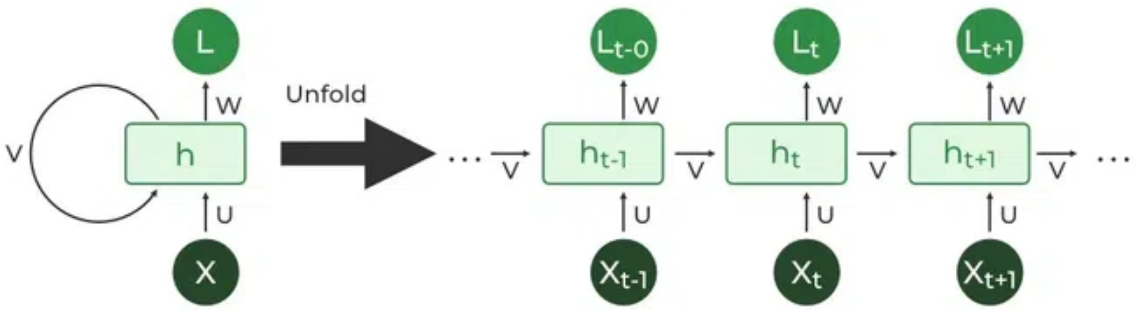
Module 4: Recurrent Neural Networks

Recurrent neural networks - Computational graphs. RNN design.
Encoder - decoder sequence to sequence architectures.
Language modelling example of RNN. Deep recurrent networks.
Recursive neural networks. Challenges of training Recurrent Networks.
Gated RNNs LSTM and GRU.

https://youtu.be/AsNTP8Kwu80?si=XRJe_3Wq0yHEfsUo - RNN

Recurrent Neural Networks (RNNs) work a bit different from regular neural networks. In neural network the information flows in one direction from input to output. However in RNN information is fed back into the system after each step.

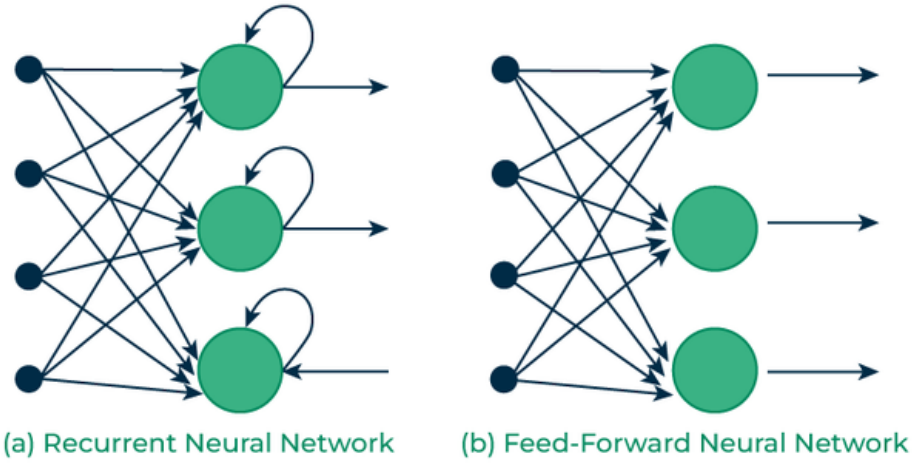
RNNs allow the network to “remember” past information by feeding the output from one step into next step. This helps the network understand the context of what has already happened and make better predictions based on that. For example when predicting the next word in a sentence the RNN uses the previous words to help decide what word is most likely to come next.



How RNN Differs from Feedforward Neural Networks?

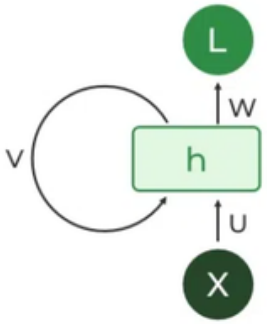
Feedforward Neural Networks (FNNs) process data in one direction from input to output without retaining information from previous inputs. This makes them suitable for tasks with independent inputs like image classification. However FNNs struggle with sequential data since they lack memory.

Recurrent Neural Networks (RNNs) solve this by incorporating loops that allow information from previous steps to be fed back into the network. This feedback enables RNNs to remember prior inputs making them ideal for tasks where context is important.



Key Components of RNNs

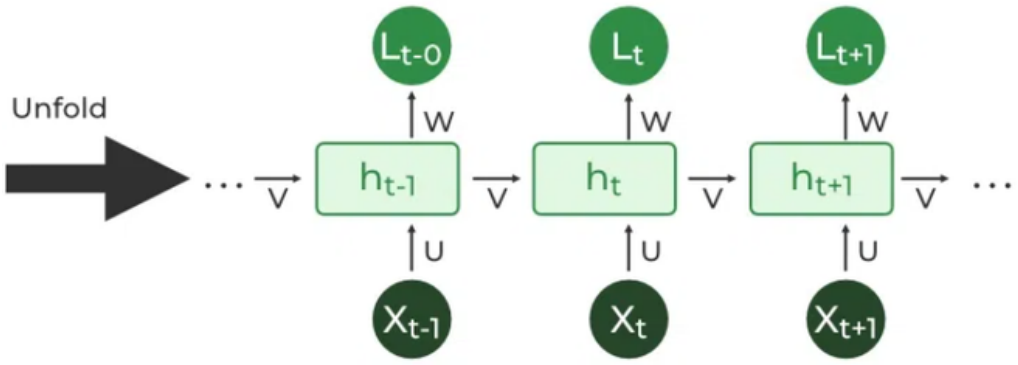
1. Recurrent Neurons



The fundamental processing unit in RNN is a Recurrent Unit. 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.

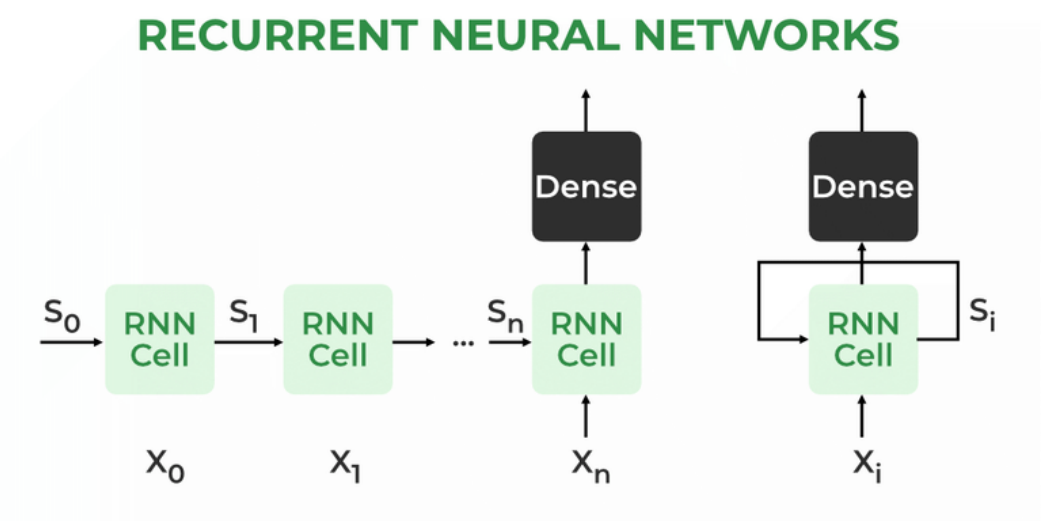
2. RNN Unfolding

RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. During unfolding 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.

Recurrent Neural Network Architecture



RNNs share similarities in input and output structures with other deep learning architectures but differ significantly in how information flows from input to output. Unlike traditional deep neural networks, where each dense layer has distinct weight matrices, RNNs use shared weights across time steps, allowing them to remember information over sequences.

In RNNs, the hidden state Hi is calculated for every input Xi to retain sequential dependencies. The computations follow these core formulas:

1. Hidden State Calculation:

$h=\sigma(U\cdot X+W\cdot ht-1+B)$

Here, h represents the current hidden state, U and W are weight matrices, and B is the bias.

2. Output Calculation:

$Y=O(V\cdot h+C)$

The output Y is calculated by applying O, an activation function, to the weighted hidden state, where V and C represent weights and bias.

3. Overall Function:

$Y=f(X,h,W,U,V,B,C)$

This function defines the entire RNN operation, where the state matrix S holds each element si representing the network's state at each time step i.

How does RNN work?

At each time step RNNs process units with a fixed activation function. These units have an internal hidden state that acts as memory that retains information from previous time steps. This memory allows the network to store past knowledge and adapt based on new inputs.

Updating the Hidden State in RNNs

The current hidden state ht depends on the previous state ht−1 and the current input xt, and is calculated using the following relations:

1. State Update:

$$h_t = f(h_{t-1}, x_t)$$

where:

- h_t is the current state
- h_{t-1} is the previous state
- x_t is the input at the current time step

2. Activation Function Application:

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

Here, Whh is the weight matrix for the recurrent neuron, and Wxh is the weight matrix for the input neuron.

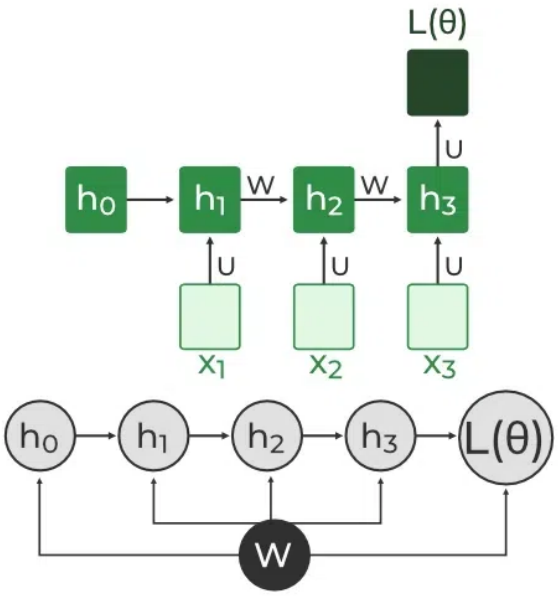
3. Output Calculation:

$$y_t = W_{hy} \cdot h_t$$

where yt is the output and Why is the weight at the output layer.

These parameters are updated using backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as backpropagation through time.

Backpropagation Through Time (BPTT) in RNNs



Since RNNs process sequential data Backpropagation Through Time (BPTT) is used to update the network's parameters. The loss function L(θ) depends on the final hidden state h3 and each hidden state relies on preceding ones forming a sequential dependency chain:

h3 depends on depends on h2,h2 depends on h1,…,h1 depends on h0 depends on h2,h2 depends on h1,…,h1 depends on h0.

In BPTT, gradients are backpropagated through each time step. This is essential for updating network parameters based on temporal dependencies.

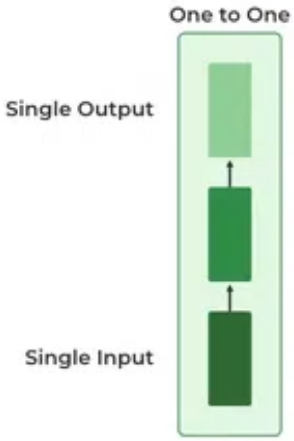
This iterative process is the essence of backpropagation through time.

