Recurrent Neural Network(RNN)

(DOWNLOAD THE FREE PDF)

Definition: Recurrent Neural Network(RNN) are a type of Neural Network where the output from previous steps are fed as input to the current step.
Example(Real world): Language Translation
Example(Imaginary): Vehicle next Position Predict, Winner predict from a race(Car, Horse, Human)
**** Need data for game to predict next move , next behaviour prediction**
Explanation
In a RNN output of the current layer is the input of the next layer.
In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words.
It uses Long Short Term Memory(LSTM) to remember the previous output.

In a RNN same weights and bias are used for input of every layer. Because it performs the same task on all the inputs of hidden layers to produce the output.
The main and most important feature of RNN is Hidden state, which remembers some information about a sequence By LSTM.

A RNN is the best use for sequential model data like predicting the next word of a sentence or next position of a running ball.

The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit. This hidden state signifies the past knowledge that that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following the recurrence relation:-

Building Blocks of RNN:

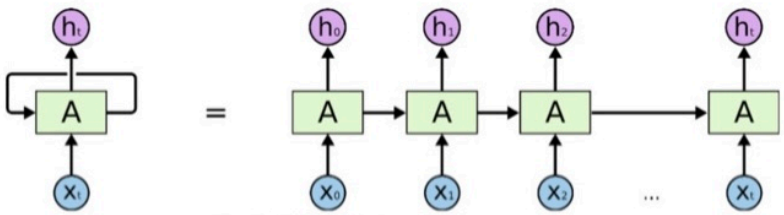


Fig-1:Building Blocks of RNN

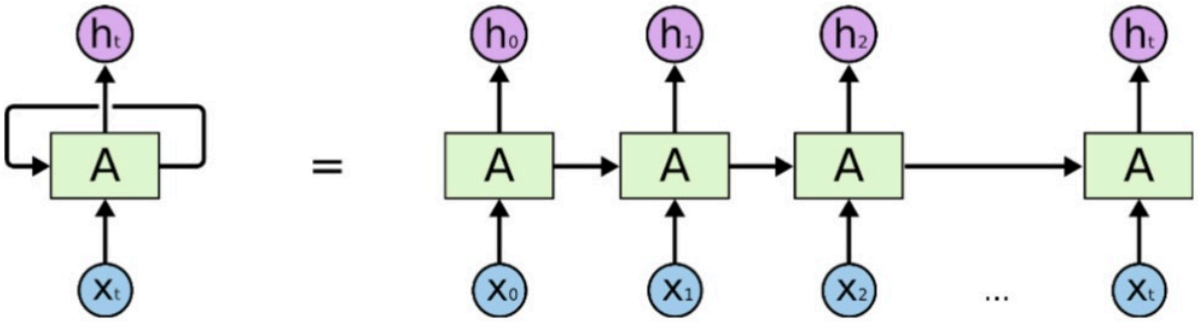


Fig-1:Building Blocks of RNN

RNN uses a loop to remember the relation among all the previous outputs to predict the next output.
So first **X(0)** is the input and **h(0)** is the output. And also **h(0)** is input for the next step where the already existing input is x(1). Similarly output of this layer which is **h(1),** and the **x(2)** will be the input for the next step. This way it keeps remembering the context in training time.

Here A is hidden layer activation function.
Activation function takes decision for the output to be qualified or not for our result
Activation function is used based on my desired output pattern or type.

The different types of Recurrent Neural Network are:

1. **One to One RNN:** One to One RNN (xt=yt=1) is the most basic and traditional type of Neural network giving a single output for a single input, as can be seen in the below image.

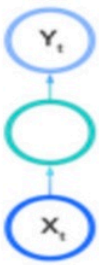


Fig-2.1:One to One RNN

2. **One to Many RNN:** One to Many (x=1,yt>1) is a kind of RNN architecture is applied in situations that give multiple output for a single input , as can be seen in

the below image. A basic example of its application would be Music generation. In Music generation models, RNN models are used to generate a music piece(multiple output) from a single musical note(single input).
Example: Music Generation, Image Captioning.

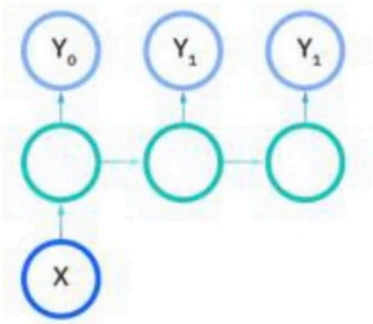


Fig-2.2:One to Many RNN

3. **Many to One RNN :** Many-to-one RNN architecture (xt>1,y=1) is usually seen for sentiment analysis model as a common example. As the name suggests, this kind of model is used when multiple inputs are required to give a single output, as can be seen in the below image.

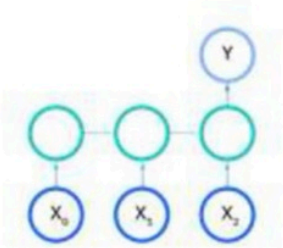


Fig-2.3:Many to One RNN

For example: The Twitter sentiment analysis model. In that model, a text input (words as multiple inputs) gives its fixed sentiment (single output).
Another example: could be movie ratings model that takes review texts as input to provide a rating to a movie that may range from 1 to 5.

