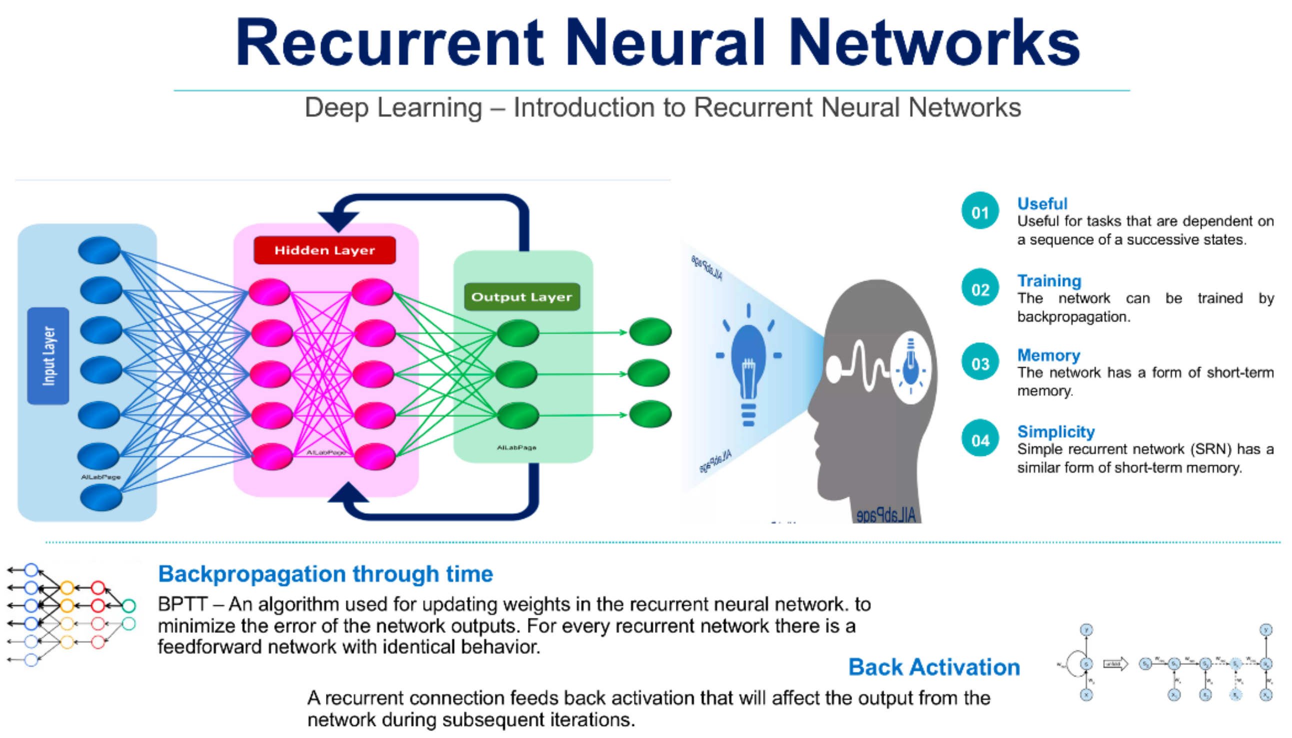
Artificial Intelligence Techniques


Recurrent Neural Networks

Deep Learning – Introduction to Recurrent Neural Networks

01 **Useful**
Useful for tasks that are dependent on a sequence of a successive states.

02 **Training**
The network can be trained by backpropagation.

03 **Memory**
The network has a form of short-term memory.

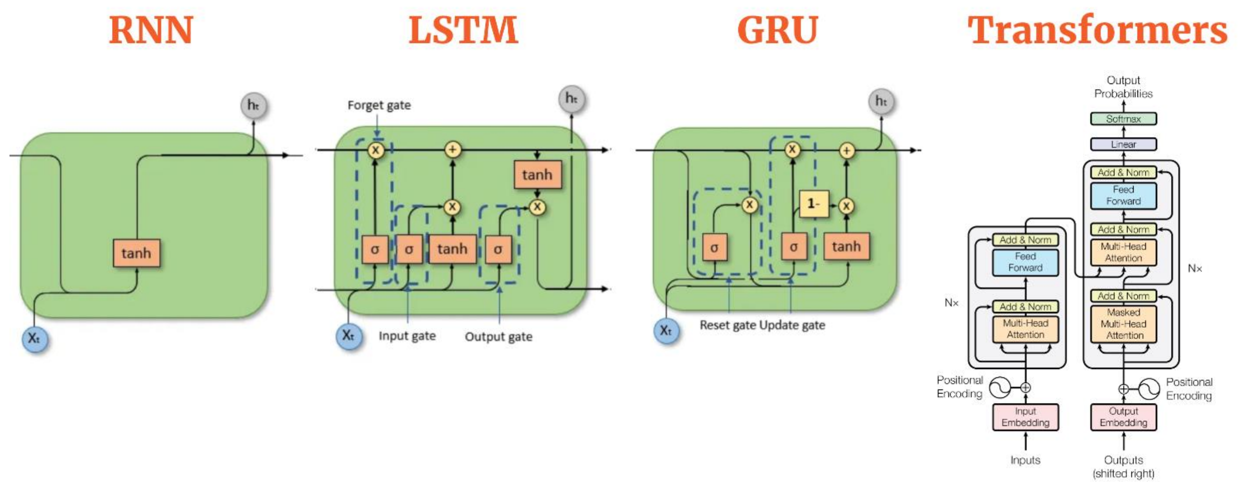
04 **Simplicity**
Simple recurrent network (SRN) has a similar form of short-term memory.

Backpropagation through time

BPTT – An algorithm used for updating weights in the recurrent neural network. to minimize the error of the network outputs. For every recurrent network there is a feedforward network with identical behavior.

Back Activation

A recurrent connection feeds back activation that will affect the output from the network during subsequent iterations.



