NASA Cosmic Origins AI/ML STIG

Recurrent Neural Networks

From Theory to Astronomical Light Curve Classification

Daniel Muthukrishna

MIT / Harvard & Smithsonian CfA

23rd February 2025

Massachusetts Institute of Technology

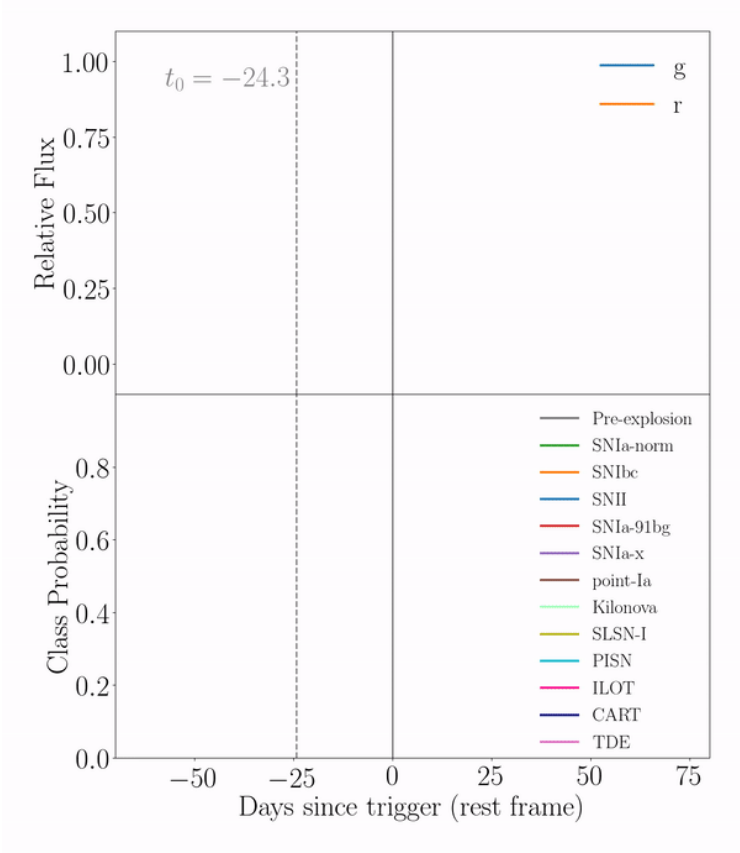
CENTER FOR ASTROPHYSICS HARVARD & SMITHSONIAN

Lecture Outline

1. **Why RNNs?**
   Motivation for sequential models in astronomy?
2. **RNN Fundamentals**
   Vanilla RNNs, hidden states, backpropagation through time
3. **LSTM & GRU**
   Gating mechanisms and long-range dependencies
4. **RNNs vs Transformers**
   Trade-offs, strengths, and practical guidance
5. **Case Study: Supernova Light Curve Classification**
   LSTM-based supernova classification for ZTF / LSST
6. **Hands-on Tutorial**
   Jupyter notebook walkthrough

Classifying Light Curves with RNNs

- Unlike traditional Feedforward neural networks, RNNs allow information to persist through time
- By the end of this lecture, we'll be able to build a classifier of light curves as a function of time



Muthukrishna et al. 2019

1. Why Recurrent Neural Networks?

The challenge of sequential data

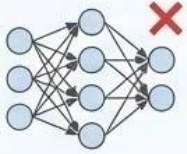
In astronomy, many of our most important datasets are time-series:

- Transient light curves — supernovae, kilonovae, tidal disruption events
- Variable star photometry — Cepheids, RR Lyrae, eclipsing binaries
- Radial velocity measurements — exoplanet detection
- Gravitational wave strain data

These datasets share a critical property: the order of observations matters. A supernova rising in brightness tells us something very different from one that is fading.

What's wrong with feedforward networks?

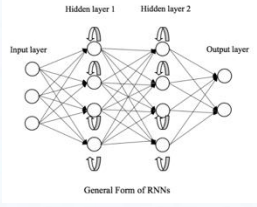
A standard feedforward network (MLP) or CNN treats each input independently or requires a fixed-size input window.



For time series of variable length and irregular cadence (common in astronomy!), we need architectures that can:

- Process sequences of arbitrary length
- Maintain a memory of what came before
- Make predictions at any point in the sequence

This is precisely what RNNs are designed to do.



From Feedforward to Recurrent

Feedforward Network

- Fixed-size input → Fixed-size output
- No memory of previous inputs
- Each sample processed independently
- Cannot handle variable-length sequences
- Must engineer features from time-series

Recurrent Neural Network

- Processes sequences step by step
- Maintains hidden state ("memory")
- Shares weights across time steps
- Handles variable-length inputs natively
- Learns temporal features automatically