4. **Many to Many RNN:** As is pretty evident, Many-to-Many RNN (Tx>1,Ty>1) Architecture takes multiple input and gives multiple output,(as can be seen in the below image) but Many-to-Many models can be two kinds as represented above:

A. Tx=Ty: (Number of input == Number of Output)

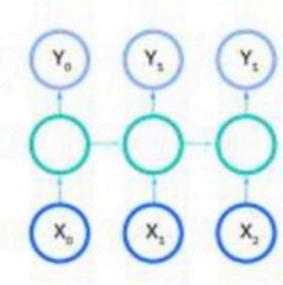


Fig-2.4:Many to Many RNN(input==output)

B.Tx!=Ty: (Number of input != Number of Output)

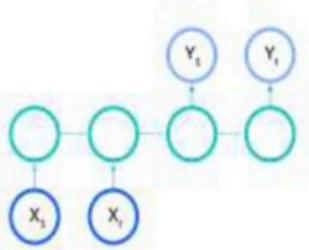


Fig-2.5:Many to Many RNN(input!=output)

Many-to-Many architecture can also be represented in models where input and output layers are of different size, and the most common application of this kind of RNN architecture is seen in Machine Translation. For example, "I Love you", the 3 magical words of the English language translates to only 2 in Spanish, "te amo". Thus, machine translation models are capable of returning words more or less than the input string because of a non-equal Many-to-Many RNN architecture works in the background.

- **Vanishing Gradients:** This occurs when the gradients become very small and tend towards zero.
- **Exploding Gradients:** This occurs when the gradients become too large due to back-propagation.

Standard RNNs (Recurrent Neural Networks) suffer from vanishing and exploding gradient problems. LSTMs (Long Short Term Memory) deal with these problems by introducing new gates, such as input and forget gates, which allow for a better control over the gradient flow and enable better preservation of “long-range dependencies”. The long range dependency in RNN is resolved by increasing the number of repeating layer in LSTM.

Vanishing Gradient:??

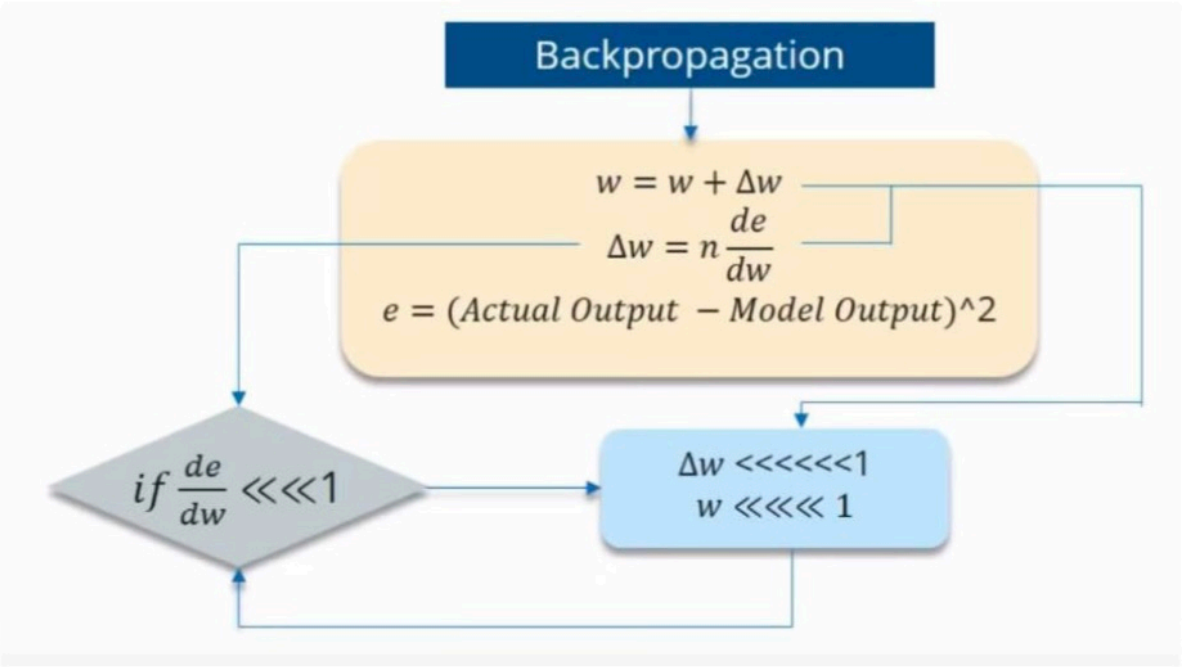


Fig-3.1: Vanishing Gradient

1) Information travels through the neural network from input neurons to the output neurons, while the error (e) is calculated and propagated back through the network to update the weights (w), change of weights needed is (Δw) .

2) While you are using Backpropagation through time, you find Error is the difference of Actual and Predicted model.

3) Now what if the partial derivation of error with respect to weight d(e)/d(w) (it is called Error gradient) is very less than 1?

4) If the partial derivation of Error is less than 1, then when it gets multiplied with the Learning rate (here n denotes learning rate) which is also very less. then Multiplying learning rate with partial derivation of Error (Δw) won't be a big change when compared with the previous iteration.