Assist Prof Dr. Mohamed Aktham Ahmed

Computer Science Department

Computer Science and Mathematics Collage

Tikrit University

Sequence Data

Sequential Data

Sequential data refers to data where the **order of the elements** or observations matters. It's commonly used in contexts where the relationship between data points is influenced by their position in a sequence. Examples of sequential data include time series data, natural language (text), audio signals, stock prices, and more.

characteristics of sequential data:

1. **Temporal or Structural Dependence**: Each element in the sequence depends on its previous elements. For instance, in a time series, the next value might be influenced by the previous ones.
2. **Order Sensitivity**: The arrangement of data points matters. Changing the order can significantly affect the interpretation or outcome, such as in sentence structure where word order determines meaning.
3. **Continuous or Discrete**: Sequential data can either be continuous (e.g., stock prices) or discrete (e.g., daily temperature).

Examples of Sequential Data

- **Text:** Sentences in a document.
- **Time Series:** Stock prices, weather data, or sensor readings over time.
- **Audio:** Speech, music, or other sound waves.
- **Video Frames:** Sequences of images forming a video.
- **DNA Sequences:** Nucleotide bases in genetics.

Sequence Models

Sequence Models

Sequence models are a class of machine learning models designed to process and analyze data where order or temporal dependencies are essential. Unlike traditional models that treat data as independent and identically distributed, sequence models account for the sequential nature of the data, making them suitable for tasks involving time steps, context, or position.

Challenges in Sequence Modeling

1. **Long-Term Dependencies:**
   - Difficulty in retaining information across long sequences.
   - Example: Understanding the subject of a paragraph after reading several sentences.
2. **Vanishing and Exploding Gradients:**
   - Problems arising in traditional recurrent networks during backpropagation.
3. **Computational Efficiency:**
   - Processing long sequences can be resource-intensive.
4. **Variable Sequence Lengths:**
   - Real-world sequences can vary in length, requiring adaptable models.

Types of Sequence Models

1. **Statistical Models:**
   - Hidden Markov Models (HMMs), Conditional Random Fields (CRFs).
2. **Recurrent Neural Networks (RNNs):**
   - Includes LSTMs, GRUs, and basic RNNs.
3. **Convolutional Sequence Models:**

    - Temporal Convolutional Networks (TCNs).

4. **Transformer Models:**
    - Self-attention mechanisms, e.g., Transformers, BERT, GPT.

Applications of Sequence Models

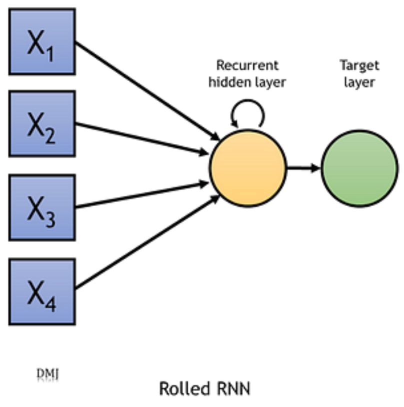
1. **Language Translation:** Converting sentences from one language to another.
2. **Speech Recognition:** Transcribing spoken words into text.
3. **Text Generation:** Writing new content, e.g., stories, articles, or code.
4. **Time-Series Prediction:** Forecasting future values based on past trends.
5. **Anomaly Detection:** Identifying unusual patterns in data sequences.
6. **Healthcare:** Monitoring patient vitals or predicting disease outcomes.

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) *were introduced in the 1980s by researchers* David Rumelhart, Geoffrey Hinton, and Ronald J. Williams. RNNs have laid the foundation for advancements in processing sequential data, such as natural language and time-series analysis, and continue to influence AI research and applications today.

RNNs are a class of artificial neural networks designed to process sequential data, where the order of the data points is significant. Unlike traditional feedforward networks, RNNs have a feedback loop that allows them to maintain a "**memory**" of previous inputs, making them particularly well-suited for tasks where context and temporal dependencies are crucial.



The key feature of RNNs is their **recurrent connection**, where the output from a previous time step is fed back into the network as input for the next time step. This architecture enables RNNs to learn patterns over time, making them effective for

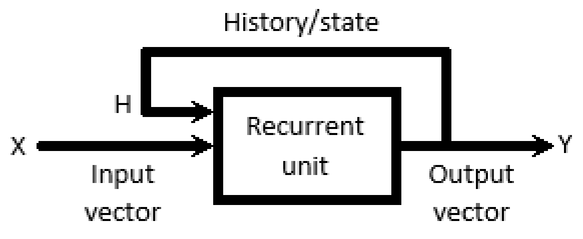
Recurrent Neural Networks introduce a mechanism where the **output from one step is fed back as input to the next**, allowing them to retain information from previous inputs. This design makes RNNs well-suited for tasks where context from earlier steps is essential, such as predicting the next word in a sentence.

The defining feature of RNNs is their ***hidden state***—also called the ***memory state***—which preserves essential information from previous inputs in the sequence. By using the same parameters across all steps, RNNs perform consistently across inputs, reducing parameter complexity compared to traditional neural networks. This capability makes RNNs highly effective for sequential tasks.

RNNs apply the same network to each element in a sequence, RNNs preserve and pass on relevant information, enabling them to learn temporal dependencies that conventional neural networks cannot

Recurrent Unit

The fundamental processing unit in a Recurrent Neural Network (RNN) is a **Recurrent Unit**, which is also called a *"Recurrent Neuron."* Recurrent units hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



The core innovation of RNNs lies in their recurrent connections. At each time step t, an RNN cell:

1. Receives an input x(t)
2. Processes its previous hidden state h(t-1)
3. Produces a new hidden state h(t)
4. Generates an output y(t)

The mathematical formulation can be expressed as:

$$h(t) = \tanh(W\_hh * h(t-1) + W\_xh * x(t) + b\_h)$$

$$y(t) = W\_hy * h(t) + b\_y$$

Where:

- W_{hh}: Hidden-to-hidden weights
- W_{xh}: Input-to-hidden weights
- W_{hy}: Hidden-to-output weights
- b_h, b_y: Bias terms

