Sequence modeling - recurrent networks

Sequential Learning Problems

A **Sequential Learning Problem** refers to any machine learning task where the data points are **ordered in time or sequence**, and where the model needs to capture dependencies between elements in that sequence.

In such problems, the current prediction or output depends not only on the current input but also on the previous inputs (and sometimes future ones).

📖 Key Characteristics

Order matters – The sequence in which data arrives is important.

Dependencies across time or position – What happens at one step affects what comes next.

Variable length sequences – Inputs or outputs can have different lengths (e.g., sentences, time series).

Context awareness – The model must remember or process past information to understand current inputs.

✅ Examples of Sequential Learning Problems

1. **Natural Language Processing (NLP)**
    - **Language modeling** → Predict the next word based on the previous words.
    - **Machine translation** → Convert sentences from one language to another.
    - **Speech recognition** → Transcribe spoken words to text.
    - **Text summarization** → Generate a summary from a long article.

2. **Time Series Forecasting**
    - Predicting stock prices, weather patterns, or energy consumption based on historical data.
3. **Video Processing**
    - Action recognition or scene understanding, where frames depend on each other.
4. **Genomics**
    - Analyzing DNA sequences to detect patterns or mutations.
5. **Reinforcement Learning**
    - The agent's actions depend on past observations and rewards.

✅ Common Challenges

Long-term dependencies → Information from much earlier in the sequence may still be relevant.

Variable sequence lengths → Sequences may not be uniform.

Noise and missing data → Real-world sequences are rarely perfect.

Exploding/vanishing gradients → Deep recurrent models struggle with long-range information flow.

✅ Models Used for Sequential Learning

1. **Recurrent Neural Networks (RNNs)**
    - Designed to process sequences step by step, passing information forward.
2. **Long Short-Term Memory (LSTM) / GRU**
    - Variants of RNNs that handle long-range dependencies better.
3. **Transformers**
    - Use self-attention mechanisms to process entire sequences in parallel while capturing global dependencies.

4. **Temporal Convolutional Networks (TCNs)**
    - Apply convolutions over sequences to model temporal patterns.

✅ When to Use Sequential Learning

When the data is time-dependent or ordered

When understanding context is necessary

For tasks like speech, text, video, and forecasting

When patterns unfold over multiple steps or positions

✅ Summary

Sequential learning problems are tasks where the order and context of data points matter, and where relationships between past, present, and future elements need to be modeled. Examples include language tasks, time series forecasting, and video analysis. Specialized models like RNNs, LSTMs, and Transformers are used to tackle these problems effectively.

Types of Connections in Sequential Learning

In sequence tasks, the relationship between inputs and outputs can vary depending on how many inputs and outputs there are in the sequence. These are described by connection patterns like **one-to-one**, **many-to-one**, **one-to-many**, and **many-to-many**.

✅ 1. One-to-One

Definition

> A single input corresponds to a single output.

Example

- **Image classification** → You give an image as input, and the model outputs one label.

Diagram

Input → Output

`x` → `y`

✅ 2. One-to-Many

Definition

A single input generates a sequence of outputs.

Example

- **Image captioning** → One image is used to generate a sequence of words describing it.

Diagram

Input → Output sequence

`x` → `y1, y2, y3, ...`

✅ 3. Many-to-One

Definition

A sequence of inputs is processed to produce a single output.

Example

- **Sentiment analysis** → A sentence with many words is processed to predict one sentiment (positive or negative).
- **Stock price prediction** → Historical prices are used to predict a future price.

Diagram

Input sequence → Output

`x1, x2, x3, ...` → `y`

✅ 4. Many-to-Many

This can be further divided based on alignment between input and output sequences:

A. Many-to-Many (Same Length)

Each input corresponds to an output at the same time step.

Example

- **Video frame labeling** → Each video frame is tagged with an action label.

Diagram

`x1, x2, x3` → `y1, y2, y3`

B. Many-to-Many (Different Lengths)

A sequence is transformed into another sequence, possibly of different lengths.

Example

- **Machine translation** → A sentence in one language is translated into another language.

