Deep Learning for Vision & Language

Natural Language Processing III: Recurrent Neural Networks

RICE UNIVERSITY

Second Assignment

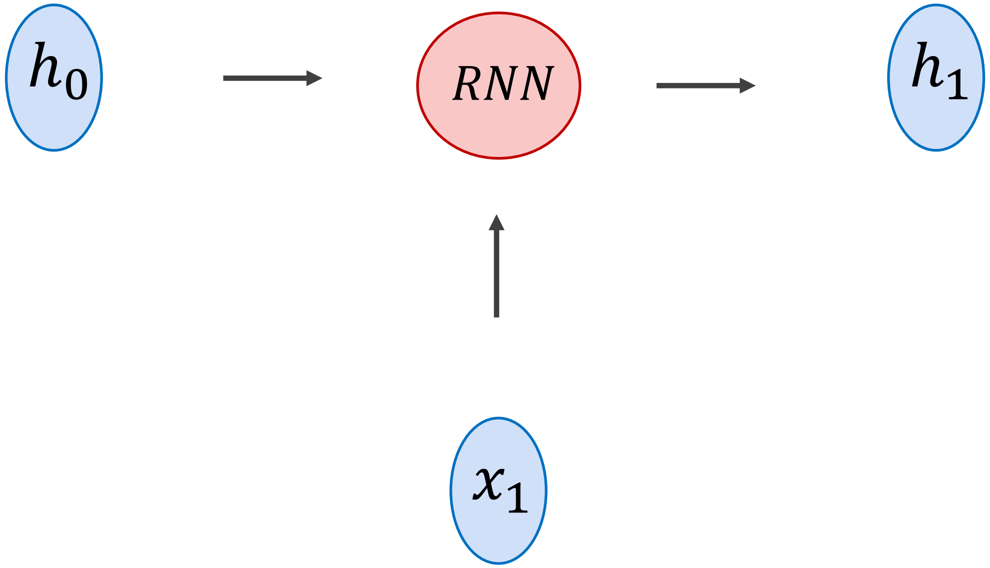
- Due Next Monday and third and final assignment to follow soon.

- Submit your project proposal – think about the amount of work it would take to a) Create an assignment 4, b) Solve assignment 4. Often in research and entrepreneurship asking a good question/finding the right problem is more important than giving a great answer/solution.

Recurrent Neural Networks

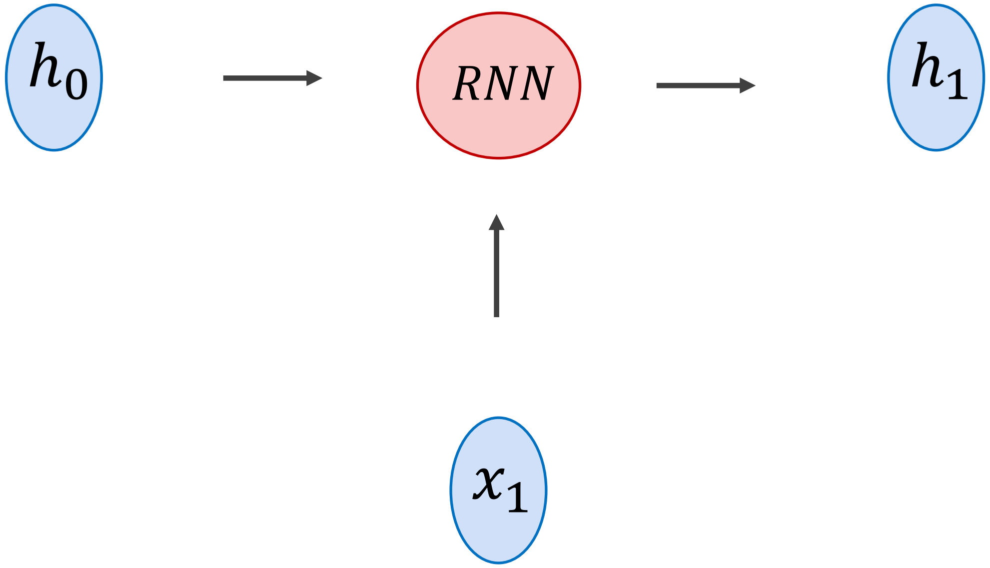
- These are models for handling sequences of things.
- Each input is not a vector but a sequence of input vectors.
- e.g. Each input can be a “word embedding” or any “word” representation – we will use in our first examples one-hot encoded tokens but in practice continuous dense word embeddings are used.