RNN Unrolling

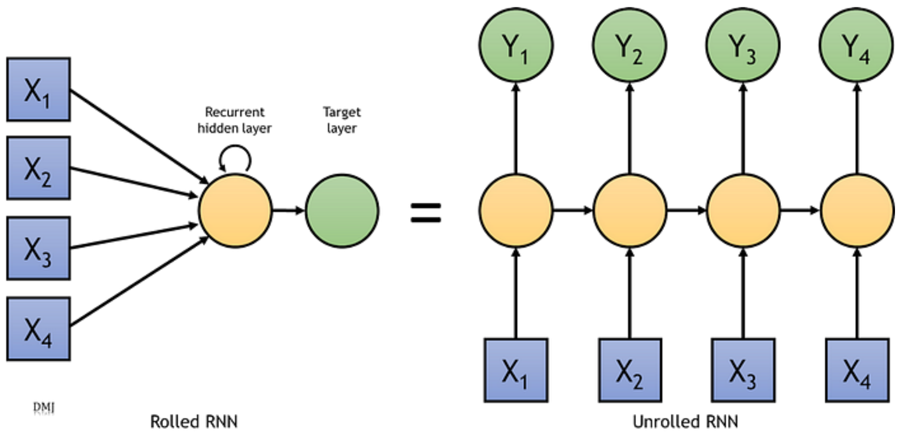
RNN unrolling or ***"unfolding"***, is the process of **expanding the recurrent structure over time steps**. During unrolling, each step of the sequence is represented as a separate layer in a series, illustrating how information flows across each time step. This unrolling enables **backpropagation through time (BPTT)**, a learning process where errors are propagated across time steps to adjust the network's weights, enhancing the RNN's ability to learn dependencies within sequential data.

When training an RNN, it is helpful to **"unroll"** it across time steps, effectively representing each time step as a separate layer in a feedforward network. This allows the model to be trained using **Backpropagation Through Time (BPTT)**, a variation of backpropagation that accounts for dependencies across time steps.

For example, given a sequence $[x_1, x_2, \ldots, x_T]$, the RNN structure can be visualized as:

1. **Time Step 1:** Computes h_1 from x_1 and the initial hidden state h_0.
2. **Time Step 2:** Computes h_2 from x_2 and h_1.
3. **...**
4. **Time Step T:** Computes h_T from x_T and h_{T-1}.

Each hidden state h_t is influenced by all previous inputs $(x_1, x_2, \ldots, x_t)$, allowing the network to capture contextual information from the sequence.

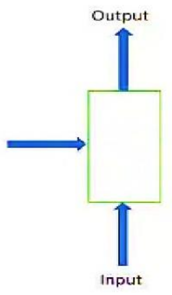


Types Of Recurrent Neural Networks

There are four types of RNNs based on the number of inputs and outputs in the network:

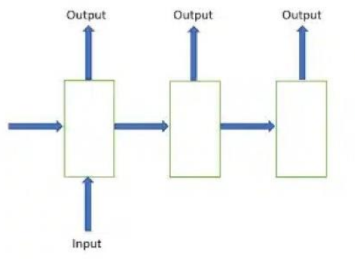
a) One-to-One:

- Single input to single output
- Used for traditional neural network tasks



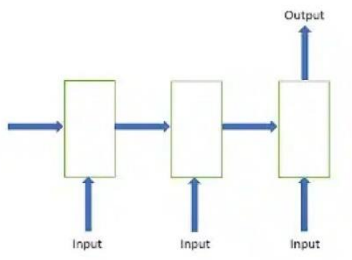
b) One-to-Many:

- Single input generates sequence output
- Example: Image captioning



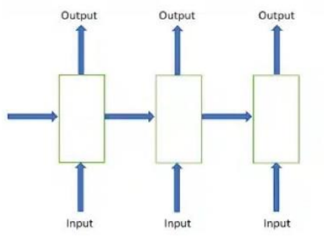
c) Many-to-One:

- Sequence input produces a single output
- Example: Sentiment analysis



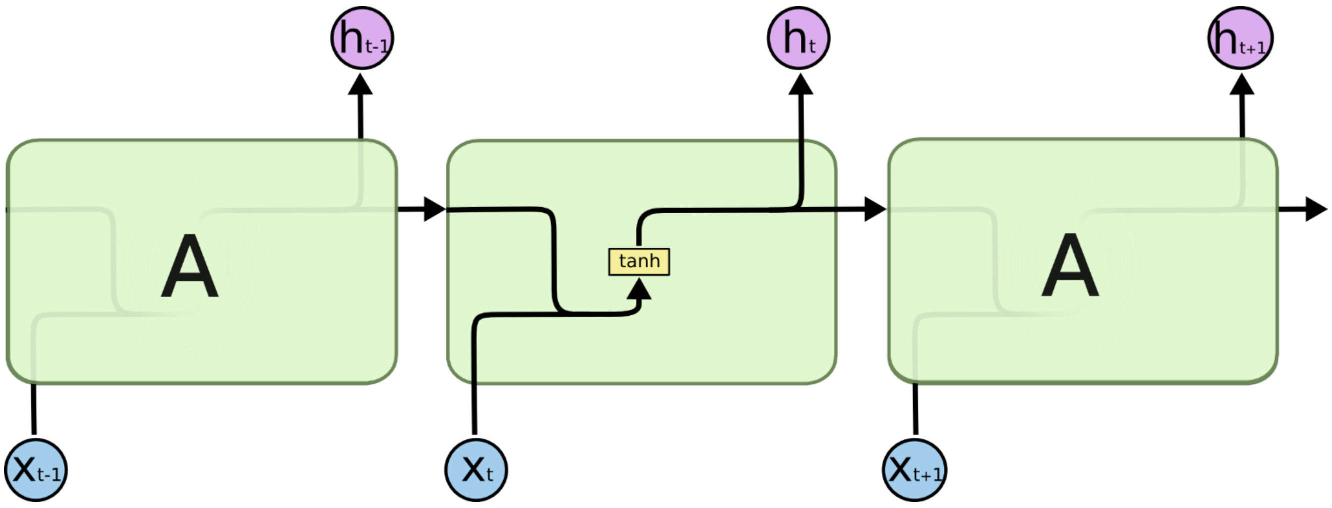
d) Many-to-Many:

- Sequence input to sequence output
- Example: Machine translation



Architecture of Recurrent Neural Networks (RNNs)

The architecture of a Recurrent Neural Network (RNN) distinguishes it from feedforward neural networks by its ability to process sequential data and maintain contextual information across time steps. Here's a breakdown of the core components of an RNN's architecture:



1. Input Layer

- The input to an RNN is a sequence of data points, represented as $\{x_1,x_2,...,x_T\}$ where T is the sequence length.
- At each time step t, an input vector x_t is provided to the network.

