

Recurrent Neural Nets: Intro

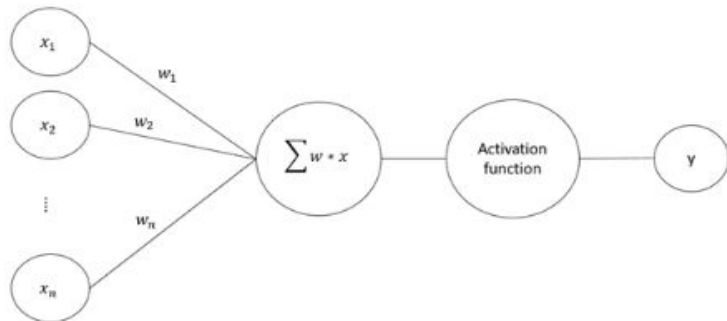
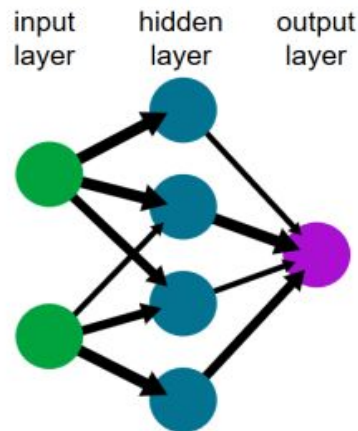
Michael(Mike) Erlihson

- Neural Nets: Intro
- What is Recurrent Neural Network (RNN)
- Types of tasks RNN is used to tackle
- Basic RNN Architecture
- More advanced RNN architectures
- Attention Layer in RNN
- RNN in Python (Keras)

Neural Network: Recap

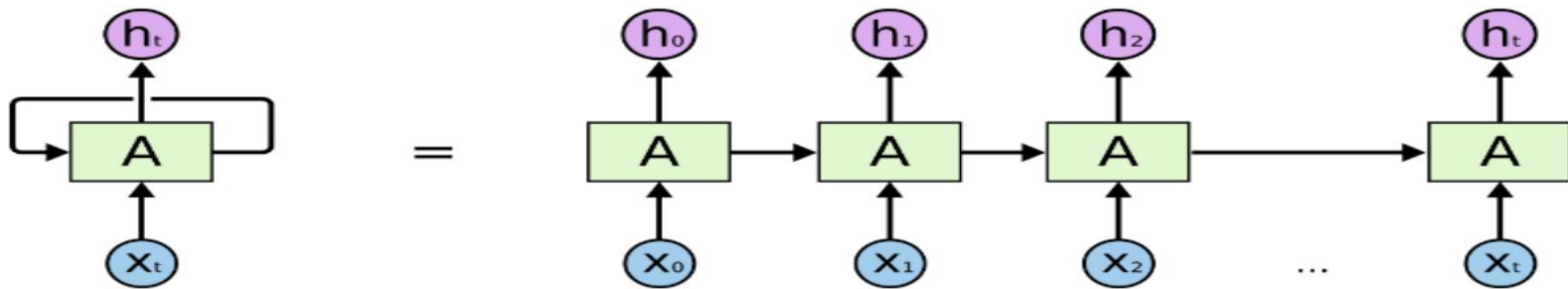


- Neural Network is a directed graph data structure
- Neural network consists of weights and nodes
- Hidden layers and output layer store a calculated value based on weights and previous layer node values
- Weights and value calculations can be represented as matrices
- Activation function is applied on the sum of weights*input values



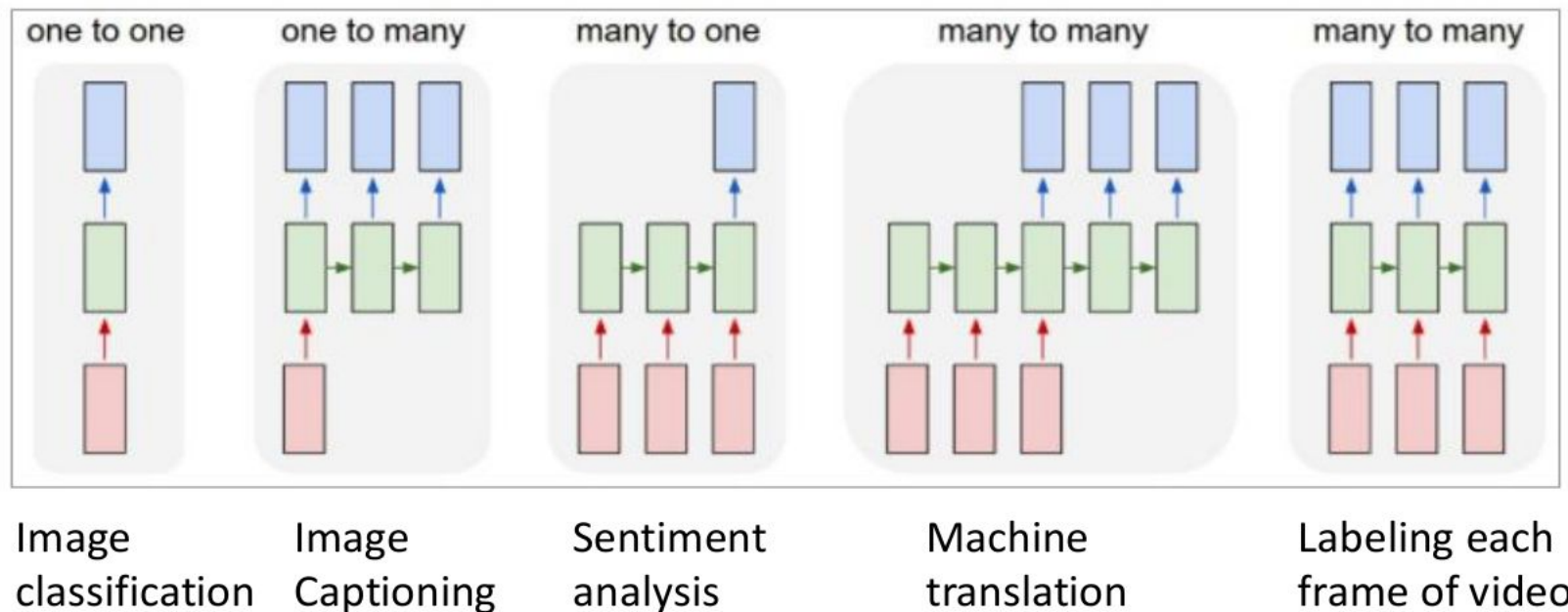
What is RNN?

- RNN is designated to tackle sequential data
 - Predict next word in a sentence, next vowel in speech, next frame in video...
- RNNs are called **recurrent** because they:
 - Perform the same task for every element of a sequence, with the output being depended on the previous computations
 - Have a “memory” capturing information about what has been calculated so far



Motivation behind CNN: Why bother?

- RNNs allow us to operate over **sequences** of vectors: **sequences** in the input, the output, or in the most general case **both**



RNN: Vast variety of possible task

< itc >

Examples of sequence data

Speech recognition



"The quick brown fox jumped over the lazy dog."

Music generation

\emptyset



Sentiment classification

"There is nothing to like in this movie."



DNA sequence analysis

AGCCCCTGTGAGGAACTAG



AGCCCCTGTGAGGAACTAG

Machine translation

Voulez-vous chanter avec moi?



Do you want to sing with me?

Video activity recognition



Running

Name entity recognition

Yesterday, Harry Potter met Hermione Granger.



Yesterday, **Harry Potter** met **Hermione Granger**.

