RNN & LSTM & GRU - Gated Architectures and Comparative Modeling

Course:
INFO-6152 Deep Learning with Tensorflow & Keras 2

FANSHAWE

Developed by:
Mohammad Noorchenarboo

November 3, 2025

Current Section

Why Do We Need RNNs?

Motivating Problem: How can we model data where the current input depends on previous inputs?
Traditional feedforward networks assume inputs are independent of one another. But what if we're modeling a sequence?

Example Use Cases

- Predicting the next word in a sentence
- Forecasting stock prices based on past prices
- Classifying sentiment from a sequence of words
- Modeling sensor data over time

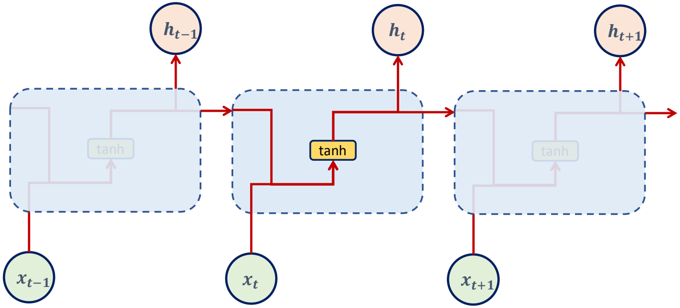
Warning

Feedforward networks flatten temporal sequences and lose ordering information. This leads to poor performance on time-dependent data.

Why Do We Need RNNs?

Standard NN: $\hat{y} = f(Wx + b)$ (no memory of previous inputs)

RNN: $h_t = f(Wx_t + Uh_{t-1} + b)$ (maintains memory)



RNN Structure: Memory and Recurrence

Core Concept: RNNs maintain a “memory” of previous time steps through a hidden state.

$$h_t = \tanh(Wx_t + Uh_{t-1} + b)$$

Term Definitions

- x_t – input vector at time step t
- h_t – hidden state at time step t
- W, U – weight matrices for input and recurrent connections
- b – bias vector

Interactive example $\rightarrow$ link

RNN Structure: Memory and Recurrence

Step-by-Step Numerical Example: 1D Time Series

Assume a time series $[0.1, 0.2, 0.3]$ and RNN parameters:

$$W = 2.0, \quad U = 0.5, \quad b = 0.1, \quad h_0 = 0$$

Step 1:

$$h_1 = \tanh(2.0 \cdot 0.1 + 0.5 \cdot 0 + 0.1) = \tanh(0.3) \approx 0.291$$

Step 2:

$$h_2 = \tanh(2.0 \cdot 0.2 + 0.5 \cdot 0.291 + 0.1) = \tanh(0.748) \approx 0.634$$

Step 3:

$$h_3 = \tanh(2.0 \cdot 0.3 + 0.5 \cdot 0.634 + 0.1) = \tanh(1.217) \approx 0.839$$

What If We Replace tanh With Other Activations?

Motivating Question: What are the consequences of using `sigmoid` or `ReLU` instead of tanh in RNN hidden states?

$$h_t = \text{activation}(Wx_t + Uh_{t-1} + b)$$

Why tanh is Common

- Outputs are centered at 0 (unlike sigmoid)
- Keeps values bounded in $(-1, 1)$
- Smooth gradient across most of the input range

Numerical Comparison (Input $z = 2.0$)

$$\tanh(2.0) \approx 0.964 \qquad \text{sigmoid}(2.0) \approx 0.88 \qquad \text{ReLU}(2.0) = 2.0$$

$$\tanh(-2.0) \approx -0.964 \qquad \text{sigmoid}(-2.0) \approx 0.12 \qquad \text{ReLU}(-2.0) = 0$$

What If We Replace tanh With Other Activations?

Weaknesses of Other Activations

Sigmoid:

- Output range is $(0, 1)$ – not zero-centered
- Leads to biased gradients that affect weight updates Activation outputs: [0.12, 0.5, 0.88] All positive $\rightarrow$ gradient flows always in same direction $\rightarrow$ biased

ReLU:

- Unbounded output – risks exploding activations
- Zero gradient for negative inputs – may cause “dead neurons”

What If We Replace tanh With Other Activations?