For example:- Lets say the value decreased like 0.863 →0.532 →0.356 →0.192 →0.117 →0.086 →0.023 →0.019..

you can see that there is not much change in the last 3 iterations. This Vanishing of Gradient is called Vanishing Gradience

Several solutions to the vanishing gradient problem have been proposed over the years. The most popular are the LSTM and GRU units.

Exploding Gradient

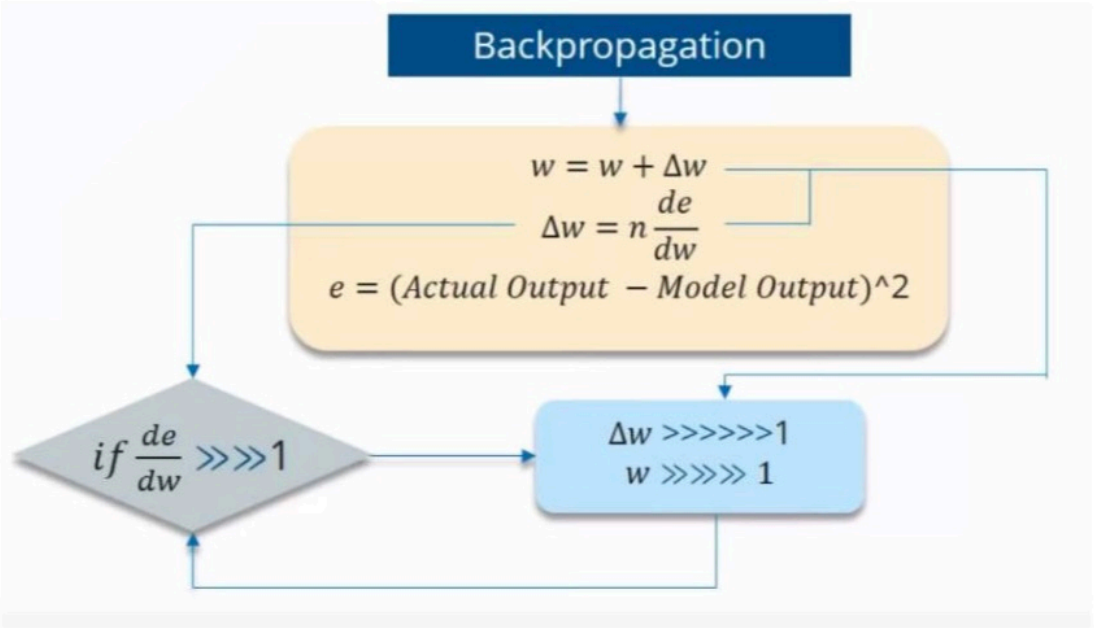


Fig-3.2:Exploding Gradient

1) Information travels through the neural network from input neurons to the output neurons, while the error (e) is calculated and propagated back through the network to update the weights (w), change of weights needed is (Δw) .

2) While you are using Backpropagation through time, you find Error is the difference of Actual and Predicted model.

3) Now what if the partial derivation of error with respect to weight d(e)/d(w) (it is called Error gradient) is very large than 1?

4) If the partial derivation of Error is large than 1, then when it gets multiplied with the Learning rate (here n denotes learning rate), then Multiplying learning rate with partial derivation of Error (Δw) will be a big change when compared with the previous iteration.

For example:- Lets say the value increased like 0.863 → 1.255→2.32→4.55→7.88 So the value increasing is very high. This is called an exploding gradient.

The Problem of Long-Term Dependencies:

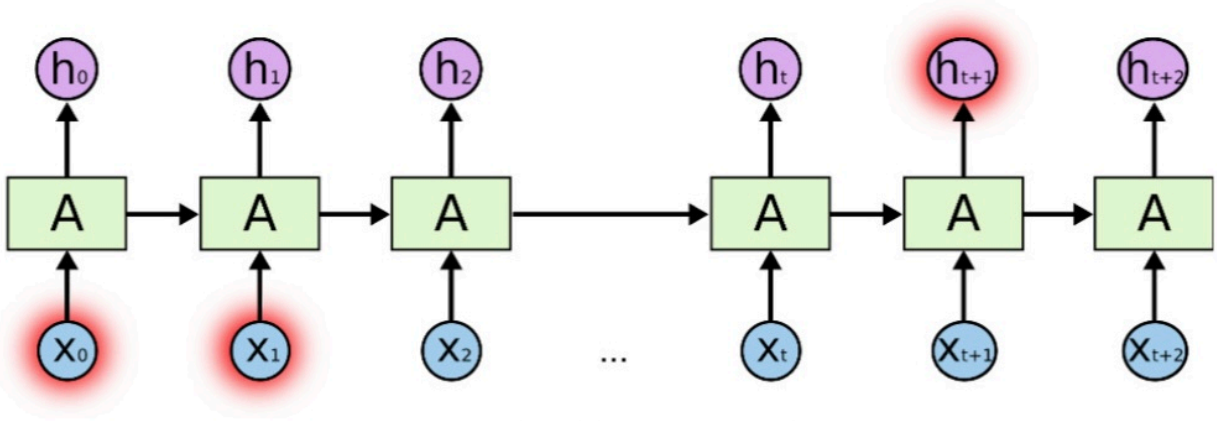


Fig-4:Long-term Dependencies

Explain the above diagram and LTD:

"A" denotes hidden activation function such as relu, sigmoid,. This is a very long sequence of LSTM layers. So if we want to predict a word for a long sentence then we have to remember the syntax of the whole sentence, In this case simple RNN fails. And for this solution comes LSTM (Long Short Term Memory).. In the above diagram h(t+1) depends on the syntax of the input data/neuron X(0) and X(1). So, for this kind of long term dependency LSTM works the best.

