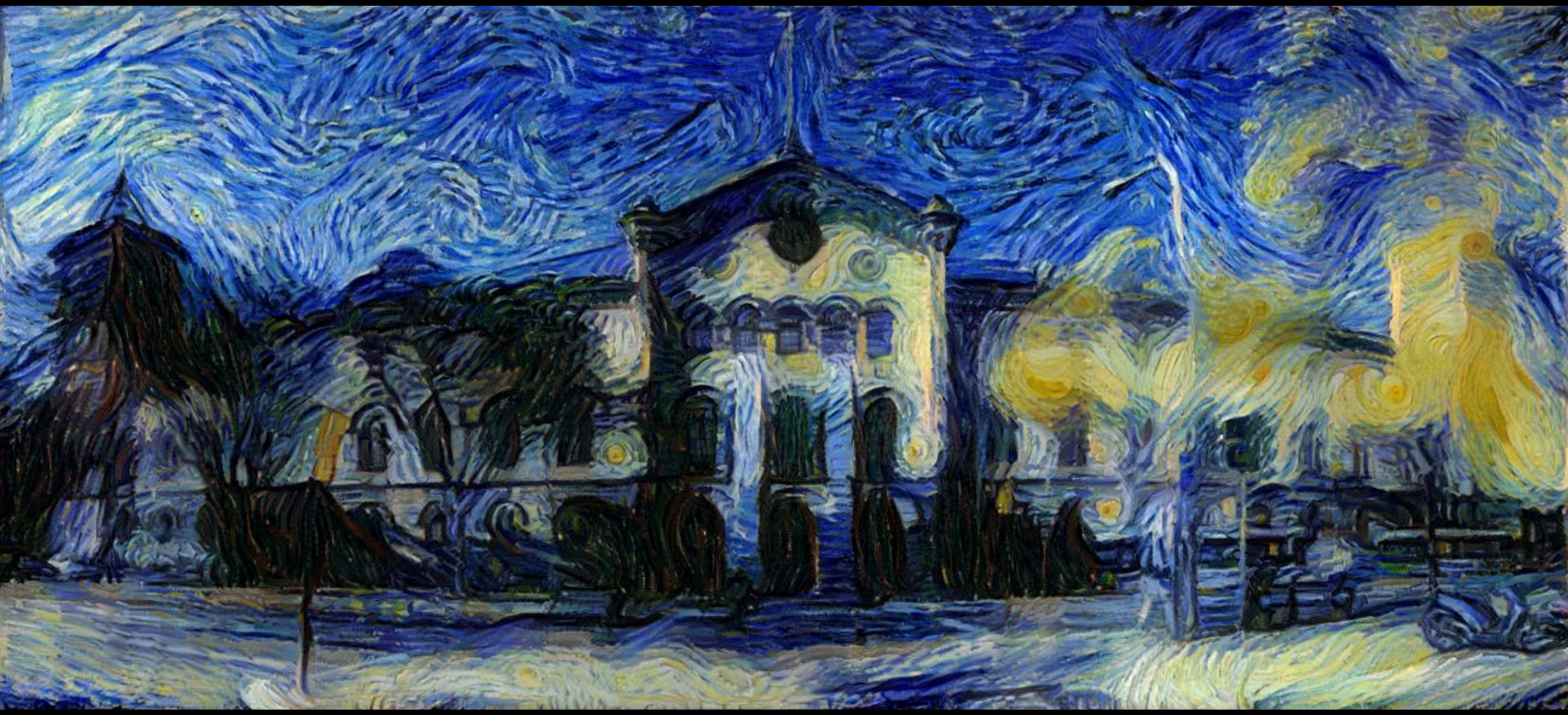


Deep Learning

Deep Sequential Models

Deep Sequential Models

Classical neural networks suffer from two severe limitations:

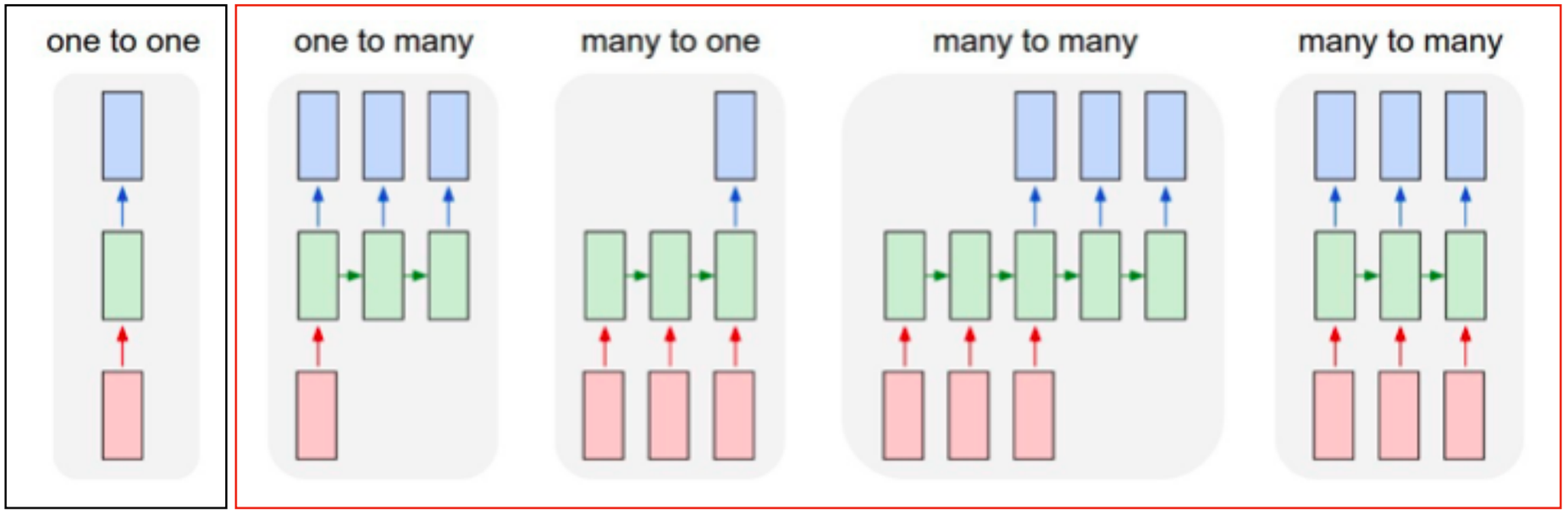
- They only accept a **fixed-sized tensors** as input and produce a fixed-sized tensor as output.
- They do not consider the **sequential nature of some data** (language, video frames, time series, etc.)

