Minor in AI: Recurrent Neural Networks (RNNs)

6 May 2025

1 Motivation for Recurrent Neural Networks (RNNs)

Many real-world tasks involve sequential or time-dependent data, where the order of the input carries crucial meaning. For example, in a sentence, the meaning of each word depends not only on itself but also on the words before it. Traditional feedforward neural networks process input in isolation, without any inherent memory or mechanism to model the temporal dependencies. This makes them unsuitable for tasks like speech recognition, machine translation, and sentiment analysis, where context is key.

Limitations of Feedforward Networks

- No memory of previous inputs — each input is processed independently.
- Cannot handle variable-length sequences — input and output dimensions must be fixed.
- No temporal modeling — cannot capture time dependencies between data points.

Why RNNs?

Recurrent Neural Networks (RNNs) were introduced to address these limitations. They introduce a feedback loop in the architecture that allows information to persist across time steps. This gives RNNs a form of memory, enabling them to learn temporal patterns and correlations in sequential data.

Key Idea

At the core of an RNN is the hidden state vector h_t, which evolves over time as the input sequence progresses. At each time step t:

- The network receives an input x_t.

- The hidden state h_t is computed based on the current input x_t and the previous hidden state h_{t-1}.
- The output y_t is computed from the current hidden state.

This structure allows the network to retain past information and use it to influence future predictions.

Intuition

Imagine reading a sentence. As you read each word, you form an understanding based on previous words. RNNs mimic this behavior by updating their internal state with every new input. This evolving state helps the model understand the overall meaning, similar to how humans use context when reading or listening.

Applications

RNNs are powerful tools for any domain involving sequential patterns. Some notable applications include:

- **Language Modeling:** Predict the next word in a sentence given the previous words.
- **Machine Translation:** Translate sequences from one language to another.
- **Speech Recognition:** Convert audio signals into text by modeling the sequence of phonemes.
- **Time Series Forecasting:** Predict future values based on previous trends in temporal data (e.g., stock prices, weather patterns).
- **Music Generation:** Create sequences of notes that form coherent melodies.
- **Video Frame Prediction:** Predict the next frame in a video sequence.
- **Handwriting Recognition:** Interpret sequences of pen strokes or images.

2 Structure and Components of RNN

RNN Components (Detailed)

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a memory of previous inputs using recurrent connections. Each component plays a crucial role in enabling temporal understanding.

- **Input Vector** x_t:
  - Represents the input at time step t.

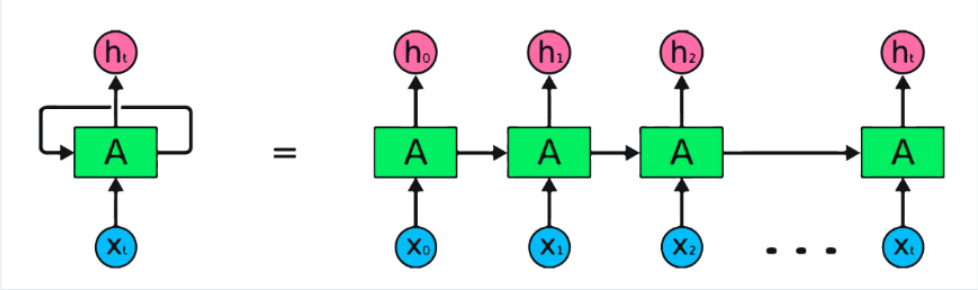

Figure 1: Unrolled RNN over multiple time steps

  - Can be a scalar, vector, or even an embedded word/feature vector depending on the task (e.g., word embeddings in NLP, pixel values in time-series images).
  - Dimension: $x_t \in \mathbb{R}^n$, where n is the size of the input feature space.

- **Hidden State** h_t:
  - Represents the memory of the network at time t.
  - Encodes past information from x_1 to x_t.
  - Updated recursively: depends on current input x_t and the previous hidden state h_{t-1}.
  - Dimension: $h_t \in \mathbb{R}^m$, where m is the number of hidden units.