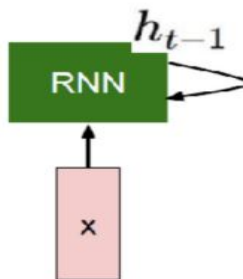
RNN: Basic Unit

< itc >

With "Recurrent" cell there is a recurrent activity and hidden state that changes in "time"

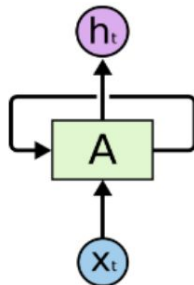
The state is updated for every input received

Input x

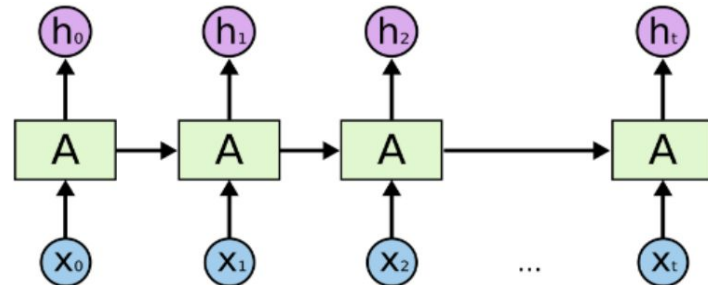


"Compressed RNN scheme"

"Unrolled RNN"



=



RNN: Hidden State

Recurrent Neural Network

$$h_t = f_W(h_{t-1}, x_t)$$

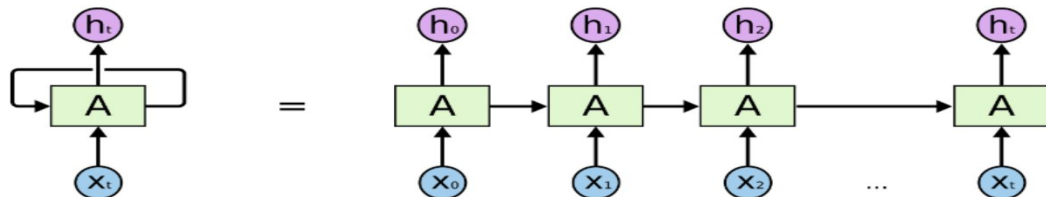
Previous hidden state

Input

New hidden state

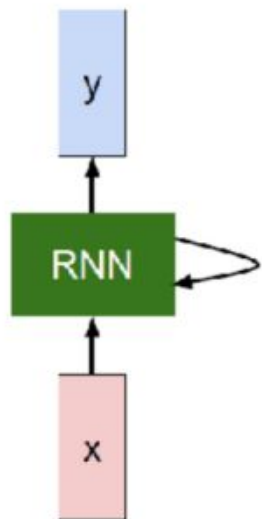
Sometimes a sequence
in a vector

Activation function
with parameters - W



(Vanilla) Recurrent Neural Network

The state consists of a single “hidden” vector h :



$$h_t = f_W(h_{t-1}, x_t)$$

In training process we are calculating the different weights

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

$$y_t = W_{hy}h_t$$

Weights for the output

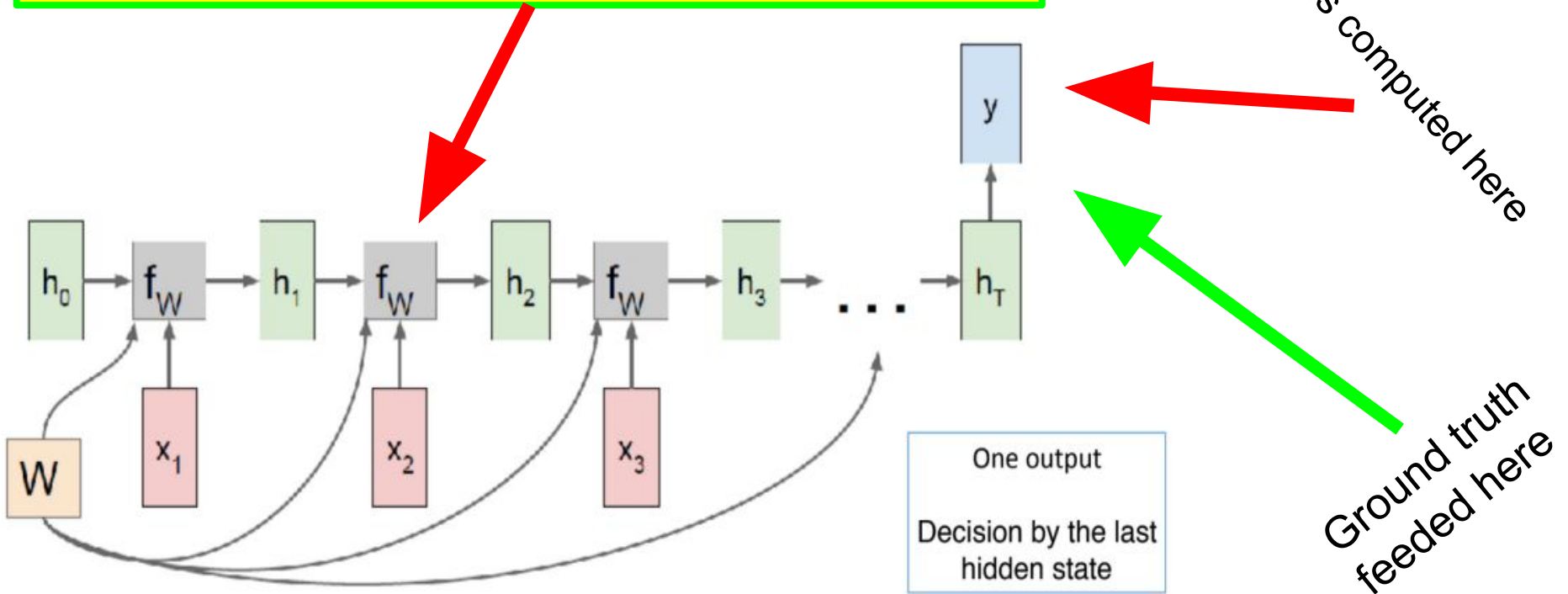
Weights for updating the new hidden state

Weights for every new input

RNN: Training Procedure (Many To One)

