# STAT 453: Introduction to Deep Learning and Generative Models

Ben Lengerich

Lecture 16: Recurrent Neural Networks

November 3, 2026

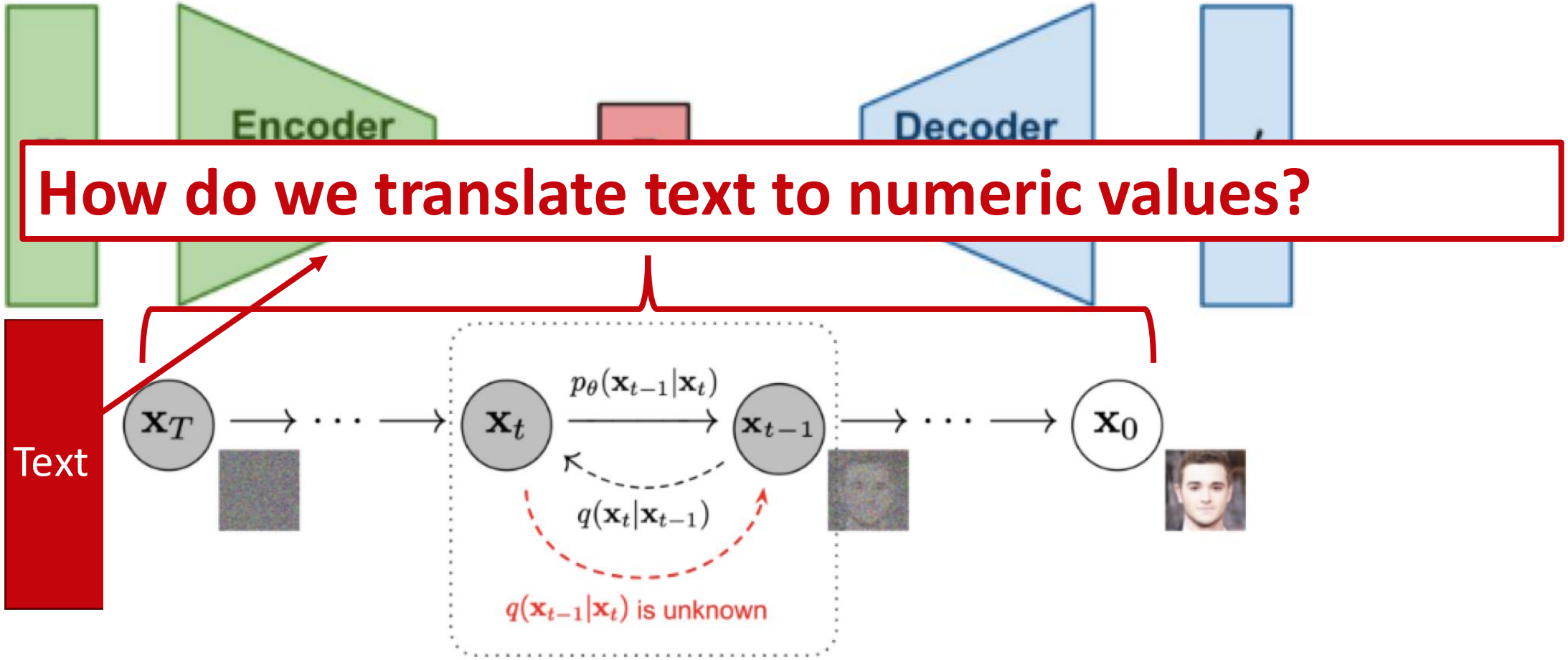
Reading: See course homepage

# This week

- HW3 due Friday EOD
- Project midway report due Friday EOD

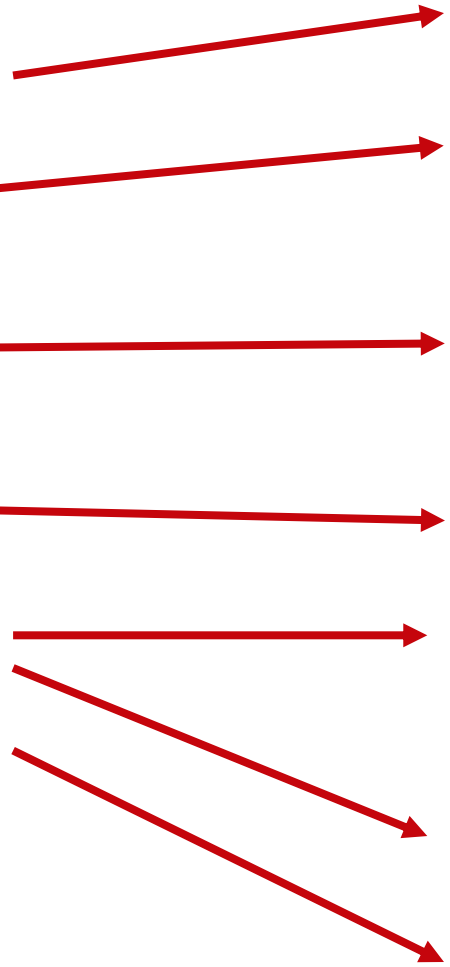
Ben Lengerich © University of Wisconsin-Madison 2026

# Recall “Conditioning on Text”…



# Challenges with text

- Variable length input
- Long-range dependencies
- Want a generative model
- Scale
- Emergent properties of large language models



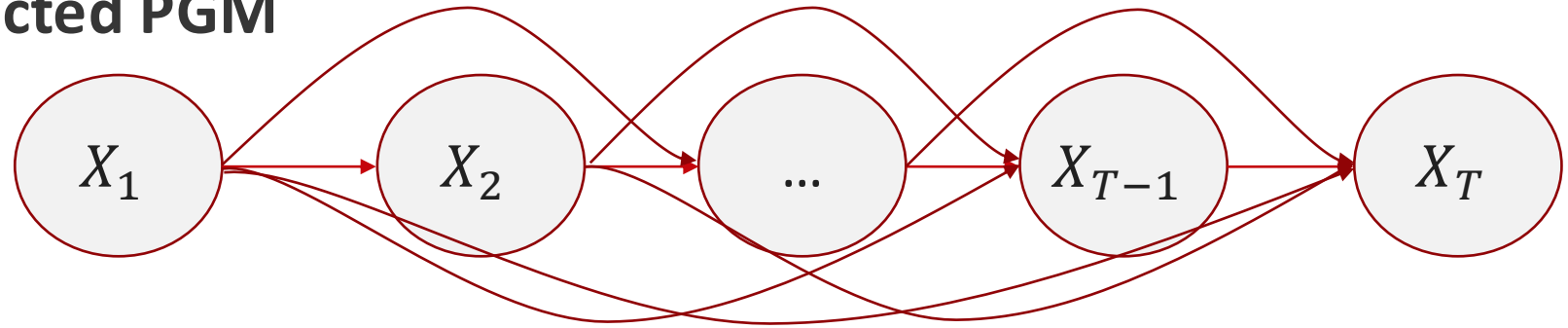
**Module 4: Sequence Models and Large Language Models**

| | |
|---|---|
| **11/3** | Lecture #16 (Prof. Lengerich):<br>**Sequence Learning with RNNs**<br>[ slides I notes ] |
| **11/5** | Lecture #17 (Prof. Lengerich):<br>**Attention, Transformers**<br>[ slides I notes ] |
| **11/10** | Lecture #18 (Prof. Lengerich):<br>**GPT Architectures**<br>[ slides I notes ] |
| **11/12** | Lecture #19 (Prof. Lengerich):<br>**Unsupervised Training of LLMs**<br>[ slides I notes ] |
| **11/17** | Lecture #20 (Prof. Lengerich):<br>**Aligning LLMs: SFT, RLHF, and DPO**<br>[ slides I notes ] |
| **11/19** | Lecture #21 (Prof. Lengerich):<br>**Reasoning Models and RL from Verifiable Rewards**<br>[ slides I notes ] |
| **11/24** | Lecture #22 (Prof. Lengerich):<br>**Parameter-Efficient Fine-tuning: LoRA, Adapters, and Prompting**<br>[ slides I notes ] |

# Where we're going

GPT = Auto-regressive probabilistic model

- **Directed PGM**




$$P_\theta(X) = \prod_i \prod_t P_\theta(X_{i,t} \mid X_{i,<t})$$

- **Probabilistic objective:** Max log-likelihood of observed seqs

$$\max_\theta \sum_i \sum_t \log P_\theta\left( X_{i,t} \mid X_{i,<t} \right)$$

[Radford et al., Improving Language Understanding by Generative Pre-Training]

# Today

- **Two challenges with Text**
- Sequence Modeling with RNNs
- Backpropagation Through Time
- Long Short-Term Memory (LSTM)

Ben Lengerich © University of Wisconsin-Madison 2026

# Two challenges with text inputs

- Turning words into numbers
  - How should we represent text so that we can forward-propagate activations and back-propagate gradients?
- Modeling sequences
  - Can we build models that accommodate **variable-length** sequences? Can contextual information be shared across long-range sequences?

Ben Lengerich © University of Wisconsin-Madison 2026

# A classic approach: Bag-of-words

"Raw" training dataset

$\mathbf{x}^{[1]} =$ "The sun is shining"

$\mathbf{x}^{[2]} =$ "The weather is sweet"

$\mathbf{x}^{[3]} =$ "The sun is shining, the weather is sweet, and one and one is two"

$$\mathbf{y} = [0, 1, 0]$$

class labels

```
vocabulary = {
    'and': 0,
    'is': 1
    'one': 2,
    'shining': 3,
    'sun': 4,
    'sweet': 5,
    'the': 6,
    'two': 7,
    'weather': 8,
}
```

Training set as design matrix

$$\mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 2 & 3 & 2 & 1 & 1 & 1 & 2 & 1 & 1 \end{bmatrix}$$

$\mathbf{y} = [0, 1, 0]$

class labels

training

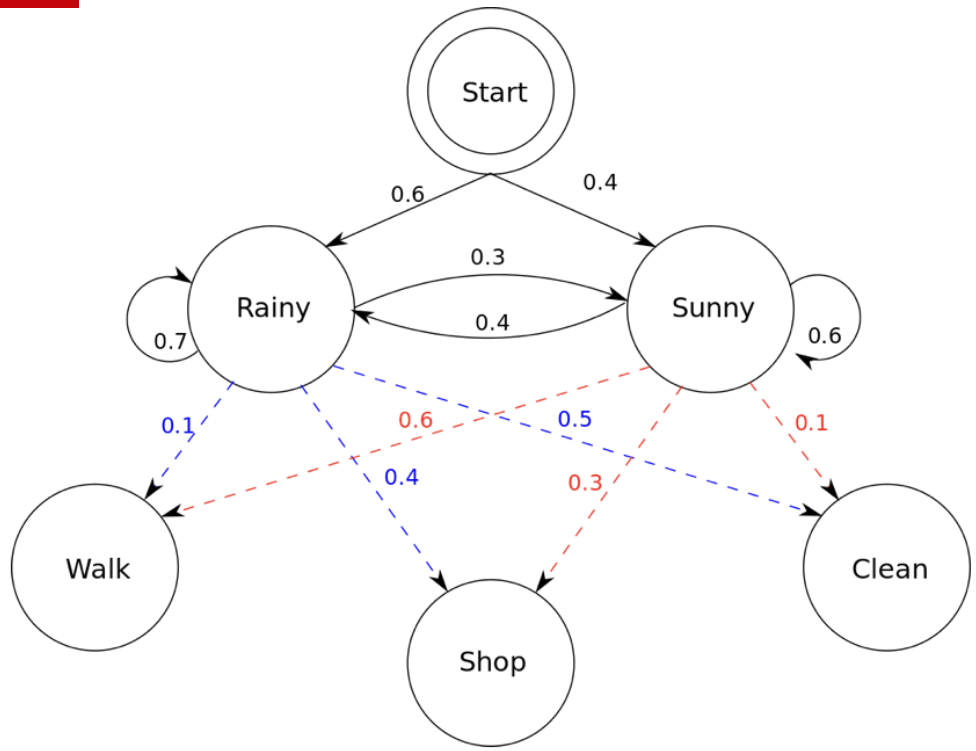
Classifier

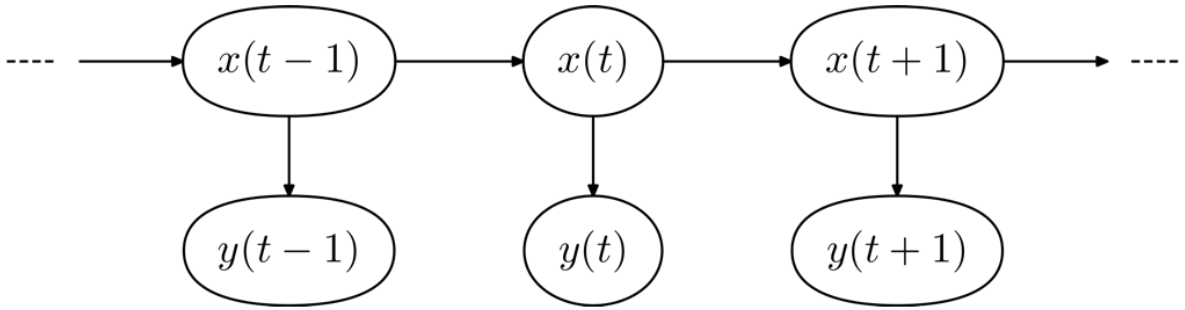
(e.g., logistic regression, MLP, ...)

Raschka & Mirjalili. Python Machine Learning 3rd Ed.
https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch08/ch08.ipynb

Simplifying assumption: Gets rid of the "sequence modeling" problem by discarding the sequential order.

