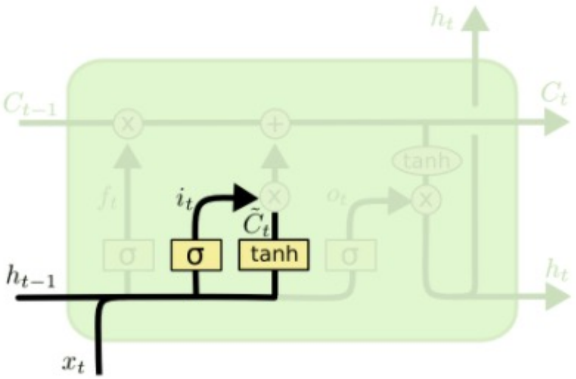
The Core Idea Behind LSTMs



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- The first step is to decide what information we’re going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.”
- A 1 represents “completely keep this” while a 0 represents “completely get rid of this.”

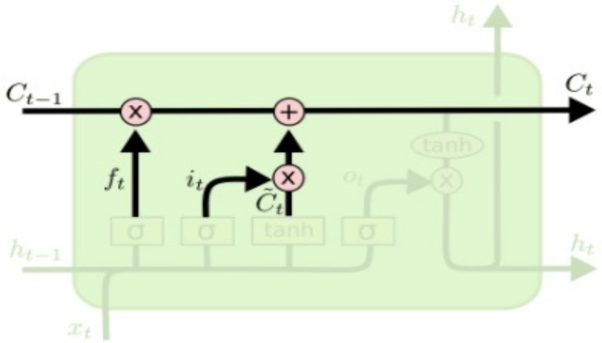
The Core Idea Behind LSTMs



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

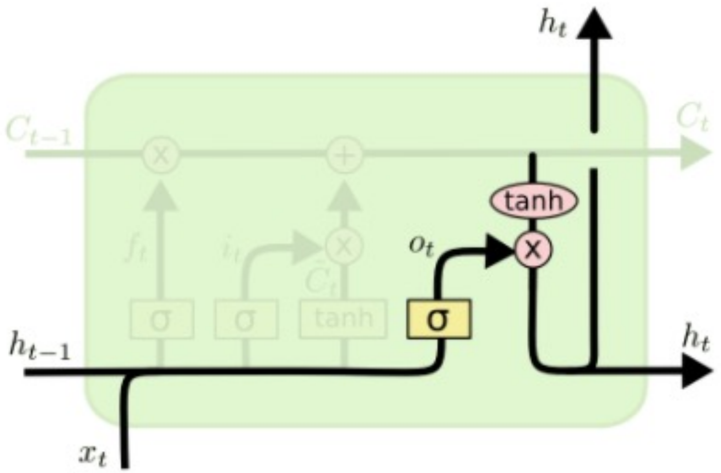
- It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.

- We multiply the old state by f_t , forgetting the things we decided to forget earlier. Then we add $i_t * \widetilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

The Core Idea Behind LSTMs



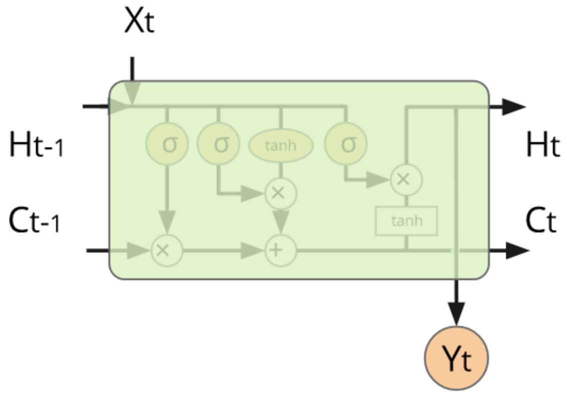
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

- Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

LSTM

		vector sizes
concatenate :	$X = X_t \mid H_{t-1}$	$p+n$
forget gate :	$f = \sigma(X.W_f + b_f)$	n
update gate :	$u = \sigma(X.W_u + b_u)$	n
result gate :	$r = \sigma(X.W_r + b_r)$	n
input :	$X' = \tanh(X.W_c + b_c)$	n
new C :	$C_t = f * C_{t-1} + u * X'$	n
new H :	$H_t = r * \tanh(C_t)$	n
output :	$Y_t = \mathrm{softmax}(H_t.W + b)$	m

$$\begin{bmatrix}1\\0\\0\\1\\2\\1\end{bmatrix} \xrightarrow{\text{Non-Linear}} \begin{bmatrix}0\\0\\0\\0\\1\\0\end{bmatrix}$$



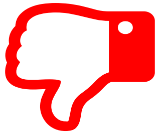
Recurrent Neural Networks (RNN)

Limitations of RNN networks

- **Length of the sentence**

I am from France

I am from France, and I'm a student in computer science at the University of Angers in France.

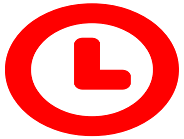


- **RNNs networks are recurrent unfortunately sentences are processed sequentially word by word**

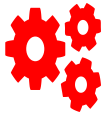
NN block

↓

I am from France



Time



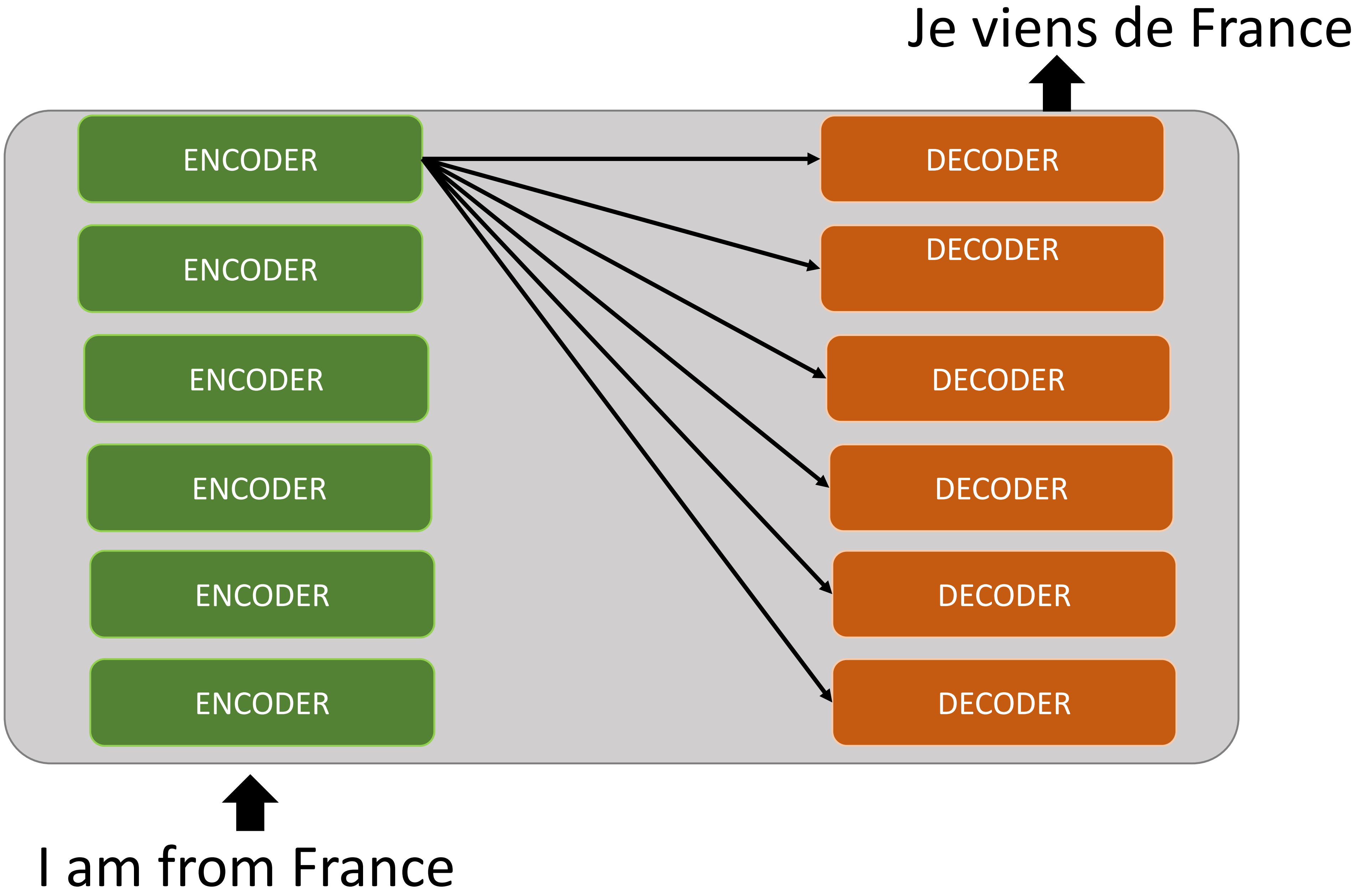
Calculation and machine power

Transformer architecture

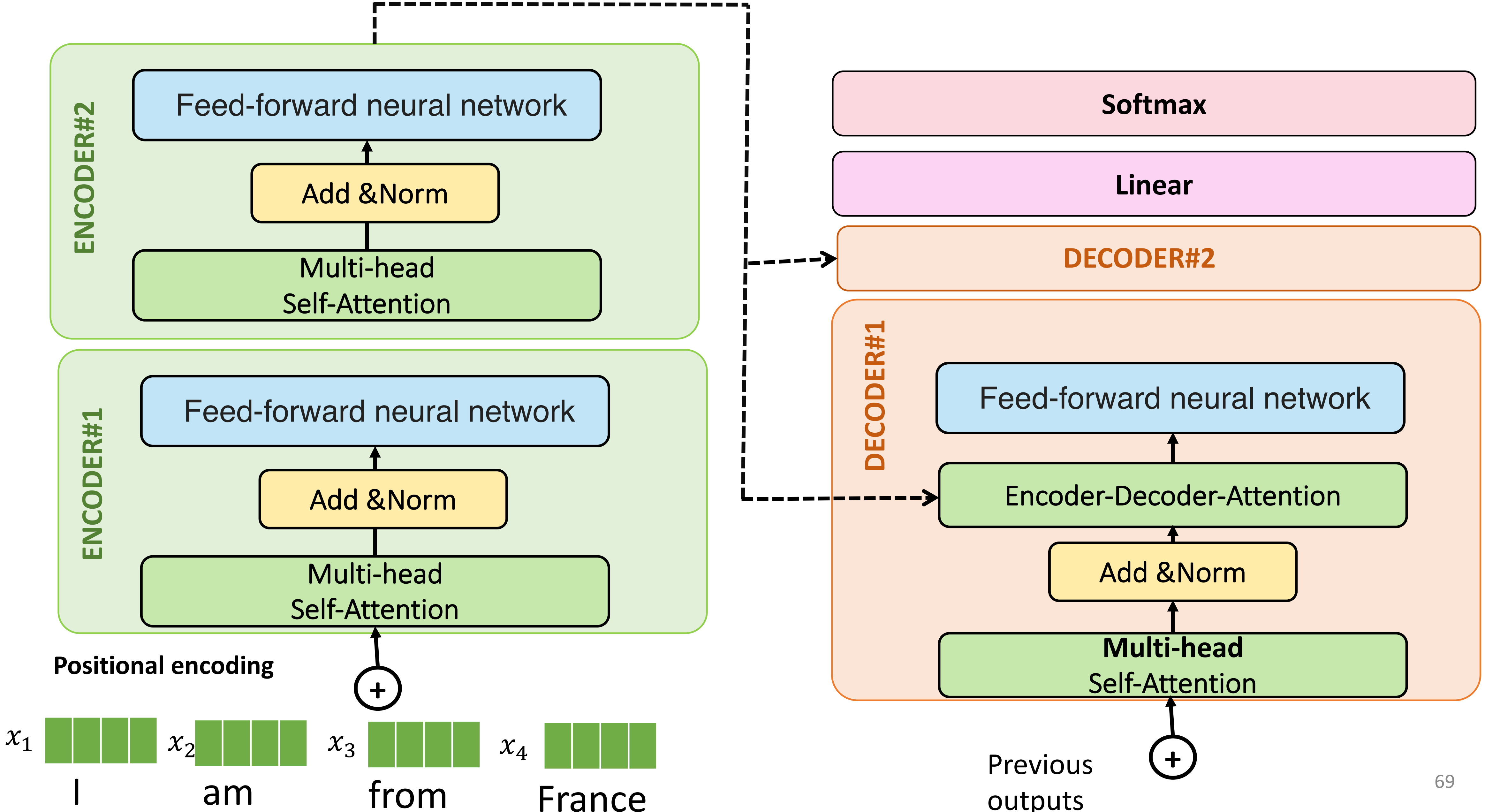
While processing a word, Attention enables the model to focus on other words in the input that are closely related to that word.

I am from France and I’m PhD student in computer science in University of Angers in France.