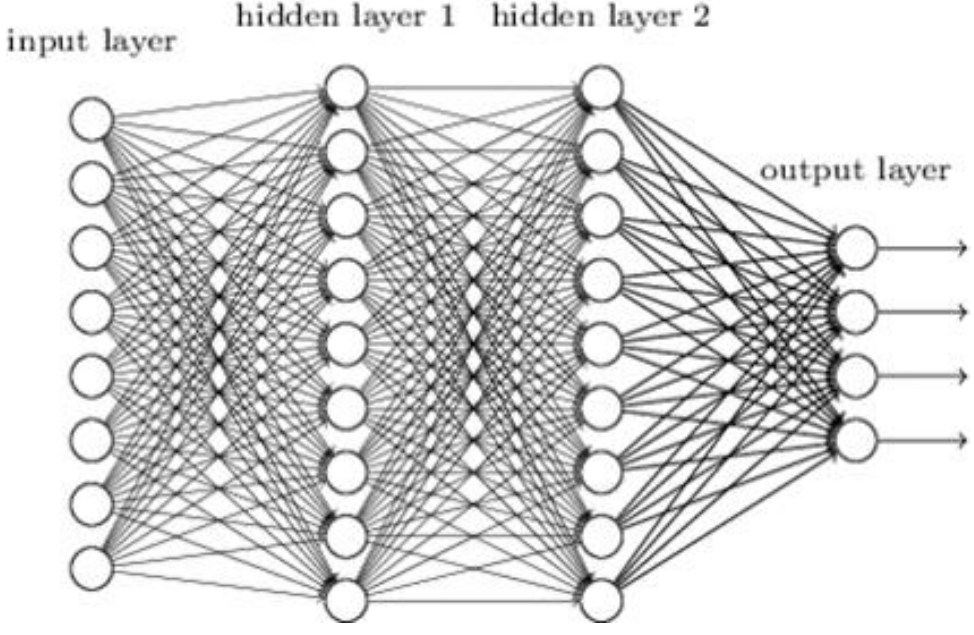
Comparison of architectures

Data Type	Architecture	Inductive Bias
Images (2D spatial)	**CNNs** (Lecture 9)	Translation equivariance, locality
Graphs / Point clouds	**GNNs** (Lecture 10)	Permutation equivariance
Set data (no ordering)	**Transformers** (Lecture 11)	Permutation equivariance + attention
Sequential / Time series	**RNNs** (This lecture)	**Temporal ordering, memory**

Key insight: The RNN inductive bias is that recent past is informative about the present, and information flows forward in time.

Feedforward Neural Network (Multilayer Perceptron)

$$\hat{y}_i = f\left(\sum_{j=1}^{M} W_{ij}\, x_j + b_i\right)$$

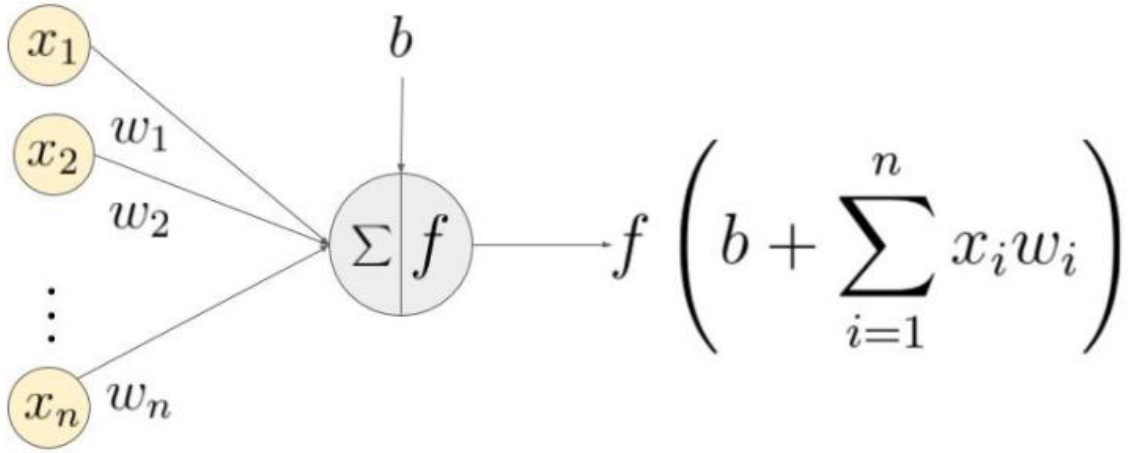


- The value of each node/neuron is computed from all the lines connected to it
- Each line has an associated *weight*
- Each node has an associated *bias*
- Feedforward NN only move in one direction
  - The information never touches a node twice and have no memory of the input they received previously

Activation Functions

$$\hat{y}_i = f\left(\sum_{j=1}^{M} W_{ij}\, x_j + b_i\right)$$

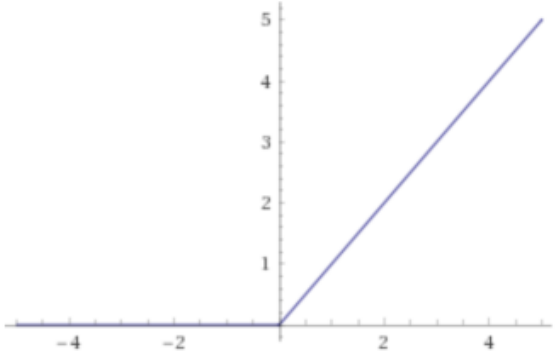
- Introduces non-linearity into the network
  - Important for stacking layers
- Can keep output values bounded



