UNIVERSITAT DE BARCELONA

Deep Sequential Models & Embeddings

Jordi Vitrià

Recurrent models

Deep Sequential Models

Classical neural networks suffer from two severe limitations:

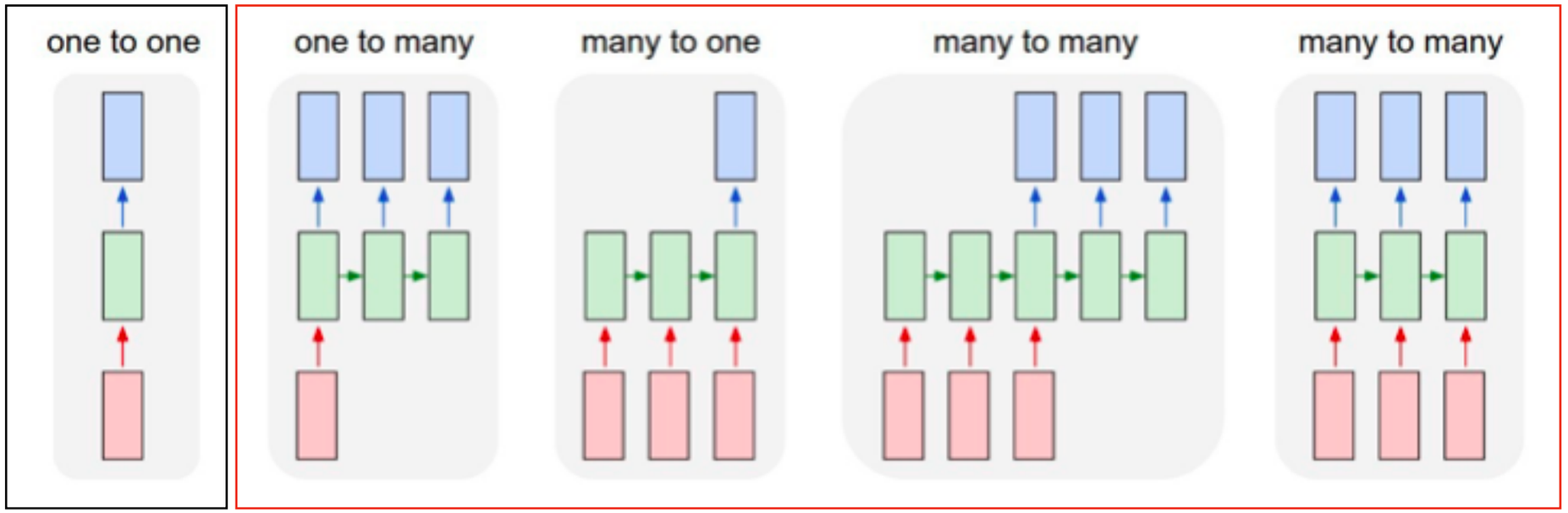
- They only accept a **fixed-sized tensors** as input and produce a fixed-sized tensor as output.
- They do not consider the **sequential nature of some data** (language, video frames, time series, etc.)

Recurrent neural networks overcome these limitations by allowing to operate over sequences of vectors (in the input, in the output, or both).

Deep Sequential Models

FCN/CNN ARCHITECTURE

Deep Sequential ARCHITECTURES

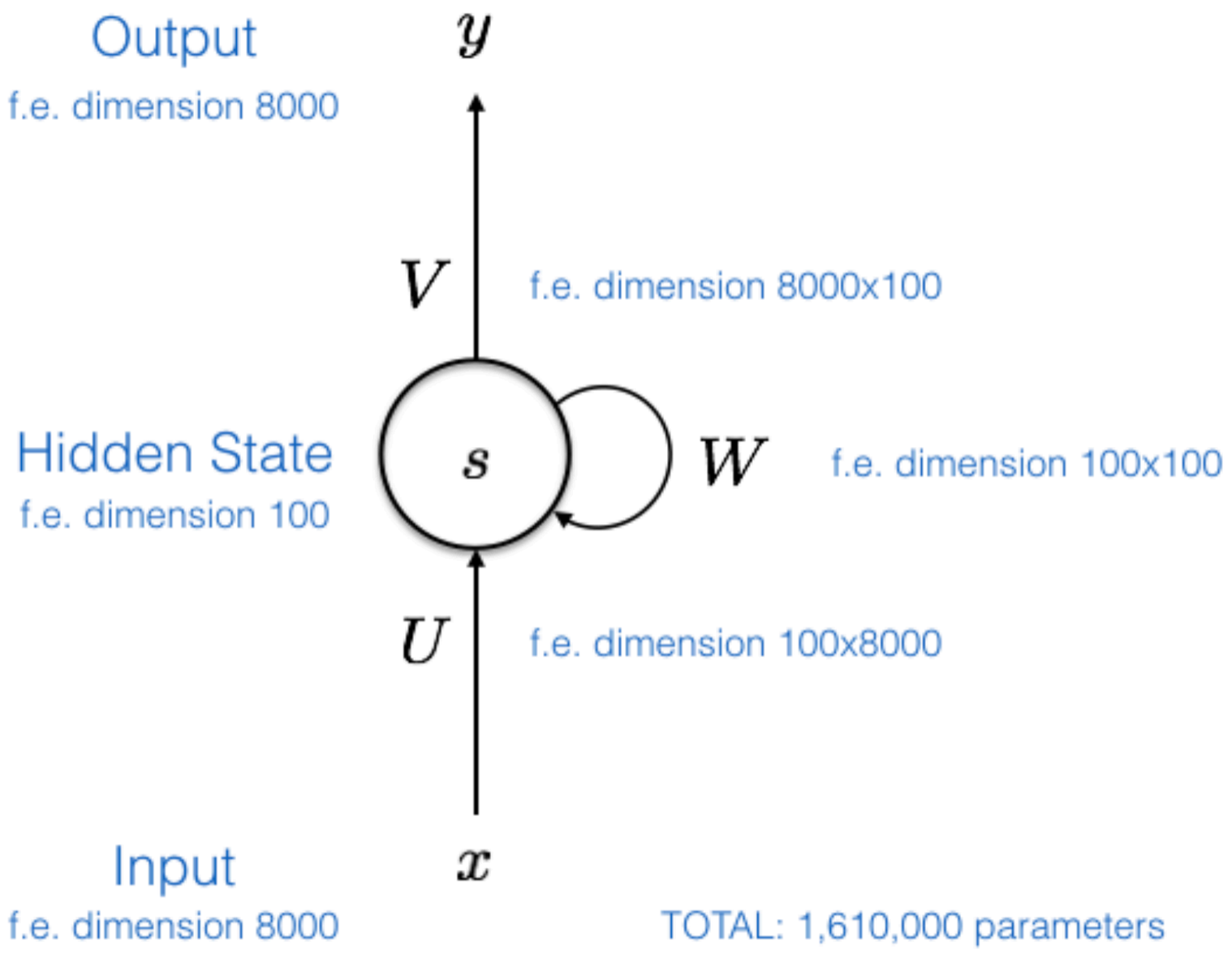


1 tensor → 1 tensor

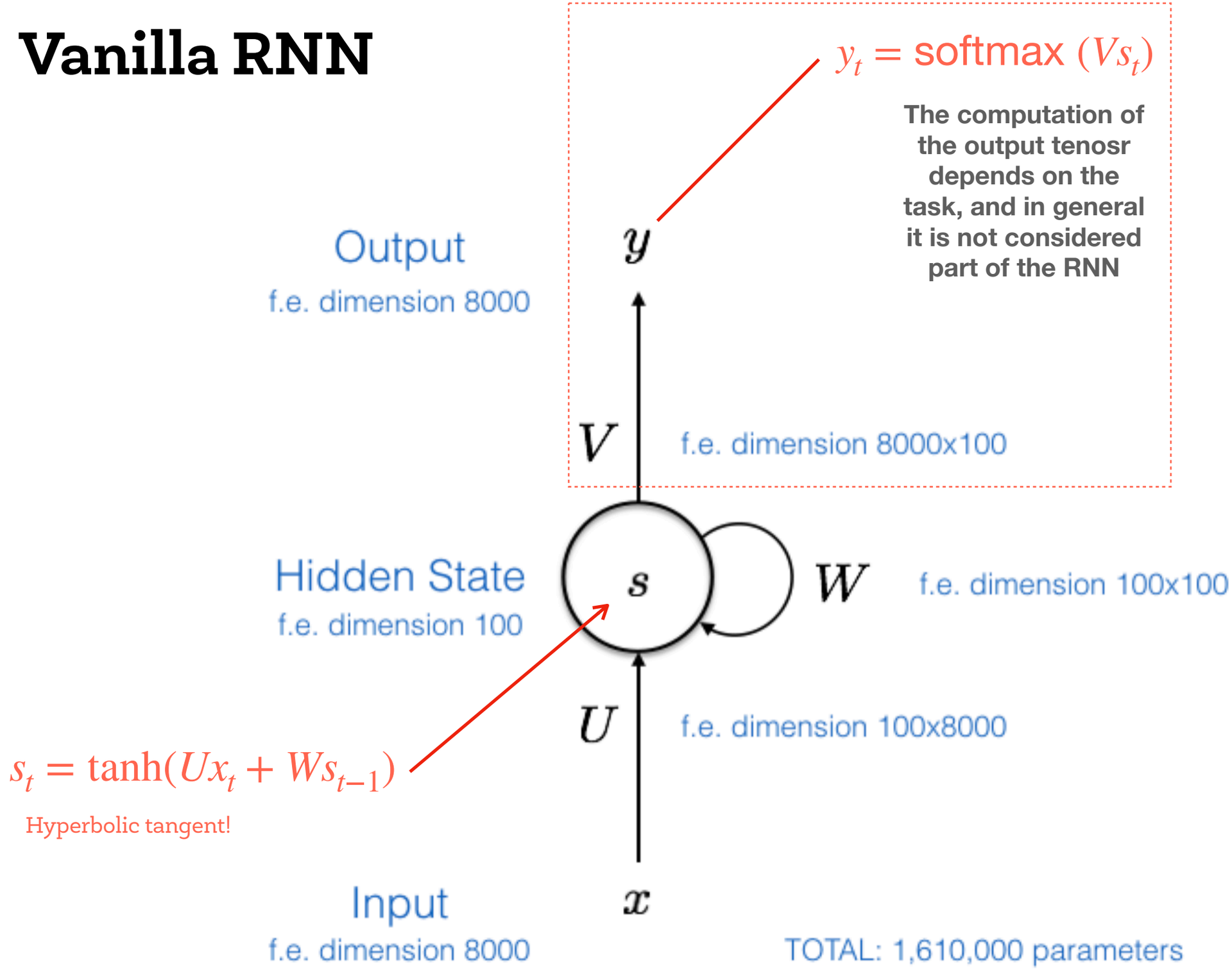
n tensors → m tensors

First approach: Vanilla RNN

Example: Recurrent model that operates on sequences of words (vocabulary = 8000 different words)



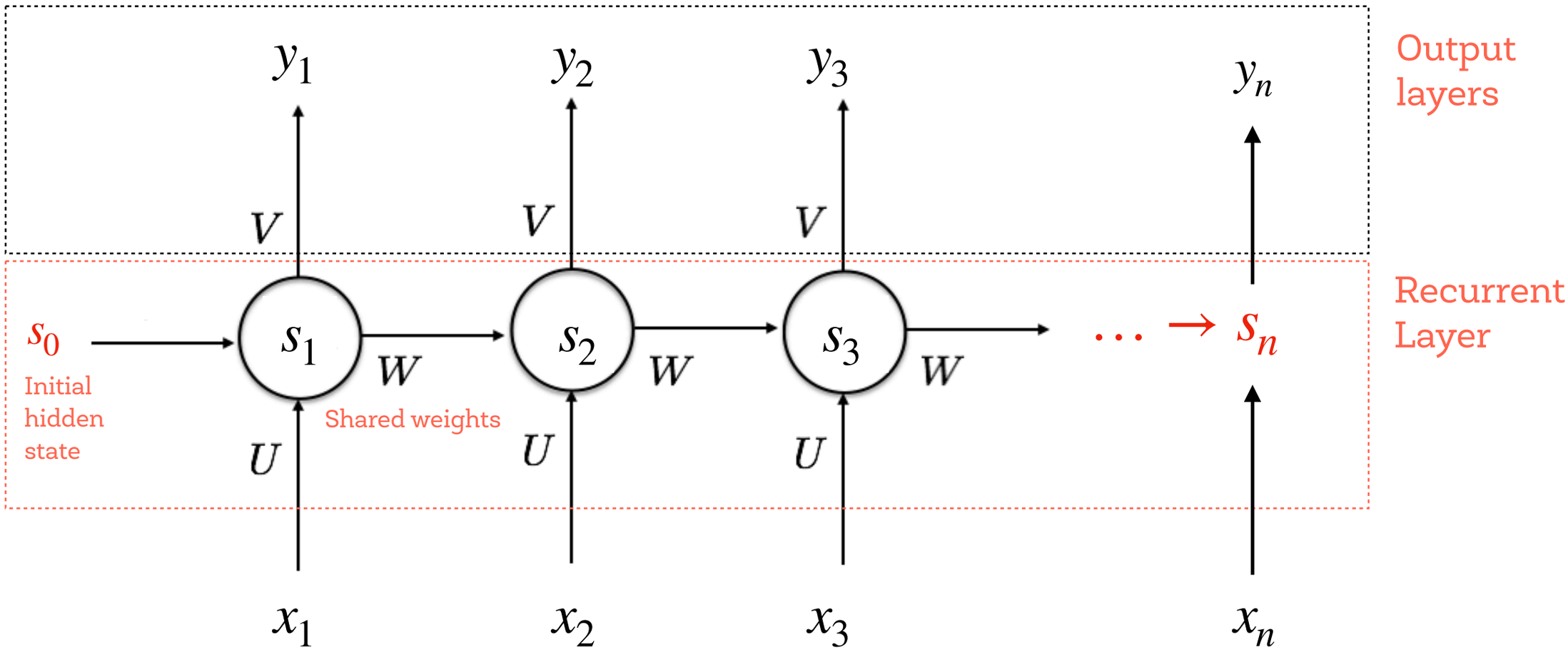
Vanilla RNN


$y_t = \text{softmax}\,(Vs_t)$
The computation of the output tenosr depends on the task, and in general it is not considered part of the RNN
Output
f.e. dimension 8000
y
V
f.e. dimension 8000x100
Hidden State
f.e. dimension 100
s
W
f.e. dimension 100x100
U
f.e. dimension 100x8000
$s_t = \tanh(Ux_t + Ws_{t-1})$
Hyperbolic tangent!
Input
f.e. dimension 8000
x
TOTAL: 1,610,000 parameters

Vanilla RNN

Instead of imagining that hidden state is being recurrently fed back into the network, it's easier to visualize the process if we **unroll** the operation into a computational graph that is composed to many time steps.

By unrolling we mean that we write out the network for the complete sequence.



Vanilla RNN

- We can think of the **hidden state** s_t as a memory of the network that **captures/stores information about the previous steps**.
- The RNN **shares the parameters** U, V, W across all time steps.
- It is not necessary to have outputs y_t at each time step.
- The value of s_0 will be learned.

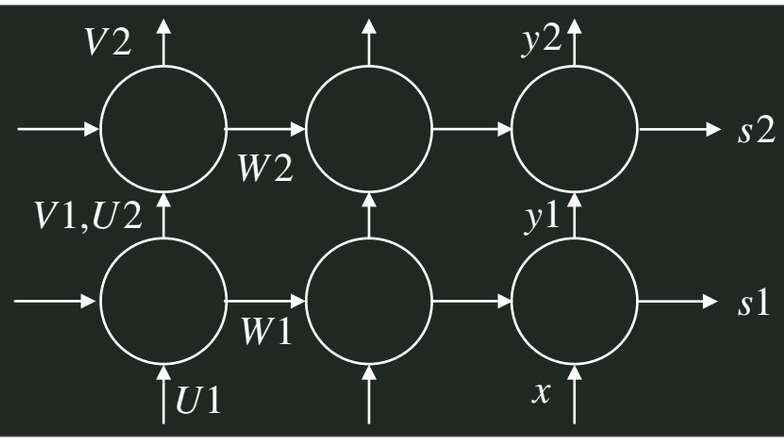
Vanilla RNN

```
def RNN_step(x):
    s = np.tanh(np.dot(W, s) + np.dot(U, x))
    y = np.dot(V, h)
    return y
```

This is heavy model because of the matrices!

We can go deep by stacking RNNs:

```
y1 = RNN.step(x)
y2 = RNN.step(y1)
```



Vanilla RNN

NumPy implementation of a simple RNN

```
import numpy as np
timesteps = 100
input_features = 32
output_features = 64
inputs = np.random.random((timesteps, input_features))
state_t = np.zeros((output_features,))
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))
successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    successive_outputs.append(output_t)
    state_t = output_t
final_output_sequence = np.concatenate(successive_outputs, axis=0)
```

Vanilla RNN for language self-learning

Let's suppose we are processing a **series of words**:

$$x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots x_T \,.$$

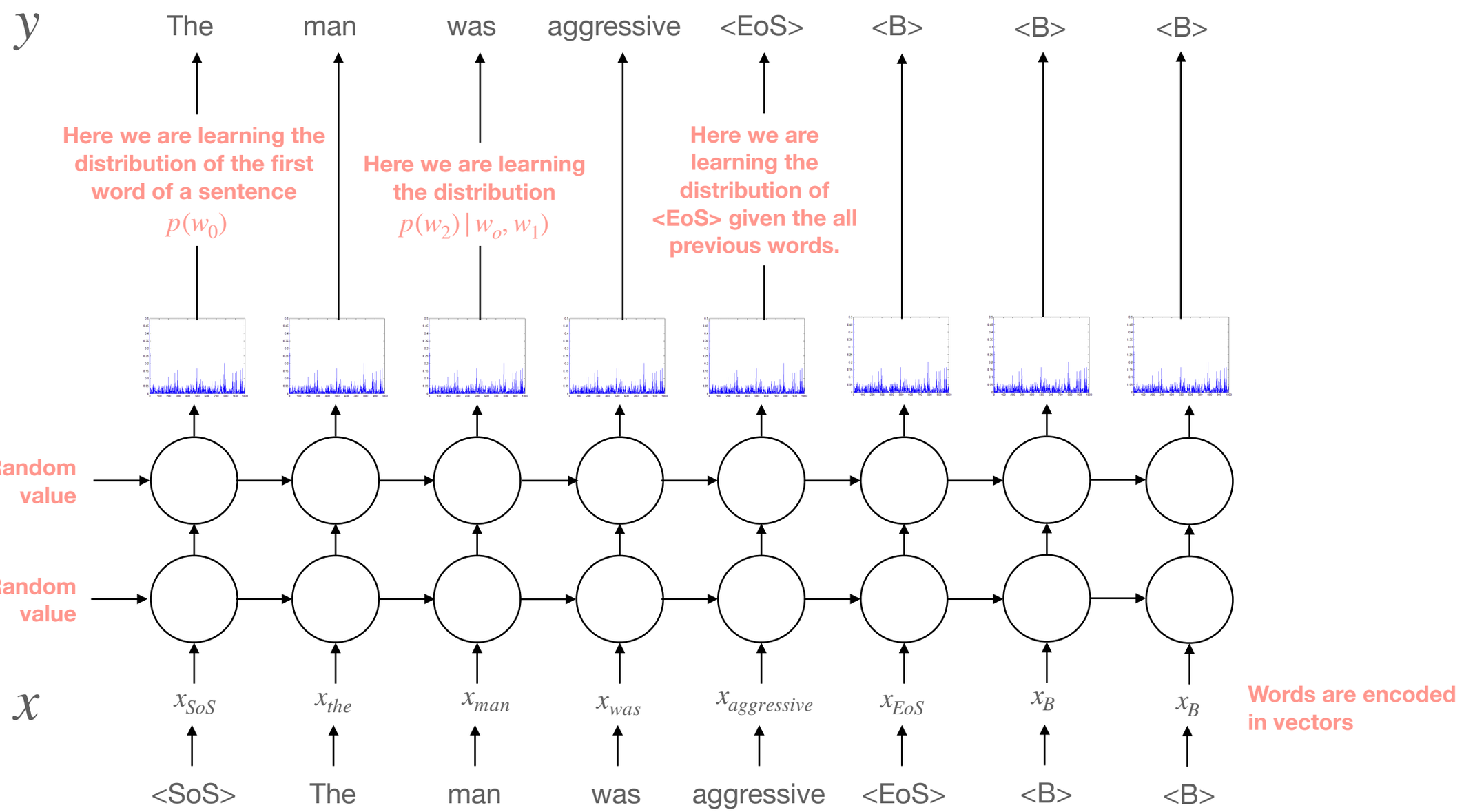
x_i are the one-hot word **vectors** corresponding to a corpus with d **symbols**.

Then, the relationship to compute the hidden layer output features at each time-step t is $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$, where:

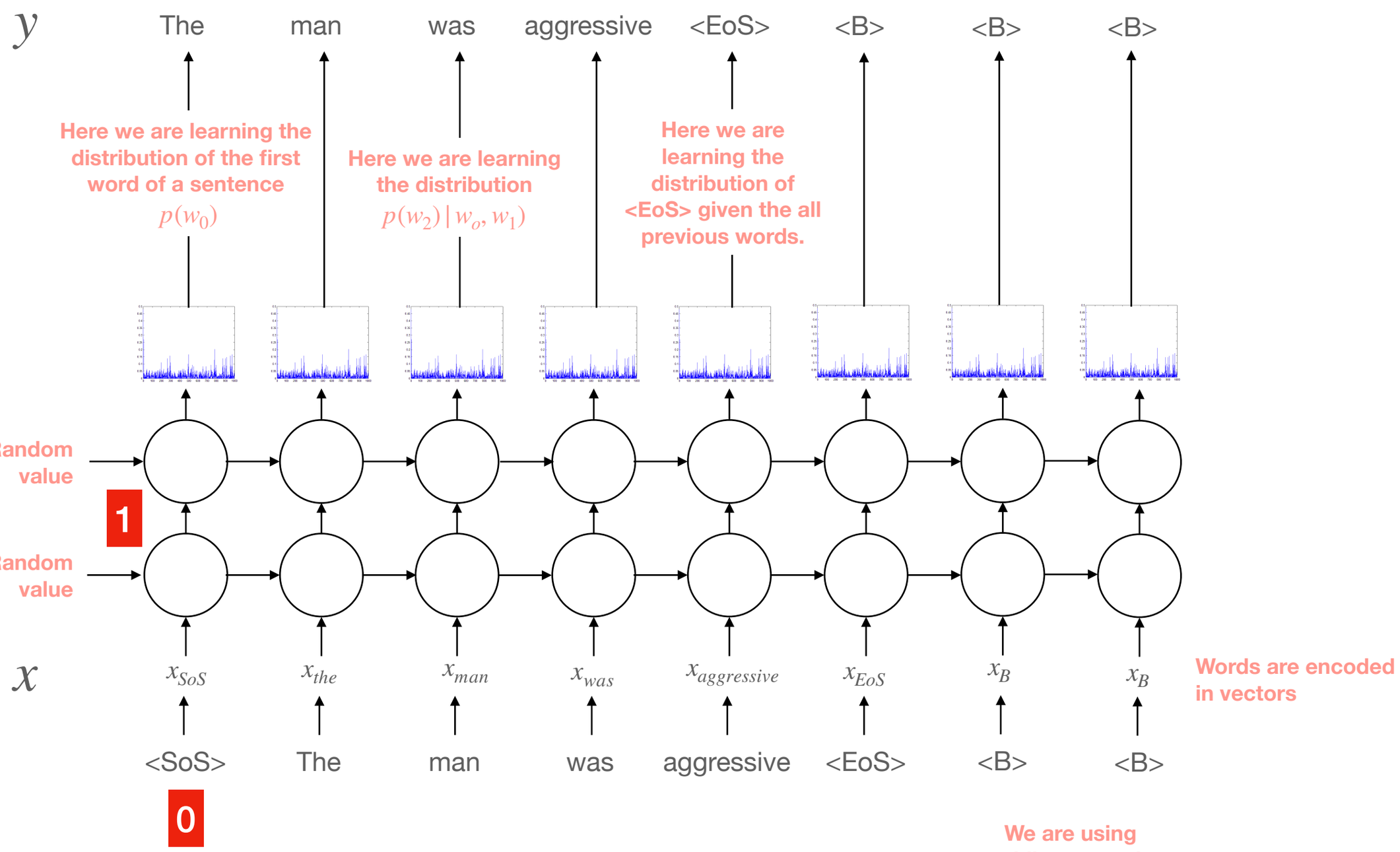
- $x_t \in \mathbb{R}^d$ is input word vector at time t.
- $W^{(hx)} \in \mathbb{R}^{D_h \times d}$ is the weights matrix used to condition the input word vector, x_t.
- $W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$ is the weights matrix used to condition the output of the previous time-step, h_{t-1}.
- $h_{t-1} \in \mathbb{R}^{D_h}$ is the output of the non-linear function at the previous time-step, $t-1$.
- $h_0 \in \mathbb{R}^{D_h}$ is an initialization vector for the hidden layer at time-step $t = 0$.
- $\sigma()$ is the non-linearity function (normally, *tanh*).

We can compute the **output probability distribution over the vocabulary** at each time-step t as $\hat{y}_t = softmax(W^{(hy)}h_t)$. Essentially, $\hat{y}_t$ is the next predicted word given the document context score so far (i.e. h_{t-1}) and the last observed word vector $x^{(t)}$. Here, $W^{(hy)} \in \mathbb{R}^{d \times D_h}$ and $\hat{y} \in \mathbb{R}^d$.