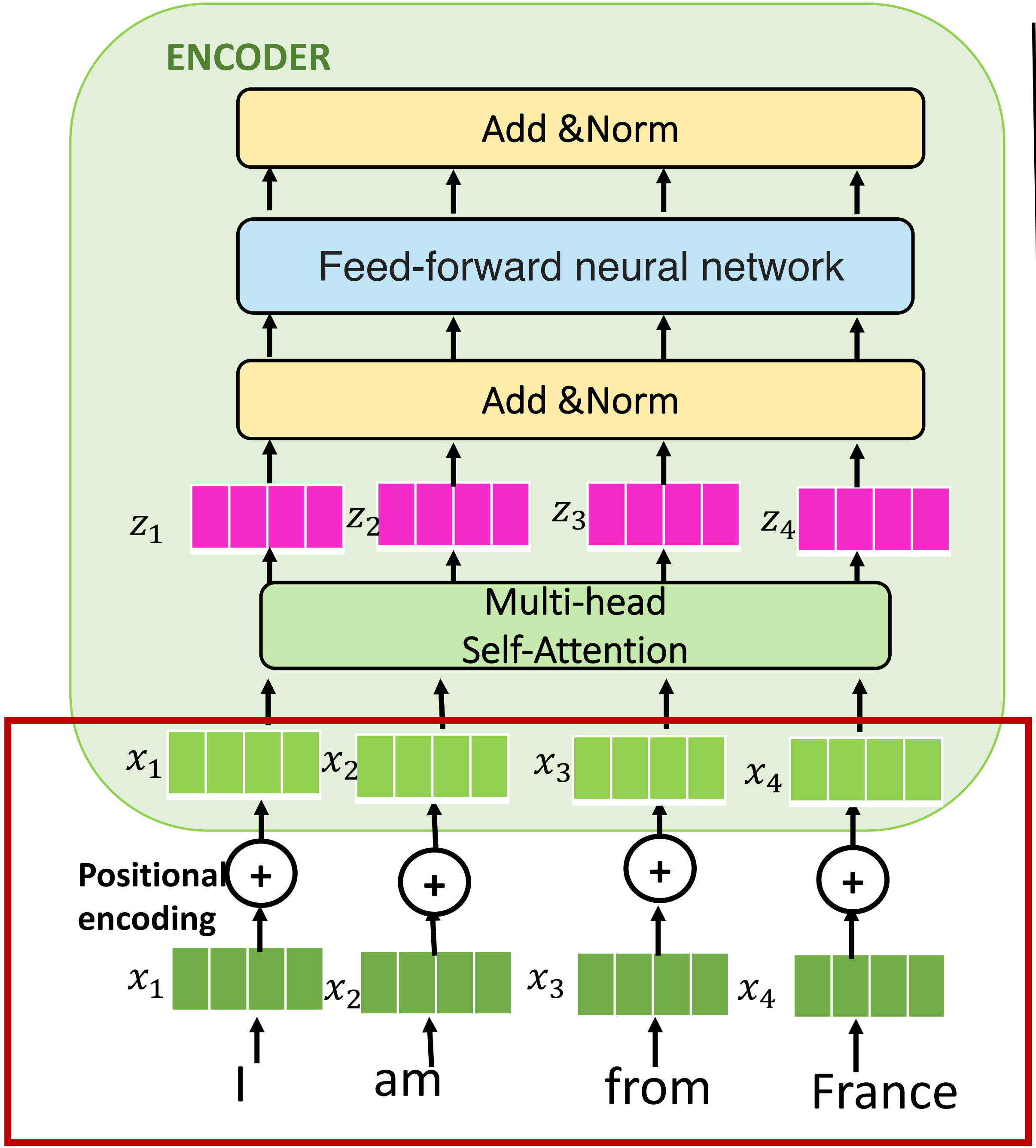
Architecture

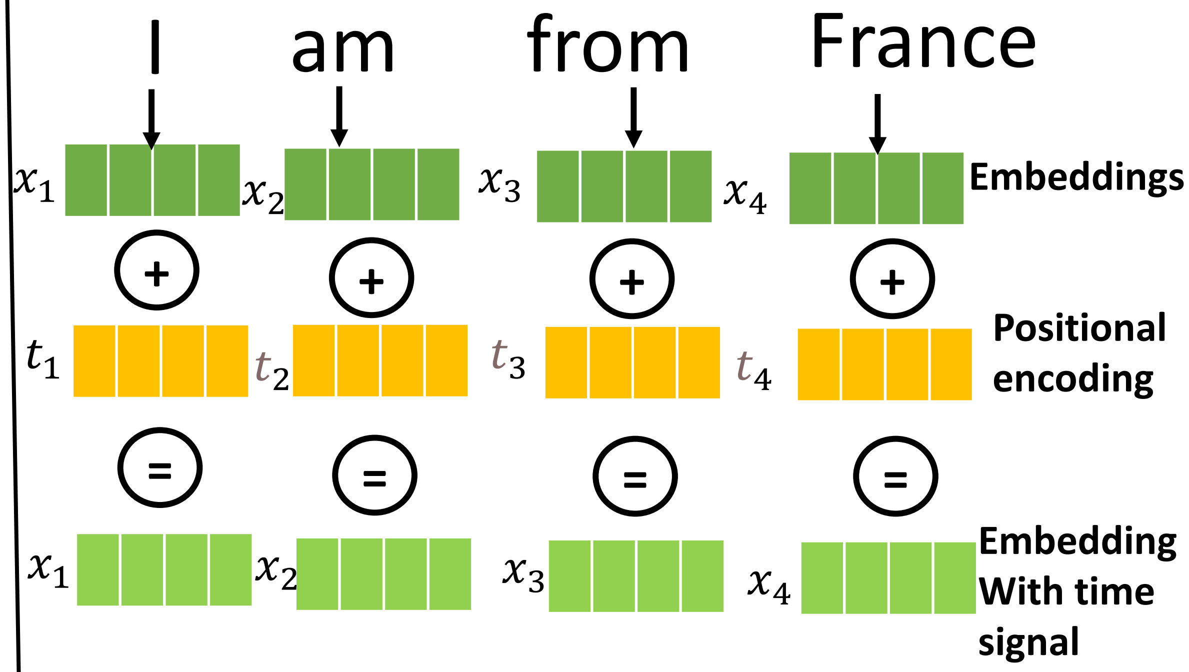


Architecture



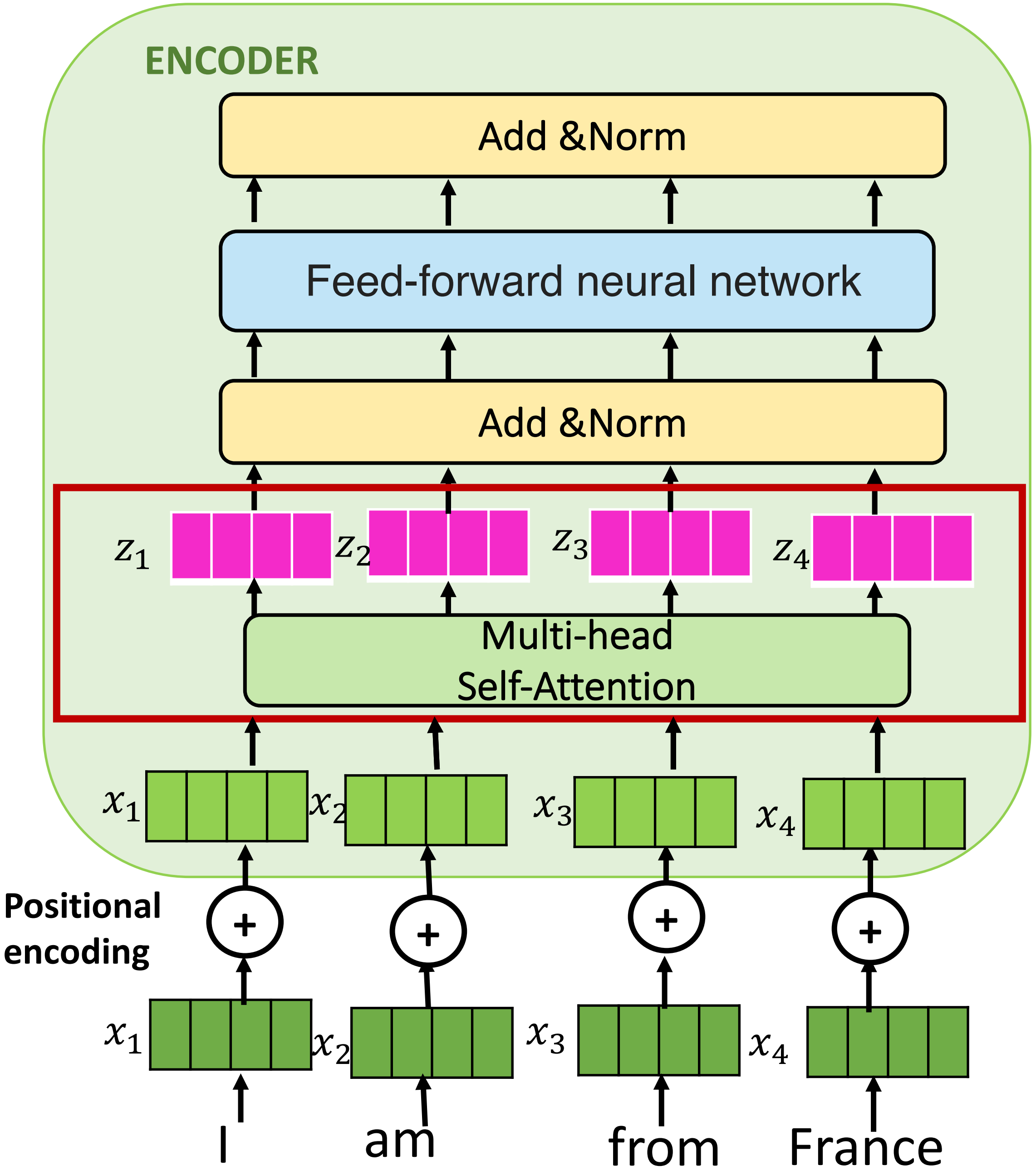
Architecture : Before encoder



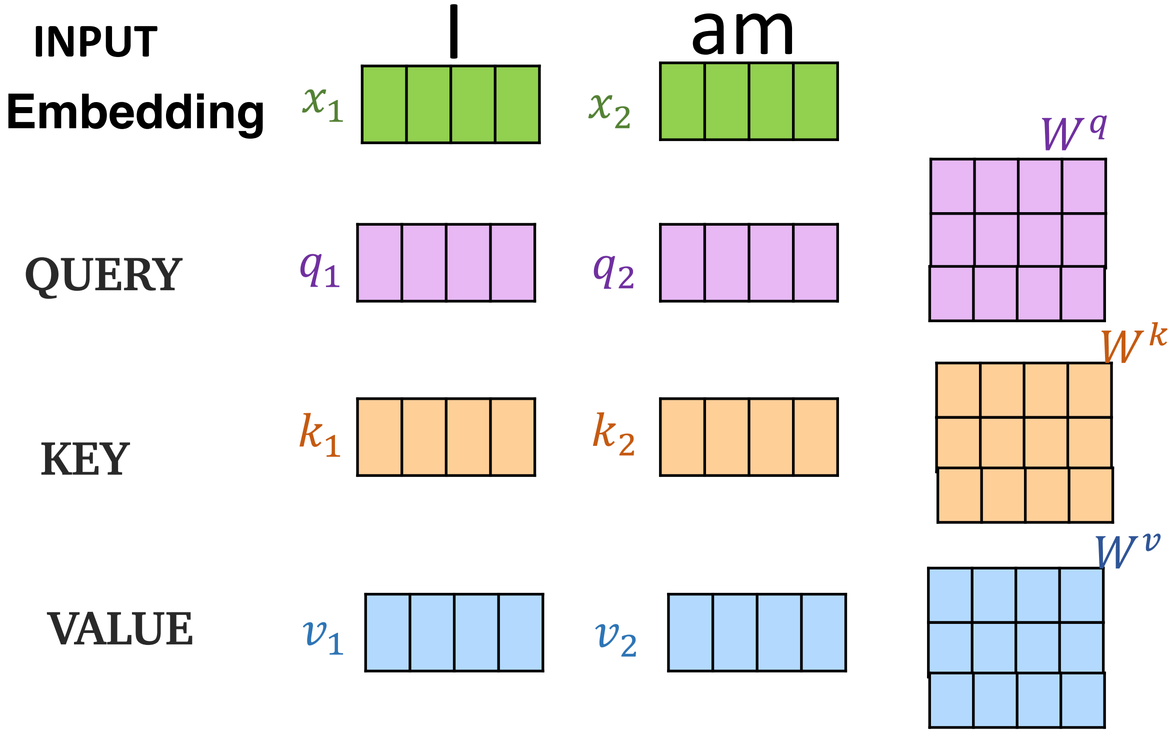


* To give the model a sense of the order of the words, we add positional encoding vectors –
* the values of which follow a specific pattern.

