Large Language Models

Recurrent Neural Networks

Flavio Giobergia

Some limits of FCNN

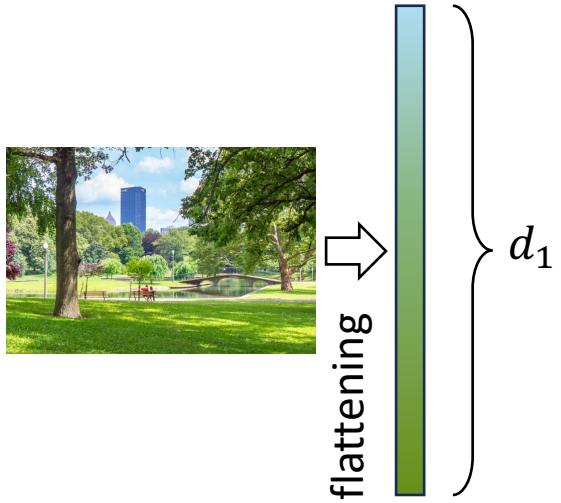
- Remember: Fully-connected NN are cascades of matrices/non-linearities
  - E.g., $f(x) = \sigma(W_{out} ReLU(W_{hid} ReLU(W_{in} x))$

- This type of models have some architectural constraints!

1. The *input* is expected to be *fixed in size*
2. Each input is associated with its set of weights
3. The *output* is expected to be *fixed in size*

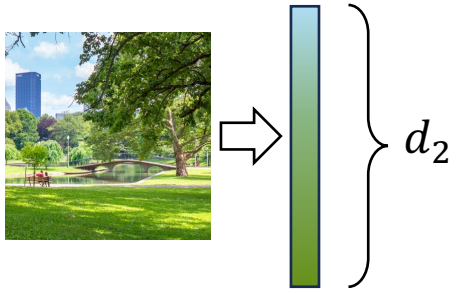
The *input* is expected to be *fixed in size*

- As defined by the number of columns in W_{in}



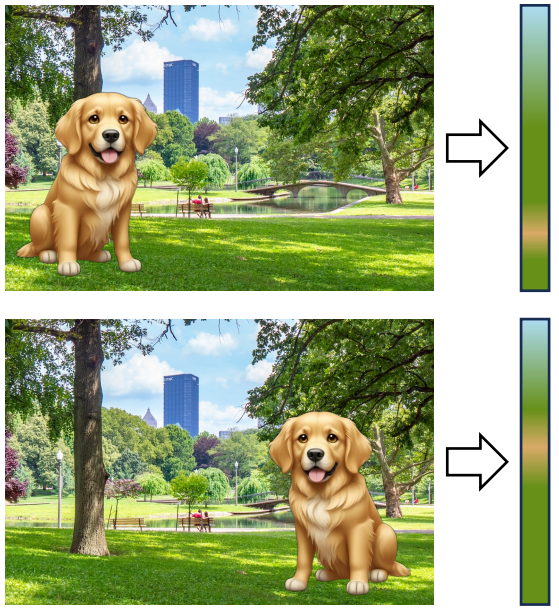
- For *structured data* (e.g., tabular) this is generally fine
  - We typically have a fixed number of input attributes
  - (Missing values need to be handled in some way)

- For *unstructured data*, this is more troubling!
  - Images may have different resolutions/ratios
  - Audios have different sampling frequencies and lengths
  - Texts come in all lengths
  - …

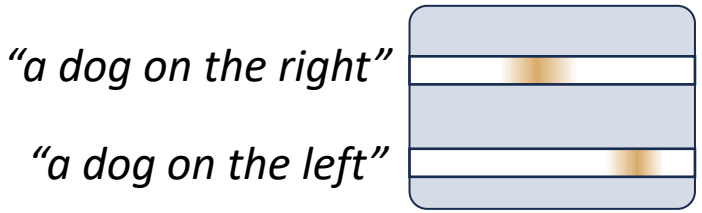


Each input is associated with its set of weights

- If the same pattern occurs in different portions of the input, the model needs to learn the same pattern multiple times



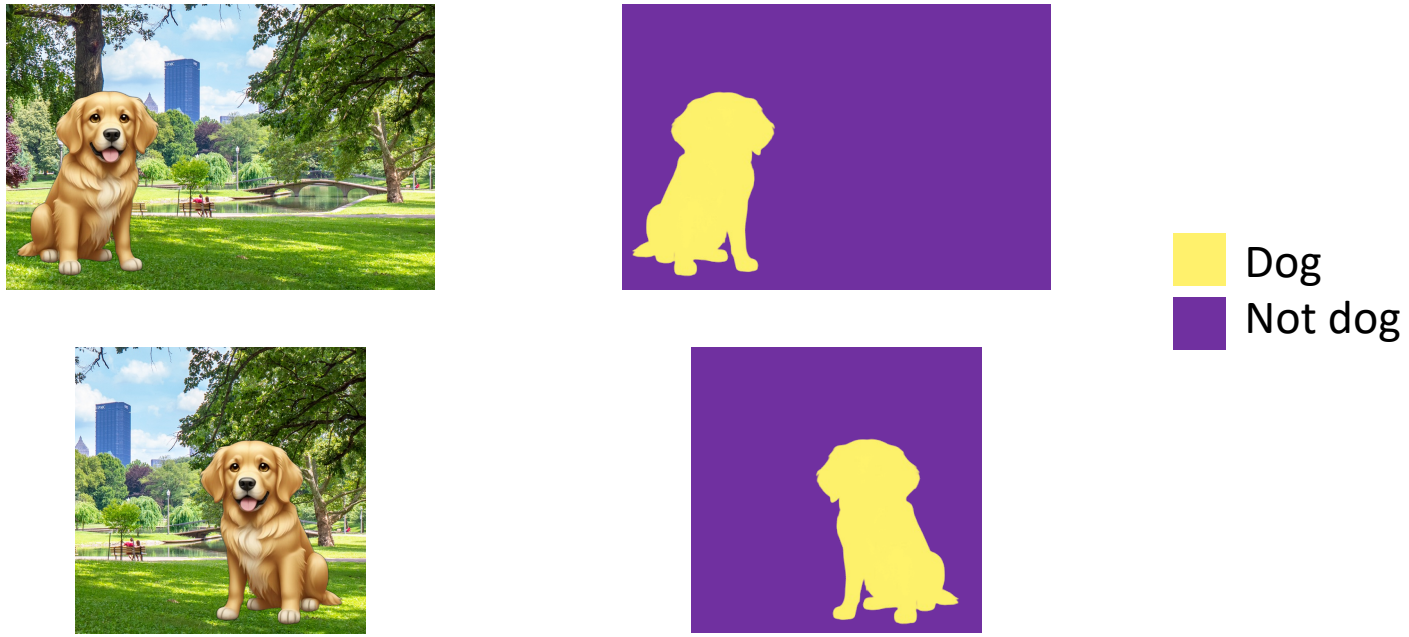
- This is expensive when patterns can occur in multiple ”places”
  - In images, a dog is a dog, regardless of where it is
  - However, A FCNN needs to separately learn what “a dog is” (pattern) in each place separately. This is:
  1. Inefficient
  2. More data-intensive (as we need pictures of dogs in different places)



Same pattern, just found in different places of the input!

The *output* is expected to be *fixed in size*

- As defined by the number of rows in W_{out}
- Sometimes, we may want outputs to have varying sizes
  - In object detection, we want to *segment* the pixels of an image