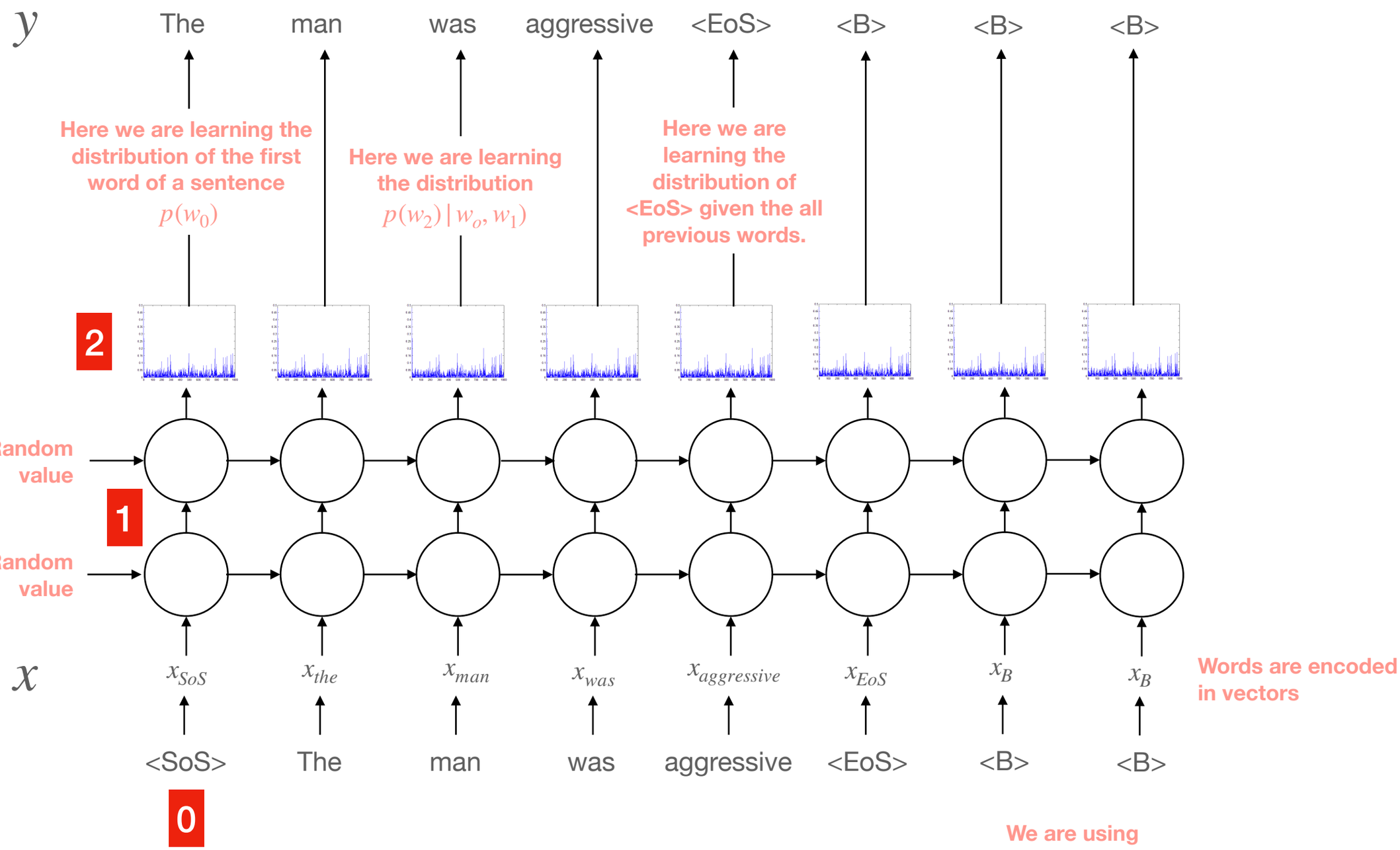
Vanilla RNN for language self-learning



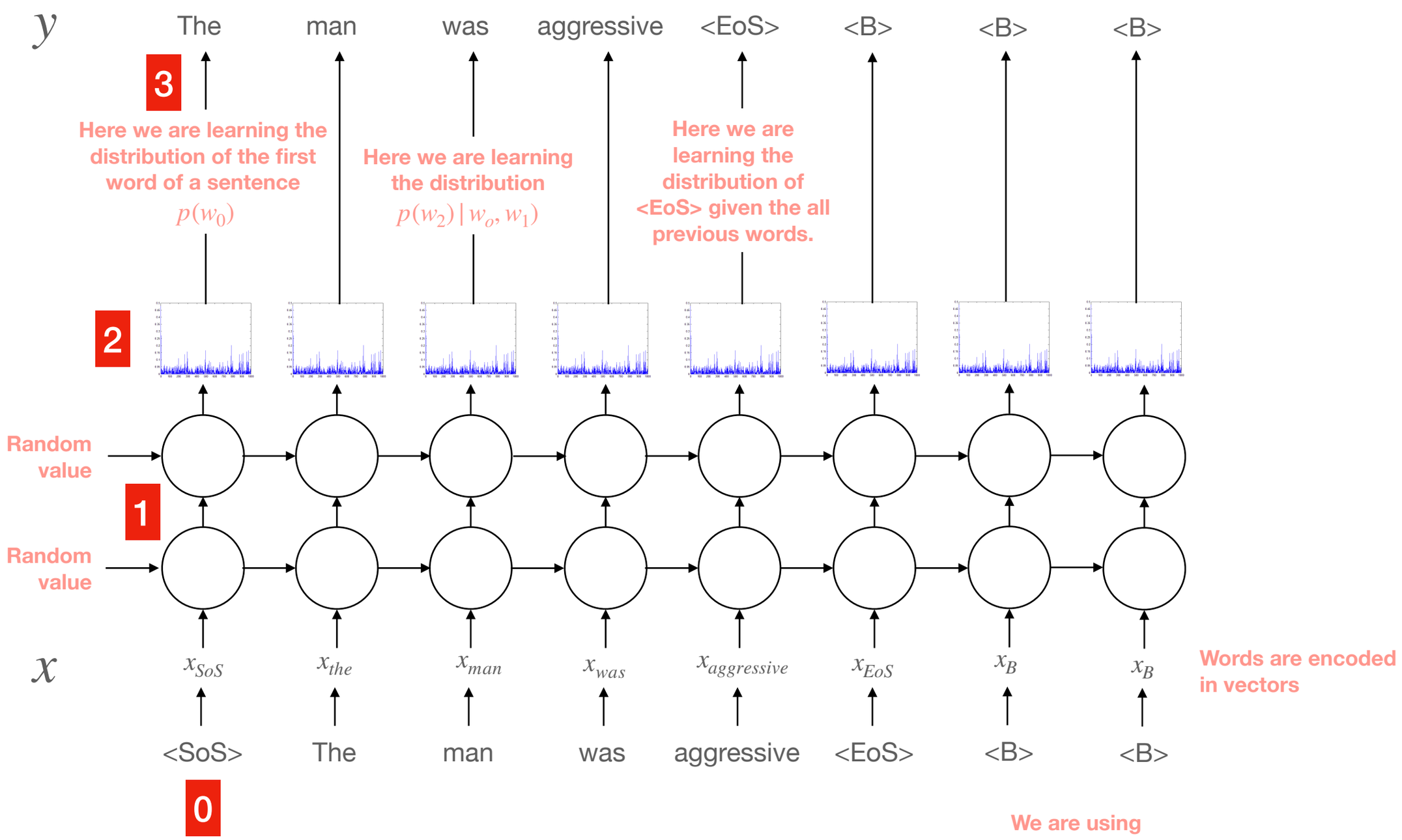
Vanilla RNN for language self-learning



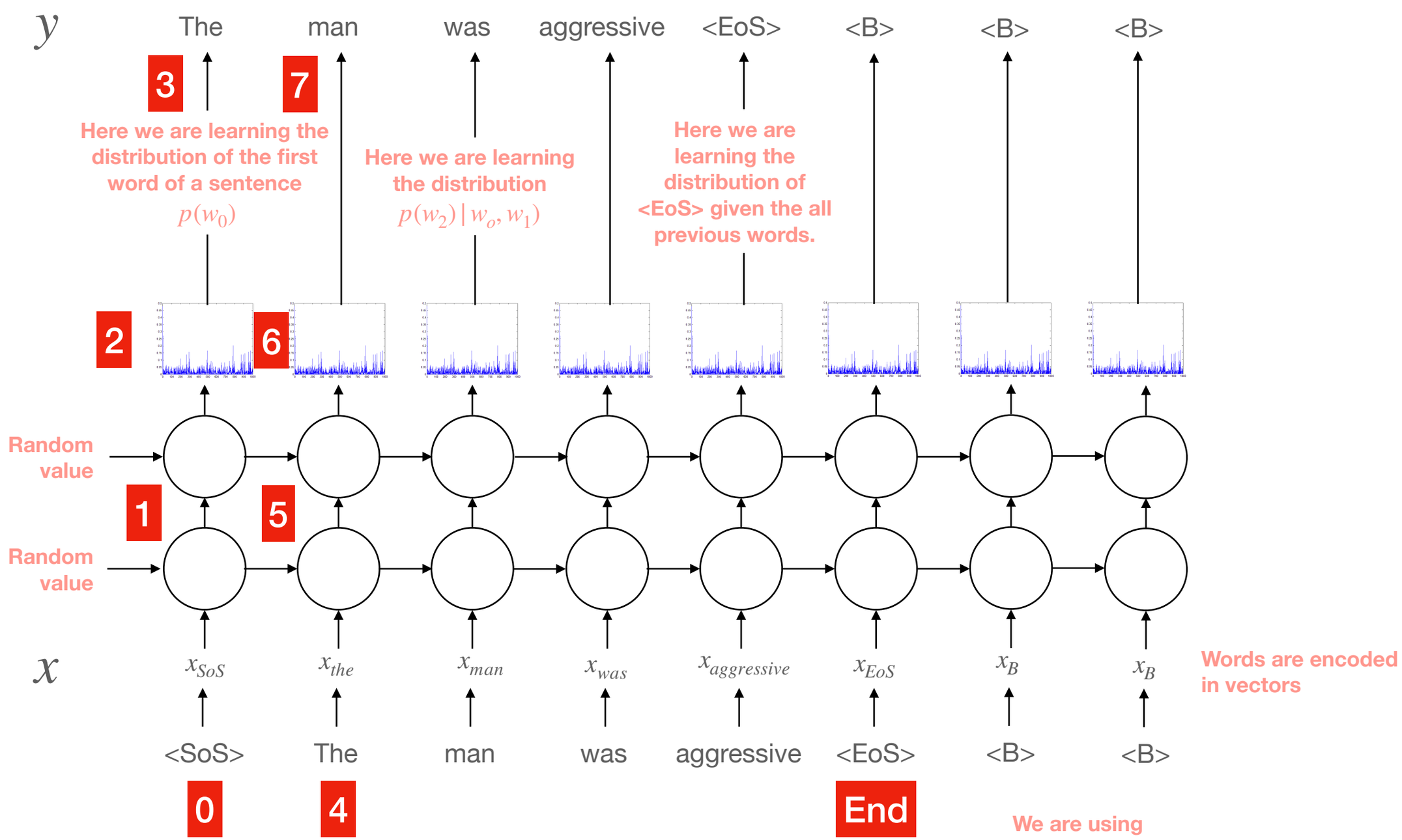
Vanilla RNN for language self-learning



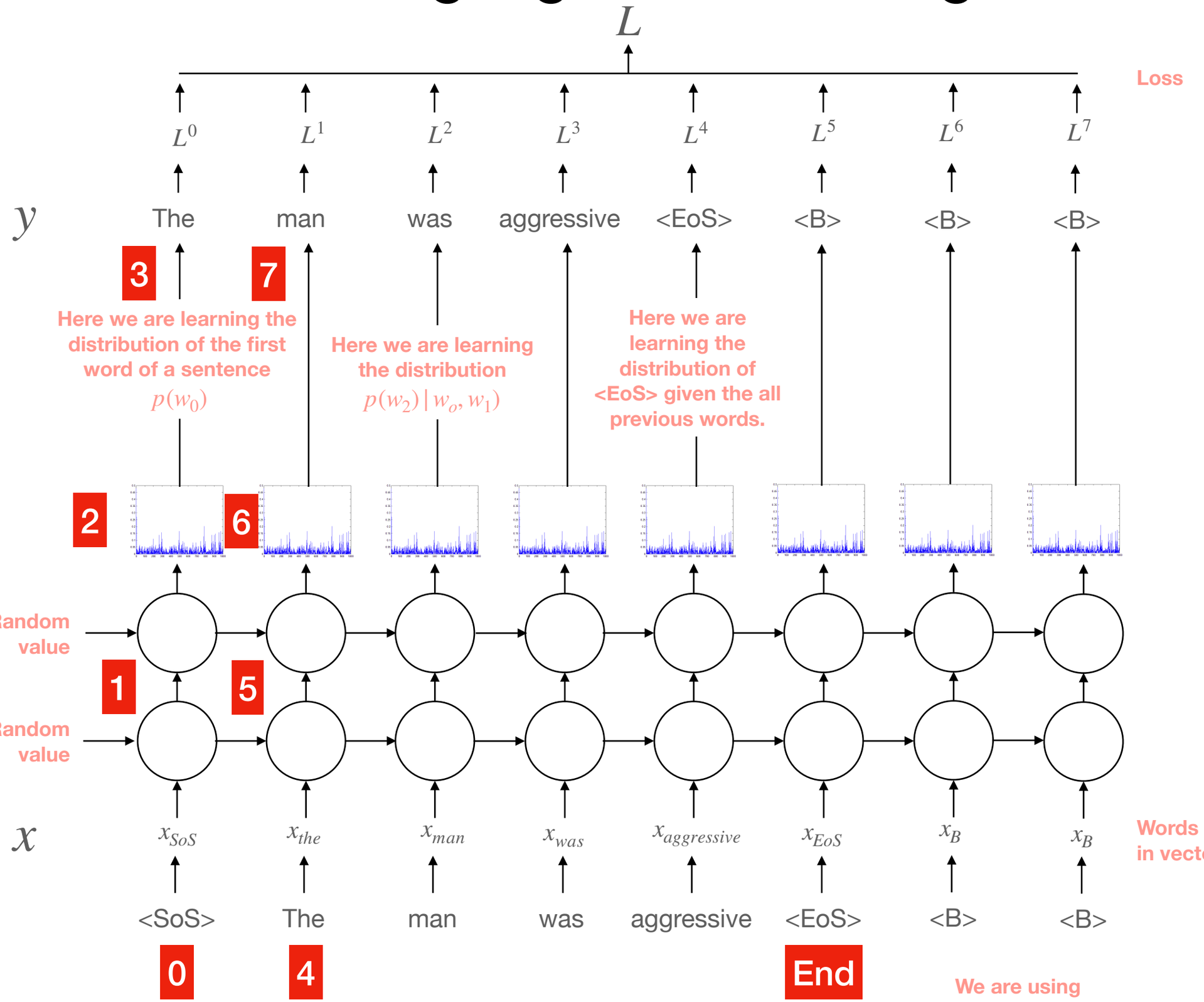
Vanilla RNN for language self-learning



Vanilla RNN for language self-learning



Vanilla RNN for language self-learning



Vanilla RNN for language self-learning

The loss function is the cross-entropy (classification) error:

$$L^{(t)}(W) = -\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j})$$

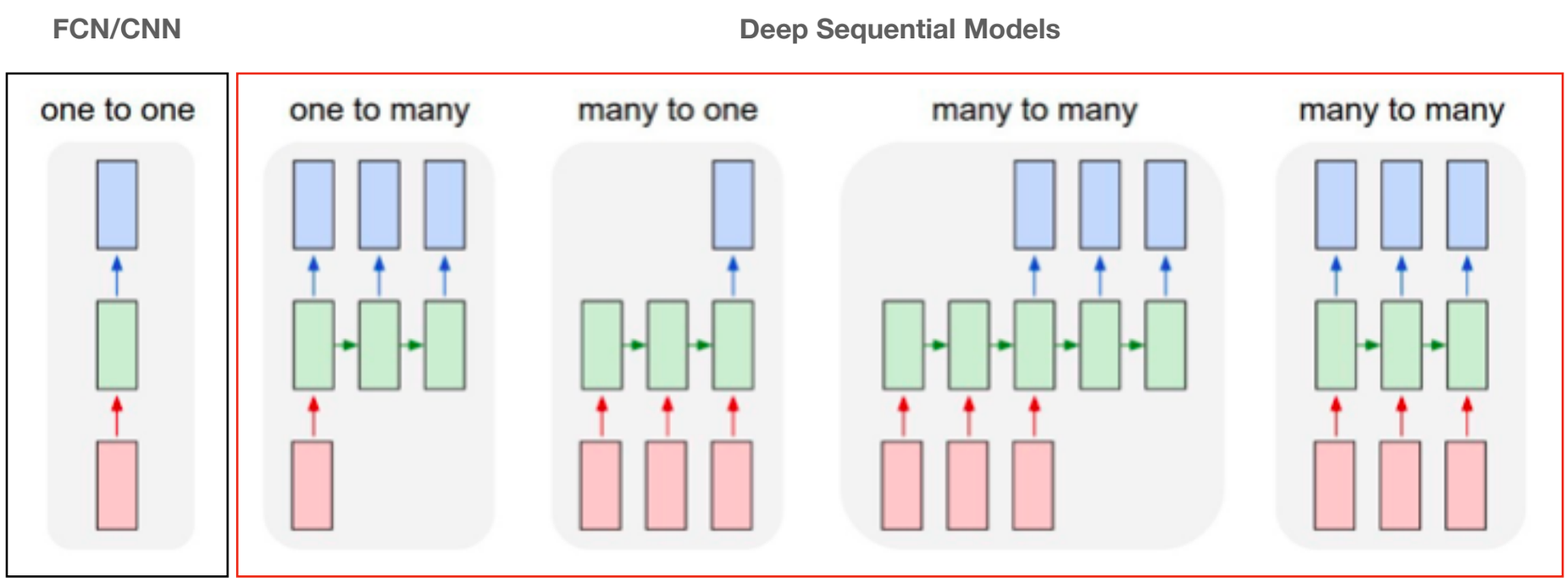
The cross entropy error over a sentence of size T is:

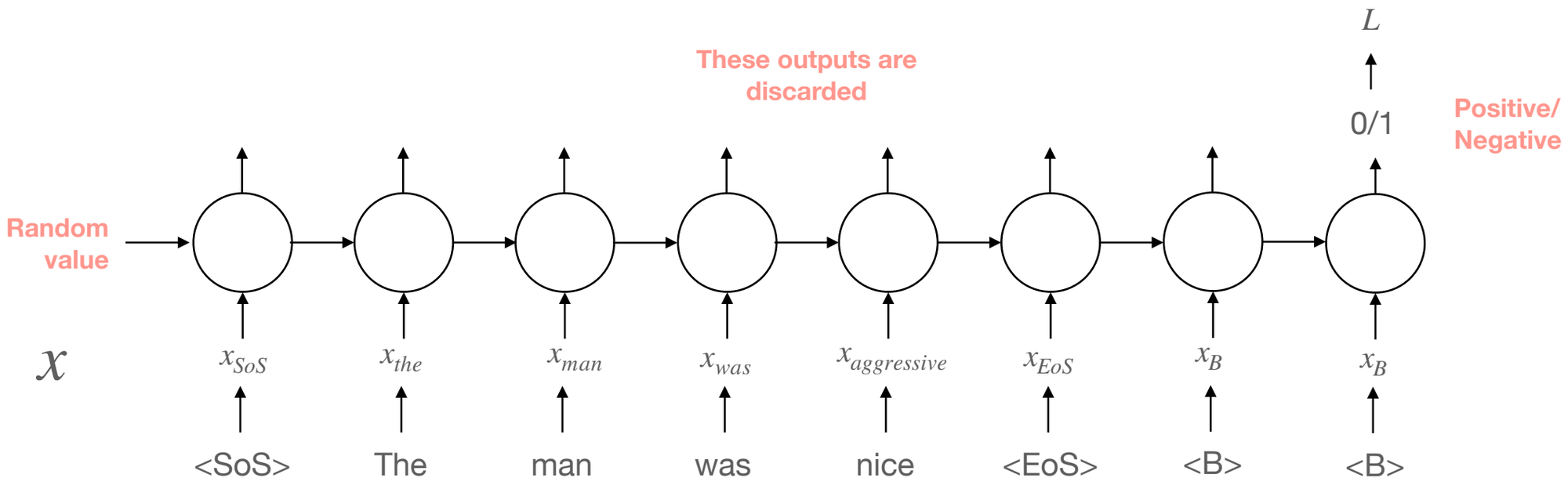
$$L = \frac{1}{T}\sum_{t=1}^{T} L^{(t)}(W) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j})$$

Vanilla RNN

```
tf.keras.layers.SimpleRNN(
    units,
    activation="tanh",
    use_bias=True,
    kernel_initializer="glorot_uniform",
    recurrent_initializer="orthogonal",
    bias_initializer="zeros",
    kernel_regularizer=None,
    recurrent_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    recurrent_constraint=None,
    bias_constraint=None,
    dropout=0.0,
    recurrent_dropout=0.0,
    return_sequences=False,
    return_state=False,
    go_backwards=False,
    stateful=False,
    unroll=False,
    **kwargs
)
```

Deep Sequential Models



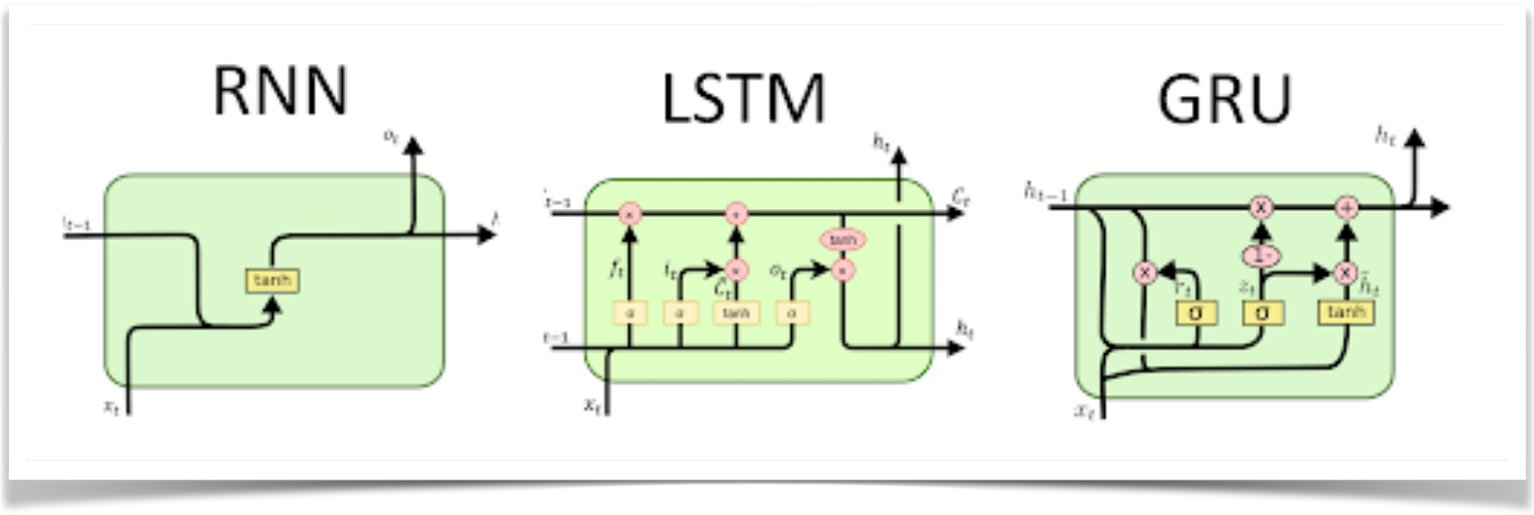


Example: Sentiment Analysis - Many to One

Gated Units

The most important types of **gated RNN**s are:

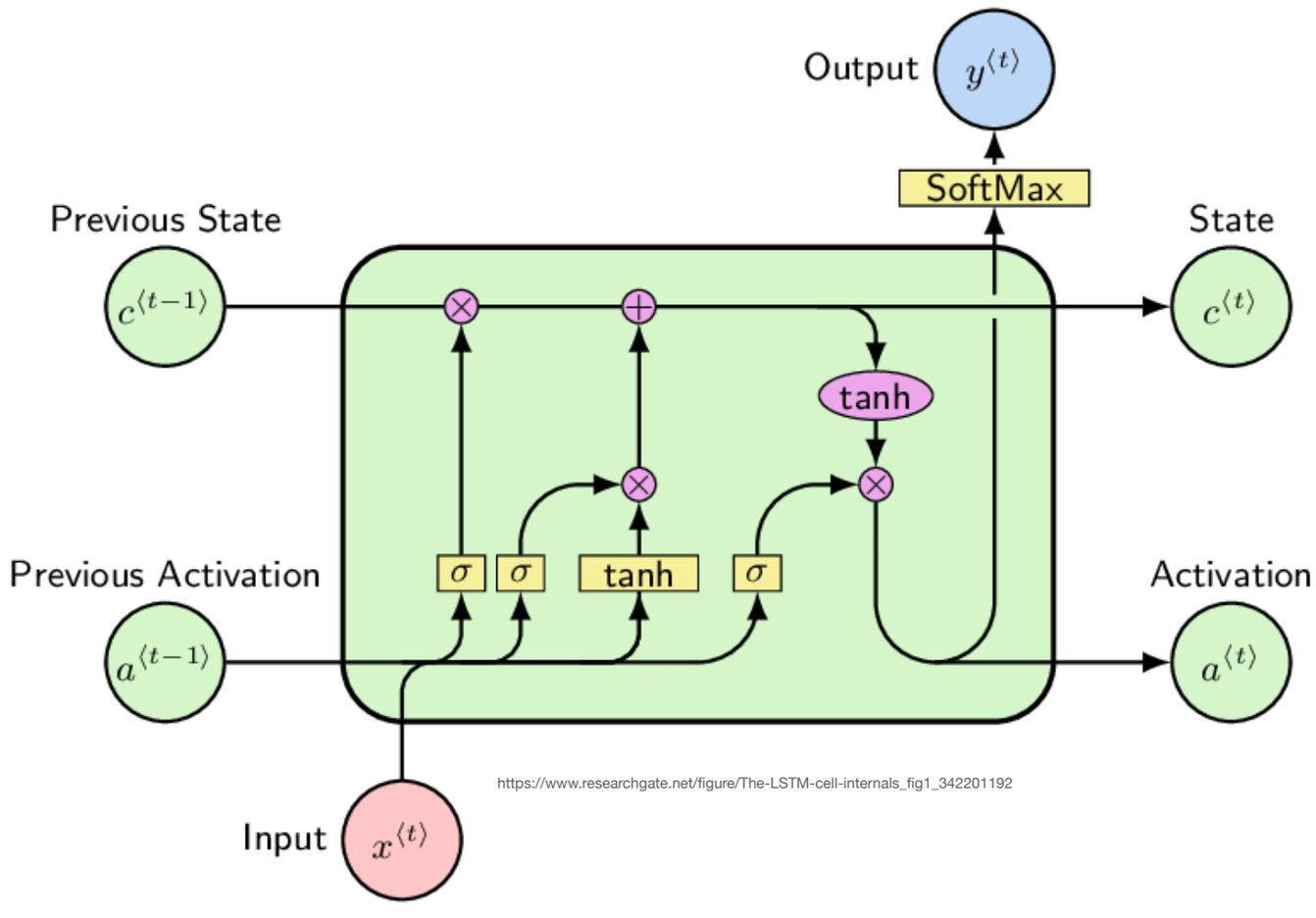
- **Long Short Term Memories** (LSTM). It was introduced by S.Hochreiter and J.Schmidhuber in 1997 and is widely used. LSTM is very good in the long run due to its high complexity.
- **Gated Recurrent Units** (GRU). It was introduced by K.Cho. It is simpler than LSTM, fasters and optimizes quicker.



LSTM

The key idea of LSTMs is the cell state C, the horizontal line running through the top of the diagram.

The cell state is kind of like a **conveyor belt**. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

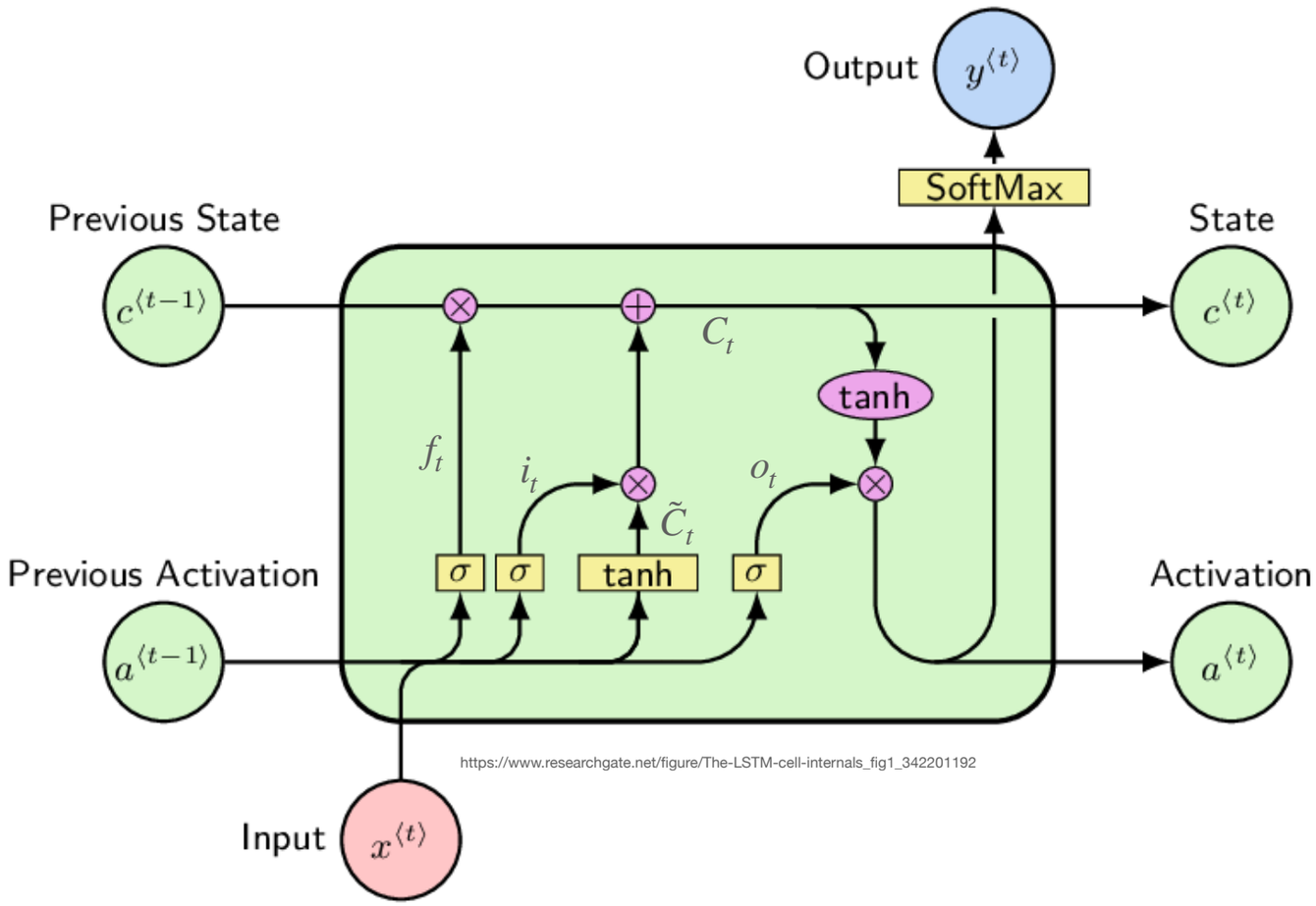


https://www.researchgate.net/figure/The-LSTM-cell-internals_fig1_342201192

LSTM

LSTM has the ability to **remove** or **add** information to the cell state, carefully regulated by structures called gates.

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



https://www.researchgate.net/figure/The-LSTM-cell-internals_fig1_342201192

$f_t = \sigma(W_f \cdot [a_{t-1}, x_t])$ (Forget gate)

$i_t = \sigma(W_i \cdot [a_{t-1}, x_t])$ (Input gate)

$\tilde{C}_t = \tanh(W_C \cdot [a_{t-1}, x_t])$

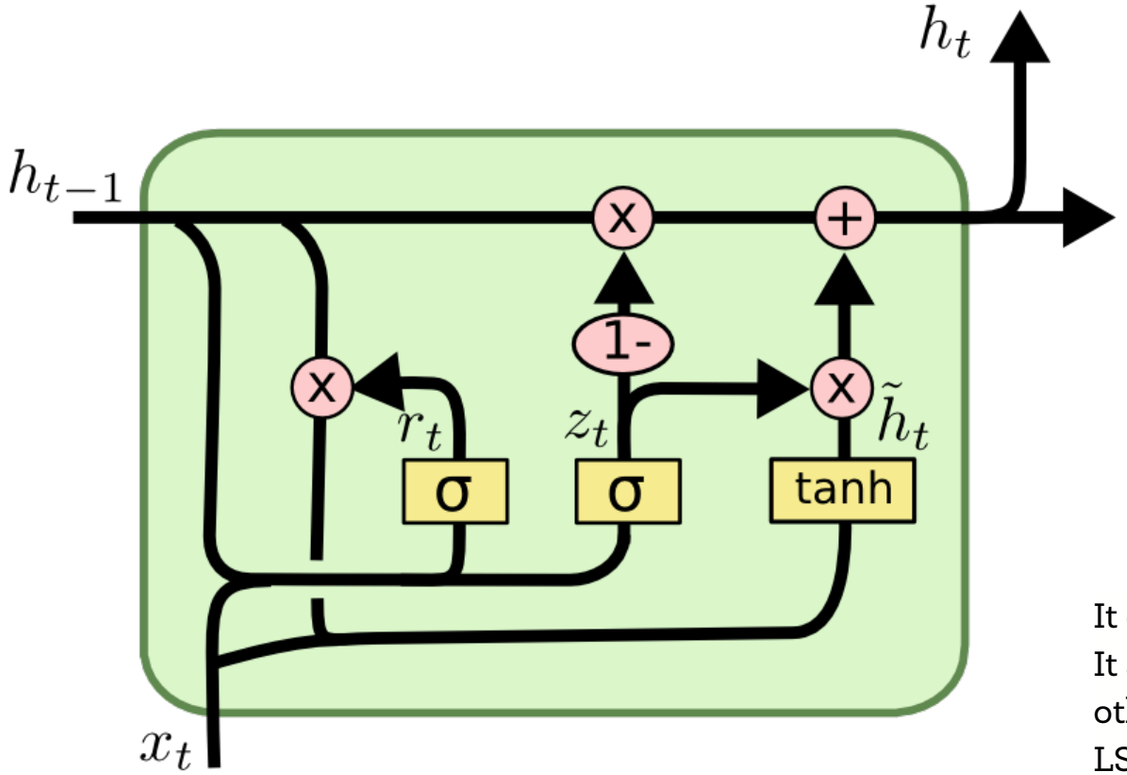
$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$ (Update gate)

$o_t = \sigma(W_o \cdot [a_{t-1}, x_t])$

$h_t = o_t * \tanh(C_t)$ (Output gate)

GRU

Gated recurrent units are designed in a manner to have persistent memory, like LSTMs, but they are lighter in terms of parameters.



$z_t = \sigma(W_z \cdot [x_t, h_{t-1}])$ (Update gate)

$r_t = \sigma(W_r \cdot [x_t, h_{t-1}])$ (Reset gate)

$\tilde{h}_t = \tanh(r_t \cdot [x_t, r_t \circ h_{t-1}])$ (New memory)

$h_t = (1 - z_t) \circ \tilde{h}_{t-1} + z_t \circ h_t$ (Hidden state)

It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models.

RNN in Keras

All recurrent layers in Keras (SimpleRNN, LSTM, and GRU) can be run in two different modes: they can return either **full sequences** of successive outputs for each timestep (a 3D tensor of shape (batch_size, timesteps, output_features)) or only the **last output for each input sequence** (a 2D tensor of shape (batch_size, output_features)).

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)  ❶
>>> print(outputs.shape)
(None, 16)
```

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)  ❶
>>> print(outputs.shape)
(120, 16)
```

These two modes are controlled by the **return_sequences** constructor argument.

RNN in Keras

In addition, a RNN layer can return its **final internal state**(s).

The returned states can be used to resume the RNN execution later, or to initialize another RNN.

To configure a RNN layer to return its internal state, set the **return_state** parameter to True when creating the layer. Note that LSTM has 2 state tensors, but GRU only has one.

To configure the initial state of the layer, just call the layer with additional keyword argument **initial_state.**

RNN in Keras

We can use LSTM/GRU layers with **multiple input sizes**. But, you need to process them before they are feed to the LSTM.

You need the **pad the sequences** of varying length to a fixed length. For this preprocessing, you need to determine the max length of sequences in your dataset.

You can do this in Keras with :

```
y = keras.preprocessing.sequence.pad_sequences(x,maxlen=10)
```

If the sequence is shorter than the max length, then zeros (default value) will appended till it has a length equal to the max length.

If the sequence is longer than the max length then, the sequence will be trimmed to the max length.