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

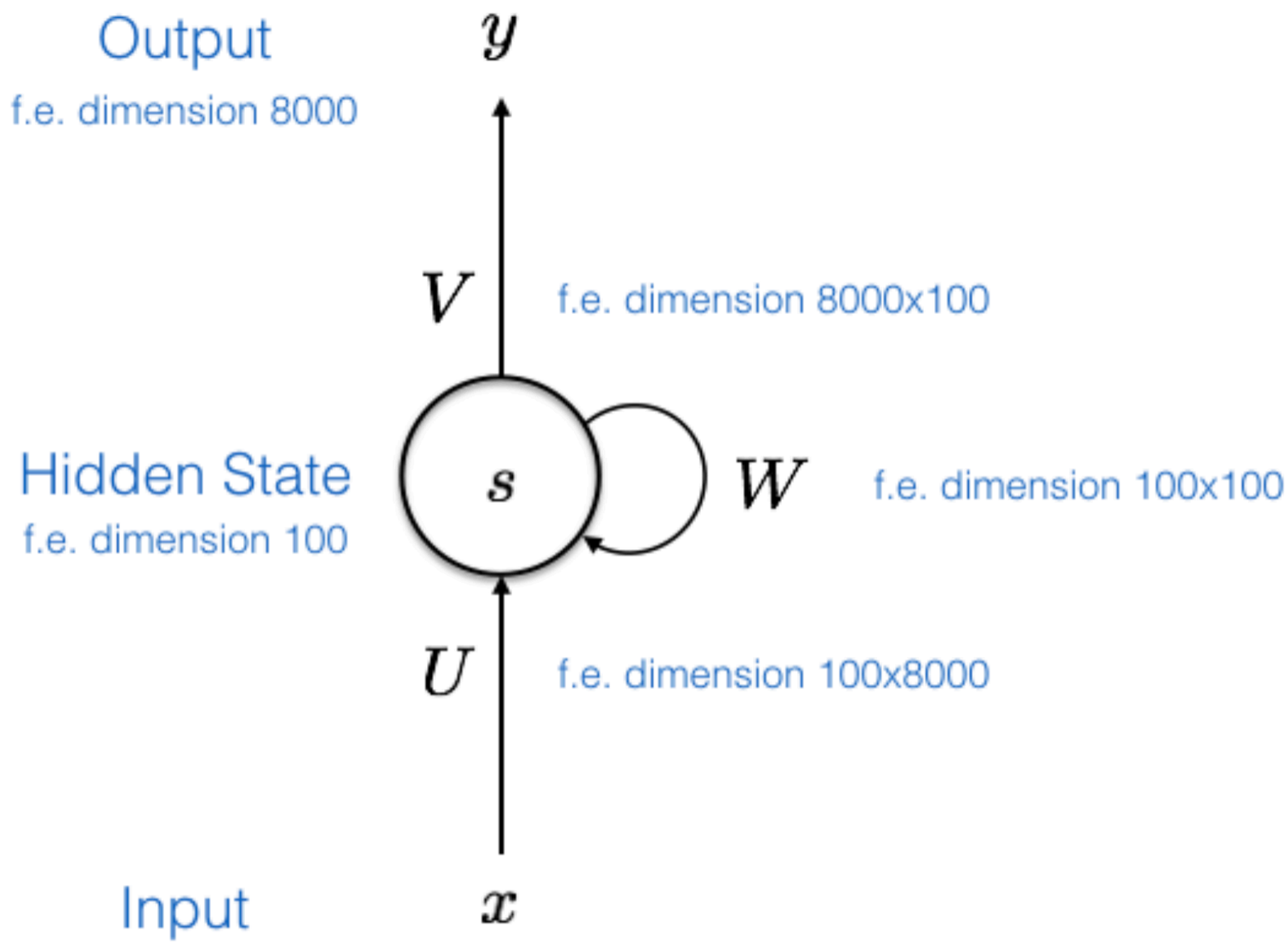


1 tensor → 1 tensor

n tensors → m tensors

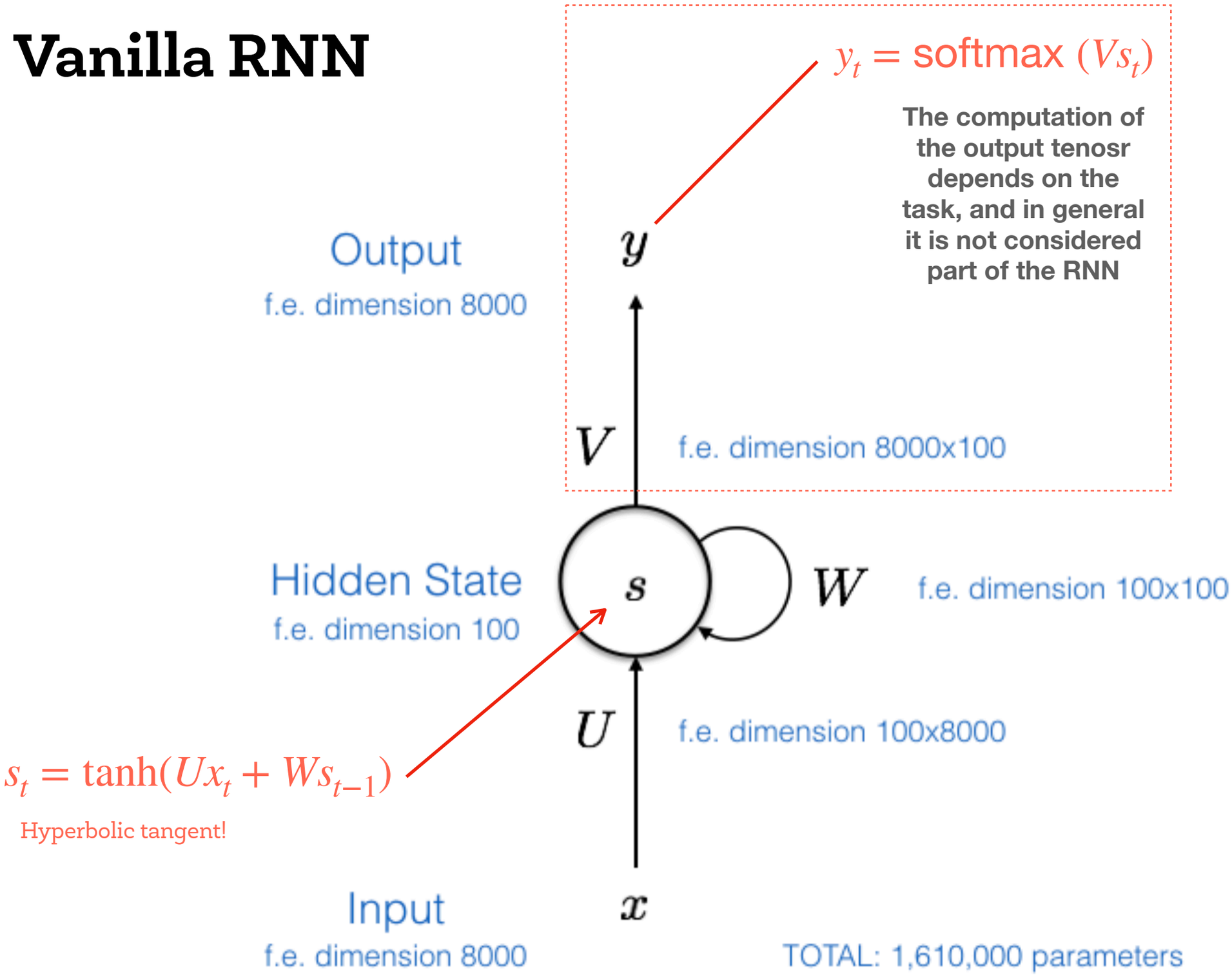
First approach: Vanilla RNN

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



TOTAL: 1,610,000 parameters

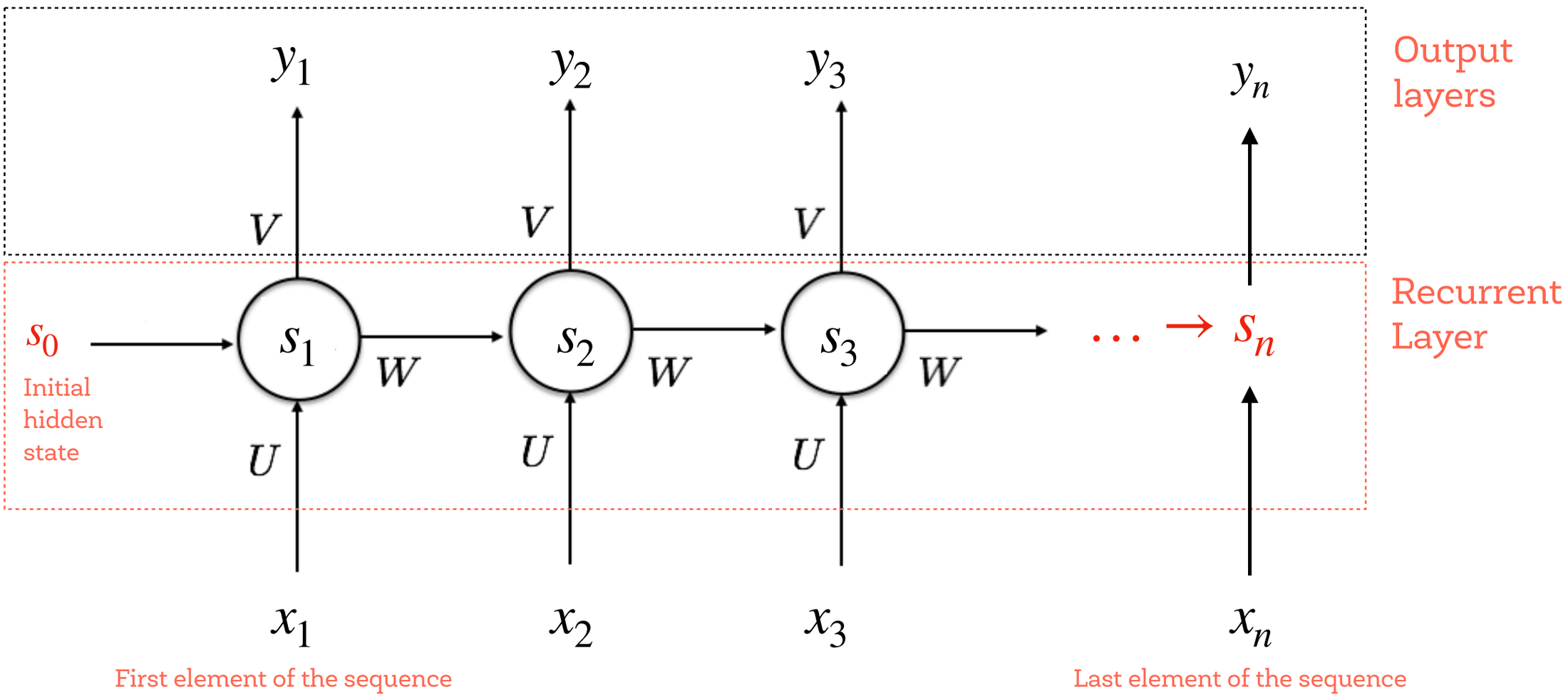
Vanilla RNN


$y_t = \text{softmax}\,(Vs_t)$
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.

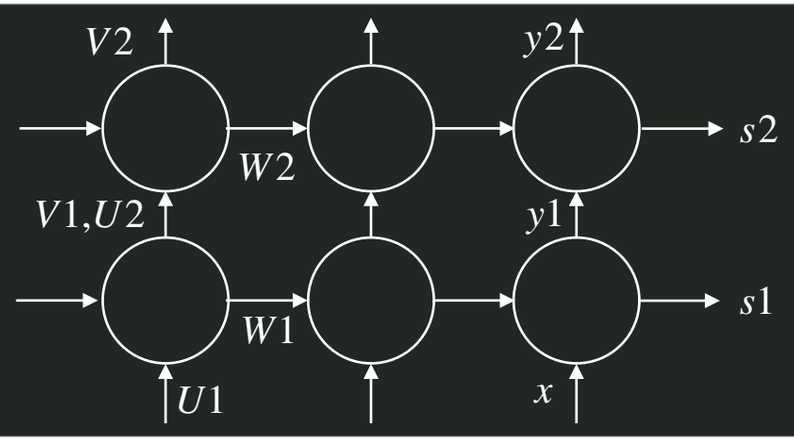
Vanilla RNN

```
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

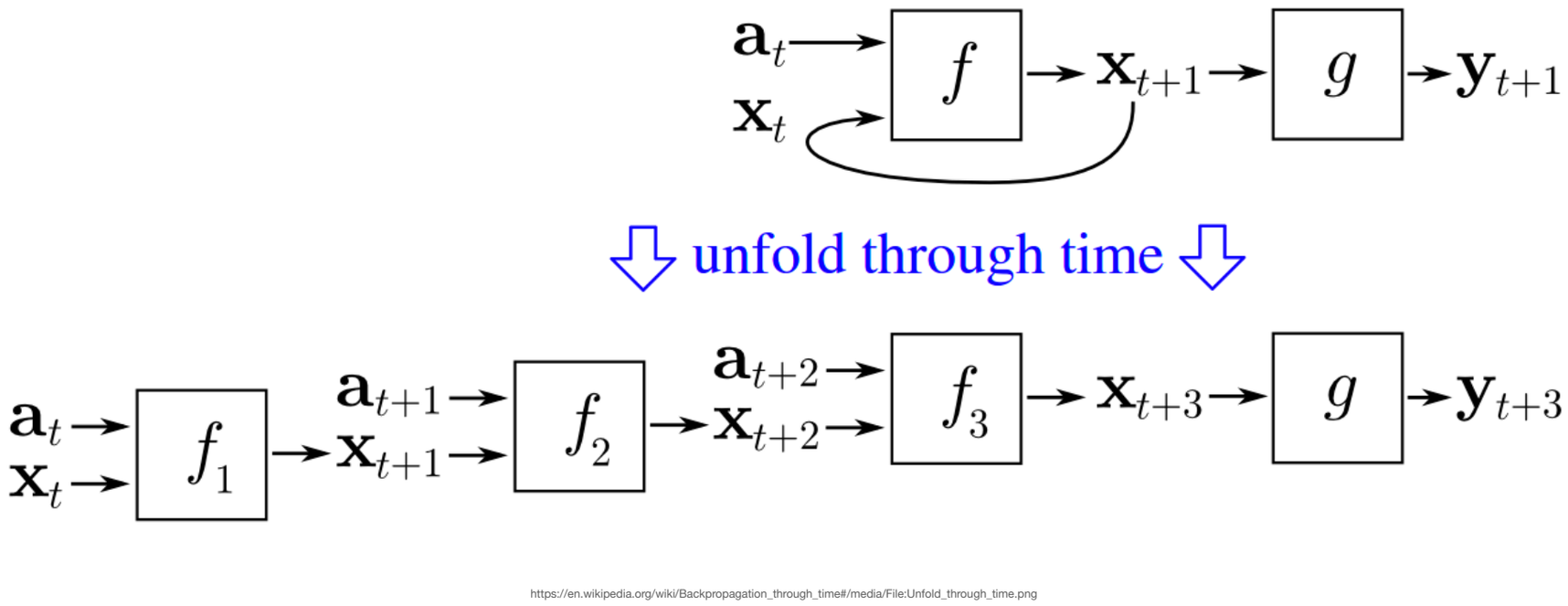
NumPy implementation of a simple RNN

```
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

Training a RNN is similar to training a traditional NN, but with some modifications.

The main reason is that parameters are shared by all time steps: in order to compute the gradient at $t = 4$, we need to propagate 3 steps and **sum up the gradients**. This is called **Backpropagation through time (BPTT)**.



https://en.wikipedia.org/wiki/Backpropagation_through_time#/media/File:Unfold_through_time.png

Vanilla RNN for language self-learning

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

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

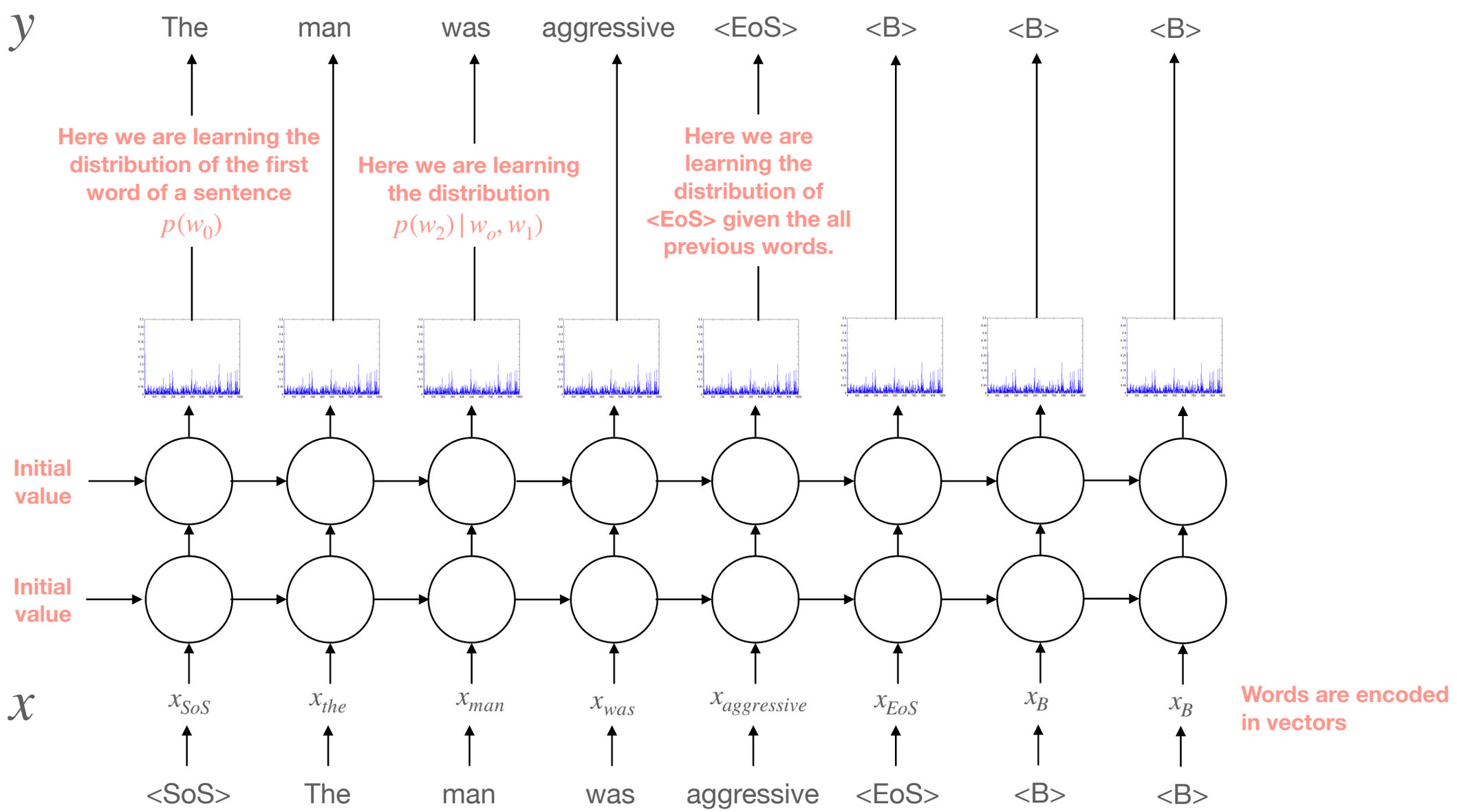
x_i are the one-hot word **vectors** corresponding to a corpus with d **symbols**.

