Time-Series & Sequence Modeling with RNNs

Advanced Topics in Machine Learning for Bioinformatics and Biomedical Engineering

Joana Gelabert Dr. Alexandre Perera Lluna

2026-03-14

Introduction: Sequence Models for Bioinformatics

Why sequence models?

Most biologically meaningful information is carried in ordered sequences, not in flat feature vectors.

Learning goals

1. Recognise the sequential structure of core biological data types.
2. State the modelling requirements that arise from variable-length, ordered input.
3. Identify the two fundamental limitations of MLPs on sequence data.
4. Explain why those limitations motivate dedicated sequence architectures.

Why sequence models?

Biological sequences are everywhere:

- DNA: `ATCGATCGTA...`
- Proteins: `MKTAYIAKQR...`
- ECG: $x_1, x_2, \ldots, x_T$
- EEG signals over time
- Gene expression time-courses

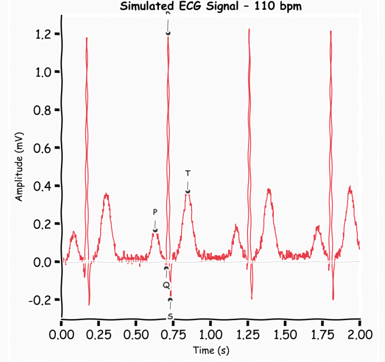


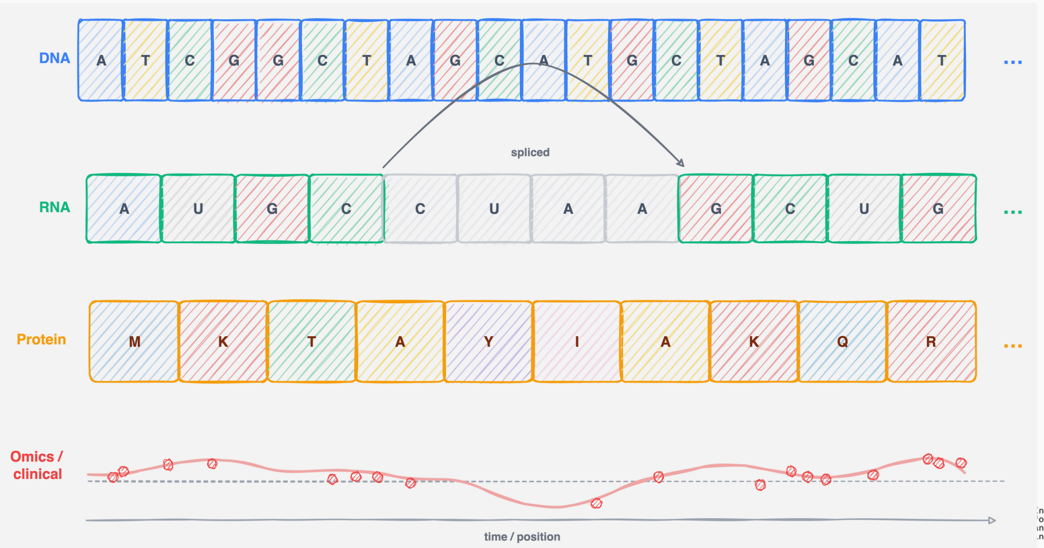
Figure 1: Simulated ECG signal at 110 bpm, with labelled P, Q, R, S, T waves.

The biological world is sequential

Biological function is encoded in the **order** of discrete or continuous measurements taken over time or position.

- **Genome:** linear string of nucleotides; regulatory elements depend on precise arrangement
- **Transcriptome:** spliced mRNA sequences; reading frame determines the amino acid coded at every position
- **Proteome:** amino acid chain whose fold is determined by residue sequence
- **Omics time series:** gene expression or metabolite levels measured at successive time points
- **Clinical signals:** ECG traces, longitudinal lab values, patient event histories
- **Clinical text:** Electronic free records, written reports from medical image

Sequential biological data: four modalities



Institute
for Research
and Innovation
in Health

DNA and RNA: ordered nucleotide alphabets

A DNA or RNA sequence is a string over a small finite alphabet:

- **DNA:** $\{A, T, C, G\}$, **RNA (mRNA):** $\{A, U, C, G\}$

Each position t carries a symbol $x_t \in \mathcal{V}$, typically encoded as a one-hot vector $x_t \in \{0, 1\}^{|\mathcal{V}|}$.

Key properties that require order-awareness:

- **Codons** are non-overlapping triplets; reading frame depends on start position
- **Splice sites** are defined by short consensus sequences at precise locations
- **Promoters and enhancers** act through specific motifs embedded in context
- **Distance** between elements (e.g., TATA box to transcription start) is functionally relevant
- **General annotations** on the genome, such as the (ENCODE Project Consortium 2012).

B2S LAB IRIS Institute for Research and Innovation in Health

Proteins: function encoded in residue sequence

A protein is a sequence of amino acids over an alphabet of 20 standard residues, $x_t \in \mathcal{V}_{\text{aa}}$ with $|\mathcal{V}_{\text{aa}}| = 20$.

- **Primary structure** (the sequence) determines all higher-order structure under physiological conditions (Anfinsen 1973)
- **Domains** are compact functional units; their boundaries are sequence-defined
- **Active sites** depend on residues that may be far apart in the sequence but close in 3D space – a long-range dependency problem
- **Sequence homology** is used to infer function across species
- **Secondary Structure** Structure prediction from sequence alone reached near-experimental accuracy, (Jumper et al. 2021)

Downstream tasks include contact map prediction, variant effect scoring, and subcellular localisation.

B2S LAB IRIS Institute for Research and Innovation in Health

Omics time series and clinical signals

Not all sequential data in bioinformatics is discrete or genomic.

- **Bulk RNA-seq time courses:** gene expression measured at T time points; $x_t \in \mathbb{R}^{d_{\text{genes}}}$, often $d > 10{,}000$
- **Single-cell trajectory inference:** ordered pseudo-time series of cell states along a differentiation path
- **Metabolomics and proteomics time series:** continuous concentration profiles during a perturbation or disease progression
- **EHR longitudinal data:** irregularly sampled lab values, diagnoses, and drug events for each patient; T varies across patients
- **Electrophysiology:** continuous voltage signals (EEG, local field potential) sampled at kHz rates

Common challenges: irregular sampling, missing values, very long T, and patient-level variability in length.

B2S LAB IRIS Institute for Research and Innovation in Health

What defines a sequence modelling problem?

Three properties jointly distinguish sequence problems from standard supervised learning:

Order matters: The output changes if the input is permuted. $f(x_1, x_2) \neq f(x_2, x_1)$ in general. A bag-of-words or feature-vector model ignores this.

Length is variable: Different instances have different T. A single fixed architecture must handle all lengths without retraining.

Long-range dependencies: The label at position t can depend on context far back (or ahead) in the sequence – dependencies may span hundreds or thousands of positions.

These three properties are not independent: handling variable length and long range simultaneously is the central challenge that motivates architecture.

The MLP as a baseline

Recall the standard MLP: a composition of affine maps and nonlinearities.

For a single input $\mathbf{x} \in \mathbb{R}^d$ and layer ℓ out of L layers:

$$\mathbf{h}^{(0)} = \mathbf{x}, \qquad \mathbf{h}^{(\ell)} = \phi\Big(W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\Big), \qquad \hat{y} = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$$

The MLP is a universal approximator (Hornik, Stinchcombe, and White 1989) and the workhorse of most non-sequential tasks.

Where MLPs work well in bioinformatics:

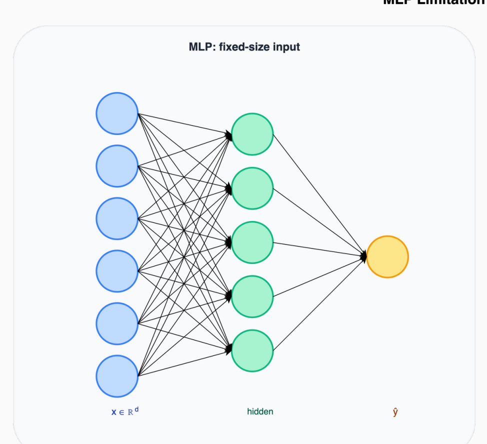
- Fixed-size tabular inputs (gene expression matrices, k-mer count vectors)
- Classification of fixed-length feature representations
- Final prediction head on top of a learned representation

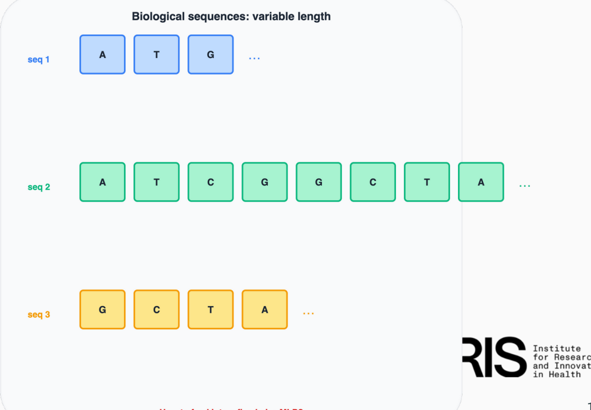
MLP limitation 1: fixed input size

Fixed Input Size

An MLP with input layer of width d requires every input to have **exactly** d **features**. There is no mechanism to handle variable-length sequences natively.

MLP Limitation 1: Fixed Input Size





RIS Institute for Research and Innovation in Health

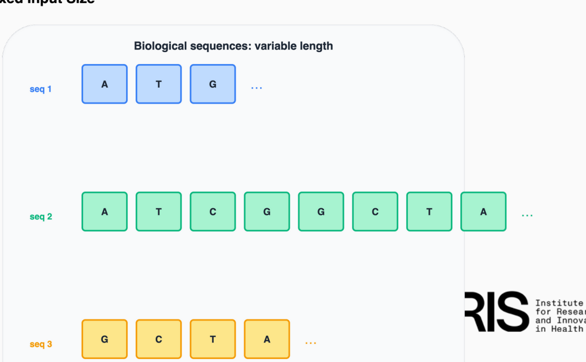
Fixed Input Size

The two engineering workarounds both lose information:

- **Truncation:** discard positions beyond a fixed cutoff – relevant context is dropped
- **Padding + flattening:** pad short sequences to length T_{max} – the model must waste capacity learning to ignore padding

MLP Limitation 1: Fixed Input Size

MLP: fixed-size input



MLP limitation 2: no memory across positions

Even if we fix the length problem –(e.g., by applying an MLP independently at each position)– the MLP has **no way to pass information from one position to the next**.

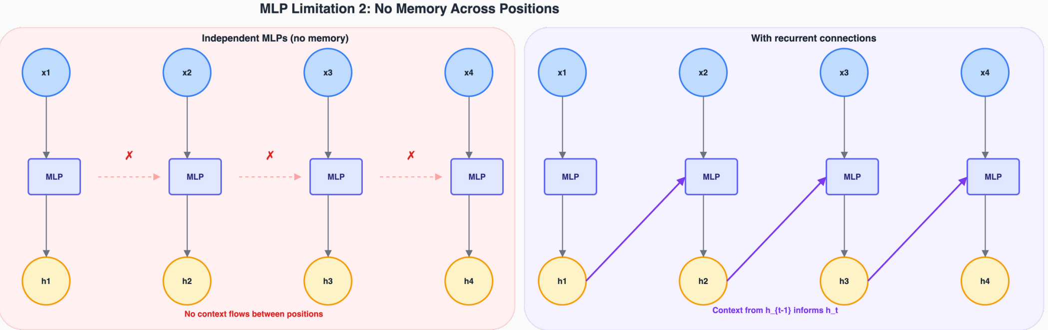


Figure 5: Left: independent MLPs applied at each position share no information (×). Right: recurrent connections allow context from h_{t-1} to influence h_t (purple arrows).

