Rizvi College of Engineering

DEEP LEARNING

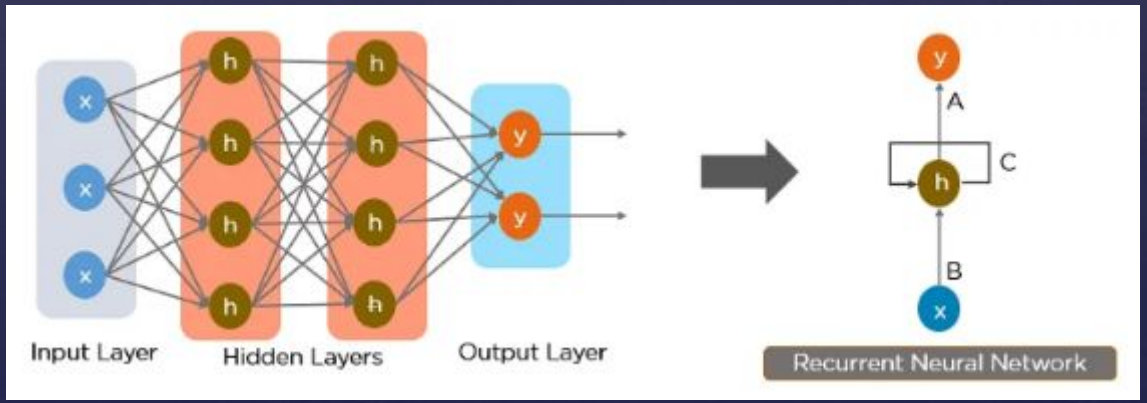
5		**Recurrent Neural Networks (RNN)**
	5.1	Sequence Learning Problem, Unfolding Computational graphs, Recurrent Neural Network, Bidirectional RNN, Backpropagation Through Time (BTT), Limitation of " vanilla RNN" Vanishing and Exploding Gradients, Truncated BTT
	5.2	Long Short Term Memory(LSTM): Selective Read, Selective write, Selective Forget, Gated Recurrent Unit (GRU)

Recurrent neural networks(RNN)

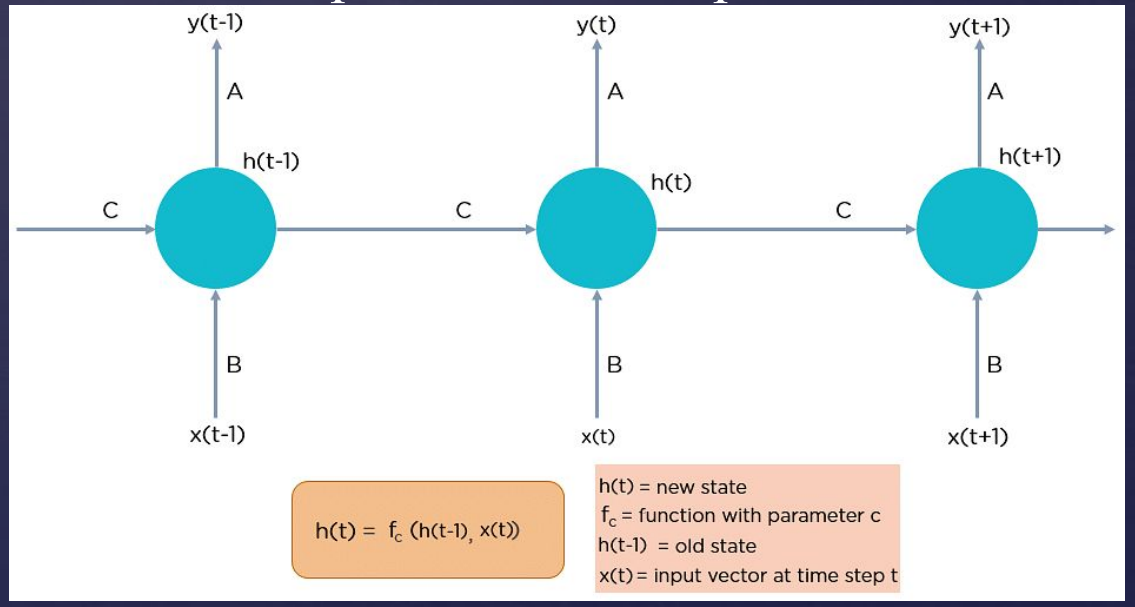
- A recurrent neural network is a neural network that is specialized for processing a sequence of data $x(t)= x(1), \ldots, x(\tau)$ with the time step index t ranging from 1 to τ. For tasks that involve sequential inputs, such as speech and language, it is often better to use RNNs.

- **RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far.**
- Example: Chef making a dish.

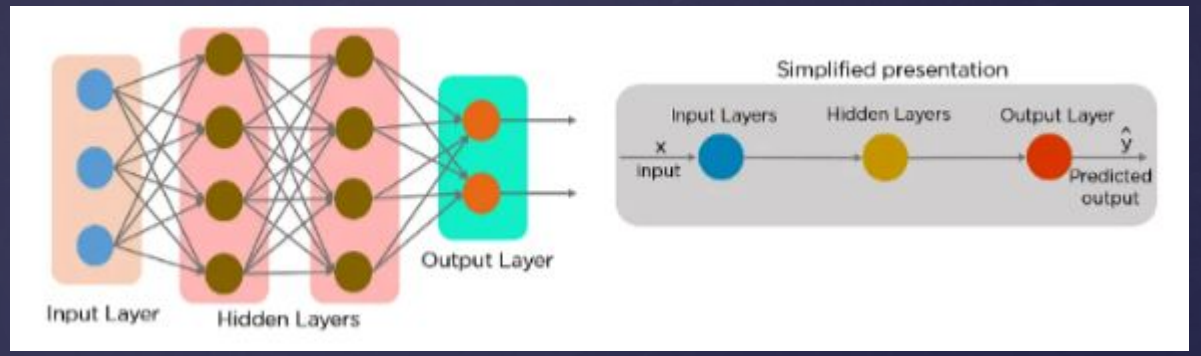
- Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step are fed as input to the current step.
- RNN have a “memory” which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.