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:

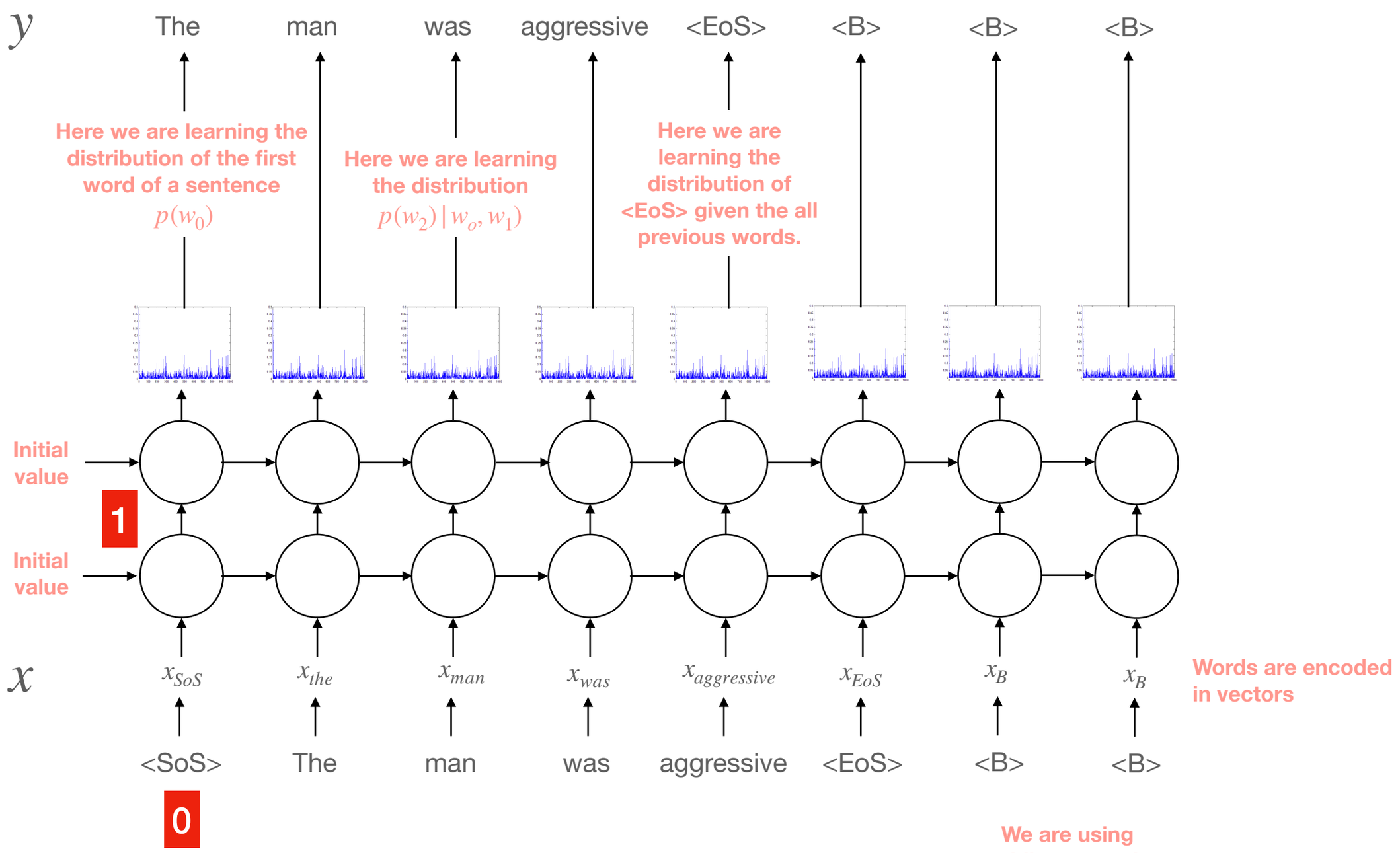
- $x_t \in \mathbb{R}^d$ is input word vector at time t.
- $W^{(hx)} \in \mathbb{R}^{D_h \times d}$ is the weights matrix used to condition the input word vector, x_t.
- $W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$ is the weights matrix used to condition the output of the previous time-step, h_{t-1}.
- $h_{t-1} \in \mathbb{R}^{D_h}$ 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 $\hat{y} \in \mathbb{R}^d$.