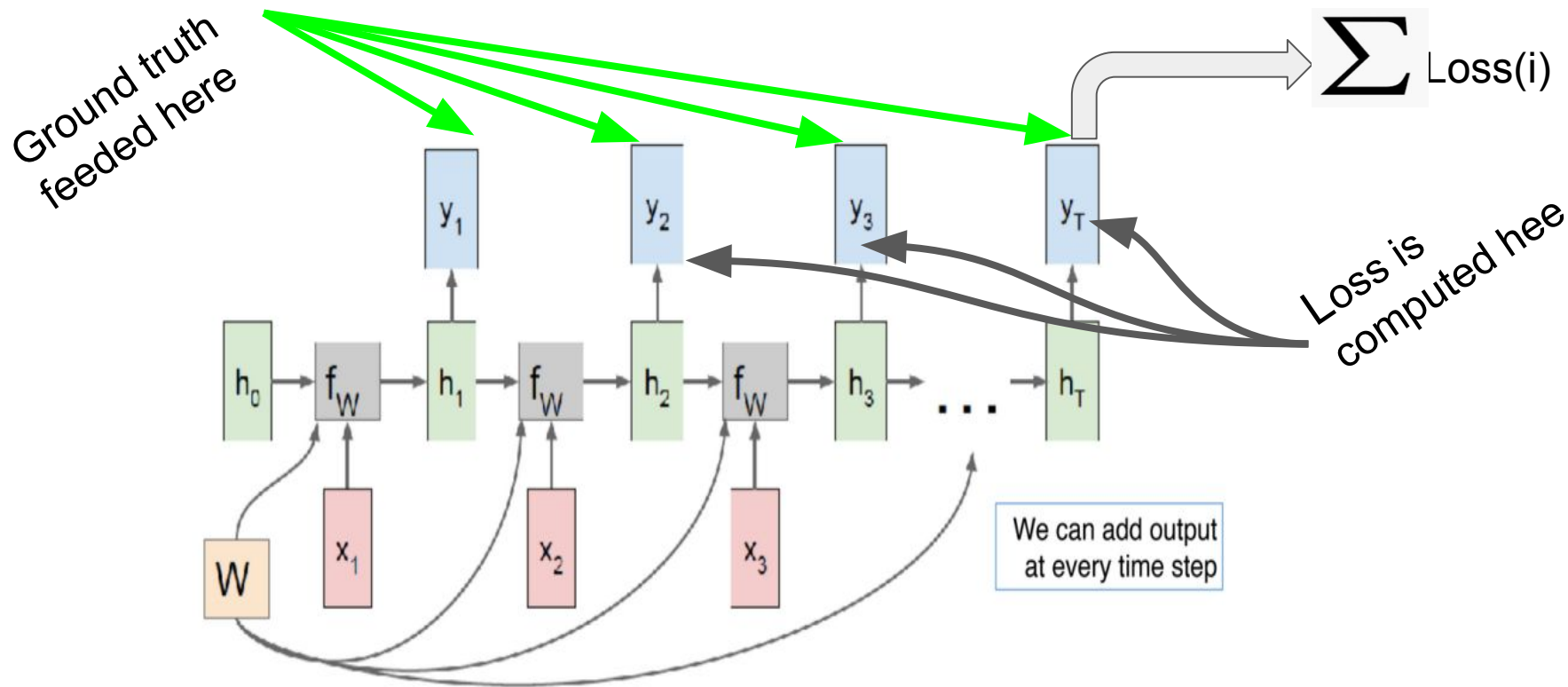
< itc >

The same weight matrices are used in ALL cells



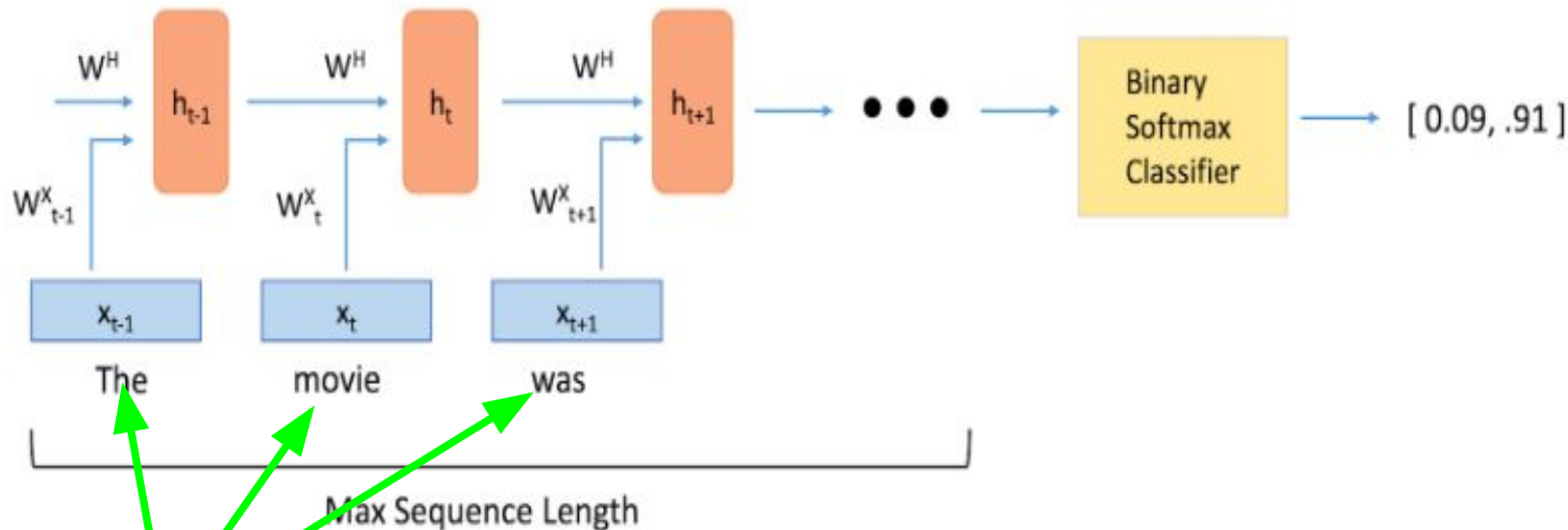
RNN: Training Procedure (Many To Many)