Architecture : Encoder



Self-attention



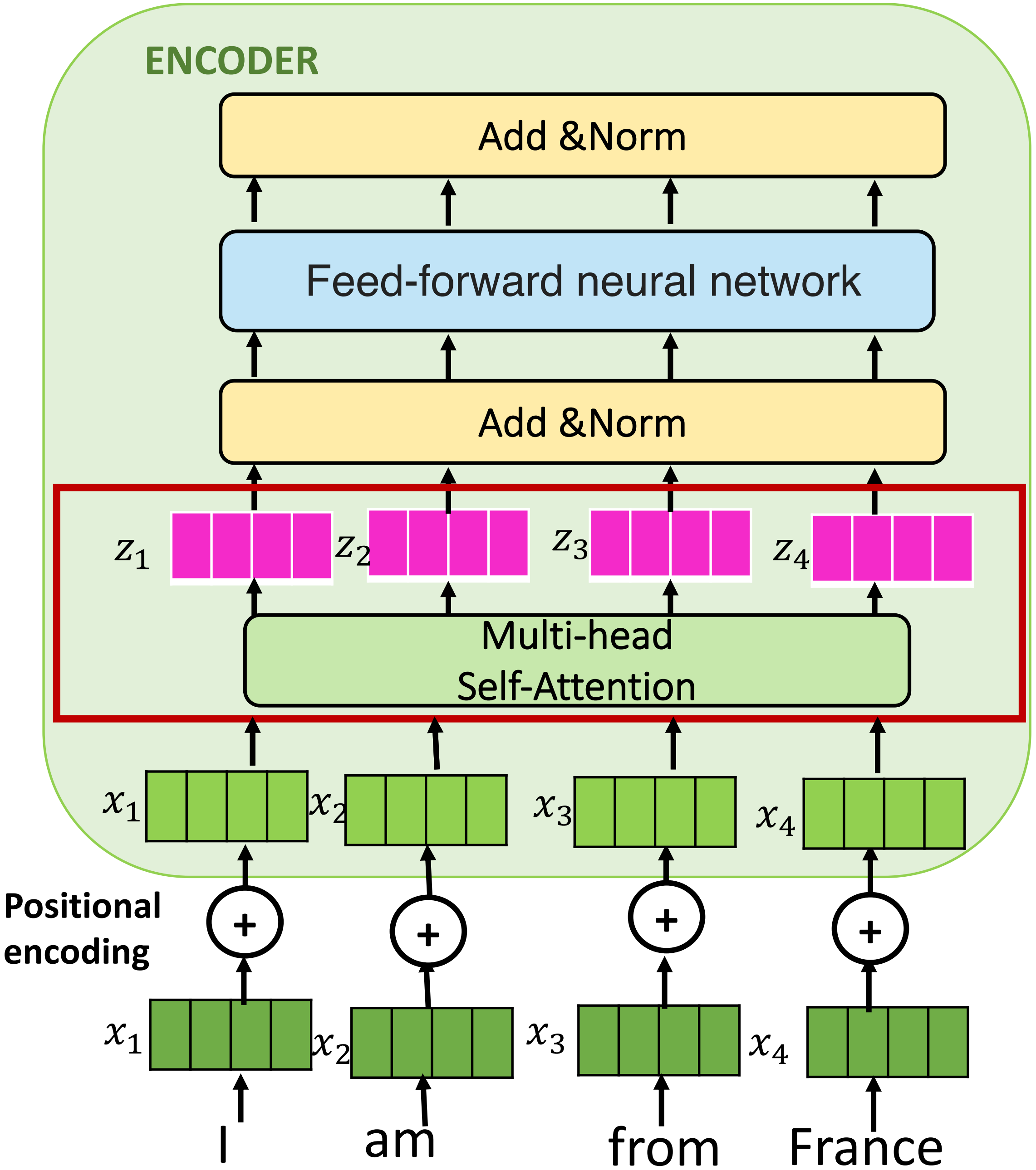
Step 1 :

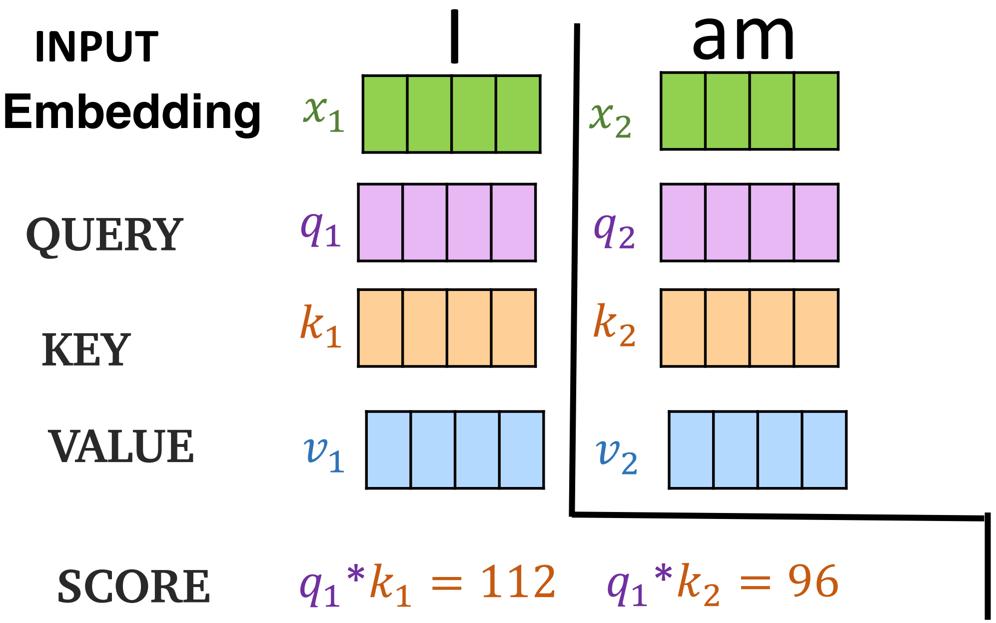
- Create a **Query** vector, a **Key** vector, and a **Value** vector.

- These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

Architecture : Encoder

Self-attention



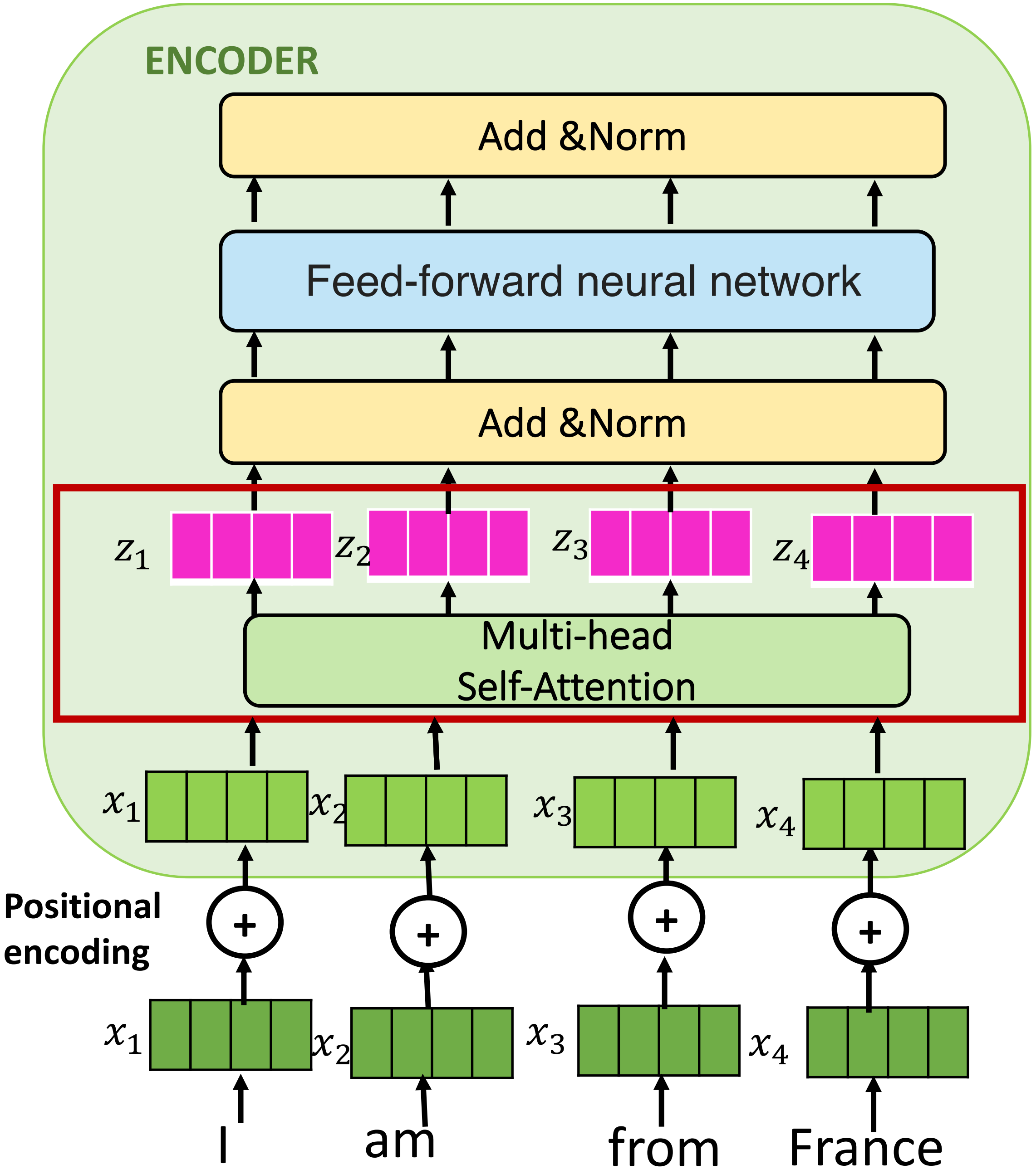


Step 2 :.

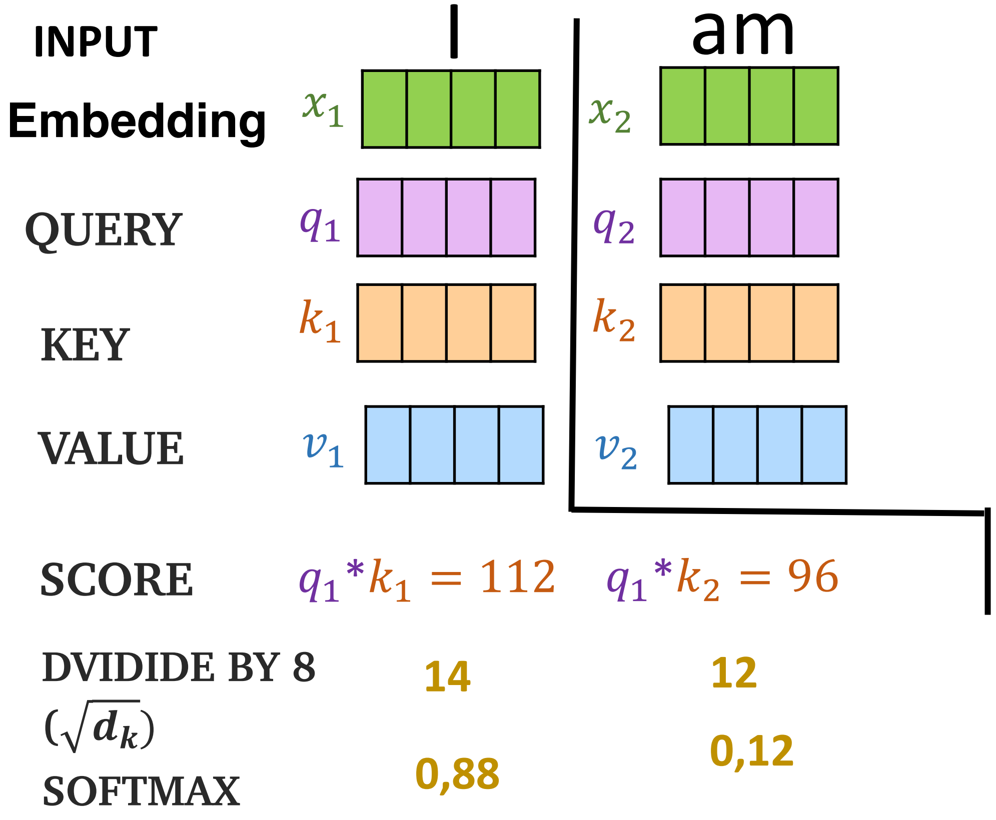
- Score each word of the input sentence against this word