Activation Functions

ReLU Function

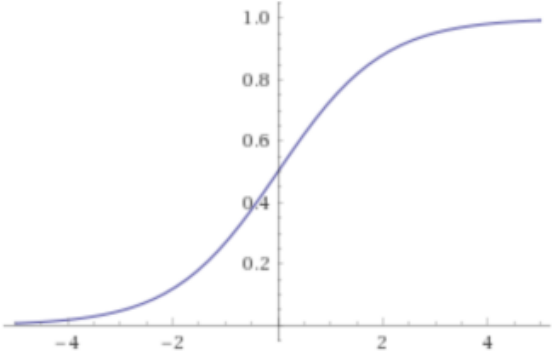
$$f(x) = \max(0, x)$$



Fast, reduces risk of *vanishing gradient problem*

Sigmoid function

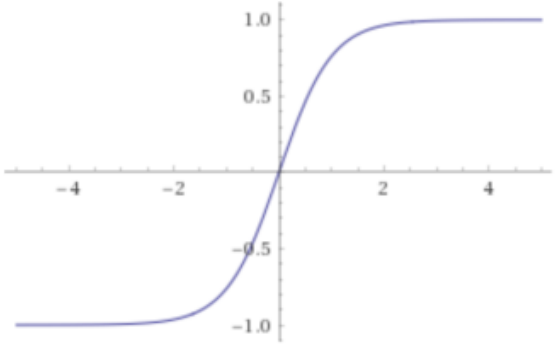
$$f(x) = \frac{1}{1 + e^{-x}}$$



Smooth activation outputs values between 0 and 1. Useful for classification

Tanh function

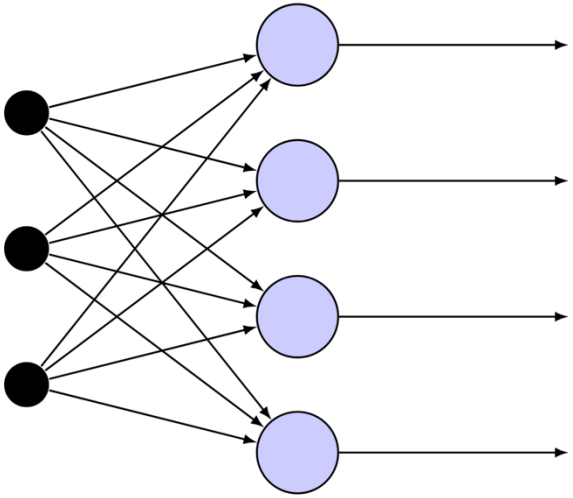
$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$



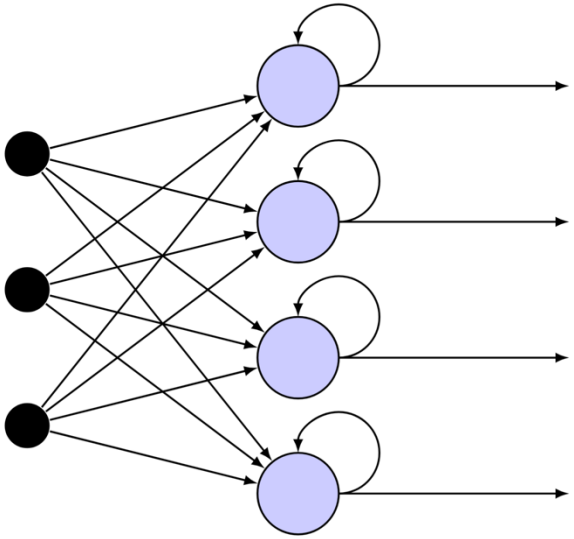
Steeper gradient strength than sigmoid. Vanishing gradient problem as saturate neurons

Feedforward vs Recurrent Neural Networks

- RNNs use *backpropagation through time* to update network weight parameters
- They are able to *remember* information in a sequence

