Recurrent neural networks

* Feedforward Neural Networks (FFNNs)
  + Treat each input independently; they do not consider the order or context of inputs.
  + Works on fixed-size, static inputs.
  + Fast to train and simple.
  + Cannot model sequences or temporal dependencies.
  + Use cases: tabular data, single-image classification.
* Convolutional Neural Networks (CNNs)
  + Detects local spatial/temporal patterns using filters.
  + Good for grid-like or structured data (images, short sequences).
  + Good for short-term dependencies (e.g. local trends) but struggle with long-term dependencies such as yearly seasonality.
  + Highly parallelizable
  + Use cases: images, short-term time series, audio feature extraction.
* Recurrent Neural Networks (RNNs)
  + Maintains a hidden state (memory) that evolves over time allows predictions to depend on the history of inputs seen so far
  + Designed for sequential and time-dependent data.
  + Harder to parallelize; vanilla RNNs suffer from vanishing/exploding gradients.
  + Use cases: language modeling, translation, speech recognition, forecasting, anomaly detection.

# Recurrent Neural Networks (RNNs)

## Mathematical Foundation

The core RNN equations define how information flows through time:

* : Input-to-hidden weight matrix - determines how the current input affects the hidden state.
* : Hidden-to-hidden weight matrix - controls how information from the past is retained (“memory”).
* : Hidden-to-output weight matrix - converts the hidden state to output/predictions
* : Activation function (typically tanh or ReLU)
* : Output activation (depends on task: linear for regression, softmax for classification)

Forecasting Convention: For time series prediction, we use to predict the next value in the sequence.

### Single RNN Cell Architecture

The internal structure of a single RNN cell at time step t:

knitr::include\_graphics("rnn\_block.png")

* Input: (current time step input)
* Previous hidden state: (memory from previous time step)
* Current hidden state: (output of current cell)
* Final output: (prediction or output at time t)

Operations:

1. : Input transformation - the input is multiplied by weight matrix
2. : Hidden state transformation - previous hidden state is multiplied by recurrent weight matrix
3. : Bias addition - bias term is added to the sum
4. tanh: Activation function applied to create new hidden state
5. : Output transformation - hidden state is transformed to produce output

Mathematical representation:

The gray box emphasizes the core recurrent computation that creates the memory mechanism.

### RNN block unfolded through time

RNN block unfolded through time, revealing the sequential computation:

knitr::include\_graphics("rnn\_unfold.png")

Left side (Folded):

* Shows the compact representation with the recurrent loop
* The circular arrow represents the temporal connection
* Parameters U, V, W represent the weight matrices

Right side (Unfolded):

* Shows the same network expanded across time steps: t−1, t, t+1, etc.
* Each blue box represents the same RNN cell applied at different time steps
* Key insight: The same parameters (U, V, W) are shared across all time steps

Information flow:

* Horizontal arrows: Hidden states flowing forward in time
* Upward arrows: Outputs at each time step
* Input connections: Each time step receives its own input

Why unfolding matters:

* Training: Backpropagation through time (BPTT) operates on this unfolded graph
* Memory: Shows how information can flow across multiple time steps
* Parameter sharing: Demonstrates that the same weights are used at every time step
* Sequential processing: Illustrates the inherent sequential nature of RNN computation

## Hidden-to-Hidden Matrix

The eigenvalues of control how past states affect over time.

* Small eigenvalues () memory fades quickly; past inputs have little influence.
* Eigenvalues near 1 information persists, like a clock or moving average.
* Rotational matrices (eigenvalues on unit circle) produce oscillations or periodic patterns.
* Large eigenvalues () risk of exploding states, causing instability in training.
* Properly chosen can capture temporal patterns: decays, cycles, or oscillations.

## Simple numerical example

* State update (simplified, no bias or activation):
* One step ahead forecast:
* Consider scalar input/output , , hidden state :
* Matrix sizes: is , is , is .

## (a) Weighted Moving Average via RNN

* (exponential smoothing with = 0.25)
* (unused dimension)
* (direct readout of smoothed state)

This creates a classic exponential moving average with decay factor 0.75, commonly used in financial analysis and signal processing.

