Machine Learning and Optimization

Lecture 17: Sequential Models and the Attention Mechanism

CSCI 4140/6960 — Spring 2026

Course website: `cs.rpi.edu/.../mlandopt.html`

Learning Objectives

By the end of this lecture, you should be able to:

1. **Identify** the components of state-space models (state, input, update rule, output) and explain why CNNs are not naturally suited to variable-length sequences.
2. **Describe** the tokenization pipeline (text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings), the role of the embedding matrix E, and the word/character/subword granularity tradeoff.
3. **State** the Elman RNN forward equations and describe the unfolded computational graph.
4. **Explain** vanishing/exploding gradients via the Jacobian product $\prod_{t'} \frac{\partial h_{t'+1}}{\partial h_{t'}}$ and connect it to $\rho(W_h)$.
5. **Describe** how LSTM gating resolves gradient pathology via the additive cell-state update and the "Jacobian $\approx$ identity" principle.

Learning Objectives (continued)

6. **Identify** the information bottleneck in encoder–decoder architectures and explain how attention resolves it.
7. **State** the core abstraction of attention: a probability distribution over positions composed with a weighted average of values.
8. **Write** the dot-product attention formula and the general scaled dot-product attention formula.
9. **Define** the query–key–value (QKV) abstraction and explain how it separates matching from aggregation.
10. **Describe** the BPE algorithm (frequency-based merging, byte-level initialization) and explain why byte-level BPE eliminates the unknown-token problem.

Lecture Roadmap

Recurrent Neural Networks as Dynamical Systems

Attention as the Resolution to the Information Bottleneck

Tokenization — A Universal Preprocessing Step

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$
- Time-series forecasting, speech recognition, music generation, ...

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$
- Time-series forecasting, speech recognition, music generation, ...

Classical sequential models (ODEs, HMMs, LDS) share common features:

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$
- Time-series forecasting, speech recognition, music generation, ...

Classical sequential models (ODEs, HMMs, LDS) share common features:

- A **state** h_t describing a dynamical system at time t

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$
- Time-series forecasting, speech recognition, music generation, ...

Classical sequential models (ODEs, HMMs, LDS) share common features:

- A **state** h_t describing a dynamical system at time t
- **Inputs** x_t to the system

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$
- Time-series forecasting, speech recognition, music generation, ...

Classical sequential models (ODEs, HMMs, LDS) share common features:

- A **state** h_t describing a dynamical system at time t
- **Inputs** x_t to the system
- An **update rule** $h_t = f(h_{t-1}, x_t)$

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$
- Time-series forecasting, speech recognition, music generation, ...

Classical sequential models (ODEs, HMMs, LDS) share common features:

- A **state** h_t describing a dynamical system at time t
- **Inputs** x_t to the system
- An **update rule** $h_t = f(h_{t-1}, x_t)$
- An optional **output** $\hat{y}_t = g(h_t)$

From Spatial to Sequential Structure

MLPs take *fixed-size* inputs. Similarly, CNNs exploit spatial locality and translation equivariance for *fixed-size*, grid-structured inputs.

Many tasks involve **variable-length sequential** data where order matters:

- Sentiment classification: variable-length text $\rightarrow$ single label
- Machine translation: source $\rightarrow$ target with $T \neq T'$
- Time-series forecasting, speech recognition, music generation, ...

Classical sequential models (ODEs, HMMs, LDS) share common features:

- A **state** h_t describing a dynamical system at time t
- **Inputs** x_t to the system
- An **update rule** $h_t = f(h_{t-1}, x_t)$
- An optional **output** $\hat{y}_t = g(h_t)$

Idea: parameterize f and g with learnable weights $\implies$ **recurrent neural network** (RNN).

Where We're Headed: Large Language Models

For the rest of this course, we focus on **text** as our sequential domain. Our goal: build up, step by step, the full architecture behind modern large language models (LLMs).

Where We're Headed: Large Language Models

For the rest of this course, we focus on **text** as our sequential domain. Our goal: build up, step by step, the full architecture behind modern large language models (LLMs).



Where We're Headed: Large Language Models

For the rest of this course, we focus on **text** as our sequential domain. Our goal: build up, step by step, the full architecture behind modern large language models (LLMs).



The roadmap from here to there:

1. **Today:** RNNs $\rightarrow$ LSTMs $\rightarrow$ attention mechanism $\rightarrow$ tokenization
2. **Next lectures:** Transformer architecture $\rightarrow$ pre-training $\rightarrow$ post-training (RLHF, DPO)

Where We're Headed: Large Language Models

For the rest of this course, we focus on **text** as our sequential domain. Our goal: build up, step by step, the full architecture behind modern large language models (LLMs).



The roadmap from here to there:

1. **Today:** RNNs → LSTMs → attention mechanism → tokenization
2. **Next lectures:** Transformer architecture → pre-training → post-training (RLHF, DPO)

By the end, you will understand the internals well enough to *read the papers*, *work with these models*, and *build your own*.

From Text to Vectors

Before defining the RNN equations, we need to ground x_t. In language tasks, inputs are words or subwords—not numbers.

From Text to Vectors

Before defining the RNN equations, we need to ground x_t. In language tasks, inputs are words or subwords—not numbers.

1. **Tokenize:** Split raw text into **tokens** from a fixed vocabulary $\mathcal{V}$.

From Text to Vectors

Before defining the RNN equations, we need to ground x_t. In language tasks, inputs are words or subwords—not numbers.

1. **Tokenize:** Split raw text into **tokens** from a fixed vocabulary $\mathcal{V}$.
2. **Index:** Map each token to an integer $j \in \{1, \ldots, |\mathcal{V}|\}$.

From Text to Vectors

Before defining the RNN equations, we need to ground x_t. In language tasks, inputs are words or subwords—not numbers.

1. **Tokenize:** Split raw text into **tokens** from a fixed vocabulary $\mathcal{V}$.
2. **Index:** Map each token to an integer $j \in \{1, \ldots, |\mathcal{V}|\}$.
3. **Embed:** Look up via a learnable **embedding matrix** $E \in \mathbb{R}^{d \times |\mathcal{V}|}$: $x_t = E\, e_{j_t} = E_{:,j_t} \in \mathbb{R}^d$

From Text to Vectors

Before defining the RNN equations, we need to ground x_t. In language tasks, inputs are words or subwords—not numbers.

1. **Tokenize:** Split raw text into **tokens** from a fixed vocabulary $\mathcal{V}$.
2. **Index:** Map each token to an integer $j \in \{1, \ldots, |\mathcal{V}|\}$.
3. **Embed:** Look up via a learnable **embedding matrix** $E \in \mathbb{R}^{d \times |\mathcal{V}|}$: $x_t = E\, e_{j_t} = E_{:,j_t} \in \mathbb{R}^d$

Example: "Muy bien y tu"

Stage	**Token 1**	**Token 2**	**Token 3–4**
Raw	"Muy"	"bien"	"y", "tu"
Index	1423	892	37, 516
Embedding	$E_{:,1423}$	$E_{:,892}$	$E_{:,37}$, $E_{:,516}$

From Text to Vectors

Before defining the RNN equations, we need to ground x_t. In language tasks, inputs are words or subwords—not numbers.

1. **Tokenize:** Split raw text into **tokens** from a fixed vocabulary $\mathcal{V}$.
2. **Index:** Map each token to an integer $j \in \{1, \ldots, |\mathcal{V}|\}$.
3. **Embed:** Look up via a learnable **embedding matrix** $E \in \mathbb{R}^{d \times |\mathcal{V}|}$: $x_t = E\, e_{j_t} = E_{:,j_t} \in \mathbb{R}^d$

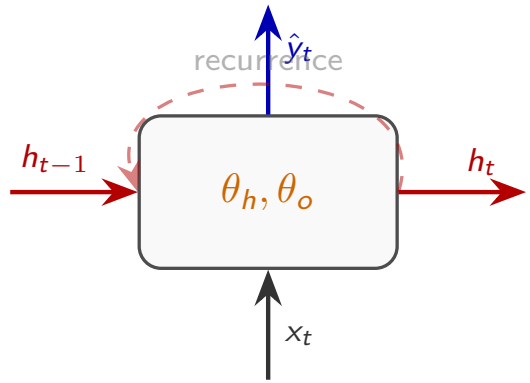
Example: "Muy bien y tu"

Stage	**Token 1**	**Token 2**	**Token 3–4**
Raw	"Muy"	"bien"	"y", "tu"
Index	1423	892	37, 516
Embedding	$E_{:,1423}$	$E_{:,892}$	$E_{:,37}$, $E_{:,516}$

E is trained end-to-end. How $\mathcal{V}$ is constructed (e.g., via BPE) is deferred to the end of the lecture.

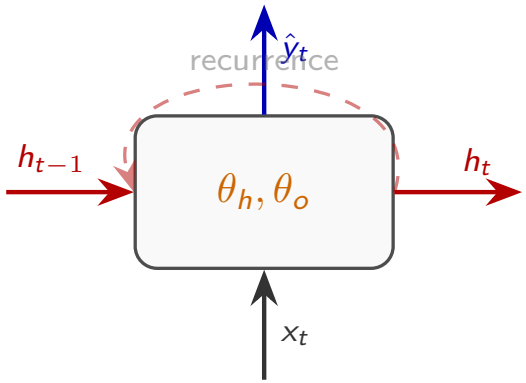
RNN Variables and the Recurrent Cell

- $h_t \in \mathbb{R}^{d_h}$: **hidden state** (internal memory)



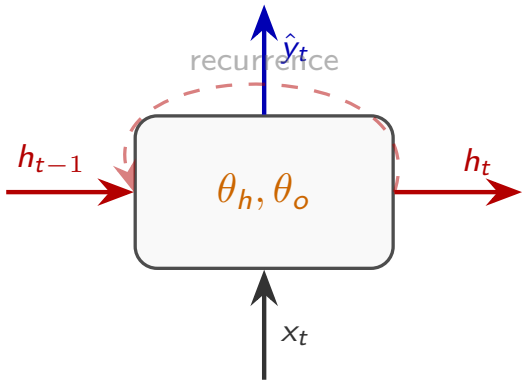
RNN Variables and the Recurrent Cell

- $h_t \in \mathbb{R}^{d_h}$: **hidden state** (internal memory)
- $x_t \in \mathbb{R}^{d_x}$: **input** (embedding vector)



RNN Variables and the Recurrent Cell

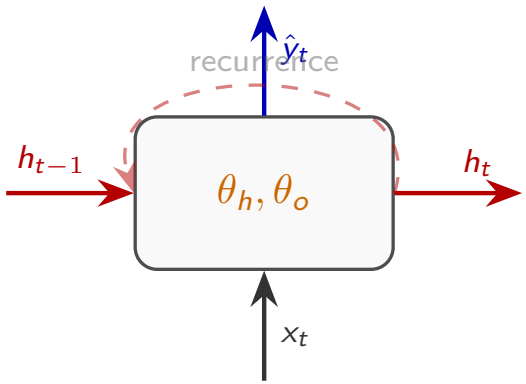
- $h_t \in \mathbb{R}^{d_h}$: **hidden state**
  (internal memory)
- $x_t \in \mathbb{R}^{d_x}$: **input**
  (embedding vector)
- $\hat{y}_t \in \mathbb{R}^{d_y}$: **output**
  (optional per step)



RNN Variables and the Recurrent Cell

- $h_t \in \mathbb{R}^{d_h}$: **hidden state** (internal memory)
- $x_t \in \mathbb{R}^{d_x}$: **input** (embedding vector)
- $\hat{y}_t \in \mathbb{R}^{d_y}$: **output** (optional per step)

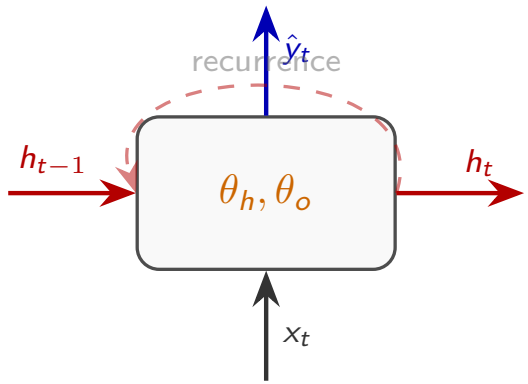
Note: d_h, d_x, and d_y can all be *different*. Typical values: $d_x = 300$ (embedding), $d_h = 512$ (hidden), $d_y = |\mathcal{V}|$ (vocabulary size).



RNN Variables and the Recurrent Cell

- $h_t \in \mathbb{R}^{d_h}$: **hidden state** (internal memory)
- $x_t \in \mathbb{R}^{d_x}$: **input** (embedding vector)
- $\hat{y}_t \in \mathbb{R}^{d_y}$: **output** (optional per step)

Note: d_h, d_x, and d_y can all be *different*. Typical values: $d_x = 300$ (embedding), $d_h = 512$ (hidden), $d_y = |\mathcal{V}|$ (vocabulary size).



Generic RNN equations:

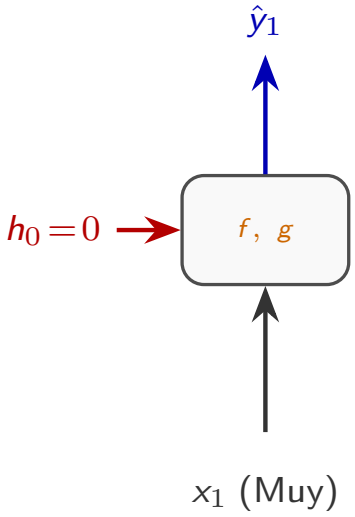
$$h_t = f(x_t, h_{t-1}, \theta_h)$$

$$\hat{y}_t = g(h_t, \theta_o)$$

Weight sharing: same θ_h, θ_o at every step.

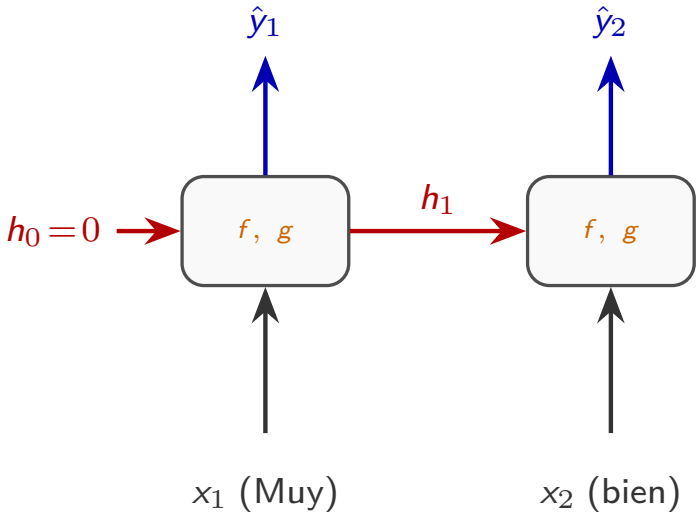
Processing a Sequence

How does the learned dynamical system consume “Muy bien y tu”?



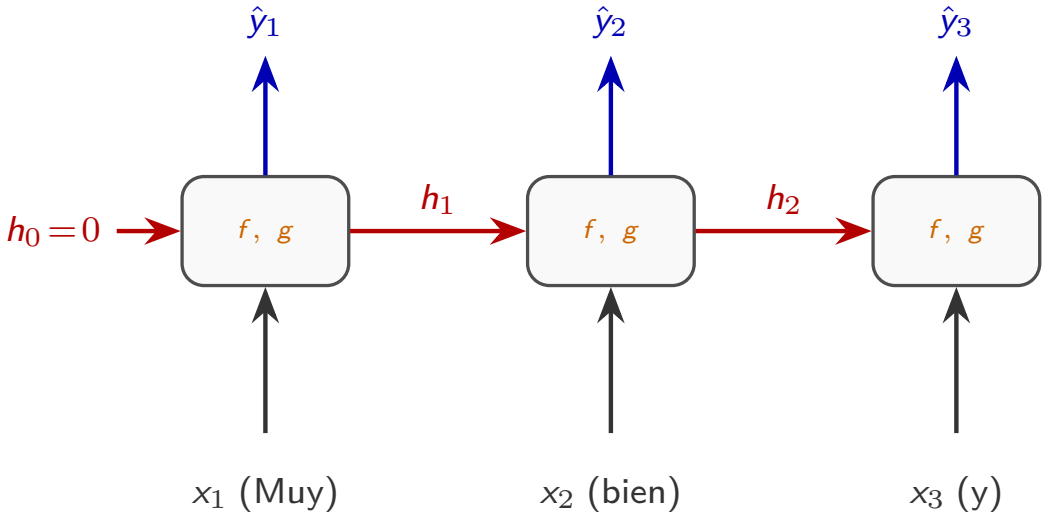
Processing a Sequence

How does the learned dynamical system consume “Muy bien y tu”?



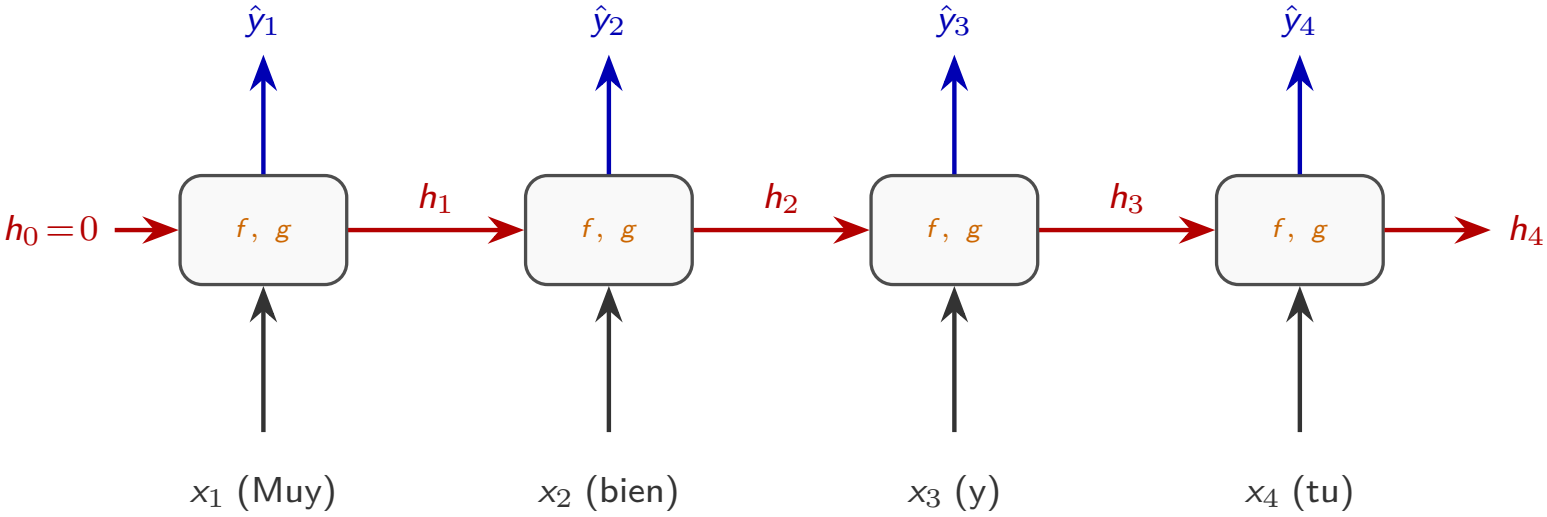
Processing a Sequence

How does the learned dynamical system consume “Muy bien y tu”?