Gradient Behavior Warning

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z) \quad \text{(smooth and bounded)}$$

$$\frac{d}{dz}\text{ReLU}(z) = \begin{cases} 1 & z > 0 \\ 0 & z \le 0 \end{cases}$$

Sigmoid and tanh both suffer from vanishing gradients, but ReLU can completely cut off learning in some neurons.

Summary: Recurrent Neural Networks (RNNs)

Concept	**Details**
Problem Addressed	Sequences where current output depends on previous inputs
Mathematical Formula	$h_t = \tanh(Wx_t + Uh_{t-1} + b)$
State Representation	Hidden state h_t captures temporal dependencies
Use Cases	Text, time series, signals, sequence modeling tasks
Numerical Example	Time series $[0.1, 0.2, 0.3]$ with $W = 2.0$, $U = 0.5$, $b = 0.1$ shows gradual state evolution

Current Section

Why Do We Need LSTM Networks?

Motivating Question: How can a neural network remember information from 50 or 100 time steps ago?

$$\text{RNN Gradient: } \frac{\partial \mathscr{L}}{\partial W} = \prod_{t=1}^{T} \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial \mathscr{L}}{\partial h_T}$$

The RNN Limitation

- Gradients shrink across time steps.
- Information from the distant past is overwritten.
- Long-term dependencies are hard to capture.

Interactive example $\rightarrow$ link

Why Do We Need LSTM Networks?

LSTM to the Rescue

LSTM networks introduce memory cells and gates to:

- Retain important information over time.
- Forget irrelevant parts.
- Selectively expose memory to output.

Motivating Example

Understanding a sentence like: “The boy who wore the red hat went to the park. He was happy.” requires linking “He” to “boy” many words earlier.

Concept	**RNN vs LSTM**
Memory Retention	LSTM can hold long-term context via cell state
Control Mechanism	LSTM uses gates to manage memory flow
Output Flexibility	LSTM can output selectively at each step

LSTM Memory: Cell State vs Hidden State

$$\mathbf{C}_t = f_t \cdot \mathbf{C}_{t-1} + i_t \cdot \tilde{C}_t, \quad \mathbf{h}_t = o_t \cdot \mathtt{tanh}(\mathbf{C}_t)$$

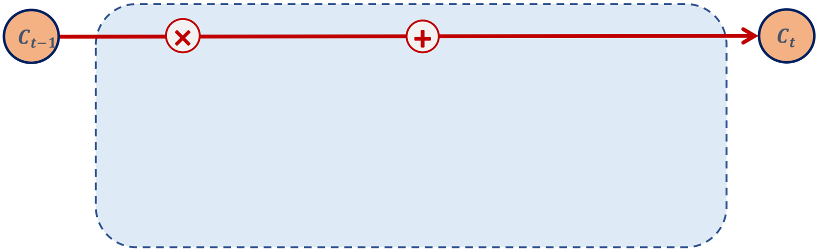
Memory Roles: A Mathematical View

- $\mathbf{C}_t$ accumulates past knowledge through additive updates. Since it avoids repeated nonlinear squashing (like `tanh`), it can carry gradients over long durations. This makes it suitable for **long-term memory**.
- $\mathbf{h}_t$ is recomputed every time step and includes a nonlinearity via $\mathtt{tanh}(\mathbf{C}_t)$, followed by modulation from o_t. It is more sensitive to recent inputs and serves as **short-term memory**.

Clarifying the Difference

While $\mathbf{h}_t$ is what gets passed to the next layer or output at time t, it is $\mathbf{C}_t$ that *remembers* the accumulation of information across time.

LSTM Memory: Cell State vs Hidden State



LSTM Memory: Cell State vs Hidden State

Numerical Example: Partial Memory Retention

Let:

$$\mathbf{C}_{t-1} = 0.9, \quad f_t = 0.4, \quad i_t = 0.5, \quad \tilde{C}_t = 0.3$$

Then:

$$\mathbf{C}_t = 0.4 \cdot 0.9 + 0.5 \cdot 0.3 = 0.36 + 0.15 = 0.51$$

$$\text{Then } \mathbf{h}_t = o_t \cdot \tanh(0.51)$$

Interpretation: The cell keeps 40% of the old memory and adds new information scaled by 50%. The hidden state is derived only after computing the updated memory.