< itc >



RNN Application: Sentiment Analysis

< itc >

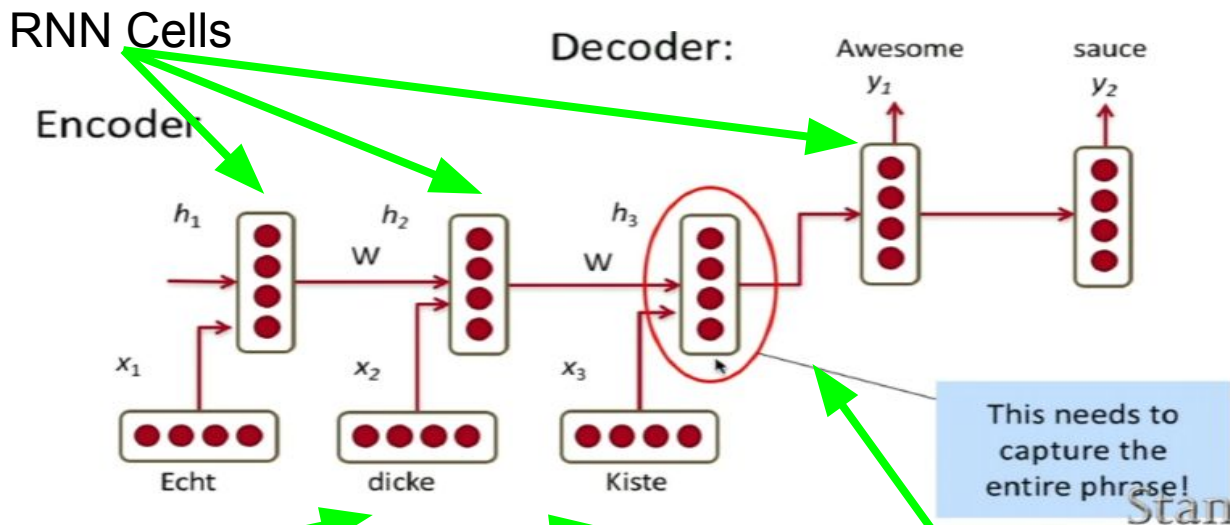


Word Embeddings

RNN Application: Machine translation



Main Task: Predict next word



Word Embeddings

Phrase numerical representation: vector

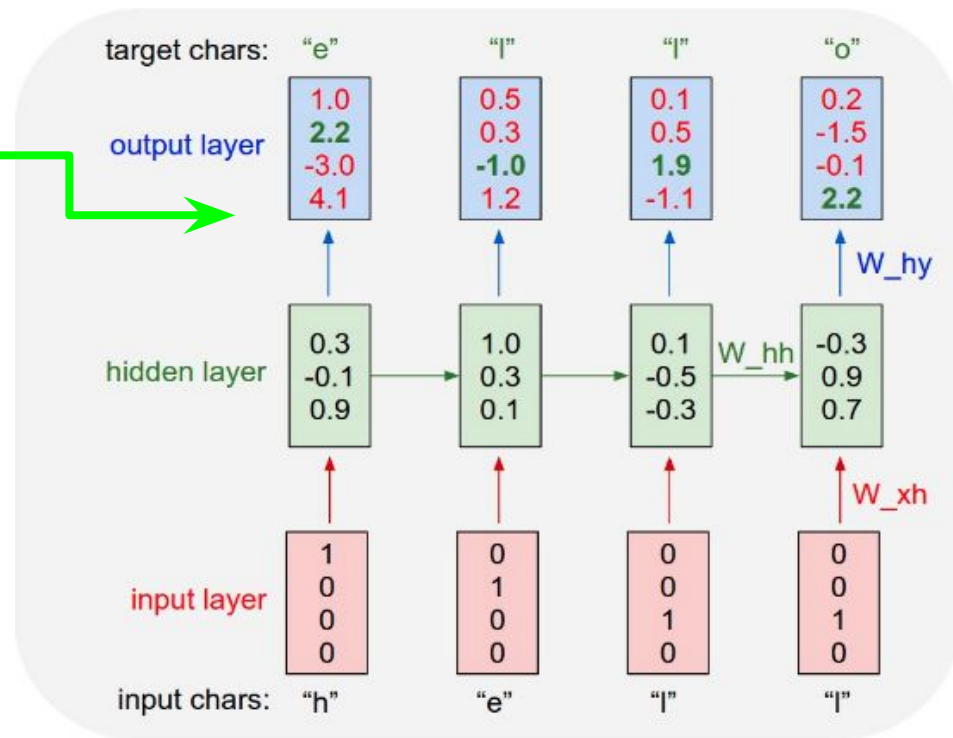
RNN: Character-level Language Model

< itc >

**Tries to predict the next
chars in the sequence**

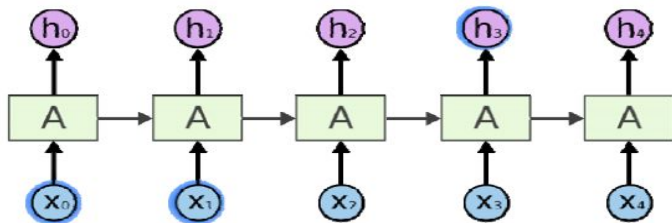
Vocabulary = [h, e, l, o]

Example training sequence:
"hello"

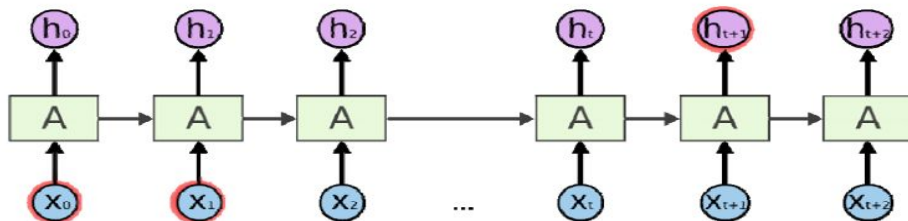


Limitations of Vanilla RNN

- Vanilla RNN works well for a small time step (remembers two-three previous hidden states)
- However, the sensitivity of the input values decays over time in a standard RNN



"the clouds are in the sky"

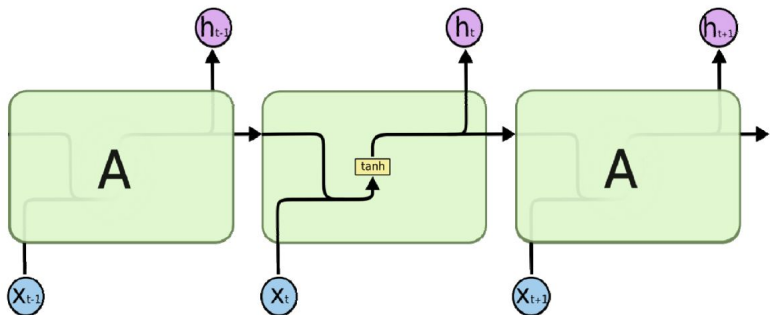


*"I grew up in France
...
I speak fluent French."*

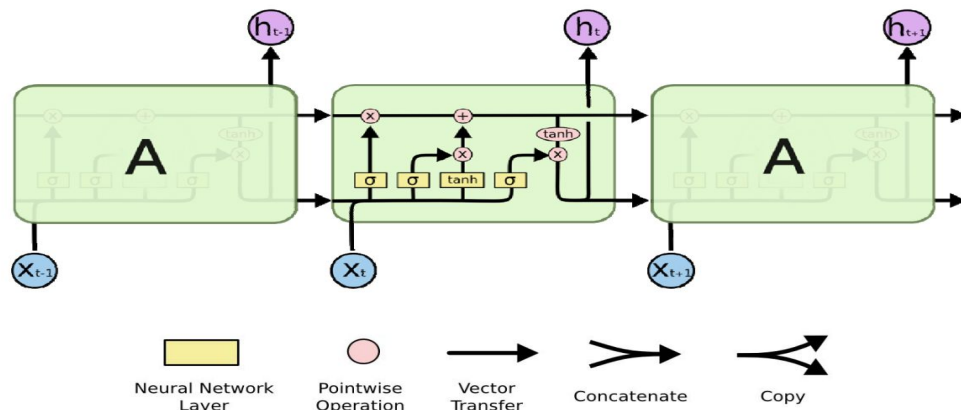
LSTM: long-short term memory



A **standard RNN** contains a single layer in the repeating module



LSTM: A special kind of RNN for learning long-term dependencies



Hmm, looks complicated.... Don't worry, we are going to figure it out

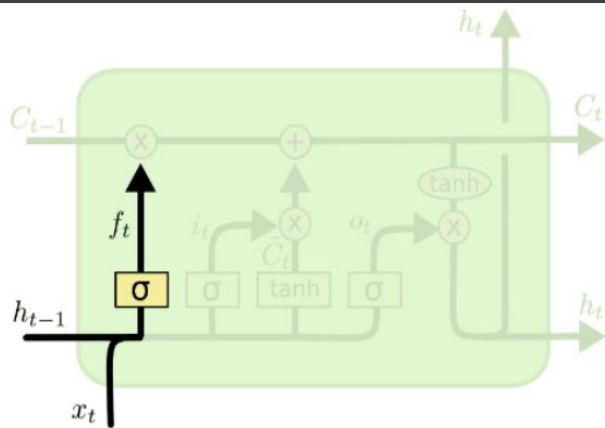
LSTM: Main Principle (in simple words)

< itc >

- **LSTM vs RNN:** similar control flow as a recurrent neural network
- **LSTM vs RNN:** both process data passing on as it propagates forward
- **LSTM vs RNN:** differences are the **operations** within the LSTM's cell
- LSTM cell operations are used to allow it **to keep or forget**

LSTM, Step by Step: Keep Gate

< itc >



“How much” of the previous cell state we want to keep/forget

$N \times (N+M)$

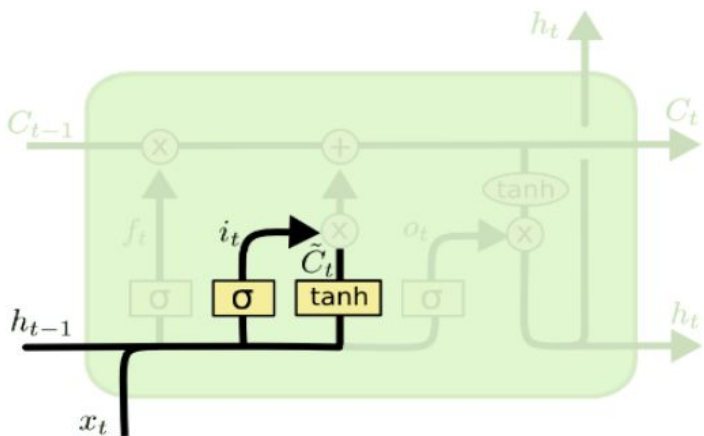
$$f_t = \sigma \left(W_f \cdot \begin{matrix} N \times 1 \\ h_{t-1}, x_t \end{matrix} + b_f \right)$$