- **Output** y_t:
  - Prediction made by the RNN at time step t.
  - Depends only on the current hidden state h_t.
  - In classification tasks, often passed through a softmax activation; for regression, can be linear.
  - Dimension: $y_t \in \mathbb{R}^k$, where k is the size of the output space (e.g., number of classes).

- **Weights**:
  - W_{xh} (Input-to-hidden weights): Projects input vector x_t into hidden space.
  - W_{hh} (Hidden-to-hidden weights): Responsible for recurrence; connects h_{t-1} to h_t.
  - W_{hy} (Hidden-to-output weights): Maps the hidden state to output vector y_t.
  - Dimensions:
    * $W_{xh} \in \mathbb{R}^{m \times n}$
    * $W_{hh} \in \mathbb{R}^{m \times m}$
    * $W_{hy} \in \mathbb{R}^{k \times m}$

- **Bias Terms**:
  - $b_h \in \mathbb{R}^m$: Bias added before applying activation in the hidden layer.
  - $b_y \in \mathbb{R}^k$: Bias added before producing the output.

3 Forward Pass Equations (Detailed)

The forward pass computes the hidden and output states for each time step t in the input sequence.

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

Explanation of each term:

- $W_{xh}x_t$: Linear transformation of input vector into hidden space.
- $W_{hh}h_{t-1}$: Recurrence — brings memory of previous time step.
- b_h: Bias added before applying the non-linear activation.
- $\tanh(\cdot)$: Activation function that bounds output to $[-1, 1]$, introducing non-linearity and enabling complex mappings.
- h_t: Hidden state carrying temporal information to the next time step.
- $W_{hy}h_t$: Projection of hidden state to output space.
- b_y: Bias added to the output.

—

4 Backpropagation Through Time (BPTT) (Detailed)

Objective

We aim to minimize the cumulative loss over the sequence:

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

- T: Total time steps in the input sequence.
- y_t: Model's prediction at time t.
- $\hat{y}_t$: Ground truth label at time t.
- $\mathcal{L}(y_t, \hat{y}_t)$: Loss function (e.g., MSE for regression, Cross-Entropy for classification).

—

Gradient Computation

Since RNNs share parameters across time, we must compute gradients not just through current activations but also across time steps.

$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_{hy}}$$

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial L}{\partial h_t} \cdot \left(\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \right) \cdot \frac{\partial h_k}{\partial W_{hh}}$$

Explanation:

1. **Gradient of loss w.r.t output weights W_{hy}**:

$$\frac{\partial L}{\partial W_{hy}} = \sum_{t} \frac{\partial L}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_{hy}} = \sum_{t} \delta_t^y \cdot h_t^\top$$

- $\delta_t^y = \frac{\partial L}{\partial y_t}$: Error at output layer at time t.
- $h_t^\top$: Transpose of hidden state; since $y_t = W_{hy} h_t + b_y$, the gradient w.r.t W_{hy} is the outer product of δ_t^y and h_t.

2. **Gradient of loss w.r.t recurrent weights W_{hh}**:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t} \sum_{k=1}^{t} \delta_t^h \cdot \left(\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \right) \cdot \frac{\partial h_k}{\partial W_{hh}}$$

- $\delta_t^h = \frac{\partial L}{\partial h_t}$: Error backpropagated to hidden layer at time t.
- $\frac{\partial h_i}{\partial h_{i-1}}$: Jacobian of hidden state at i w.r.t. previous hidden state — depends on $\tanh'$, W_{hh}, and current activations.
- $\prod_{i=k+1}^{t}$: Represents the recursive chain of derivatives over time — this is where vanishing/exploding gradients emerge due to repeated multiplication.
- $\frac{\partial h_k}{\partial W_{hh}}$: Gradient of the hidden state w.r.t. recurrent weights — contains the influence of past hidden states.

