Deep Learning for Vision & Language

Natural Language Processing II: Introduction

How to represent a phrase/sentence?

bag-of-words representation

person holding dog	{1, 3, 4}	[1 0 1 1 0 0 0 0 0 0]
person holding cat	{2, 3, 4}	[0 1 1 1 0 0 0 0 0 0]
person using computer	{3, 7, 6}	[0 0 1 0 0 1 1 0 0 0]

dog, cat, person, holding, tree, computer, using

person using computer person holding cat	{3, 3, 7, 6, 2}	[0 1 2 1 0 1 1 0 0 0]

What if vocabulary is very large?

Sparse Representation

bag-of-words representation

person holding dog	{1, 3, 4}	indices = [1, 3, 4]	values = [1, 1, 1]
person holding cat	{2, 3, 4}	indices = [2, 3, 4]	values = [1, 1, 1]
person using computer	{3, 7, 6}	indices = [3, 7, 6]	values = [1, 1, 1]

person using computer person holding cat	{3, 3, 7, 6, 2}	indices = [3, 7, 6, 2]	values = [2, 1, 1, 1]

Recap

- Bag-of-words encodings for text (e.g. sentences, paragraphs, captions, etc)

You can take a set of sentences/documents and classify them, cluster them, or compute distances between them using this representation.

Problem with this bag-of-words representation

my friend makes a nice meal

These would be the same using bag-of-words

my nice friend makes a meal

Bag of Bi-grams

indices = [10132, 21342, 43233, 53123, 64233]
values = [1, 1, 1, 1, 1]

my friend makes a nice meal

{my friend, friend makes, makes a,

a nice, nice meal}

indices = [10232, 43133, 21342, 43233, 54233]
values = [1, 1, 1, 1, 1]

my nice friend makes a meal

{my nice, nice friend, friend makes,

makes a, a meal}

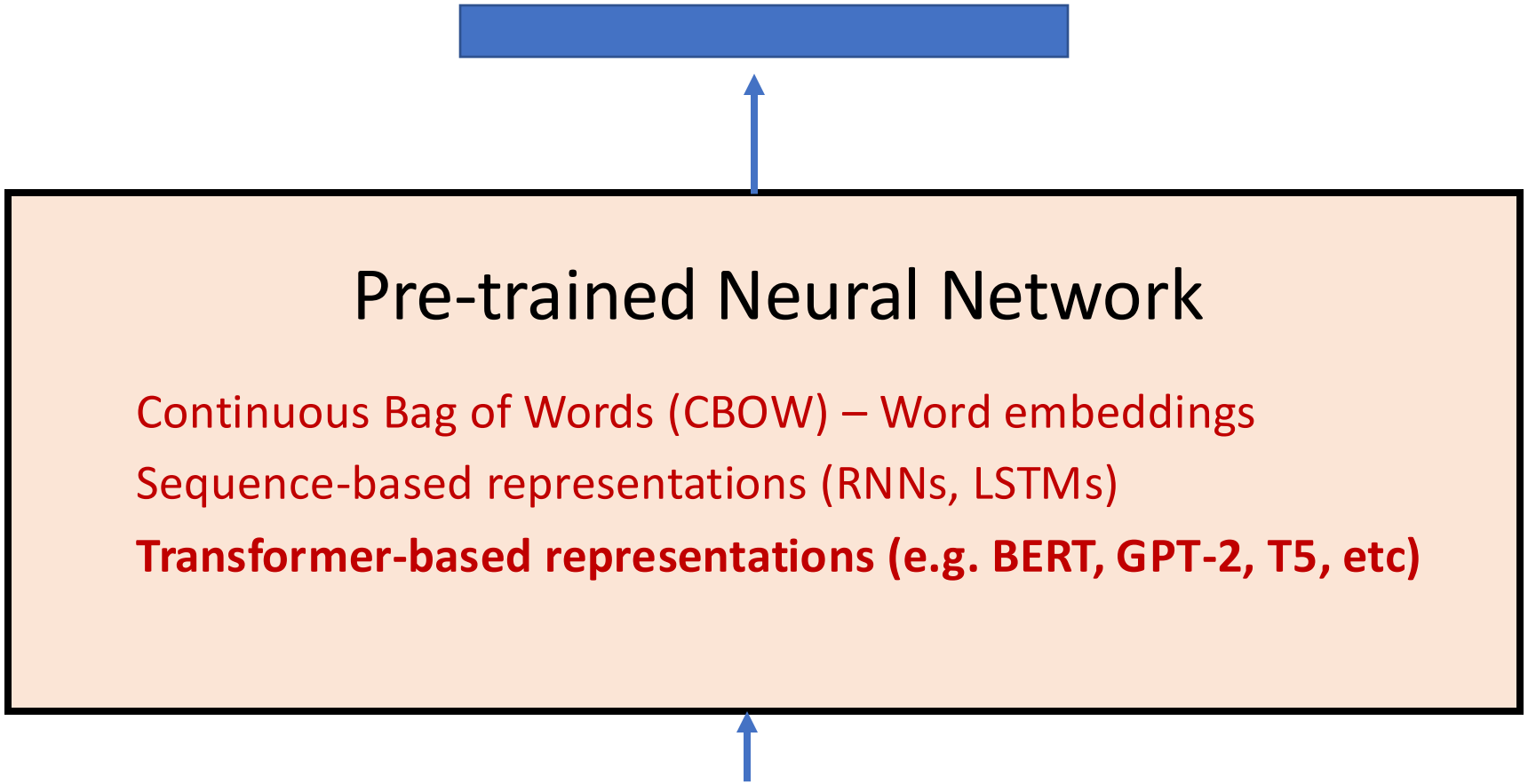
A dense vector-representation would be very inefficient
Think about tri-grams and n-grams

Recommended reading: n-gram language models

Yejin Choi's course on Natural Language Processing

http://www3.cs.stonybrook.edu/~ychoi/cse628/lecture/02-ngram.pdf

Modern way of representing Phrases/Text



Back to how to represent a word?

Problem: distance between words using one-hot encodings always the same

dog	1	$[1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$
cat	2	$[0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$
person	3	$[0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$

Idea: Instead of one-hot-encoding use a histogram of commonly co-occurring words.

Distributional Semantics

Dogs are man’s best friend.

I saw a dog on a leash walking in the park.

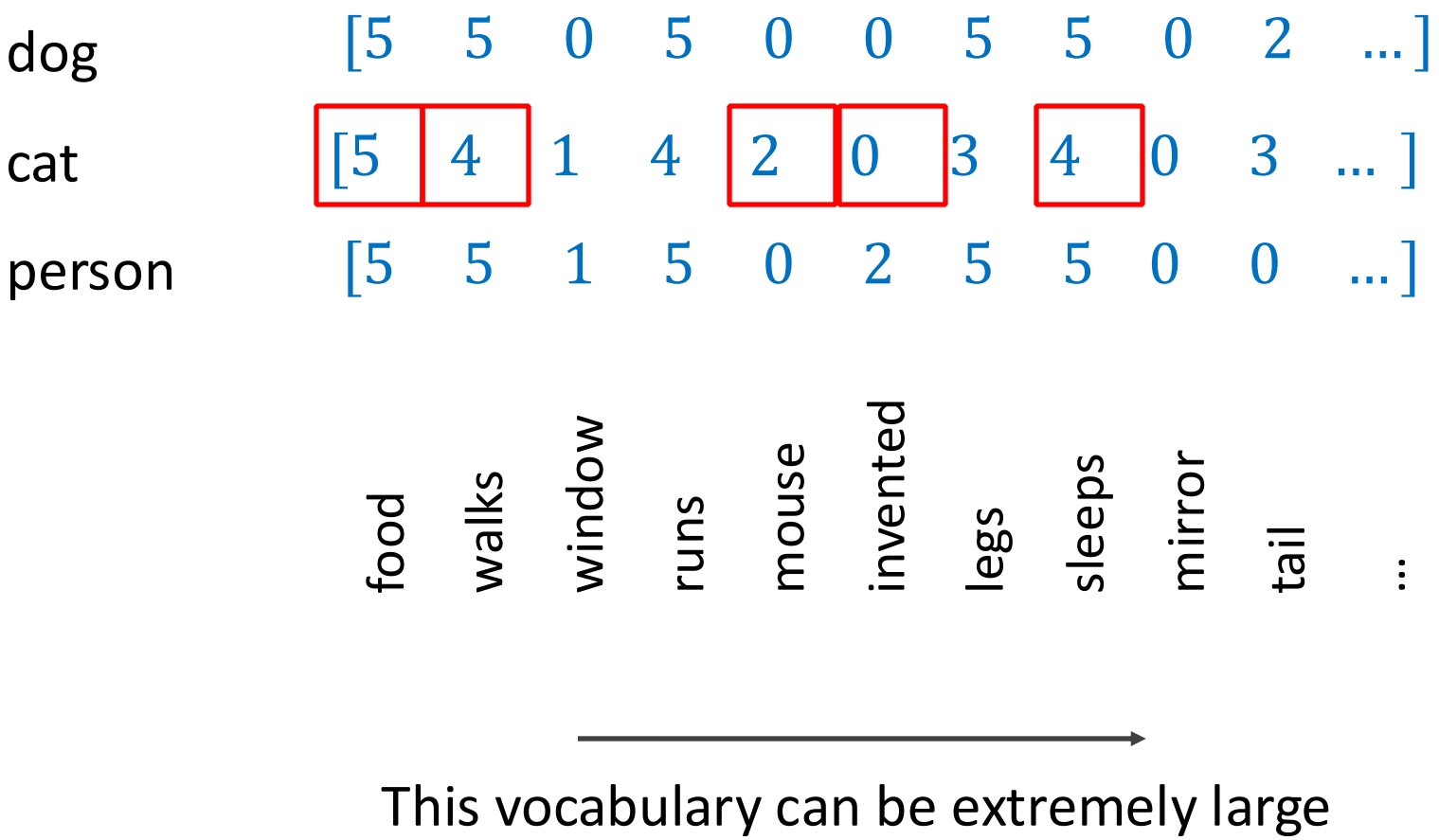
His dog is his best companion.

He walks his dog in the late afternoon

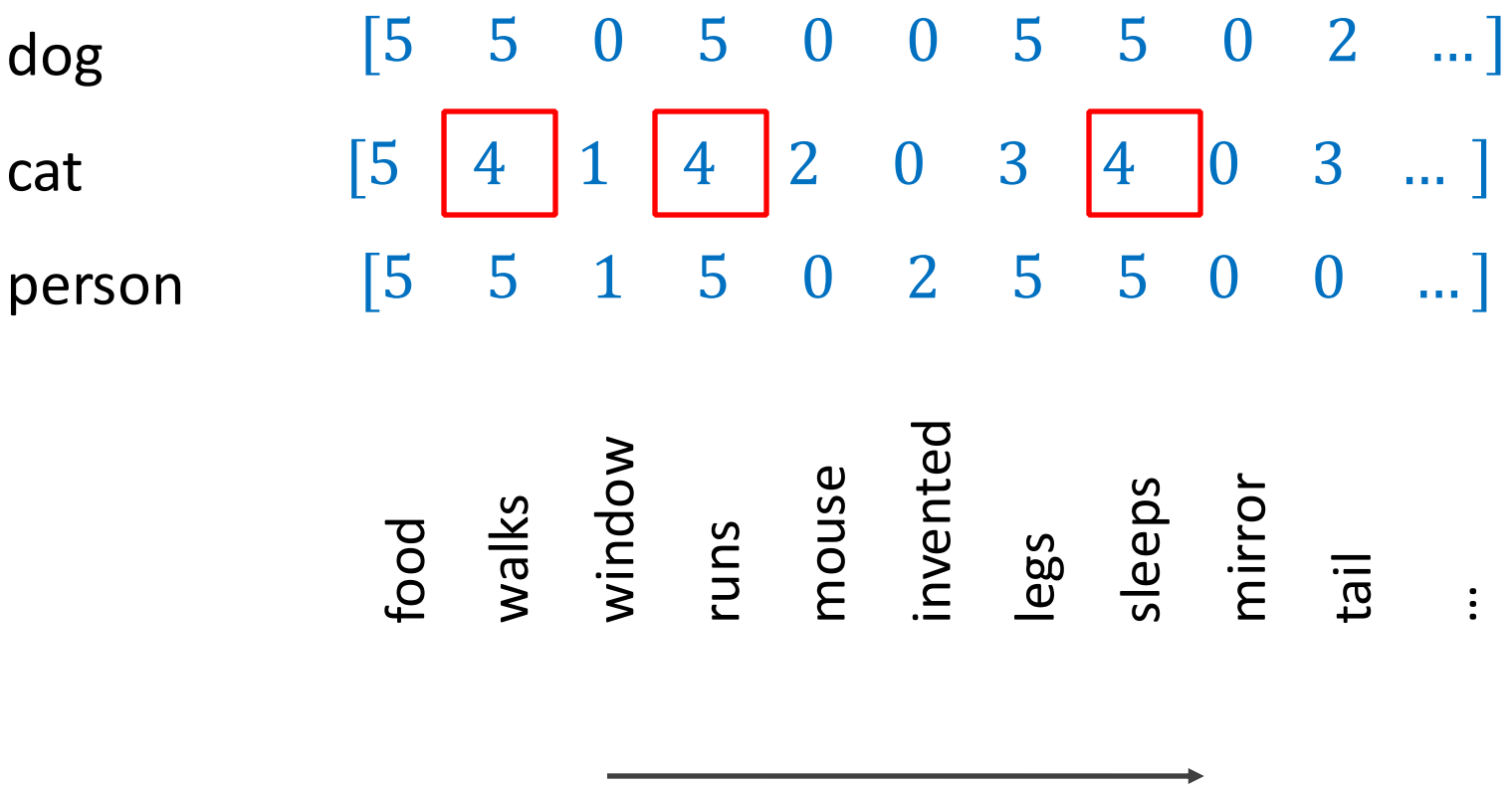
…

	friend	leash	park	walking	walks	food	legs	runs	sleeps	sits	…
dog	[3	2	3	4	2	4	3	5	6	7	…]

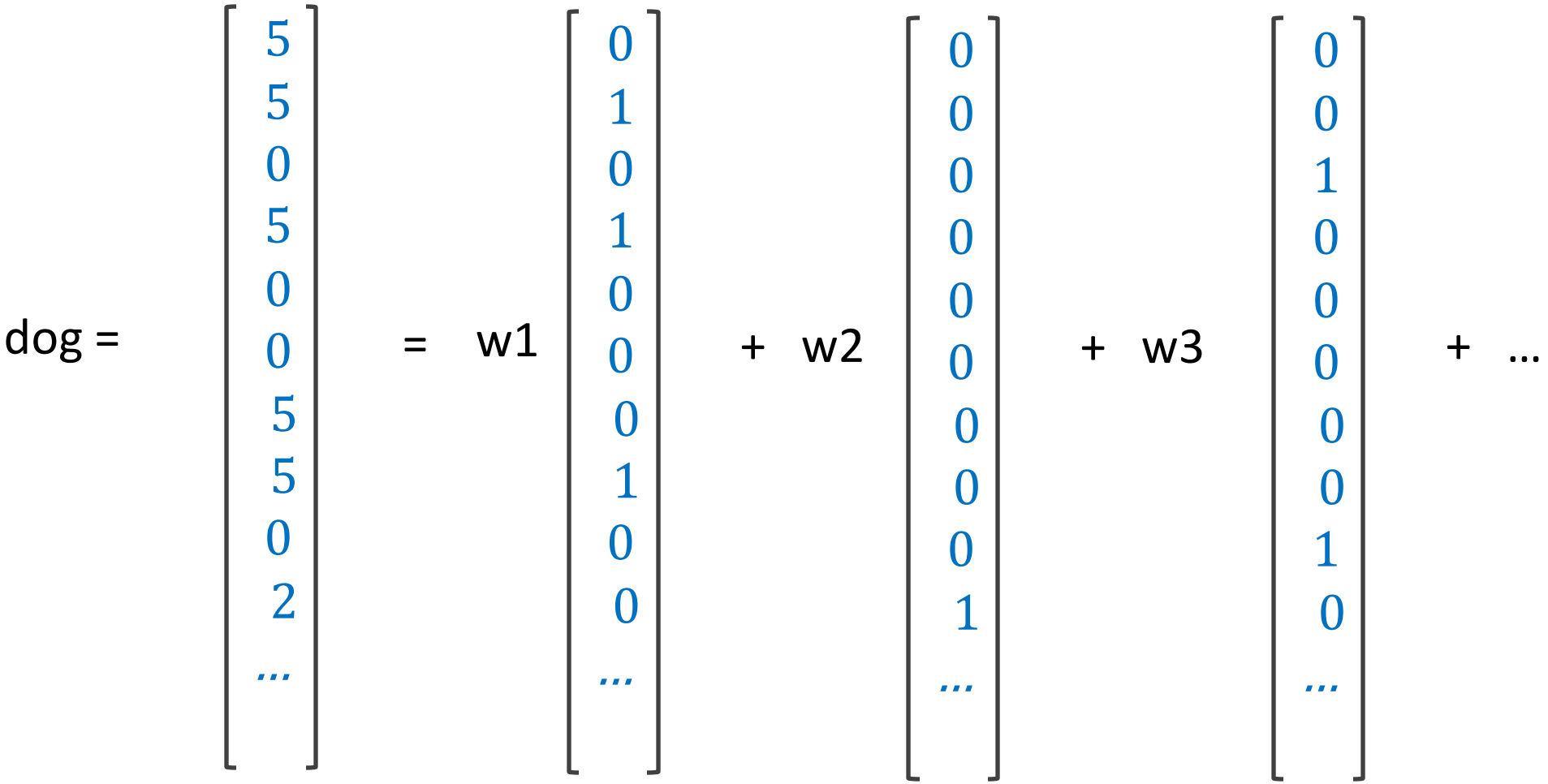
Distributional Semantics



Toward more Compact Representations



Toward more Compact Representations



Toward more Compact Representations

dog = [w1 w2 w3]

The basis vectors can be found using Principal Component Analysis (PCA)

This is known as Latent Semantic Analysis sometimes in NLP, maybe not anymore?

