Unit III	**Sequence Modeling: Recurrent and Recursive nets:** Unfolding Computational Graphs	[T

Recurrent Neural Networks
Teacher forcing and networks with output recurrence
Computing the gradient in a Recurrent Neural Network
Recurrent Networks as directed graphical models
Modelling sequences conditioned on context with RNNs
Bidirectional RNNs
Encoder-Decoder sequence-to –sequence architectures
Deep Recurrent networks
Recursive neural networks
The Challenge of Long-Term Dependencies
Echo State Networks
Leaky Units & Other strategies for multiple timescales
The Long short-Term memory -LSTM

① Unfolding Computational Graphs:

A computation graph is a way of formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs & loss.

Unfolding the graph results in the sharing of Parameters across the deep networks structures.

Consider the classical form of a dynamical system:

$$s^{(t)} = f(s^{(t-1)}; \theta) \quad \text{(1)}$$

(θ → Parameters, $s^{(t)}$ ↓ State of System.)

The (1) Eq is a Recurrent function coz the S (state of system at time (t) refers back to the same state of system at time $(t-1)$

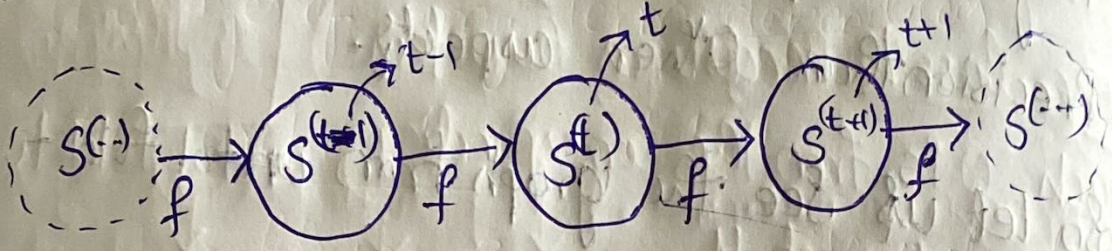
for Example: Let $t=3$.

$$s^{(3)} = f(s^{(2)}; \theta)$$

$$= f(\underbrace{f(s^{(1)}; \theta)}_{s^{(2)}}; \theta); \quad \text{(2)}$$

unfolding the Equation by Repeatedly

apply the definition formula in this way has yielded an Expression that is not Recurrence. Such Expressions can be represented by a traditional directed acyclic graph as shown below:



Now let us consider a dynamic System by an External Signal $x^{(t)}$ (a input at Previous output as)

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta) \longrightarrow (3)$$

now the State contains the info about Previous Sequences.

Many Recurrent neural networks use the below Equation to represent the hidden Units (h).

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \longrightarrow (4)$$

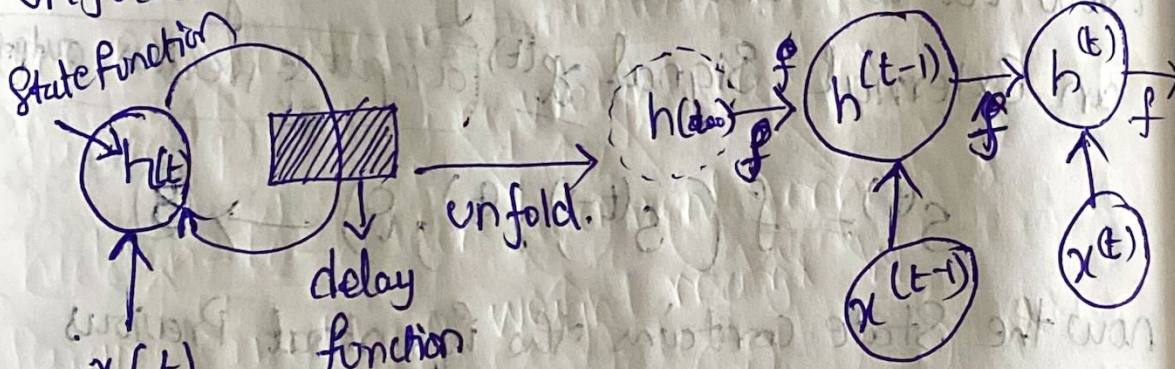
When the Recurrent network is trained to perform some tasks that need ~~future~~ Past Values, the network typically learns to use $h^{(t)}$ as a kind of lossy Summary of the Past Sequence of inputs up to t.