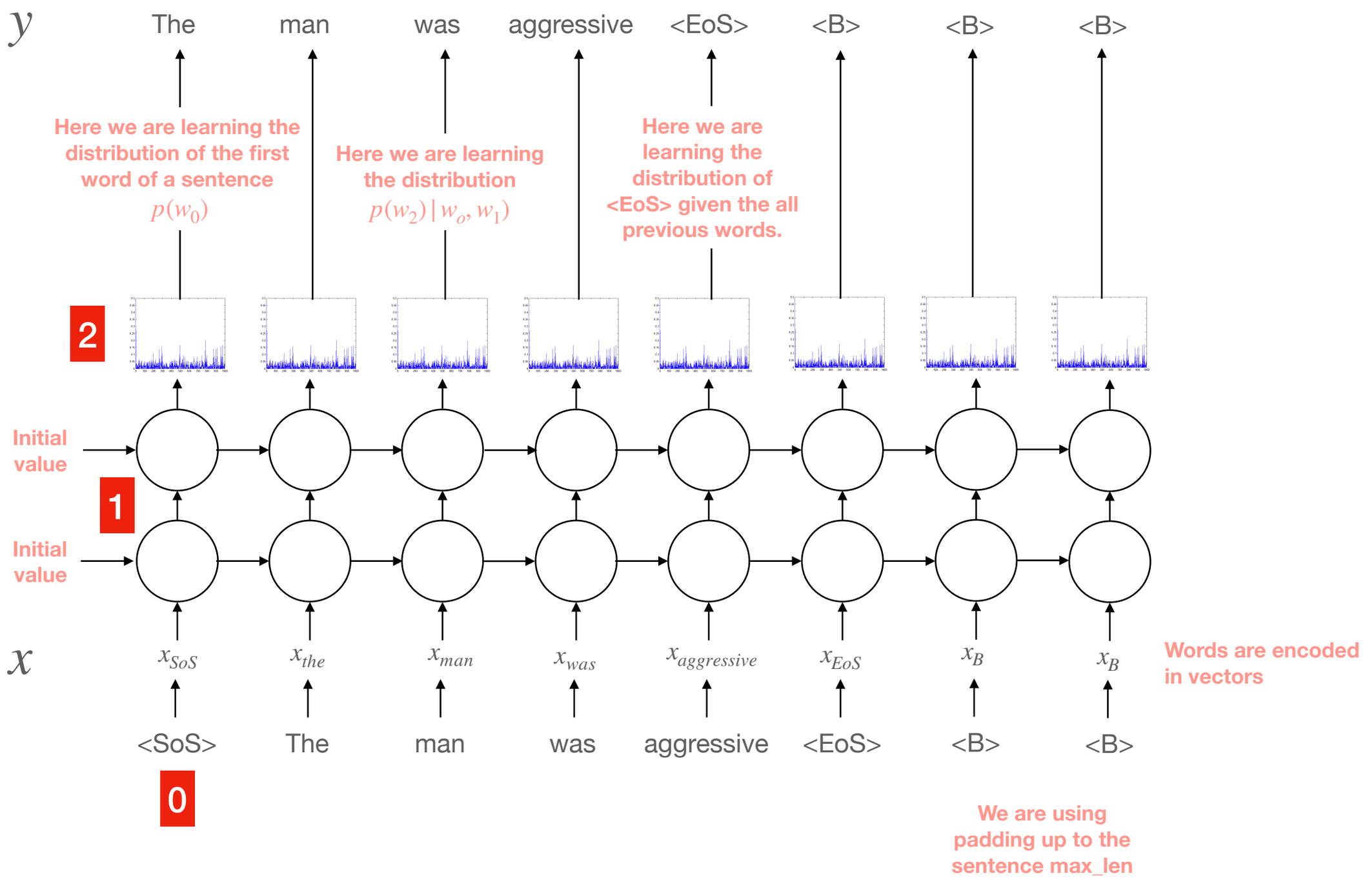
Vanilla RNN for language self-learning



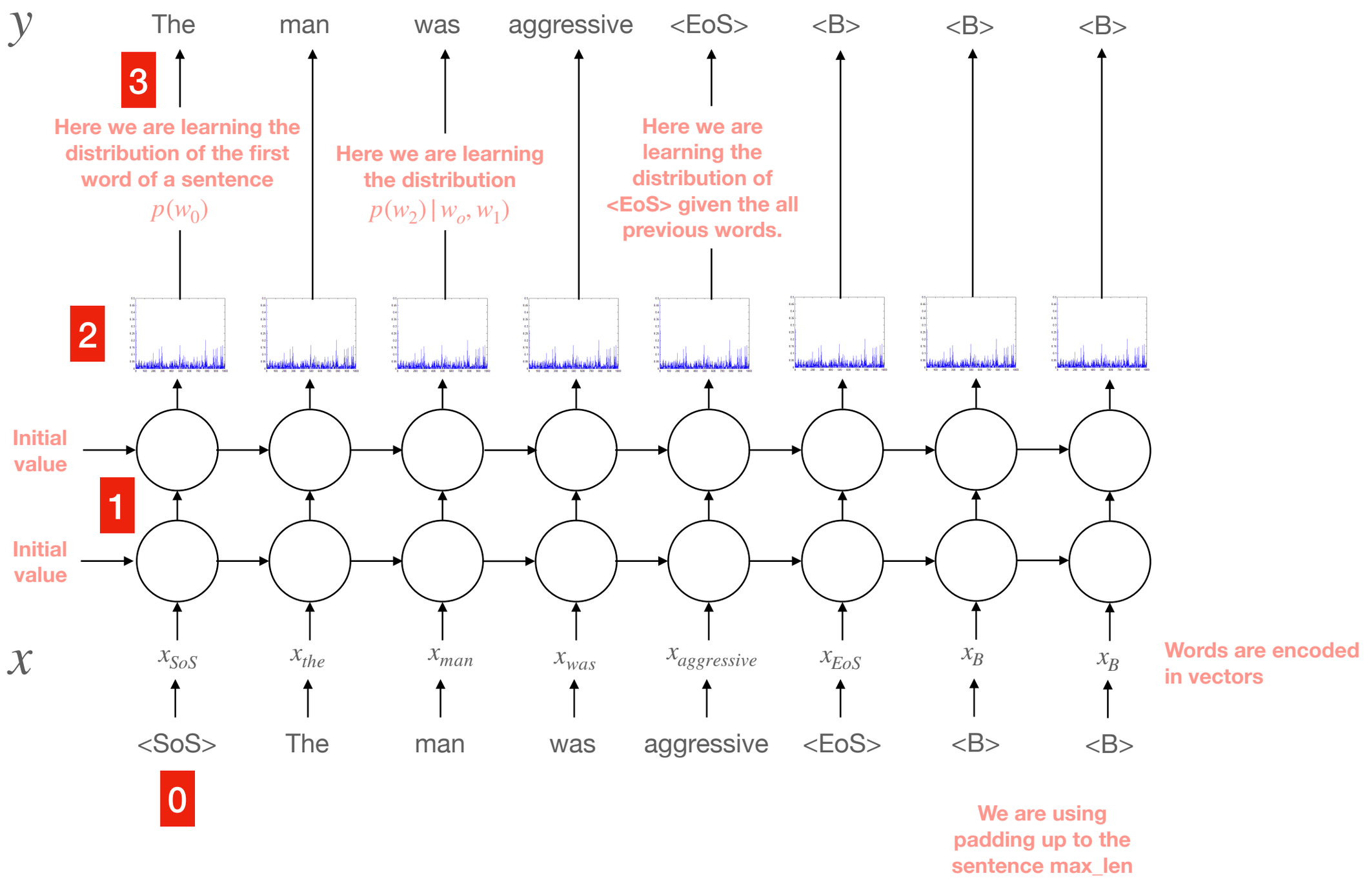
Vanilla RNN for language self-learning



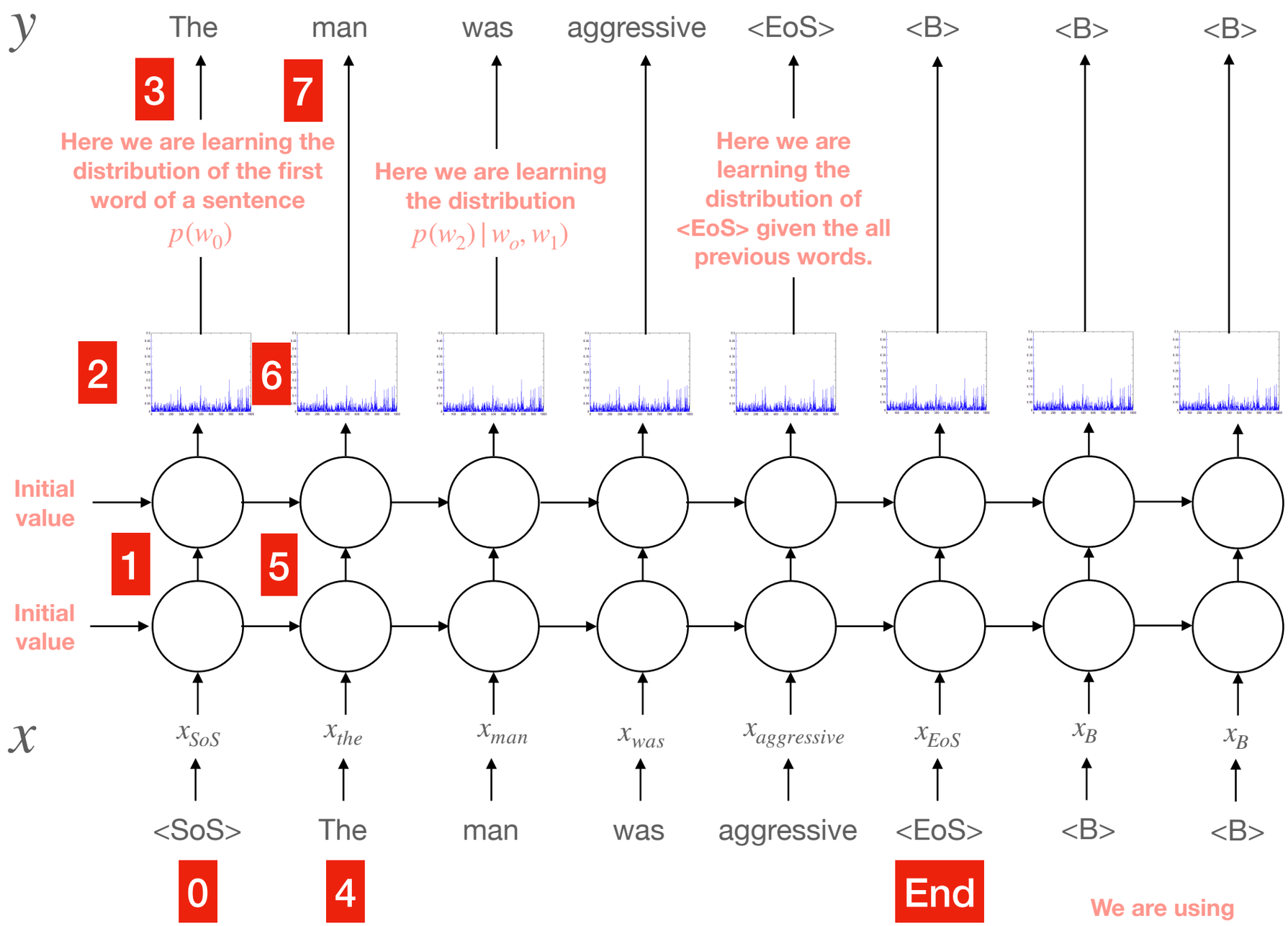
Vanilla RNN for language self-learning



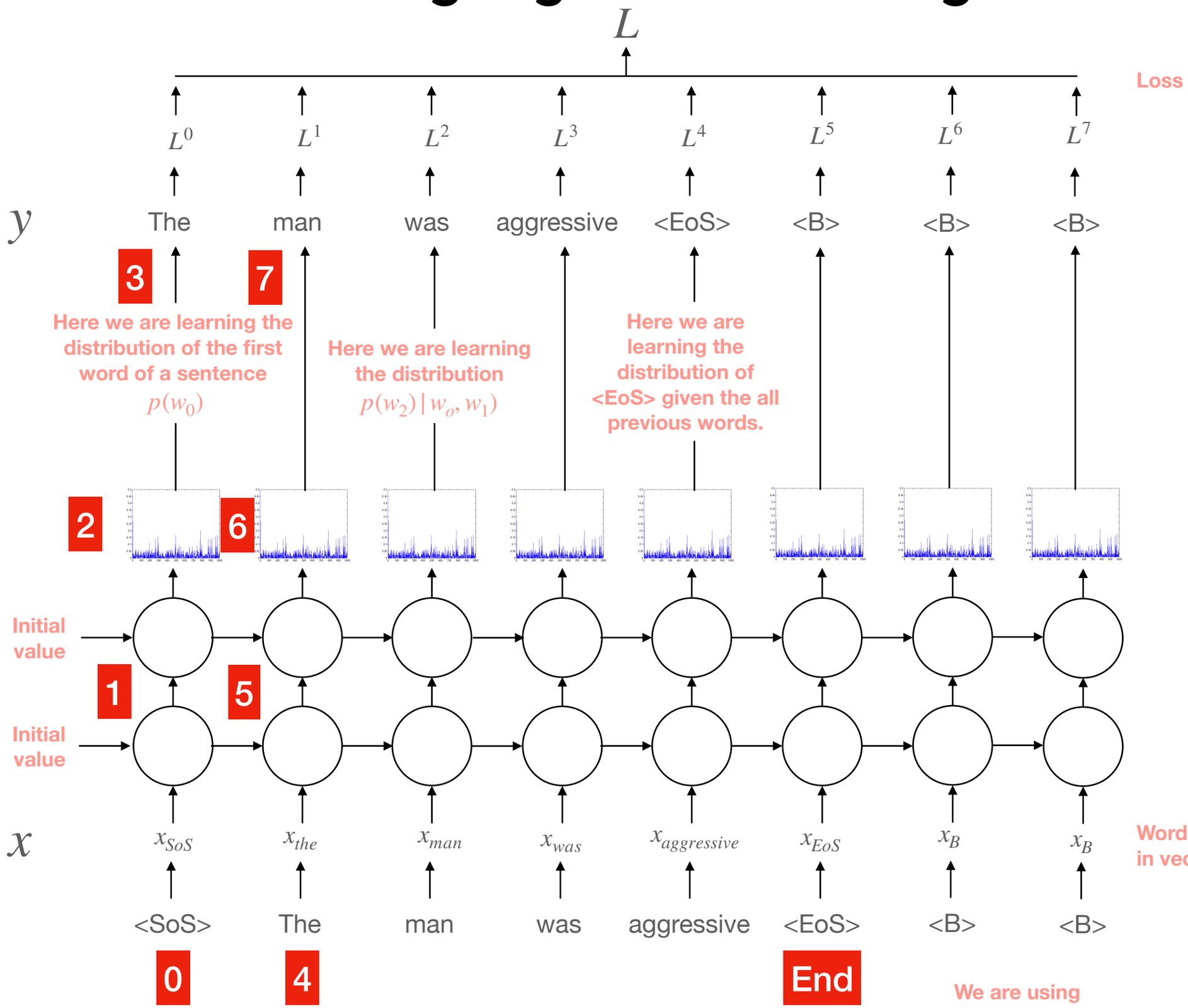
Vanilla RNN for language self-learning



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

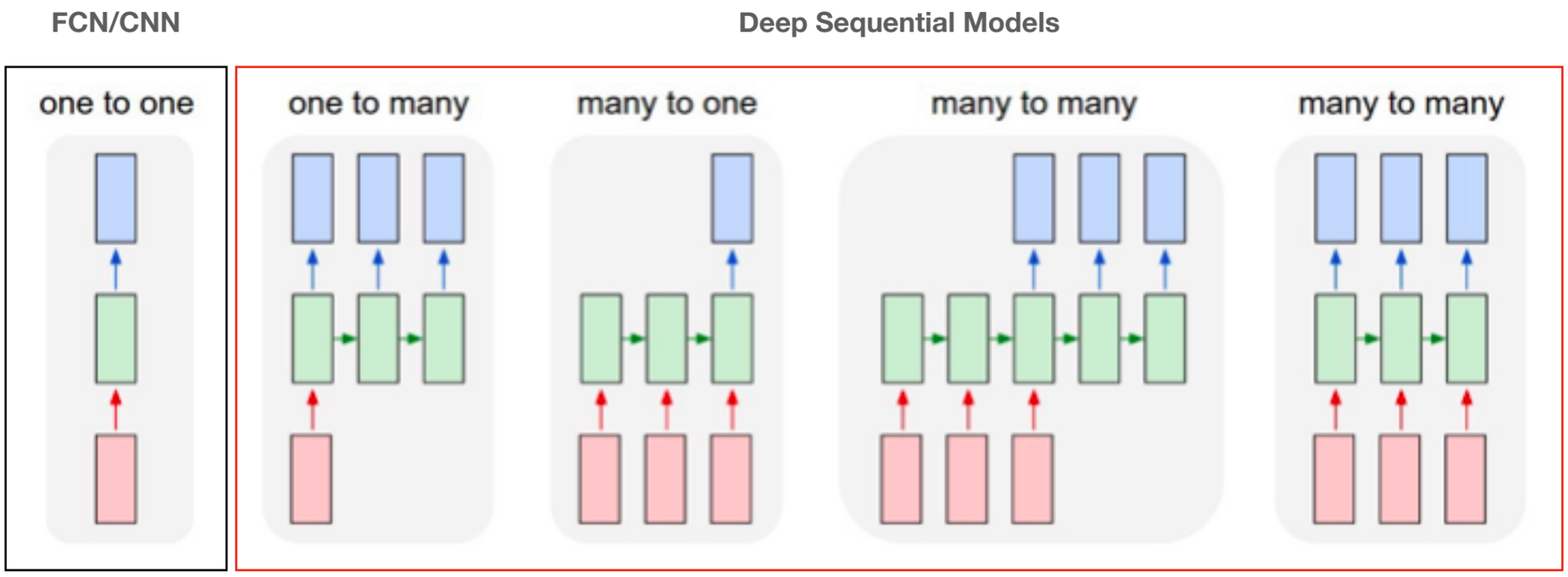
```
tf.keras.layers.SimpleRNN(
    units,
    activation="tanh",
    use_bias=True,
    kernel_initializer="glorot_uniform",
    recurrent_initializer="orthogonal",
    bias_initializer="zeros",
    kernel_regularizer=None,
    recurrent_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    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
)
```

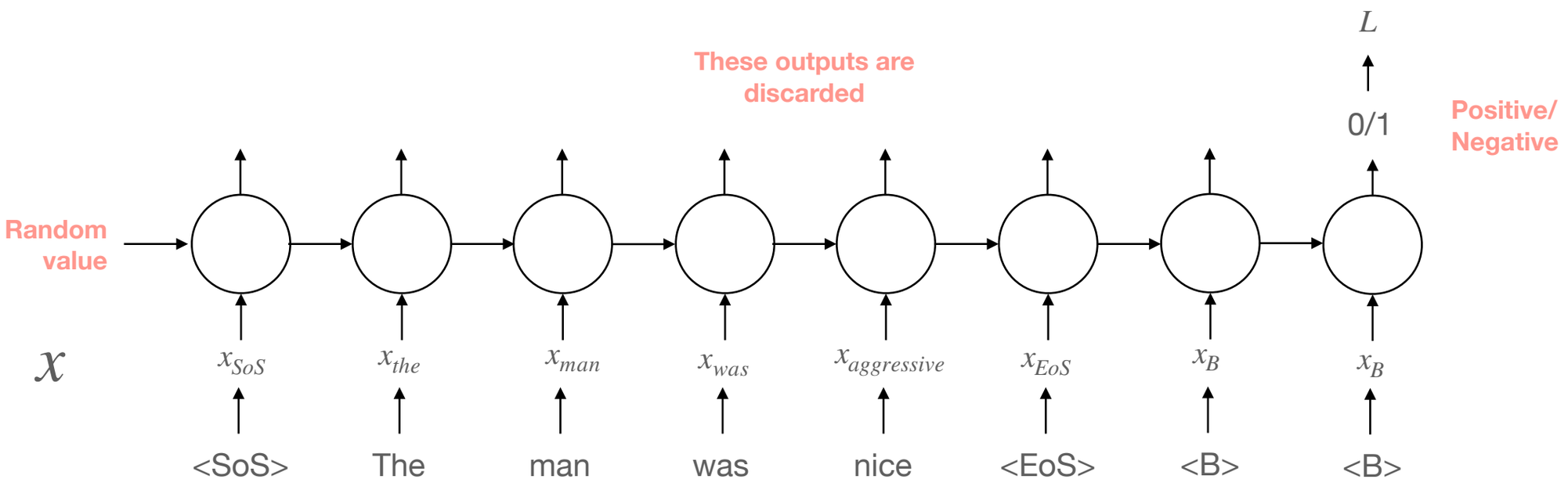
Vanilla RNN

```
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
)
```

Draws samples from a uniform distribution within `[-limit, limit]`, where `limit = sqrt(6 / (fan_in + fan_out))` (`fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units).

Deep Sequential Models

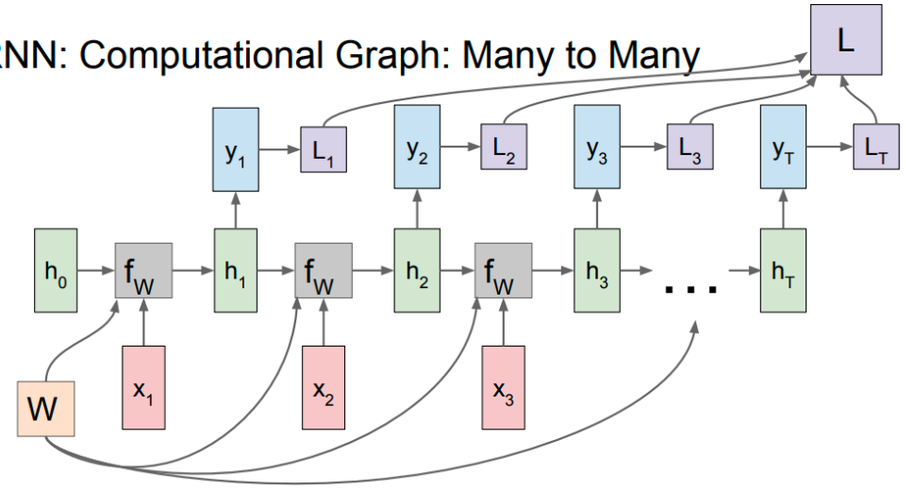




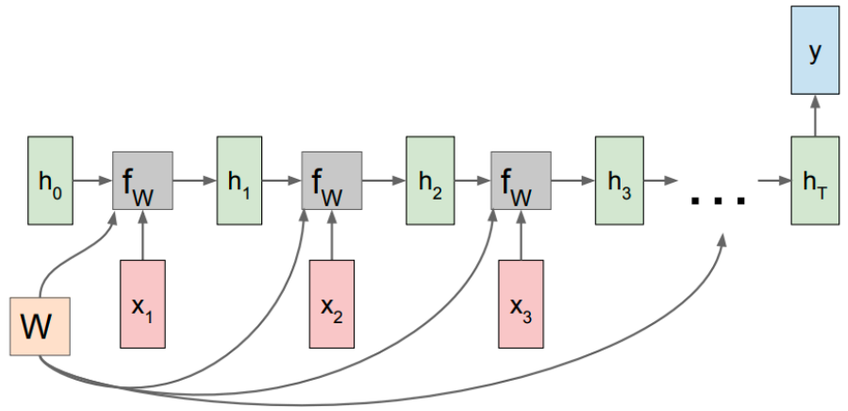
Example: Sentiment Analysis - Many to One