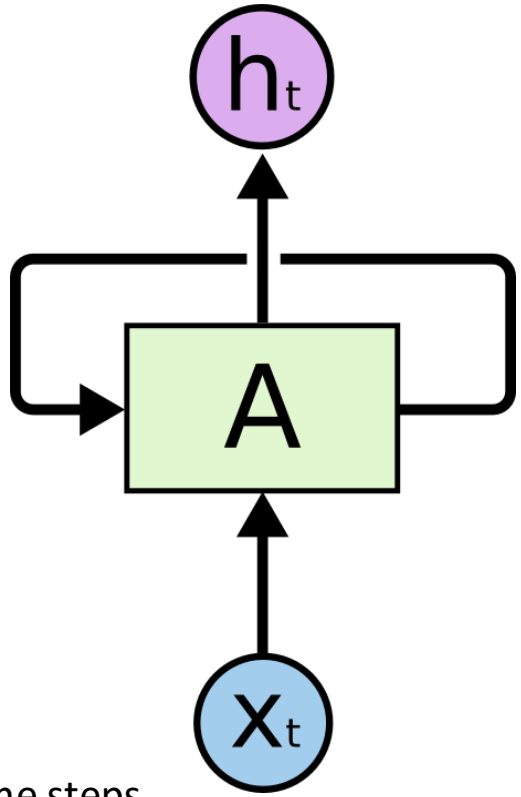


Feedforward Neural Network



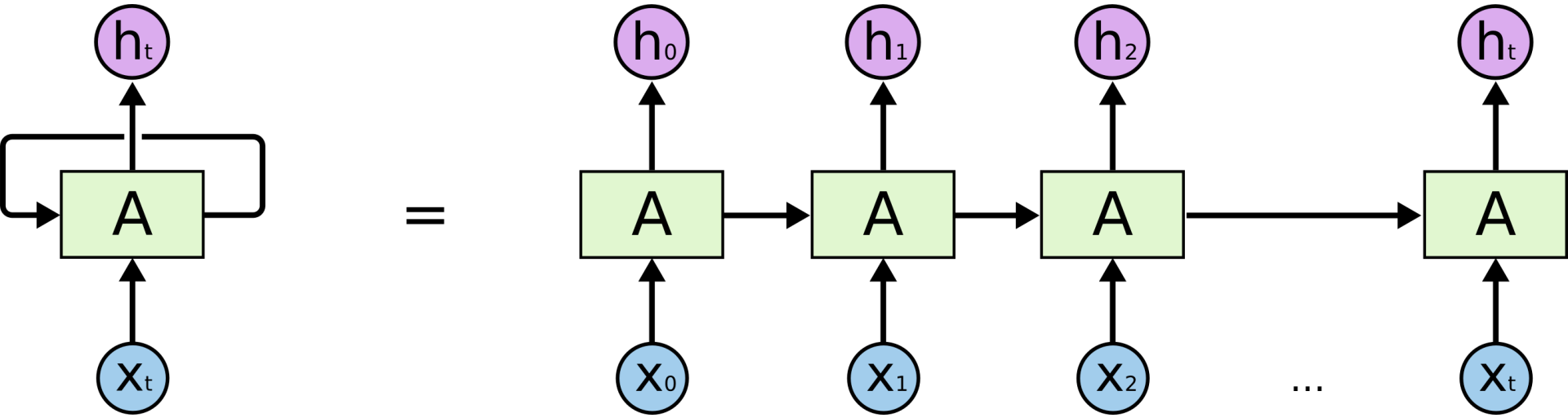
Recurrent Neural Network

Recurrent Neural Network



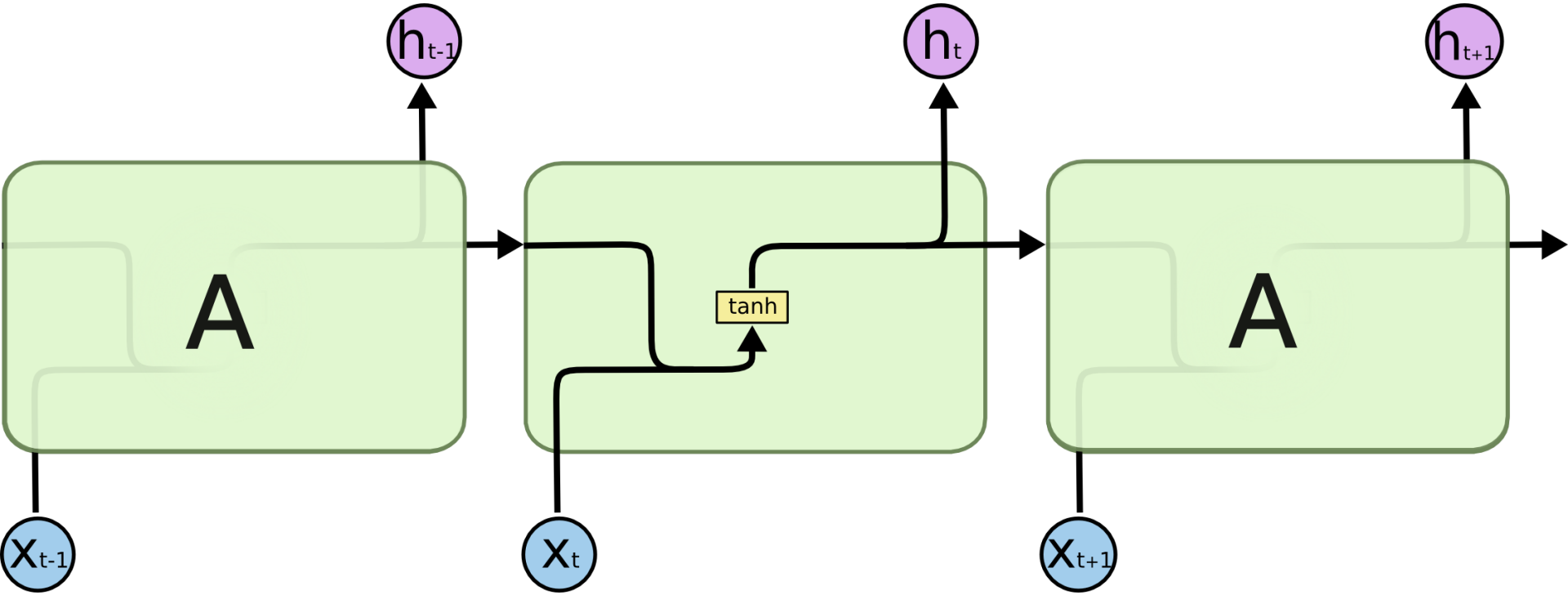
- Each node has two inputs
  1. Current timestep input
  2. Output of previous node
- Can retain a *memory* of previous time steps

Recurrent Neural Network



- Each node has two inputs
  1. Current timestep input
  2. Output of previous node
- Can retain a *memory* of previous time steps

Vanilla RNN



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

2. Vanilla RNN

An RNN processes a sequence one element at a time, maintaining a **hidden state $\mathbf{h}_t$** that acts as the networks "memory" of what it has seen so far.

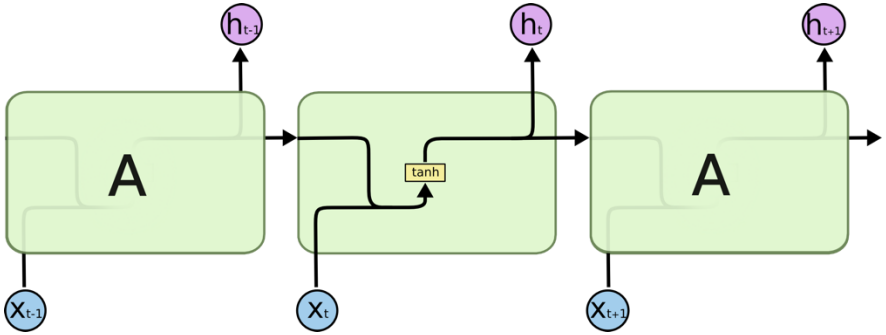
At each time step t, given input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

Where:

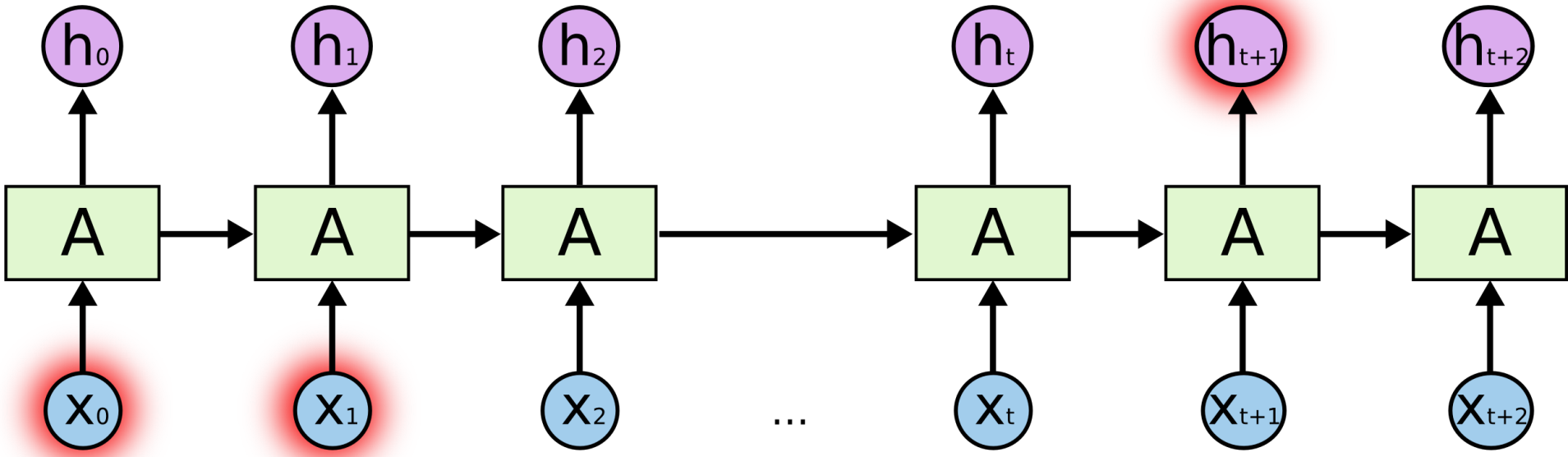
- $\mathbf{x}_t \in \mathbb{R}^d$ is the input at time t
- $\mathbf{h}_t \in \mathbb{R}^n$ is the hidden state
- $\mathbf{y}_t$ is the output
- $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{hy}$ are **shared weight matrices** (same at every time step)