$N \times 1 \quad N \times 1 \quad M \times 1$

f close to 0 means that we want to **forget most** of the previous state

LSTM, Step by Step: Write Gate

< itc >



**“How much” of the cell state
“candidate” we want to keep/forget**

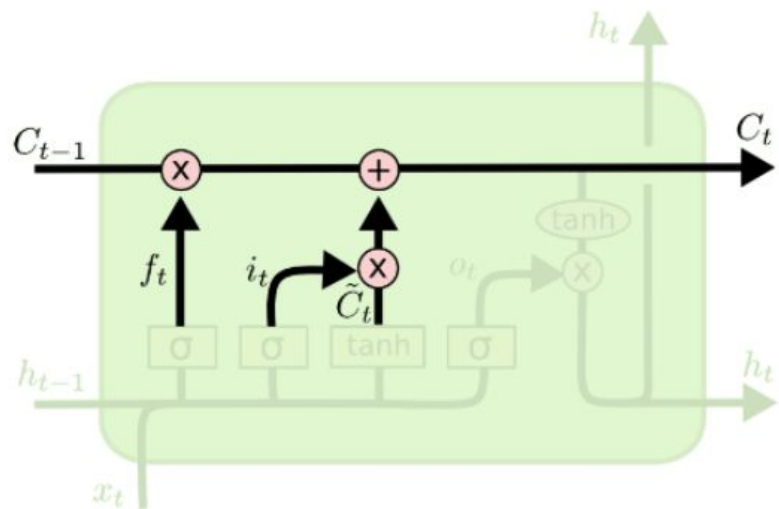
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

Cell state candidate

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

LSTM, Step by Step: Update gate

< itc >



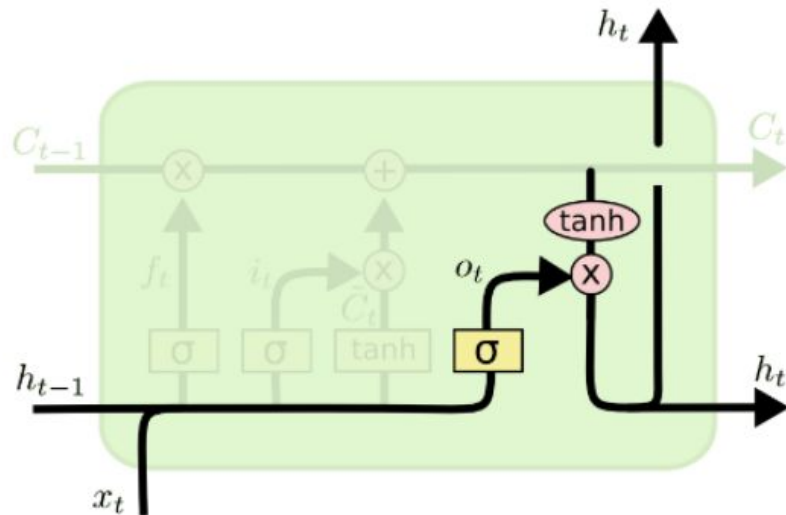
New cell state

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

Combine “portion” of the previous cell state
and “portion” the candidate cell state: **Long-Short Memory**

LSTM, Step by Step: Read(Output) gate

< itc >



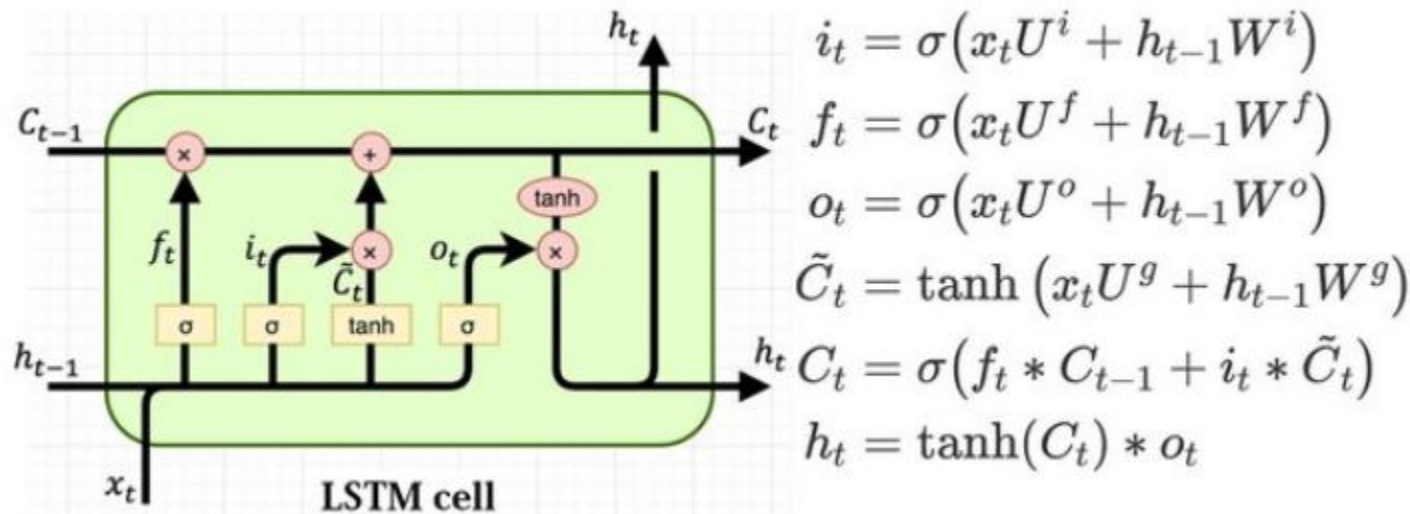
Standard RNN Output Computation

$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

Use cell state to compute new hidden state

LSTM: All pieces together

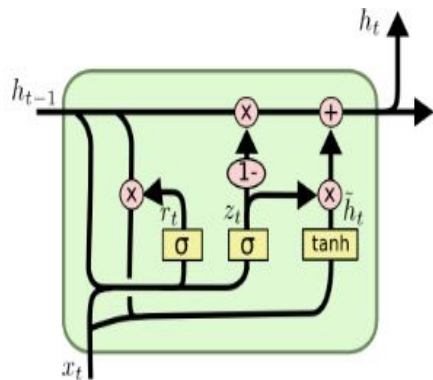
< itc >



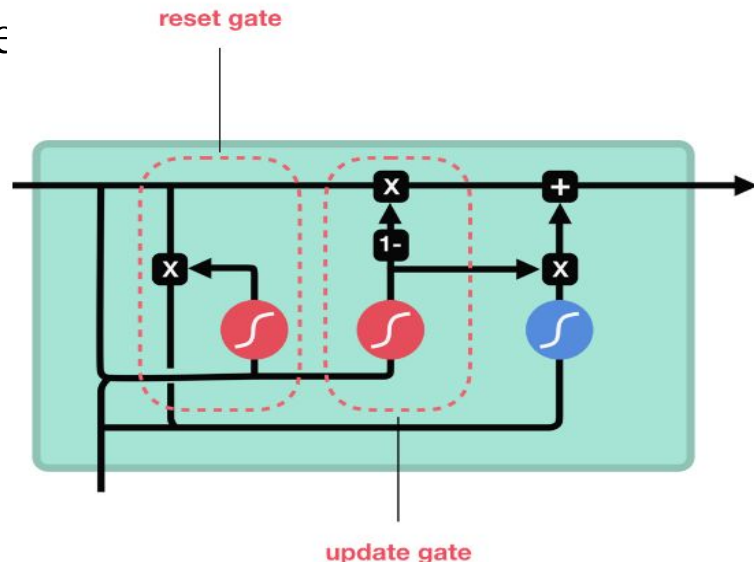
Not so complicated, right?