Recurrent Neural Network Cell

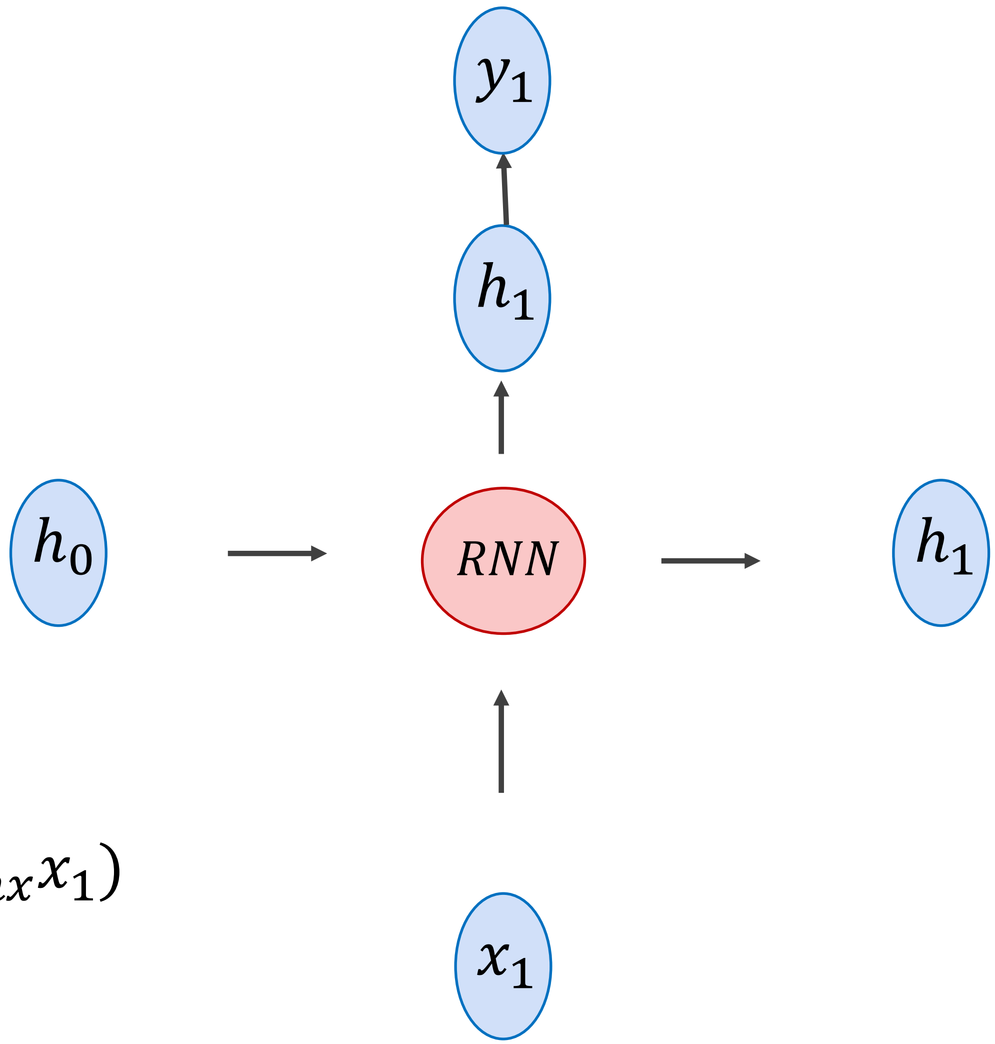


Recurrent Neural Network Cell

$$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1)$$



Recurrent Neural Network Cell



$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1)$

$y_1 = \mathrm{softmax}(W_{hy}h_1)$

Recurrent Neural Network Cell

$$y_1 = [0.1, 0.05, 0.05, 0.1, 0.7]$$

$$h_1 = [0.1 \quad 0.2\ 0 \ -0.3 \ -0.1\ ]$$

$$h_0 = [0\ 0\ 0\ 0\ 0\ 0\ 0\ ]$$

RNN

$$h_1 = [0.1 \quad 0.2\ 0 \ -0.3 \ -0.1\ ]$$

$$x_1 = \ [0\ 0\ 1\ 0\ 0]$$

$$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1)$$

$$y_1 = \text{softmax}(W_{hy}h_1)$$

Recurrent Neural Network Cell

$y_1 = [0.1, 0.05, 0.05, 0.1, 0.7]$ → **e (0.7)**

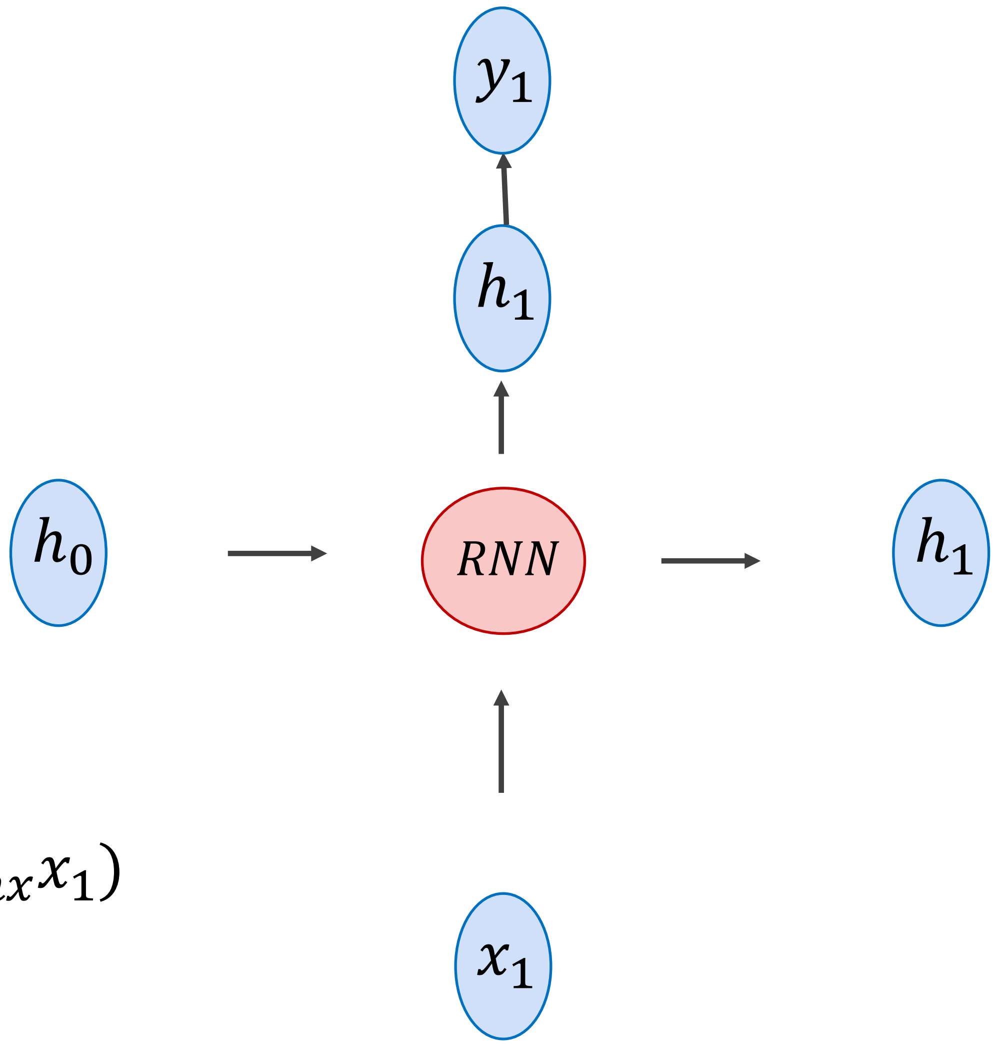
$h_1 = [0.1 \quad 0.2 \; 0 \; -0.3 \; -0.1\,]$

$h_0 = [0\;0\;0\;0\;0\;0\;0\,]$ → RNN → $h_1 = [0.1 \quad 0.2 \; 0 \; -0.3 \; -0.1\,]$

$x_1 = [0\;0\;1\;0\;0]$

a b **c** d e

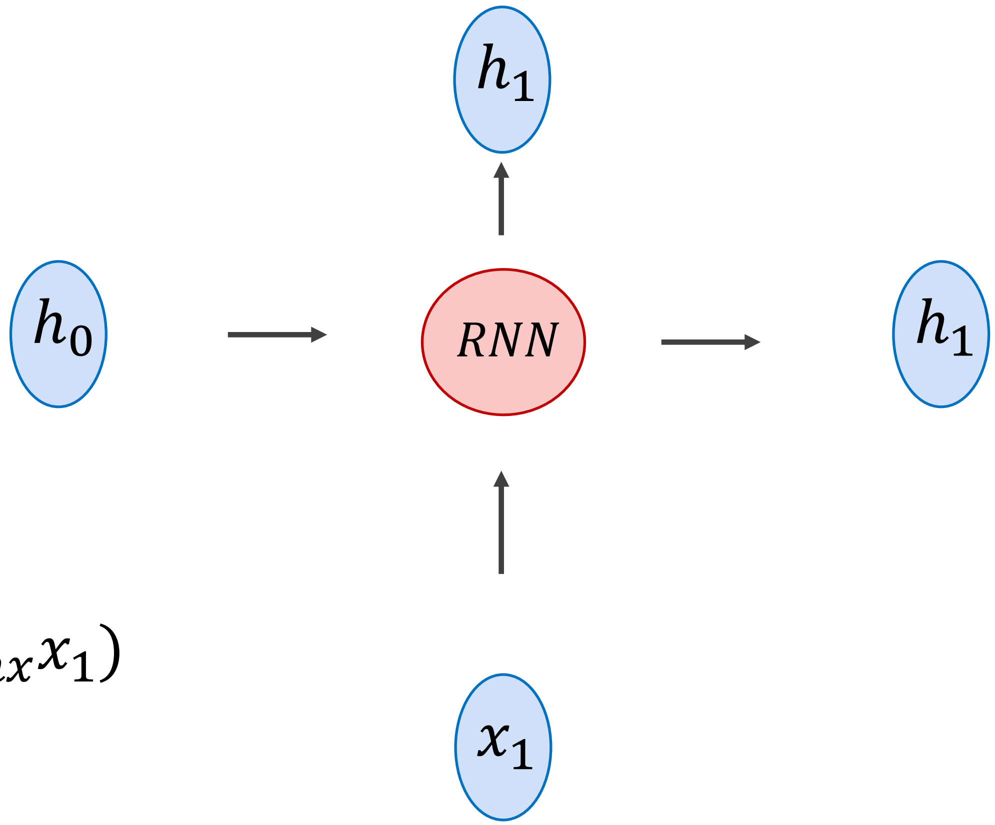
Recurrent Neural Network Cell



$$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1)$$

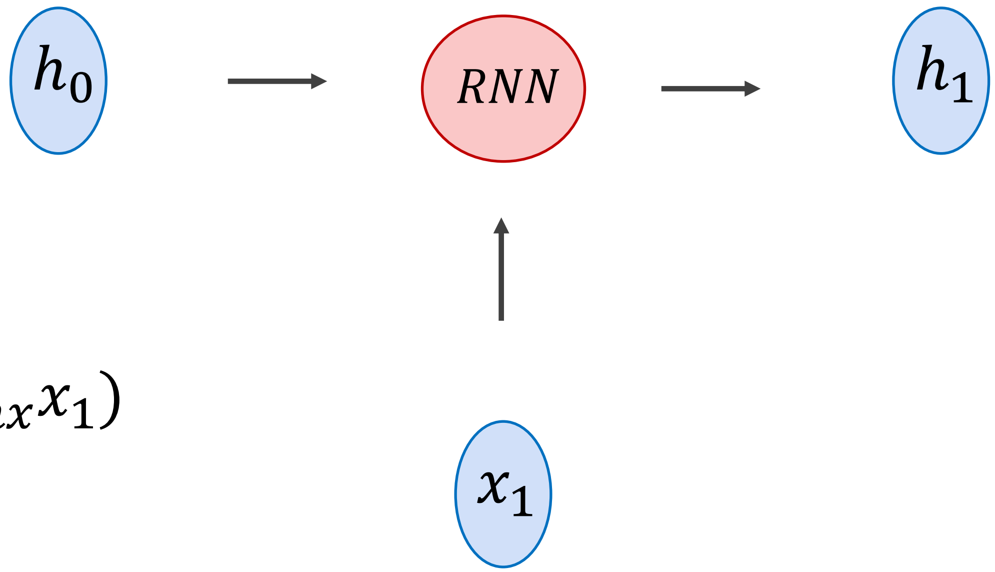
$$y_1 = \text{softmax}(W_{hy}h_1)$$

Recurrent Neural Network Cell



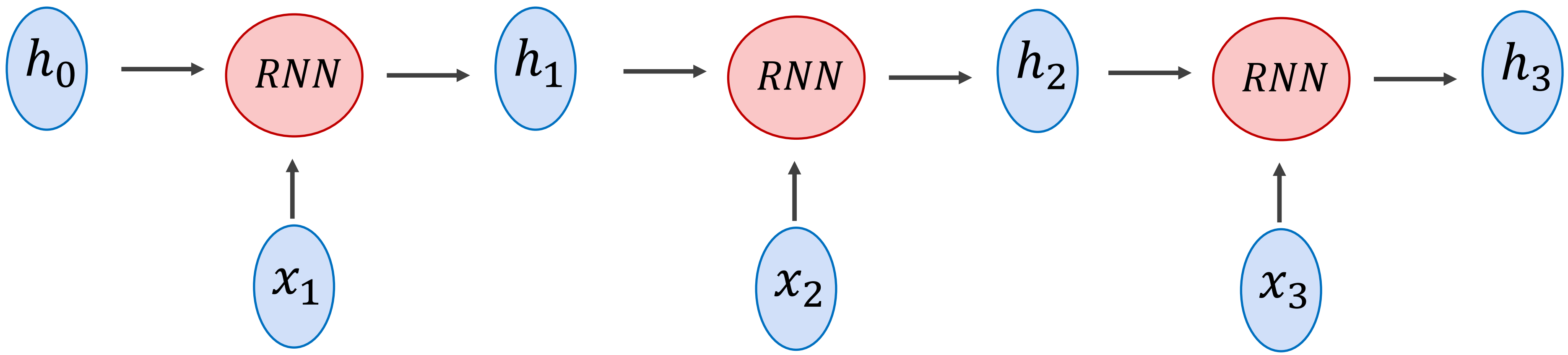
$$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1)$$