The key property: **the same weights are applied at every time step**. This is analogous to how CNN filters are shared across spatial locations — here, the weights are shared across **time**.

Vanilla RNNs

- The disadvantage of a vanilla RNN is that as the timesteps increase, it can't derive context from timesteps that are too far behind



Long Short Term Memory networks (LSTMs) were introduced to deal with this long-term dependency problem (Hocreiter & Schmidhuber, 1997)

3. The Vanishing Gradient Problem and Backpropagation Through Time

How BPTT Works

1. Unroll the RNN for T time steps
2. Forward pass: compute all hidden states and outputs
3. Compute loss at each time step
4. Backward pass: propagate gradients back through time
5. Sum gradients across all time steps
6. Update shared weights W_hh, W_xh, W_hy

The Gradient Problem

Gradient of loss w.r.t. early hidden states requires multiplying a chain of many Jacobian matrices:

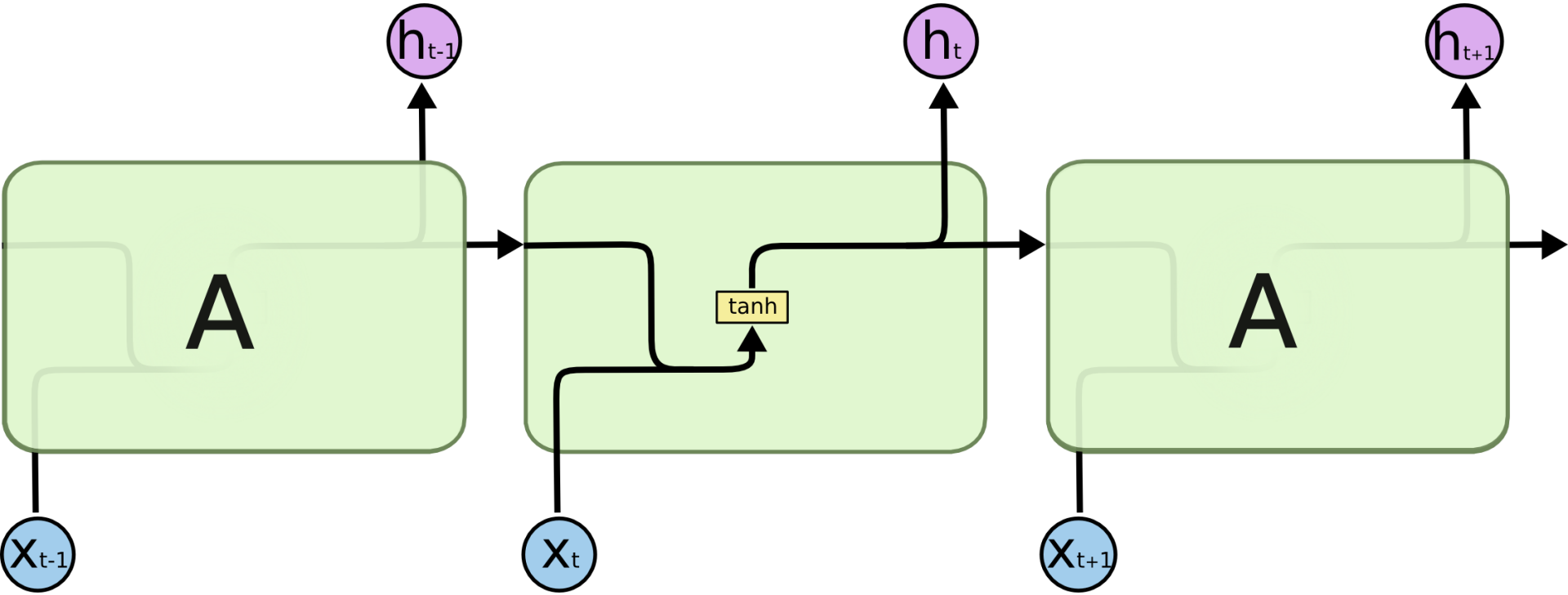
$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_1} = \prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \prod_{t=2}^{T} \mathrm{diag}(\tanh'(\cdot)) \cdot \mathbf{W}_{hh}$$

Since $|\tanh'(x)| \leq 1$, the product of many values < 1 shrinks exponentially:

Vanishing Gradients: If eigenvalues of $W_{hh} < 1$, gradients shrink exponentially — network forgets distant past

Exploding Gradients: If eigenvalues of $W_{hh} > 1$, gradients grow exponentially — training becomes unstable. (Gradient clipping helps)