Forget Gate: Erasing Old Memory

Key Idea: The forget gate determines how much of the previous cell state $\mathbf{C}_{t-1}$ should be retained.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \quad 0 \leq f_t \leq 1$$

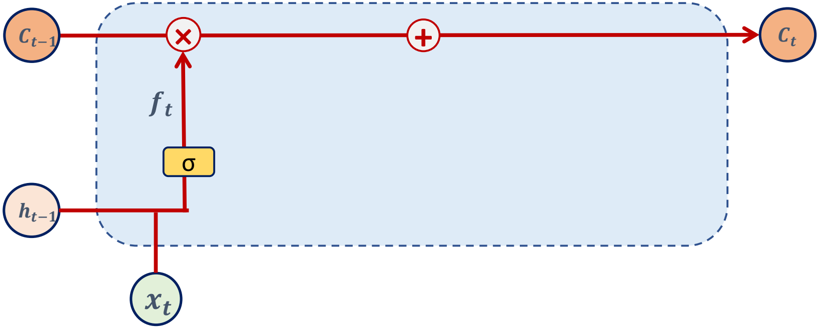
Forget Gate Purpose

The forget gate f_t acts as a memory filter:

- $f_t = 1$: retain 100% of the previous memory.
- $f_t = 0$: discard all previous memory.
- $0 < f_t < 1$: keep a weighted portion of the past.

It enables selective forgetting of long-term memory content.

Forget Gate: Erasing Old Memory



Forget Gate: Erasing Old Memory

Numerical Example: Keeping 40% of Old Memory

Let:

$$\mathbf{C}_{t-1} = 1.2, \quad f_t = 0.4$$

Then:

$$\mathbf{C}_t = f_t \cdot \mathbf{C}_{t-1} = 0.4 \cdot 1.2 = 0.48$$

Interpretation: Only 40% of the previous memory is preserved. The rest is forgotten, making space for new information to be written by the input gate.

Interpretation

The forget gate dynamically removes outdated or irrelevant information. This ensures that long-term memory $\mathbf{C}_t$ remains focused and relevant over time, without overwhelming the network with old data.

Input Gate: Storing New Information

Key Idea: The input gate determines how much of the new candidate memory should be added to the current cell state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \quad \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

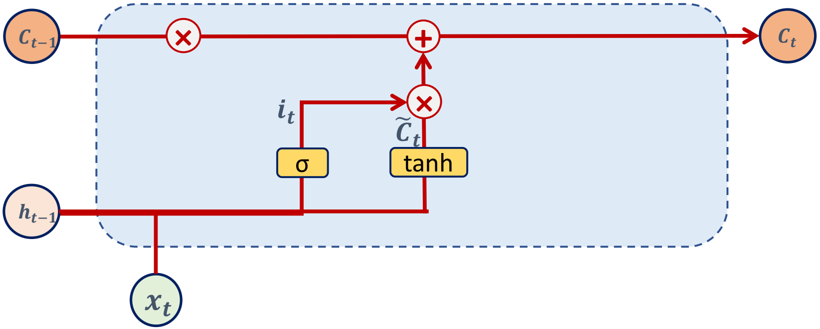
$$\mathbf{C}_t = f_t \cdot \mathbf{C}_{t-1} + i_t \cdot \tilde{C}_t$$

Input Gate Purpose

- i_t determines how much of the candidate memory $\tilde{C}_t$ is allowed into the cell state.
- $\tilde{C}_t$ is the newly computed content from the current input and previous hidden state.

This mechanism ensures that only valuable new information is written into long-term memory.

Input Gate: Storing New Information



Input Gate: Storing New Information

Numerical Example: Adding New Memory

Let:

$$\mathbf{C}_{t-1} = 0.9, \quad f_t = 0.5, \quad i_t = 0.3, \quad \tilde{C}_t = 0.6$$

Then:

$$\mathbf{C}_t = 0.5 \cdot 0.9 + 0.3 \cdot 0.6 = 0.45 + 0.18 = 0.63$$

Interpretation: Half of the old memory is retained. A small portion (30%) of the candidate value 0.6 is written into memory, raising the total.