Toward more Compact Representations: Word Embeddings

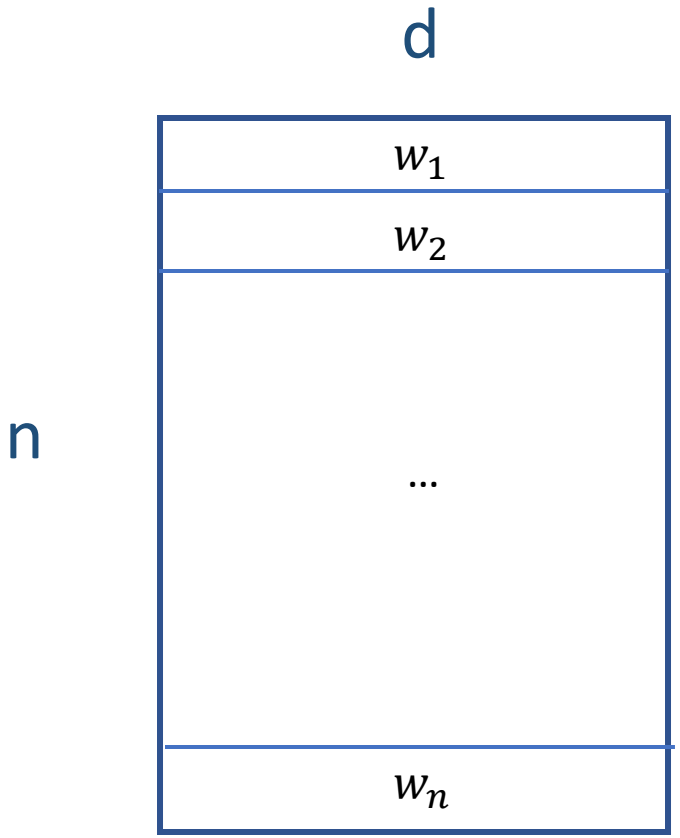
dog = [w1 w2 w3]

The weights w1, …, wn are found using a neural network

Word2Vec: https://arxiv.org/abs/1301.3781

Word2Vec – CBOW Version

- First, create a huge matrix of word embeddings initialized with random values – where each row is a vector for a different word in the vocabulary.



Efficient Estimation of Word Representations in Vector Space

Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

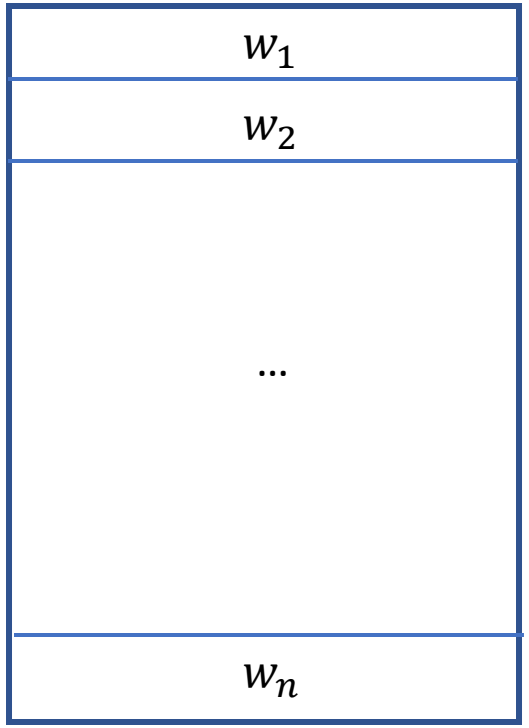
Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

Word2Vec – CBOW Version

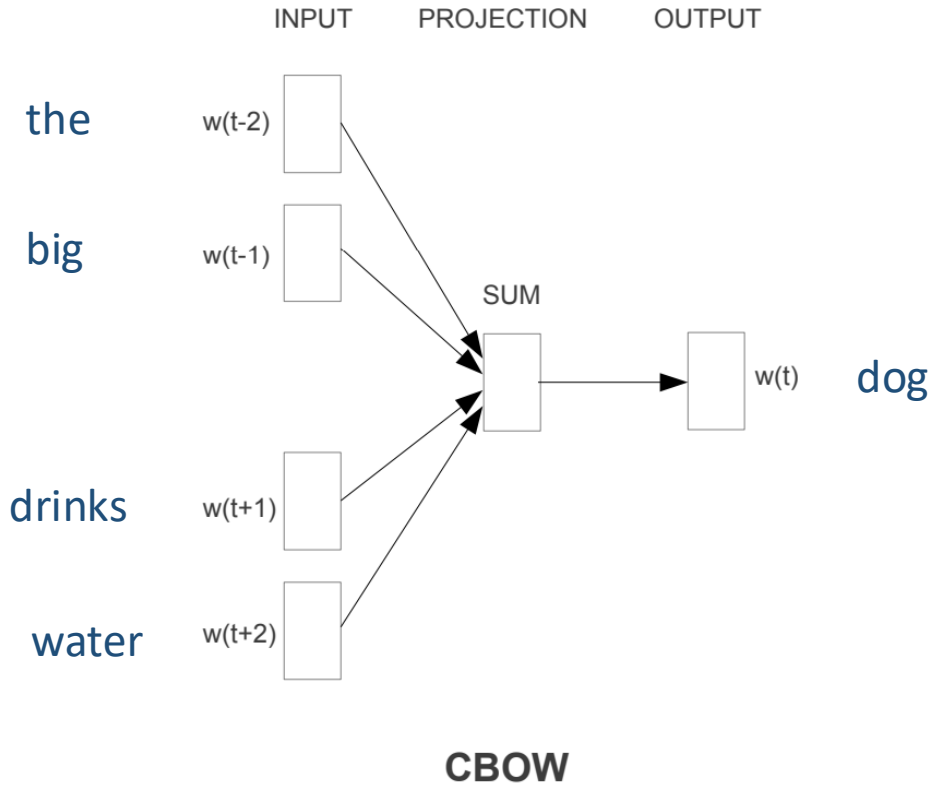
- Then, collect a lot of text, and solve the following regression problem for a large corpus of text:

d

n



"the big dog drinks water"



The Embedding Layer nn.Embedding

EMBEDDING

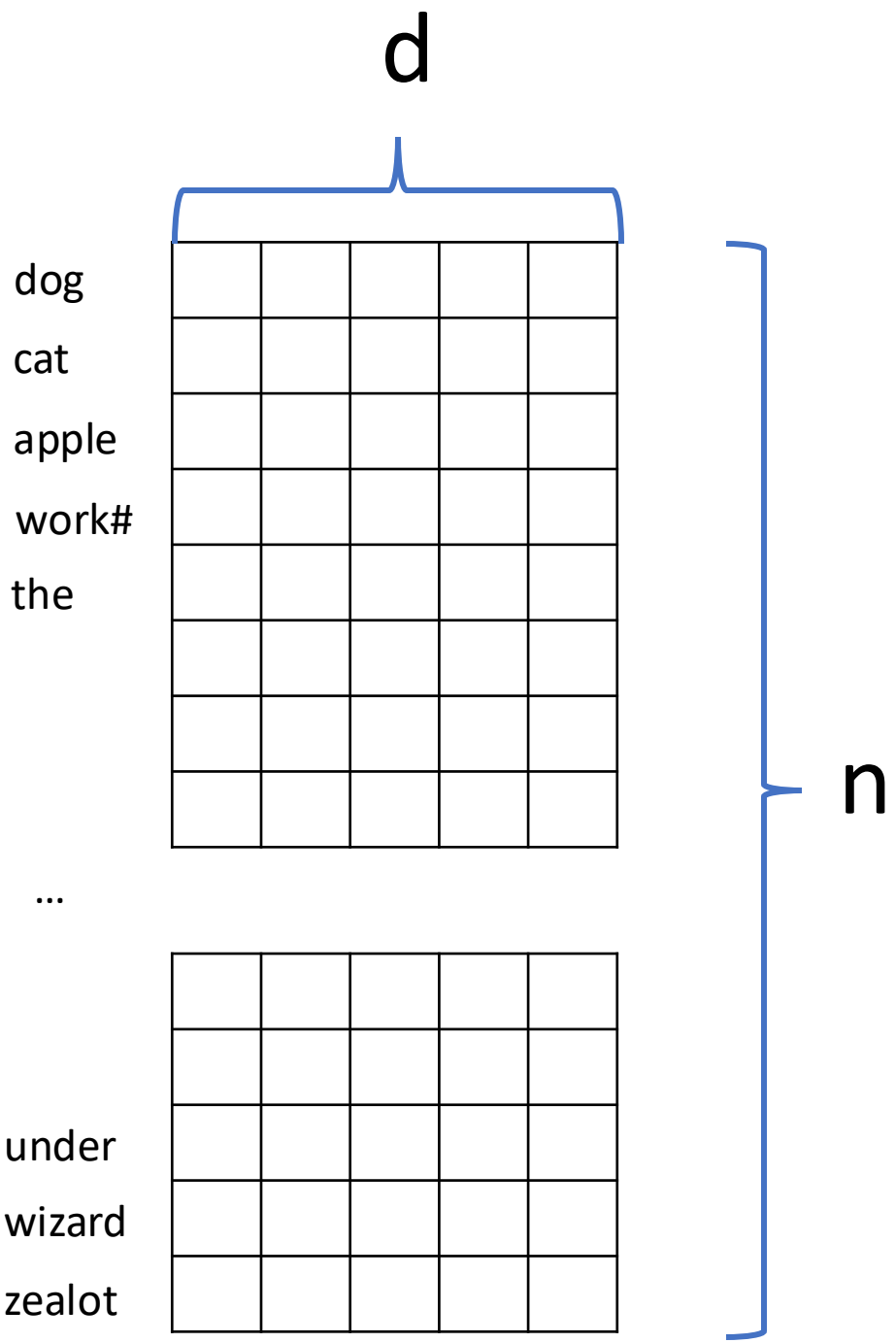
CLASS torch.nn.Embedding(*num_embeddings*, *embedding_dim*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*, *_weight=None*, *device=None*, *dtype=None*) [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Parameters:

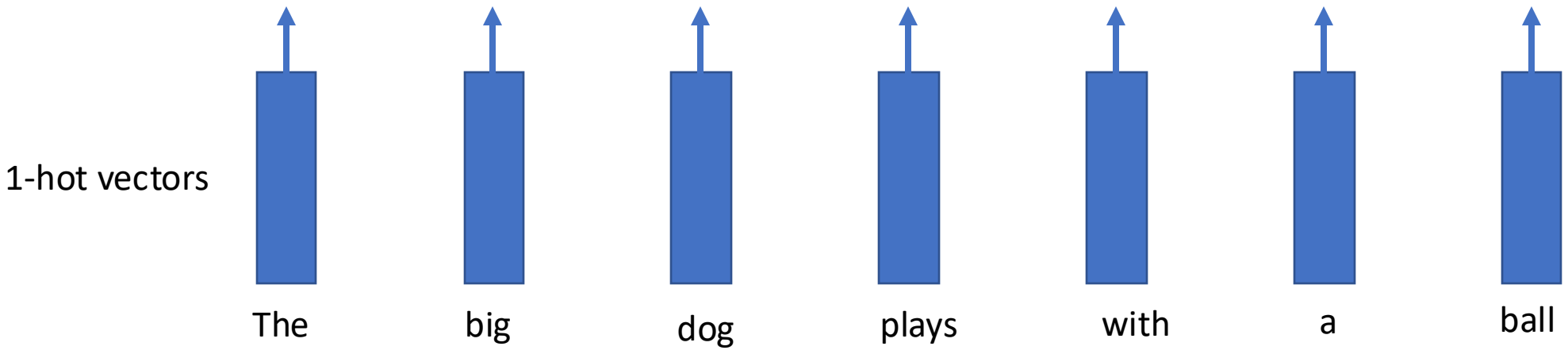
- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **padding_idx** (*int*, *optional*) – If specified, the entries at `padding_idx` do not contribute to the gradient; therefore, the embedding vector at `padding_idx` is not updated during training, i.e. it remains as a fixed "pad". For a newly constructed Embedding, the embedding vector at `padding_idx` will default to all zeros, but can be updated to another value to be used as the padding vector.



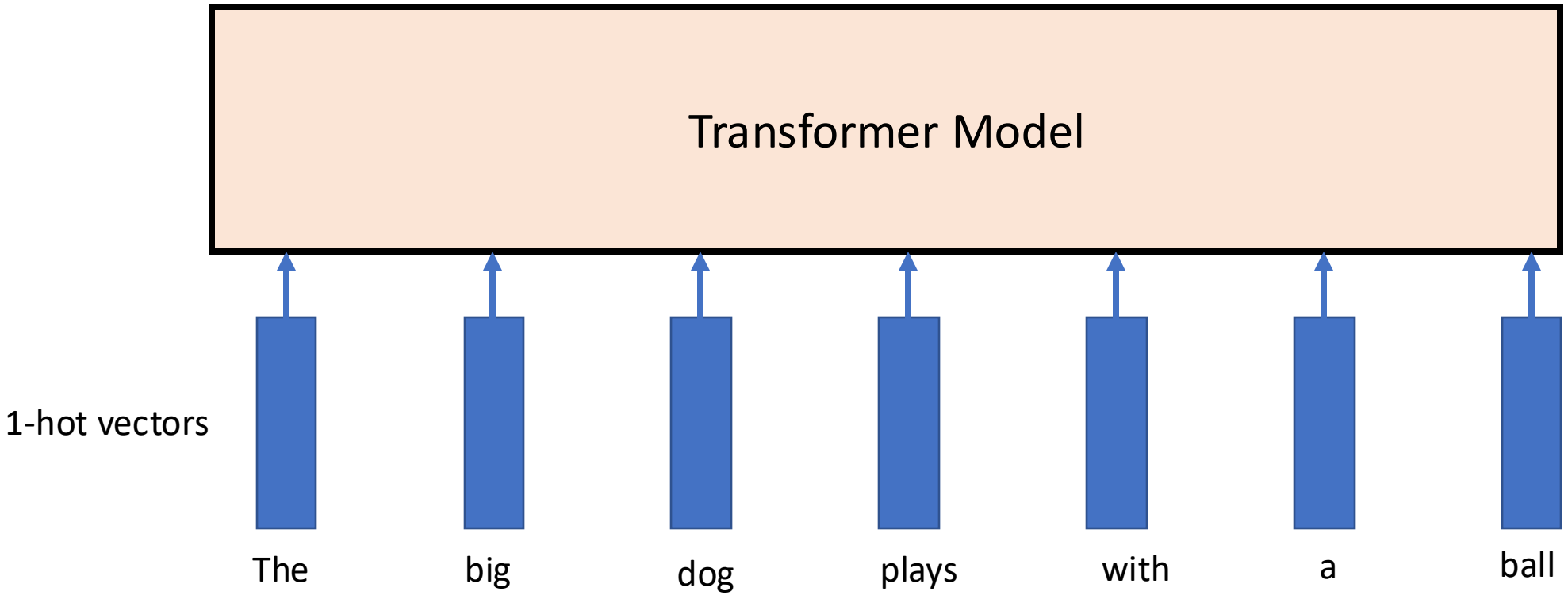
The Embedding Layer nn.Embedding

nn.Embedding(n, d)

