Recurrent Neural Network (RNN)

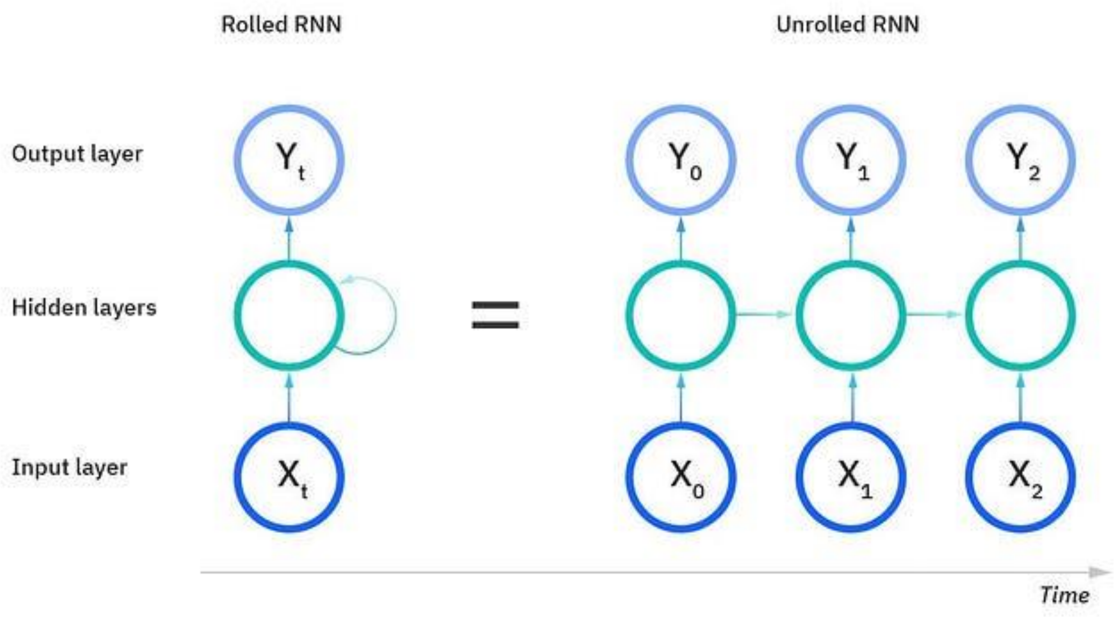
In the world of machine learning and deep learning, we’ve made significant strides in handling data with fixed input and output sizes. Traditional neural networks, like feedforward and convolutional neural networks, excel at tasks where inputs and outputs are of a predetermined size, such as image classification. However, they fall short when dealing with sequential data or data with varying input and output lengths.

Imagine trying to generate a coherent sentence. In any language, sentences aren't just random words thrown together; they follow grammatical rules and convey specific meanings. To generate meaningful sentences, a model needs to understand the sequence and context of words. This is where **Recurrent Neural Networks (RNNs)** come into play.

RNNs are designed to handle sequential data by taking the output from previous steps and using it as input for the current step. They are essential for tasks that involve time series data or any data where the order matters.

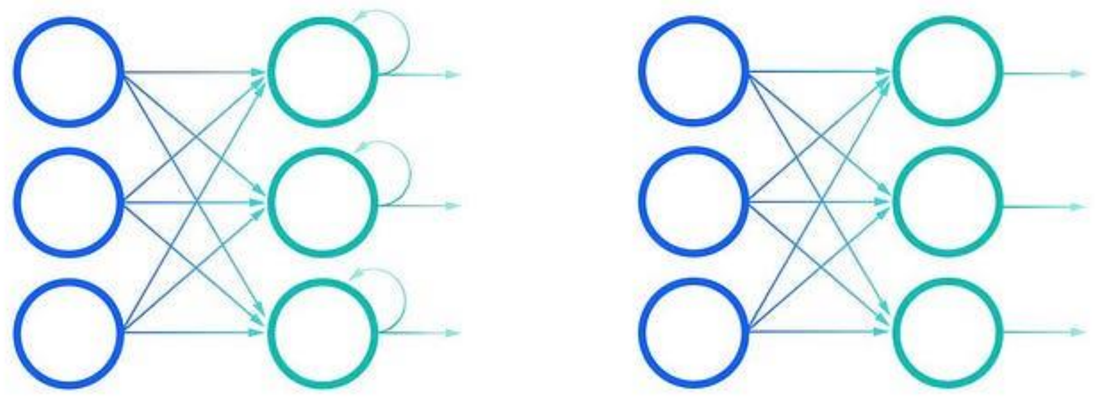
Feedforward vs. Recurrent Neural Networks

Feedforward Neural Networks



Feedforward neural networks process inputs in a single direction — from input to output — without any notion of sequence or memory. They assume that all inputs and outputs are independent of each other. This makes them unsuitable for tasks where the context or sequence of data points is crucial.

Recurrent Neural Networks



RNNs, on the other hand, are designed to recognize patterns in sequences of data. They have a "memory" that captures information about what has been calculated so far. This memory allows RNNs to use prior inputs to influence the current output.

Example: Consider the idiom "feeling under the weather., which is commonly used when someone is ill, to aid us in the explanation of RNNs. In order for the idiom to make sense, it needs to be expressed in that specific order. An RNN can understand this sequence because it processes each word in the context of the previous words.

Looking at the visuals above, the "rolled" visual of the RNN represents the whole neural network, or rather the entire predicted phrase, like “feeling under the weather.” The “unrolled” visual represents the individual layers, or time steps, of the neural network. Each layer maps to a single word in that phrase, such as “weather”. Prior inputs, such as “feeling” and “under”, would be represented as a hidden state in the third timestep to predict the output in the sequence, “the”.

Challenges in RNNs

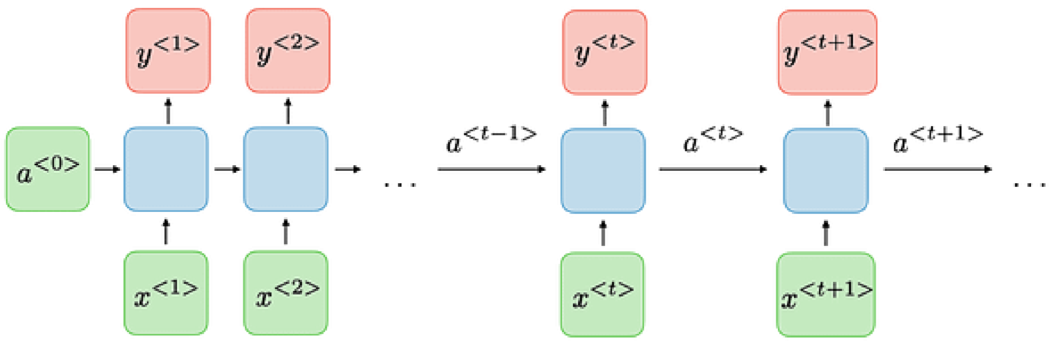
While RNNs are powerful, they come with their own set of challenges, primarily the **exploding and vanishing gradient** problems. These issues are defined by the size of the gradient, which is the slope of the loss function along the error curve.