Recurrent Neural Network Cell

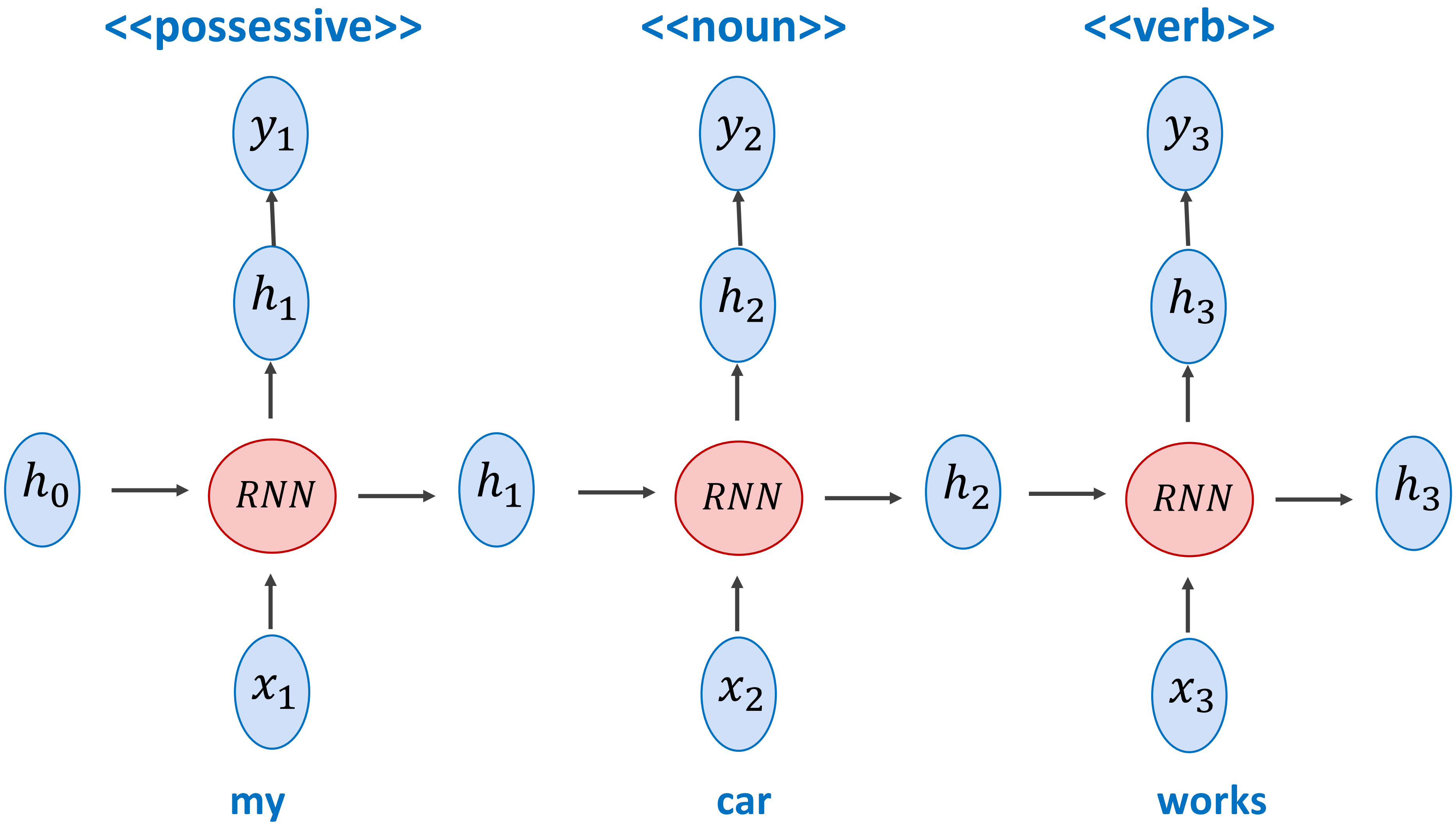


$$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1)$$

(Unrolled) Recurrent Neural Network



How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems



How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Training examples don’t need to be the same length!

input	output
my car works	<<possessive>> <<noun>> <<verb>>
my dog ate the assignment	<<possessive>> <<noun>> <<verb>> <<pronoun>> <<noun>>
my mother saved the day	<<possessive>> <<noun>> <<verb>> <<pronoun>> <<noun>>
the smart kid solved the problem	<<pronoun>> <<qualifier>> <<noun>> <<verb>> <<pronoun>> <<noun>>

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Training examples don’t need to be the same length!

input	**output**
L(my car works) = 3	**L (<<possessive>> <<noun>> <<verb>>) = 3**
L(my dog ate the assignment) = 5	**L (<<possessive>> <<noun>> <<verb>> <<pronoun>> <<noun>>) = 5**
L(my mother saved the day) = 5	**L (<<possessive>> <<noun>> <<verb>> <<pronoun>> <<noun>>) = 5**
L(the smart kid solved the problem) = 6	**L (<<pronoun>> <<qualifier>> <<noun>> <<verb>> <<pronoun>> <<noun>>) = 6**

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Training examples don't need to be the same length!

If we assume a vocabulary of a 1000 possible words and 20 possible output tags

input	**output**
T: 1000 x 3	**T: 20 x 3**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 6	**T: 20 x 6**

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Training examples don't need to be the same length!

If we assume a vocabulary of a 1000 possible words and 20 possible output tags

input	output
T: 1000 x 3	T: 20 x 3
T: 1000 x 5	T: 20 x 5
T: 1000 x 5	T: 20 x 5
T: 1000 x 6	T: 20 x 6

How do we create batches if inputs and outputs have different shapes?

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Training examples don't need to be the same length!

If we assume a vocabulary of a 1000 possible words and 20 possible output tags

input	output
T: 1000 x 3	**T: 20 x 3**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 6	**T: 20 x 6**

How do we create batches if inputs and outputs have different shapes?

Solution 1: Forget about batches, just process things one by one.

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Training examples don't need to be the same length!

If we assume a vocabulary of a 1000 possible words and 20 possible output tags

input	output
T: 1000 x 3	T: 20 x 3
T: 1000 x 5	T: 20 x 5
T: 1000 x 5	T: 20 x 5
T: 1000 x 6	T: 20 x 6

How do we create batches if inputs and outputs have different shapes?

Solution 2: Zero padding. We can put the above vectors in T: 4 x 1000 x 6

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Training examples don't need to be the same length!

If we assume a vocabulary of a 1000 possible words and 20 possible output tags

input	**output**
T: 1000 x 3	**T: 20 x 3**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 6	**T: 20 x 6**

How do we create batches if inputs and outputs have different shapes?

Solution 3: Advanced. Dynamic Batching or Auto-batching
https://dynet.readthedocs.io/en/latest/tutorials_notebooks/Autobatching.html

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Solution 4: Pytorch stacking, padding, and sorting combination

pad_sequence

`torch.nn.utils.rnn.pad_sequence(sequences, batch_first=False, padding_value=0)` [SOURCE]

Pad a list of variable length Tensors with `padding_value`

`pad_sequence` stacks a list of Tensors along a new dimension, and pads them to equal length. For example, if the input is list of sequences with size `L x *` and if batch_first is False, and `T x B x *` otherwise.

B is batch size. It is equal to the number of elements in `sequences`. *T* is length of the longest sequence. *L* is length of the sequence. * is any number of trailing dimensions, including none.

Example

```
>>> from torch.nn.utils.rnn import pad_sequence
>>> a = torch.ones(25, 300)
>>> b = torch.ones(22, 300)
>>> c = torch.ones(15, 300)
>>> pad_sequence([a, b, c]).size()
torch.Size([25, 3, 300])
```

• NOTE

This function returns a Tensor of size `T x B x *` or `B x T x *` where *T* is the length of the longest sequence. This function assumes trailing dimensions and type of all the Tensors in sequences are same.

Parameters

- **sequences** (*list*[*Tensor*]) – list of variable length sequences.
- **batch_first** (*bool, optional*) – output will be in `B x T x *` if True, or in `T x B x *` otherwise
- **padding_value** (*python:float, optional*) – value for padded elements. Default: 0.

Returns

Tensor of size `T x B x *` if `batch_first` is `False`. Tensor of size `B x T x *` otherwise

How can it be used? – e.g. Tagging a Text Sequence
One-to-one Sequence Mapping Problems

Solution 4: Pytorch stacking, padding, and sorting combination

pack_sequence

`torch.nn.utils.rnn.pack_sequence(sequences, enforce_sorted=True)` [SOURCE]

Packs a list of variable length Tensors

`sequences` should be a list of Tensors of size `L x *`, where *L* is the length of a sequence and * is any number of trailing dimensions, including zero.

For unsorted sequences, use *enforce_sorted = False*. If `enforce_sorted` is `True`, the sequences should be sorted in the order of decreasing length. `enforce_sorted = True` is only necessary for ONNX export.

Example

```
>>> from torch.nn.utils.rnn import pack_sequence
>>> a = torch.tensor([1,2,3])
>>> b = torch.tensor([4,5])
>>> c = torch.tensor([6])
>>> pack_sequence([a, b, c])
PackedSequence(data=tensor([ 1,  4,  6,  2,  5,  3]), batch_sizes=tensor([ 3,  2,  1]))
```

Parameters

- **sequences** (*list*[*Tensor*]) – A list of sequences of decreasing length.
- **enforce_sorted** (*bool, optional*) – if `True`, checks that the input contains sequences sorted by length in a decreasing order. If `False`, this condition is not checked. Default: `True`.

Returns

a `PackedSequence` object

Pytorch RNN

RNN

CLASS `torch.nn.RNN(*args, **kwargs)` [SOURCE]

Applies a multi-layer Elman RNN with $tanh$ or $ReLU$ non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(W_{ih} x_t + b_{ih} + W_{hh} h_{(t-1)} + b_{hh})$$