- The **score** determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

Architecture : Encoder



Self-attention

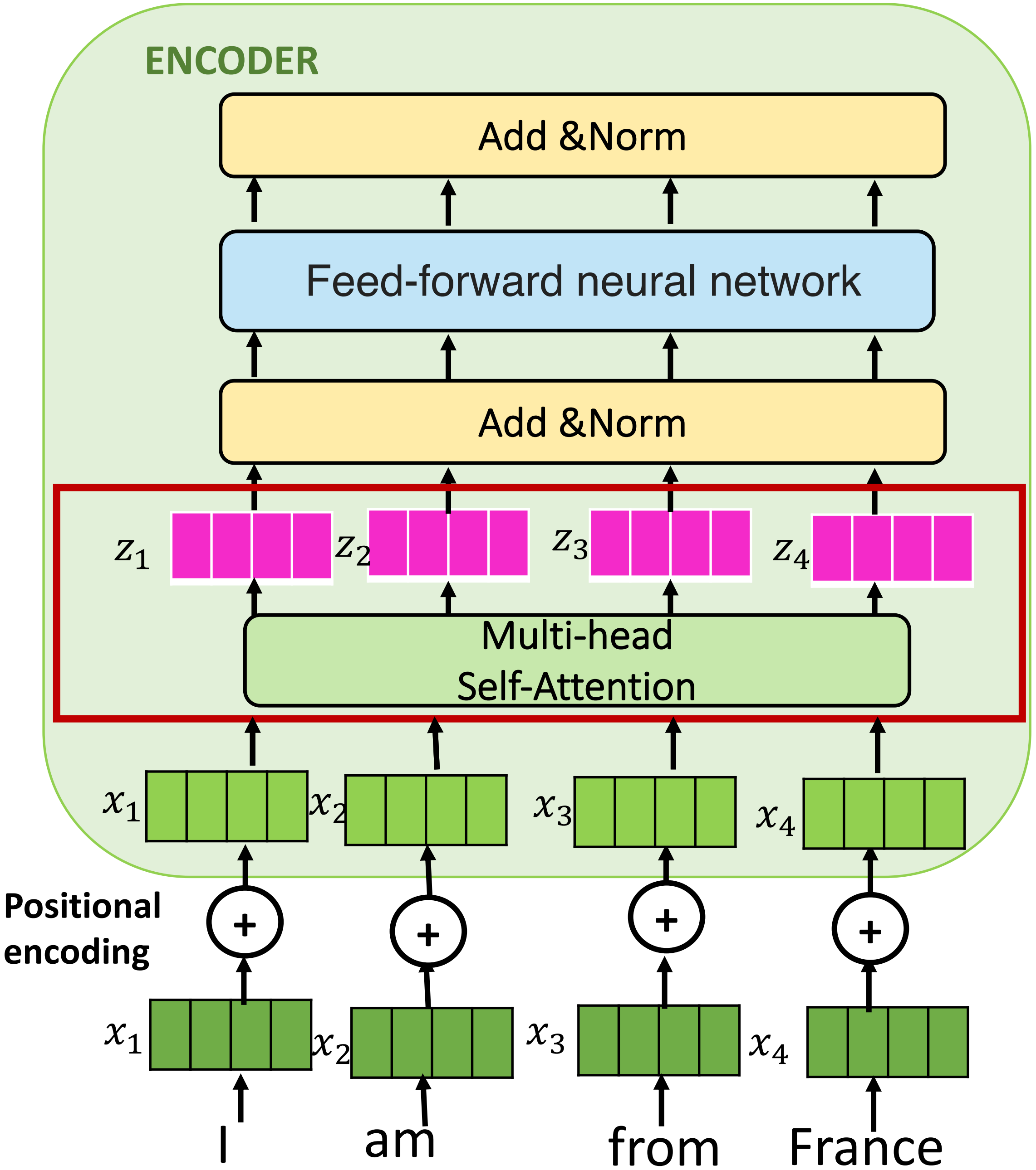


With d_k : dimension of x_n vector (64 in the original paper)

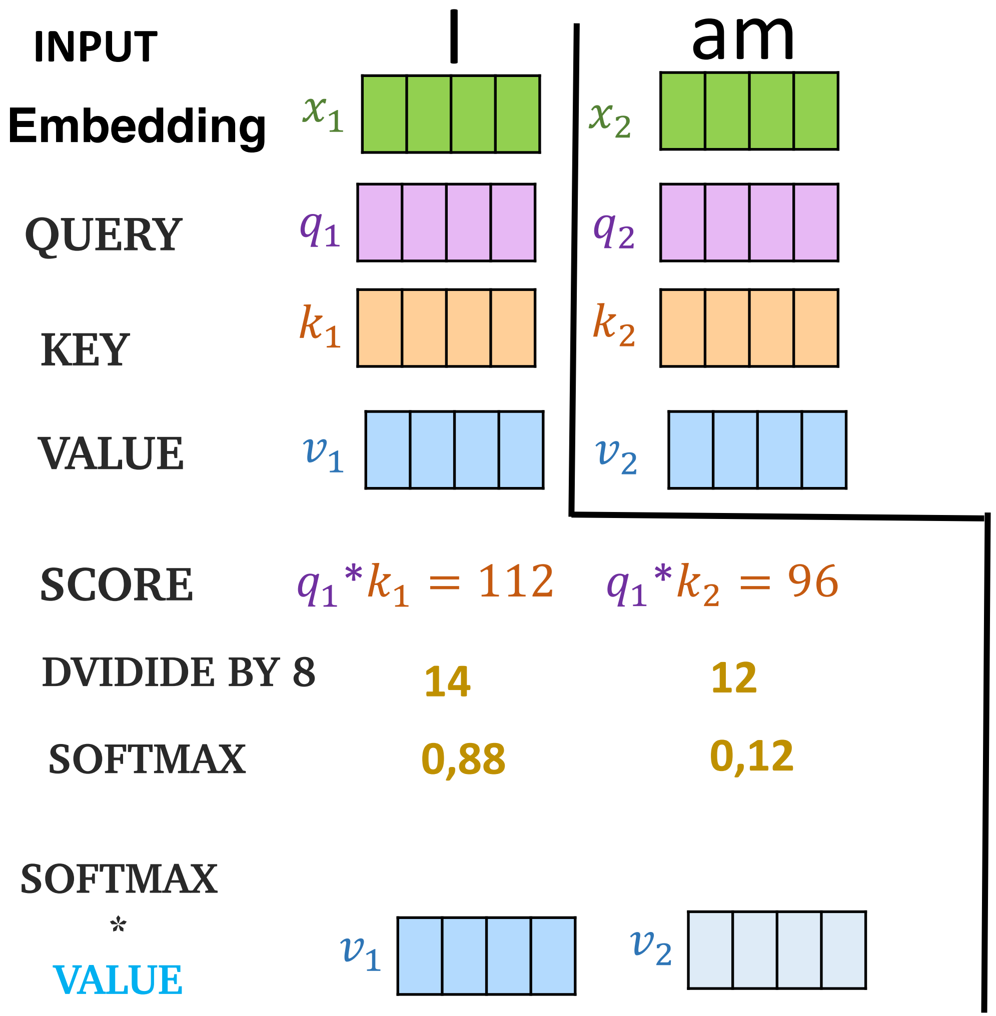
Step 3 & 4:

- **Divide** the scores by 8 then pass the result through a softmax operation.
- **Softmax** score determines how much each word will be expressed at this position.

Architecture : Encoder



Self-attention

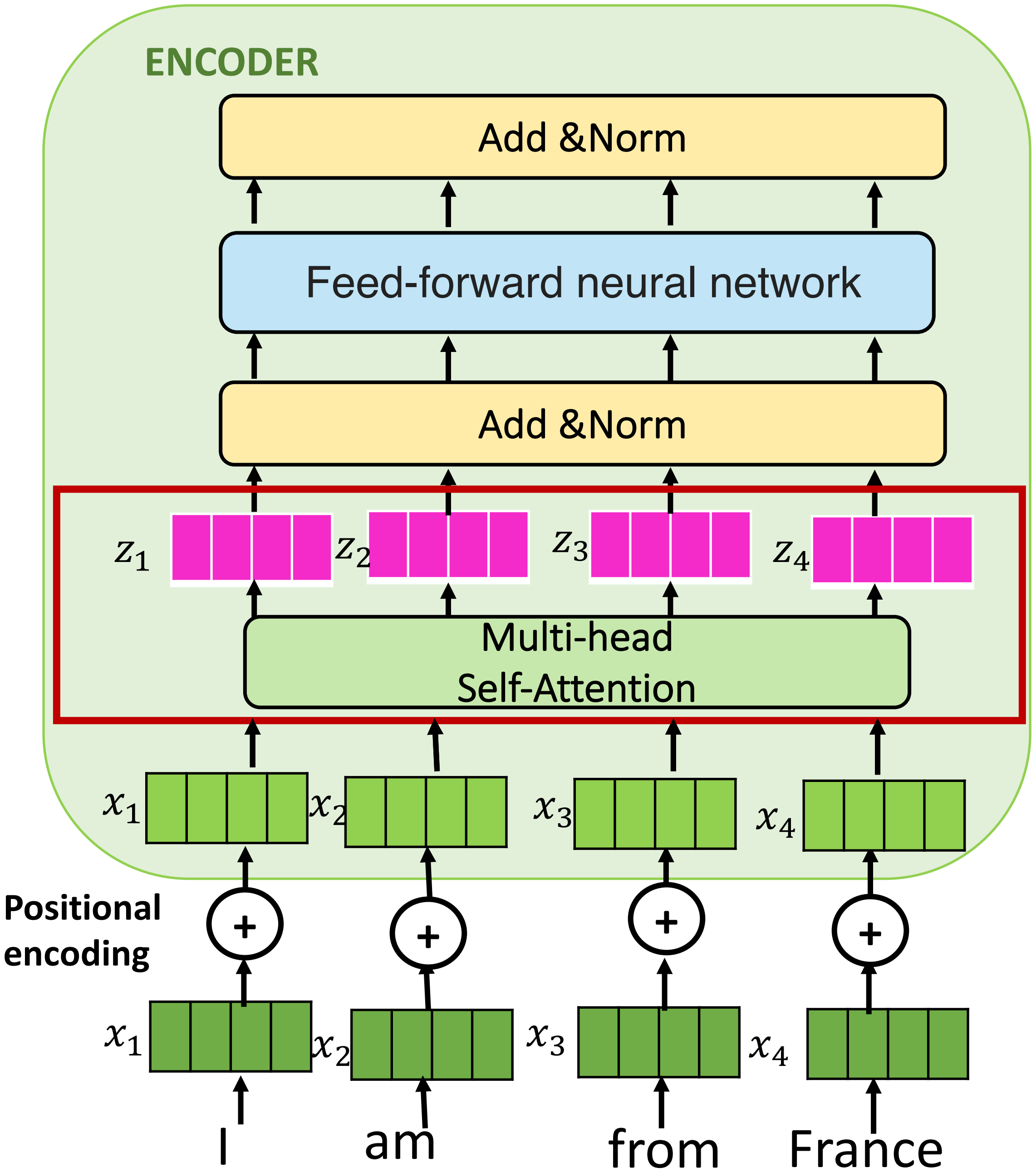


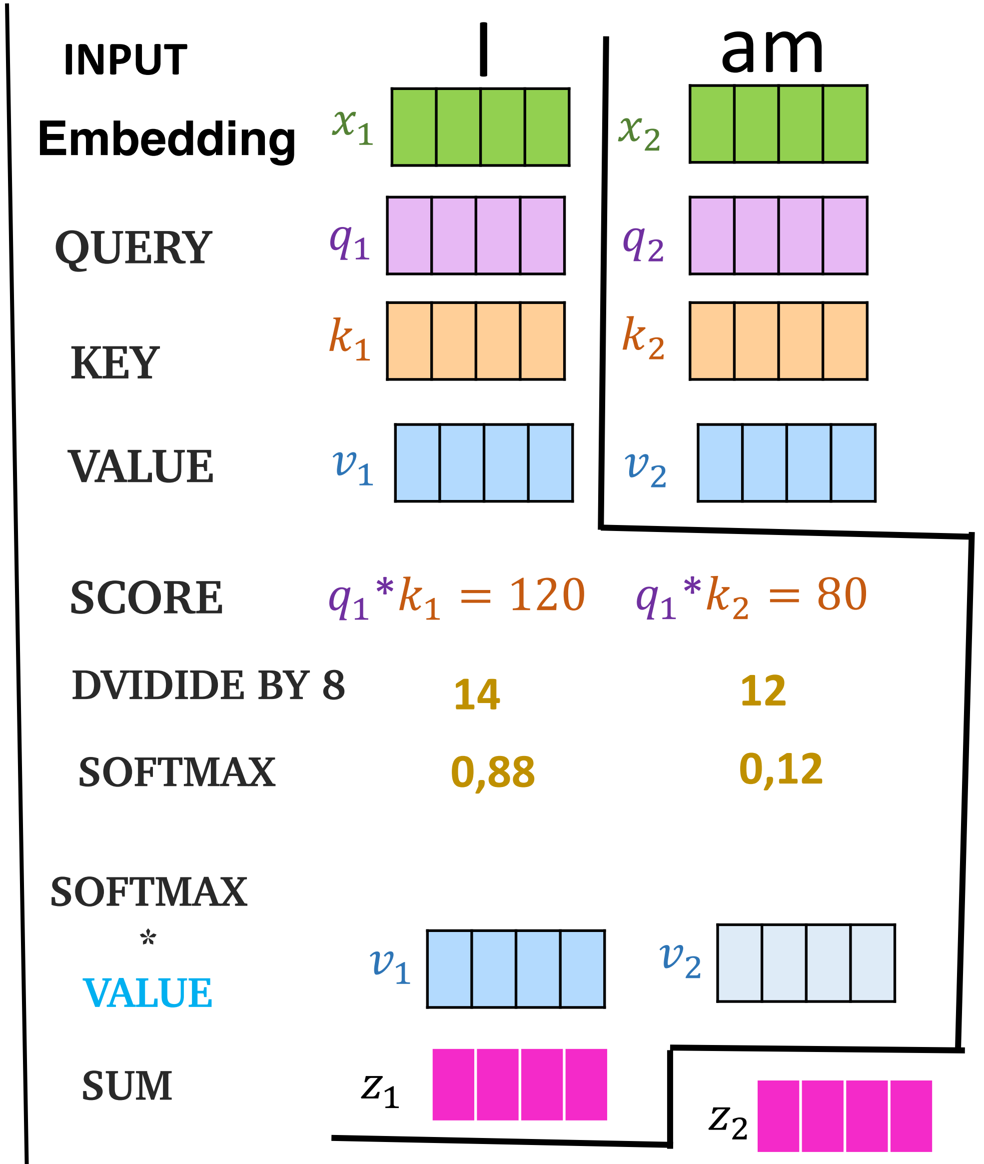
Step 5 :