# Another classic approach: Hidden Markov Model




Wikipedia example: each day, weather follows a Markov chain, and activities are observables

$$\mathbb{P}(Y_n = y \,|\, X_1 = x_1, \ldots, X_n = x_n) = \mathbb{P}(Y_n = y \,|\, X_n = x_n)$$

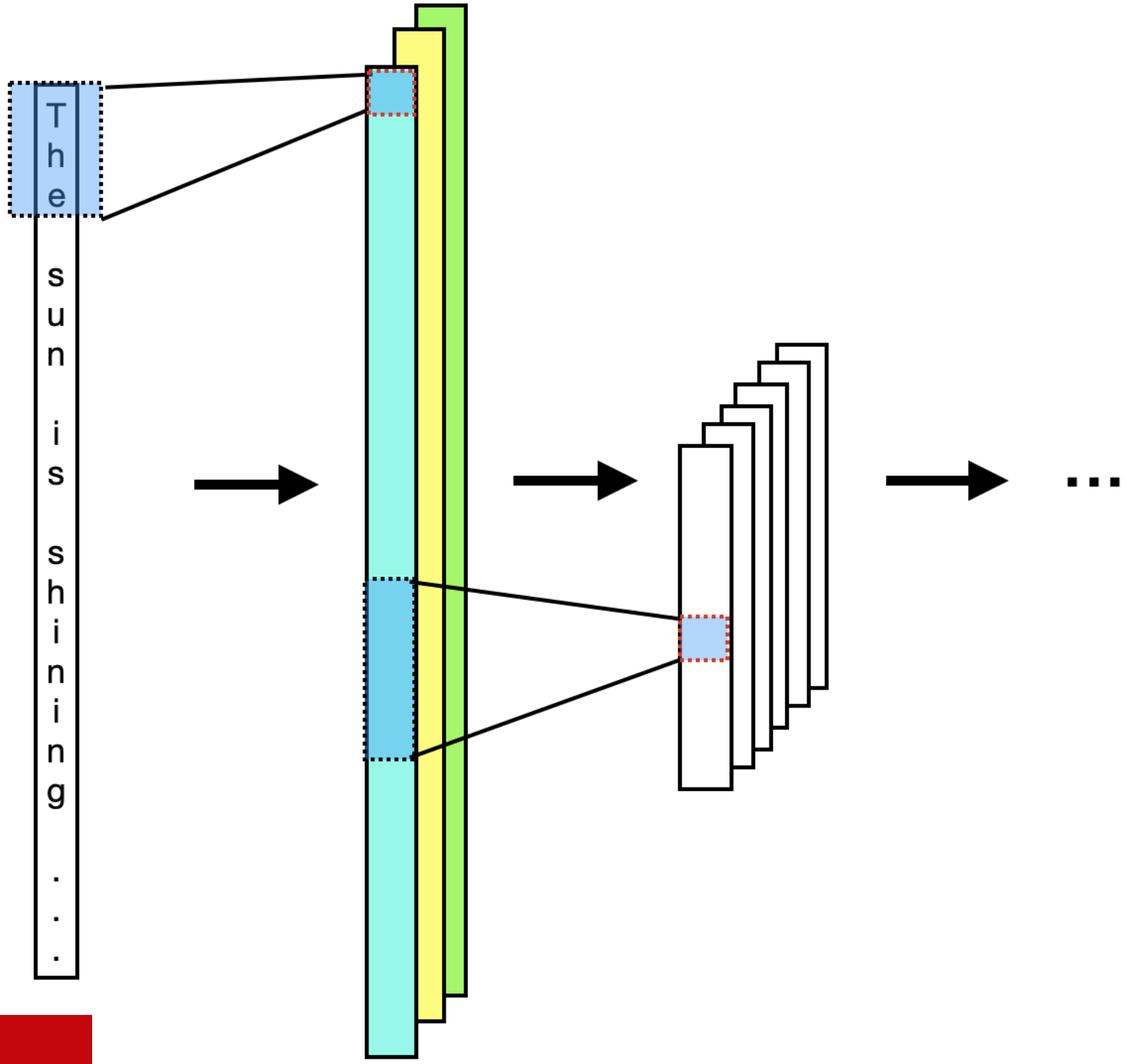
Simplifying assumption: Gets rid of the "sequence modeling" problem by assuming each state depends only on the last one.

Ben Lengerich © University of Wisconsin-Madison 2026

# Another approach: CNNs

Can’t handle variable length input,
→ need padding to max input length

Simplifying assumption: Gets rid of the “sequence modeling” problem by assuming there is a reasonably-short maximum length input.




Ben Lengerich © University of Wisconsin-Madison 2026

# **Word embeddings:** Convert words into continuous values.

Step 1: Convert words into numbers by a vocabulary (lookup table).

"Raw" training dataset

$\mathbf{x}^{[1]}$ = "The sun is shining"

$\mathbf{x}^{[2]}$ = "The weather is sweet"

$\mathbf{x}^{[3]}$ = "The sun is shining, the weather is sweet, and one and one is two"

```
vocabulary = {
    '<unk>': 0,
    'and': 1,
    'is': 2
    'one': 3,
    'shining': 4,
    'sun': 5,
    'sweet': 6,
    'the': 7,
    'two': 8,
    'weather': 9,
    '<pad>': 10
}
```

$\mathbf{x}^{[1]}$ = "The sun is shining"

```
[7   5   2   4   ...   10   10   10]
```

$\mathbf{x}^{[2]}$ = "The weather is sweet"

```
[7   9   2   6   ...   10   10   10]
```

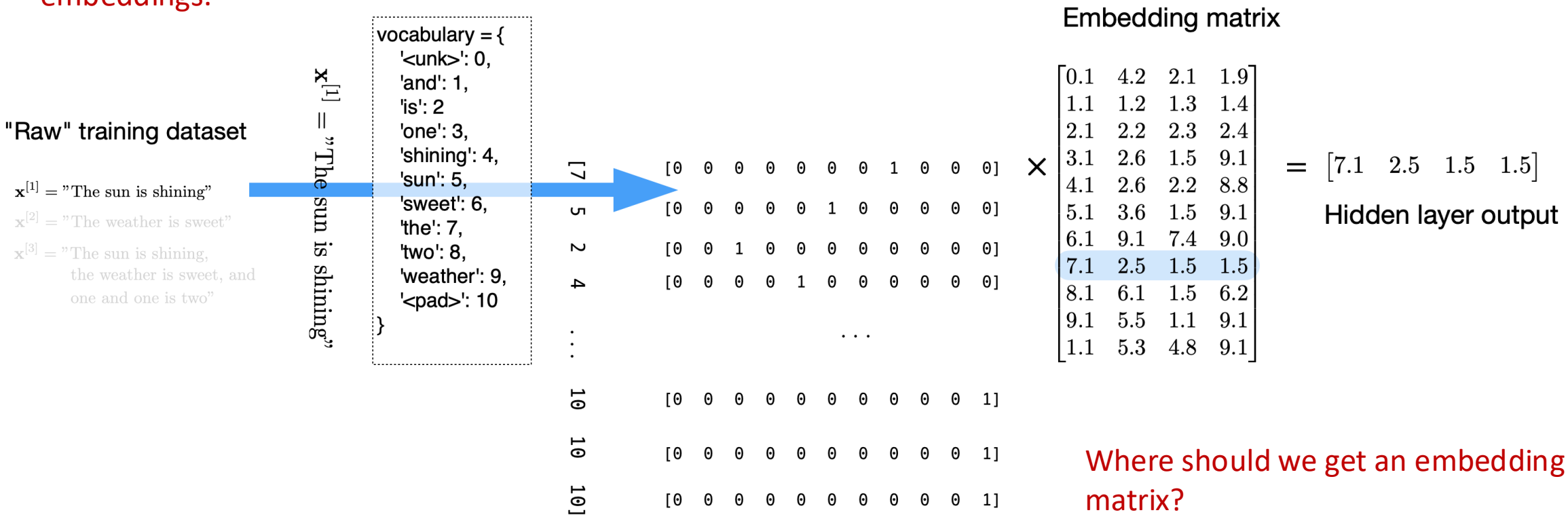
$\mathbf{x}^{[3]}$ = "The sun is shining, the weather is sweet, and one and one is two"

```
[7   5   2   4   ...    3    2    8]
```

Ben Lengerich © University of Wisconsin-Madison 2026

# **Word embeddings:** Convert words into continuous values.

Step 2: Convert indices to one-hot embeddings, and then dense embeddings.




Where should we get an embedding matrix?

Ben Lengerich © University of Wisconsin-Madison 2026

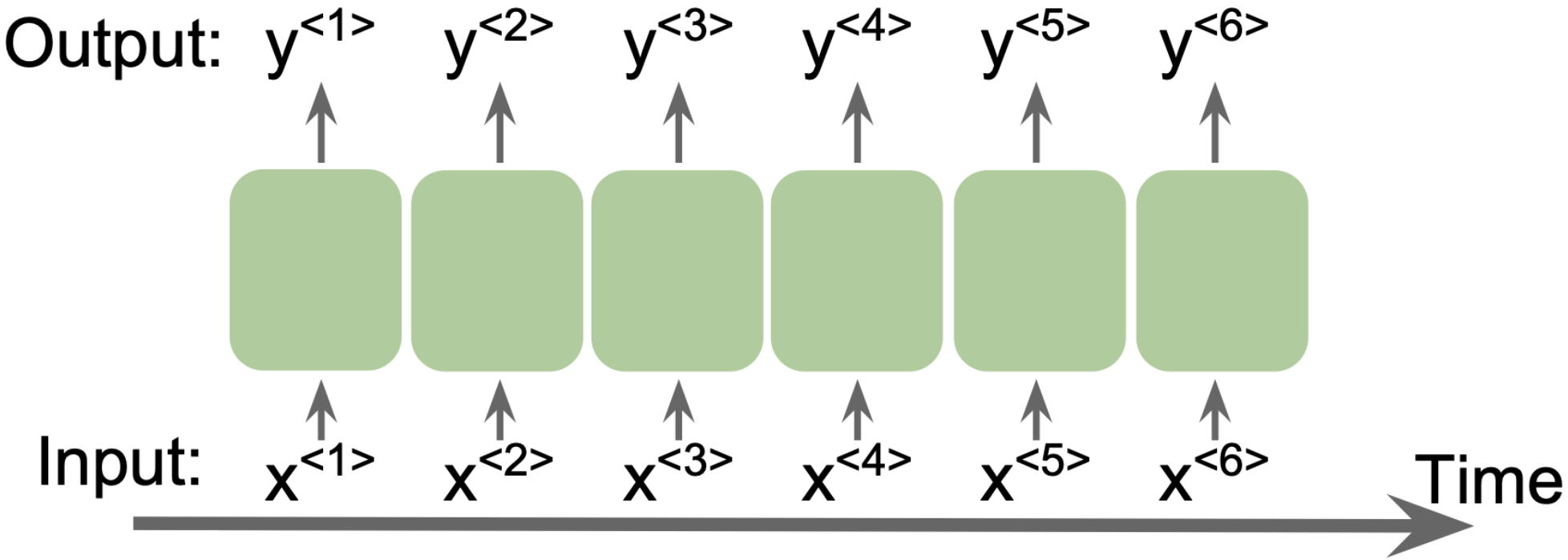
# Today

- Two challenges with Text
- **Sequence Modeling with RNNs**
- Backpropagation Through Time
- Long Short-Term Memory (LSTM)

Ben Lengerich © University of Wisconsin-Madison 2026

# Sequence data: order matters

The movie my friend has **not** seen is good
The movie my friend has seen is **not** good

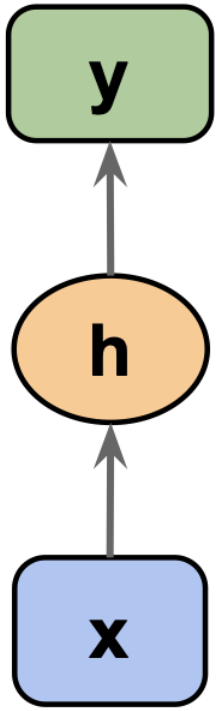



Ben Lengerich © University of Wisconsin-Madison 2026

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Recurrent Neural Networks (RNNs)

Networks we used previously: also called feedforward neural networks




Recurrent Neural Network (RNN)

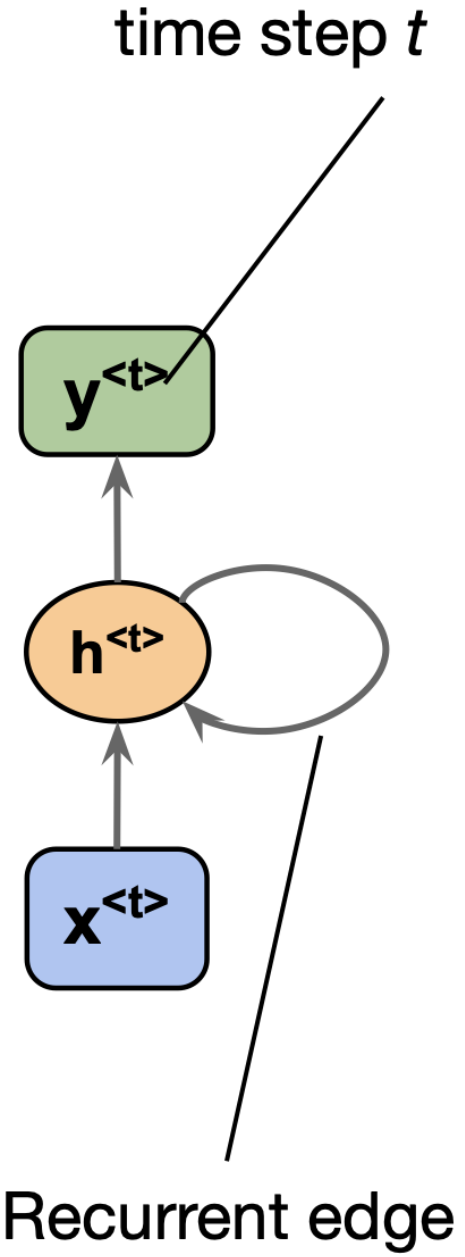



Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Recurrent Neural Networks (RNNs)

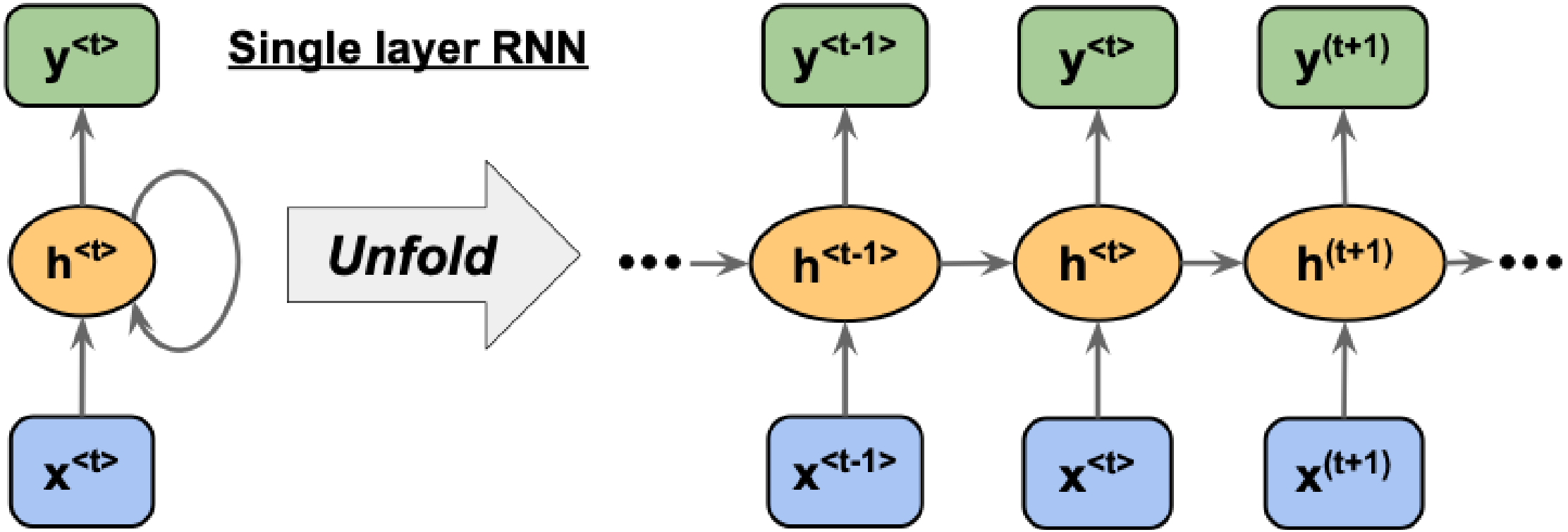



Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Recurrent Neural Networks (RNNs)