RNN in Keras

Advanced features:

- **Recurrent dropout** — A variant of dropout, used to fight overfitting in recurrent layers.
- **Stacking recurrent layers** — This increases the representational power of the model (at the cost of higher computational loads).
- **Bidirectional recurrent layers** — These present the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

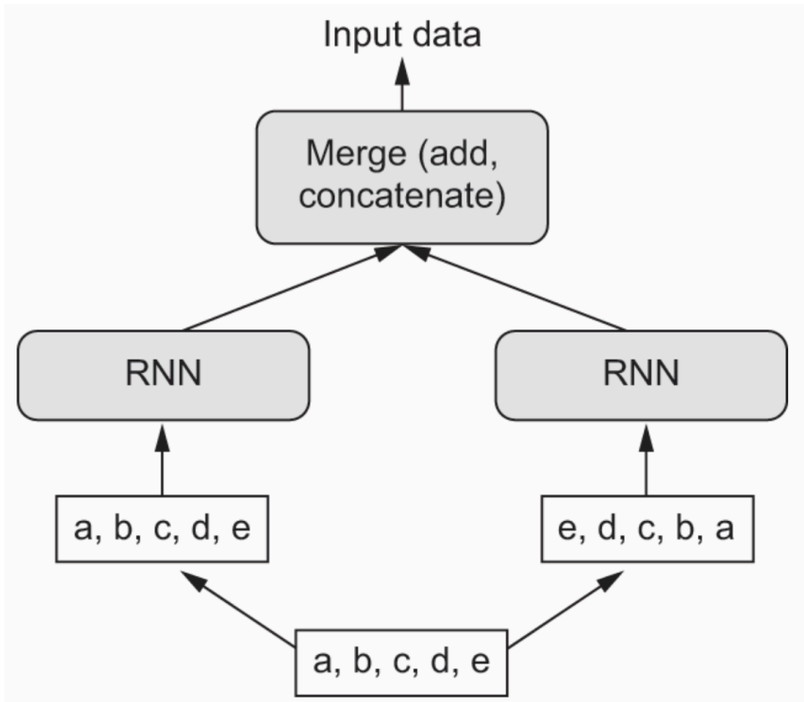
RNN in Keras

Bidirectional:

RNNs are notably order-dependent: they process the time steps of their input sequences in order, and shuffling or reversing the time steps can completely change the representations the RNN extracts from the sequence.

A **bidirectional RNN** exploits the order sensitivity of RNNs: it consists of using two regular RNNs, such as the GRU and LSTM layers you're already familiar with, each of which processes the input sequence in one direction (chronologically and anti-chronologically), and then merging their representations.

RNN in Keras



The equations are (arrows are for designing left-to-right and right-to-left tensors):

$$\overrightarrow{h}_t = f(\overrightarrow{W}x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b})$$
$$\hat{y}_t = g(Uh_t + c) = g(U[\overrightarrow{h}_t; \overleftarrow{h}_t] + c)$$

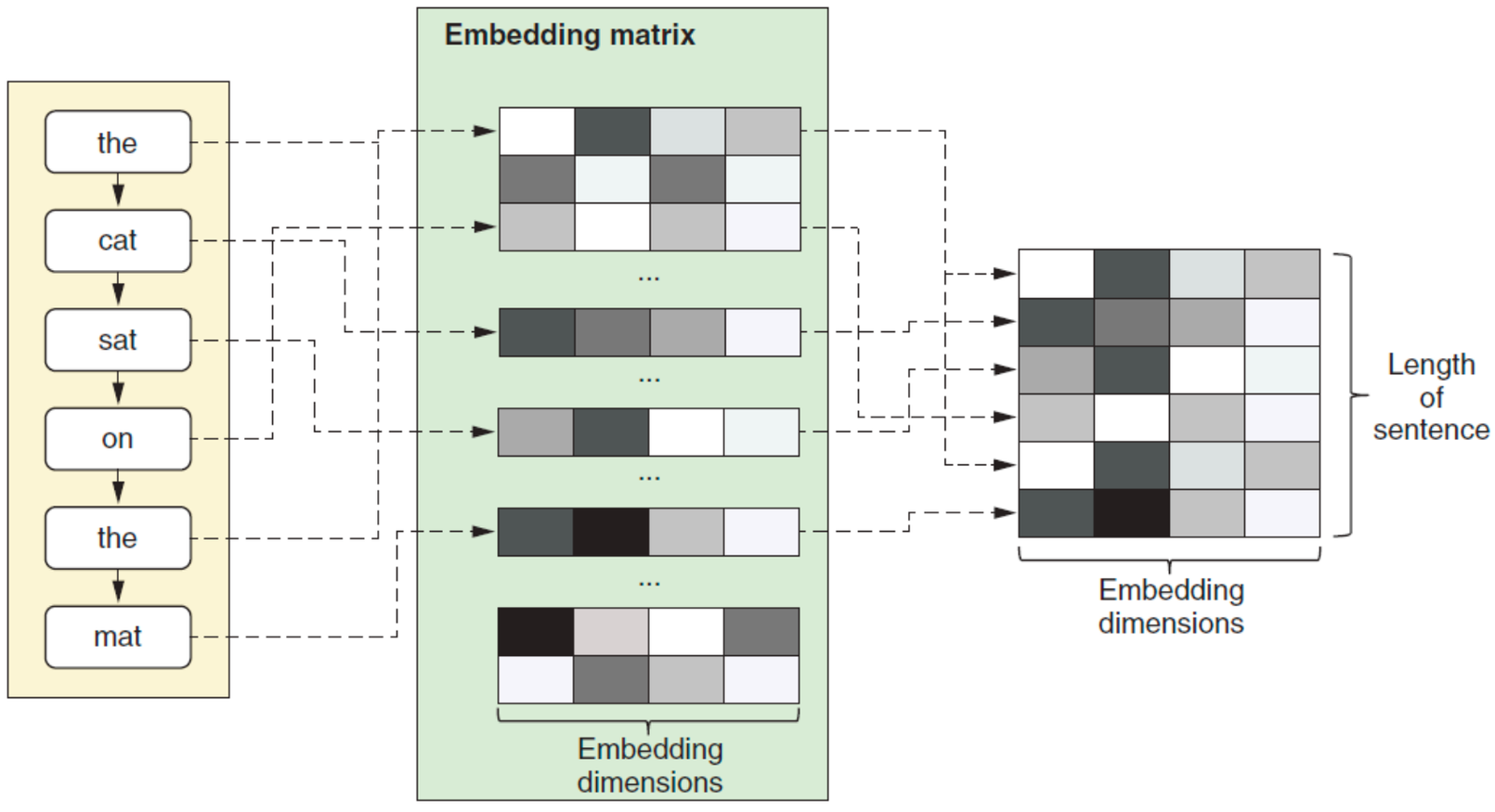
$[\overrightarrow{h}_t; \overleftarrow{h}_t]$ summarizes the past and future of a single element of the sequence.

Bidirectional RNNs can be stacked as usual!

Advancement: Embedding Layers

The model begins with an **embedding layer** which turns the input integer indices into the corresponding word vectors.

Word embedding is a way to represent a word as a vector. Word embeddings allow the value of the vector's element to be trained. After training, words with similar meanings often have the similar vectors.



https://djajafer.medium.com/multi-class-text-classification-with-keras-and-lstm-4c5525bef592

Example in Keras

```
1 model = Sequential()
2
3 model.add(Embedding(vocab_size, embedding_dim))
4 model.add(Dropout(0.5))
5 model.add(LSTM(embedding_dim,return_sequences=True))
6 model.add(LSTM(embedding_dim))
7 model.add(Dense(6, activation='softmax'))
8
9 model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 64)          320000
_________________________________________________________________
dropout (Dropout)            (None, None, 64)          0
_________________________________________________________________
lstm (LSTM)                  (None, None, 64)          33024
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                33024
_________________________________________________________________
dense (Dense)                (None, 6)                 390
=================================================================
Total params: 386,438
Trainable params: 386,438
Non-trainable params: 0
```

Time Series Analysis

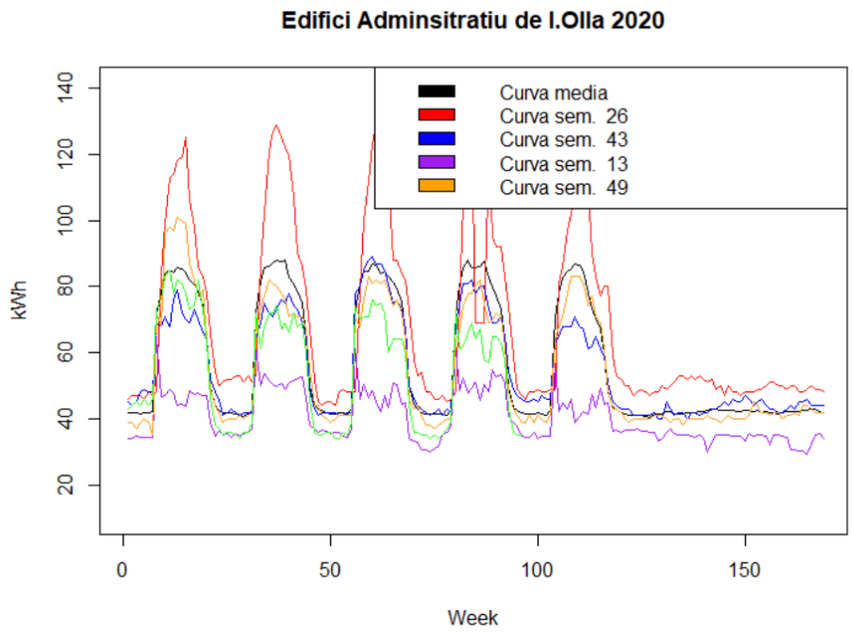
- **Forecasting:** prediction of events through a sequence of time. It predicts future events by analyzing the trends of the past, on the assumption that future trends will hold similar to historical trends.

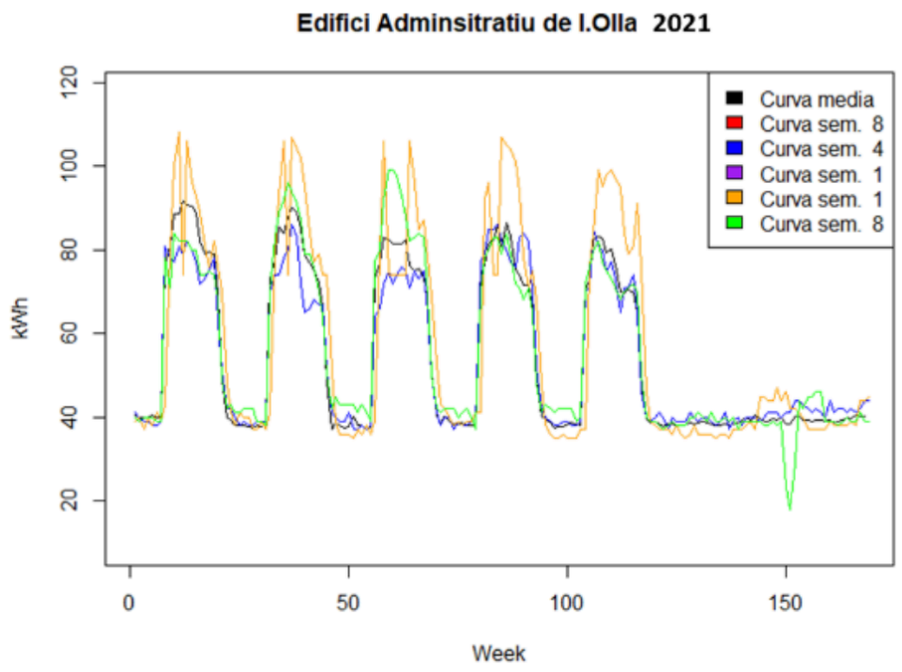
- **Classification**: assign one or more categorical labels to a time series.
  For instance, given the time series of activity of a visitor on a website, classify whether the visitor is a bot or a human.

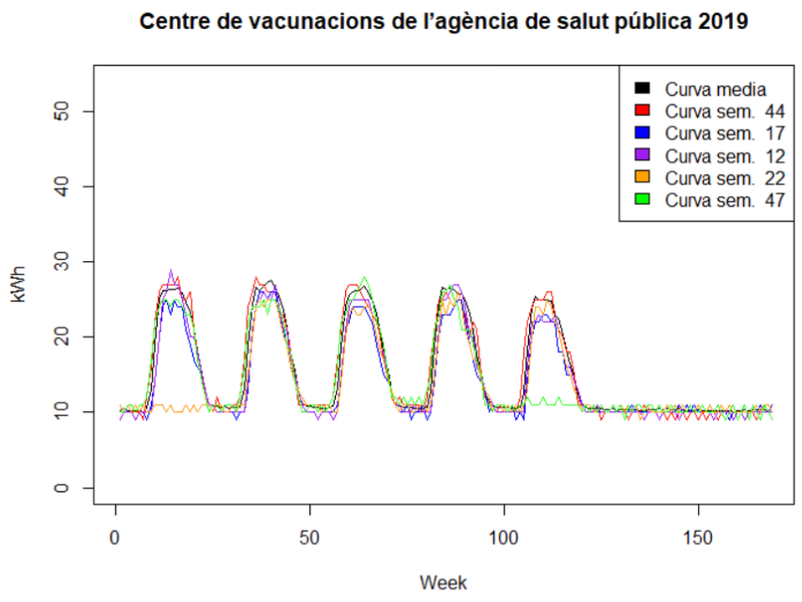
- **Event detection**: identify the occurrence of a specific, expected event within a continuous data stream.
  A particularly useful application is “hot word detection", where a model monitors a audio stream and detects utterances like "Ok Google" or "Hey Alexa”.

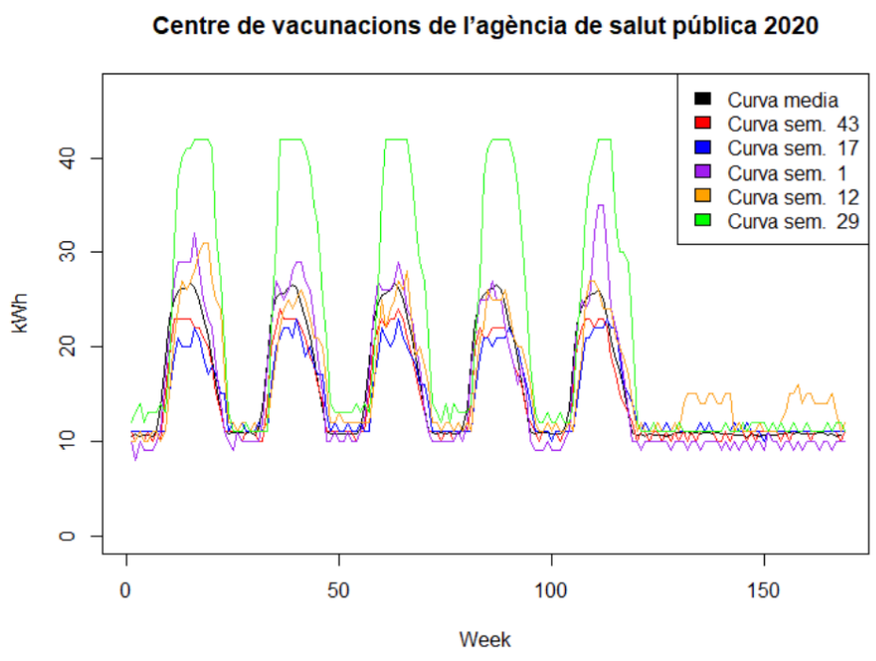
- **Anomaly detection**: detect anything unusual happening within a continuous data stream. Unusual activity on your corporate network? Might be an attacker. Unusual readings on a manufacturing line? Time for a human to go take a look. Anomaly detection is typically done via unsupervised learning, because you often don’t know what kind of anomaly you’re looking for, and thus you can’t train on specific anomaly examples.

Time Series Analysis









Time Series Analysis

Autoregressive model $$y_{t+1} = f(y_{1,\ldots,t})$$

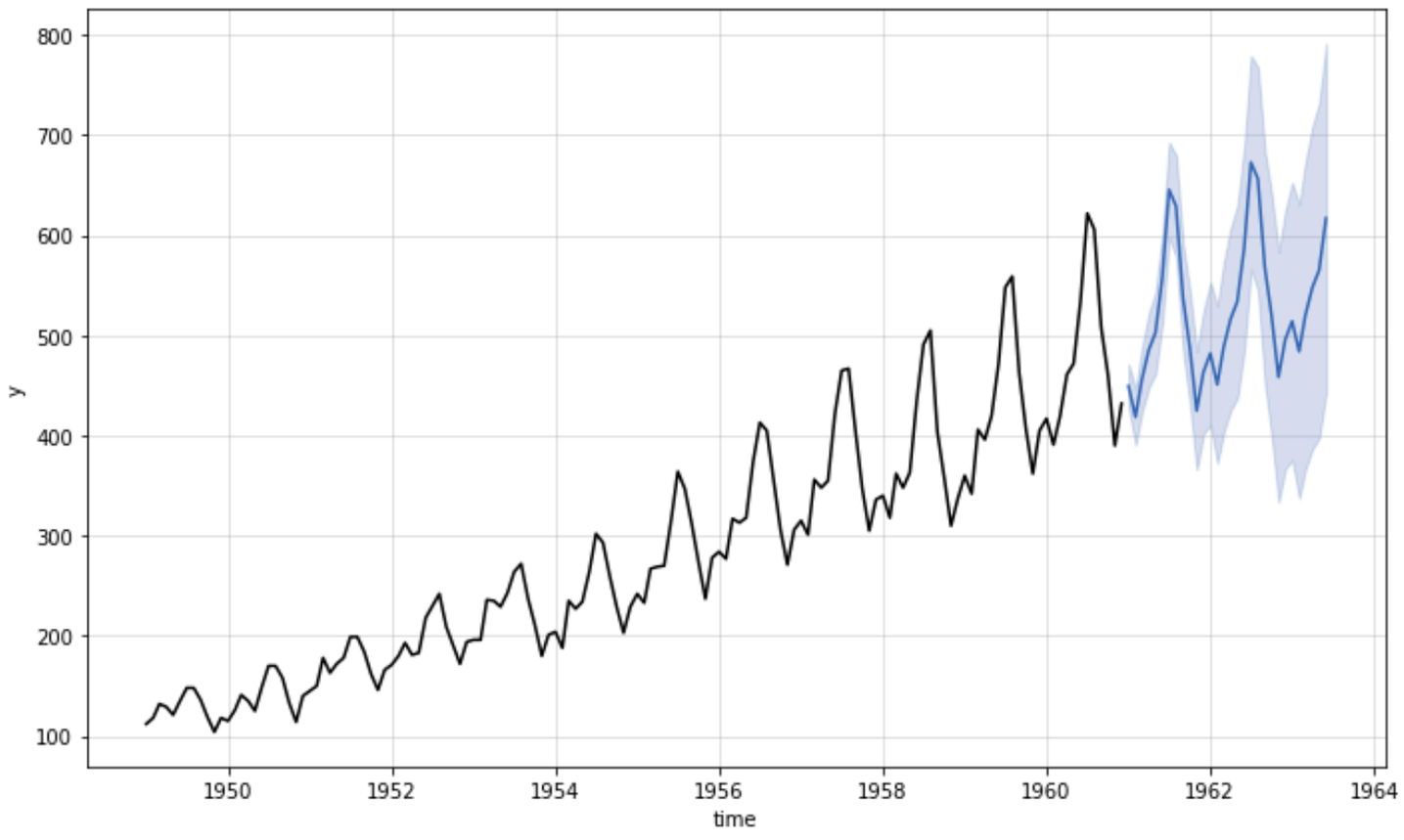
Typical case in Deep Learning $$y_{t+1} = f(x_{1,\ldots,t+1}, y_{1,\ldots,t})$$

Multiple Times Series $$y_{t+1} = f(x^{1,\ldots,k}_{1,\ldots,t+1}, y^{1,\ldots,k}_{1,\ldots,t})$$

x represents (possibly multimensional) covariate data that is useful for predicting.

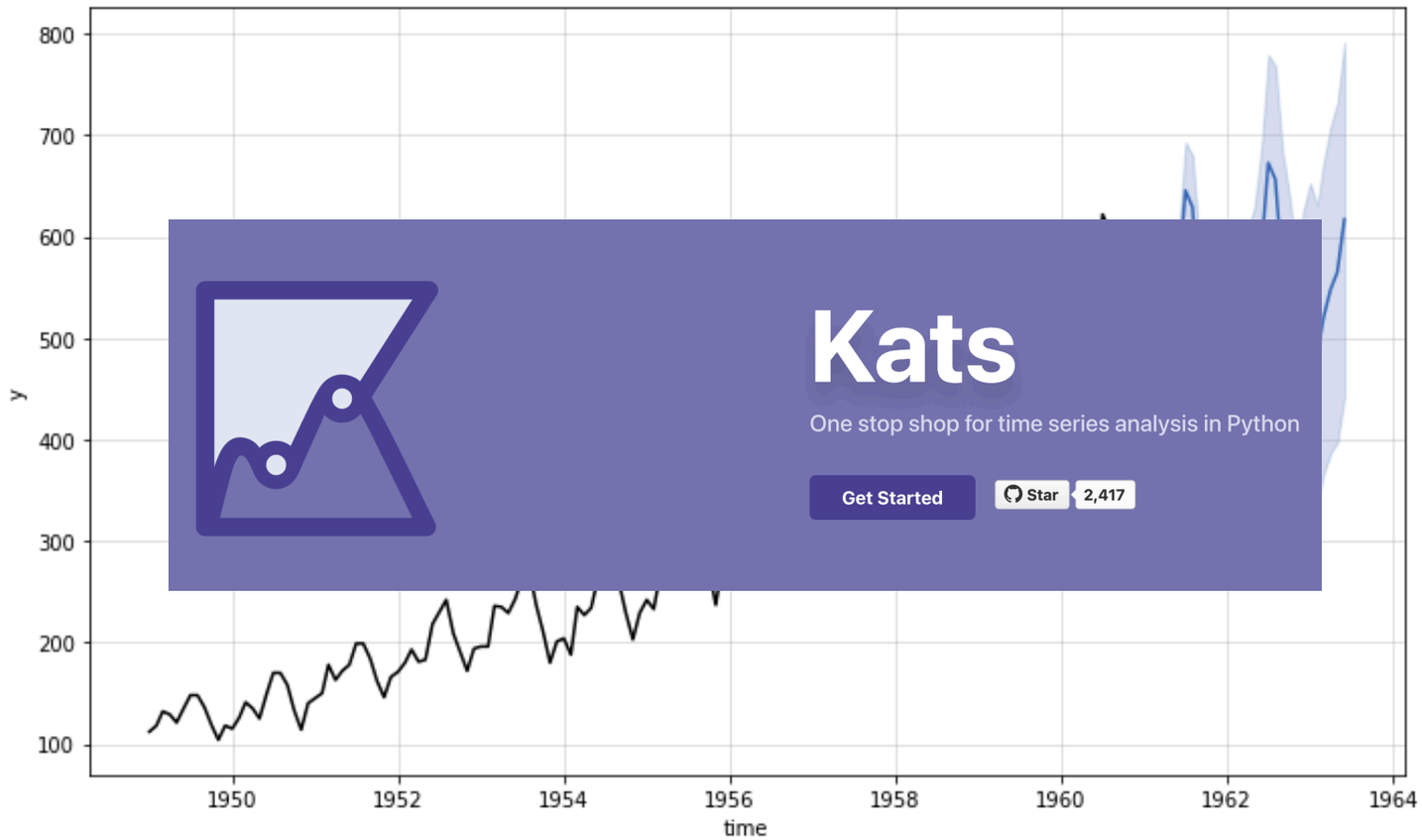
Time Series Analysis

Autoregressive model $y_{t+1} = f(y_{1,\ldots,t})$



Time Series Analysis

Numerical autoregressive model $y_{t+1} = f(y_{1,\ldots,t})$



Time Series Analysis

52,700 Catalan names

Categorical autoregressive model $y_{t+1} = f(y_{1,\ldots,t})$

Time Series Analysis

52,700 Catalan names

- Alzinetes, torrent de les
- Alzinetes, vall de les
- **Alzinó, Mas d'**
- Alzinosa, collada de l'
- Alzinosa, font de l'

-

- Benavent, roc de
- Benaviure, Cal
- **Benca**
- Bendiners, pla de
- Benedi, roc del

-

- Fiola, la
- Fiola, puig de la
- **Fiper, Granja del**
- Firassa, Finca
- Firell

-

- Regueret, lo
- Regueret, lo
- **Regueró**
- Reguerols, els
- Reguerons, els

-

- Vallverdú, Mas de
- Vallverdú, serrat de
- **Vallvicamanyà**
- Vallvidrera
- Vallvidrera, riera de

-

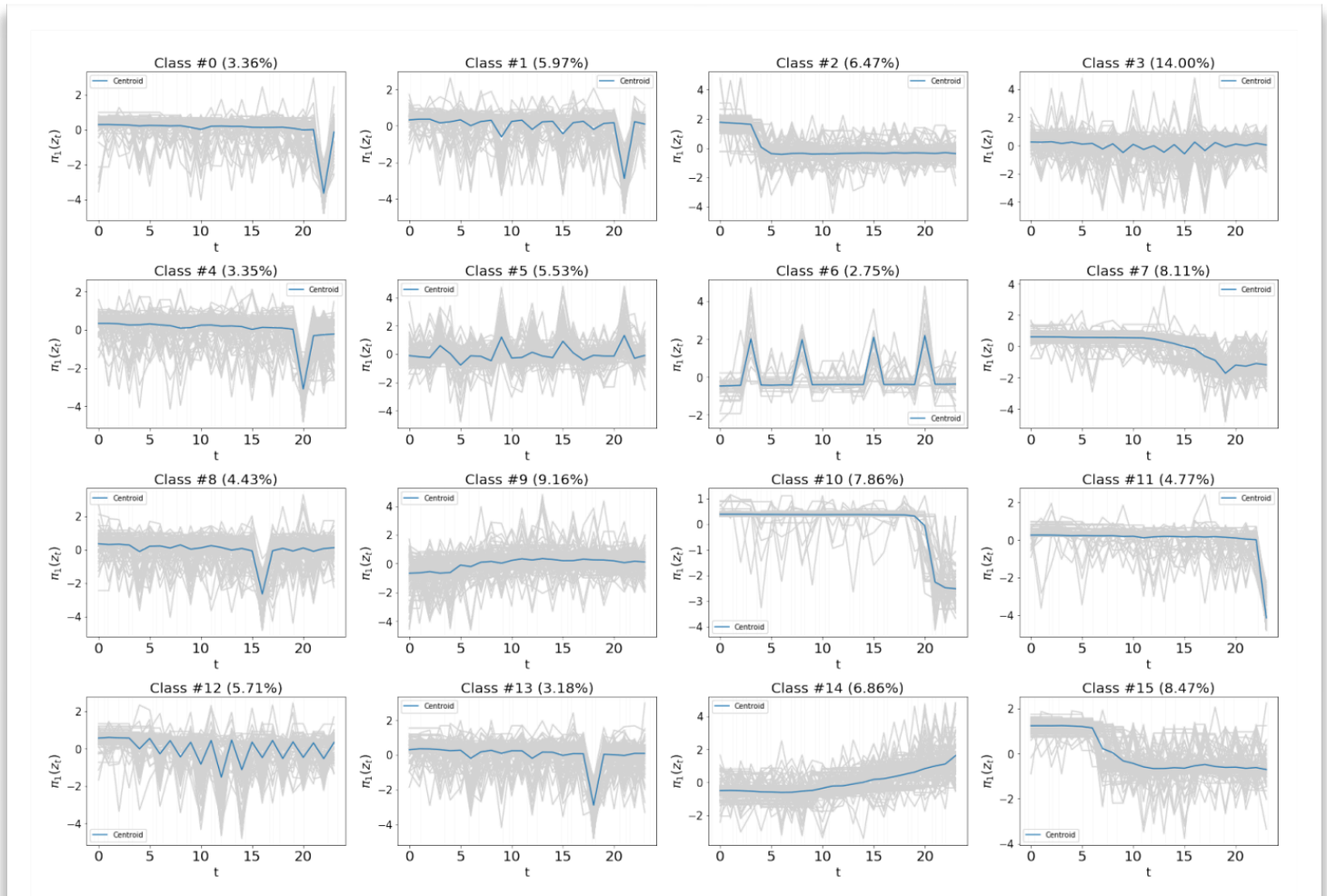
- Terraubella, Corral de
- Terraubes
- **Terravanca**
- Terrer Nou, Can
- Terrer Roig, lo

Name generator from a few letters

Time Series Analysis

Multiple Times Series

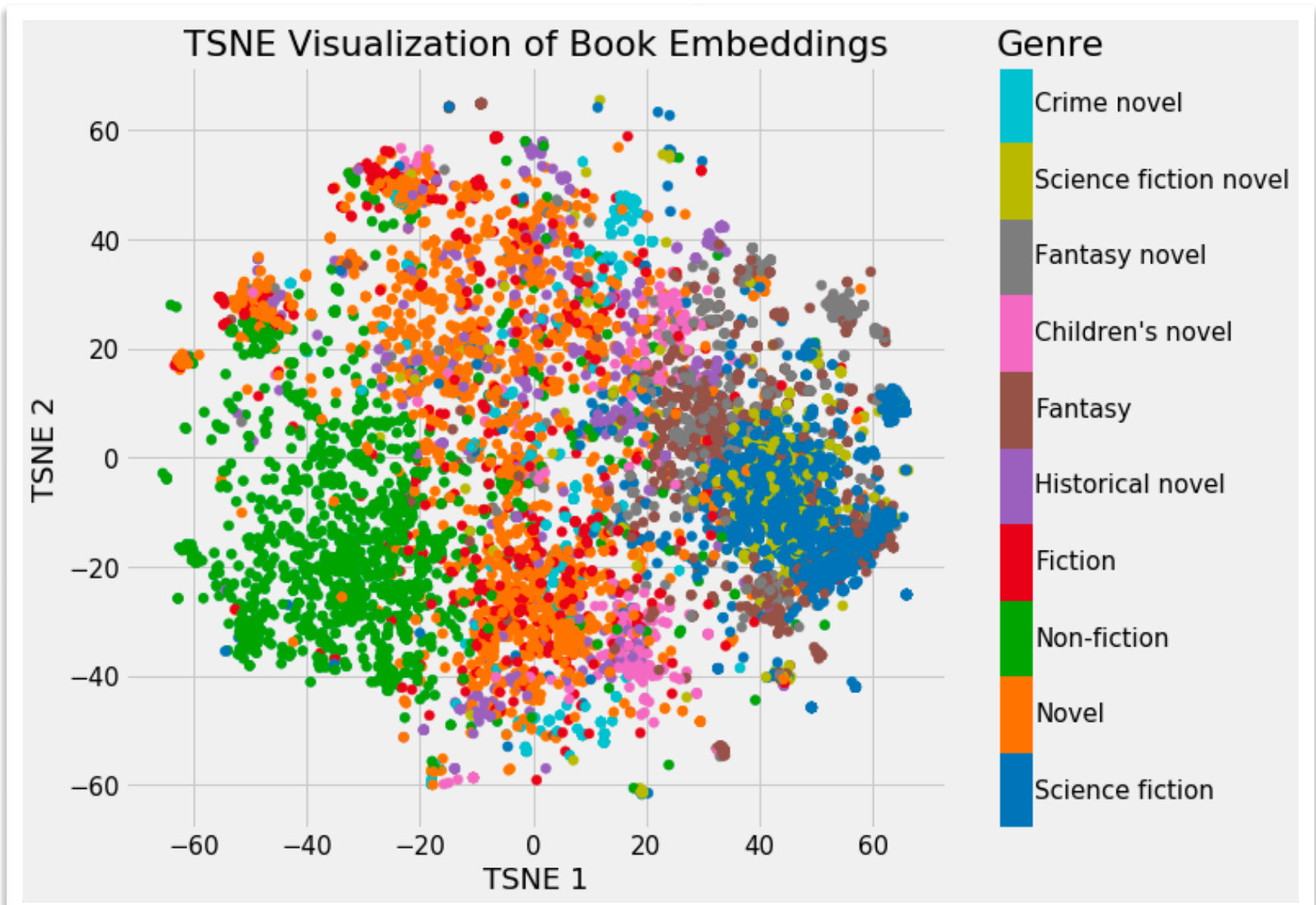
$$y_{t+1} = f(x^{1,\ldots,k}_{1,\ldots,t+1}, y^{1,\ldots,k}_{1,\ldots,t})$$



Embeddings

Embedding

An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors.
Ideally, an embedding captures some of the semantics of the input by placing **semantically similar inputs close together** in the embedding space.



Item representation: One-hot encoding

Words in documents and other categorical features such as **user/product ids** in recommenders, **names of places**, visited **URL**s, etc. are usually represented by using a one-of-K scheme (**one-hot encoding**).

If we represent every English word as an $\mathbb{R}^{|V|\times 1}$ vector with all 0's and one 1 at the index of that word in the sorted english language, word vectors in this type of encoding would appear as the following:

$$w^{aardvark} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{a} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \ldots, w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

There are an estimated 13 million tokens for the English language. The dimensionality of these vectors is huge!