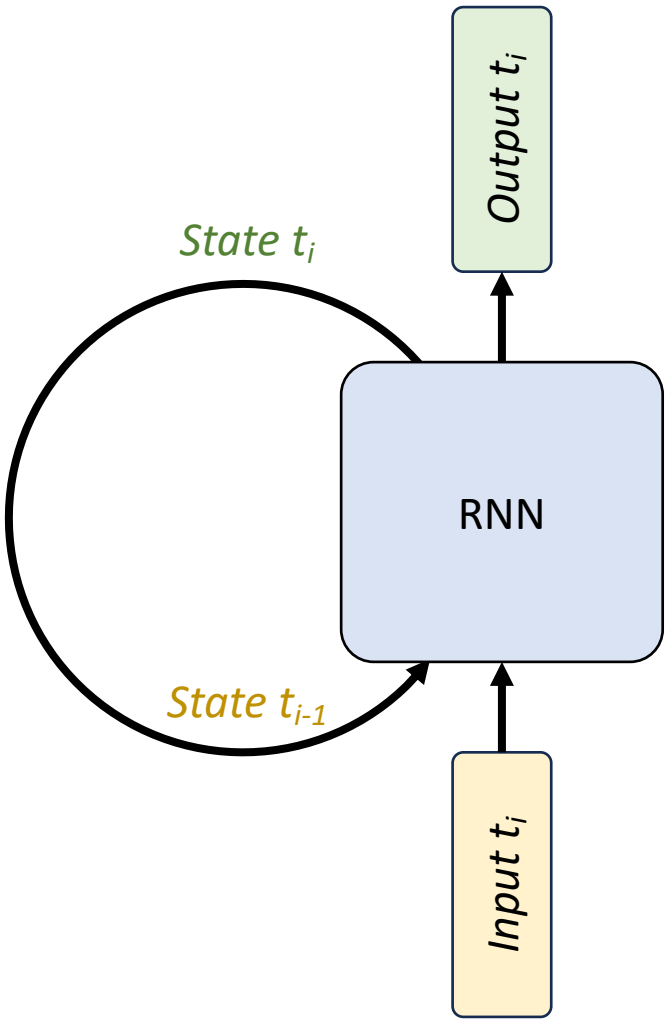
Recurrent Neural Networks (RNN)

- RNNs are a type of NN designed to process sequential data
  - Can receive a sequence as input
  - And/or produce a sequence as output
- The input/output sequences can be of varying sizes!

- An RNN processes inputs one *element* at a time
  - *element* = "one unit of the sequence"
    - E.g., a word of sentence, a character of a string, a day of the year, etc.
- And maintains an internal (*hidden*) state
  - The hidden state summarizes the information seen so far

RNN architecture

- The general RNN architecture is charaterized by two *inputs* and two *outputs*
  - The *first input* is the element at the *current* time step of the input sequence
  - The *second input* is the hidden state produced in the *previous* time stemp
  - The *first output* is the model's prediction for the *current* time step
  - The *second output* is the model's hidden state for the *current* time step
    - Which will be used as the *second input* for the *next* time step



RNN architecture, unfolded

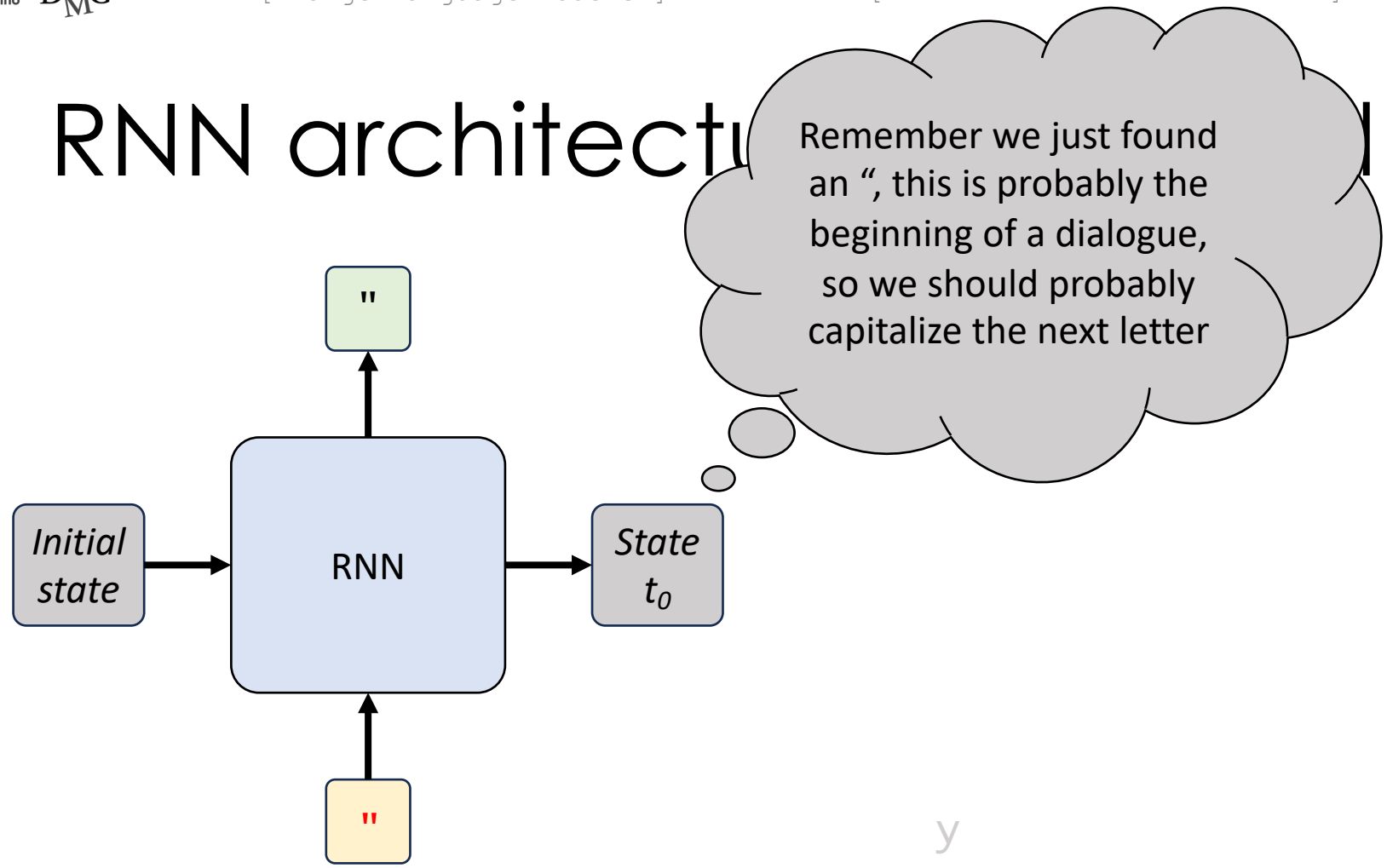
- To understand how an RNN handles an input sequence, let's consider a step-by-step case

- We feed a lowercase input string to the model (one character at a time), and expect the output to be the same string, with the correct case.
- Input sequence
  - "you are very kind, mr. holmes, but i cannot do that"
- Desired output sequence
  - "You are very kind, Mr. Holmes, but I cannot do that"

Note
For these sequences, each character represents an element of the sequence.
- s_0 = "
- s_1 = y
- s_2 = o
- s_3 = u

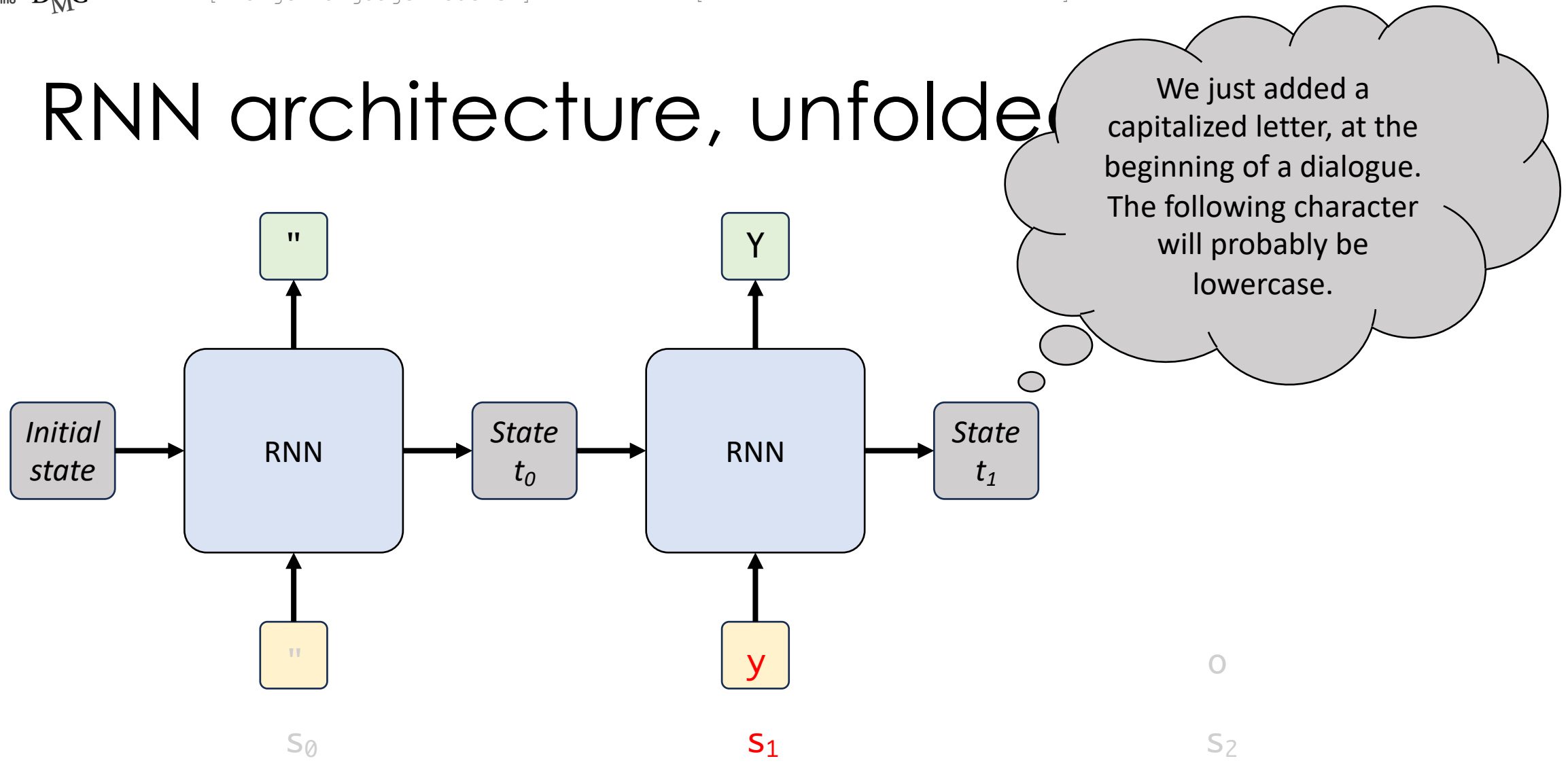
…
Each element can be represented as a 256-dimensional one-hot vector