B2S LAB IRIS Institute for Research and Innovation in Health

MLP limitation 2: consequences for biology

For example, an MLP applied position-by-position cannot:

- **Detect a motif that spans multiple positions** without seeing all positions simultaneously (defeats variable-length handling)
- **Track codon reading frame** – the interpretation of position t depends on $t \bmod 3$, which requires knowing where the sequence started
- **Model splice site context** – both the donor and acceptor site must be jointly considered, though they are tens to thousands of nucleotides apart
- **Capture promoter–TSS distance** – a regulatory element's effect depends on its distance from the transcription start, not on its local identity alone

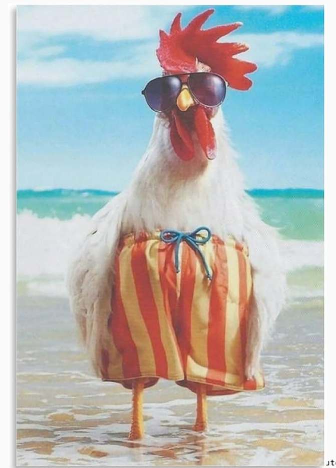


Figure 6: Houston, we have

and Innovation in Health

What a sequence model must provide

A model suited to sequential biological data needs at least four properties:

Property	Requirement
Variable-length input	Same parameters for any T
Position awareness	Representations respect order
Context propagation	Information at t influences representations at $t' > t$
Parameter efficiency	Parameters do not grow with T

Models we will study in this course, in order of increasing capacity:

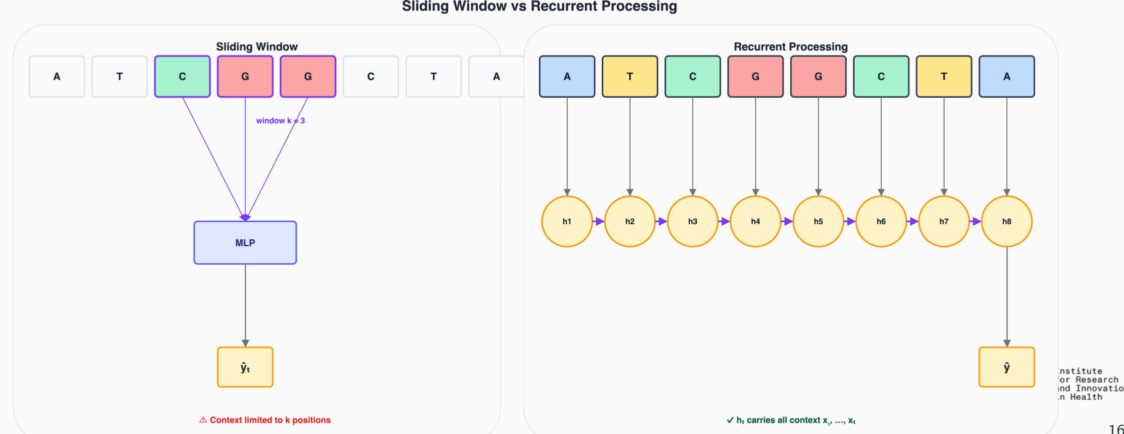
- **Recurrent Neural Networks (RNN):** sequential hidden state; $h_t = f(x_t, h_{t-1})$
- **Long Short-Term Memory (LSTM):** gated memory cell; selectively retains long-range context
- **Convolutional sequence models:** parallel local context; used in motif detection
- **Transformers and attention:** all-pairs context; the basis of protein language

B2S LAB IRIS Institute for Research and Innovation in Health

Sequence Modelling Basics

The sliding window

Before recurrent networks, the standard approach was a **sliding window**: extract a fixed-width context of k positions around each target position and apply an MLP to that window.



Institute for Research and Innovation in Health

Sliding window: properties and limits

For a window of size k centred at position t, the input to the MLP is:

$$\mathbf{v}_t = \left[x_{t-\lfloor k/2 \rfloor}, \; \ldots, \; x_t, \; \ldots, \; x_{t+\lfloor k/2 \rfloor}\right] \in \mathbb{R}^{k \cdot d_{\text{in}}}$$

Strengths:

- Simple to implement; fully parallelisable across positions
- Captures local motifs (e.g., TATA box, splice consensus) efficiently
- Still used in convolutional layers as a learned multi-scale window

Limitations:

- **Hard context horizon:** positions more than $k/2$ steps away are invisible
- **Boundary artefacts:** positions near the start or end of the sequence need padding
- **No global state:** the model cannot detect that a motif seen at position 20 is relevant to the label at position 200
- **Width is a hyperparameter:** too small misses context; too large inflates input

Sequence modelling

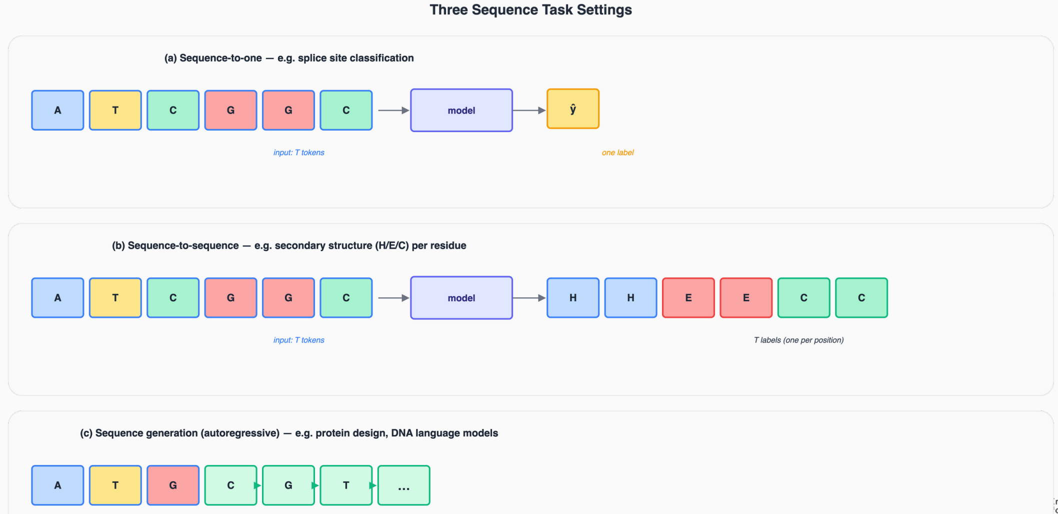
- Sequence models differ in what they predict and when.
- The choice of **task setting** determines
  - the loss,
  - the output layer, and
  - how we read off predictions at inference time.

Sequence mapping

one to one | one to many | many to one | many to many | many to many

Figure 8: Generally, we will find these sequence mapping variants

Sequence task settings



Sequence-to-one

A single label is produced for the **entire input sequence** of length T.

$$f_\theta : \mathbb{R}^{T \times d_{\text{in}}} \longrightarrow \mathbb{R}^k$$

The model reads all T positions and emits one prediction vector.

Some examples:

- Promoter / enhancer classification from a fixed-length DNA window
- Splice site prediction: does this sequence contain a donor or acceptor?
- Protein family classification from the full amino acid sequence
- Virulence or antibiotic resistance prediction from a gene sequence
- Patient-level outcome prediction from a longitudinal EHR record

Sequence-to-sequence

A label is produced at **every position** in the input sequence.

$$f_\theta : \mathbb{R}^{T \times d_{\text{in}}} \longrightarrow \mathbb{R}^{T \times k}$$

The output has the same length T as the input; labels are aligned position by position.

Some examples:

- Secondary structure prediction: helix / sheet / coil label per residue
- Intrinsic disorder prediction: probability of disorder at each residue
- CpG island annotation: methylation state per cytosine position
- Named entity recognition in clinical text: drug / disease / dosage per token
- Base-calling from raw nanopore signals: nucleotide label per signal sample

Sequence generation (autoregressive)

The model **generates** one token at a time, conditioning on all previously generated tokens.

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1})$$

At each step the model samples the next token, then feeds it back as input.

Some examples:

- Protein sequence design: generate amino acid sequences compatible with a target fold or function (Madani et al. 2023)
- DNA language models: predict the next nucleotide given genomic context
- scRNA-seq data augmentation: generate plausible synthetic expression profiles

This is the same principle used in large language models. The only difference in biology is that the our vocabulary is small ($|\mathcal{V}| = 4$ or 20) but the sequences can be very long.

B2S IRIS Institute for Research and Innovation in Health

Recurrent processing: the key idea

A **recurrent model** maintains a **hidden state** h_t that is updated at every step:

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

- $h_t \in \mathbb{R}^{d_h}$ is a fixed-size summary of **all past input** $x_1, \ldots, x_t$, $h_0 \in \mathbb{R}^{d_h}$ is an initial state (typically $\mathbf{0}$)
- f is the same function applied at every step (defined by learned parameters) .

Recurrent processing: the key idea

A **recurrent model** maintains a **hidden state** h_t that is updated at every step:

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

Three consequences that directly address the MLP limitations:

- **Unbounded context:** h_t can in principle depend on x_1, however far back
- **Variable-length input:** the same recurrence runs for any T
- **Fixed parameter count:** the size of the model does not grow with T

The gradient signal must propagate through many time steps to update weights that influenced early states, –> **vanishing gradient** problem – the main motivation for LSTMs and GRUs.

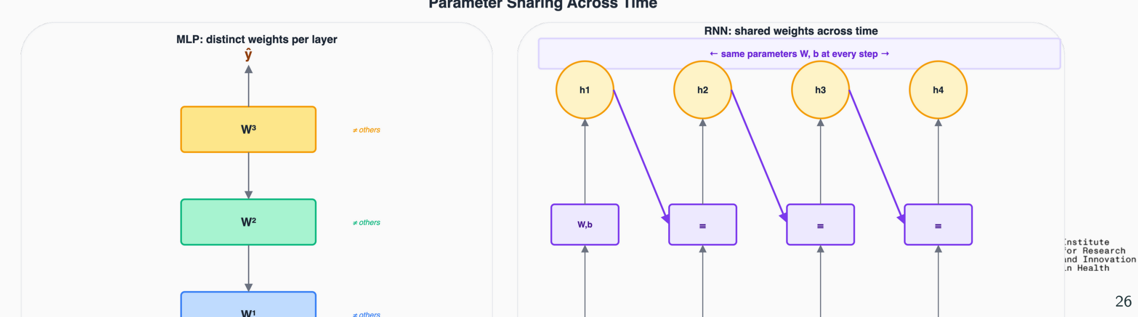
B2S LAB IRIS Institute for Research and Innovation in Health

Parameter sharing across time

The function f uses the **same parameters at every time step**. For the simplest recurrent model:

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

where $W_x \in \mathbb{R}^{d_h \times d_{\text{in}}}$, $W_h \in \mathbb{R}^{d_h \times d_h}$, and $b \in \mathbb{R}^{d_h}$ are **shared across all T steps**.



Sharing parameters across time has two important consequences.

Generalisation to unseen lengths

The model is trained on sequences of length up to $T_{\max}$, but the same recurrence $h_t = f(x_t, h_{t-1})$ can be unrolled for any T at test time. No architecture change or retraining is required.

Regularisation by structure

Using the same W_x, W_h, b everywhere forces the model to learn a universal rule for updating context – not a position-specific rule. This reduces the effective number of parameters from $O(T \cdot d_h^2)$ to $O(d_h^2)$.

Note

Parameter sharing is the same principle as weight tying in convolutional networks: a filter is applied at every spatial position. In RNNs the axis of sharing is time rather than space.

Recurrent neural networks

The RNN unit

At each time step t, the unit receives two inputs:

- $x_t \in \mathbb{R}^{d_{\text{in}}}$ — the current observation
- $h_{t-1} \in \mathbb{R}^{d_h}$ — the hidden state carried from the previous step

It produces a new hidden state:

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

- $W_h \in \mathbb{R}^{d_h \times d_h}$ mixes the recurrent memory
- $W_x \in \mathbb{R}^{d_h \times d_{\text{in}}}$ projects the current input
- tanh keeps values in $(-1, 1)$, preventing unbounded growth
- h_t is both the **output** at step t and the **input** to step $t+1$

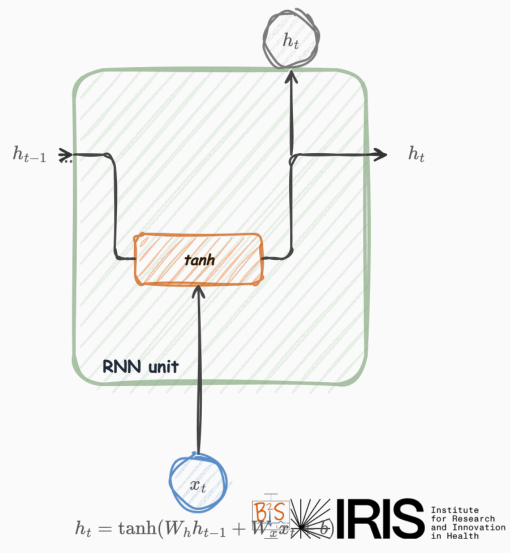


Figure 11: RNN unit: h_{t-1} and x_t enter

The RNN unit

h_{t-1} h_t h_{t+1}

h_{t-2} h_{t-1} h_t h_{t+1}

tanh tanh tanh

RNN unit RNN unit RNN unit

x_{t-1} x_t x_{t+1}

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

Figure 12: Unfolded

The RNN unit animated

link

The RNN unit animated

link

Hidden state as a lossy summary

The hidden state h_t compresses the entire history $(x_1, x_2, \ldots, x_t)$ into a fixed-size vector.