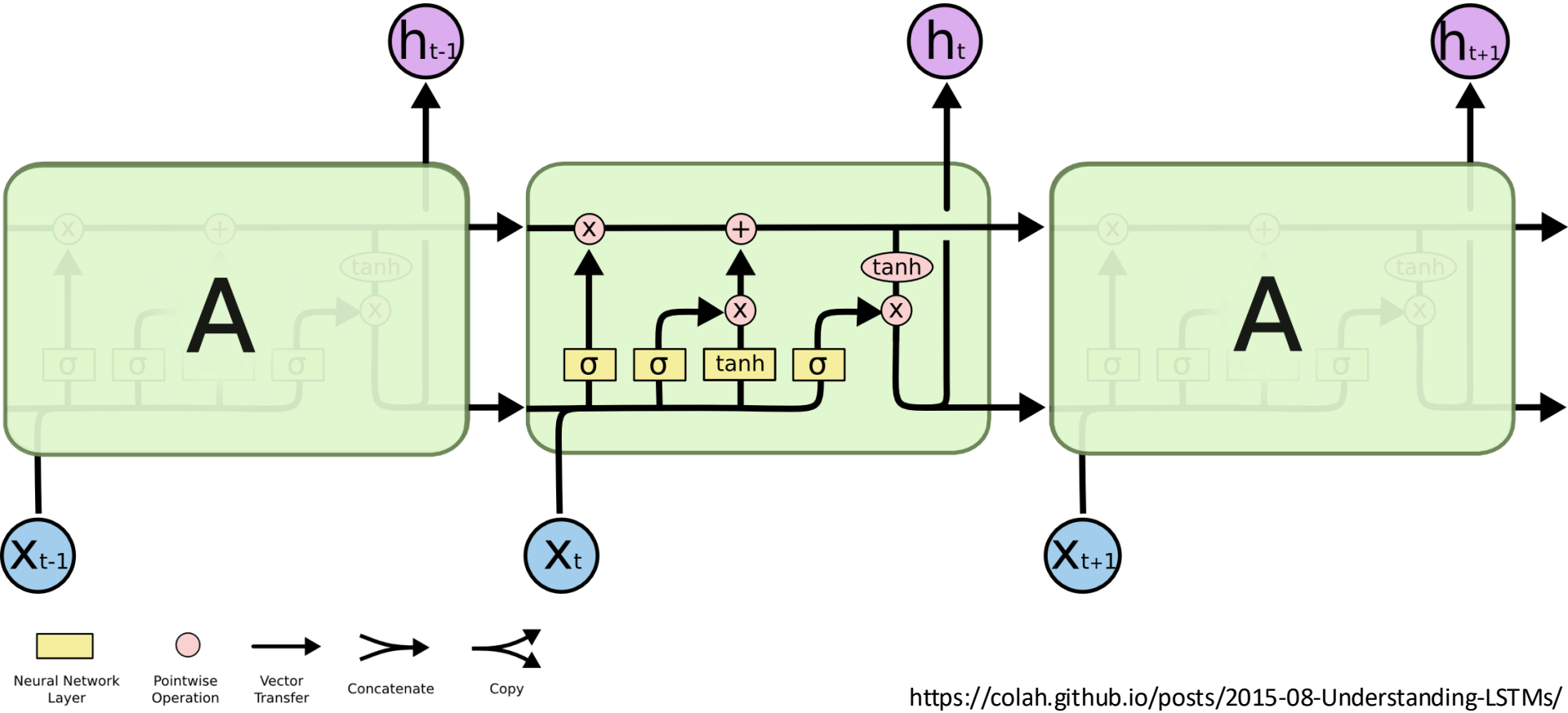
Vanilla RNN



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory (LSTM)

Hochreiter & Schmidhuber, 1997 — The architecture that solved vanishing gradients



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

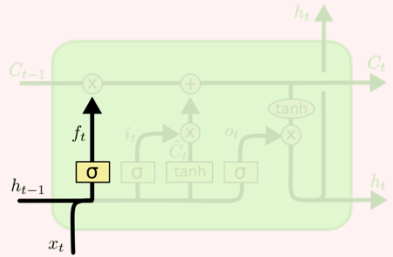
4. Long Short-Term Memory (LSTM)

Hochreiter & Schmidhuber, 1997 — The architecture that solved vanishing gradients

Forget Gate

Decides what information to discard from the cell state. Outputs values between 0 (forget) and 1 (keep).

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

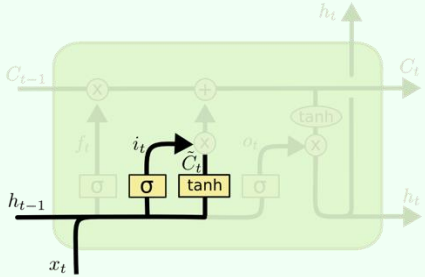


Input Gate

Decides what new information to store. Creates candidate values scaled by gate activation.

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

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

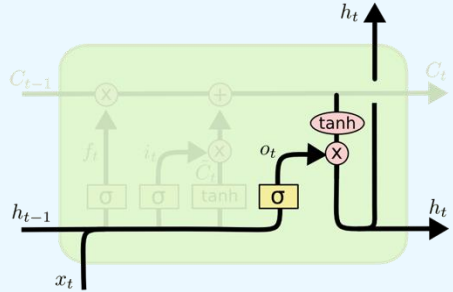


Output Gate

Controls what part of the cell state becomes the hidden state output.

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

$$h_t = o_t * \tanh\left(C_t\right)$$




```
Cell State Update:   C(t) = f(t) * C(t-1) + i(t) * C̃(t)    ←  The 'conveyor belt' that allows gradients to flow unimpeded
```

Gated Recurrent Unit (GRU)

Cho et al., 2014 — A simpler alternative to LSTM with comparable performance

Update Gate (z)

Controls how much of the previous hidden state to keep vs. replace with new candidate. Combines LSTM's forget + input gates into one.