Diagram

`x1, x2, x3` → `y1, y2, y3, y4`

✅ Applications by Connection Type

Connection Type	Example Use Case
One-to-One	Image classification, speech command recognition
One-to-Many	Image captioning, music generation
Many-to-One	Sentiment analysis, time series forecasting
Many-to-Many (same)	Video frame annotation, pose estimation
Many-to-Many (diff)	Machine translation, text summarization, speech-to-text

✅ Why These Connections Matter

The type of connection determines:

- The architecture you choose (RNN, Transformer, etc.)
- Whether you need sequence padding or attention mechanisms
- How data flows and how models handle dependencies

Choosing the right connection pattern helps the model capture relationships effectively and solve tasks more efficiently.

✅ Summary

1. **One-to-One** → Single input, single output → Image classification.
2. **One-to-Many** → Single input, sequence output → Image captioning.
3. **Many-to-One** → Sequence input, single output → Sentiment analysis.
4. **Many-to-Many** → Sequence input, sequence output → Machine translation, video analysis.

Understanding these connection patterns is crucial for designing models that suit the problem at hand and learning how information flows across time or space.

Recurrent Neural Network (RNN)

A **Recurrent Neural Network (RNN)** is a type of neural network designed to process **sequential data**. Unlike traditional feedforward neural networks, RNNs have **loops** in them, allowing information to be carried from one step of the sequence to the next.

This makes RNNs suitable for tasks where the current input depends on previous inputs — such as time series, language modeling, or speech recognition.

✅ Why do we need RNNs?

In many problems, data isn't independent:

- Words in a sentence depend on previous words.

- Stock prices today are related to past prices.
- A musical note depends on preceding notes.

Feedforward networks treat each input separately and don't "remember" past inputs, while RNNs maintain **context** or memory through time.

✅ Architecture

At its core, an RNN processes sequences step by step. It consists of:

1. **Input layer** – where the sequence is fed in.
2. **Recurrent hidden layer** – where the memory (hidden state) is maintained and updated.
3. **Output layer –** where predictions or representations are produced at each step or at the end.

> The recurrence is what makes it special — it's like having a "loop" that feeds the output of one step as input to the next.

Components of the Architecture

📥 Input

At each time step tt, the network takes an input vector x_t.

- For text, this could be a word embedding vector.
- For audio, this could be a frame of sound features.
- For time series, this could be the observed value.

Example:

$$x_1, x_2, x_3, \ldots, x_T$$

🧠 Hidden State

The hidden state hth_t is what stores memory.

- It depends on the previous hidden state h_{t-1} and the current input x_t.

- It is updated at every time step using a function like:

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

Where:

- W_{xh}: Weights from input to hidden layer.
- W_{hh}: Weights from the previous hidden state to the current.
- b_h: Bias term.

This update allows the network to "remember" information from past inputs.

📤 Output

The output at time step t, y_t, is computed from the hidden state:

$$y_t = W_{hy}h_t + b_y$$

Where:

- W_{hy}: Weights from hidden to output.
- b_y: Bias for the output layer.

Depending on the problem, the output could be produced at each time step or only after processing the entire sequence.

Unrolling the Architecture

One way to understand how an RNN works is to “unroll” it over time.

Example – Sequence length = 3

```
x1 → h1 → y1
      ↓
x2 → h2 → y2
      ↓
x3 → h3 → y3
```

> Each arrow represents the computation at a time step, with the hidden state flowing from one step to the next.

✅ Weight Sharing

A key aspect is that the same weights W_{xh}, W_{hh}, W_{hy} are used at every time step. This is known as **parameter sharing**, which:

- Reduces the total number of parameters.
- Allows the model to generalize across sequences of different lengths.
- Ensures consistency in learning temporal relationships.

✅ Types of RNN Architectures

1. **Many-to-Many**
   - Example: Machine translation
   - Input: Sequence of words → Output: Sequence of translated words
2. **Many-to-One**
   - Example: Sentiment analysis
   - Input: Sequence of words → Output: Single label
3. **One-to-Many**
   - Example: Image captioning
   - Input: Single image → Output: Sequence of words

✅ Activation Functions

- Typically, **tanh** or **ReLU** is used for the hidden state.
- **Softmax** is often used at the output layer when the task is classification.

✅ How Information Flows

1. **Input Encoding:** At each time step, the input x_t is transformed by weights.
2. **Memory Update:** The hidden state h_t is updated based on x_t and h_{t-1}.

3. **Output Generation:** The updated hidden state is used to produce output y_t.
4. **Feedback Loop:** The hidden state h_t feeds into the next time step's computation.

This loop is what allows the RNN to model dependencies over time.