The input vectors such as

$(x(t), x(t-1), x(t-2), \cdots x(2), x(1))$.

Depending on the training, this above inputs might keep the past sequences with more Precision than other aspects.

So let us see the graph from ~~unfol~~ fold to unfold.



$x(t)$

↓

at each step of RNN the input is given.

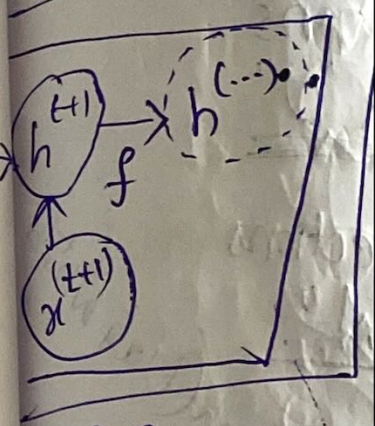
→ The Right Same network seen as an unfolded computational graph; where each node is now associated with one Particular time instance.

We Represent the unfolded recurrence (Right Side diagram) after t steps with a function $g^{(t)}$:

$h^{(t)} = g^{(t)}(x$

$= f($

② Recurr



2) Recurrens
output at e
Connections
Step to th
Step.

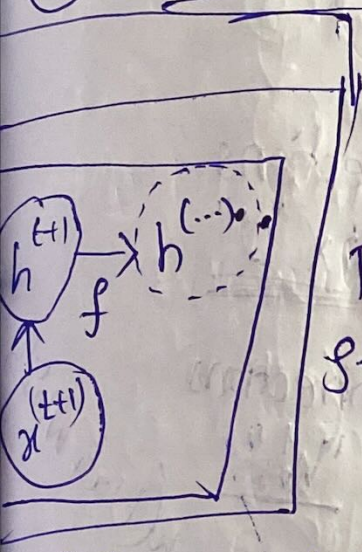
3) Recurre
Connection
entire Seq
Single outp

$$h^{(t)} = g^{(t)}\left(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)}\right) \rightarrow (5)$$

(function) ← (whole past sequence)

$$= f\left(h^{(t-1)}, x^{(t)}; \theta\right) \rightarrow (6)$$

(2) Recurrent Neural Network



Design Patterns: (3)

1) The Recurrent networks that Produce an output at each time Step and have recurrent (output-output) connections b/w hidden units.

2) Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step. (output → hidden layer)

3) Recurrent network with recurrent connections b/w hidden units, that read an entire sequence and then produce an single output. (Single output).

① ⇒

A recurrent network that maps an input sequence of "x" values to a corresponding sequence of output "o" values.

The loss L measures how far each O is from the corresponding training target "y".

The RNN has input to hidden connections Parameterized by a weight matrix "U"

hidden-to-hidden recurrent connections Parameterized by a weight matrix "W"

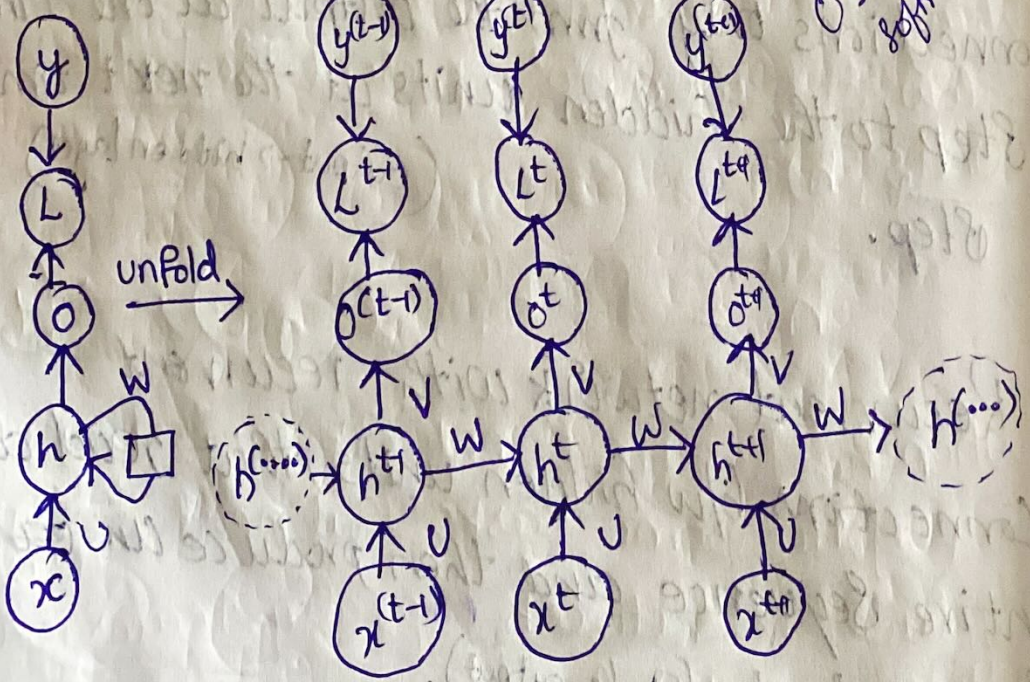
hidden-to-output → "V"