Item representation: One-hot encoding

One hot encoding represents each word as a completely **independent** entity:

$$(w^{hotel})^T w^{motel} = (w^{hotel})^T w^{cat} = 0$$

But words are not independent (from the point of view of their meaning)!

What other alternatives are there?

Self-supervised pre-training of word vectors/embeddings.

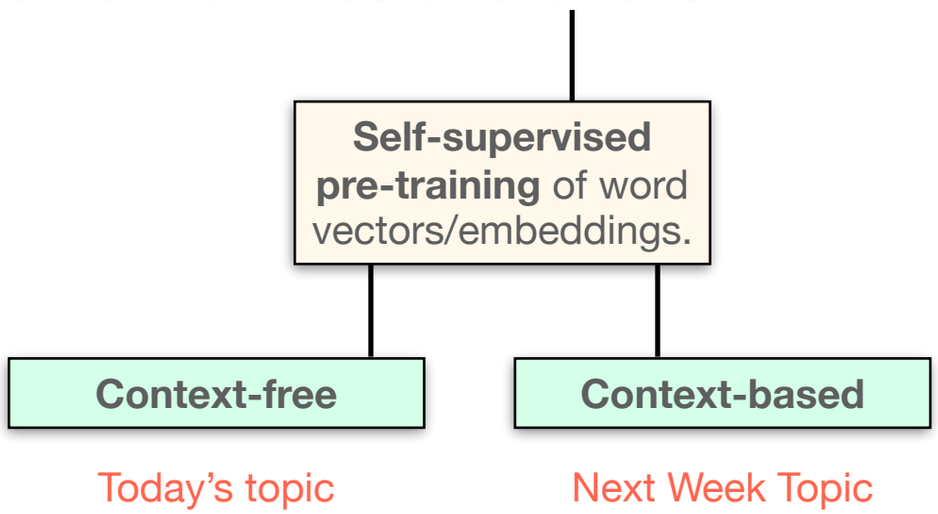
Item representation: One-hot encoding

One hot encoding represents each word as a completely independent entity:

$$(w^{hotel})^T w^{motel} = (w^{hotel})^T w^{cat} = 0$$

But words are not independent (from the point of view of their meaning)!

What other alternatives are there?



Embedding

Embeddings can be thought of as an alternative to one-hot encoding vectors along with dimensionality reduction.

Insight:
In order to use one-hot encoding within a machine learning system, we can represent each sparse vector as a dense vector of numbers **so that semantically similar items** (movies, words, products, etc.) **have similar representations in the new vector space**.

Problems:

- How can we learn a mapping between items and dense vectors? (*mathematical* question)?
- Do we need a task-dependent or a task-independent representation?

Embedding

Let's consider **linear embedding maps:**

Embedding matrix

$$[0,0,1]^T \cdot \begin{bmatrix} 0.29572738 & 0.88443109 & 0.21831979 \\ 0.03157878 & 0.71250614 & 0.22703532 \\ 0.12386669 & 0.74266196 & 0.91580261 \end{bmatrix} = \begin{bmatrix} 0.21831979 \\ 0.22703532 \\ 0.91580261 \end{bmatrix}$$

You can learn a **linear embedding mapping** (that maps indices to real valued vectors) for your data in two different ways:

- By considering the elements of the embedding matrix as **parameters** and optimizing their value with respect to a loss function that represents a task (f.e. classifying the polarity of a tweet).
- By "learning" a set of elements of the embedding matrix that are "good" for a large series of tasks.

Keras embedding layer

The Keras embedding layer receives a sequence of **non-negative integer indices** and learns to embed those into a high dimensional vector (the size of which is specified by output dimension).

$$[[4], [20]] \text{ is turned into } [[0.25, 0.1], [0.6, -0.2]]$$

This layer can only be used as the first layer in a model (after the input layer). **It learns a representation that is optimal for a task.**

```
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# summarize the model
print(model.summary())
```

Keras embedding layer

Lets see how to do it using restaurant reviews data.

```
# Define 10 restaurant reviews
reviews =[
          'Never coming back!',
          'horrible service',
          'rude waitress',
          'cold food',
          'horrible food!',
          'awesome',
          'awesome services!',
          'rocks',
          'poor work',
          'couldn\'t have done better'
]
#Define labels
labels = array([1,1,1,1,1,0,0,0,0,0])

Vocab_size = 50
encoded_reviews = [one_hot(d,Vocab_size) for d in reviews]
```

```
encoded reviews:

[[18, 39, 17], [27, 27],
[5, 19], [41, 29], [27,
29], [2], [2, 1], [49],
[26, 9], [6, 9, 11, 21]]
```

Keras embedding layer

Lets see how to do it using restaurant reviews data.

```
max_length = 4
padded_reviews = pad_sequences(encoded_reviews,maxlen=max_length,padding='post')
print(padded_reviews)
```

```
padded encoded reviews:

[[18 39 17  0]
 [27 27  0  0]
 [ 5 19  0  0]
 [41 29  0  0]
 [27 29  0  0]
 [ 2  0  0  0]
 [ 2  1  0  0]
 [49  0  0  0]
 [26  9  0  0]
 [ 6  9 11 21]]
```

Keras embedding layer

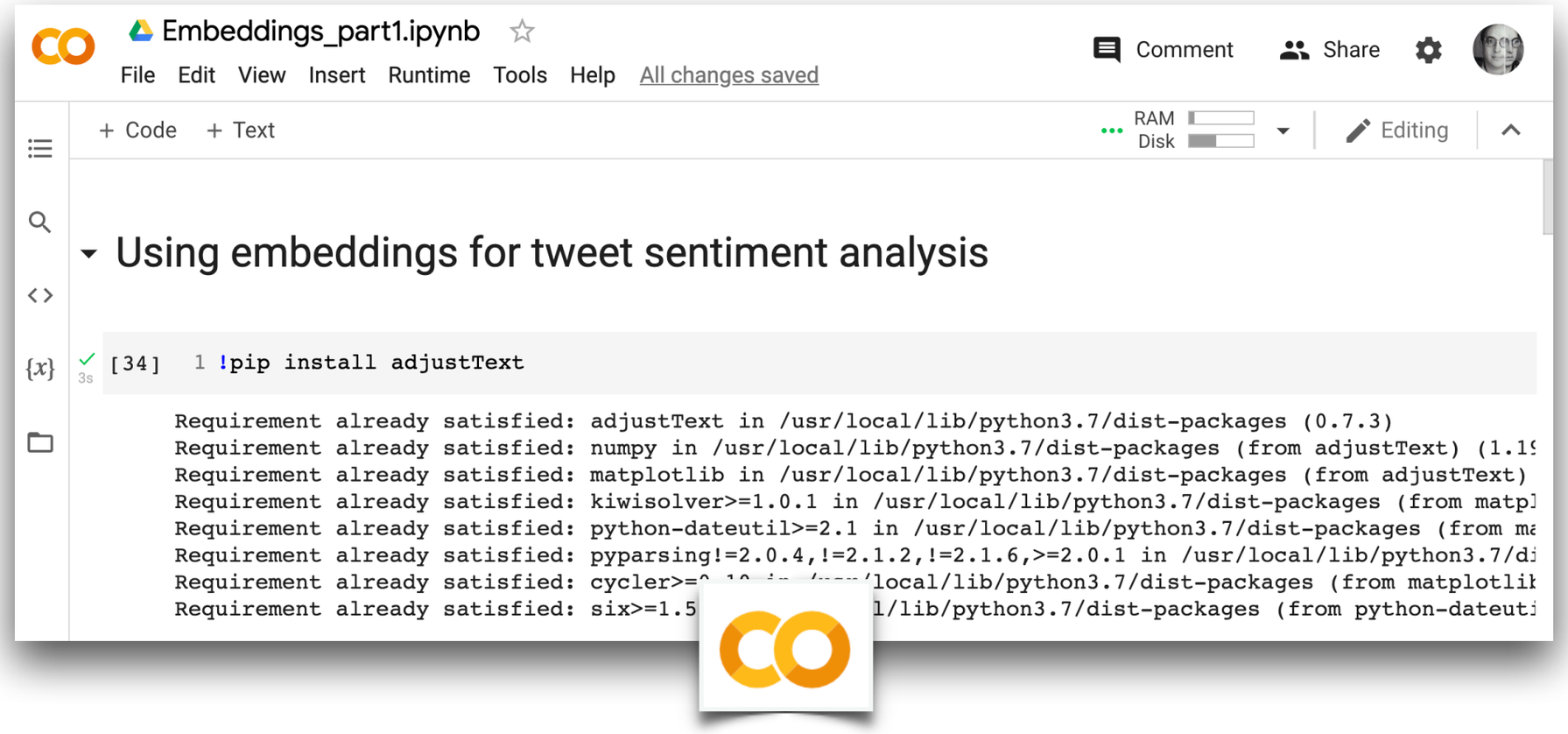
Lets see how to do it using restaurant reviews data.

```
model = Sequential()
embedding_layer = Embedding(input_dim=Vocab_size,output_dim=8,input_length=max_length)
model.add(embedding_layer)
model.add(Flatten())
model.add(Dense(1,activation='sigmoid'))
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['acc'])
```

This embedding matrix is essentially a lookup table of 50 rows and 8 columns.

```
[0] -> [ 0.056933 0.0951985 0.07193055 0.13863552 -0.13165753 0.07380469 0.10305451 -0.10652688]
```

Embeddings and Sentiment Analysis



Distributional and compositional semantics

Linguistics assumes two important hypotheses:

1. **Distributional Hypothesis**: words that occur in the same contexts tend to have similar meanings (Harris, 1954).

government debt problems turning into **banking** crises as has happened in
These words will represent banking
These words will represent banking
saying that Europe needs unified **banking** regulation to replace the hodgepodge

The meaning of a word is determined by its contexts

2. **Compositionality**: Semantic complex entities can be built from its simpler constituents by using **compositional** rules

```
morphomes > words > sentences > paragraphs > text
```

Word models.

Researchers have developed various techniques for **training** general purpose word models (**vector/embedding models**) using the enormous piles of unannotated text on the web (this is known as **pre-training**).

Assumption: These general purpose pre-trained models are general and can then be **fine-tuned** on smaller task-specific datasets, e.g., when working with problems like question answering and sentiment analysis or even be used as it.

Word models.

Word models can either be **context-free** or **context-based**.

Context-free models learn **a single** word embedding representation (a **vector** of numbers) for each word in the vocabulary.

On the other hand, **context-based** models generate (at training or at inference time) a representation of each word that **is based on the other words in the sentence**.

> Context-based representations can then be unidirectional or bidirectional. For example, in the sentence "*I accessed the bank account*," a unidirectional contextual model would represent "*bank*" based on "*I accessed the*" but not "*account*."

Context-free Word Embeddings

Simply put, context-free **word embeddings** are a way to represent, in a mathematical space, words which "live" in similar contexts in a real-world collection of text, otherwise known as a **text corpus**.

Word embeddings take the notion of distributional similarity, with words simply mapped to their company and stored in vector spaces.

These vectors can then be used across a **wide range of natural language understanding tasks**.

F.e. Analyzing word co-ocurrences by using SVD

Some previous methods were based on **counting** co-ocurrences of words, but today the most successful are those based on **prediction**:

$y = f(x)$

Instead of counting how often each word y occurs near x, train a classifier on a binary prediction task: Is y likely to show up near x?

x

government debt problems turning into **banking** crises as has happened in

y y

Context-free Word Embeddings

Simply put, context-free **word embeddings** are a way to represent, in a mathematical space, words which "live" in similar contexts in a real-world collection of text, otherwise known as a **text corpus**.

Word embeddings take the notion of distributional similarity, with words simply mapped to their company and stored in vector spaces.

These vectors can then be used across a **wide range of natural language understanding tasks**.

F.e. Analyzing word co-ocurrences by using SVD

Some previous methods were based on **counting** co-ocurrences of words, but today the most successful are those based on **prediction**:

$y = f(x)$

Instead of counting how often each word y **occurs near** x**, train a classifier on a binary prediction task: Is** y **likely to show up near** x**?**

y

government debt problems turning into **banking** crises as has happened in

x x

Word2Vec

The **context of a word** is the set of m surrounding words.

For instance, the $m = 2$ context of the word `fox` in the sentence

`The quick brown fox jumped over the lazy dog`

is `quick`, `brown`, `jumped`, `over`.

The idea is to design a model whose parameters are the word vectors. Then, train the (discriminative) model on a certain objective.

Mikolov presented a simple, probabilistic model in 2013 that is known as **word2vec**. In fact, **word2vec** includes 2 algorithms (**CBOW** and **skip-gram**).

Continuous Bag of Words Model (CBOW)

The approach is to treat {`The`,`cat`, `over`,`the`, `puddle`} as a context and from these words, be able to predict or generate the center word `jumped`.

First, we set up our known parameters. Let the known parameters in our model be the sentence represented by **one-hot word vectors**.

- The **input** one-hot vectors or context will be represented with an $x^{(c)}$.
- And the **output** as y which is the one hot vector of the unknown center word.

Continuous Bag of Words Model (CBOW)

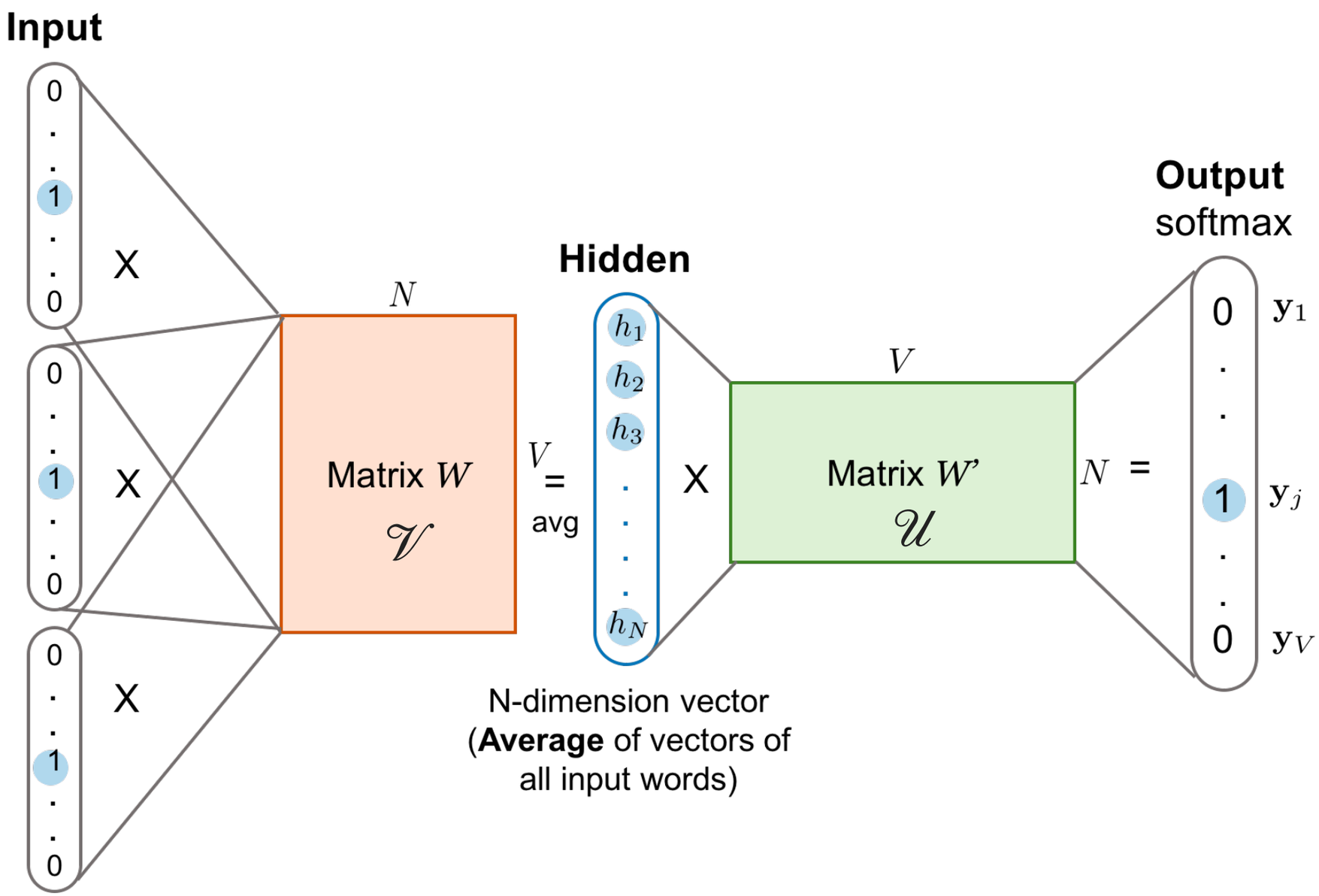
We create two matrices, $\mathscr{V} \in \mathbb{R}^{n \times |V|}$ and $\mathscr{U} \in \mathbb{R}^{|V| \times n}$, where n is an arbitrary size which defines the size of our embedding space.

$\mathscr{V}$ is the input word matrix such that the i-th column of $\mathscr{V}$ is the n-dimensional embedded vector for word w_i when it is an input to this model. We denote this $n \times 1$ vector as v_i.

Similarly, $\mathscr{U}$ is the output word matrix. The j-th row of $\mathscr{U}$ is an n-dimensional embedded vector for word w_j when it is an output of the model. We denote this row of $\mathscr{U}$ as u_j.

Note that we do in fact learn two vectors for every word w_i (i.e. input word vector v_i and output word vector u_i).

Continuous Bag of Words Model (CBOW)

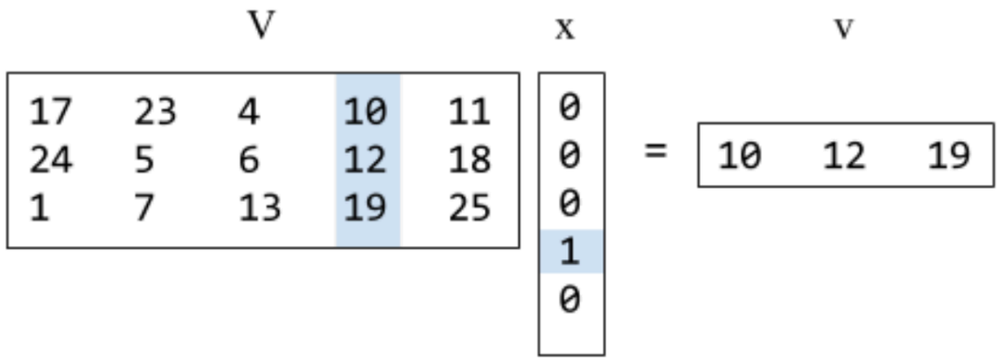


https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html#loss-functions

Continuous Bag of Words Model (CBOW)

We breakdown the way this model works in these steps:

- We generate our one hot word vectors for the input context of size m : $(x^{(c-m)}, \ldots, x^{(c-1)}, x^{(c+1)}, \ldots, x^{(c+m)} \in \mathbb{R}^{|V|})$.
- We get our embedded word vectors for the context ($v_{c-m} = \mathcal{V}x^{(c-m)}, v_{c-m+1} = \mathcal{V}x^{(c-m+1)}, \ldots, v_{c+m} = \mathcal{V}x^{(c+m)} \in \mathbb{R}^n$)



- Average these vectors to get $h = \frac{v_{c-m}+v_{c-m+1}+\ldots+v_{c+m}}{2m} \in \mathbb{R}^n$
- Generate a score vector $z = \mathcal{U}h \in \mathbb{R}^{|V|}$. As the dot product of similar vectors is higher, it will push similar words close to each other in order to achieve a high score.
- Turn the scores into probabilities $\hat{y} = \text{softmax}(z) \in \mathbb{R}^{|V|}$.
- We desire our probabilities generated, $\hat{y} \in \mathbb{R}^{|V|}$, to match the true probabilities, $y \in \mathbb{R}^{|V|}$, which also happens to be the one hot vector of the actual word.

Continuous Bag of Words Model (CBOW)

How can we learn these two matrices?

Well, we need to create an **objective function**. We choose a classification loss function. Here, we use a popular choice: *cross entropy*, **as a proxy function of the conditional probability** $P(u_c \mid h)$.

We thus formulate our optimization objective as:

$$\text{minimize } J = -\log P(w_c \mid w_{c-m}, \ldots, w_{c-1}, w_{c+1}, \ldots, w_{c+m})$$

$$= -\log P(u_c \mid h)$$

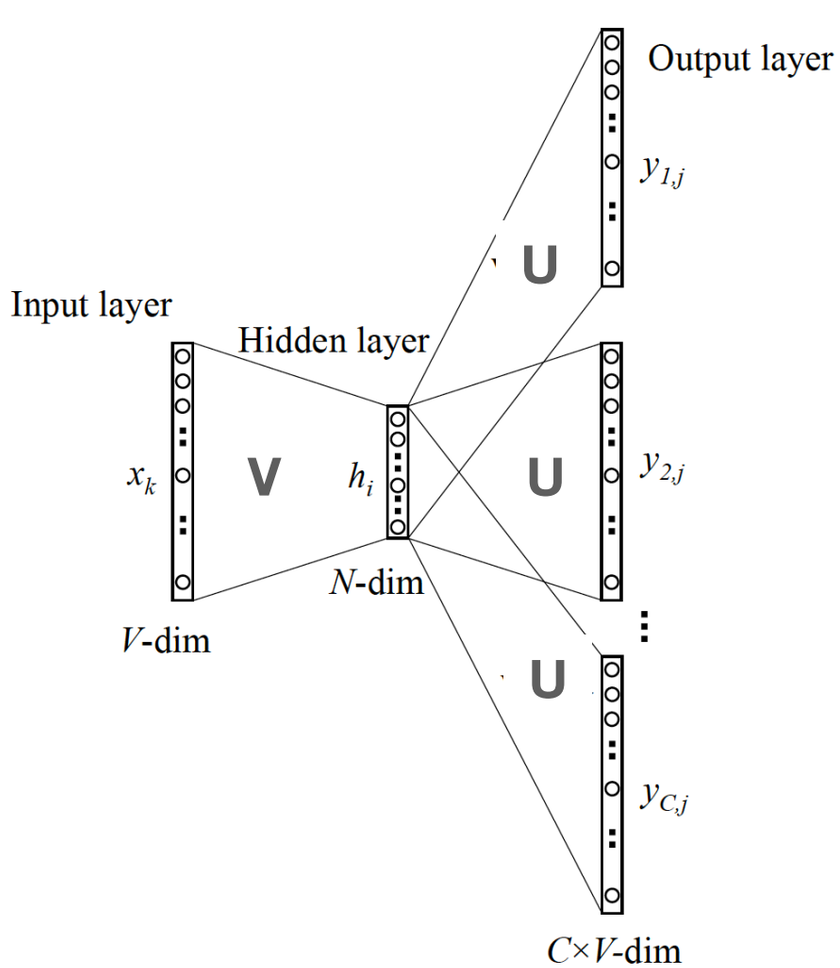
$$= -\log \frac{\exp(u_c^T h)}{\sum_{j=1}^{|V|} \exp(u_j^T h)}$$

$$= -u_c^T h + \log \sum_{j=1}^{|V|} \exp(u_j^T h)$$

Skip-gram model

Another approach, the **skip-gram model,** is to create a model such that given the center word `jumped`, the model will be able to predict or generate the surrounding words `The`, `cat`, `over`, `the`, `puddle`.

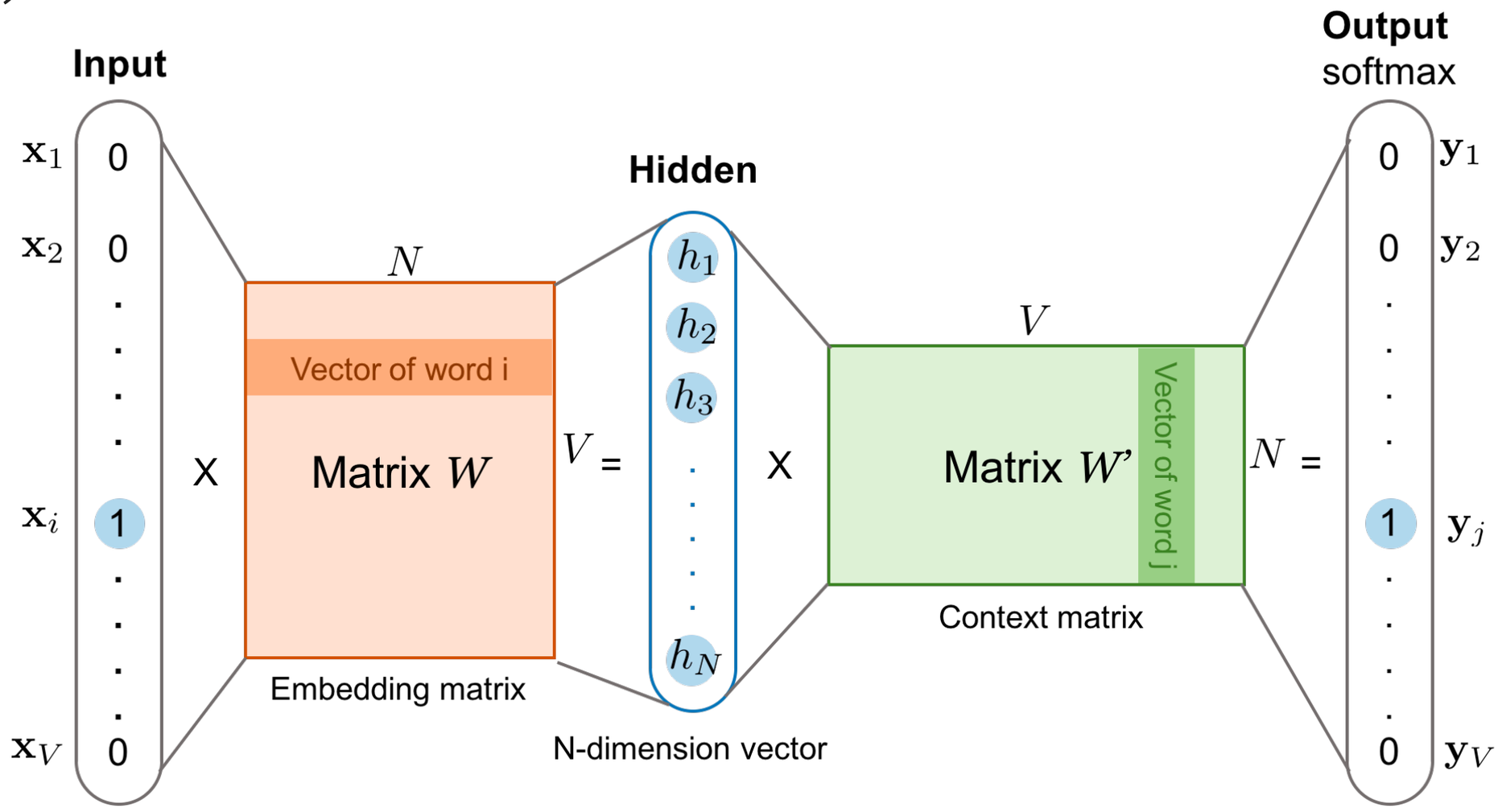
Here we call the word `jumped` the context. We call this type of model a **Skip-Gram model**.



Skip-gram model

In fact, we can consider that **each context-target pair** is treated as a new observation in the data.

For example, the target word “swing” in the above case produces four training samples: (“*swing*”, “*sentence*”), (“*swing*”, “*should*”), (“*swing*”, “*the*”), and (“*swing*”, “*sword*”).

- 

https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html#loss-functions

Skip-gram model

We thus formulate our optimization objective as:

$$\text{minimize } J = -\log P(w_{c-m}, \ldots, w_{c-1}, w_{c+1}, \ldots, w_{c+m} \mid w_c)$$

$$= -\log \prod_{j=0; j \neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|} \exp(u_k^T v_c)}$$

$$= -\log \prod_{j=0; j \neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{k=1}^{|V|} \exp(u_k^T v_c)$$

Skip-gram model

We thus formulate our optimization objective as:

$$\text{minimize } J = -\log P(w_{c-m}, \ldots, w_{c-1}, w_{c+1}, \ldots, w_{c+m} \mid w_c)$$

$$= -\log \prod_{j=0; j \neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|} \exp(u_k^T v_c)}$$

$$= -\log \prod_{j=0; j \neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{k=1}^{|V|} \exp(u_k^T v_c)$$

Negative Sampling

The summation over $|V|$ is computationally huge!

> Any update we do or **evaluation of the objective function** would take $O(|V|)$ time which if we recall is in the millions.

A simple idea is we could instead just **approximate** it.

> For every training step, instead of looping over the entire vocabulary, we can just sample several negative examples!

But instead of this, we can also optimize a different objective function (**Negative Sampling Loss**) that takes into account this fact...

(See https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html#loss-functions)

Example

Let's try a little experiment in word embedding extracted from "the Games of Thrones corpus".

Extract words:

```
import sys
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize

STOP_WORDS = set(stopwords.words('english'))

def get_words(txt):
    return filter(
        lambda x: x not in STOP_WORDS,
        re.findall(r'\b(\w+)\b', txt)
    )

def parse_sentence_words(input_file_names):
   """Returns a list of a list of words. Each sublist is a sentence."""
    sentence_words = []
    for file_name in input_file_names:
        for line in open(file_name):
            line = line.strip().lower()
            line = line.decode('unicode_escape').encode('ascii','ignore')
            sent_words = map(get_words, sent_tokenize(line))
            sent_words = filter(lambda sw: len(sw) > 1, sent_words)
            if len(sent_words) > 1:
                sentence_words += sent_words
    return sentence_words

# You would see five .txt files after unzip 'a_song_of_ice_and_fire.zip'
input_file_names = ["001ssb.txt", "002ssb.txt", "003ssb.txt",
                    "004ssb.txt", "005ssb.txt"]
GOT_SENTENCE_WORDS= parse_sentence_words(input_file_names)
```

Example

Let's try a little experiment in word embedding extracted from "the Games of Thrones corpus".

Feed a word2vec model:

```
from gensim.models import Word2Vec

# size: the dimensionality of the embedding vectors.
# window: the maximum distance between the current and predicted word within a sentence.
model = Word2Vec(GOT_SENTENCE_WORDS, size=128, window=3, min_count=5, workers=4)
model.wv.save_word2vec_format("got_word2vec.txt", binary=False)
```

Example

Let's try a little experiment in word embedding extracted from "the Games of Thrones corpus".

Check the results:

`model.most_similar('king', topn=10)`	`model.most_similar('queen', topn=10)`
(word, similarity with 'king')	(word, similarity with 'queen')
('kings', 0.897245)	('cersei', 0.942618)
('baratheon', 0.809675)	('joffrey', 0.933756)
('son', 0.763614)	('margaery', 0.931099)
('robert', 0.708522)	('sister', 0.928902)
('lords', 0.698684)	('prince', 0.927364)
('joffrey', 0.696455)	('uncle', 0.922507)
('prince', 0.695699)	('varys', 0.918421)
('brother', 0.685239)	('ned', 0.917492)
('aerys', 0.684527)	('melisandre', 0.915403)
('stannis', 0.682932)	('robb', 0.915272)

Limitations

- **Low probability words** are represented by one symbol [UNK]. How to deal with rare words, names, etc?

- Always the same representation for a word type regardless of the context in which a word token occurs. We do not have fine-grained **word sense disambiguation**.

- We just have one representation for a word, but words have **different aspects**, including semantics, syntactic behavior, and register/ connotations.

Example

We can load a pre-trained word embeddings matrix into an Embedding layer!

Keras

Search Keras documentation...

Star 52,540

» Code examples / Natural Language Processing / Using pre-trained word embeddings

- About Keras
- Getting started
- Developer guides
- Keras API reference
- **Code examples**
  - Computer Vision
  - Natural Language Processing
  - Structured Data
  - Timeseries
  - Audio Data
  - Generative Deep Learning

Using pre-trained word embeddings

Author: fchollet
Date created: 2020/05/05
Last modified: 2020/05/05
Description: Text classification on the Newsgroup20 dataset using pre-trained GloVe word embeddings.