- At any given time t, the current input is a combination of input at x(t) and x(t-1). The output at any given time is fetched back to the network to improve on the output.
- Fully connected Neural Network is shown.



- Feed-Forward Neural Networks vs Recurrent Neural Networks:
- A feed-forward neural network allows information to flow only in the forward direction, from the input nodes, through the hidden layers, and to the output nodes. There are no cycles or loops in the network.
- In a feed-forward neural network, the decisions are based on the current input. It doesn't memorize the past data, and there's no future scope. Feed-forward neural networks are used in general regression and classification problems.



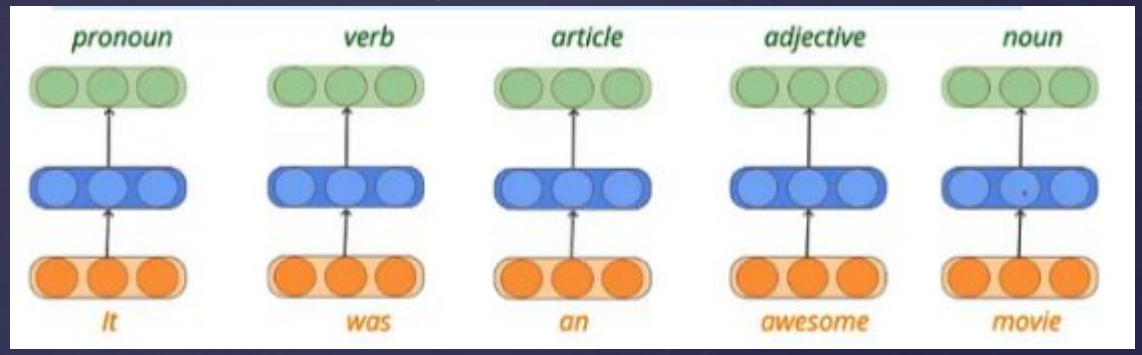
If we trained a feedforward network that processes sentences of fixed length. A traditional fully connected feedforward network would have separate parameters for each input feature, so it would need to learn all of the rules of the language separately at each position in the sentence. By comparison, a recurrent neural network shares the same weights across several time steps.

I. Sequence Learning Problems

- In all of the networks that we have covered so far(Fully Connected Neural Network(FCNN), Convolutional Neural Network(CNN)):
- the output at any time step is independent of the previous layer input/output
- the input was always of the fixed-length/size

- In "Sequence Learning Problems", the "two properties of FCNN and CNNs do not hold" and the output at any timestep depends on previous input/output and the length of the input is not fixed.
- Few examples:

- Part of speech tagging:
  - Given a “sequence of words”, the idea is to “predict part of the speech tag for each word” (whether that word is a pronoun, noun, verb, article, adjective, and so on)
  - Here also the “output depends” not only on the “current input” but “also on the previous input(s)” for example:



  - Say the current input as ‘movie’ (last layer in the snippet above) and the previous input was ‘awesome’ which is an adjective; “the moment model sees an adjective” it would be more or less confident that the “next word is actually going to be a noun”

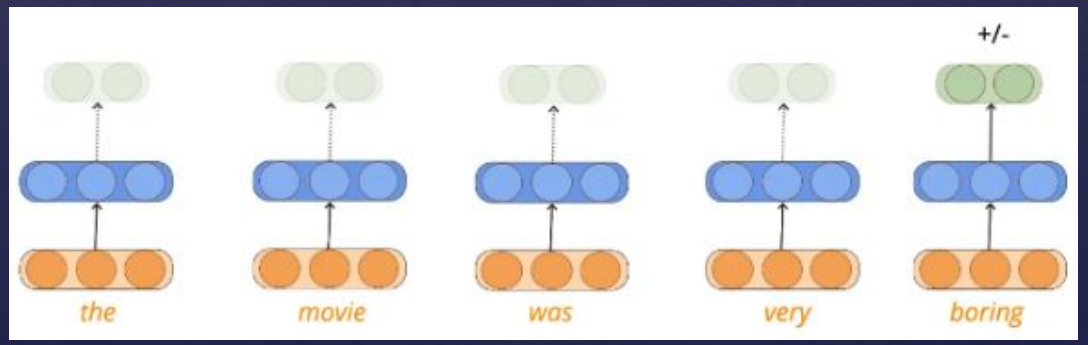
- Sentimental Analysis:

- It’s not mandatory to produce an output at each step(time step/layer) for example for sentiment analysis, the model looks at all the words in a sentence and gives/predicts a final output as positive/negative sentiment conveyed by the sentence.
- This could be considered as output is

produced at every time step but the model

ignore those output and reports only the final

output (and somehow this final output is dependent on all the previous inputs as well). This is also termed a Sequence Classification Problem.



- Sequence Learning problems using video and speech data:
  - *Speech Recognition* —Think of speech as a sequence of phonemes and give the speech signal as the input, the idea would be to map each signal to its respective phoneme in the language
  - *Video Labeling* — A video is a sequence of frames (there might be some processing on these frames), one task could be to label every frame in the video.