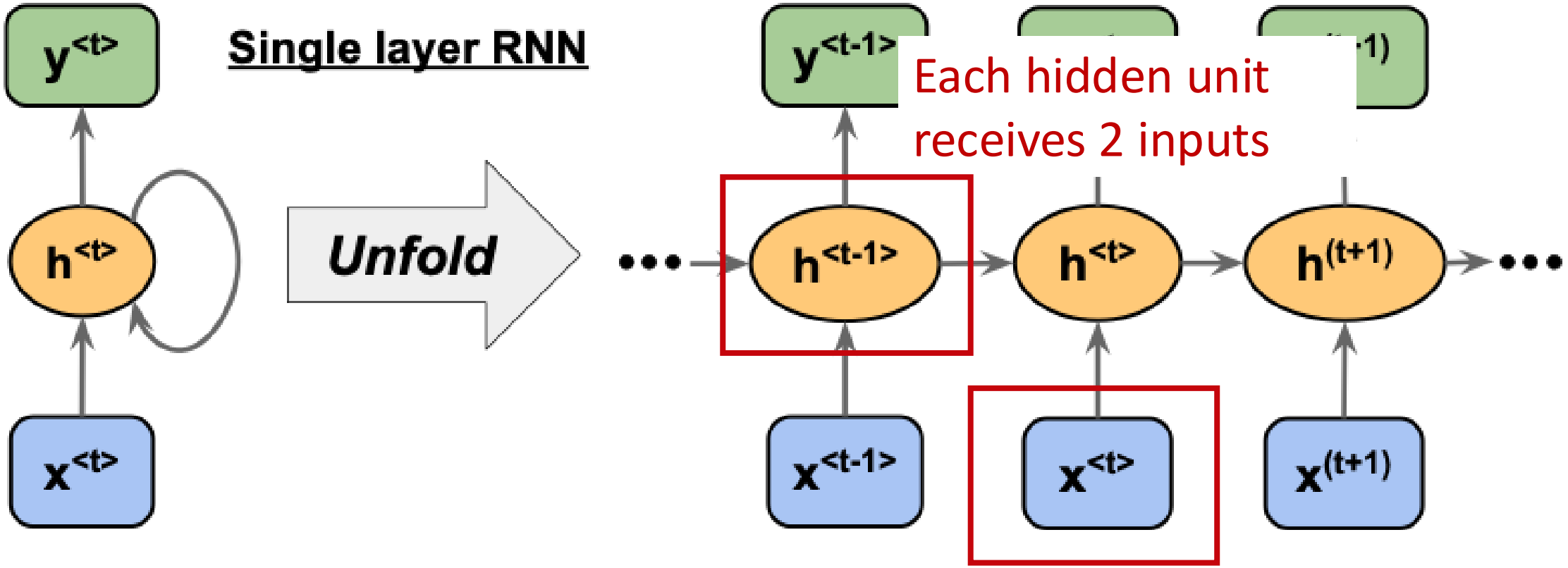



Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Multilayer RNNs

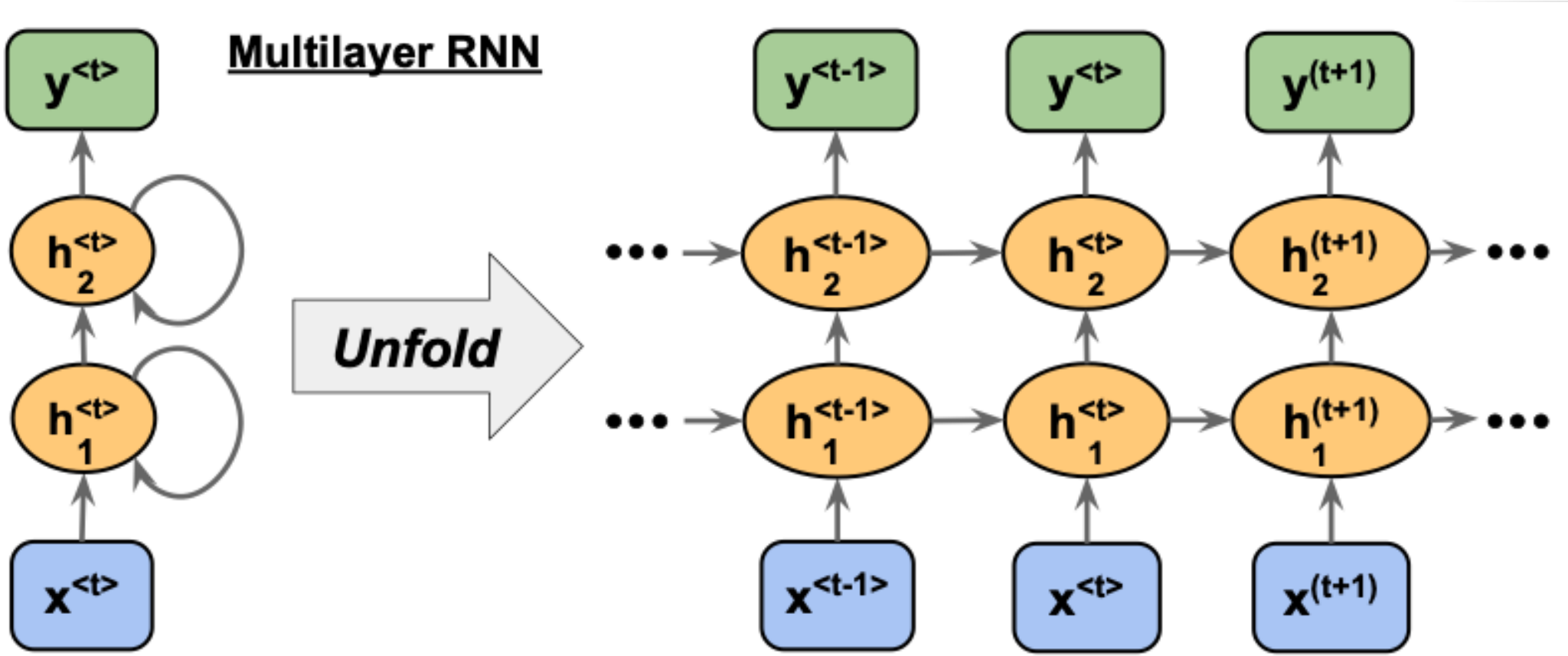
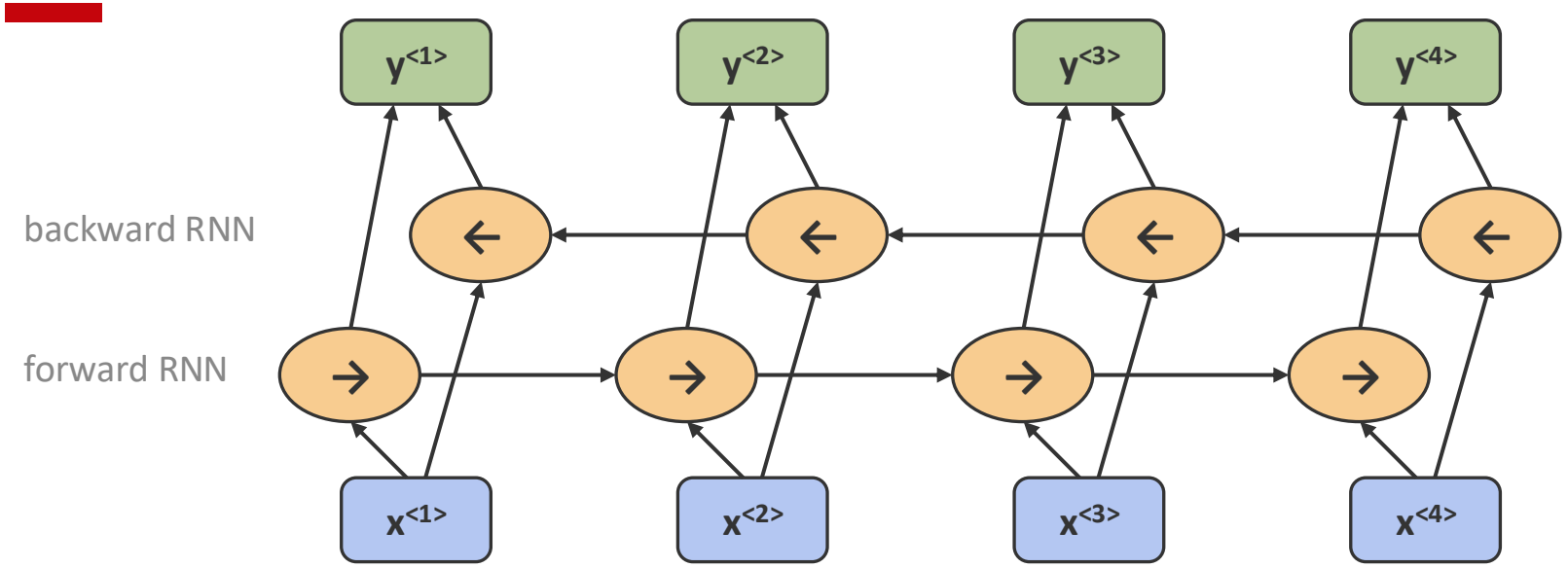



Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Bidirectional RNNs



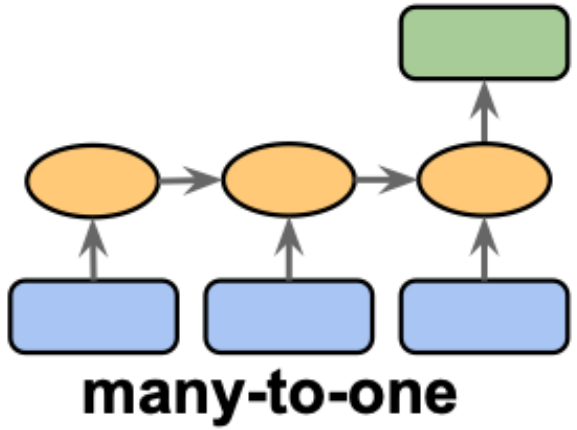

$$\mathbf{h}^{\langle t \rangle} = \left[ \overrightarrow{\mathbf{h}}^{\langle t \rangle} ; \overleftarrow{\mathbf{h}}^{\langle t \rangle} \right]$$

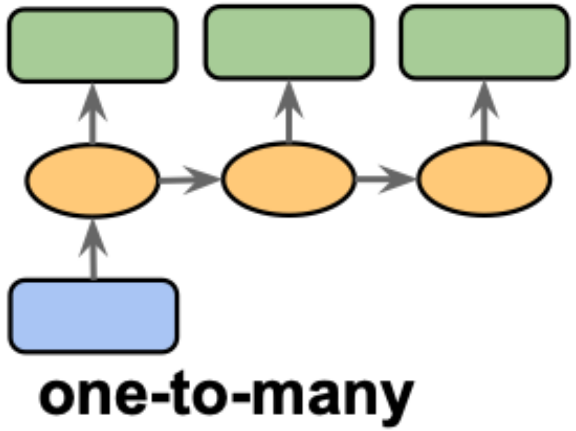
concatenate both directions

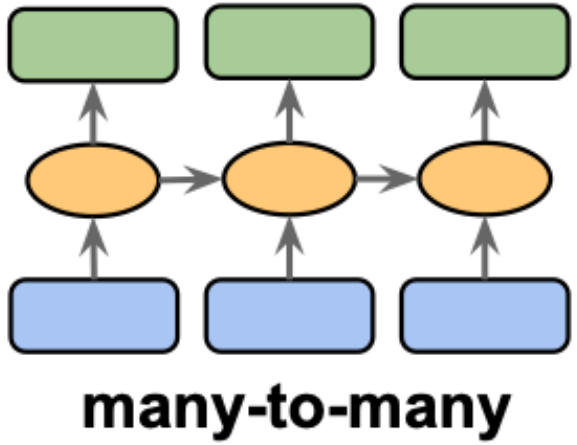
- Run one RNN forward and a second RNN backward, then **concatenate** their hidden states at each step
- Every output sees the **whole sequence**, past and future: “The movie my friend has **not** seen is good”
- Great for **encoding** an input you already have: tagging, classification, the encoder in translation (the attention examples next lecture use one)
- Can’t be used to **generate** text: the future tokens haven’t been written yet

