Machine Learning for Robotics: **Recurrent Neural Networks**

Prof. Navid Dadkhah Tehrani

In many applications we deal with sequential data:

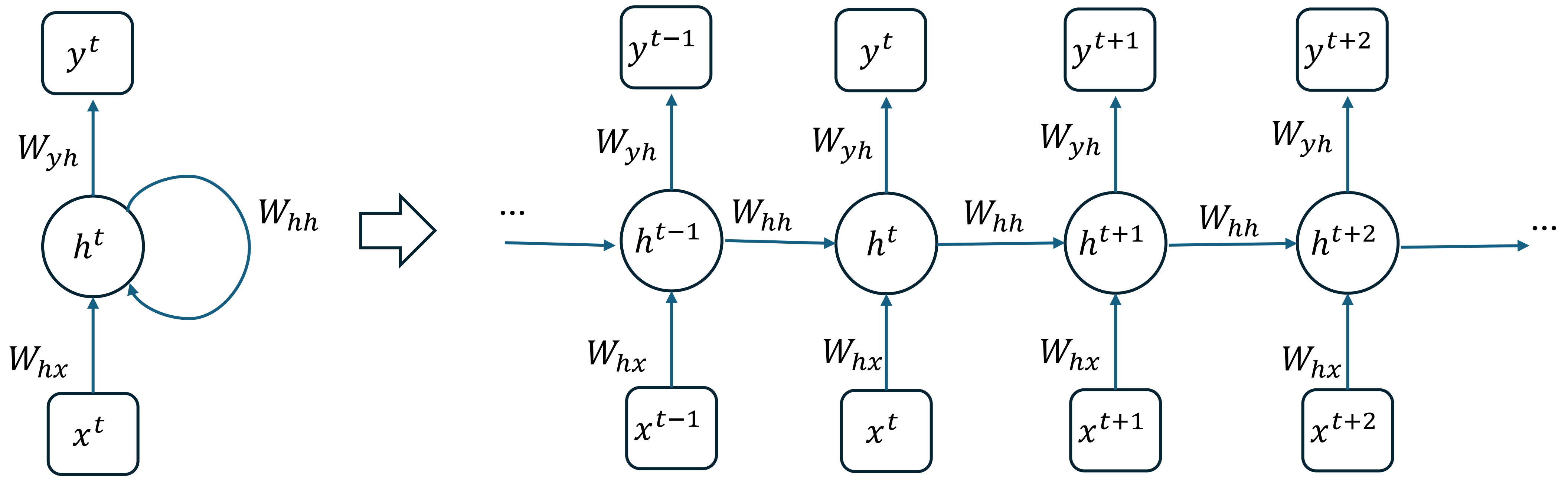
- Speech analysis
- Text analysis
- Dynamic model prediction
- Video analysis

- RNNs are deep learning models that can capture the dynamics of the sequence via recurrent connection.

- MLPs and CNNs are stateless but RNNs have internal (hidden) states.

Single Layer RNN

In RNNs we have two weight matrices for hidden layer W_{hx}, W_{hh}; one connecting input to the hidden layer and the other one connecting previous hidden layer to the current hidden layer.



Note that the weights W_{hx}, W_{hh}, W_{yh} are the same across time.

Single Layer RNN

$$z_h^t = W_{hx}\ x^t + W_{hh}\ h^{t-1} + b_h$$
$$h^t = \sigma_h\ (z_h^t)$$
$$z_y^t = W_{yh} h^t + b_y$$
$$y^t = \sigma_y(z_y^t)$$

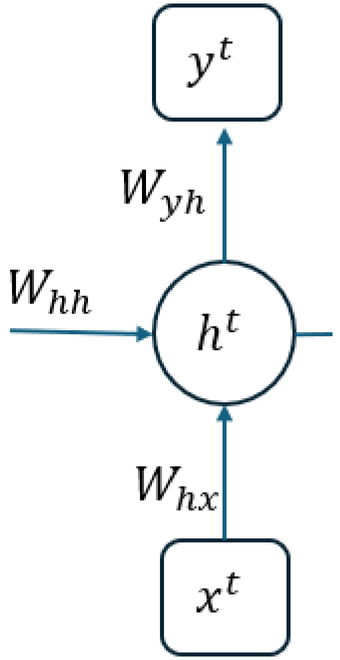
$given\ h^t\ at\ t = 0$:

Note that since RNN has hidden state, it has to be initialized.

----

recall from MLP, it does not have hidden state $z_h^t = W_{hx}\ x^t + b_h$

$y^t = \sigma\ (z_h^t)$



Backpropagation through time:

Q: how is the loss calculated in RNN since we have more that one output?
A: it really depend on the application. You can use all the outputs $y^1, \dots, y^t$ to calculate the loss or you can use the last or first one.

$$L = \sum_{t=1}^{T} L^t$$

$$\frac{\partial L^t}{\partial W_{hh}} = \frac{\partial L^t}{\partial y^t}\frac{\partial y^t}{\partial h^t}\left(\sum_{k=1}^{t}\frac{\partial h^t}{\partial h^k}\frac{\partial h^k}{\partial W_{hh}}\right)$$

Where $\frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^{t}\frac{\partial h^i}{\partial h^{i-1}}$ (for example: $\frac{\partial h^5}{\partial h^1} = \frac{\partial h^5}{\partial h^4} * \frac{\partial h^4}{\partial h^3} * \frac{\partial h^3}{\partial h^2} * \frac{\partial h^2}{\partial h^1}$)