Types Of Recurrent Neural Networks

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

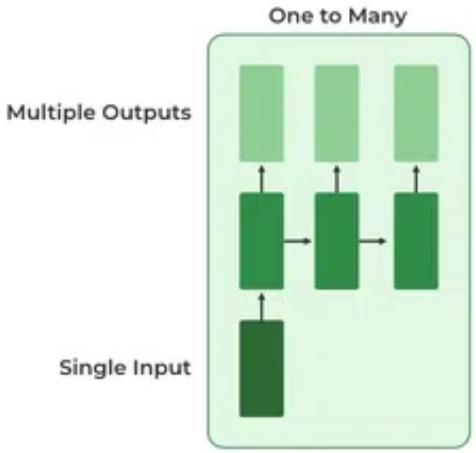
1. One-to-One RNN

This is the simplest type of neural network architecture where there is a single input and a single output. It is used for straightforward classification tasks such as binary classification where no sequential data is involved.



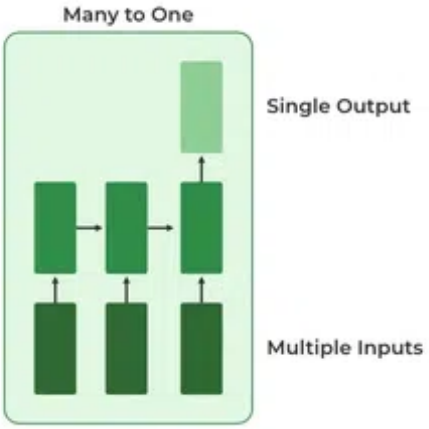
2. One-to-Many RNN

In a One-to-Many RNN the network processes a single input to produce multiple outputs over time. This is useful in tasks where one input triggers a sequence of predictions (outputs). For example in image captioning a single image can be used as input to generate a sequence of words as a caption.



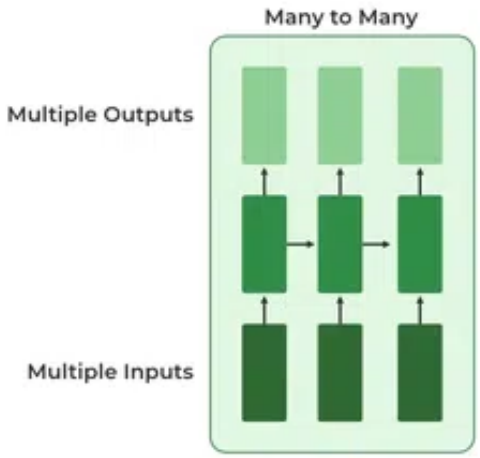
3. Many-to-One RNN

The Many-to-One RNN receives a sequence of inputs and generates a single output. This type is useful when the overall context of the input sequence is needed to make one prediction. In sentiment analysis the model receives a sequence of words (like a sentence) and produces a single output like positive, negative or neutral.



4. Many-to-Many RNN

The Many-to-Many RNN type processes a sequence of inputs and generates a sequence of outputs. In language translation task a sequence of words in one language is given as input, and a corresponding sequence in another language is generated as output.



Variants of Recurrent Neural Networks (RNNs)

There are several variations of RNNs, each designed to address specific challenges or optimize for certain tasks:

1. Vanilla RNN

2. Bidirectional RNNs

Bidirectional RNNs process inputs in both forward and backward directions, capturing both past and future context for each time step. This architecture is ideal for tasks where the entire sequence is available, such as named entity recognition and question answering.

3. Long Short-Term Memory Networks (LSTMs)

Long Short-Term Memory Networks (LSTMs) introduce a memory mechanism to overcome the vanishing gradient problem. Each LSTM cell has three gates:

- Input Gate: Controls how much new information should be added to the cell state.
- Forget Gate: Decides what past information should be discarded.
- Output Gate: Regulates what information should be output at the current step. This selective memory enables LSTMs to handle long-term dependencies, making them ideal for tasks where earlier context is critical.

4. Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs) simplify LSTMs by combining the input and forget gates into a single update gate and streamlining the output mechanism. This design is computationally efficient,

often performing similarly to LSTMs, and is useful in tasks where simplicity and faster training are beneficial.

Encoder - decoder sequence to sequence architectures.

In Deep Learning, Many Complex problems can be solved by constructing better neural network architecture. The RNN(**Recurrent Neural Network**) and its variants are much useful in sequence to sequence learning. The RNN variant LSTM (**Long Short-term Memory**) is the most used cell in seq-seq learning tasks.

*The encoder-decoder architecture for recurrent neural networks is the standard neural **machine translation method** that rivals and in some cases outperforms classical statistical machine translation methods.*

Encoder-Decoder Model

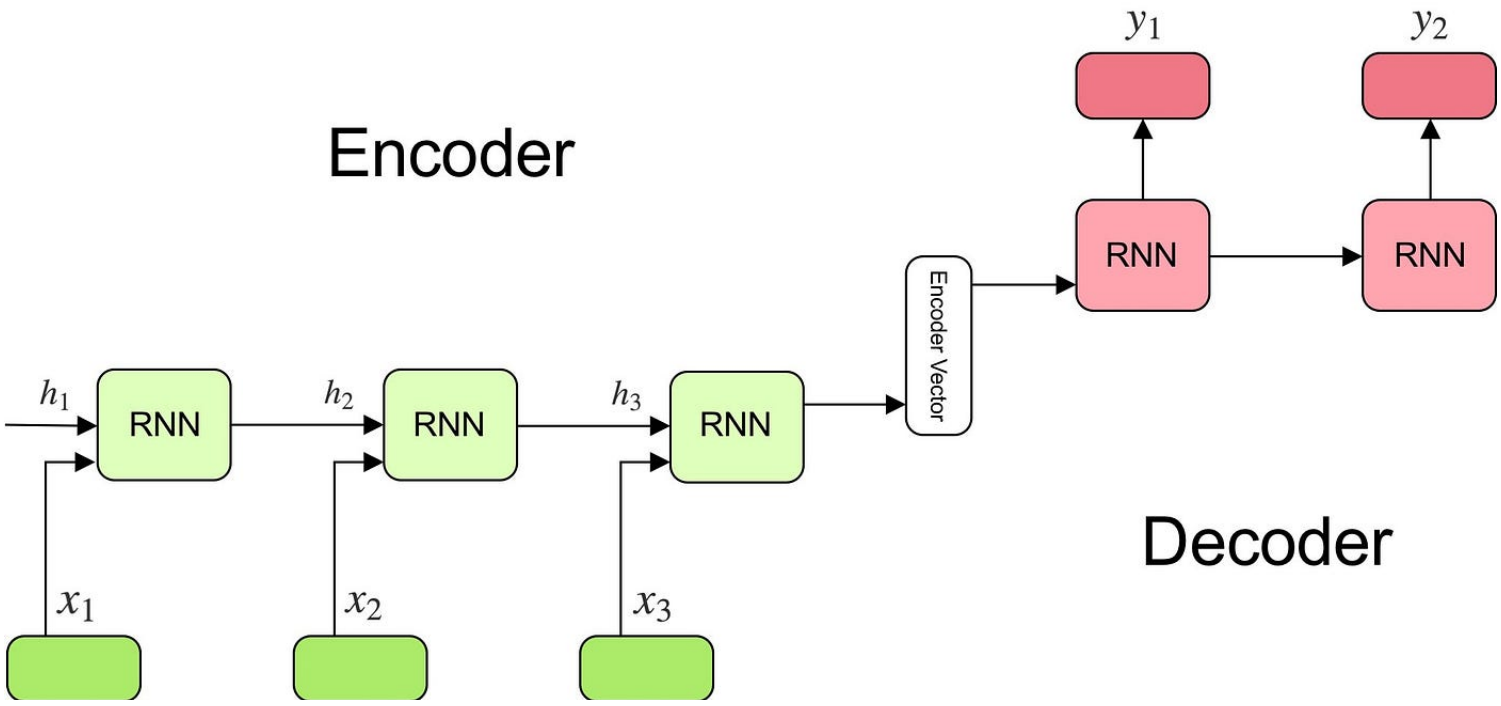
There are three main blocks in the encoder-decoder model,

- Encoder
- Hidden Vector
- Decoder

The Encoder will convert the input sequence into a single-dimensional vector (hidden vector). The decoder will convert the hidden vector into the output sequence.

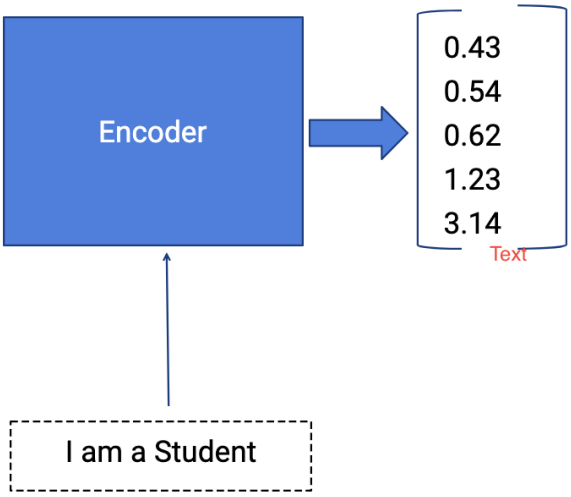
Encoder-Decoder models are jointly trained to maximize the conditional probabilities of the target sequence given the input sequence.

How the Sequence to Sequence Model works?



Encoder

- Multiple RNN cells can be stacked together to form the encoder. RNN reads each inputs sequentially
- For every timestep (each input) t, the hidden state (hidden vector) h is updated according to the input at that timestep X[i].
- After all the inputs are read by encoder model, the final hidden state of the model represents the context/summary of the whole input sequence.
- Example: Consider the input sequence “I am a Student” to be encoded. There will be totally 4 timesteps (4 tokens) for the Encoder model. At each time step, the hidden state h will be updated using the previous hidden state and the current input.



- At the first timestep t1, the previous hidden state h0 will be considered as zero or randomly chosen. So the first RNN cell will update the current hidden state with the first input and h0. Each layer outputs two things — updated hidden state and the output for each stage. The outputs at each stage are rejected and only the hidden states will be propagated to the next layer.
- The hidden states h_i are computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$$

- At second timestep t2, the hidden state h1 and the second input X[2] will be given as input , and the hidden state h2 will be updated according to both inputs. Then the hidden state h1 will be updated with the new input and will produce the hidden state h2. This happens for all the four stages wrt example taken.
- A stack of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence, collects information for that element, and propagates it forward.
- In the question-answering problem, the input sequence is a collection of all words from the question. Each word is represented as x_i where i is the order of that word.

This simple formula represents the result of an ordinary recurrent neural network. As you can see, we just apply the appropriate weights to the previously hidden state h_(t-1) *and the input vector* x_t.