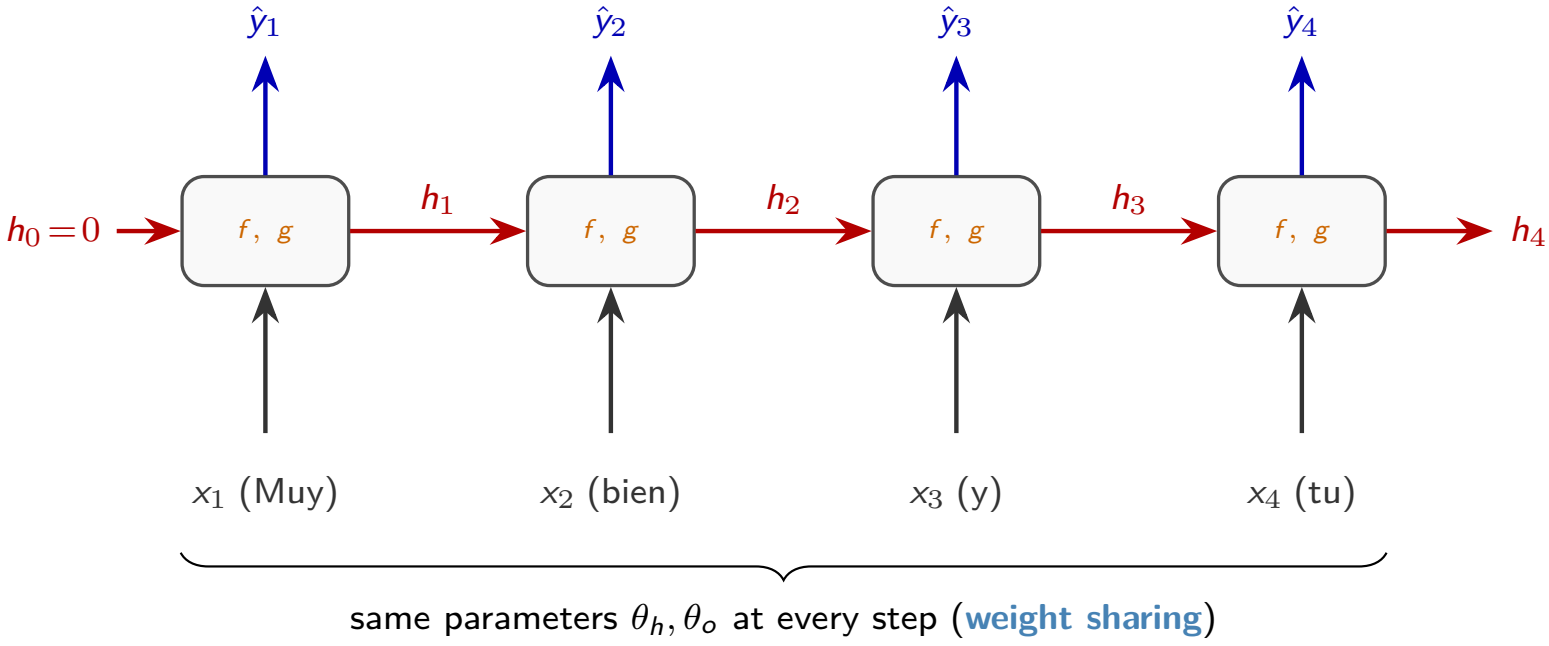
Processing a Sequence

How does the learned dynamical system consume “Muy bien y tu”?



Processing a Sequence

How does the learned dynamical system consume "Muy bien y tu"?



At each step: read x_t, update $h_t = f(x_t, h_{t-1}, \theta_h)$, optionally emit $\hat{y}_t = g(h_t, \theta_o)$.

The hidden state h_t is a lossy, fixed-dimensional summary of all inputs so far: $x_1, \ldots, x_t$.

Vanilla RNN Equations (Elman RNN)

Hidden state update:

$$a_t = W_h\, h_{t-1} + W_x\, x_t + b, \qquad h_t = \tanh(a_t)$$

Vanilla RNN Equations (Elman RNN)

Hidden state update:

$$a_t = W_h\, h_{t-1} + W_x\, x_t + b, \qquad h_t = \tanh(a_t)$$

Output (e.g., classification):

$$\hat{y}_t = \mathrm{softmax}(W_o\, h_t + c)$$

Vanilla RNN Equations (Elman RNN)

Hidden state update:

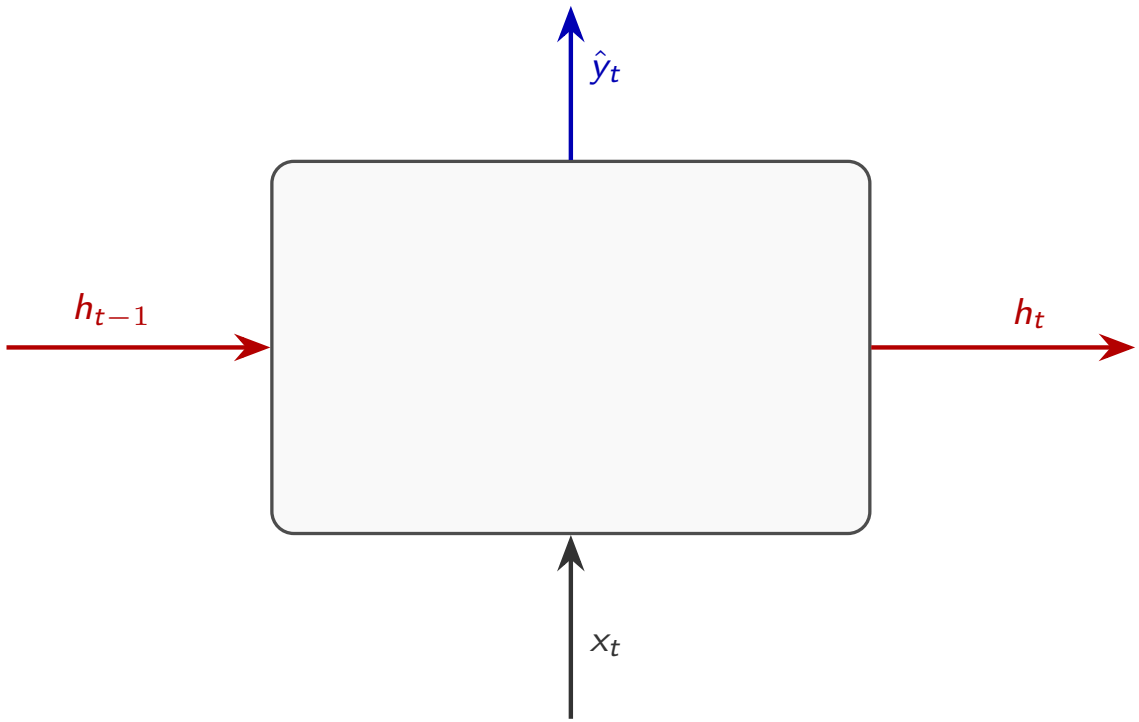
$$a_t = W_h\, h_{t-1} + W_x\, x_t + b, \qquad h_t = \tanh(a_t)$$

Output (e.g., classification):

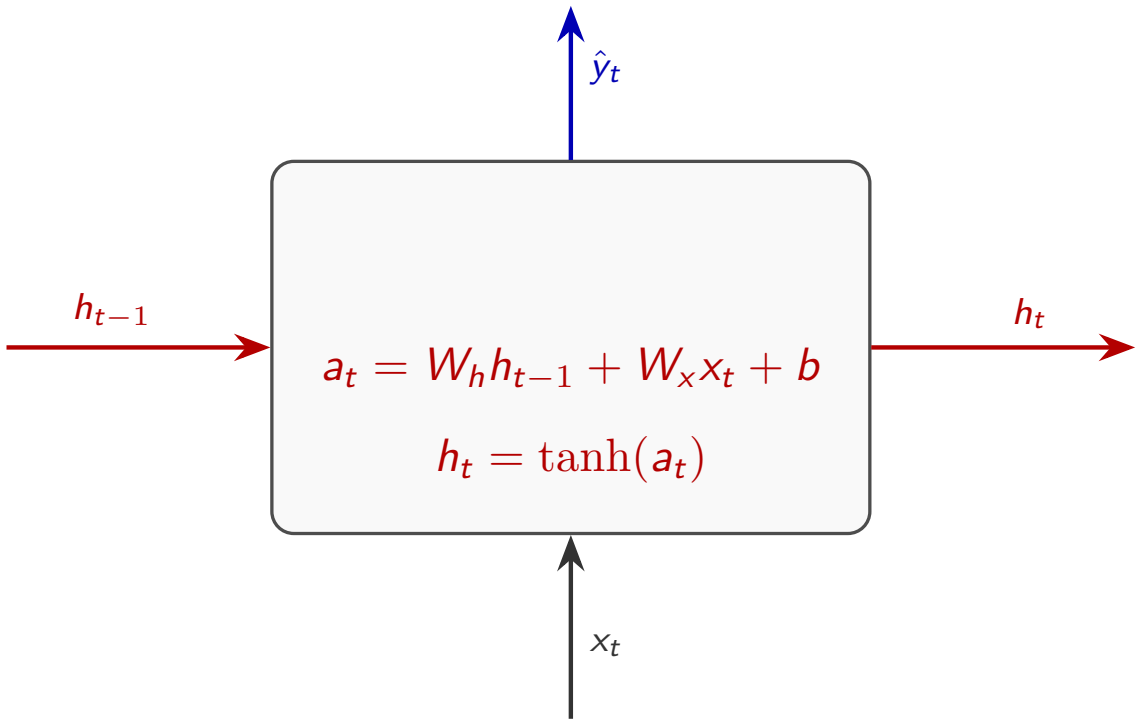
$$\hat{y}_t = \mathrm{softmax}(W_o\, h_t + c)$$

Parameter	Dimensions	Role
W_h	$\mathbb{R}^{d_h \times d_h}$	Hidden-to-hidden (recurrence)
W_x	$\mathbb{R}^{d_h \times d_x}$	Input-to-hidden
b	$\mathbb{R}^{d_h}$	Hidden bias
W_o	$\mathbb{R}^{d_y \times d_h}$	Hidden-to-output
c	$\mathbb{R}^{d_y}$	Output bias

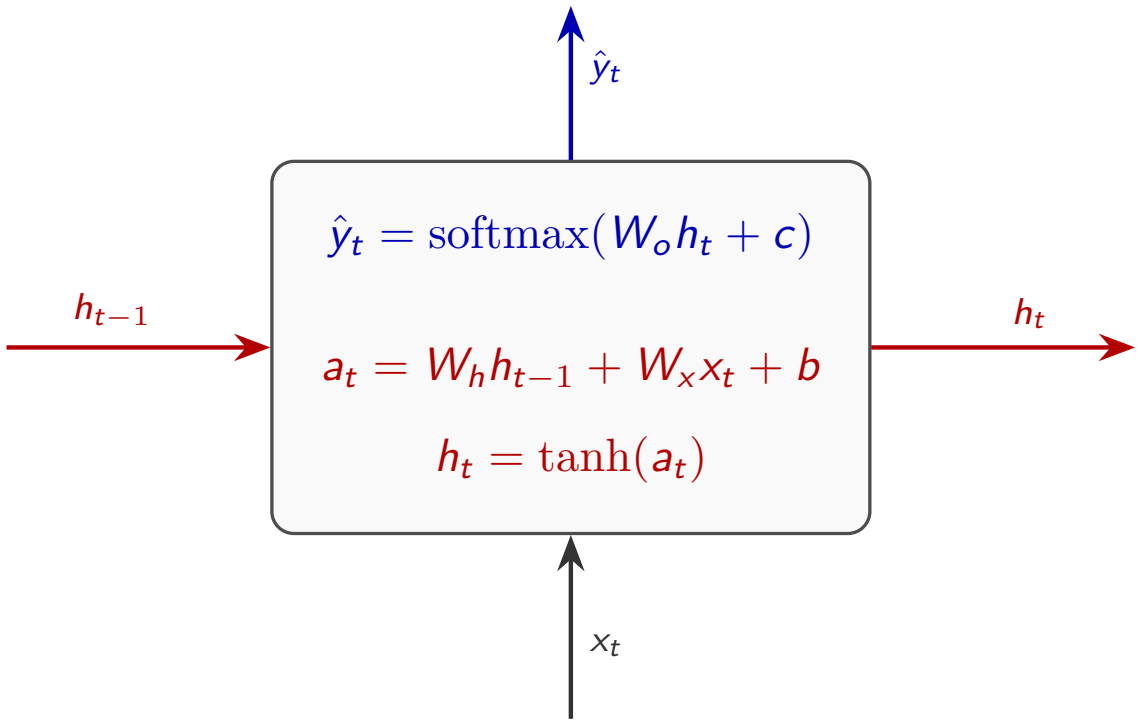
The RNN Cell



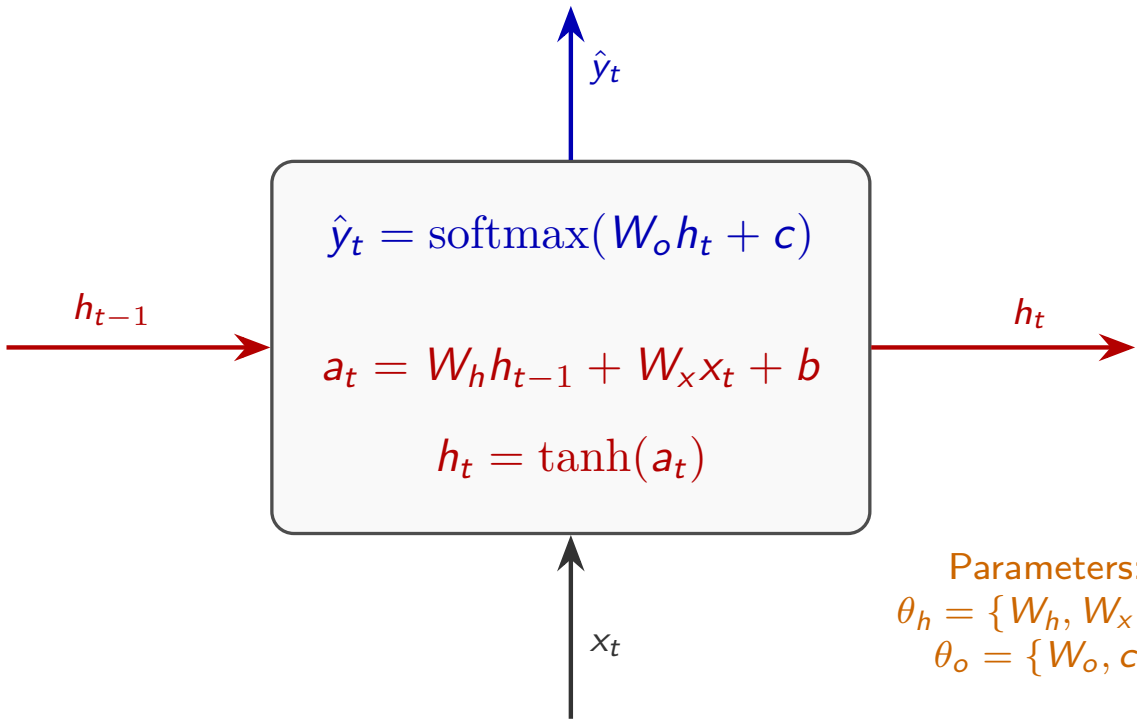
The RNN Cell



The RNN Cell


$\hat{y}_t$
$\hat{y}_t = \text{softmax}(W_o h_t + c)$
h_{t-1}
$a_t = W_h h_{t-1} + W_x x_t + b$
h_t
$h_t = \tanh(a_t)$
x_t

The RNN Cell


$\hat{y}_t$
$\hat{y}_t = \text{softmax}(W_o h_t + c)$
h_{t-1}
$a_t = W_h h_{t-1} + W_x x_t + b$
h_t
$h_t = \tanh(a_t)$
x_t
Parameters:
$\theta_h = \{W_h, W_x, b\}$
$\theta_o = \{W_o, c\}$

Pause and Check

1. Why does the vanilla RNN use $\tanh$ rather than, say, ReLU or sigmoid as its hidden-state nonlinearity?

Pause and Check

1. Why does the vanilla RNN use $\tanh$ rather than, say, ReLU or sigmoid as its hidden-state nonlinearity?

   $\tanh$ is zero-centered (range $(-1, 1)$), so hidden states don't accumulate a systematic positive bias across time steps. Sigmoid (σ) maps to $(0, 1)$, which would make all hidden units positive. ReLU can cause unbounded growth of h_t since it has no saturation for positive inputs—dangerous in a recurrence where the same W_h is applied repeatedly.

Pause and Check

1. Why does the vanilla RNN use $\tanh$ rather than, say, ReLU or sigmoid as its hidden-state nonlinearity?

   $\tanh$ is zero-centered (range $(-1, 1)$), so hidden states don't accumulate a systematic positive bias across time steps. Sigmoid (σ) maps to $(0, 1)$, which would make all hidden units positive. ReLU can cause unbounded growth of h_t since it has no saturation for positive inputs—dangerous in a recurrence where the same W_h is applied repeatedly.

2. If $d_h = 256$ and $d_x = 300$, how many learnable parameters are in the vanilla RNN's hidden-state update (excluding the output layer)?

Pause and Check

1. Why does the vanilla RNN use $\tanh$ rather than, say, ReLU or sigmoid as its hidden-state nonlinearity?

   $\tanh$ is zero-centered (range $(-1, 1)$), so hidden states don't accumulate a systematic positive bias across time steps. Sigmoid (σ) maps to $(0, 1)$, which would make all hidden units positive. ReLU can cause unbounded growth of h_t since it has no saturation for positive inputs—dangerous in a recurrence where the same W_h is applied repeatedly.

2. If $d_h = 256$ and $d_x = 300$, how many learnable parameters are in the vanilla RNN's hidden-state update (excluding the output layer)?

   $W_h \in \mathbb{R}^{256\times256}$, $W_x \in \mathbb{R}^{256\times300}$, $b \in \mathbb{R}^{256}$: total
   $= 256^2 + 256 \cdot 300 + 256 = 65{,}536 + 76{,}800 + 256 = 142{,}592.$

Unfolding in Time

Task: Tag each word in “Muy bien y tu” with its part of speech (ADV, ADV, CONJ, PRON). We need one output per input.

Unfolding in Time

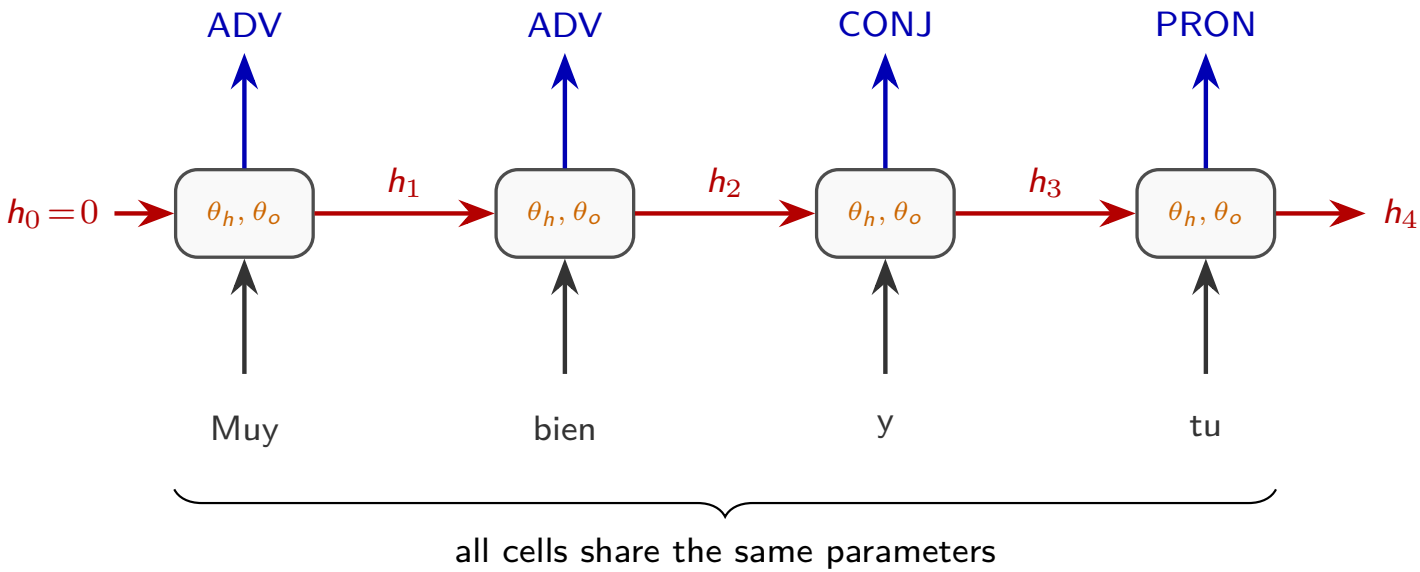
Task: Tag each word in "Muy bien y tu" with its part of speech (ADV, ADV, CONJ, PRON). We need one output per input.