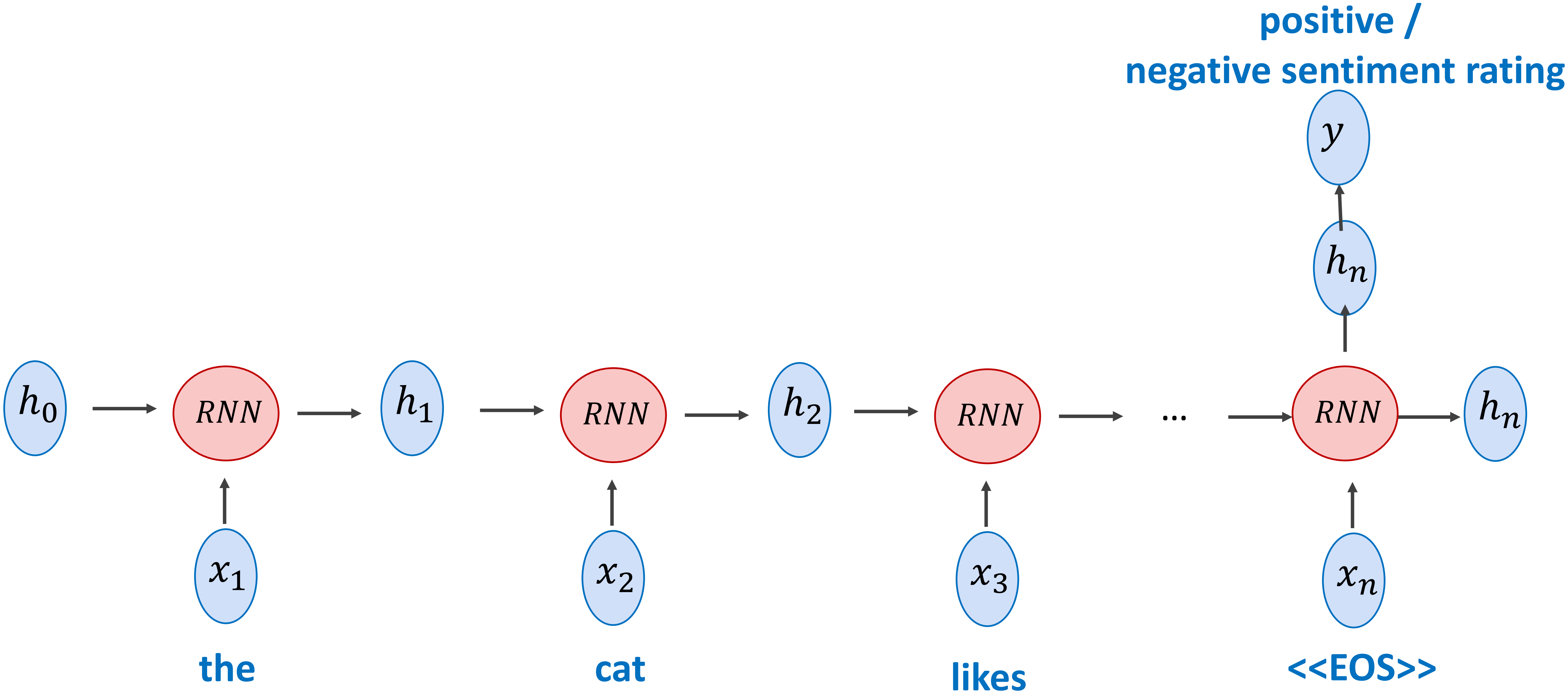
where h_t is the hidden state at time *t*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then *ReLU* is used instead of *tanh*.

Inputs: input, h_0

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.

How can it be used? – e.g. Scoring the Sentiment of a Text Sequence

Many-to-one Sequence to score problems



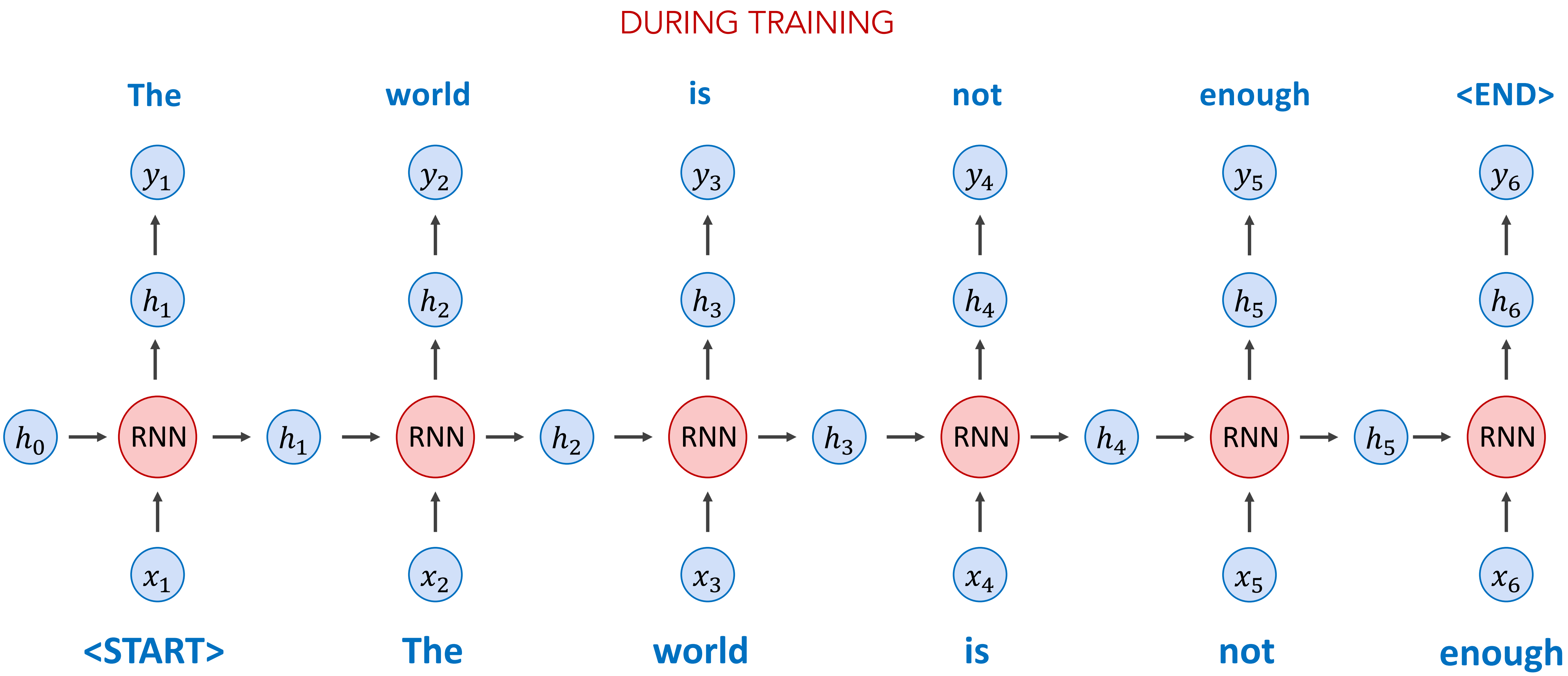
How can it be used? – e.g. Sentiment Scoring
Many to one Mapping Problems

Input training examples don't need to be the same length!
In this case outputs can be.

input	output
this restaurant has good food	Positive
this restaurant is bad	Negative
this restaurant is the worst	Negative
this restaurant is well recommended	Positive

How can it be used? – e.g. Text Generation

Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test



How can it be used? – e.g. Text Generation
Auto-regressive Models

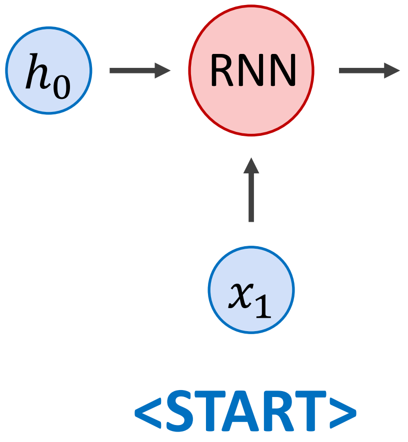
Input training examples don't need to be the same length!
In this case outputs can be.

input	**output**
<START> this restaurant has good food	**this restaurant has good food <END>**
<START> this restaurant is bad	**this restaurant is bad <END>**
<START> this restaurant is the worst	**this restaurant is the worst <END>**
<START> this restaurant is well recommended	**this restaurant is well recommended <END>**