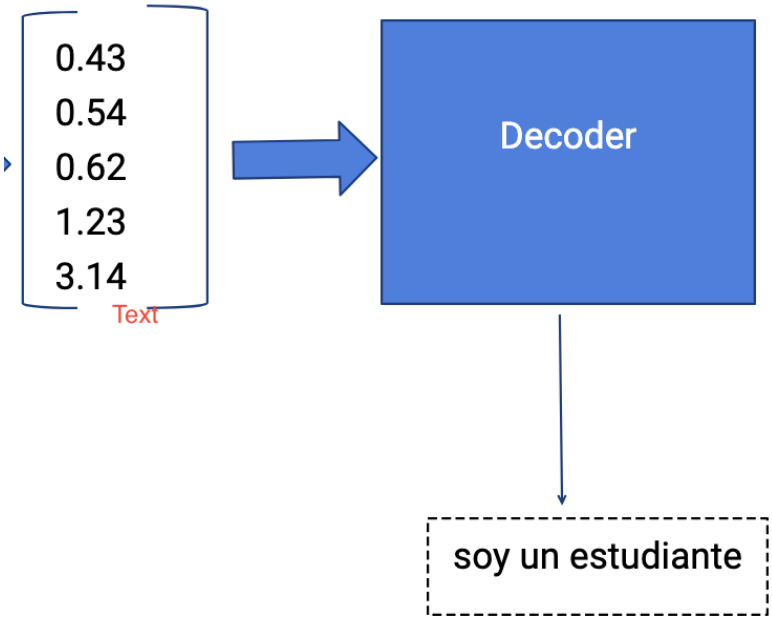
Encoder Vector

- This is the final hidden state produced from the encoder part of the model. It is calculated using the formula above.
- This vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions.
- It acts as the initial hidden state of the decoder part of the model.

Decoder

- The Decoder generates the output sequence by predicting the next output Yt given the hidden state ht.
- The input for the decoder is the final hidden vector obtained at the end of encoder model.
- Each layer will have three inputs, hidden vector from previous layer ht-1 and the previous layer output yt-1, original hidden vector h.
- At the first layer, the output vector of encoder and the random symbol START, empty hidden state ht-1 will be given as input, the outputs obtained will be y1 and updated hidden state h1 (the information of the output will be subtracted from the hidden vector).
- The second layer will have the updated hidden state h1 and the previous output y1 and original hidden vector h as current inputs, produces the hidden vector h2 and output y2.
- The outputs occurred at each timestep of decoder is the actual output. The model will predict the output until the END symbol occurs.
- A stack of several recurrent units where each predicts an output y_t at a time step t.
- Each recurrent unit accepts a hidden state from the previous unit and produces an output as well as its own hidden state.

- In the question-answering problem, the output sequence is a collection of all words from the answer. Each word is represented as y_i where i is the order of that word.



- Any hidden state h_i is computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1})$$

As you can see, we are just using the previous hidden state to compute the next one.

Output Layer

- We use Softmax activation function at the output layer.
- It is used to produce the probability distribution from a vector of values with the target class of high probability.
- The output y_t at time step t is computed using the formula:

$$y_t = softmax(W^S h_t)$$

We calculate the outputs using the hidden state at the current time step together with the respective weight W(S). Softmax is used to create a

probability vector that will help us determine the final output (e.g. word in the question-answering problem).

The power of this model lies in the fact that it can map sequences of different lengths to each other**.** As you can see the inputs and outputs are not correlated and their lengths can differ. This opens a whole new range of problems that can now be solved using such architecture.

Applications

It possesses many applications such as

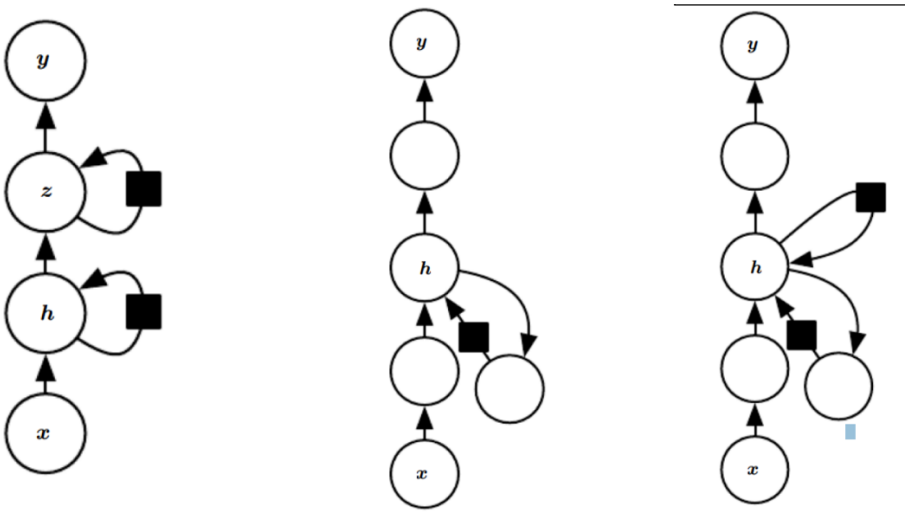
- Google's Machine Translation
- Question answering chatbots
- Speech recognition
- Time Series Application etc.,

Language modelling example of RNN ?????????

Deep Recurrent Networks

The computation in most recurrent neural networks can be decomposed into three blocks of parameters and associated transformations:
1. From the input to the hidden state
2. From the previous hidden state to the next hidden state
3. From the hidden state to the output

Ways of making an RNN deep
1. Hidden recurrent state can be broken down into groups organized hierarchically
2. Deeper computation can be introduced in the input-hidden, hidden-hidden and hidden-output parts. This may lengthen the shortest path linking different time steps
3. The path lengthening effect can be mitigated by introducing skip connections.



Recurrent Neural Networks (RNNs) perform computations based on three fundamental parameter blocks that determine transformations across time steps:

1. **From Input to Hidden State** – Converts input into a hidden representation.
2. **From Previous Hidden State to Next Hidden State** – Captures temporal dependencies.
3. **From Hidden State to Output** – Maps the learned representation to the final prediction.

These transformations typically use **a single weight matrix per block**, forming a shallow transformation equivalent to a single-layer feedforward network in deep architectures.

Depth in RNN Parameter Blocks

- A deeper transformation for each block (input-to-hidden, hidden-to-hidden, hidden-to-output) could enhance representational capacity.
- However, deepening these transformations increases optimization challenges due to **longer gradient propagation paths**, making training more difficult.

Ways to Make an RNN Deep

To improve performance, deep architectures can be introduced in three ways:

1. **Hierarchical Hidden Recurrent States**
    - Hidden states are broken into hierarchical groups where lower layers refine raw input features before passing them to higher layers.
    - This improves feature representation at each level.
2. **Deeper Computation in Hidden-to-Hidden Transitions**
    - Instead of using a single transformation, a deep **Multi-Layer Perceptron (MLP)** is introduced in each of the three blocks.
    - This increases network capacity but can slow down training due to the **lengthened shortest path between time steps**.
3. **Introducing Skip Connections**
    - Direct connections are added between non-consecutive time steps to shorten gradient paths and improve training stability.
    - This mitigates the depth-related challenges of longer dependency paths.

Deep Recurrent Networks

A deep recurrent network introduces multiple layers into the hidden state computations:

- **Multi-layer RNNs**: Stacking multiple RNN layers where the output of one layer becomes the input of the next.
- **LSTMs/GRUs with multiple layers**: Improves long-term dependencies while mitigating vanishing gradients.
- **Residual or Highway Connections**: Help in training deep RNNs by allowing gradients to flow efficiently.

By incorporating depth in a structured manner, deep RNNs achieve **better representation learning** while addressing common training challenges

Recursive neural networks

Recursive Neural Networks are a type of neural network architecture that is specially designed to process hierarchical structures and capture dependencies within recursively structured data. Unlike traditional feedforward neural networks (RNNs), Recursive Neural Networks or RvNN can efficiently handle tree-structured inputs which makes them suitable for tasks involving nested and hierarchical relationships. Recursion is very preliminary concept of DSA where a function or a structure is defined in terms of itself in such way that during execution it calls itself still the used defined criteria or condition is satisfied. When it is used in neural networks, it is just an ability to process data in a hierarchical or nested manner. Here, information from lower-level structures is used to form representations at higher levels. By using this concept, models can capture more complex patterns present in the nested data.

The network processes the input recursively, combining information from child nodes to form representations for parent nodes. RvNN mainly used in some NLP tasks like sentiment analysis etc. by processing data which is in the format of parse tree. RvNN processes parse trees by assigning vectors to each word or subphrase based on the information from its children which allows the network to capture hierarchical relationships and dependencies within the sentence.

Working Principals of RvNN

Some of the key-working principals of RvNN is discussed below:

1. Recursive Structure Handling: RvNN is designed to handle recursive structures which means it can naturally process hierarchical relationships in data by combining information from child nodes to form representations for parent nodes.

2. Parameter Sharing: RvNN often uses shared parameters across different levels of the hierarchy which enables the model to generalize well and learn from various parts of the input structure.

3. Tree Traversal: RvNN traverses the tree structure in a bottom-up or top-down manner by simultaneously updating node representations based on the information gathered from their children.

4. Composition Function: The composition function in RvNN combines information from child nodes to create a representation for the parent node. This function is crucial in capturing the hierarchical relationships within the data.

Difference Facts	RvNN	CNN
Structure	It is designed to process hierarchical or tree-structured data, capturing dependencies within recursively structured information.	It primarily used for grid-structured data like images, where convolutional layers are applied to local receptive fields.
Processing Mechanism	It processes data in a recursive manner, combining information from child nodes to form representations for parent nodes.	It uses convolutional layers to extract local features and spatial hierarchies from input data.
Applications	It is suitable for tasks involving nested structures like natural language parsing or molecular structure analysis.	It excels in tasks related to computer vision, image recognition and other grid-based data.

Differencing facts	RNN	RvNN
Data Structure	It is designed for sequential data like time series or sequences of words.	It is specially designed to tailor for hierarchical data structures like trees.
Processing Mechanism	It processes sequences by maintaining hidden states which capture temporal dependencies.	It processes hierarchical structures by recursively combining information from child nodes.

Differencing facts	RNN	RvNN
Applications	Commonly used in tasks like natural language processing, speech recognition and time series prediction.	Commonly used in applications which involves hierarchical relationships like parsing in NLP, analyzing molecular structures or image segmentation.
Topology	It has linear topology where information flows sequentially through time steps.	It has a tree-like or hierarchical topology which allows them to capture nested relationships within the data.

Challenges of Training Recurrent Neural Networks (RNNs)

1. Vanishing and Exploding Gradients

- **Vanishing Gradient**: During backpropagation through time (BPTT), gradients shrink exponentially, making it difficult for earlier time steps to influence later ones. This results in **poor learning of long-term dependencies**.
- **Exploding Gradient**: Conversely, gradients may grow exponentially, causing instability and large weight updates, leading to **diverging loss values**.
- **Solutions**:
  - Gradient clipping (limits large updates)
  - Using architectures like **LSTMs** and **GRUs**, which help preserve gradients

2. Long-Term Dependencies are Hard to Learn

- Standard RNNs struggle to retain information over long sequences due to **short-term memory bias**.
- The longer the sequence, the harder it is to **propagate relevant information** from the past to the future.
- **Solution**: LSTMs and GRUs introduce **gating mechanisms** to selectively retain or discard information.