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

- At $t = 1$: h_1 depends only on x_1 (and the initial state h_0, often zero)
- At $t = T$: h_T must encode everything relevant from T steps

Note

The representation is **lossy**: h_t has d_h entries regardless of how long the sequence is. Early inputs can only influence the output if their effect survives repeated multiplication and squashing through tanh.

Output at each step

The hidden state h_t can be read out at every step or only at the end, depending on the task.

Sequence-to-sequence (e.g., per-residue secondary structure prediction):

$$y_t = W_y\, h_t + b_y, \qquad t = 1, \ldots, T$$

Sequence-to-one (e.g., classify a promoter sequence as active/inactive):

$$y = W_y\, h_T + b_y$$

- $W_y \in \mathbb{R}^{k \times d_h}$, $b_y \in \mathbb{R}^k$ are the output-layer parameters
- $k =$ number of output classes or targets
- The loss is computed on y_t (per step) or y (final), then backpropagated through the unrolled graph

Institute for Research and Innovation in Health

Parameters are shared across time

A key property: W_h, W_x, and b are **the same** at every step.

- The RNN applies the same function $f(h_{t-1}, x_t; \theta)$ at $t = 1, 2, \ldots, T$
- This means one set of parameters handles sequences of **any length**
- Gradients from every time step contribute to the same W_h, W_x, b

Tip

Weight sharing is what makes RNNs fundamentally different from a feed-forward network with T separate layers. It also makes gradient computation more involved, because changing W_h affects h_1, which affects h_2, and so on.

Backpropagation Through Time

From backprop to BPTT

In a feed-forward network, backprop walks backward through layers.

In an RNN, the computational graph is the **unrolled** chain $h_1 \to h_2 \to \cdots \to h_T$.

Backpropagation Through Time (BPTT) is simply backprop applied to this unrolled graph:

1. **Forward:** compute $h_1, h_2, \ldots, h_T$ and the loss L
2. **Backward:** propagate $\frac{\partial L}{\partial h_t}$ from $t = T$ back to $t = 1$
3. **Accumulate:** sum the gradient contributions from every step into $\frac{\partial L}{\partial W_h}, \frac{\partial L}{\partial W_x}, \frac{\partial L}{\partial b}$

Note

BPTT is not a new algorithm. It is standard backprop on a graph that happens to repeat the same operation T times with shared weights.

B2S LAB IRIS Institute for Research and Innovation in Health

The chain rule through time

Consider a loss that depends only on the final state: $L = \ell(h_T)$.

The gradient of L with respect to h_t (for $t < T$) requires chaining through every intermediate step:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_T} \prod_{s=t+1}^{T} \frac{\partial h_s}{\partial h_{s-1}}$$

Each Jacobian factor comes from the recurrence $h_s = \tanh(W_h\, h_{s-1} + W_x\, x_s + b)$:

$$\frac{\partial h_s}{\partial h_{s-1}} = \mathrm{diag}\big(\tanh'(z_s)\big)\; W_h$$

where $z_s = W_h\, h_{s-1} + W_x\, x_s + b$ and $\tanh'(z) = 1 - \tanh^2(z)$.

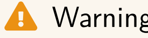
The gradient is a product of many matrices

Expanding the product:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_T} \prod_{s=t+1}^{T} \mathrm{diag}(1 - h_s^2)\ W_h$$

This is a product of $(T - t)$ matrices, all involving W_h.

- If $T - t$ is large, this product can **shrink to zero** or **blow up**
- The behaviour depends on the spectral properties of W_h and the saturation of tanh


Warning

This is the fundamental difficulty of training RNNs on long sequences: the gradient signal must survive many multiplicative steps to reach early time steps.

Gradient w.r.t. shared parameters

Since W_h appears at every step, the total gradient is a sum over time:

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \left. \frac{\partial h_t}{\partial W_h} \right|_{\text{local}}$$

where the local term is:

$$\left. \frac{\partial h_t}{\partial W_h} \right|_{\text{local}} = \mathrm{diag}(1 - h_t^2)\ h_{t-1}^{\top}$$

The same pattern applies to W_x and b. Each step contributes a term, and the difficulty lies in the factor $\frac{\partial L}{\partial h_t}$, which requires the long product from the previous slide.

Institute for Research and Innovation in Health

Vanishing and exploding gradients

Why gradients vanish

Recall the product that links h_T back to h_t:

$$\prod_{s=t+1}^{T} \mathrm{diag}(1 - h_s^2)\, W_h$$

Vanishing occurs when the spectral norm of each factor is less than 1:

- $\tanh'(z) \in (0, 1]$, and is close to 0 when $|z|$ is large (saturation)
- If the largest singular value of W_h is moderate, each multiplication **shrinks** the gradient
- After many steps the gradient reaching early time steps is negligibly small

Consequence: the network cannot learn long-range dependencies. Parameters are updated based almost entirely on the last few steps.

B2S LAB IRIS Institute for Research and Innovation in Health

Why gradients explode

Exploding occurs when the spectral norm of W_h is large enough that the product grows exponentially:

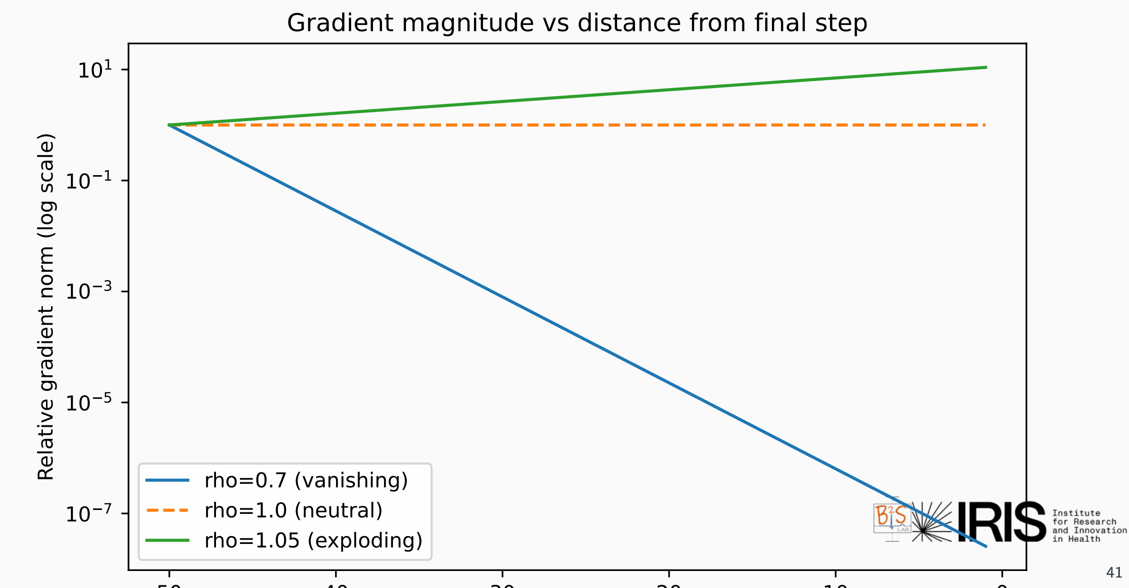
$$\left\| \prod_{s=t+1}^{T} \mathrm{diag}(1 - h_s^2)\, W_h \right\| \approx \rho^{T-t}$$

where ρ is an effective per-step amplification factor.

- Even $\rho = 1.1$ gives $1.1^{50} \approx 117$ after 50 steps
- Gradients become extremely large, causing huge parameter updates
- Training diverges: loss jumps to NaN or oscillates wildly

> **Note**
>
> Vanishing is **silent** (the model simply fails to learn). Exploding is **loud** (training crashes). Both stem from the same cause: a long chain of multiplications.

A numerical example



Vanishing gradients in bioinformatics

Long-range dependencies are common in biological sequences:

- **Protein folding:** residues distant in sequence can form contacts in 3D structure
- **Gene regulation:** enhancer elements can be thousands of base pairs from the promoter
- **Clinical time series:** a drug administered early may affect outcomes weeks later

If the gradient vanishes over 20–50 steps, the RNN treats these dependencies as invisible. The model learns only **local** patterns.

Tip

This limitation motivated the development of gated architectures (LSTM, GRU) and, later, attention-based models (Transformers), which provide alternative gradient pathways.

IRIS Institute for Research and Innovation in Health

Gradient clipping

Gradient clipping: taming explosions

Gradient clipping caps the norm of the gradient vector before each parameter update.

Norm clipping (most common): if $\|\nabla_\theta L\| > \tau$, rescale:

$$\nabla_\theta L \leftarrow \frac{\tau}{\|\nabla_\theta L\|} \nabla_\theta L$$

- The **direction** of the gradient is preserved
- Only its **magnitude** is reduced when it exceeds the threshold τ
- Typical values: $\tau \in [1, 5]$

Note

Clipping does not solve vanishing gradients. It only prevents the training process from crashing when gradients explode.

B2S LAB IRIS Institute for Research and Innovation in Health

Effect of clipping on the gradient landscape

Gradient clipping (norm clipping)