Caution

The input gate does not control erasure of memory; it only adds new content. Forgetting is handled independently by f_t.

Output Gate: Producing Final Output

Key Idea: The output gate decides what part of the internal memory should influence the next step or be passed to the next layer.

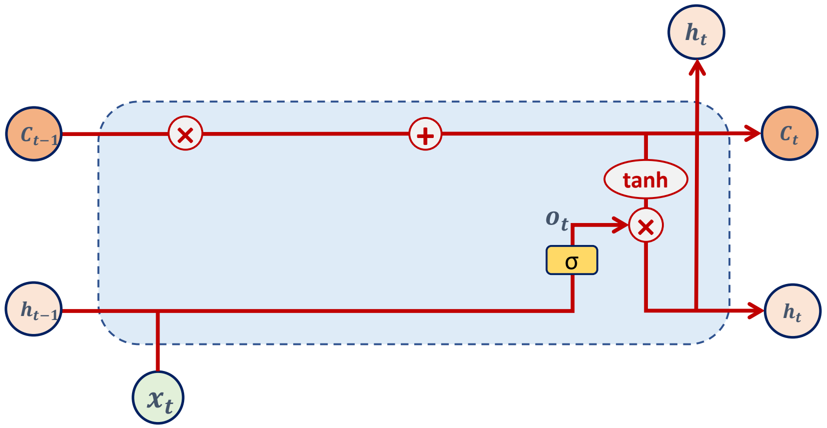
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o), \quad h_t = o_t \cdot \tanh(\mathbf{C}_t)$$

Output Gate Purpose

- Filters the updated cell state through tanh to constrain values between $[-1, 1]$.
- Scales this output using o_t to determine the visible short-term response (h_t).

This ensures the model doesn't expose the full internal memory unless needed.

Output Gate: Producing Final Output



Output Gate: Producing Final Output

Numerical Example: Controlled Exposure

Let:

$$\mathbf{C}_t = 0.8, \quad o_t = 0.5, \quad \texttt{tanh}(0.8) \approx 0.664$$

$$h_t = 0.5 \cdot 0.664 = 0.332$$

Interpretation: The internal memory content (0.8) is first squashed to 0.664. Only 50% of that value is passed as output, making $h_t = 0.332$.

Key Insight

The output gate doesn't change the memory $\mathbf{C}_t$ itself. It only determines how much of it is visible externally at this time step.

Full LSTM Update: End-to-End Numerical Pass

Let the values at time t be:

$$\mathbf{C}_{t-1} = 1.0$$
$$f_t = 0.4, \quad i_t = 0.6, \quad \tilde{C}_t = 0.5, \quad o_t = 0.7$$

Step 1: Update Cell State

$$\mathbf{C}_t = f_t \cdot \mathbf{C}_{t-1} + i_t \cdot \tilde{C}_t = 0.4 \cdot 1.0 + 0.6 \cdot 0.5 = 0.4 + 0.3 = 0.7$$

Step 2: Compute Hidden State

$$h_t = o_t \cdot \tanh(\mathbf{C}_t) = 0.7 \cdot \tanh(0.7) \approx 0.7 \cdot 0.604 = 0.423$$

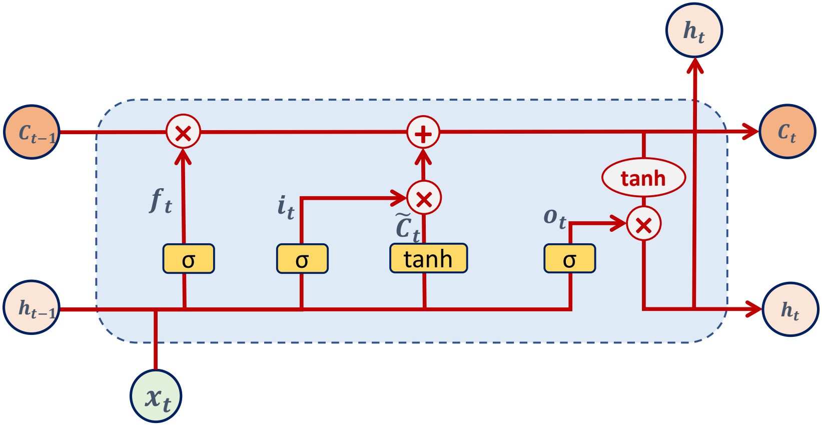
Result

Final updated states:

$$\mathbf{C}_t = 0.7, \quad h_t \approx 0.423$$

Interpretation: The forget gate retains 40% of the old memory. The input gate contributes new information, and the output gate controls the visibility of the memory. The hidden state h_t carries this filtered, context-aware signal forward.

Full LSTM Update: End-to-End Numerical Pass



Short-Term vs Long-Term Memory in LSTM

Mathematical Foundations:

$$\mathbf{C}_t = f_t \cdot \mathbf{C}_{t-1} + i_t \cdot \tilde{C}_t, \quad \mathbf{h}_t = o_t \cdot \tanh(\mathbf{C}_t)$$

Long-Term Memory: $\mathbf{C}_t$

- Accumulates information gradually across time steps.
- Modified by additive updates, preserving gradient flow.
- Serves as a memory buffer spanning multiple inputs.

Short-Term Memory: $\mathbf{h}_t$

- Derived from the updated cell state $\mathbf{C}_t$ at time t.
- Multiplied by the output gate o_t to control visibility.
- Used for immediate output to the next layer or time step.

Short-Term vs Long-Term Memory in LSTM

Numerical Example

Let:

$$\mathbf{C}_t = 1.1, \quad o_t = 0.5, \quad \texttt{tanh}(1.1) \approx 0.800$$

Then:

$$\mathbf{h}_t = 0.5 \cdot 0.800 = 0.400$$

Interpretation: The long-term memory retains accumulated knowledge (1.1), while only half of its squashed form (0.400) is exposed as short-term output.

Comparison Table

Memory Type	**Purpose**
Cell State ($\mathbf{C}_t$)	Carries persistent memory across time steps
Hidden State ($\mathbf{h}_t$)	Outputs time-local representation for current step

LSTM Components Summary

Component	**Function**
Forget Gate (f_t)	Decides how much past memory to retain
Input Gate (i_t)	Determines how much new info to store
Candidate Memory ($\tilde{C}_t$)	New memory to consider adding
Output Gate (o_t)	Controls what gets output
Cell State ($\mathbf{C}_t$)	Stores long-term memory
Hidden State ($\mathbf{h}_t$)	Carries short-term, time-step-specific info

Current Section

Why GRU? A Simpler Recurrent Architecture

Motivating Question: Can we retain the benefits of LSTM while using fewer gates and simpler equations?

Key Motivation for GRU

- GRU simplifies LSTM by merging the forget and input gates into a single **update gate**.
- It removes the separate cell state; instead, the hidden state stores all memory.
- Fewer parameters, faster training, and often similar performance.

When to Consider GRU

GRUs may outperform LSTMs on smaller datasets or tasks requiring fewer long-range dependencies. However, they lack the fine-grained memory control of LSTM.

Why GRU? A Simpler Recurrent Architecture

Comparison Table: GRU vs LSTM

Component	**LSTM** vs **GRU**
Memory Storage	LSTM: Cell + Hidden states; GRU: Hidden state only
Number of Gates	LSTM: 3 gates (forget, input, output); GRU: 2 gates (update, reset)
Control Mechanism	LSTM separates read/write logic; GRU merges it
Training Speed	GRU generally trains faster

GRU Architecture Overview

GRU Hidden State Update Equation:

$$\mathbf{h}_t = (1 - z_t) \cdot \mathbf{h}_{t-1} + z_t \cdot \tilde{\mathbf{h}}_t$$

Core Components

- z_t: **Update gate** – controls how much of the past to keep.
- r_t: **Reset gate** – controls how much of the past to forget when computing new content.
- $\tilde{\mathbf{h}}_t$: **Candidate hidden state** – the new memory content.