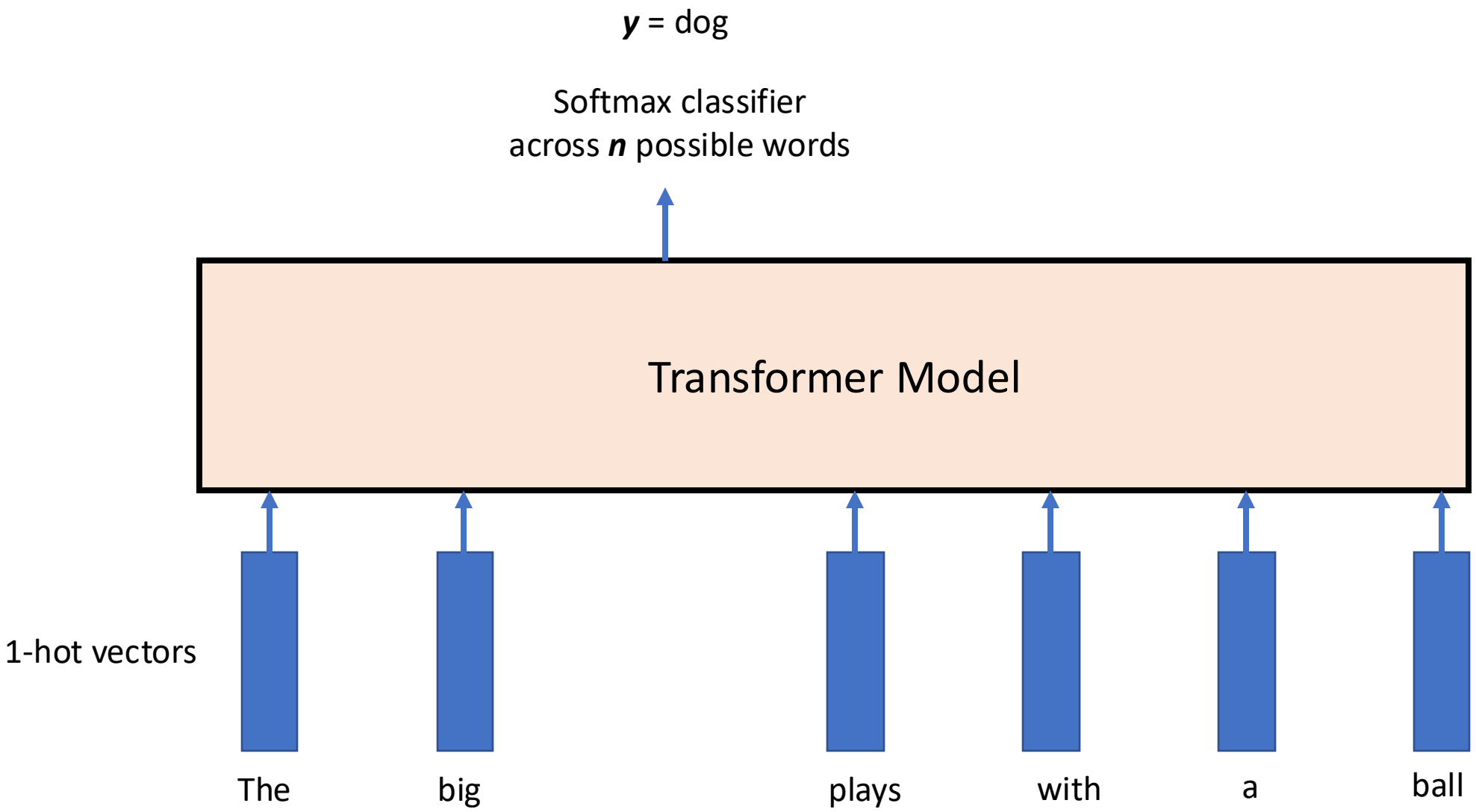
Pre-trained Language Models



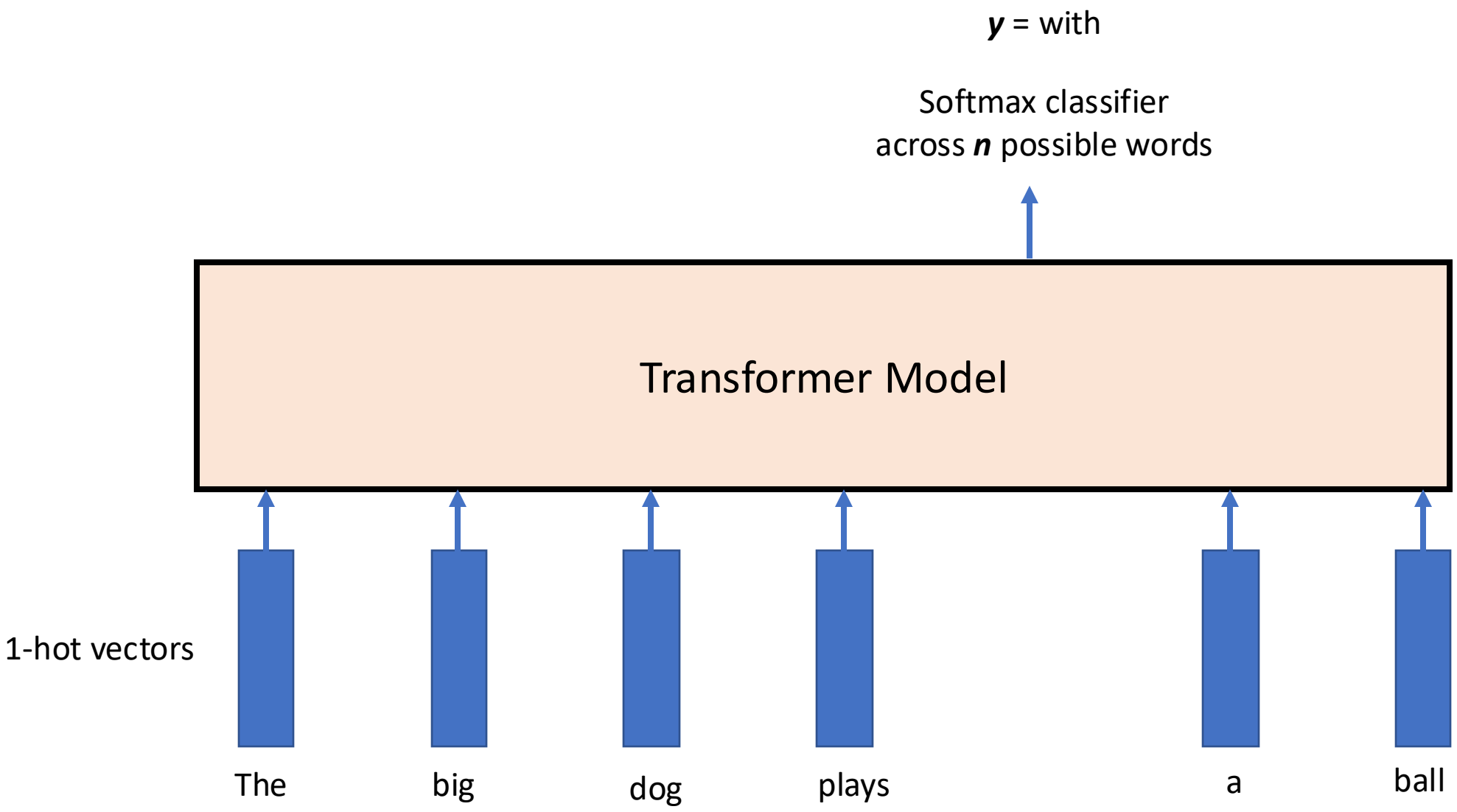
Pre-trained Language Models



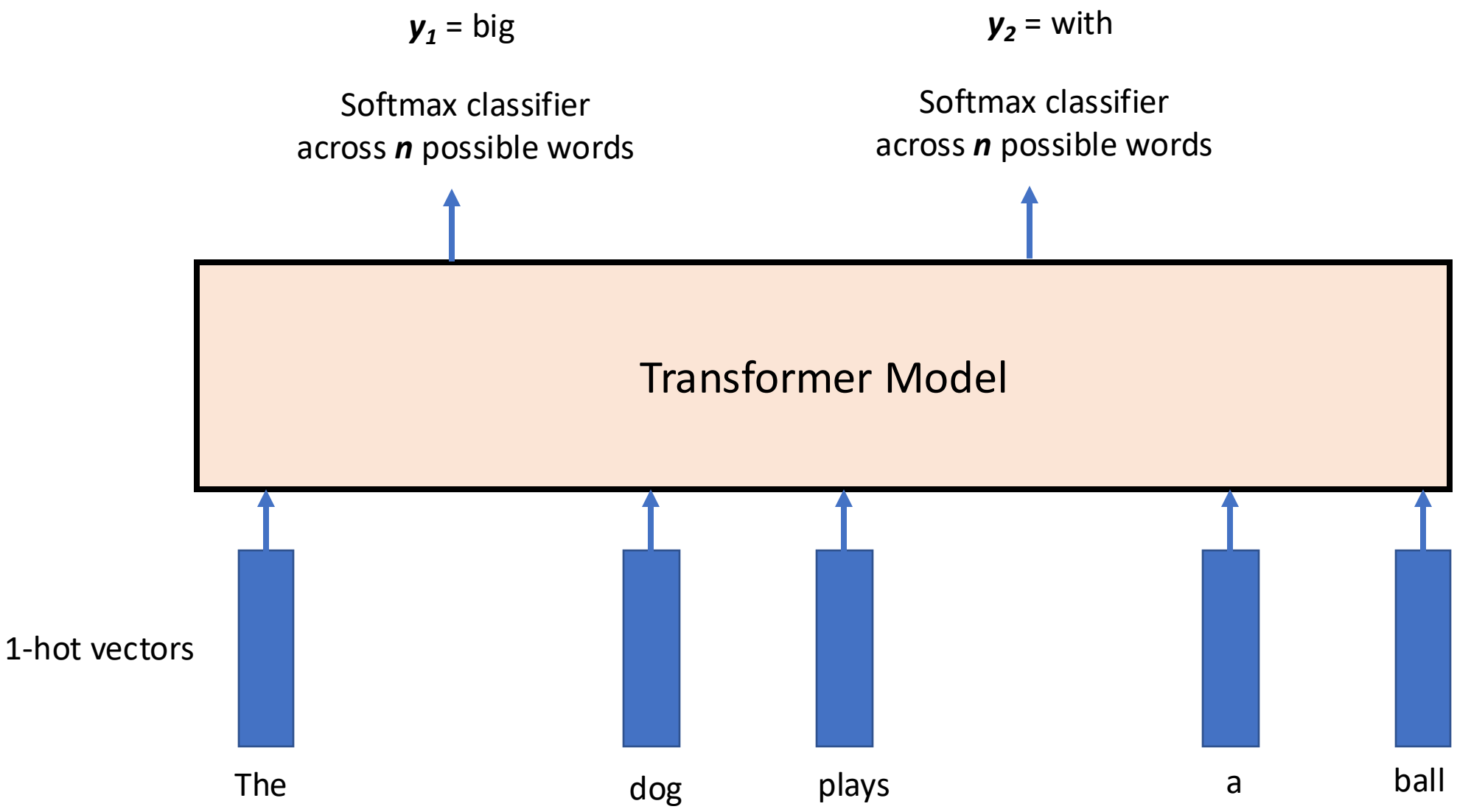
Pre-trained Language Models



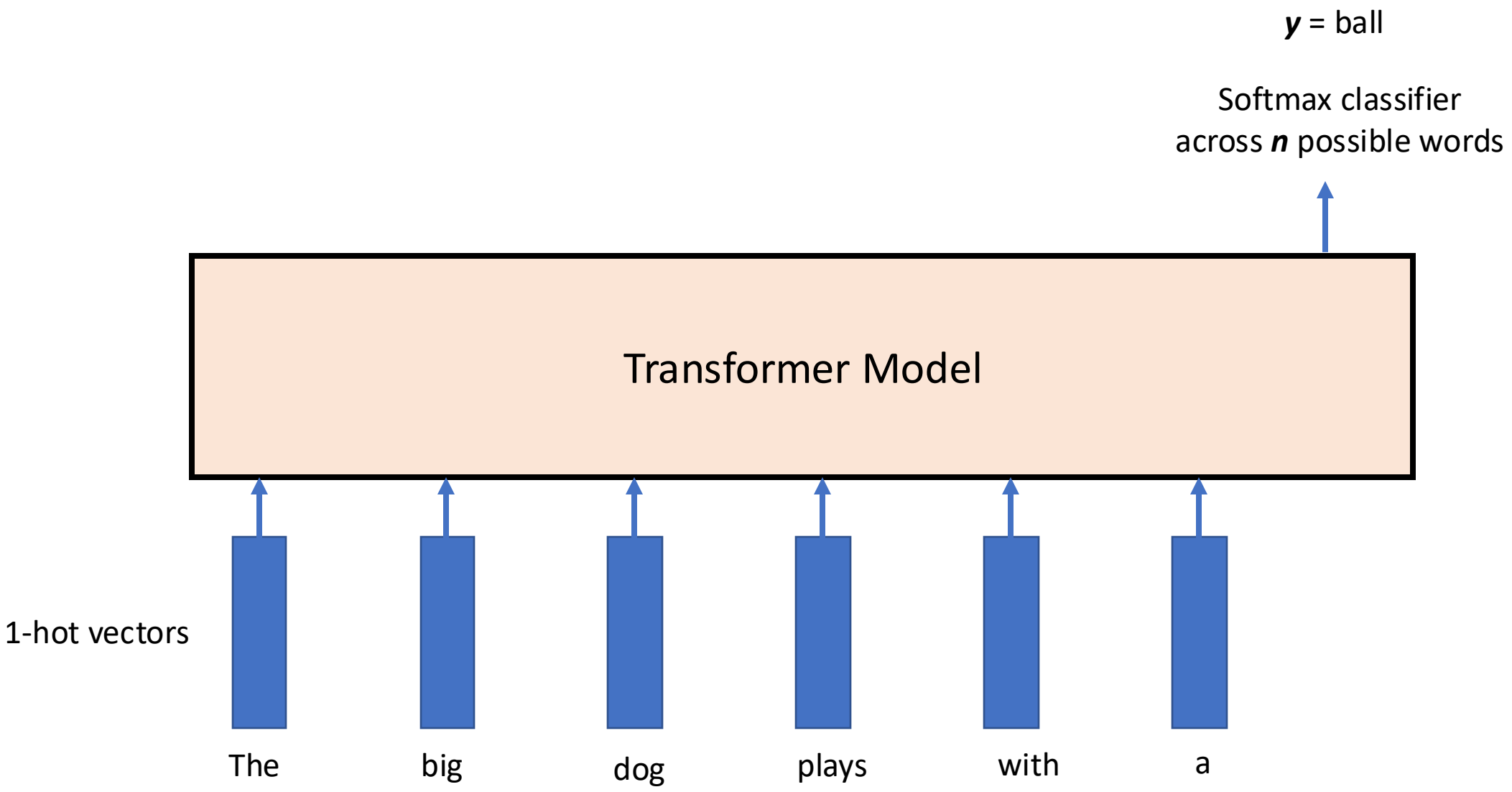
Pre-trained Language Models



Pre-trained Language Models



Generative Language Models



Practical Issues - Tokenization

- For each text representation we usually need to separate a sentence into tokens – we have assumed words in this lecture (or pairs of words) – but tokens could also be characters and anything in-between.

- Word segmentation can be used as tokenization.
  - In the assignment I was lazy I just did “my sentence”.split(“ “) and called it a day.
  - However, even English is more difficult than that because of punctuation, double spaces, quotes, etc. For English I would recommend you too look up the great word tokenization tools in libraries such as Python’s NLTK and Spacy before you try to come up with your own word tokenizer.

Issues with Word based Tokenization

- We already mentioned that tokenization can be hard even when word-based for other languages that don't use spaces in-between words.

- Word tokenization can also be bad for languages where the words can be "glued" together like German or Turkish.
  - Remember fünfhundertfünfundfünfzig? It wouldn't be feasible to have a word embedding for every number in the German language.

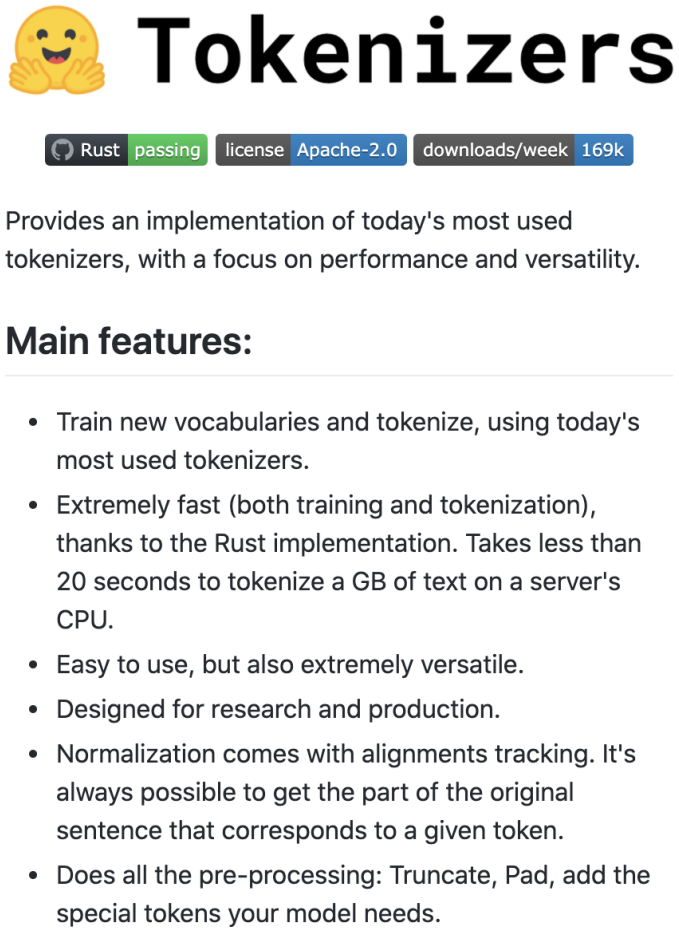
- It is problematic to handle words that are not in the vocabulary e.g. a common practice is to use a special <OOV> (out of vocabulary) token for those words that don't show up in the vocabulary.

Tokenization can be complex

- Think of Japanese
  - Three vocabularies/sets of symbols:
    Katakana and Hiragana symbols represent syllables / sounds
    く = ku, ぎ = gi, ナ = na, ア= a
    Kanji represent ideas / words (Chinese characters).
    日 = day, sun, 大 = big, 凸= convex 凹 = concave
  - They can be combined – e.g. tomorrow = 明日
  - Each symbol also has some structure within the symbols. They are not independently created. e.g. bright= 明るい , rising sun = 旭
  - And of course there are no spaces in between the characters.

Solution: Sub-word Tokenization

- Byte-pair Encoding Tokenization (BPE)
  - Start from small strings and based on substring counts iteratively use larger sequences until you define a vocabulary that maximizes informative subtokens. That way most will correspond to words at the end.
- Byte-level BPE Tokenizer
  - Do the same but at the byte representation level not at the substring representation level.



huggingface/tokenizers

BPE Tokenization Overview

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk,bhaddow@inf.ed.ac.uk

- Learn BPE operations (python code on the right) – from the paper.
- Use said operations to construct your sub-word vocabulary.
- Treat each sub-word token as a “word” in any models we will discuss.

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

https://colab.research.google.com/drive/1gUjL_h2tXdTtPSfxbBP-6MkE_BMck6gm?usp=sharing

Tokenization used in GPT-3

https://platform.openai.com/tokenizer

The cat is in the house

Tokens	Characters
6	23

The cat is in the house

[464, 3797, 318, 287, 262, 2156]

The geologist made an effort to rationalize the explanation

Tokens	Characters
11	59

The geologist made an effort to rationalize the explanation

[464, 4903, 7451, 925, 281, 3626, 284, 9377, 1096, 262, 7468]

fünfhundertfünfundfünfzig

Tokens	Characters
21	29

fünfhundertfünfundfünfzig

[69, 9116, 77, 69, 3907, 71, 4625, 83, 3907, 69, 9116, 77, 69, 3907, 917, 3907, 69, 9116, 77, 69, 38262]

La ardilla va a la universidad

Tokens	Characters
8	30

La ardilla va a la universidad

[14772, 33848, 5049, 46935, 257, 8591, 5820, 32482]

Tokenization used in GPT-3

https://platform.openai.com/tokenizer

深層学

Tokens	Characters
8	3

[162, 115, 109, 161, 109, 97, 27764, 99]

কেমন আছেন?

Tokens	Characters
20	10

[48071, 243, 156, 100, 229, 48071, 106, 48071, 101, 220, 48071, 228, 48071, 249, 156, 100, 229, 48071, 101, 30]

வணக்கம்

Tokens	Characters
21	7

[156, 106, 113, 156, 106, 96, 156, 106, 243, 156, 107, 235, 156, 106, 243, 156, 106, 106, 156, 107, 235]

Recurrent Neural Networks

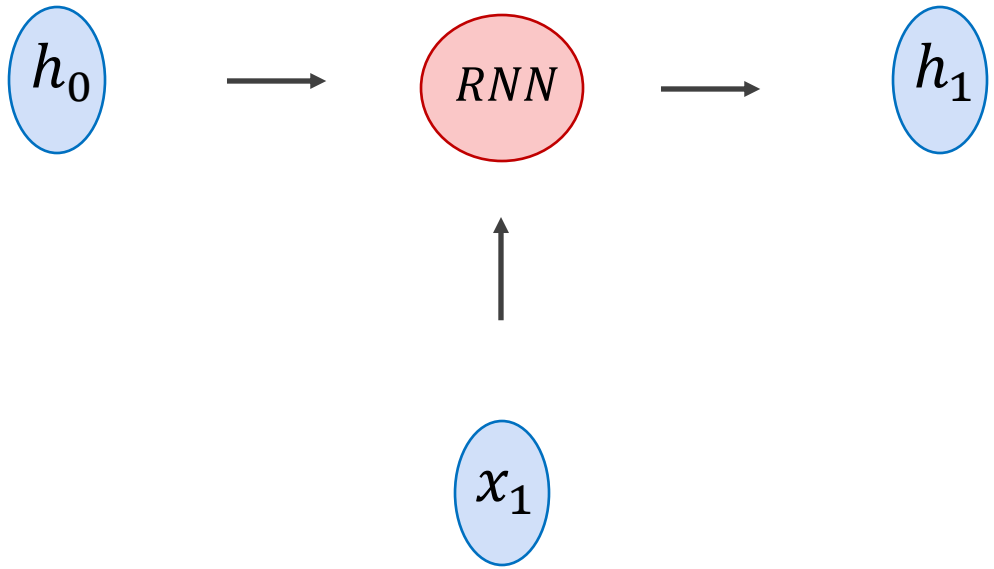
- These are models for handling sequences of things.