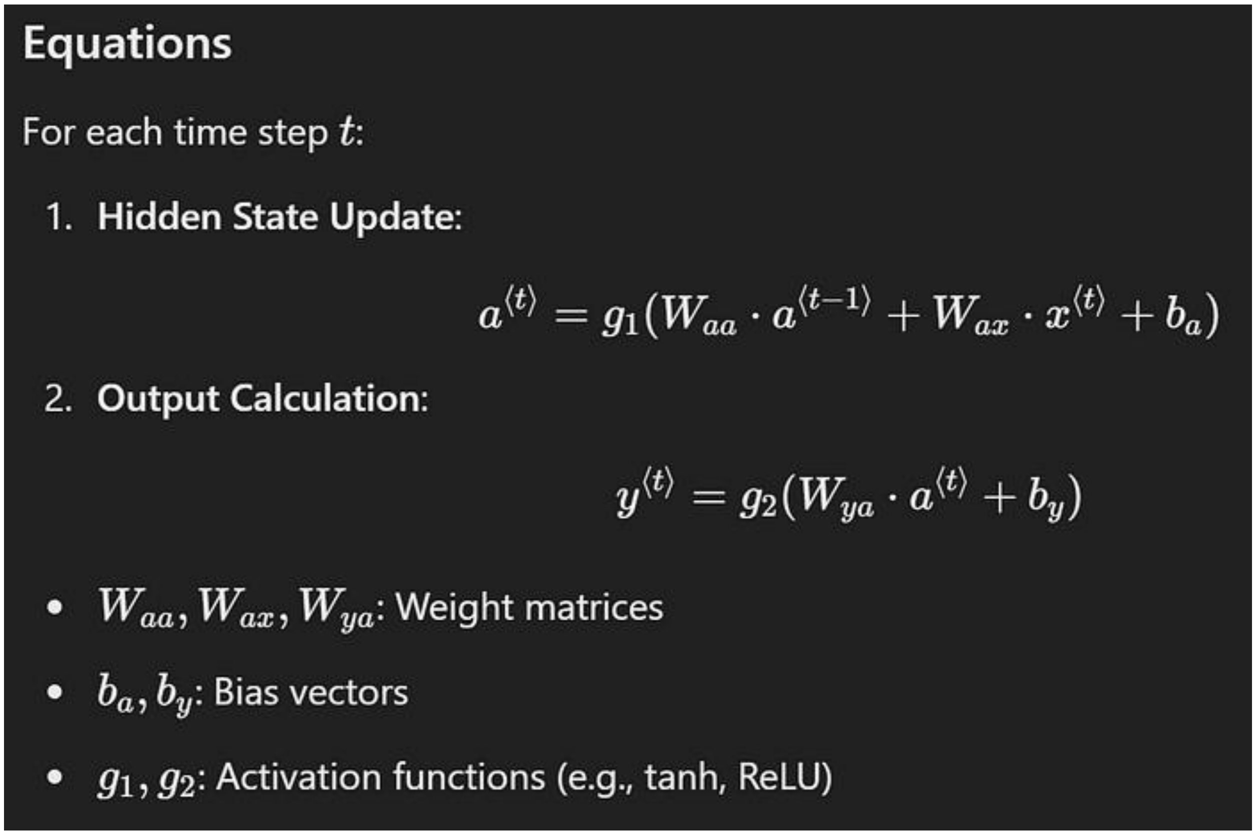
When the gradient is too small, it continues to become smaller, updating the weight parameters until they become insignificant — i.e. 0. When that occurs, the algorithm is no longer learning.
Exploding gradients occur when the gradient is too large, creating an unstable model. In this case, the model weights will grow too large, and they will eventually be represented as NaN.

One solution to these issues is to reduce the number of hidden layers within the neural network, eliminating some of the complexity in the RNN model.

How RNN works?

An RNN processes a sequence of inputs, maintaining a hidden state that captures information about the sequence.





Equations

For each time step t:

1. **Hidden State Update:**

$$a^{\langle t \rangle} = g_1(W_{aa} \cdot a^{\langle t-1 \rangle} + W_{ax} \cdot x^{\langle t \rangle} + b_a)$$

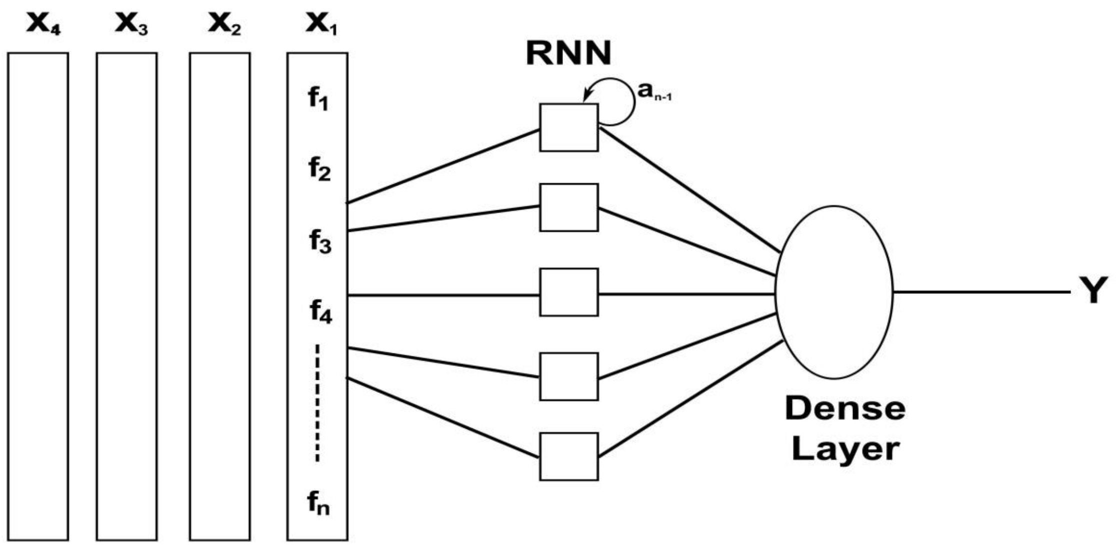
2. **Output Calculation:**

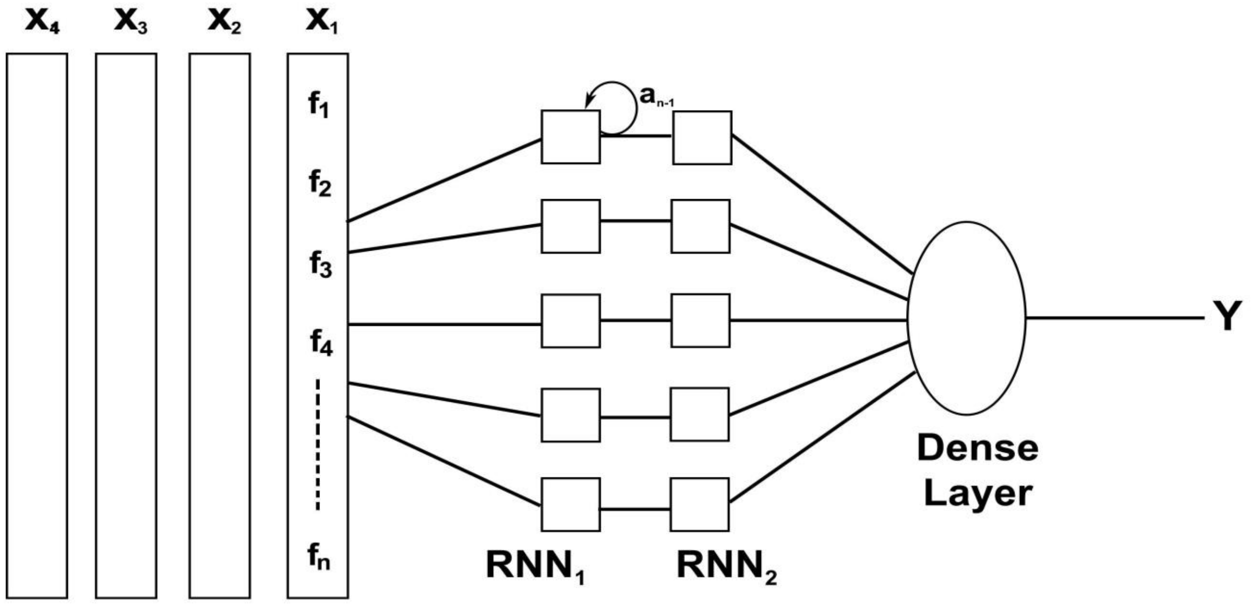
$$y^{\langle t \rangle} = g_2(W_{ya} \cdot a^{\langle t \rangle} + b_y)$$