RNN architectu



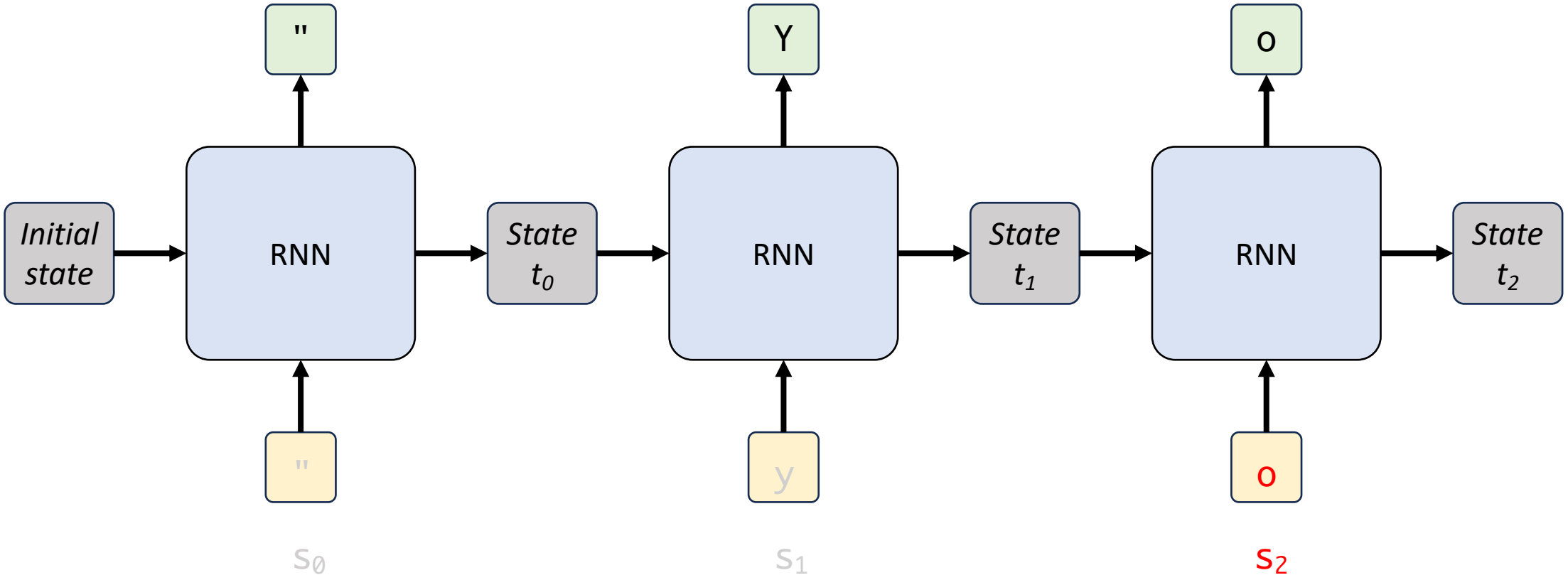
t = 0

RNN architecture, unfolde



t = 1

RNN architecture, unfolded

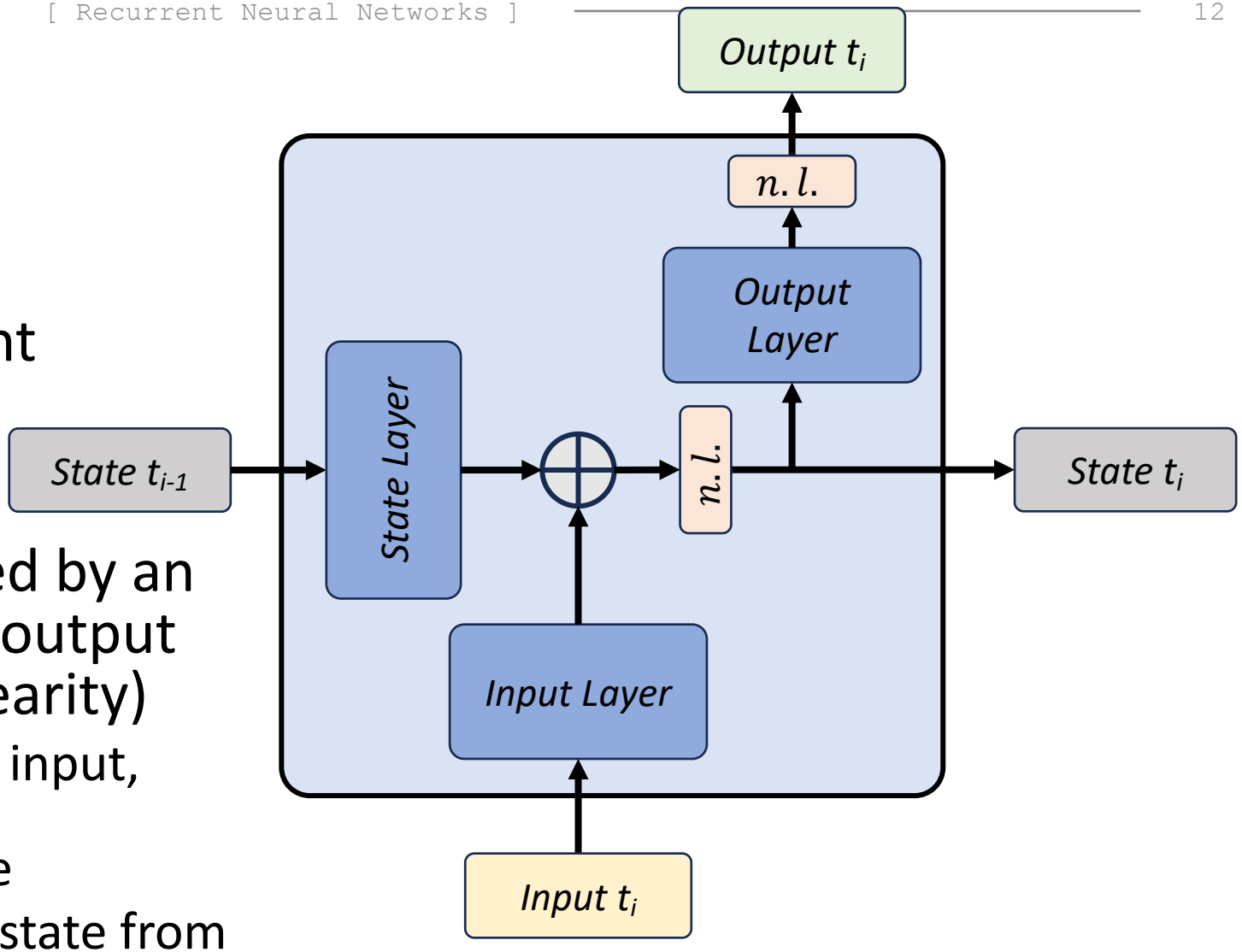


t = 2

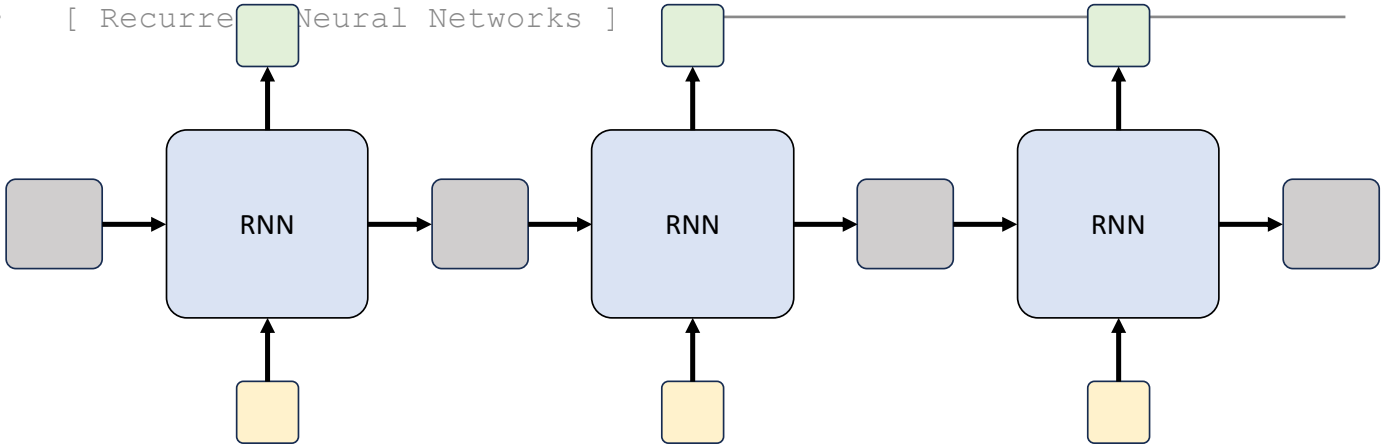
Inside an RNN

- Different RNNs can have different implementations

- A simple example is characterized by an input layer, a state layer, and an output layer (& of course some non-linearity)
  - *Input layer*: processes the current input, produces a vector with the same dimensionality as the hidden state
  - *State layer*: processes the hidden state from the previous step