2. Hidden Layer (Recurrent Layer)

- The hidden layer is where the recurrence happens. It maintains a state, h_t, that captures information from the current input and the previous **hidden state**.
- The **hidden state** is updated at each time step using the formula:

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

  - W_{xh}: Weights for the input to the hidden state.
  - W_{hh}: Weights for the previous hidden state to the current hidden state.

  - b_h: Bias term for the hidden state.
  - f: Activation function, typically a non-linear function like **tanh** or **ReLU**.
- This recurrent connection allows the network to retain information about previous time steps, creating a "memory" of the sequence.

3. Output Layer

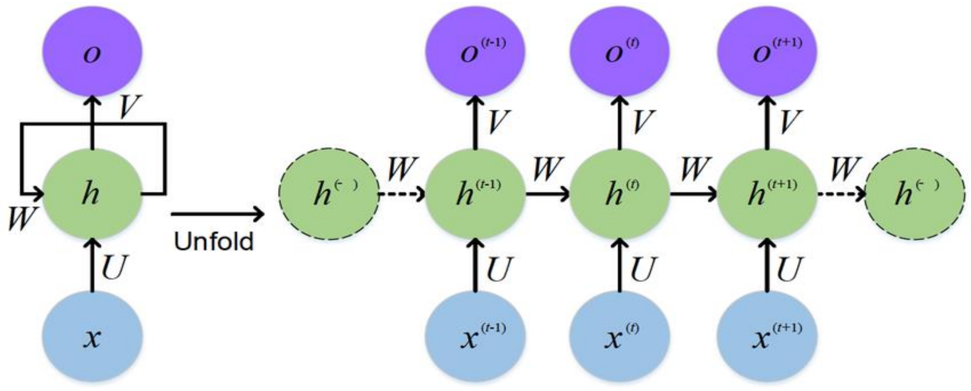
- The output at each time step, y_t, is computed using the current hidden state:

$$y_t = g(W_{hy}h_t + b_y)$$

  - W_{hy}: Weights for the hidden state to output mapping.
  - b_y: Bias term for the output.
  - g: Activation function, which varies based on the task (e.g., softmax for classification, linear for regression).
- Depending on the task, the RNN may output:
  - One value for the entire sequence (many-to-one).
  - One value at each time step (many-to-many).

4. Backpropagation Through Time (BPTT)

- During training, errors are propagated backward through all time steps to update the weights. This is called **Backpropagation Through Time (BPTT)**.
- The process involves computing gradients for each time step and summing them to update the shared parameters.



Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT) is an extension of the backpropagation algorithm designed to train recurrent neural networks (RNNs). It adjusts the weights of the RNN by accounting for the dependencies across time steps in sequential data.

In standard backpropagation, errors are propagated backward through a feedforward network. In BPTT, errors are propagated not only through the layers of the network but also backward through the time steps of the sequence.

BPTT Work

1. Forward Pass

1. Process the input sequence step-by-step over time.
2. At each time step t:
    - The input x_t and the hidden state from the previous step h_{t-1} are used to compute the new hidden state h_t:

$$h_t = f(W_h h_{t-1} + W_x x_t + b)$$

3. If there is an output y_t, it is computed as:

$$y_t = g(W_y h_t + c)$$

4. The loss is computed for each time step based on the output y_t and the true value y.

2. Backward Pass (Unrolling the Network)

- The RNN is "unrolled" across time steps to form a computational graph.
- **Gradients** are computed for each weight by summing up contributions from all time steps.

Steps:

1. **Error Signal:**
   Calculate the error signal at the final time step T:

$$\delta_T = \frac{\partial L_T}{\partial h_T}$$

   where L_T is the loss at time step T.

2. **Backpropagation Through Time Steps:**
   Propagate the error backward through time:

$$\delta_t = \frac{\partial L_t}{\partial h_t} + \frac{\partial h_{t+1}}{\partial h_t}\delta_{t+1}$$

3. **Gradient Computation:**
   Compute gradients for weights W_h, W_x, and W_y by summing their contributions across all time steps:

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \delta_t \frac{\partial h_t}{\partial W_h}$$

4. **Weight Updates:**
   Update weights using a gradient descent algorithm:

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

   where η is the learning rate.

Example: Sentence Processing

Imagine processing the sentence *"I love deep learning"*:

1. The first word ("I") is input, and the RNN generates a hidden state h1.
2. The second word ("love") is combined with h1 to compute h2.
3. This process continues for all words, with the output at each step depending on the sequence so far.

This enables the RNN to understand context and meaning across the sequence.

Example: Predicting Stock Prices

Let's consider an example where an RNN is used to predict stock prices based on past price data. The input sequence consists of daily stock prices over several days, and the goal is to predict the price on the next day.

Step-by-Step Process

1. **Input Sequence:**
    - Suppose the stock prices for the past 5 days are: {100,102,105,107,110}.
      Each value represents the stock price for a day, where:
        - x1=100,
        - x2=102,
        - x3=105,
        - x4=107,
        - x5=110.
2. **Initial Hidden State:**
    - The RNN starts with an initial hidden state h0, typically initialized to zeros.
3. **Hidden State Updates:**

- For each day:
  - The RNN takes the stock price of the current day as input (xt).
  - It updates the hidden state ht using xt and the previous hidden state ht−1.