- Each input is not a vector but a sequence of input vectors.

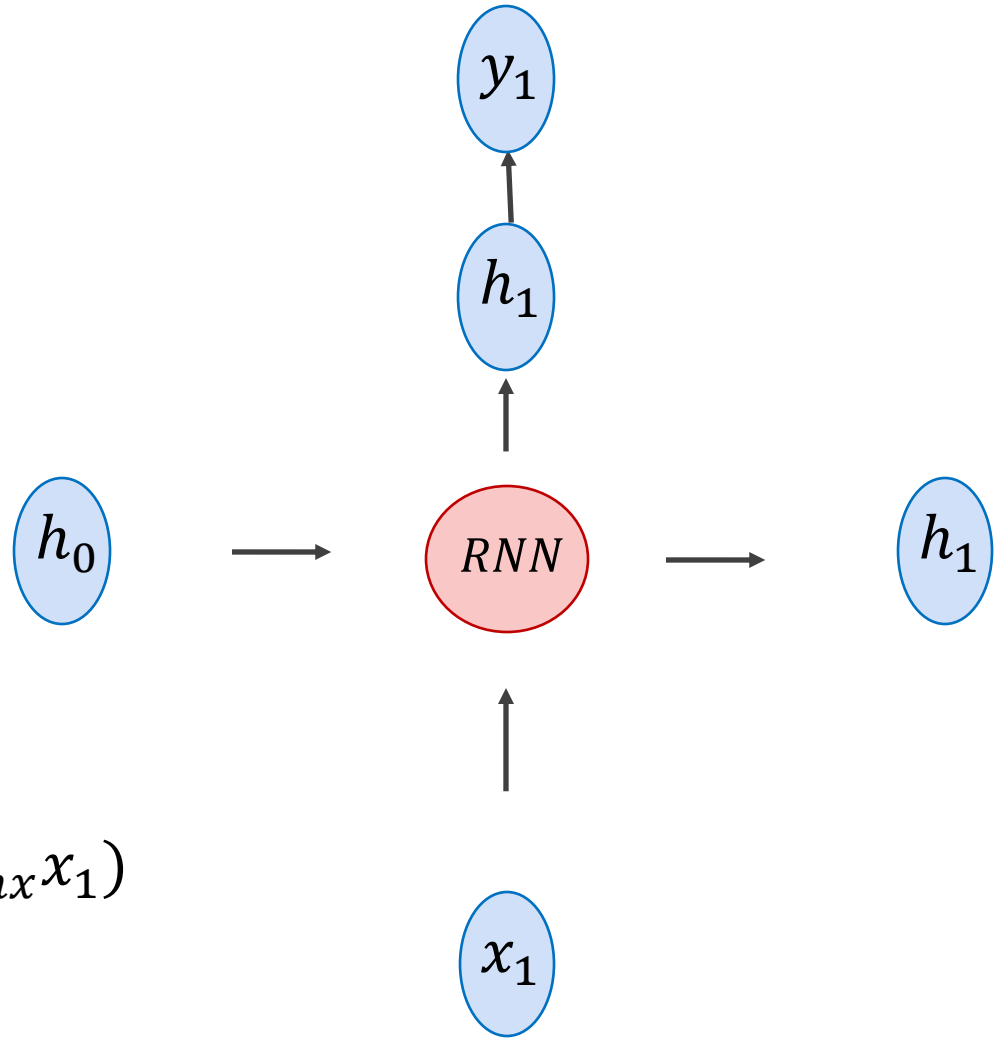
- e.g. Each input can be a “word embedding” or any “word” representation – we will use in our first examples one-hot encoded tokens but in practice continuous dense word embeddings are used.

Recurrent Neural Network Cell

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



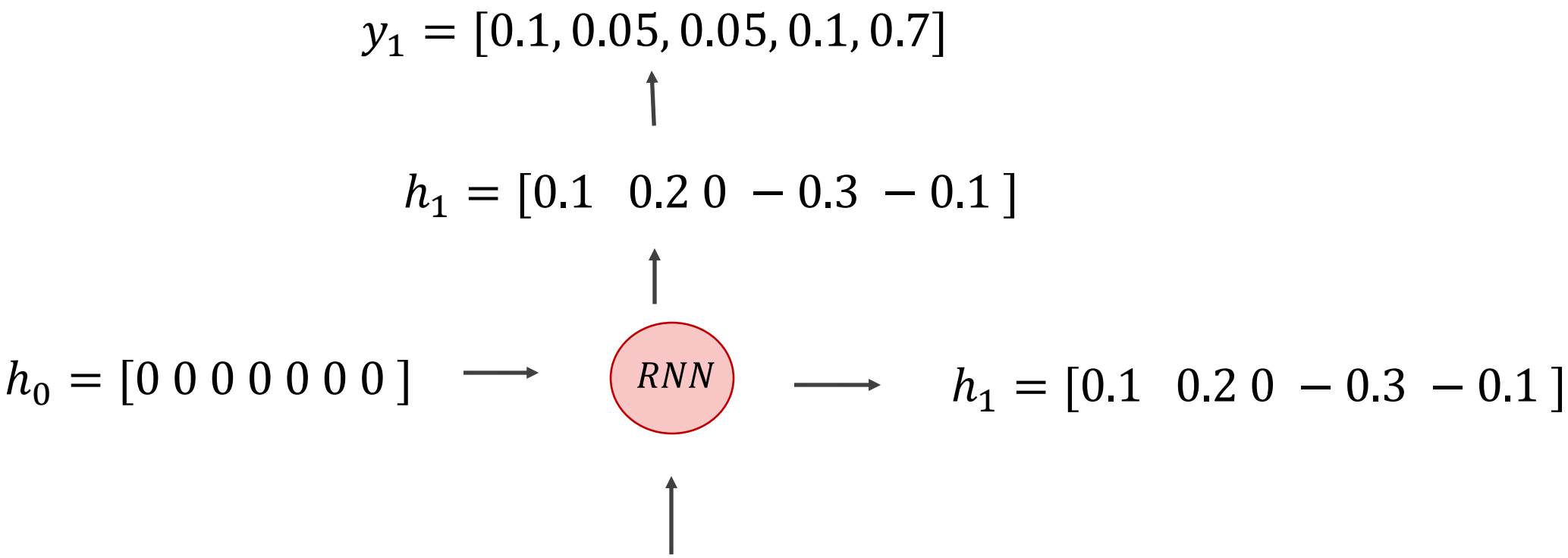
Recurrent Neural Network Cell



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

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

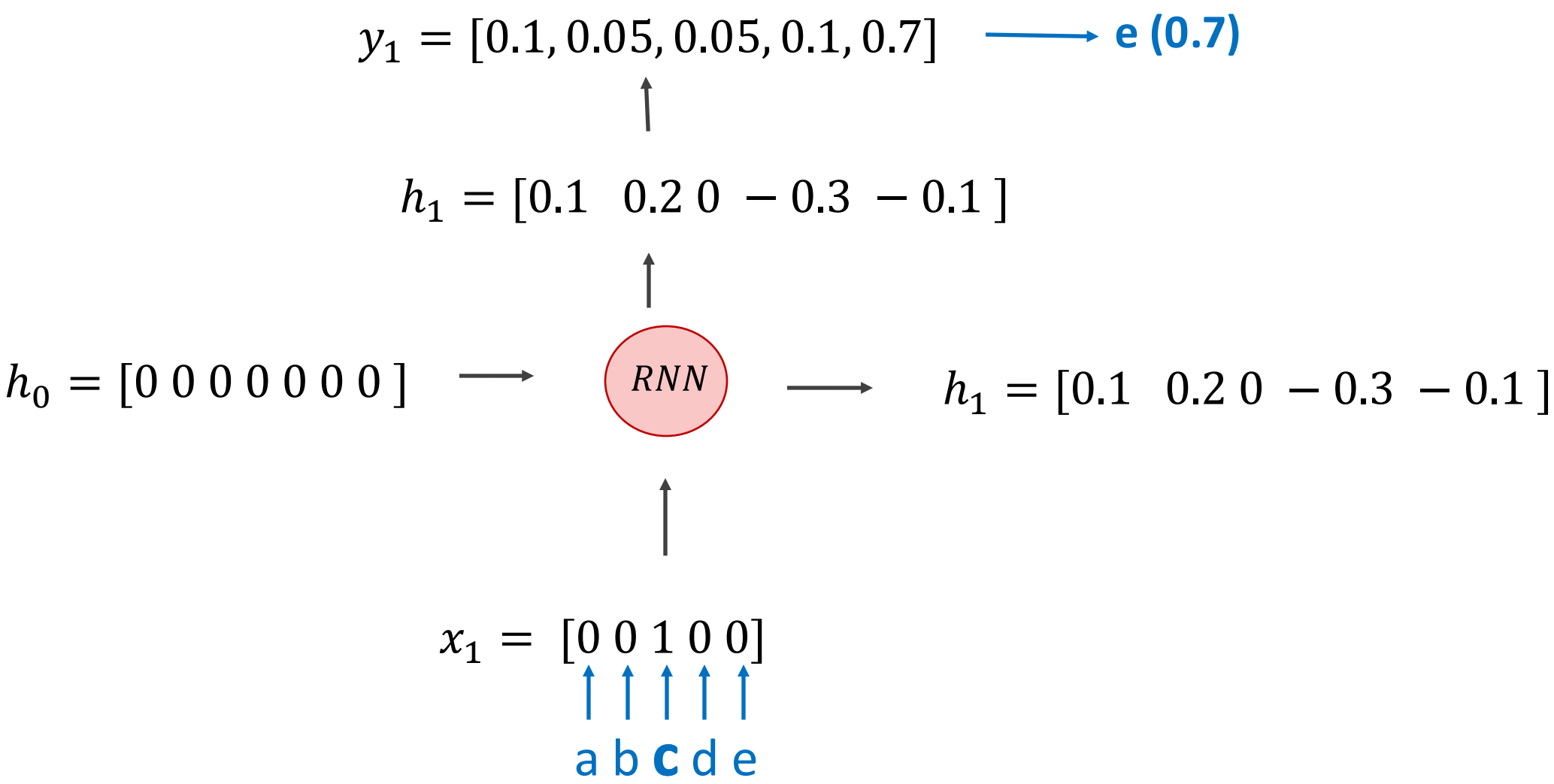
Recurrent Neural Network Cell



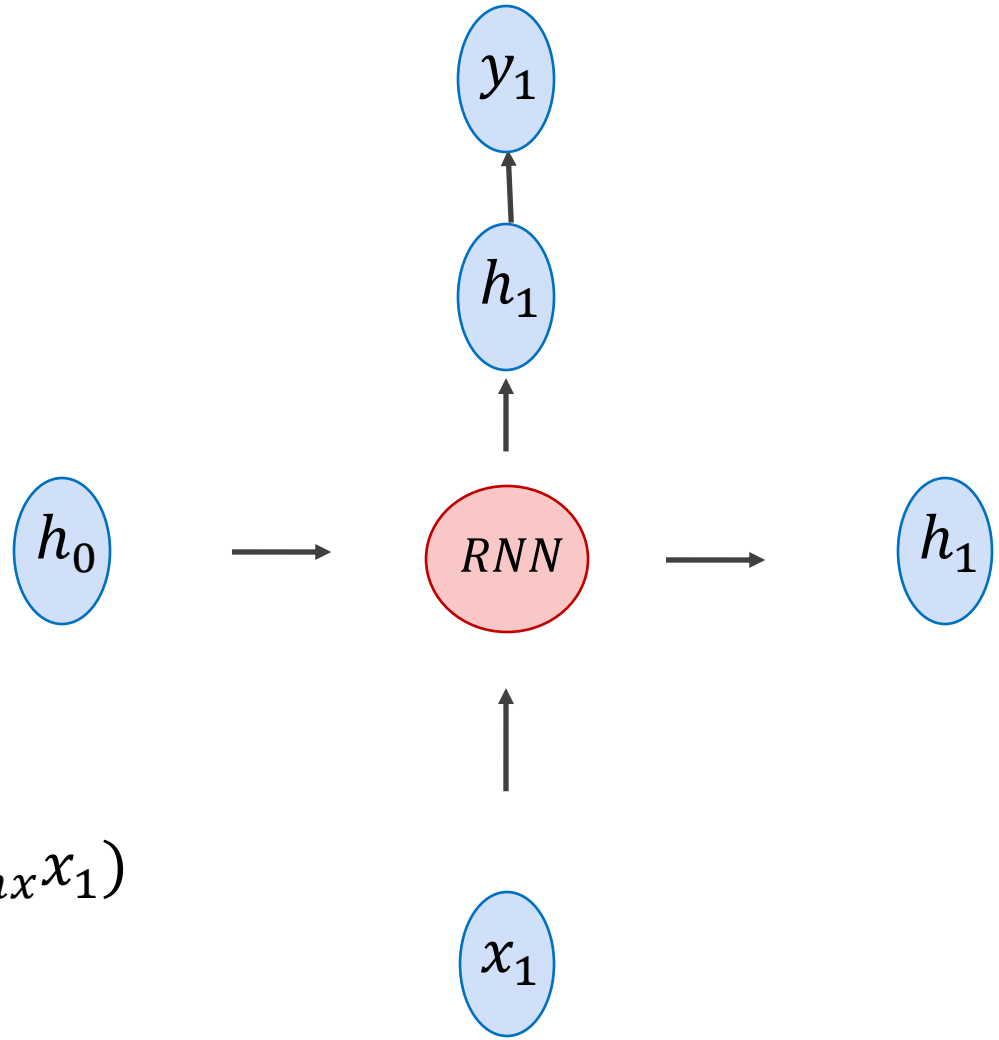
$$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1)$$

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

Recurrent Neural Network Cell



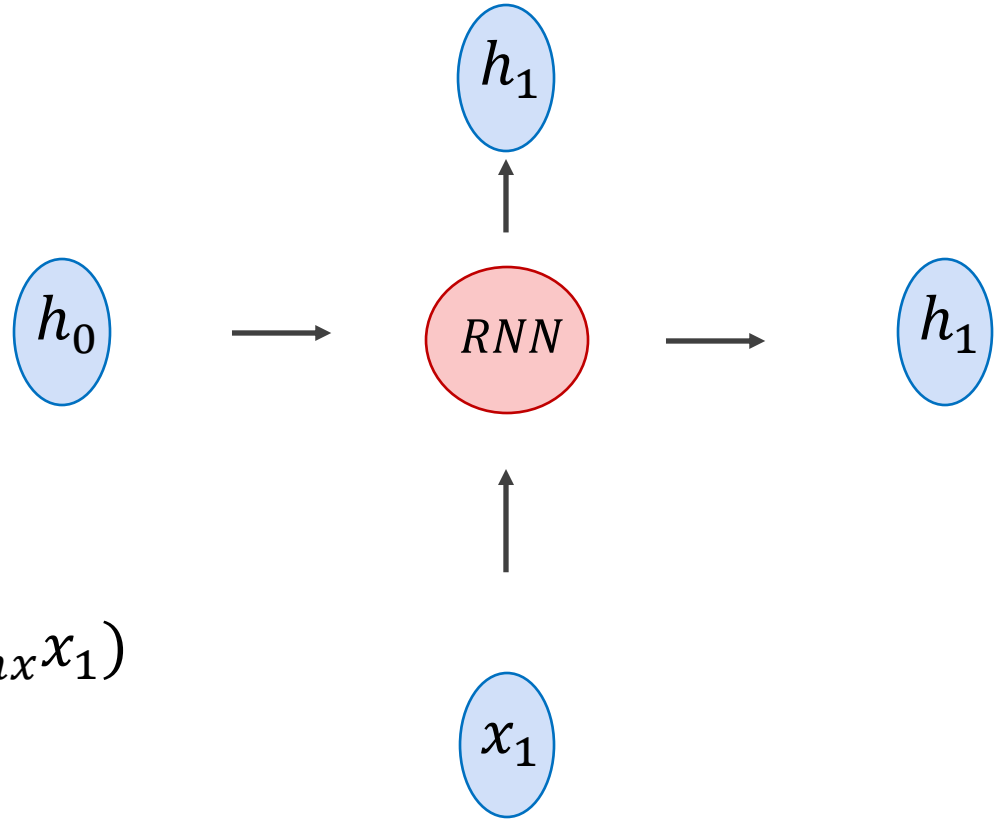
Recurrent Neural Network Cell



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

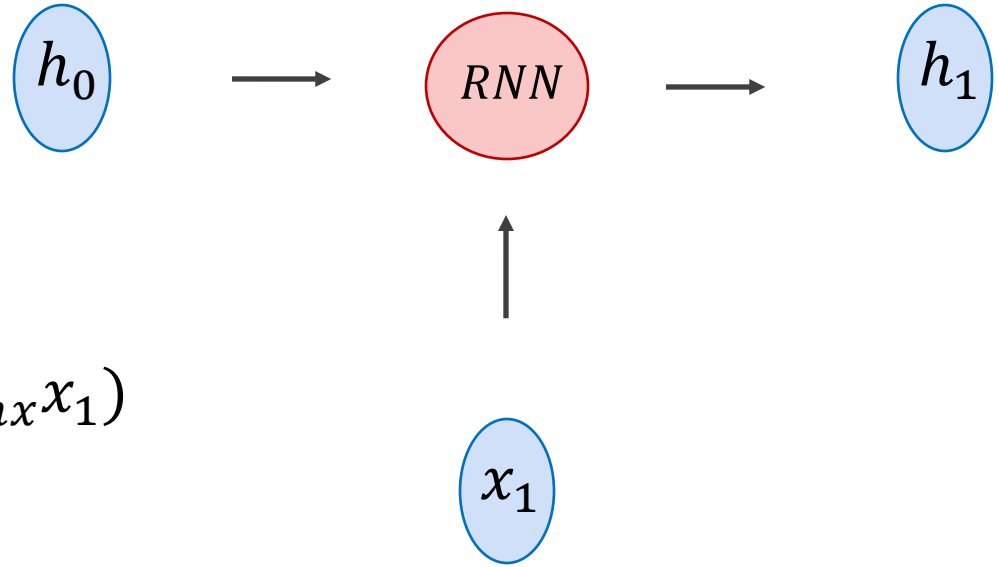
$$y_1 = \text{softmax}(W_{hy}h_1)$$

Recurrent Neural Network Cell



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

Recurrent Neural Network Cell



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

RNN

CLASS torch.nn.RNN(*self*, *input_size*, *hidden_size*, *num_layers=1*, *nonlinearity='tanh'*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *device=None*, *dtype=None*) [SOURCE]

Apply 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:

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

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 $\tanh$.