After clipping
Before clipping (exceeded)
Threshold tau=1.0

$\partial L / \partial \theta_2$

3
2
1
0
−1
−2
−3

Clipping in practice

In PyTorch, gradient clipping is a single line before the optimizer step:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)
optimizer.step()
```

Practical guidelines:

- Monitor the gradient norm during training; plot it alongside the loss
- If norms spike by orders of magnitude, clipping is necessary
- Start with $\tau = 5.0$; lower to 1.0 if instability persists
- Clipping is **standard practice** for any RNN-based model

Warning

If you must clip aggressively ($\tau < 0.5$), the model is likely ill-conditioned. Consider reducing the learning rate, changing the architecture (e.g., LSTM/GRU), or shortening the sequences via truncated BPTT.

Institute for Research and Innovation in Health

Truncated BPTT

Full BPTT unrolls all T steps, which can be expensive in memory and computation.

Truncated BPTT limits the backward pass to the last K steps:

- Forward: process all T steps normally, carrying h_t forward
- Backward: compute gradients only through $t = T, T-1, \ldots, T-K+1$
- Gradients for dependencies longer than K steps are **discarded**

Trade-off:

- $K = T$: full BPTT, exact gradients, expensive
- $K \ll T$: cheap, but long-range dependencies are invisible to the optimizer
- Typical choice: $K = 20$–200, depending on available memory and sequence length

ℹ Note

Truncated BPTT does not prevent vanishing gradients within the K-step window.
It simply limits how far back the gradient is computed.

Summary: vanilla RNN strengths and limitations

Property	Vanilla RNN
Parameter sharing	Same W_h, W_x, b at every step
Handles variable-length input	Yes, by construction
Long-range dependencies	Poor — gradients vanish exponentially
Training stability	Requires gradient clipping
Typical effective memory	10–20 steps in practice

Secondary Structure Demo

link

Long Short-Term Memory

Motivation: why gates?

The vanilla RNN updates its state by **overwriting** it at every step:

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

- The entire hidden state is recomputed through a squashing nonlinearity
- Information from early steps must survive repeated multiplication by W_h and compression through tanh
- After 20–50 steps, early signals are effectively erased

The core idea

Instead of forcing all information through a single multiplicative bottleneck, introduce a **separate memory channel** with **additive** updates and **learnable gates** that control what to remember, what to forget, and what to output.

Two state vectors

An LSTM cell maintains **two** vectors at each time step:

- **Cell state** $C_t \in \mathbb{R}^{d_h}$ — the long-term memory. Updated mostly by **addition**, so gradients flow without repeated matrix multiplication.
- **Hidden state** $h_t \in \mathbb{R}^{d_h}$ — the short-term output. A filtered, squashed view of C_t, exposed to the rest of the network.

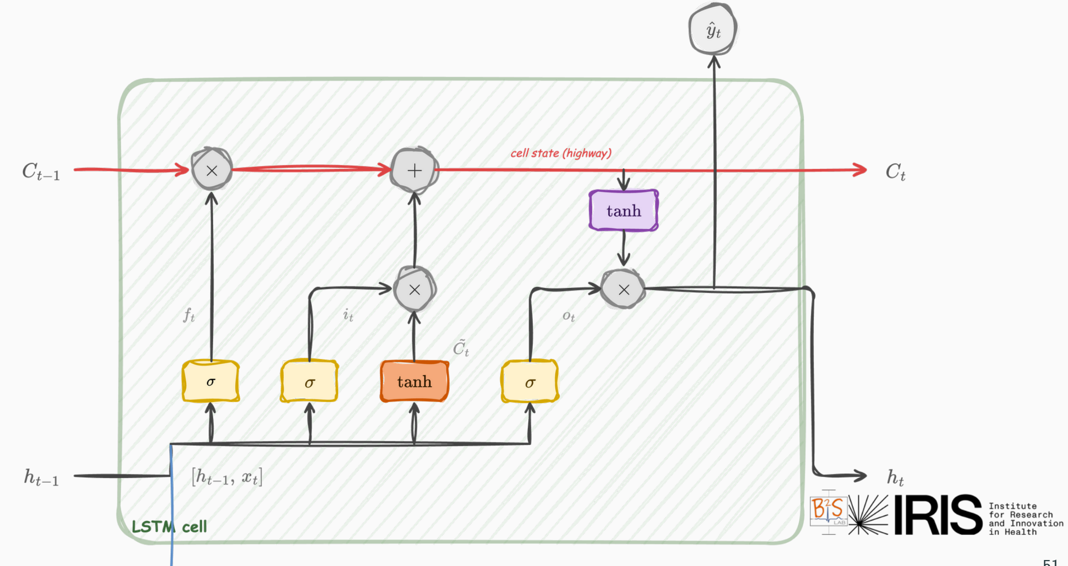
$$\begin{aligned} C_t &\longrightarrow \text{carries information across many steps (highway)} \\ h_t &\longrightarrow \text{used for predictions and passed to the next step} \end{aligned}$$

Note

The cell state C_t is what distinguishes an LSTM from a vanilla RNN. It acts as a **conveyor belt**: information can travel along it with minimal interference, unless a gate explicitly modifies it.

IRIS Institute for Research and Innovation in Health

The LSTM cell at a glance



Four gates, one cell

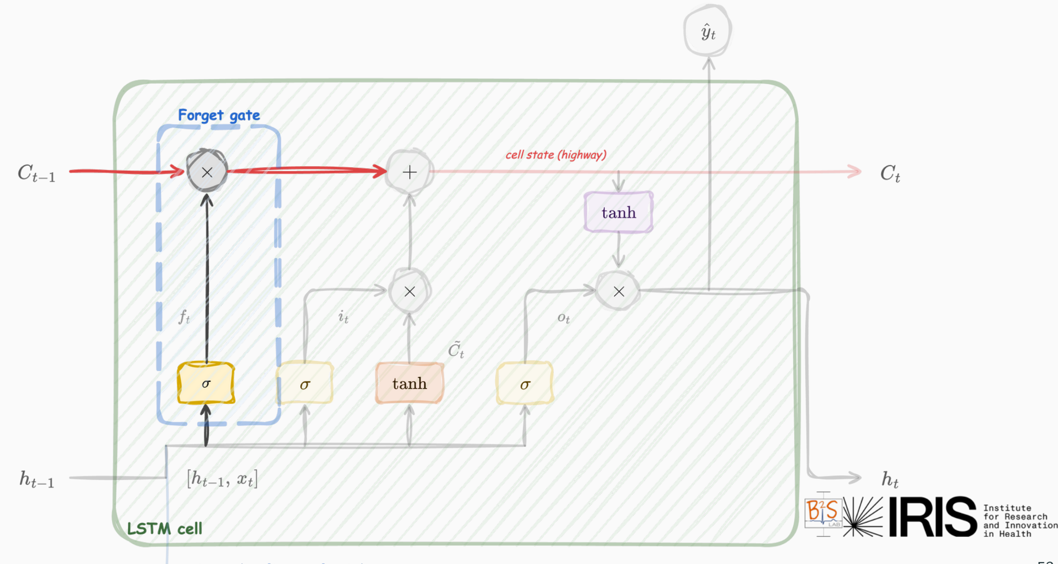
The LSTM cell computes four intermediate quantities at each step, all from a linear combination of h_{t-1} and x_t:

Gate	Symbol	Activation	Role
Forget	f_t	sigmoid	How much of C_{t-1} to **keep**
Input	i_t	sigmoid	How much of the new candidate to **write**
Candidate	$\tilde{C}_t$	tanh	What new information to propose
Output	o_t	sigmoid	What part of C_t to **expose** as h_t

- Sigmoid gates produce values in $(0, 1)$: they act as **soft switches**
- The candidate uses tanh to produce values in $(-1, 1)$: it proposes **content**, not control

B2S LAB IRIS Institute for Research and Innovation in Health

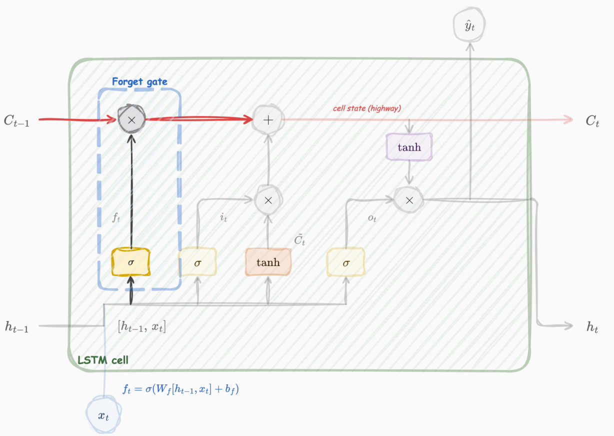
Forget gate



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

Forget gate

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



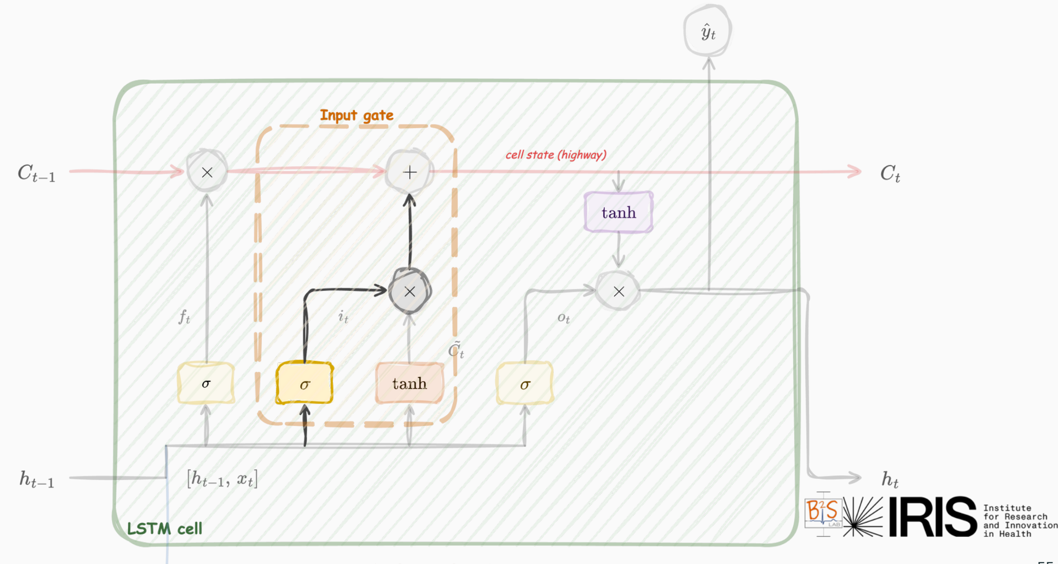
Dimensions:

- $[h_{t-1}, x_t] \in \mathbb{R}^{d_h + d_{\text{in}}}$
- $W_f \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$, $b_f \in \mathbb{R}^{d_h}$
- $f_t \in (0,1)^{d_h}$: one val. per dim of C_{t-1}

Per-dimension behaviour:

- $f_t^{(j)} \approx 1$: dim j **preserved**
- $f_t^{(j)} \approx 0$: dim j **erased**

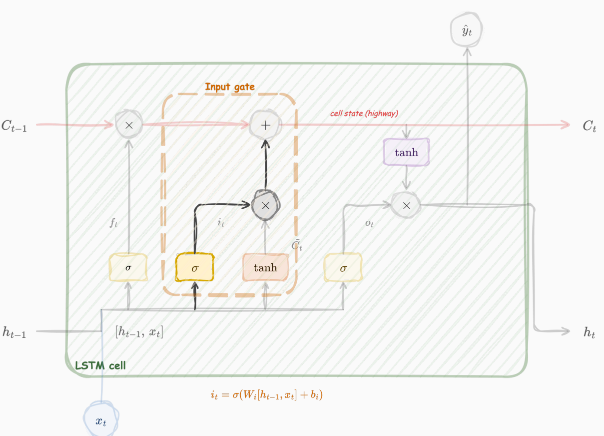
Input gate



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

Input gate

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



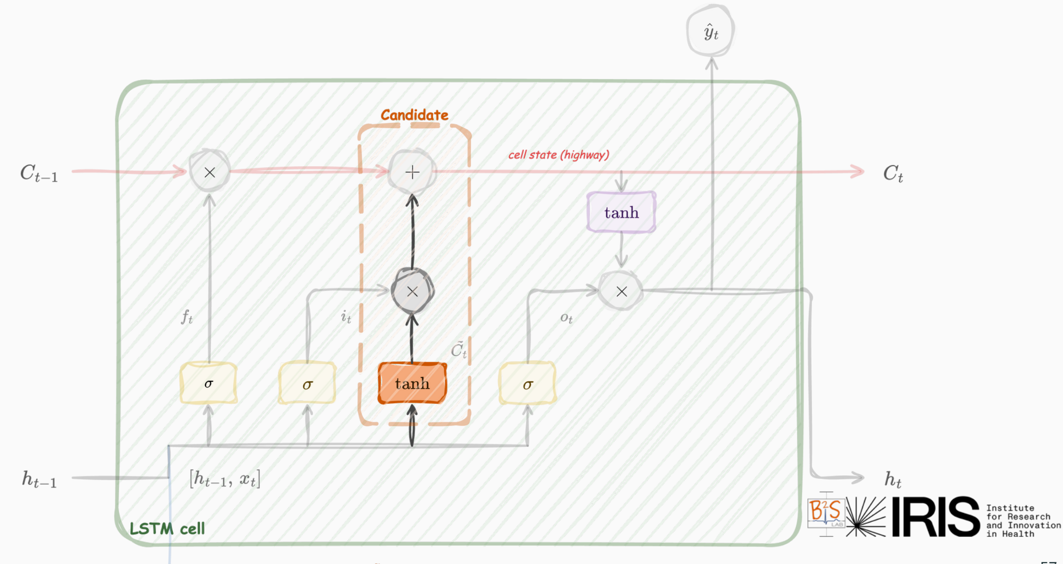
Dimensions:

- $W_i \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$, $b_i \in \mathbb{R}^{d_h}$
- $i_t \in (0, 1)^{d_h}$: per-dimension write strength

Separation of roles:

- **Input gate** decides *how much* to write (control)
- **Candidate** decides *what* to write (content)

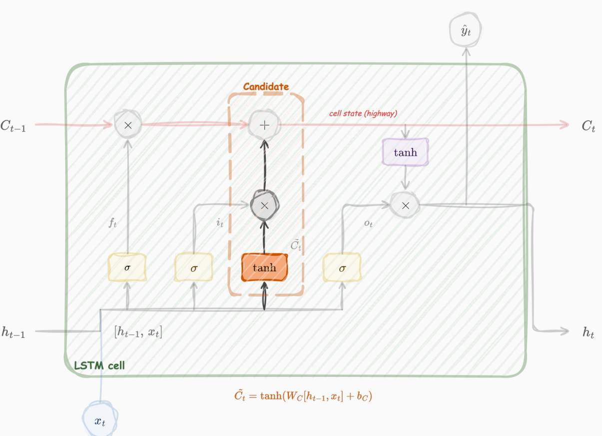
Candidate value



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

Candidate value

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



Dimensions:

- $W_C \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$, $b_C \in \mathbb{R}^{d_h}$
- $\tilde{C}_t \in (-1, 1)^{d_h}$: proposed content

Key properties:

- Uses tanh (+,-)
- Only non-gate component
- Gated contribution: $i_t \odot \tilde{C}_t$

Cell state update

The cell state is updated by **forgetting** selectively and **adding** selectively:

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

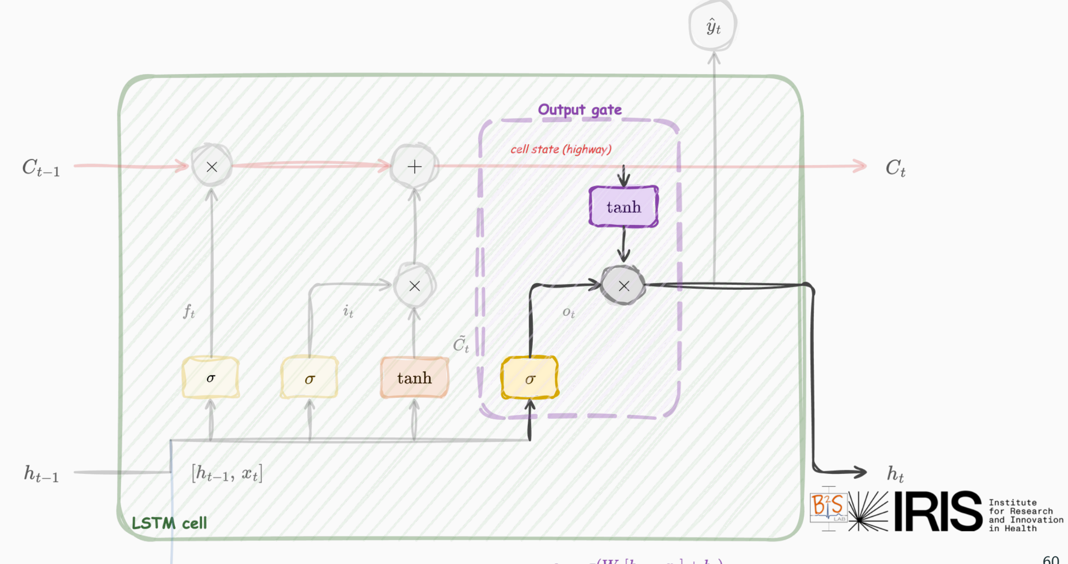
- $f_t \odot C_{t-1}$: keep selected parts of the old memory
- $i_t \odot \tilde{C}_t$: write selected parts of the new candidate
- $\odot$ denotes element-wise (Hadamard) product

The critical difference from vanilla RNNs

This update is **additive** in C_{t-1}. When $f_t \approx 1$ and $i_t \approx 0$, the cell state passes through **unchanged** — no matrix multiplication, no tanh squashing. This is why gradients can survive over long distances.

B2S LAB IRIS Institute for Research and Innovation in Health

Output gate



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

Output gate

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

Dimensions:

- $W_o \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$, $b_o \in \mathbb{R}^{d_h}$
- $o_t \in (0,1)^{d_h}$

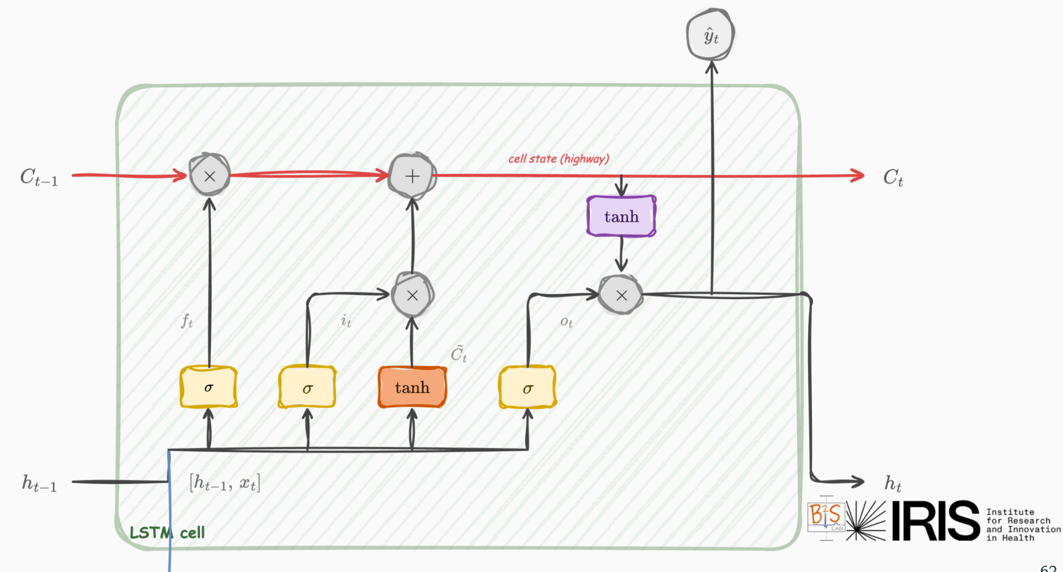
Two-step output:

- $\tanh(C_t)$ $(-1, 1)$
- o_t selects dims to **reveal**
- h_t feeds preds **and** next step

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

C_t can store values outside $(-1, 1)$ because only the output view h_t is squashed. This

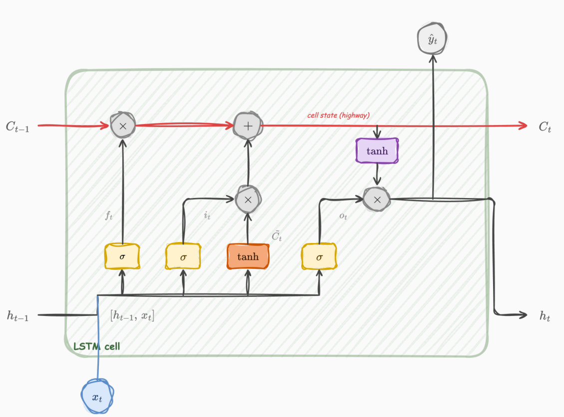
Putting it all together



Putting it all together

All four equations for one time step:

$$\begin{aligned} f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) && (1) \\ i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) && (2) \\ \tilde{C}_t &= \tanh(W_C[h_{t-1}, x_t] + b_C) && (3) \\ C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t && (4) \\ o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) && (5) \\ h_t &= o_t \odot \tanh(C_t) && (6) \end{aligned}$$



Inputs: h_{t-1}, C_{t-1}, x_t. **Outputs:** h_t, C_t.

A step-by-step walkthrough

To consolidate, trace the computation for a single time step:

1. **Concatenate** inputs: form $[h_{t-1}, x_t] \in \mathbb{R}^{d_h + d_{\text{in}}}$
2. **Forget:** compute $f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$ — decide what to erase from memory
3. **Input + Candidate:** compute i_t and $\tilde{C}_t$ — decide what new content to write
4. **Update cell:** $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$ — the additive highway update
5. **Output:** compute o_t and $h_t = o_t \odot \tanh(C_t)$ — decide what to expose

Tip

In practice, the four matrix multiplications (W_f, W_i, W_C, W_o) are **fused** into a single large multiplication and then split. This is why you will see implementations using a single weight matrix of size $4d_h \times (d_h + d_{\text{in}})$.

Parameter count

Each gate has its own weight matrix and bias:

- Each of the 4 weight matrices: $W \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$
- Each of the 4 bias vectors: $b \in \mathbb{R}^{d_h}$

Total parameters per LSTM layer:

$$4\, d_h\,(d_h + d_{\text{in}}) + 4\, d_h = 4\, d_h\,(d_h + d_{\text{in}} + 1)$$

> **Note**
>
> An LSTM has roughly **4 times** the parameters of a vanilla RNN with the same d_h and d_{in}. This is the cost of gating. In efficient implementations, the four matrix multiplications are fused into one large product.

B2S LAB IRIS Institute for Research and Innovation in Health

Why LSTMs mitigate vanishing gradients

The LSTM cell state evolves as:

$$C_s = f_s \odot C_{s-1} + i_s \odot \tilde{C}_s$$

where:

- f_s: forget gate
- i_s: input gate
- $\tilde{C}_s$: candidate cell update

The key term is:

$$f_s \odot C_{s-1}$$

which carries the previous memory forward by **elementwise scaling**.

Remember about cell state C_s

The cell state at time (s) is a vector:

$$C_s \in \mathbb{R}^{d_h} \qquad \text{with} \qquad C_s = \begin{bmatrix} C_s^{(1)} \\ C_s^{(2)} \\ \vdots \\ C_s^{(d_h)} \end{bmatrix}$$

- s indexes the **time step**
- j indexes the **position inside the vector**
- $C_s^{(j)}$ means the j**-th component** of the cell state at time s

So j simply labels one memory dimension of the LSTM.

Remember about cell state C_s

(j) is the **index of one component** of the hidden/cell vector, (s) is the time-steop. If the cell state is

$$C_s \in \mathbb{R}^{d_h}, C_s = \begin{bmatrix} C_s^{(1)} \\ C_s^{(2)} \\ \vdots \\ C_s^{(d_h)} \end{bmatrix}$$

So:

$$C_s^{(j)} = f_s^{(j)} C_{s-1}^{(j)} + \cdots$$

Each coordinate evolves independently in that derivative, which is why the Jacobian is diagonal.

One-step gradient

Differentiating the cell state with respect to the previous one gives:

$$\frac{\partial C_s}{\partial C_{s-1}} = \mathrm{diag}(f_s)$$

This is because each coordinate satisfies:

$$C_s^{(j)} = f_s^{(j)} C_{s-1}^{(j)} + \cdots$$

so:

$$\frac{\partial C_s^{(j)}}{\partial C_{s-1}^{(j)}} = f_s^{(j)}$$

Thus, the Jacobian is **diagonal**: each memory dimension is scaled independently.

B2S LAB IRIS Institute for Research and Innovation in Health

Backpropagation through many timesteps

From time T back to time t, the gradient along the cell-state path is:

$$\frac{\partial C_T}{\partial C_t} = \prod_{s=t+1}^{T} \mathrm{diag}(f_s)$$

For each coordinate (j), this becomes:

$$\frac{\partial C_T^{(j)}}{\partial C_t^{(j)}} = \prod_{s=t+1}^{T} f_s^{(j)}$$

So the gradient is passed backward by repeated mult. of the **forget gates**.

If $f_s \approx 1$, product stays close to 1, and gradients flow across long time spans.

Comparing gradient paths

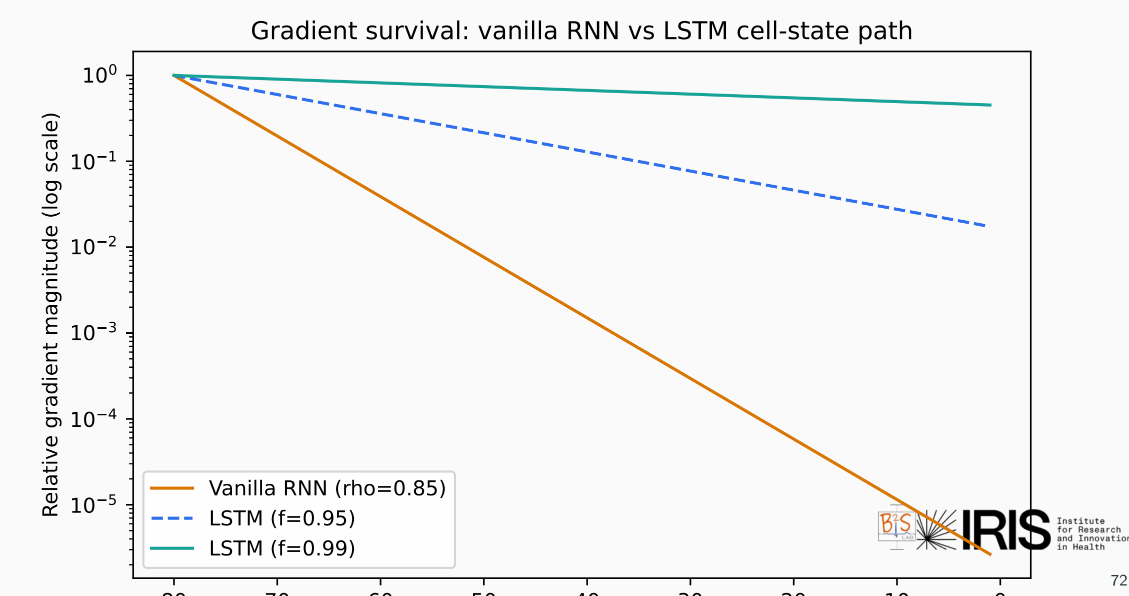
	Vanilla RNN	LSTM (cell state path)
Gradient factor per step	$\mathrm{diag}(\tanh'(z_s))\ W_h$	$\mathrm{diag}(f_s)$
Involves weight matrix	Yes	No
Spectral norm	Can be > 1 or $\ll 1$	In $(0, 1)$ per element
Additive path	No	Yes
Effective memory	10–20 steps	100+ steps possible

Tip

The forget gate acts as a **learnable decay rate** per dimension. The network decides, based on the input, how long to retain each piece of information. This is far more flexible than the fixed decay imposed by repeated multiplication through W_h.

BIS LAB IRIS Institute for Research and Innovation in Health

Gradient flow: a visual comparison



LSTMs in bioinformatics

LSTMs have been widely applied to biological sequence tasks:

- **Protein secondary structure prediction:** per-residue classification over sequences of 100–1000 residues, where long-range contacts matter
- **Splice site detection:** dependencies between donor and acceptor sites span hundreds of nucleotides
- **Clinical time series:** ICU monitoring, longitudinal patient records with irregular sampling and long-term dependencies
- **Gene expression forecasting:** time-course experiments where early perturbations affect late responses

Note

In many of these tasks, LSTMs were the dominant architecture before Transformers. They remain a strong baseline and are often preferred when data is limited, because they have fewer parameters than attention-based models of comparable capacity.

Practical considerations

- **Initialization:** set forget gate biases to 1.0 so it starts by remembering
- **Gradient clipping:** recommended, even though LSTMs behave better than vanilla RNNs
- **Stacking:** multiple LSTM layers can be stacked; the output h_t of one layer becomes the input x_t of the next
- **Bidirectional LSTMs:** run one LSTM forward and one backward, concatenate their hidden states — useful when the full sequence is available (e.g., protein classification, but not autoregressive generation)
- **Hidden size:** typical values are $d_h \in [64, 512]$ for biomedical tasks; larger values risk overfitting on small datasets

Warning

LSTMs are slower to train than vanilla RNNs (4x more parameters, 4x more computation per step) and cannot be parallelised across time steps. For very long sequences ($T > 1000$), consider Transformers or structured state-space models.

Gated Recurrent Unit (GRU)

Motivation: simplifying the LSTM

The LSTM introduced four components (three gates plus a candidate) and two state vectors (C_t, h_t). This works well, but:

- The parameter count is **4x** that of a vanilla RNN
- Two separate state vectors add implementation complexity
- For many tasks, the full gating machinery is more than necessary

The core idea

The GRU merges the cell state and hidden state into a **single vector** h_t, and reduces the gate count from three to **two**: a reset gate and an update gate. It retains the key benefit of LSTMs — learnable, per-dimension control over memory retention — with fewer parameters.

One state vector, two gates

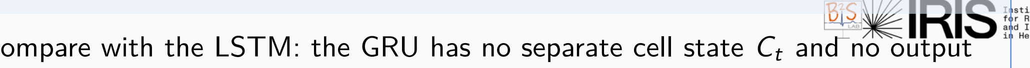
A GRU cell maintains a **single** state vector:

- **Hidden state** $h_t \in \mathbb{R}^{d_h}$ — serves as both memory and output

Two gates control how h_t is updated:

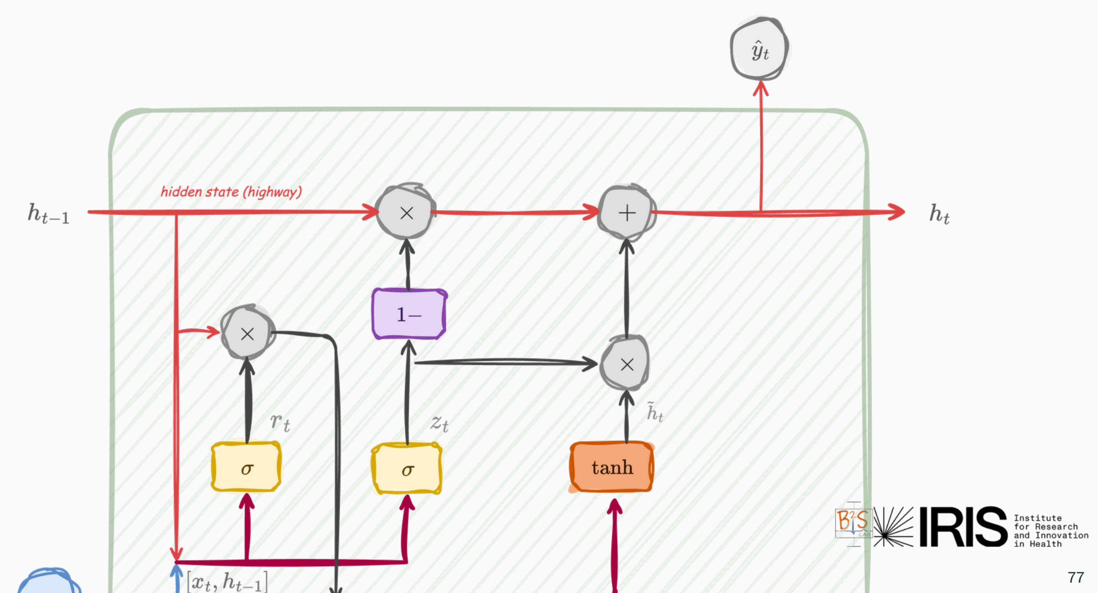
Gate	Symbol	Role
Reset	r_t	How much of h_{t-1} to **use** when computing the candidate
Update	z_t	How much of h_{t-1} to **carry forward** vs replace

Note

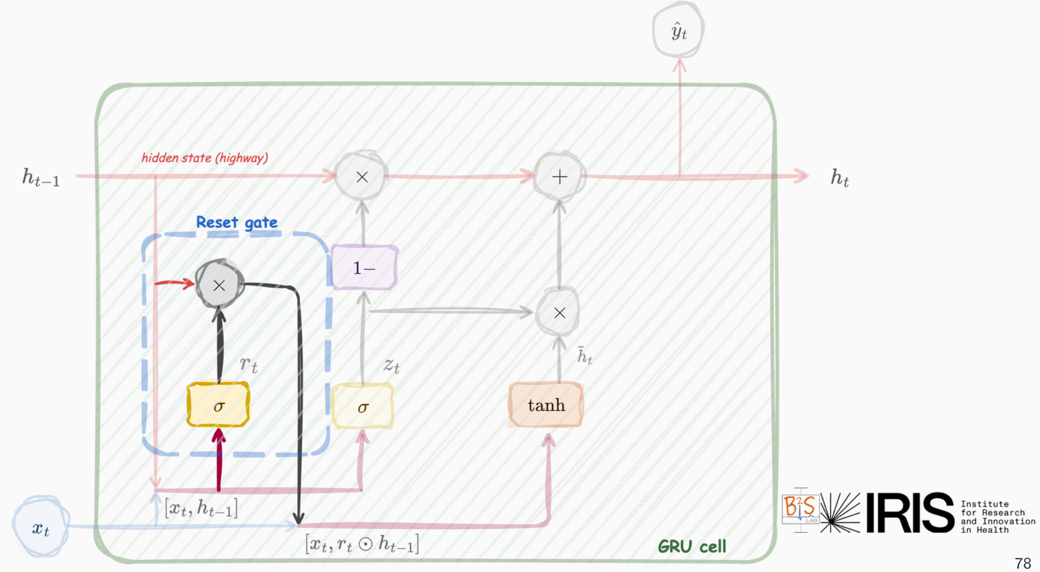


Compare with the LSTM: the GRU has no separate cell state C_t and no output gate. The update gate z_t plays the combined role of the LSTM forget and input

The GRU cell at a glance

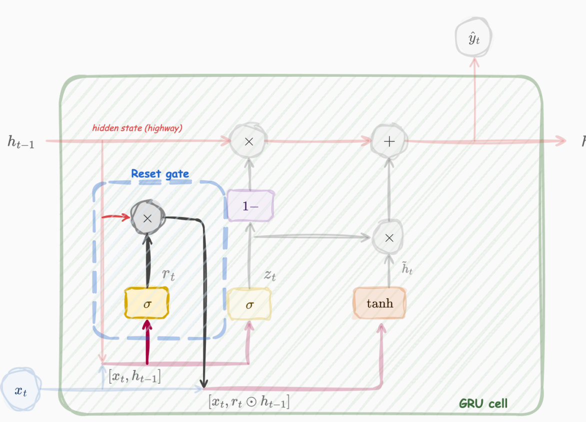


Reset gate



Reset gate

$$r_t = \sigma\left(W_r\,[h_{t-1},\, x_t] + b_r\right)$$



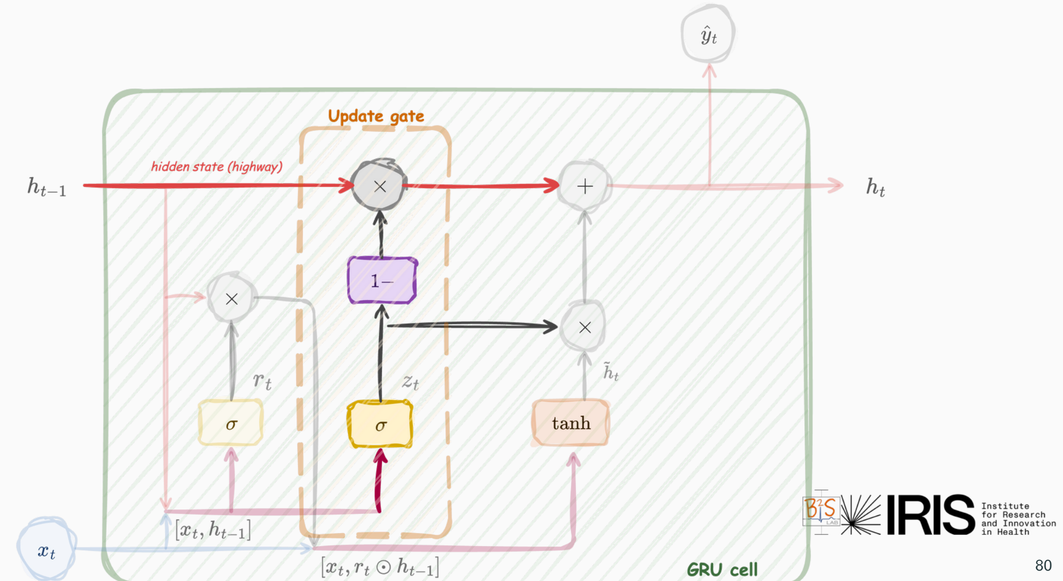
Dimensions:

- $[h_{t-1}, x_t] \in \mathbb{R}^{d_h + d_{\text{in}}}$
- $W_r \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$, $b_r \in \mathbb{R}^{d_h}$
- $r_t \in (0, 1)^{d_h}$

Per-dimension behaviour:

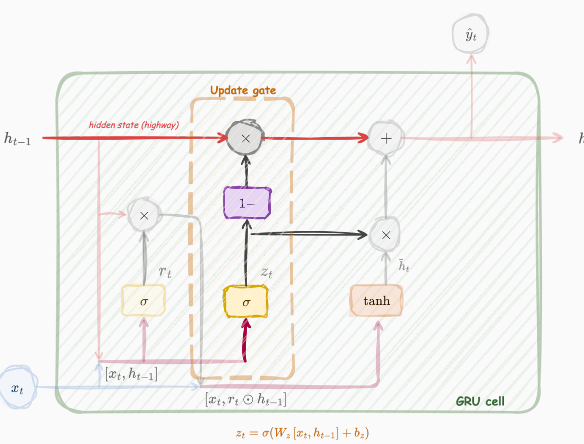
- $r_t^{(j)} \approx 1$: past **visible** to candidate
- $r_t^{(j)} \approx 0$: past **ignored** by candidate

Update gate



Update gate

$$z_t = \sigma\left(W_z\,[h_{t-1},\, x_t] + b_z\right)$$

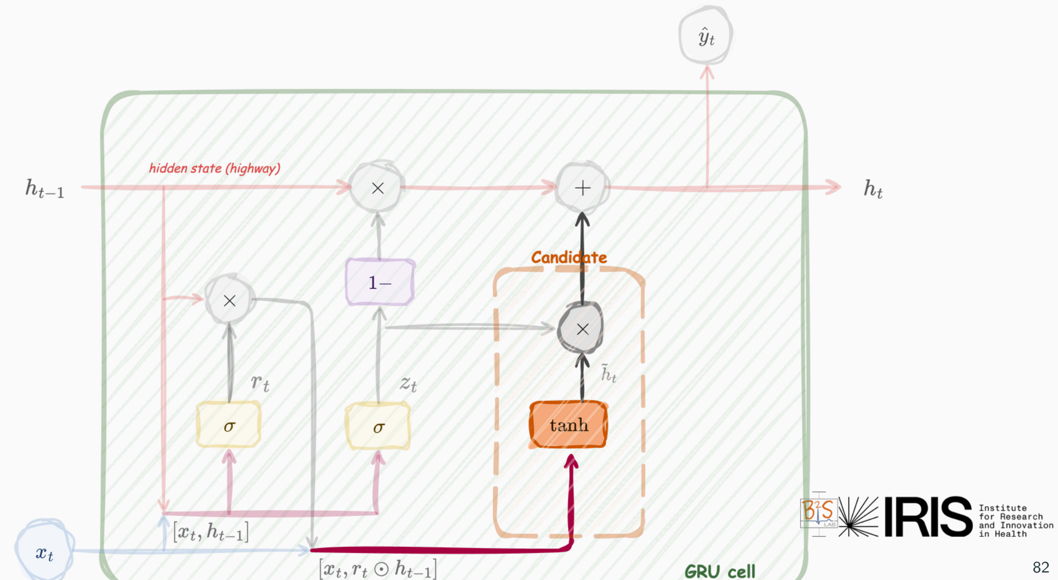


Dimensions:

- $W_z \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$, $b_z \in \mathbb{R}^{d_h}$
- $z_t \in (0, 1)^{d_h}$

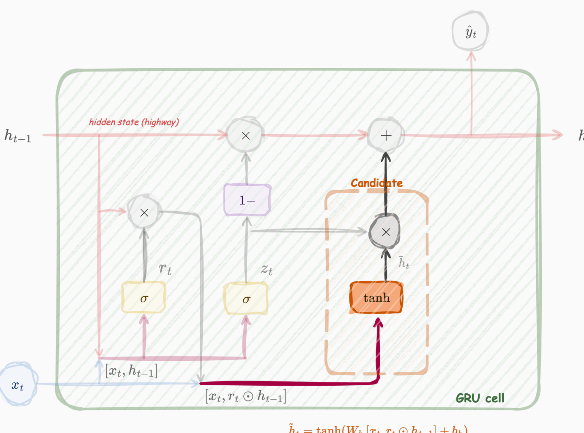
Dual role:

- z_t controls **retention** of h_{t-1}
- $(1 - z_t)$ controls **admission** of $\tilde{h}_t$
- One gate replaces both LSTM forget and input gates


$\hat{y}_t$
h_{t-1}
hidden state (highway)
h_t
Candidate
$1-$
r_t
z_t
$\tilde{h}_t$
σ
σ
tanh
$[x_t, h_{t-1}]$
x_t
$[x_t, r_t \odot h_{t-1}]$
GRU cell
IRIS
Institute for Research and Innovation in Health

Candidate hidden state

$$\tilde{h}_t = \tanh\big(W_h\,[r_t \odot h_{t-1},\, x_t] + b_h\big)$$



Dimensions:

- $W_h \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$, $b_h \in \mathbb{R}^{d_h}$
- $\tilde{h}_t \in (-1, 1)^{d_h}$

Key difference from LSTM:

- The reset gate modulates h_{t-1} **before** it enters the linear combination
- When $r_t \approx 0$, the candidate sees only x_t

Hidden state update

The new hidden state is a **convex interpolation** between the old state and the candidate:

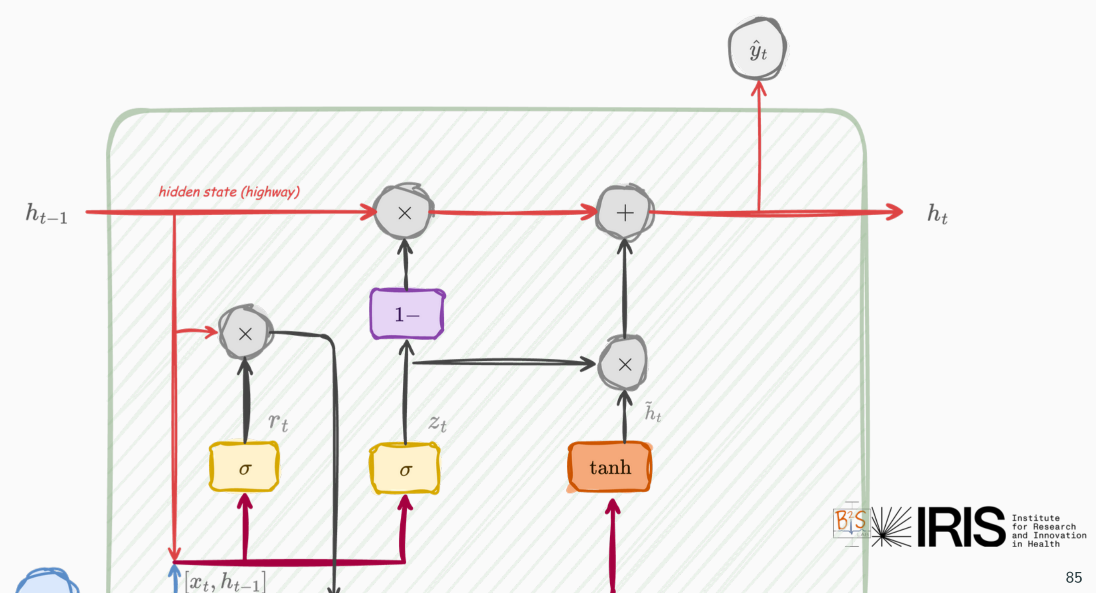
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

- $z_t \odot h_{t-1}$: the fraction of old memory that **survives**
- $(1 - z_t) \odot \tilde{h}_t$: the fraction of new content that is **written**
- Since $z_t + (1 - z_t) = 1$, this is a **weighted average** per dimension

The additive memory mechanism

When $z_t \approx 1$, the hidden state passes through nearly unchanged — no matrix multiplication, no tanh compression. This is the same principle that gives LSTMs long-range gradient flow, achieved here with a simpler structure.

Putting it all together



Putting it all together

All three equations for one time step:

$$