GRU Simplifies Memory Control

There is no separate cell state. Memory is stored directly in the hidden state $\mathbf{h}_t$, simplifying backpropagation and reducing parameters.

Update Gate z_t: Balancing Past and Present

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

$$\mathbf{h}_t = (1 - z_t) \cdot \mathbf{h}_{t-1} + z_t \cdot \tilde{\mathbf{h}}_t$$

What Does the Update Gate Control?

The update gate z_t determines how much of the new candidate memory $\tilde{h}_t$ should replace the old hidden state h_{t-1}.

- $z_t \approx 0$: mostly retain past knowledge.
- $z_t \approx 1$: quickly adopt new information.

It performs the role of both the **forget** and **input** gates from LSTM in one equation.

Intuitive Analogy

If $z_t = 0$, we're listening only to past memory. If $z_t = 1$, we're fully replacing it with new information. Values in between create a **weighted mix**.

Update Gate z_t: Balancing Past and Present

Numerical Example: Gradual Memory Update

Let:

$$\mathbf{h}_{t-1} = 0.9, \quad \tilde{\mathbf{h}}_t = 0.2, \quad z_t = 0.4$$

Then:

$$\mathbf{h}_t = (1-0.4)\cdot 0.9 + 0.4\cdot 0.2 = 0.54 + 0.08 = 0.62$$

Interpretation: The GRU retains 60% of the past and adds 40% of the present. This allows the network to adapt gradually without abrupt memory shifts.

Reset Gate r_t: Controlling Past Influence

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

$$\tilde{\mathbf{h}}_t = \tanh(W \cdot [r_t \cdot h_{t-1}, x_t] + b)$$

Reset Gate Purpose

The reset gate r_t determines how much of the previous hidden state h_{t-1} should influence the **candidate hidden state** $\tilde{h}_t$.

- $r_t \approx 1$: use full memory of the past.
- $r_t \approx 0$: ignore past memory and rely on the current input only.

This enables the network to “reset” part of its memory when the context changes.

Intuitive Analogy

Think of r_t as a **context reset switch**. When encountering new topics or abrupt transitions, the GRU can “clear out” its previous thoughts by reducing r_t.

Reset Gate r_t: Controlling Past Influence

Numerical Example: Forgetting the Past

Let:

$$h_{t-1} = 0.7, \quad x_t = 0.3, \quad r_t = 0.2$$

$$r_t \cdot h_{t-1} = 0.2 \cdot 0.7 = 0.14$$

Assume:

$$W = [0.5,\ 0.5], \quad b = 0.1$$

$$\tilde{h}_t = \tanh(0.5 \cdot 0.14 + 0.5 \cdot 0.3 + 0.1) = \tanh(0.07 + 0.15 + 0.1) \approx 0.309$$

Interpretation: Only a small influence (20%) from the past is used. The reset gate favors the new input, helping the model adapt to a fresh pattern.

Full GRU Update: End-to-End Numerical Example

Given:

$$h_{t-1} = 0.8, \quad z_t = 0.3, \quad r_t = 0.5, \quad x_t = 0.4$$

$$\begin{aligned}\tilde{h}_t &= \tanh(0.5 \cdot (r_t \cdot h_{t-1}) + 0.5 \cdot x_t) \\ &= \tanh(0.5 \cdot 0.4 + 0.2) = \tanh(0.4) \approx 0.379\end{aligned}$$

Final Hidden State:

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t = 0.7 \cdot 0.8 + 0.3 \cdot 0.379 = 0.56 + 0.114 = 0.674$$

Result

Final memory at time t:

$$h_t \approx 0.674$$

Interpretation: The GRU keeps most of the past and adds a fraction of new content based on reset-modified memory.

GRU Summary: What to Remember

Key Components of GRU

- **Reset Gate** r_t – controls how much past information to use when computing new candidate memory.
- **Update Gate** z_t – controls how much of the new candidate should be used to update the hidden state.
- **Candidate Memory** $\tilde{h}_t$ – the possible update to the hidden state, created from the input and selectively reset past memory.
- **Final Hidden State** h_t – a blend of old and new, controlled entirely by z_t.