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. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'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*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

RNN

RNN

Inputs: input, h_0

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` 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.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for the input sequence batch. Defaults to zeros if not provided.

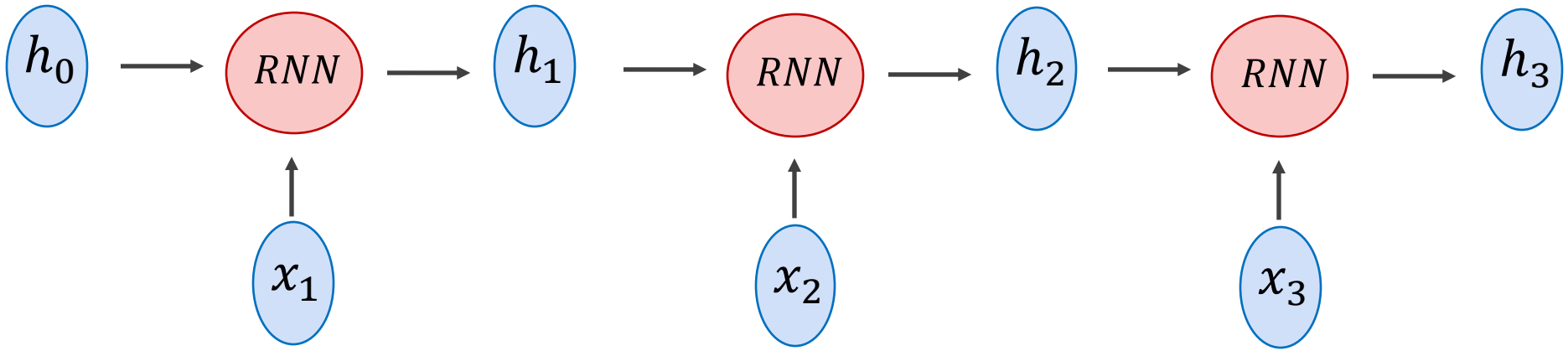
where:

$$
\begin{aligned}
N &= \text{batch size} \\
L &= \text{sequence length} \\
D &= 2 \text{ if bidirectional=True otherwise } 1 \\
H_{in} &= \text{input\_size} \\
H_{out} &= \text{hidden\_size}
\end{aligned}
$$

Outputs: output, h_n

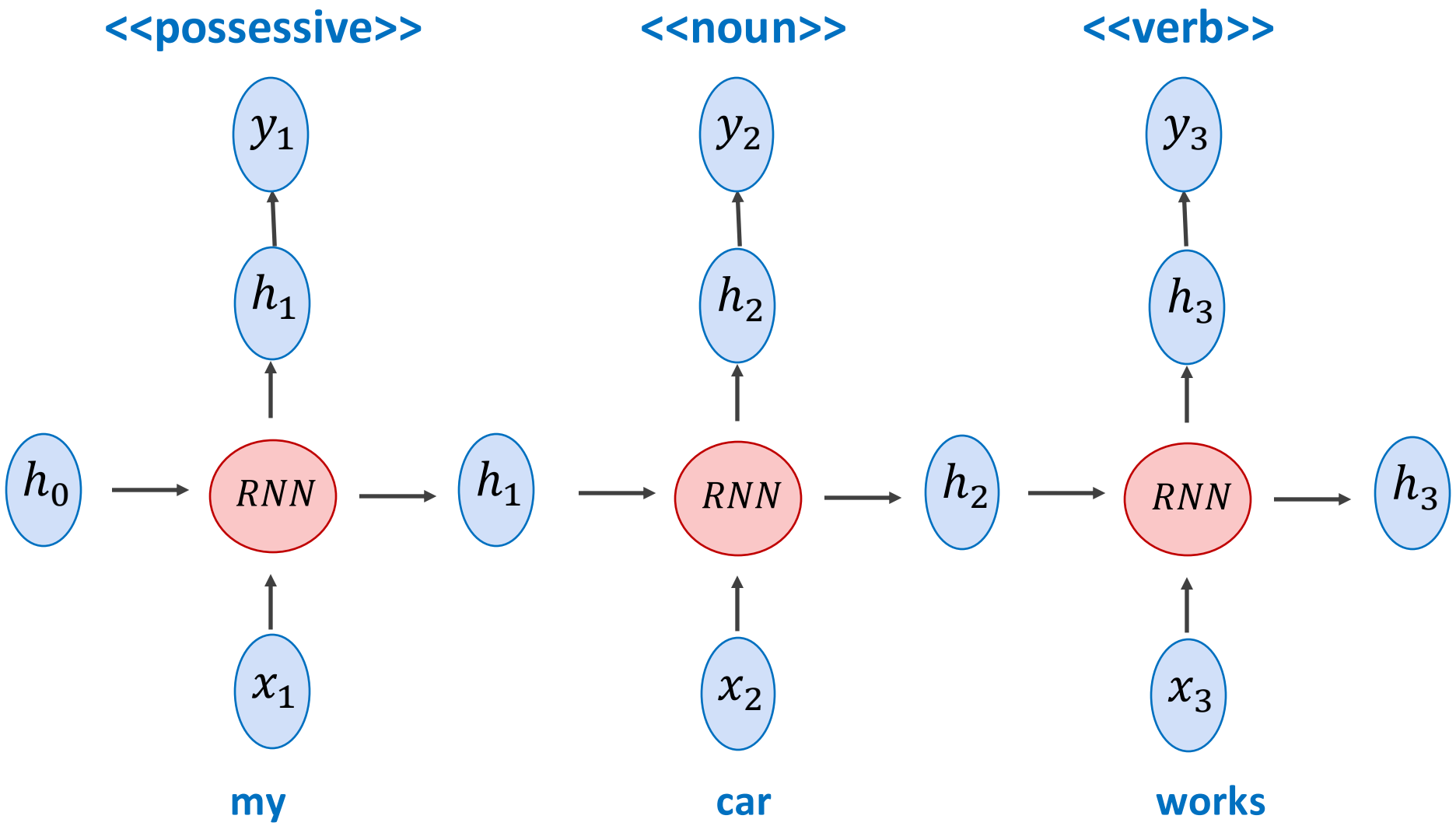
- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (*h_t*) from the last layer of the RNN, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.

(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

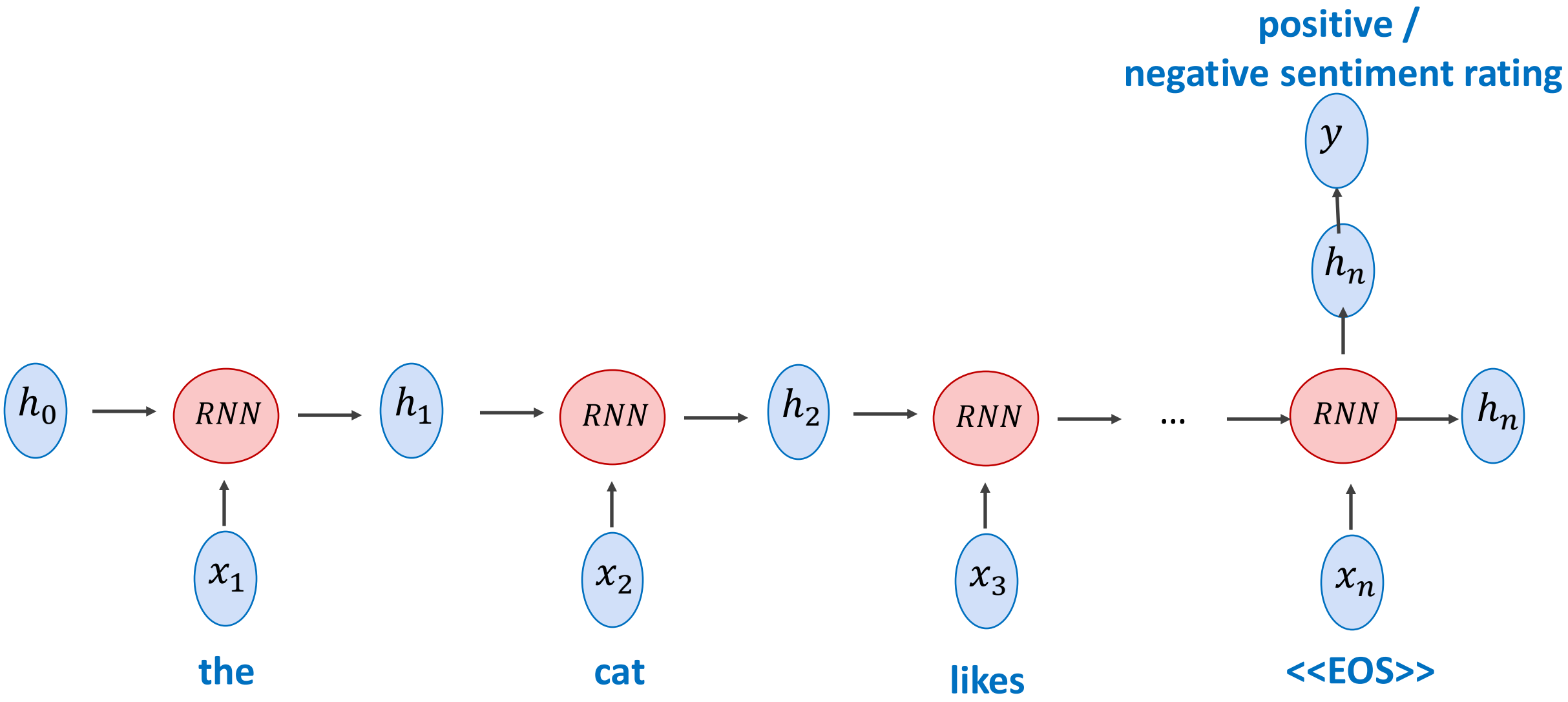
input	output
T: 1000 x 3	**T: 20 x 3**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 5	**T: 20 x 5**
T: 1000 x 6	**T: 20 x 6**

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. Scoring the Sentiment of a Text Sequence

Many-to-one Sequence to score problems



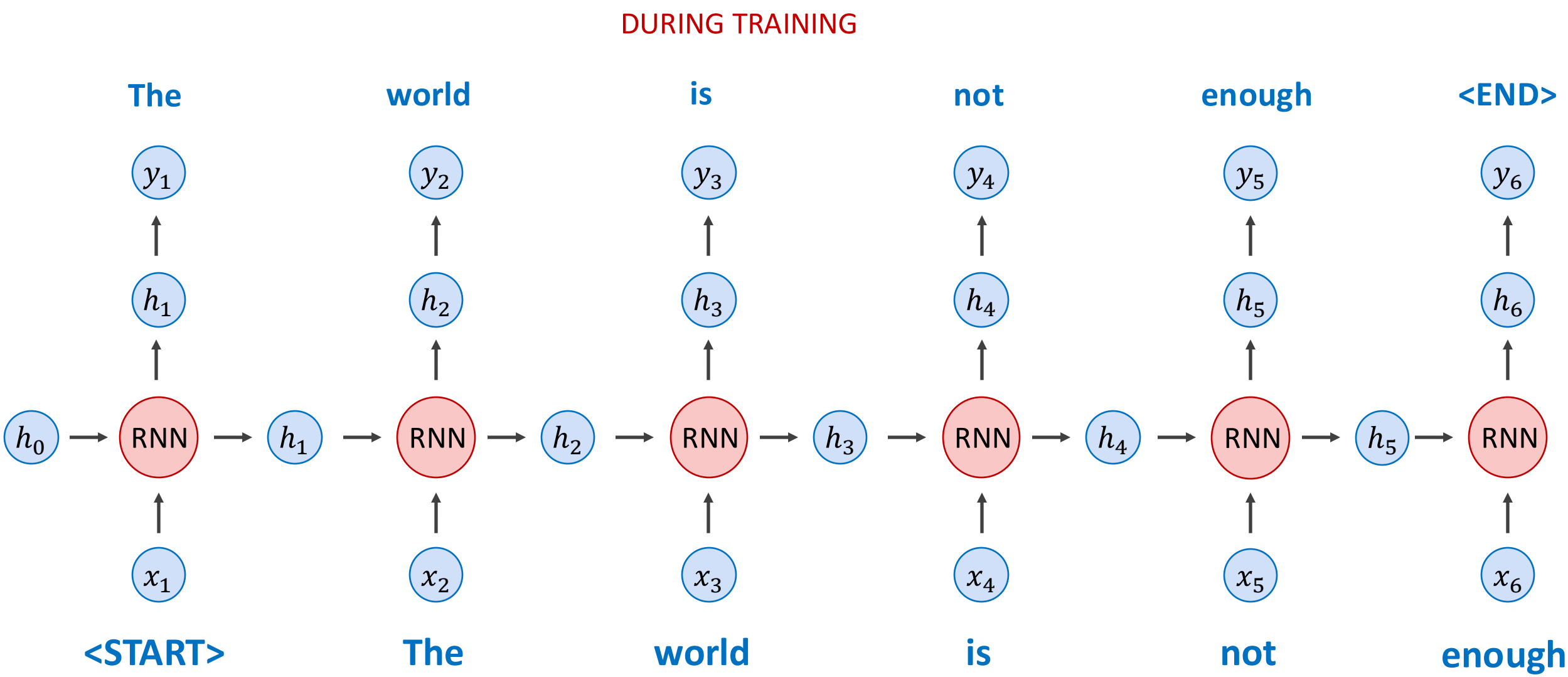
How can it be used? – e.g. Sentiment Scoring
Many to one Mapping Problems

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

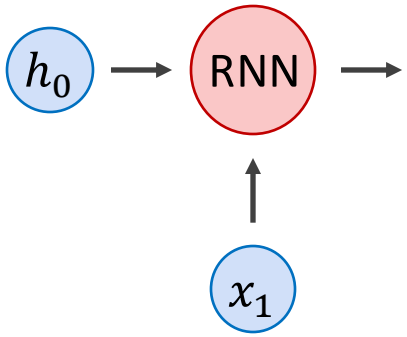
Input training examples don’t need to be the same length!
In this case outputs can be.