\begin{aligned}
r_t &= \sigma\left(W_r[h_{t-1}, x_t] + b_r\right) && (7)\\
z_t &= \sigma\left(W_z[h_{t-1}, x_t] + b_z\right) && (8)\\
\tilde{h}_t &= \tanh\left(W_h[r_t \odot h_{t-1}, x_t] + b_h\right) && (9)\\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t && (10)
\end{aligned}
$$

Input: h_{t-1}, x_t. **Output:** h_t.

A step-by-step walkthrough

To consolidate, trace the computation for a single time step:

1. **Concatenate** inputs: form $[h_{t-1}, x_t] \in \mathbb{R}^{d_h+d_{\text{in}}}$
2. **Reset:** compute $r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$ — decide how much past to show the candidate
3. **Candidate:** compute $\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h)$ — propose new content
4. **Update:** compute $z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$ — decide how much to keep vs replace
5. **Interpolate:** $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$

Tip

In practice, the three matrix multiplications (W_r, W_z, W_h) are **fused** into a single product using a weight matrix of size $3d_h \times (d_h + d_{\text{in}})$, then split and routed to the appropriate activations.

Parameter count

Each of the 3 components has its own weight matrix and bias:

- Each weight matrix: $W \in \mathbb{R}^{d_h \times (d_h + d_{\text{in}})}$
- Each bias vector: $b \in \mathbb{R}^{d_h}$

Total parameters per GRU layer:

$$3\,d_h\,(d_h + d_{\text{in}}) + 3\,d_h = 3\,d_h\,(d_h + d_{\text{in}} + 1)$$

> **Note**
>
> A GRU has **3/4 the parameters** of an LSTM with the same d_h and d_{in}. For the same parameter budget, a GRU can use a larger hidden size, which sometimes compensates for the simpler gating structure.

Why GRUs mitigate vanishing gradients

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

Differentiating with respect to the previous state (dominant term):

$$\frac{\partial h_t}{\partial h_{t-1}} \approx \mathrm{diag}(z_t)$$

Over many steps, the gradient along the direct path is:

$$\frac{\partial h_T^{(j)}}{\partial h_t^{(j)}} \approx \prod_{s=t+1}^{T} z_s^{(j)}$$

When $z_s \approx 1$, the product stays close to 1 and gradients survive.

ℹ Note

Comparing gradient paths: RNN vs LSTM vs GRU

	Vanilla RNN	LSTM (cell path)	GRU (hidden path)
Gradient factor	$\mathrm{diag}(\tanh') \, W_h$	$\mathrm{diag}(f_s)$	$\mathrm{diag}(z_s)$
Weight matrix	Yes	No	No
Additive path	No	Yes (C_t)	Yes (h_t)
State vectors	1 (h_t)	2 (h_t, C_t)	1 (h_t)
Gates	0	3	2
Params (relative)	1x	4x	3x

GRU vs LSTM: when to choose which

Prefer GRU when

- Dataset is **small** (fewer parameters to overfit)
- Sequence lengths are moderate ($T < 200$)
- Training speed matters (faster per step)
- A simpler model is easier to debug and interpret
- Initial baseline before scaling up

Prefer LSTM when

- Sequences are **very long** ($T > 500$)
- The task requires storing information and revealing it later (output gate helps)
- You need fine-grained independent control of forgetting and writing
- Empirical results on your data favour it

Note

On many benchmarks, GRUs and LSTMs perform **comparably**. The choice often depends on dataset size, sequence length, and computational budget rather than on a clear architectural winner. When in doubt, try both and compare on validation data.

IRIS Institute for Research and Innovation in Health

GRUs in bioinformatics

GRUs have been applied to many of the same tasks as LSTMs:

- **Protein function prediction:** per-residue or per-sequence classification where moderate-length dependencies suffice
- **Drug-target interaction:** encoding molecular SMILES strings, where sequences are typically shorter than protein sequences
- **Clinical event prediction:** patient trajectories with tens to hundreds of time points, where the smaller parameter count reduces overfitting on limited cohorts
- **Enhancer-promoter interaction:** DNA sequence classification tasks where bidirectional GRUs capture local and mid-range regulatory patterns

Tip

When data is scarce — a common situation in clinical and rare-disease settings — the GRU's lower parameter count can be a decisive advantage over the LSTM.

IRIS Institute for Research and Innovation in Health

Secondary Structure Demo

link

Practical considerations

- **Initialization:** unlike the LSTM, there is no standard trick for bias initialization; default zeros work in most frameworks
- **Gradient clipping:** still recommended, as with any recurrent architecture
- **Stacking:** multiple GRU layers stack the same way as LSTMs; h_t of one layer becomes x_t of the next
- **Bidirectional GRUs:** same principle as bidirectional LSTMs; concatenate forward and backward h_t vectors
- **Hidden size:** since GRUs have 3/4 the parameters of LSTMs, you can afford a slightly larger d_h for the same compute budget

Warning

Like LSTMs, GRUs process time steps **sequentially** and cannot be parallelised across the time axis. For very long sequences or large-scale pretraining, Transformers are generally preferred.

IRIS Institute for Research and Innovation in Health

Training and implementation

Why recurrent training is harder than feedforward

- In an MLP, gradients flow through a fixed, shallow graph.
- In an RNN, gradients flow through a graph unrolled T times – the effective depth equals the sequence length.
- Long sequences amplify two failure modes:
  - **Vanishing gradients:** signal decays exponentially with T
  - **Exploding gradients:** signal grows exponentially with T
- Initialisation, clipping, and gating are the main tools to keep training stable.

> **Note**
>
> Gates (LSTM, GRU) address vanishing gradients architecturally. The practical choices on this slide address the remaining engineering problems.

Initialisation of recurrent weights

The recurrent weight matrix $W_h \in \mathbb{R}^{d_h \times d_h}$ is applied at every time step:

$$h_t = \phi\big(W_x\, x_t + W_h\, h_{t-1} + b\big)$$

After T steps, the effective Jacobian involves products of W_h:

$$\frac{\partial h_T}{\partial h_0} \propto \prod_{t=1}^{T} \mathrm{diag}\big(\phi'(z_t)\big)\; W_h$$

- If $\|W_h\| < 1$ repeatedly, gradients **vanish**.
- If $\|W_h\| > 1$ repeatedly, gradients **explode**.

Initialisation strategies

Orthogonal initialisation (recommended default)

Sample W_h as a random orthogonal matrix: all singular values equal 1.

$$W_h^\top W_h = I \quad \Rightarrow \quad \sigma_i(W_h) = 1 \;\; \forall i$$

- Preserves gradient norms at initialisation.
- Especially effective for vanilla RNNs with tanh activation.

Xavier / He for input-to-hidden

- W_x: use **Xavier** (tanh/sigmoid) or **He** (ReLU), same as in feedforward layers.
- Biases: initialise to zero.

Tip

For LSTM forget gates, initialise the **forget bias to 1** (or higher). This keeps the gate open at the start of training, so information flows through the cell before the network has learned what to forget.

LSTM bias initialisation in detail

Recall the LSTM forget gate:

$$f_t = \sigma\big(W_f\, x_t + U_f\, h_{t-1} + b_f\big)$$

- At random initialisation, $\sigma(\cdot) \approx 0.5$ on average.
- Half-open gates lose cell information from the first epoch.
- Setting $b_f = 1$ (or $b_f = 2$) pushes f_t close to 1 initially.

This simple trick, proposed by Jozefowicz et al. (2015), often matters more than the choice of weight initialisation scheme.

> **Note**
>
> In PyTorch, `nn.LSTM` packs all four gate biases into one vector. You must index the correct slice to set the forget bias independently.

B2S LAB IRIS Institute for Research and Innovation in Health

Gradient clipping

Even with good initialisation, occasional gradient spikes can destabilise training.

Gradient norm clipping rescales the full gradient vector when its norm exceeds a threshold τ:

$$g \leftarrow \begin{cases} g & \text{if } \|g\| \leq \tau \\ \tau \dfrac{g}{\|g\|} & \text{if } \|g\| > \tau \end{cases}$$

- Preserves gradient **direction**; only reduces **magnitude**.
- Typical values: $\tau \in [1, 5]$.
- Applied **after** computing all gradients, **before** the optimiser step.

Warning

Clipping does not fix vanishing gradients. It only prevents explosions. For vanishing gradients, use gated architectures (LSTM, GRU).

Monitoring gradient norms

Logging $\|g\|$ per training step is one of the most useful diagnostics for recurrent models.

- **Healthy training:** gradient norms fluctuate within a stable range (e.g., 0.1–10).
- **Exploding gradients:** sudden spikes to 10^3 or higher; training loss jumps.
- **Vanishing gradients:** norms collapse to $< 10^{-4}$; loss plateaus early.

In PyTorch, compute the total norm before clipping:

```
total_norm = torch.nn.utils.clip_grad_norm_(
    model.parameters(), max_norm=tau
)
# total_norm is the pre-clip value -- log it
```

Tip

Plot gradient norms alongside the learning curve. If norms spike exactly when the loss jumps, you have an exploding-gradient event.

IRIS Institute for Research and Innovation in Health

Monitoring hidden state saturation

For tanh or sigmoid activations, **saturation** means most hidden units are near ± 1 (tanh) or near 0 / 1 (sigmoid).

- Saturated units produce near-zero gradients: $\tanh'(x) \approx 0$ for $|x| \gg 1$.
- The network stops learning through those units.

What to check

- Histogram of h_t values across the batch at a few time steps.
- Fraction of units with $|h_t| > 0.95$ (for tanh).
- If most units are saturated early in training, reduce the learning rate or check initialisation.

> **Note**
>
> LSTM cell states C_t are **unbounded** by design – they do not saturate. This is a key reason why LSTMs handle long-range dependencies better than vanilla RNNs.

Practical optimisation choices

Optimiser

- **Adam** is the standard default for recurrent models.
- Learning rate: start with 10^{-3}; reduce on plateau or use a scheduler.

Sequence length and batching

- Sort sequences by length; group similar lengths into batches.
- Use **packed sequences** (PyTorch `pack_padded_sequence`) to avoid computing on padding tokens.

Regularisation

- **Dropout between layers**, not between time steps (naive temporal dropout breaks the recurrent signal).
- Variational dropout (same mask at every step) is a principled alternative.

Practical optimisation choices (continued)

Learning rate scheduling

- **ReduceLROnPlateau:** halve the learning rate when validation loss stalls for k epochs.
- **Cosine annealing:** smooth decay; common in longer training runs.
- Warm-up (a few hundred steps at a low rate) helps if gradients are noisy at the start.

When training diverges

Symptom	Likely cause	Remedy
Loss becomes NaN	Exploding gradients	Lower τ, reduce learning rate
Loss plateaus immediately	Vanishing gradients or dead gates	Check initialisation, try LSTM/GRU
Train loss drops, val loss rises	Overfitting	Add dropout, reduce model size

IRIS Institute for Research and Innovation in Health

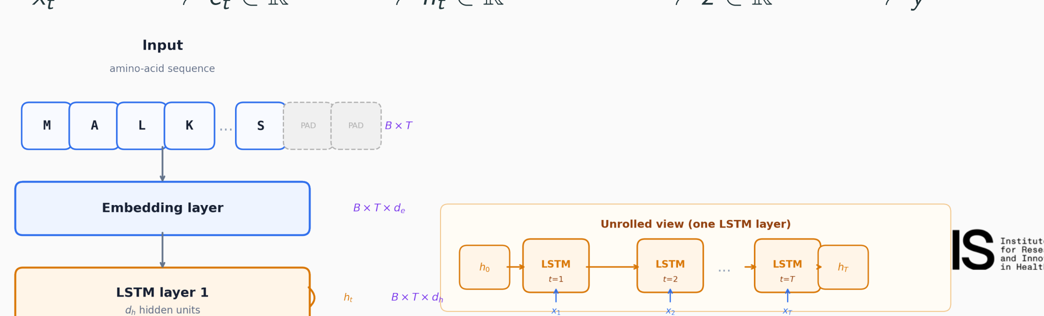
Worked example: sequence classification with an LSTM

Task: classify protein subcellular localisation from amino-acid sequences.

- Input: one-hot or embedding of amino acids, $x_t \in \mathbb{R}^{d_{\text{in}}}$
- Sequence length: variable (T ranges from tens to thousands)
- Output: one of k compartments (nucleus, cytoplasm, membrane, ...)

Architecture

$$x_t \xrightarrow{\text{embedding}} e_t \in \mathbb{R}^{d_e} \xrightarrow{\text{LSTM}} h_t \in \mathbb{R}^{d_h} \xrightarrow{\text{pool + linear}} z \in \mathbb{R}^k \xrightarrow{\text{softmax}} \hat{y}$$



Institute for Research and Innovation in Health

Worked example: pooling the hidden states

The LSTM produces one hidden vector h_t per time step. We need a single vector for classification.

Common pooling strategies:

- **Last hidden state:** h_T (simple, but biased toward the end of the sequence)
- **Mean pooling:** $\bar{h} = \frac{1}{T'} \sum_{t=1}^{T'} h_t$, where T' is the true (unpadded) length
- **Max pooling:** $\bar{h}_j = \max_t h_{t,j}$ (picks the strongest activation per dimension)

Tip

For protein sequences, **mean pooling over unpadded positions** is a robust default. It treats all residues equally and avoids the positional bias of using only h_T.

The padding mask $M \in \{0, 1\}^{B \times T}$ is essential here: pool only over positions where $M_{b,t} = 1$.

Worked example: training loop (pseudocode)