Long Short-Term Memory(LSTM):

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems. This is a behavior required in complex problem domains like machine translation, speech recognition, and more.

- Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies.
- Generally LSTM is composed of a cell (the memory part of the LSTM unit) and three "regulators", usually called gates, of the flow of information inside the LSTM unit: an input gate, an output gate and a forget gate.
- Intuitively, the cell is responsible for keeping track of the dependencies between the elements in the input sequence.
- The input gate controls the extent to which a new value flows into the cell, the forget gate controls the extent to which a value remains in the cell and the output gate controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit.

- The activation function of the LSTM gates is often the logistic sigmoid function.
- There are connections into and out of the LSTM gates, a few of which are recurrent. The weights of these connections, which need to be learned during training, determine how the gates operate.
- LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

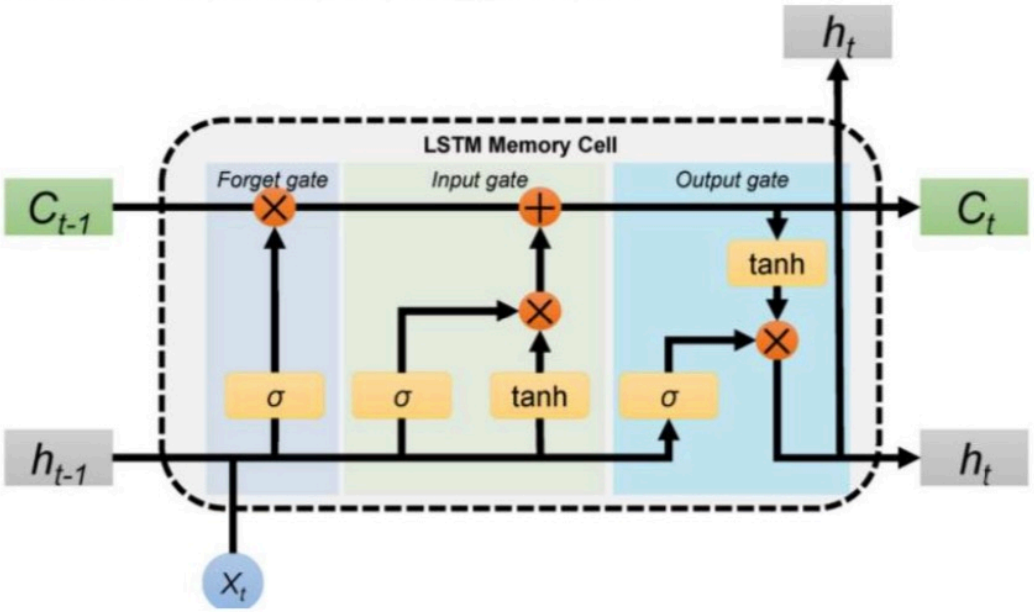
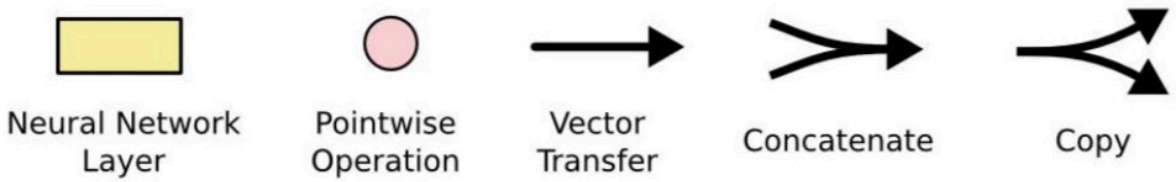


Fig-5:Long Short-Term Memory(LSTM)

So, from the above figure C_{t-1} is the long term memory and the h_{t-1} is the short term memory and thus they create this above figure (also called cell) which works as LSTM (Long short term memory).



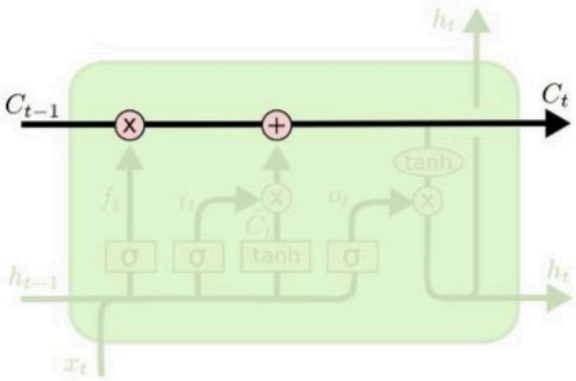
In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied and the copies going to different locations.