  For example:

  - At $t = 1$: $h_1 = f(W_{xh}x_1 + W_{hh}h_0 + b_h)$,
  - At $t = 2$: $h_2 = f(W_{xh}x_2 + W_{hh}h_1 + b_h)$,
  - And so on.

4. **Output:**
   - The RNN generates an output yt for each time step, which represents the predicted stock price for the next day:

   - At $t = 1$: $y_1 = g(W_{hy}h_1 + b_y)$,
   - At $t = 2$: $y_2 = g(W_{hy}h_2 + b_y)$,
   - And so on.

   For instance:

   - After processing x1=100, the network might predict y1=101.
   - After x2=102, it might predict y2=104, and so on.
5. **Final Prediction:**
   - After processing the entire sequence (x1,x2,...,x5), the RNN uses the last hidden state h5 to predict the stock price for the next day (y6).

Context is Retained

- The RNN "remembers" trends and patterns, such as whether the stock price is generally increasing or if there are periodic fluctuations.

- For instance, if the prices are consistently rising, the RNN captures this trend in the hidden states, allowing it to predict a higher price for the next day.

Output Example

If the actual sequence of stock prices was:
{100,102,105,107,110,113},
the RNN might predict:
{101,104,106,109,112,115},
with some errors that gets minimized during training.

Vanishing and Exploding Gradient Problems in RNN

The **vanishing** gradient problem arises when **weights that are too small** cause the values being pushed back through backpropagation to drop down to almost zero.

Conversely, the **exploding** gradient occurs when the weight of the current network **is too large**, causing the number to blow up to infinity during backpropagation.

The Vanishing Gradient Problem

One of the primary challenges in training RNNs is the vanishing gradient problem. During backpropagation through time (BPTT), gradients can:

- Exponentially shrink as they flow backward through time steps
- Lead to difficulties in learning long-term dependencies
- Result in the network primarily focusing on recent inputs

Solutions and Variants

1. Long Short-Term Memory (LSTM)

LSTMs introduce specialized gates:

- Forget gate: Controls information to discard
- Input gate: Regulates new information flow
- Output gate: Manages information propagation
- Memory cell: Maintains long-term dependencies

2. Gated Recurrent Units (GRU)

GRUs simplify the LSTM architecture by:

- Combining forget and input gates into an update gate
- Merging cell state and hidden state
- Reducing computational complexity while maintaining performance

Applications of Recurrent Neural Network (RNN)

1. **Natural Language Processing**

   NLP is an application of RNN that is used to comprehend natural language. It involves analyzing the sentence structure, identifying the parts of speech and then making meaning out of the sentences.

- **Sentiment Analysis**: Sentiment analysis is an application of RNN that analyzes text to determine if it has a positive, negative, or neutral sentiment. RNNs are particularly useful in this application because of their ability to analyze text effectively.
- **Speech Recognition** :Speech recognition is an application of RNN that involves speech to text conversion. The network is trained using audio and text data, making it possible for the network to recognize spoken words and convert them to text.

- **Machine translation** is an application of RNN that involves the translation of one language to another in real-time. The network is exposed to a significant number of sentences in multiple languages, enabling the network to learn about grammar, sentence structure, and meaning of sentences.
- **Handwriting recognition** is an application of RNN that involves recognizing handwritten text and converting it to digital text. RNN is particularly useful for this application because of its ability to recognize complex patterns in text.
- **Text Generation** where a model generates coherent and contextually relevant text based on a given input. It has applications in tasks such as chatbots, content creation, translation, storytelling, and code generation.

2. Time Series Analysis

- Financial Forecasting
- Weather Prediction
- Sensor Data Processing

3. Music Generation

- Melody Composition
- Rhythm Pattern Recognition
- Harmonic Progression Prediction

Recent Advances in RNN

1. Attention Mechanisms

Attention mechanisms are a key innovation in deep learning, particularly for handling sequential and structured data. They allow models to dynamically focus on the most relevant parts of the input data when making predictions, improving efficiency and performance in tasks like translation, text summarization, image captioning, and more. The benefit of attention include:

- Dynamically adjusts focus on important parts of input data.
- Handles long-range dependencies effectively.
- Improves interpretability by showing which parts of the input were most influential in the output.

Types of Attention Mechanisms

a. Global Attention

- Considers all input tokens when producing each output token.
- Each token's relevance is determined by computing weights for all inputs.
- Example: Machine Translation, where the entire source sentence is attended to.

b. Local Attention

- Focuses on a subset of input tokens for each output token.
- Reduces computational cost while maintaining focus on relevant regions.

c. Self-Attention (Intra-Attention)

- Each token in the input sequence attends to every other token in the sequence.
- Forms the basis of Transformer models.
- Captures long-range dependencies efficiently.

d. Multi-Head Attention

- Extends self-attention by splitting it into multiple subspaces (heads).
- Each head learns a different aspect of the input, enhancing model capacity.