3. Difficulty in Optimization

- Deeper RNNs have **longer backpropagation paths**, making it harder for optimization algorithms (e.g., SGD, Adam) to converge effectively.
- Training deep RNNs can be **computationally expensive** and slow.
- **Solutions**:
    - Use **pre-trained embeddings** to speed up learning.
    - Apply **batch normalization** or **layer normalization** to stabilize training.

4. High Computational Cost

- RNNs process inputs sequentially, making them **slower** compared to parallelizable models like CNNs or Transformers.
- Training long sequences requires high memory and **efficient hardware (e.g., GPUs, TPUs)**.
- **Solutions**:
    - Use **truncated BPTT** (only backpropagate for a limited number of time steps).
    - Explore **Transformer models**, which avoid sequential dependencies.

5. Overfitting on Small Datasets

- RNNs have **high capacity**, making them prone to memorizing training data rather than generalizing.
- **Solution**:
    - **Dropout in RNNs** (applied between layers)
    - **Regularization techniques** (L2 norm, weight decay)

6. Exposure Bias in Sequence Generation

- During training, the model sees **true previous words**, but during inference, it uses **its own predictions**, leading to **error accumulation**.
- **Solution**: Scheduled sampling (gradually transitioning from true labels to model predictions).

7. Difficulty in Handling Long Sequences

- Processing long sequences leads to **memory constraints** and slow computations.
- **Solution**:
    - Use **attention mechanisms** to focus on relevant parts of the sequence.
    - Consider alternative architectures like **Transformers**, which handle long-range dependencies more efficiently.

8. Non-Stationary Data Distributions

- In real-world applications (e.g., stock market predictions, speech recognition), input distributions change over time.
- **Solution**: Adaptive learning rates, continual learning techniques.

LSTM

Long Short-Term Memory (LSTM) is an enhanced version of the Recurrent Neural Network (RNN) . LSTMs can capture long-term dependencies in sequential data making them ideal for tasks like language translation, speech recognition and time series forecasting.

Unlike traditional RNNs which use a single hidden state passed through time LSTMs introduce a memory cell that holds information over extended periods addressing the challenge of learning long-term dependencies.

Problem with Long-Term Dependencies in RNN

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. However they often face challenges in learning long-term dependencies where information from distant time steps becomes crucial for making accurate predictions for current state. This problem is known as the vanishing gradient or exploding gradient problem.

• Vanishing Gradient: When training a model over time, the gradients (which help the model learn) can shrink as they pass through many steps. This makes it hard for the model to learn long-term patterns since earlier information becomes almost irrelevant.

• Exploding Gradient: Sometimes, gradients can grow too large, causing instability. This makes it difficult for the model to learn properly, as the updates to the model become erratic and unpredictable.

Both of these issues make it challenging for standard RNNs to effectively capture long-term dependencies in sequential data.

LSTM Architecture

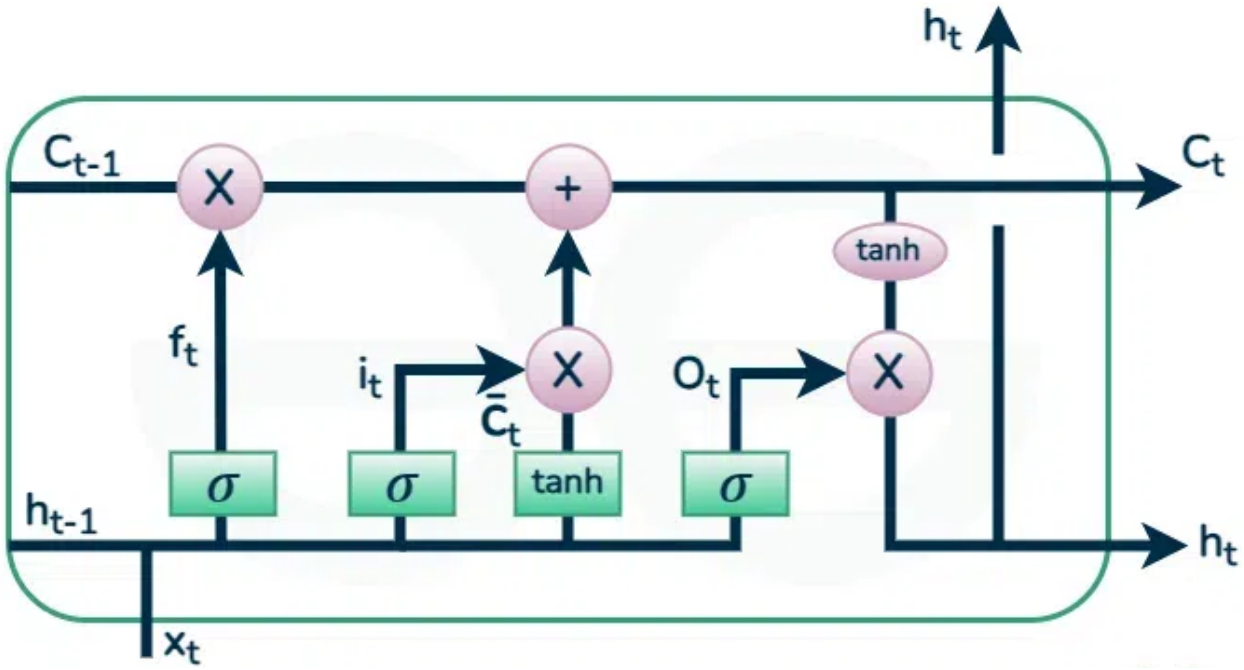
LSTM architectures involves the memory cell which is controlled by three gates: the input gate, the forget gate and the output gate. These gates decide what information to add to, remove from and output from the memory cell.

- Input gate: Controls what information is added to the memory cell.
- Forget gate: Determines what information is removed from the memory cell.
- Output gate: Controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

Working of LSTM

LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells.



Information is retained by the cells and the memory manipulations are done by the gates. There are three gates –

Forget Gate

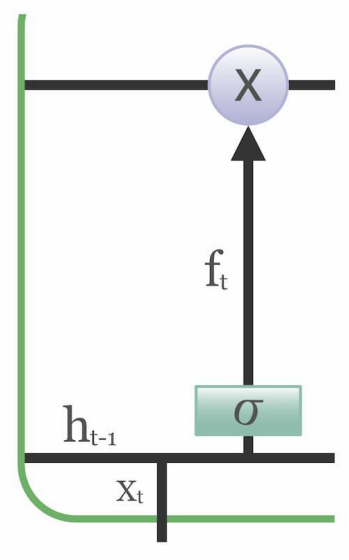
The information that is no longer useful in the cell state is removed with the forget gate. Two inputs xt (input at the particular time) and ht-1 (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.

The equation for the forget gate is:

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

where:

- W_f represents the weight matrix associated with the forget gate.
- [h_t-1, x_t] denotes the concatenation of the current input and the previous hidden state.
- b_f is the bias with the forget gate.
- σ is the sigmoid activation function.



Input gate

The addition of useful information to the cell state is done by the input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs ht-1 and xt. . Then, a vector is created using tanh function that gives an output from -1 to +1, which contains all the possible values from ht-1 and xt. At last, the values of the vector and the regulated values are multiplied to obtain the useful information. The equation for the input gate is:

$it=\sigma(Wi\cdot[ht-1,xt]+bi)$

$C\hat{}t=\tanh(Wc\cdot[ht-1,xt]+bc)$

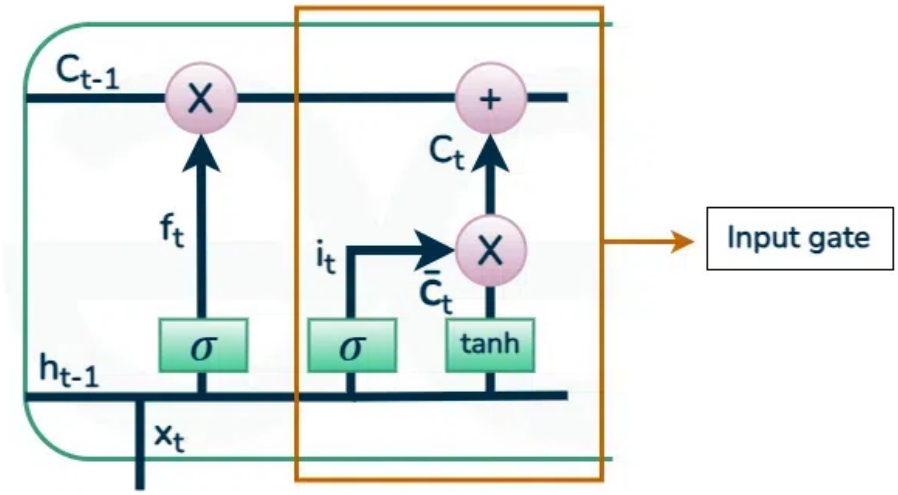
We multiply the previous state by ft, disregarding the information we had previously chosen to ignore. Next, we include it∗Ct. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

$Ct=ft\odot Ct-1+it\odot C\hat{}t$

where

• ⊙ denotes element-wise multiplication

• tanh is tanh activation function



Output gate

The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter by the values to be remembered using inputs ht−1and xt. At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell. The equation for the output gate is:

$o_t=\sigma(W_o\cdot[h_{t-1},x_t]+b_o)$

Bidirectional LSTM Model

Bidirectional LSTM (Bi LSTM/ BLSTM) is a variation of normal LSTM which processes sequential data in both forward and backward directions. This allows Bi LSTM to learn longer-range dependencies in sequential data than traditional LSTMs which can only process sequential data in one direction.

• Bi LSTMs are made up of two LSTM networks one that processes the input sequence in the forward direction and one that processes the input sequence in the backward direction.

• The outputs of the two LSTM networks are then combined to produce the final output.

LSTM models including Bi LSTMs have demonstrated state-of-the-art performance across various tasks such as machine translation, speech recognition and text summarization.

LSTM networks can be stacked to form deeper models allowing them to learn more complex patterns in data. Each layer in the stack captures different levels of information and time-based relationships in the input.

Applications of LSTM

Some of the famous applications of LSTM includes:

• Language Modeling: Used in tasks like language modeling, machine translation and text summarization. These networks learn the dependencies between words in a sentence to generate coherent and grammatically correct sentences.

• Speech Recognition: Used in transcribing speech to text and recognizing spoken commands. By learning speech patterns they can match spoken words to corresponding text.

• Time Series Forecasting: Used for predicting stock prices, weather and energy consumption. They learn patterns in time series data to predict future events.

• Anomaly Detection: Used for detecting fraud or network intrusions. These networks can identify patterns in data that deviate drastically and flag them as potential anomalies.