Gated Recurrent Unit(GRU): Simplified LSTM

- Introduced by Cho, et al. (2014).
- Got rid of the cell state and used the hidden state to transfer information.
- Has only two gates, a reset gate and update

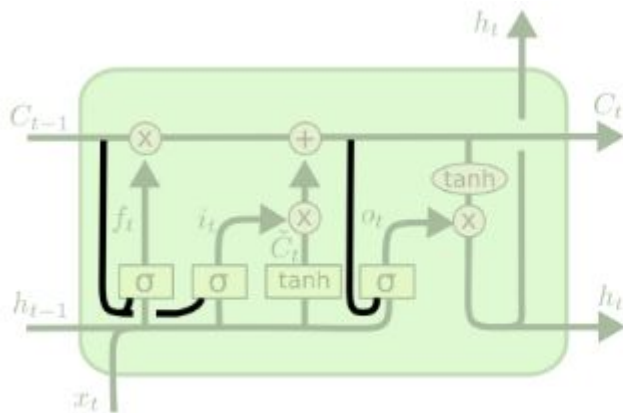


$$\begin{aligned}z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\\tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t\end{aligned}$$



LSTM with Peephole Connection

- Adds “peepholes” to all the gates
- Let the gate layers look at the cell state
- May use some peepholes and not others.



$$f_t = \sigma (W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma (W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i)$$

$$o_t = \sigma (W_o \cdot [C_t, h_{t-1}, x_t] + b_o)$$

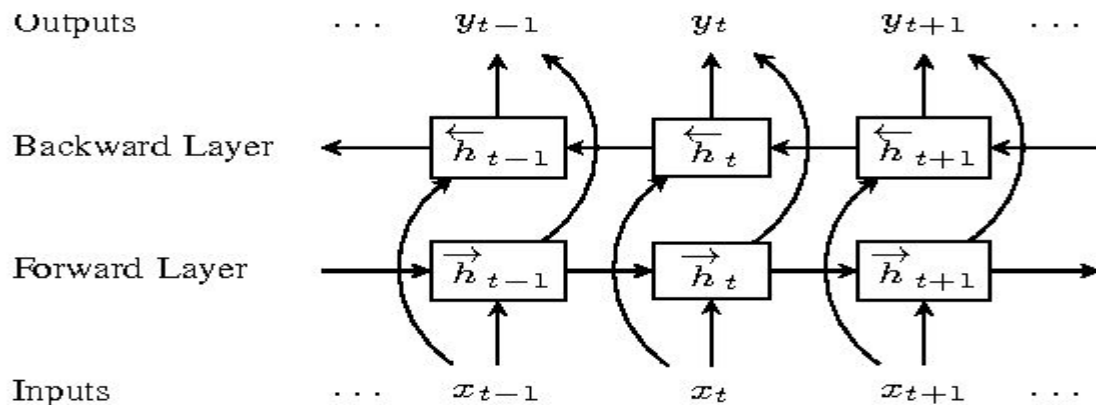
Let's say that we want to do name entity recognition. Following two examples are included in the training set; 'He said, "Teddy bears are on sale!"', and 'He said, "Teddy Roosevelt was a great President!"'.

If the information flows only from left to right, our model has no way to learn that "Teddy" in the second example is a name.

Solution: Bidirectional RNN

BiDirectional RNNs, Cont'

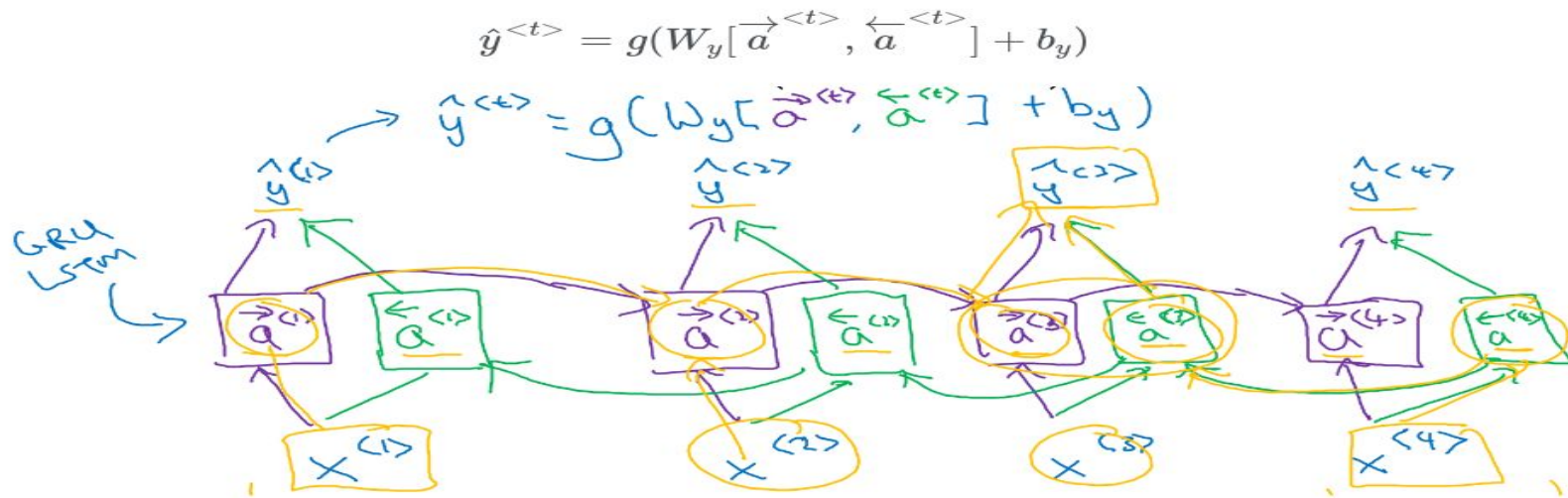
- 2 forward propagations; left to right and right to left
- For each layer combine activations from two forward props to output prediction.