View in Colab • GitHub source

Setup

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

Using pre-trained word embeddings

Example

Embeddings, Transformers and Transfer Learning

Using transformer embeddings like BERT in spaCy

spaCy supports a number of **transfer and multi-task learning** workflows that can often help improve your pipeline's efficiency or accuracy. Transfer learning refers to techniques such as word vector tables and language model pretraining. These techniques can be used to import knowledge from raw text into your pipeline, so that your models are able to generalize better from your annotated examples.

You can convert **word vectors** from popular tools like FastText and Gensim, or you can load in any pretrained **transformer model** if you install `spacy-transformers` </> . You can also do your own language model pretraining via the `spacy pretrain` command. You can even **share** your transformer or other contextual embedding model across multiple components, which can make long pipelines several times more efficient. To use transfer learning, you'll need at least a few annotated examples for what you're trying to predict. Otherwise, you could try using a "one-shot learning" approach using vectors and similarity.

What's the difference between word vectors and language models? ⊕

When should I add word vectors to my model? ⊕

How to learn word embeddings?

Embeddings_part2.ipynb ☆

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text Reconnect Editing

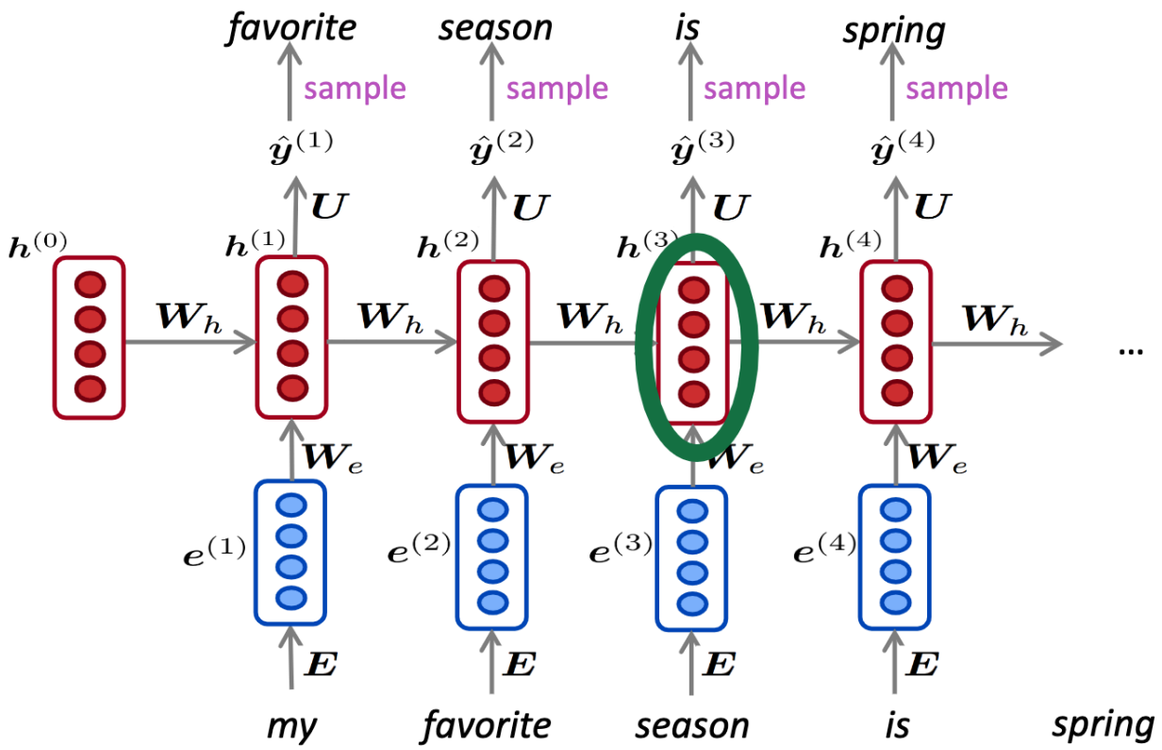
Product Embedding with Word2Vect

```
1 !pip install adjustText
```

```
Requirement already satisfied: adjustText in /usr/local/lib/python3.7/dist-packages (0.7.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from adjustText) (
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from adjustTe
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplc
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (frc
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from m
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dat
```

From context-free to context-based representations

Recurrent models can generate **context-dependent** word representations!



Limitations:

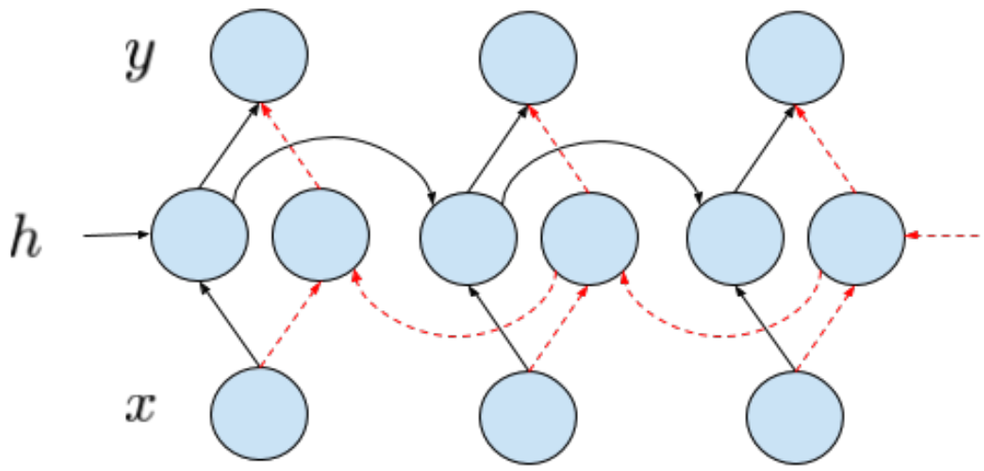
- This is a right-to-left model.
- This is task-dependent.

Bi-directional LSTM

Bidirectional LSTMs are an extension of traditional LSTMs that can improve model performance on sequence classification problems by extending left-to-right processing.

Bi-directional deep neural networs, at each time-step, t, maintain two hidden layers, one for the left-to-right propagation and another for the right-to-left propagation (hence, consuming twice as much memory space).

The final classification result, $\hat{y}$, is generated through combining the score results produced by both RNN hidden layers.



Bi-directional LSTM

The equations are (arrows are for designing left-to-right and right-to-left tensors):

$$\overrightarrow{h}_t = f(\overrightarrow{W}x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\overrightarrow{h}_t; \overleftarrow{h}_t] + c)$$

$[\overrightarrow{h}_t; \overleftarrow{h}_t]$ summarizes the past and future of a single element of the sequence.

Bidirectional RNNs can be stacked as usual!

Bi-directional LSTM

```
# Input for variable-length sequences of integers
inputs = keras.Input(shape=(None,), dtype="int32")
# Embed each integer in a 128-dimensional vector
x = layers.Embedding(max_features, 128)(inputs)
# Add 2 bidirectional LSTMs
x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)
x = layers.Bidirectional(layers.LSTM(64))(x)
# Add a classifier
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.summary()

```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding (Embedding)        (None, None, 128)         2560000
_________________________________________________________________
bidirectional (Bidirectional (None, None, 128)         98816
_________________________________________________________________
bidirectional_1 (Bidirection (None, 128)               98816
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 2,757,761
Trainable params: 2,757,761
Non-trainable params: 0
```

Bi-directional LSTM

```
# Input for variable-length sequences of integers
inputs = keras.Input(shape=(None,), dtype="int32")
# Embed each integer in a 128-dimensional vector
x = layers.Embedding(max_features, 128)(inputs)
# Add 2 bidirectional LSTMs
x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)
x = layers.Bidirectional(layers.LSTM(64))(x)
# Add a classifier
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.summary()

```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding (Embedding)        (None, None, 128)         2560000
_________________________________________________________________
bidirectional (Bidirectional (None, None, 128)         98816
_________________________________________________________________
bidirectional_1 (Bidirection (None, 128)               98816
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 2,757,761
Trainable params: 2,757,761
Non-trainable params: 0
```

Here we can use *Word2Vec* embeddings instead of learning task-specific vectors.

This could be a good word representation …

Large Language Models

Large Language Models

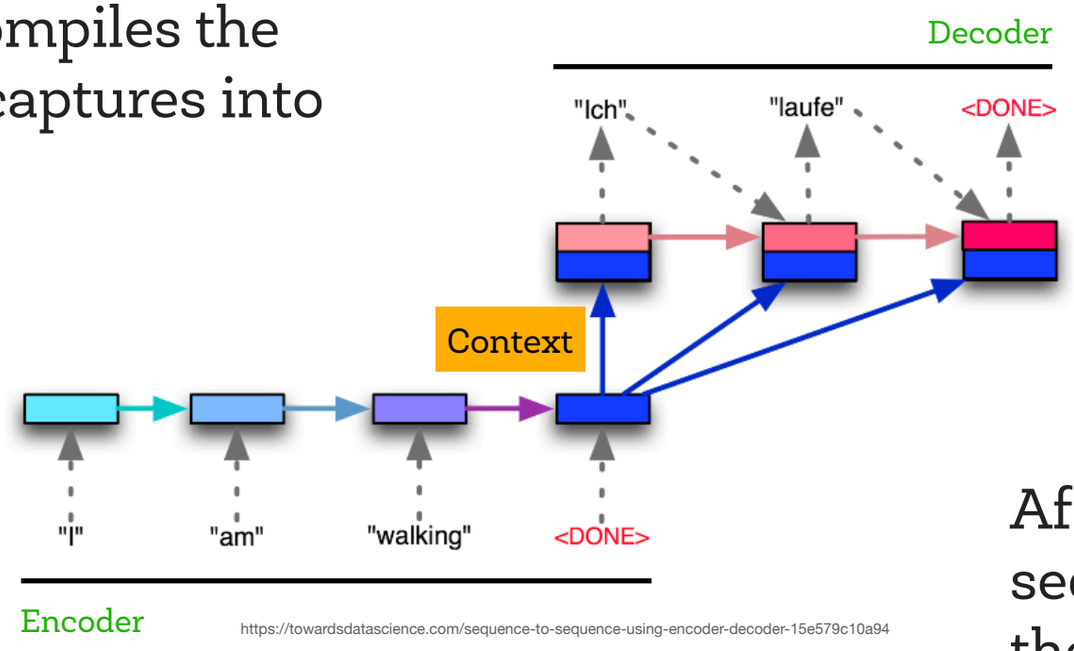
Large language models (LMM) are characterized by:

- **Attention** mechanisms.
- **Context-depending embeddings**.
- **Multitask**-training using huge amounts of text.
- **Parallel processing** instead of left-to-right or bidirectional models.

Attention

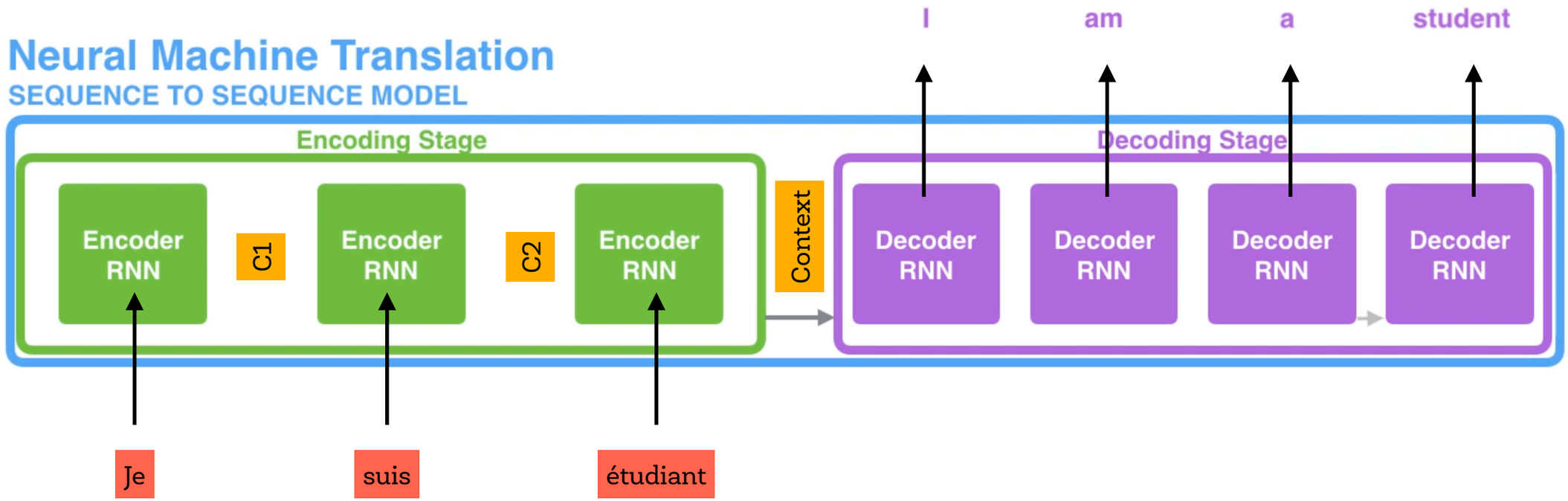
Let's consider that we have a **sequence-to-sequence recurrent model** that takes a sequence of items (words, letters, image frames, etc) and outputs another sequence of items.

The encoder compiles the information it captures into a vector (called the **context**).



After processing the entire input sequence, the encoder sends the **context** over to the **decoder**, which begins producing the output sequence item by item.

Attention

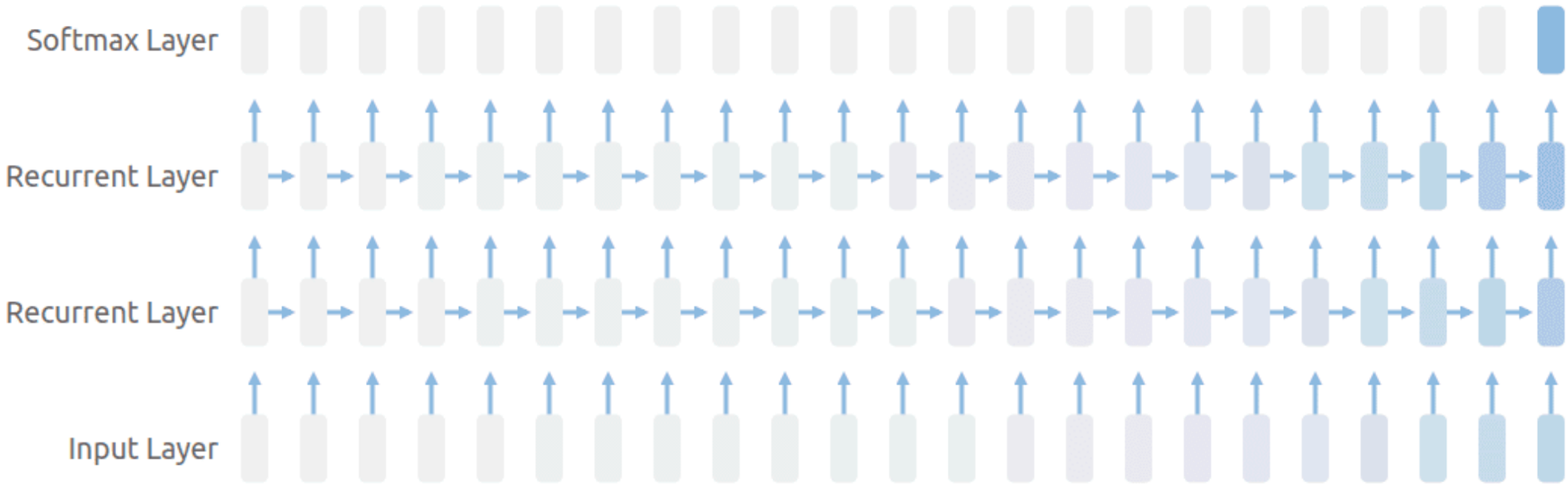


https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

The **context** vector turned out to be a bottleneck for these types of models. It made it challenging for the models to deal with **long sentences (vanishing gradient problem)**.

Attention

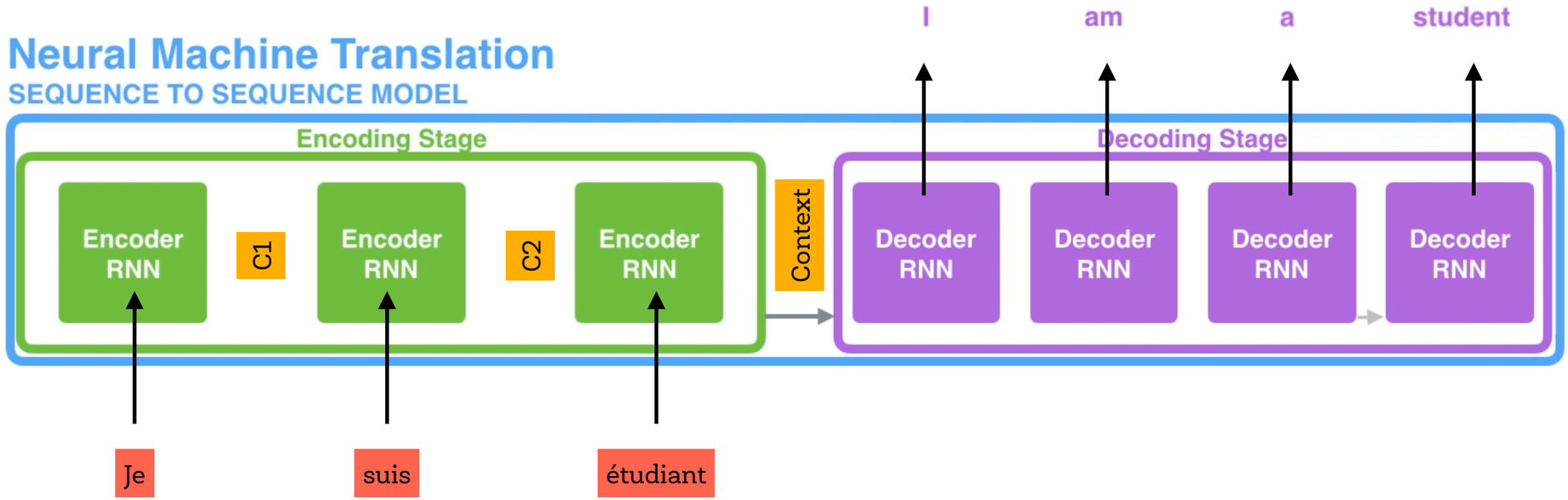
vanishing gradient problem



Vanishing Gradient: where the contribution from the earlier steps becomes insignificant in the gradient for the vanilla RNN unit.

https://theaisummer.com/attention/

Attention



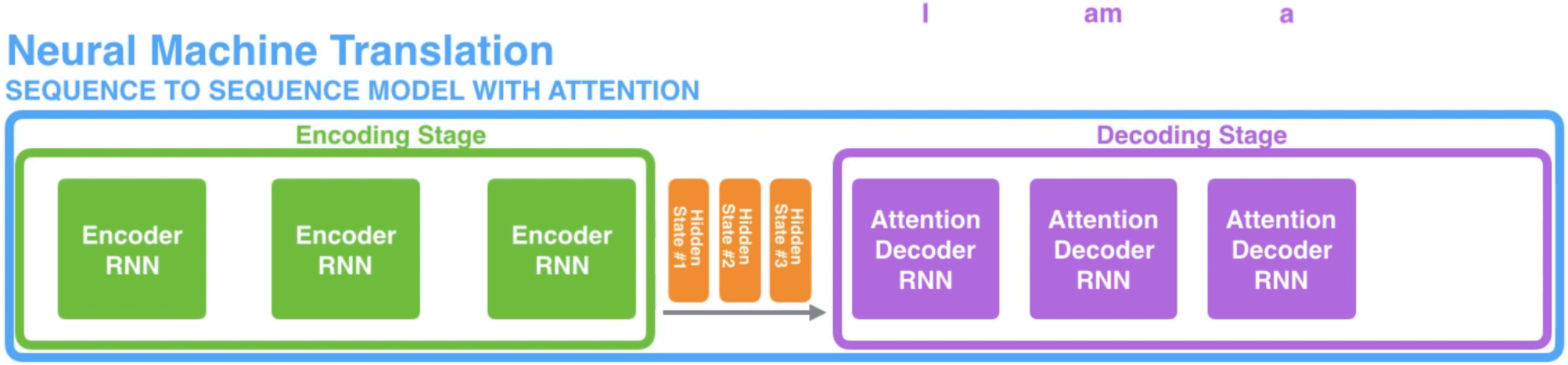
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

The **context** vector turned out to be a bottleneck for these types of models. It made it challenging for the models to deal with **long sentences**.

A solution was proposed in Bahdanau et al., 2014 and Luong et al., 2015. These papers introduced and refined a technique called "**Attention**", which allows the model to focus on the relevant parts of the input sequence as needed.

Attention

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**:



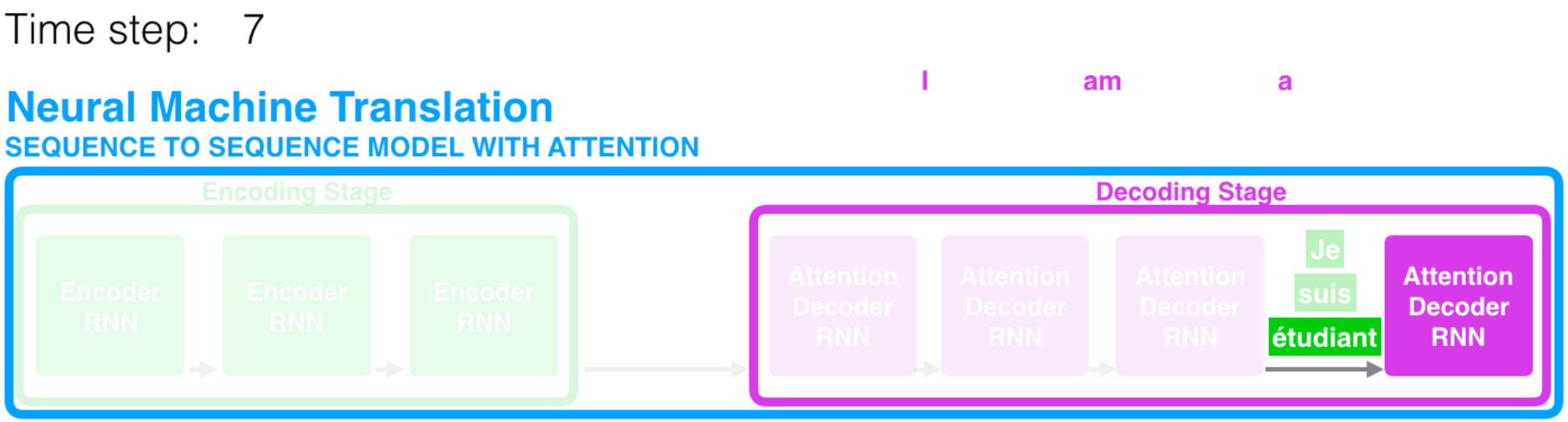
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Attention

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:

- **Look at the set of encoder hidden states it received (presumably, each encoder hidden states is most associated with a certain word in the input sentence).**
- **Give each hidden states a score (*let's ignore how the scoring is done for now*).**
- Multiply each hidden state by its *softmaxed* score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores.
- Build a new representation (**context state**) from these "weighted" encoder hidden states.

Attention



At time step 7, the attention mechanism enables the decoder to focus on the word "*étudiant*" before it generates the English translation.

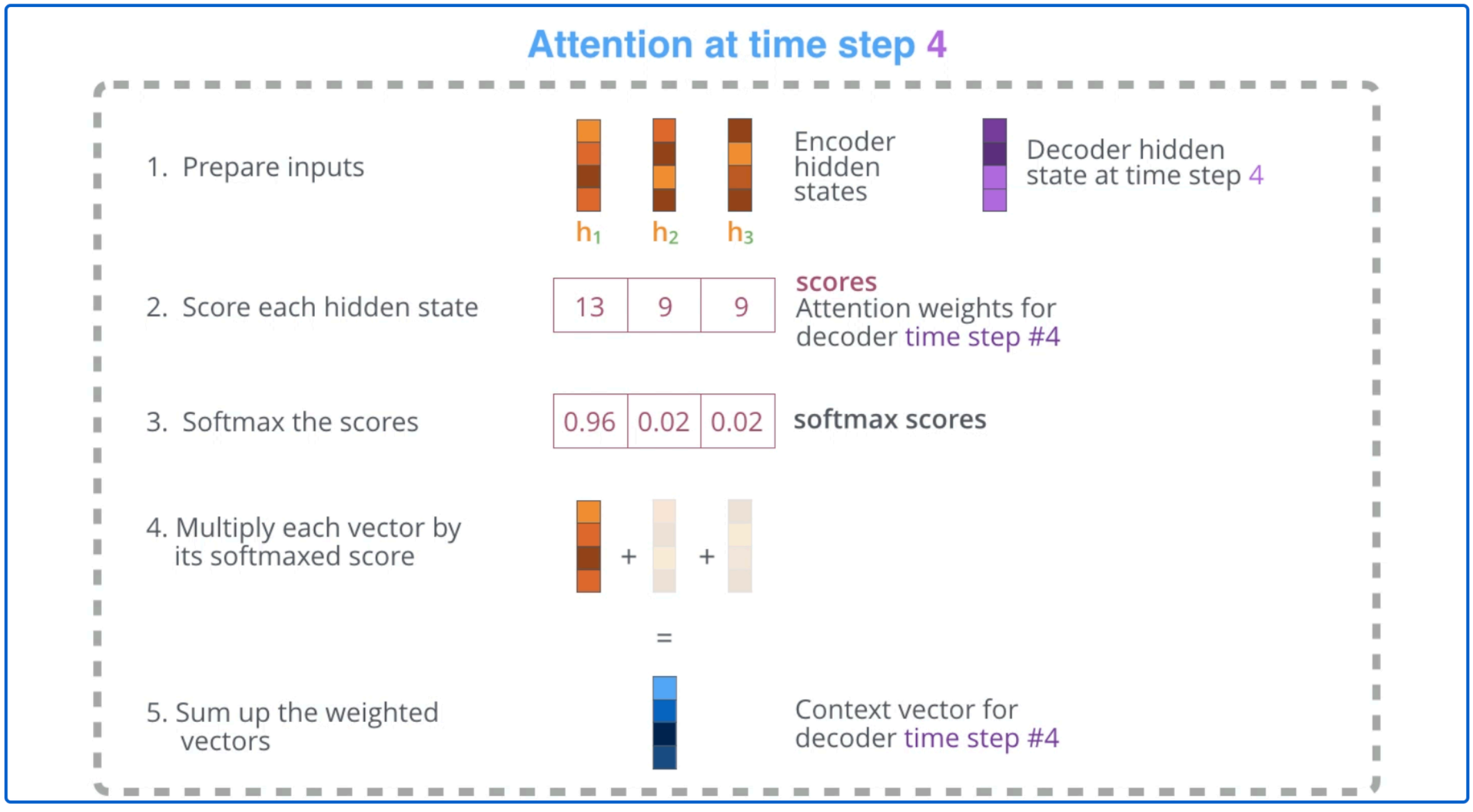
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Attention

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:

- Look at the set of encoder hidden states it received (presumably, each encoder hidden states is most associated with a certain word in the input sentence).
- Give each hidden states a **score** (*let's ignore how the scoring is done for now*)
- **Multiply each hidden state by its *softmaxed* score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores.**
- **Build a new representation (context state) from these "weighted" encoder hidden states.**

Attention



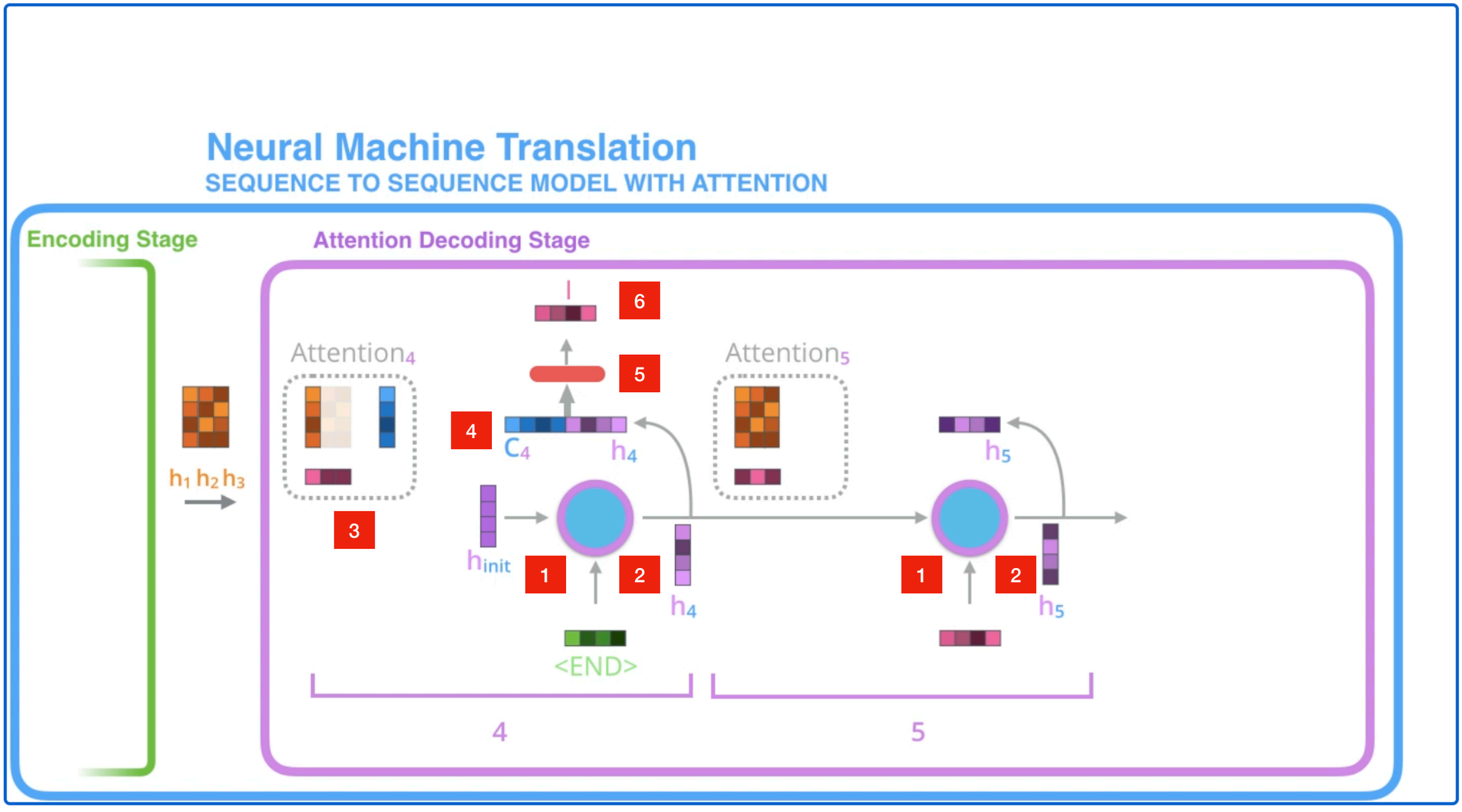
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Attention

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

1. The attention decoder RNN takes in the previous decoder hidden state and the mebedding of the last output.
2. The RNN processes these inputs to produce a new hidden state vector (*h*).
3. **(Attention Step)** We use the encoder hidden states and the *h* vector to calculate a context vector (*C*) for this time step.
4. We concatenate *h* and *C* 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.