  - *Output layer*: uses the current state (combination of input and state) to produce the current output



Weight sharing

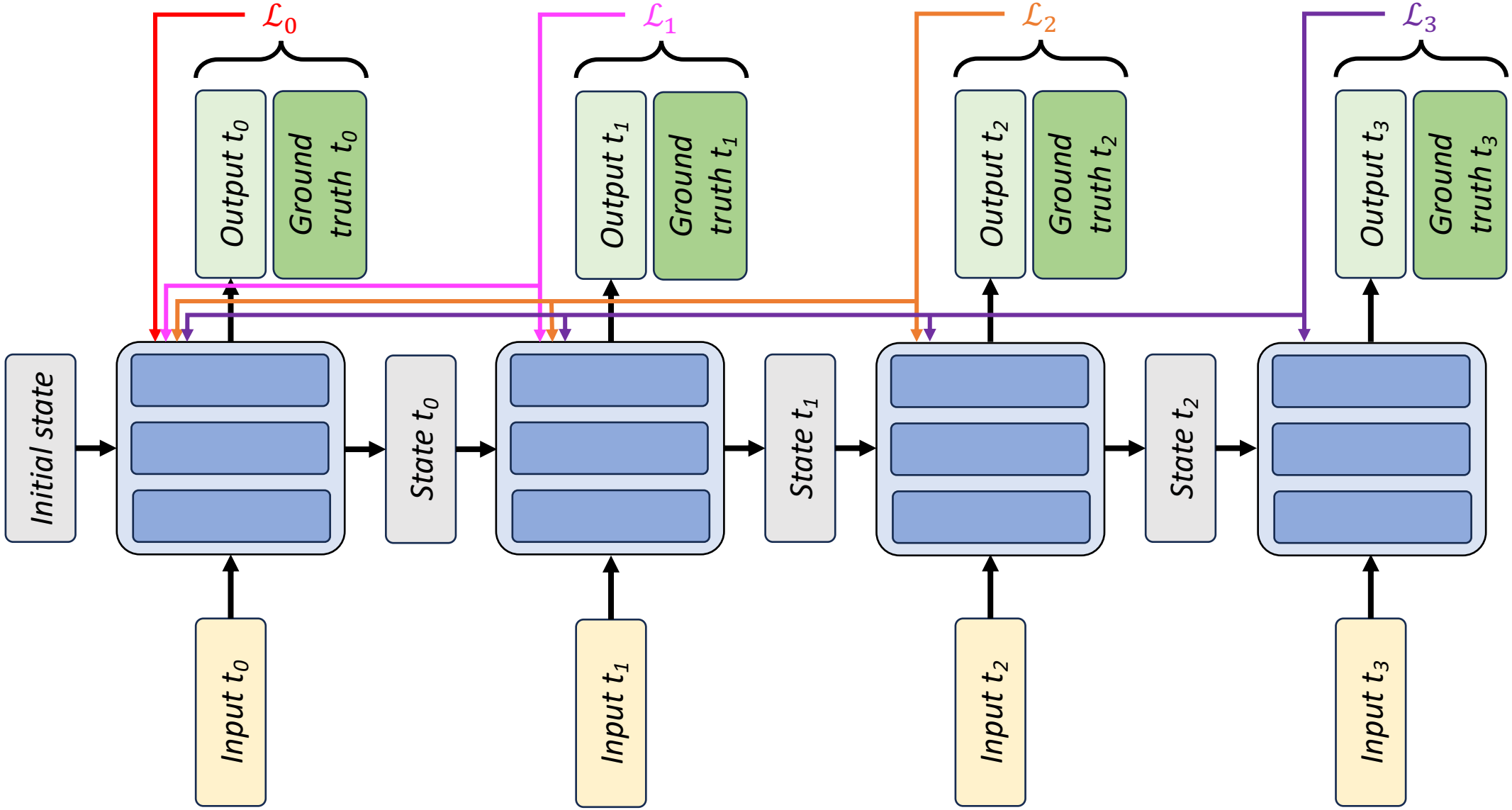


- The same model (i.e., same weights!) is applied multiple times throughout the sequence
- The state is used to provide the model with the context of what happened before
- So if the same context occurs in different moments of the sequence, the model's behavior will be the same, regardless of position
- No need to learn the same patterns in different positions of the input!
  - The model should capitalize the first letter of words after a period, regardless of where it is.