✅ Challenges in Architecture

- **Vanishing gradients** when information fades over long sequences.
- **Exploding gradients** when values grow too large.
- Solutions involve more advanced variants like **LSTM** and **GRU**, which add gating mechanisms.

✅ Visualization Summary

```
Time t-1    x_(t-1)  → h_(t-1) → y_(t-1)
                       ↓
Time t      x_t      → h_t     → y_t
                                  ↓
Time t+1    x_(t+1) → h_(t+1) → y_(t+1)
```

Each hidden state captures information from the past and influences future outputs.

✅ Key Features of RNNs

- **Shared parameters**: The same weights are used at each time step.
- **Memory**: Hidden states act as memory, capturing information from previous time steps.
- **Sequential processing**: Inputs are processed one at a time.

✅ Forward Propagation in an RNN

At each time step tt, the RNN performs the following computations:

Step 1 – Update the hidden state

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

- x_t: Input at time t
- h_{t-1}: Previous hidden state
- W_{xh}, W_{hh}: Weights
- b_h: Bias
- $\tanh$: Activation function

This step updates the memory.

Step 2 – Compute the output

$y_t = W_{hy}h_t + b_y$

- W_{hy}: Weight from hidden state to output
- b_y: Output bias

This gives the prediction at time tt.

Repeat for each time step from t=1 to T

The outputs $y_1, y_2, \ldots, y_T$ are compared with the true outputs $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_T$, and the loss is computed:

$$L = \sum_{t=1}^{T} \text{Loss}(y_t, \hat{y}_t)$$

Where "Loss" could be cross-entropy for classification or mean squared error for regression.

✅ Backpropagation Through Time (BPTT)

After the forward pass, we need to compute gradients and update the weights. But unlike feedforward networks, the hidden state depends on all previous states, so gradients need to be propagated backward through time.

Key steps in BPTT

1. **Compute the gradient of the loss w.r.t the output**:

$$\frac{\partial L}{\partial y_t}$$

2. **Compute the gradient of the loss w.r.t the hidden state**:

$$\frac{\partial L}{\partial h_t}$$

This involves not only how h_t affects y_t, but also how it affects future states $h_{t+1}, h_{t+2}, ..., h_T$.

3. **Compute gradients for the weights**:

$$\frac{\partial L}{\partial W_{hy}}, \quad \frac{\partial L}{\partial W_{hh}}, \quad \frac{\partial L}{\partial W_{xh}}$$

4. **Propagate gradients backward through all time steps**:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_t} + \frac{\partial L}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_t}$$

This recursive relationship shows how error flows backward in time.

✅ Mathematical Intuition

➤ Why recursion matters

The total gradient at time tt depends on:

- Direct influence on the output at t: $\frac{\partial L}{\partial y_t}$
- Indirect influence through future hidden states: $\frac{\partial L}{\partial h_{t+1}}$, etc.

This creates a chain of multiplications as gradients are passed back through time steps.

➤ Vanishing / Exploding Gradients

When computing the recursive gradient:

$$\frac{\partial h_{t+1}}{\partial h_t} = W_{hh} \cdot \text{diag}(1 - h_t^2)$$

Repeated multiplication through time steps can:

- Shrink exponentially → **Vanishing gradients** (hard to learn long-term dependencies)
- Grow exponentially → **Exploding gradients** (training becomes unstable)

✅ Summary of Backpropagation Through Time

1. Forward pass computes h_t and y_t at each time step.
2. Loss LL is calculated by comparing predicted outputs y_t with true outputs.
3. In the backward pass:
   - Gradients of the loss are computed w.r.t outputs and hidden states.
   - Gradients are recursively passed back through time using the chain rule.
   - Weights W_{xh}, W_{hh}, W_{hy} are updated using gradient descent.
4. Issues like vanishing and exploding gradients arise from repeated multiplication through time steps.

✅ Final Thoughts

- BPTT is essentially the chain rule applied over time.
- The hidden state's dependence on past states makes training harder compared to feedforward networks.

- Understanding how gradients flow helps explain why architectures like LSTM or GRU are used to control gradient flow.

✅ Variants and Improvements

LSTM (Long Short-Term Memory)

- Designed to remember information over longer sequences.
- Uses gates (input, forget, output) to control the flow of information.

GRU (Gated Recurrent Unit)

- A simpler version of LSTM.
- Combines some gates and performs similarly in many tasks.