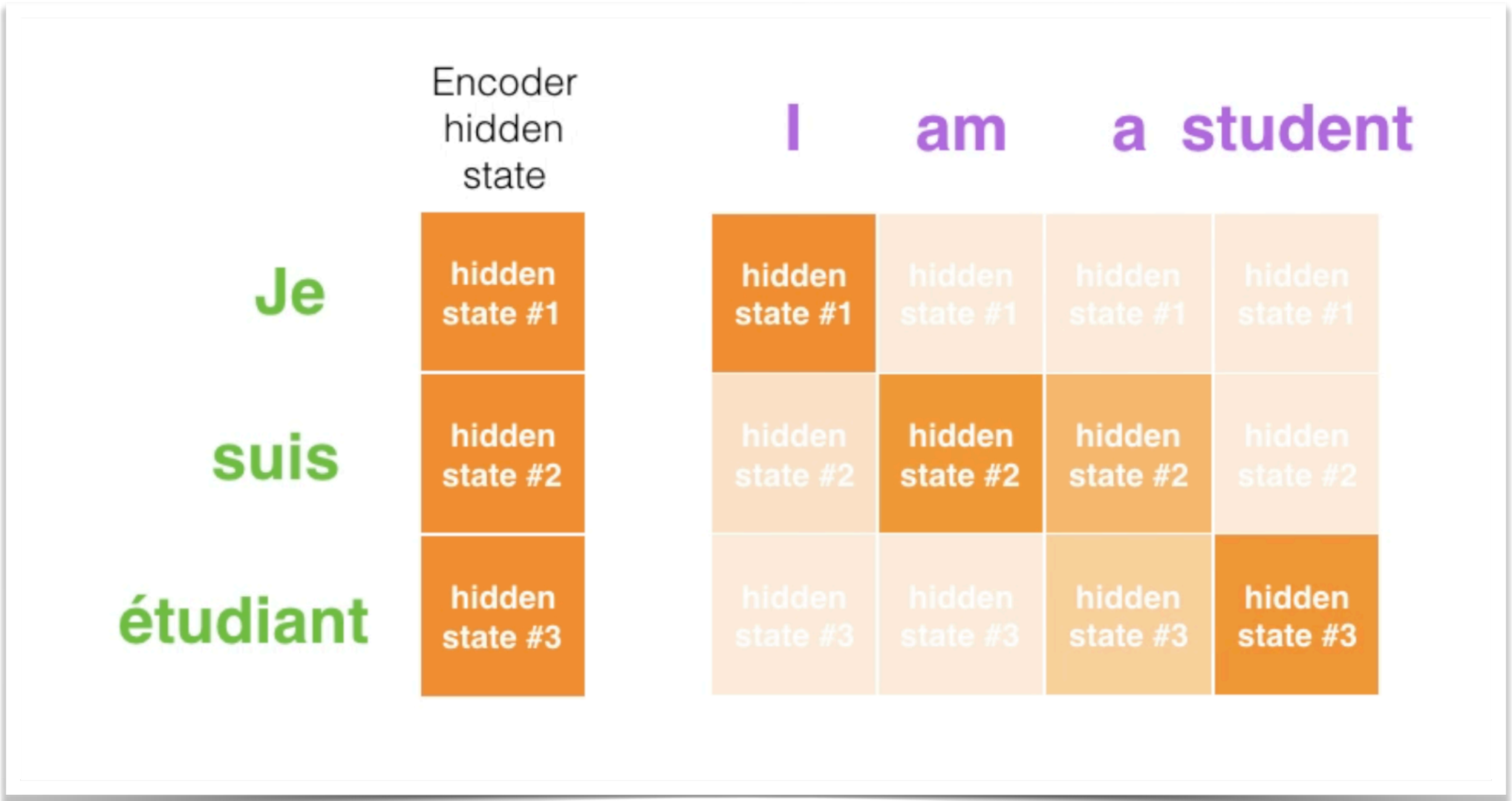
Attention



https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

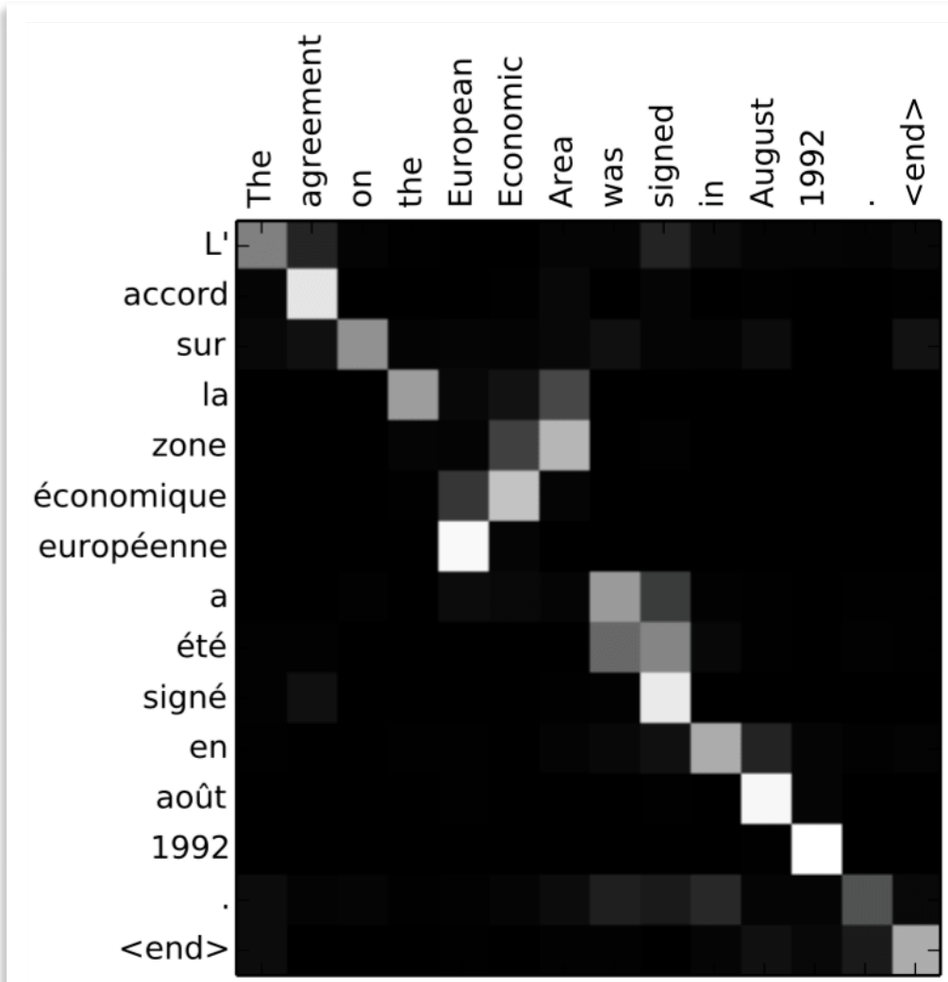
Attention

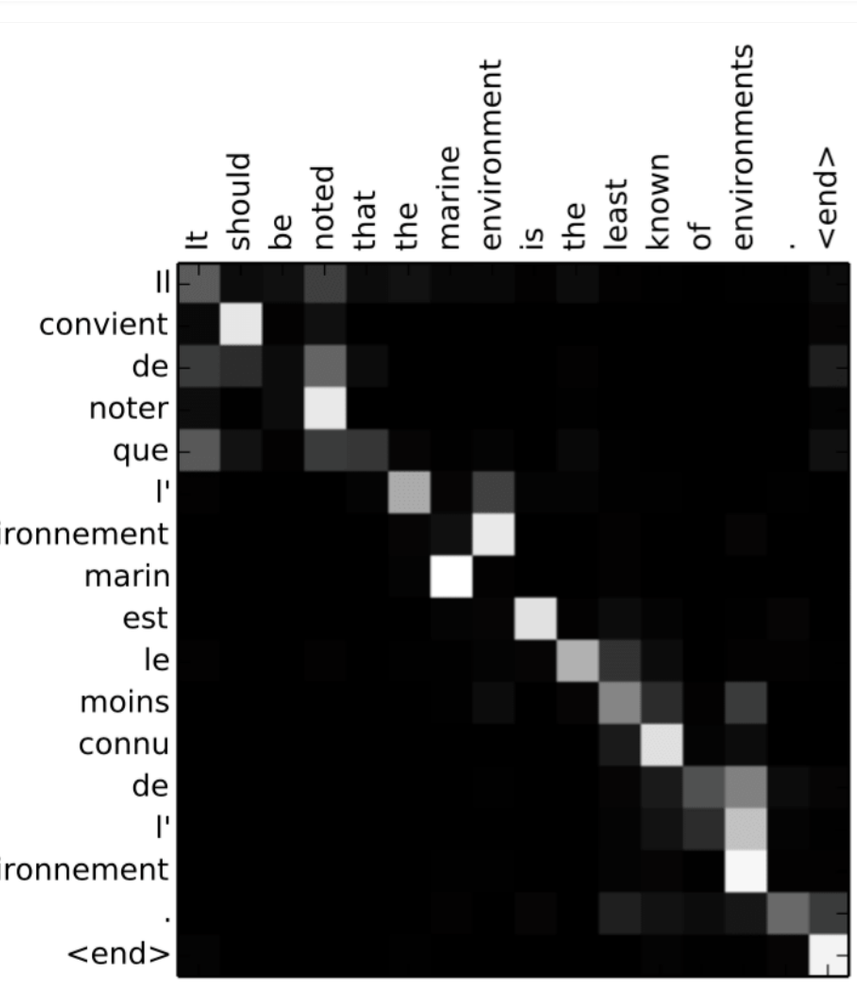
This is another way to look at which part of the input sentence we're paying attention to at each decoding step:



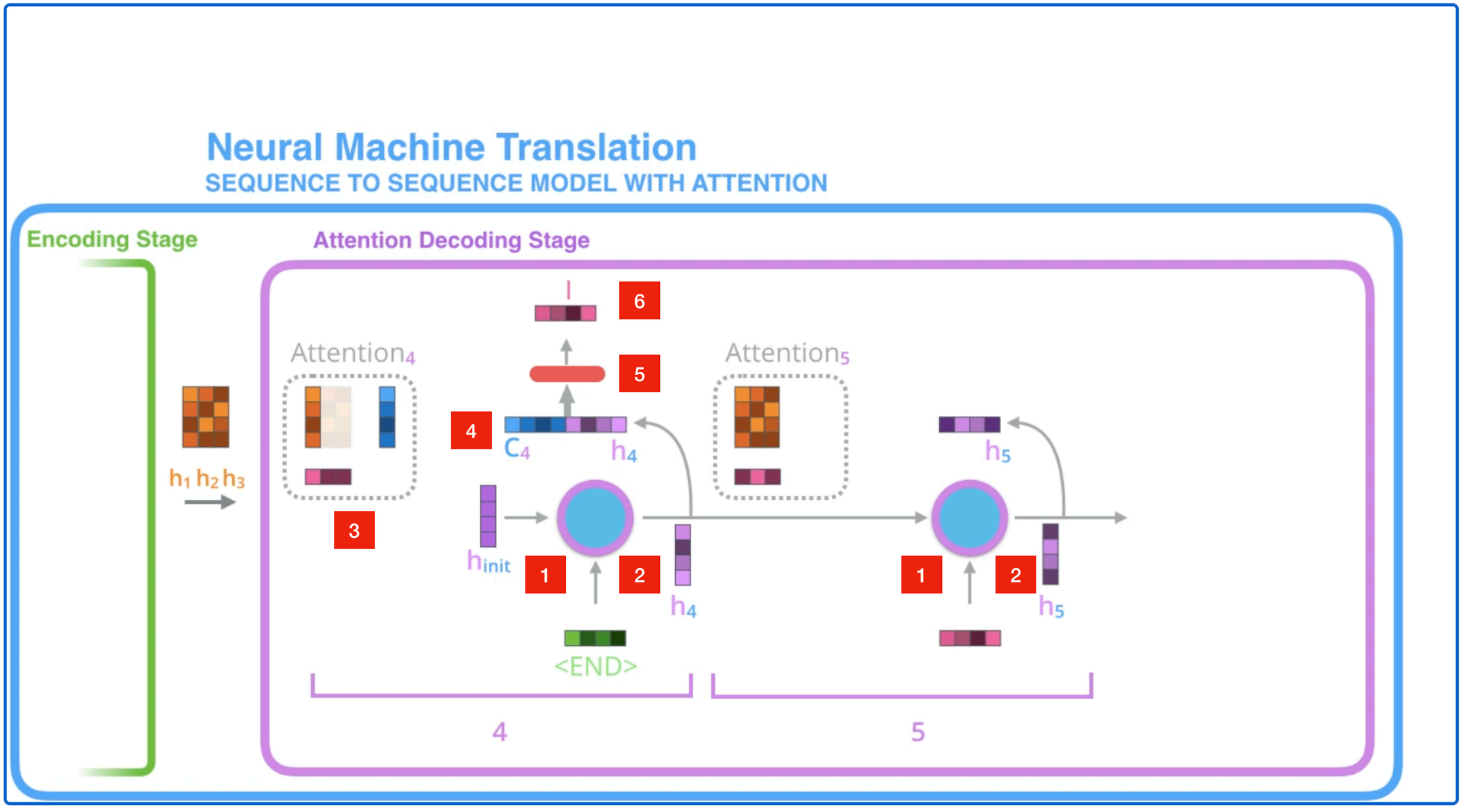
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Attention


The
agreement
on
the
European
Economic
Area
was
signed
in
August
1992
.
<end>
L'
accord
sur
la
zone
économique
européenne
a
été
signé
en
août
1992
.
<end>

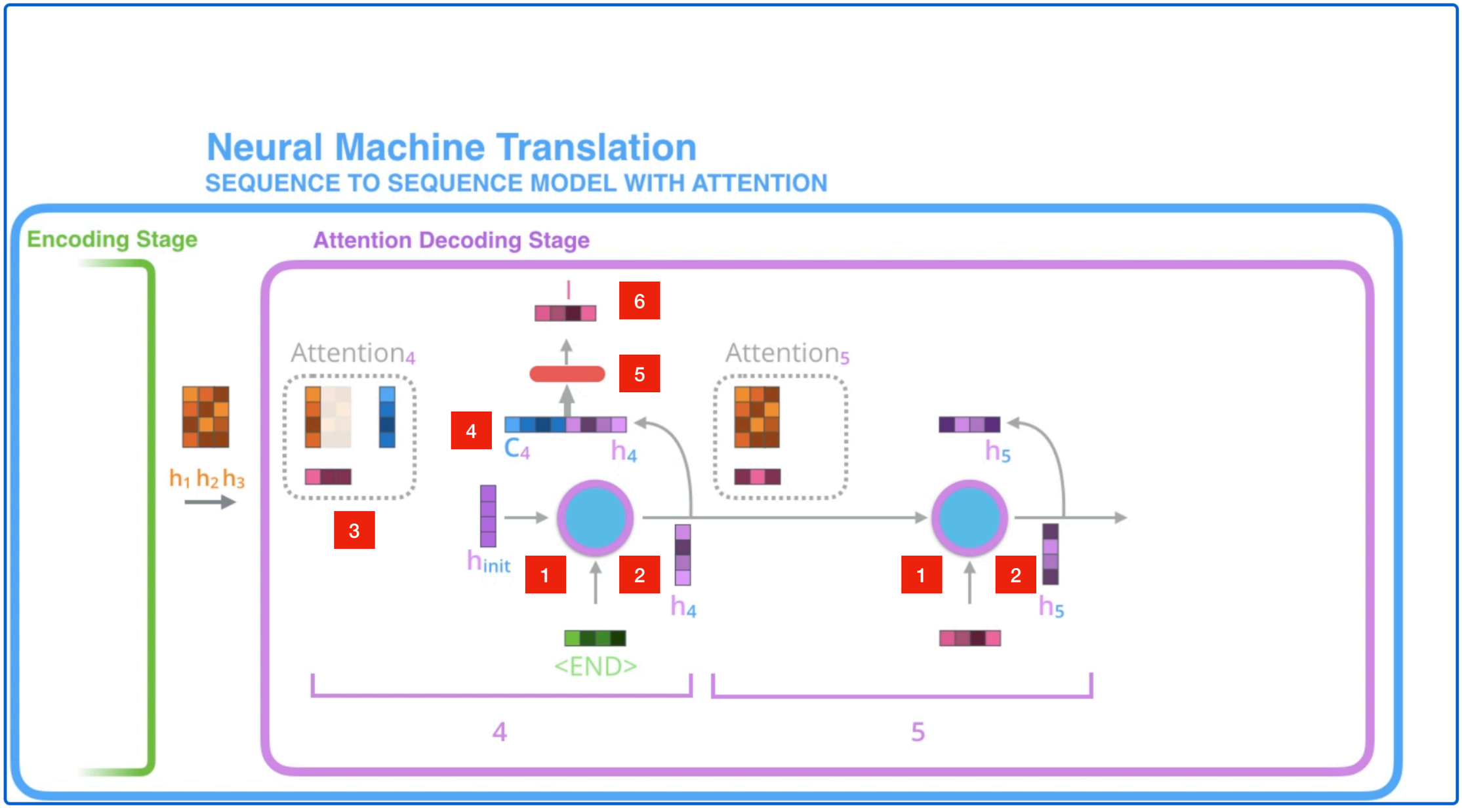

It
should
be
noted
that
the
marine
environment
is
the
least
known
of
environments
.
<end>
Il
convient
de
noter
que
l'
environnement
marin
est
le
moins
connu
de
l'
environnement
.
<end>

Attention: Luong Model



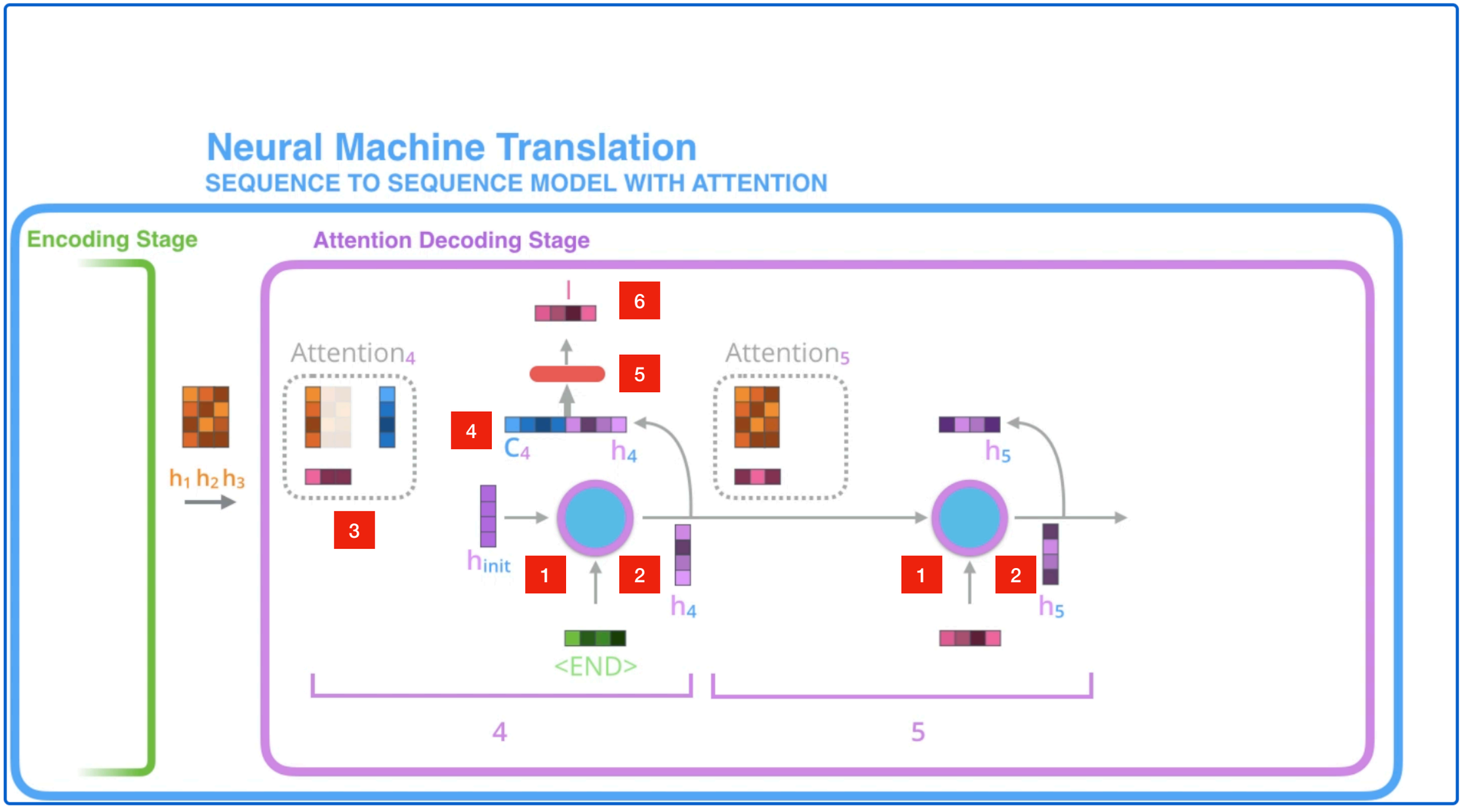
3 $score(h_i, \bar{h}_4) = h_i^T \bar{h}_4$ No training parameters!

Attention: Luong Multiplicative



3 $score(h_i, \bar{h}_4) = h_i^T W \bar{h}_4$ More training parameters!

Attention: Bahdanau

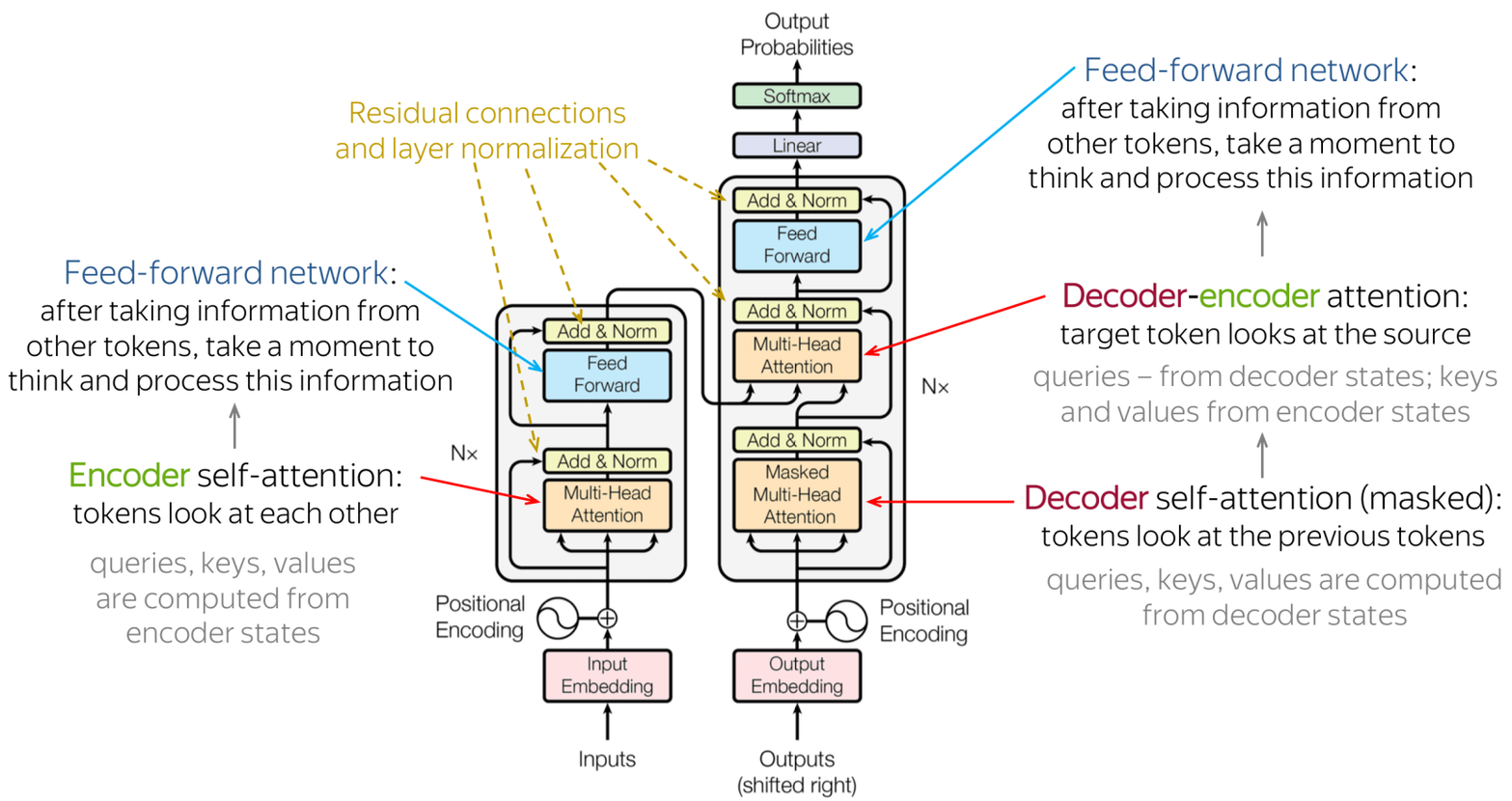


3 $score(h_i, \bar{h}_4) = v^T \tanh(W[h_i; \bar{h}_])$

Transformer

The Transformer Neural Networks — usually just called "**Transformers**" — were introduced by a Google-led team in 2017 in a paper titled "*Attention Is All You Need*".

The Transformer has an **encoder-**decoder architecture but it is based solely on attention mechanisms: i.e., without recurrence or convolutions.



https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Transformer

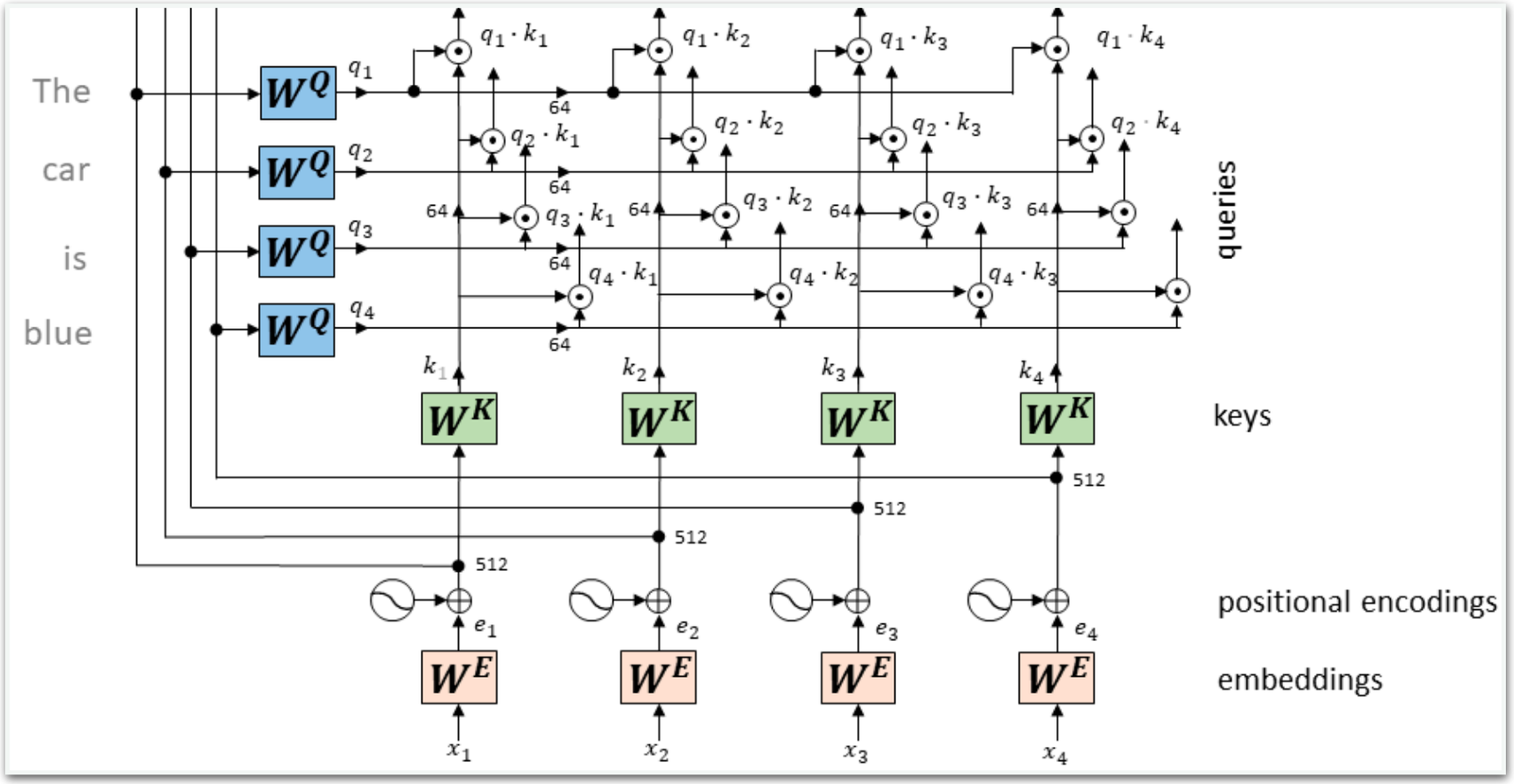
In the last few years, several architectures based on the basic transformer introduced in the 2017 paper have been developed and trained for complex natural language processing tasks:

- BERT
- DistilBERT
- T5
- GPT-3 and GPT-4
- Etc.

HuggingFace has implemented a Python package for transformers that is really easy to use. It is open-source and you can find it on GitHub.

Transformer Encoder

We will focus on the encoder (self-attention) architecture.



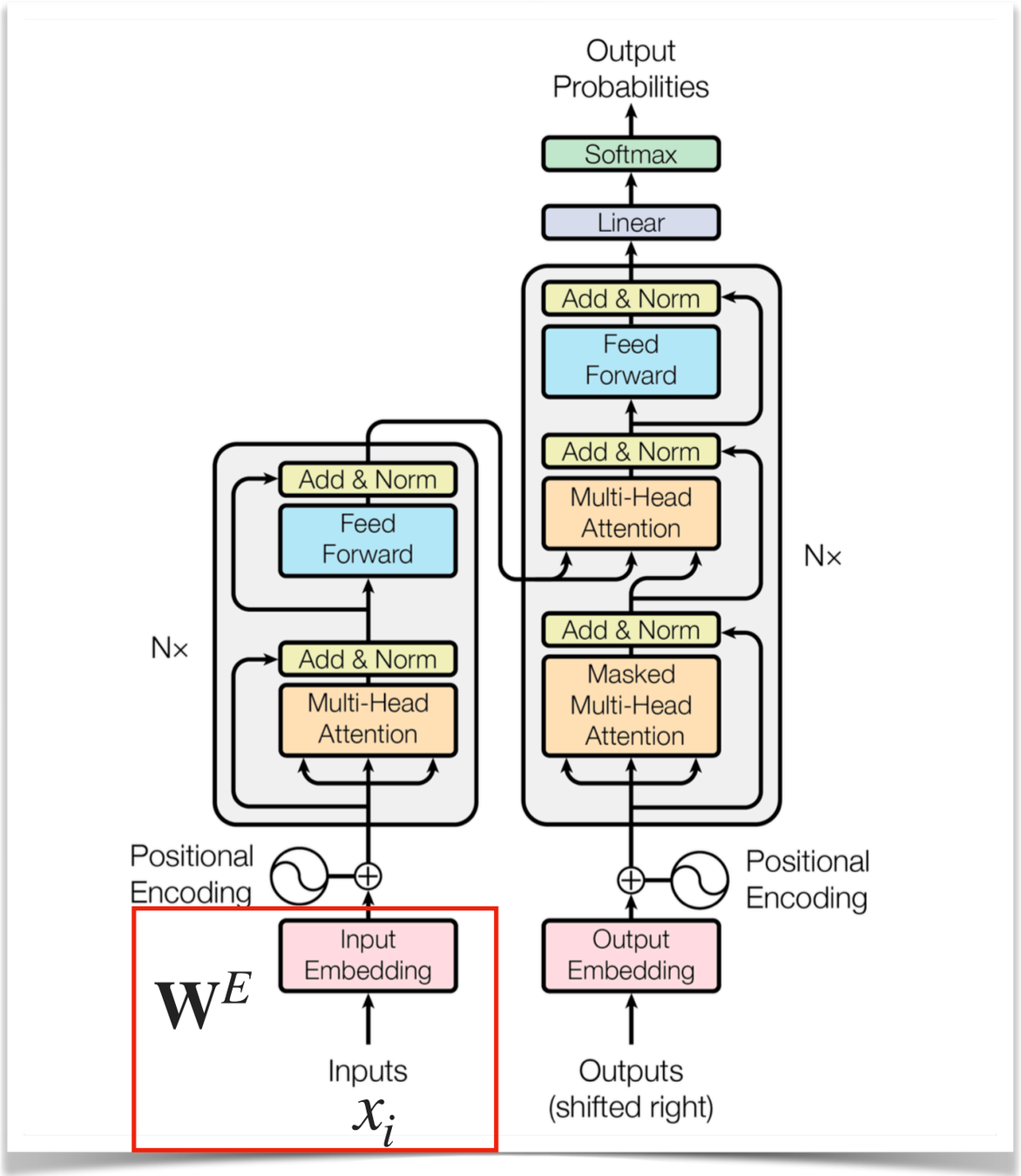
https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

Transformer Input

A Transformer takes as input a sequence of tokens x_i, which are presented to the network as **one-hot encodings**.