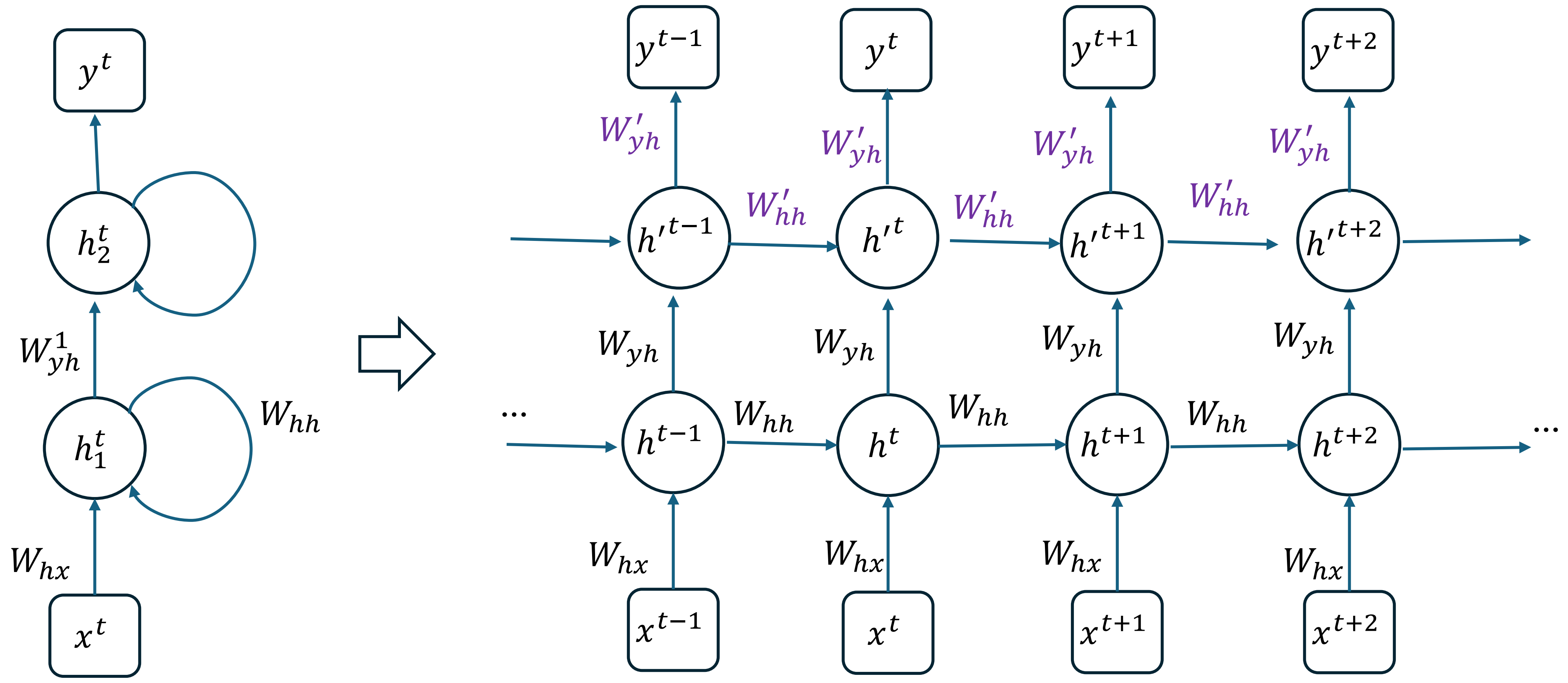
For more details: Werbos, P. "Backpropagation through time: what it does and how to do it" 1990.
Also section 9.7 of the textbook.

The multiplication of adjacent time steps above is problematic and can lead to vanishing gradient.

How to avoid vanishing gradient in RNNs:

- Gradient clipping: set a max value for gradients. This can be done for other type of NN as well.
- Truncated backpropagation through time: limits the number of time steps the signal can backpropagate after each forward pass. This leads to approximation of gradient.
- Using LTSMs (Long Short Term Memory).

Multiple Layer RNNs



Memory mechanism in RNN:

RNNs have a hidden state that gets updated at each time step based
on the current input and the previous hidden state.

Due to the vanishing gradient problem, RNNs struggle to learn long-term dependencies.
They tend to forget information quickly as the sequence length increases.

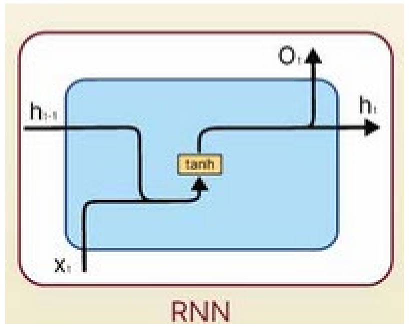
LSTMs have a more complex memory structure, including a cell state (often referred to as c_t)
and a hidden state (often referred to as h_t).

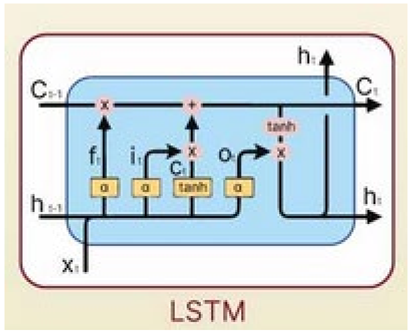
The cell state acts as a long-term memory, which is controlled by three gates: input gate, forget gate, and output gate.
These gates regulate the flow of information into and out of the cell state.

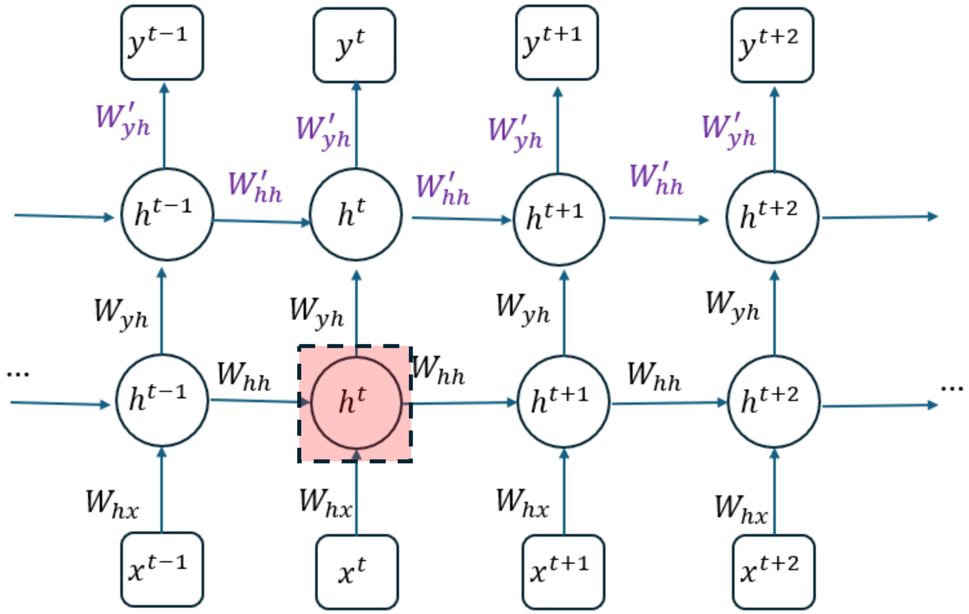
LSTM cell

Before start, let look at LSTM cell versus RNN cell.

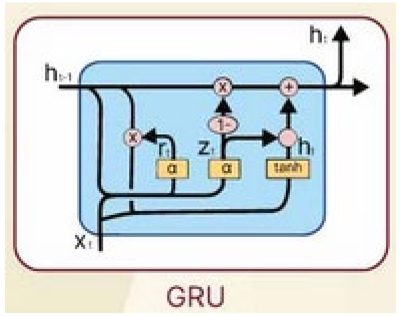
This is where a cell is located in the big picture.