- W_{aa}, W_{ax}, W_{ya}: Weight matrices
- b_a, b_y: Bias vectors
- g_1, g_2: Activation functions (e.g., tanh, ReLU)

RNN Architectures

RNNs can be structured in various ways to suit different tasks:





(Upper image) X_1, X_2, X_3 sequence will be passed through RNN and we would get Y_1, Y_2, Y_3. But we care only about the final output Y, not the intermediate outputs. (Lower image) we need the sequence output from RNN1 so that it can become the sequential input given inside RNN2.

RNNs can be structured in various ways to suit different tasks:

One-to-One: Standard neural network (e.g., image classification).

One-to-Many: Single input, sequence output (e.g., image captioning).

Many-to-One: Sequence input, single output (e.g., sentiment analysis).

Many-to-Many (Same Length): Sequence input and output of the same length (e.g., named entity recognition).

Many-to-Many (Different Length): Sequence input and output of different lengths (e.g., machine translation).

Applications of RNNs

RNNs are widely used in fields that require sequence processing:

Natural Language Processing (NLP): Language modeling, text generation.

Speech Recognition: Transcribing spoken words into text.

Machine Translation: Translating text from one language to another.

Time Series Prediction: Stock market prediction, weather forecasting.

Music Generation: Composing music by predicting sequences of notes.

Variants of RNNs

1. Bidirectional Recurrent Neural Networks (BRNN)

BRNNs process data in both forward and backward directions, allowing the network to have both past and future context.

Use-Case: Useful when the output at time ***t*** depends on future inputs.

2. Long Short-Term Memory Networks (LSTM)

LSTMs are designed to handle the vanishing gradient problem by introducing a memory cell that can maintain information over long periods.

Components:

Cell State: Stores long-term dependencies.
Gates:

— — → **Input Gate**: Controls how much new information flows into the cell.

— — → **Forget Gate**: Decides what information to discard from the cell.

— — → **Output Gate**: Determines what information to output.

Use-Case: Ideal for tasks requiring long-term memory, like essay writing or speech recognition.

3. Gated Recurrent Units (GRU)

GRUs are a simplified version of LSTMs with only two gates: reset gate and update gate.

Advantages:

Fewer parameters than LSTM, making them faster to train.

Comparable performance to LSTMs on many tasks.

REF:

https://medium.com/@RobuRishabh/recurrent-neural-network-rnn-8412b9abd755

https://www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn/

LSTM: Architecture, Pros and Cons, and Implementation

What is LSTM and How it works?

LSTM stands for Long Short-Term Memory, and it is a type of recurrent neural network (RNN) architecture that is commonly used in natural language processing, speech recognition, and other sequence modeling tasks.

Unlike a traditional RNN, which has a simple structure of input, hidden state, and output, an LSTM has a more complex structure with additional memory cells and gates that allow it to selectively remember or forget information from previous time steps.

An LSTM cell consists of several components:

Input gate: This gate controls the flow of information from the current input and the previous hidden state into the memory cell.

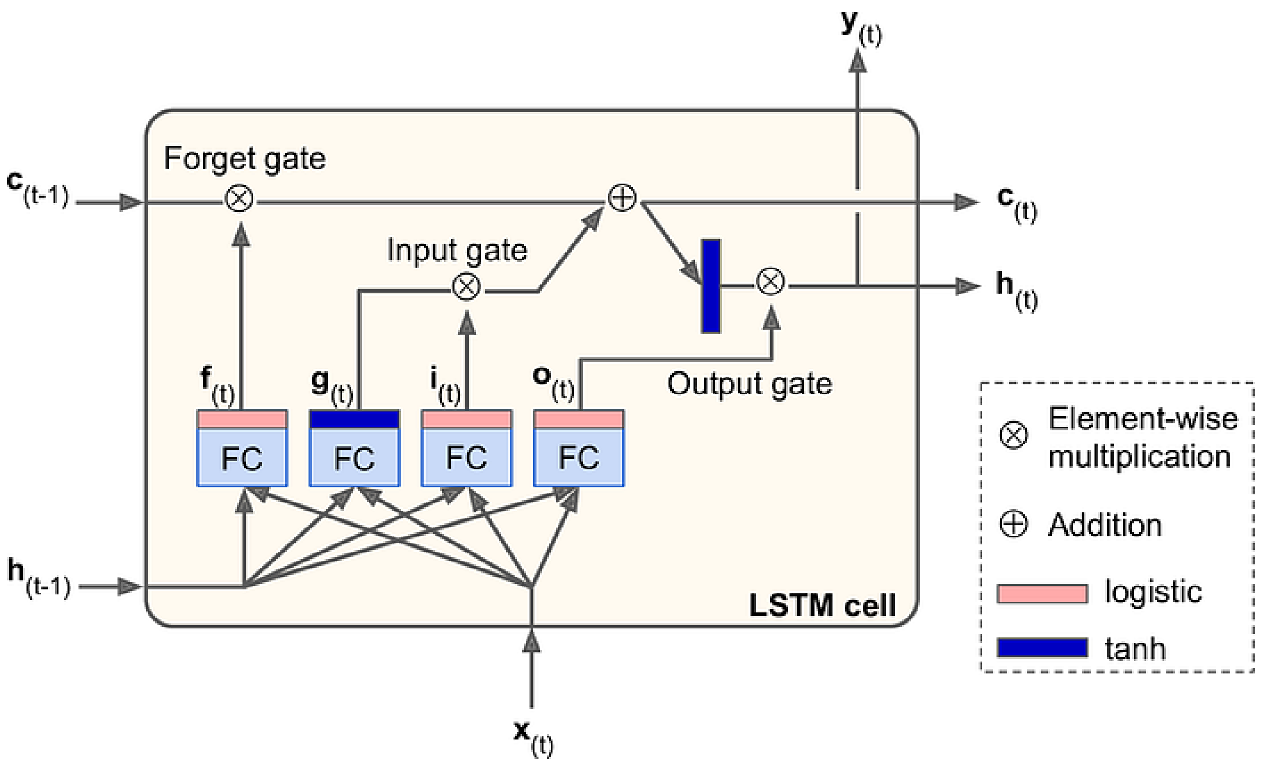
Forget gate: This gate controls the flow of information from the previous memory cell to the current memory cell. It allows the LSTM to selectively forget or remember information from previous time steps.

Memory cell: This is the internal state of the LSTM. It stores information that can be selectively modified by the input and forget gates.
Output gate: This gate controls the flow of information from the memory cell to the current hidden state and output.

During the forward pass, the LSTM takes in a sequence of inputs and updates its memory cell and hidden state at each time step. The input gate and forget gate use sigmoid functions to decide how much information to let into or out of the memory cell, while the output gate uses a sigmoid function and a tanh function to produce the current hidden state and output.