Ben Lengerich © University of Wisconsin-Madison 2026

# Recurrence unlocks many types of sequence tasks



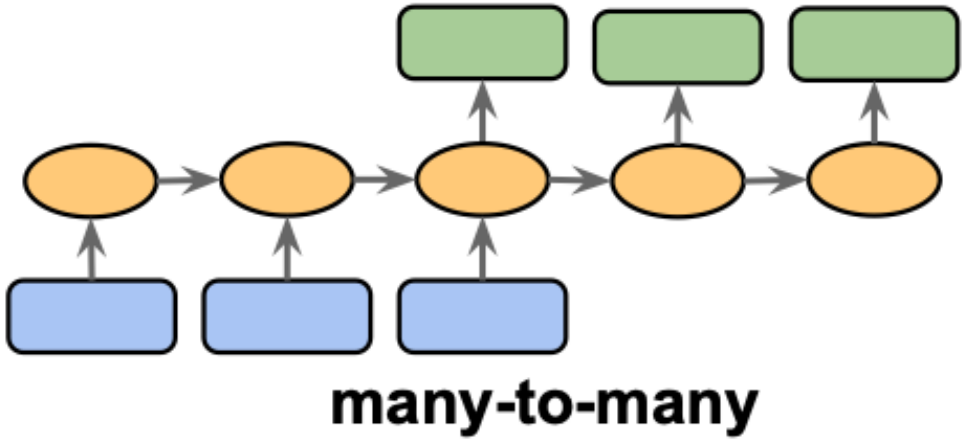
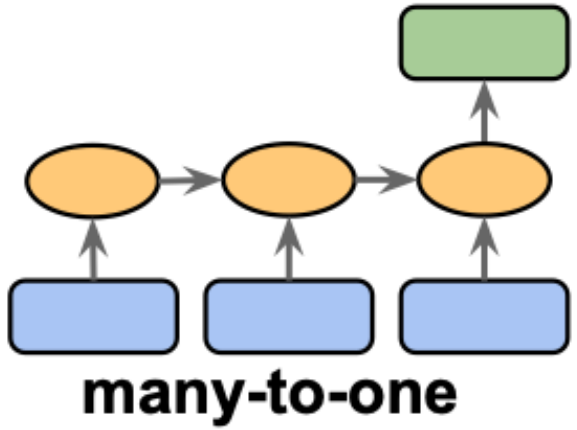




Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Recurrence unlocks many types of sequence tasks



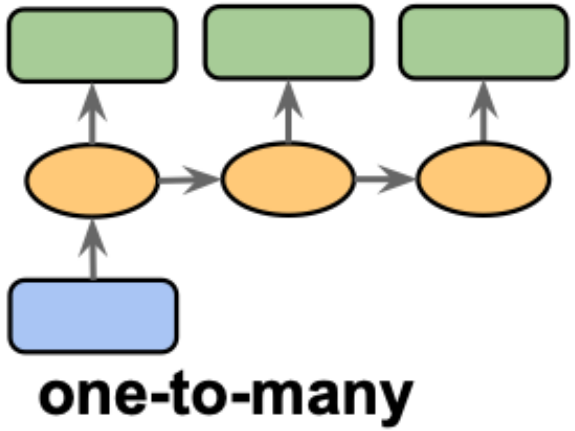

**Many-to-one:** The input data is a sequence, but the output is a fixed-size vector, not a sequence.

**Example:** sentiment analysis, the input is some text, and the output is a class label.

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# Recurrence unlocks many types of sequence tasks




**One-to-many:** Input data is in a standard format (not a sequence), the output is a sequence.

**Example:** Image captioning, where the input is an image, the output is a text description of that image

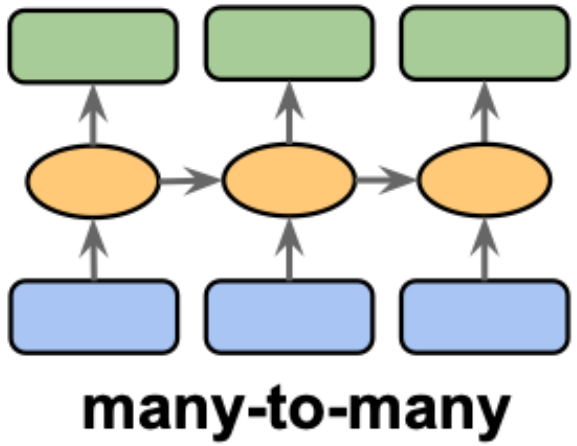
Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# Recurrence unlocks many types of sequence tasks

**Many-to-many:** Both inputs and outputs are sequences. Can be direct or delayed.

**Example:** Video-captioning, i.e., describing a sequence of images via text (direct). Translation.



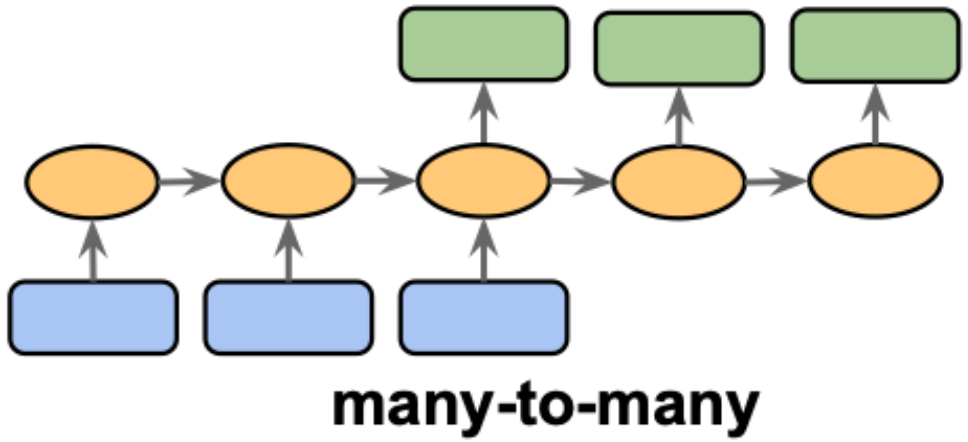
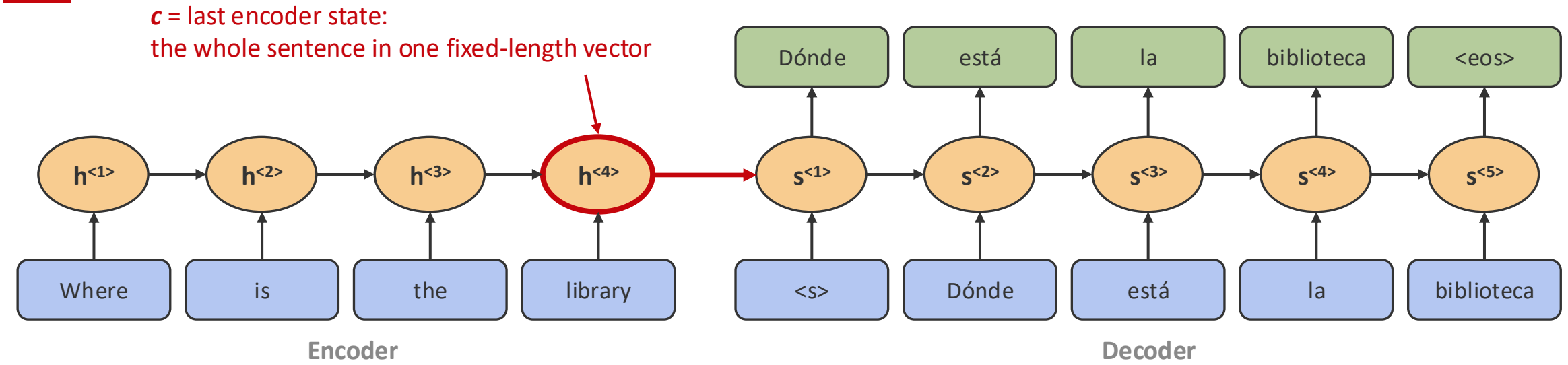




Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# Encoder–decoder (seq2seq)




- **Encoder** RNN reads the whole input; its last hidden state *c* summarizes the sentence
- **Decoder** RNN starts from *c* and writes the output one token at a time, feeding each output back in as the next input
- Input and output lengths can differ: translation, summarization, question answering (Sutskever et al. 2014; Cho et al. 2014)
- **Bottleneck:** everything must squeeze into one fixed-length vector *c*. Long sentences suffer. Next lecture: attention.

Ben Lengerich © University of Wisconsin-Madison 2026

# Today

- Two challenges with Text
- Sequence Modeling with RNNs
- **Backpropagation Through Time**
- Long Short-Term Memory (LSTM)

Ben Lengerich © University of Wisconsin-Madison 2026

# Under the hood: weight matrices in an RNN

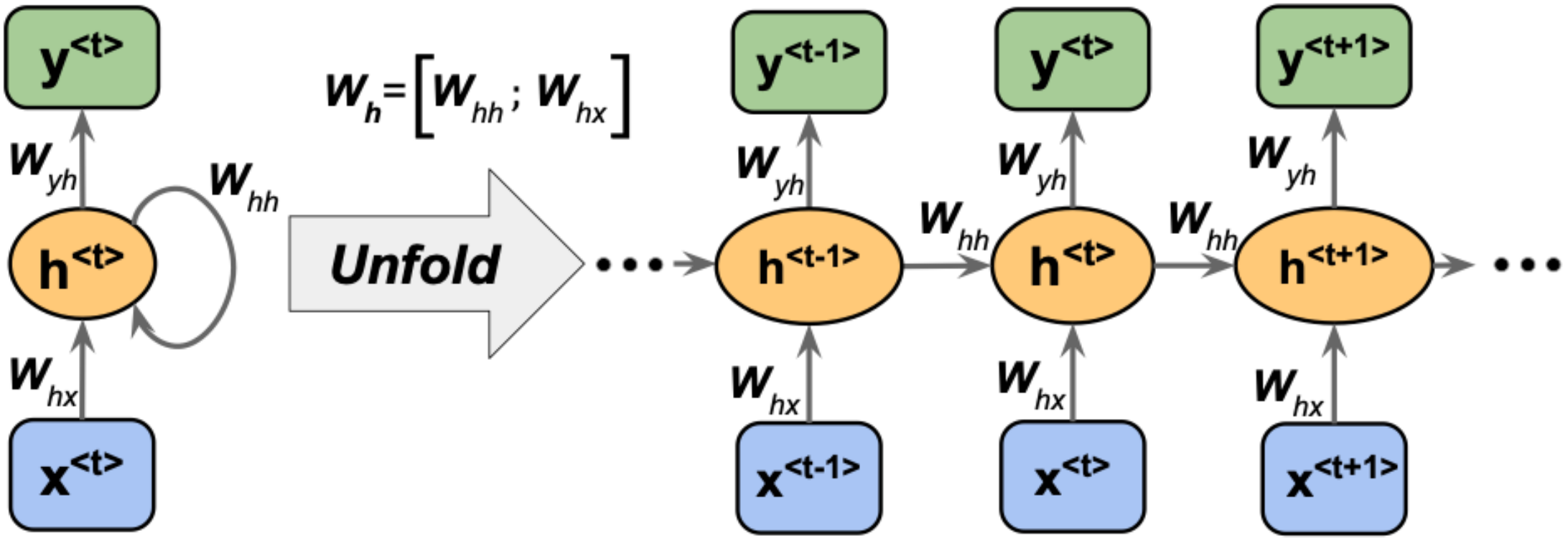
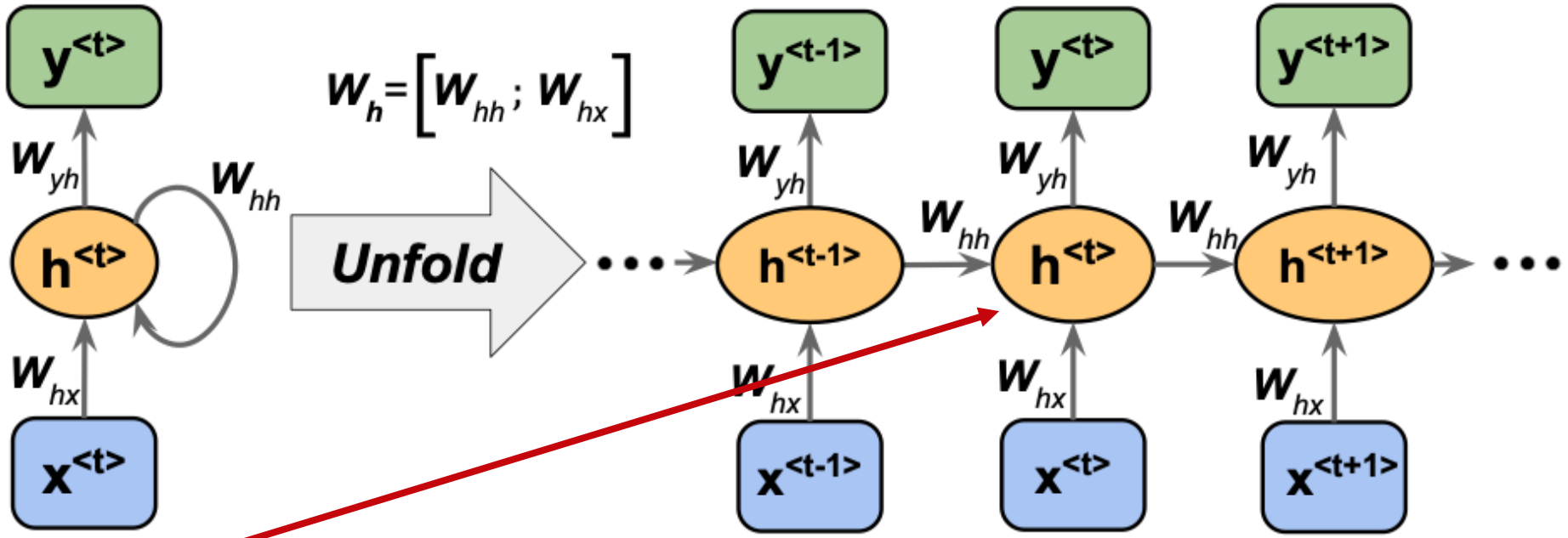



Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Under the hood: weight matrices in an RNN

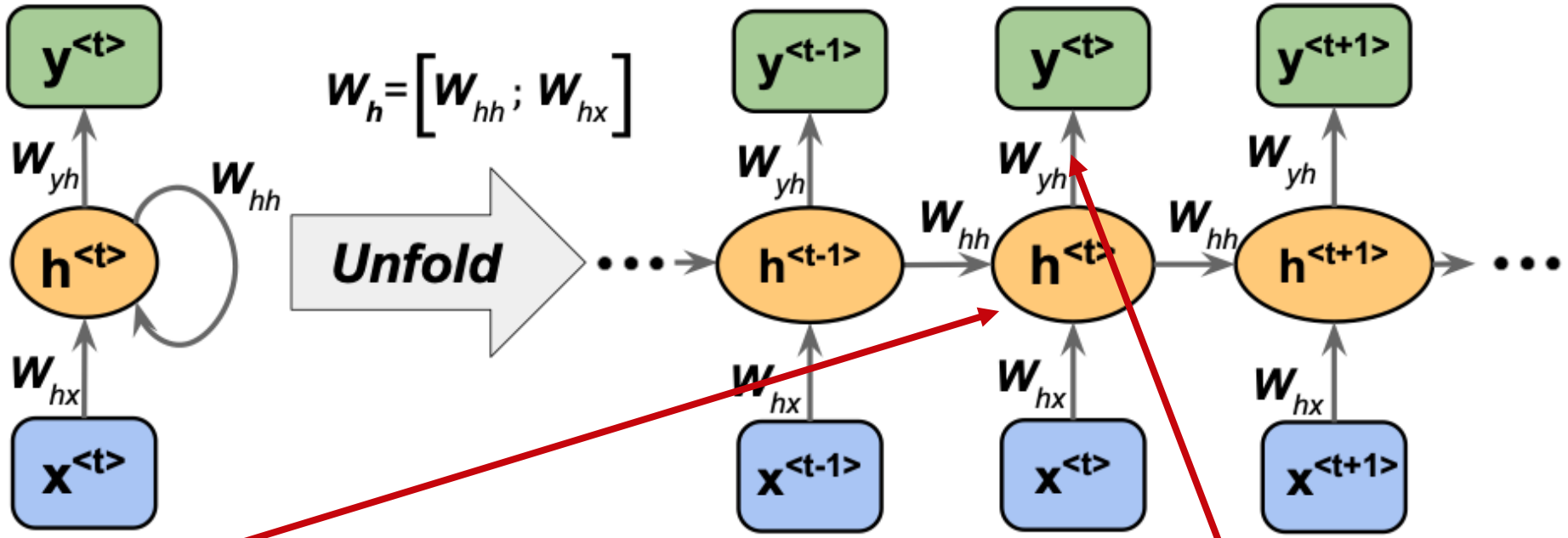



Net input: $$\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$$

Activation: $$\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$$

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Under the hood: weight matrices in an RNN