- **Multiply** each value vector by the softmax score

Architecture : Encoder

Self-attention



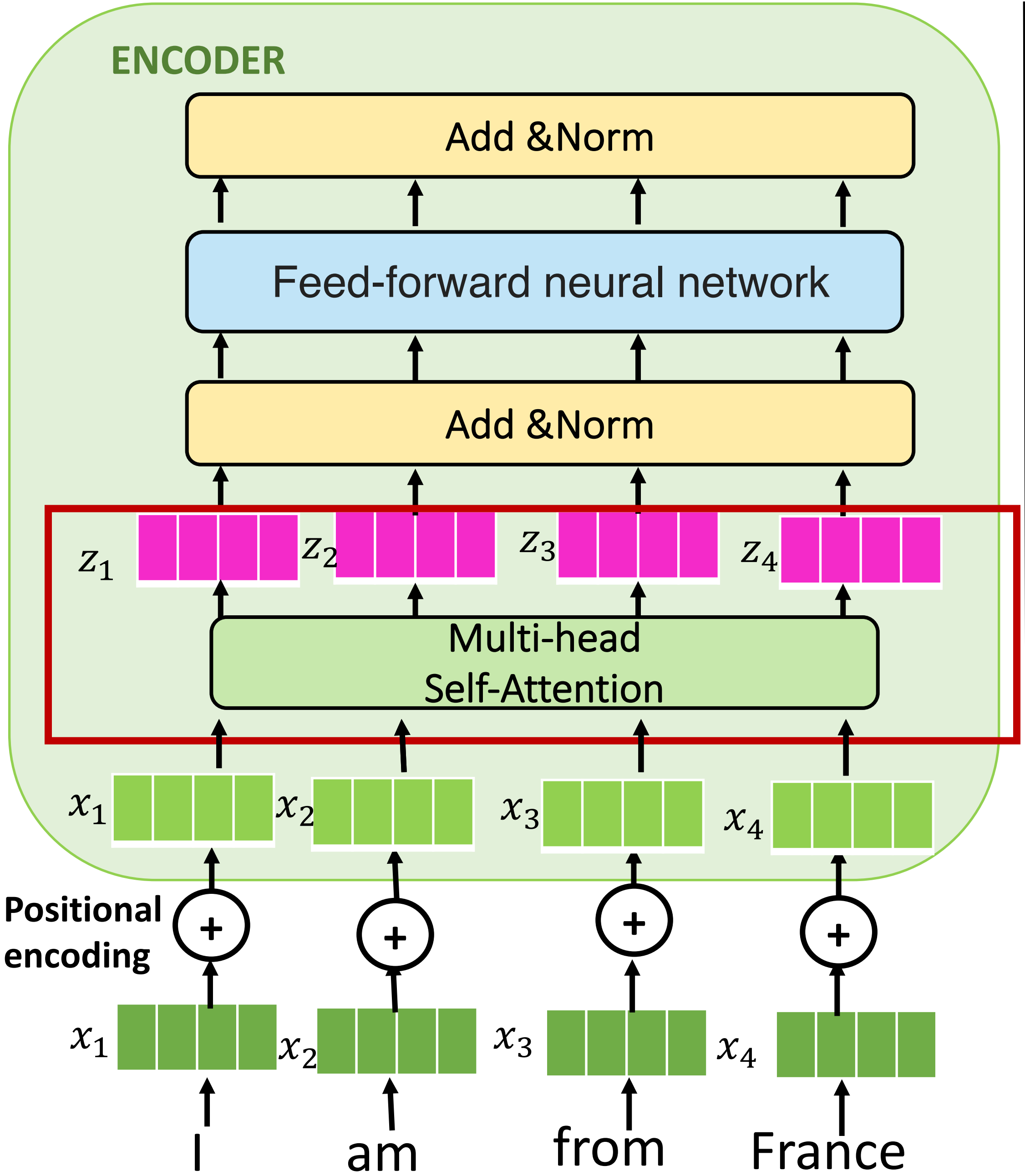


Step 6 :

- Sum up the weighted value vectors

Architecture : Encoder

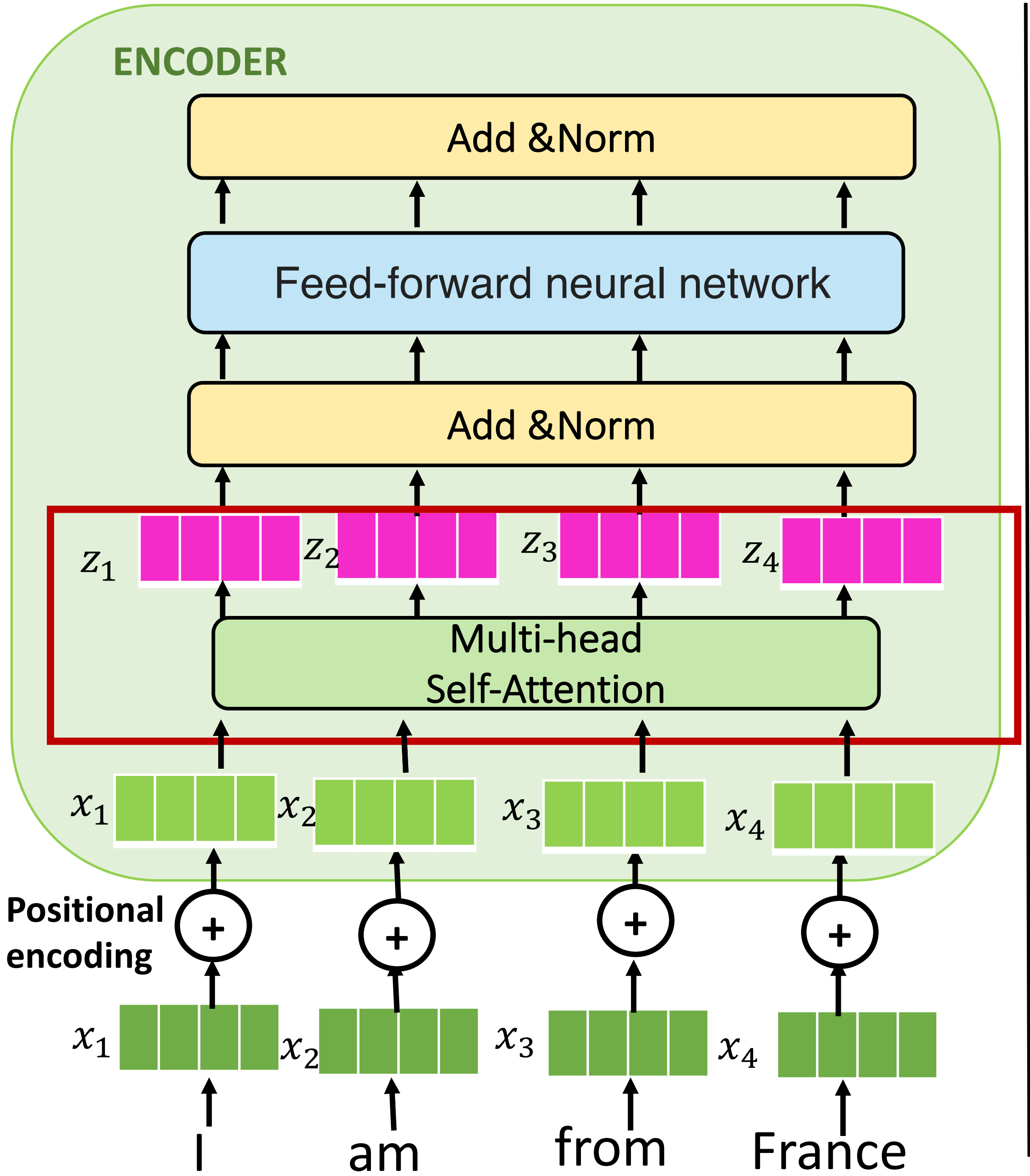
Multi-head of self-attention



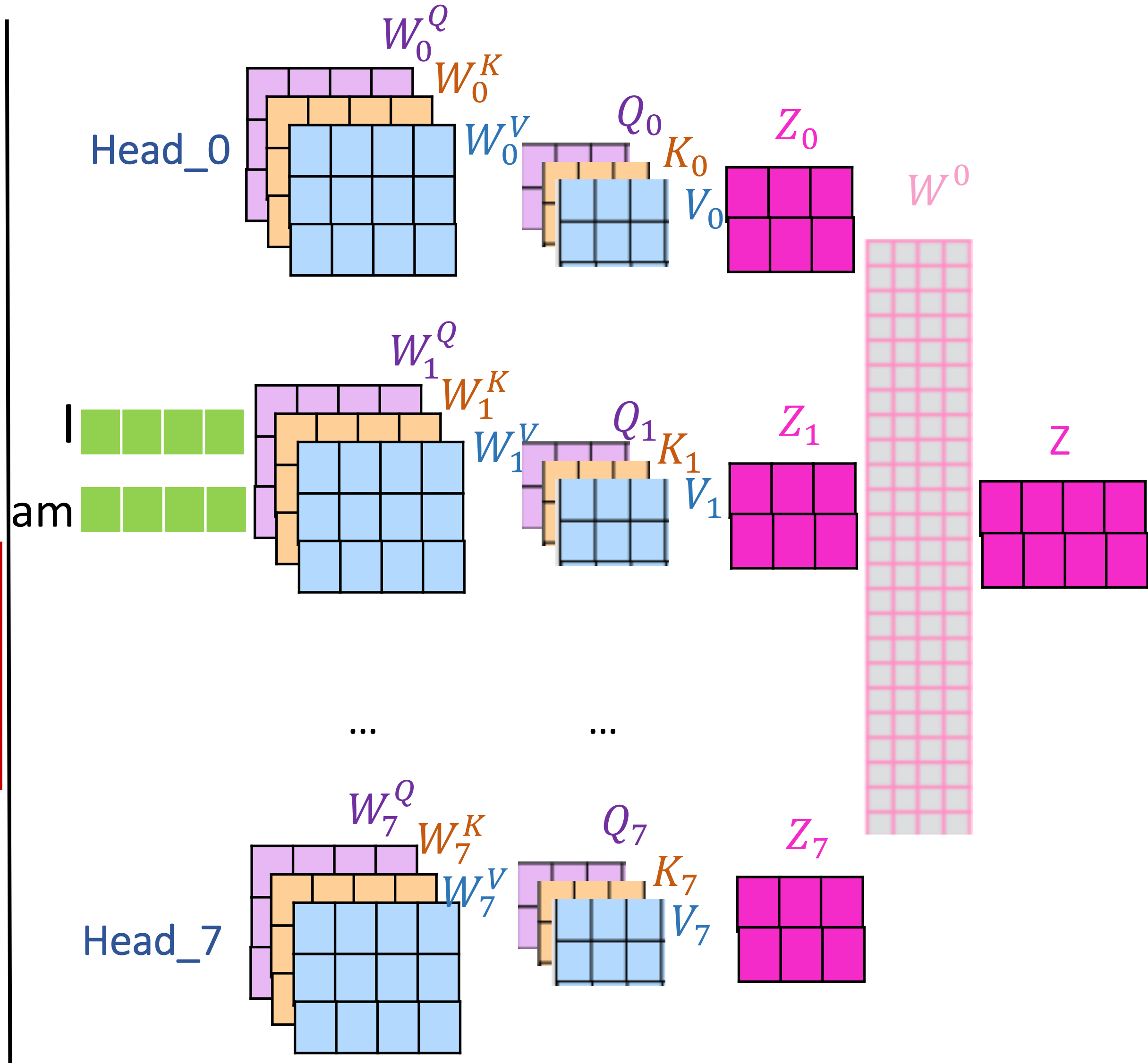
- Expands the model's ability to focus on different positions

- Gives the attention layer multiple "representation subspaces".