The recurrent cell processes one token at a time. To handle an entire sequence, we **unfold** it into a chain—one copy per time step:

Unfolding in Time

Task: Tag each word in "Muy bien y tu" with its part of speech (ADV, ADV, CONJ, PRON). We need one output per input.

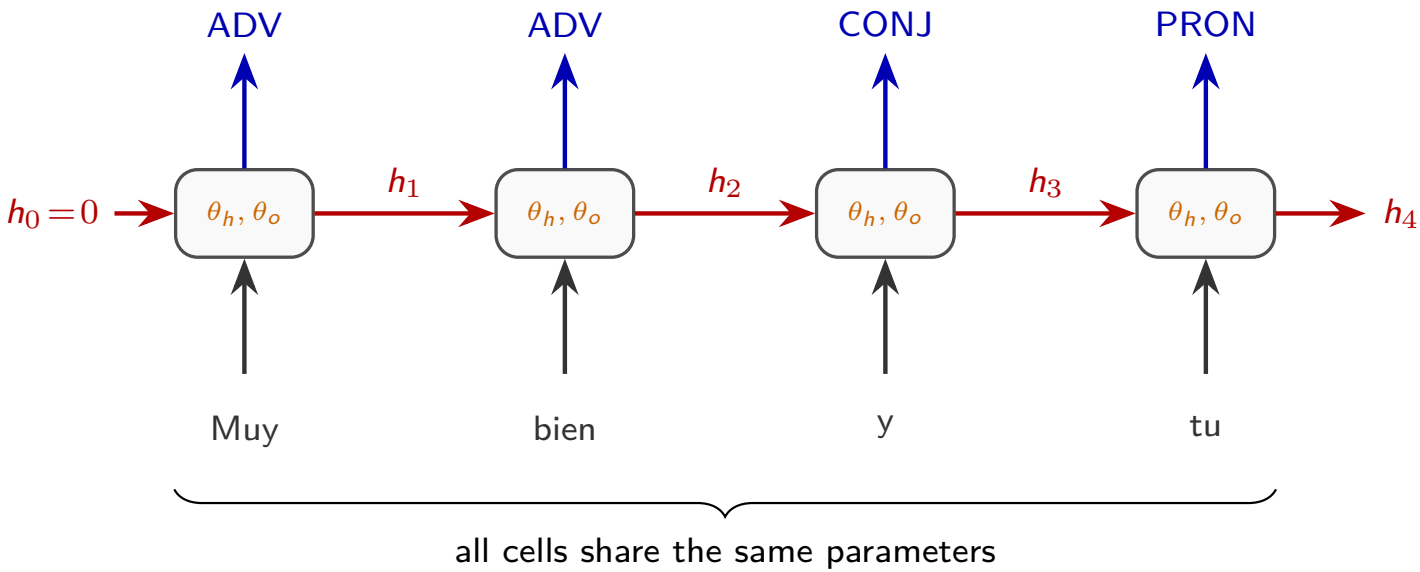
The recurrent cell processes one token at a time. To handle an entire sequence, we **unfold** it into a chain—one copy per time step:



Unfolding in Time

Task: Tag each word in "Muy bien y tu" with its part of speech (ADV, ADV, CONJ, PRON). We need one output per input.

The recurrent cell processes one token at a time. To handle an entire sequence, we **unfold** it into a chain—one copy per time step:



The unfolded graph is a standard feedforward computation graph (a chain). Backpropagation applies directly—this is called **backpropagation through time** (BPTT).

Sequence Loss and BPTT

Given training sequence $(x_1, y_1), \ldots, (x_T, y_T)$, the per-sequence loss is:

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \ell(\hat{y}_t, y_t)$$

Sequence Loss and BPTT

Given training sequence $(x_1, y_1), \ldots, (x_T, y_T)$, the per-sequence loss is:

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \ell(\hat{y}_t, y_t)$$

The critical dependency chain: computing $\hat{y}_t$ requires h_t, which requires h_{t-1}, ..., which requires h_0. Every output depends (indirectly) on the entire input history.

Sequence Loss and BPTT

Given training sequence $(x_1, y_1), \ldots, (x_T, y_T)$, the per-sequence loss is:

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \ell(\hat{y}_t, y_t)$$

The critical dependency chain: computing $\hat{y}_t$ requires h_t, which requires h_{t-1}, ..., which requires h_0. Every output depends (indirectly) on the entire input history.

Gradient computation via BPTT:

$$\nabla_{\theta_h} \mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta_h} \ell(\hat{y}_t, y_t), \qquad \nabla_{\theta_o} \mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta_o} \ell(\hat{y}_t, y_t)$$

Sequence Loss and BPTT

Given training sequence $(x_1, y_1), \ldots, (x_T, y_T)$, the per-sequence loss is:

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \ell(\hat{y}_t, y_t)$$

The critical dependency chain: computing $\hat{y}_t$ requires h_t, which requires h_{t-1}, ..., which requires h_0. Every output depends (indirectly) on the entire input history.

Gradient computation via BPTT:

$$\nabla_{\theta_h} \mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta_h} \ell(\hat{y}_t, y_t), \qquad \nabla_{\theta_o} \mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta_o} \ell(\hat{y}_t, y_t)$$

The gradient w.r.t. θ_o is local (depends only on h_t). The gradient w.r.t. θ_h involves recursive dependence through the hidden states—this is the source of the **vanishing gradient problem**. See Appendix B for the full recurrence.

Gradient Flow Through Time — The Jacobian Product

BPTT propagates gradient information backward through $h_1 \to h_2 \to \cdots \to h_T$.

Gradient Flow Through Time — The Jacobian Product

BPTT propagates gradient information backward through $h_1 \to h_2 \to \cdots \to h_T$.

The sensitivity of h_T to h_t:

$$\frac{\partial h_T}{\partial h_t} = \prod_{t'=t+1}^{T} \frac{\partial h_{t'}}{\partial h_{t'-1}}$$

Gradient Flow Through Time — The Jacobian Product

BPTT propagates gradient information backward through $h_1 \to h_2 \to \cdots \to h_T$.

The sensitivity of h_T to h_t:

$$\frac{\partial h_T}{\partial h_t} = \prod_{t'=t+1}^{T} \frac{\partial h_{t'}}{\partial h_{t'-1}}$$

For $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$, each local Jacobian is:

$$\frac{\partial h_t}{\partial h_{t-1}} = \operatorname{diag}\big(\tanh'(a_t)\big)\, W_h \in \mathbb{R}^{d_h \times d_h}$$

where $\tanh'(a_t) = 1 - \tanh^2(a_t) \in (0, 1]^{d_h}$ elementwise.

Gradient Flow Through Time — The Jacobian Product

BPTT propagates gradient information backward through $h_1 \to h_2 \to \cdots \to h_T$.

The sensitivity of h_T to h_t:

$$\frac{\partial h_T}{\partial h_t} = \prod_{t'=t+1}^{T} \frac{\partial h_{t'}}{\partial h_{t'-1}}$$

For $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$, each local Jacobian is:

$$\frac{\partial h_t}{\partial h_{t-1}} = \operatorname{diag}\big(\tanh'(a_t)\big)\, W_h \in \mathbb{R}^{d_h \times d_h}$$

where $\tanh'(a_t) = 1 - \tanh^2(a_t) \in (0, 1]^{d_h}$ elementwise.

Therefore:

$$\frac{\partial h_T}{\partial h_t} = \prod_{t'=t+1}^{T} \operatorname{diag}\big(\tanh'(a_{t'})\big)\, W_h$$

A product of $T - t$ matrices, each of the form (diagonal scaling) $\times$ (fixed recurrence weight).

Spectral Analysis — Why Gradients Vanish or Explode

Since $|\tanh'(z)| \leq 1$, the diagonal factors can only shrink. In the simplified case $\mathrm{diag}(\tanh'(a_{t'})) \approx I$:

$$\frac{\partial h_T}{\partial h_t} \approx W_h^{T-t}$$

Spectral Analysis — Why Gradients Vanish or Explode

Since $|\tanh'(z)| \leq 1$, the diagonal factors can only shrink. In the simplified case $\mathrm{diag}(\tanh'(a_{t'})) \approx I$:

$$\frac{\partial h_T}{\partial h_t} \approx W_h^{T-t}$$

The behavior is controlled by the **spectral radius** $\rho(W_h) = \max_i |\lambda_i(W_h)|$:

Spectral Analysis — Why Gradients Vanish or Explode

Since $|\tanh'(z)| \leq 1$, the diagonal factors can only shrink. In the simplified case $\mathrm{diag}(\tanh'(a_{t'})) \approx I$:

$$\frac{\partial h_T}{\partial h_t} \approx W_h^{T-t}$$

The behavior is controlled by the **spectral radius** $\rho(W_h) = \max_i |\lambda_i(W_h)|$:

- $\rho(W_h) < 1$: $\|W_h^{T-t}\| \to 0$ exponentially — **vanishing gradients**. Cannot learn long-range dependencies.

Spectral Analysis — Why Gradients Vanish or Explode

Since $|\tanh'(z)| \leq 1$, the diagonal factors can only shrink. In the simplified case $\mathrm{diag}(\tanh'(a_{t'})) \approx I$:

$$\frac{\partial h_T}{\partial h_t} \approx W_h^{T-t}$$

The behavior is controlled by the **spectral radius** $\rho(W_h) = \max_i |\lambda_i(W_h)|$:

- $\rho(W_h) < 1$: $\|W_h^{T-t}\| \to 0$ exponentially — **vanishing gradients**. Cannot learn long-range dependencies.
- $\rho(W_h) > 1$: $\|W_h^{T-t}\| \to \infty$ exponentially — **exploding gradients**. Mitigation: **gradient clipping** $g \leftarrow g \cdot \min(1, \tau/\|g\|)$. A patch, not a cure.

Spectral Analysis — Why Gradients Vanish or Explode

Since $|\tanh'(z)| \leq 1$, the diagonal factors can only shrink. In the simplified case $\mathrm{diag}(\tanh'(a_{t'})) \approx I$:

$$\frac{\partial h_T}{\partial h_t} \approx W_h^{T-t}$$

The behavior is controlled by the **spectral radius** $\rho(W_h) = \max_i |\lambda_i(W_h)|$:

- $\rho(W_h) < 1$: $\|W_h^{T-t}\| \to 0$ exponentially — **vanishing gradients**. Cannot learn long-range dependencies.
- $\rho(W_h) > 1$: $\|W_h^{T-t}\| \to \infty$ exponentially — **exploding gradients**. Mitigation: **gradient clipping** $g \leftarrow g \cdot \min(1, \tau/\|g\|)$. A patch, not a cure.
- $\rho(W_h) \approx 1$: an unstable condition that is difficult to maintain during training.

Spectral Analysis — Why Gradients Vanish or Explode

Since $|\tanh'(z)| \leq 1$, the diagonal factors can only shrink. In the simplified case $\mathrm{diag}(\tanh'(a_{t'})) \approx I$:

$$\frac{\partial h_T}{\partial h_t} \approx W_h^{T-t}$$

The behavior is controlled by the **spectral radius** $\rho(W_h) = \max_i |\lambda_i(W_h)|$:

- $\rho(W_h) < 1$: $\|W_h^{T-t}\| \to 0$ exponentially — **vanishing gradients**. Cannot learn long-range dependencies.
- $\rho(W_h) > 1$: $\|W_h^{T-t}\| \to \infty$ exponentially — **exploding gradients**. Mitigation: **gradient clipping** $g \leftarrow g \cdot \min(1, \tau/\|g\|)$. A patch, not a cure.
- $\rho(W_h) \approx 1$: an unstable condition that is difficult to maintain during training.

For large T, vanilla RNNs fail to propagate useful gradient information to early steps.

The Central Idea — Additive State Updates

The vanilla RNN state update is **multiplicative**: $h_t = \tanh(W_h h_{t-1} + \cdots)$. Gradients must pass through both $\tanh$ and W_h at every step.

The Central Idea — Additive State Updates

The vanilla RNN state update is **multiplicative**: $h_t = \tanh(W_h h_{t-1} + \cdots)$. Gradients must pass through both $\tanh$ and W_h at every step.

Hochreiter & Schmidhuber (1997): the **Long Short-Term Memory** (LSTM) adds a **cell state** $c_t \in \mathbb{R}^{d_h}$ with **additive** update:

$$\boxed{c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t}$$

The Central Idea — Additive State Updates

The vanilla RNN state update is **multiplicative**: $h_t = \tanh(W_h h_{t-1} + \cdots)$. Gradients must pass through both $\tanh$ and W_h at every step.

Hochreiter & Schmidhuber (1997): the **Long Short-Term Memory** (LSTM) adds a **cell state** $c_t \in \mathbb{R}^{d_h}$ with **additive** update:

$$\boxed{c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t}$$

The Central Idea — Additive State Updates

The vanilla RNN state update is **multiplicative**: $h_t = \tanh(W_h h_{t-1} + \cdots)$. Gradients must pass through both $\tanh$ and W_h at every step.

Hochreiter & Schmidhuber (1997): the **Long Short-Term Memory** (LSTM) adds a **cell state** $c_t \in \mathbb{R}^{d_h}$ with **additive** update:

$$\boxed{c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t}$$

- c_{t-1} carried forward directly, scaled by the **forget gate** $f_t \in (0,1)^{d_h}$

The Central Idea — Additive State Updates

The vanilla RNN state update is **multiplicative**: $h_t = \tanh(W_h h_{t-1} + \cdots)$. Gradients must pass through both $\tanh$ and W_h at every step.

Hochreiter & Schmidhuber (1997): the **Long Short-Term Memory** (LSTM) adds a **cell state** $c_t \in \mathbb{R}^{d_h}$ with **additive** update:

$$\boxed{c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t}$$

- c_{t-1} carried forward directly, scaled by the **forget gate** $f_t \in (0,1)^{d_h}$
- New information $\tilde{c}_t$ added, scaled by the **input gate** $i_t \in (0,1)^{d_h}$

The Central Idea — Additive State Updates

The vanilla RNN state update is **multiplicative**: $h_t = \tanh(W_h h_{t-1} + \cdots)$. Gradients must pass through both $\tanh$ and W_h at every step.

Hochreiter & Schmidhuber (1997): the **Long Short-Term Memory** (LSTM) adds a **cell state** $c_t \in \mathbb{R}^{d_h}$ with **additive** update:

$$\boxed{c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t}$$

- c_{t-1} carried forward directly, scaled by the **forget gate** $f_t \in (0,1)^{d_h}$
- New information $\tilde{c}_t$ added, scaled by the **input gate** $i_t \in (0,1)^{d_h}$
- Structurally analogous to a residual connection: $c_t \approx c_{t-1} +$ update when $f_t \approx \mathbf{1}$

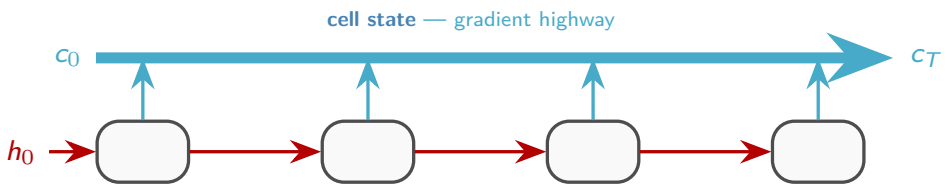
The Central Idea — Additive State Updates

The vanilla RNN state update is **multiplicative**: $h_t = \tanh(W_h h_{t-1} + \cdots)$. Gradients must pass through both $\tanh$ and W_h at every step.

Hochreiter & Schmidhuber (1997): the **Long Short-Term Memory** (LSTM) adds a **cell state** $c_t \in \mathbb{R}^{d_h}$ with **additive** update:

$$\boxed{c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t}$$

- c_{t-1} carried forward directly, scaled by the **forget gate** $f_t \in (0,1)^{d_h}$
- New information $\tilde{c}_t$ added, scaled by the **input gate** $i_t \in (0,1)^{d_h}$
- Structurally analogous to a residual connection: $c_t \approx c_{t-1} +$ update when $f_t \approx \mathbf{1}$



LSTM Gates — Compact Reference

All gates are sigmoid activations of affine combinations of h_{t-1} and x_t:

LSTM Gates — Compact Reference

All gates are sigmoid activations of affine combinations of h_{t-1} and x_t:

$$f_t = \sigma(W_{h,f}\, h_{t-1} + W_{x,f}\, x_t + b_f) \qquad \textbf{(forget gate)}$$

$$i_t = \sigma(W_{h,i}\, h_{t-1} + W_{x,i}\, x_t + b_i) \qquad \textbf{(input gate)}$$

$$\tilde{c}_t = \tanh(W_{h,c}\, h_{t-1} + W_{x,c}\, x_t + b_c) \qquad \textbf{(candidate)}$$

$$o_t = \sigma(W_{h,o}\, h_{t-1} + W_{x,o}\, x_t + b_o) \qquad \textbf{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t)$$

LSTM Gates — Compact Reference

All gates are sigmoid activations of affine combinations of h_{t-1} and x_t:

$$f_t = \sigma(W_{h,f}\, h_{t-1} + W_{x,f}\, x_t + b_f) \qquad \textbf{(forget gate)}$$

$$i_t = \sigma(W_{h,i}\, h_{t-1} + W_{x,i}\, x_t + b_i) \qquad \textbf{(input gate)}$$

$$\tilde{c}_t = \tanh(W_{h,c}\, h_{t-1} + W_{x,c}\, x_t + b_c) \qquad \textbf{(candidate)}$$

$$o_t = \sigma(W_{h,o}\, h_{t-1} + W_{x,o}\, x_t + b_o) \qquad \textbf{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t)$$

LSTM Gates — Compact Reference

All gates are sigmoid activations of affine combinations of h_{t-1} and x_t:

$$f_t = \sigma(W_{h,f}\, h_{t-1} + W_{x,f}\, x_t + b_f) \qquad \textbf{(forget gate)}$$

$$i_t = \sigma(W_{h,i}\, h_{t-1} + W_{x,i}\, x_t + b_i) \qquad \textbf{(input gate)}$$

$$\tilde{c}_t = \tanh(W_{h,c}\, h_{t-1} + W_{x,c}\, x_t + b_c) \qquad \textbf{(candidate)}$$

$$o_t = \sigma(W_{h,o}\, h_{t-1} + W_{x,o}\, x_t + b_o) \qquad \textbf{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t)$$

LSTM Gates — Compact Reference

All gates are sigmoid activations of affine combinations of h_{t-1} and x_t:

$$f_t = \sigma(W_{h,f}\, h_{t-1} + W_{x,f}\, x_t + b_f) \qquad \textbf{(forget gate)}$$

$$i_t = \sigma(W_{h,i}\, h_{t-1} + W_{x,i}\, x_t + b_i) \qquad \textbf{(input gate)}$$

$$\tilde{c}_t = \tanh(W_{h,c}\, h_{t-1} + W_{x,c}\, x_t + b_c) \qquad \textbf{(candidate)}$$

$$o_t = \sigma(W_{h,o}\, h_{t-1} + W_{x,o}\, x_t + b_o) \qquad \textbf{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t)$$