Luckily, recent RNN variants such as LSTM (Long Short-Term Memory) have been able to overcome the vanishing/exploding gradient problem, so RNNs can safely be applied

to extremely long sequences, even ones that contain millions of elements. In fact, LSTMs addressing the gradient problem have been largely responsible for the recent successes in very deep NLP applications such as speech recognition, language modeling, and machine translation.

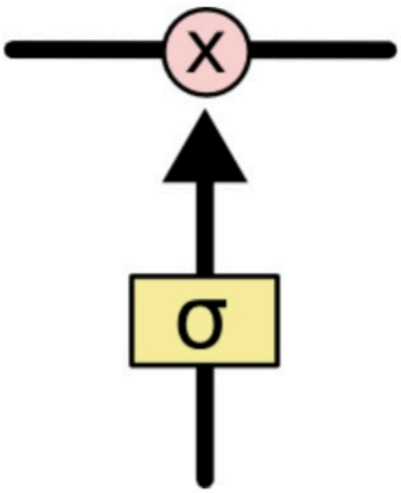
The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
The cell state is kind of like a conveyor (পরিবাহক) belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

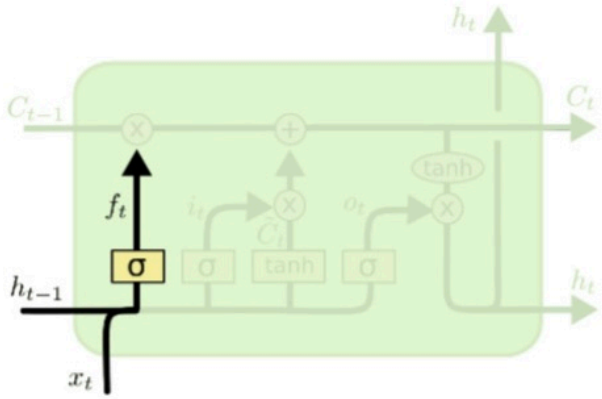
Gates are a way to optionally let information through. They are composed out of a sigmoid (**σ**) neural net layer and a pointwise multiplication (X-multiplication) operation.



The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”
An LSTM has three of these gates, to protect and control the cell state.

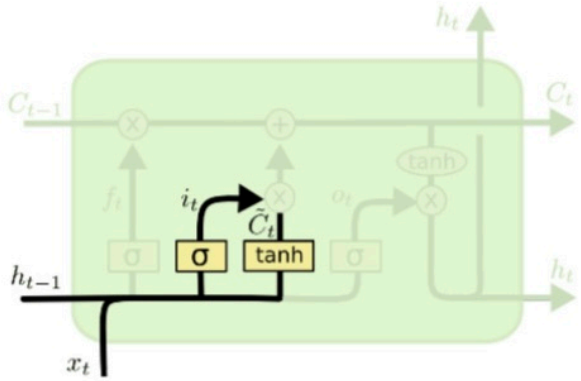
Step-by-Step LSTM Walk Through

The first step in our LSTM is to decide what information we’re going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.” It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A “1” represents “completely keep this” while a “0” represents “completely get rid of this.”
For example let’s say, a language model is trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



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

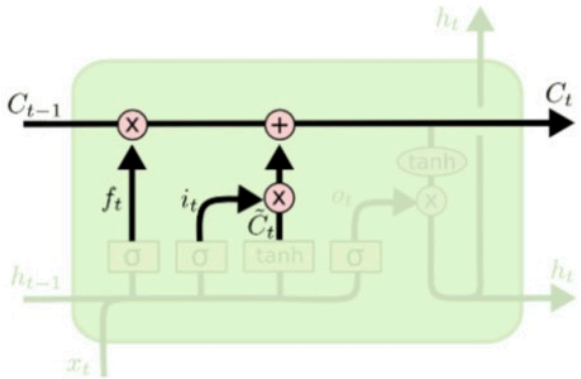
The next step is to decide what new information we’re going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we’ll update. Next, a tanh layer creates a vector of new candidate values, C_t (bar), that could be added to the state. In the next step, we’ll combine these two to create an update to the state.
In the example of our language model, we’d want to add the gender of the new subject to the cell state, to replace the old one we’re forgetting.



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

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

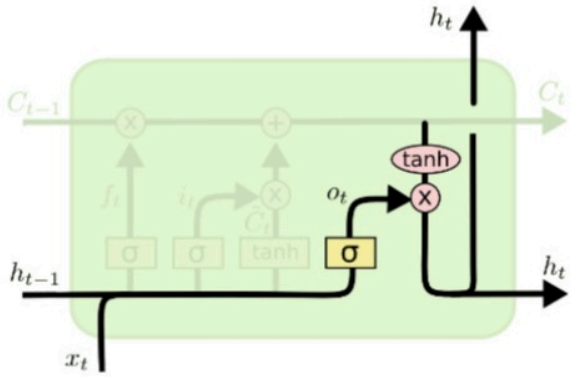
It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.
We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * C_t$ (bar) . This is the new candidate values, scaled by how much we decided to update each state value.
In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