- II. Unfolding Computational graphs:
- **A Computational Graph is a way to formalize the structure of a set of computations. Such as mapping inputs and parameters to outputs and loss**. We can unfold a recursive or recurrent computation into a computational graph that has a repetitive structure corresponding to a chain of events.
- Unfolding this graph results in sharing of parameters across a deep network structure.

1. Unfolding equation by repeatedly applying the definition in this way has yielded expression without recurrence.

- **Dynamical system driven by external signal**:
- As another example, consider a dynamical system driven by external (input) signal x(t)
- $s(t)=f(s(t-1), x(t); \theta)$.
- State now contains information about the whole past input sequence.
- Note that the previous dynamic system was simply $s(t)=f(s(t-1); \theta)$.

Classical form of a dynamical system is

$$s^{(t)} = f\left(s^{(t-1)}; \boldsymbol{\theta}\right)$$

- where $s^{(t)}$ is called the state of the system
- Equation is recurrent because the definition of s at time t refers back to the same definition at time t-1

For a finite no. of time steps τ, the graph can be unfolded by applying the definition τ-1 times

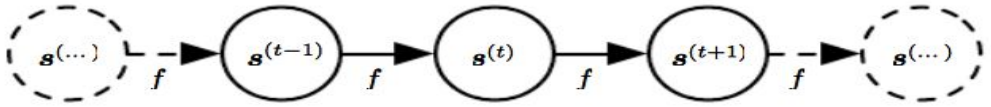
- E,g, for $\tau = 3$ time steps we get

$$s^{(3)} = f\left(s^{(2)}; \boldsymbol{\theta}\right)$$
$$= f\left(f\left(s^{(1)}; \boldsymbol{\theta}\right); \boldsymbol{\theta}\right)$$

The classical dynamical system described by

$$s^{(t)} = f\left(s^{(t-1)}; \boldsymbol{\theta}\right) \text{ and } s^{(3)} = f\left(f\left(s^{(1)}; \boldsymbol{\theta}\right); \boldsymbol{\theta}\right)$$

is illustrated as an unfolded computational graph



Each node represents state at some time t

Function f maps state at time t to the state at $t+1$

The same parameters (the same value of $\boldsymbol{\theta}$ used to parameterize f) are used for all time steps

2. Recurrent neural networks can be built in many ways:

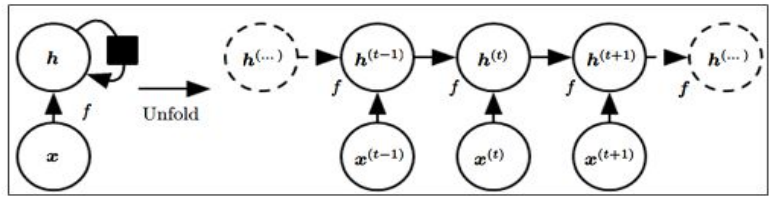
- Much as almost any function is a feedforward neural network, any function involving recurrence can be considered to be a recurrent neural network.

Many recurrent neural nets use same equation (as dynamical system with external input) to define values of hidden units

- To indicate that the state is hidden rewrite using variable $\boldsymbol{h}$ for state:

$$\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

- Illustrated below:



Typical RNNs have extra architectural features such as output layers that read information out of state $\boldsymbol{h}$ to make predictions

Predicting the Future from the Past

When RNN is required to perform a task of predicting the future from the past, network typically learns to use $\boldsymbol{h}^{(t)}$ as a lossy summary of the task-relevant aspects of the past sequence of inputs upto t

The summary is in general lossy since it maps a sequence of arbitrary length $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, .., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ to a fixed length vector $\boldsymbol{h}^{(t)}$

- Example, If RNN is used in statistical language modelling, typically trying to predict the next word given previous words, it may not be necessary to store all the information in the input sequence up to time t, but rather only enough information to predict the rest of the sentence.

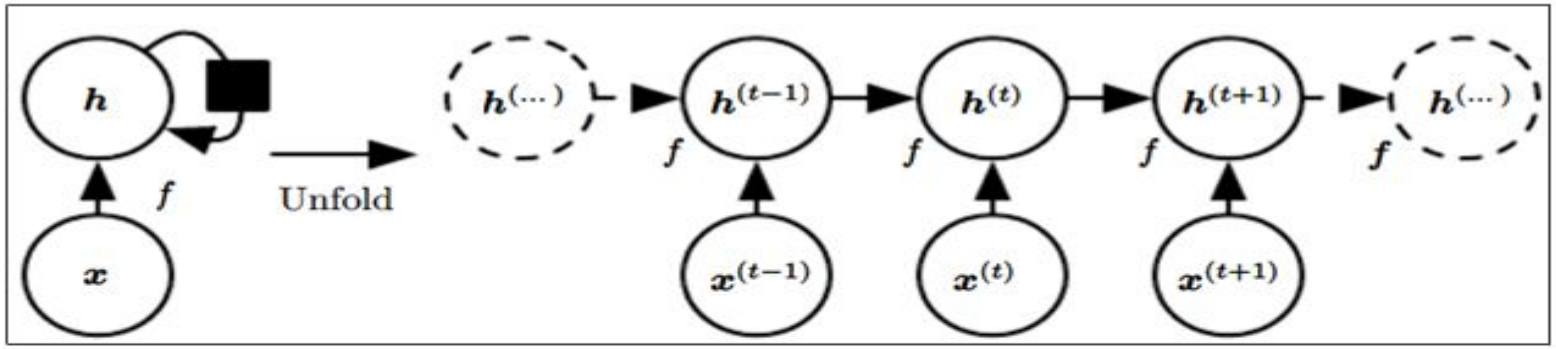
- 3.