Input sequence , with initial state .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Interpretation |
| 1 | 0 |  |  | No signal yet |
| 2 | 0 |  |  | Still quiet |
| 3 | 0 |  |  | Baseline |
| 4 | 5 |  |  | **Spike detected, predicting echo** |
| 5 | 0 |  |  | Exponential decay begins |
| 6 | 0 |  |  | Continued decay |
| 7 | 0 |  |  | Memory fading |
| 8 | 5 |  |  | **Second spike, stronger prediction** |
| 9 | 0 |  |  | New decay cycle |

The weighted moving average provides a reactive forecasting approach, predicting based on recent weighted history.

## (b) 4-Step Cycle

Rotates the hidden state each step generates periodic behavior.

Inputs . Initialize .

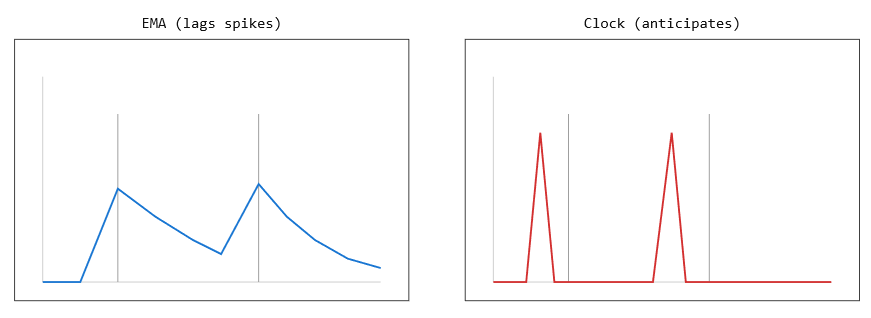
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Phase |
| 1 | 0 |  |  | 90° |
| 2 | 0 |  |  | 180° |
| **3** | 0 |  |  | **270° - Prediction phase** |
| 4 | 5 |  |  | 0° (reset + input) |
| 5 | 0 |  |  | 90° |
| 6 | 0 |  |  | 180° |
| **7** | 0 |  |  | **270° - Prediction phase** |
| 8 | 5 |  |  | 0° (reset + input) |

* rotates by each step (period 4).
* Spike inputs to **re-sync** phase
* Output reads pre-spike phase
* Linear readout gives a sinusoidal score; gates/thresholds would help remove the bottom trough.

## (c) 12-Step Cycle

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| t | phase |  |  |  |
| 0 |  | [1.000, 0.000] | 0.866 | 0.0 |
| 1 |  | [0.866, 0.500] | 0.500 | 0.0 |
| 2 |  | [0.500, 0.866] | 0.000 | 0.0 |
| 3 |  | [0.000, 1.000] | −0.500 | 0.0 |
| 4 |  | [−0.500, 0.866] | −0.866 | 0.0 |
| 5 |  | [−0.866, 0.500] | −1.000 | 0.0 |
| 6 |  | [−1.000, 0.000] | −0.866 | 0.0 |
| 7 |  | [−0.866, −0.500] | −0.500 | 0.0 |
| 8 |  | [−0.500, −0.866] | 0.000 | 0.0 |
| 9 |  | [0.000, −1.000] | 0.500 | 0.0 |
| 10 |  | [0.500, −0.866] | 0.866 | 0.0 |
| 11 |  | [**0.866, −0.500**] | **1.000** | **5.0** |
| 12 |  | [1.000, 0.000] | 0.866 | 0.0 |

* The hidden state rotates steadily around the unit circle.
* The raw score is just a cosine wave, peaking when reaches the pre-spike phase (330°).
* The nonlinearity converts this smooth score into a sharp forecast spike at .
* At , the observed spike resets the hidden state to , ready for the next cycle.



## EMA vs Clock

* EMA captures decaying average of past inputs.
* Clock/oscillator approach anticipates periodic events by tracking phase.

# Training RNNs

## Backpropagation Through Time (BPTT)

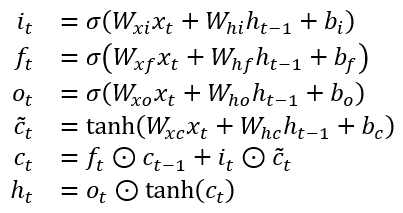
Idea: Unroll the RNN across all timesteps and apply standard backpropagation.

For sequence of length :

loss ,

* controls gradient flow across time.
* Over steps, gradients accumulate products .
* If , then across steps the product scales as .
* Gives rise to problem of exploding or vanishing gradients
* Vanishing: , so early timesteps contribute almost nothing.
* Exploding: , so unstable updates.
* Other kinds of RNN units (LSTM/GRU) help avoid this by “gating”

## LSTM

Input gate , forget gate , output gate .