LSTM Gates — Compact Reference

All gates are sigmoid activations of affine combinations of h_{t-1} and x_t:

$$f_t = \sigma(W_{h,f}\, h_{t-1} + W_{x,f}\, x_t + b_f) \qquad \textbf{(forget gate)}$$

$$i_t = \sigma(W_{h,i}\, h_{t-1} + W_{x,i}\, x_t + b_i) \qquad \textbf{(input gate)}$$

$$\tilde{c}_t = \tanh(W_{h,c}\, h_{t-1} + W_{x,c}\, x_t + b_c) \qquad \textbf{(candidate)}$$

$$o_t = \sigma(W_{h,o}\, h_{t-1} + W_{x,o}\, x_t + b_o) \qquad \textbf{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t)$$

LSTM Gates — Compact Reference

All gates are sigmoid activations of affine combinations of h_{t-1} and x_t:

$$f_t = \sigma(W_{h,f}\, h_{t-1} + W_{x,f}\, x_t + b_f) \qquad \textbf{(forget gate)}$$

$$i_t = \sigma(W_{h,i}\, h_{t-1} + W_{x,i}\, x_t + b_i) \qquad \textbf{(input gate)}$$

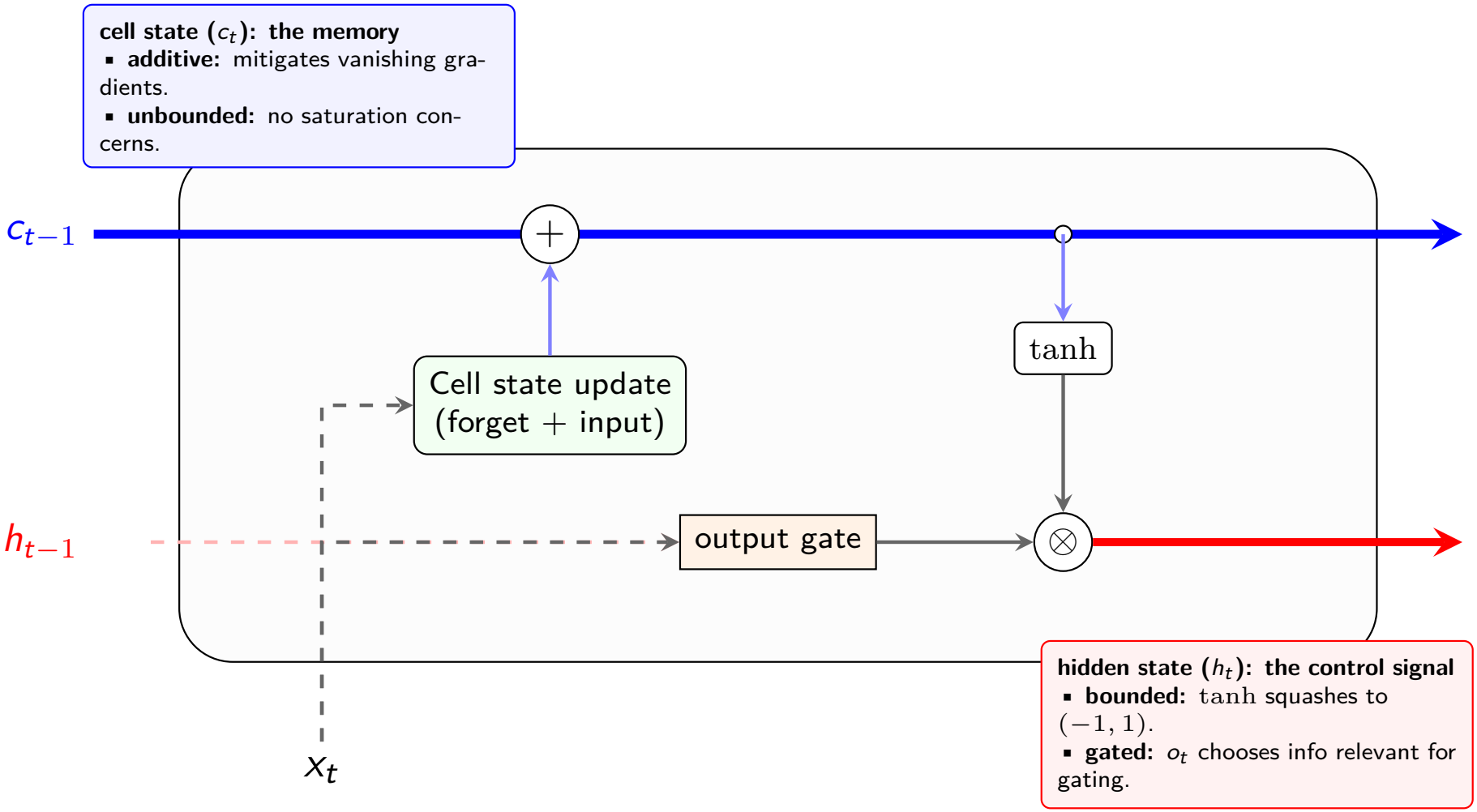
$$\tilde{c}_t = \tanh(W_{h,c}\, h_{t-1} + W_{x,c}\, x_t + b_c) \qquad \textbf{(candidate)}$$

$$o_t = \sigma(W_{h,o}\, h_{t-1} + W_{x,o}\, x_t + b_o) \qquad \textbf{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t)$$

Every gate has the same form $\sigma(\text{affine}(h_{t-1}, x_t))$, differing only in learned parameters. The gates are *control signals*; the cell-state update is the *mechanism*. See Appendix A for full dimensional reference.

c_t (memory) vs. h_t (control)



c_t is optimized for *gradient propagation* (the "memory").

h_t is optimized for *control*.

Pause and Check

1. The forget gate bias b_f is often initialized to a *large positive value* (e.g., 1 or 2). Why?

Pause and Check

1. The forget gate bias b_f is often initialized to a *large positive value* (e.g., 1 or 2). Why?

   At initialization, $\sigma(b_f) \approx 1$, so $f_t \approx \mathbf{1}$ and the cell state is carried forward nearly unchanged. This ensures gradient flow from the start of training. Without this, randomly initialized forget gates would be ≈ 0.5, causing the cell state to decay geometrically and losing the gradient highway property before training even begins.

Pause and Check

1. The forget gate bias b_f is often initialized to a *large positive value* (e.g., 1 or 2). Why?

   At initialization, $\sigma(b_f) \approx 1$, so $f_t \approx \mathbf{1}$ and the cell state is carried forward nearly unchanged. This ensures gradient flow from the start of training. Without this, randomly initialized forget gates would be ≈ 0.5, causing the cell state to decay geometrically and losing the gradient highway property before training even begins.

2. The LSTM has four sets of weight matrices ($W_{h,f}, W_{h,i}, W_{h,c}, W_{h,o}$ and their W_x counterparts). How does its parameter count compare to the vanilla RNN?

Pause and Check

1. The forget gate bias b_f is often initialized to a *large positive value* (e.g., 1 or 2). Why?

   At initialization, $\sigma(b_f) \approx 1$, so $f_t \approx \mathbf{1}$ and the cell state is carried forward nearly unchanged. This ensures gradient flow from the start of training. Without this, randomly initialized forget gates would be ≈ 0.5, causing the cell state to decay geometrically and losing the gradient highway property before training even begins.

2. The LSTM has four sets of weight matrices ($W_{h,f}, W_{h,i}, W_{h,c}, W_{h,o}$ and their W_x counterparts). How does its parameter count compare to the vanilla RNN?

   Roughly $4\times$. The vanilla RNN has one hidden-to-hidden matrix (d_h^2) and one input-to-hidden matrix ($d_h \cdot d_x$). The LSTM has four of each, plus four biases: $4(d_h^2 + d_h \cdot d_x + d_h)$.

Why Gating Helps

The Jacobian of the cell-state update (ignoring small second-order terms):

$$\frac{\partial c_t}{\partial c_{t-1}} = \mathrm{diag}(f_t)$$

Why Gating Helps

The Jacobian of the cell-state update (ignoring small second-order terms):

$$\frac{\partial c_t}{\partial c_{t-1}} = \mathrm{diag}(f_t)$$

When the forget gate is open ($f_t \approx \mathbf{1}$):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx I \qquad \Longrightarrow \qquad \frac{\partial c_T}{\partial c_t} = \prod_{t'=t+1}^{T} \mathrm{diag}(f_{t'}) \approx I$$

Why Gating Helps

The Jacobian of the cell-state update (ignoring small second-order terms):

$$\frac{\partial c_t}{\partial c_{t-1}} = \mathrm{diag}(f_t)$$

When the forget gate is open ($f_t \approx \mathbf{1}$):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx I \qquad \Longrightarrow \qquad \frac{\partial c_T}{\partial c_t} = \prod_{t'=t+1}^{T} \mathrm{diag}(f_{t'}) \approx I$$

Gradients flow through the cell state **without vanishing or exploding**. The cell state acts as a **gradient highway**: a direct, additive path through time bypassing the multiplicative bottleneck.

Why Gating Helps

The Jacobian of the cell-state update (ignoring small second-order terms):

$$\frac{\partial c_t}{\partial c_{t-1}} = \mathrm{diag}(f_t)$$

When the forget gate is open ($f_t \approx \mathbf{1}$):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx I \qquad \Longrightarrow \qquad \frac{\partial c_T}{\partial c_t} = \prod_{t'=t+1}^{T} \mathrm{diag}(f_{t'}) \approx I$$

Gradients flow through the cell state **without vanishing or exploding**. The cell state acts as a **gradient highway**: a direct, additive path through time bypassing the multiplicative bottleneck.

Caveat: In practice f_t is not identically $\mathbf{1}$—the forget gate is trained and varies across time steps and dimensions. The argument shows why the *architecture allows gradient flow*, not that gradients are perfectly preserved.

Why Gating Helps

The Jacobian of the cell-state update (ignoring small second-order terms):

$$\frac{\partial c_t}{\partial c_{t-1}} = \mathrm{diag}(f_t)$$

When the forget gate is open ($f_t \approx \mathbf{1}$):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx I \qquad \Longrightarrow \qquad \frac{\partial c_T}{\partial c_t} = \prod_{t'=t+1}^{T} \mathrm{diag}(f_{t'}) \approx I$$

Gradients flow through the cell state **without vanishing or exploding**. The cell state acts as a **gradient highway**: a direct, additive path through time bypassing the multiplicative bottleneck.

Caveat: In practice f_t is not identically $\mathbf{1}$—the forget gate is trained and varies across time steps and dimensions. The argument shows why the *architecture allows gradient flow*, not that gradients are perfectly preserved.

Analogy: residual connections add skip paths through depth; LSTM gating adds skip paths through time. Both turn multiplicative chains into approximately additive ones.

The Sequence-to-Sequence Problem

Many tasks require mapping a variable-length input $(x_1, \ldots, x_T)$ to a variable-length output $(y_1, \ldots, y_{T'})$ where $T \neq T'$ in general.

The Sequence-to-Sequence Problem

Many tasks require mapping a variable-length input $(x_1, \ldots, x_T)$ to a variable-length output $(y_1, \ldots, y_{T'})$ where $T \neq T'$ in general.

The standard many-to-many RNN produces one output per input—it requires $T = T'$ with synchronized alignment. This does not apply to translation, summarization, or question answering.

The Sequence-to-Sequence Problem

Many tasks require mapping a variable-length input $(x_1, \ldots, x_T)$ to a variable-length output $(y_1, \ldots, y_{T'})$ where $T \neq T'$ in general.

The standard many-to-many RNN produces one output per input—it requires $T = T'$ with synchronized alignment. This does not apply to translation, summarization, or question answering.

Example:

- Input: "Muy bien y tu" ($T = 4$)
- Output: "Very well and you" ($T' = 4$ here, but in general $T \neq T'$)

Introduced in Sutskever et al., "Sequence to Sequence Learning with Neural Networks" (2014).

The Encoder–Decoder Architecture

Introduced in Cho et al. (2014) and Sutskever et al. (2014): split the model into two components.

The Encoder–Decoder Architecture

Introduced in Cho et al. (2014) and Sutskever et al. (2014): split the model into two components.

- **Encoder:** a many-to-one RNN reading $x_1, \ldots, x_T$, producing hidden states $h_1, \ldots, h_T$. The final state $h_T \in \mathbb{R}^{d_h}$ summarizes the entire input.

The Encoder–Decoder Architecture

Introduced in Cho et al. (2014) and Sutskever et al. (2014): split the model into two components.

- **Encoder:** a many-to-one RNN reading $x_1, \ldots, x_T$, producing hidden states $h_1, \ldots, h_T$. The final state $h_T \in \mathbb{R}^{d_h}$ summarizes the entire input.
- **Decoder:** a one-to-many RNN initialized with $s_0 = h_T$ that generates outputs autoregressively:
  - At each step: update $s_t = f_{\text{dec}}(s_{t-1}, \hat{y}_{t-1}, \theta_{\text{dec}})$, predict via linear + softmax.
  - Continue until $\langle \texttt{EOS} \rangle$ or max length.

The Encoder–Decoder Architecture

Introduced in Cho et al. (2014) and Sutskever et al. (2014): split the model into two components.

- **Encoder:** a many-to-one RNN reading $x_1, \ldots, x_T$, producing hidden states $h_1, \ldots, h_T$. The final state $h_T \in \mathbb{R}^{d_h}$ summarizes the entire input.
- **Decoder:** a one-to-many RNN initialized with $s_0 = h_T$ that generates outputs autoregressively:
  - At each step: update $s_t = f_{\text{dec}}(s_{t-1}, \hat{y}_{t-1}, \theta_{\text{dec}})$, predict via linear + softmax.
  - Continue until $\langle \texttt{EOS} \rangle$ or max length.
- During training, the decoder receives the ground-truth y_{t-1} (**teacher forcing**) rather than its own $\hat{y}_{t-1}$ (see Appendix C).

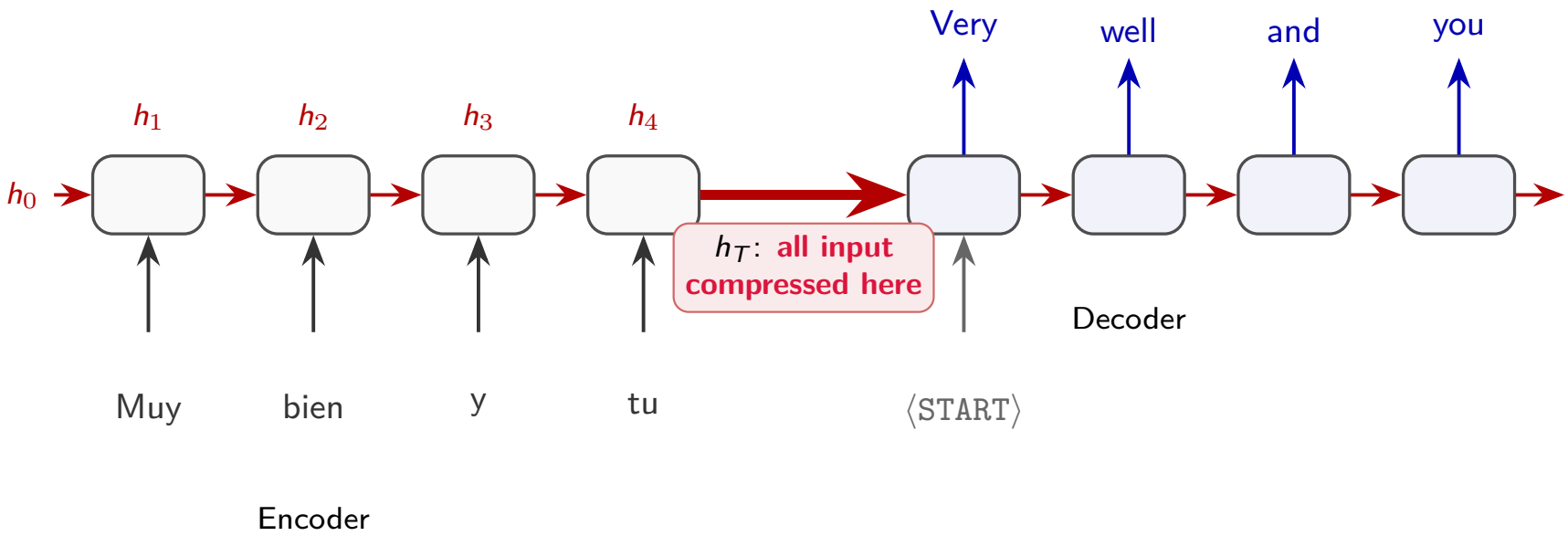
Encoder–Decoder (without Attention)

Input: Source tokens $x_1, \ldots, x_T$; target tokens $y_1, \ldots, y_{T'}$ (training)
Output: Predictions $\hat{y}_1, \ldots, \hat{y}_{T'}$

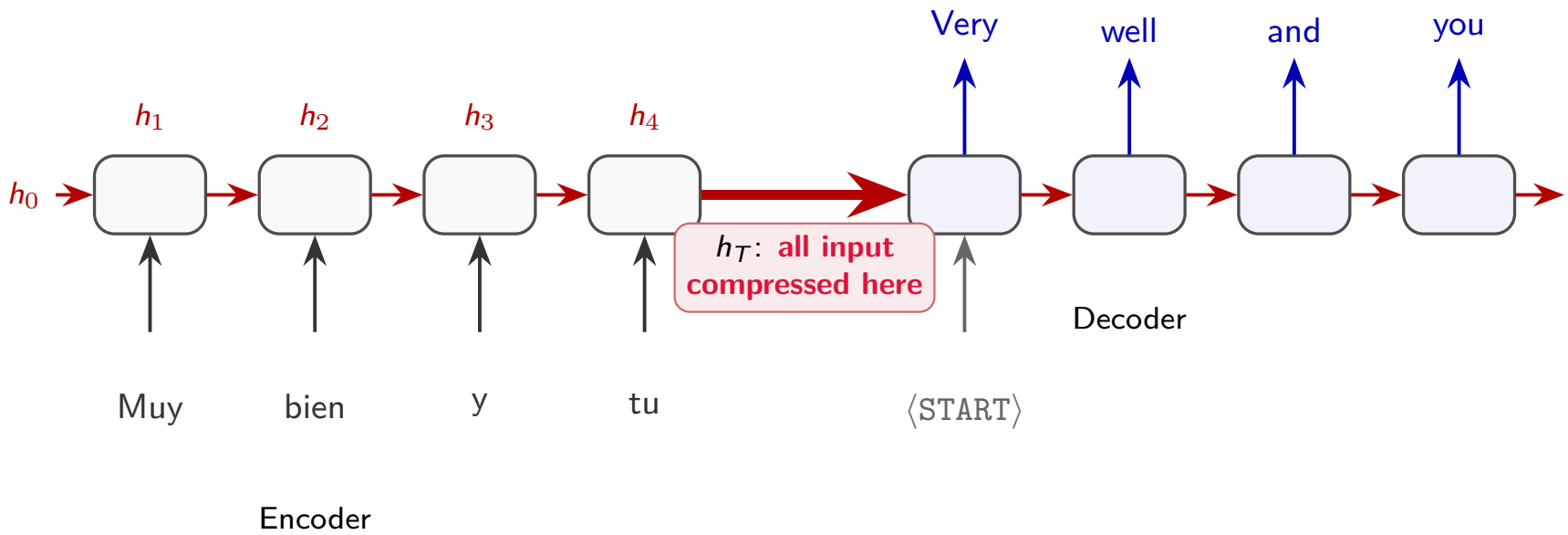
`/* Encode */`
$h_0 \leftarrow \mathbf{0}$
for $t = 1, \ldots, T$ **do**
  $h_t \leftarrow \tanh(W_h\, h_{t-1} + W_x\, x_t + b)$
end

`/* Decode */`
$s_0 \leftarrow h_T$ `// bottleneck: all info in` h_T
for $t = 1, \ldots, T'$ **do**
  $s_t \leftarrow f_{\text{dec}}(s_{t-1},\ y_{t-1},\ \theta_{\text{dec}})$ `// teacher forcing`
  $\hat{y}_t \leftarrow \text{softmax}(W_o\, s_t + c)$
end
return $\hat{y}_1, \ldots, \hat{y}_{T'}$

The Information Bottleneck

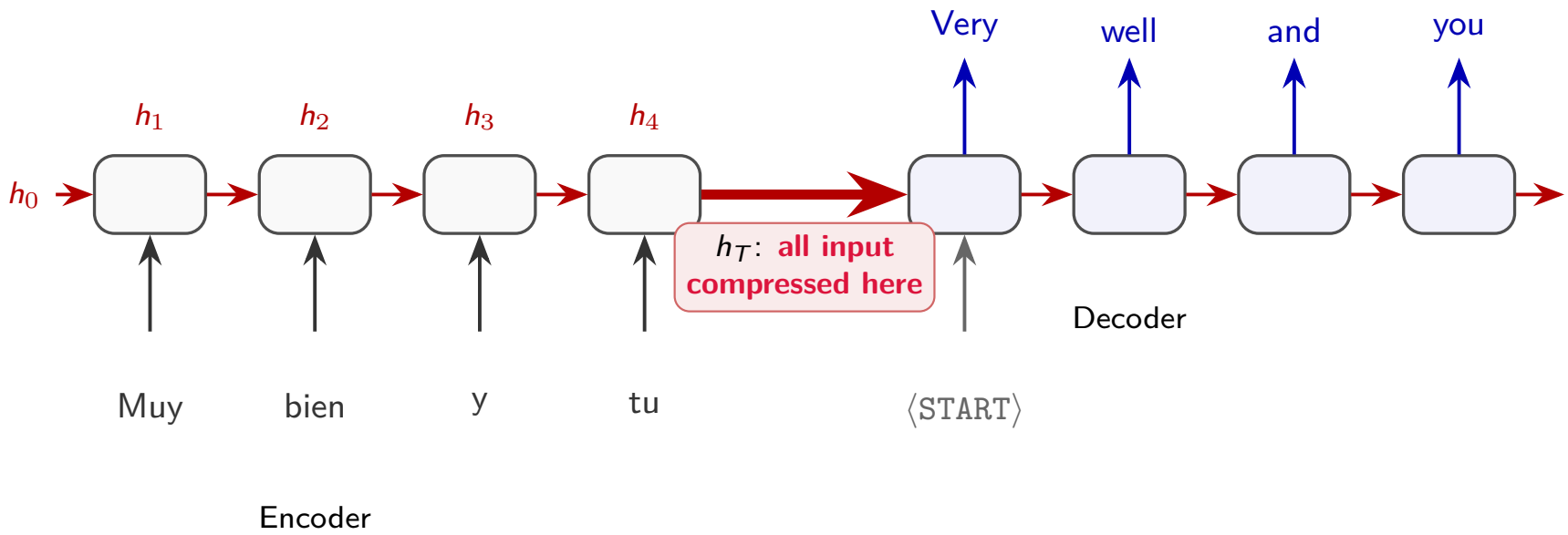


The Information Bottleneck



Problem: Regardless of T, all input information is compressed into a single d_h-dimensional vector h_T. As T grows, h_T becomes increasingly lossy. This is the **information bottleneck**.