Deep Sequential Models

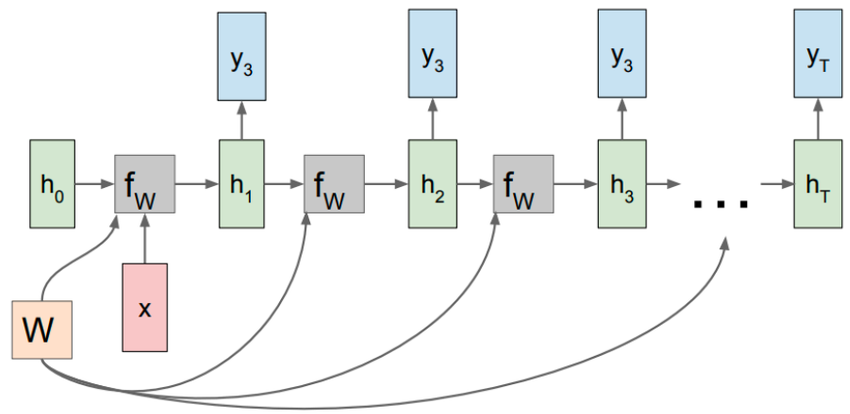
RNN: Computational Graph: Many to Many



RNN: Computational Graph: Many to One



RNN: Computational Graph: One to Many



https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

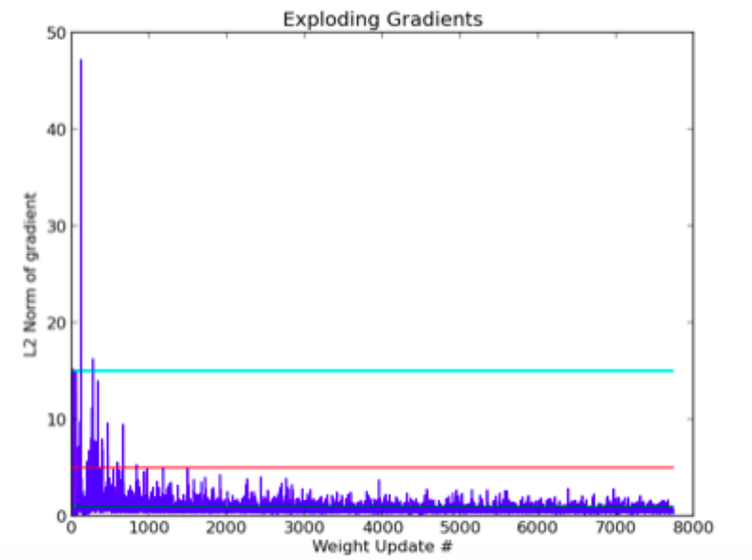
RNN Training Tricks

Recurrent neural networks propagate weight matrices from one time-step to the next. Recall the goal of a RNN implementation is to enable propagating context information through faraway time-steps. When these propagation results in a **long series of matrix multiplications**, weights can vanish or explode.

Once the gradient value grows extremely large, it causes an overflow (i.e. NaN) which is easily detectable at runtime; this issue is called the **Gradient Explosion Problem**.

When the gradient value goes to zero, however, it can go undetected while drastically reducing the learning quality of the model for far-away words in the corpus; this issue is called the **Vanishing Gradient Problem**.

To solve the problem of exploding gradients, Thomas Mikolov first introduced a simple heuristic solution that clips gradients to a small number whenever they explode. That is, whenever they reach a certain threshold, they are set back to a small number.



RNN Training Tricks

To solve the problem of vanishing gradients, instead of initializing $W^{(hh)}$ randomly, starting off from **random orthogonal matrices** works better, i.e., a square matrix W for which $W^T W = I$.

There are two properties of orthogonal matrices that are useful for training deep neural networks:

- they are norm-preserving, i.e., $\|Wx\|^2 = \|x\|^2$, and
- their columns (and rows) are all orthonormal to one another.

At least at the start of training, the first of these should help to keep the norm of the input constant throughout the network, which can help with the problem of exploding/vanishing gradients.

Similarly, an intuitive understanding of the second is that having orthonormal weight vectors encourages the weights to learn different input features.

RNN Training Tricks

You can obtain a random $n \times n$ orthogonal matrix W, (uniformly distributed) by performing a QR factorization of a $n \times n$ matrix with elements i.i.d. Gaussian random variables of mean 0 and variance 1.

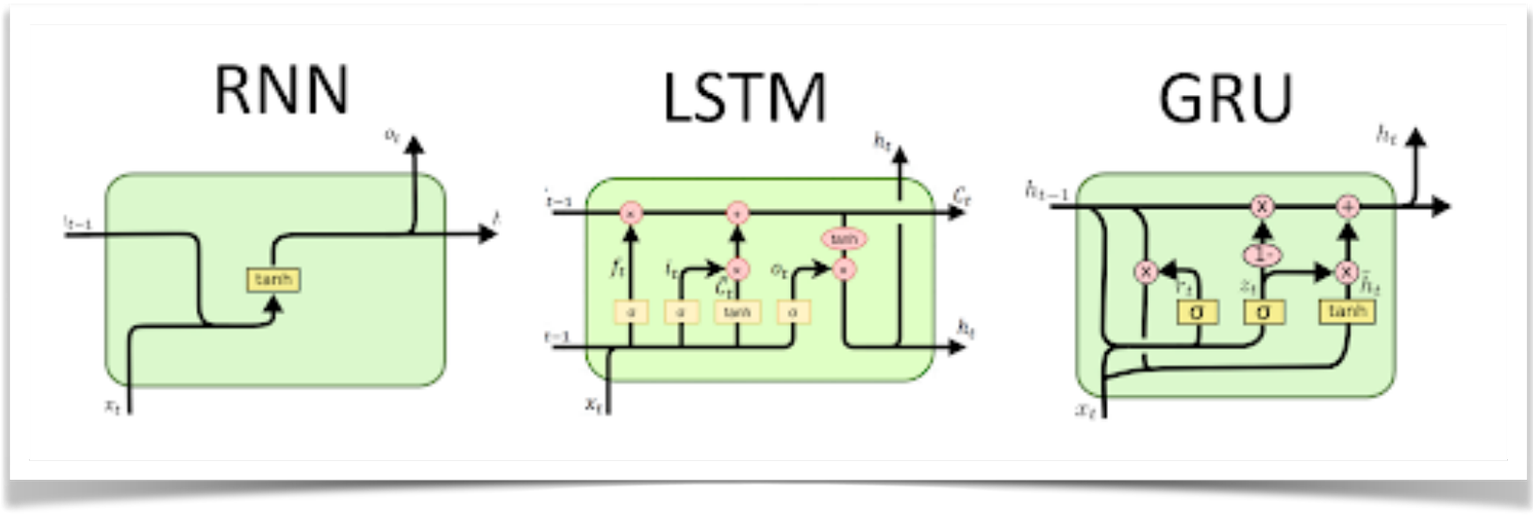
```python
import numpy as np
from scipy.linalg import qr

n = 3
H = np.random.randn(n, n)
print(H)
print('\n')

Q, R = qr(H)

print (Q.dot(Q.T))
```

```
[[ 0.06351849  0.75175046 -0.10405964]
 [ 1.11485701  1.78923717 -0.9788983 ]
 [-0.08515308 -1.16475846  0.0640979 ]]


[[ 1.00000000e+00  5.49819565e-16 -1.29542567e-16]
 [ 5.49819565e-16  1.00000000e+00  6.99257794e-17]
 [-1.29542567e-16  6.99257794e-17  1.00000000e+00]]
```

Gated Units

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

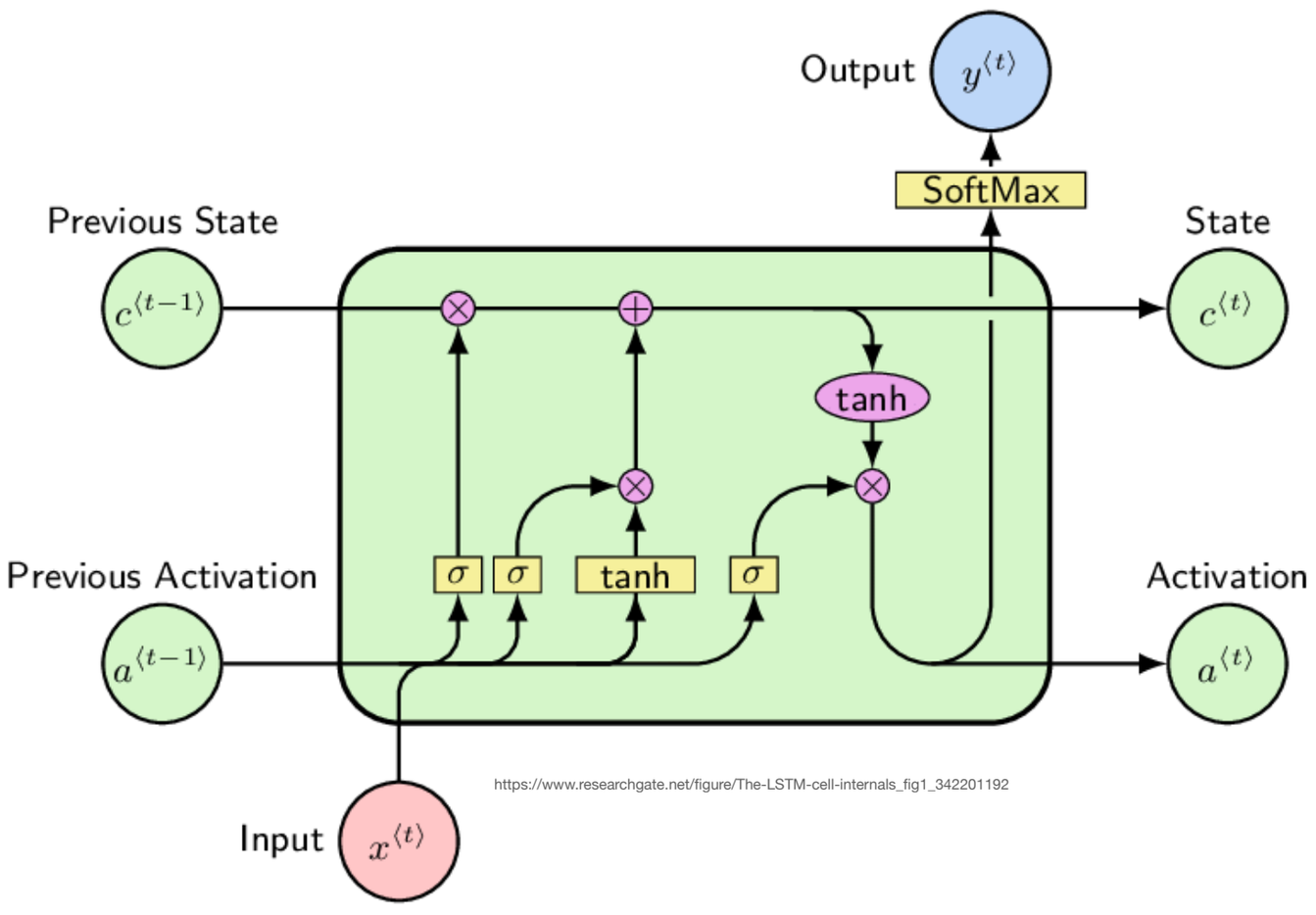
- **Long Short Term Memories** (LSTM). It was introduced by S.Hochreiter and J.Schmidhuber in 1997 and is widely used. LSTM is very good in the long run due to its high complexity.
- **Gated Recurrent Units** (GRU). It was introduced by K.Cho in 2014. It is simpler than LSTM, fasters and optimizes quicker.



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.

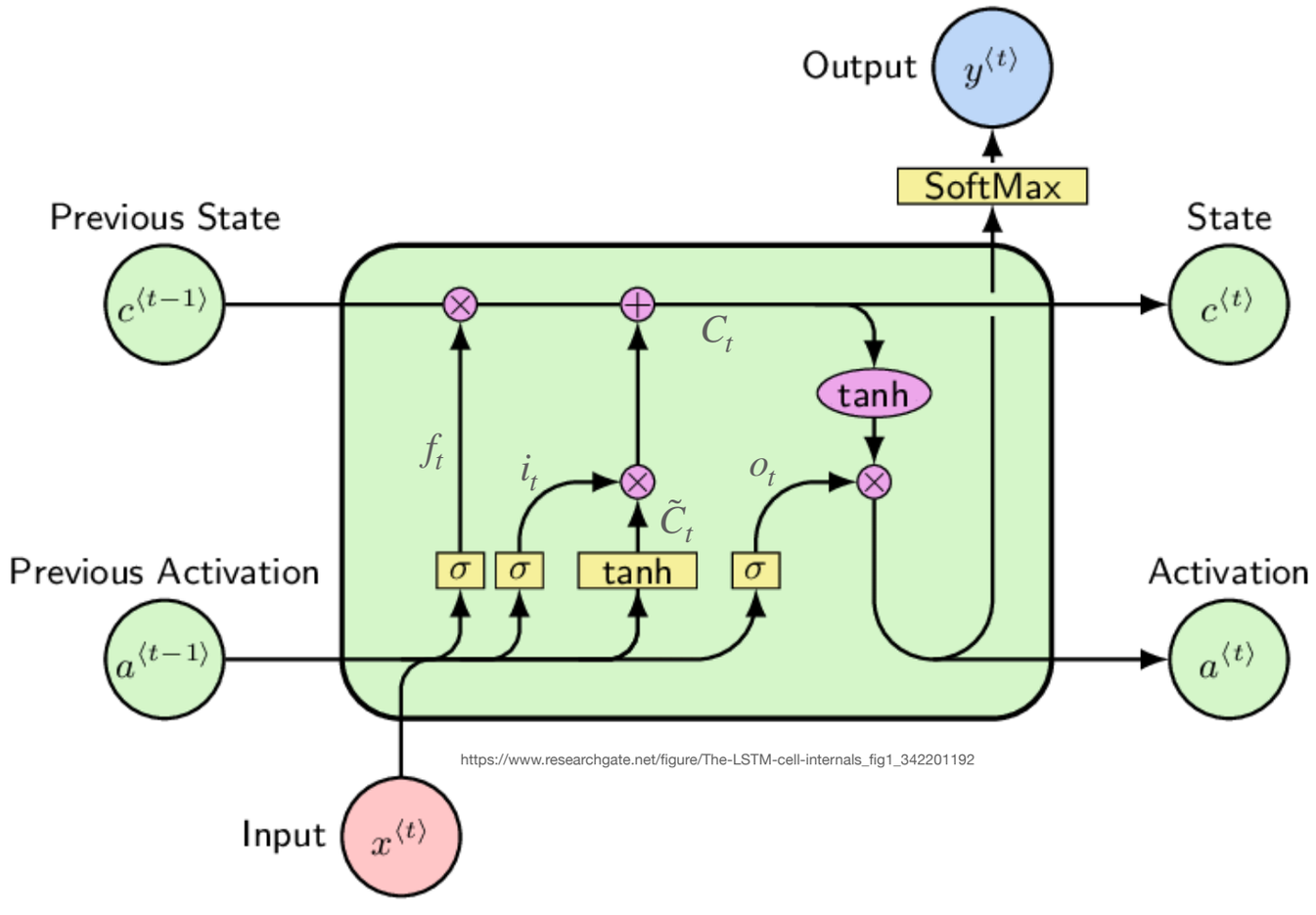


https://www.researchgate.net/figure/The-LSTM-cell-internals_fig1_342201192

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)