The LSTM's ability to selectively remember or forget information from previous time steps makes it well-suited for tasks that require modeling long-term dependencies, such as language translation or sentiment analysis.

LSTM Architecture



Source

The architecture of an LSTM can be visualized as a series of repeating "blocks" or "cells", each of which contains a set of interconnected nodes. Here's a high-level overview of the architecture:

Input: At each time step, the LSTM takes in an input vector, x_t, which represents the current observation or token in the sequence.

Hidden State: The LSTM maintains a hidden state vector, h_t, which represents the current "memory" of the network. The hidden state is initialized to a vector of zeros at the beginning of the sequence.

Cell State: The LSTM also maintains a cell state vector, c_t, which is responsible for storing long-term information over the course of the sequence. The cell state is initialized to a vector of zeros at the beginning of the sequence.

Gates: The LSTM uses three types of gates to control the flow of information through the network:

a). Forget Gate: This gate takes in the previous hidden state, h_{t-1}, and the. current input, x_t, and outputs a vector of values between 0 and 1 that represent how much of the previous cell state to "forget" and how much to retain. This gate allows the LSTM to selectively "erase" or "remember" information from the previous time step.

b). Input Gate: This gate takes in the previous hidden state, h_{t-1}, and the current input, x_t, and outputs a vector of values between 0 and 1 that represent how much of the current input to add to the cell state. This gate allows the LSTM to selectively "add" or "discard" new information to the cell state.

c). Output Gate: This gate takes in the previous hidden state, h_{t-1}, and the current input, x_t, and the current cell state, c_t, and outputs a vector of values between 0 and 1 that represent how much of the current cell state to output as the current hidden state, h_t. This gate allows the LSTM to selectively "focus" or "ignore" certain parts of the cell state when computing the output.

5. **Output**: At each time step, the LSTM outputs a vector, y_t, that represents the network's prediction or encoding of the current input.

The combination of the cell state, hidden state, and gates allows the LSTM to selectively "remember" or "forget" information over time, making it well-suited for tasks that require modeling long-term dependencies or sequences.

Equations of each gates

Here are the equations for each of the three gates in an LSTM:

Forget Gate:

The forget gate takes as input the previous hidden state, h_{t-1}, and the current input, x_t, and outputs a vector of values between 0 and 1 that represent how much of the previous cell state to “forget” and how much to retain. The equation for the forget gate is:

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

where:

σ is the sigmoid function

W_f is the weight matrix for the forget gate

$[h_{t-1}, x_t]$ is the concatenation of the previous hidden state and the current input

b_f is the bias vector for the forget gate

f_t is the vector of forget gate values for the current time step

2. **Input Gate:**

The input gate takes as input the previous hidden state, h_{t-1}, and the current input, x_t, and outputs a vector of values between 0 and 1 that represent how much of the current input to add to the cell state. The equation for the input gate is:

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

where:

σ is the sigmoid function

W_i and W_c are the weight matrices for the input gate

$[h_{t-1}, x_t]$ is the concatenation of the previous hidden state and the current input

b_i and b_c are the bias vectors for the input gate

i_t is the vector of input gate values for the current time step

$\sim C_t$ is the candidate cell state vector for the current time step, which is produced by applying the tanh activation function to a linear combination of the previous hidden state and the current input.

3. **Output Gate:**

The output gate takes as input the previous hidden state, h_{t-1}, the current input, x_t, and the current cell state, c_t, and outputs a vector of values between 0 and 1 that represent how much of the current cell state to output as the current hidden state, h_t. The equation for the output gate is:

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

where:

σ is the sigmoid function

W_o is the weight matrix for the output gate

[h_{t-1}, x_t] is the concatenation of the previous hidden state and the current input

b_o is the bias vector for the output gate

o_t is the vector of output gate values for the current time step

h_t is the current hidden state, which is produced by applying the tanh activation function to the current cell state and multiplying it element-wise with the output gate values.

Python Implementation

Here is a simple implementation of LSTM in Python using the Keras library:

```
from keras.models import Sequential
from keras.layers import LSTM, Dense

# define the LSTM model
model = Sequential()
model.add(LSTM(100, input_shape=(timesteps, features)))
model.add(Dense(1, activation='sigmoid'))

# compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
# fit the model to the training data
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
```

In this example, we first import the necessary modules from Keras. We then define the LSTM model using the `Sequential` API. The LSTM layer is added using the `LSTM` function, which takes as input the number of units (100 in this case) and the input shape (a tuple of `(timesteps, features)`). We then add a dense output layer with a sigmoid activation function.

We then compile the model using the `compile` function, specifying the loss function (binary crossentropy), optimizer (Adam), and metrics (accuracy). Finally, we fit the model to the training data using the `fit` function, specifying the number of epochs, batch size, and validation data.

Note that this is just a simple example, and there are many variations and customization options for LSTM models in Keras.

Pros and Cons of using LSTM

Pros:

Modeling long-term dependencies: LSTMs are well-suited for modeling long-term dependencies in sequential data, since they can selectively “remember” or “forget” information over time. This makes them useful for tasks like speech recognition, machine translation, and text generation.

Robustness to noisy data: LSTMs are more robust to noisy or missing data than other types of recurrent neural networks, since they can selectively filter out irrelevant or noisy information using the forget gate.

Flexibility: LSTMs can be used for a wide variety of tasks, including classification, regression, and sequence-to-sequence learning.

Interpretability: Since LSTMs maintain a cell state vector that represents the network’s “memory” at each time step, they can be more interpretable than other types of recurrent neural networks.

Cons:

Computationally expensive: LSTMs can be computationally expensive to train and evaluate, especially for long sequences or large datasets.

Prone to overfitting: LSTMs can be prone to overfitting on small datasets, especially if the model architecture is too complex.

Hyperparameter tuning: LSTMs have many hyperparameters that need to be tuned in order to achieve optimal performance, including the number of hidden units, the learning rate, and the dropout rate.

Data requirements: LSTMs require a large amount of training data to learn complex patterns in the data. If there is not enough data available, the model may not perform well.

What is LSTM? Introduction to Long Short-Term Memory

Long Short-Term Memory Networks or LSTM in deep learning, is a sequential neural network that allows information to persist. It is a special type of Recurrent Neural Network which is capable of handling the vanishing gradient problem faced by RNN. LSTM was designed by Hochreiter and Schmidhuber that resolves the problem caused by traditional rnns and machine learning algorithms. LSTM Model can be implemented in Python using the Keras library.

Let’s say while watching a video, you remember the previous scene, or while reading a book, you know what happened in the earlier chapter. RNNs work similarly; they remember the previous information and use it for processing the current input. The shortcoming of RNN is they cannot remember long-term dependencies due to vanishing gradient. LSTMs are explicitly designed to avoid long-term dependency problems.

Table of contents

What is LSTM?