Net input: $\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$

Net input: $\mathbf{z}_y^{\langle t \rangle} = \mathbf{W}_{yh}\mathbf{h}^{\langle t \rangle} + \mathbf{b}_y$

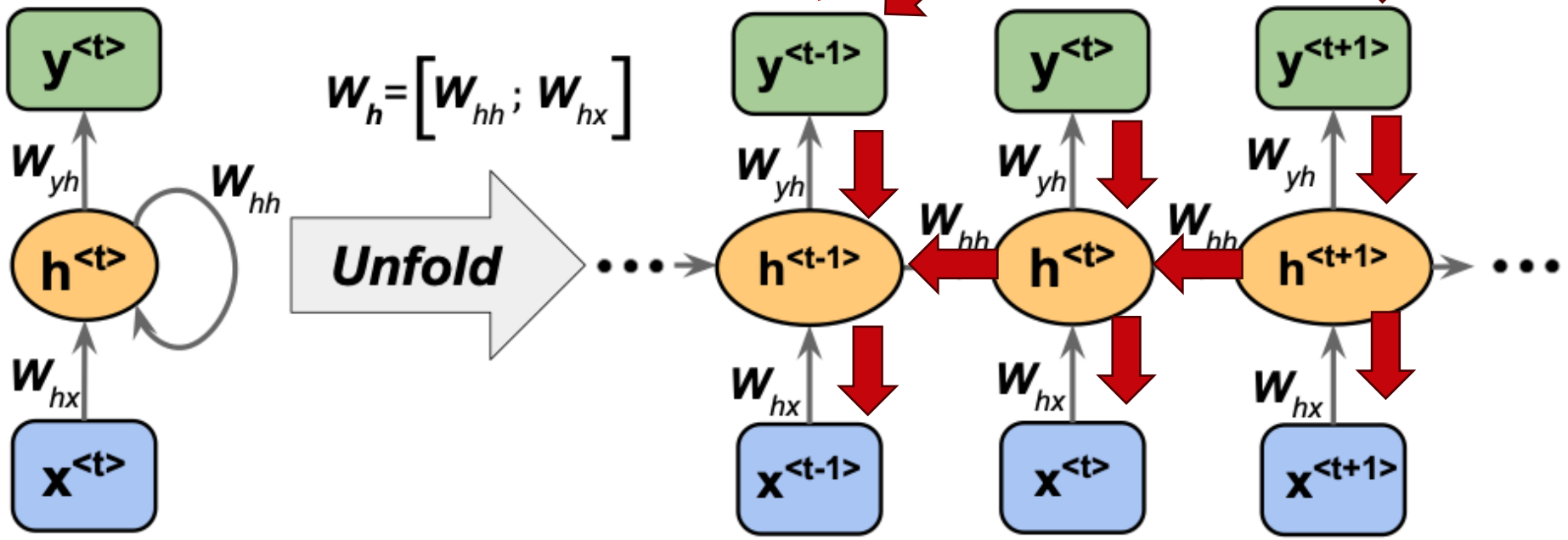
Activation: $\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$

Output: $\mathbf{y}^{\langle t \rangle} = \sigma_y\left(\mathbf{z}_y^{\langle t \rangle}\right)$

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Backpropagation through time

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

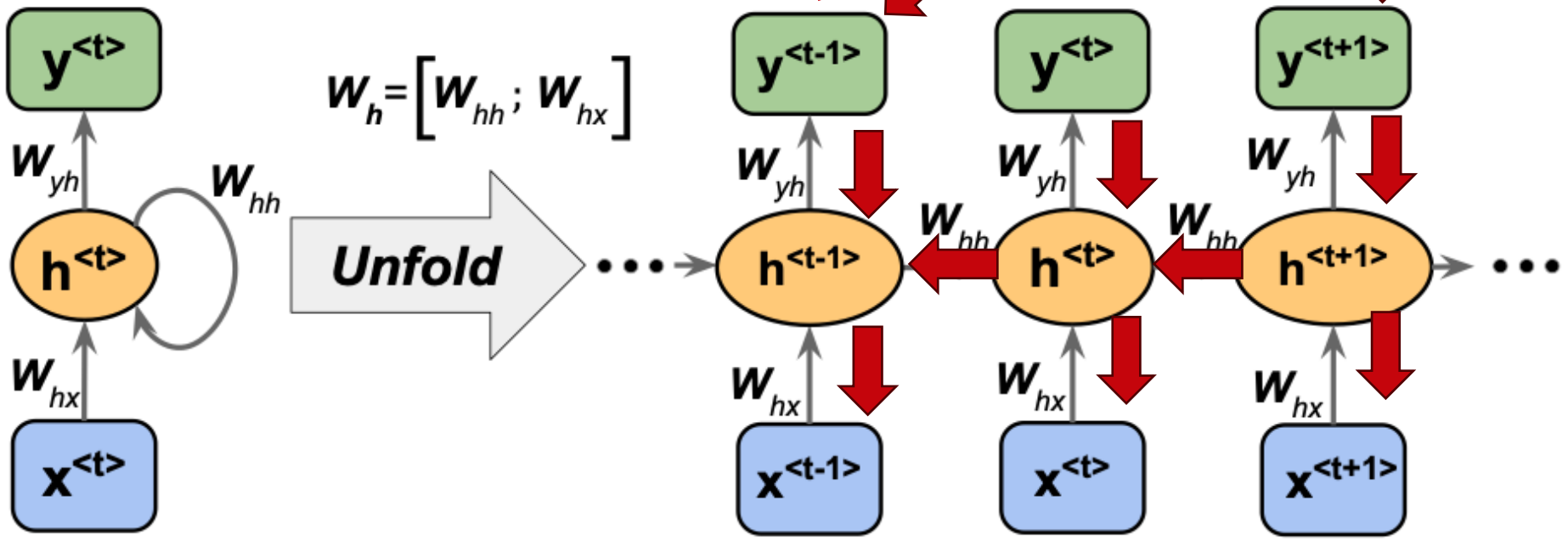



The overall loss can be computed as the sum over all time steps

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Backpropagation through time

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

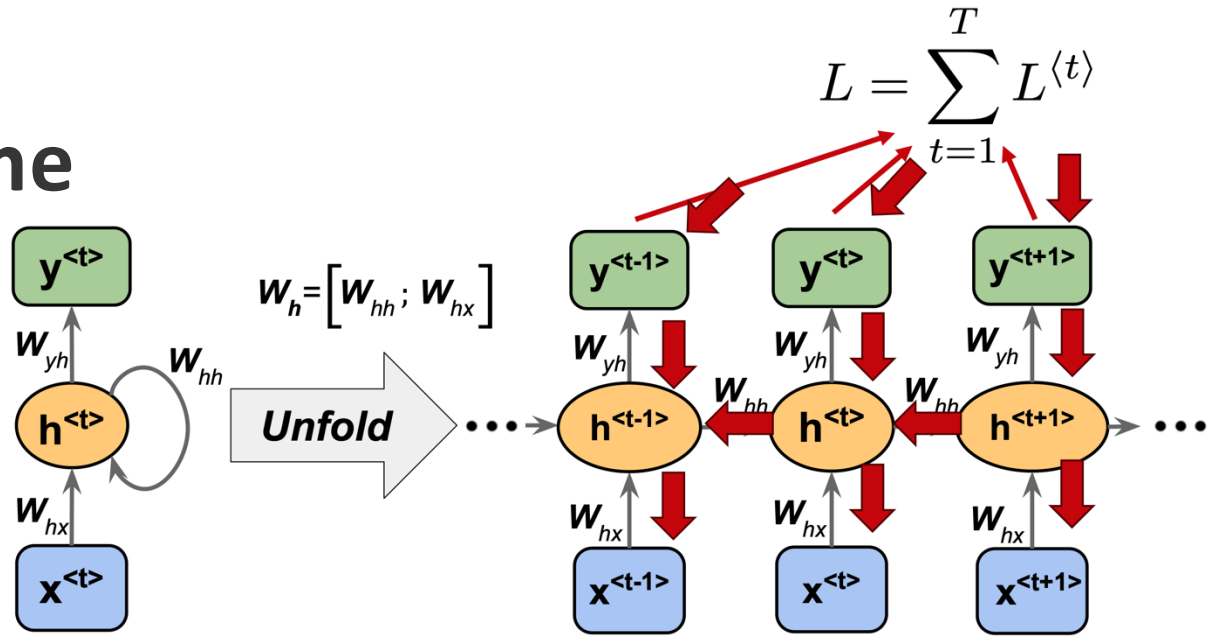



$$L = \sum_{t=1}^{T} L^{(t)}$$

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

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Backpropagation through time




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

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

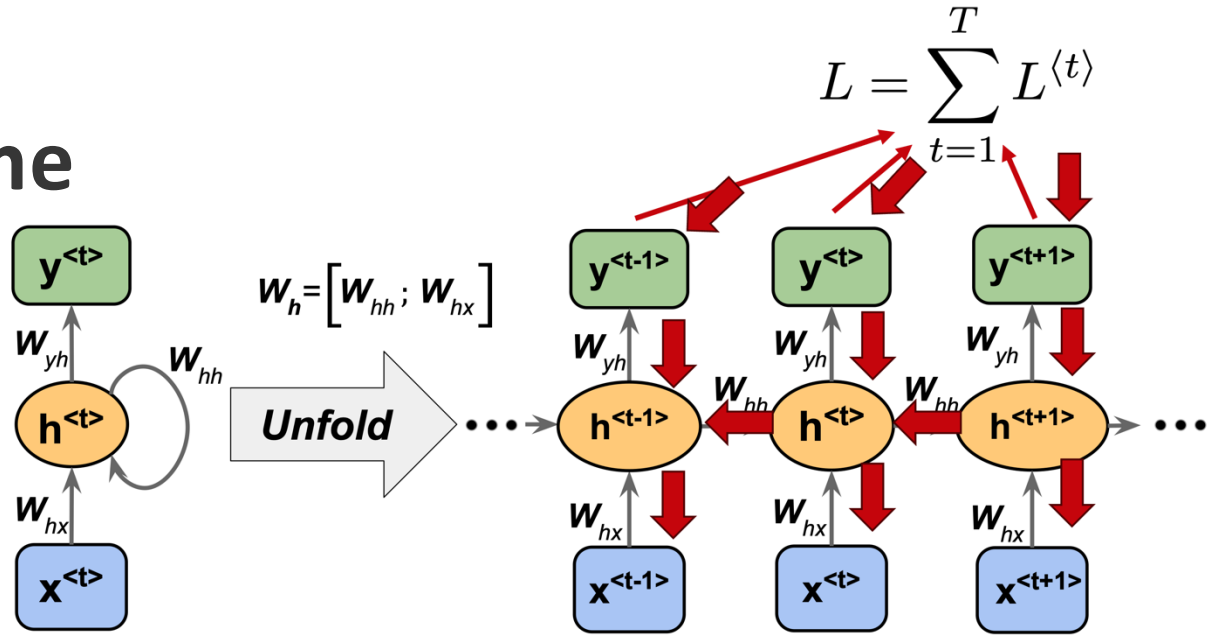
Computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Backpropagation through time

Straightforward, but problematic for long chains: vanishing / exploding gradients!




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

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

Computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Packt, 2019

# Solutions to Vanishing / Exploding Gradients

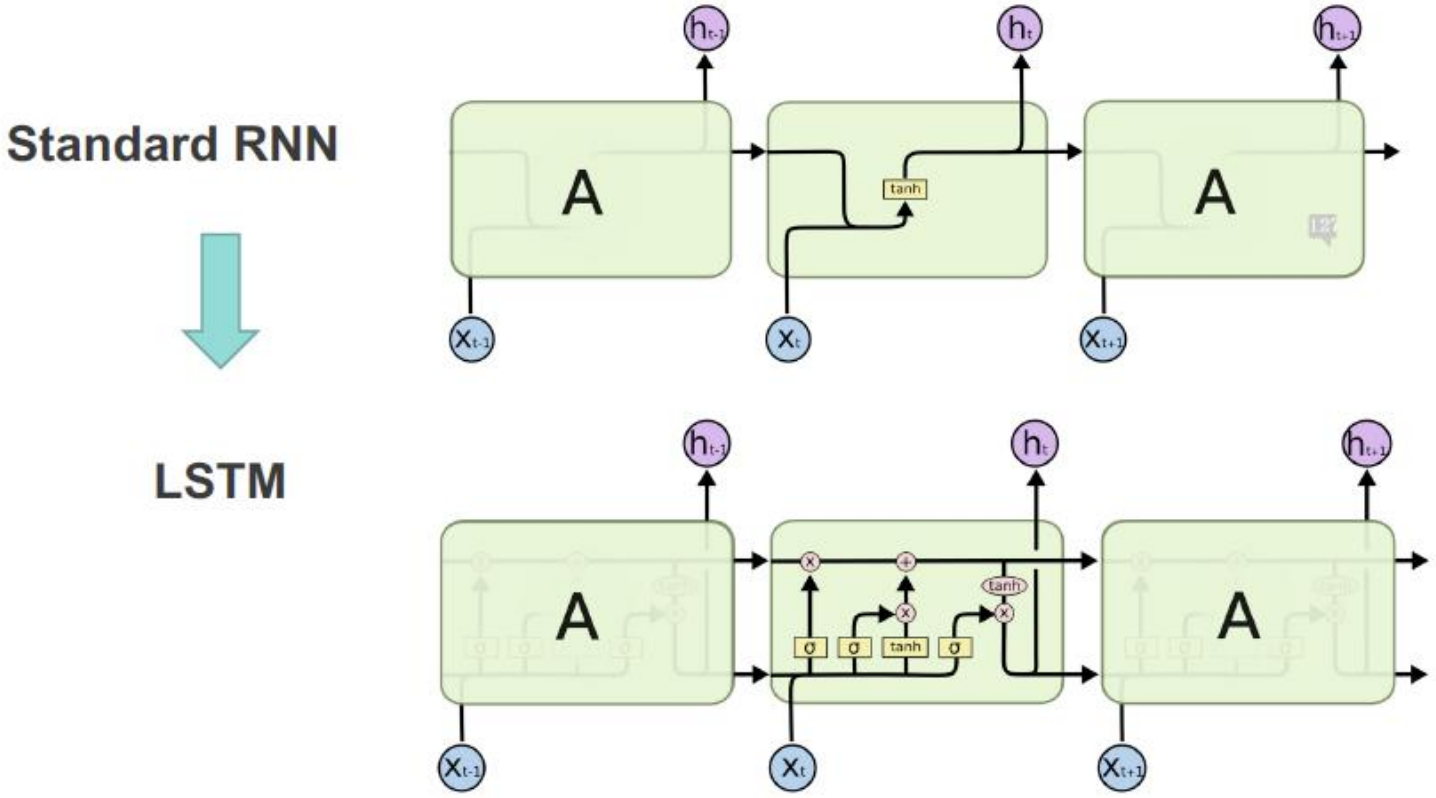
- **Gradient Clipping:** if the gradient's norm exceeds a threshold, rescale it down to that norm (fixes only exploding gradients, not vanishing ones)

- **Truncated backpropagation through time (TBPTT):** limit the number of time steps the signal can backpropagate after each forward pass.
  - E.g., even if the sequence has 100 elements/steps, we may only backpropagate through 20 or so.