A recurrent network with no outputs

- This network just processes information from input x by incorporating it into state h that is passed forward through time

Circuit diagram:
Black square indicates
Delay of one time step

Unfolded computational graph:
each node is now
associated with one time instance

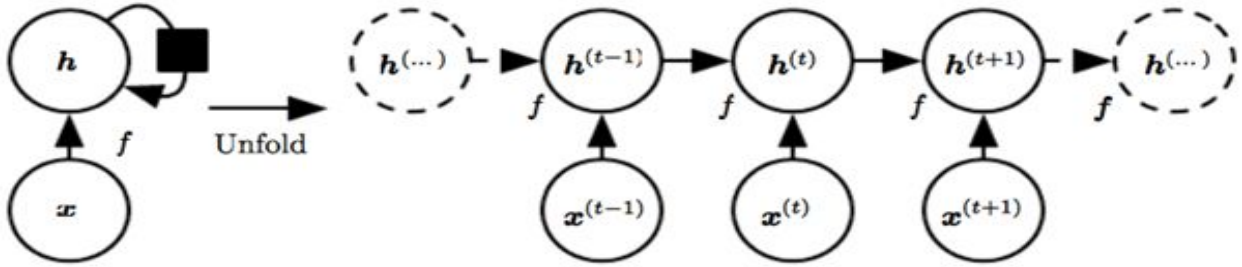


Typical RNNs will add extra architectural features such as output layers to read information out of the state h to make predictions

- 4.

Unfolding: from circuit diagram to computational graph

- Equation $\boldsymbol{h}^{(t)}=f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$ can be written in two different ways: circuit diagram or an unfolded computational graph



- Unfolding is the operation that maps a circuit to a computational graph with repeated pieces
- The unfolded graph has a size dependent on the sequence length

Process of Unfolding

We can represent unfolded recurrence after t steps with a function $g^{(t)}$:

$$\boldsymbol{h}^{(t)} = g^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, .., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right)$$
$$= f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

Function $g^{(t)}$ takes in whole past sequence $\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, .., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right)$ as input and produces the current state but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f

The unfolding process introduces two major advantages as discussed next.

- **5.** The unfolding process introduces two major advantages:

1. Regardless of sequence length, learned model has same input size, because it is specified in terms of transition from one state to another state rather than specified in terms of a variable length history of states

2. Possible to use same function f with same parameters at every step. These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths rather than needing separate model g(t) for all possible time steps

- Learning a single shared model allows:
  - Generalization to sequence lengths that did not appear in the training
  - Allows model to be estimated with far fewer training examples than would be required without parameter sharing.

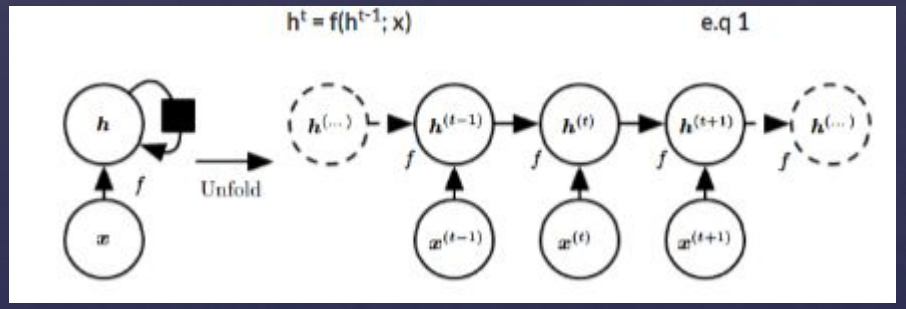
Both recurrent graph and unrolled graph are useful

Recurrent graph is succinct

Unrolled graph provides explicit description of what computations to perform

- Helps illustrate the idea of information flow forward in time
  - Computing outputs and losses
- And backwards in time
  - Computing gradients
- By explicitly showing path along which information flows

- **Recurrent Neural Network:**
- **Recurrent Neural Networks (RNN) are a part of the neural network's family used for processing sequential data. RNN perform same task for every element with output being dependent on previous computations.**



- **Using graph unfolding and parameter sharing ideas in the above equation and diagram, we can design a wide variety of recurrent neural networks.**

Architecture of recurrent neural network

- Architecture of recurrent neural network where x, h, o, L, y represents input, hidden state, output, loss, and target value respectively.
- Recurrent Neural Network maps an input sequence x values to a

corresponding sequence of output ‘o’ values.