```
for epoch in 1 ... max_epochs:
    for batch in dataloader:
        x, lengths, y = batch             # padded input, true lengths, labe
        x_packed = pack(x, lengths)        # avoid computing on padding
        _, (h_T, _) = lstm(x_packed)       # forward pass
        logits = linear(h_T.squeeze(0))    # map to k classes
        loss = cross_entropy(logits, y)

        loss.backward()                    # backward pass
        norm = clip_grad_norm_(params, tau)  # clip + log
        optimiser.step()
        optimiser.zero_grad()

    val_loss = evaluate(val_loader)
    scheduler.step(val_loss)               # reduce LR on plateau
    if val_loss < best:
```

IRIS Institute for Research and Innovation in Health

Worked example: what to monitor

During training, log at every epoch:

- **Training loss** and **validation loss** (learning curves)
- **Gradient norm** before clipping (stability diagnostic)
- **Validation accuracy** or AUROC (task-level performance)

After training, inspect:

- **Confusion matrix** on the held-out test set
- **Per-class performance** (localisation classes are often imbalanced)
- **Attention to sequence length:** does the model perform differently on short vs long proteins?

Warning

Split proteins by **sequence identity**, not by random sampling. Homologous sequences in both train and test sets inflate performance and hide poor generalisation.

IRIS Institute for Research and Innovation in Health

Summary: training recurrent models

Concern	Recommendation
Weight init (W_h)	Orthogonal; forget bias $= 1$ for LSTM
Gradient explosions	Clip global norm, $\tau \in [1, 5]$
Vanishing gradients	Use LSTM or GRU
Optimiser	Adam, lr $\approx 10^{-3}$
Regularisation	Dropout between layers; weight decay
Monitoring	Log gradient norms + learning curves every epoch
Stopping criterion	Early stopping on validation loss
Batching	Pack sequences; sort by length

Notation

Notation for sequences, states, and batches

We will use the following conventions **throughout the course**.

Single sequence

- Input at one step: $x_t \in \mathbb{R}^{d_{\text{in}}}$, for $t \in \{1, \ldots, T\}$
- Full sequence as a matrix: $X \in \mathbb{R}^{T \times d_{\text{in}}}$
- Hidden state at step t: $h_t \in \mathbb{R}^{d_h}$
- Cell state (LSTM only): $C_t \in \mathbb{R}^{d_h}$

Batch of sequences (training)

- Batch size: B (number of sequences processed together)
- Padded input tensor: $X \in \mathbb{R}^{B \times T \times d_{\text{in}}}$, where $T = T_{\text{max}}$ for the batch
- Padding mask: $M \in \{0, 1\}^{B \times T}$, where $M_{b,t} = 0$ marks a padding position for sequence b

Tip

Sequences in a batch typically have **different true lengths**. Padding extends each

Notation (we will reuse everywhere

We will keep the same conventions as in the MLP slides:

- Scalars: b, vectors: $\mathbf{x}$, matrices: W
- Time index: $t \in \{1, \dots, T\}$
- Batch size: B

Sequence input

- Single sequence: $x_t \in \mathbb{R}^{d_{\text{in}}}$, collected as $X \in \mathbb{R}^{T \times d_{\text{in}}}$
- Batch of padded sequences: $X \in \mathbb{R}^{B \times T \times d_{\text{in}}}$
- Optional mask: $M \in \{0, 1\}^{B \times T}$, where $M_{b,t} = 0$ means "padding"

Hidden state

- $h_t \in \mathbb{R}^{d_h}$ (the recurrent "memory" at time t)
- For gated models we will also use a **cell state** $C_t \in \mathbb{R}^{d_h}$ (LSTM)

Questions

References

Anfinsen, Christian B. 1973. "Principles That Govern the Folding of Protein Chains." *Science* 181: 223–30.

ENCODE Project Consortium. 2012. "An Integrated Encyclopedia of DNA Elements in the Human Genome." *Nature* 489: 57–74.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. 1989. "Multilayer Feedforward Networks Are Universal Approximators." *Neural Networks* 2: 359–66.

Jumper, John et al. 2021. "Highly Accurate Protein Structure Prediction with AlphaFold." *Nature* 596: 583–89.

Madani, Ali et al. 2023. "Large Language Models Generate Functional Protein Sequences Across Diverse Families." *Nature Biotechnology* 41: 1099–1106.