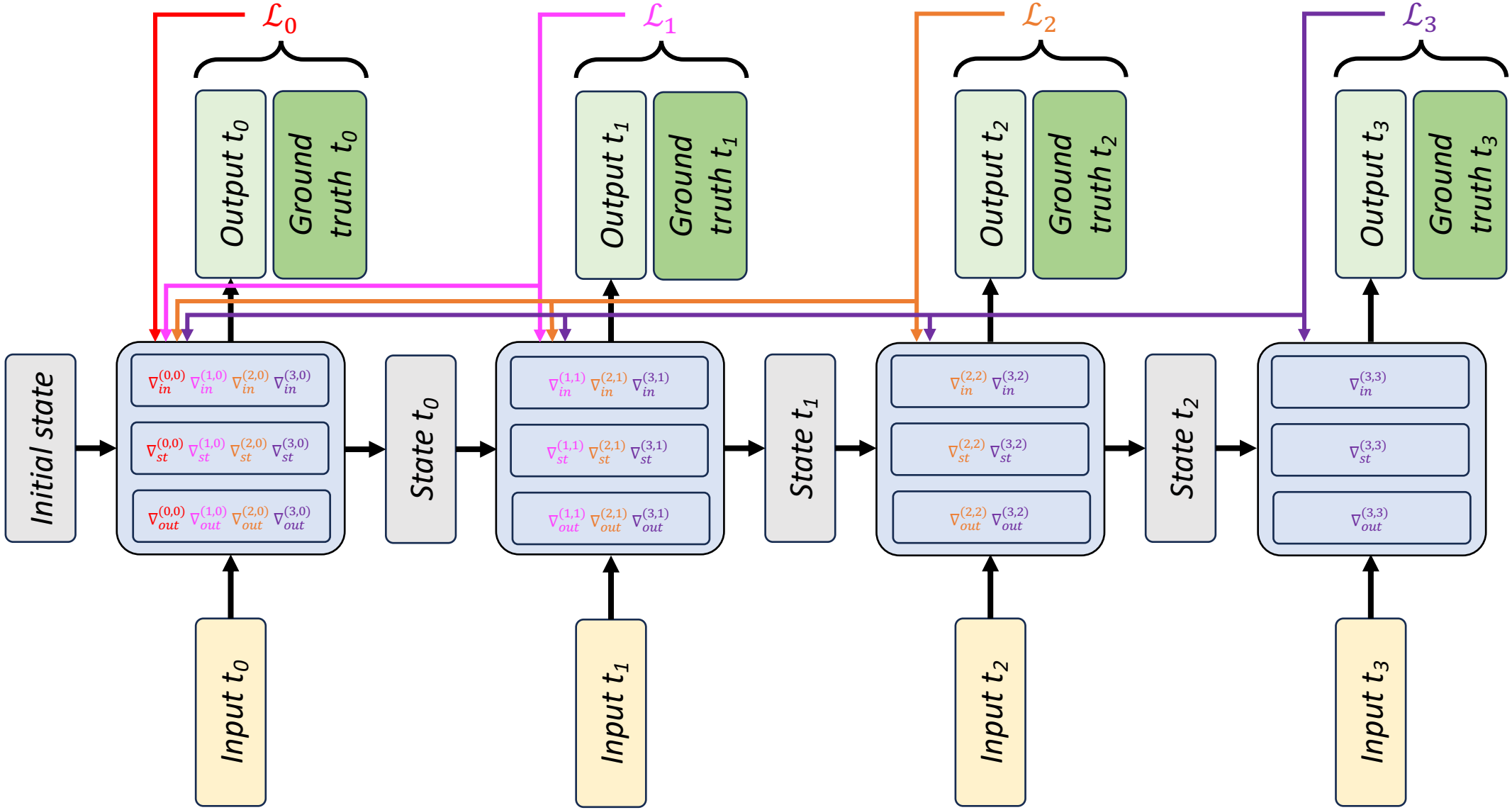
Backpropagation-through-time

- An RNN is nothing other than a bunch of NNs applied in cascade.
- At training time, we first unfold the RNN across the entire sequence
- Then, the loss function is computed and propagated
- The gradient of each layer will have multiple terms (one for each step of the sequence)
  - We simply add them up

Backprop-through-time (BPTT)

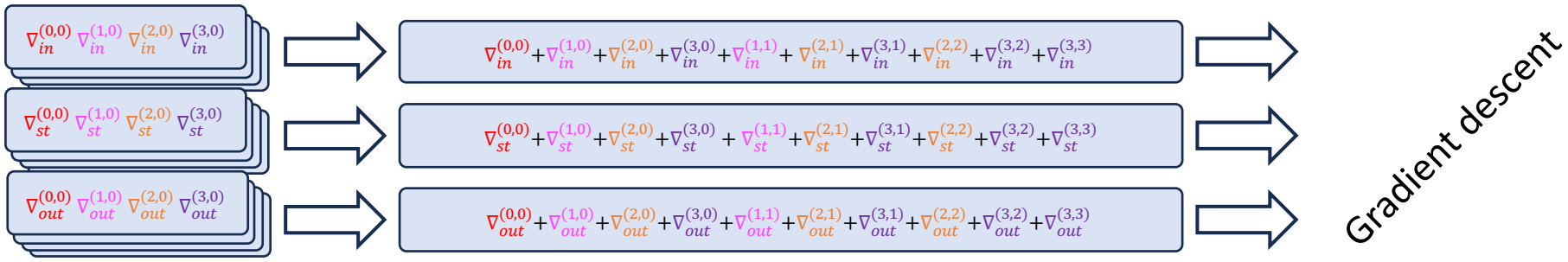


Backprop-through-time (BPTT)



Backprop-through-time (BPTT)

- Although the model is applied multiple times, the weights to be updated are actually the same
- The gradient from the various steps are accumulated into a single gradient
- The gradient descent step is taken once after all gradient contributions are backpropagated
  - (Backpropagation-through-time)

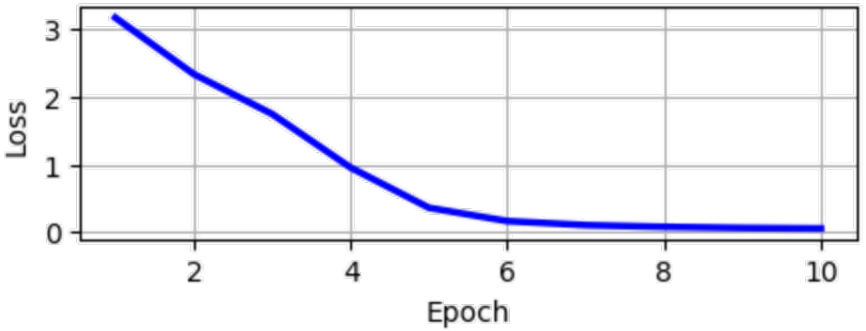


Case correction – RNN demo

- *Dataset*: The Adventures of Sherlock Holmes (Arthur Conan Doyle)
  - Randomly cropped sequences of characters, 10 to 100 ASCII characters long
    - E.g., “My cabby drove fast. I don't think I ever drove faster, but the others”
  - (the book is in the public domain, you can find it in txt format, online)
- *Input*: lowercase version of the sequences (1 ASCII char at a time)
- *Target output*: the correctly-cased sequences
- *Model*: RNN with 32-dimensional hidden state (initial state, all zeros), Adam optimizer, learning rate 0.001
- *Input/output encoding*: 256-dimensional vector
- *Loss function*: cross-entropy between predicted probability distribution and correct character
- *Training*: 10 epochs (batch size 32)

Case correction – RNN demo

- We keep track of the training loss throughout the training process (10 epochs).
- We test the RNN after each epoch on a fixed sentence, to observe how the predicted output varies




```
Input test sentence :  upon his pale face. "it may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
Target test sentence:  upon his pale face. "It may be so, or it may not, Mr. Holmes," said he, "but if you are so very sharp
================================================================================================================================
Epoch 01, loss: 3.1681 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 02, loss: 2.3334 eXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 03, loss: 1.7436 tst  X   s te e eee t e e e te se  ee se e e  ee  eee XXXee st s ss  e  t ee se eee  ee se eeee s  e_
Epoch 04, loss: 0.9519 upon his pale face. cit may be sos or it may notts mr. holresst said hes cbut if wou are so very sharp
Epoch 05, loss: 0.3579 upon his pale face. "it may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
Epoch 06, loss: 0.1599 upon his pale face. "it may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
Epoch 07, loss: 0.1009 upon his pale face. "It may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
Epoch 08, loss: 0.0752 upon his pale face. "It may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
Epoch 09, loss: 0.0599 upon his pale face. "It may be so, or it may not, mr. Holmes," said he, "but if you are so very sharp
Epoch 10, loss: 0.0515 upon his pale face. "It may be so, or it may not, mr. Holmes," said he, "but if you are so very sharp
```

The "X" represents the prediction of a null (0x00) byte – the implementation uses null bytes to pad sentences of different lengths. The model learns that null bytes show up a bunch of times, so it initially predicts mostly that character.

Next, the models figures out that spaces should be forwarded "as is", and gets most of them correctly. It also notices that the letter "e" occurs quite often (most frequent character in English), so it also predicts a bunch of e's.

A breakthrough (notice the drop in loss) occurs when the model realizes it can forward most letters "as is". It still needs some time to figure out punctuation, but it has most parts of the string under control.

Later, it figures out that after periods (.) -- even if followed by quotes ("), it should capitalize the next letter. Notice how the model capitalizes Holmes not because it is a name (remember, the model only knows that there's an "h", it cannot see the future), but because the "h" follows a period.

```
Input test sentence :  upon his pale face. "it may be so, or it may no
Target test sentence:  upon his pale face. "It may be so, or it may no
==========================================================================
Epoch 01, loss: 3.1681 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 02, loss: 2.3334 eXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 03, loss: 1.7436 tst X  s te e eee t e e e te se ee se e e e
Epoch 04, loss: 0.9519 upon his pale face. cit may be sos or it may nots mr. nolresst said hes cbut if wou are so very sharp
Epoch 05, loss: 0.3579 upon his pale face. "it may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
Epoch 06, loss: 0.1599 upon his pale face. "it may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
                       face. "It may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
                       face. "It may be so, or it may not, mr. holmes," said he, "but if you are so very sharp
                       face. "It may be so, or it may not, mr. Holmes," said he, "but if you are so very sharp
                       face. "It may be so, or it may not, mr. Holmes," said he, "but if you are so very sharp
```

The model still cannot see the future, so it will unlikely learn that "m" should be capitalized because it is the beginning of "Mr."!