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

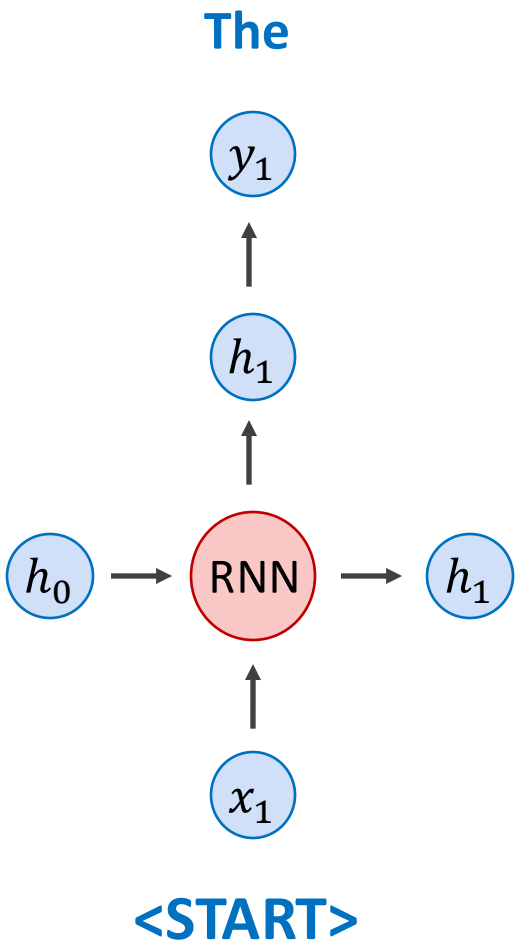
DURING TESTING



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

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

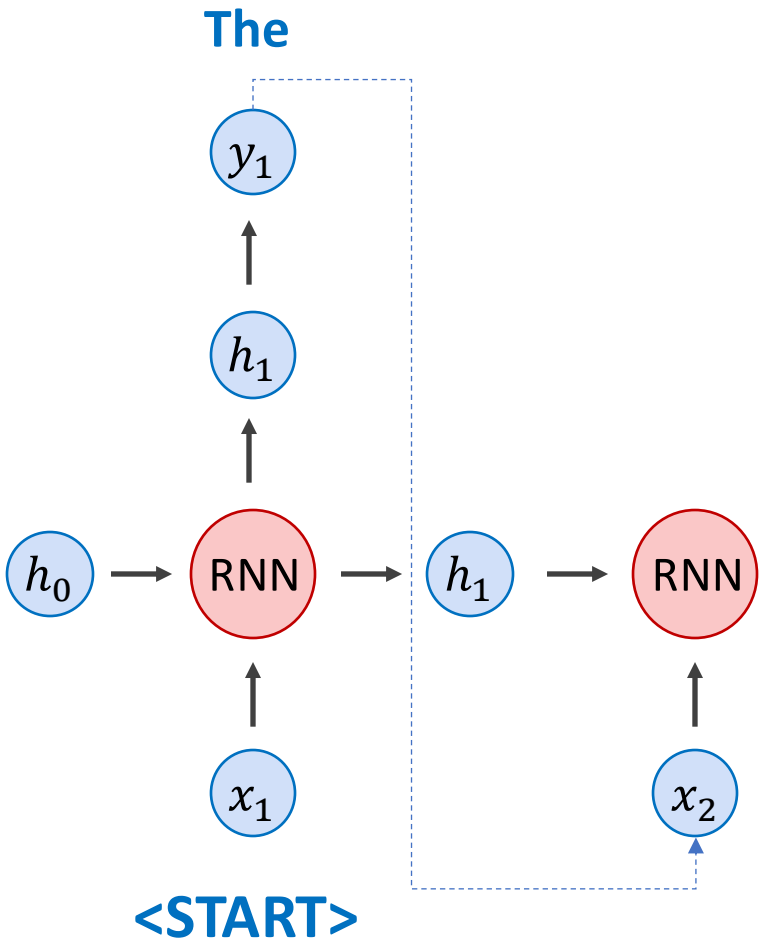
DURING TESTING



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

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

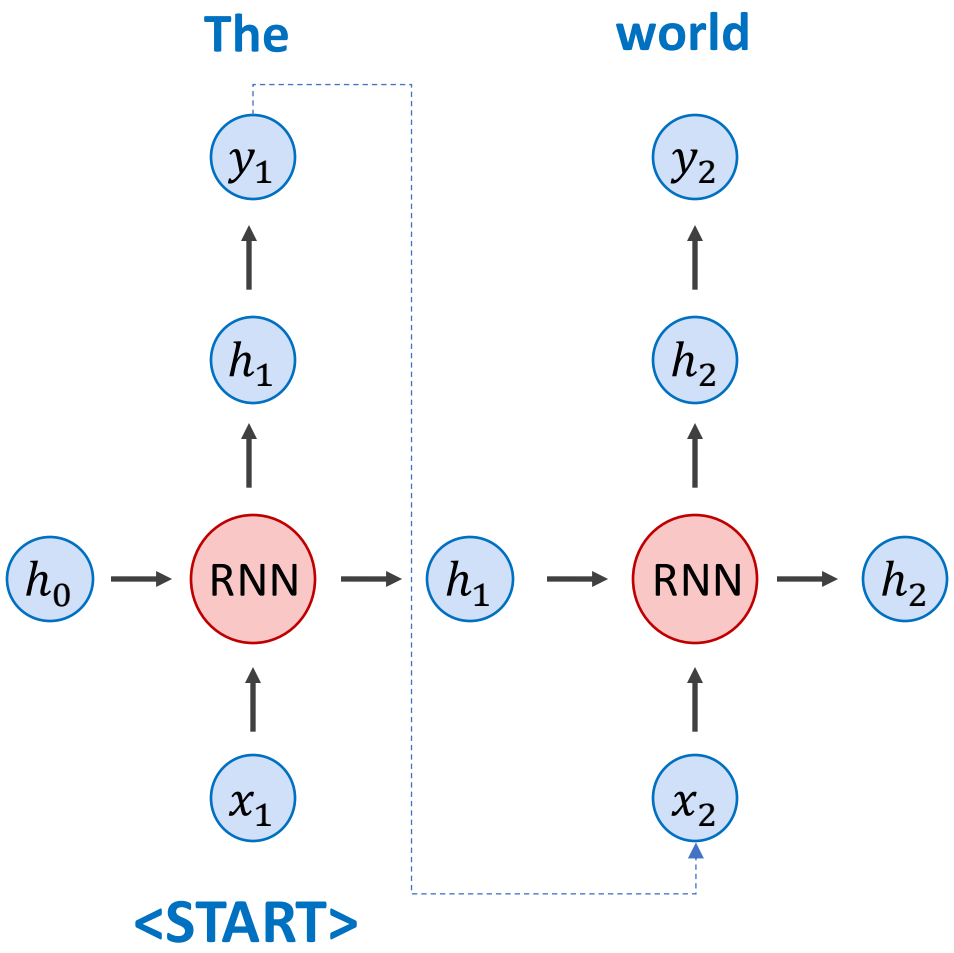
DURING TESTING



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

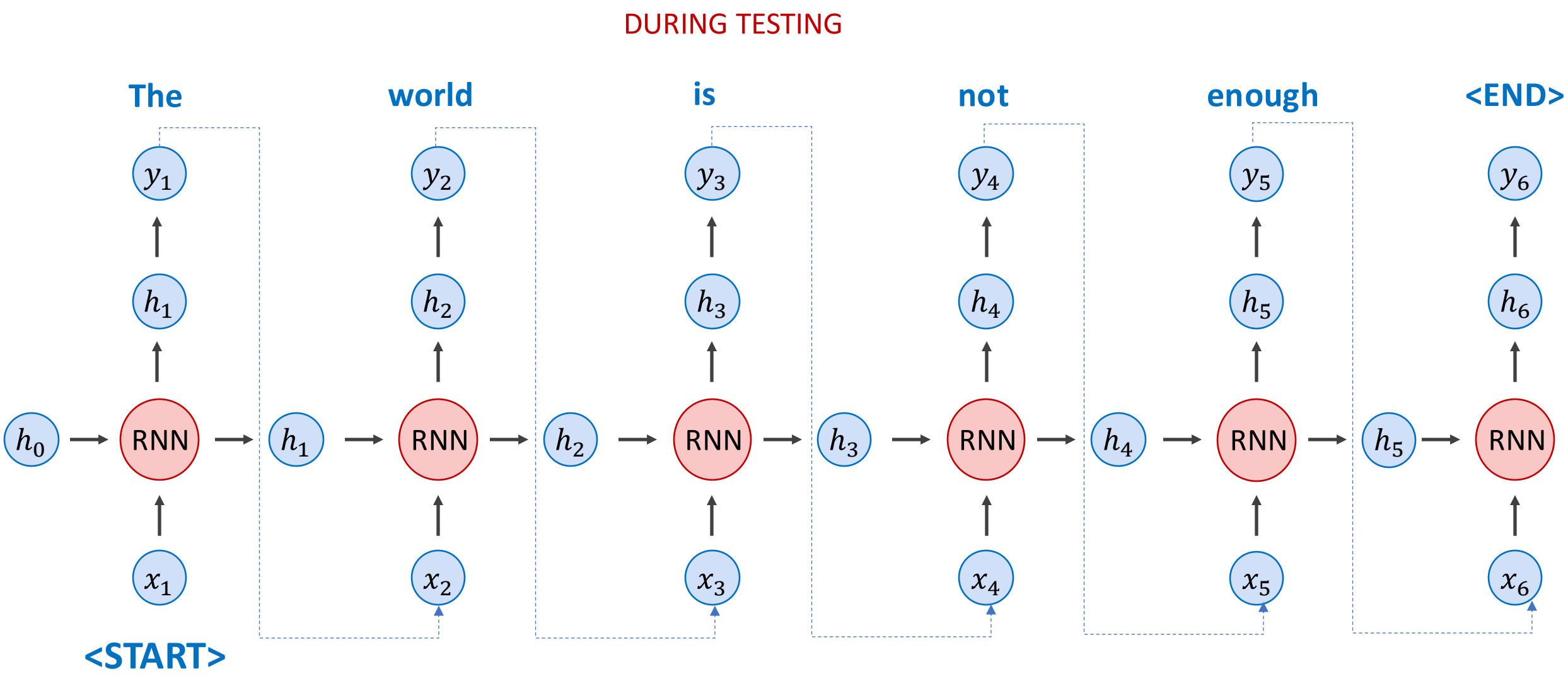
Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test

DURING TESTING



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

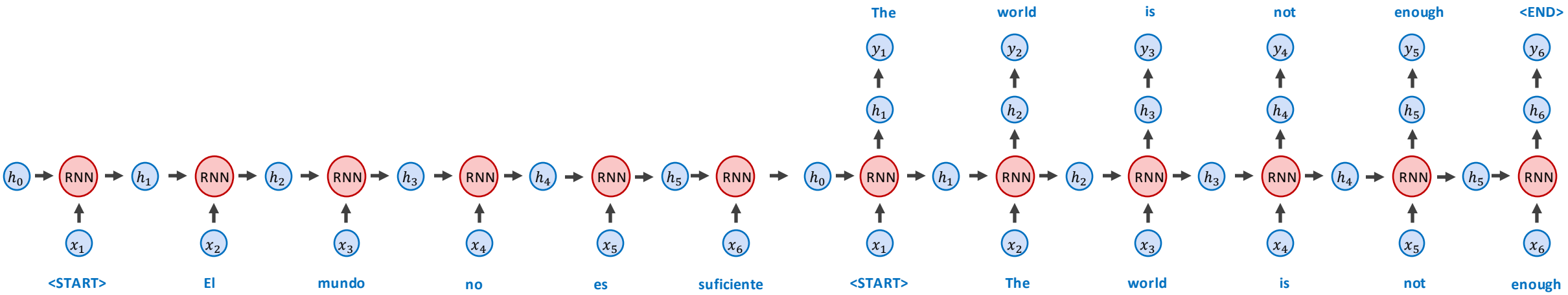
Auto-regressive model – Sequence to Sequence during Training, Auto-regressive during test



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

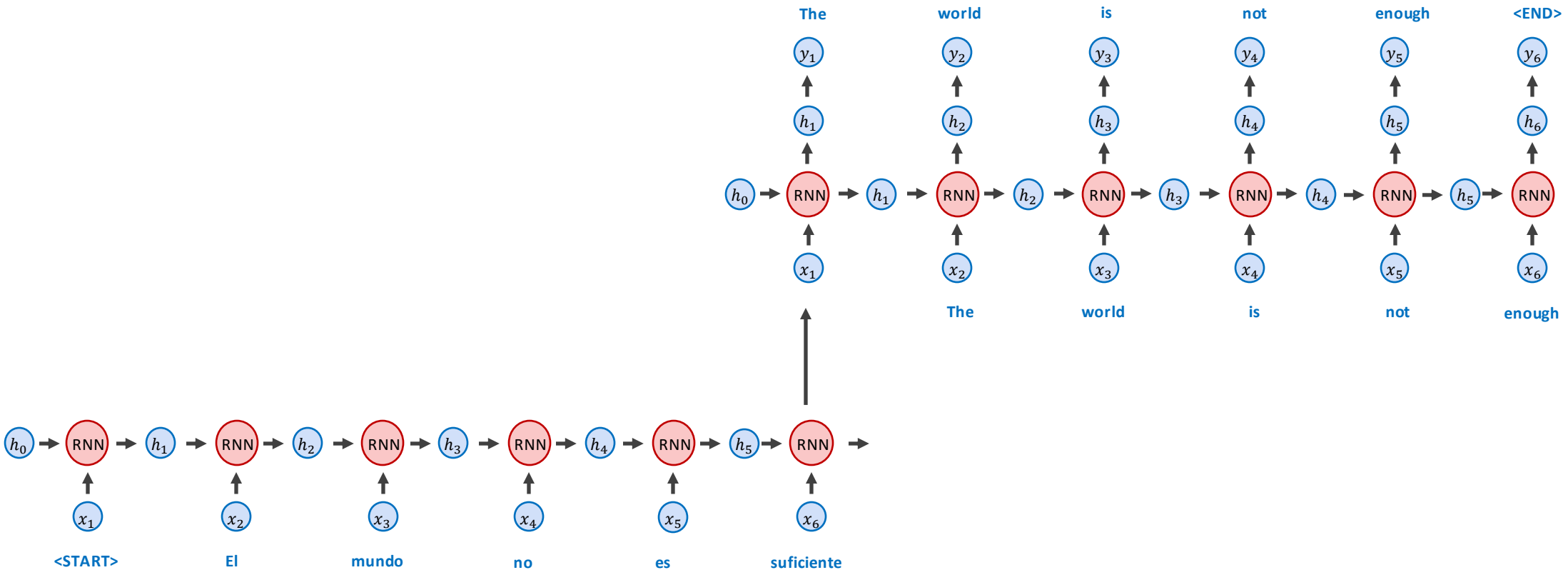
Input training examples don't need to be the same length!
In this case outputs can be.

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)



Problems

- Long Sequences lead to vanishing

- Hidden states can not carry information in a long sequence (Telephone Game problem)

Solutions Proposed

- Use another hidden state variable and experiment with more complex transition functions than $h = \tanh(W_1 h + W_2 x)$.
  - Read about LSTMs, GRUs, etc

LSTM Cell (Long Short-Term Memory)

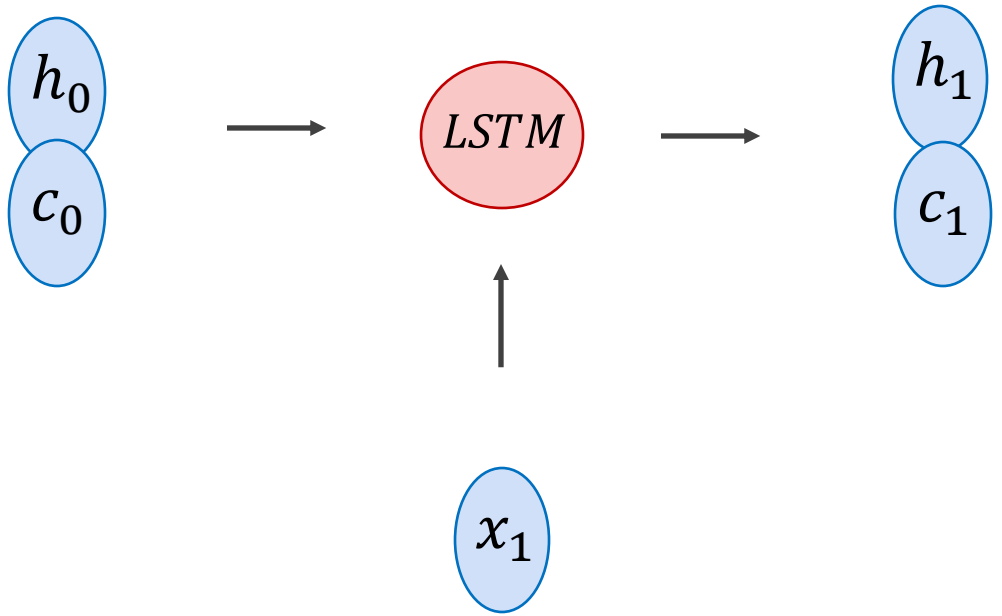
$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \quad (7)$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \quad (8)$$

$$c_t = f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \quad (9)$$

$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right) \quad (10)$$

$$h_t = o_t \tanh(c_t) \quad (11)$$

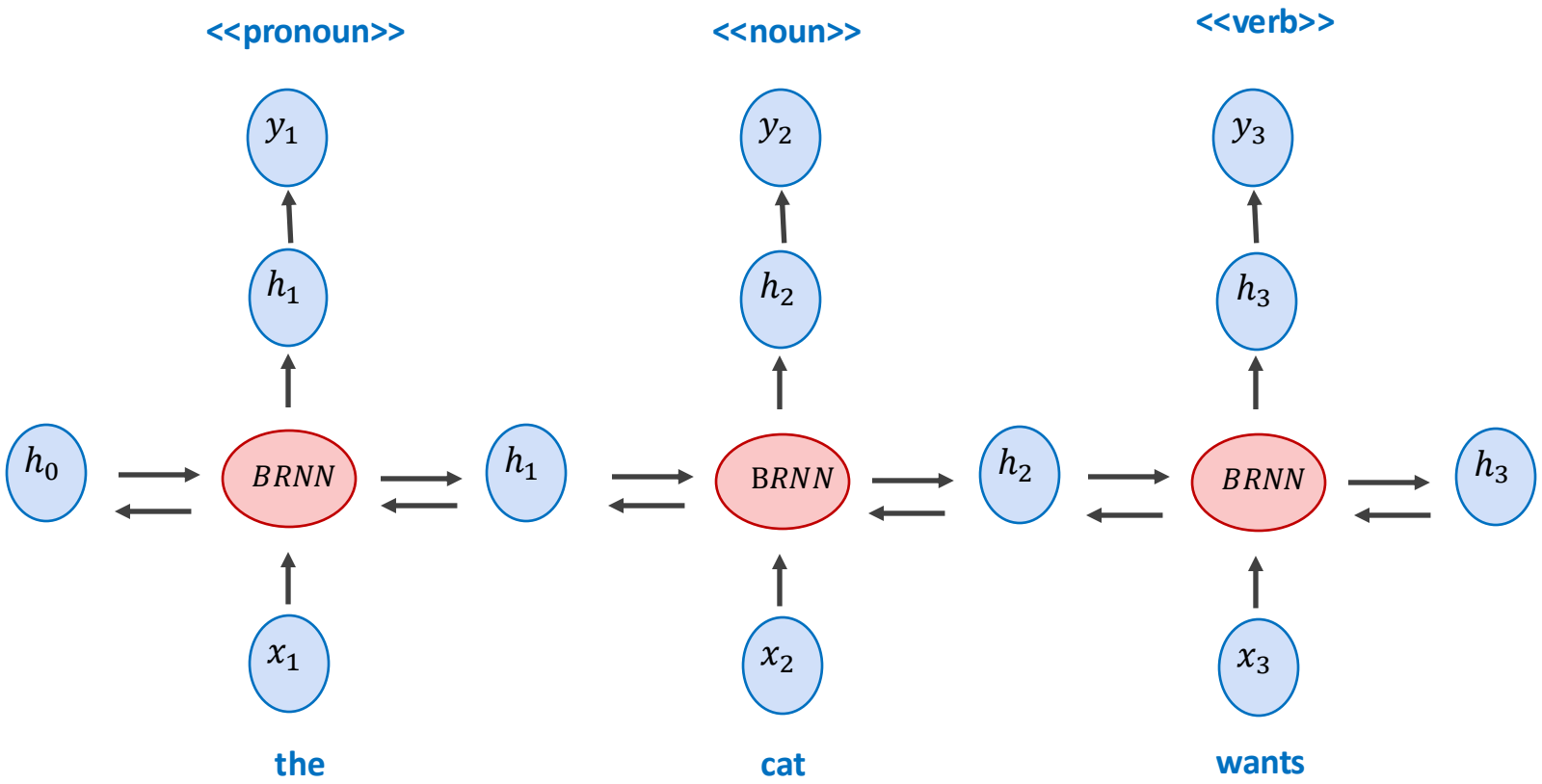


Solutions Proposed

- Use another hidden state variable and experiment with more complex transition functions than $h = \tanh(W_1 h + W_2 x)$.
  - Read about LSTMs, GRUs, etc

- Encode the sentences both from left-to-right and right-to-left using two RNNs and combine the final hidden states from each direction.
  - Read about Bidirectional RNNs (BiRNNs), BiLSTMs, BiGRUs