$a^t = f(W a^{t-1} + U x^t + W h^{t-1} + b)$

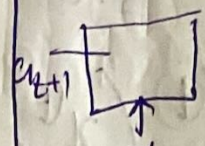
$h^t = \tanh(U x^t + W h^{t-1} + b)$

$o^t = (hV + c)$

softmax



$a^t = f(W a^{t-1} + U x^t + b_a)$



$(n\_x, 1)$

↓

Vocabulary Size.

$a = f(W_{aa} * a_{t-1} + W_{ax} * X + b_a)$

→ Here we apply the hyperbolic tangent activation.

→ also we assume that output is discrete as the RNN is used to predict words or characters.

→ We can apply the Softmax operation as a post processing step.

$$a^{(t)} = b + W h^{(t-1)} + U x^{(t)} \rightarrow (1)$$

(b: bias; $h^{(t-1)}$: previous hidden output; $x^{(t)}$: input)

$$h^{(t)} = \tanh(a^{(t)})$$

$$o^{(t)} = c + V h^{(t)}$$

(c: bias; $h^{(t)}$: hidden output)

$$y^{(t)} = \text{Softmax}(o^{(t)}).$$

Where the parameters are the bias vectors b, c along with the weight matrices U, V and W, respectively.

The total loss for a given sequence of "x" values paired with a sequence of "y" values would then be just the sum of the losses over all the time steps.

If $L^{(t)}$ is negative log-likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(t)}$ then [BPTT]