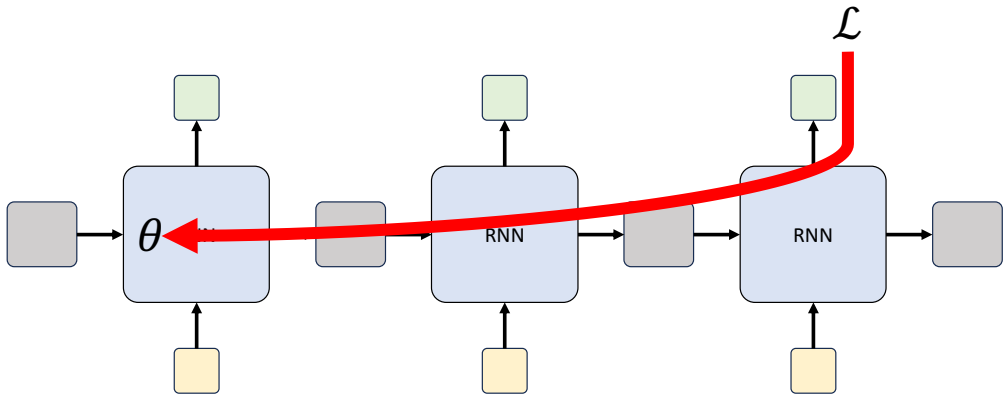
Problems with RNNs

- *Vanishing & Exploding gradients*
  - The gradients are computed as a product of many terms – this can make the gradient go to 0, or to infinity!
- *Long-Term Dependency Issues*
  - The hidden state should keep track of the history of the sequence, but in practice it mostly remembers the most recent inputs
- *Computational inefficiency*
  - RNNs cannot be parallelized, since the execution of each step depends on the previous one

Vanishing and Exploding Gradients (I)

- Remember: gradients are backpropagated by means of the chain rule
  - $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial x}$

- If a sequence has length N, we will unroll the RNN N times!
  - The gradients will be multiplied backward across "unrolls"
  - E.g., $\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial b}\frac{\partial b}{\partial c} \dots \frac{\partial z}{\partial \theta}$



Vanishing and Exploding Gradients (II)

- If |gradients| are mostly < 1, the final gradient will be ~ 0
  - 😐 *Vanishing gradient*
  - $0.9^{50} = 0.005$
  - Some activation functions (e.g., sigmoid) also aren't helping, since they have ~ 0 derivative almost everywhere
- If |gradients| are mostly > 1, the final gradient will grow to large values
  - 💥 *Exploding gradients*
  - $1.1^{50} = 117.4$
- *Problem*: Gradients shrink (vanishing) or grow (exploding) exponentially during backpropagation, leading to very small/large updates to the network weights.

Long-Term Dependency Issues

- For RNNs, the only way to remember the past is to maintain a *hidden state*, updated at each time step based on:
  - the current input, and
  - the previous hidden state
- Ideally, this hidden state *should* capture all relevant information from previous time steps
- However, for long sequences, earlier steps are *forgotten* from the hidden state, in favor of the more recent ones:
  - The state is updated often, and the RNN has no explicit way of prioritizing/remembering important things from earlier
  - Vanishing/exploding gradients make the contribution of earlier inputs less significant

Long-Term Dependency Issues

- We can devise a “next word prediction” task
- At step t, the model predicts what the t+1th word should be

- “The mouse was getting chased by the cat through the big house. After a long chase, the cat finally managed to catch the ” ➔ ?

- The answer is (of course) mouse

- However, RNNs struggle with this kind of task, since they “forget” what they saw at earlier stages.

Gated RNNs

- Gated RNNs are a class of RNNs designed to address the *long-term dependency* and *gradient issues* of RNNs
  - Gates used to control the flow of information

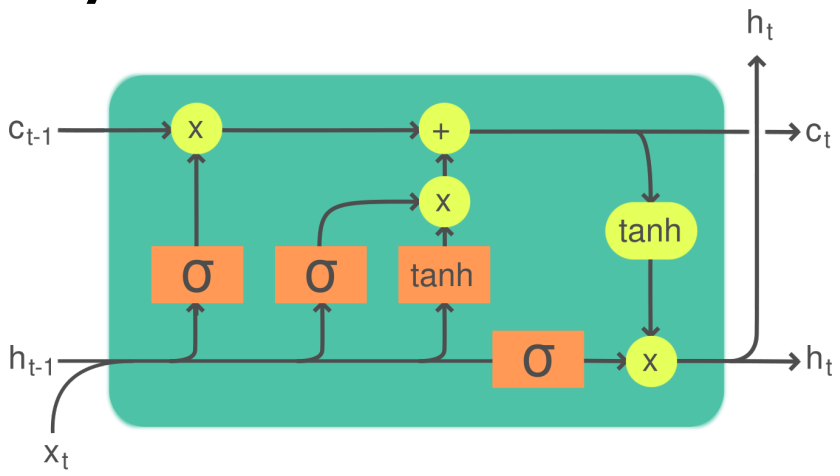
- Common types:
  - *LSTM* (Long Short-Term Memory)
    - Introduce gates that manage memory more effectively
  - *GRU* (Gated Recurrent Unit)
    - A simplified version of LSTM with fewer gates but similar effectiveness

Key Components of Gated RNNs

- *Input Gate*
  - Controls how much of the new input should influence the current memory
- *Forget Gate*
  - Decides which parts of the previous memory should be forgotten/retained
- *Output Gate*
  - Determines (1) how much of the current memory should be passed on to the next time step and (2) how much should be output.
- *Update Gate* (in GRU)
  - A single gate that combines the functionality of the input and forget gates
  - (makes the architecture simpler)

Long Short-Term Memory

- *Inputs* (x_t)
  - The input is immediately concatenated to the pervious hidden state
- *Cell states* (c_t)
  - The cell state passes, with only small changes, through the LSTM cell
  - It represents the "long-term memory"
- *Hidden states* (h_t)
  - It depends on both cell state, input and previous hidden state
  - It represents the "short-term memory"
- No output states: the hidden state works as an output – if we want step-by-step results we can further process the current output



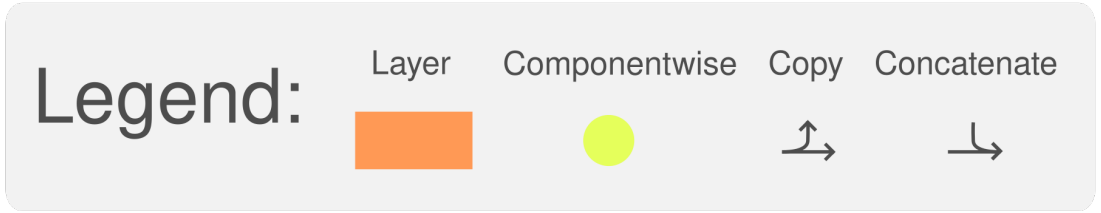
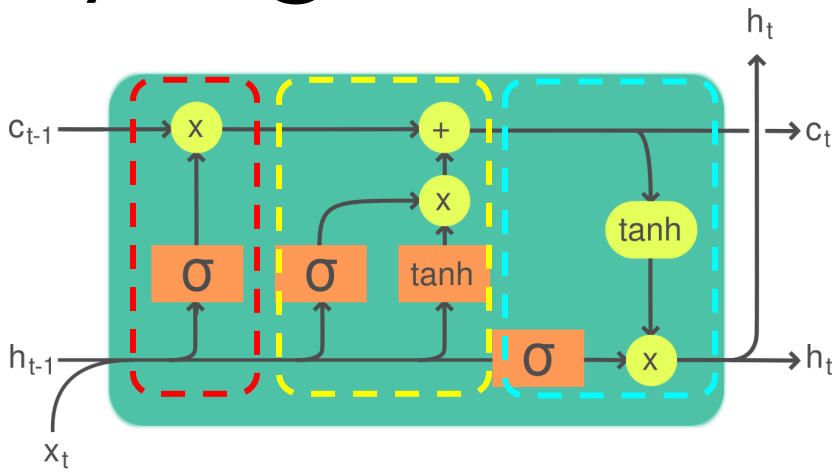


Image source: wikipedia

Note
In the above schema, we summarize a linear layer + non-linearity with a single orange block. For instance, σ indicates a linear layer, followed by a sigmoid (i.e., the block also has *learnable parameters*)