How can it be used? – e.g. Text Generation

Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test

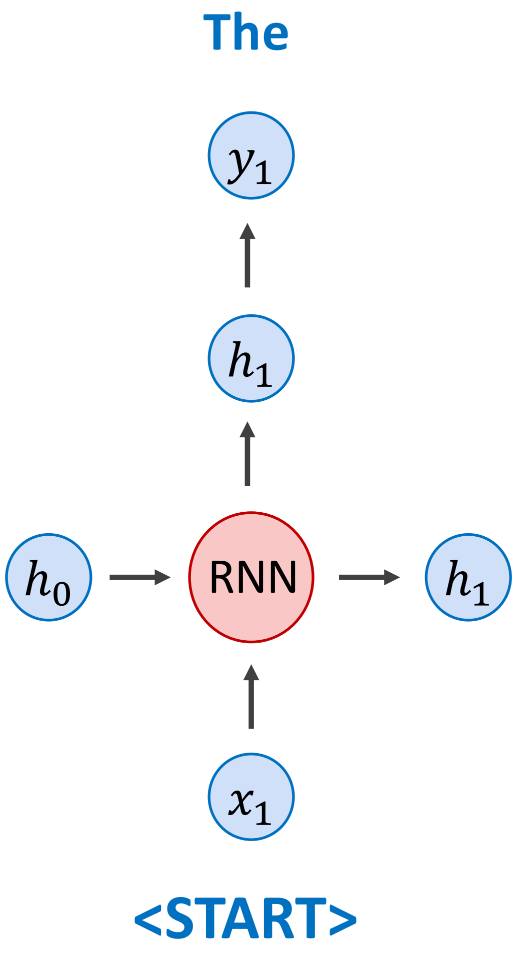
DURING TESTING



How can it be used? – e.g. Text Generation

Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test

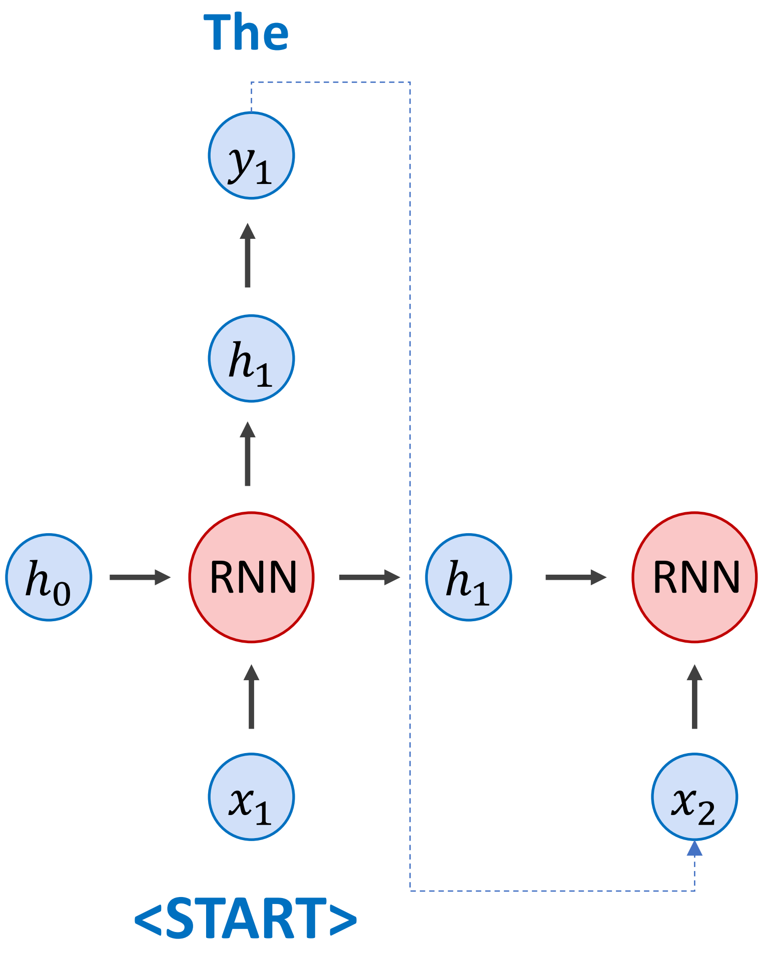
DURING TESTING



How can it be used? – e.g. Text Generation

Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test

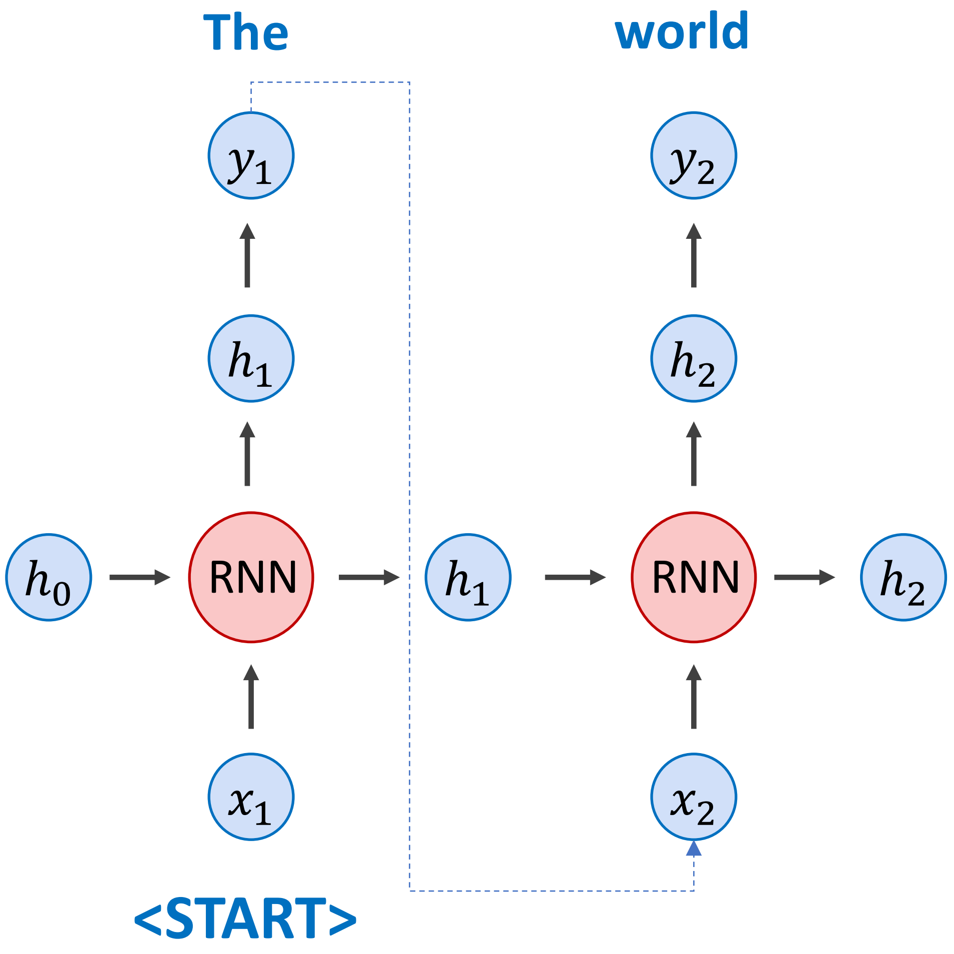
DURING TESTING



How can it be used? – e.g. Text Generation

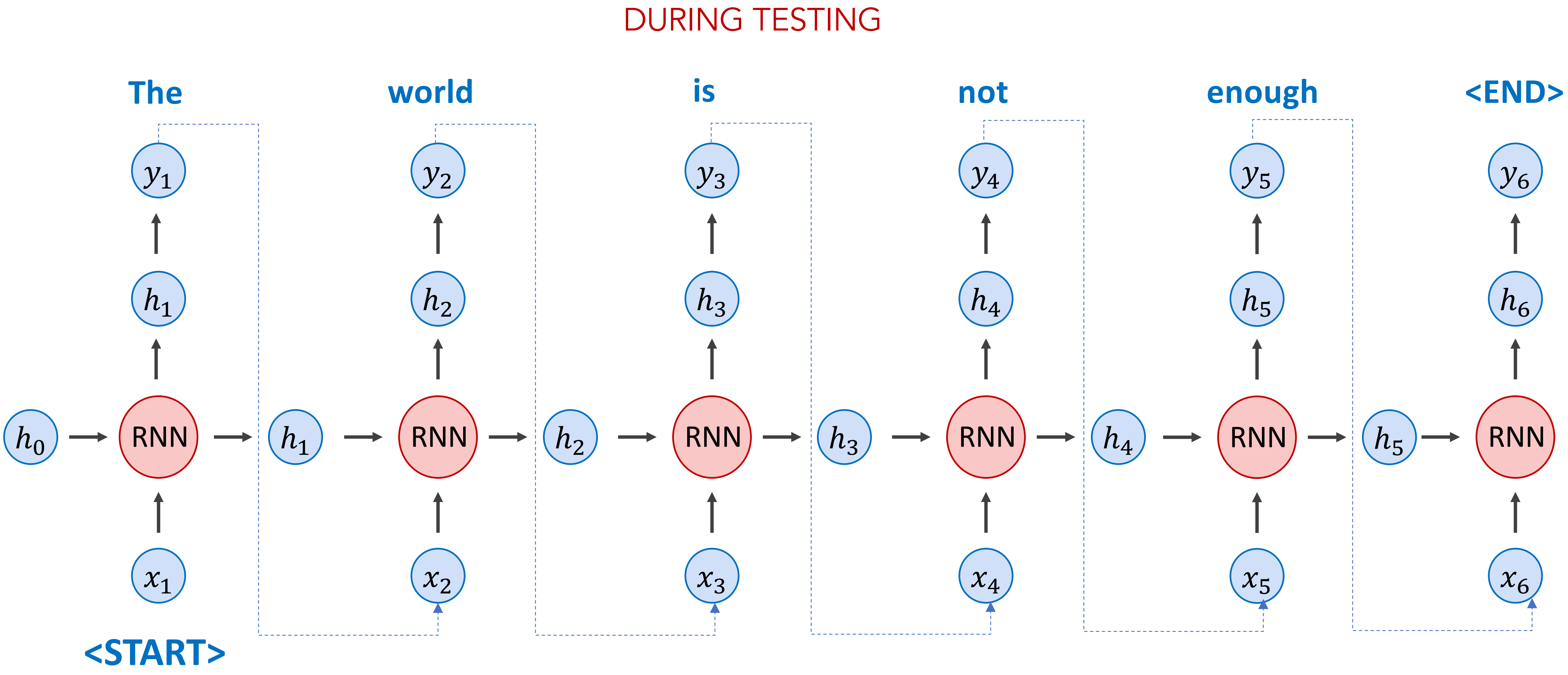
Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test