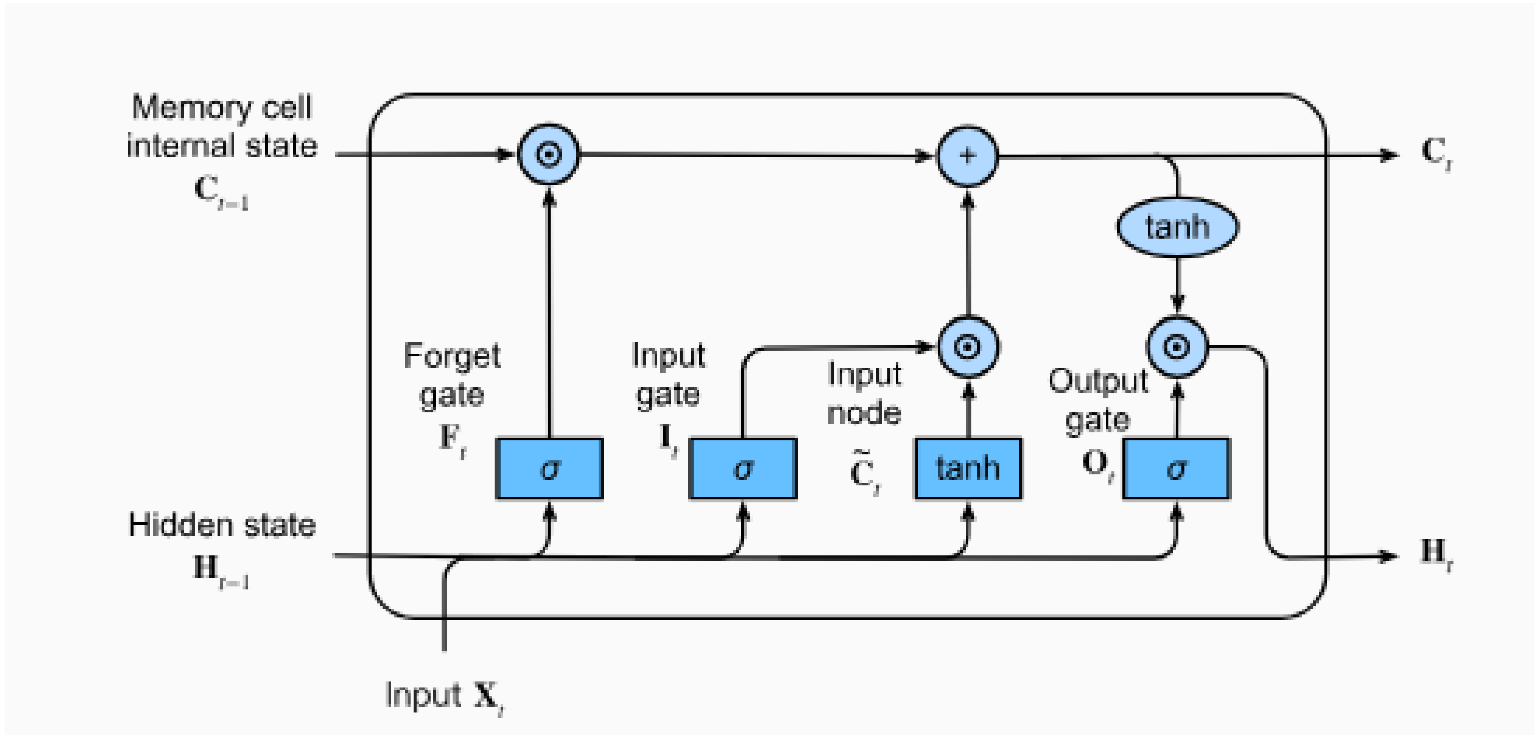


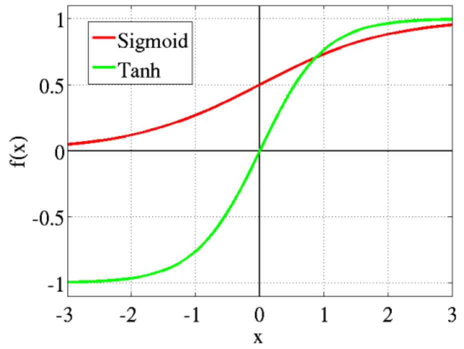
A simpler version of LSTM called GRU (gated recurrent unit)



Essentially, an LSTM cell is an RNN cell with extra cell state (c_t).
And RNN is an MLP with extra hidden state and weight associated with it.

LSTM cell





If the forget gate is always 1 and input gate is always 0, the memory cell internal state C_{t-1} remains constant.

When the output gate is 1, we allow the memory cell state to impact the subsequent layers. And when output gate is 0, we prevent the current memory from impacting other layers.

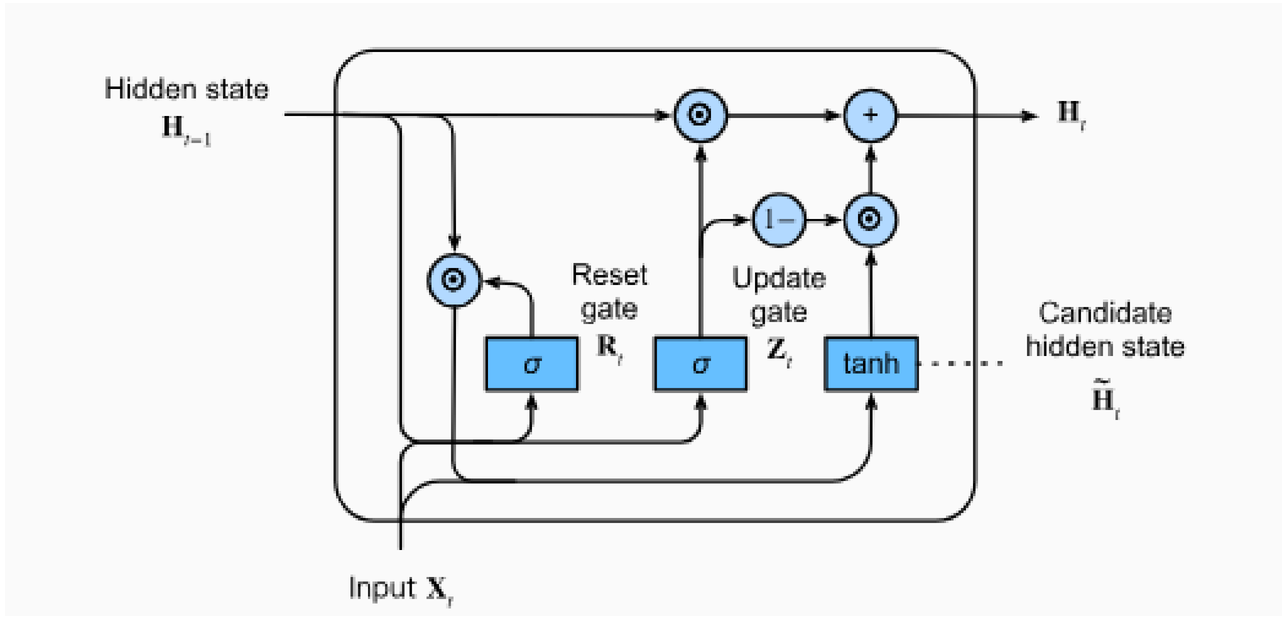
$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{\text{xi}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hi}} + \mathbf{b}_\text{i}),$$
$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{\text{xf}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hf}} + \mathbf{b}_\text{f}),$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{\text{xo}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{ho}} + \mathbf{b}_\text{o}),$$

$$\tilde{\mathbf{C}}_t = \text{tanh}(\mathbf{X}_t \mathbf{W}_{\text{xc}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hc}} + \mathbf{b}_\text{c}),$$

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

GRU cell



Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
σ
σ
tanh
1−
+
Candidate hidden state $\tilde{\mathbf{H}}_t$
Input $\mathbf{X}_t$

If reset gate is close to 1, we recover vanilla RNN.
If the reset gate is close to 0, the candidate hidden state is an MLP.