Bi Directional RNN for word sequence

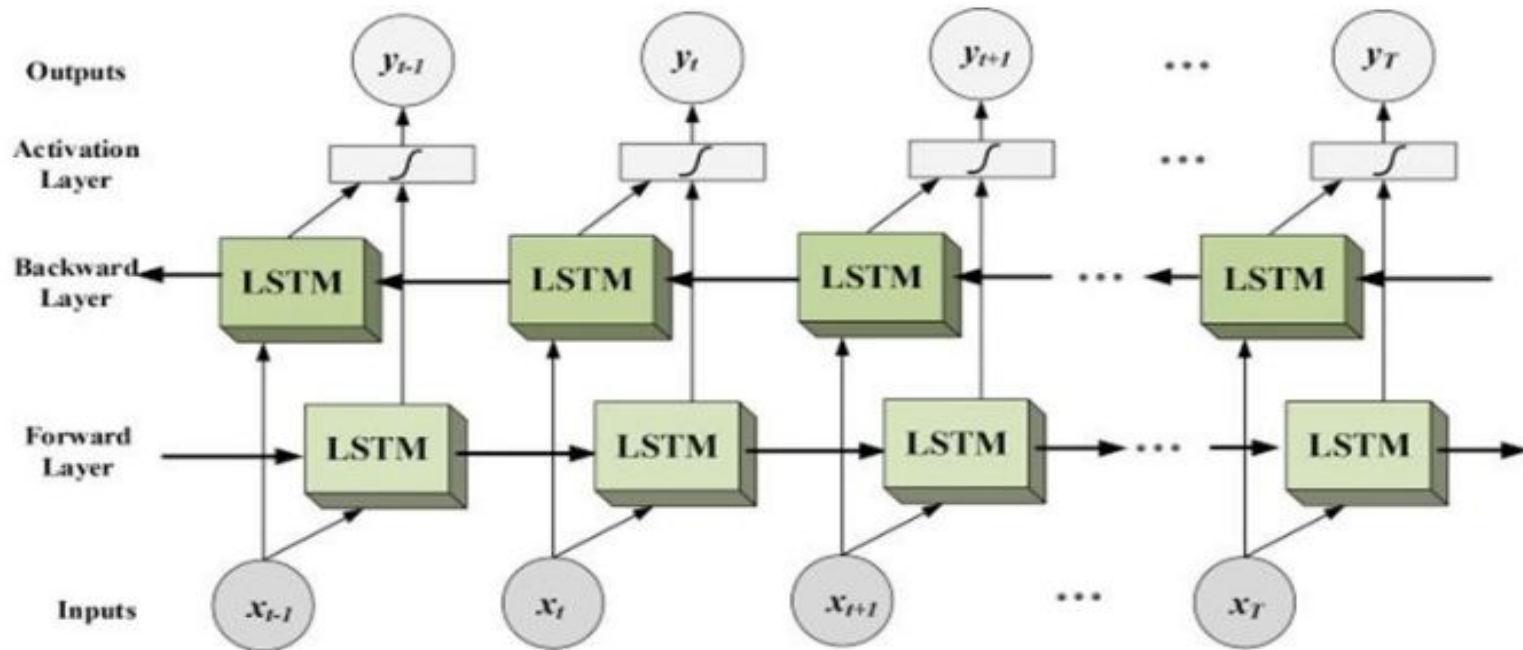
BiDirectional RNNs, more detailed view

< itc >



BiDirectional RNNs with LSTM

< itc >

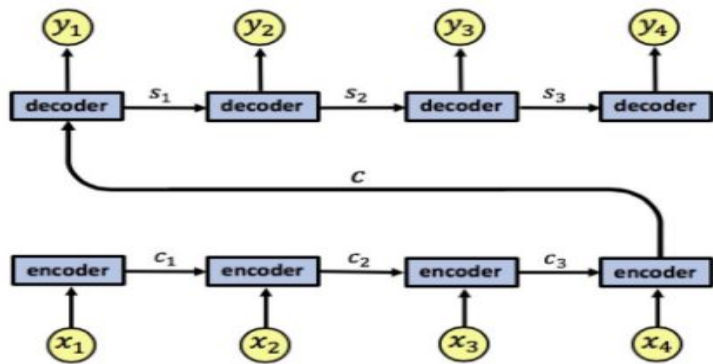


- Even LSTMs struggle to effectively encode(and decode) long sequences
- NN has to find connections between long input and output sentences (dozens of words) - machine translation, sentence similarity etc
- **Attention** (in RNN) is a mechanism allowing it to focus on certain parts of the input sequence when predicting a certain part of the output sequence
- **Combination of attention mechanisms** enabled improved performance in many tasks making it an integral part of modern RNN networks

Attention in RNN: Machine Translation Issue



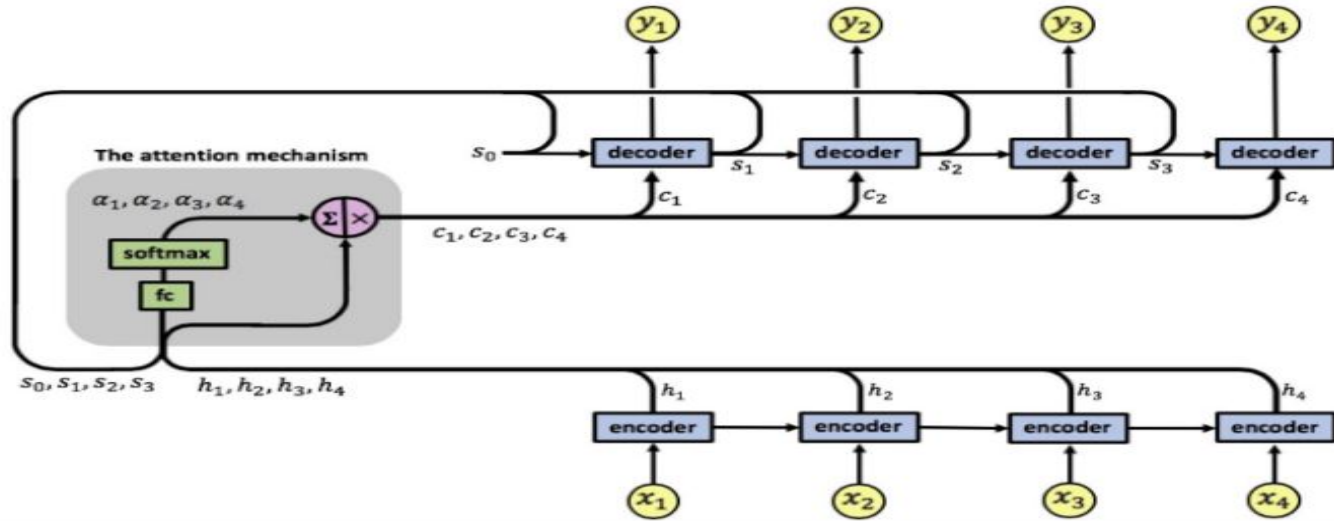
- Encoder needs to represent the entire input sequence as a single vector, which can cause information loss
- Decoder needs to decipher the passed information from this single vector, a complex task in itself.



A potential issue with this encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector. This may make it difficult for the neural network to cope with long sentences, especially those that are longer than the sentences in the training corpus.


Attention mechanism:

< itc >



The attention mechanism is located between the encoder and the decoder, its input is composed of the encoder's output vectors h_1, h_2, h_3, h_4 and the states of the decoder s_0, s_1, s_2, s_3 , the attention's output is a sequence of vectors called context vectors (c_1, c_2, c_3, c_4)

- The context vectors enable the decoder to focus on certain parts of the input when predicting its output.
- Each context vector is a weighted sum of the encoder's output vectors h_1, h_2, h_3, h_4
- The vectors h_1, h_2, h_3, h_4 are scaled by weights α_{ij} capturing the degree of relevance of input x_j to output at time i, y_i .

Context vector 

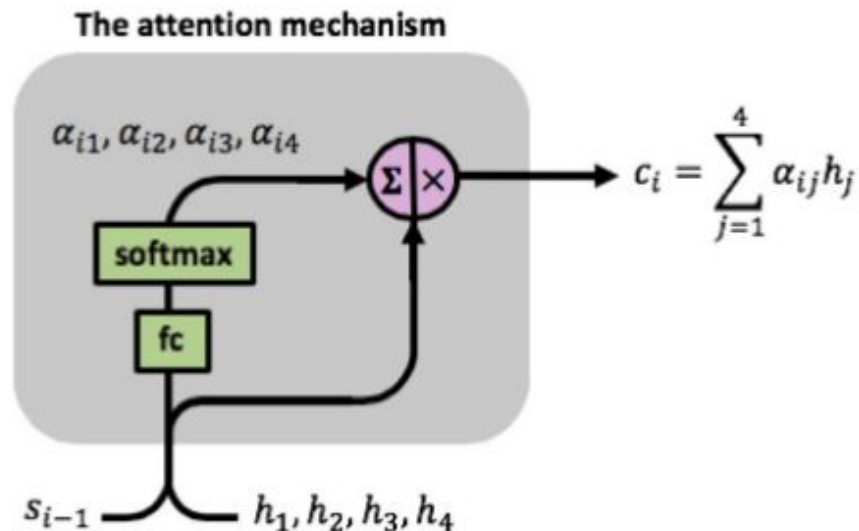
$$c_i = \sum_{j=1}^4 \alpha_{ij} h_j$$

Attention weights

- The attention weights are learned using an additional FC shallow network
- FC NN receives the concatenation of vectors $[s_{i-1}, h_i]$ as input at time step i and has a single fully-connect