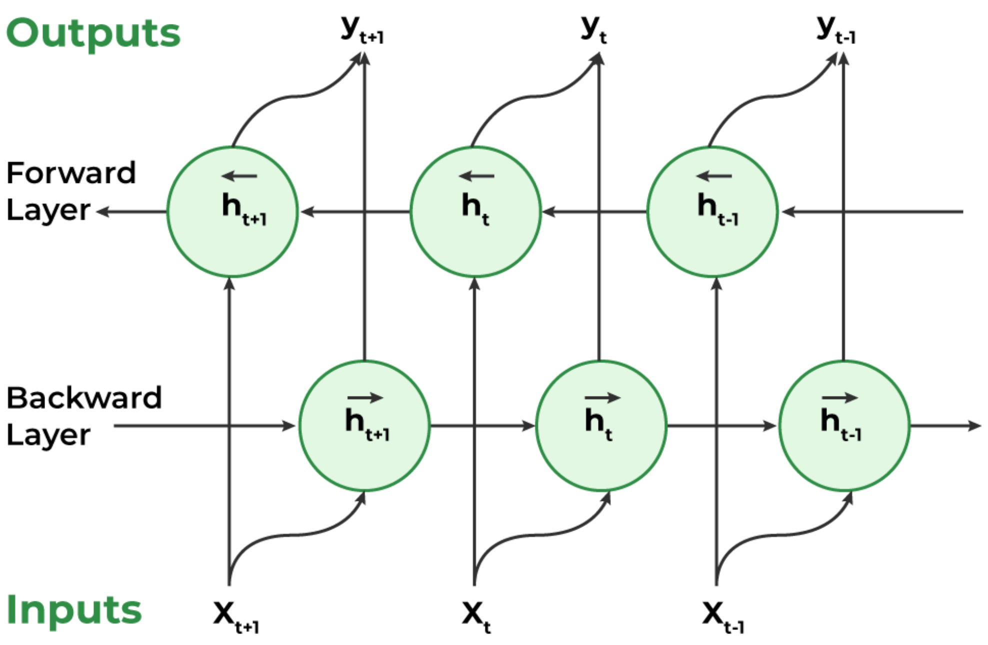
2.Bidirectional RNNs

A Bidirectional Recurrent Neural Network (BiRNN) is an extension of the standard RNN architecture that processes input sequences in both forward and backward directions. This allows the network to capture contextual information from both the past and the future, improving performance in tasks where context is critical, such as speech recognition, language modeling, and machine translation.

These networks process sequences in both forward and backward directions, providing:

- Better context understanding
- Improved performance on many tasks
- More robust feature extraction

Structure of BiRNN



1. **Forward RNN:**
   - Processes the input sequence from the first time step to the last.

$$\overrightarrow{h_t} = f(W_x x_t + W_h \overrightarrow{h_{t-1}} + b)$$

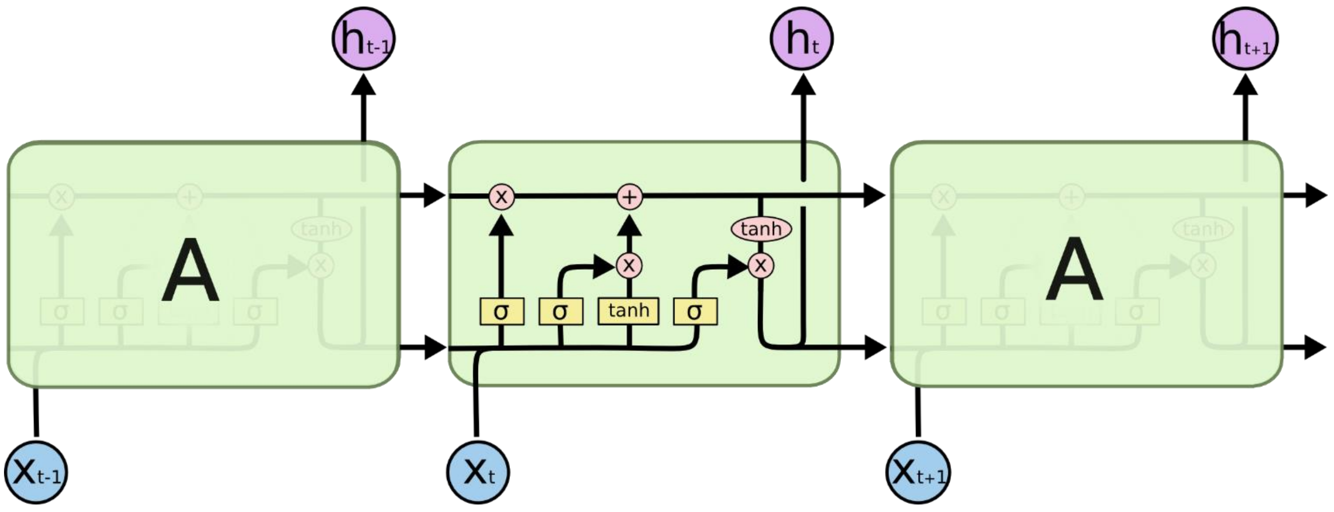
2. **Backward RNN:**
   - Processes the input sequence in reverse, from the last time step to the first.

$$\overleftarrow{h_t} = f(W_x x_t + W_h \overleftarrow{h_{t+1}} + b)$$

3. **Combining States:**
   - The hidden states from both directions are combined at each time step, typically by concatenation or summation:
   - $h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$
   - ht represents the context-aware encoding for time step ttt.

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a type of **Recurrent Neural Network (RNN)** designed to handle sequential data and address the vanishing gradient problem common in traditional RNNs. LSTMs are widely used in applications like natural language processing, time-series analysis, and speech recognition. This guide provides a comprehensive introduction to LSTMs, their components, and their applications.



LSTMs can capture long-term dependencies in sequential data by introducing specialized structures called ***gates***. These gates control the flow of information, enabling the network to retain important information and forget irrelevant details.

Key Features of LSTMs

1. **Memory Cell (Ct):**
   - Stores long-term information, acting as the core memory of the network.
2. **Gating Mechanisms:**
   - Regulate the addition, removal, and retention of information in the memory cell.
   - The gates include:

a) **Forget Gate (*f_t*):** Decides what information to discard.
b) **Input Gate (*i_t*):** Decides what new information to store.
c) **Output Gate (*o_t*):** Decides what information to output at the current step.

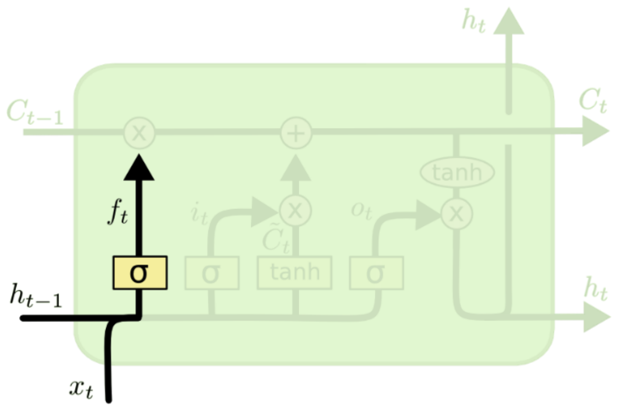
3. **Efficient Handling of Long-Term Dependencies:**
   - LSTMs are explicitly designed to overcome the limitations of traditional RNNs, making them suitable for long sequences.