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

Reset Gate (r)

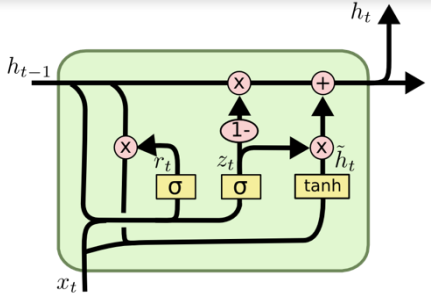
Controls how much of the previous hidden state is used when computing the new candidate. Allows the network to 'forget' past information.

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

Hidden State Update

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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



GRU vs LSTM: 2 gates instead of 3 · No separate cell state · ~25% fewer parameters · Often matches LSTM performance on shorter sequences

5. Bidirectional RNNs

Process the sequence in both forward and backward directions, capturing context from past and future.

Uni-directional

$h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4$

Information flows forward in time only

At time t, only observations ≤ t are used

Can classify at any point during the event

Natural for real-time / streaming applications

Bi-directional

$h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4$
$h_1 \leftarrow h_2 \leftarrow h_3 \leftarrow h_4$

Uses both past and future context

Each step sees the full sequence

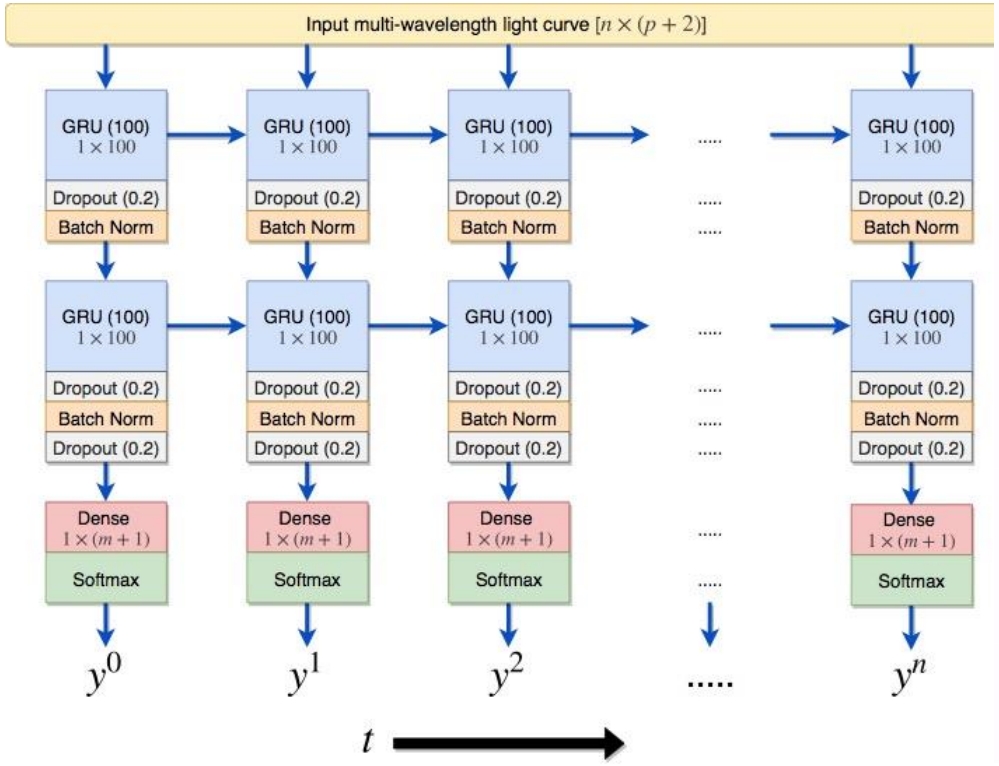
Better accuracy on complete sequences

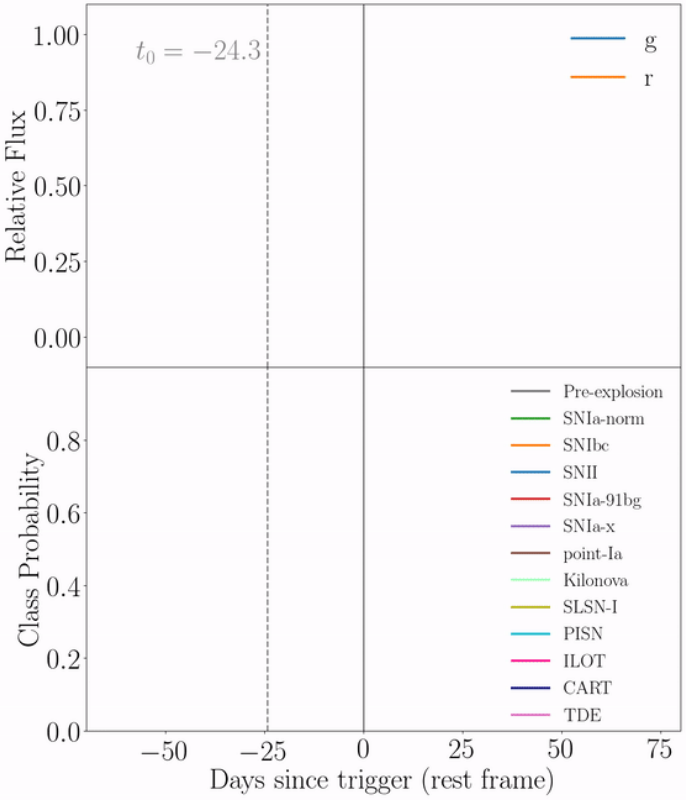
Should not be used to train real-time systems

Output: $h(t) = [\ h\rightarrow(t)\ ;\ h\leftarrow(t)\ ]$ — concatenation of both directions

Use when the full sequence is available (e.g., complete light curve). Not suitable for real-time streaming where future data is unknown.