DURING TESTING

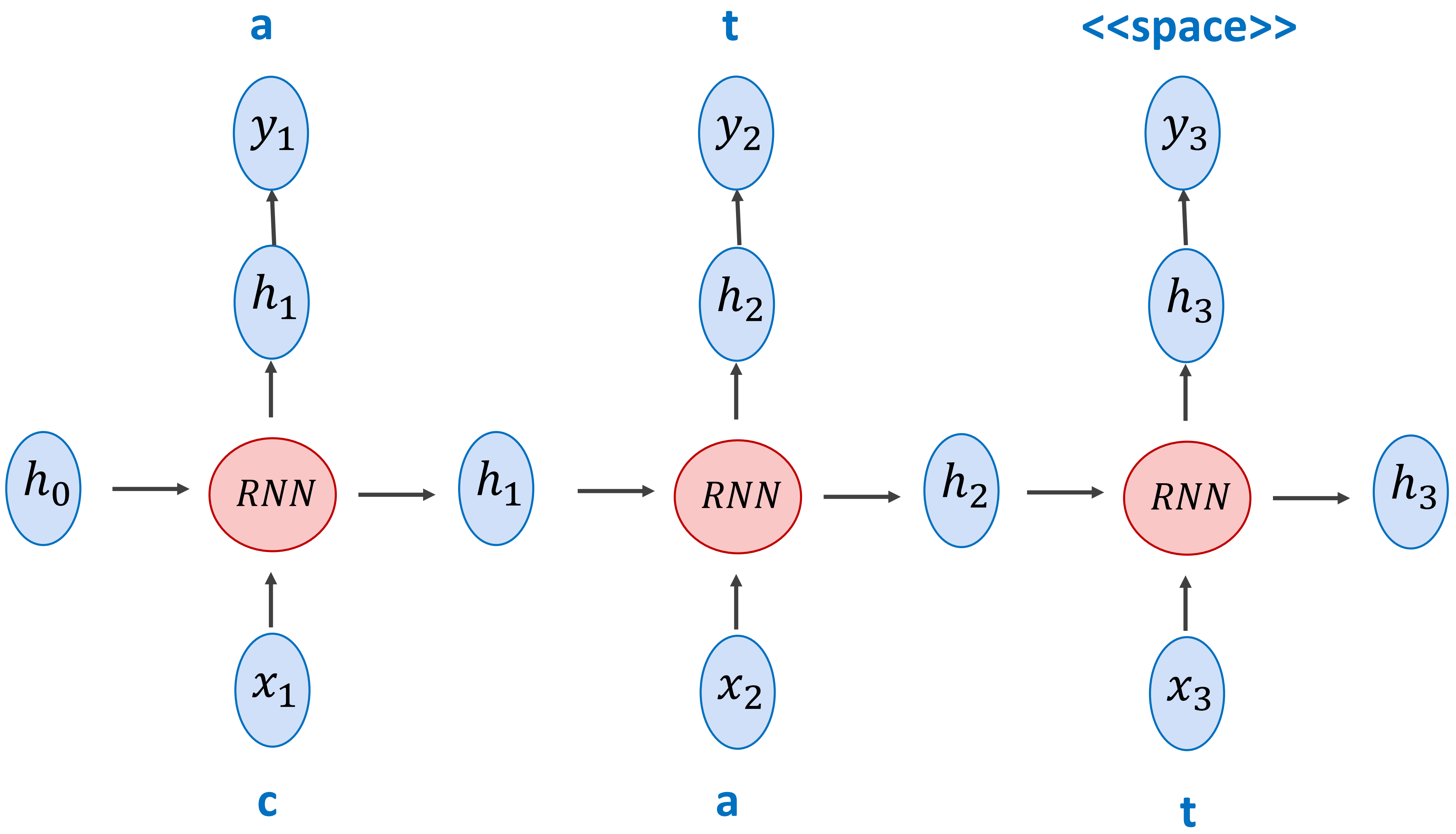


How can it be used? – e.g. Text Generation

Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test



Character-level Models



Generating Sequences With Recurrent Neural Networks

Alex Graves
Department of Computer Science
University of Toronto
graves@cs.toronto.edu

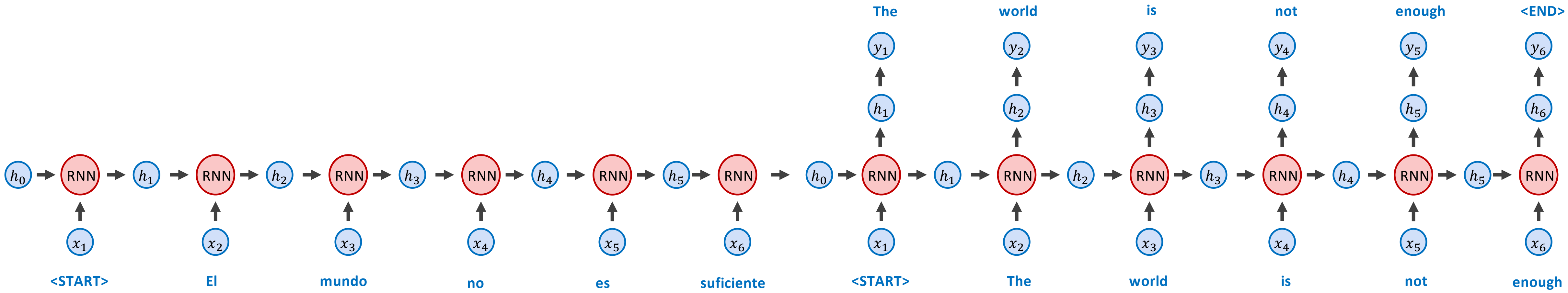
Abstract

This paper shows how Long Short-term Memory recurrent neural networks can be used to generate complex sequences with long-range structure, simply by predicting one data point at a time. The approach is demonstrated for text (where the data are discrete) and online handwriting (where the data are real-valued). It is then extended to handwriting synthesis by allowing the network to condition its predictions on a text sequence. The resulting system is able to generate highly realistic cursive handwriting in a wide variety of styles.

How can it be used? – e.g. Machine Translation

Sequence to Sequence – Encoding – Decoding – Many to Many mapping

DURING TRAINING



How can it be used? – e.g. Machine Translation
Sequence to Sequence Models

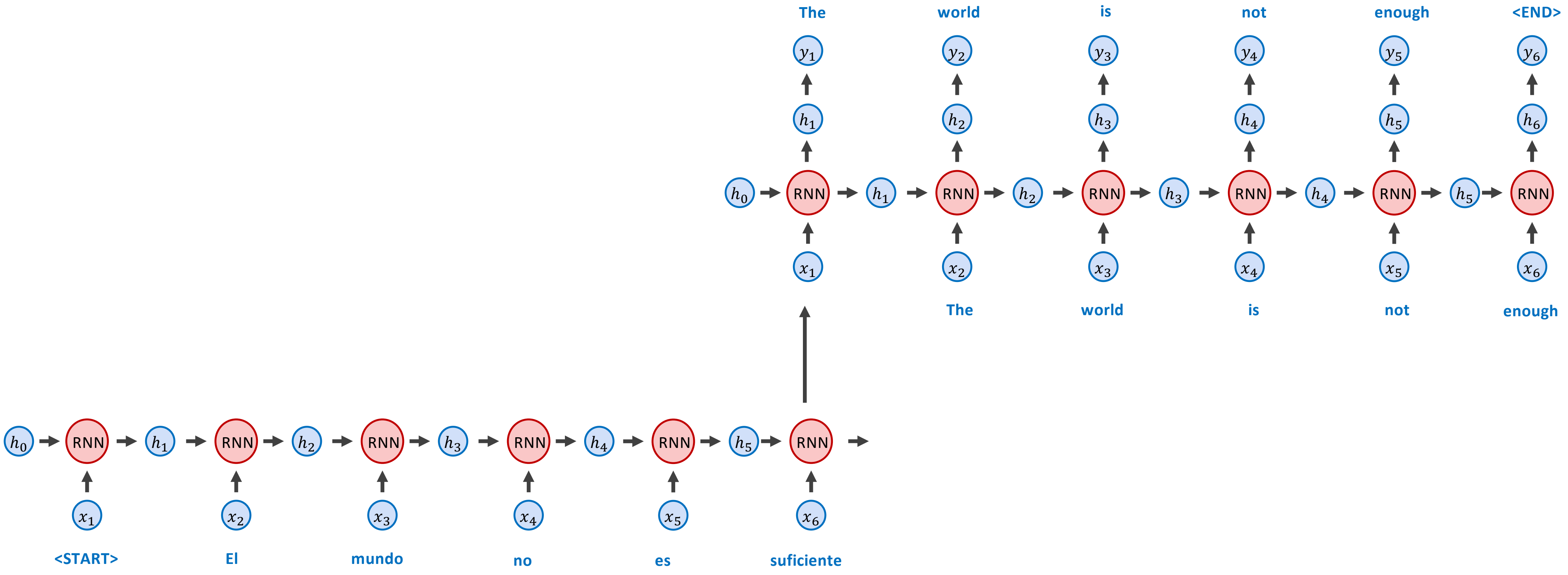
Input training examples don't need to be the same length!
In this case outputs can be.

input	**output**
<START> este restaurante tiene buena comida	this restaurant has good food <END>
<START> this restaurant has good food	
<START> el mundo no es suficiente	the world is not enough <END>
<START> the world is not enough	

How can it be used? – e.g. Machine Translation

Sequence to Sequence – Encoding – Decoding – Many to Many mapping

DURING TRAINING – (Alternative)



Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

Kyunghyun Cho
Bart van Merriënboer **Caglar Gulcehre**
Université de Montréal
firstname.lastname@umontreal.ca

Dzmitry Bahdanau
Jacobs University, Germany
d.bahdanau@jacobs-university.de

Fethi Bougares **Holger Schwenk**
Université du Maine, France
firstname.lastname@lium.univ-lemans.fr

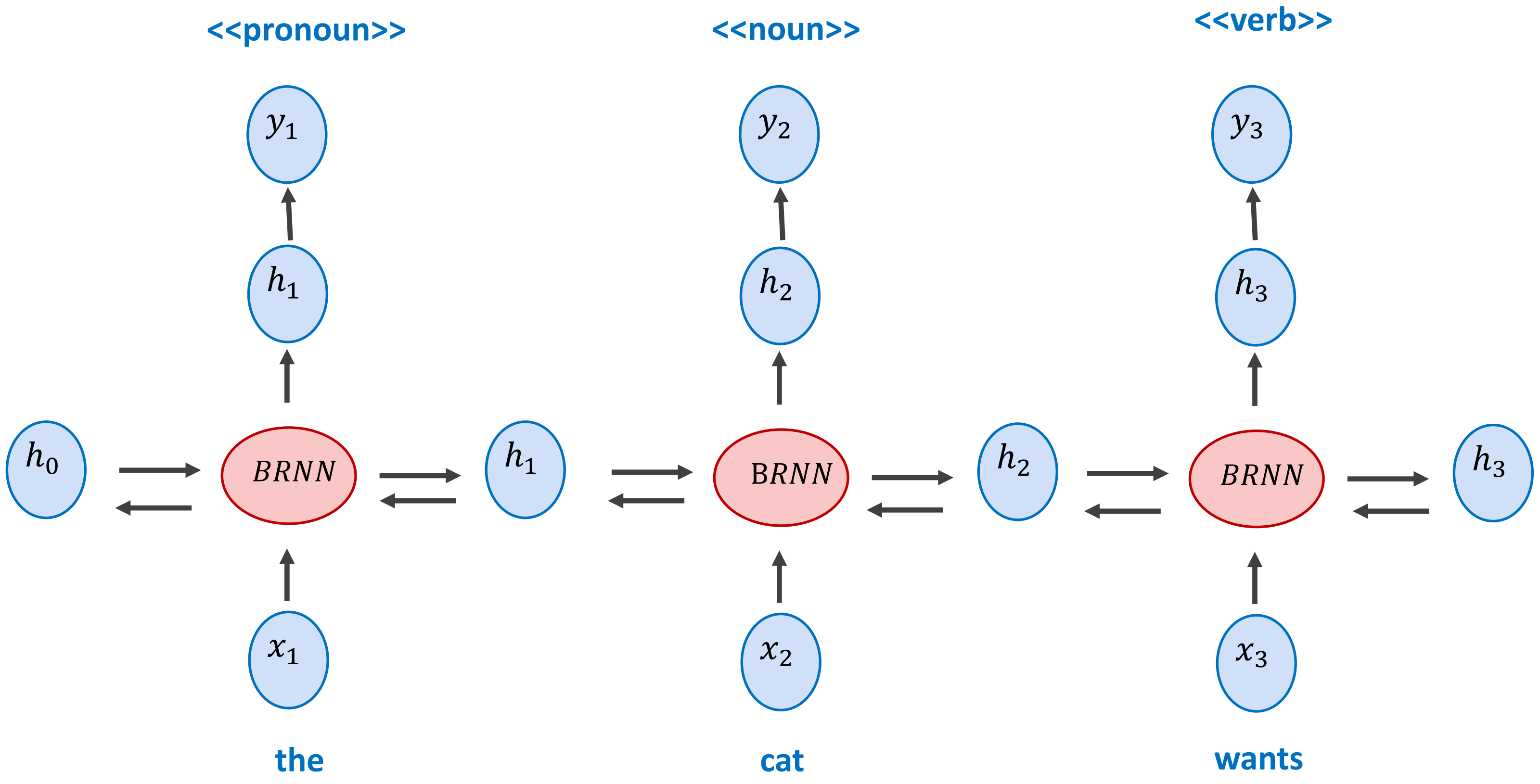
Yoshua Bengio
Université de Montréal, CIFAR Senior Fellow
find.me@on.the.web