Ben Lengerich © University of Wisconsin-Madison 2026

# Solutions to Vanishing / Exploding Gradients

**Long short-term memory (LSTM):** uses a *memory cell* for modeling long-range dependencies and mitigates vanishing gradients



# Today

- Two challenges with Text
- Sequence Modeling with RNNs
- Backpropagation Through Time
- **Long Short-Term Memory (LSTM)**

Ben Lengerich © University of Wisconsin-Madison 2026

# Long short-term memory (LSTM)

- Not an oxymoron: **2 paths** of memory

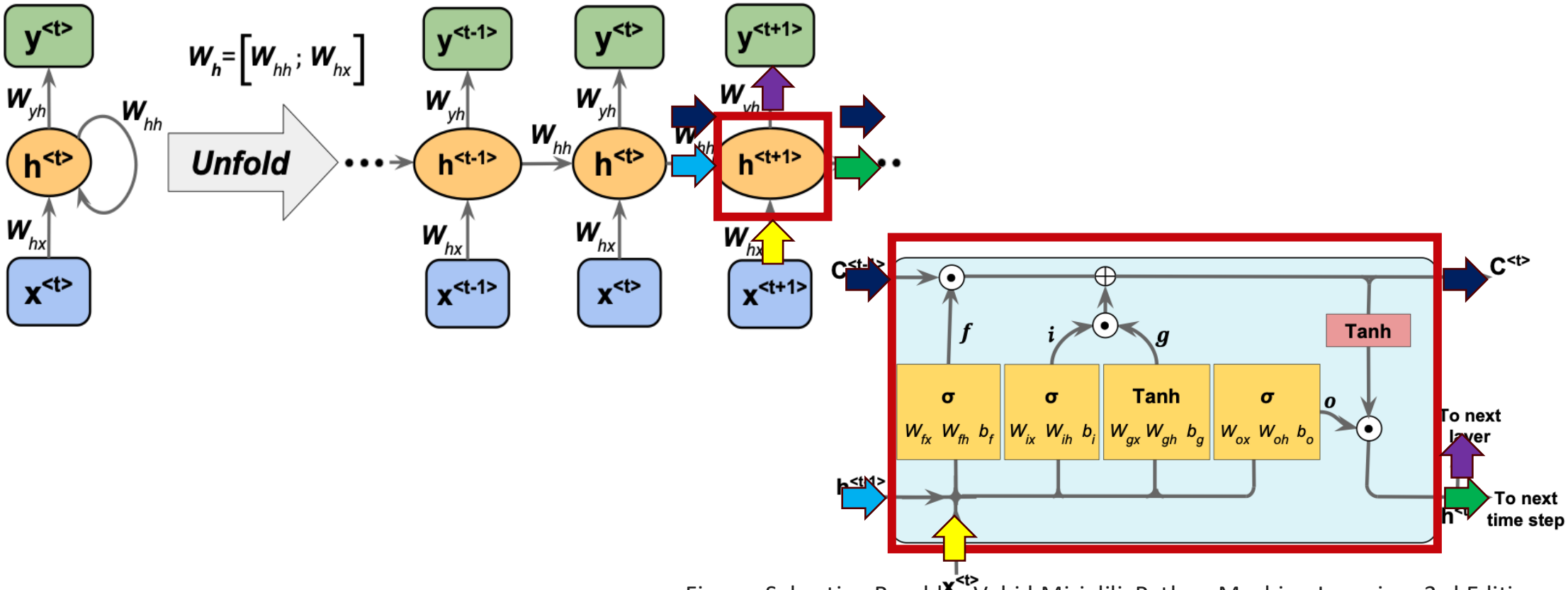



Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

# Long short-term memory (LSTM)

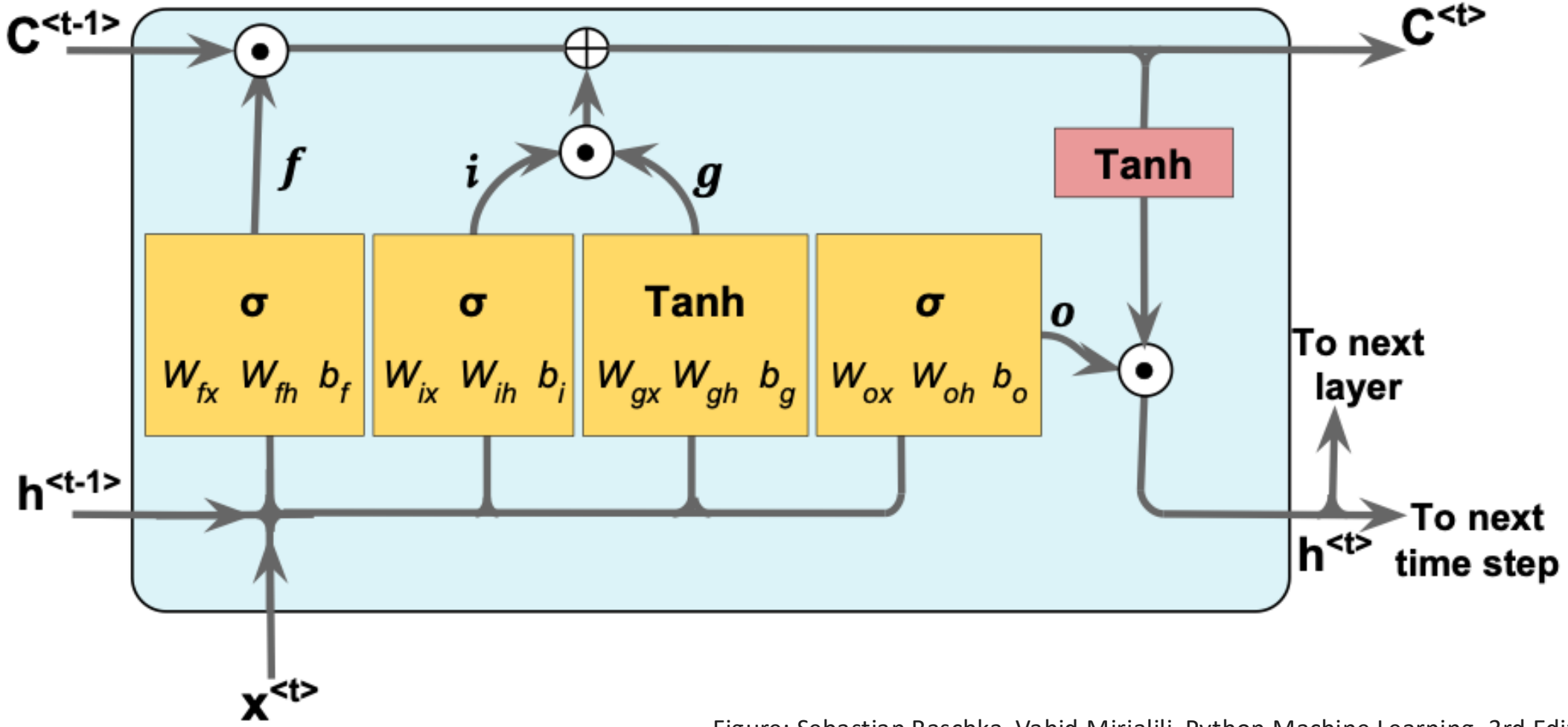



Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# Inside LSTM

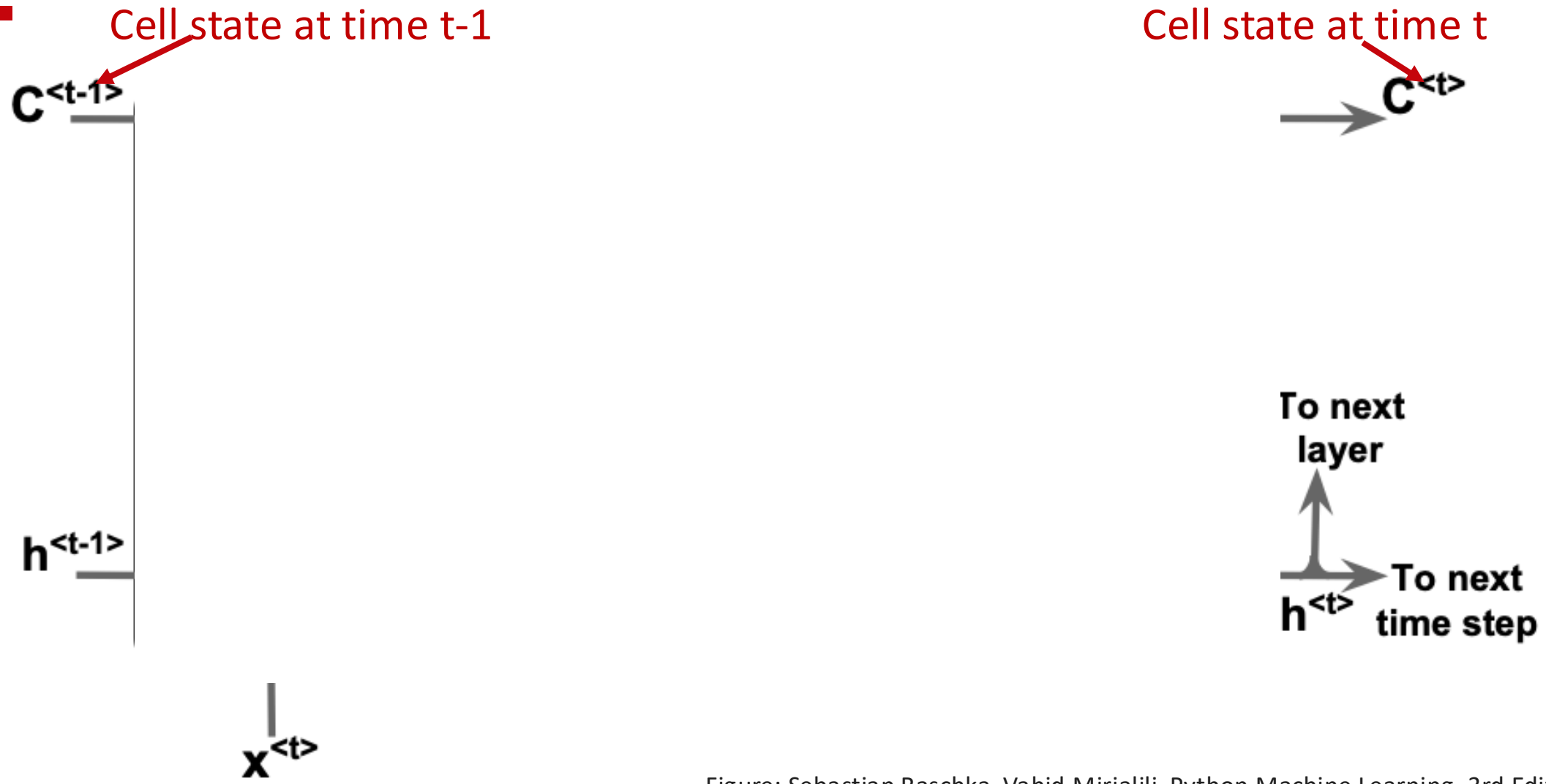
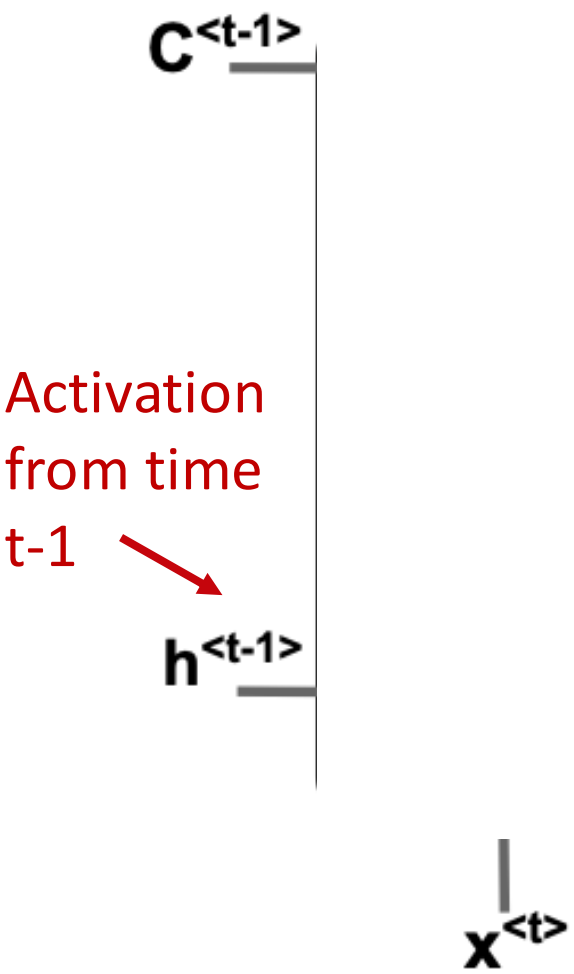



Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# Inside LSTM



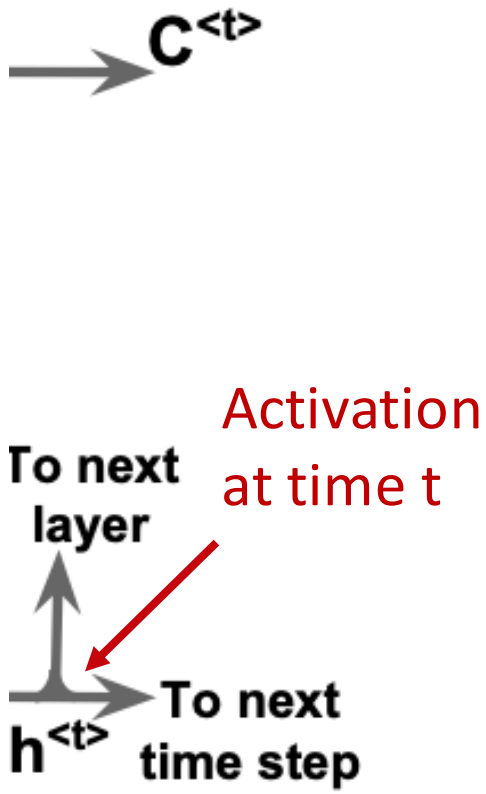




Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# Inside LSTM

"Forget gate": controls which information is remembered and which is forgotten

$$f_t = \sigma\left(\mathbf{W}_{fx}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{fh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_f\right)$$

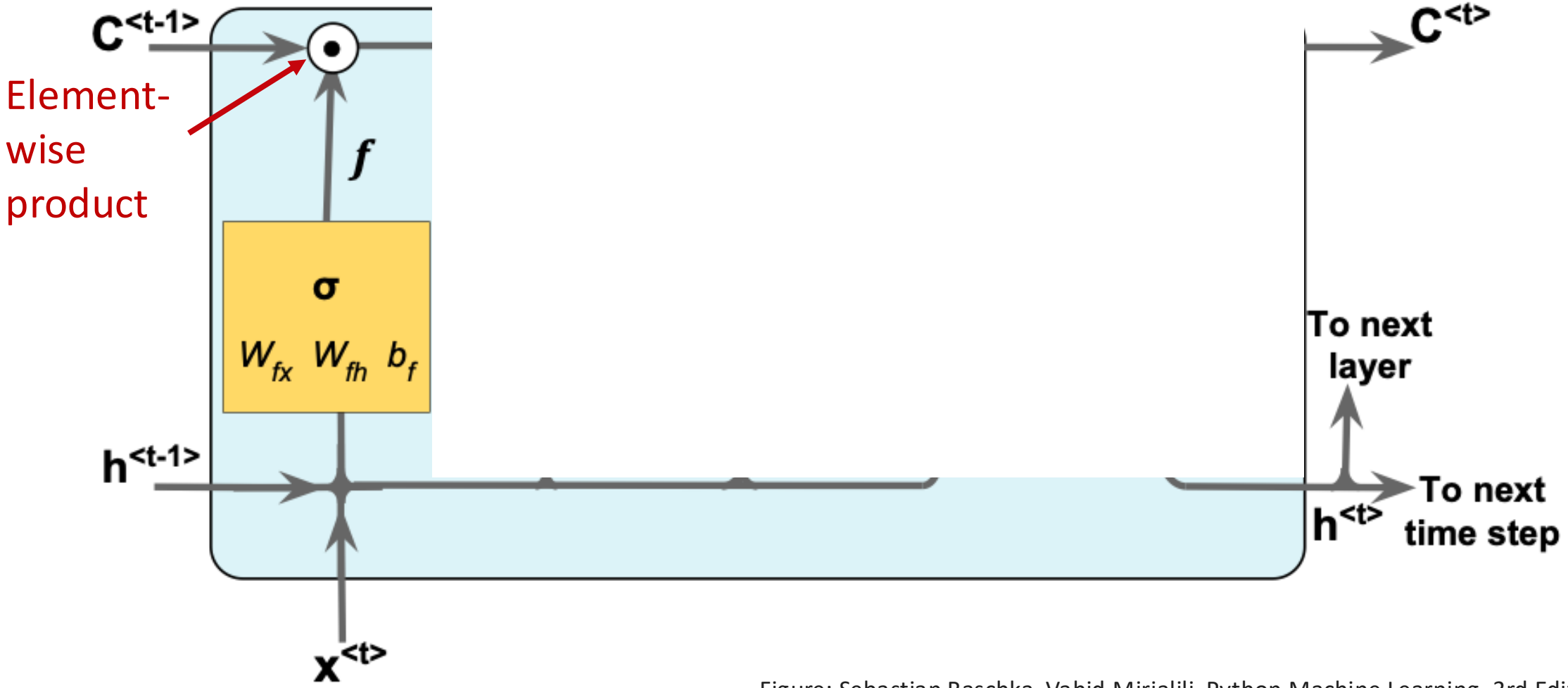



Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# Inside LSTM

"Input gate": $\mathbf{i}_t = \sigma\left(\mathbf{W}_{ix}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{ih}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_i\right)$

"Input node": $\mathbf{g}_t = \tanh\left(\mathbf{W}_{gx}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{gh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_g\right)$

Element-wise addition

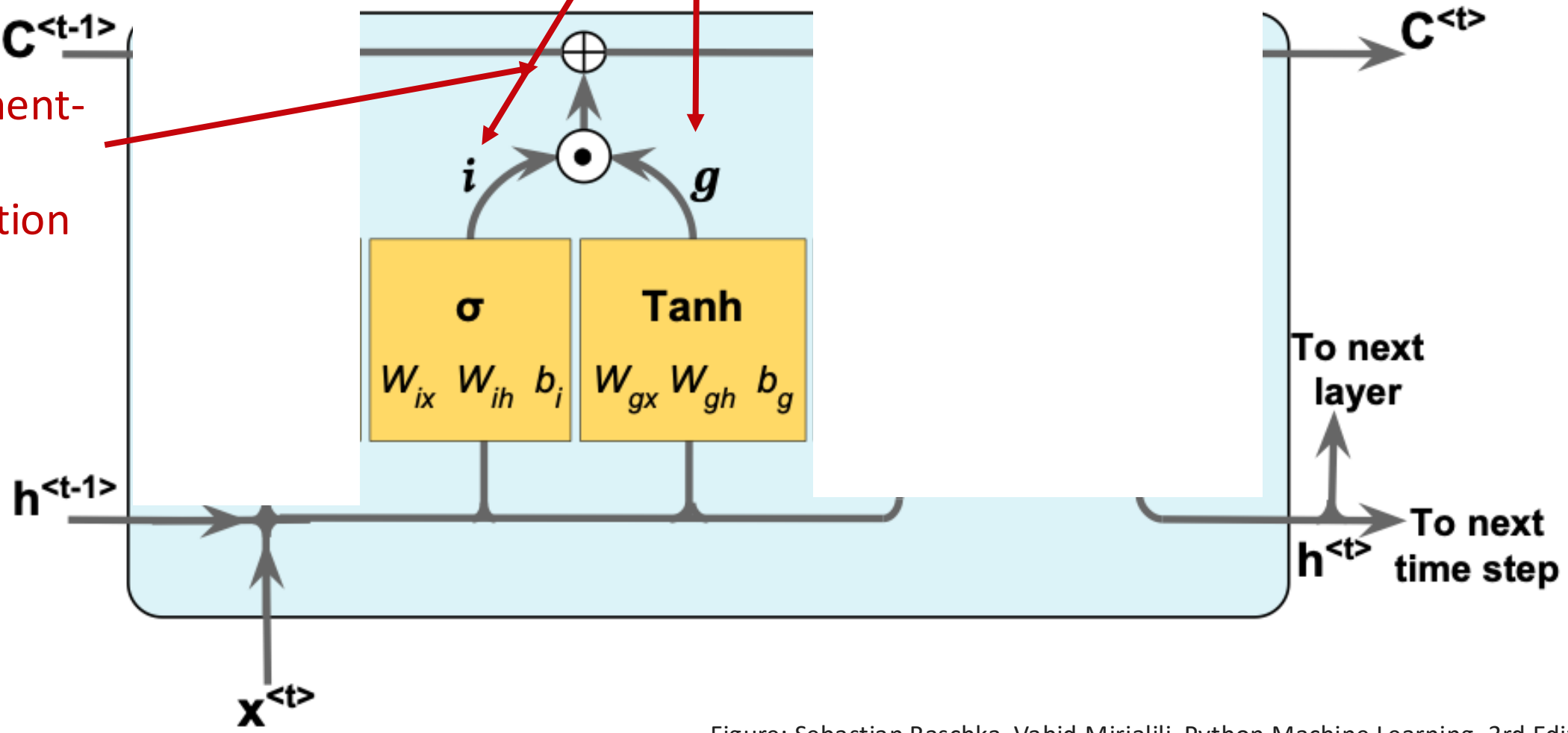



Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

# Inside LSTM

$$C^{\langle t \rangle} = \left(C^{\langle t-1 \rangle} \odot f_t\right) \oplus (i_t \odot g_t)$$