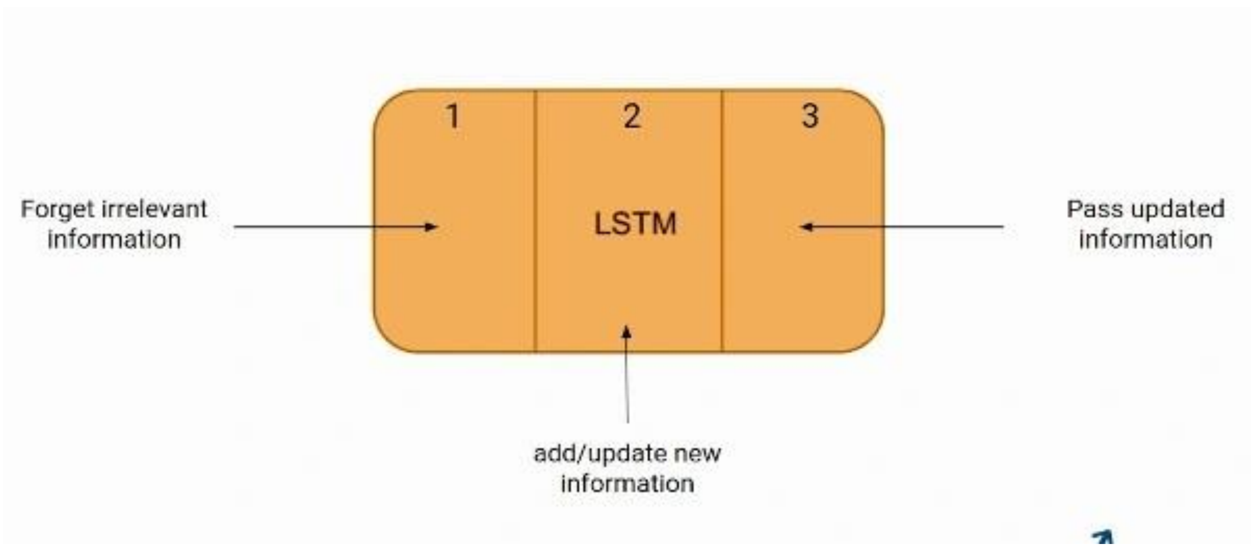
LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning. It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks.

Unlike traditional neural networks, LSTM incorporates feedback connections, allowing it to process entire sequences of data, not just individual data points. This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.

LSTM has become a powerful tool in artificial intelligence and deep learning, enabling breakthroughs in various fields by uncovering valuable insights from sequential data.

LSTM Architecture

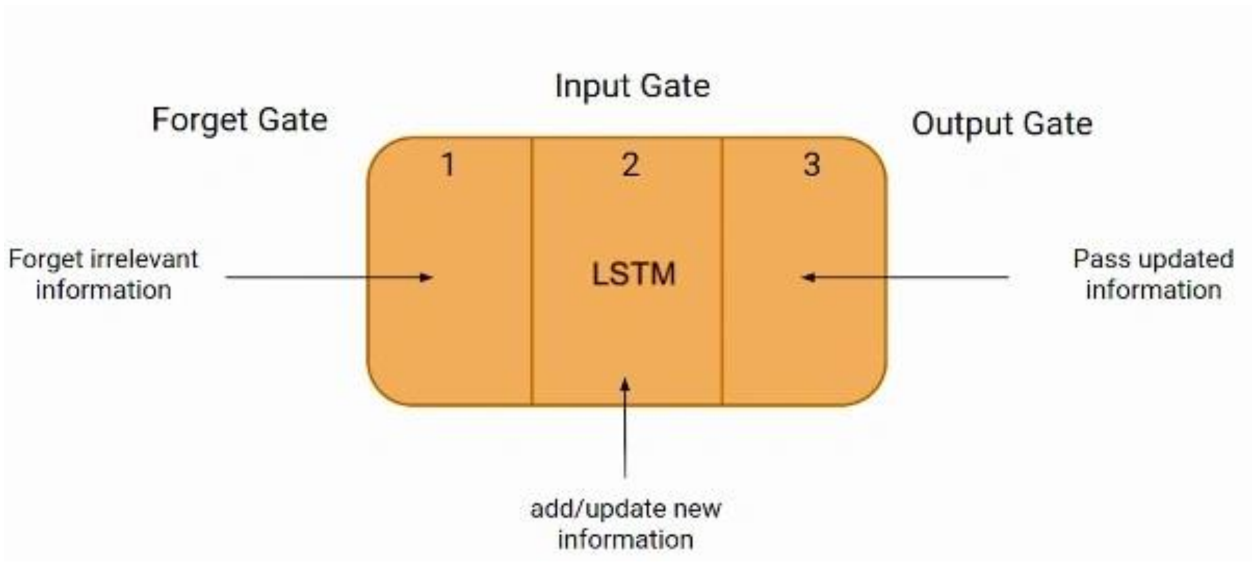
In the introduction to long short-term memory, we learned that it resolves the vanishing gradient problem faced by RNN, so now, in this section, we will see how it resolves this problem by learning the architecture of the LSTM. At a high level, LSTM works very much like an RNN cell. Here is the internal functioning of the **LSTM network**. The LSTM network architecture consists of three parts, as shown in the image below, and each part performs an individual function.



The Logic Behind LSTM

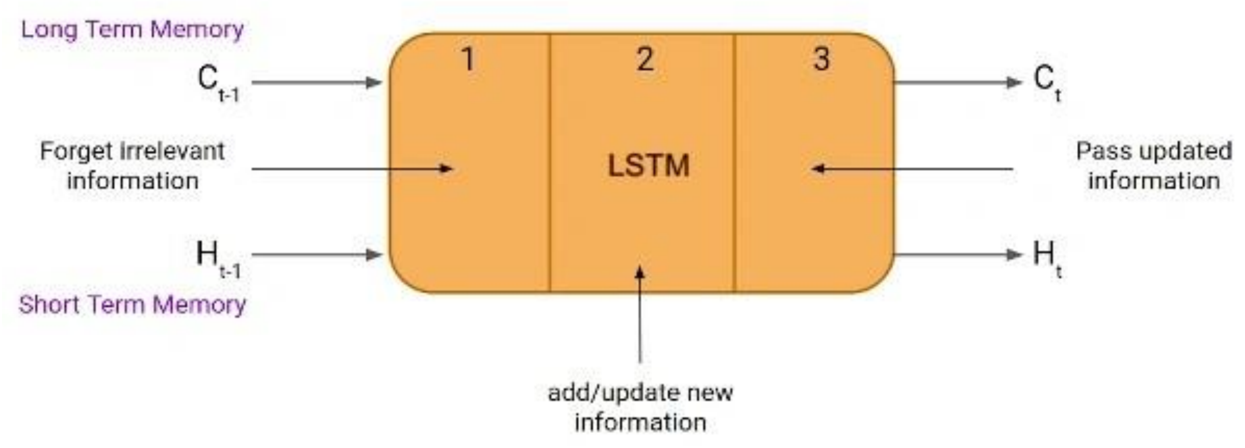
The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten. In the second part, the cell tries to learn new information from the input to this cell. At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp. This one cycle of LSTM is considered a single-time step.

These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or lstm cell. The first gate is called Forget gate, the second gate is known as the Input gate, and the last one is the Output gate. An LSTM unit that consists of these three gates and a memory cell or lstm cell can be considered as a layer of neurons in traditional feedforward neural network, with each neuron having a hidden layer and a current state.

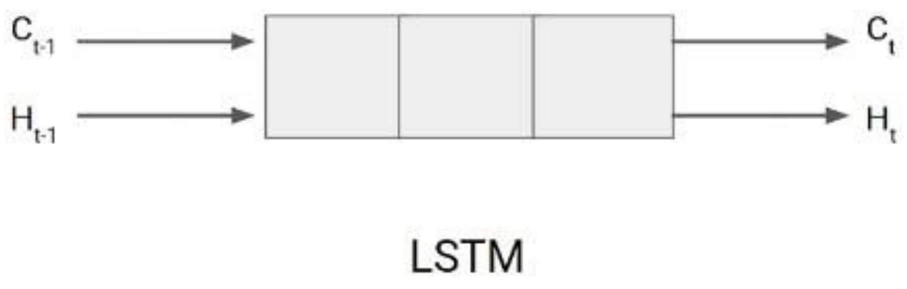


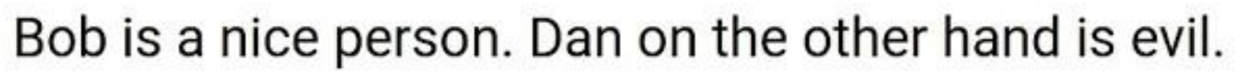
Just like a simple RNN, an LSTM also has a hidden state where H(t-1) represents the hidden state of the previous timestamp and Ht is the hidden state of the current timestamp. In addition to that, LSTM also has a cell state represented by C(t-1) and C(t) for the previous and current timestamps, respectively.