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

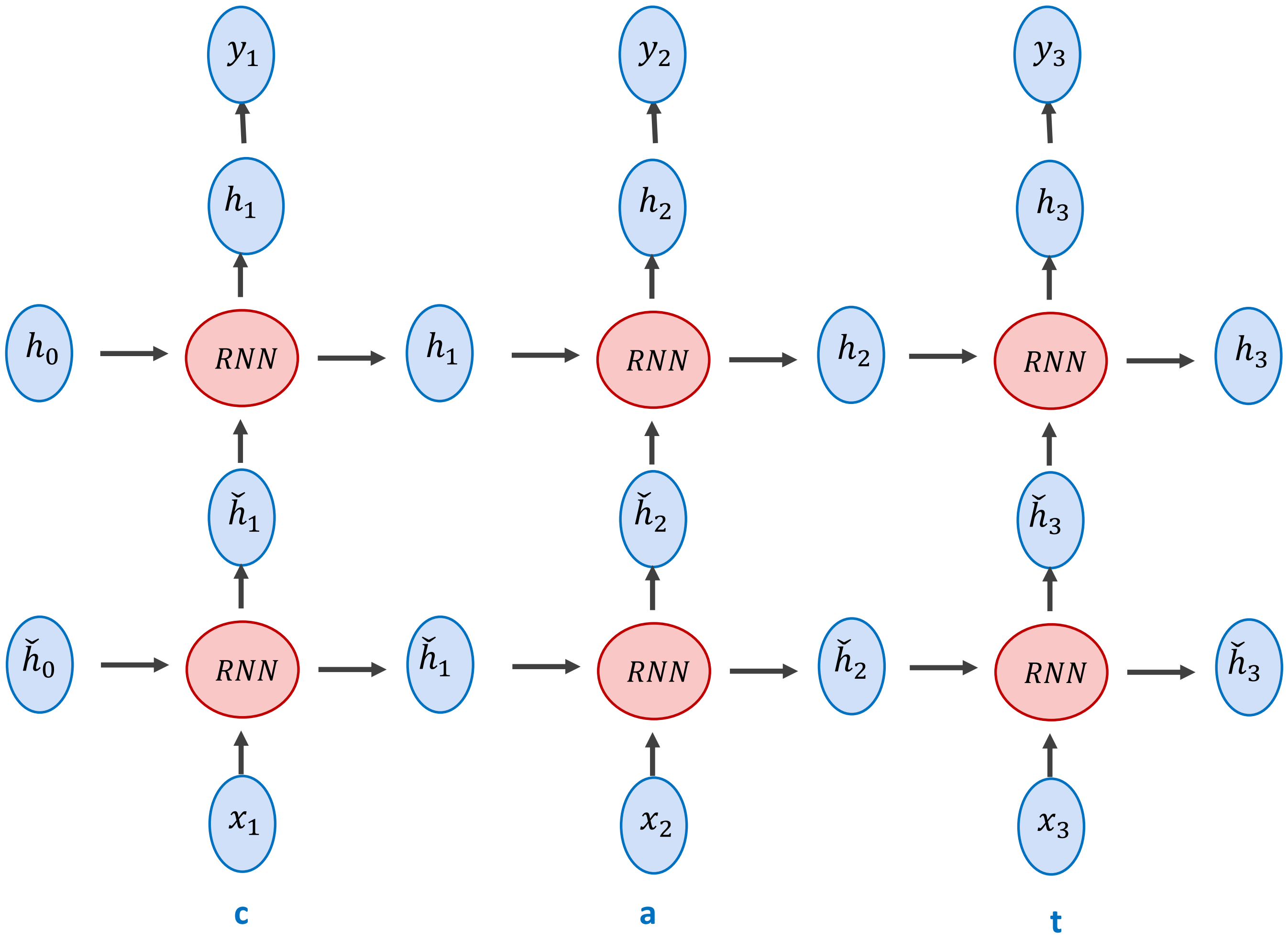
Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

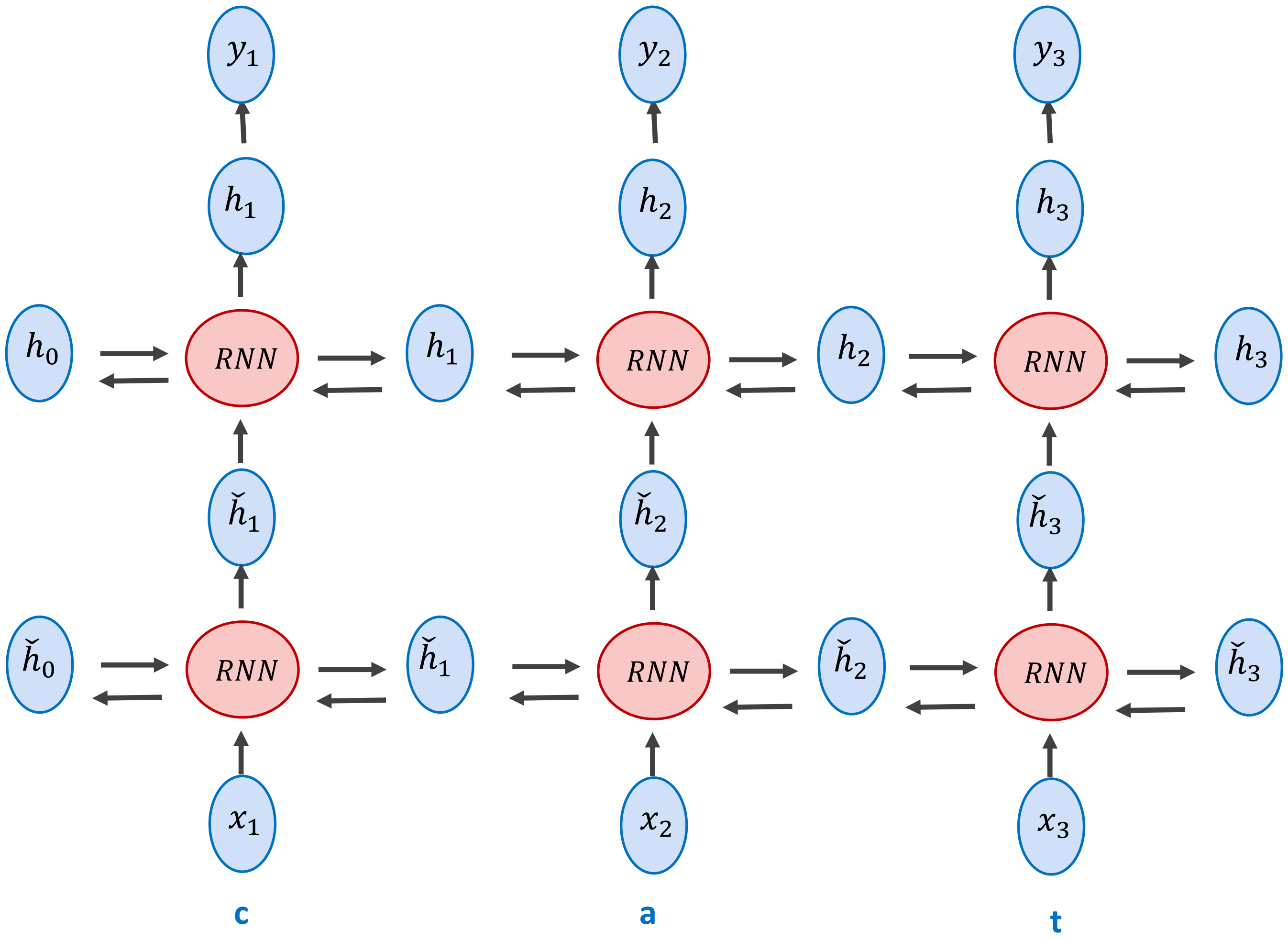
Bidirectional Recurrent Neural Network



Stacked Recurrent Neural Network



Stacked Bidirectional Recurrent Neural Network



RNN in Pytorch

Recurrent layers

class `torch.nn.RNN`(**args*, ***kwargs*) [source]

Applies a multi-layer Elman RNN with tanh or ReLU non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(w_{ih} * x_t + b_{ih} + w_{hh} * h_{(t-1)} + b_{hh})$$

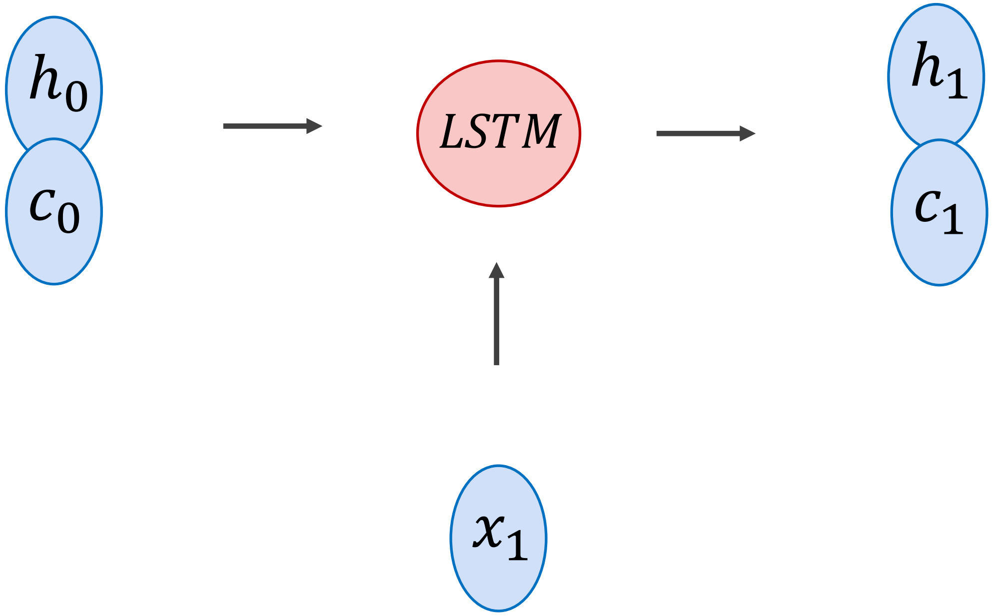
where h_t is the hidden state at time *t*, and x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer. If nonlinearity='relu', then *ReLU* is used instead of *tanh*.