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

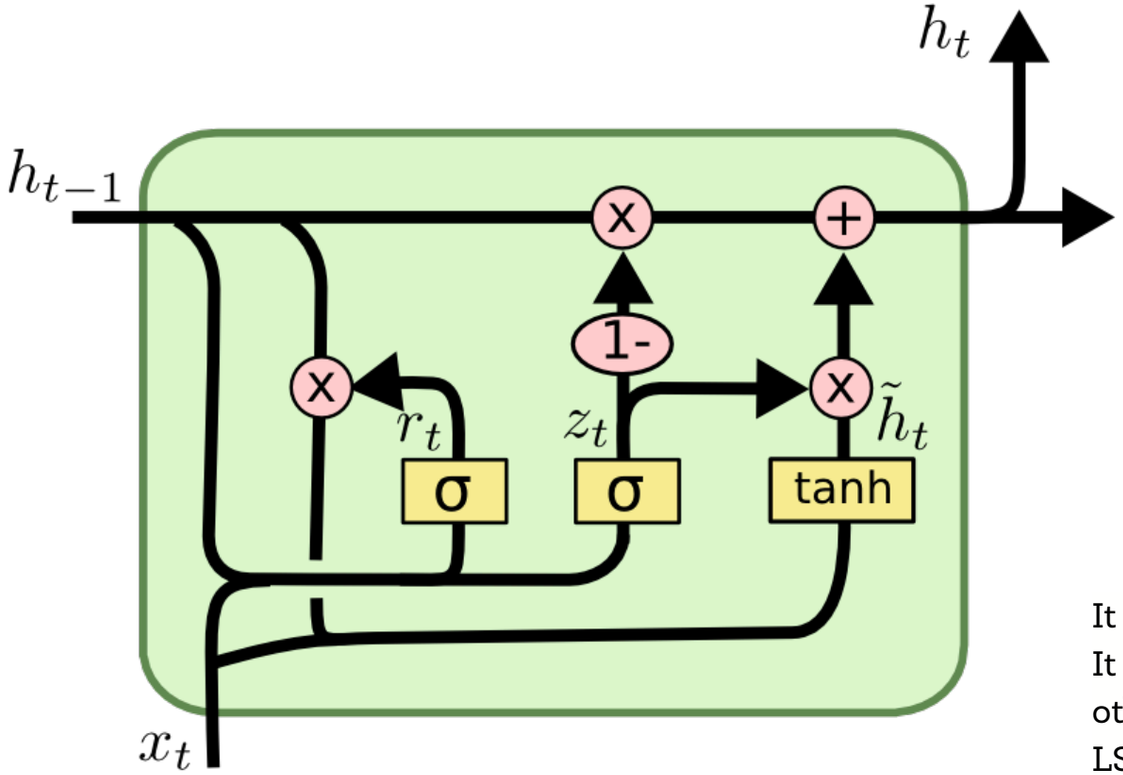
$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$ (Update gate)

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

$h_t = o_t * \tanh(C_t)$ (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.



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

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

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

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

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

When processing very long sequences (possibly infinite), you may want to use the pattern of **cross-batch statefulness**.

Normally, the internal state of a RNN layer is reset every time it sees a new batch (i.e. every sample seen by the layer is assumed to be independent of the past). The layer will only maintain a state while processing a given sample.

If you have very long sequences though, it is useful to break them into shorter sequences, and to feed these shorter sequences sequentially into a RNN layer without resetting the layer's state. That way, the layer can retain information about the entirety of the sequence, even though it's only seeing one sub-sequence at a time.

You can do this by setting **stateful**=True in the constructor.

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

Dropout:

> The proper way to use dropout with a recurrent network is:
>
> > The same **dropout** mask (the same pattern of dropped units) should be applied at every time step, instead of a dropout mask that varies randomly from time step to time step.
>
> Using the same dropout mask at every time step allows the network to properly propagate its learning error through time; a temporally random dropout mask would disrupt this error signal and be harmful to the learning process.

RNN in Keras

Staking:

If you're no longer overfitting, but seem to have hit a performance bottleneck, you should consider increasing the capacity and expressive power of the network.

Recall the description of the universal machine-learning workflow: **it's generally a good idea to increase the capacity of your model until overfitting becomes the primary obstacle** (assuming you're already taking basic steps to mitigate overfitting, such as using dropout). As long as you aren't overfitting too badly, you're likely under capacity.

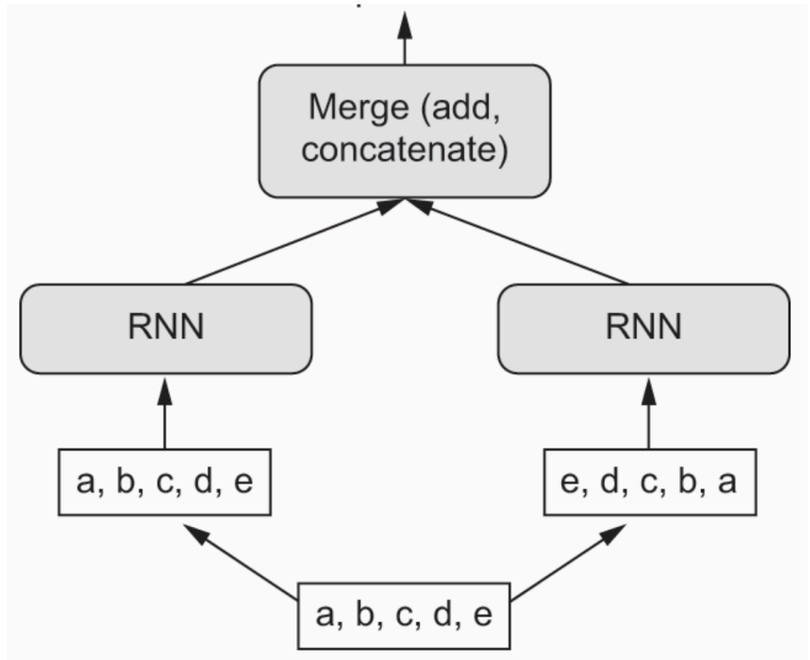
RNN in Keras

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!

Example in Keras

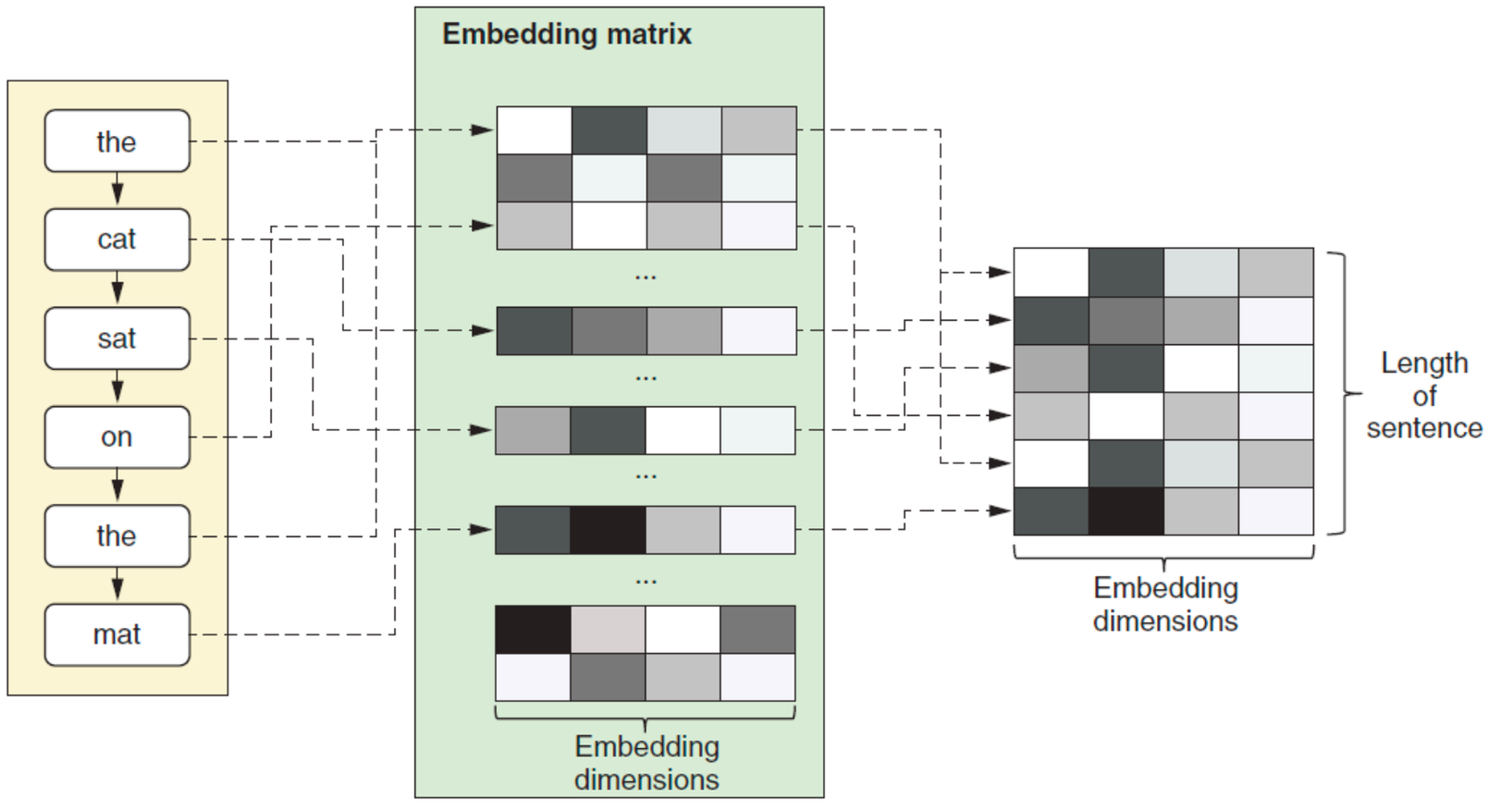
```
model = Sequential()

model.add(Embedding(vocab_size, embedding_dim))
model.add(Dropout(0.5))
model.add(LSTM(embedding_dim,return_sequences=True))
model.add(LSTM(embedding_dim))
model.add(Dense(6, activation='softmax'))

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
```

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

Time Series Analysis

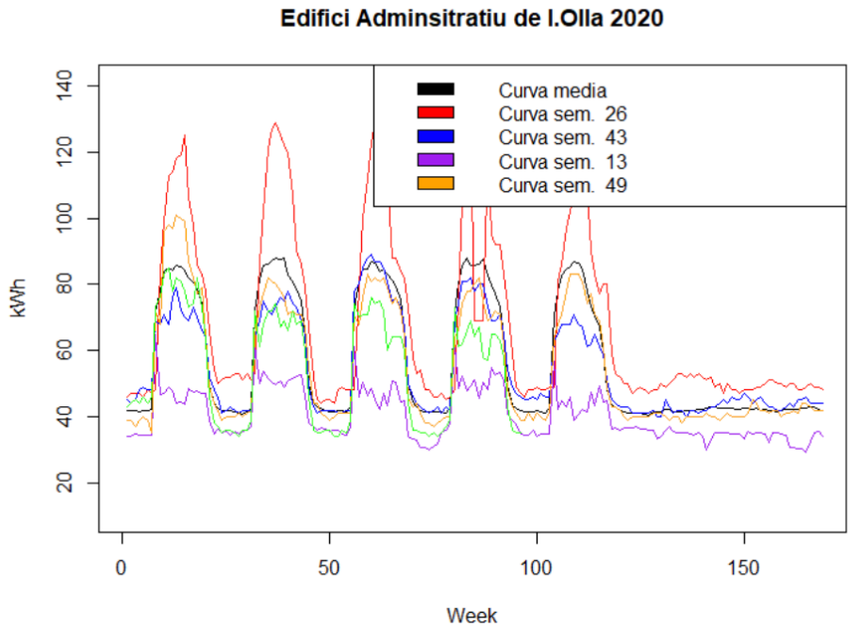
- **Forecasting:** prediction of events through a sequence of time. It predicts future events by analyzing the trends of the past, on the assumption that future trends will hold similar to historical trends.