• Recommender Systems: In recommendation tasks like suggesting movies, music and books. They learn user behavior patterns to provide personalized suggestions.

• Video Analysis: Applied in tasks such as object detection, activity recognition and action classification. When combined with Convolutional Neural Networks (CNNs) they help analyze video data and extract useful information.

LSTM VS RNN

Feature	LSTM (Long Short-term Memory)	RNN (Recurrent Neural Network)
Memory	Has a special memory unit that allows it to learn long-term dependencies in sequential data	Does not have a memory unit
Directionality	Can be trained to process sequential data in both forward and backward directions	Can only be trained to process sequential data in one direction
Training	More difficult to train than RNN due to the complexity of the gates and memory unit	Easier to train than LSTM

Feature	LSTM (Long Short-term Memory)	RNN (Recurrent Neural Network)
Long-term dependency learning	Yes	Limited
Ability to learn sequential data	Yes	Yes
Applications	Machine translation, speech recognition, text summarization, natural language processing, time series forecasting	Natural language processing, machine translation, speech recognition, image processing, video processing

Long Short-Term Memory (LSTM) Networks

Introduction to LSTM

- LSTMs address the **context management problem** by breaking it into two sub-problems:
  1. **Removing unnecessary information** from the context.
  2. **Adding important information** for later decision-making.
- The key to LSTM is **learning how to manage context** dynamically, instead of using a hard-coded strategy.
- LSTMs achieve this by:
  - Adding an **explicit context layer** to the architecture.
  - Using **specialized neural units** with **gates** that control information flow.
- These **gates** use additional weights that sequentially process:
  - The input,
  - The previous hidden layer,
  - The previous context layers.

The Core Idea Behind LSTMs

- The **cell state** (represented by a horizontal line in diagrams) acts like a **conveyor belt**, carrying information with minimal modification.
- LSTMs use **gates** to control what information is removed or added to the cell state.
- Each gate consists of:
  - A **feedforward layer**,
  - A **sigmoid activation function** (values between 0 and 1),

    - A **pointwise multiplication** with the layer being gated.
- A **sigmoid value of 0** means “block everything,” while **1 means let everything pass.”

Three Types of LSTM Gates

1. **Forget Gate** 🛑
    - Determines **what information should be removed** from the cell state.
    - Uses a **sigmoid layer** that outputs values between **0 (forget) and 1 (retain)**.
    - Example:
        - Sentence 1: *"Ravi plays basketball."*
        - Sentence 2: *"He is good at web designing."*
        - Since the context has changed, the forget gate **removes information about basketball** to focus on web designing.
2. **Input Gate** ✍️
    - Determines **what new information should be added** to the cell state.
    - Uses:
        - A **sigmoid layer** to decide what values to update.
        - A **tanh layer** to generate new candidate values (-1 to 1 range).
    - Example:
        - Sentence: *"Ravi is a university topper."*
        - Important information (*"university topper"*) is **added**, while unnecessary parts (*"yesterday he told me"*) are **discarded**.
3. **Output Gate** 📤
    - Determines **what part of the cell state should be output**.
    - Uses:
        - A **sigmoid layer** to filter what gets passed through.
        - A **tanh layer** to scale values between -1 and 1.
    - Example:
        - Sentence: *"Dan grew up in France. He speaks fluent French."*
        - The output gate ensures that the **subject's singular/plural nature** is passed on to ensure correct verb conjugation.

Step-by-Step LSTM Walkthrough

1. Forgetting Irrelevant Information (Forget Gate)

- The first step is to **decide what information to remove** from the cell state.
- A **sigmoid layer** (forget gate) checks each value and assigns a retention score (0 → forget, 1 → keep).
- Example:
    - **Before Forget Gate:**
        - Subject: *Dan*
        - Gender: *Male*
    - **After Forget Gate (New Subject Found):**
        - Old subject’s gender is **forgotten**.

2. Storing Important Information (Input Gate)

- The **input gate** determines which **new information should be stored** in the cell state.
- Two steps:
  - A **sigmoid layer** filters what should be updated.
  - A **tanh layer** generates new values for the cell state.
- Example:
  - **Old Cell State:** *Subject: Dan, Gender: Male*
  - **New Cell State:** *Subject: He, Gender: Male* (as inferred from "He speaks fluent French").

3. Updating the Cell State

- The old cell state **Ct-1** is updated to the new **Ct** based on the decisions made in steps 1 and 2.
- The forget gate's output **removes old, unnecessary information**.
- The input gate's output **adds new, useful information**.

4. Generating the Output (Output Gate)

- The final step is **deciding what to output** from the LSTM unit.
- The **output gate**:
  - Uses a **sigmoid layer** to determine what parts of the cell state are important.
  - Passes the filtered cell state through **tanh** to ensure values stay between -1 and 1.
- Example:
  - Sentence: *"Dan grew up in France. He speaks fluent French."*
  - The LSTM **outputs subject information (Singular/Plural)** to ensure correct **subject-verb agreement**.

LSTM Gates in Action (Examples)

Forget Gate Example

- Given the sentence: *"Bob is a nice person. Dan, on the other hand, is evil."*
- After encountering the **full stop**, the forget gate **removes Bob's subject information**, making room for Dan.

Input Gate Example

- Given: *"Bob knows swimming. He told me over the phone that he had served in the navy."*
- The **important information** ("Bob knows swimming, served in the navy") is **stored**, while **irrelevant details** ("told me over the phone") are **discarded**.

Output Gate Example

- Given: *"Bob fought single-handedly with the enemy and died for his country. For his contributions, brave ____."*

- Since "brave" is an **adjective**, the output gate predicts that the missing word is **likely a noun (e.g., soldier, warrior, etc.)**.

LSTM Architecture Overview

Key Mathematical Operations in LSTM

The LSTM unit involves **four matrix computations** that run in parallel:

1. **Forget Gate:** Controls what information to remove.

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

2. **Input Gate:** Controls what new information to add.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

3. **Updating Cell State:**

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

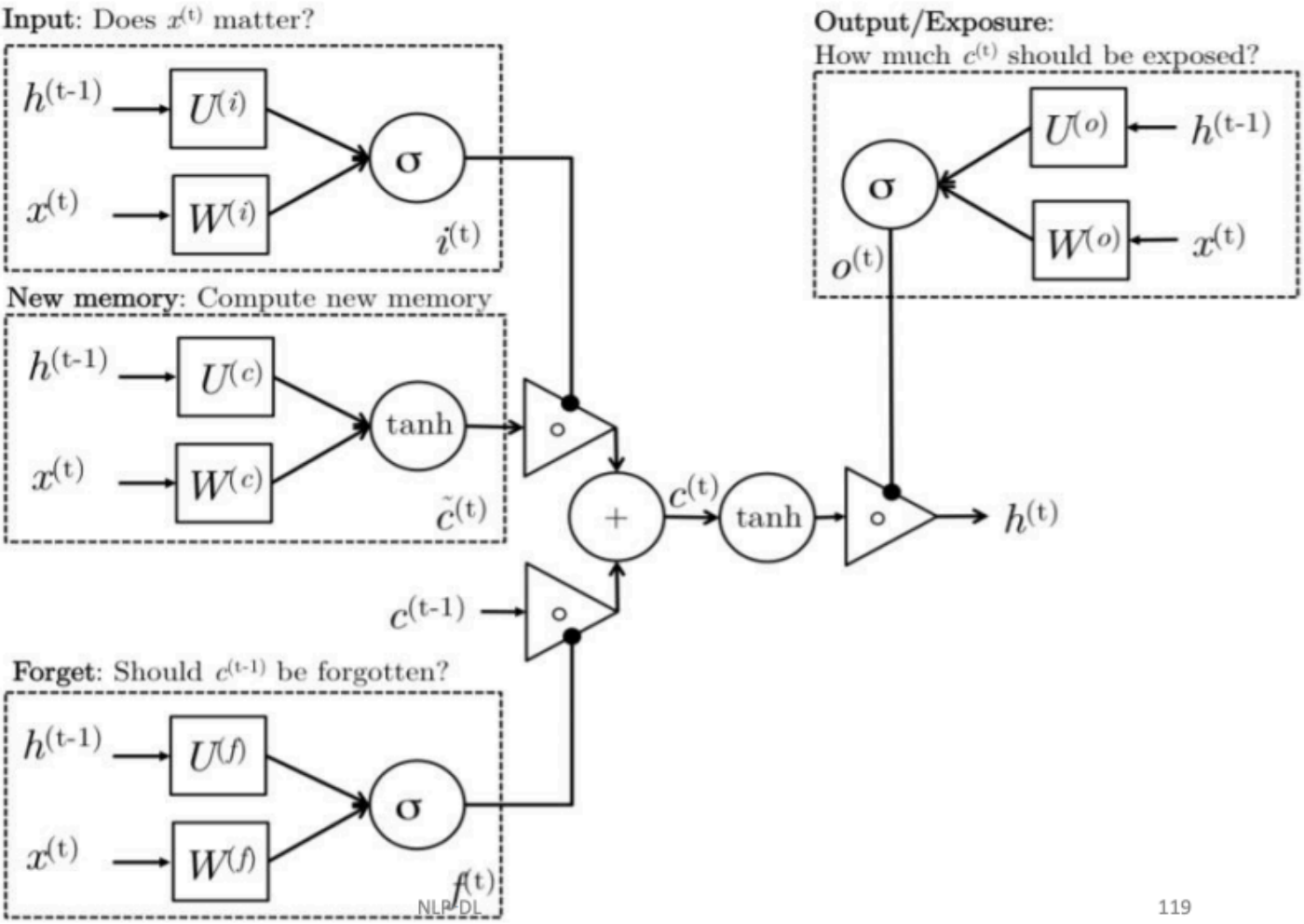
4. **Output Gate:** Controls what information to output.