The Information Bottleneck



Problem: Regardless of T, all input information is compressed into a single d_h-dimensional vector h_T. As T grows, h_T becomes increasingly lossy. This is the **information bottleneck**.

Empirically, encoder–decoder performance degrades sharply for long sequences.

The Attention Mechanism (Bahdanau et al., 2015)

Popularized in Bahdanau, Cho & Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" (2015).

The Attention Mechanism (Bahdanau et al., 2015)

Popularized in Bahdanau, Cho & Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" (2015).

Central idea: Instead of compressing the entire input into h_T, let the decoder **attend** to each encoder state $h_1, \ldots, h_T$ at every output step.

The Attention Mechanism (Bahdanau et al., 2015)

Popularized in Bahdanau, Cho & Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" (2015).

Central idea: Instead of compressing the entire input into h_T, let the decoder **attend** to each encoder state $h_1, \ldots, h_T$ at every output step.

The Attention Mechanism (Bahdanau et al., 2015)

Popularized in Bahdanau, Cho & Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" (2015).

Central idea: Instead of compressing the entire input into h_T, let the decoder **attend** to each encoder state $h_1, \ldots, h_T$ at every output step.

- When generating "well," attend primarily to "bien."

The Attention Mechanism (Bahdanau et al., 2015)

Popularized in Bahdanau, Cho & Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" (2015).

Central idea: Instead of compressing the entire input into h_T, let the decoder **attend** to each encoder state $h_1, \ldots, h_T$ at every output step.

- When generating "well," attend primarily to "bien."
- When generating "you," attend primarily to "tu."

The Attention Mechanism (Bahdanau et al., 2015)

Popularized in Bahdanau, Cho & Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" (2015).

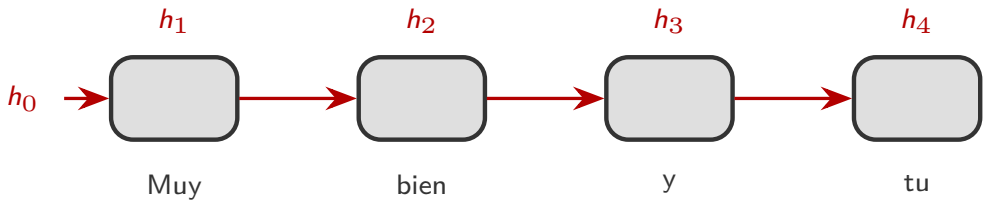
Central idea: Instead of compressing the entire input into h_T, let the decoder **attend** to each encoder state $h_1, \ldots, h_T$ at every output step.

- When generating "well," attend primarily to "bien."
- When generating "you," attend primarily to "tu."

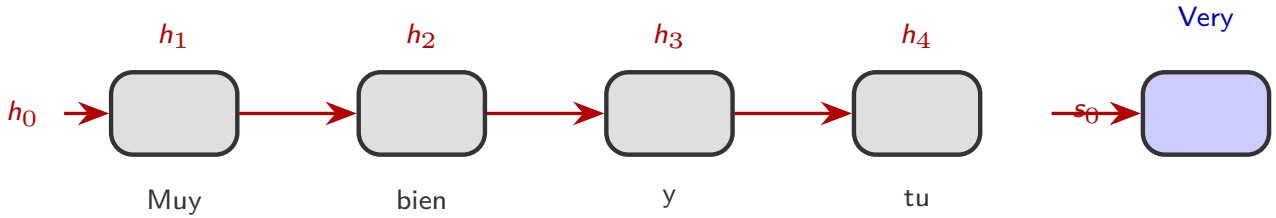
The decoder has direct access to every encoder state—no need to compress everything into h_T.

The attention weights $\alpha_t \in \mathbb{R}^T$ form a distribution over encoder positions, visualizable as a $T' \times T$ **alignment matrix** (heatmap).

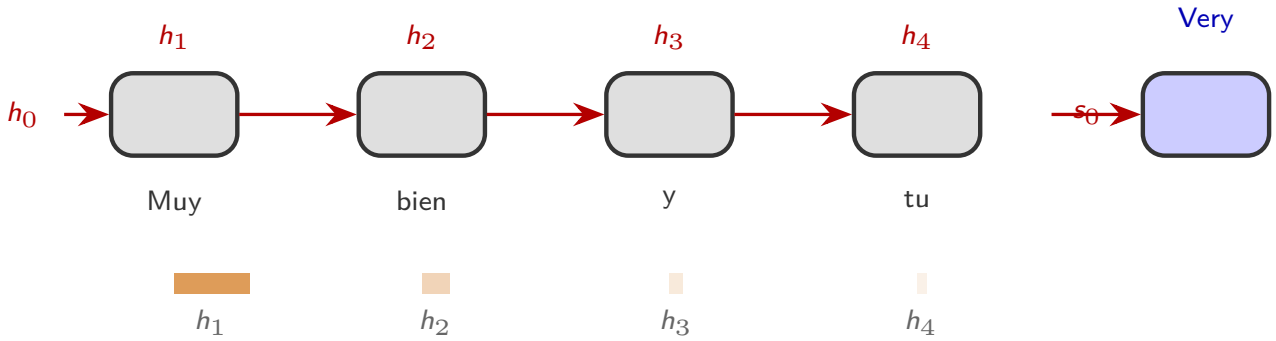
Attention Step by Step — Generating “Very” and “well”



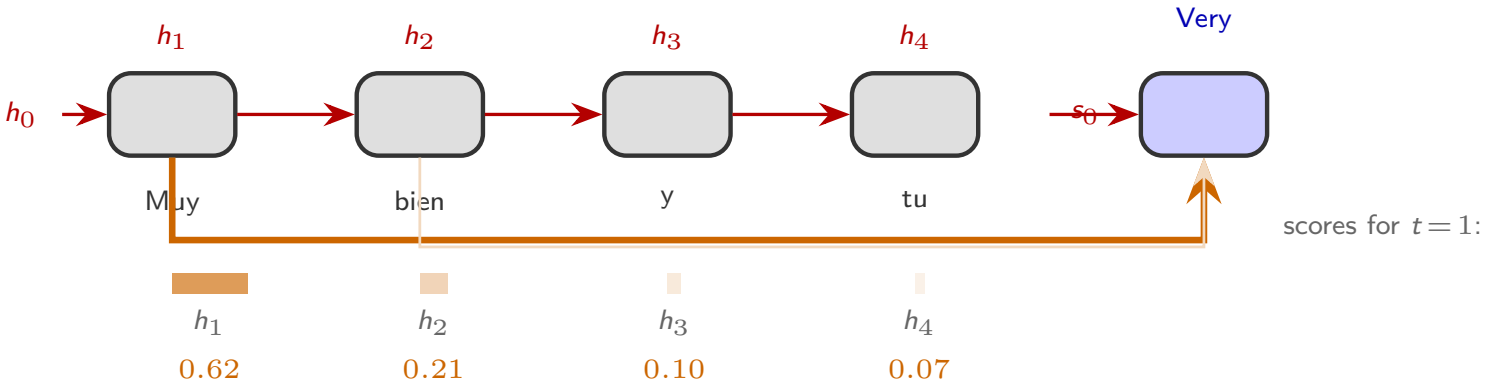
Attention Step by Step — Generating “Very” and “well”



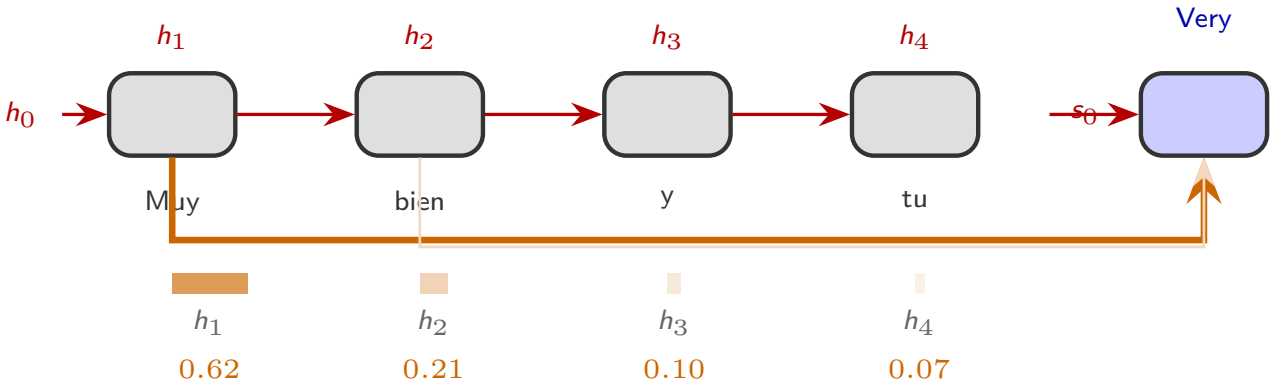
Attention Step by Step — Generating “Very” and “well”



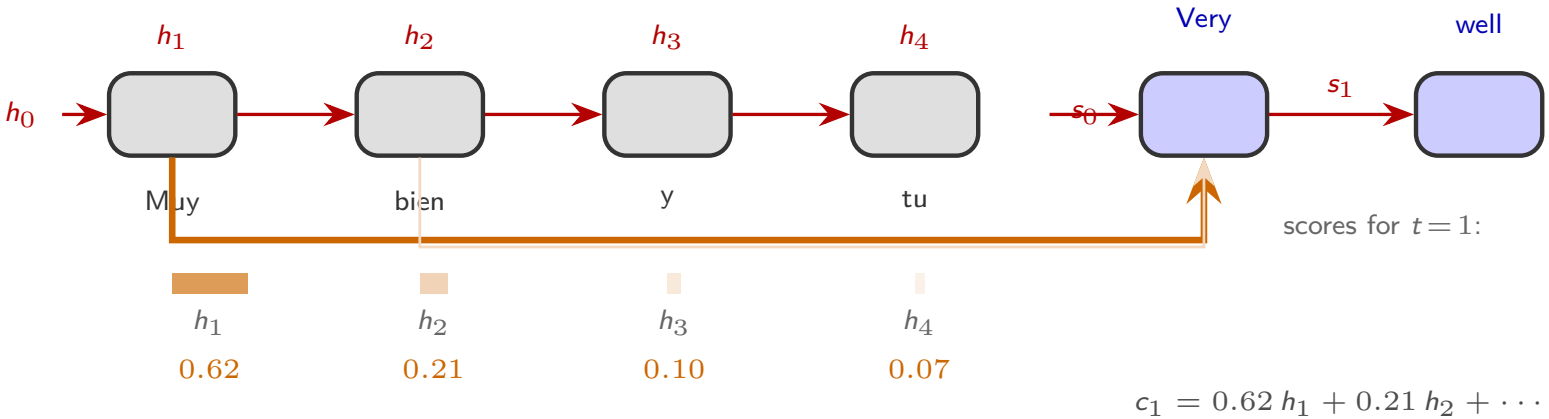
Attention Step by Step — Generating “Very” and “well”



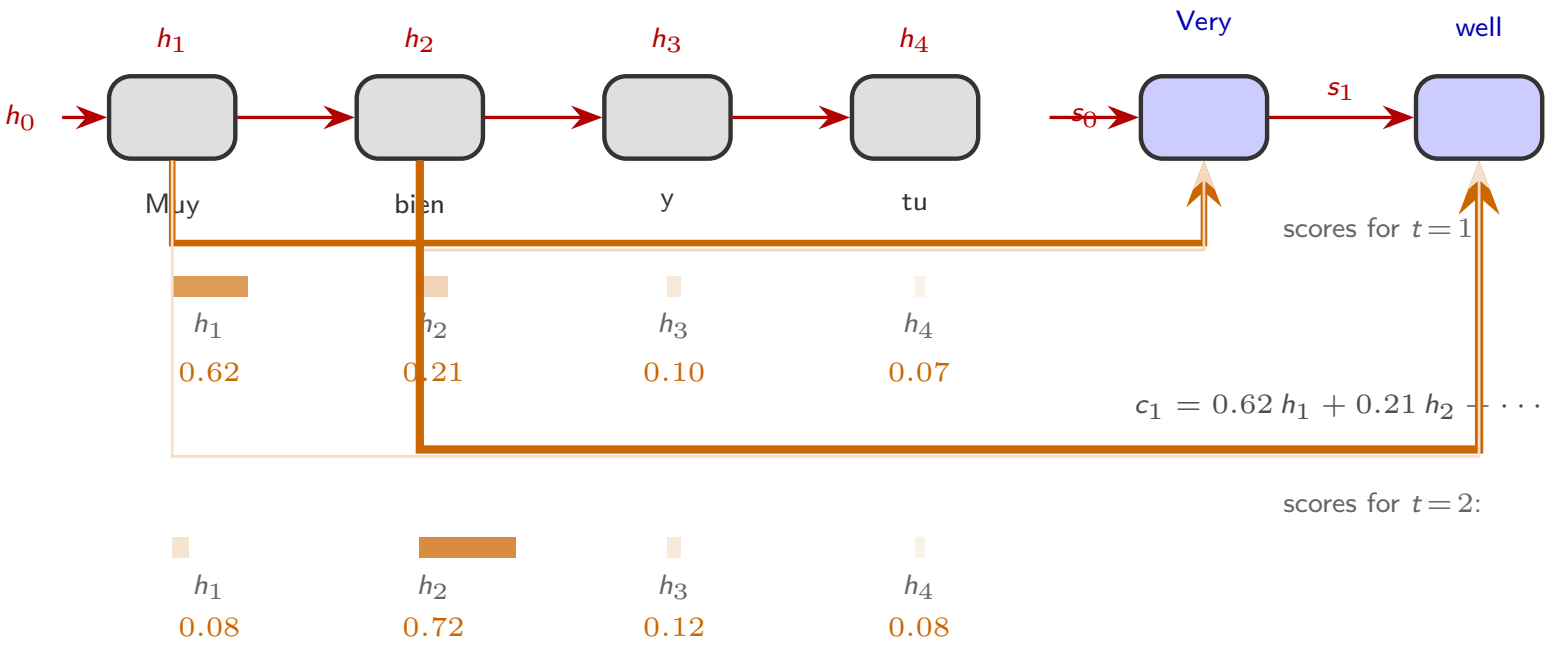
Attention Step by Step — Generating “Very” and “well”



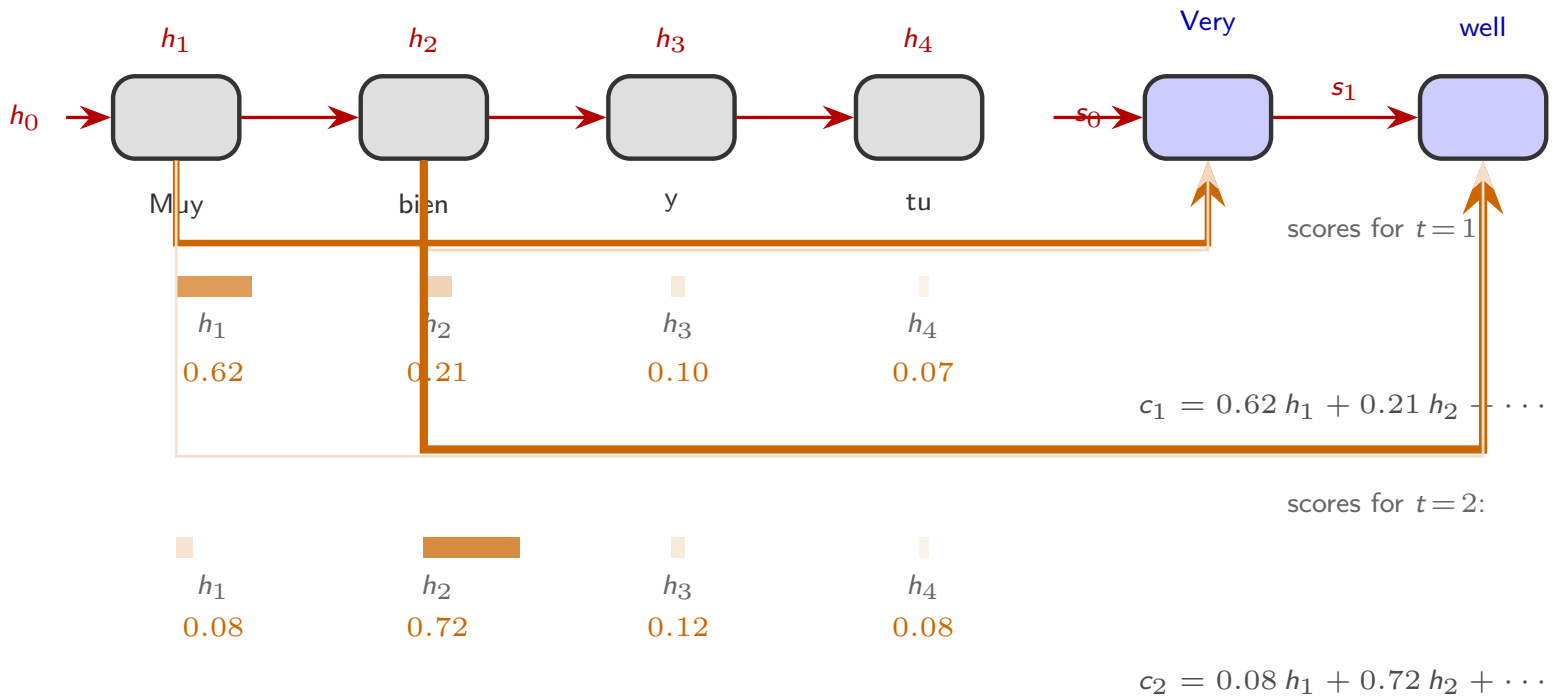
Attention Step by Step — Generating “Very” and “well”



Attention Step by Step — Generating “Very” and “well”



Attention Step by Step — Generating “Very” and “well”



The attention pattern *shifts* as the decoder advances: “Very” attends to “Muy”; “well” attends to “bien”. Each step computes its own distribution over the encoder.