Here the hidden state is known as Short term memory, and the cell state is known as Long term memory. Refer to the following image.



It is interesting to note that the cell state carries the information along with all the timestamps.




Bob is a nice person. Dan on the other hand is evil.

Example of LTSM Working

Let's take an example to understand how LSTM works. Here we have two sentences separated by a full stop. The first sentence is "Bob is a nice person," and the second sentence is "Dan, on the Other hand, is evil". It is very clear, in the first sentence, we are talking about Bob, and as soon as we encounter the full stop(.), we started talking about Dan.

As we move from the first sentence to the second sentence, our network should realize that we are no more talking about Bob. Now our subject is Dan. Here, the Forget gate of the network allows it to forget about it. Let’s understand the roles played by these gates in LSTM architecture.

Forget Gate

In a cell of the LSTM neural network, the first step is to decide whether we should keep the information from the previous time step or forget it. Here is the equation for forget gate.

Forget Gate:

- $f_t = \sigma (x_t * U_f + H_{t-1} * W_f)$

Let’s try to understand the equation, here

- Xt: input to the current timestamp.
- Uf: weight associated with the input
- Ht-1: The hidden state of the previous timestamp
- Wf: It is the weight matrix associated with the hidden state

Later, a sigmoid function is applied to it. That will make ft a number between 0 and 1. This ft is later multiplied with the cell state of the previous timestamp, as shown below.

$$C_{t-1} * f_t = 0 \quad \ldots\text{if } f_t = 0 \text{ (forget everything)}$$

$$C_{t-1} * f_t = C_{t-1} \quad \ldots\text{if } f_t = 1 \text{ (forget nothing)}$$

Input Gate

Let’s take another example.

“Bob knows swimming. He told me over the phone that he had served the navy for four long years.”

So, in both these sentences, we are talking about Bob. However, both give different kinds of information about Bob. In the first sentence, we get the information that he knows swimming. Whereas the second sentence tells, he uses the phone and served in the navy for four years.

Now just think about it, based on the context given in the first sentence, which information in the second sentence is critical? First, he used the phone to tell, or he served in the navy. In this context, it doesn’t matter whether he used the phone or any other medium of communication to pass on the information. The fact that he was in the navy is important information, and this is something we want our model to remember for future computation. This is the task of the Input gate.

The input gate is used to quantify the importance of the new information carried by the input.

Here is the equation of the input gate

Input Gate:

- $i_t = \sigma(x_t * U_i + H_{t-1} * W_i)$

Here,

- Xt: Input at the current timestamp t
- Ui: weight matrix of input
- Ht-1: A hidden state at the previous timestamp
- Wi: Weight matrix of input associated with hidden state

Again we have applied the sigmoid function over it. As a result, the value of I at timestamp t will be between 0 and 1.

New Information

- $N_t = \tanh(x_t * U_c + H_{t-1} * W_c)$ (new information)

Now the new information that needed to be passed to the cell state is a function of a hidden state at the previous timestamp t-1 and input x at timestamp t. The activation function here is tanh.

Due to the tanh function, the value of new information will be between -1 and 1. If the value of Nt is negative, the information is subtracted from the cell state, and if the value is positive, the information is added to the cell state at the current timestamp.

However, the Nt won't be added directly to the cell state. Here comes the updated equation:

$$C_t = f_t * C_{t-1} + i_t * N_t \text{ (updating cell state)}$$

Here, Ct-1 is the cell state at the current timestamp, and the others are the values we have calculated previously.

Output Gate

Now consider this sentence.

"Bob single-handedly fought the enemy and died for his country. For his contributions, brave______."

During this task, we have to complete the second sentence. Now, the minute we see the word brave, we know that we are talking about a person. In the sentence, only Bob is brave, we can not say the enemy is brave, or the country is brave. So based on the current expectation, we have to give a relevant word to fill in the blank. That word is our output, and this is the function of our Output gate.

Here is the equation of the Output gate, which is pretty similar to the two previous gates.

Output Gate:

- $o_t = \sigma (x_t * U_o + H_{t-1} * W_o)$

Its value will also lie between 0 and 1 because of this sigmoid function. Now to calculate the current hidden state, we will use Ot and tanh of the updated cell state. As shown below.

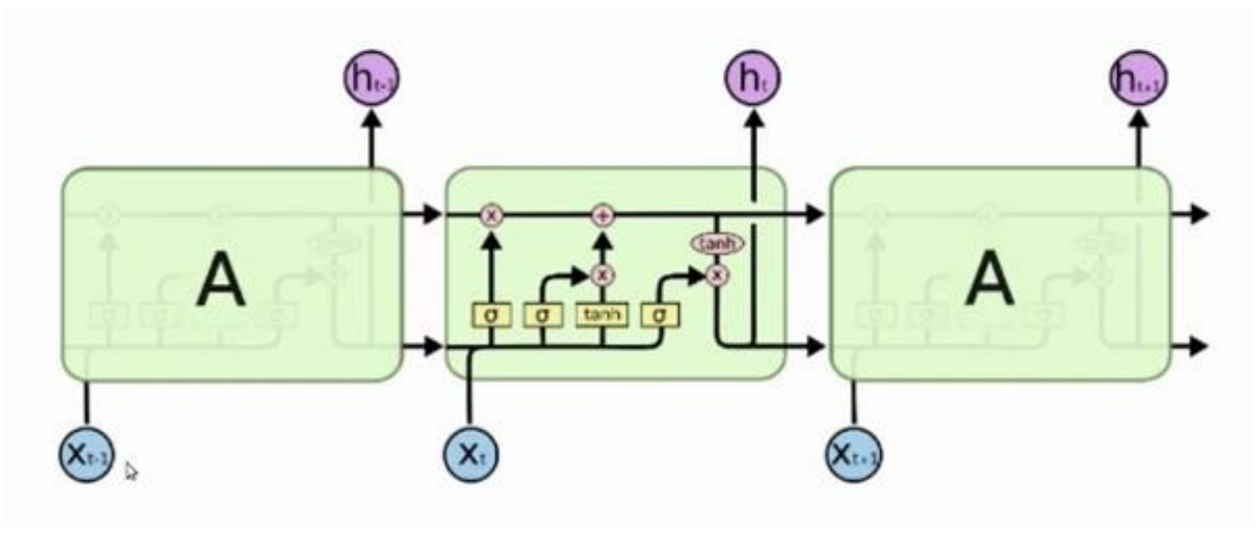
$$H_t = o_t * \tanh(C_t)$$

It turns out that the hidden state is a function of Long term memory (Ct) and the current output. If you need to take the output of the current timestamp, just apply the SoftMax activation on hidden state Ht.

$$\text{Output} = \text{Softmax}(H_t)$$

Here the token with the maximum score in the output is the prediction.

This is the More intuitive diagram of the LSTM network.