Bidirectional Recurrent Neural Network



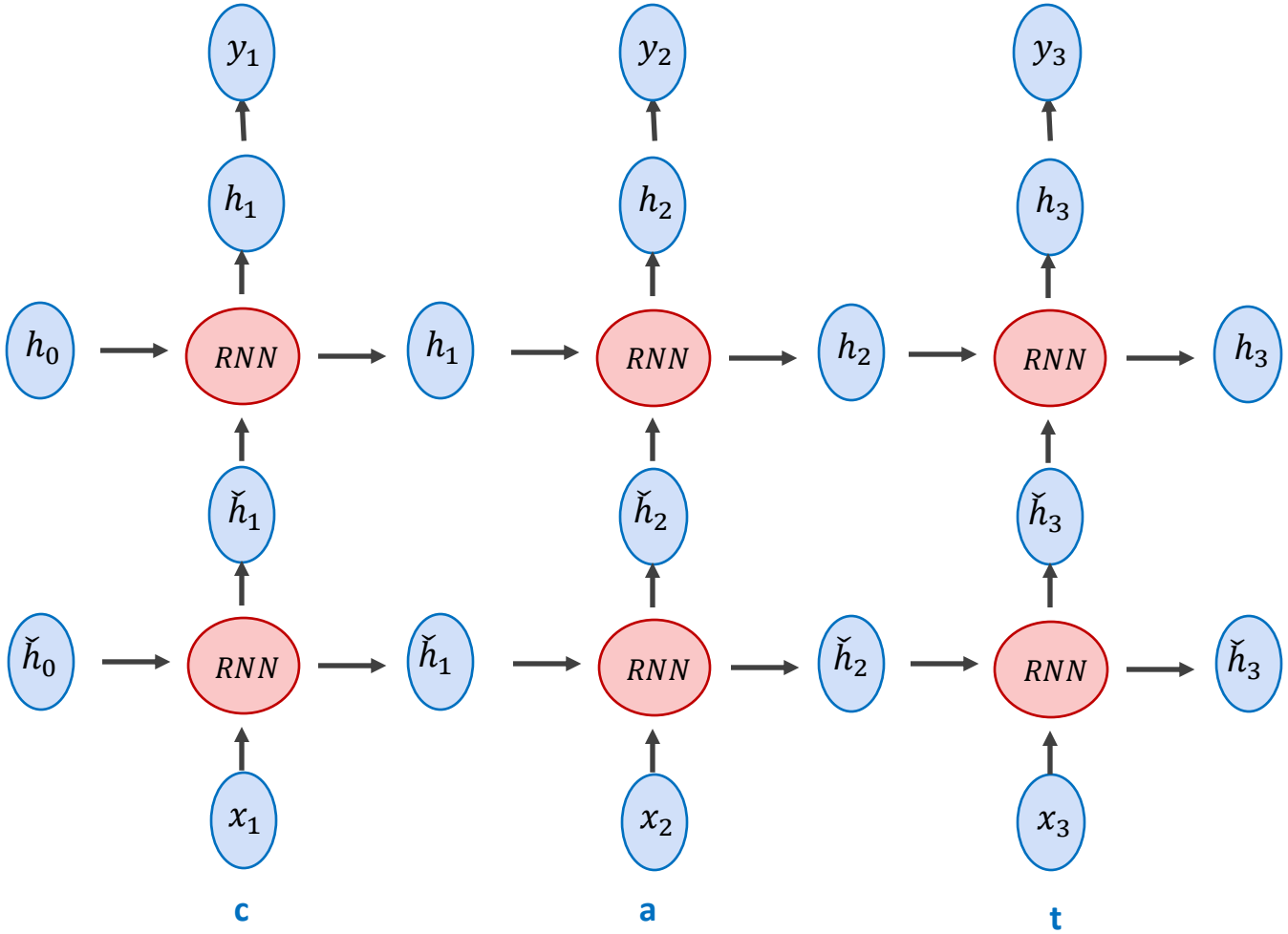
Solutions Proposed

- Use another hidden state variable and experiment with more complex transition functions than $h = \tanh(W_1 h + W_2 x)$.
  - Read about LSTMs, GRUs, etc

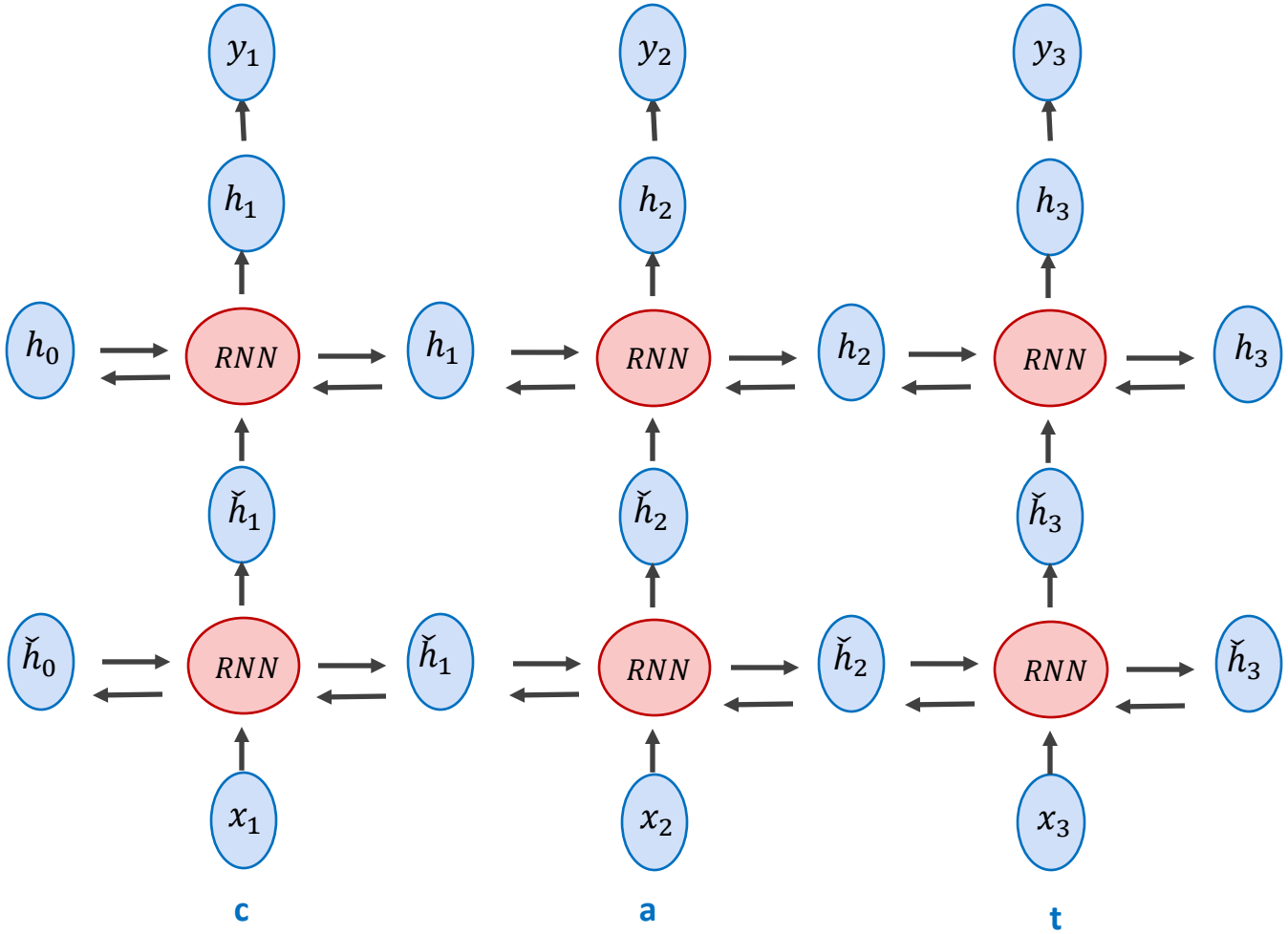
- Encode the sentences both from left-to-right and right-to-left using two RNNs and combine the final hidden states from each direction.
  - Read about Bidirectional RNNs (BiRNNs), BiLSTMs, BiGRUs

- Stack RNNs, use an RNN that feeds its output states to another RNN and this second RNN outputs the final output states.
  - Stacked RNNs, or Deep RNNs.

Stacked Recurrent Neural Network

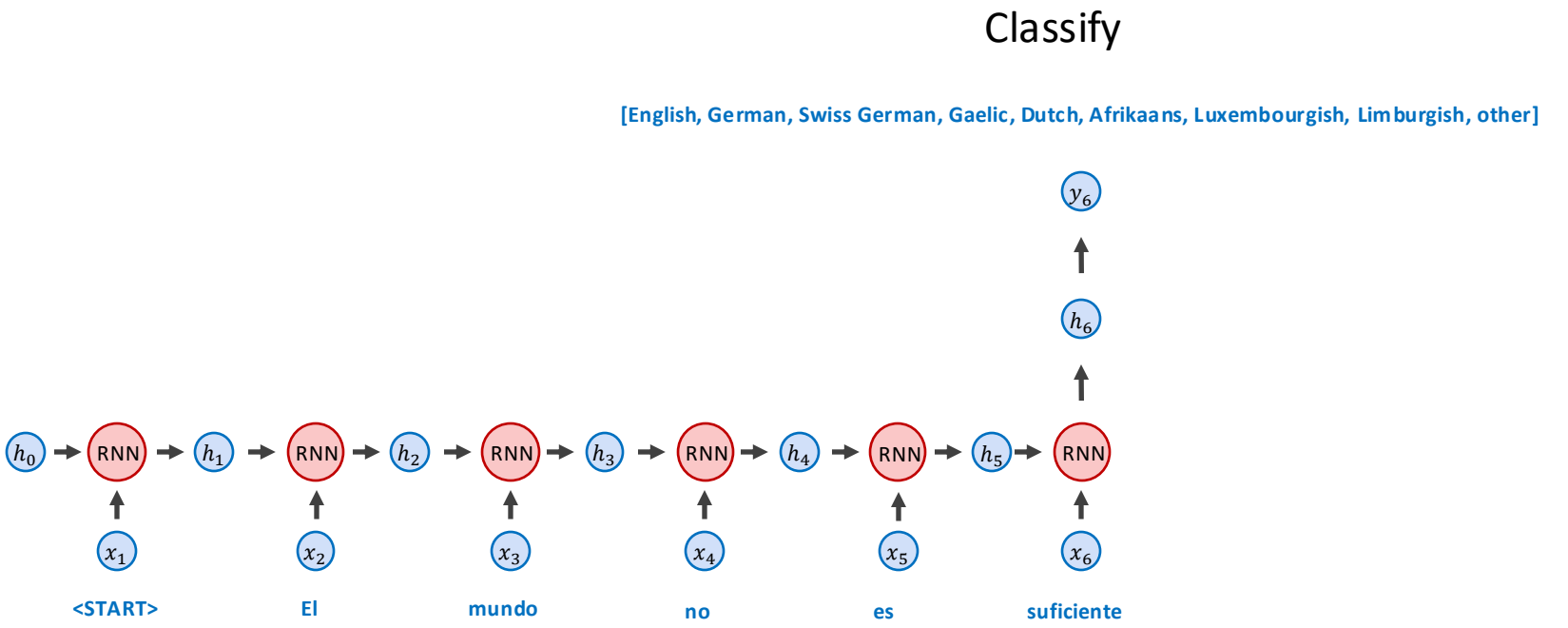


Stacked Bidirectional Recurrent Neural Network

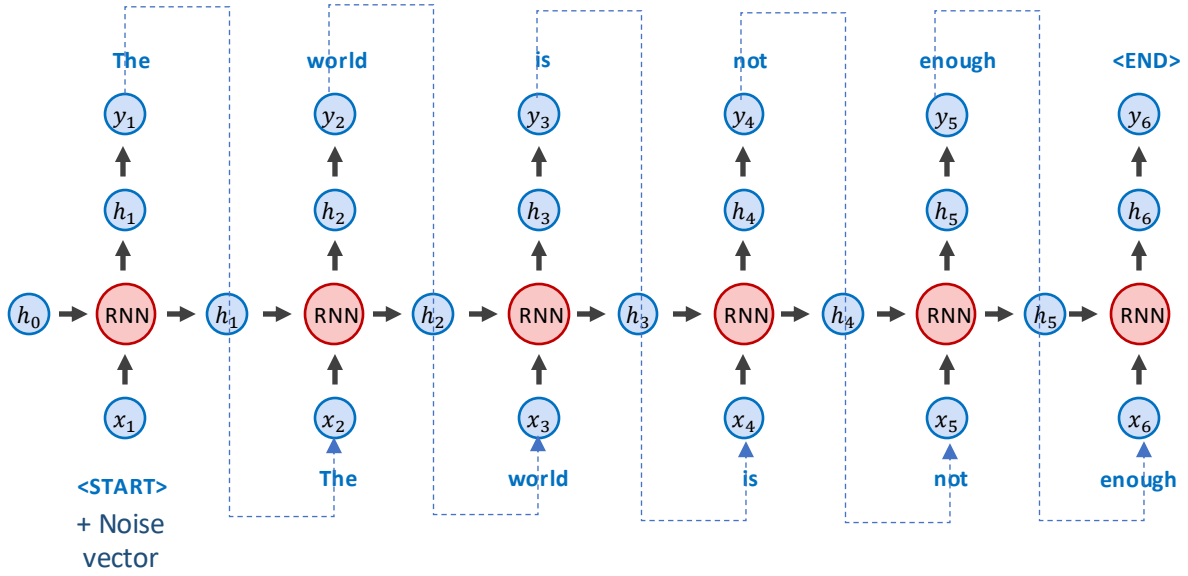


Best Solution: Learning Attention Weights

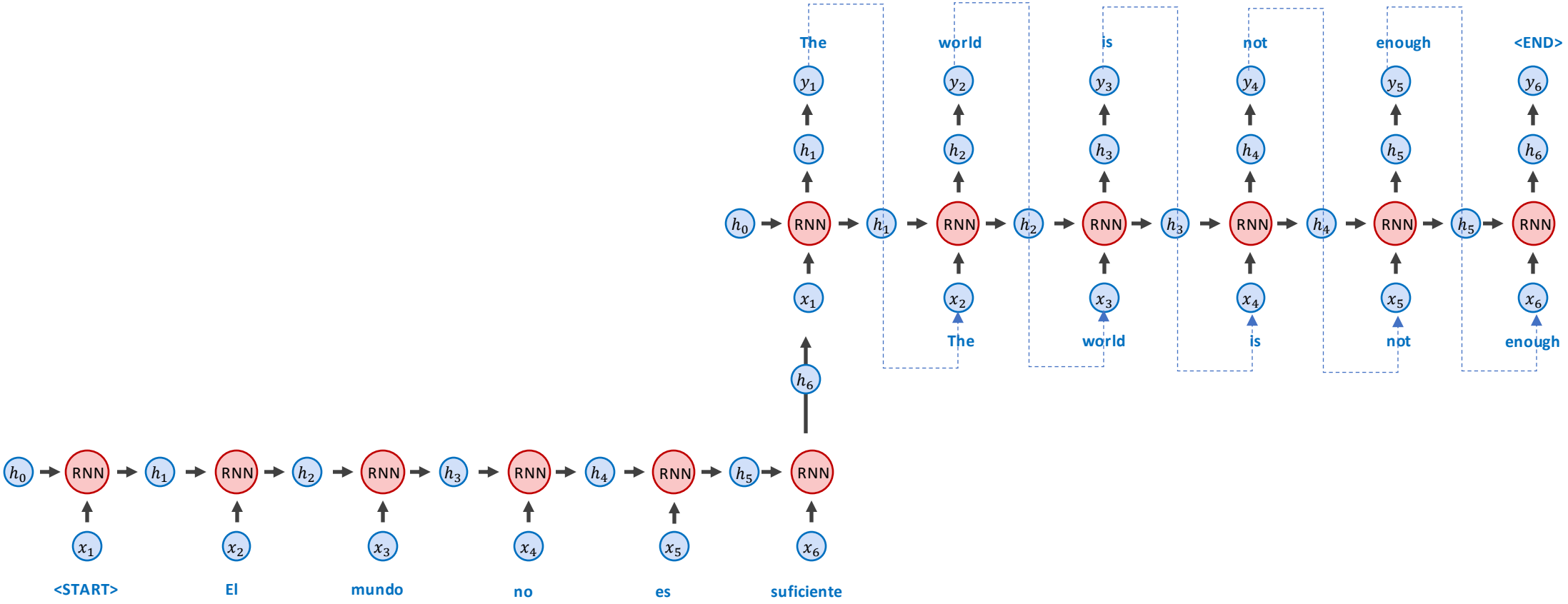
RNNs – Sequence to score prediction



RNNs for Text Generation (Auto-regressive)