Attention — The Three-Step Computation

At decoder step t (given decoder state s_{t-1} and encoder states $h_1, \ldots, h_T$):

Attention — The Three-Step Computation

At decoder step t (given decoder state s_{t-1} and encoder states $h_1, \ldots, h_T$): **Step 1 — Score.** For each encoder position $i = 1, \ldots, T$:

$$\tilde{\alpha}_{t,i} = \mathsf{score}(s_{t-1}, h_i) \in \mathbb{R}$$

Attention — The Three-Step Computation

At decoder step t (given decoder state s_{t-1} and encoder states $h_1, \ldots, h_T$): **Step 1 — Score.** For each encoder position $i = 1, \ldots, T$:

$$\tilde{\alpha}_{t,i} = \mathsf{score}(s_{t-1}, h_i) \in \mathbb{R}$$

Step 2 — Normalize. Convert scores to a probability distribution:

$$(\alpha_t)_i = \frac{\exp(\tilde{\alpha}_{t,i})}{\sum_{j=1}^{T} \exp(\tilde{\alpha}_{t,j})}, \qquad \alpha_t \in \Delta^{T-1}$$

Attention — The Three-Step Computation

At decoder step t (given decoder state s_{t-1} and encoder states $h_1, \ldots, h_T$): **Step 1 — Score.** For each encoder position $i = 1, \ldots, T$:

$$\tilde{\alpha}_{t,i} = \text{score}(s_{t-1}, h_i) \in \mathbb{R}$$

Step 2 — Normalize. Convert scores to a probability distribution:

$$(\alpha_t)_i = \frac{\exp(\tilde{\alpha}_{t,i})}{\sum_{j=1}^{T} \exp(\tilde{\alpha}_{t,j})}, \qquad \alpha_t \in \Delta^{T-1}$$

Step 3 — Aggregate. Form the **context vector**:

$$c_t = \sum_{i=1}^{T} (\alpha_t)_i \, h_i \in \mathbb{R}^{d_h}$$

Attention — The Three-Step Computation

At decoder step t (given decoder state s_{t-1} and encoder states $h_1, \ldots, h_T$): **Step 1 — Score.** For each encoder position $i = 1, \ldots, T$:

$$\tilde{\alpha}_{t,i} = \text{score}(s_{t-1}, h_i) \in \mathbb{R}$$

Step 2 — Normalize. Convert scores to a probability distribution:

$$(\alpha_t)_i = \frac{\exp(\tilde{\alpha}_{t,i})}{\sum_{j=1}^{T} \exp(\tilde{\alpha}_{t,j})}, \qquad \alpha_t \in \Delta^{T-1}$$

Step 3 — Aggregate. Form the **context vector**:

$$c_t = \sum_{i=1}^{T} (\alpha_t)_i \, h_i \in \mathbb{R}^{d_h}$$

The decoder then updates: $s_t = g(s_{t-1}, c_t, \hat{y}_{t-1})$.

Pause and Check

1. What happens if the attention weights are *uniform*, i.e., $(\alpha_t)_i = 1/T$ for all i? What does the context vector become?

Pause and Check

1. What happens if the attention weights are *uniform*, i.e., $(\alpha_t)_i = 1/T$ for all i? What does the context vector become?

   The context vector becomes $c_t = \frac{1}{T}\sum_{i=1}^{T} h_i$, the simple average of all encoder states. This is equivalent to a bag-of-words representation—all positional and sequential information is lost. Attention is useful precisely because it produces *non-uniform* weights that focus on the relevant positions.

Pause and Check

1. What happens if the attention weights are *uniform*, i.e., $(\alpha_t)_i = 1/T$ for all i? What does the context vector become?

   The context vector becomes $c_t = \frac{1}{T}\sum_{i=1}^{T} h_i$, the simple average of all encoder states. This is equivalent to a bag-of-words representation—all positional and sequential information is lost. Attention is useful precisely because it produces *non-uniform* weights that focus on the relevant positions.

2. In the encoder–decoder without attention, the decoder is initialized with $s_0 = h_T$. With attention, is $s_0 = h_T$ still needed? Why or why not?

Pause and Check

1. What happens if the attention weights are *uniform*, i.e., $(\alpha_t)_i = 1/T$ for all i? What does the context vector become?

   The context vector becomes $c_t = \frac{1}{T}\sum_{i=1}^{T} h_i$, the simple average of all encoder states. This is equivalent to a bag-of-words representation—all positional and sequential information is lost. Attention is useful precisely because it produces *non-uniform* weights that focus on the relevant positions.

2. In the encoder–decoder without attention, the decoder is initialized with $s_0 = h_T$. With attention, is $s_0 = h_T$ still needed? Why or why not?

   If s_0 does not depend on the input, the initial scores carry no information about the input, so the first attention distribution is content-independent—the decoder attends blind. With $s_0 = h_T$, the query is informed by a summary of the entire input, so attention is well-directed from the very first step.

The Attention Abstraction

The three-step structure is the same across all forms of attention. There are multiple ways to compute scores, but the overall pattern is fixed:

The Attention Abstraction

The three-step structure is the same across all forms of attention. There are multiple ways to compute scores, but the overall pattern is fixed:

The Core Abstraction: Score $\rightarrow$ Normalize $\rightarrow$ Aggregate

The Attention Abstraction

The three-step structure is the same across all forms of attention. There are multiple ways to compute scores, but the overall pattern is fixed:

The Core Abstraction: Score → Normalize → Aggregate

1. Compute a **score** for each source position (measuring relevance).

The Attention Abstraction

The three-step structure is the same across all forms of attention. There are multiple ways to compute scores, but the overall pattern is fixed:

The Core Abstraction: Score $\rightarrow$ Normalize $\rightarrow$ Aggregate

1. Compute a **score** for each source position (measuring relevance).
2. Normalize scores into a **probability distribution** $\alpha \in \Delta^{T-1}$ via softmax.

The Attention Abstraction

The three-step structure is the same across all forms of attention. There are multiple ways to compute scores, but the overall pattern is fixed:

The Core Abstraction: Score $\rightarrow$ Normalize $\rightarrow$ Aggregate

1. Compute a **score** for each source position (measuring relevance).
2. Normalize scores into a **probability distribution** $\alpha \in \Delta^{T-1}$ via softmax.
3. Form a **weighted average** $c = \sum_i \alpha_i \, v_i$ of source representations.

The Attention Abstraction

The three-step structure is the same across all forms of attention. There are multiple ways to compute scores, but the overall pattern is fixed:

The Core Abstraction: Score → Normalize → Aggregate

1. Compute a **score** for each source position (measuring relevance).
2. Normalize scores into a **probability distribution** $\alpha \in \Delta^{T-1}$ via softmax.
3. Form a **weighted average** $c = \sum_i \alpha_i v_i$ of source representations.

The output of attention is always a **convex combination of value vectors**. What changes between variants is *only Step 1*: how scores are computed.

Scoring via Dot Products

The dominant modern scoring function is **dot-product attention** (Luong et al., 2015), and the basis for the Transformer.

Scoring via Dot Products

The dominant modern scoring function is **dot-product attention** (Luong et al., 2015), and the basis for the Transformer.

Generate **queries** and **keys** as linear projections:

$$q_t = W_Q\, s_{t-1} \in \mathbb{R}^{d_k}, \qquad k_i = W_K\, h_i \in \mathbb{R}^{d_k}$$

Scoring via Dot Products

The dominant modern scoring function is **dot-product attention** (Luong et al., 2015), and the basis for the Transformer.

Generate **queries** and **keys** as linear projections:

$$q_t = W_Q\, s_{t-1} \in \mathbb{R}^{d_k}, \qquad k_i = W_K\, h_i \in \mathbb{R}^{d_k}$$

The score is the inner product:

$$\text{score}(s_{t-1}, h_i) = q_t^\top k_i = s_{t-1}^\top W_Q^\top W_K\, h_i$$

This is a bilinear form—cheap and parallelizable.

Queries, Keys, and Values

In dot-product scoring we introduced $q_t = W_Q s_{t-1}$ and $k_i = W_K h_i$.

Queries, Keys, and Values

In dot-product scoring we introduced $q_t = W_Q s_{t-1}$ and $k_i = W_K h_i$.

The context vector is $c_t = \sum_i (\alpha_t)_i \, h_i$—we are averaging the *raw encoder states*. But what if the information useful for *matching* differs from the information useful for *aggregation*?

Queries, Keys, and Values

In dot-product scoring we introduced $q_t = W_Q s_{t-1}$ and $k_i = W_K h_i$.

The context vector is $c_t = \sum_i (\alpha_t)_i \, h_i$—we are averaging the *raw encoder states*. But what if the information useful for *matching* differs from the information useful for *aggregation*?

Introduce a third projection: the **value** $v_i = W_V h_i \in \mathbb{R}^{d_v}$.

Queries, Keys, and Values

In dot-product scoring we introduced $q_t = W_Q s_{t-1}$ and $k_i = W_K h_i$.

The context vector is $c_t = \sum_i (\alpha_t)_i \, h_i$—we are averaging the *raw encoder states*. But what if the information useful for *matching* differs from the information useful for *aggregation*?

Introduce a third projection: the **value** $v_i = W_V h_i \in \mathbb{R}^{d_v}$.

Now the three-step computation becomes:

1. **Score:** $\tilde{\alpha}_{t,i} = q_t^\top k_i = (W_Q s_{t-1})^\top (W_K h_i)$
2. **Normalize:** $\alpha_t = \mathrm{softmax}(\tilde{\alpha}_t)$
3. **Aggregate:** $c_t = \sum_i (\alpha_t)_i \, v_i$ (using v_i, not h_i)

Queries, Keys, and Values

In dot-product scoring we introduced $q_t = W_Q s_{t-1}$ and $k_i = W_K h_i$.

The context vector is $c_t = \sum_i (\alpha_t)_i \, h_i$—we are averaging the *raw encoder states*. But what if the information useful for *matching* differs from the information useful for *aggregation*?

Introduce a third projection: the **value** $v_i = W_V h_i \in \mathbb{R}^{d_v}$.

Now the three-step computation becomes:

1. **Score:** $\tilde{\alpha}_{t,i} = q_t^\top k_i = (W_Q s_{t-1})^\top (W_K h_i)$
2. **Normalize:** $\alpha_t = \mathrm{softmax}(\tilde{\alpha}_t)$
3. **Aggregate:** $c_t = \sum_i (\alpha_t)_i \, v_i$ (using v_i, not h_i)

This is the **query–key–value (QKV) abstraction**. Matching (K, Q) and aggregation (V) are *decoupled*: the model can learn to match on one set of features while returning different features as output.

Building the Score Matrix

Stack all queries and keys into matrices:

$$Q = [q_1 \mid \cdots \mid q_{T'}] \in \mathbb{R}^{d_k \times T'}, \quad K = [k_1 \mid \cdots \mid k_T] \in \mathbb{R}^{d_k \times T}$$

$$V = [v_1 \mid \cdots \mid v_T] \in \mathbb{R}^{d_v \times T}$$

Building the Score Matrix

Stack all queries and keys into matrices:

$$Q = [q_1 \mid \cdots \mid q_{T'}] \in \mathbb{R}^{d_k \times T'}, \quad K = [k_1 \mid \cdots \mid k_T] \in \mathbb{R}^{d_k \times T}$$
$$V = [v_1 \mid \cdots \mid v_T] \in \mathbb{R}^{d_v \times T}$$

Step 1. Compute all scores simultaneously:

$$K^\top Q = \begin{bmatrix} k_1^\top q_1 & k_1^\top q_2 & k_1^\top q_3 & k_1^\top q_4 \\ k_2^\top q_1 & k_2^\top q_2 & k_2^\top q_3 & k_2^\top q_4 \\ k_3^\top q_1 & k_3^\top q_2 & k_3^\top q_3 & k_3^\top q_4 \\ k_4^\top q_1 & k_4^\top q_2 & k_4^\top q_3 & k_4^\top q_4 \end{bmatrix} \in \mathbb{R}^{T \times T'}$$

Building the Score Matrix

Stack all queries and keys into matrices:

$$Q = [q_1 \mid \cdots \mid q_{T'}] \in \mathbb{R}^{d_k \times T'}, \quad K = [k_1 \mid \cdots \mid k_T] \in \mathbb{R}^{d_k \times T}$$
$$V = [v_1 \mid \cdots \mid v_T] \in \mathbb{R}^{d_v \times T}$$

Step 1. Compute all scores simultaneously:

$$K^\top Q = \begin{bmatrix} k_1^\top q_1 & k_1^\top q_2 & k_1^\top q_3 & k_1^\top q_4 \\ k_2^\top q_1 & k_2^\top q_2 & k_2^\top q_3 & k_2^\top q_4 \\ k_3^\top q_1 & k_3^\top q_2 & k_3^\top q_3 & k_3^\top q_4 \\ k_4^\top q_1 & k_4^\top q_2 & k_4^\top q_3 & k_4^\top q_4 \end{bmatrix} \in \mathbb{R}^{T \times T'}$$

	q_1 (Very)	q_2 (well)	q_3 (and)	q_4 (you)
k_1 (Muy)	**3.1**	0.4	0.2	0.1
k_2 (bien)	0.5	**3.8**	0.3	0.2
k_3 (y)	0.2	0.1	**2.9**	0.3
k_4 (tu)	0.1	0.3	0.4	**3.5**

Building the Score Matrix

Stack all queries and keys into matrices:

$$Q = [q_1 \mid \cdots \mid q_{T'}] \in \mathbb{R}^{d_k \times T'}, \quad K = [k_1 \mid \cdots \mid k_T] \in \mathbb{R}^{d_k \times T}$$
$$V = [v_1 \mid \cdots \mid v_T] \in \mathbb{R}^{d_v \times T}$$

Step 1. Compute all scores simultaneously:

$$K^\top Q = \begin{bmatrix} k_1^\top q_1 & k_1^\top q_2 & k_1^\top q_3 & k_1^\top q_4 \\ k_2^\top q_1 & k_2^\top q_2 & k_2^\top q_3 & k_2^\top q_4 \\ k_3^\top q_1 & k_3^\top q_2 & k_3^\top q_3 & k_3^\top q_4 \\ k_4^\top q_1 & k_4^\top q_2 & k_4^\top q_3 & k_4^\top q_4 \end{bmatrix} \in \mathbb{R}^{T \times T'}$$

	q_1 (Very)	q_2 (well)	q_3 (and)	q_4 (you)
k_1 (Muy)	**3.1**	0.4	0.2	0.1
k_2 (bien)	0.5	**3.8**	0.3	0.2
k_3 (y)	0.2	0.1	**2.9**	0.3
k_4 (tu)	0.1	0.3	0.4	**3.5**

Entry (i, t) measures: "how relevant is source position i to output position t?"

Softmax and Aggregation — Completing the Computation

Step 2. Scale by $\sqrt{d_k}$ and apply column-wise softmax: $\mathrm{softmax}\left(\frac{K^\top Q}{\sqrt{d_k}}\right) = [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4] \in \mathbb{R}^{T \times T'}$. Each column α_t is a probability distribution over the T source positions.

Softmax and Aggregation — Completing the Computation

Step 2. Scale by $\sqrt{d_k}$ and apply column-wise softmax: $\mathrm{softmax}\left(\frac{K^\top Q}{\sqrt{d_k}}\right) = [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4] \in \mathbb{R}^{T \times T'}$. Each column α_t is a probability distribution over the T source positions.

Applying to the score matrix from the previous slide (with $d_k = 4$):

$$\begin{bmatrix} 0.62 & 0.05 & 0.04 & 0.03 \\ 0.06 & 0.75 & 0.04 & 0.03 \\ 0.04 & 0.03 & 0.66 & 0.05 \\ 0.03 & 0.05 & 0.06 & 0.72 \end{bmatrix} \approx [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4]$$

Softmax and Aggregation — Completing the Computation

Step 2. Scale by $\sqrt{d_k}$ and apply column-wise softmax: $\mathrm{softmax}\left(\frac{K^\top Q}{\sqrt{d_k}}\right) = [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4] \in \mathbb{R}^{T \times T'}$. Each column α_t is a probability distribution over the T source positions.

Applying to the score matrix from the previous slide (with $d_k = 4$):

$$\begin{bmatrix} 0.62 & 0.05 & 0.04 & 0.03 \\ 0.06 & 0.75 & 0.04 & 0.03 \\ 0.04 & 0.03 & 0.66 & 0.05 \\ 0.03 & 0.05 & 0.06 & 0.72 \end{bmatrix} \approx [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4]$$

Each column sums to 1. The near-diagonal structure reflects the word-to-word alignment.

Softmax and Aggregation — Completing the Computation

Step 2. Scale by $\sqrt{d_k}$ and apply column-wise softmax:
$\mathrm{softmax}\left(\frac{K^\top Q}{\sqrt{d_k}}\right) = [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4] \in \mathbb{R}^{T \times T'}$. Each column α_t is a probability distribution over the T source positions.

Applying to the score matrix from the previous slide (with $d_k = 4$):

$$\begin{bmatrix} 0.62 & 0.05 & 0.04 & 0.03 \\ 0.06 & 0.75 & 0.04 & 0.03 \\ 0.04 & 0.03 & 0.66 & 0.05 \\ 0.03 & 0.05 & 0.06 & 0.72 \end{bmatrix} \approx [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4]$$

Each column sums to 1. The near-diagonal structure reflects the word-to-word alignment.

Step 3. Multiply by the value matrix:
$[c_1 \mid \cdots \mid c_{T'}] = V[\alpha_1 \mid \cdots \mid \alpha_{T'}] \in \mathbb{R}^{d_v \times T'}$, where $c_t = \sum_{i=1}^{T} (\alpha_t)_i \, v_i$.

Softmax and Aggregation — Completing the Computation

Step 2. Scale by $\sqrt{d_k}$ and apply column-wise softmax:
$\mathrm{softmax}\left(\frac{K^\top Q}{\sqrt{d_k}}\right) = [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4] \in \mathbb{R}^{T \times T'}$. Each column α_t is a probability distribution over the T source positions.

Applying to the score matrix from the previous slide (with $d_k = 4$):

$$\begin{bmatrix} 0.62 & 0.05 & 0.04 & 0.03 \\ 0.06 & 0.75 & 0.04 & 0.03 \\ 0.04 & 0.03 & 0.66 & 0.05 \\ 0.03 & 0.05 & 0.06 & 0.72 \end{bmatrix} \approx [\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4]$$

Each column sums to 1. The near-diagonal structure reflects the word-to-word alignment.

Step 3. Multiply by the value matrix:
$[c_1 \mid \cdots \mid c_{T'}] = V[\alpha_1 \mid \cdots \mid \alpha_{T'}] \in \mathbb{R}^{d_v \times T'}$, where $c_t = \sum_{i=1}^{T} (\alpha_t)_i \, v_i$.