Forget Gate

Input Gate

Input Node

For updating the cell state

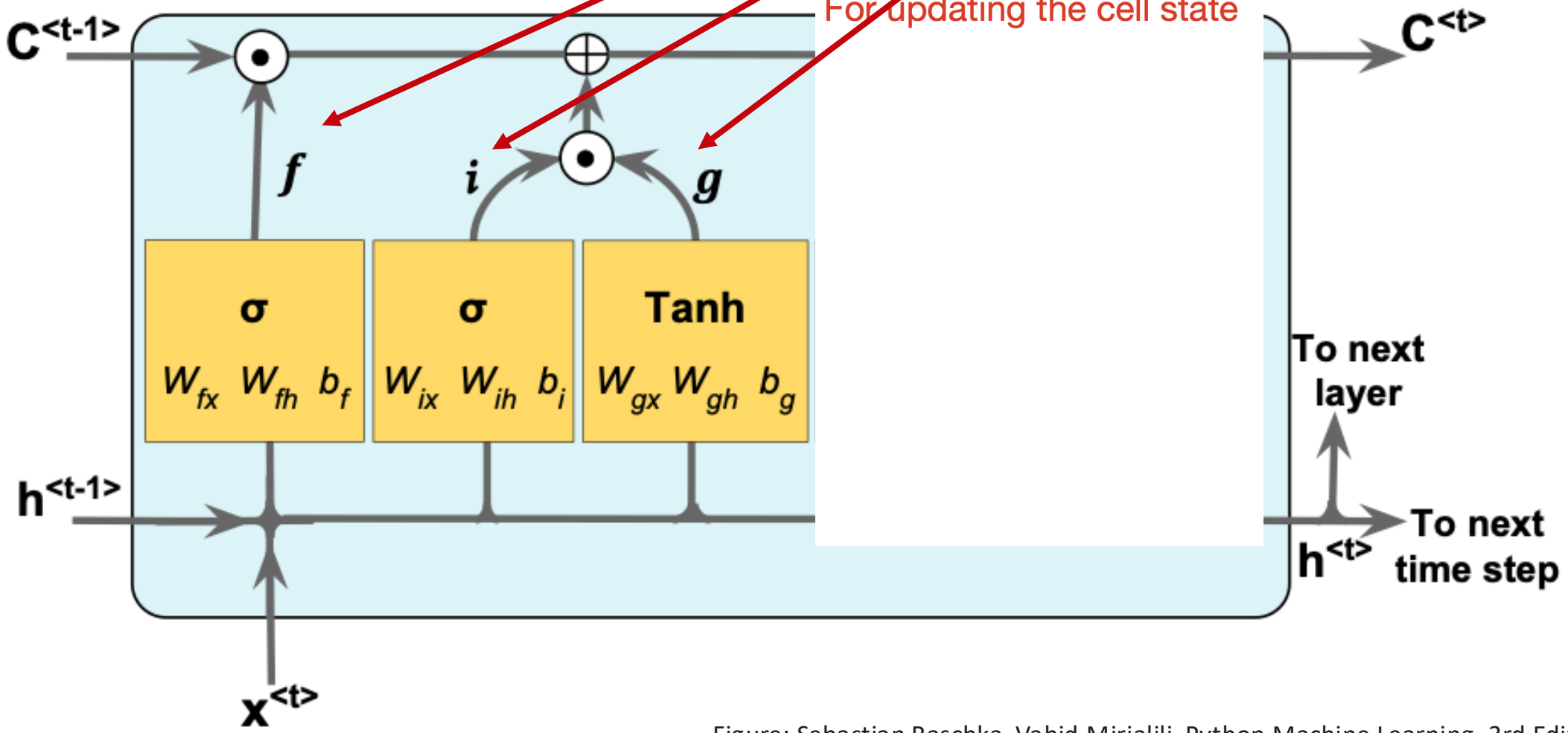



Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

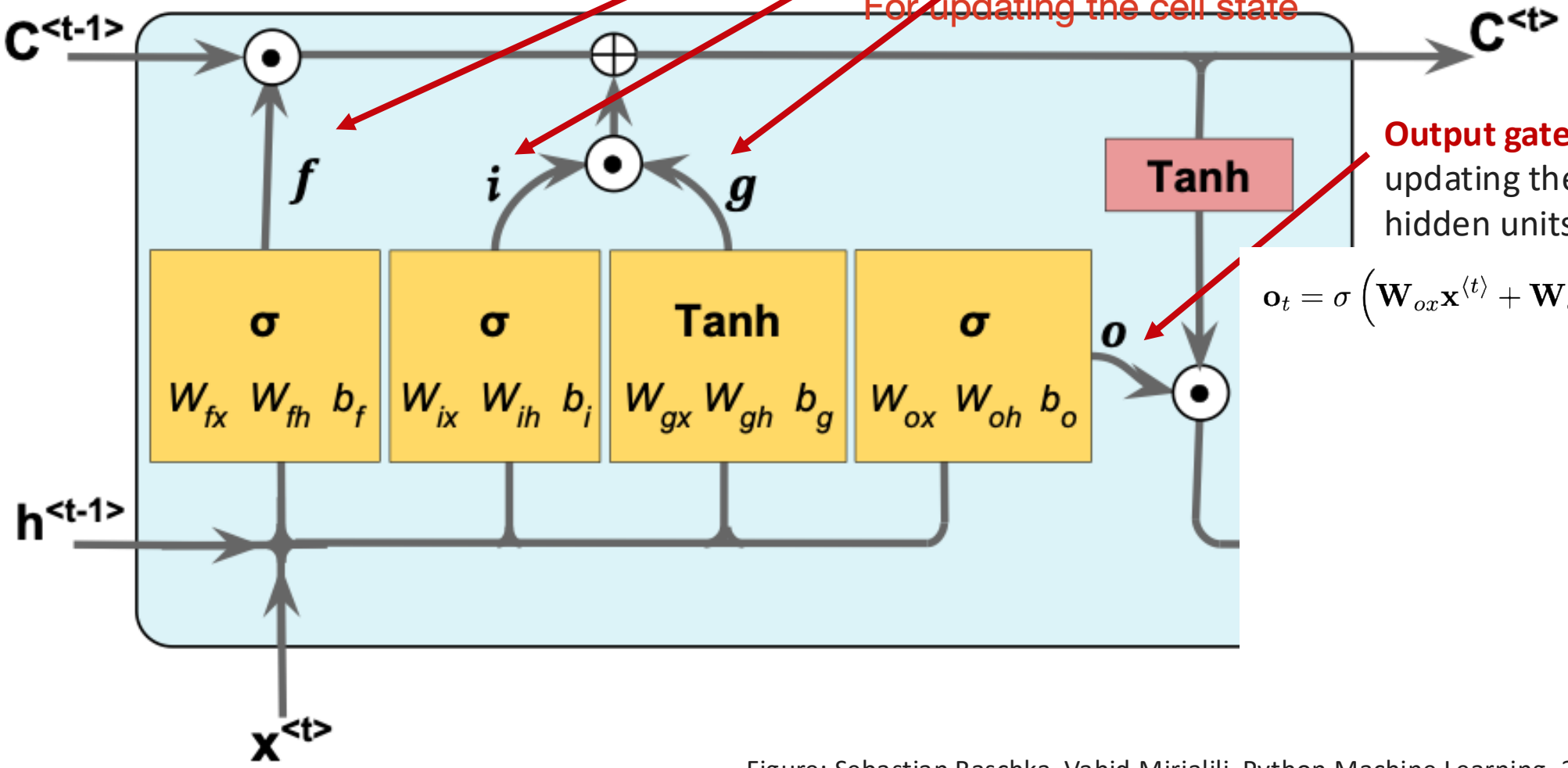
Ben Lengerich © University of Wisconsin-Madison 2026

# Inside LSTM

Forget Gate Input Gate Input Node

$$C^{\langle t \rangle} = \left( C^{\langle t-1 \rangle} \odot f_t \right) \oplus \left( i_t \odot g_t \right)$$

For updating the cell state




**Output gate** for updating the values of hidden units:

$$\mathbf{o}_t = \sigma \left( \mathbf{W}_{ox} \mathbf{x}^{\langle t \rangle} + \mathbf{W}_{oh} \mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_o \right)$$

Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

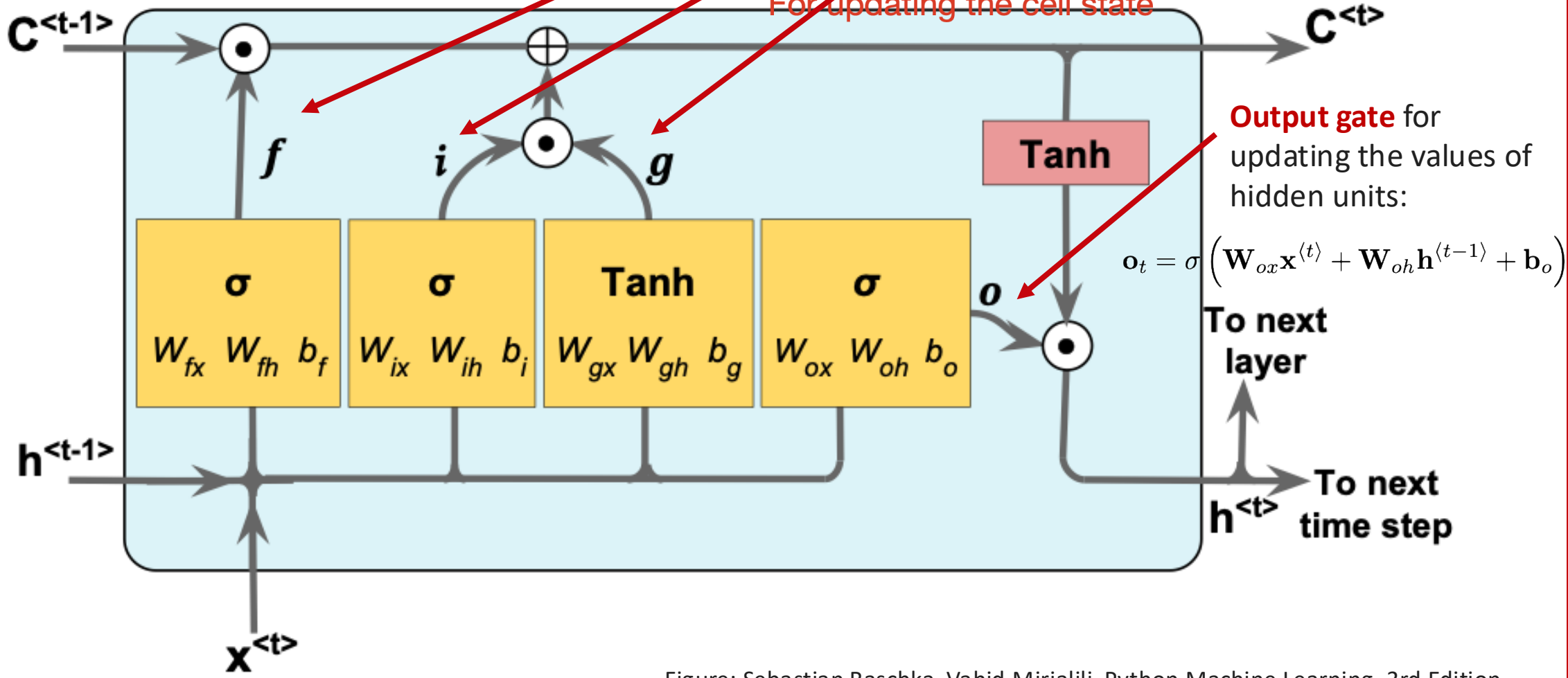
Ben Lengerich © University of Wisconsin-Madison 2026

# Inside LSTM

Forget Gate Input Gate Input Node

$$C^{\langle t \rangle} = \left(C^{\langle t-1 \rangle} \odot f_t\right) \oplus (i_t \odot g_t)$$

For updating the cell state




**Output gate** for updating the values of hidden units:

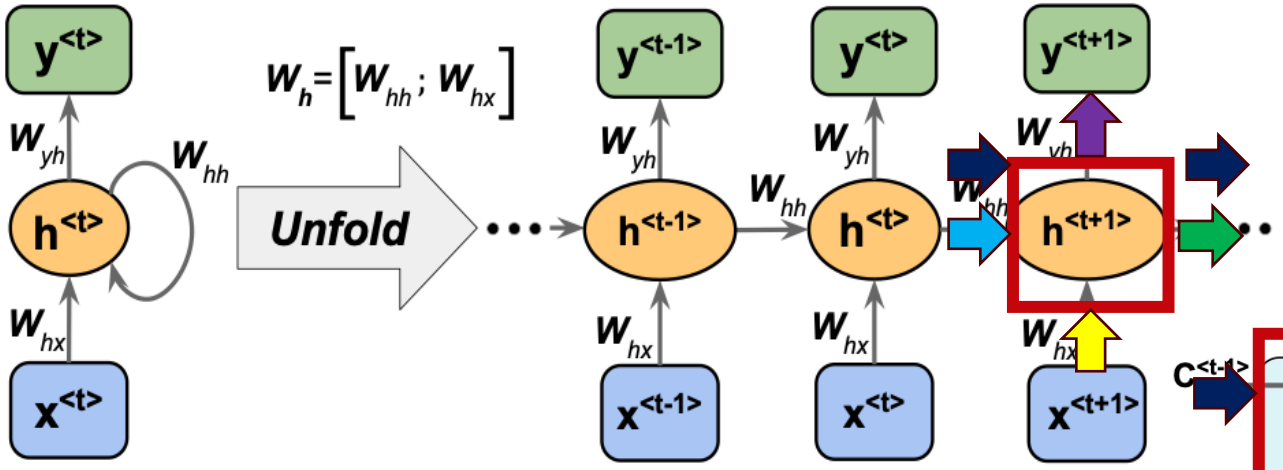
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_{ox}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{oh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_o\right)$$

Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

Ben Lengerich © University of Wisconsin-Madison 2026

# LSTM Back Together

$$\mathbf{h}^{\langle t \rangle} = \mathbf{o}_t \odot \tanh\left(\mathbf{C}^{\langle t \rangle}\right)$$



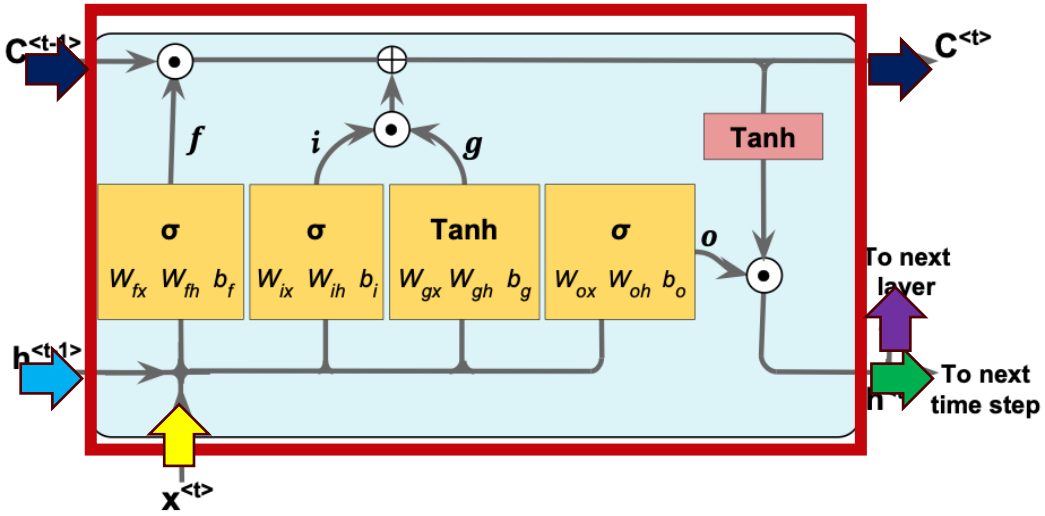




Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

# LSTMs in PyTorch

https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

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 LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **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). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

Examples:

```
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```

# Gated Recurrent Unit (GRU)

$$\mathbf{r}^{\langle t \rangle} = \sigma\left(\mathbf{W}_{rx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{rh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_r\right)$$ reset gate

$$\mathbf{z}^{\langle t \rangle} = \sigma\left(\mathbf{W}_{zx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{zh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_z\right)$$ update gate

$$\tilde{\mathbf{h}}^{\langle t \rangle} = \tanh\left(\mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\left(\mathbf{r}^{\langle t \rangle} \odot \mathbf{h}^{\langle t-1 \rangle}\right) + \mathbf{b}_h\right)$$ candidate state

$$\mathbf{h}^{\langle t \rangle} = \left(1 - \mathbf{z}^{\langle t \rangle}\right) \odot \mathbf{h}^{\langle t-1 \rangle} + \mathbf{z}^{\langle t \rangle} \odot \tilde{\mathbf{h}}^{\langle t \rangle}$$ **new hidden state**

- **A simpler gated RNN (Cho et al. 2014):** 2 gates instead of 3, and no separate cell state
- **Update gate *z***: how much of the old state to overwrite. Like tying the LSTM's forget and input gates together ($f = 1 - z$)
- **Reset gate *r***: lets the candidate ignore the past when it isn't relevant
- 3 weight blocks instead of 4 → about 25% fewer parameters than an LSTM of the same size; accuracy is usually similar (Chung et al. 2014)
- torch.nn.GRU has the same interface as nn.LSTM, minus the cell state. Note: PyTorch's docs swap the roles of $z$ and $1 - z$.

# Why did RNNs fade?

- **Training is sequential:** $h^{<t>}$ can't be computed until $h^{<t-1>}$ is done. A length-$T$ sequence is $T$ dependent steps, so GPUs can't parallelize across time. Transformers process all $T$ positions at once.
- **Memory bottleneck:** the entire past is squeezed into one fixed-size vector $h^{<t>}$. Attention can look back at any earlier token directly.
- **Long-range gradients still fade,** even with LSTM gating (just more slowly).
- **Net effect:** transformers turned more data + compute into better models (BERT, GPT; 2018 onward). By ~2020, RNNs had mostly disappeared from frontier language modeling.

**Transformers won on *training* parallelism. Pay for it at *inference* time.**

Ben Lengerich © University of Wisconsin-Madison 2026

# Recurrence is back

- **The cost of attention:** every new token attends to every previous one, so total compute is $O(T^2)$ and the KV cache grows with $T$. An RNN carries a fixed-size state: O(1) memory per generated token.
- **The trick: remove the nonlinearity between time steps**

  Classic RNN: $h^{<t>} = \tanh(W_{hh}\, h^{<t-1>} + W_{hx}\, x^{<t>})$

  Linear recurrence: $h^{<t>} = A_t\, h^{<t-1>} + B_t\, x^{<t>}$, $y^{<t>} = C_t\, h^{<t>}$

  Linear in $h$ ⇒ train all steps in parallel (parallel scan), then generate one step at a time like an RNN
- **Timeline:** linear attention (Katharopoulos et al. 2020) → S4 state-space models (Gu et al. 2021) → Mamba (Gu & Dao 2023; $A_t$, $B_t$, $C_t$ depend on the input), RWKV, RetNet (2023) → xLSTM (Beck et al. 2024)
- **Today: hybrids.** Mostly recurrent layers plus a few attention layers (Jamba 2024; Nemotron-H, Qwen3-Next 2025).
- **Why keep any attention?** A fixed-size state can't store everything. Exact recall and copying from long contexts are still hard for pure recurrent models (Jelassi et al. 2024).

# Good reading

- The Unreasonable Effectiveness of Recurrent Neural Networks by Andrej Karpathy
- On the difficulty of training recurrent neural networks by Razvan Pascanu, Tomas Mikolov, Yoshua Bengio

# Questions?