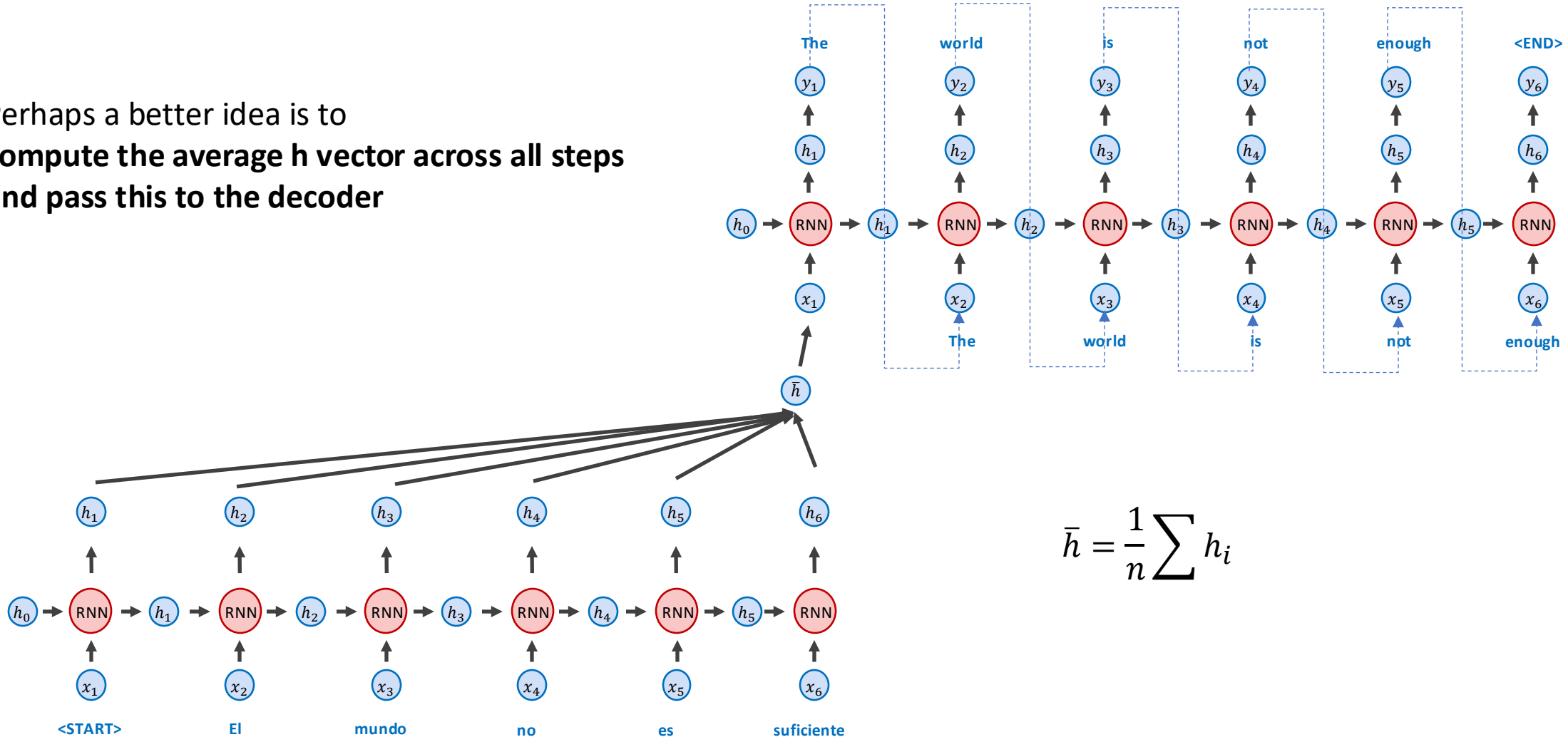


RNNs for Machine Translation Seq-to-Seq



RNNs for Machine Translation Seq-to-Seq

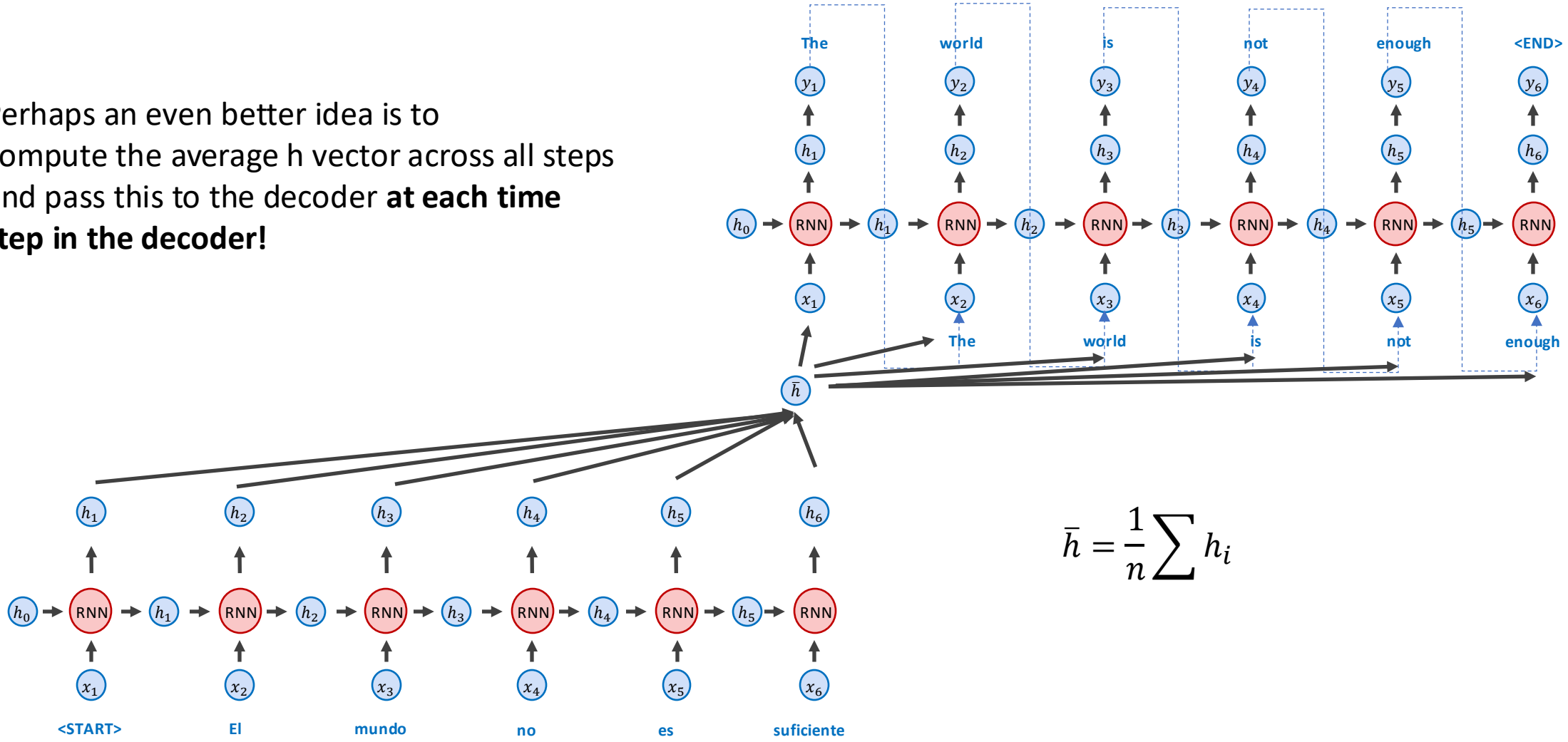
Perhaps a better idea is to
compute the average h vector across all steps and pass this to the decoder



$$\bar{h} = \frac{1}{n}\sum h_i$$

RNNs for Machine Translation Seq-to-Seq

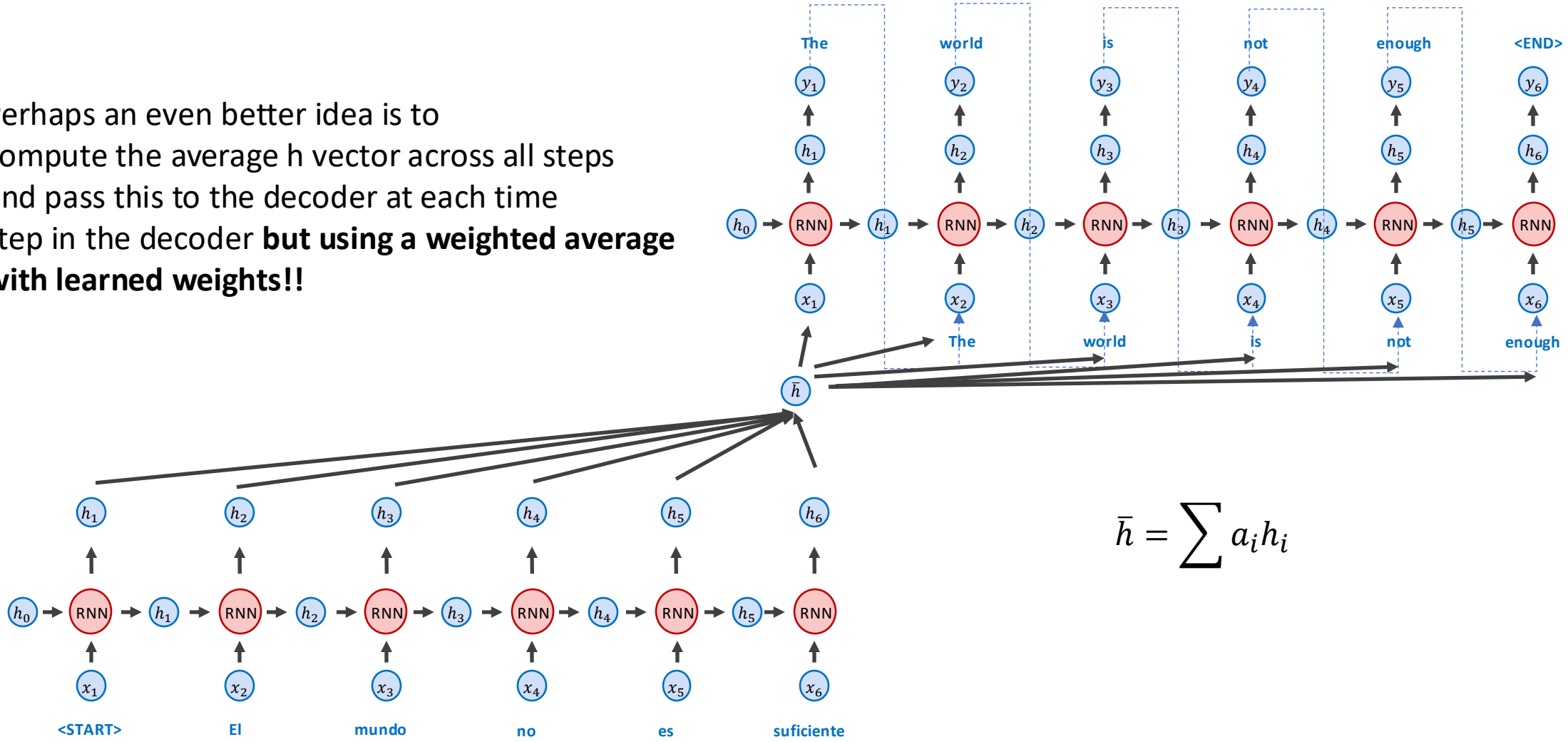
Perhaps an even better idea is to compute the average h vector across all steps and pass this to the decoder **at each time step in the decoder!**



$$\bar{h} = \frac{1}{n}\sum h_i$$

RNNs for Machine Translation Seq-to-Seq

Perhaps an even better idea is to compute the average h vector across all steps and pass this to the decoder at each time step in the decoder **but using a weighted average with learned weights!!**

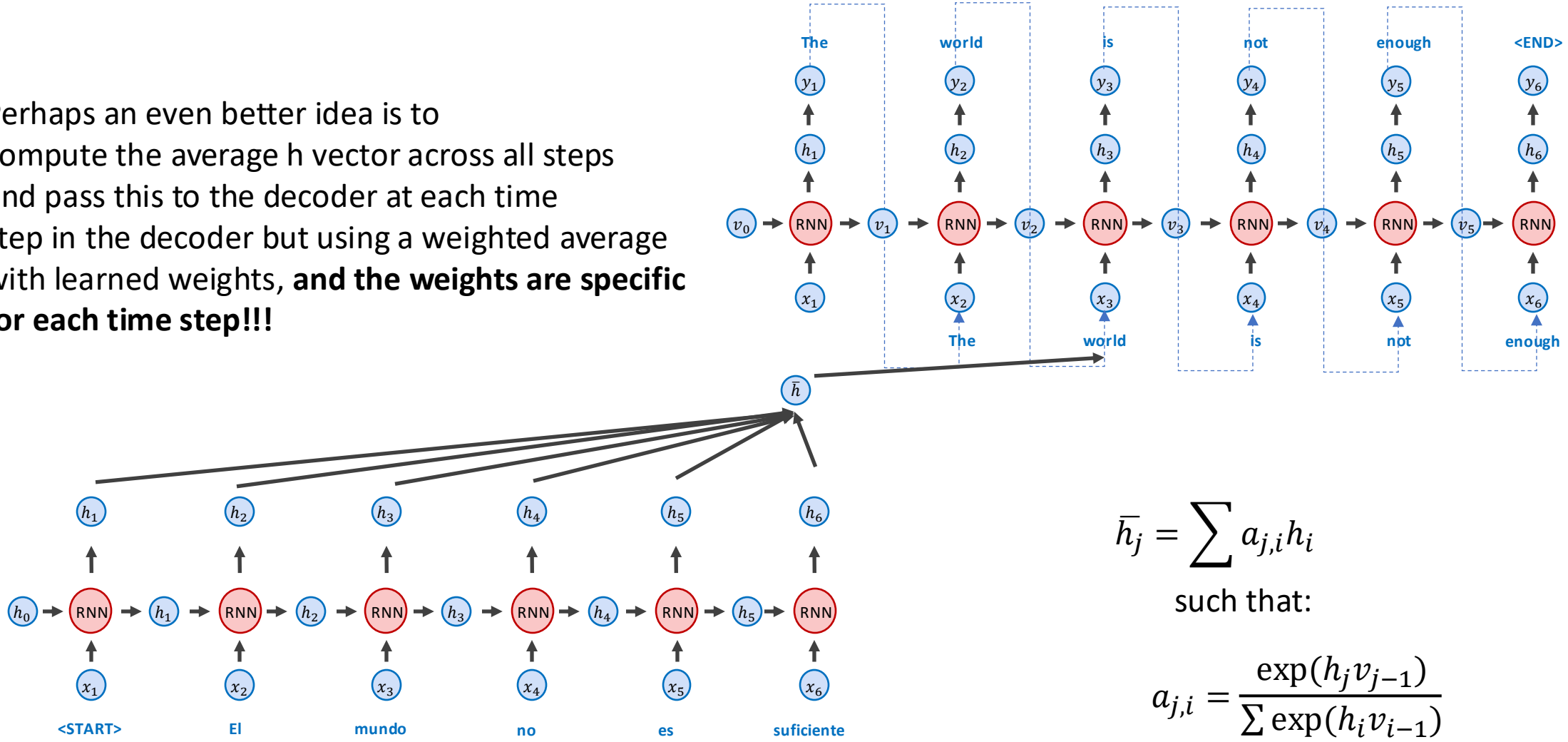


$$\bar{h} = \sum a_i h_i$$

RNNs for Machine Translation Seq-to-Seq

Only showing the third time step encoder-decoder connection

Perhaps an even better idea is to compute the average h vector across all steps and pass this to the decoder at each time step in the decoder but using a weighted average with learned weights, **and the weights are specific for each time step!!!**



$$\bar{h}_j = \sum a_{j,i} h_i$$

such that:

$$a_{j,i} = \frac{\exp(h_j v_{j-1})}{\sum \exp(h_i v_{i-1})}$$

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

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

Let's take a look at one of the first papers introducing this idea.

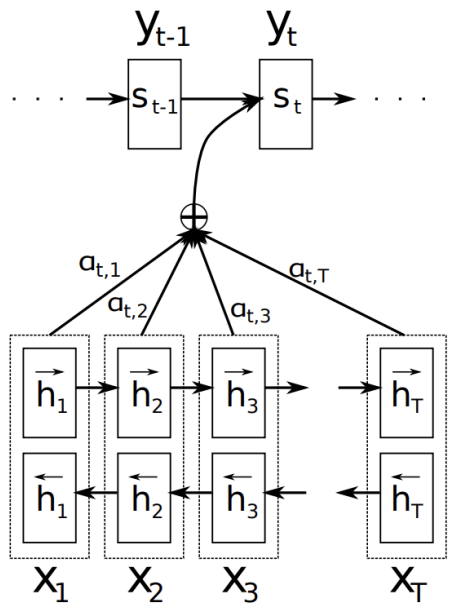


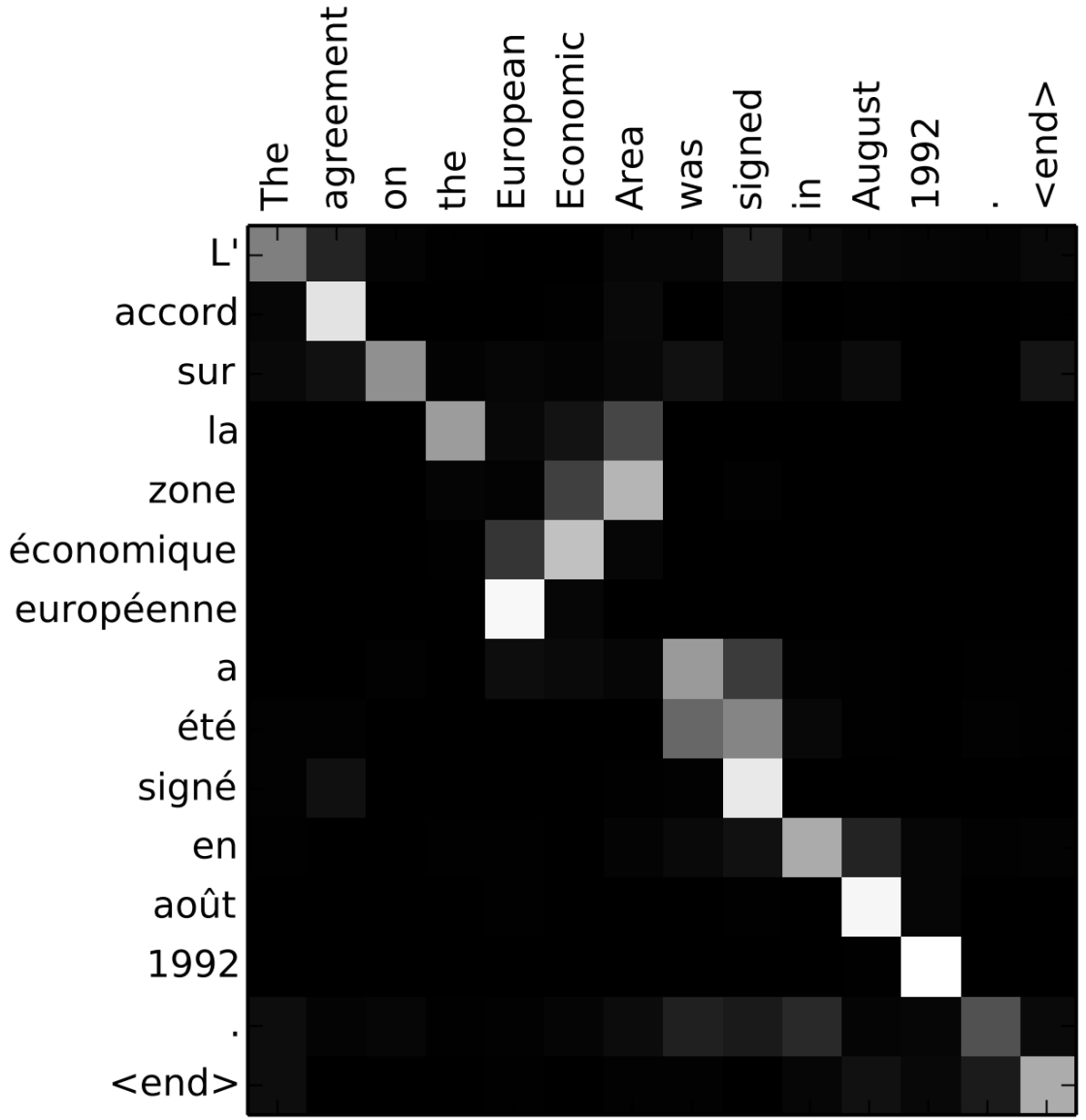
Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

$$e_{ij} = a(s_{i-1}, h_j)$$

Let’s look at the Attention weights



Transformers: Attention is All You Need

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Questions?