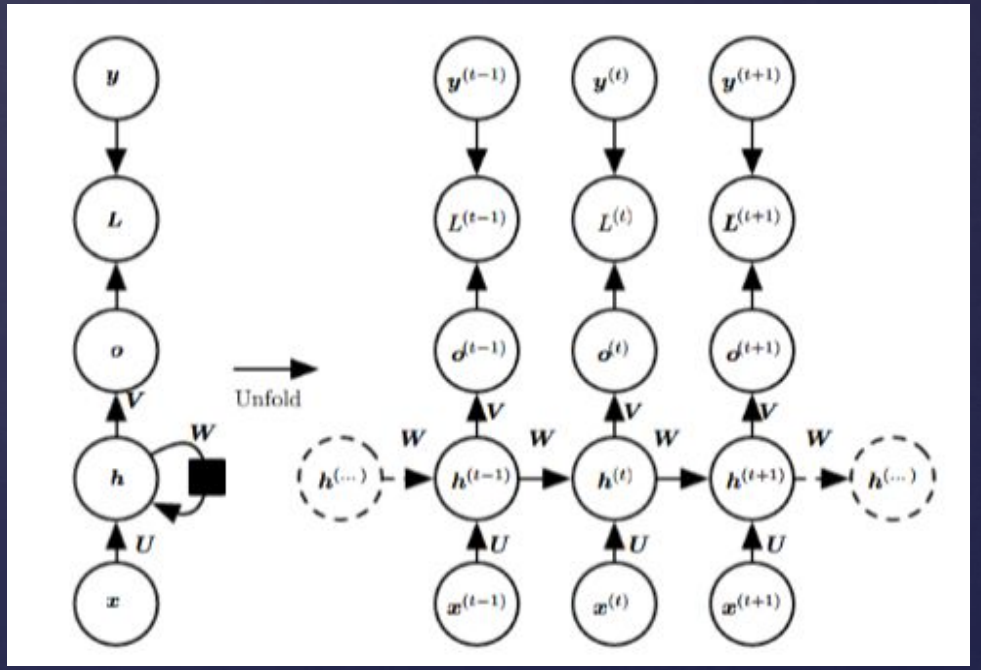
- A loss ‘L’ measure the difference between the actual output y

and the predicted output o.

- The RNN has also input to hidden connection parameterized by

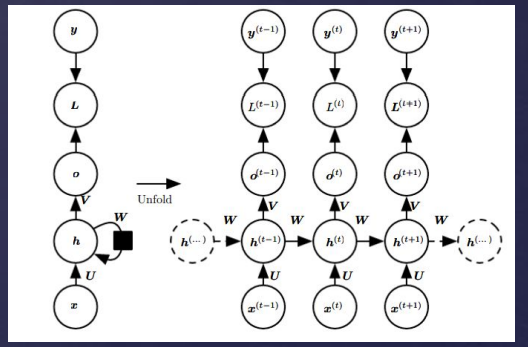
a weight matrix U, hidden to hidden connections parameterized by a weight matrix W, and hidden-to- output connections parameterized by a weight matrix V.

- The softmax operation is applies as a post-processing step to obtain a vector y^ of normalized probability of the output.



- The softmax operation is applies as a post-processing step to obtain a vector y^ of normalized probability of the output.
- Then from time step t = 1 to t = n we apply the following equation:

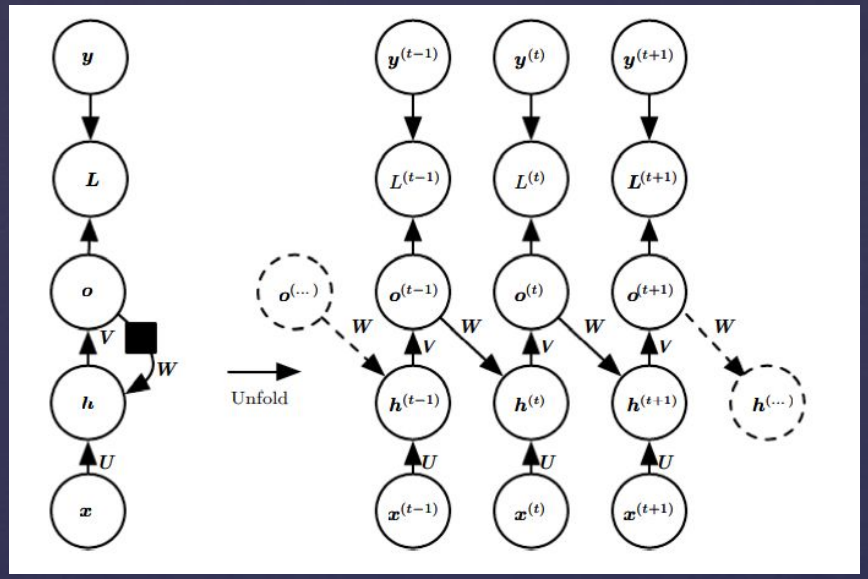
$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$



- These are the forward propagation equations of the recurrent neural network where U, V, W are the weight matrix that is shared among each time step.
- The above equations are also known as forwarding propagation of RNN where the b and c are the bias vectors and tanh and softmax are the activation functions.

- To update the weight matrix U, V, W we calculate the gradient of the loss function for each weight matrix i.e. $\partial L/\partial U$, $\partial L/\partial V$, $\partial L/\partial W$, and update each weight matrix with the help of a back-propagation algorithm.
- When a back-propagation algorithm is applied to RNN, it is sometimes also known as BPTT i.e. backpropagation through time.

- Another variation that can be done in recurrent neural network architecture is that we can change the recurrent connection from hidden to hidden state and make it from output to hidden state.



(The negative log-likelihood function is commonly used as a loss function in machine learning because it is a natural measure of the distance between the predicted probability distribution and the true probability distribution)

- Example of RNN:

This is an example of an RNN that maps an input sequence to an output sequence of the same length

Total loss for a given sequence for a given sequence of $\boldsymbol{x}$ values with a sequence of $\boldsymbol{y}$ values is the sum of the losses over the time steps