Next, we reduce the dimensionality of the one-hot encoded vectors by multiplying them with a so called "**embedding matrix**" $\mathbf{W}^E$. The resulting vectors e_i are called token embeddings.

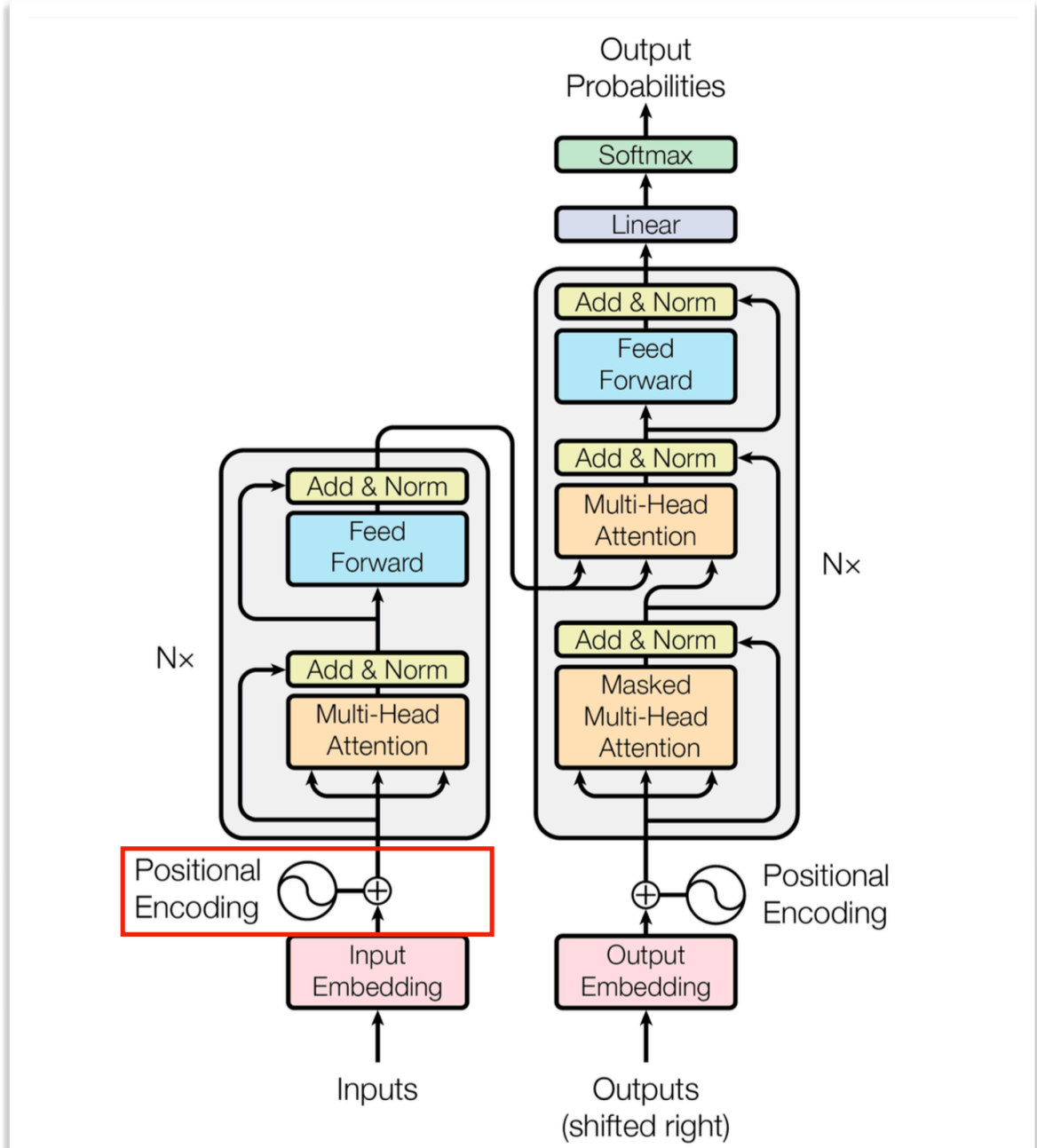
The size of the token (word) embeddings in the original paper is $d = 512$.



Transformer Input

All the tokens are presented to the Transformer **simultaneously**. However, this means that the **order in which tokens occur in the input sequence is lost**.

To address this, the Transformer **adds a vector to each input** embedding, thus injecting some information about the position.



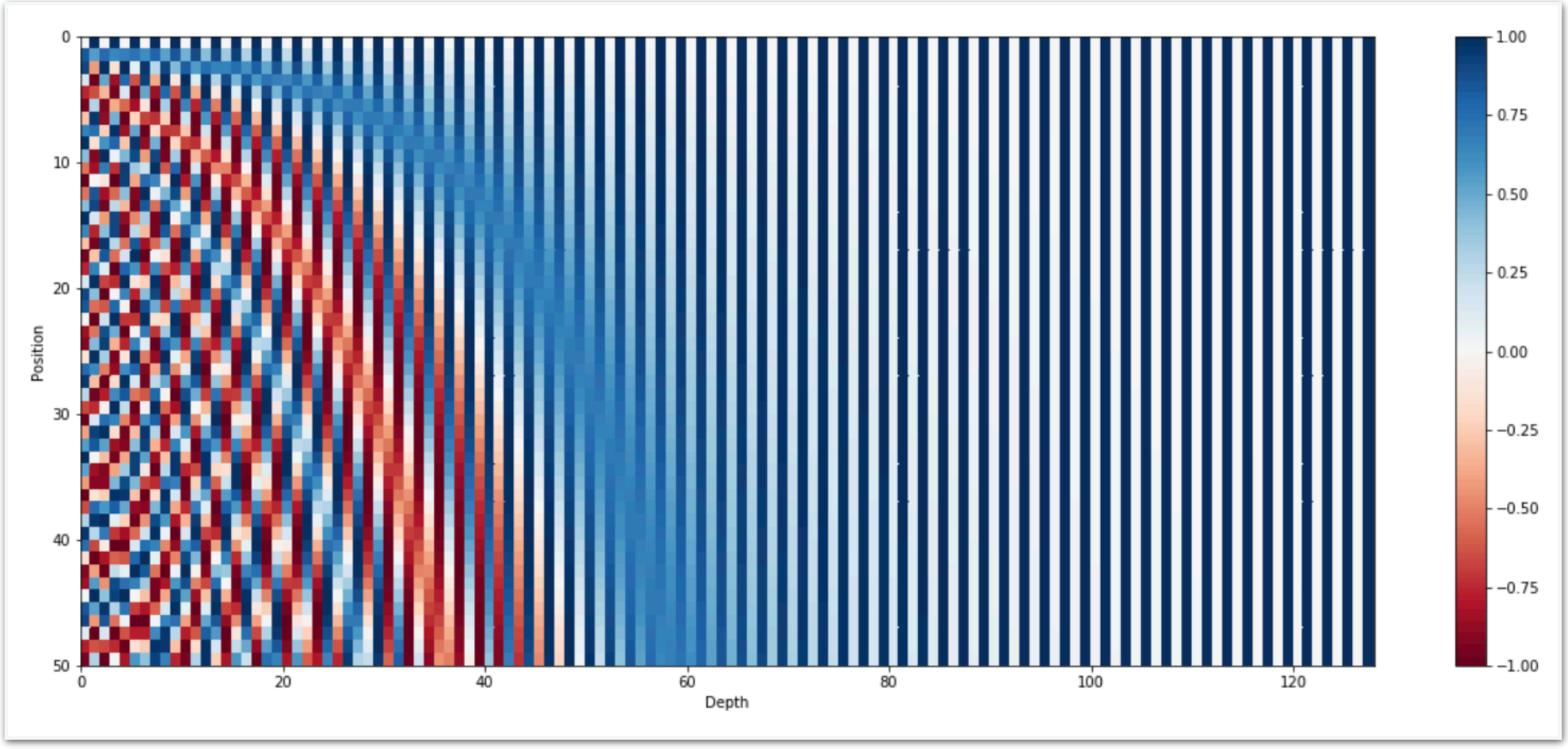
Transformer Input

The encoding proposed by the authors is a d-dimensional vector that contains information about a specific **position** in a sentence.

It must satisfy these criteria:

- It should output a unique encoding for each time-step (token's **position** in a sentence)
- Distance between any two time-steps should be consistent across sentences with different lengths.
- The model should generalize to longer sequences without any efforts.
- Its values should be bounded.
- It must be deterministic.

Transformer Input



The 128-dimensional positonal encoding for a sentence with the maximum lenght of 50. Each row represents the embedding vector $\overrightarrow{p_t}$

Transformer Input

In the original paper **added the positional encoding on top of the actual embeddings**.

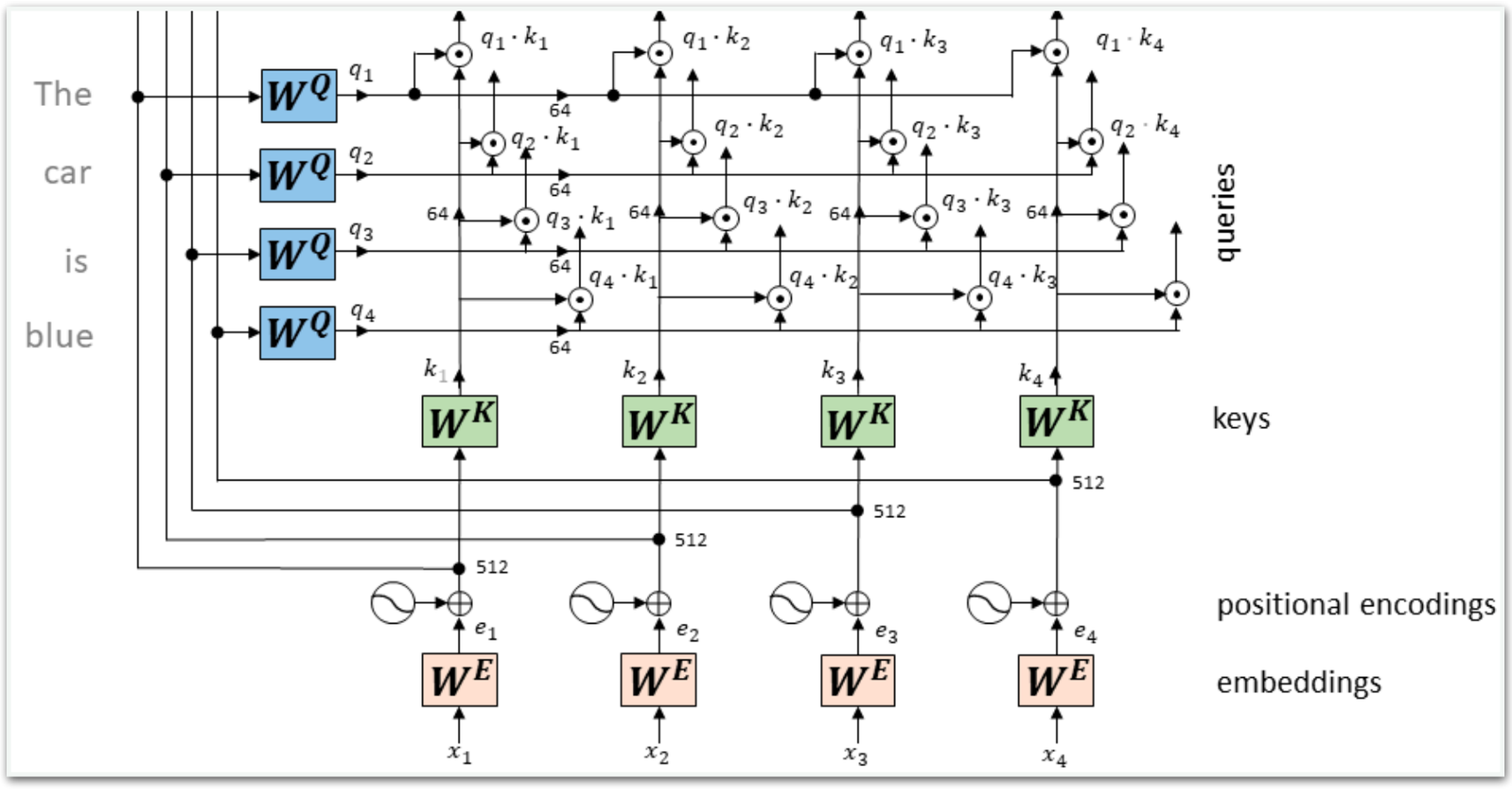
That is for every word embedding e_t in a sentence of words $[x_1, \ldots x_n]$ the correspondent "position-aware" embedding which is fed to the model is as follows:

$$\psi'(e_t) = \psi(e_t) + \overrightarrow{p_t}$$

To make this summation possible, we keep the positional embedding's dimension equal to the word embeddings' dimension

Keys and Queries

Next, we multiply the position-aware word embeddings by two matrices, W^Q and W^K, to obtain the "**query vectors**" and "**key vectors**", each of size 64.

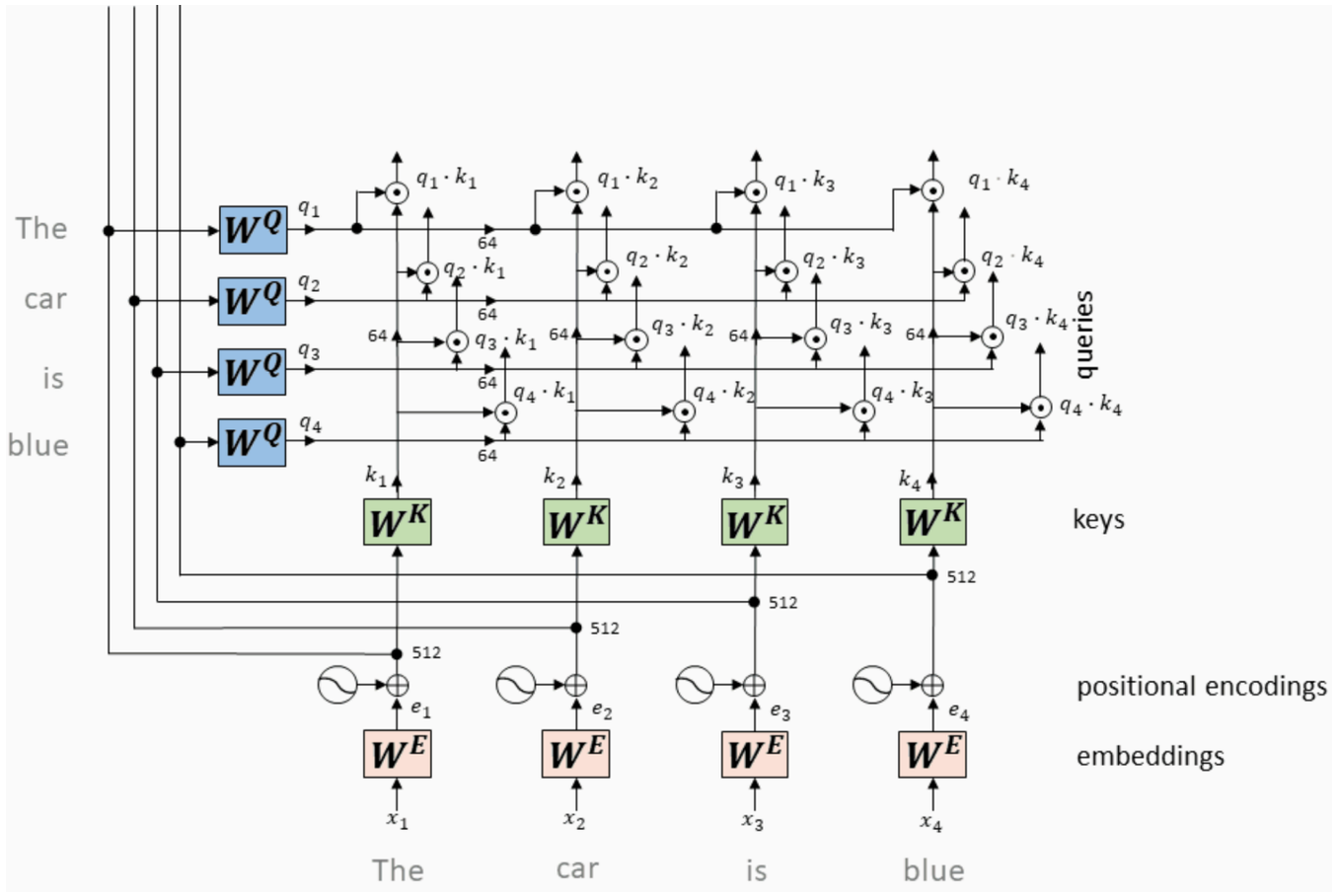


https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

Keys and Queries

We calculate the dot products **for all possible combinations of "query vectors" and "key vectors"**.

The result of a dot product is a single number, which in a later step will be used as a weight factor. **The weights factors tell us, how much two words at different positions of the input sentence depend on each other**. This is called **self-attention** in the original paper.

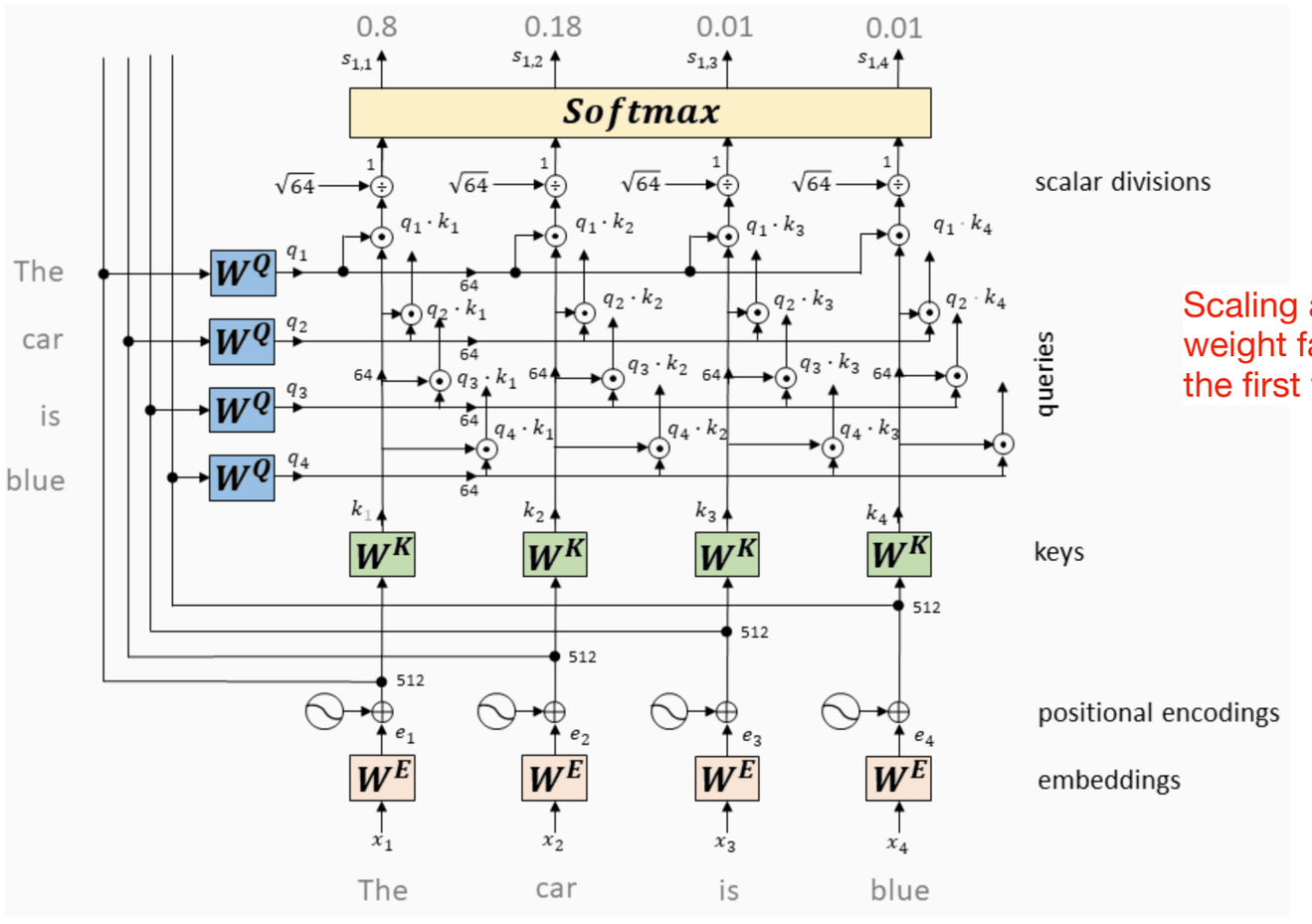


https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

Keys and Queries

Subsequently, all weight factors are divided by 8 (the square root of the dimension of the key vectors 64), to have more stable gradients.

The scaled factors are put through a *softmax* function, which normalizes them so they are all positive and sum up to 1 to get $s_{i,j}$.

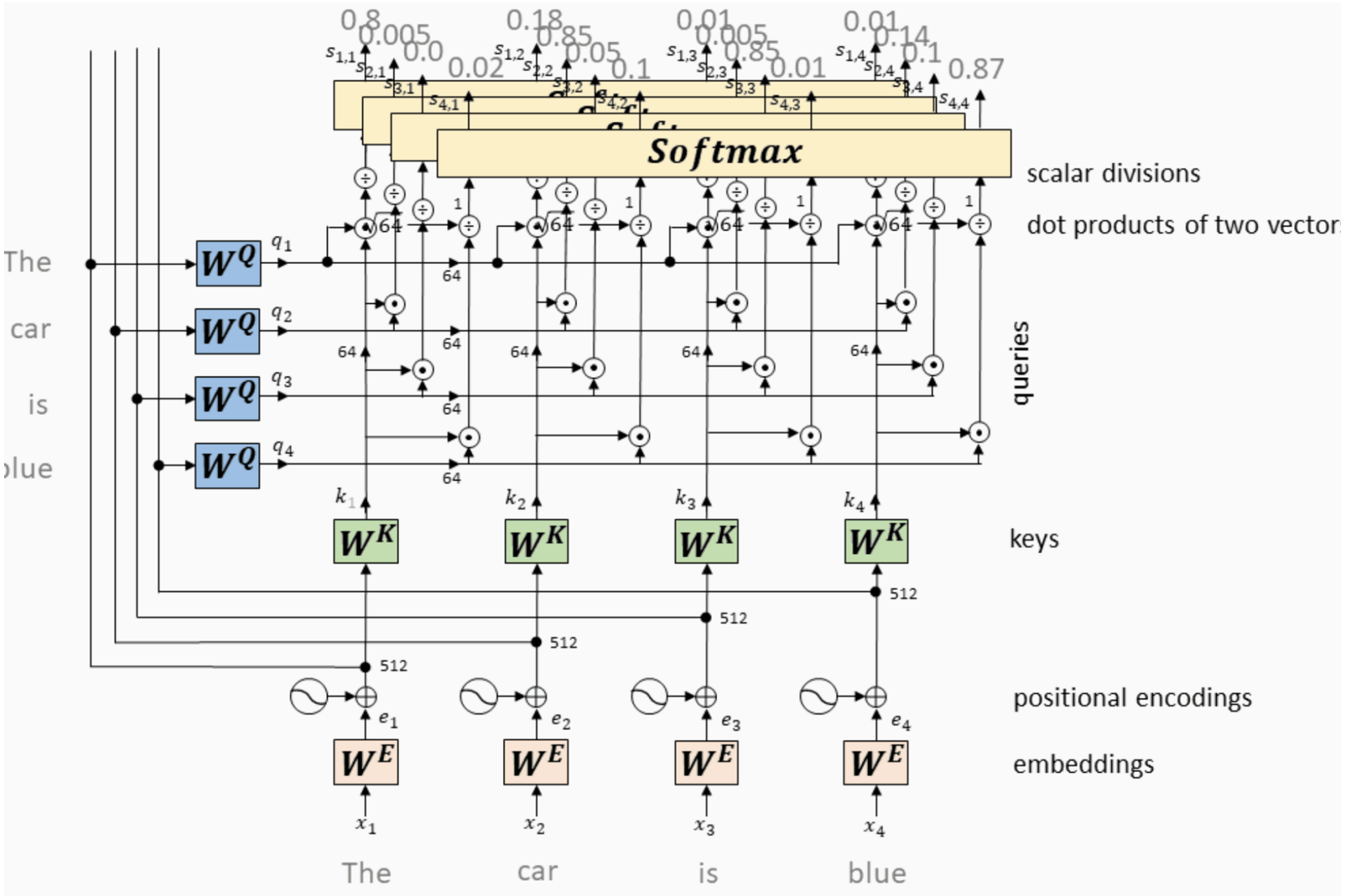


Scaling and softmax of the weight factors belonging to the first word “The”

https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

Keys and Queries

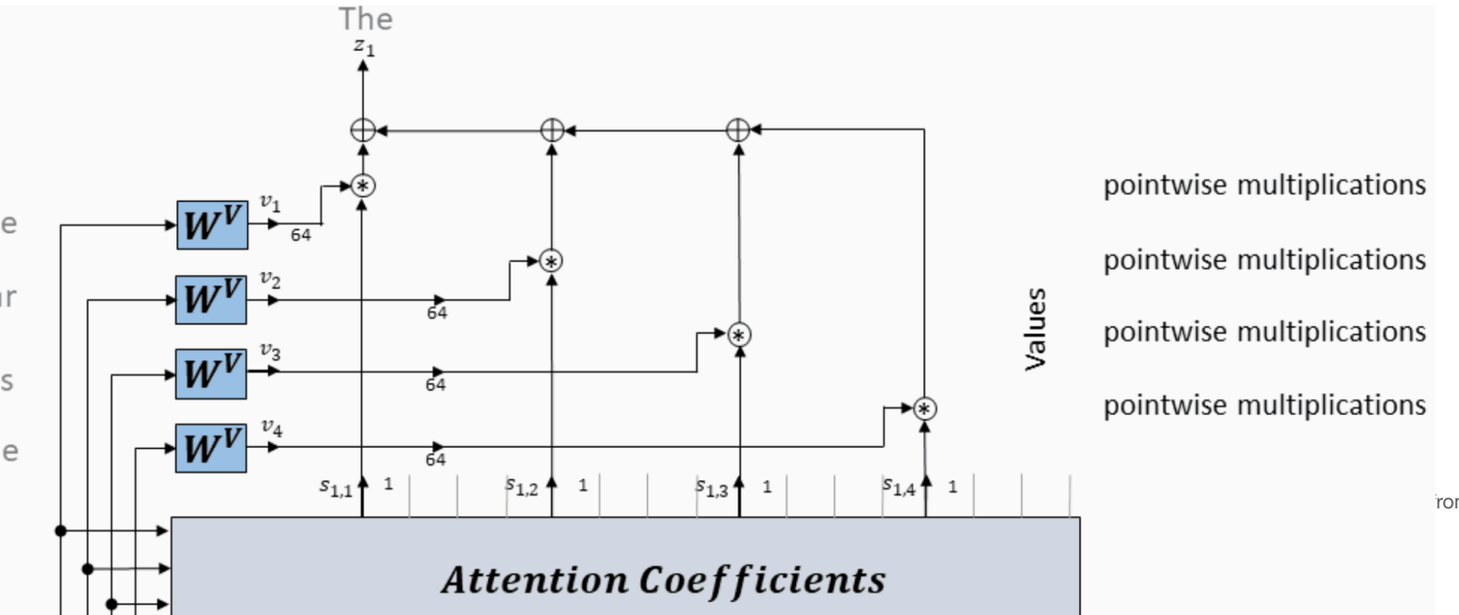
Analogously, for the other words “car”, “is” and “blue” in our input sequence to get all $s_{i,j}$:



https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

Values, weighting and summation

Identical to the computation of the "key vector" and "query vectors" we obtain the "value vectors" v_k by multiplying the word embeddings by a matrix W^V.
Each "value vector" v_k is multiplied by its corresponding "weight factors" $s_{k,i}, \forall i$.
Finally, we add toghether all these values to get z_k.
This way we only keep the words we want to focus on, while irrelevant words are suppressed by weighting them by tiny numbers like 0.001



rom-scratch-part-1-9269ed9a2c5e

Values, weighting and summation

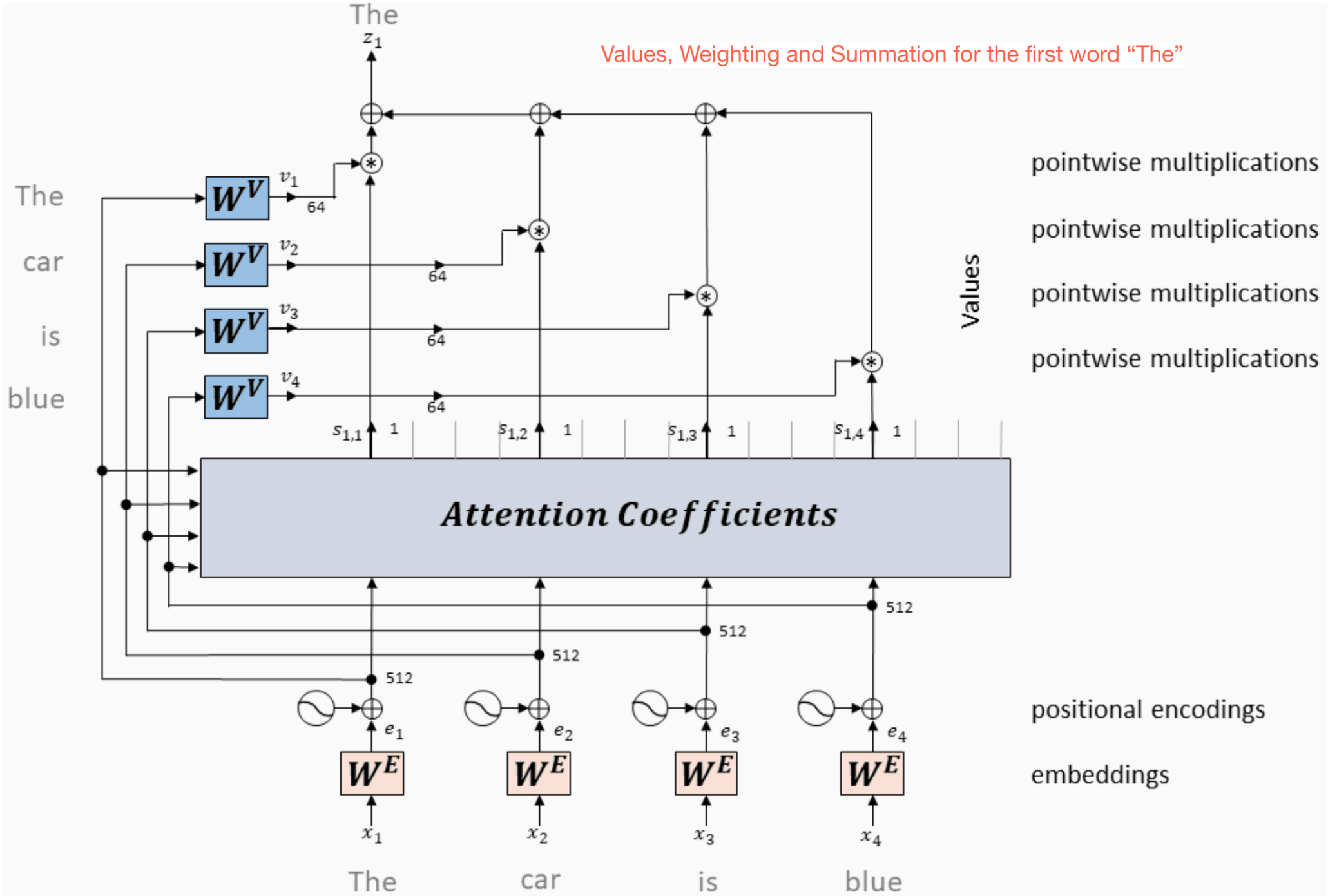
Identical to the computation of the "key vector" and "query vectors" we obtain the "value vectors" v_k by multiplying the word embeddings by a matrix W^V.