When the update gate is 1, we retain the old state; this means that the information from X_t is ignored, i.e. skipping time step t in the dependency chain.
If the update gate is 0, the new hidden state approaches the candidate hidden state.

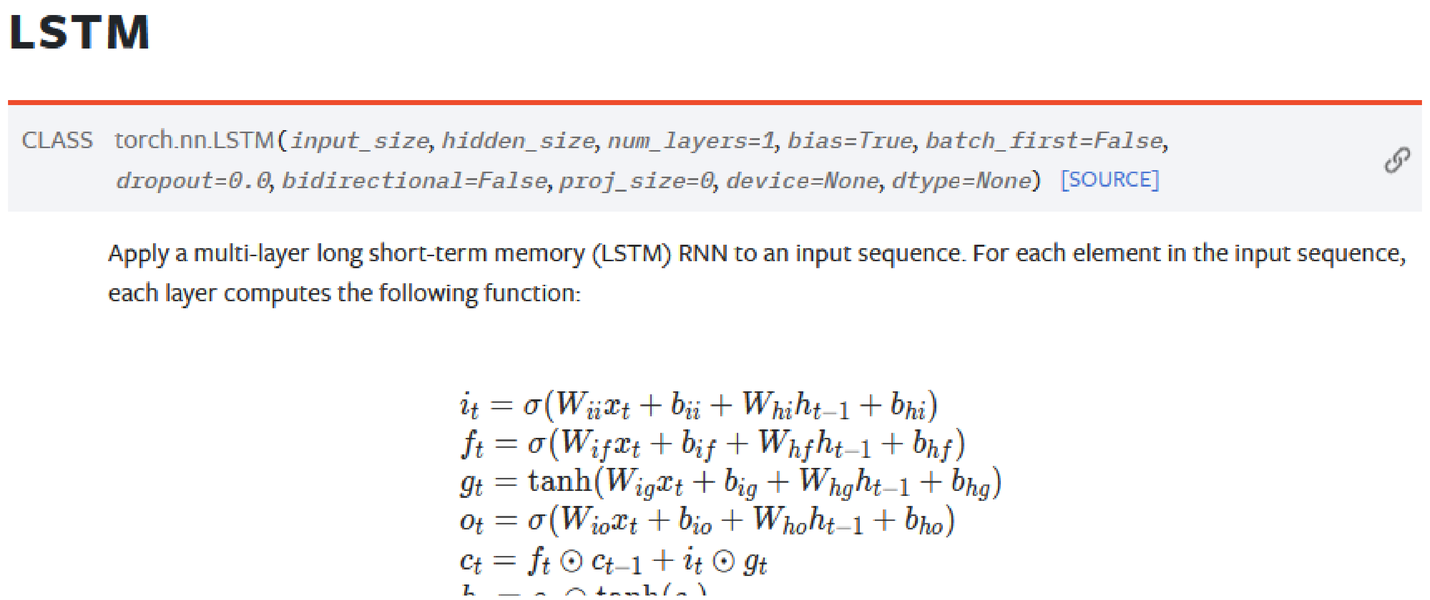
$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{\mathrm{xr}} + \mathbf{H}_{t-1}\mathbf{W}_{\mathrm{hr}} + \mathbf{b}_{\mathrm{r}}),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{\mathrm{xz}} + \mathbf{H}_{t-1}\mathbf{W}_{\mathrm{hz}} + \mathbf{b}_{\mathrm{z}}),$$

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{\mathrm{xh}} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{\mathrm{hh}} + \mathbf{b}_{\mathrm{h}}),$$

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

Pytorch LSTM and LSTM cell

In Pytorch you can build an RNN completely from LSTM cells
or just use one LSTM cell in your architecture.

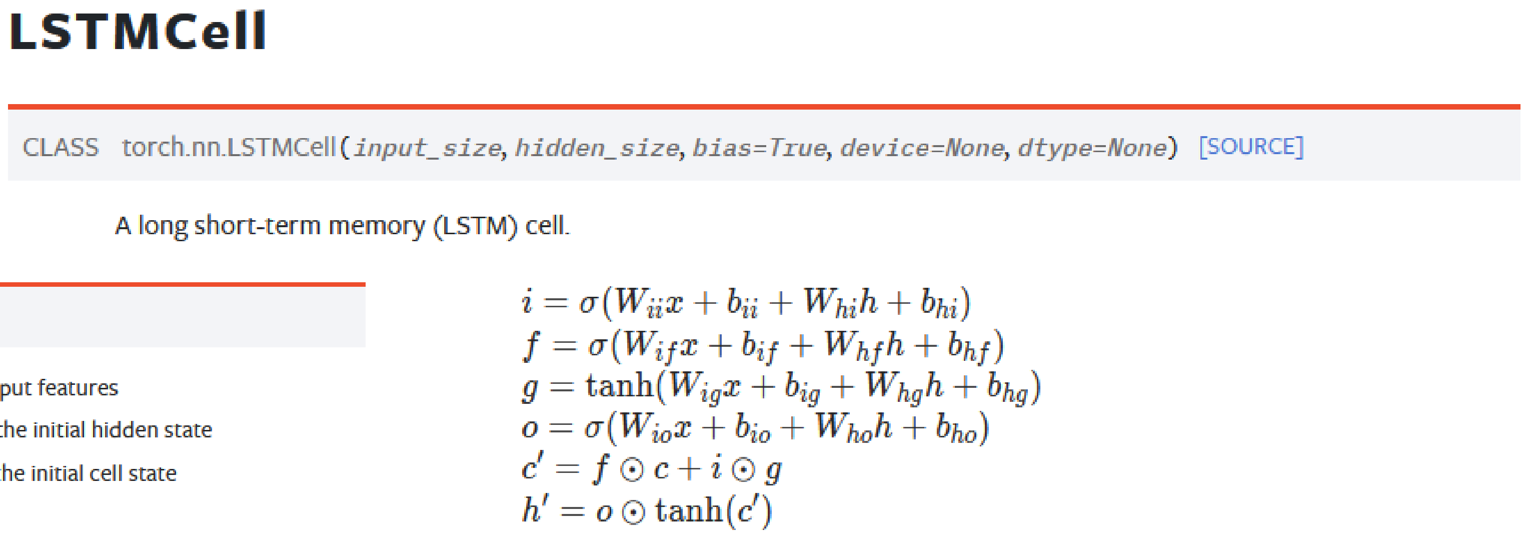


LSTM

CLASS torch.nn.LSTM(*input_size, hidden_size, num_layers=1, bias=True, batch_first=False, dropout=0.0, bidirectional=False, proj_size=0, device=None, dtype=None*) [SOURCE]