An LSTM has **two hidden vectors** at each step:

* : the hidden state / output
* : the cell state / long-term memory track
* controls how much of is allowed into .
* controls how much of is retained vs. discarded.
* controls how much of is retained.

### The Forget Gate

controls how much of is carried forward.

* : how current input influences forgetting.
* : how past hidden state influences forgetting.
* squashes values into .
* : keep the -th cell’s past memory.
* : forget (wipe out) the -th cell’s past memory.

### The Input Gate

controls how much of to admit.

* : decide where new info should be written.
* : transform input and hidden state into a candidate content vector.

### Updating the Cell State

Now combine forget and input:

This equation is the core of the LSTM:

* carries over past memory where says so.
* adds new content where allows.
* Updating is updated by addition rather than full replacement avoids vanishing/exploding gradients.

### The Output Gate

decides how much of is exposed.

* squashes the cell state to before exposing.
* is then fed forward to the next step.

## Summary

At each time step:

1. Forget gate () decides what old info to erase.
2. Input gate ( + ) decides what new info to add.
3. Cell state update combines old and new: .
4. Output gate () decides what part of becomes visible in .

## Gated Recurrent Units (GRUs)

* one hidden vector (no separate cell state).
* “reset gate” and “update gate” .
* blends old vs new (acts like / combo).
* If : keep old .
* If : replace with candidate .

### The Update Gate

* : how current input influences whether to keep old memory.
* : how past hidden state influences this decision.
* gives values in .
* : keep the -th unit from the past.
* : replace it with a new candidate.

### The Reset Gate

is the **new candidate hidden state** the GRU could switch to:

with

controlling how much the past hidden state contributes when making a new candidate.

## The Reset Gate

* : decide where new info should be written.
* : transform input and (reset version of last) hidden state into a candidate content vector.
* : ignore the -th unit of when building the candidate.
* : fully use the past hidden info.

## Summary

* Update gate weighs old memory vs. new memory
* Reset gate weighs past info when making a new memory
* is the proposed new memory

# RNN Time Series Forecasting with Keras

This code demonstrates how to build and compare different RNN architectures (SimpleRNN, LSTM, GRU) for time series forecasting using Keras in R. The task involves predicting periodic spikes in a synthetic time series.

## 1. Setup and Data Generation

library(keras3)  
set.seed(42)  
N <- 600  
period <- 12  
  
spikes <- rep(c(rep(0, period-1), 5), length.out = N)  
noise <- rnorm(N, sd = 0.2)  
series <- spikes + noise

Purpose: Create a synthetic periodic time series with predictable patterns.

Components:

* N <- 600: Generate 600 time points
* period <- 12: Set periodicity to 12 time steps (like monthly data)
* spikes <- rep(c(rep(0, period-1), 5), length.out = N):
  + Creates pattern: [0,0,0,0,0,0,0,0,0,0,0,5] repeated for 600 points
  + Spike of magnitude 5 every 12th time step
* noise <- rnorm(N, sd = 0.2): Add Gaussian noise with standard deviation 0.2
* series <- spikes + noise: Combine clean periodic signal with noise

Result: A noisy time series with spikes at positions 12, 24, 36, 48, etc.

## 2. Data Preprocessing for Supervised Learning

Twin <- 24  
X <- array(0, dim = c(N - Twin, Twin, 1))  
y <- array(0, dim = c(N - Twin, 1))  
  
for (i in seq\_len(N - Twin)) {  
 X[i,,1] <- series[i:(i+Twin-1)]  
 y[i,1] <- series[i+Twin]  
}

Purpose: Transform the time series into a supervised learning problem using a sliding window approach.

Parameters:

* Twin <- 24: Window size of 24 time steps (input sequence length)
* X: Input array with dimensions [samples, timesteps, features]
  + samples: N - Twin = 576 training examples
  + timesteps: 24 (lookback window)
  + features: 1 (univariate time series)
* y: Output array with dimensions [samples, outputs]

Sliding Window Logic:

* For each position i, use 24 consecutive values as input: series[i:(i+23)]
* Predict the next value: series[i+24]
* Example:
  + Sample 1: X = series[1:24], y = series[25]
  + Sample 2: X = series[2:25], y = series[26]
  + And so on…

Result: 576 input-output pairs suitable for training RNNs.