Each "value vector" v_k is multiplied by its corresponding "weight factors" $s_{k,i}, \forall i$.

Finally, we add toghether all these values to get z_k.

This way we only keep the words we want to focus on, while irrelevant words are suppressed by weighting them by tiny numbers like 0.001

Values, weighting and summation

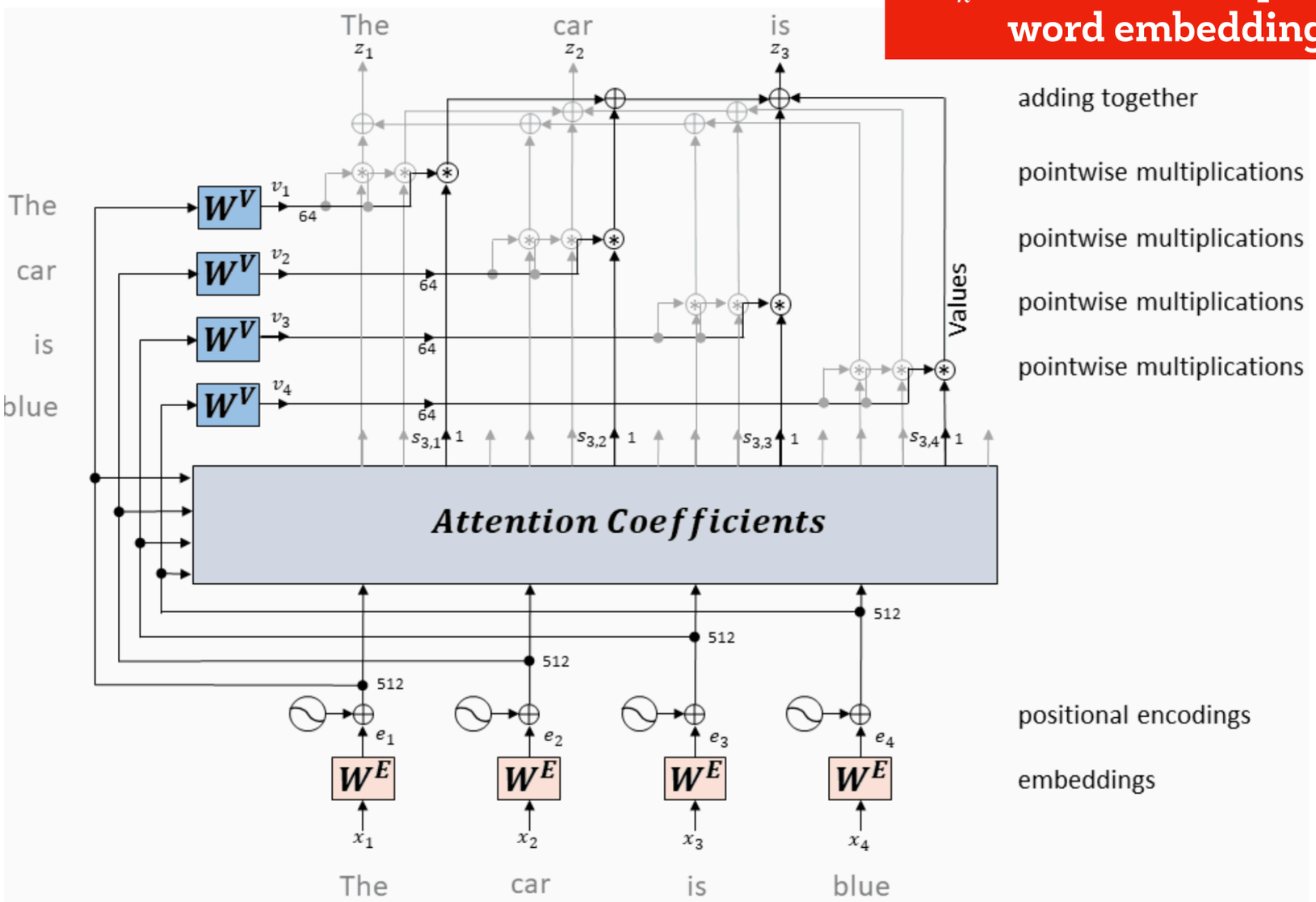


https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

Values, weighting and summation

Analogously for the other words “car”, “is”, “blue” in our input sequence.

z_k are **context-dependent word embeddings**



https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

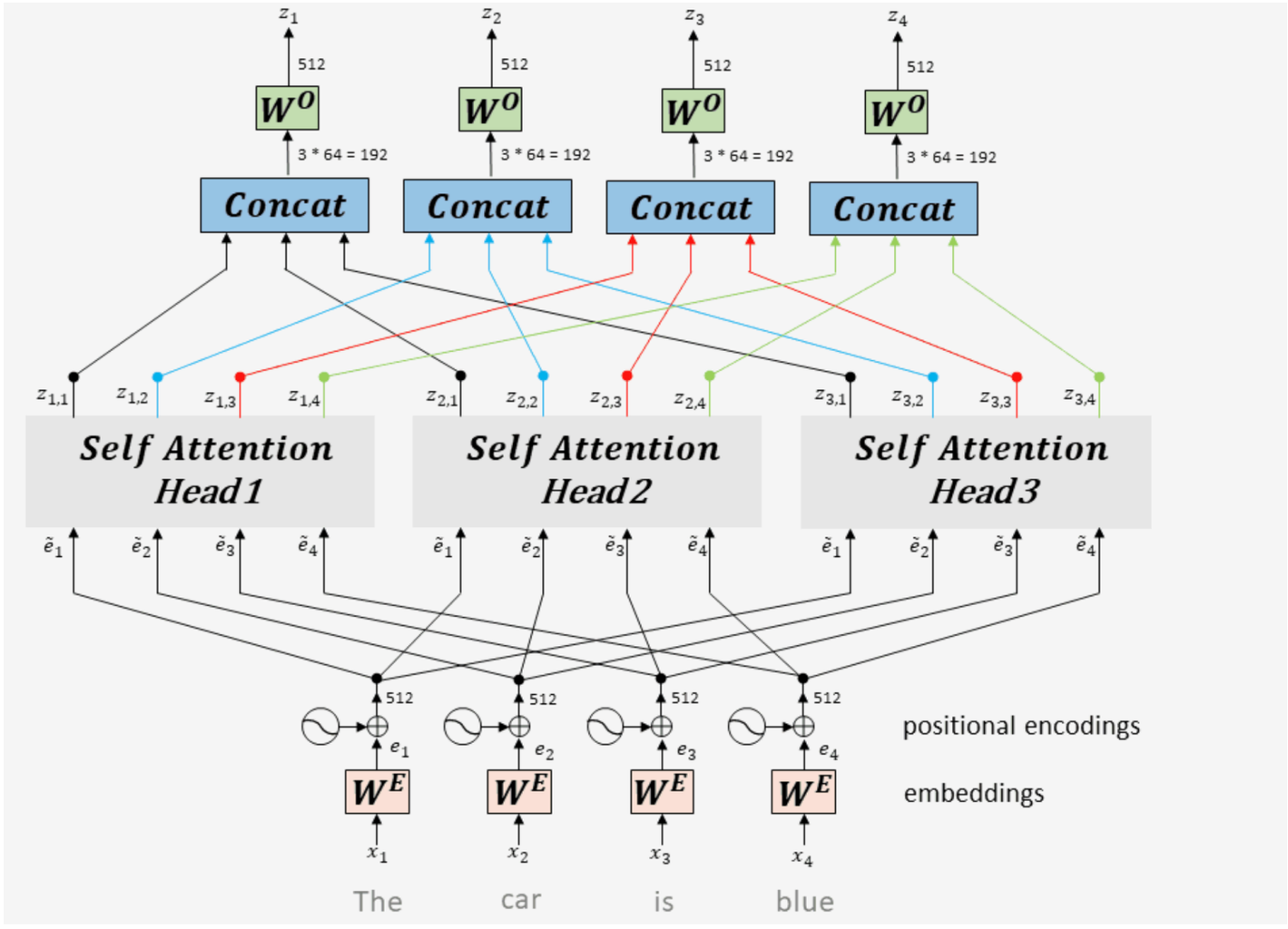
Attention in matrix form

Since we're dealing with matrices, we can condense some steps in one formula to calculate the outputs of the **self-attention layer:**

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

Self-attention

That could conclude a **self-attention calculation**. The output of the self-attention layer can be considered as a **context enriched word embedding**, but instead of performing a single self-attention function, the authors employ **multiple self-attention heads**, each with different weight matrices.

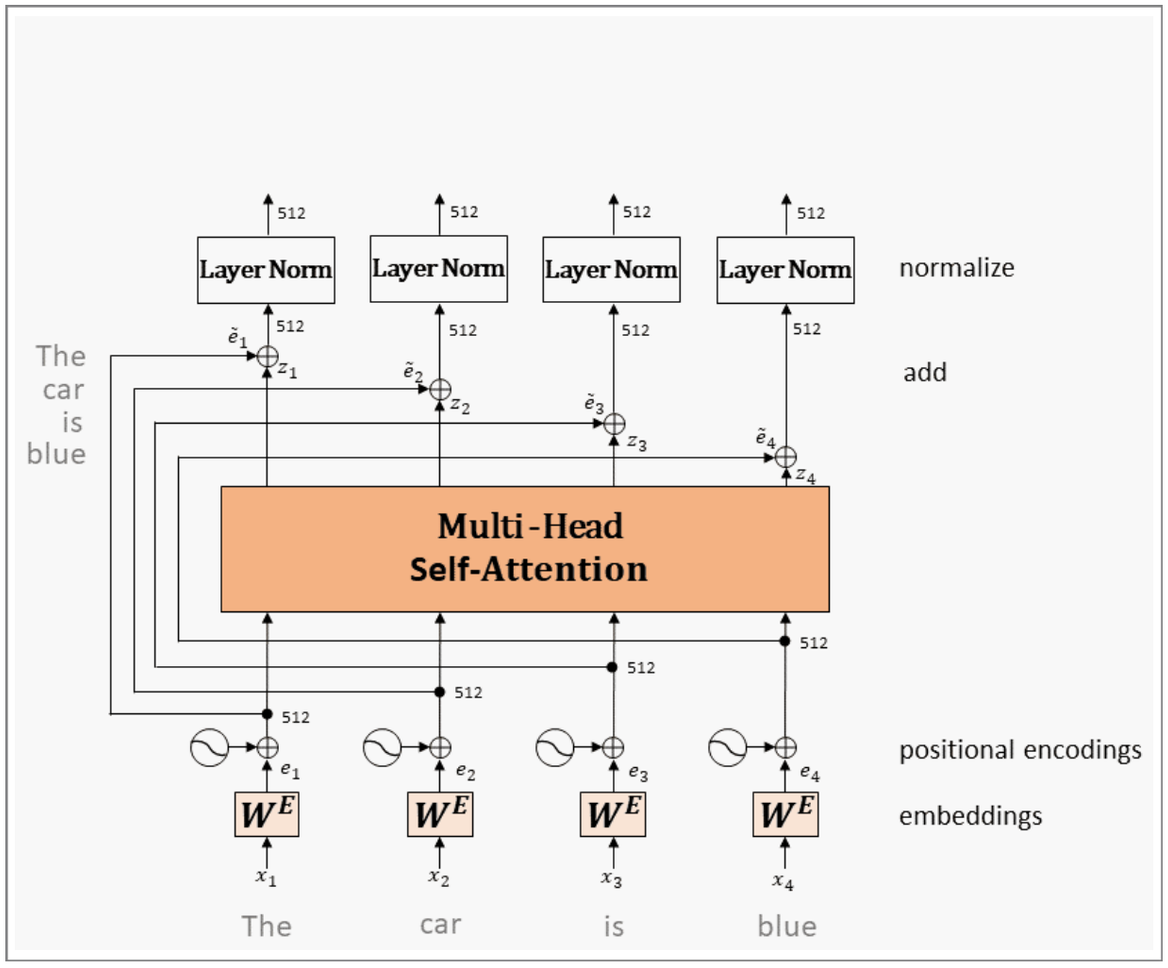


https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

The outputs of the attention heads are concatenated and once again multiplied by an additional weights matrix to get new z_k.

Feed-forward

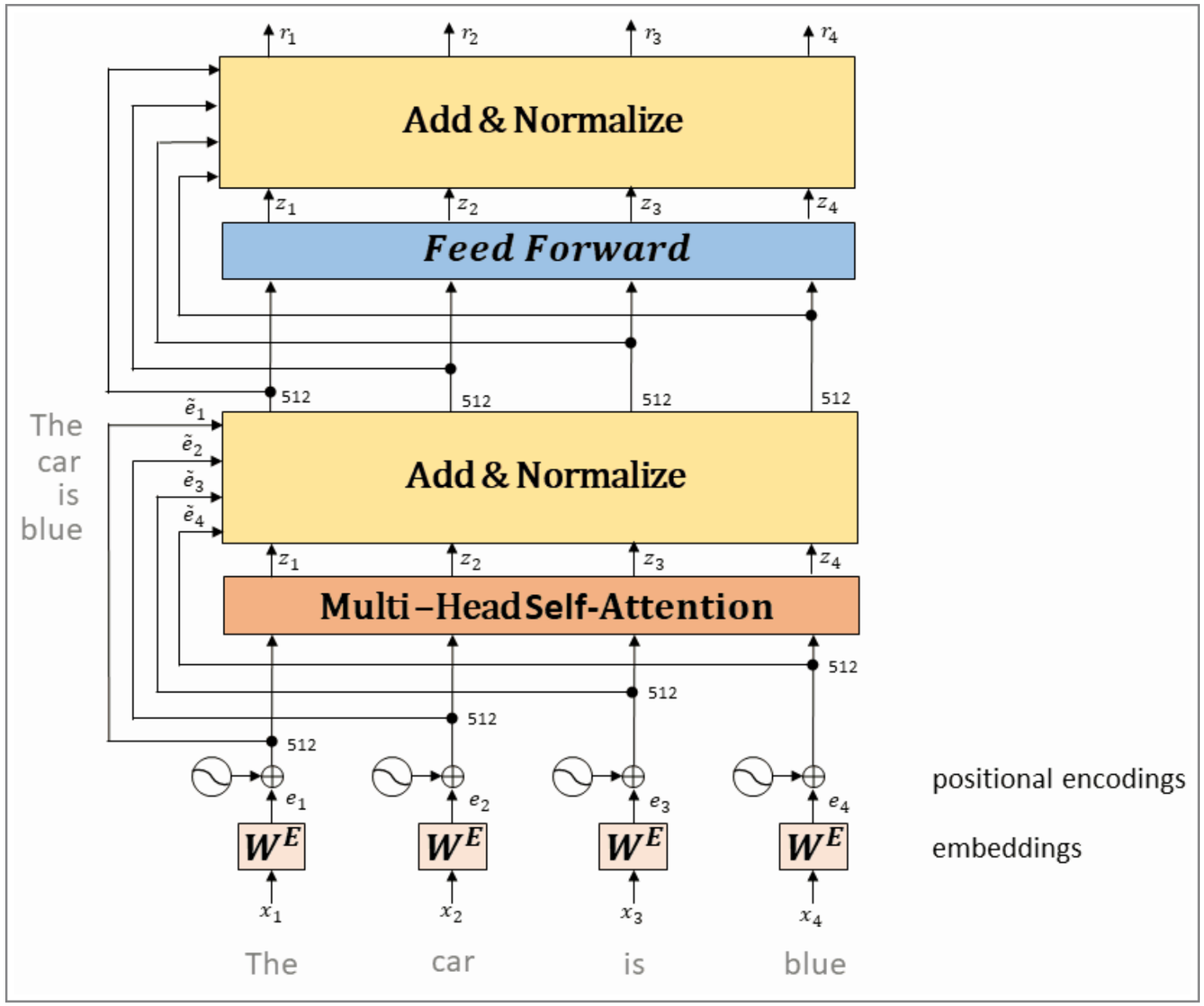
The outputs of the self-attention layer are fed to a fully connected feed-forward network. This consists of two linear transformations with a ReLU activation in between. The dimensionality of input and output is 512, and the inner-layer has dimensionality 2048. The exact same feed-forward network is independently applied to each position, i.e. for each word in the input sequence.

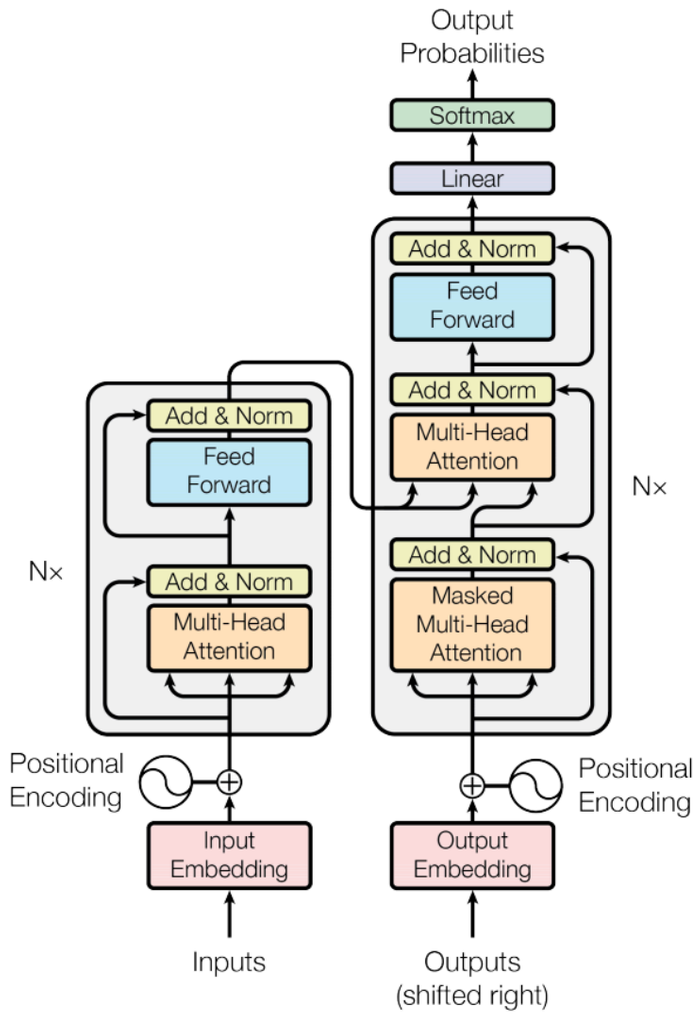


https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

Stack of encoders

The entire encoding component is a stack of six encoders. The encoders are all identical in structure, yet they do not share weights.





https://towardsdatascience.com/drawing-the-transformer-network-from-scratch-part-1-9269ed9a2c5e

How to train a Transformer: BERT loss functions

BERT (Bidirectional Encoder Representations from Transformers) is a specific Transformer implementation.

It is based on the following observation: many models predict the next word in a sequence, a directional approach which inherently limits context learning.

To overcome this challenge, BERT uses **two** training strategies: Masked LM (**MLM**) and Next Sentence Prediction (**NSP**).

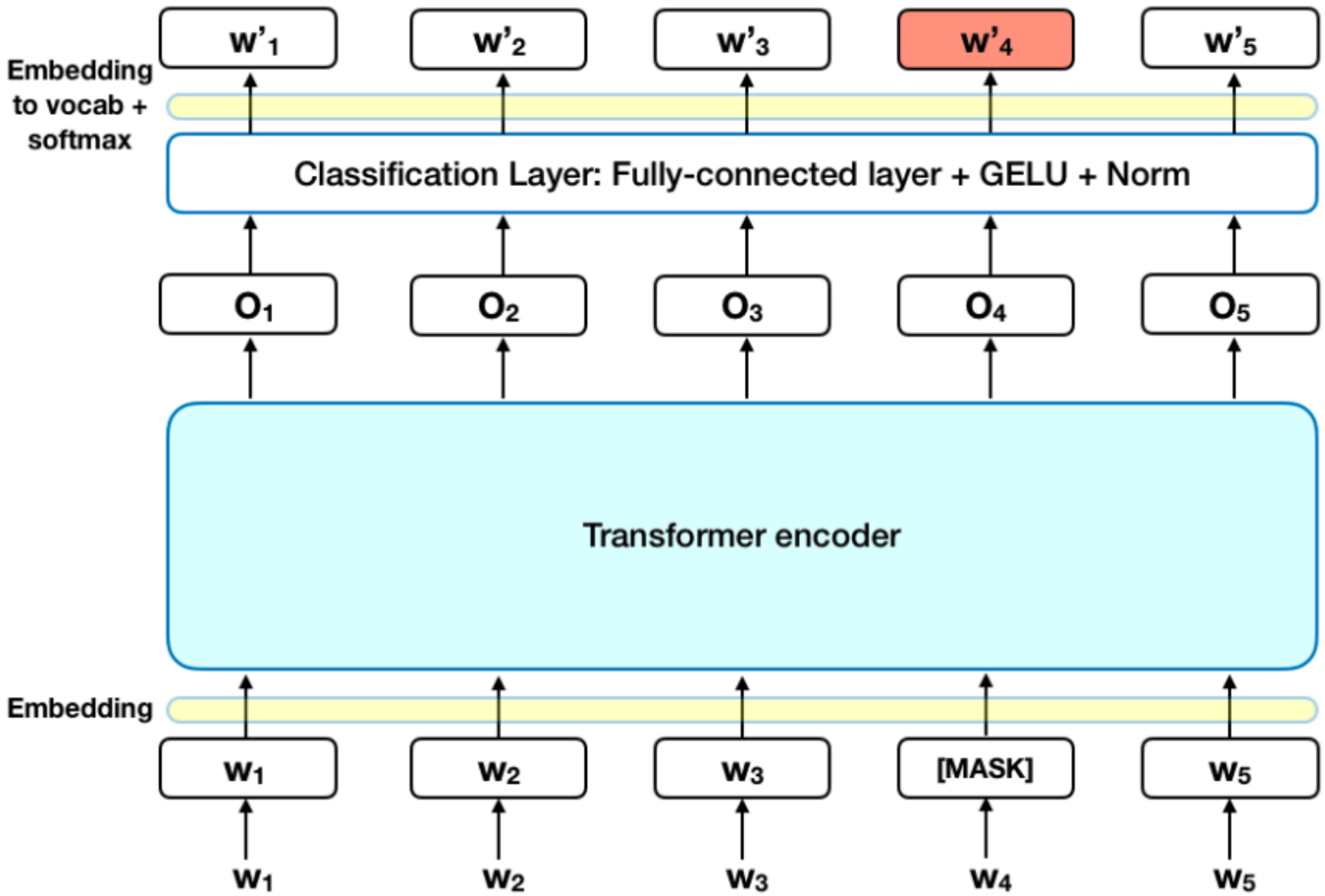
Masked LM (MLM)

Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence.

In technical terms, the prediction of the output words requires:

- Adding a classification layer on top of the encoder output.
- Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
- Calculating the probability of each word in the vocabulary with softmax.

Masked LM (MLM)



https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words.

Next Sentence Prediction (NSP)

In the BERT training process, the model **receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document**.

During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence.

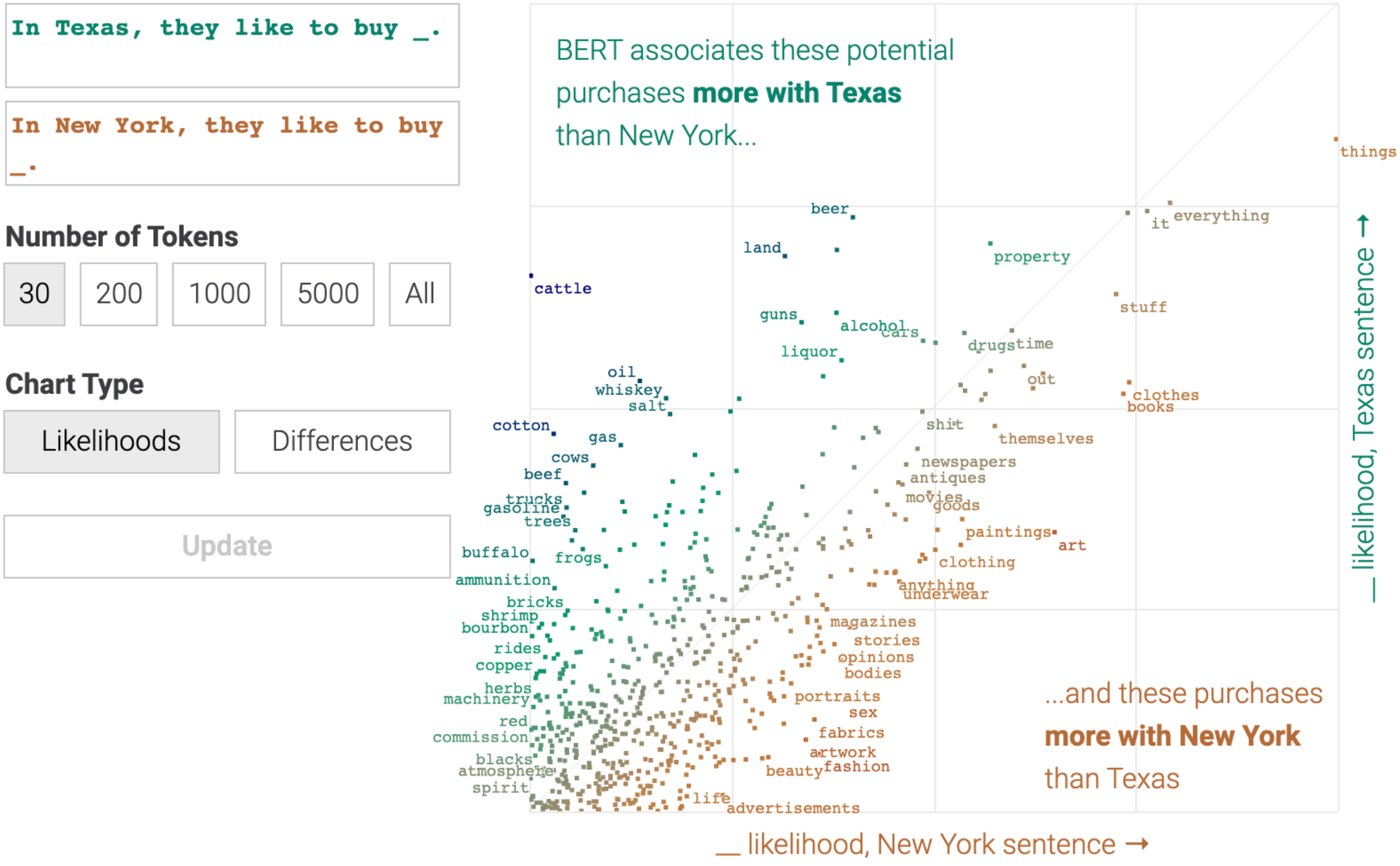
The assumption is that the random sentence will be disconnected from the first sentence.

Next Sentence Prediction (NSP)

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

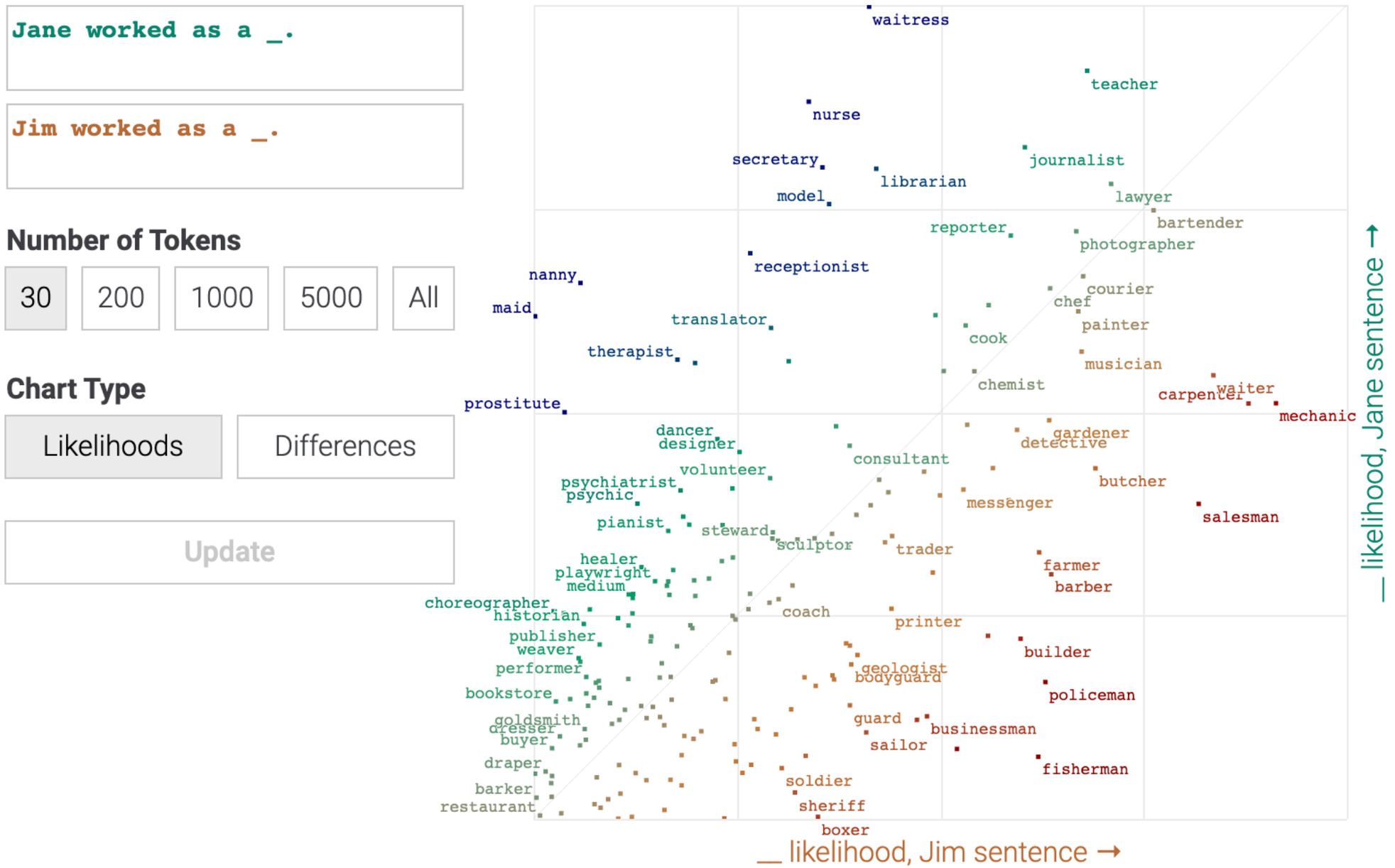
- A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
- A sentence embedding indicating Sentence A or Sentence B is added to each token.
- A positional embedding is added to each token to indicate its position in the sequence.

What Have Language Models Learned?



https://pair.withgoogle.com/explorables/fill-in-the-blank/

What Have Language Models Learned?



https://pair.withgoogle.com/explorables/fill-in-the-blank/