This diagram is taken from an interesting blog. I urge you all to go through it. Here is the link: ***Understanding LSTM Networks***!

LTSM vs RNN

If you are looking to kick-start your Data Science Journey and want every topic under one roof, your search stops here. Check out Analytics Vidhya's Certified AI & ML BlackBelt Plus Program.

What are Bidirectional LSTMs?

Bidirectional LSTMs (Long Short-Term Memory) are a type of recurrent neural network (RNN) architecture that processes input data in both forward and backward directions. In a traditional LSTM, the information flows only from past to future, making predictions based on the preceding context. However, in bidirectional LSTMs, the network also considers future context, enabling it to capture dependencies in both directions.

The bidirectional LSTM comprises two LSTM layers, one processing the input sequence in the forward direction and the other in the backward direction. This allows the network to access information from past and future time steps simultaneously. As a result, bidirectional LSTMs are particularly useful for tasks that require a comprehensive understanding of the input sequence, such as natural language processing tasks like sentiment analysis, machine translation, and named entity recognition.

By incorporating information from both directions, bidirectional LSTMs enhance the model's ability to capture long-term dependencies and make more accurate predictions in complex sequential data.

Conclusion

In this article, we covered the basics and sequential architecture of a Long Short-Term Memory Network model. Knowing how it works helps you design an LSTM model with ease and better understanding. It is an important topic to cover as LSTM models are widely used in artificial intelligence for natural language processing tasks like language modeling and machine translation. Some other applications of lstm are speech recognition, image captioning, handwriting recognition, time series forecasting by learning time series data, etc.

REF:

https://medium.com/@anishnama20/understanding-lstm-architecture-pros-and-cons-and-implementation-3e0cca194094

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

Gated Recurrent Unit (GRU)

Introduction

In the ever-evolving world of artificial intelligence, Moreover, where algorithms mimic the human brain's ability to learn from data, Recurrent Neural Networks (RNNs) have emerged as a powerful deep learning algorithm for processing sequential data. However, RNNs struggle with long-term dependencies within sequences. This is where Gated Recurrent Units (GRUs) come in. As a type of RNN equipped with a specific learning algorithm, GRUs address this limitation by utilizing gating mechanisms to control information flow, making them a valuable tool for various tasks in machine learning.

Objective

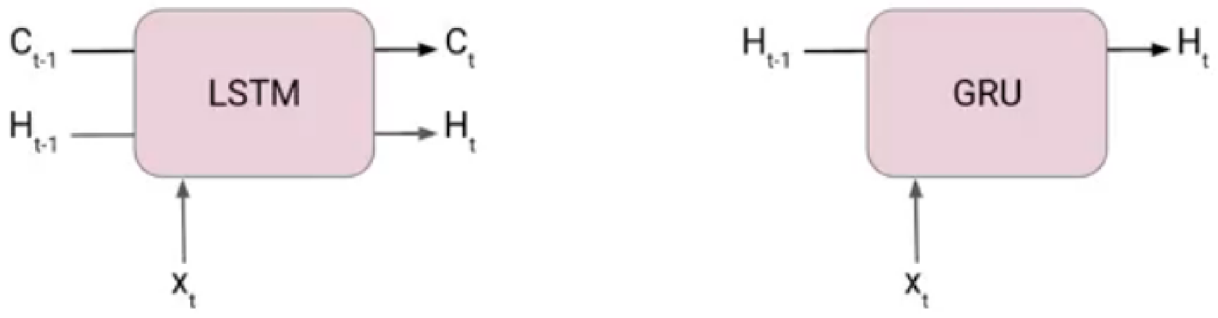
- In sequence modeling techniques, the Gated Recurrent Unit is the newest entrant after RNN and LSTM, hence it offers an improvement over the other two.
- Understand the working of GRU Activation Function and how it is different from LSTM

Note: If you are more interested in learning concepts in an Audio-Visual format, We have this entire article explained in the video below. If not, you may continue reading.

What is GRU?

GRU or Gated recurrent unit is an advancement of the standard RNN i.e recurrent neural network. It was introduced by Kyunghyun Cho et al in the year 2014.

GRUs are very similar to Long Short Term Memory(LSTM). Just like LSTM, GRU uses gates to control the flow of information. They are relatively new as compared to LSTM. This is the reason they offer some improvement over LSTM and have simpler architecture.



Another Interesting thing about GRU network is that, unlike LSTM, it does not have a separate cell state (Ct). It only has a hidden state(Ht). Due to the simpler architecture, GRUs are faster to train.

In case you are unaware of the LSTM network, I will suggest you go through the following article-*Introduction to Long Short term Memory(LSTM)*.

Limitations of Standard RNN

Here are the limitations of standard RNNs in bullet points:

- **Vanishing Gradient problem :** This is a major limitation that occurs when processing long sequences. As information propagates through the network over many time steps, the gradients used to update the network weights become very small (vanish). This makes it difficult for the network to learn long-term dependencies in the data.
- **Exploding Gradients:** The opposite of vanishing gradients, exploding gradients occur when the gradients become very large during backpropagation. This can lead to unstable training and prevent the network from converging to an optimal solution.
- **Limited Memory:** Standard RNNs rely solely on the hidden state to capture information from previous time steps. This hidden state has a limited capacity, making it difficult for the network to remember information over long sequences.
- **Difficulty in Training:** Due to vanishing/exploding gradients and limited memory, standard RNNs can be challenging to train, especially for complex tasks involving long sequences.

How GRU Solve the Limitations of Standard RNN?

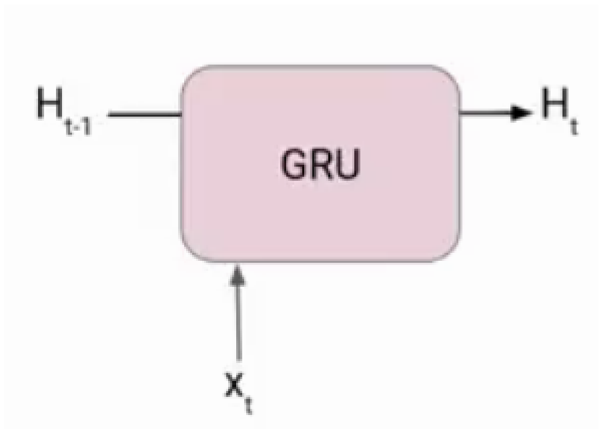
There are various types of recurrent neural network to solve the issues with standard RNN, GRU is one of them. Here’s how GRUs address the limitations of standard RNNs:

- **Gated Mechanisms:** Unlike standard RNNs, GRUs use special gates (Update gate and Reset gate) to control the flow of information within the network. These gates act as filters, deciding what information from the past to keep, forget, or update.

- **Mitigating Vanishing Gradients:** By selectively allowing relevant information through the gates, GRUs prevent gradients from vanishing entirely. This allows the network to learn long-term dependencies even in long sequences.
- **Improved Memory Management:** The gating mechanism allows GRU Activation Function to effectively manage the flow of information. The Reset gate can discard irrelevant past information, and the Update gate controls the balance between keeping past information and incorporating new information. This improves the network's ability to remember important details for longer periods.
- **Faster Training:** Due to the efficient gating mechanisms, GRU Activation Function can often be trained faster than standard RNNs on tasks involving long sequences. The gates help the network learn more effectively, reducing the number of training iterations required.

The **Architecture** of Gated Recurrent Unit

Now lets' understand how GRU works. Here we have a GRU cell which more or less similar to an LSTM cell or RNN cell.