Long Short-Term Memory – gates

- *Forget gate*
  - Current input + previous hidden state are used to decide *how much* of the previous cell state should be forgotten
- *Input gate*
  - Current input + previous hidden state are used to decide *how much* of the input should be added to the next cell state
- *Output gate*
  - Current input + previous hidden state are used to decide *how much* of the cell state should become the new hidden state



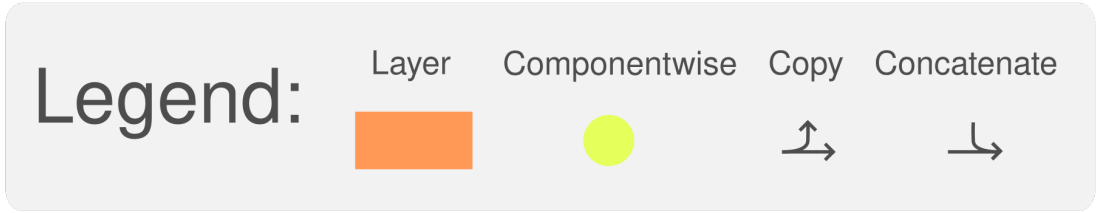
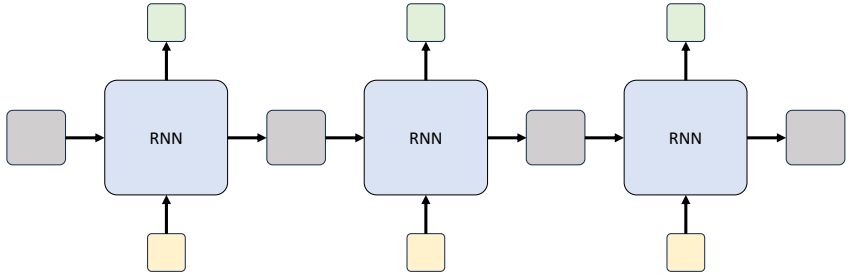


Image source: wikipedia

Note
When we need to choose how much of something should be passed, we multiply that something with the output of a sigmoid layer, which is bounded in [0,1].
0 ➔ keep nothing
1 ➔ keep everything

Sequence to sequence (seq2seq) tasks

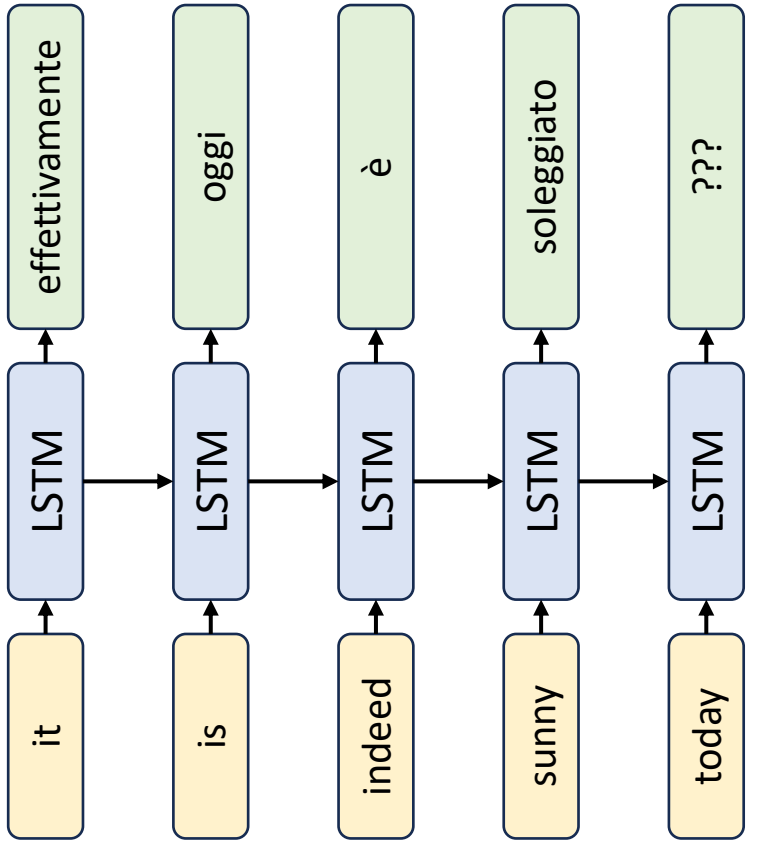
- We have discussed various tasks that consist in mapping an input sequence to an output sequence
  - These tasks are referred to as *seq2seq*



- We have currently considered situations where <u>input and output sequences have the same length</u>
  - One RNN step takes *1 input element* and produces *1 output element*
  - *One-to-one mapping* (between inputs and outputs)
  - In the previous demo ("case correction" problem) we associated, to each lowercase input letter, its correctly cased version

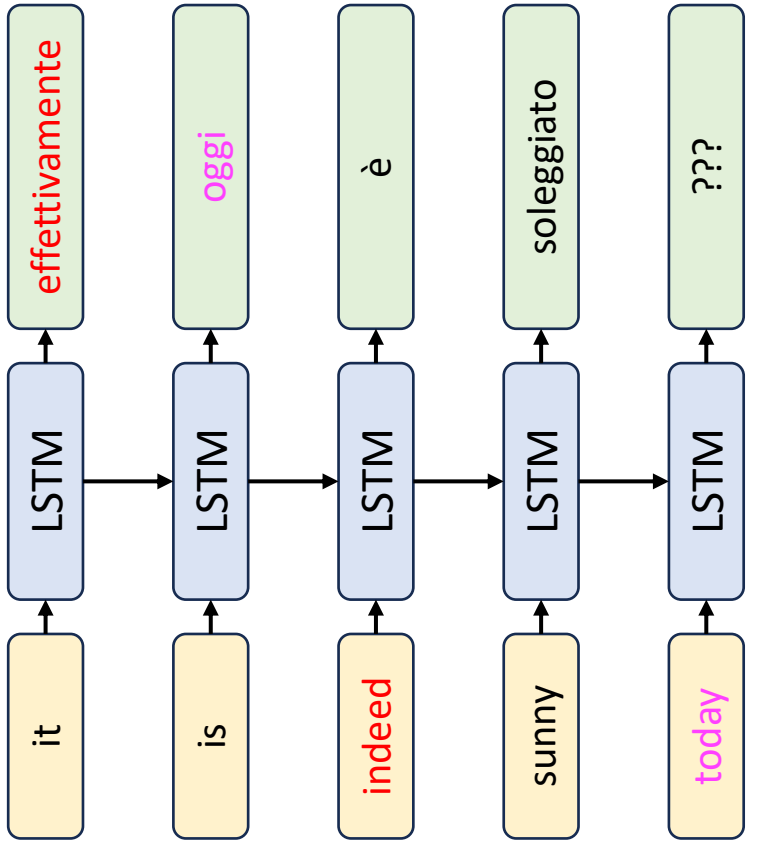
Limitations of one-to-one mapping (I)

- Inputs and outputs may have different lengths
  - For machine translation tasks, source and destination languages may encode the same sentence with different-length vectors
  - E.g., for English-to-Italian translactions
  - [it] [is] [indeed] [sunny] [today] ➔ [effettivamente] [oggi] [è] [soleggiato]
  - 5 words vs 4 words



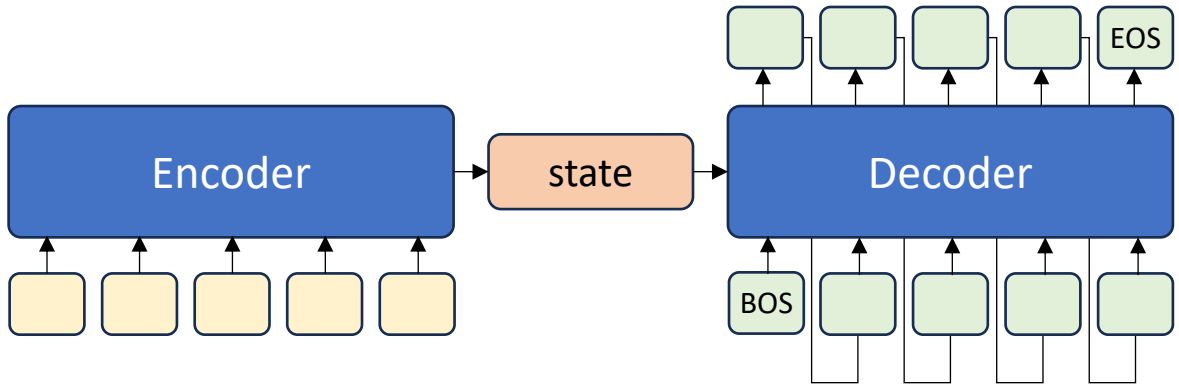
Limitations of one-to-one mapping (II)