Architecture : Encoder

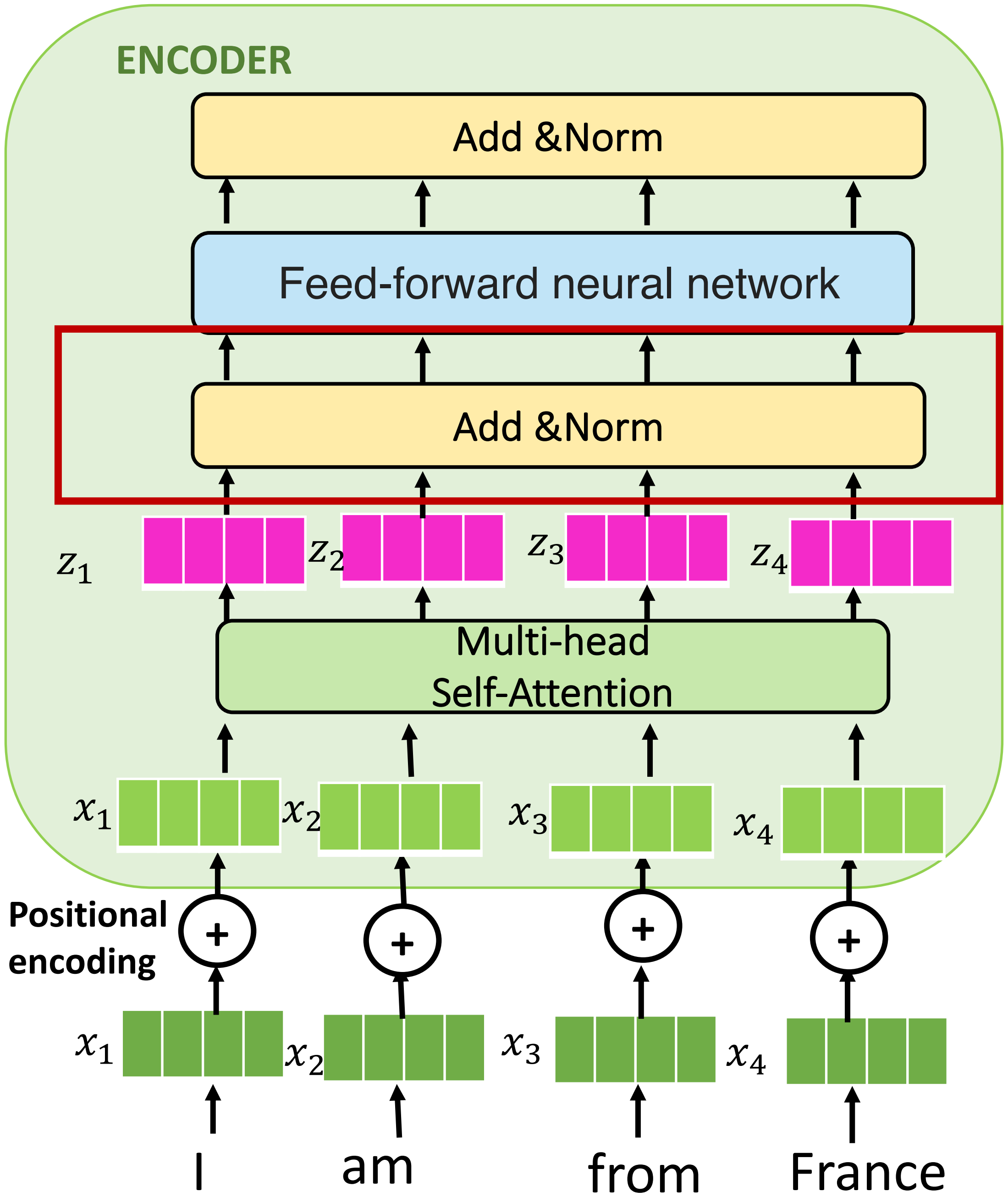


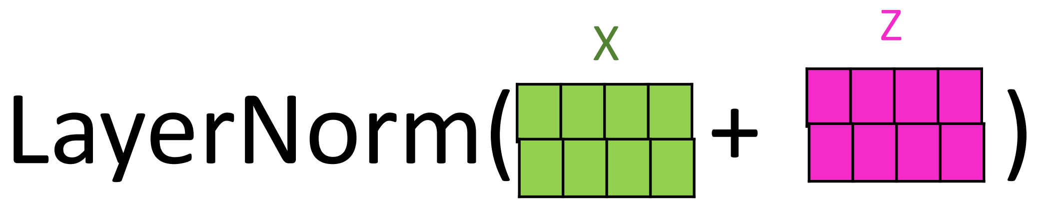
Multi-head of self-attention



Architecture : Encoder

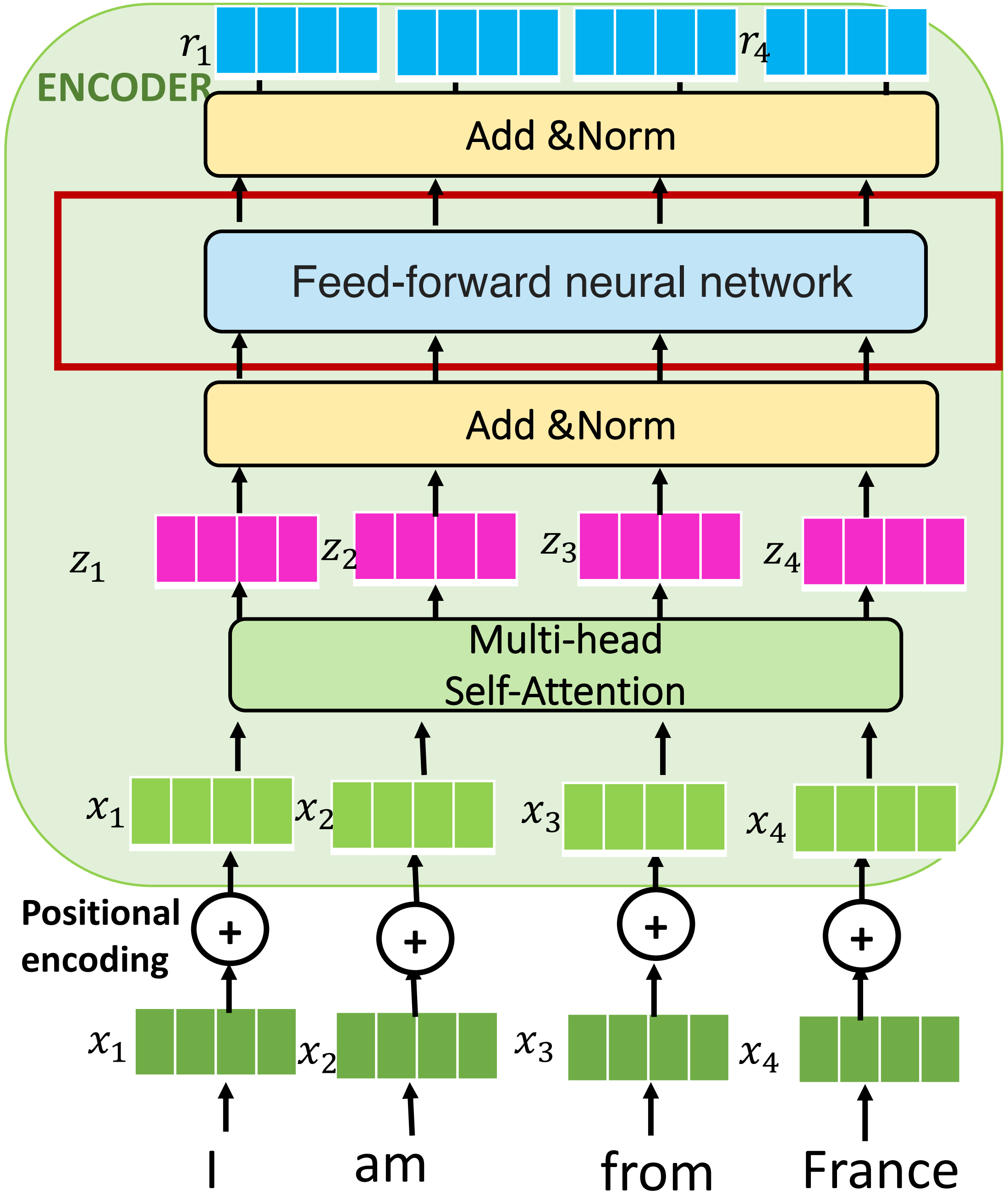
Add & Normalisation





Architecture : Encoder

Feed-forward neural network

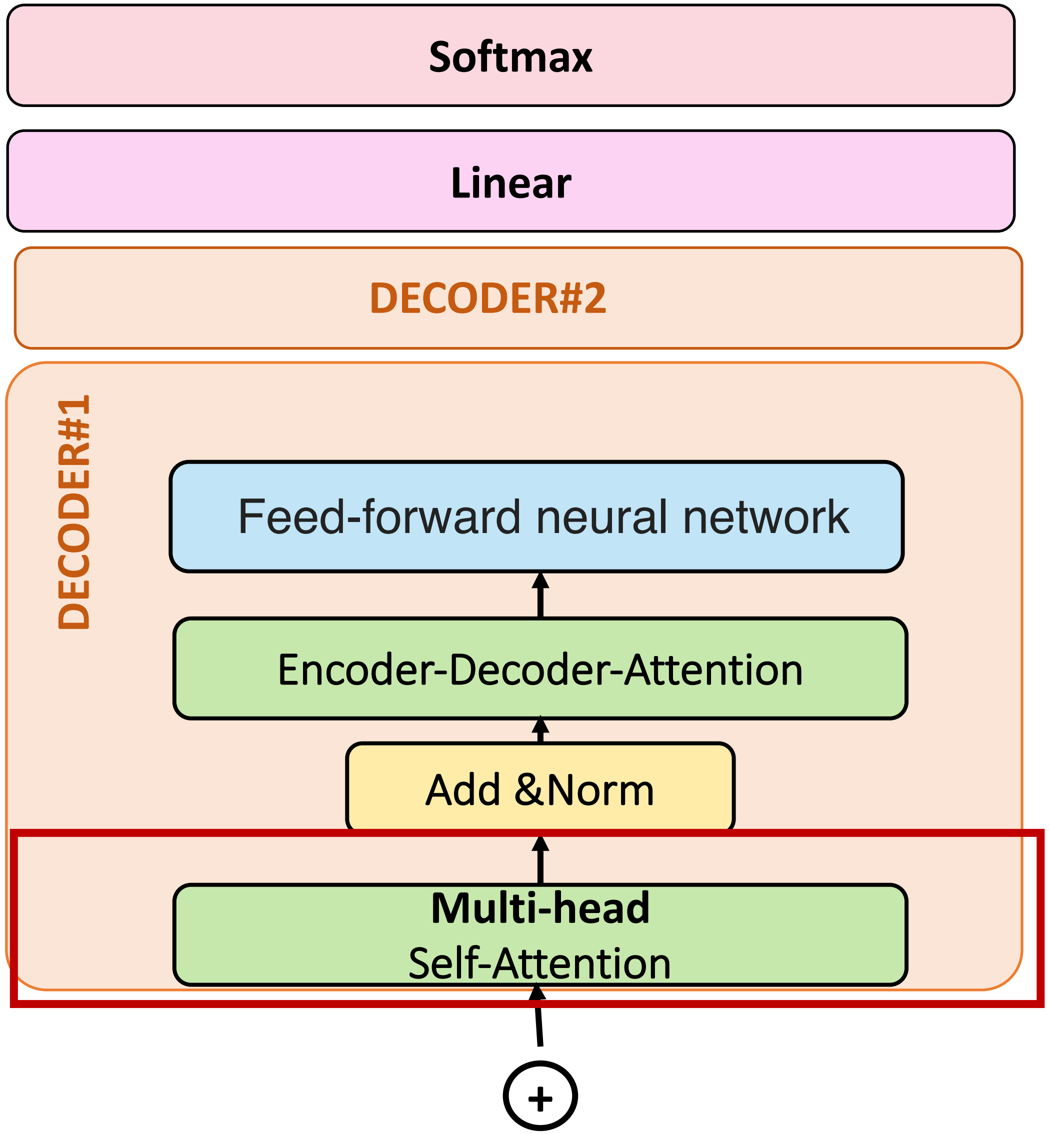


- **FFNs** are fully connected layers
- **FFN** is a position-wise network
- **FFN** contains two layers and applies a ReLu activation function