## 3. Model Architecture 1: SimpleRNN (EMA-like Baseline)

inputs <- layer\_input(shape = c(Twin, 1))  
rnn <- layer\_simple\_rnn(inputs, units = 2, activation = "tanh")  
outputs <- layer\_dense(rnn, units = 1, activation = "linear")  
model\_ema <- keras\_model(inputs, outputs)

Architecture:

* Input Layer: Accepts sequences of shape (24, 1)
* SimpleRNN Layer:
  + units = 2: Only 2 hidden units (minimal capacity)
  + activation = "tanh": Standard RNN activation
  + return\_sequences = FALSE (default): Returns only the last hidden state
* Dense Output Layer:
  + units = 1: Single output for next value prediction
  + activation = "linear": No activation for regression

Purpose: Serves as a baseline that mimics exponential moving average behavior due to limited capacity.

## 4. Model Architecture 2: LSTM Forecaster

inputs <- layer\_input(shape = c(Twin, 1))  
lstm <- layer\_lstm(inputs, units = 16, return\_sequences = FALSE)  
outputs <- layer\_dense(lstm, units = 1)  
model\_lstm <- keras\_model(inputs, outputs)

Architecture:

* Input Layer: Same shape (24, 1)
* LSTM Layer:
  + units = 16: 16 LSTM cells (8x more capacity than SimpleRNN)
  + return\_sequences = FALSE: Return only final hidden state
  + Includes forget, input, and output gates for better memory control
* Dense Output Layer: Linear output for regression

Purpose: Leverage LSTM’s gating mechanisms to capture long-term dependencies and periodic patterns.

## 5. Model Architecture 3: GRU Forecaster

inputs <- layer\_input(shape = c(Twin, 1))  
gru <- layer\_gru(inputs, units = 16, return\_sequences = FALSE)  
outputs <- layer\_dense(gru, units = 1)  
model\_gru <- keras\_model(inputs, outputs)

Architecture:

* Input Layer: Same shape (24, 1)
* GRU Layer:
  + units = 16: 16 GRU cells (same capacity as LSTM)
  + return\_sequences = FALSE: Return only final hidden state
  + Uses update and reset gates (simpler than LSTM)
* Dense Output Layer: Linear output for regression

Purpose: Test whether GRU’s simpler gating mechanism can match LSTM performance with fewer parameters.

## 6. Model Compilation

model\_ema %>% compile(  
 optimizer = optimizer\_adam(learning\_rate = 1e-3, clipnorm = 1.0),  
 loss = "mse", metrics = "mae")

Training Configuration (same for all models):

* Optimizer: Adam with learning rate 0.001
* Gradient Clipping: clipnorm = 1.0 prevents exploding gradients
* Loss Function: Mean Squared Error (MSE) for regression
* Metrics: Mean Absolute Error (MAE) for interpretable performance tracking

Why Gradient Clipping? RNNs are prone to exploding gradients, especially during early training. Clipping ensures stable updates.

## 7. Model Evaluation and Visualization

pred\_ema <- model\_ema %>% predict(X)  
pred\_lstm <- model\_lstm %>% predict(X)  
pred\_gru <- model\_gru %>% predict(X)  
  
plot\_idx <- 1:200  
plot(y[plot\_idx,1], type="l", col="black", lwd=2, main="Validation forecasts")  
lines(pred\_ema[plot\_idx,1], col="steelblue", lwd=2)  
lines(pred\_lstm[plot\_idx,1], col="tomato", lwd=2)  
lines(pred\_gru[plot\_idx,1], col="darkgreen", lwd=2)  
legend("topleft", legend=c("true","EMA","LSTM","GRU"),  
 col=c("black","steelblue","tomato","darkgreen"), lwd=2, bty="n")

Anticipated Performance:

* SimpleRNN (EMA): Likely to provide smoothed, reactive forecasts
* LSTM: Should capture the 12-step periodicity and provide anticipatory predictions
* GRU: Expected to perform similarly to LSTM with potentially faster training

Key Learning Points:

1. Capacity Matters: The 2-unit SimpleRNN has limited representational power
2. Memory Mechanisms: LSTM/GRU gates help retain information about the 12-step cycle
3. Anticipatory vs. Reactive: Advanced architectures can predict spikes before they occur
4. Gradient Stability: Clipping ensures stable training across all architectures