Key Point:

- Because hidden states are recursively dependent, the gradient at time t affects not only parameters at t but also at all earlier time steps.
- This leads to **temporal credit assignment**, where the network must determine how far back to assign blame for errors.
- The Jacobian chain (product of $\frac{\partial h_i}{\partial h_{i-1}}$) can cause:
  - **Vanishing Gradients**: Gradients shrink exponentially → earlier time steps receive almost no update.
  - **Exploding Gradients**: Gradients grow exponentially → leads to numerical instability.

5 Vanishing and Exploding Gradients (Detailed)

Problem Description

Training Recurrent Neural Networks (RNNs) involves **backpropagating gradients through time**, which means errors must flow backward through potentially hundreds of time steps. However, due to repeated multiplication of derivatives (Jacobians), gradients can:

- **Vanishing Gradient Problem**:
  - When gradients become exponentially smaller with each time step.
  - Early layers (or time steps) receive nearly zero gradient during backpropagation.
  - Learning long-term dependencies becomes extremely difficult.
  - Symptoms: network "forgets" distant inputs; training stagnates.
- **Exploding Gradient Problem**:
  - When gradients grow exponentially as they propagate backward.
  - Causes numerical instability: weights grow uncontrollably, loss becomes NaN.
  - Often occurs in deep or long-sequence models.

Mathematical Insight

Let us consider the recursive dependency of hidden states:

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

During backpropagation, the derivative of the loss L with respect to a hidden state h_{t-n} is influenced by the chain of partial derivatives:

$$\frac{\partial h_t}{\partial h_{t-n}} = \prod_{k=t-n+1}^{t} \frac{\partial h_k}{\partial h_{k-1}}$$

Each term $\frac{\partial h_k}{\partial h_{k-1}}$ is a **Jacobian matrix**, involving:

$$\frac{\partial h_k}{\partial h_{k-1}} = \text{diag}(1 - h_k^2) \cdot W_{hh}$$

- $\text{diag}(1 - h_k^2)$: Derivative of tanh, where $h_k \in (-1, 1)$, hence $1 - h_k^2 \in (0, 1)$.
- So, this Jacobian is a product of a bounded term (from the nonlinearity) and the recurrent weight matrix W_{hh}.

The behavior of the product $\prod_k \frac{\partial h_k}{\partial h_{k-1}}$ depends on the eigenvalues of W_{hh}:

- If the largest eigenvalue $\lambda_{\max} < 1$, the product decays exponentially $\rightarrow$ **vanishing gradients**.

- If $\lambda_{\max} > 1$, the product grows exponentially $\rightarrow$ **exploding gradients**.

This explains why learning dependencies across many time steps becomes either impossible or unstable in vanilla RNNs.

Consequences

- In **vanishing gradients**, earlier time steps fail to influence the learning, leading the network to rely only on recent inputs.
- In **exploding gradients**, training can diverge rapidly unless constrained, causing numerical overflow.

Solutions and Mitigations

- **Gradient Clipping**:
  - Apply thresholding: rescale gradients if their norm exceeds a defined threshold.
  - Prevents exploding gradients.
  - Commonly used:
$$\text{if } \|\nabla_\theta L\| > \tau, \quad \nabla_\theta L \leftarrow \tau \cdot \frac{\nabla_\theta L}{\|\nabla_\theta L\|}$$
- **Use of Gated Architectures (LSTM / GRU)**:
  - Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) explicitly control information flow using gates.
  - Mitigate vanishing gradients by using additive paths (not multiplicative only), preserving gradient flow across time.
  - Forget and input gates regulate what to keep or discard from the memory.
- **Weight Initialization Strategies**:
  - Properly scale weights (e.g., Xavier/Glorot or He initialization) to ensure eigenvalues of W_{hh} remain near 1.
  - Helps reduce the chance of both vanishing and exploding gradients.
- **Normalization Techniques**:
  - Use Layer Normalization or Batch Normalization adapted to sequences.
  - Stabilizes activations and gradients over time steps.
- **Shorter Unroll Lengths (Truncated BPTT)**:
  - Instead of backpropagating over the entire sequence, limit to a fixed number of steps (e.g., 20).
  - Reduces risk of gradient explosion/vanishing, but sacrifices learning from long-range dependencies.