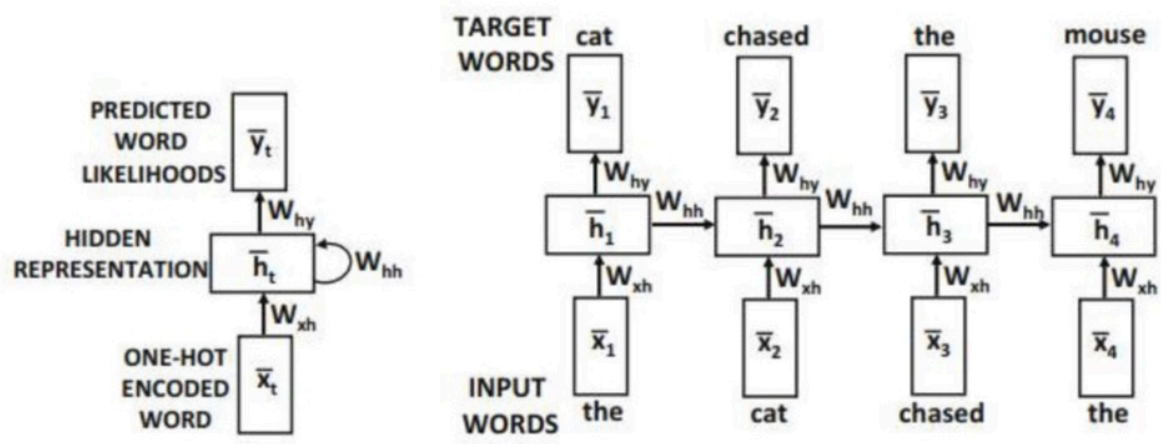
For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



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

RNN Workflow:

The simplest recurrent neural network is shown in Figure (a). A key point here is the presence of the self-loop in Figure (a), which will cause the hidden state of the neural network to change after the input of each word in the sequence. In practice, one only works with sequences of finite length, and it makes sense to unfold the loop into a "time-layered" network that looks more like a feed-forward network. This network is shown in Figure (b). Note that in this case, we have a different node for the hidden state at each time-stamp and the self-loop has been unfolded into a feed-forward network. This representation is mathematically equivalent to Figure (a), but is much easier to comprehend because of its similarity to a traditional network. The weight matrices in different temporal layers are shared to ensure that the same function is used at each time-stamp. The annotations Wxh (weight of the connection of input to hidden node), Whh (weight of hidden node of time t-1, to hidden node of time t), and Why(weight of the connection of hidden to output node) in Figure (b) make the sharing evident.



(a) RNN (b) Time-layered representation of (a)

the input vector at time t is xt(bar), the hidden state at time t is ht (bar), and the output vector at time t (e.g., predicted probabilities of the (t + 1)th word) is yt(bar).

ht(bar) = fw(xt,ht-1)

ht (bar) = The new hidden state
ht-1 (bar) = The old hidden state
xt (bar) = The current input
fw = The fixed function with trainable weights

A separate function yt = g(ht) is used to learn the output probabilities from the hidden states.

Next, we describe the functions f(·, ·) and g(·) more concretely. We define a p×d input-hidden matrix Wxh, a p × p hidden-hidden matrix Whh, and a d × p hidden-output matrix Why. Then, one can expand the Equation **ht = fw(xt,ht-1)** and also write the condition for the outputs as follows:

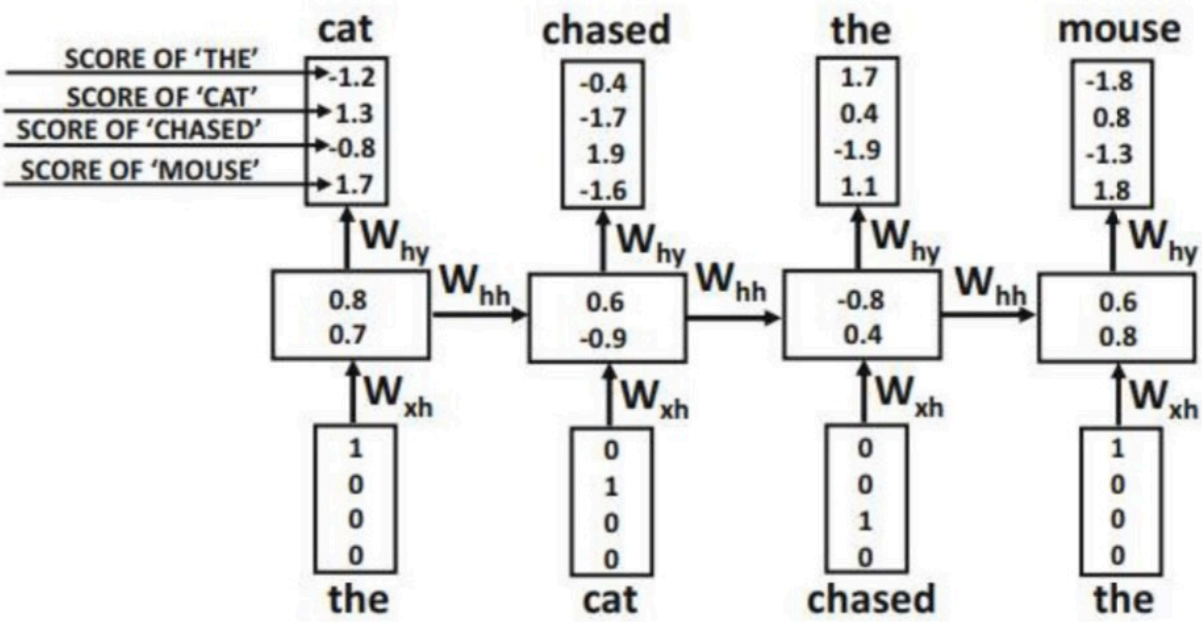
ht (bar) = tanh(Wxh * xt(bar) + Whh * ht−1(bar))

yt (bar) = Why * ht(bar)