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \\
\tilde{h}_t &= \tanh(W \cdot [r_t \cdot h_{t-1}, x_t] + b) \\
h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t
\end{aligned}
$$

GRU vs LSTM: Side-by-Side Comparison

Aspect	**GRU**	**LSTM**
Memory Units	Single hidden state (h_t) stores all memory	Separate cell state (C_t) and hidden state (h_t)
Number of Gates	2 (Update, Reset)	3 (Forget, Input, Output)
Update Strategy	Blends old and new hidden state using z_t	Updates cell state additively, then exposes it via o_t
Complexity	Simpler structure, fewer parameters	More complex, more parameters
Training Speed	Generally faster to train	Slightly slower due to more operations
Long-Term Memory Control	Coarser control with merged gates	Finer control via separate forget/input/output gates
Use Cases	Good for short to mid-length sequences	Stronger for long-term dependencies

Current Section

RNN Forecasting Setup: 24 Past Hours, 12 Future Predictions

We consider a multivariate time series forecasting problem with:

- 5 input features per time step (e.g., temperature, humidity, wind speed, etc.)
- 24 past time steps as input
- Predicting the next 12 time steps (forecast horizon = 12)

Input and Output Shapes

- **Input shape for model:** $(\text{batch\_size}, 24, 5)$
- **Output shape:** $(\text{batch\_size}, 12)$

$$\text{Input sequence: } X = [x_1, x_2, \ldots, x_{24}], \quad x_t \in \mathbb{R}^5$$

$$\text{Output vector: } \hat{Y} = [\hat{y}_{25}, \hat{y}_{26}, \ldots, \hat{y}_{36}], \quad \hat{y}_t \in \mathbb{R}$$

RNN Forecasting Setup: 24 Past Hours, 12 Future Predictions

Preparing RNN Model With return_sequences=False

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

model = Sequential()
model.add(SimpleRNN(64, return_sequences=False, input_shape=(24, 5)))
model.add(Dense(12))  # Predict 12 future steps

# model.summary() output:
# Layer (type)        Output Shape         Param #
# simple_rnn (RNN)    (None, 64)           ...
# dense (Dense)       (None, 12)           ...
```

RNN Forecasting Setup: 24 Past Hours, 12 Future Predictions

Explanation Based on Formula

RNN computes hidden states as:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \quad t = 1, \ldots, 24$$

$$\hat{Y} = W_o \cdot h_{24} + b_o$$

Only the final hidden state h_{24} is used to predict the entire output horizon.

Best Practice

Use `return_sequences=False` when the output depends only on the final state of the input sequence, which is common in one-shot forecasting.

RNN Forecasting Setup: 24 Past Hours, 12 Future Predictions

Preparing RNN Model With return_sequences=True

```
model = Sequential()
model.add(SimpleRNN(64, return_sequences=True, input_shape=(24, 5)))
model.add(tf.keras.layers.Flatten())   # Shape: (None, 24*64)
model.add(Dense(128, activation='relu'))
model.add(Dense(12))  # Output 12 time steps
```

RNN Forecasting Setup: 24 Past Hours, 12 Future Predictions

Explanation Based on Formula

When `return_sequences=True`, the RNN returns:

$$[h_1, h_2, \dots, h_{24}] \in \mathbb{R}^{24\times 64}$$

These are flattened and passed into a Dense layer:

$$\hat{Y} = W_o \cdot \text{Flatten}(H) + b_o$$

This allows the model to use hidden states from **all time steps**.

Warning

Using `return_sequences=True` increases the number of parameters and may lead to overfitting if not necessary.

RNN Forecasting Setup: 24 Past Hours, 12 Future Predictions

Model Setting	**Effect on Input/Output and Prediction**
`return_sequences=False`	RNN outputs only h_{24}; Dense layer maps it to 12 future steps.
`return_sequences=True` + Flatten	RNN outputs h_1 to h_{24}; all hidden states are flattened and used for prediction.
`Dense(12)`	Directly maps either h_{24} or `Flatten`(H) to 12 future time steps.
`Input shape: (24, 5)`	Means 24 time steps and 5 features at each step.