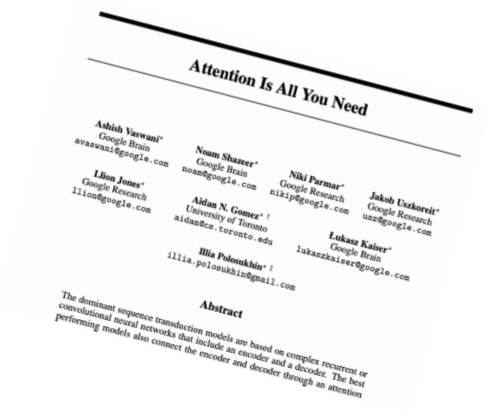
6. Training an RNN to classify supernova light curves





Did Transformers kill RNNs?

Maybe… but RNNs are still shaping new architectures



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Lukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention

Prefer RNNs when:

- Compute or memory constrained
  - RNNs are lightweight and memory friendly
  - Edge devices where latency is important
- Limited training data available
- Strong temporal locality assumptions

Were RNNs All We Needed?

Leo Feng
Mila – Université de Montréal & Borealis AI
leo.feng@mila.quebec

Frederick Tung
Borealis AI
frederick.tung@borealisai.com

Mohamed Osama Ahmed
Borealis AI
mohamed.o.ahmed@borealisai.com

Yoshua Bengio
Mila – Université de Montréal
yoshua.bengio@mila.quebec

Hossein Hajimirsadeghi
Borealis AI
hossein.hajimirsadeghi@borealisai.com

Abstract

The introduction of Transformers in 2017 reshaped the landscape of deep learning. Originally proposed for sequence modelling, Transformers have since achieved widespread success across various domains. However, the scalability limitations of Transformers—particularly with respect to sequence length—have sparked renewed interest in novel recurrent models that are parallelizable during training, offer comparable performance, and scale more effectively. In this work, we revisit sequence modelling from a historical perspective, focusing on Recurrent Neural Networks (RNNs), which dominated the field for two decades before the rise of Transformers. Specifically, we examine LSTMs (1997) and GRUs (2014). We demonstrate that by simplifying these models, we can derive minimal versions (minLSTMs and minGRUs) that (1) use fewer parameters than their traditional counterparts, (2) are fully parallelizable during training, and (3) achieve surprisingly competitive performance on a range of tasks, rivalling recent models including Transformers.

RNNs vs Transformers

Aspect	RNNs (LSTM/GRU)	Transformers
Sequence Processing	Sequential (step-by-step)	Parallel (all at once)
Long-Range Deps	Struggles with very long sequences	Handles well via attention
Computation	O(T) sequential steps	$O(T^2)$ but parallelizable
Memory Efficiency	Constant memory per step	Grows quadratically with T
Training Speed	Slow (can't parallelize time)	Fast (fully parallelizable)
Inductive Bias	Strong temporal bias built in	Learns position from data
Parameter Count	Compact, weight sharing	Typically larger
Variable Length	Naturally handles any length	Fixed context window (handles variable length input with padding and masking)

When to Choose RNNs Over Transformers

RNNs Shine When...

✓ Limited training data — strong inductive bias helps generalization with fewer examples

✓ Resource-constrained environments — embedded systems, real-time pipelines, edge computing

✓ Streaming / online data — process one observation at a time without storing full sequence

✓ Clear causal/temporal structure — physics-based time-series where order matters intrinsically

✓ Small to medium sequence lengths — competitive performance with far fewer parameters

RNN Pitfalls

✗ Slow training — sequential nature prevents GPU parallelism across time steps

✗ Vanishing gradients still possible — even LSTMs struggle with sequences > 500–1000 steps

✗ Difficulty with global patterns — hard to learn dependencies spanning entire sequence

✗ No pre-trained foundation models — unlike Transformers, few large pre-trained RNNs exist

✗ Harder to interpret — hidden states are opaque; attention maps are more interpretable

✗ Being supplanted — Transformers + linear attention variants closing the efficiency gap

RNN Applications in Astronomy

Light Curve Classification

RAPID, SuperNNova, PELICAN

Classify variable stars, supernovae types, and transient events from photometric time-series. RNNs handle irregular cadences with time-aware modifications.

Anomaly Detection in Surveys

Real-time broker systems

Flag unusual or novel objects in real-time survey streams (ZTF, LSST/Rubin). RNNs process observations sequentially as they arrive.

Radial Velocity Time Series

Exoplanet confirmation pipelines

Model stellar RV variations to disentangle planetary signals from stellar activity in exoplanet detection.

Spectral Sequence Analysis

Spectroscopic pipelines

Process wavelength-ordered spectra for redshift estimation, line identification, and stellar parameter estimation.

Gravitational Wave Detection

GW trigger classification

Identify and classify gravitational wave signals in noisy detector data streams from LIGO/Virgo.

The Modern Landscape: Beyond Classic RNNs

New architectures inherit RNN principles while addressing their limitations

State Space Models (Mamba)

Structured state space models (S4, Mamba) combine the recurrent formulation with efficient parallel training. Linear complexity O(T) at inference with O(T log T) training. Selective state spaces allow input-dependent gating, similar to LSTM gates but continuous.

Linear Attention / RWKV

Reformulate attention to be computable as an RNN at inference time. RWKV achieves Transformer-level quality with RNN-level inference cost. Bridges the gap between the two paradigms.

Temporal Fusion Transformers

Combine LSTM encoders for local temporal processing with Transformer attention for capturing long-range patterns. State-of-the-art for multi-horizon time-series forecasting.

RNNs aren’t quite dead and still help shape new architectures. Understanding RNN fundamentals is essential for grasping these modern architectures.

Key Takeaways

1. RNNs process sequential data naturally — hidden states provide built-in memory for temporal patterns, ideal for light curves.

2. Gating solves the vanishing gradient — LSTMs and GRUs use learned gates to control information flow over long sequences.

3. RNNs complement Transformers — RNNs excel with limited data and streaming; Transformers dominate with large datasets.

4. RAPID showcases RNNs in astronomy — GRU-based real-time transient classification for ZTF and LSST alert streams.

5. The landscape is evolving — state-space models (Mamba) and hybrid architectures blend the best of both worlds.