Because of the recursive nature of the above Equation, the recurrent network has the ability to compute a function of variable-length inputs. In other words, one can expand the recurrence of above Equation to define the function for ht in terms of t inputs. For example, starting at h0, which is typically fixed to some constant vector (such as the zero vector), we have h1 = f(h0, x1) and h2 = f(f(h0, x1), x2). Note that h1 is a function of only x1, whereas h2 is a function of both x1 and x2. In general, ht is a function of x1 ... xt. Since the output yt is a function of ht, these properties are inherited by yt as well. In general, we can write the following: yt = Ft(x1, x2,... xt)

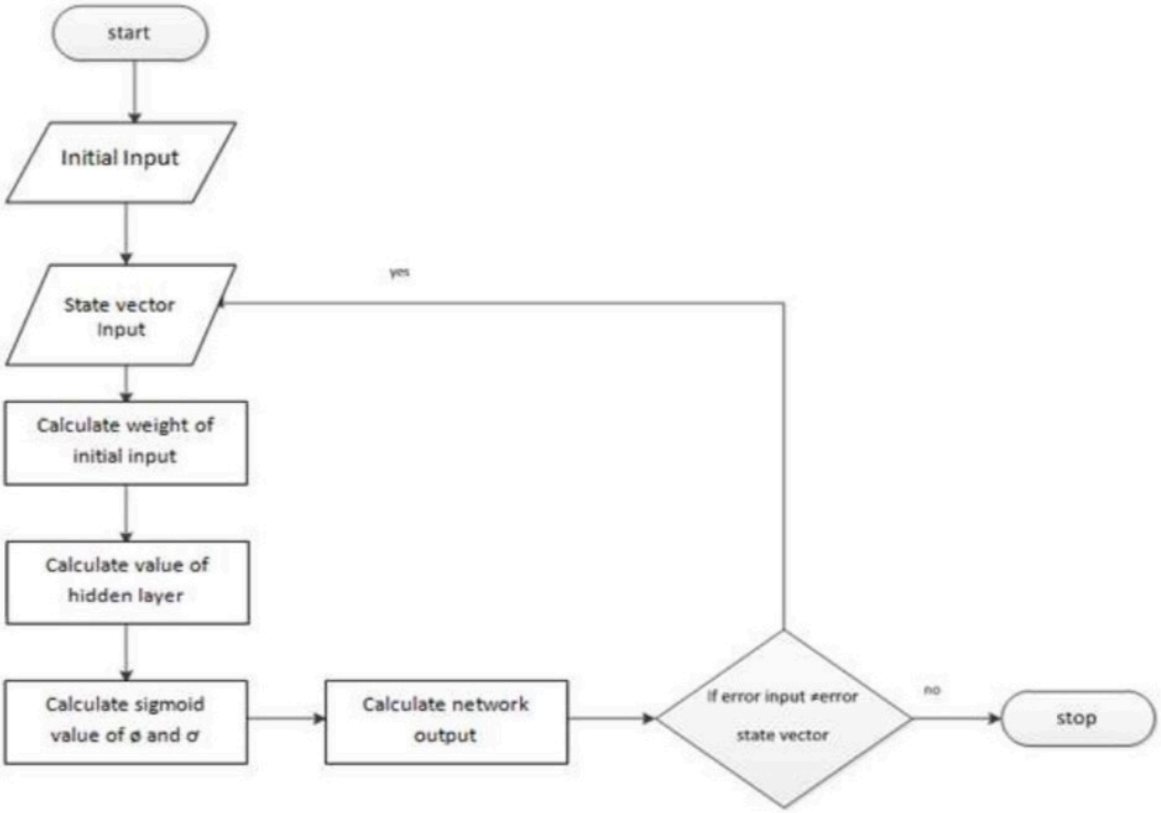
the function Ft(·) indicates the probability of the next word, taking into account all the previous words in the sentence.

RNN with examples:



The word “cat” is predicted in the first time-stamp with a value of 1.3, although this value seems to be (incorrectly) outstripped by “mouse” for which the corresponding value is 1.7. However, the word “chased” seems to be predicted correctly at the next time-stamp. As in all learning algorithms, one cannot hope to predict every value exactly, and such errors are more likely to be made in the early iterations of the backpropagation algorithm. However, as the network is repeatedly trained over multiple iterations, it makes fewer errors over the training data.