Apply a multi-layer long short-term memory (LSTM) RNN to an input sequence. For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t
\end{aligned}
$$

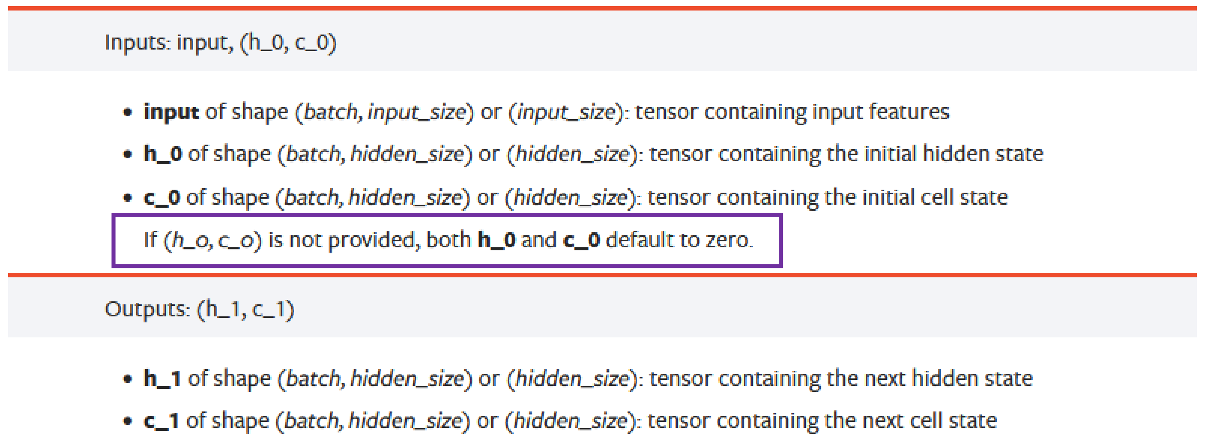


LSTMCell

CLASS torch.nn.LSTMCell(*input_size, hidden_size, bias=True, device=None, dtype=None*) [SOURCE]

A long short-term memory (LSTM) cell.

$$
\begin{aligned}
i &= \sigma(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
f &= \sigma(W_{if}x + b_{if} + W_{hf}h + b_{hf}) \\
g &= \tanh(W_{ig}x + b_{ig} + W_{hg}h + b_{hg}) \\
o &= \sigma(W_{io}x + b_{io} + W_{ho}h + b_{ho}) \\
c' &= f \odot c + i \odot g \\
h' &= o \odot \tanh(c')
\end{aligned}
$$



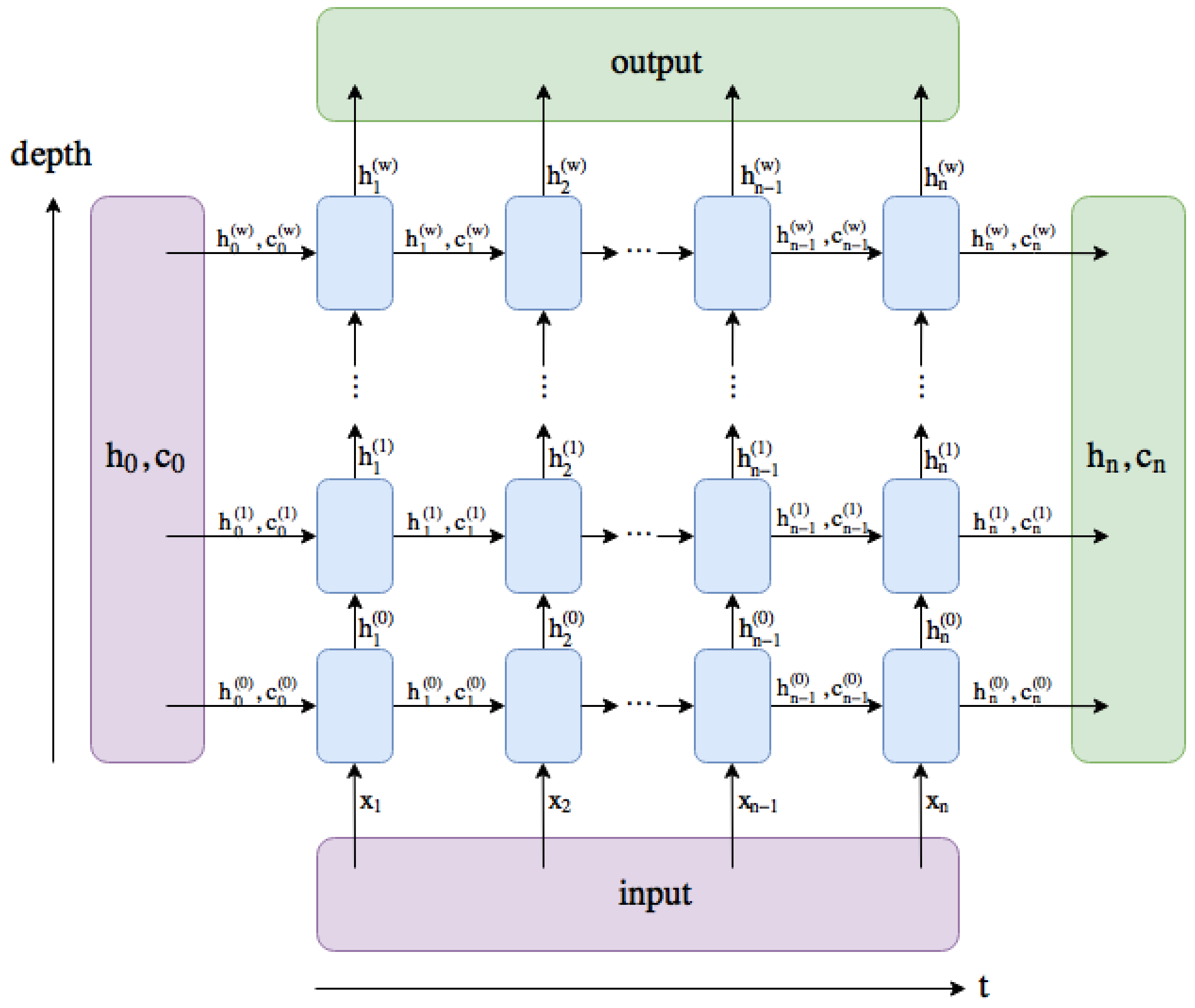
Inputs: input, (h_0, c_0)

- **input** of shape (*batch, input_size*) or (*input_size*): tensor containing input features
- **h_0** of shape (*batch, hidden_size*) or (*hidden_size*): tensor containing the initial hidden state
- **c_0** of shape (*batch, hidden_size*) or (*hidden_size*): tensor containing the initial cell state

If (*h_0, c_0*) is not provided, both **h_0** and **c_0** default to zero.