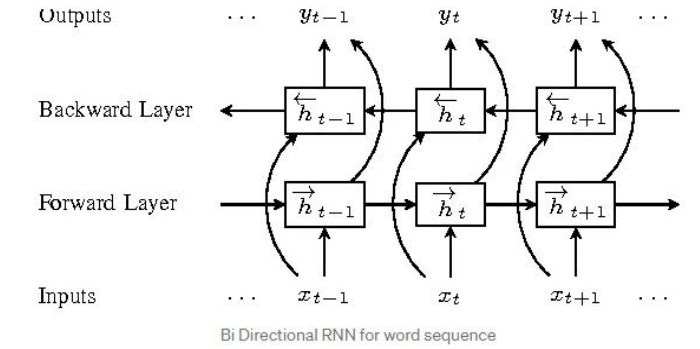
If $L^{(t)}$ is the negative log-likelihood of $\boldsymbol{y}^{(t)}$ given $\boldsymbol{x}^{(1)},..\boldsymbol{x}^{(t)}$ then

$$L\left(\left\{\boldsymbol{x}^{(t)},..\boldsymbol{x}^{(t)}\right\},\left\{\boldsymbol{y}^{(t)},..\boldsymbol{y}^{(t)}\right\}\right)=\sum_{t}L^{(t)}$$

$$=-\sum_{t}\log p_{model}\left(\boldsymbol{y}^{(t)}|\left\{\boldsymbol{x}^{(t)},..\boldsymbol{x}^{(t)}\right\}\right)$$

- where p_{model} is given by reading the entry for $\boldsymbol{y}^{(t)}$ from the model's output vector $\hat{\boldsymbol{y}}^{(t)}$

- **Bidirectional RNNs:**
  - Combine an RNN that moves forward through time from the start of the sequence Another RNN that moves backward through time beginning from the end of the sequence
  - A bidirectional RNN consists of two RNNs which are stacked on the top of each other. The one that processes the input in its original order and the one that processes the reversed input sequence.
  - The output is then computed based on the hidden state of both RNNs.



Bi Directional RNN for word sequence

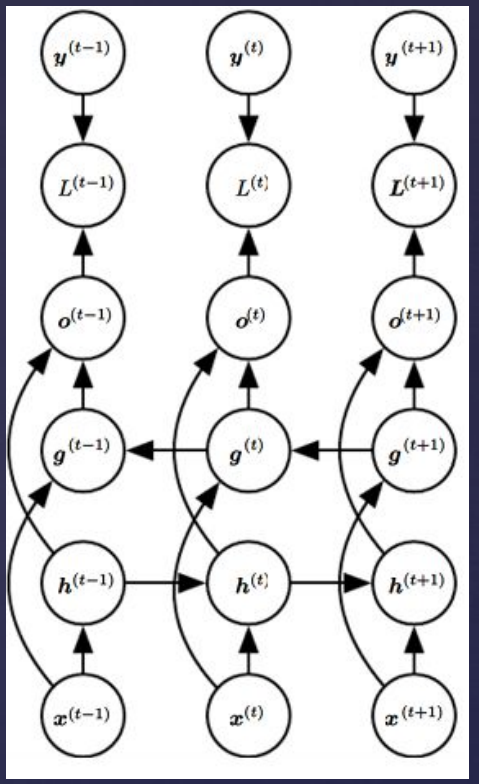
Consider the word sequence “I love mango juice”. The forward layer would feed the sequence as such. But, the Backward Layer would feed the sequence in the reverse order “juice mango love I”. Now, the outputs would be generated by concatenating the word sequences at each time and generating weights accordingly. This can be used for POS tagging problems as well.

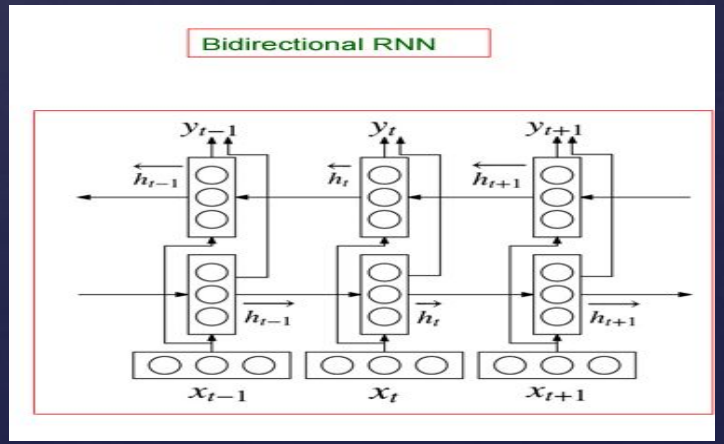
- Maps input sequences x to target sequences y with loss L(t) at each step t.

h recurrence propagates to the right g recurrence propagates to the left.

- This allows output units o(t) to compute a representation that depends both the past and the future.

- Need for bidirectionality:
- In speech recognition, the correct interpretation of the current sound may depend on the next few phonemes because of co-articulation and the next few words because of linguistic dependencies.





- **Back Propagation Through Time(BPTT)**
- Nodes of our computational graph include the parameters U, V, W, b, c as well as the sequence of nodes indexed by t for x(t), h(t), o(t) and L(t).
- For each node N we need to compute the gradient ∇NL recursively.
- Start recursion with nodes immediately preceding final loss
- We assume that the outputs o(t) are used as arguments to softmax to obtain vector y^ of probabilities over the output.
- Also that loss is the negative log-likelihood of the true target y(t) given the input so far.
- The gradient on the outputs at time step t is, for all i, t is as follows:

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}$$

- Work backwards starting from the end of the sequence. At the final time step τ, h(τ) has only o(τ) as a descendent.
- In other words, the output at the final time step τ depends only on the final hidden state, and not on any future hidden states or inputs.
- Iterate backwards in time iterating to backpropagate gradients through time from t =τ-1 down to t =1
- Once gradients on internal nodes of the computational graph are obtained, we can obtain gradients on the parameters.