FFN(x) = max(0, xW1 + b1)W2 + b2

Architecture : DECODER

Multi-head of self-attention

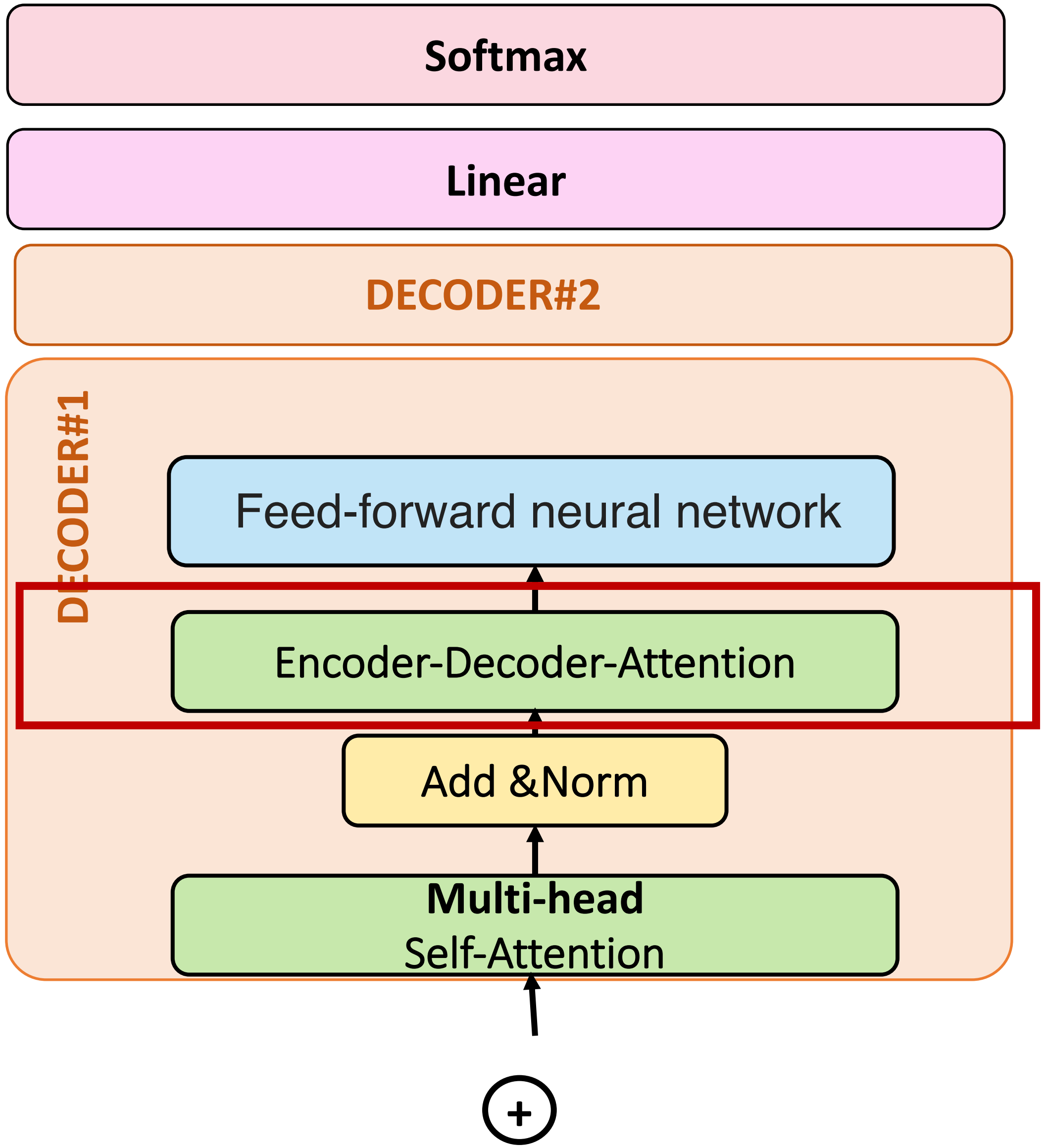


- In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence.

- This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

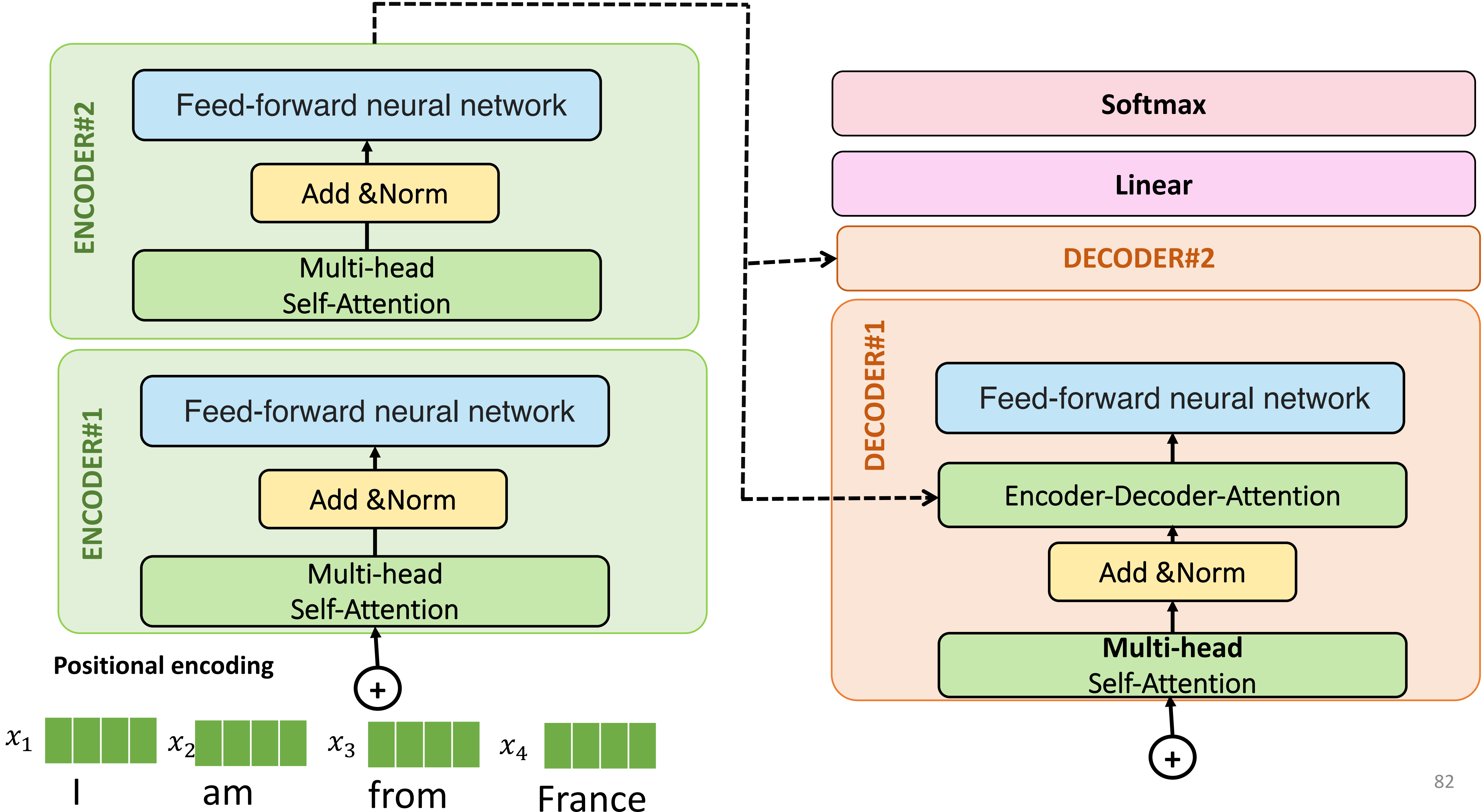
Architecture : DECODER

Encoder-Decoder-Attention



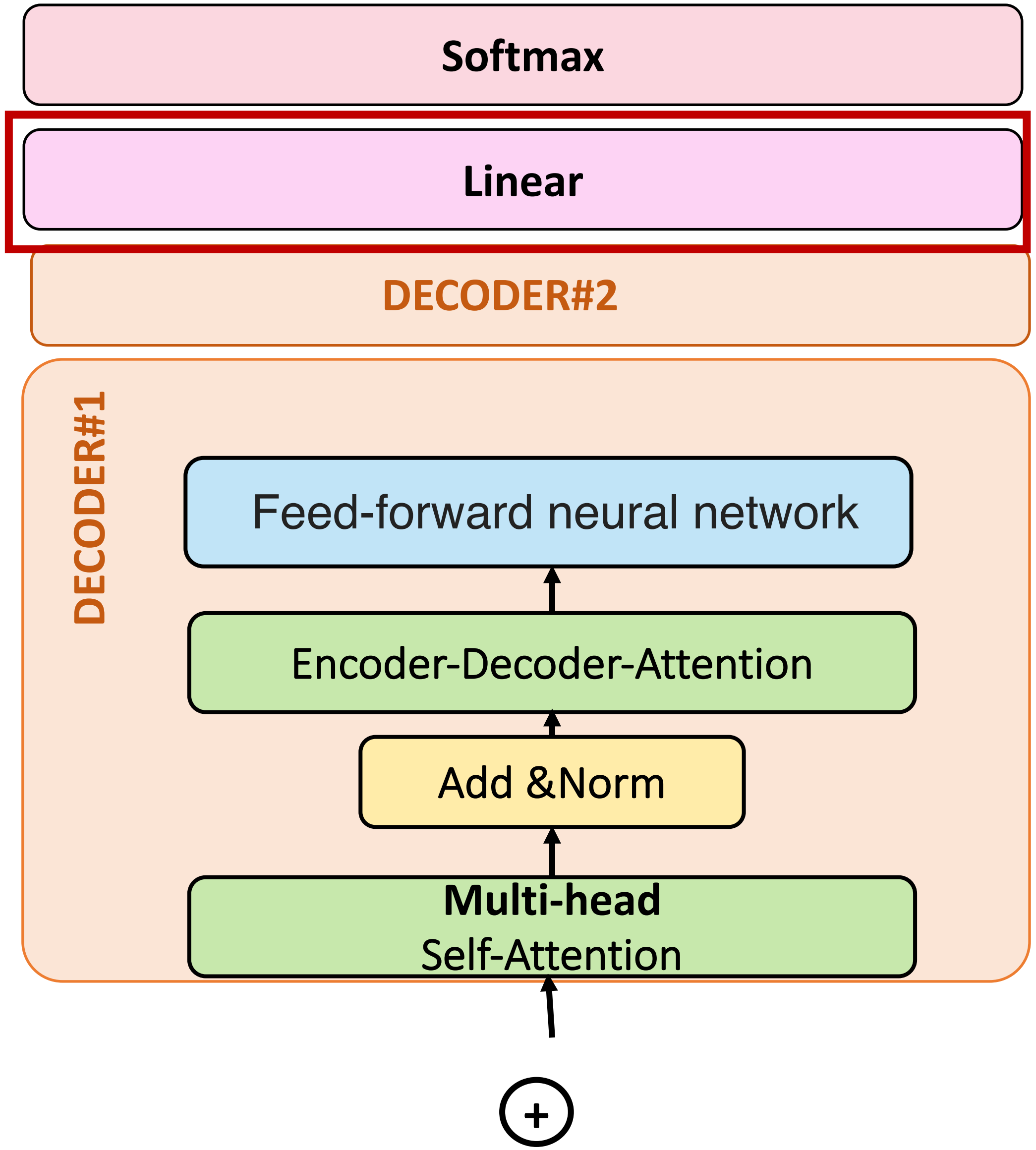
- The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