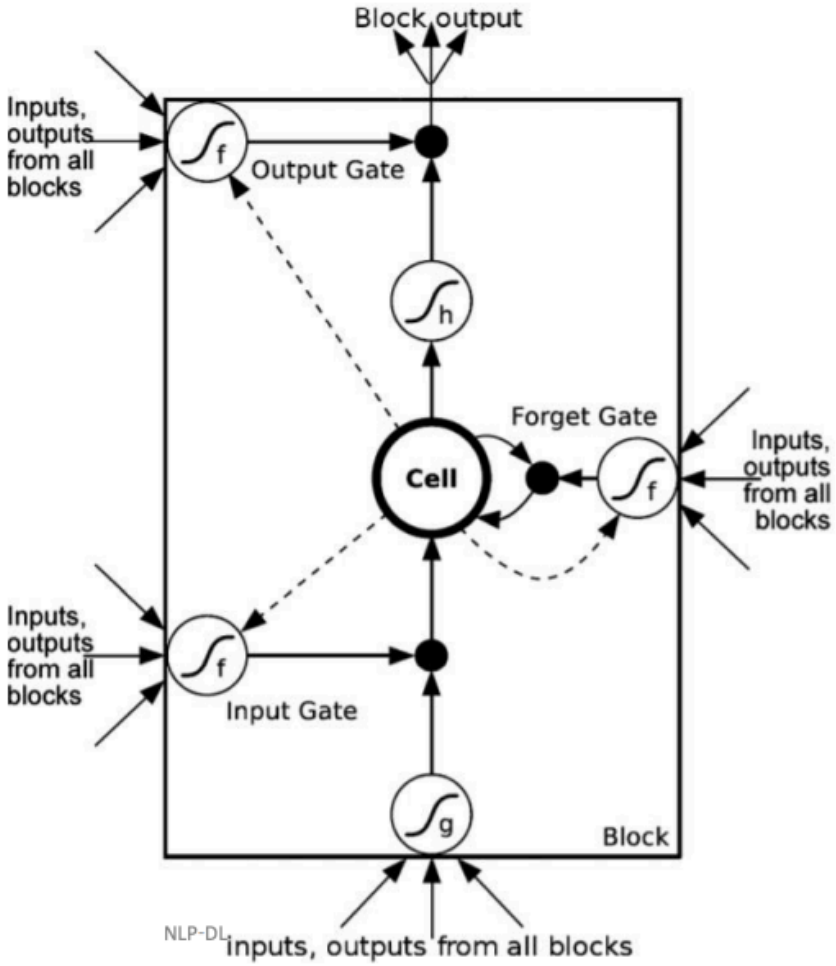
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

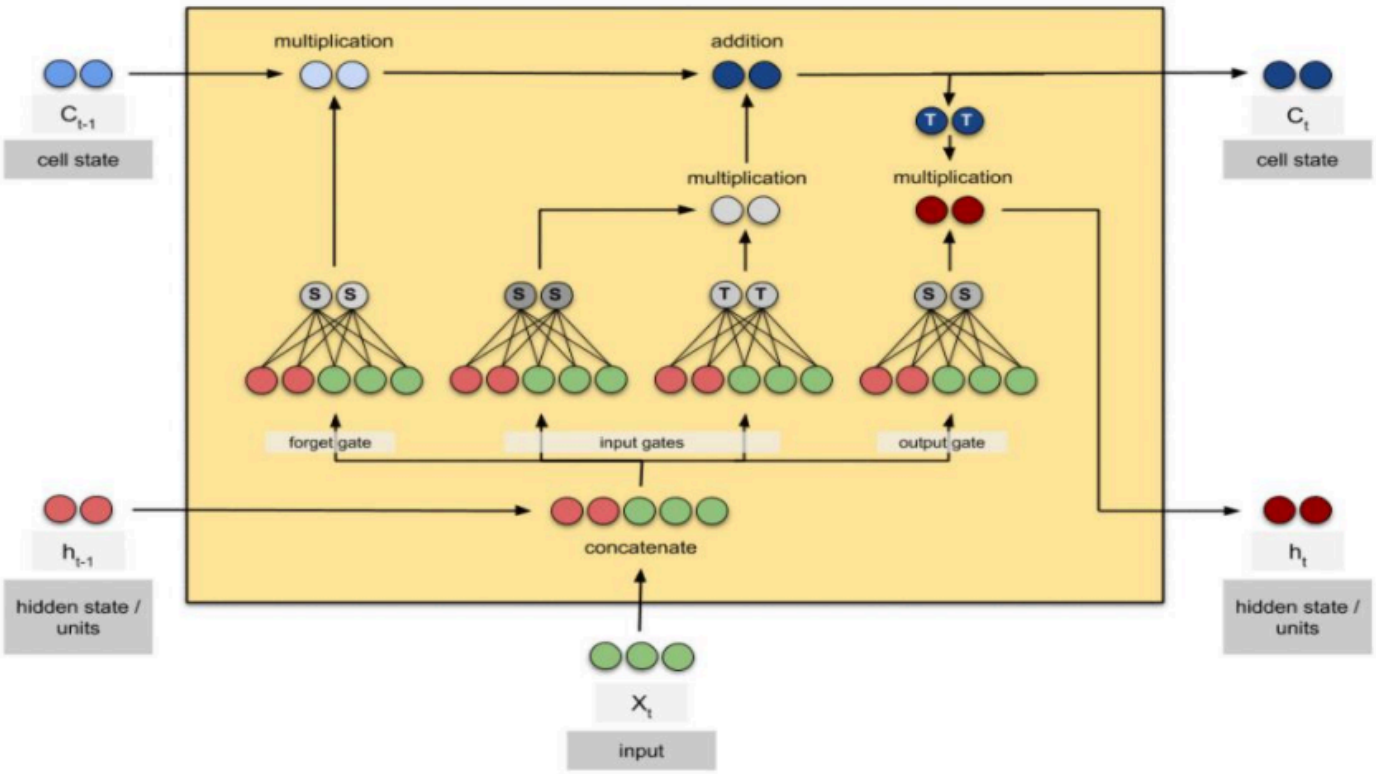
Conclusion

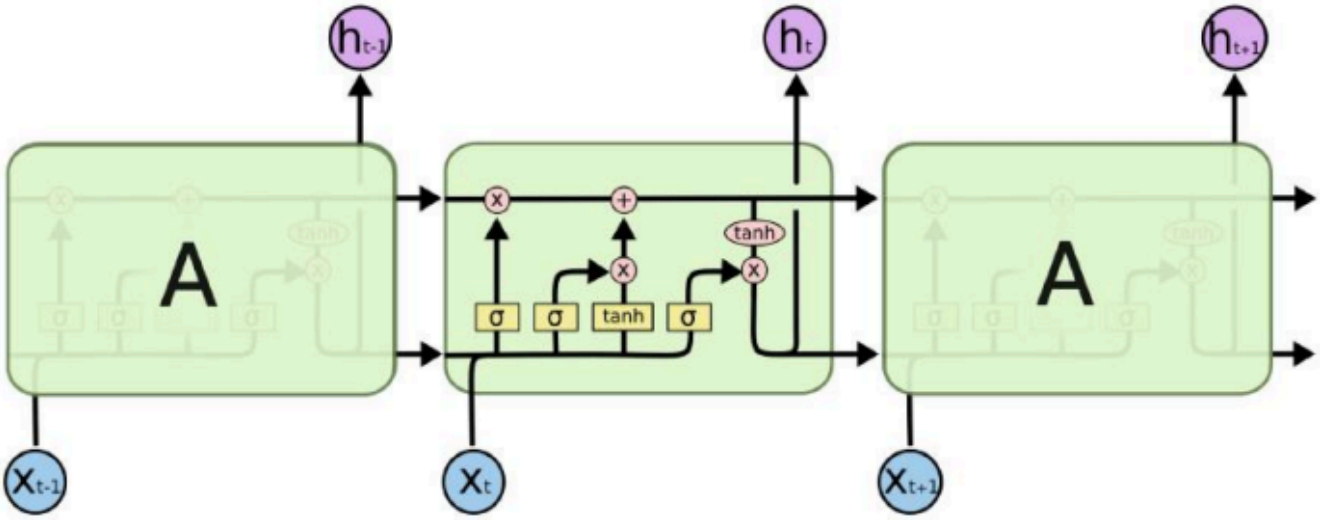
- LSTMs **overcome vanishing gradient problems** by maintaining long-term dependencies in sequential data.
- Their **gate mechanisms (forget, input, output)** make them **powerful** for tasks like text prediction, translation, speech recognition, and time series forecasting.
- The **step-by-step processing** allows LSTMs to dynamically adjust their memory, making them **superior to vanilla RNNs** for handling **long-range dependencies**.

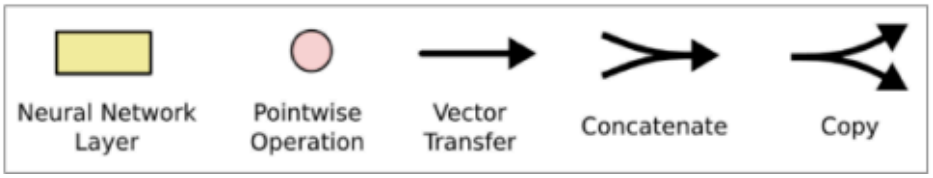

Input: Does $x^{(t)}$ matter?
$h^{(t-1)}$
$U^{(i)}$
$x^{(t)}$
$W^{(i)}$
σ
$i^{(t)}$
Output/Exposure:
How much $c^{(t)}$ should be exposed?
$U^{(o)}$
$h^{(t-1)}$
σ
$W^{(o)}$
$x^{(t)}$
$o^{(t)}$
New memory: Compute new memory
$h^{(t-1)}$
$U^{(c)}$
tanh
$x^{(t)}$
$W^{(c)}$
$\tilde{c}^{(t)}$
$c^{(t)}$
tanh
$h^{(t)}$
$c^{(t-1)}$
Forget: Should $c^{(t-1)}$ be forgotten?
$h^{(t-1)}$
$U^{(f)}$
σ
$x^{(t)}$
$W^{(f)}$
$f^{(t)}$

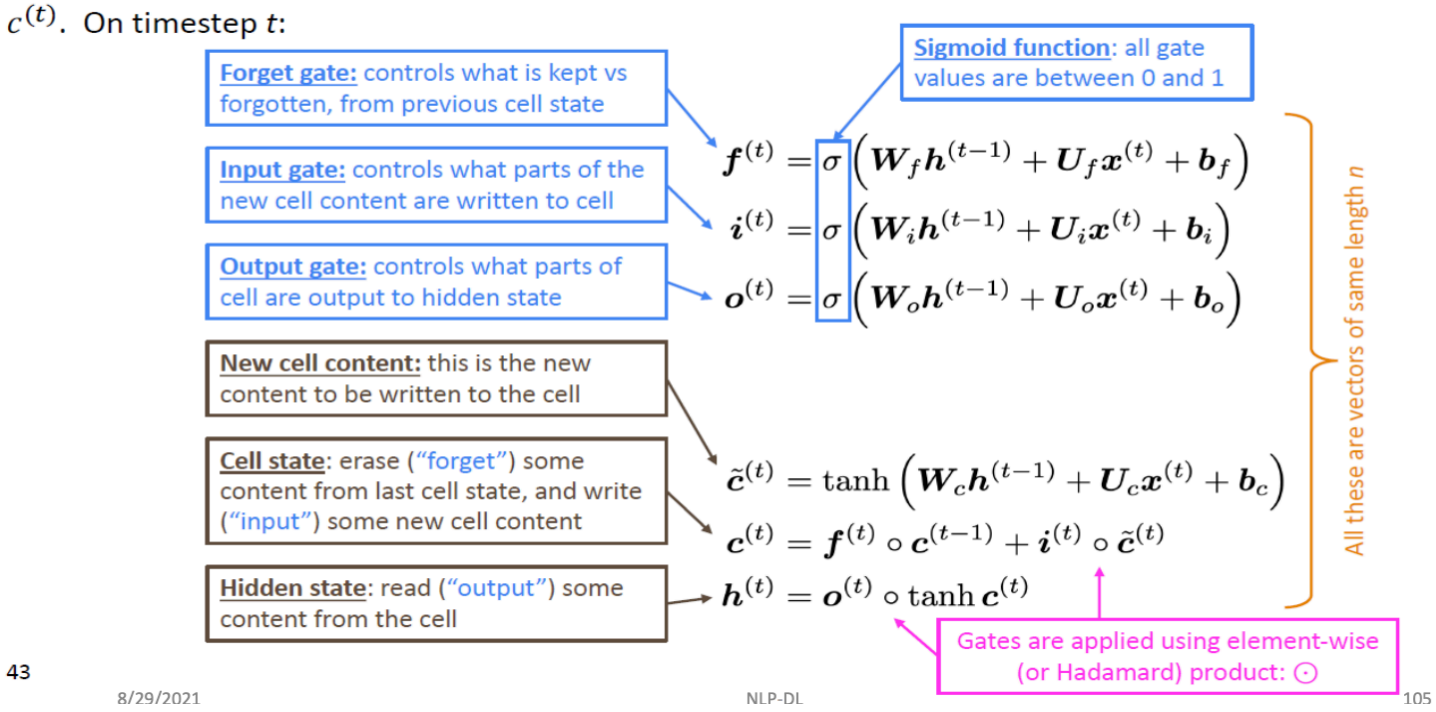

Block output
Inputs, outputs from all blocks
f
Output Gate
h
Forget Gate
Cell
f
Inputs, outputs from all blocks
Inputs, outputs from all blocks
f
Input Gate
g
Block
inputs, outputs from all blocks