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L$$

$$\begin{aligned}\nabla_{\boldsymbol{h}^{(t)}} L &= \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^\top (\nabla_{\boldsymbol{o}^{(t)}} L) \\ &= \boldsymbol{W}^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^\top (\nabla_{\boldsymbol{o}^{(t)}} L)\end{aligned}$$

where $\operatorname{diag}\left(1 - (\boldsymbol{h}^{(t+1)})^2\right)$ indicates a diagonal matrix with elements within parentheses. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t+1$

$$
\begin{aligned}
\nabla_{\boldsymbol{c}} L &= \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^{\top} \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L \\
\nabla_{\boldsymbol{b}} L &= \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^{\top} \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L \\
\nabla_{\boldsymbol{V}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top} \\
\nabla_{\boldsymbol{W}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)\top} \\
\nabla_{\boldsymbol{U}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)\top}
\end{aligned}
$$

- **Vanishing/Exploding Gradients Problem:**
- Common problems that occur during the backpropogation of time series data are the vanishing and exploading gradients.
- The vanishing gradient problem is essentially in which deep multilayer feed forward network or a RNN does not have ability to propogate useful gradient information from the output end of model back to the layers near the input end of model.
- As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the vanishing gradients problem.

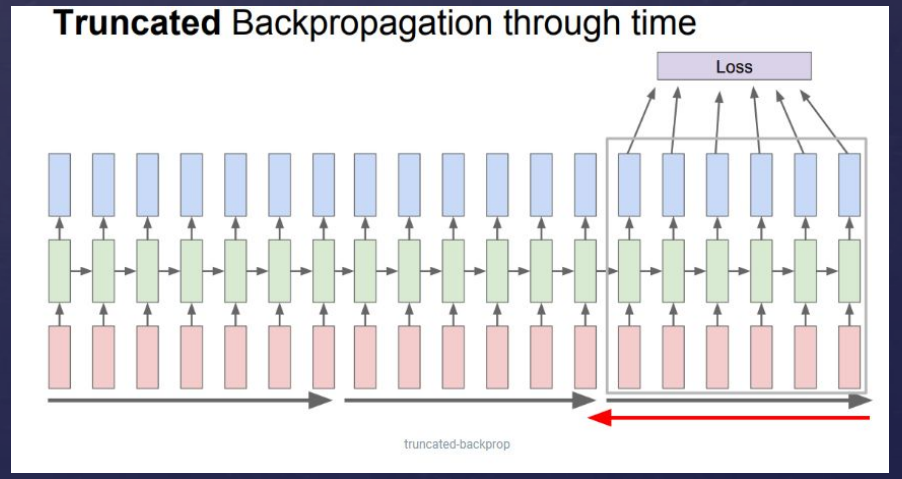
- Exploding gradients are a problem when large error gradients accumulate and result in very large updates to neural network model weights during training. A gradient calculates the direction and magnitude during the training of neural network and it is used to teach network weights in the right direction by right amount.
- In some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the exploding gradients problem.

Exploding	Vanishing
• There is an exponential growth in the model parameters. • The model weights may become NaN during training. • The model experiences avalanche learning.(the model's accuracy "avalanches" or rapidly declines when presented with new and unfamiliar data.)	• The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all). • The model weights may become 0 during training. • The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

- Methods that are proposed to overcome the vanishing gradient problem are:
  - Residual neural networks (ResNets)
  - Multi-level hierarchy
  - Long short term memory (LSTM)
  - Faster hardware
  - ReLU
  - Batch normalization
- Methods that are proposed to overcome the exploding gradient problem are:
  - Use LSTM network
  - Use Gradient clipping
  - Use Regularization
  - Redesign the neural network

Truncated Backpropagation Through Time:

- Truncated Backpropagation Through Time (truncated BPTT) is a widespread method for learning recurrent computational graphs. Truncated BPTT keeps the computational benefits of Backpropagation Through Time (BPTT) while relieving the need for a complete backtrack through the whole data sequence at every step.



- Reduced memory requirements: By truncating the backpropagation algorithm, TBPTT reduces the amount of memory required to store intermediate activations during training. This is particularly important for RNNs, which can have very long sequences of inputs.

- Faster training: By reducing the number of time steps over which the backpropagation algorithm is applied, TBPTT can speed up the training process. This can be especially useful for large and complex models.