The complete formula:

$$\boxed{\mathrm{Attention}(Q, K, V) = V \, \mathrm{softmax}\left(\frac{K^\top Q}{\sqrt{d_k}}\right) \in \mathbb{R}^{d_v \times T'}}$$

Instantiation and Complexity

In the encoder–decoder setting:

Instantiation and Complexity

In the encoder–decoder setting:

- Queries from the **decoder**: $q_t = W_Q\, s_{t-1}$ for $t = 1, \ldots, T'$

Instantiation and Complexity

In the encoder–decoder setting:

- Queries from the **decoder**: $q_t = W_Q\, s_{t-1}$ for $t = 1, \ldots, T'$
- Keys from the **encoder**: $k_i = W_K\, h_i$ for $i = 1, \ldots, T$

Instantiation and Complexity

In the encoder–decoder setting:

- Queries from the **decoder**: $q_t = W_Q\, s_{t-1}$ for $t = 1, \ldots, T'$
- Keys from the **encoder**: $k_i = W_K\, h_i$ for $i = 1, \ldots, T$
- Values from the **encoder**: $v_i = W_V\, h_i$ for $i = 1, \ldots, T$

Instantiation and Complexity

In the encoder–decoder setting:

- Queries from the **decoder**: $q_t = W_Q\, s_{t-1}$ for $t = 1, \ldots, T'$
- Keys from the **encoder**: $k_i = W_K\, h_i$ for $i = 1, \ldots, T$
- Values from the **encoder**: $v_i = W_V\, h_i$ for $i = 1, \ldots, T$

Projection matrices: $W_Q \in \mathbb{R}^{d_k \times d_s}$, $W_K \in \mathbb{R}^{d_k \times d_h}$, $W_V \in \mathbb{R}^{d_v \times d_h}$.

Instantiation and Complexity

In the encoder–decoder setting:

- Queries from the **decoder**: $q_t = W_Q\, s_{t-1}$ for $t = 1, \ldots, T'$
- Keys from the **encoder**: $k_i = W_K\, h_i$ for $i = 1, \ldots, T$
- Values from the **encoder**: $v_i = W_V\, h_i$ for $i = 1, \ldots, T$

Projection matrices: $W_Q \in \mathbb{R}^{d_k \times d_s}$, $W_K \in \mathbb{R}^{d_k \times d_h}$, $W_V \in \mathbb{R}^{d_v \times d_h}$.
Complexity:

- Computing $K^\top Q$: $O(T \cdot T' \cdot d_k)$
- Forming context vectors: $O(T \cdot T' \cdot d_v)$
- Total: $\boxed{O(T \cdot T' \cdot \max(d_k, d_v))}$ — **quadratic in sequence lengths**

Encoder–Decoder with Attention

Input: Source tokens $x_1, \ldots, x_T$; target tokens $y_1, \ldots, y_{T'}$
Output: Predictions $\hat{y}_1, \ldots, \hat{y}_{T'}$

`/* Encode */`

$h_0 \leftarrow \mathbf{0}$

for $t = 1, \ldots, T$ **do**

$\quad h_t \leftarrow \tanh(W_h\, h_{t-1} + W_x\, x_t + b)$

end

Compute $k_i = W_K h_i$,
$v_i = W_V h_i$ for all i

`/* Decode with attention */`

$s_0 \leftarrow h_T$

for $t = 1, \ldots, T'$ **do**

$\quad q_t \leftarrow W_Q\, s_{t-1}$

$\quad \alpha_t \leftarrow \mathrm{softmax}\big([q_t^\top k_1,\ \ldots,\ q_t^\top k_T]/\sqrt{d_k}\big)$

$\quad c_t \leftarrow \sum_{i=1}^{T} (\alpha_t)_i\, v_i$ `// attention context`

$\quad s_t \leftarrow f_{\mathrm{dec}}(s_{t-1},\ [c_t;\ y_{t-1}],\ \theta_{\mathrm{dec}})$

$\quad \hat{y}_t \leftarrow \mathrm{softmax}(W_o\, s_t + c)$

end

return $\hat{y}_1, \ldots, \hat{y}_{T'}$

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

1. **Sequential computation.**

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

1. **Sequential computation.**
   - The encoder processes tokens one at a time: h_t depends on h_{t-1}.

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

1. **Sequential computation.**
   - The encoder processes tokens one at a time: h_t depends on h_{t-1}.
   - Modern GPUs execute thousands of operations in parallel, but the RNN forces a serial chain of T steps.

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

1. **Sequential computation.**
   - The encoder processes tokens one at a time: h_t depends on h_{t-1}.
   - Modern GPUs execute thousands of operations in parallel, but the RNN forces a serial chain of T steps.
2. **Long path lengths in recurrent representations.**

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

1. **Sequential computation.**
   - The encoder processes tokens one at a time: h_t depends on h_{t-1}.
   - Modern GPUs execute thousands of operations in parallel, but the RNN forces a serial chain of T steps.

2. **Long path lengths in recurrent representations.**
   - Path length between x_i and x_j in the encoder is $|i - j|$ steps.

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

1. **Sequential computation.**
   - The encoder processes tokens one at a time: h_t depends on h_{t-1}.
   - Modern GPUs execute thousands of operations in parallel, but the RNN forces a serial chain of T steps.

2. **Long path lengths in recurrent representations.**
   - Path length between x_i and x_j in the encoder is $|i - j|$ steps.
   - Attention gives $O(1)$ paths between encoder and decoder—but not *within* the encoder.

What Attention Does Not Fix

Attention resolves the information bottleneck, but two problems remain:

1. **Sequential computation.**
   - The encoder processes tokens one at a time: h_t depends on h_{t-1}.
   - Modern GPUs execute thousands of operations in parallel, but the RNN forces a serial chain of T steps.

2. **Long path lengths in recurrent representations.**
   - Path length between x_i and x_j in the encoder is $|i - j|$ steps.
   - Attention gives $O(1)$ paths between encoder and decoder—but not *within* the encoder.

The Question That Motivates the Next Lecture

What if we **removed the recurrence entirely** and used attention as the **sole mechanism** for contextual representations?

— Vaswani et al., "Attention Is All You Need" (2017) $\rightarrow$ next lecture.

Transition: From Architecture to Preprocessing

We have now seen three architectures for sequential data:

1. Vanilla RNN (sequential, bottleneck in gradients)
2. LSTM (gating resolves gradient bottleneck)
3. Encoder–decoder + attention (resolves information bottleneck)

Transition: From Architecture to Preprocessing

We have now seen three architectures for sequential data:

1. Vanilla RNN (sequential, bottleneck in gradients)
2. LSTM (gating resolves gradient bottleneck)
3. Encoder–decoder + attention (resolves information bottleneck)

All three require the same first step: converting discrete text into continuous vectors. At the start of the lecture we deferred the question of how the vocabulary $\mathcal{V}$ is constructed.

Transition: From Architecture to Preprocessing

We have now seen three architectures for sequential data:

1. Vanilla RNN (sequential, bottleneck in gradients)
2. LSTM (gating resolves gradient bottleneck)
3. Encoder–decoder + attention (resolves information bottleneck)

All three require the same first step: converting discrete text into continuous vectors. At the start of the lecture we deferred the question of how the vocabulary $\mathcal{V}$ is constructed.

This matters in practice: the choice of tokenization affects sequence length, vocabulary size, and even what the model can compute (e.g., arithmetic). We now develop this machinery.

What Should a Token Be?

The design choice is the **granularity** of the vocabulary $\mathcal{V}$:

What Should a Token Be?

The design choice is the **granularity** of the vocabulary $\mathcal{V}$:

Granularity	$\lvert\mathcal{V}\rvert$	**Seq. length**	**Tradeoff**
Word-level	$\sim 10^5$–10^6	Short	Huge vocab; OOV problem
Character-level	~ 100	Very long	Tiny vocab; hard to learn
Subword	$\sim 3\times10^4$–10^5	Moderate	**Balanced**

What Should a Token Be?

The design choice is the **granularity** of the vocabulary $\mathcal{V}$:

Granularity	$\lvert\mathcal{V}\rvert$	**Seq. length**	**Tradeoff**
Word-level	$\sim 10^5$–10^6	Short	Huge vocab; OOV problem
Character-level	~ 100	Very long	Tiny vocab; hard to learn
Subword	$\sim 3\times10^4$–10^5	Moderate	**Balanced**

Subword tokenization is the modern standard: common words $\rightarrow$ single tokens; rare/novel words $\rightarrow$ decomposed into known subwords. The dominant algorithm: **Byte Pair Encoding** (BPE).

Tokenization in Action — Three Granularities

Consider the input: “unhappiness is overwhelming”

Tokenization in Action — Three Granularities

Consider the input: “unhappiness is overwhelming”

Word-level ($|\mathcal{V}| \sim 10^5$–10^6):

`unhappiness` `is` `overwhelming` — 3 tokens, but “unhappiness” must be in $\mathcal{V}$

Tokenization in Action — Three Granularities

Consider the input: “unhappiness is overwhelming”

Word-level ($|\mathcal{V}| \sim 10^5$–10^6):

`unhappiness` `is` `overwhelming` — 3 tokens, but “unhappiness” must be in $\mathcal{V}$

Character-level ($|\mathcal{V}| \sim 100$):

`u` `n` `h` `a` `p` `p` `i` `n` `e` `s` `s` `␣` `i` `s` `␣` $\cdots$ — 27 tokens

Tokenization in Action — Three Granularities

Consider the input: "unhappiness is overwhelming"

Word-level ($|\mathcal{V}| \sim 10^5$–10^6):

`unhappiness` `is` `overwhelming` — 3 tokens, but "unhappiness" must be in $\mathcal{V}$

Character-level ($|\mathcal{V}| \sim 100$):

`u` `n` `h` `a` `p` `p` `i` `n` `e` `s` `s` `␣` `i` `s` `␣` $\cdots$ — 27 tokens

Subword / BPE ($|\mathcal{V}| \sim 3{\times}10^4$–$10^5$):

`un` `happiness` `_is` `_over` `whelm` `ing` — 6 tokens, morphologically meaningful splits

Tokenization in Action — Three Granularities

Consider the input: "unhappiness is overwhelming"

Word-level ($|\mathcal{V}| \sim 10^5$–10^6):

`unhappiness` `is` `overwhelming` — 3 tokens, but "unhappiness" must be in $\mathcal{V}$

Character-level ($|\mathcal{V}| \sim 100$):

`u` `n` `h` `a` `p` `p` `i` `n` `e` `s` `s` `_` `i` `s` `_` $\cdots$ — 27 tokens

Subword / BPE ($|\mathcal{V}| \sim 3{\times}10^4$–$10^5$):

`un` `happiness` `_is` `_over` `whelm` `ing` — 6 tokens, morphologically meaningful splits

Subword tokenization decomposes rare words into known pieces while keeping common words intact. This is the modern standard.

Vocabulary Size Tradeoff and Tokenization Artifacts

Larger $|\mathcal{V}|$ (e.g., 100K):

- Shorter sequences, more info per context window
- Larger E, more parameters, sparser statistics

Smaller $|\mathcal{V}|$ (e.g., 8K):

- Longer sequences, more computation
- Smaller E, denser statistics, better rare-word handling

Model	$\lvert\mathcal{V}\rvert$	**Type**
GPT-2	50K	BPE
BERT	30K	WordPiece
LLaMA	32K	SentencePiece
GPT-4	$\sim$100K	BPE

Vocabulary Size Tradeoff and Tokenization Artifacts

Larger $|\mathcal{V}|$ (e.g., 100K):

- Shorter sequences, more info per context window
- Larger E, more parameters, sparser statistics

Smaller $|\mathcal{V}|$ (e.g., 8K):

- Longer sequences, more computation
- Smaller E, denser statistics, better rare-word handling

Tokenization artifacts:

Model	$\lvert\mathcal{V}\rvert$	**Type**
GPT-2	50K	BPE
BERT	30K	WordPiece
LLaMA	32K	SentencePiece
GPT-4	$\sim$100K	BPE

Vocabulary Size Tradeoff and Tokenization Artifacts

Larger $|\mathcal{V}|$ (e.g., 100K):

- Shorter sequences, more info per context window
- Larger E, more parameters, sparser statistics

Smaller $|\mathcal{V}|$ (e.g., 8K):

- Longer sequences, more computation
- Smaller E, denser statistics, better rare-word handling

Model	$\lvert\mathcal{V}\rvert$	**Type**
GPT-2	50K	BPE
BERT	30K	WordPiece
LLaMA	32K	SentencePiece
GPT-4	$\sim$100K	BPE

Tokenization artifacts:

- **Arithmetic:** "123+456" $\rightarrow$ ["12","3","+","4","56"]—digit boundaries misalign with token boundaries, degrading arithmetic.

Vocabulary Size Tradeoff and Tokenization Artifacts

Larger $|\mathcal{V}|$ (e.g., 100K):

- Shorter sequences, more info per context window
- Larger E, more parameters, sparser statistics

Smaller $|\mathcal{V}|$ (e.g., 8K):

- Longer sequences, more computation
- Smaller E, denser statistics, better rare-word handling

Model	$\lvert\mathcal{V}\rvert$	**Type**
GPT-2	50K	BPE
BERT	30K	WordPiece
LLaMA	32K	SentencePiece
GPT-4	$\sim$100K	BPE

Tokenization artifacts:

- **Arithmetic:** "123+456" $\rightarrow$ ["12","3","+","4","56"]—digit boundaries misalign with token boundaries, degrading arithmetic.
- **Multilingual:** Non-English text is over-segmented (2–$3\times$ more tokens), reducing effective context window.

The BPE Algorithm (Sennrich et al., 2016)

Input: Training corpus $\mathcal{C}$;
target vocabulary size N
Output: Vocabulary $\mathcal{V}$;
ordered list of merge rules

$\mathcal{V} \leftarrow$ all individual bytes in $\mathcal{C} \cup \{\langle \texttt{BOS} \rangle, \langle \texttt{EOS} \rangle, \langle \texttt{PAD} \rangle, \langle \texttt{UNK} \rangle\}$
Tokenize $\mathcal{C}$ at the byte level
while $|\mathcal{V}| < N$ **do**
  Count all adjacent token pairs (a, b) in the tokenized corpus
  $(a^*, b^*) \leftarrow$ most frequent pair
  Create new token $a^* b^*$ by concatenation
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{a^* b^*\}$
  Replace every adjacent (a^*, b^*) in the corpus with $a^* b^*$
end
return $\mathcal{V}$, merge rules (in order learned)

At inference: tokenize new text by applying merge rules greedily, in the order learned.

BPE Starts from Bytes — Why This Matters

A critical detail: modern BPE implementations (GPT-2 onward) initialize $\mathcal{V}$ with the 256 **byte values**, not just ASCII characters. Every byte sequence is representable, so:

BPE Starts from Bytes — Why This Matters

A critical detail: modern BPE implementations (GPT-2 onward) initialize $\mathcal{V}$ with the 256 **byte values**, not just ASCII characters. Every byte sequence is representable, so:

- **Any Unicode document is tokenizable.** Emoji, CJK characters, mathematical symbols—all decompose into bytes that are in the base vocabulary.

BPE Starts from Bytes — Why This Matters

A critical detail: modern BPE implementations (GPT-2 onward) initialize $\mathcal{V}$ with the 256 **byte values**, not just ASCII characters. Every byte sequence is representable, so:

- **Any Unicode document is tokenizable.** Emoji, CJK characters, mathematical symbols—all decompose into bytes that are in the base vocabulary.
- **No $\langle$`UNK`$\rangle$ needed.** Word-level tokenizers must map unseen words to a special unknown token. BPE never encounters an unknown input.

BPE Starts from Bytes — Why This Matters

A critical detail: modern BPE implementations (GPT-2 onward) initialize $\mathcal{V}$ with the 256 **byte values**, not just ASCII characters. Every byte sequence is representable, so:

- **Any Unicode document is tokenizable.** Emoji, CJK characters, mathematical symbols—all decompose into bytes that are in the base vocabulary.
- **No $\langle$`UNK`$\rangle$ needed.** Word-level tokenizers must map unseen words to a special unknown token. BPE never encounters an unknown input.

Example: Consider the input "I'm ☺ today".

- **Word-level:** `[I'm, ⟨UNK⟩, today]` — the emoji is lost
- **Byte-level BPE:** `[I, 'm, _☺, _today]` — ☺ is encoded via its UTF-8 bytes, then merged if frequent enough in training data

BPE Starts from Bytes — Why This Matters

A critical detail: modern BPE implementations (GPT-2 onward) initialize $\mathcal{V}$ with the 256 **byte values**, not just ASCII characters. Every byte sequence is representable, so:

- **Any Unicode document is tokenizable.** Emoji, CJK characters, mathematical symbols—all decompose into bytes that are in the base vocabulary.
- **No $\langle$`UNK`$\rangle$ needed.** Word-level tokenizers must map unseen words to a special unknown token. BPE never encounters an unknown input.

Example: Consider the input "I'm ☺ today".

- **Word-level:** `[I'm, ⟨UNK⟩, today]` — the emoji is lost
- **Byte-level BPE:** `[I, 'm, _☺, _today]` — ☺ is encoded via its UTF-8 bytes, then merged if frequent enough in training data

This universality is one reason BPE became the standard: the tokenizer never fails on valid input.

BPE — Example

Corpus: `"low low lower lowest lowly"`

BPE — Example

Corpus: `"low low lower lowest lowly"`

Iter.	Most frequent pair	New token	Added to $\mathcal{V}$
0	(initial)	—	`l, o, w, e, r, s, t, y, _`, …
1	(`l`, `o`) — 5 times	`lo`	✓
2	(`lo`, `w`) — 5 times	`low`	✓
3	(`low`, `_`) — 2 times	`low_`	✓
4	(`low`, `e`) — 2 times	`lowe`	✓
5	(`lowe`, `r`)	`lower`	✓

BPE — Example

Corpus: `"low low lower lowest lowly"`

Iter.	Most frequent pair	New token	Added to $\mathcal{V}$
0	(initial)	—	`l, o, w, e, r, s, t, y, _`, …
1	(`l`, `o`) — 5 times	`lo`	✓
2	(`lo`, `w`) — 5 times	`low`	✓
3	(`low`, `_`) — 2 times	`low_`	✓
4	(`low`, `e`) — 2 times	`lowe`	✓
5	(`lowe`, `r`)	`lower`	✓

BPE implicitly learns morphological structure—common stems and affixes emerge as tokens.

Pause and Check

Using the five merge rules from the previous slide, tokenize each word by applying merges greedily in the order learned.

Pause and Check

Using the five merge rules from the previous slide, tokenize each word by applying merges greedily in the order learned.

1. **“lowest”**

Pause and Check

Using the five merge rules from the previous slide, tokenize each word by applying merges greedily in the order learned.

1. **"lowest"**
   `l,o,w,e,s,t` $\xrightarrow{1}$ `lo,w,e,s,t` $\xrightarrow{2}$ `low,e,s,t` $\xrightarrow{4}$ `lowe,s,t`.
   No further merges apply. Result: `lowe+s+t` (3 tokens).

   Note: `est` is *not* in $\mathcal{V}$—BPE does not "know" English morphology, only learned merges.

Pause and Check

Using the five merge rules from the previous slide, tokenize each word by applying merges greedily in the order learned.

1. **"lowest"**
   `l,o,w,e,s,t` $\xrightarrow{1}$ `lo,w,e,s,t` $\xrightarrow{2}$ `low,e,s,t` $\xrightarrow{4}$ `lowe,s,t`.
   No further merges apply. Result: `lowe+s+t` (3 tokens).

   Note: `est` is *not* in $\mathcal{V}$—BPE does not "know" English morphology, only learned merges.

2. **"lowly"**

Pause and Check

Using the five merge rules from the previous slide, tokenize each word by applying merges greedily in the order learned.

1. **"lowest"**
   `l,o,w,e,s,t` $\xrightarrow{1}$ `lo,w,e,s,t` $\xrightarrow{2}$ `low,e,s,t` $\xrightarrow{4}$ `lowe,s,t`.
   No further merges apply. Result: `lowe+s+t` (3 tokens).

   Note: `est` is *not* in $\mathcal{V}$—BPE does not "know" English morphology, only learned merges.

2. **"lowly"**
   `l,o,w,l,y` $\xrightarrow{1}$ `lo,w,l,y` $\xrightarrow{2}$ `low,l,y`.
   No further merges apply. Result: `low+l+y` (3 tokens).