At each timestamp t, it takes an input Xt and the hidden state Ht-1 from the previous timestamp t-1. Later it outputs a new hidden state Ht which again passed to the next timestamp.

Now there are primarily two gates in a GRU as opposed to three gates in an LSTM cell. The first gate is the Reset gate and the other one is the update gate.

Reset Gate (Short term memory)

The Reset Gate is responsible for the short-term memory of the network i.e the hidden state (Ht). Here is the equation of the Reset gate.

$$r_t = \sigma(x_t * U_r + H_{t-1} * W_r)$$

If you remember from the LSTM gate equation it is very similar to that. The value of **rt** will range from 0 to 1 because of the sigmoid function. Here Ur and Wr are weight matrices for the reset gate.

Update Gate (Long Term memory)

Similarly, we have an Update gate for long-term memory and the equation of the gate is shown below.

$$u_t = \sigma\left(x_t * U_u + H_{t-1} * W_u\right)$$

The only difference is of weight metrics i.e Uu and Wu.

How GRU Works?

Prepare the Inputs:

- The GRU takes two inputs as vectors: the current input (X_t) and the previous hidden state (h_(t-1)).

Gate Calculations:

- There are three gates in a GRU: Reset Gate, Update Gate, and Forget Gate (sometimes combined with Reset Gate). We’ll calculate the values for each gate.
- To do this, we perform an element-wise multiplication (like a dot product for each element) between the current input and the previous hidden state vectors. This is done separately for each gate, essentially creating “parameterized” versions of the inputs specific to each gate.

- Finally, we apply an activation function (a function that transforms the values) element-wise to each element in these parameterized vectors. This activation function typically outputs values between 0 and 1, which will be used by the gates to control information flow.

Now let's see the functioning of these gates in detail. To find the Hidden state Ht in GRU, it follows a two-step process. The first step is to generate what is known as the candidate hidden state. As shown below

Candidate Hidden State

$$\hat{H}_t = \tanh(x_t * U_g + (r_t \circ H_{t-1}) * W_g)$$

It takes in the current input and the hidden state from the previous timestamp t-1 which is multiplied by the reset gate output rt. Later passed this entire information to the tanh function, the resultant value is the candidate's hidden state.

$$\hat{H}_t = \tanh(x_t * U_g + (r_t \circ H_{t-1}) * W_g)$$

The most important part of this equation is how we are using the value of the reset gate to control how much influence the previous hidden state can have on the candidate state.

If the value of rt is equal to 1 then it means the entire information from the previous hidden state Ht-1 is being considered. Likewise, if the value of rt is 0 then that means the information from the previous hidden state is completely ignored.

Hidden State

Once we have the candidate state, it is used to generate the current hidden state Ht. It is where the Update gate comes into the picture. Now, this is a very interesting equation, instead of using a separate gate like in LSTM and GRU Architecture we use a single update gate to control both the historical information which is Ht-1 as well as the new information which comes from the candidate state.

$$H_t = u_t \circ H_{t-1} + (1-u_t) \circ \hat{H}_t$$

Now assume the value of ut is around 0 then the first term in the equation will vanish which means the new hidden state will not have much information from the previous hidden state. On the other hand, the second part becomes almost one that essentially means the hidden state at the current timestamp will consist of the information from the candidate state only.

$$H_t = u_t \circ H_{t-1} + (1-u_t) \circ \hat{H}_t$$

Similarly, if the value of ut is on the second term will become entirely 0 and the current hidden state will entirely depend on the first term i.e the information from the hidden state at the previous timestamp t-1.

$$H_t = u_t \circ H_{t-1} + (1-u_t) \circ \hat{H}_t$$

Hence we can conclude that the value of ut is very critical in this equation and it can range from 0 to 1.

In case, you are interested to know more about LSTM and GRU Architecture I suggest you read this Paper.

Advantages and Disadvantages of GRU

Advantages of GRU

- **Faster Training and Efficiency:** Compared to LSTMs (Long Short-Term Memory networks), GRUs have a simpler architecture with fewer parameters. This makes them faster to train and computationally less expensive.
- **Effective for Sequential Tasks:** GRUs excel at handling long-term dependencies in sequential data like language or time series. Their gating mechanisms allow them to selectively remember or forget information, leading to better performance on tasks like machine translation or forecasting.
- **Less Prone to Gradient Problems:** The gating mechanisms in GRUs help mitigate the vanishing/exploding gradient problems that plague standard RNNs. This allows for more stable training and better learning in long sequences.

Disadvantages of GRU

- **Less Powerful Gating Mechanism:** While effective, GRUs have a simpler gating mechanism compared to LSTMs which utilize three gates. This can limit their ability to capture very complex relationships or long-term dependencies in certain scenarios.
- **Potential for Overfitting:** With a simpler architecture, LSTM and GRU Architecture might be more susceptible to overfitting, especially on smaller datasets. Careful hyperparameter tuning is crucial to avoid this issue.
- **Limited Interpretability:** Understanding how a GRU Activation Function arrives at its predictions can be challenging due to the complexity of the gating mechanisms. This makes it difficult to analyze or explain the network's decision-making process.

Applications of Gated Recurrent Unit

Here are some applications of GRUs where their ability to handle sequential data shines:

Natural Language Processing (NLP)

- **Machine translation:** GRUs can analyze the context of a sentence in one language and generate a grammatically correct and fluent translation in another language.
- **Text summarization:** By processing sequences of sentences, LSTM and GRU Architecture can identify key points and generate concise summaries of longer texts.
- **Chatbots:** GRUs can be used to build chatbots that can understand the context of a conversation and respond in a natural way.

- **Sentiment Analysis:** GRUs excel at analyzing the sequence of words in a sentence and understanding the overall sentiment (positive, negative, or neutral).

Speech Recognition

GRUs can analyze the sequence of audio signals in speech to transcribe it into text. They can be particularly effective in handling variations in speech patterns and accents.

Time Series Forecasting

GRUs can analyze historical data like sales figures, website traffic, or stock prices to predict future trends. Their ability to capture long-term dependencies makes them well-suited for forecasting tasks.

Anomaly Detection

GRUs can identify unusual patterns in sequences of data, which can be helpful for tasks like fraud detection or network intrusion detection.

Music Generation

GRUs can be used to generate musical pieces by analyzing sequences of notes and chords. They can learn the patterns and styles of different musical genres and create new music that sounds similar.

These are just a few examples, and the potential applications of GRUs continue to grow as researchers explore their capabilities in various fields.

Conclusion

Gated Recurrent Units (GRUs) represent a significant advancement in recurrent neural networks, addressing the limitations of standard RNNs. With their efficient gating mechanisms, GRUs effectively manage long-term dependencies in sequential data, making them valuable for various applications in natural language processing, speech recognition, and time series forecasting. While offering advantages like faster training and effective memory management, GRUs also have limitations such as potential overfitting and reduced interpretability. As AI continues to evolve, GRUs remain a powerful tool in the machine learning toolkit, balancing efficiency and performance for sequential data processing tasks.

Key Takeaways:

- Moreover, GRUs represent an advancement over standard RNNs, addressing their limitations by using gating mechanisms to control information flow.
- Specifically, the Reset Gate manages short-term memory, while the Update Gate controls long-term memory in GRUs.
- Additionally, GRUs feature a simpler architecture compared to Long Short-Term Memory (LSTM) networks, making them faster to train and computationally less expensive.

- Furthermore, GRUs excel at handling long-term dependencies in sequential data, making them valuable for tasks like machine translation, text summarization, and time series forecasting.

REF:

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-gated-recurrent-unit-gru/

https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2