6 Case Study: Language Modeling with RNN

Problem Definition

Goal: Given a sequence of words (or tokens), the objective is to predict the most likely next word in the sequence.

Formally, given an input sequence of tokens $(w_1, w_2, \ldots, w_T)$, we want to learn the conditional probability distribution:

$$P(w_{t+1} \mid w_1, w_2, \ldots, w_t)$$

This task forms the foundation for many NLP problems such as text generation, autocomplete, and translation.

Model Description

To model the probability distribution over the next word, we use a recurrent neural network that updates a hidden state based on the current input and past hidden states.

- **Vocabulary Size** V: The total number of unique tokens in the corpus.
- **Input** x_t:
  - One-hot encoded vector of size V representing the word at time t.
  - Example: For the word "cat" at position t, x_t has all zeros except a 1 at the index corresponding to "cat".
- **Hidden State** h_t:
  $$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$
  Represents a compressed representation of the sequence history up to time t.
- **Output** y_t:
  $$y_t = \text{softmax}(W_{hy}h_t + b_y)$$
  A probability distribution over all V words in the vocabulary. The index with the highest probability is selected as the predicted next word.
- **Loss Function**:
  $$\mathcal{L}_t = -\sum_{i=1}^{V} \hat{y}_t^{(i)} \log(y_t^{(i)})$$
  where $\hat{y}_t$ is the one-hot vector of the actual next word at time t. This is the categorical cross-entropy loss, encouraging the network to assign high probability to the true next word.

Training Procedure

- **Backpropagation Through Time (BPTT)**:
  - Gradients are computed by unrolling the RNN over time steps.
  - Each time step's loss contributes to gradients for shared parameters ($W_{xh}, W_{hh}, W_{hy}, b_h, b_y$).
  - Errors are propagated backward through time using the chain rule.
- **Gradient Clipping**:
  - Applied to prevent exploding gradients.
  - If the norm of the gradient vector exceeds a predefined threshold τ, scale the gradients:
$$\nabla \leftarrow \tau \cdot \frac{\nabla}{\|\nabla\|}$$
- **Monitoring and Optimization**:
  - Monitor loss over epochs to ensure convergence.
  - Optionally track perplexity:
$$\text{Perplexity} = \exp\left(\frac{1}{T}\sum_{t=1}^{T}\mathcal{L}_t\right)$$
  - Optimizer: SGD, RMSprop, or Adam (preferred for fast convergence).

Evaluation and Inference

At inference time, we use the trained model to generate text:

- Start with a seed word w_1.
- Generate the next word w_2 using $P(w_2 \mid w_1)$.
- Feed w_2 back as input to generate w_3, and so on.
- Sampling strategies include:
  - Greedy decoding
  - Beam search
  - Temperature-controlled sampling

Concluding Note on RNNs

Recurrent Neural Networks (RNNs) marked a major leap in sequence modeling by enabling the learning of temporal dependencies in data like language, speech, and time series. Despite their conceptual elegance, vanilla RNNs struggle with learning long-term dependencies due to vanishing and exploding gradient problems. Nonetheless, they serve as the foundational architecture for more advanced models like LSTMs and GRUs.

In language modeling, RNNs can capture contextual information over sequences and learn meaningful statistical regularities. However, they are limited in scalability, parallelizability, and context range. These limitations have led to the rise of more powerful architectures like Transformers, which now dominate NLP.

Takeaway: RNNs are a crucial stepping stone in deep learning history. Understanding their mechanics and challenges gives deep insight into the evolution of sequential modeling and the rationale behind architectural innovations in modern neural networks.