LSTM Architecture

An LSTM cell contains several components designed to manage the flow of information:

a. Forget Gate

Decides which information to discard from the cell state.



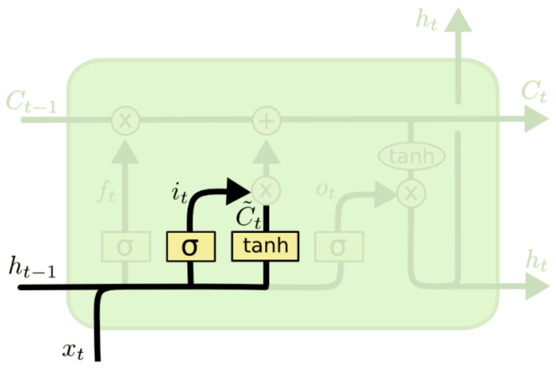
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Where:

- f_t: Forget gate output
- σ: Sigmoid activation function
- W_f, b_f: Weights and bias for the forget gate
- h_{t-1}: Previous hidden state
- x_t: Current input

b. Input Gate

Determines what new information to store in the cell state.



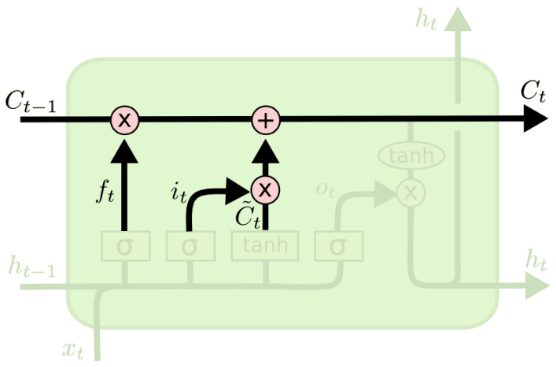
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Where:

- i_t: Input gate output
- $\tilde{C}_t$: Candidate values for cell state

c. Cell State

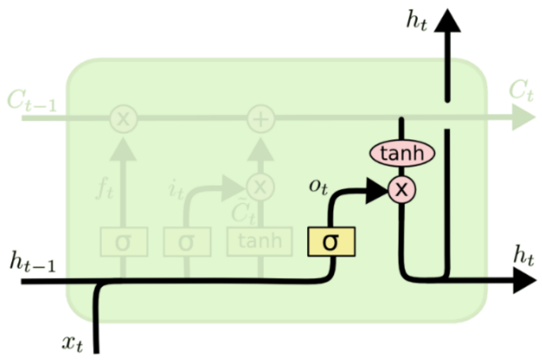
Maintains long-term memory.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

d. Output Gate

Controls what part of the cell state to output as the hidden state.



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

Where:

- o_t: Output gate output
- h_t: New hidden state

LSTEM Training

1. LSTM Forward Pass

1. Receive input x_t and previous hidden state h_{t-1}.
2. Compute gate activations (f_t, i_t, o_t).
3. Update cell state C_t.
4. Generate new hidden state h_t.

This process repeats for every timestep in the sequence.

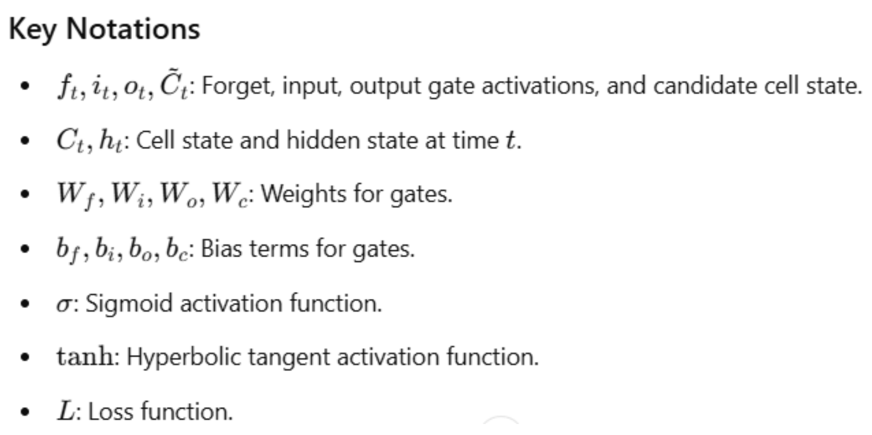
2.LSTM Backward Pass

The **backward pass** in an LSTM is crucial for training the network using backpropagation through time (**BPTT**). During the backward pass, gradients of the loss with respect to LSTM parameters are computed to update the weights. This section provides a detailed breakdown of the backward pass in an LSTM.

The backward pass computes gradients with respect to:

1. **Loss with respect to cell states (**C_t**).**
2. **Loss with respect to hidden states (**h_t **).**
3. **Gradients of gates (**f_t, i_t, o_t, $\tilde{C}_t$ **).**
4. **Gradients of parameters** **(**W_f,W_i,W_o,W_c,b_f,b_i,b_o,b_c **).**

The process propagates gradients from the final timestep backward through time.



Key Notations

- $f_t, i_t, o_t, \tilde{C}_t$: Forget, input, output gate activations, and candidate cell state.
- C_t, h_t: Cell state and hidden state at time t.
- W_f, W_i, W_o, W_c: Weights for gates.
- b_f, b_i, b_o, b_c: Bias terms for gates.
- σ: Sigmoid activation function.
- $\tanh$: Hyperbolic tangent activation function.
- L: Loss function.

Backward Pass Steps

1. Compute gradients for o_t, f_t, i_t, and $\tilde{C}_t$.

2. Accumulate gradients for cell state C_t.
3. Backpropagate gradients to previous hidden and cell states.
4. Update weight and bias gradients.