Architecture



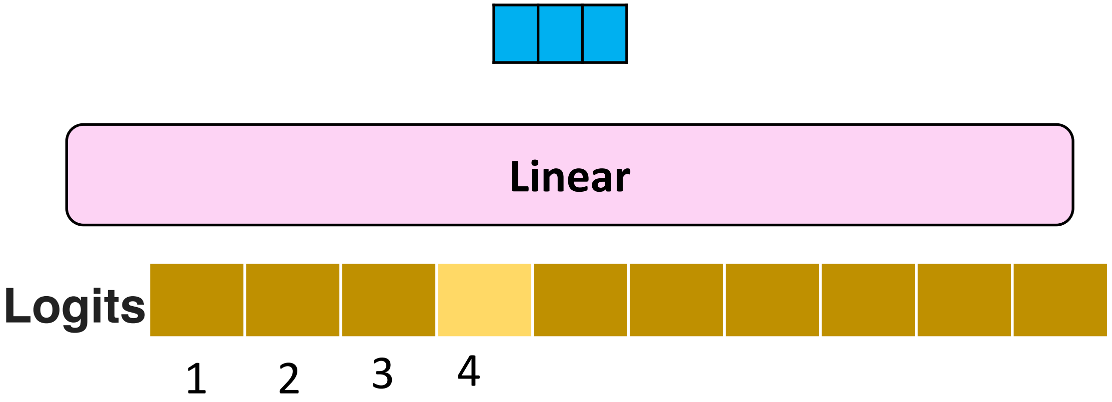
Architecture : DECODER

Linear



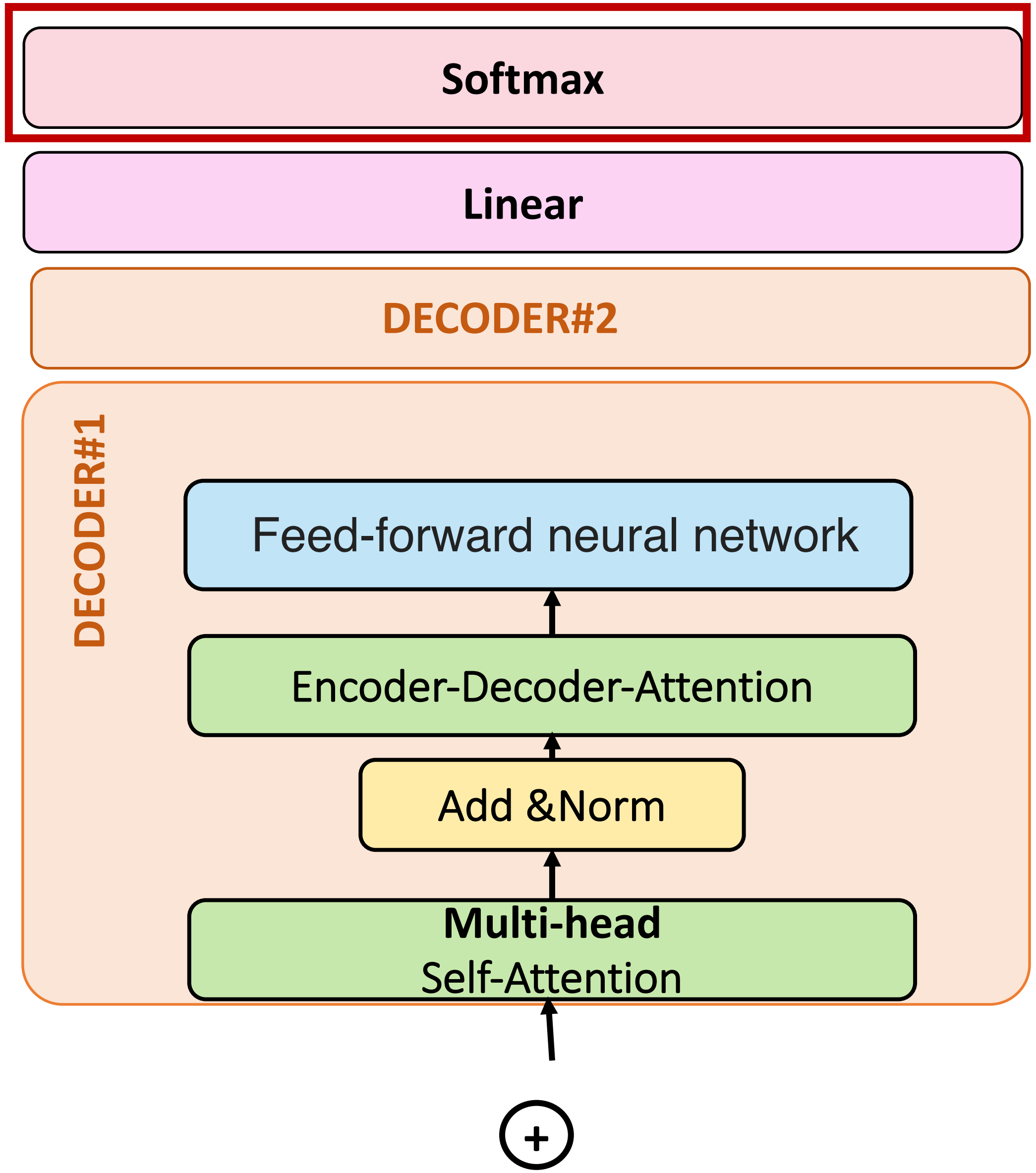
The decoder stack outputs a vector of floats. How do we turn that into a word?

- The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a **logits vector.**

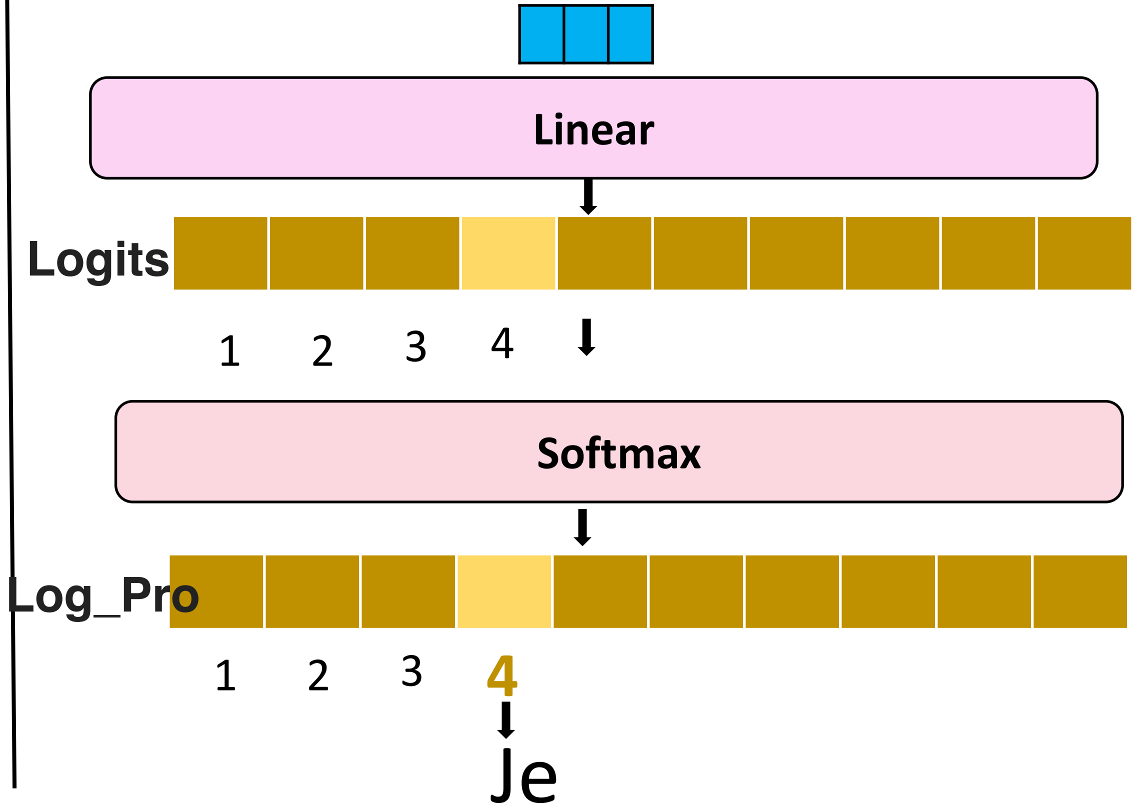


Architecture : DECODER

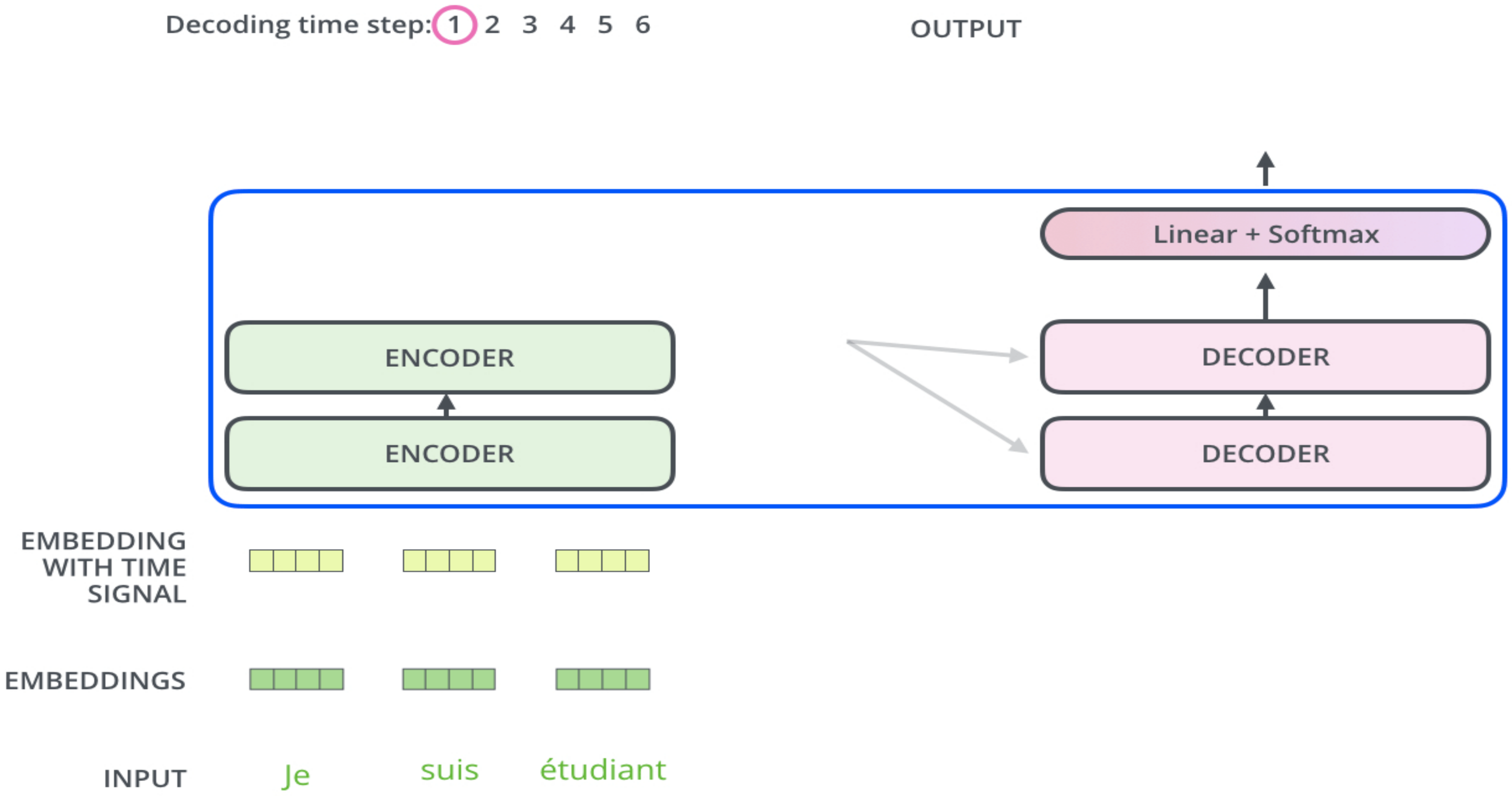
Softmax



- The softmax layer then turns those scores into probabilities.
- The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

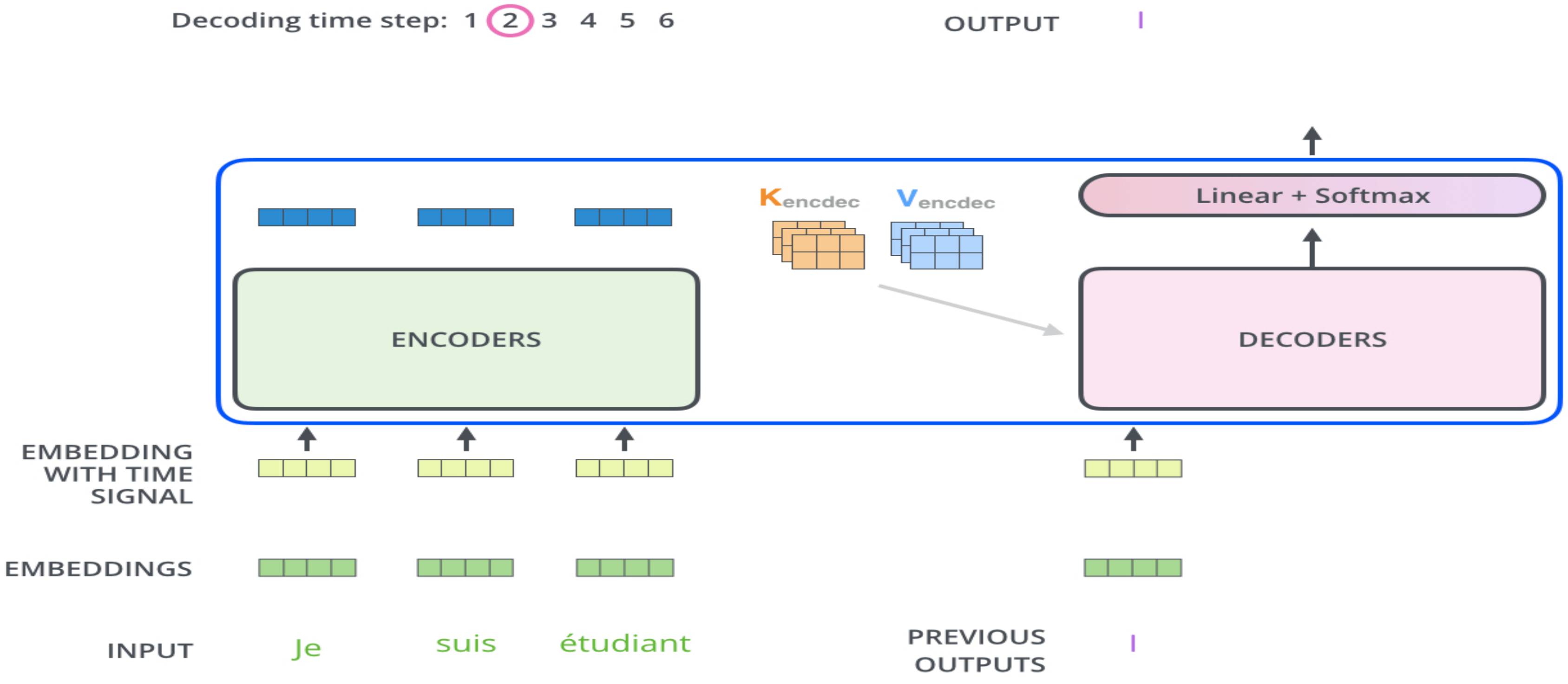


Architecture : ENCODER &DECODER



- The encoder start by processing the input sequence.
- The output of the top encoder is then transformed into a set of attention vectors K and V.

Architecture : ENCODER &DECODER



- The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results
- Embed and add positional encoding to those decoder inputs to indicate the position of each word