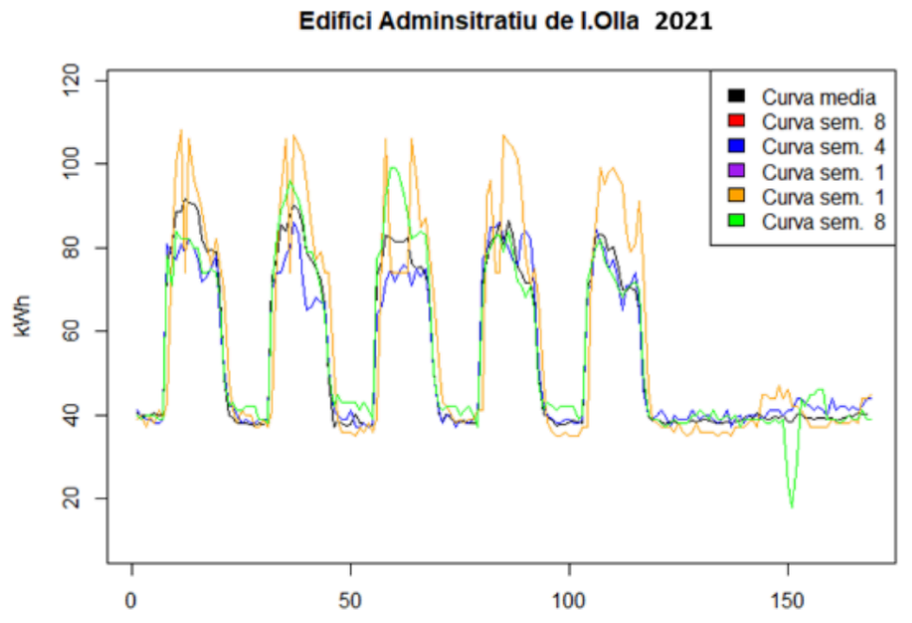
- **Classification**: assign one or more categorical labels to a time series.
  For instance, given the time series of activity of a visitor on a website, classify whether the visitor is a bot or a human.

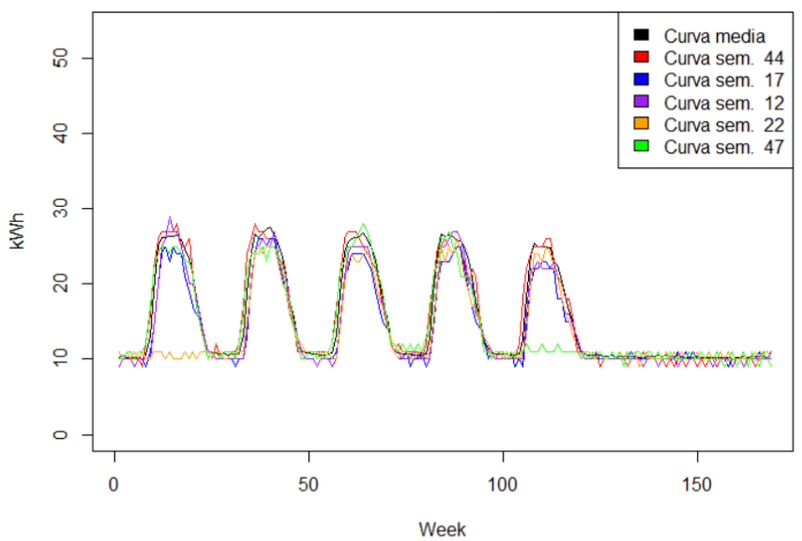
- **Event detection**: identify the occurrence of a specific, expected event within a continuous data stream.
  A particularly useful application is “hot word detection", where a model monitors a audio stream and detects utterances like "Ok Google" or "Hey Alexa”.

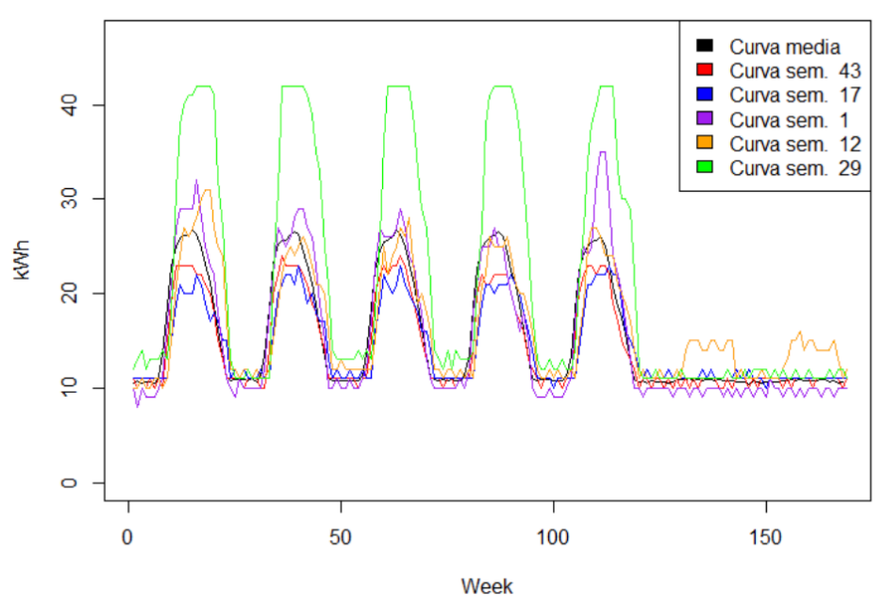
- **Anomaly detection**: detect anything unusual happening within a continuous data stream. Unusual activity on your corporate network? Might be an attacker. Unusual readings on a manufacturing line? Time for a human to go take a look. Anomaly detection is typically done via unsupervised learning, because 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 $$y_{t+1} = f(y_{1,\ldots,t})$$

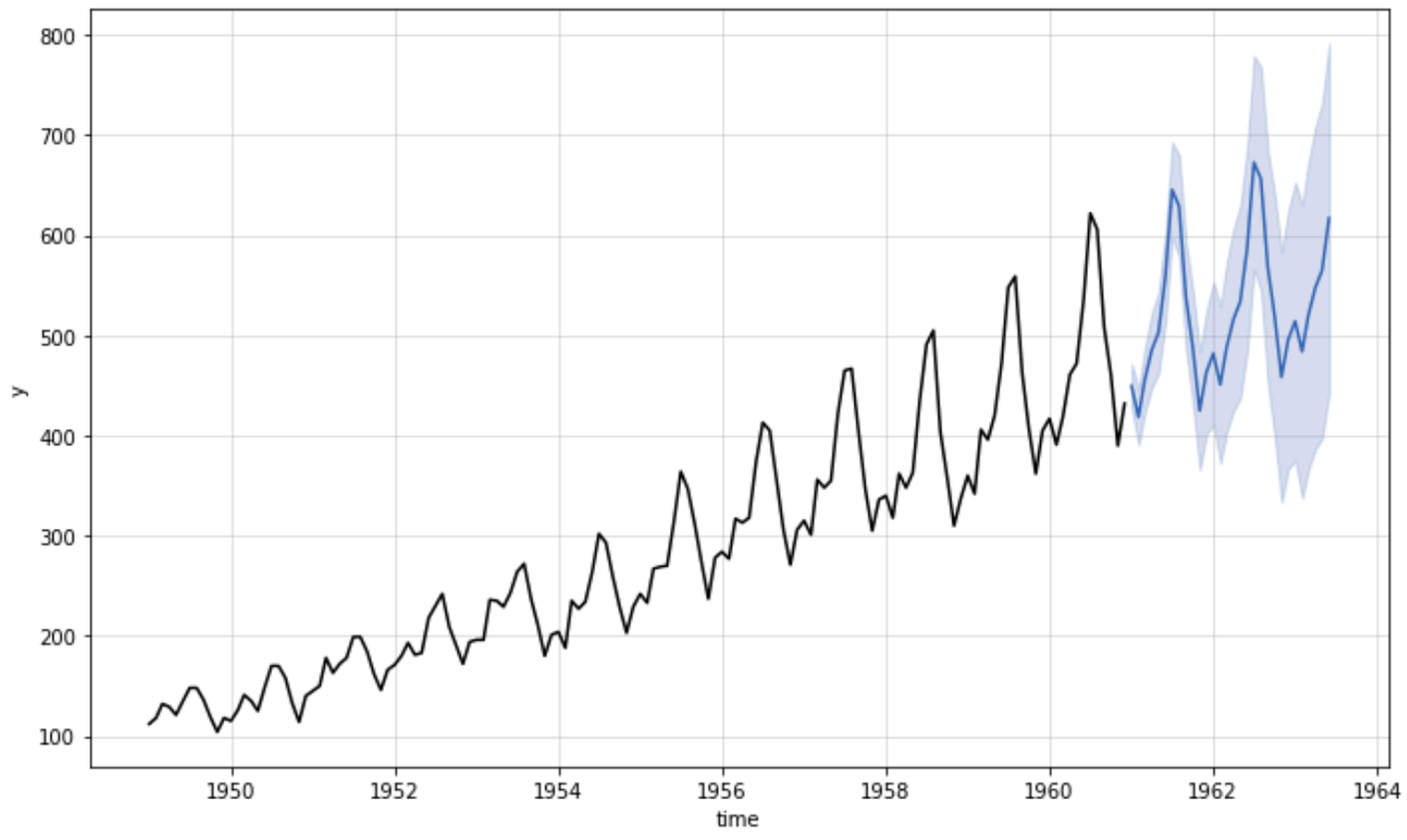
Typical case in Deep Learning $$y_{t+1} = f(x_{1,\ldots,t+1}, y_{1,\ldots,t})$$

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.