The backward pass ensures all LSTM parameters are adjusted to minimize the loss function, enabling the network to learn long-term dependencies effectively.

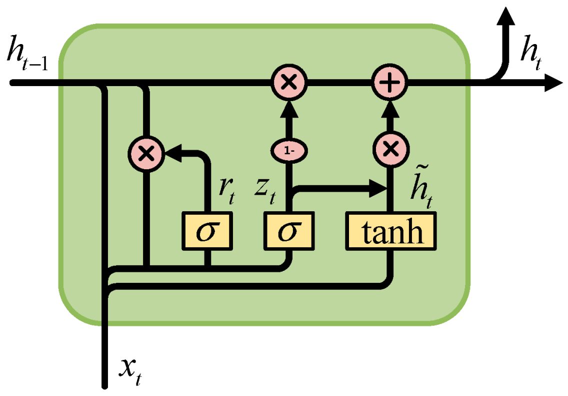
Layers of LSTM

An LSTM layer consists of multiple LSTM cells stacked to process sequences more effectively. Each LSTM layer can be described as follows:

1. **Single LSTM Layer:**
   - Processes a single sequence at a time.
   - Useful for simple sequence modeling tasks like time-series forecasting.
2. **Stacked LSTM Layers:**
   - Combines multiple LSTM layers, where the output of one layer serves as the input to the next.
   - Enables the model to learn hierarchical representations of sequences, improving performance on complex tasks like speech recognition or machine translation.
3. **Bidirectional LSTM (BiLSTM):**
   - Processes sequences in both forward and backward directions, capturing context from both past and future inputs.
4. **LSTM with Dropout:**
   - Adds dropout regularization to prevent overfitting during training.

Gated Recurrent Units (GRU)

The Gated Recurrent Unit (GRU) is a simplified variant of the Long Short-Term Memory (LSTM) network. GRUs are designed to retain the ability to handle long-term dependencies in sequential data while simplifying the architecture by reducing the number of gates and parameters. GRUs combine the functionalities of the forget and input gates into a single gate, making them computationally more efficient than LSTMs.



Key Features of GRUs

1. **Simplified Architecture:**
   - GRUs have fewer gates compared to LSTMs, making them faster to train and computationally lighter.
2. **Update Gate (z_t):**
   - Controls how much of the past information to keep and how much of the new input to incorporate.
3. **Reset Gate (r_t):**
   - Determines how much of the past information to forget.
4. **Direct Hidden State Update:**
   - Unlike LSTMs, GRUs do not maintain a separate memory cell (C_t) and instead directly update the hidden state (h_t).

GRU Architecture

A GRU has two primary gates: the **Update Gate** and the **Reset Gate**. These gates control the flow of information through the network.

a. Update Gate (zt)

The update gate determines how much of the previous hidden state (ht−1) to retain and how much of the new candidate activation $(\tilde{h}_t)$ to use.

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

b. Reset Gate (rt)

The reset gate determines how much of the previous hidden state to forget.

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

c. Candidate Activation $(\tilde{h}_t)$

The candidate activation computes the potential new hidden state, using the reset gate to filter the influence of the past state.

$$\tilde{h}_t = \tanh(W_h \cdot [x_t, r_t \odot h_{t-1}] + b_h)$$

d. Hidden State (ht)

The hidden state is a combination of the previous state and the candidate activation, weighted by the update gate.

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

GRU Training

1. GRU Forward Pass

1. Compute the reset gate (rt).
2. Compute the update gate (zt).
3. Compute the candidate activation $(\tilde{h}_t)$
4. Update the hidden state (ht).

This process is repeated for each timestep in the sequence.

2. GRU Backward Pass

The backward pass in a GRU involves **Backpropagation Through Time (BPTT)**. Gradients are computed for each gate and parameter:

a. Gradients at Hidden State (ht)

The gradient of the loss with respect to the hidden state combines contributions from:

- The current timestep's output.
- The next timestep's backpropagated gradient.

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

b. Gradients at Update Gate (zt)

$$\frac{\partial L}{\partial z_t} = \frac{\partial L}{\partial h_t} \cdot (h_{t-1} - \tilde{h}_t)$$

c. Gradients at Reset Gate (rt)

$$\frac{\partial L}{\partial r_t} = \frac{\partial L}{\partial \tilde{h}_t} \cdot (W_h \cdot h_{t-1})$$

d. Gradients at Candidate $(\tilde{h}_t)$ Activation

$$\frac{\partial L}{\partial \tilde{h}_t} = \frac{\partial L}{\partial h_t} \cdot (1 - z_t)$$

The chain rule is applied to compute parameter gradients for

$W_z, W_r, W_h, b_z, b_r, b_h.$

Layers of GRUs

1. **Single GRU Layer:**
   - Processes a sequence step by step using a single layer of GRU cells.
   - Suitable for simpler tasks like basic time-series forecasting.
2. **Stacked GRU Layers:**

- Multiple GRU layers are stacked, with the output of one layer serving as the input to the next.
- Used for more complex sequence modeling tasks.

3. **Bidirectional GRUs:**
    - Processes the input sequence in both forward and backward directions to capture full context.
    - Commonly used in natural language processing tasks like sentiment analysis and translation.
4. **GRUs with Dropout:**
    - Adds regularization to reduce overfitting, especially in large models or when the dataset is small.

Comparison Between RNN, GRU, and LSTM

Aspect	**RNN**	**GRU**	**LSTM**
Architecture	Simple recurrent structure	Gated structure with two gates	Gated structure with three gates
Memory Management	Relies on hidden states alone	Combines hidden state with gates	Uses separate memory cell and gates
Gates	None	Update, Reset	Forget, Input, Output
Ability to Handle Long-Term Dependencies	Limited due to vanishing gradients	Good due to gating mechanisms	Excellent due to explicit memory cell
Parameters	Few	Moderate	More
Computational Complexity	Low	Moderate	High
Training Speed	Fast	Faster than LSTM	Slower due to complexity
Performance on Complex Sequences	Poor	Competitive	Better for intricate dependencies