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^4 \exp(e_{ik})}$$

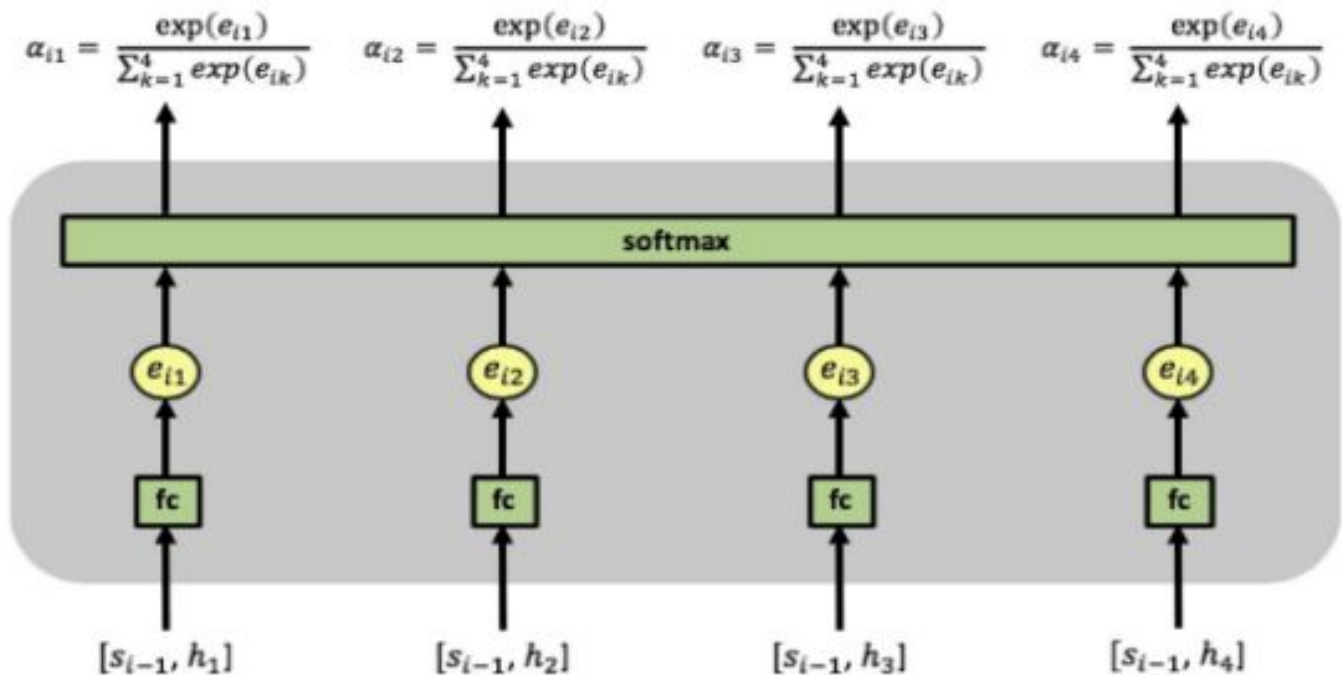
where $e_{ij} = \text{fc}(s_{i-1}, h_j)$



* Attention weights computation can be performed with more complex mechanisms

Attention Mechanism: All pieces together

< itc >



Keras has layer objects for various types of RNNs :

- ***tf.keras.layers.SimpleRNN***
- ***tf.keras.layers.GRU***
- ***tf.keras.layers.LSTM***

It also has wrapper ***tf.keras.layers.Bidirectional*** for creating bidirectional RNNs.

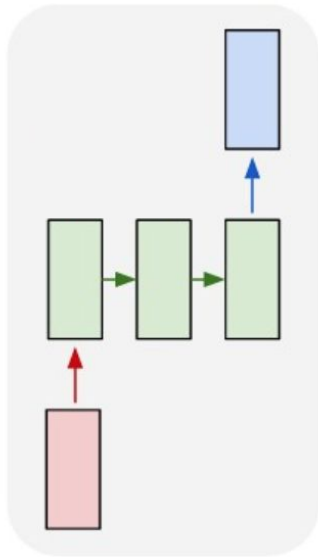
See also: [Keras recurrent layers documentation](#)

```
tf.keras.layers.SimpleRNN(  
    5, # number of units  
    return_state=False  
    return_sequences=False  
)
```

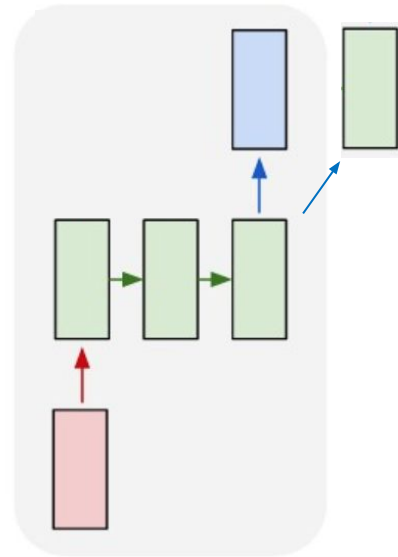
Parameters:

- number of hidden units
- **return_state:** if **True**, return the last hidden state as well as the last output
- **return_sequences:** if **True**, return sequence of outputs instead of just the last output

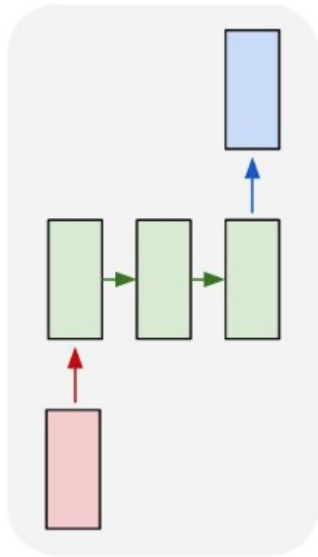
SimpleRNN(5, return_state=**False**)



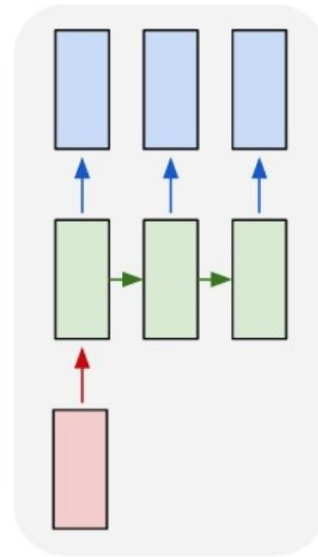
SimpleRNN(5, return_state=**True**)



SimpleRNN(5,
return_sequences=False)



SimpleRNN(5,
return_sequences=True)



Example of a simple RNN on input with 10 time steps and 4 features for each time sample:

```
In [1]: 1 from tensorflow.keras.layers import Input, SimpleRNN  
        2 from tensorflow.keras.models import Model
```

```
In [2]: 1 input_layer = Input((10, 4))  
        2 rnn_output = SimpleRNN(5)(input_layer)  
        3 model = Model(inputs=input_layer, outputs=rnn_output)
```

```
In [3]: 1 model.input_shape, model.output_shape
```

```
Out[3]: ((None, 10, 4), (None, 5))
```

Example with ***return_state=True***:

```
In [1]: 1 from tensorflow.keras.layers import Input, SimpleRNN
        2 from tensorflow.keras.models import Model
```

```
In [2]: 1 input_layer = Input((10, 4))
        2 rnn_output, final_state = SimpleRNN(5, return_state=True)(input_layer)
        3 model = Model(inputs=input_layer, outputs=[rnn_output, final_state])
```

```
In [3]: 1 model.input_shape, model.output_shape
```

```
Out[3]: ((None, 10, 4), [(None, 5), (None, 5)])
```


Example with ***return_sequences=True***:

```
In [1]: 1 from tensorflow.keras.layers import Input, SimpleRNN  
        2 from tensorflow.keras.models import Model
```

```
In [2]: 1 input_layer = Input((10, 4))  
        2 rnn_sequence_output = SimpleRNN(5, return_sequences=True)(input_layer)  
        3 model = Model(inputs=input_layer, outputs=rnn_sequence_output)
```

```
In [3]: 1 model.input_shape, model.output_shape
```

```
Out[3]: ((None, 10, 4), (None, 10, 5))
```

`tf.keras.layers.GRU/LSTM` take the same parameters as **`SimpleRNN`**

The syntax is mostly the same, but **`LSTM`** has both hidden and cell states that are returned with **`return_state=True`**:

```
In [1]: 1 from tensorflow.keras.layers import Input, LSTM
        2 from tensorflow.keras.models import Model
```

```
In [2]: 1 input_layer = Input((10, 4))
        2 lstm_output, final_hidden_state, final_cell_state = LSTM(5, return_state=True)(input_layer)
        3 model = Model(inputs=input_layer, outputs=[lstm_output, final_hidden_state, final_cell_state])
```

```
In [3]: 1 model.input_shape, model.output_shape
```

```
Out[3]: ((None, 10, 4), [(None, 5), (None, 5), (None, 5)])
```

Topics Not Covered :(

< itc >

- Multi-layer RNNs
- Back propagation in RNNs
- Gradient explode/vanishing problem of RNNs
- CNN and RNN combined together (e.g. image captioning, video processing)

This is only what came to my mind