Outputs: (h_1, c_1)

- **h_1** of shape (*batch, hidden_size*) or (*hidden_size*): tensor containing the next hidden state
- **c_1** of shape (*batch, hidden_size*) or (*hidden_size*): tensor containing the next cell state



Ref: https://stackoverflow.com/questions/48302810/whats-the-difference-between-hidden-and-output-in-pytorch-lstm

Text Analysis with RNNs

Initially the main applications of RNNs were text processing.
It’s important to understand text processing when studying RNNs and Transformers.

It’s also very important for you to understand how we convert a sentence to a tensor format. These ideas are very useful and can be applied to different applications.

As an example, we want to build a RNN neural network than can take a sentence as input and classify its sentiment.

This is a great movie → positive
This film is a waste of time → negative

In the next few slides we go over the steps that are need to accomplish this.

Step one: building the vocabulary:

Let say the vocabulary in movie reviews contains these words:

{the, movie, film, a , great, this, is , time, of, waste, ..., <unknown>, <padding> }

We assign an index to each word.

{the→0, movie→1, film→2, a→3 , great→4, this→5, is→6 , time→7, of→8, waste→9, ..., <unknown>→10, <padding>→11 }

Then our example sentences are can be converted to list of indices. We assume that the max sentence length is 10 words

Step two: convert sentences to indices

This is a great movie → [5, 6, 3, 4, 1, 11,11,11,11,11]
This film is a waste of time → [5, 2, 6,3,9,8,7,11,11, 11]
This movie rocks → [5, 1, 10, 11,11,11,11,11,11,11]

Step 3: indices to one-hot encoding

$$\text{This is a great movie} \rightarrow \begin{bmatrix} 5 \\ 6 \\ 3 \\ 4 \\ 1 \\ 11 \\ 11 \\ 11 \\ 11 \\ 11 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}_{10*12}$$

number of rows of the one-hot matrix is the maximum sentence length.
number of column of the one-hot matrix is the maximum vocabulary size.

Step 4: convert the sentences to float numbers via Embedding Matrix

Embedding matrix is a linear weight matrix that is initialized randomly and it can also be learned and is multiplied by the one-hot encoding of the words.

For example if the embedding size if 5, the embedding matrix has dimension of “size_of_vocab”.

$$\underset{10*12}{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{12*5}{\begin{bmatrix} 0.1 & 0.5 & 0.8 & 0.2 & 0.3 \\ 0.3 & 0.5 & 0.4 & 0.1 & 0.2 \\ 0.1 & 0.2 & 0.8 & 0.2 & 0.1 \\ 0.9 & 0.2 & 0.1 & 0.05 & 0.3 \\ 0.8 & 0.5 & 0.2 & 0.1 & 1 \\ 0.5 & 0.3 & 0.6 & 0.1 & 0.8 \\ 0.4 & 0.1 & 0.5 & 0.7 & 0.1 \\ 0.4 & 0.2 & 0.6 & 0.2 & 0.4 \\ 0.5 & 0.7 & 0.4 & 0.2 & 0.5 \\ 0.3 & 0.6 & 0.5 & 0.1 & 0.4 \\ 0.5 & 0.8 & 0.3 & 0.7 & 0.3 \\ 0.2 & 0.9 & 0.1 & 0.4 & 0.3 \end{bmatrix}} = \underset{10*5}{\begin{bmatrix} 0.5 & 0.3 & 0.6 & 0.1 & 0.8 \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix}} \rightarrow \text{Embedded sentence}$$

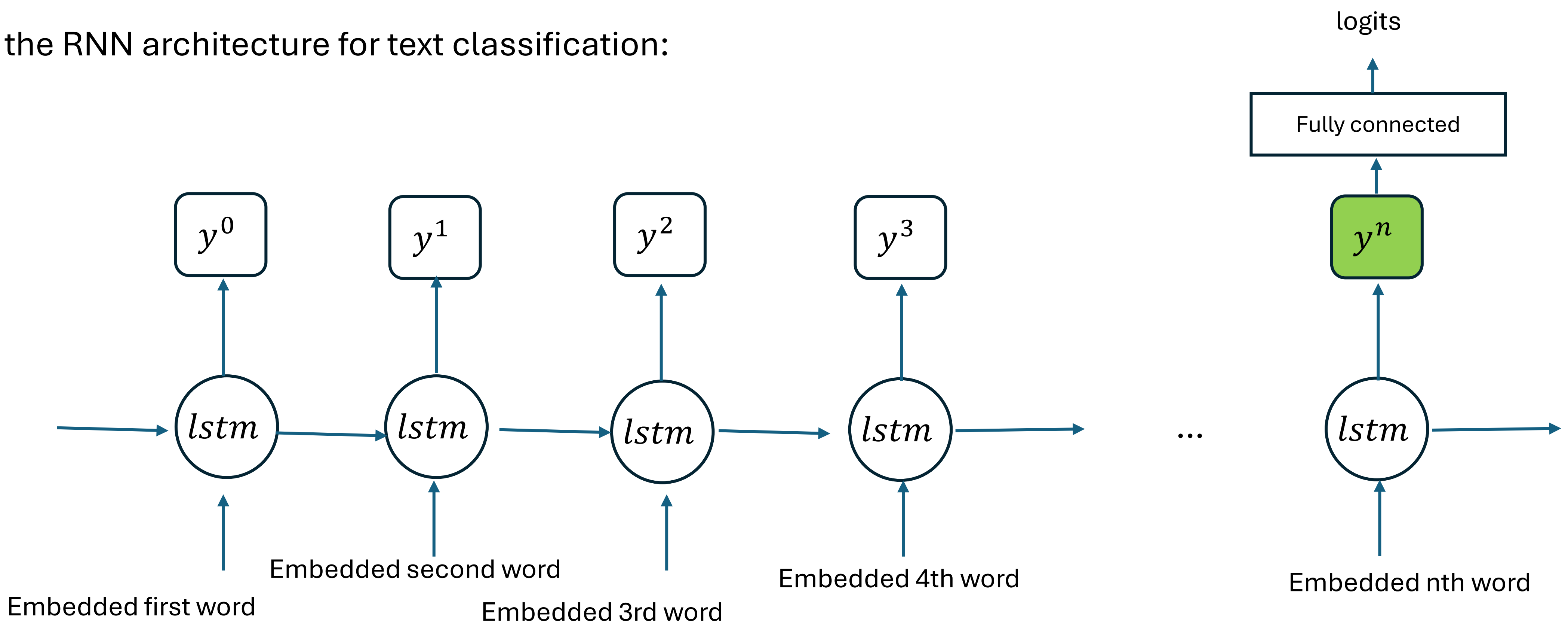
Some models such as Word2vec and Glove are trained to generate word embedding.