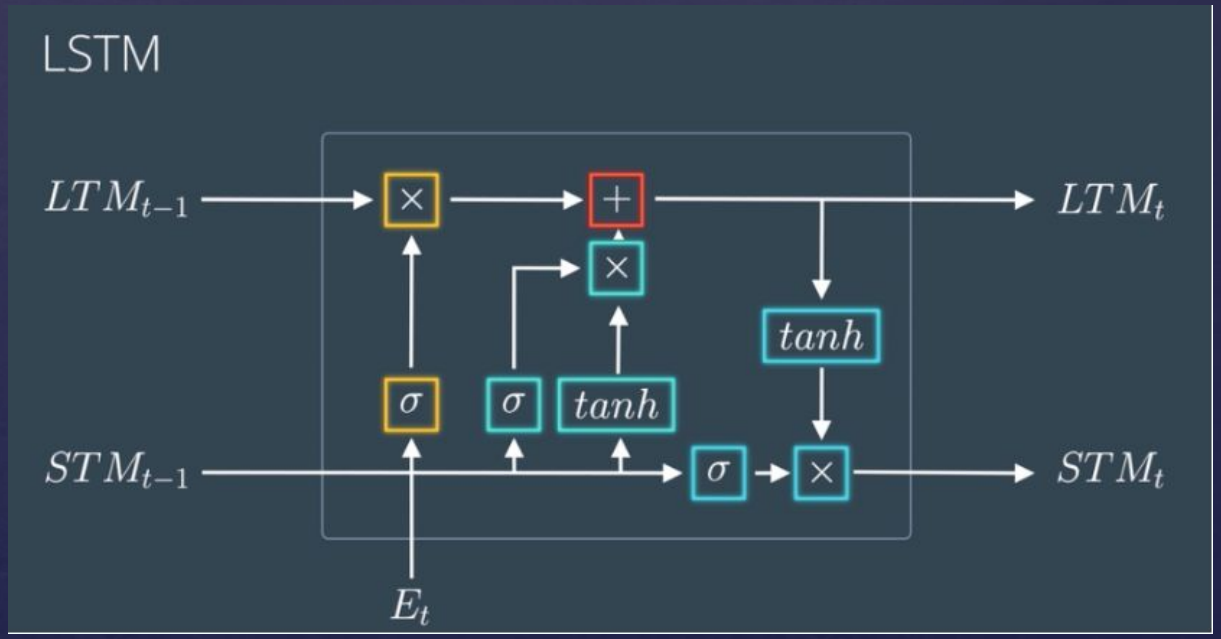
- Improved generalization: By truncating the backpropagation algorithm, TBPTT can help prevent overfitting by limiting the amount of influence that earlier time steps have on the final output. This can help the model generalize better to new data.

- Flexibility: The number of time steps used in TBPTT can be adjusted to balance between computational efficiency and model performance. This allows for greater flexibility in designing and training RNNs.

- In the backpropagation training, there is a forward pass and a backward pass through the entire sequence to compute the loss and the gradient. By taking a window, we also improve the training performance from the training duration aspect.

Long Short-Term Memory Network:

- <u>The architecture of LSTM:</u>
- LSTMs deal with both Long Term Memory (LTM) and Short Term Memory (STM) and for making the calculations simple and effective it uses the concept of gates.

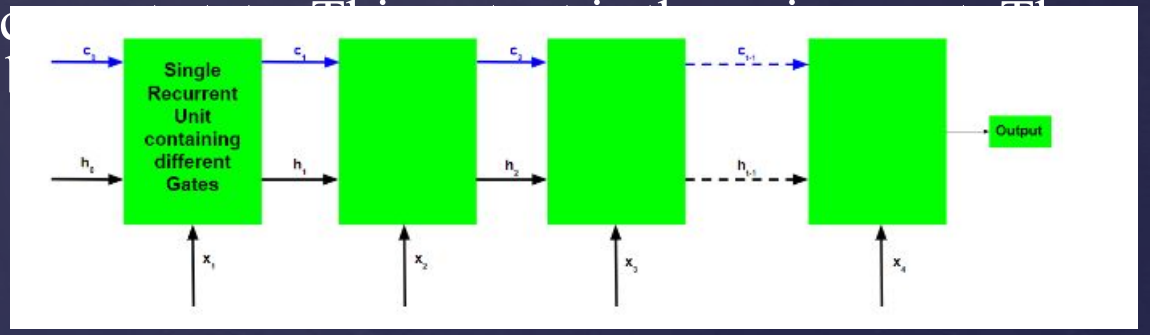


- LSTM networks are the most commonly used variation of Recurrent Neural Networks (RNNs). The critical component of the LSTM is the memory cell and the gates (including the forget gate but also the input gate), inner contents of the memory cell are modulated by the input gates and forget gates. Assuming that both of the segue he are closed, the contents of the memory cell will remain unmodified between one time-step and the next gradients gating structure allows information to be retained across many time-steps, and consequently also allows group that to flow across many time-steps. This allows the LSTM model to overcome the vanishing gradient properly occurs with most Recurrent Neural Network models.
- Forget Gate(f): (LSTM decides what to forget) At forget gate the input is combined with the previous output to generate a fraction between 0 and 1, that determines how much of the previous state need to be preserved (or in other words, how much of the state should be forgotten). This output is then multiplied with the previous state. Note: An activation output of 1.0 means “remember everything” and activation output of 0.0 means “forget everything.” From a different perspective, a better name for the forget gate might be the “remember gate”

- Input Gate(i): (LSTM decides what new information is worth adding). Input gate operates on the same signals as the forget gate, but here the objective is to decide which new information is going to enter the state of LSTM. The output of the input gate (again a fraction between 0 and 1) is multiplied with the output of tan h block that produces the new values that must be added to previous state. This gated vector is then added to previous state to generate current state

- Input Modulation Gate(g): It is often considered as a sub-part of the input gate and much literature on LSTM's does not even mention it and assume it is inside the Input gate. It is used to modulate the information that the Input gate will write onto the Internal State Cell by adding non-linearity to the information and making the information Zero-mean. This is done to reduce the learning time as Zero-mean input has faster convergence. Although this gate's actions are less important than the others and are often treated as a finesse-providing concept, it is good practice to include this gate in the structure of the LSTM unit.

- Output Gate(o): (LSTM decides what information to reveal or use). At output gate, the input and previous state are gated as before to generate another scaling fraction that is combined with the output of tanh block that brings the c
output and state are fed back into the LSTM



- In LSTM's we can selectively read, write and forget information by regulating the flow of information using gates.
- In the following few sections, we will discuss how we can implement the selective read, write and forget strategy. We will also discuss how do we know which information to read and which information to forget.

- Continuous flow: LSTM manages information. It can let in data, forget irrelevant data, pass what is important to the next step.
- Long term dependencies: Captures relationship or patterns over long term to remember things and makes LSTM powerful.