LSTM Gates operation









GRU

Gated Recurrent Units (GRU) & Variants of LSTMs

1. Overview of RNN Variants

Different architectures of **Recurrent Neural Networks (RNNs)** have been developed to address issues like **vanishing gradients** and **long-term dependencies**:

Architecture	Description
Simple RNN	Uses a direct multiplication of input (xt) and previous hidden state (ht−1), passed through a **tanh** activation function. No gating mechanisms.
Gated Recurrent Unit (GRU)	Introduces an **Update Gate** that decides whether to pass the previous output (ht−1) to the next cell or update it.
Long Short-Term Memory (LSTM)	Extends GRUs by adding **two more gates** (Forget & Output gates) alongside the Update Gate. Provides more control over information flow.

---

2. Comparison: LSTM vs. GRU

Feature	LSTM	GRU
Number of Gates	3 (Forget, Input, Output)	2 (Update, Reset)

Feature	LSTM	GRU
Hidden State Management	Uses both **cell state** and **hidden state**	Uses **only hidden state** (no separate cell state)
Training Complexity	Higher (8 sets of weights to learn)	Lower (fewer parameters, faster training)
Memory Efficiency	More parameters lead to **higher memory usage**	Fewer parameters, **more memory-efficient**
Performance	Handles long dependencies better	Faster and computationally **less expensive**
Use Case Preference	Text generation, complex language models	Real-time applications, speech recognition

---

3. GRU Architecture

- GRUs simplify LSTMs by reducing the number of **gates** and eliminating the **cell state**.
- Instead of three gates (Forget, Input, Output), **GRUs use only two gates**:
    1. **Update Gate (z)** – Decides **what information to keep** from the previous state and **what to update**.
    2. **Reset Gate (r)** – Determines **how much past information to forget**.

---

4. Working of GRUs

Step-by-Step Breakdown

1. **Reset Gate (r)**
    - Decides which aspects of the previous hidden state should be ignored.
    - Element-wise multiplication of rrr with the previous hidden state controls this.
2. **Intermediate Hidden State**
    - The masked previous hidden state is used to compute a new intermediate representation.
3. **Update Gate (z)**
    - Determines the final hidden state by interpolating between the **previous** and **new** hidden states.
    - If z is close to **0**, the network **forgets** past information.
    - If z is close to **1**, past information is **retained**.

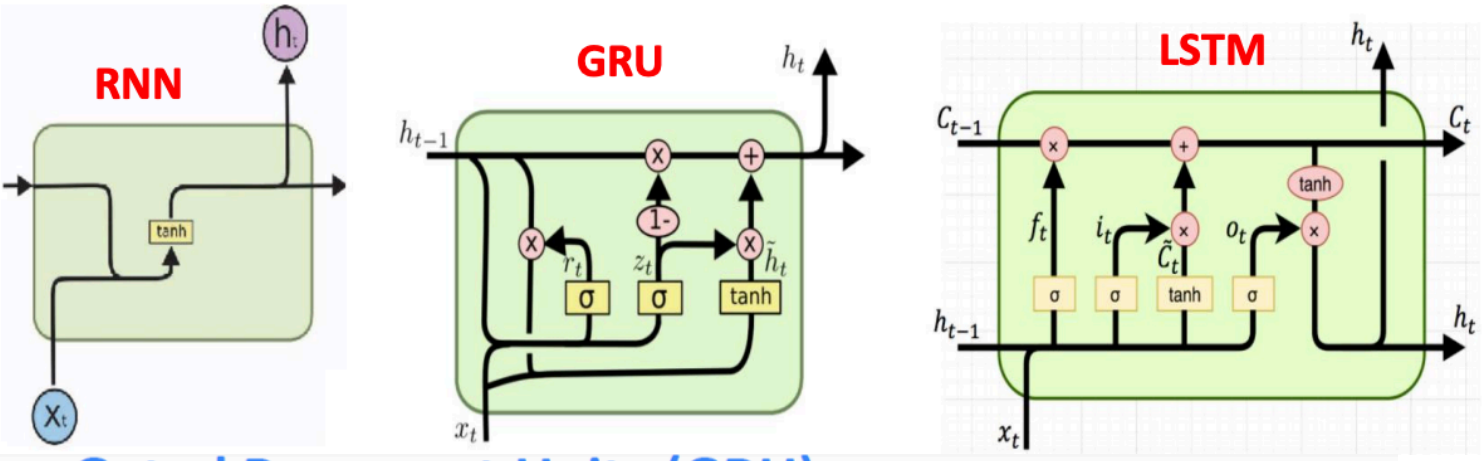
---

5. Advantages of GRUs

- **Fewer parameters than LSTMs**, making them **faster to train**.
- **Performs well on smaller datasets** where **LSTMs might overfit**.
- **Less computational complexity**, making them suitable for **real-time applications**.

6. Key Observations

- **LSTMs provide more fine-grained control** over information retention, making them better for **complex sequential tasks**.
- **GRUs are computationally cheaper**, making them useful for **faster training and real-time tasks**.
- **There is no clear winner** between GRUs and LSTMs; the choice depends on the **task requirements**.



Gated Recurrent Units (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep t we have input $\boldsymbol{x}^{(t)}$ and hidden state $\boldsymbol{h}^{(t)}$ (no cell state).

Update gate: controls what parts of hidden state are updated vs preserved

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{W}_u\boldsymbol{h}^{(t-1)} + \boldsymbol{U}_u\boldsymbol{x}^{(t)} + \boldsymbol{b}_u\right)$$

Reset gate: controls what parts of previous hidden state are used to compute new content

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{W}_r\boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r\boldsymbol{x}^{(t)} + \boldsymbol{b}_r\right)$$

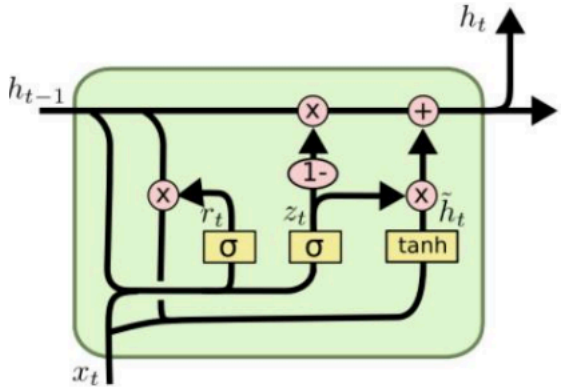
New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{W}_h(\boldsymbol{r}^{(t)} \circ \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}_h\boldsymbol{x}^{(t)} + \boldsymbol{b}_h\right)$$

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{u}^{(t)}) \circ \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \circ \tilde{\boldsymbol{h}}^{(t)}$$

How does this solve vanishing gradient?
Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)



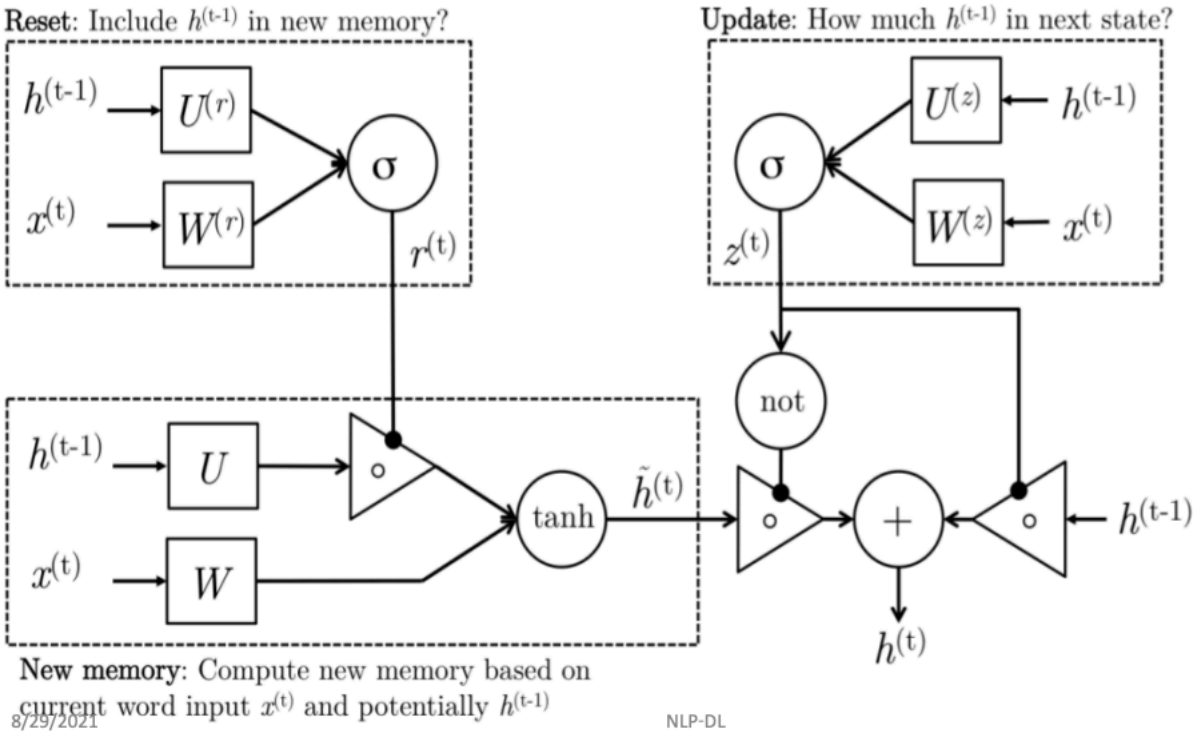
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