✅ Applications of RNNs

- Language Modeling and Text Generation
- Machine Translation
- Speech Recognition
- Time Series Forecasting
- Video Analysis
- Music Composition
- Handwriting Recognition

✅ Limitations

- Struggles with long-range dependencies in practice.
- Computationally expensive for long sequences.
- Training can be unstable due to vanishing/exploding gradients.
- Parallelization is harder compared to CNNs or Transformers.

✅ Example

Let's say you're predicting the next word in a sentence:

Input sequence:

`I am going to the`

At each time step, the network takes one word, remembers previous words, and predicts the next likely word based on both the current input and the memory of prior inputs.

✅ Summary

Aspect	Description
Type	Neural network for sequences
Memory	Uses hidden states to retain information across time
Works on	Text, audio, video, time series
Key issue	Vanishing/exploding gradients
Solutions	LSTM, GRU, gradient clipping
Applications	NLP, speech, forecasting, etc.

Use Case: Combining CNN and RNN – How It Works

Combining **CNNs (Convolutional Neural Networks)** and **RNNs (Recurrent Neural Networks)** is common when you need to process **spatial** and **temporal/sequential** information together.

Example Use Cases

1. **Video classification or activity recognition**
    - CNN extracts spatial features from each video frame.
    - RNN models how these features change over time.
2. **Image captioning**
    - CNN encodes the image into a feature vector.
    - RNN generates a sequence of words describing the image.
3. **Handwriting recognition**
    - CNN extracts features from handwritten strokes.
    - RNN models the sequence of strokes to recognize letters or words.
4. **Medical imaging sequences**
    - CNN processes each frame or slice (like MRI scans).

    - RNN captures temporal or progressive patterns over time.

✅ How the Architecture Works – Step by Step

Example: Video Action Recognition

Input

- A video sequence → frames $F_1, F_2, ..., F_T$

Step 1 – CNN for Spatial Feature Extraction

- Each frame F_t is passed through a CNN.
- The CNN extracts spatial features → f_t
    - Example: A convolutional backbone like ResNet or VGG is used.

    $$f_t = \text{CNN}(F_t)$$

Now, instead of raw pixels, each frame is represented by a feature vector f_t.

Step 2 – RNN for Temporal Modeling

- The sequence of features $f_1, f_2, ..., f_T$ is fed into an RNN.
- The RNN learns how patterns evolve over time.

  $$h_t = \text{RNN}(f_t, h_{t-1})$$

$$y = \text{Final classification based on } h_T$$

The output could be:

- A class label like "running" or "jumping"
- A sequence output in other tasks like captioning or translation.

✅ Why This Combination Works

Component	What It Does	Why It's Needed
CNN	Extracts spatial or structural features from frames/images	Recognizes patterns, edges, objects

Component	What It Does	Why It's Needed
RNN	Captures temporal dependencies across frames	Understands motion, changes over time

This separation of concerns makes the architecture efficient and powerful.

✅ Detailed Example: Image Captioning

Step 1 – CNN encodes the image

$$f = \text{CNN}(image)$$

This generates a feature vector representing the content of the image.

Step 2 – RNN generates the caption

The RNN is conditioned on f, and generates a sequence of words:

$$h_0 = f$$

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

$$y_t = \text{softmax}(W_{hy} h_t + b_y)$$

Where:

- x_t: Input word embedding at time tt
- h_t: Hidden state
- y_t: Predicted word at time tt

This continues until an "end" token is produced.

✅ Flowchart – Video Example

```
Frames (F1, F2, F3, ..., FT)
       ↓
CNN → f1, f2, f3, ..., fT
       ↓
RNN → h1, h2, h3, ..., hT
```

```
       ↓
Output → Action label (e.g., "walking")
```

✅ Important Notes

✔️ The CNN handles spatial data; the RNN handles temporal patterns.

✔️ The CNN can be pretrained (e.g., on ImageNet), improving performance and reducing training time.

✔️ The RNN learns sequence dependencies — how one frame relates to another.

✔️ This combination is widely used in video analysis, speech-driven gesture recognition, robotics, etc.

✅ Extensions

- **CNN + LSTM**: LSTM replaces the standard RNN to better handle long-range dependencies.
- **Attention mechanisms**: Focus on specific parts of the frame sequence rather than all frames equally.
- **3D CNNs**: Some architectures integrate spatial and temporal convolutions directly, without RNNs.