Pause and Check

Using the five merge rules from the previous slide, tokenize each word by applying merges greedily in the order learned.

1. **"lowest"**
   `l,o,w,e,s,t` $\xrightarrow{1}$ `lo,w,e,s,t` $\xrightarrow{2}$ `low,e,s,t` $\xrightarrow{4}$ `lowe,s,t`.
   No further merges apply. Result: `lowe+s+t` (3 tokens).

   Note: `est` is *not* in $\mathcal{V}$—BPE does not "know" English morphology, only learned merges.

2. **"lowly"**
   `l,o,w,l,y` $\xrightarrow{1}$ `lo,w,l,y` $\xrightarrow{2}$ `low,l,y`.
   No further merges apply. Result: `low+l+y` (3 tokens).

3. **"lowering"** (unseen during training)

Pause and Check

Using the five merge rules from the previous slide, tokenize each word by applying merges greedily in the order learned.

1. **"lowest"**
   `l,o,w,e,s,t` $\xrightarrow{1}$ `lo,w,e,s,t` $\xrightarrow{2}$ `low,e,s,t` $\xrightarrow{4}$ `lowe,s,t`.
   No further merges apply. Result: `lowe+s+t` (3 tokens).

   Note: `est` is *not* in $\mathcal{V}$—BPE does not "know" English morphology, only learned merges.

2. **"lowly"**
   `l,o,w,l,y` $\xrightarrow{1}$ `lo,w,l,y` $\xrightarrow{2}$ `low,l,y`.
   No further merges apply. Result: `low+l+y` (3 tokens).

3. **"lowering"** (unseen during training)
   `l,o,w,e,r,i,n,g` $\xrightarrow{1}$ `lo,w,e,r,i,n,g` $\xrightarrow{2}$ `low,e,r,i,n,g` $\xrightarrow{4}$ `lowe,r,i,n,g` $\xrightarrow{5}$ `lower,i,n,g`.
   Result: `lower+i+n+g` (4 tokens). BPE handles unseen words by decomposition.

Pause and Check

Each of the following LLM behaviors is a direct consequence of tokenization. Explain why.

Pause and Check

Each of the following LLM behaviors is a direct consequence of tokenization. Explain why.

1. LLMs struggle to count how many times the letter “r” appears in “strawberry.”

Pause and Check

Each of the following LLM behaviors is a direct consequence of tokenization. Explain why.

1. LLMs struggle to count how many times the letter "r" appears in "strawberry."

   A BPE tokenizer might encode "strawberry" as `str+aw+berry` or similar. The model never sees individual characters—it sees subword tokens. Counting a specific letter requires reasoning *below* the granularity of the input representation, which the model was never trained to do.

Pause and Check

Each of the following LLM behaviors is a direct consequence of tokenization. Explain why.

1. LLMs struggle to count how many times the letter "r" appears in "strawberry."

   A BPE tokenizer might encode "strawberry" as `str+aw+berry` or similar. The model never sees individual characters—it sees subword tokens. Counting a specific letter requires reasoning *below* the granularity of the input representation, which the model was never trained to do.

2. LLMs perform worse on arithmetic with large numbers (e.g., $4827 + 3916$) than small ones.

Pause and Check

Each of the following LLM behaviors is a direct consequence of tokenization. Explain why.

1. LLMs struggle to count how many times the letter "r" appears in "strawberry."

   A BPE tokenizer might encode "strawberry" as `str+aw+berry` or similar. The model never sees individual characters—it sees subword tokens. Counting a specific letter requires reasoning *below* the granularity of the input representation, which the model was never trained to do.

2. LLMs perform worse on arithmetic with large numbers (e.g., $4827 + 3916$) than small ones.

   Digit boundaries and token boundaries are misaligned: "4827" might tokenize as 48+27. The model must implicitly recover place-value structure from tokens that split digits at arbitrary positions. Small numbers are more likely to be single tokens, making the arithmetic trivial pattern matching.

Pause and Check

3. Code (e.g., Python) typically consumes 2–$3\times$ more tokens than the same "amount" of English prose.

Pause and Check

3. Code (e.g., Python) typically consumes 2–$3\times$ more tokens than the same "amount" of English prose.

   BPE vocabularies are learned primarily from natural language corpora, so common English words and phrases become single tokens. Code contains many short identifiers, punctuation sequences (`==`, `():`, `->`), and indentation that are less frequent in training data and therefore split into more tokens. This reduces the effective context window when processing code.

Pause and Check

3. Code (e.g., Python) typically consumes 2–$3\times$ more tokens than the same "amount" of English prose.

   BPE vocabularies are learned primarily from natural language corpora, so common English words and phrases become single tokens. Code contains many short identifiers, punctuation sequences (`==`, `():`, `->`), and indentation that are less frequent in training data and therefore split into more tokens. This reduces the effective context window when processing code.

4. Two prompts with identical meaning can produce different-quality outputs depending on phrasing. Why might tokenization contribute to this?

Pause and Check

3. Code (e.g., Python) typically consumes 2–$3\times$ more tokens than the same "amount" of English prose.

   BPE vocabularies are learned primarily from natural language corpora, so common English words and phrases become single tokens. Code contains many short identifiers, punctuation sequences (`==`, `():`, `->`), and indentation that are less frequent in training data and therefore split into more tokens. This reduces the effective context window when processing code.

4. Two prompts with identical meaning can produce different-quality outputs depending on phrasing. Why might tokenization contribute to this?

   Different phrasings produce different token sequences, and the model's learned distribution is over *tokens*, not meanings. Different tokenizations put the model in regions where it has seen less training data, leading to weaker predictions even if the semantic content is the same. This is one reason "prompt engineering" matters: small wording changes affect tokenization.

Summary

1. **Grounding x_t.** Text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings via E.

Summary

1. **Grounding x_t.** Text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.

Summary

1. **Grounding x_t.** Text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.
3. **Vanishing/exploding gradients.** $\prod_t \mathrm{diag}(\tanh'(a_t))W_h$ decays or grows with $\rho(W_h)$.

Summary

1. **Grounding x_t.** Text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.
3. **Vanishing/exploding gradients.** $\prod_t \mathrm{diag}(\tanh'(a_t)) W_h$ decays or grows with $\rho(W_h)$.
4. **LSTM gating.** $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ creates a gradient highway: $\partial c_t / \partial c_{t-1} \approx I$.

Summary

1. **Grounding x_t.** Text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.
3. **Vanishing/exploding gradients.** $\prod_t \mathrm{diag}(\tanh'(a_t)) W_h$ decays or grows with $\rho(W_h)$.
4. **LSTM gating.** $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ creates a gradient highway: $\partial c_t / \partial c_{t-1} \approx I$.
5. **Information bottleneck.** Encoder–decoder compresses input into h_T.

Summary

1. **Grounding x_t.** Text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.
3. **Vanishing/exploding gradients.** $\prod_t \mathrm{diag}(\tanh'(a_t)) W_h$ decays or grows with $\rho(W_h)$.
4. **LSTM gating.** $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ creates a gradient highway: $\partial c_t / \partial c_{t-1} \approx I$.
5. **Information bottleneck.** Encoder–decoder compresses input into h_T.
6. **Attention.** $c_t = \sum_i (\alpha_t)_i v_i$ resolves the bottleneck. Abstraction: score $\rightarrow$ normalize $\rightarrow$ aggregate.

Summary

1. **Grounding x_t.** Text $\rightarrow$ tokens $\rightarrow$ indices $\rightarrow$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.
3. **Vanishing/exploding gradients.** $\prod_t \mathrm{diag}(\tanh'(a_t)) W_h$ decays or grows with $\rho(W_h)$.
4. **LSTM gating.** $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ creates a gradient highway: $\partial c_t / \partial c_{t-1} \approx I$.
5. **Information bottleneck.** Encoder–decoder compresses input into h_T.
6. **Attention.** $c_t = \sum_i (\alpha_t)_i v_i$ resolves the bottleneck. Abstraction: score $\rightarrow$ normalize $\rightarrow$ aggregate.
7. **QKV.** $\mathrm{Attention}(Q, K, V) = V\,\mathrm{softmax}(K^\top Q / \sqrt{d_k})$. Separates matching from aggregation.

Summary

1. **Grounding x_t.** Text $\to$ tokens $\to$ indices $\to$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.
3. **Vanishing/exploding gradients.** $\prod_t \operatorname{diag}(\tanh'(a_t)) W_h$ decays or grows with $\rho(W_h)$.
4. **LSTM gating.** $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ creates a gradient highway: $\partial c_t / \partial c_{t-1} \approx I$.
5. **Information bottleneck.** Encoder–decoder compresses input into h_T.
6. **Attention.** $c_t = \sum_i (\alpha_t)_i v_i$ resolves the bottleneck. Abstraction: score $\to$ normalize $\to$ aggregate.
7. **QKV.** $\operatorname{Attention}(Q, K, V) = V \operatorname{softmax}(K^\top Q / \sqrt{d_k})$. Separates matching from aggregation.
8. **Tokenization.** Word/char/subword tradeoffs. **BPE**: greedily merge most-frequent byte pairs; byte-level base eliminates $\langle \texttt{UNK} \rangle$; implicitly learns morphology.

Summary

1. **Grounding x_t.** Text $\to$ tokens $\to$ indices $\to$ embeddings via E.
2. **RNNs as dynamical systems.** $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$. Weight sharing; BPTT.
3. **Vanishing/exploding gradients.** $\prod_t \operatorname{diag}(\tanh'(a_t)) W_h$ decays or grows with $\rho(W_h)$.
4. **LSTM gating.** $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ creates a gradient highway: $\partial c_t / \partial c_{t-1} \approx I$.
5. **Information bottleneck.** Encoder–decoder compresses input into h_T.
6. **Attention.** $c_t = \sum_i (\alpha_t)_i v_i$ resolves the bottleneck. Abstraction: score $\to$ normalize $\to$ aggregate.
7. **QKV.** $\operatorname{Attention}(Q, K, V) = V \operatorname{softmax}(K^\top Q / \sqrt{d_k})$. Separates matching from aggregation.
8. **Tokenization.** Word/char/subword tradeoffs. **BPE**: greedily merge most-frequent byte pairs; byte-level base eliminates $\langle \texttt{UNK} \rangle$; implicitly learns morphology.
9. **Open:** Attention does not fix sequential computation or long path lengths $\to$ next lecture.

Appendices

- Appendix A: LSTM Cell — Full Equations
- Appendix B: BPTT Gradient Recurrence
- Appendix C: Teacher Forcing
- Appendix D: LSTM Cell-State Jacobian — Full Derivation

Appendix A: LSTM Cell — Full Equations

$$i_t = \sigma(W_{h,i}\, h_{t-1} + W_{x,i}\, x_t + b_i) \qquad \textbf{(input gate)}$$

$$f_t = \sigma(W_{h,f}\, h_{t-1} + W_{x,f}\, x_t + b_f) \qquad \textbf{(forget gate)}$$

$$\tilde{c}_t = \tanh(W_{h,c}\, h_{t-1} + W_{x,c}\, x_t + b_c) \qquad \textbf{(candidate cell)}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \qquad \textbf{(cell state update)}$$

$$o_t = \sigma(W_{h,o}\, h_{t-1} + W_{x,o}\, x_t + b_o) \qquad \textbf{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t) \qquad \textbf{(hidden state)}$$

All gate weight matrices: $W_{h,\cdot} \in \mathbb{R}^{d_h \times d_h}$, $W_{x,\cdot} \in \mathbb{R}^{d_h \times d_x}$. All biases: $b_\cdot \in \mathbb{R}^{d_h}$.

Total learnable parameters per LSTM cell: $4(d_h^2 + d_h \cdot d_x + d_h)$.

Appendix B: BPTT Gradient Recurrence

Define $d_t = \frac{dh_t}{d\theta_h}$, $b_t = \frac{\partial f}{\partial \theta_h}(x_t, h_{t-1}, \theta_h)$, $e_t = \frac{\partial f}{\partial h_{t-1}}(x_t, h_{t-1}, \theta_h)$.

By the chain rule: $d_t = b_t + e_t\, d_{t-1}$, with $d_0 = 0$.

Closed form (by induction):

$$d_t = b_t + \sum_{\Delta=1}^{t-1} \left(\prod_{j=\Delta+1}^{t} e_j \right) b_\Delta$$

Plugging into the loss gradient:

$$\frac{d\mathcal{L}}{d\theta_h} = \frac{1}{T} \sum_{t=1}^{T} \underbrace{\frac{\partial \ell}{\partial \hat{y}_t} \cdot \frac{\partial g}{\partial h_t}}_{\text{local}} \cdot \underbrace{d_t}_{\text{recursive}}$$

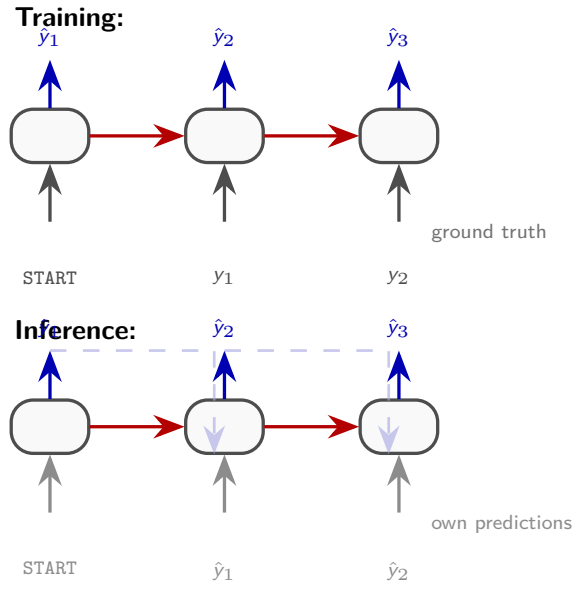
The product $\prod_{j=\Delta+1}^{t} e_j$ is the Jacobian product from the vanishing gradient discussion.

Truncated BPTT at horizon τ: approximate d_t by summing only the last τ terms. Benefits: cheaper ($O(\tau)$ vs. $O(t)$); mitigates vanishing/exploding gradients; acts as regularization.

Appendix C: Teacher Forcing

During training, the decoder receives the **ground-truth** y_{t-1} as input at step t, rather than its own prediction $\hat{y}_{t-1}$.

- **Why:** Incorrect predictions compound across the sequence (**exposure bias**).
- **Benefit:** Breaks the error chain; stable gradients.
- **Tradeoff:** Train/test mismatch—at inference the decoder uses its own (imperfect) outputs.
- **Mitigation:** **Scheduled sampling** (introduced in Bengio et al., 2015).



Appendix D: LSTM Cell-State Jacobian — Full Derivation

Recall: $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$, where each gate depends on h_{t-1} and x_t, and $h_{t-1} = o_{t-1} \odot \tanh(c_{t-1})$.

The full Jacobian by the product rule:

$$\frac{\partial c_t}{\partial c_{t-1}} = \underbrace{\operatorname{diag}(f_t)}_{\text{direct}} + \underbrace{\operatorname{diag}(c_{t-1})\frac{\partial f_t}{\partial c_{t-1}} + \operatorname{diag}(\tilde{c}_t)\frac{\partial i_t}{\partial c_{t-1}} + \operatorname{diag}(i_t)\frac{\partial \tilde{c}_t}{\partial c_{t-1}}}_{\text{indirect paths through } h_{t-1}=o_{t-1}\odot\tanh(c_{t-1})}$$

Appendix D: LSTM Cell-State Jacobian — Full Derivation

Recall: $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$, where each gate depends on h_{t-1} and x_t, and $h_{t-1} = o_{t-1} \odot \tanh(c_{t-1})$.

The full Jacobian by the product rule:

$$\frac{\partial c_t}{\partial c_{t-1}} = \underbrace{\operatorname{diag}(f_t)}_{\text{direct}} + \underbrace{\operatorname{diag}(c_{t-1})\frac{\partial f_t}{\partial c_{t-1}} + \operatorname{diag}(\tilde{c}_t)\frac{\partial i_t}{\partial c_{t-1}} + \operatorname{diag}(i_t)\frac{\partial \tilde{c}_t}{\partial c_{t-1}}}_{\text{indirect paths through } h_{t-1}=o_{t-1}\odot\tanh(c_{t-1})}$$

Each indirect term factors through $\frac{\partial h_{t-1}}{\partial c_{t-1}} = \operatorname{diag}\big(o_{t-1} \odot (1 - \tanh^2(c_{t-1}))\big)$, then through the gate's dependence on h_{t-1}.

Appendix D: LSTM Cell-State Jacobian — Full Derivation

Recall: $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$, where each gate depends on h_{t-1} and x_t, and $h_{t-1} = o_{t-1} \odot \tanh(c_{t-1})$.

The full Jacobian by the product rule:

$$\frac{\partial c_t}{\partial c_{t-1}} = \underbrace{\operatorname{diag}(f_t)}_{\text{direct}} + \underbrace{\operatorname{diag}(c_{t-1})\frac{\partial f_t}{\partial c_{t-1}} + \operatorname{diag}(\tilde{c}_t)\frac{\partial i_t}{\partial c_{t-1}} + \operatorname{diag}(i_t)\frac{\partial \tilde{c}_t}{\partial c_{t-1}}}_{\text{indirect paths through } h_{t-1}=o_{t-1}\odot\tanh(c_{t-1})}$$

Each indirect term factors through $\frac{\partial h_{t-1}}{\partial c_{t-1}} = \operatorname{diag}\big(o_{t-1} \odot (1 - \tanh^2(c_{t-1}))\big)$, then through the gate's dependence on h_{t-1}.

Why the indirect terms are small:

1. $\tanh'(c_{t-1}) \in (0, 1]$ and $o_{t-1} \in (0, 1)^{d_h}$, so $\|\partial h_{t-1}/\partial c_{t-1}\| \leq 1$.
2. Each gate Jacobian $\partial f_t/\partial h_{t-1}$ involves $\sigma'(\cdot) \leq 1/4$, providing further contraction.
3. The product makes each indirect term $O(\frac{1}{4}\|W_{h,\cdot}\|)$—a small perturbation of $\operatorname{diag}(f_t)$.

Appendix D: LSTM Cell-State Jacobian — Full Derivation

Recall: $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$, where each gate depends on h_{t-1} and x_t, and $h_{t-1} = o_{t-1} \odot \tanh(c_{t-1})$.

The full Jacobian by the product rule:

$$\frac{\partial c_t}{\partial c_{t-1}} = \underbrace{\operatorname{diag}(f_t)}_{\text{direct}} + \underbrace{\operatorname{diag}(c_{t-1})\frac{\partial f_t}{\partial c_{t-1}} + \operatorname{diag}(\tilde{c}_t)\frac{\partial i_t}{\partial c_{t-1}} + \operatorname{diag}(i_t)\frac{\partial \tilde{c}_t}{\partial c_{t-1}}}_{\text{indirect paths through } h_{t-1}=o_{t-1}\odot\tanh(c_{t-1})}$$

Each indirect term factors through $\frac{\partial h_{t-1}}{\partial c_{t-1}} = \operatorname{diag}\big(o_{t-1} \odot (1 - \tanh^2(c_{t-1}))\big)$, then through the gate's dependence on h_{t-1}.

Why the indirect terms are small:

1. $\tanh'(c_{t-1}) \in (0, 1]$ and $o_{t-1} \in (0, 1)^{d_h}$, so $\|\partial h_{t-1}/\partial c_{t-1}\| \leq 1$.
2. Each gate Jacobian $\partial f_t/\partial h_{t-1}$ involves $\sigma'(\cdot) \leq 1/4$, providing further contraction.
3. The product makes each indirect term $O(\frac{1}{4}\|W_{h,\cdot}\|)$—a small perturbation of $\operatorname{diag}(f_t)$.

When $f_t \approx \mathbf{1}$: $\frac{\partial c_t}{\partial c_{t-1}} \approx I + \epsilon$ with $\|\epsilon\| \ll 1$. Over $T - t$ steps, the product stays near I—the gradient highway property.