Parameters:

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers.
- **nonlinearity** – The non-linearity to use ['tanh'|'relu']. Default: 'tanh'
- **bias** – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature)
- **dropout** – If non-zero, introduces a dropout layer on the outputs of each RNN layer except the last layer
- **bidirectional** – If True, becomes a bidirectional RNN. Default: False

LSTM Cell (Long Short-Term Memory)

$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \tag{7}$$
$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \tag{8}$$
$$c_t = f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \tag{9}$$
$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right) \tag{10}$$
$$h_t = o_t \tanh(c_t) \tag{11}$$



LSTM in Pytorch

class `torch.nn.LSTM`(**args, **kwargs*) [source]

Applies 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:

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

where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the hidden state of the previous layer at time t or $input_t$ for the first layer, and i_t, f_t, g_t, o_t are the input, forget, cell, and out gates, respectively.

Parameters:

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers.
- **bias** – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature)
- **dropout** – If non-zero, introduces a dropout layer on the outputs of each RNN layer except the last layer
- **bidirectional** – If True, becomes a bidirectional RNN. Default: False

GRU in Pytorch

class `torch.nn.GRU`(**args, **kwargs*) [source]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

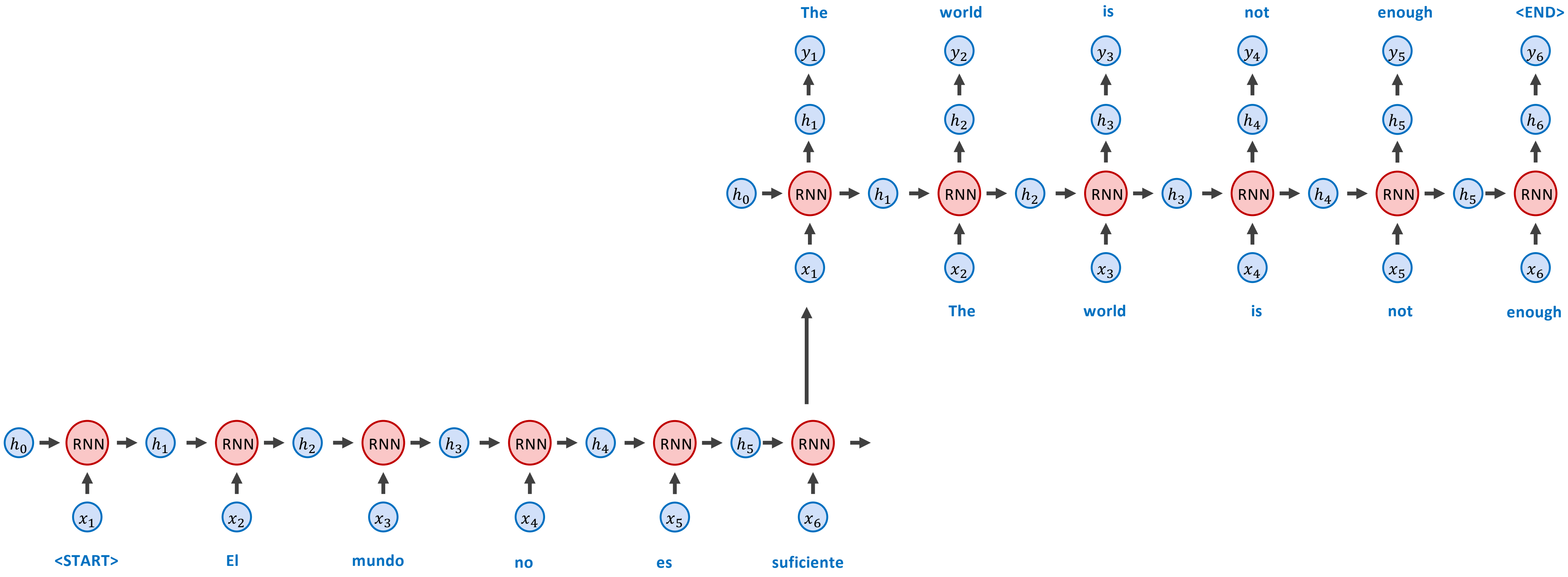
$$\begin{aligned} r_t &= \mathrm{sigmoid}(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\ z_t &= \mathrm{sigmoid}(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\ n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\ h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)} \end{aligned}$$

where h_t is the hidden state at time t, x_t is the hidden state of the previous layer at time t or $input_t$ for the first layer, and r_t, z_t, n_t are the reset, input, and new gates, respectively.

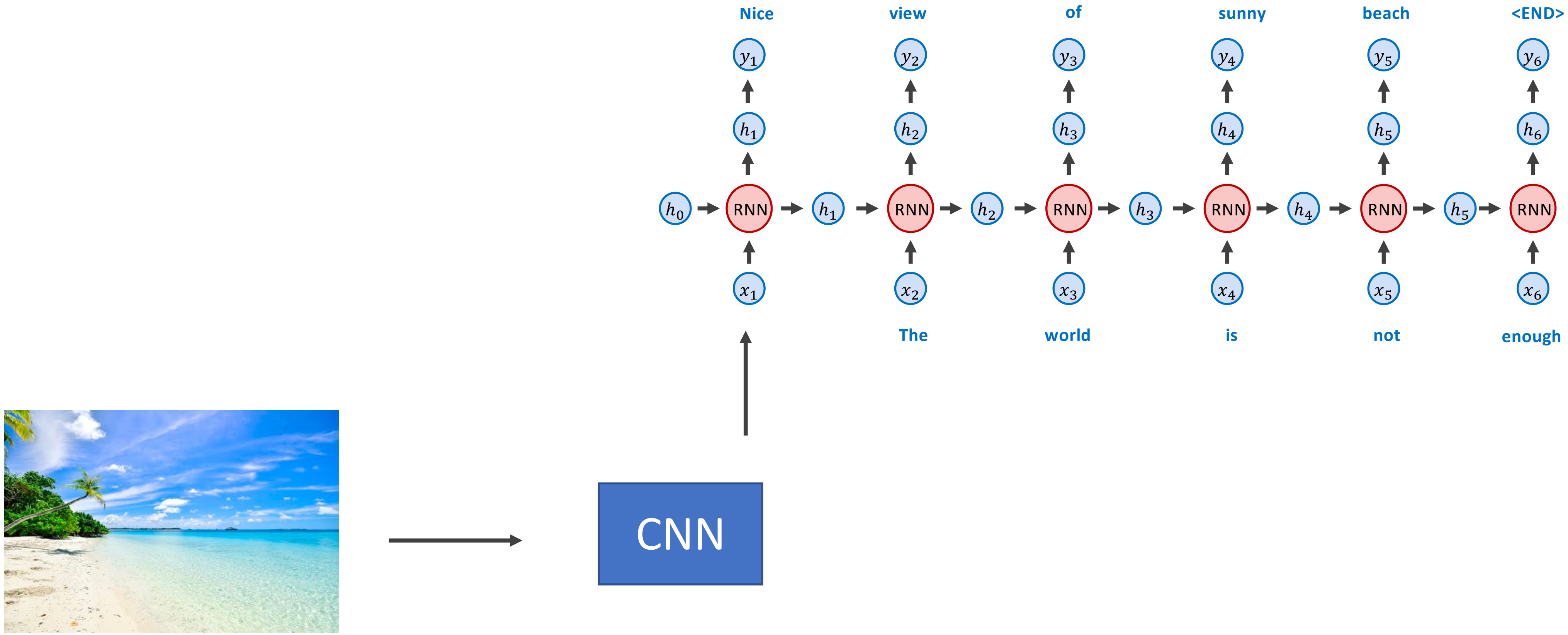
Parameters:

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers.
- **bias** – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature)
- **dropout** – If non-zero, introduces a dropout layer on the outputs of each RNN layer except the last layer
- **bidirectional** – If True, becomes a bidirectional RNN. Default: False

RNNs for Image Caption Generation



RNNs for Image Caption Generation



References (a lot of them)

- Vinyals et al. Show and Tell: A Neural Image Caption Generator https://arxiv.org/abs/1411.4555

- Mao et al. Deep Captioning with Multimodal Recurrent Neural Networks (m-RNN). https://arxiv.org/abs/1412.6632

- Karpathy and Fei-Fei. Deep Visual-Semantic Alignments for Generating Image Descriptions. https://arxiv.org/abs/1412.2306

- Fang et al. From Captions to Visual Concepts and Back. https://arxiv.org/abs/1411.4952

- Yin and Ordonez. Obj2Text: Generating Visually Descriptive Language from Object Layouts. https://arxiv.org/abs/1707.07102 (not exactly targeting image captioning specifically but locally grown paper so let me self-promote)

Questions?