Recurrent Neural Network FlowChart:

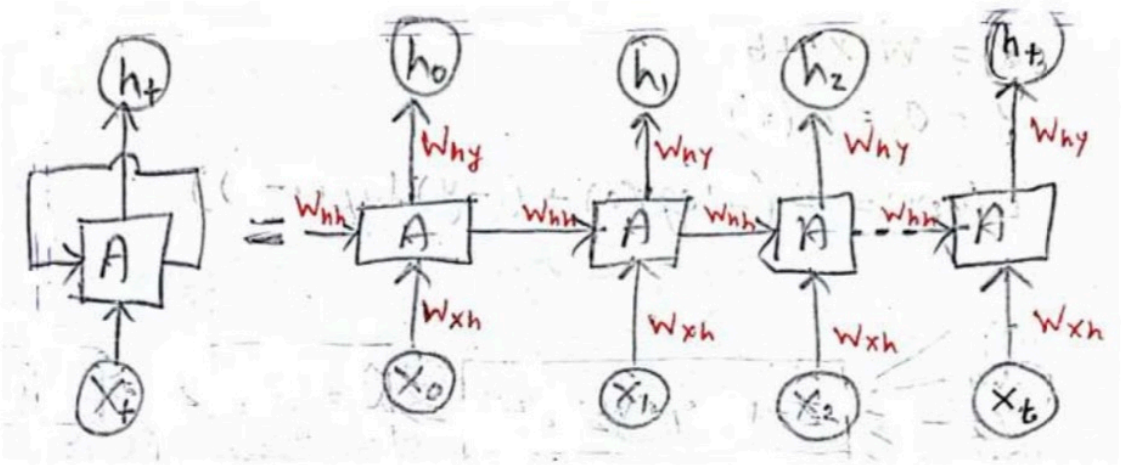


At first take inputs. After taking inputs calculate weight for each input. Then calculate hidden layer value ,sigmoid value . When error input != error state vector this condition continue while(error input != error state vector) . If condition fulfill then works stop and the output will out.

How does Recurrent Neural Network Works (ALGORITHM) :

1. A single time step of the input is provided to the network.
2. Then calculate its current state **h(t)** using set of current input and the previous state.
3. The current **ht** becomes **ht-1** for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

How does work RNN (Simulation) :



Formula for calculating current state:
ht = f(ht-1,xt) where,
ht -> current state
ht-1 ->previous state
xt -> input state
formula for applying Activation fanc(tanh)(ber)
ht = tanh(Whh ht-1 + Wxh xt)
where,
Whh -> weight at recurrent neuron
Wxh -> weight at input neuron

Formula for calculating output:
Yt=Why ht
Where,
Yt -> output
Why -> Weight at output layer

Recurrent Neural Network Real World Applications:

1. Text summarization
2. Report Generation
3. Face detection
4. Image recognition

5. Spam Detection,
6. Bot Detection
7. Next word prediction.
8. Music composition.
9. Image captioning
10. Time series anomaly detection
11. Stock market prediction

Recurrent Neural Network Advantage:

1. RNN can process inputs of any length.
2. the weight can be shared across the time steps.
3. Even if the input size is larger ,the model size does not increase.
4. An RNN model is modeled to remember each information throughout the time which is very helpful in any time series predictor.
5. RNN can use their internal memory for processing the arbitrary series of inputs which is not the case with feedforward neural networks.

Recurrent Neural Network Disadvantages:

1. Due to its recurrent nature, the computation is slow.
2. Training of RNN models can be difficult.
3. If we are using sigmoid or tanh as activation function, it becomes very difficult to process sequences that are very long.
4. problems such as exploding and gradient vanishing.

Some more extended Types of Recurrent Neural Network

Bidirectional RNNs

The use case for bidirectional recurrent neural networks is centered on scenarios in which the state of a node is affected by the state of nodes executed at a future time. The traditional RNN architecture is based on a very simple computation graph in which the state of the network at any given time is based solely on information about the past. Now let's take a simple speech recognition scenario in which the final analysis of an audio stream at any given time depends on the interpretation of a future segment of the audio stream. Suppose that a digital assistant inquires about your latest experience at the movies by asking you "how was the movie?" to what you answer "Well…" indicating a level of uncertainty. However, the final analysis will depend on your next statement. Bidirectional RNNs address the future-dependency limitations of traditional recurrent networks by combining two RNNs in the same model. The first RNN moves forward through time from the beginning of the network while the second RNN moves backward

starting at the end of a specific sequence. This simple adaptation of traditional RNNs allow any hidden unit to compute knowledge that depends both on the past and the future relative to a specific time window.

Deep Recurrent Networks

Traditional RNNs are represented using a very basic computation graph that connects the input unit to a sequence hidden units and the final hidden unit to the output unit. In that model, the computations performed by any of the hidden units have to be based on atomic transformations which often result insufficient to build more sophisticated data manipulation routines. Deep recurrent networks address that challenge by decomposing the state of a unit into a multi-layer network capable of performing arbitrarily complex operations.

The addition of depth to specific units directly expands the richness of its knowledge representation. However, is not as trivial as it sounds. Deep recurrent networks can have a negative impact by hurting the learning performance or making optimization more difficult.

Bidirectional and deep recurrent networks are two of the most popular forms of RNNs that you will find in deep learning stacks. In the next part of this article we will cover recursive neural networks as another technique to consider when processing sequential datasets.