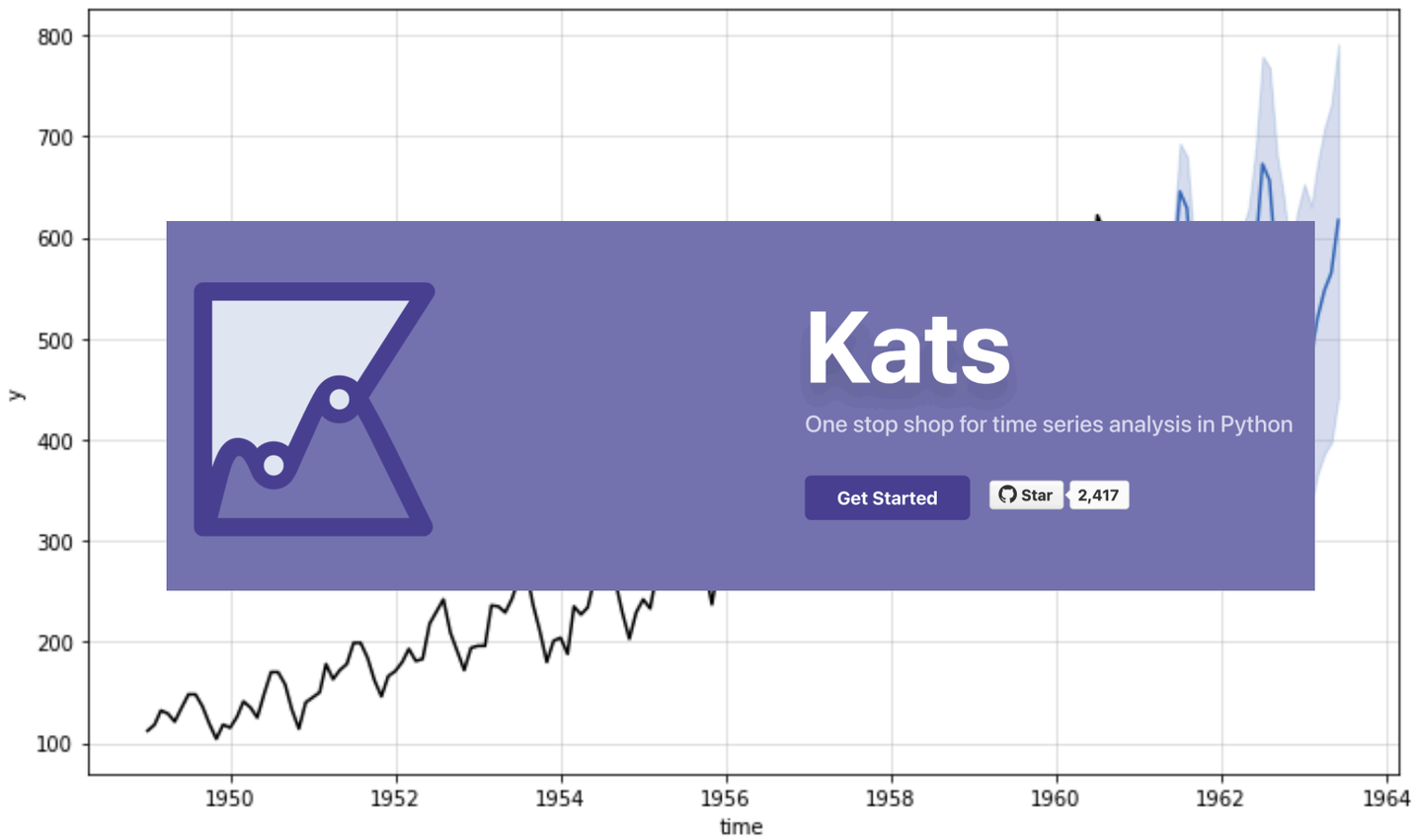
Time Series Analysis

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

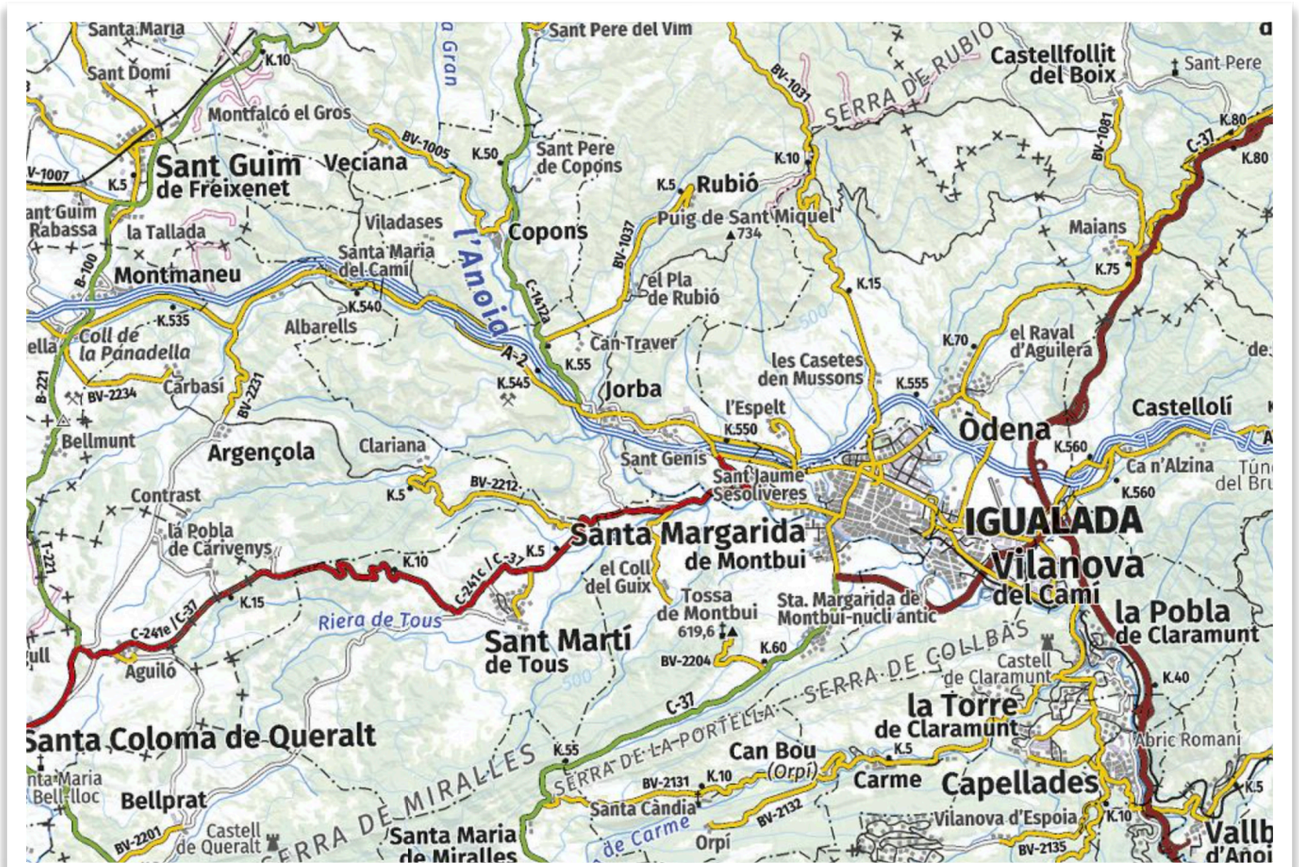


Time Series Analysis

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

-

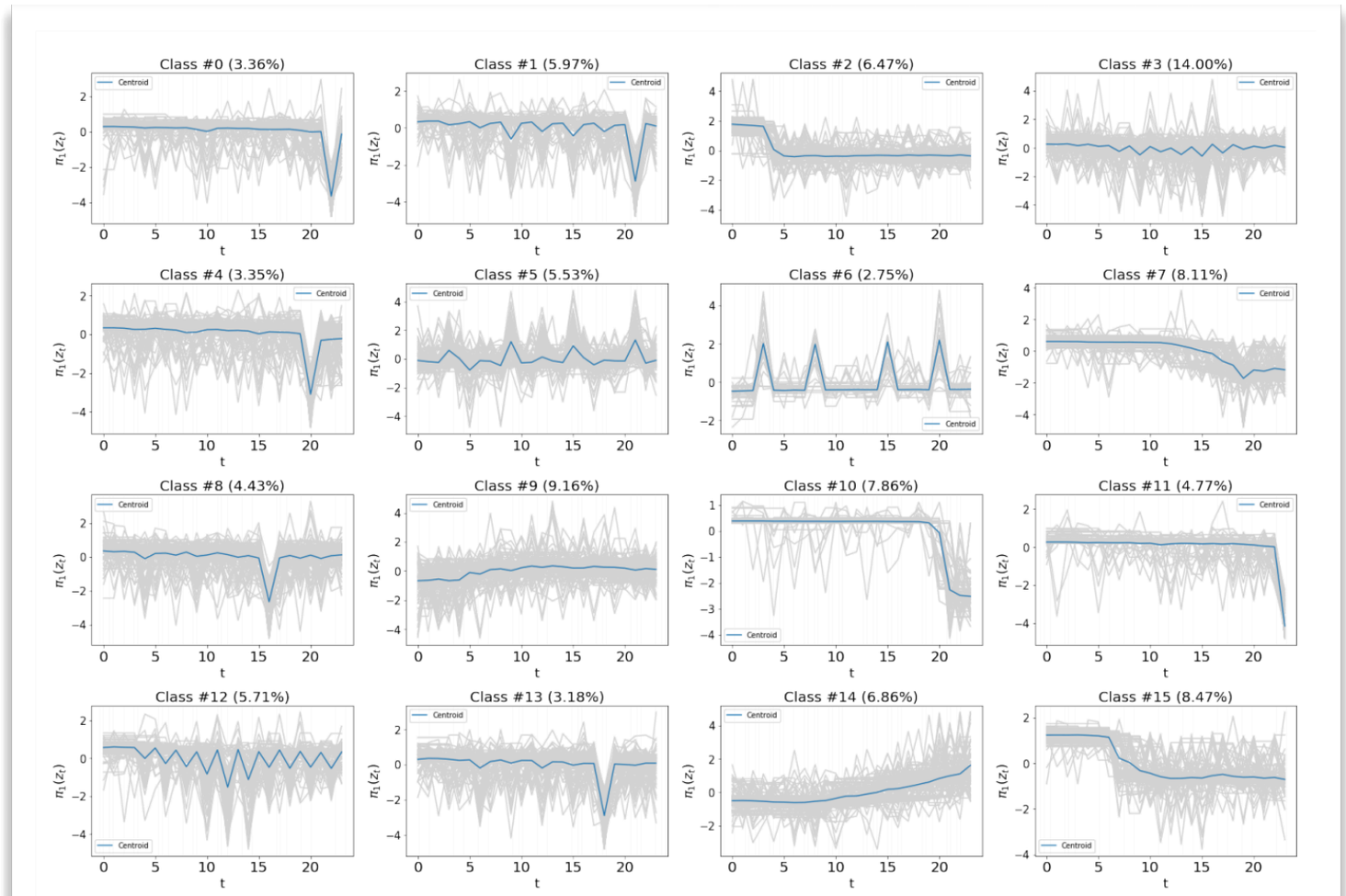
- Terraubella, Corral de
- Terraubes
- **Terravanca**
- Terrer Nou, Can
- Terrer Roig, lo

Name generator from a few letters

Time Series Analysis

Multiple Times Series

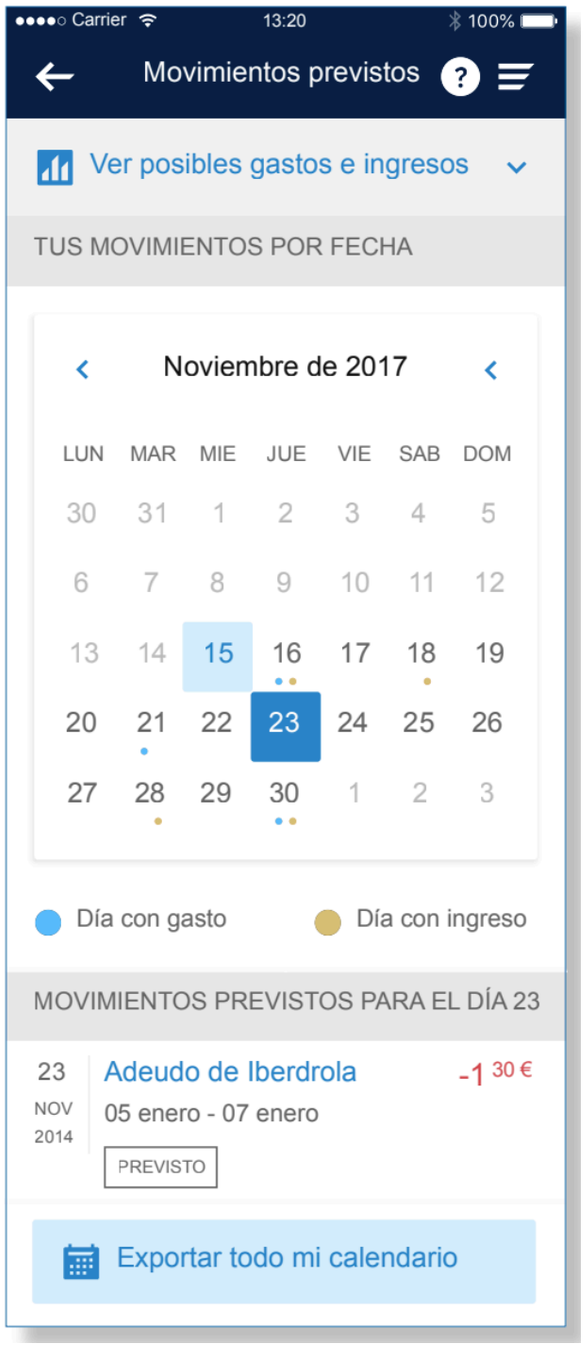
$$y_{t+1} = f(x^{1,\ldots,k}_{1,\ldots,t+1}, y^{1,\ldots,k}_{1,\ldots,t})$$



Time Series Analysis

Multiple Times Series

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



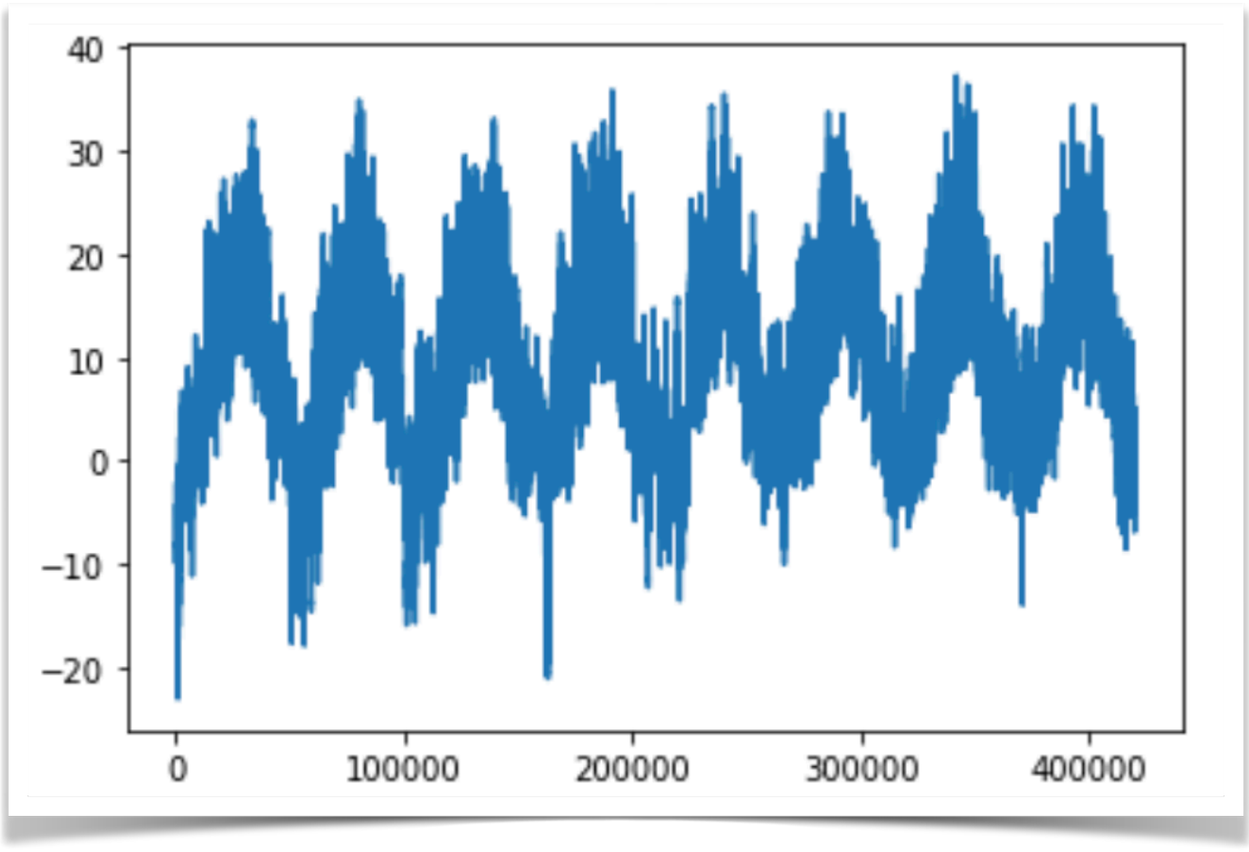
Time Series Analysis

Example: predicting the temperature 24 hours in the future, given a time series of hourly measurements of quantities such as **atmospheric pressure** and **humidity**, recorded over the recent past by a set of sensors on the roof of a building.

We'll work with a weather time series dataset recorded at the Weather Station at the Max Planck Institute for Biogeochemistry in Jena, Germany. In this dataset, 14 different quantities (such as temperature, pressure, humidity, wind direction, and so on) were recorded every 10 minutes, over several years. The subset of the data we'll download is limited from 2009 to 2016.

https://s3.amazonaws.com/keras-datasetsjena_climate_2009_2016.csv.zip

Time Series Analysis



Time Series Analysis