As a result, the entire sentence have the dimension of max_sentence_length * embedding_size.

And all the sentences in the batch have the same size as above.

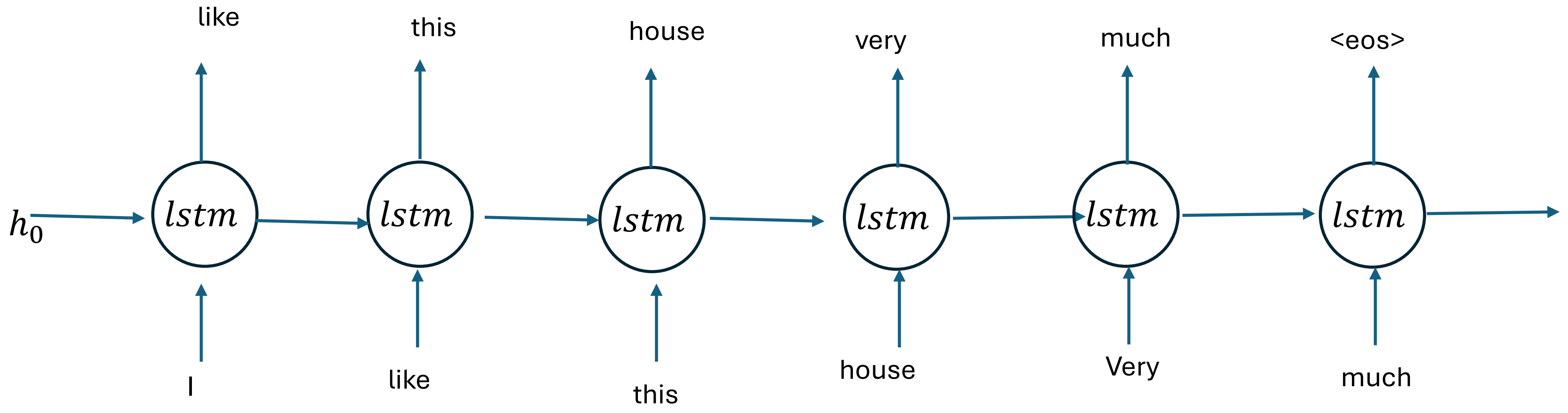
Therefore, the sentences are now like an image.
We can batch many sentences together and perform training.

This is how the RNN architecture for text classification:

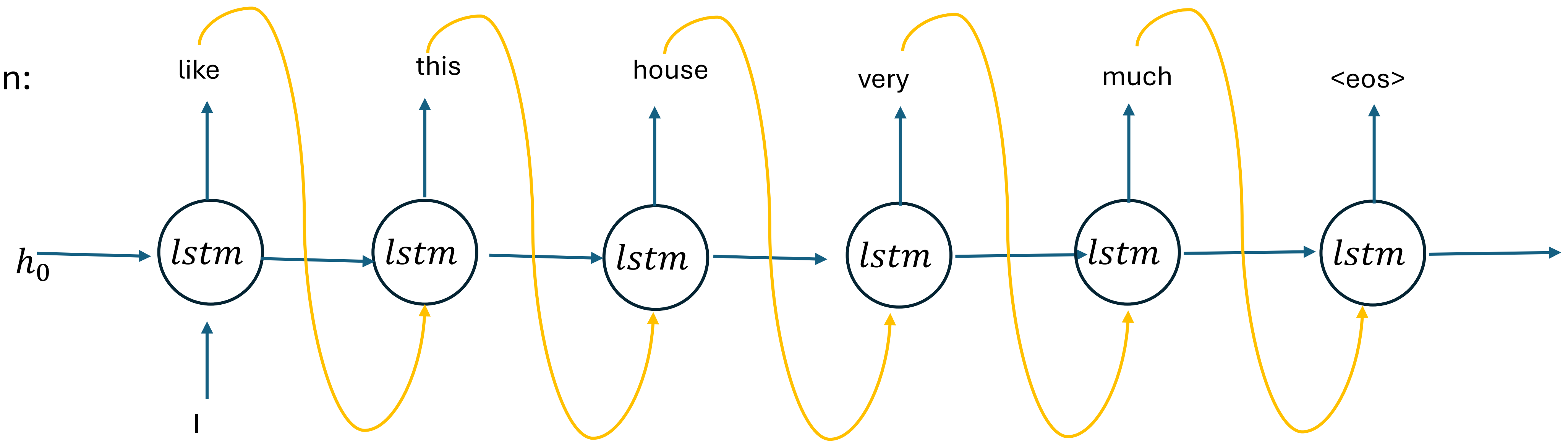


Sequence generating with RNNs

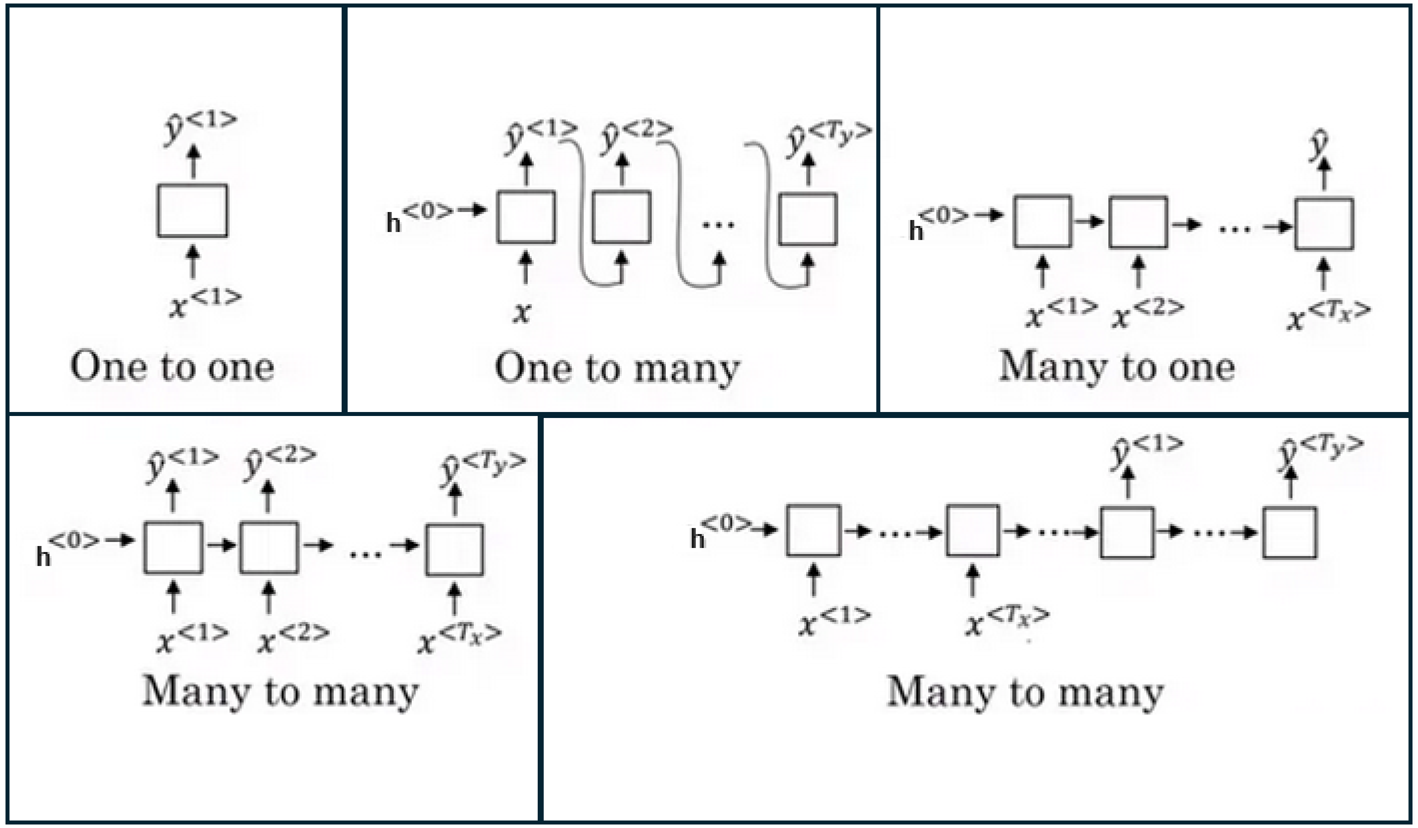
During training:

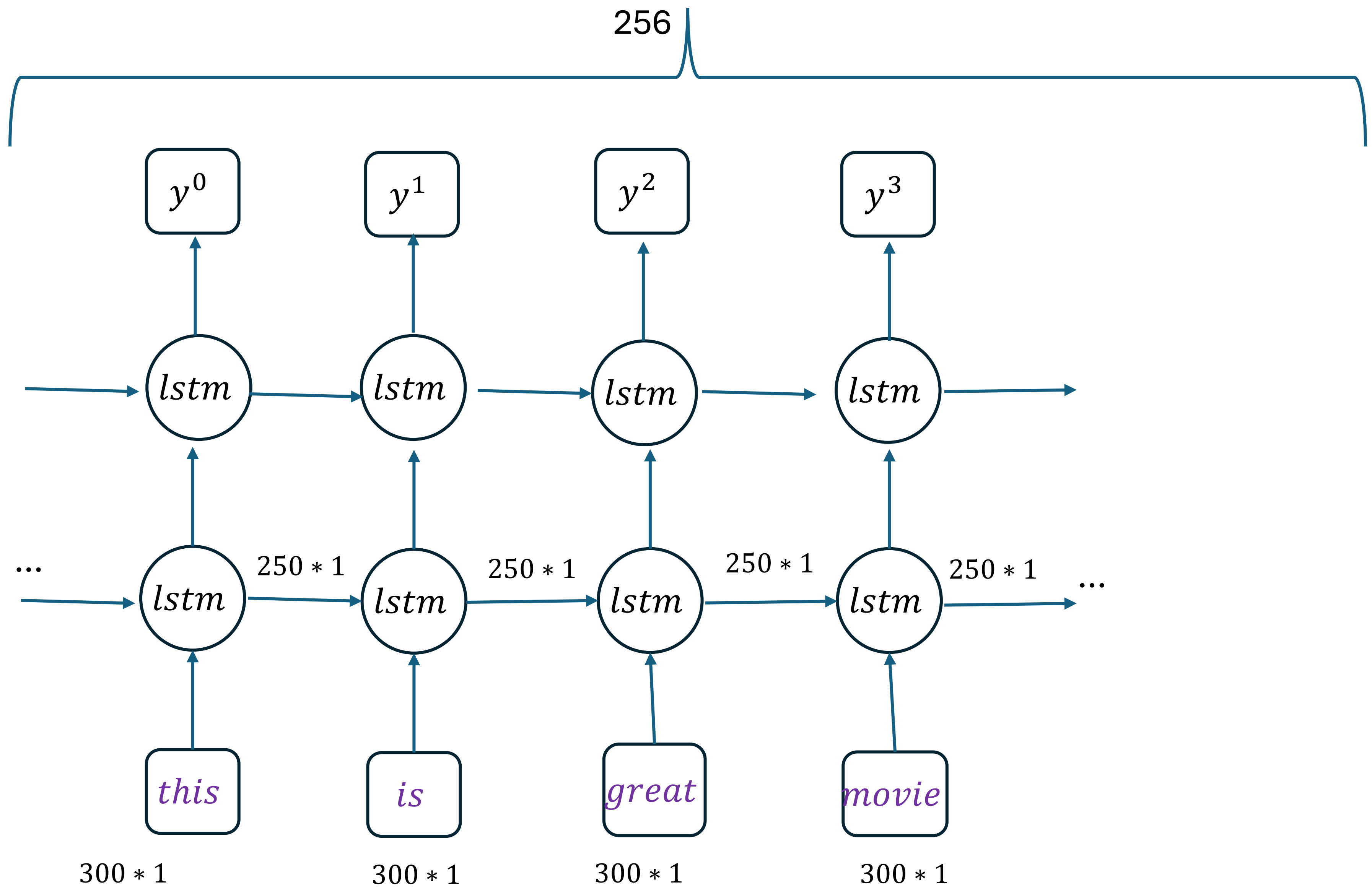


During evaluation:



Different RNN architecture


$\hat{y}^{<1>}$
$x^{<1>}$
One to one
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$
$h^{<0>}$
x
One to many
$\hat{y}$
$h^{<0>}$
$x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Many to one
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$
$h^{<0>}$
$x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Many to many
$\hat{y}^{<1>}$ $\hat{y}^{<T_y>}$
$h^{<0>}$
$x^{<1>}$ $x^{<T_x>}$
Many to many


256
y^0
y^1
y^2
y^3
$lstm$
$lstm$
$lstm$
$lstm$
...
$lstm$
$250 * 1$
$lstm$
$250 * 1$
$lstm$
$250 * 1$
$lstm$
$250 * 1$
...
$this$
is
$great$
$movie$
$300 * 1$
$300 * 1$
$300 * 1$
$300 * 1$