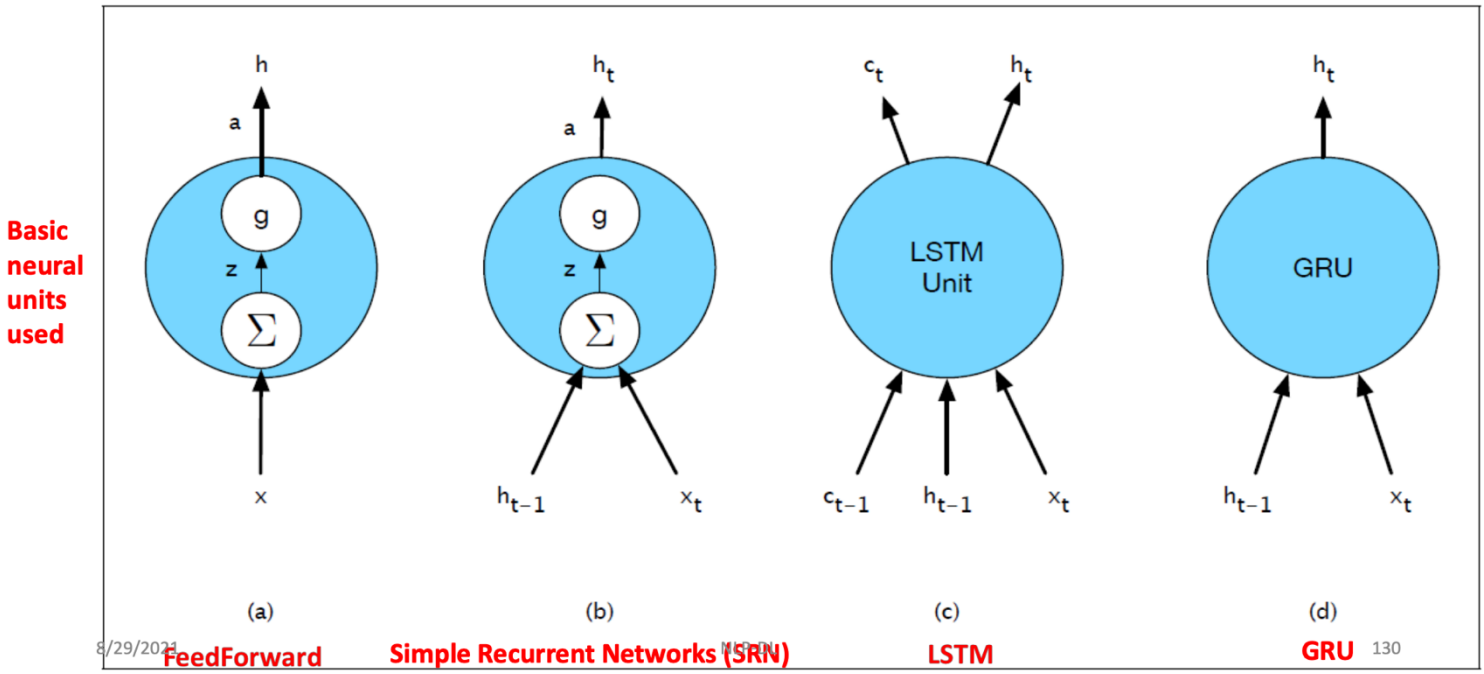
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

The detailed internals of a GRU





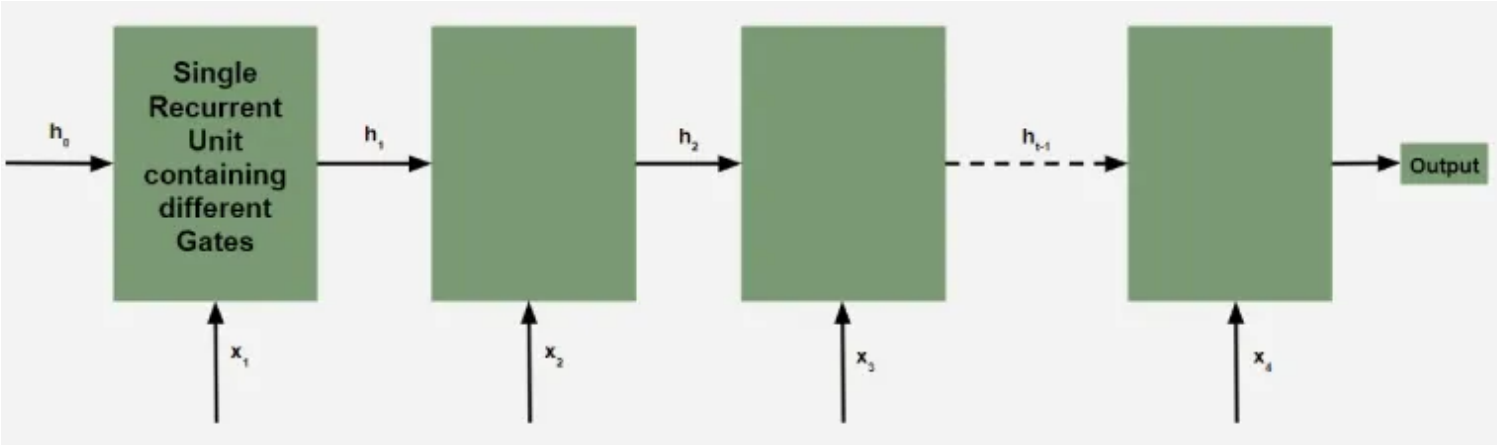
Traditional RNNs struggle with capturing long-term dependencies due to the vanishing gradient problem architectures like Long Short-Term Memory (LSTM) networks were developed to overcome this limitation.

However LSTMs are very complex structure with higher computational cost. To overcome this Gated Recurrent Unit (GRU) where introduced which uses LSTM architecture by merging its gating mechanisms offering a more efficient solution for many sequential tasks without sacrificing performance. In this article we'll learn more about them.

Getting Started with Gated Recurrent Units (GRU)

Gated Recurrent Units (GRUs) are a type of RNN. The core idea behind GRUs is to use gating mechanisms to selectively update the hidden state at each time step allowing them to remember important information while discarding irrelevant details. GRUs aim to simplify the LSTM architecture by merging some of its components and focusing on just two main gates: the update gate and the reset gate.

Core Structure of GRUs



The GRU consists of two main gates:

1. Update Gate (zt): This gate decides how much information from previous hidden state should be retained for the next time step.

2. Reset Gate (rt): This gate determines how much of the past hidden state should be forgotten.

These gates allow GRU to control the flow of information in a more efficient manner compared to traditional RNNs which solely rely on hidden state.

Equations for GRU Operations

The internal workings of a GRU can be described using following equations:

1. Reset gate:

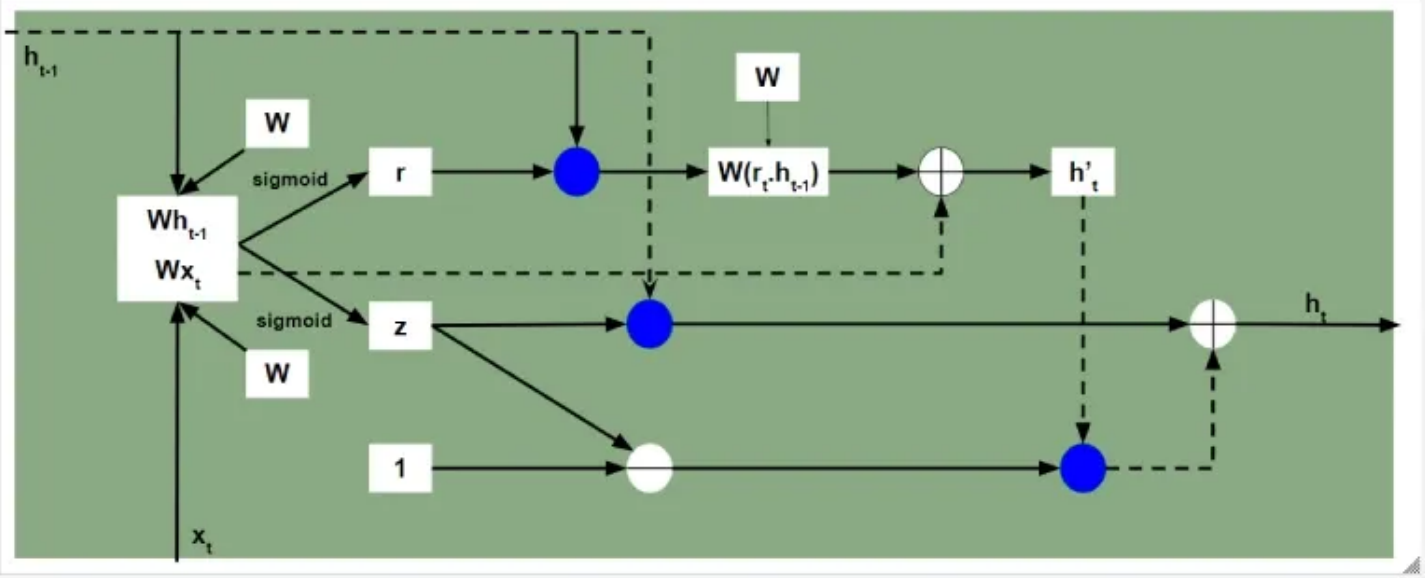
$rt=\sigma(Wr\cdot[ht-1,xt])$

The reset gate determines how much of the previous hidden state $ht-1$ should be forgotten.

2. Update gate:

$zt=\sigma(Wz\cdot[ht-1,xt])$

The update gate controls how much of the new information xt should be used to update the hidden state.



3. Candidate hidden state:

$ht'=\tanh(Wh\cdot[rt\cdot ht-1,xt])$

This is the potential new hidden state calculated based on the current input and the previous hidden state.

4. Hidden state:

$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot h_t'$

The final hidden state is a weighted average of the previous hidden state h_{t-1} and the candidate hidden state h_t' based on the update gate z_t.

How GRUs Solve the Vanishing Gradient Problem

Like LSTMs, GRUs were designed to address the vanishing gradient problem which is common in traditional RNNs. GRUs help mitigate this issue by using gates that regulate the flow of gradients during training ensuring that important information is preserved and that gradients do not shrink excessively over time. By using these gates, GRUs maintain a balance between remembering important past information and learning new, relevant data.

GRU vs LSTM

GRUs are more computationally efficient because they combine the forget and input gates into a single update gate. GRUs do not maintain an internal cell state as LSTMs do, instead they store information directly in the hidden state making them simpler and faster.

Feature	LSTM (Long Short-Term Memory)	GRU (Gated Recurrent Unit)
Gates	3 (Input, Forget, Output)	2 (Update, Reset)
Cell State	Yes it has cell state	No (Hidden state only)
Training Speed	Slower due to complexity	Faster due to simpler architecture
Computational Load	Higher due to more gates and parameters	Lower due to fewer gates and parameters
Performance	Often better in tasks requiring long-term memory	Performs similarly in many tasks with less complexity