- The LSTM doesn't get to see the entire input sequence until the end
  - We may need to know what happens ahead of time!
  - In the above example, the LSTM needs to know that the words "effettivamente" and "oggi" exist before their English versions appear in the input sequence
  - [it] [is] [indeed] [sunny] [today] ➔ [effettivamente] [oggi] [è] [soleggiato]

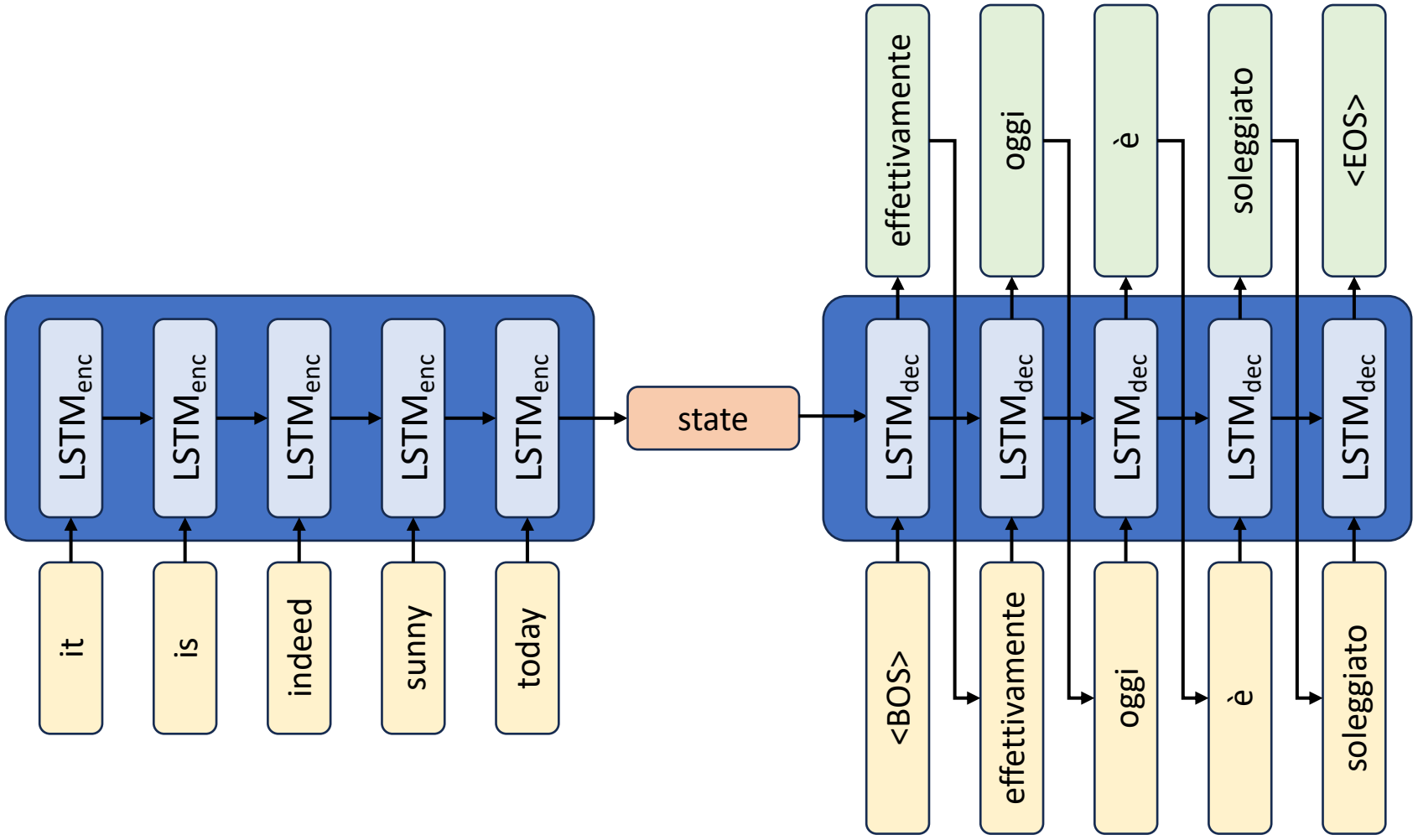


Encoder-decoder architecture

- Ideally, the model should:
  1. See the entire sequence before starting generating the output
  2. Produce a *state vector* that "encodes" the entire input sequence
  3. Use the previously mentioned vector to start generating the output
  4. Be allowed to generate until *it* decides to stop
     - i.e., it produces a special *token* that determines the End of Sequence (EOS)
- This architecture is called *encoder-decoder*



Encoder-decoder architectur



Note
The first token provided to the decoder is <BOS> (Beginning Of Sequence).

Note
The input tokens following the first one will be the outputs of the preceding steps.

Note
Sometimes during training we use *teacher forcing*, where we pass the "correct" output token instead of the predicted one. This helps reach convergence more quickly.

Note
The generation of the decoder continues until the decoder produces the <EOS> token.

Case correction problem – revised

- Target sentence:
  - "Indeed I am Sherlock Holmes,"
- Some interesting aspects:
  1. "I" of "Indeed" should be capitalized (beginning of a quote – ")
  2. "I" of "I" should be capitalized (subject pronoun)
  3. "S" of "Sherlock" should be capitalized (first name)
  4. "H" of "Holmes" should be capitalized (last name)
- What can we expect from one-to-one mapping?
  1. It should figure this one out (capitalization depends on previous character(s))
  2. It probably won't realize it (model only knows that this is the beginning of the word, it doesn't know this letter is a word in and of itself – it can't see the future!)
  3. Again, the model just sees a word starting with "s" – it cannot know that it will be followed by "herlock", thus requiring capitalization
  4. After the model sees that the previous word was sherlock, when it sees an "h", it may realize that the "h" will be the beginning of "Holmes"
     - This depends very much on how often "Sherlock Holmes" is contained in the training data

Results

Note
Note the larger number of epochs required to reach the result!

One-to-one mapping

```
Input test sequence:     "indeed i am sherlock holmes,"
Target test sequence:    "Indeed I am Sherlock Holmes,"
==========================================================
Epoch 0001, loss: 5.3212 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 0002, loss: 3.7157 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 0003, loss: 2.7964 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 0004, loss: 2.7981 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Epoch 0030, loss: 0.2264 "indeed i am sherlock holmes,"
Epoch 0050, loss: 0.0802 "indeed i am sherlock holmes,"
Epoch 0160, loss: 0.0301 "Indeed i am sherlock Holmes,"
Epoch 0530, loss: 0.0224 "Indeed i am sherlock Holmes,"
Epoch 1070, loss: 0.0232 "Indeed i am sherlock Holmes,"
```

Note
Indeed, the model can only figure out Indeed and Holmes -- but it has no way of knowing that ”I” is its own word, and that the “s” is the beginning of “Sherlock”

Encoder-decoder

```
                  "indeed i am sherlock holmes,"
                  "Indeed I am Sherlock Holmes,"
==========================================
loss: 5.4591 wwXXXXXXXXXXXXXXXXXXXXXXXXXX
loss: 4.0161 aXXXXXXXXXXXXXXXXXXXXXXXXXXX
loss: 2.8268 XXXXXXXXXXXXXXXXXXXXXXXXXXXX
loss: 2.7671 XXXXXXXXXXXXXXXXXXX
loss: 0.2315 "indeed i am sherlock holmes,"
loss: 0.0705 "indeed i am sherlock holmes,"
loss: 0.0259 "Indeed i am sherlock Holmes,"
loss: 0.0197 "Indeed I am sherlock Holmes,"
loss: 0.0156 "Indeed I am Sherlock Holmes,"
```

Note
The encoder-decoder first sees the entire sequence. It “remembers” that the “I” is a full word (so, it should be capitalized) and that the “s” is the first letter of sherlock (➔ capitalize!)

Limitations of Gated RNN

- Gated RNNs provide workarounds to the problems of vanilla RNNs. However, they only marginally improve upon the problems of RNNs

- Difficulties with long sequences are still present
  - Long-term dependencies still bottlenecked to a single vector
  - No access to the full input sequence
- Gradients still vanish/explode
  - BPTT still unrolls the recurrent model
- Added architectural complexity
  - The LSTM cell introduces additional parameters/gates
- Still cannot be parallelized