$$L\left(\{x^{(1)}, \ldots, x^{(\tau)}\}, \{y^{(1)}, \ldots y^{(t)}\}\right)$$

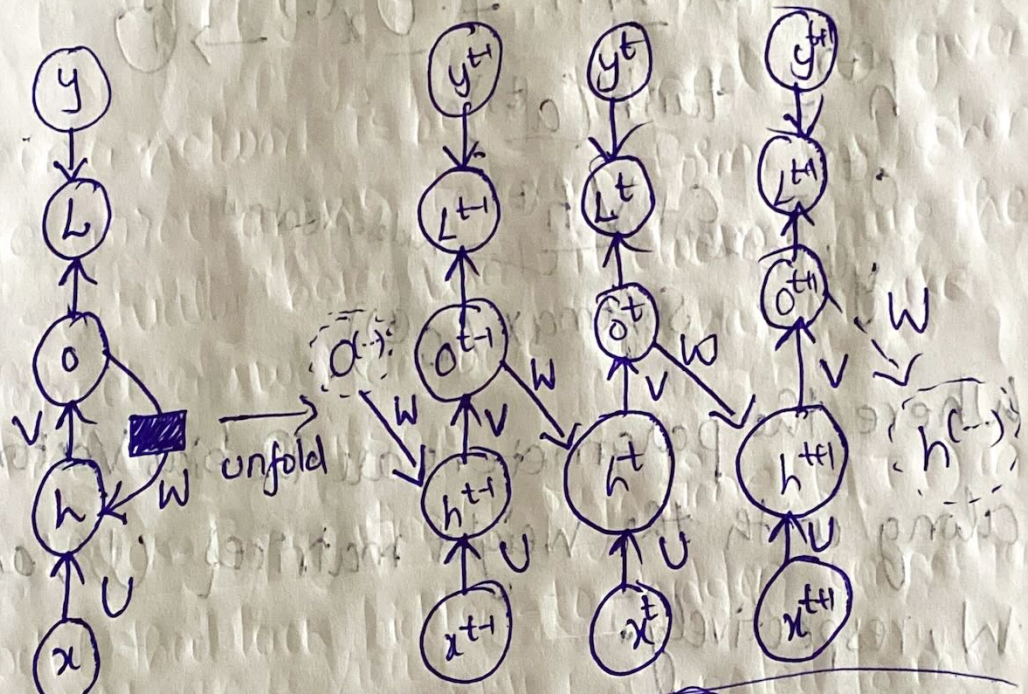
$$= \sum_{t} L^{(t)}$$

$$= -\sum_t \log p_{model}\left(y^{(t)} \mid \{x^{(1)}, \ldots, x^{(t)}\}\right),$$

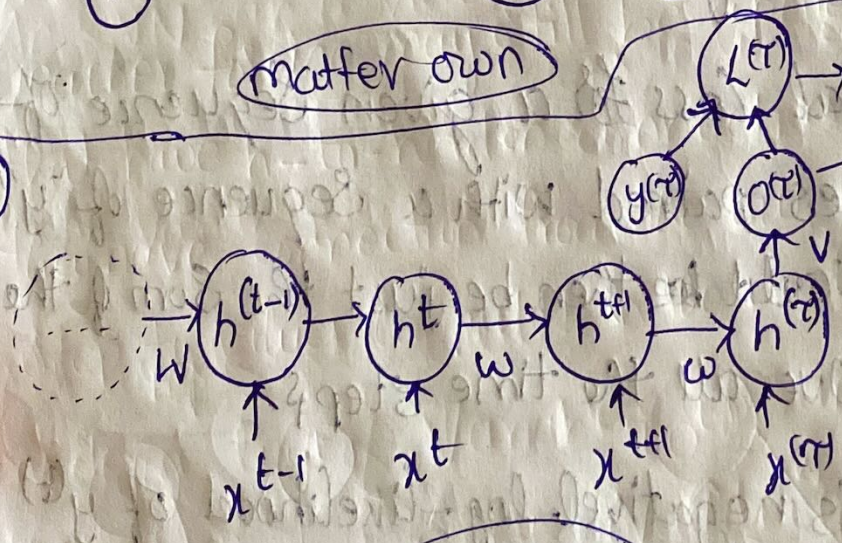
↓

given by reading the entry for $y^{(t)}$ from the output vector $\hat{y}(t)$.

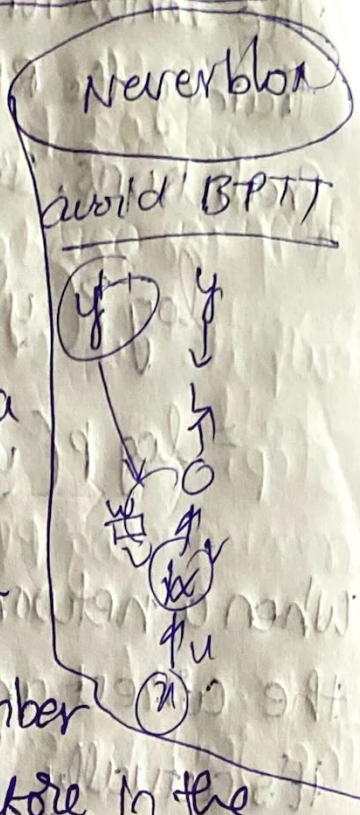
(2)



(3)



③ Teacher forcing and networks with output recurrence.



The RNN are designed in a way that they can't fully act like a Turing machine, which is a concept of a device that can calculate anything given enough time. These RNN can't Remember Everything that has happend before in the sequence, which limits their complexity.

The teacher forcing is a technique used when training RNNs. It involves using the correct expected output at the current step in the sequence to help predict the next step during the training. For example: if you we are teaching the network to predict the next word in a sentence, we give it the correct word as a clue each time.

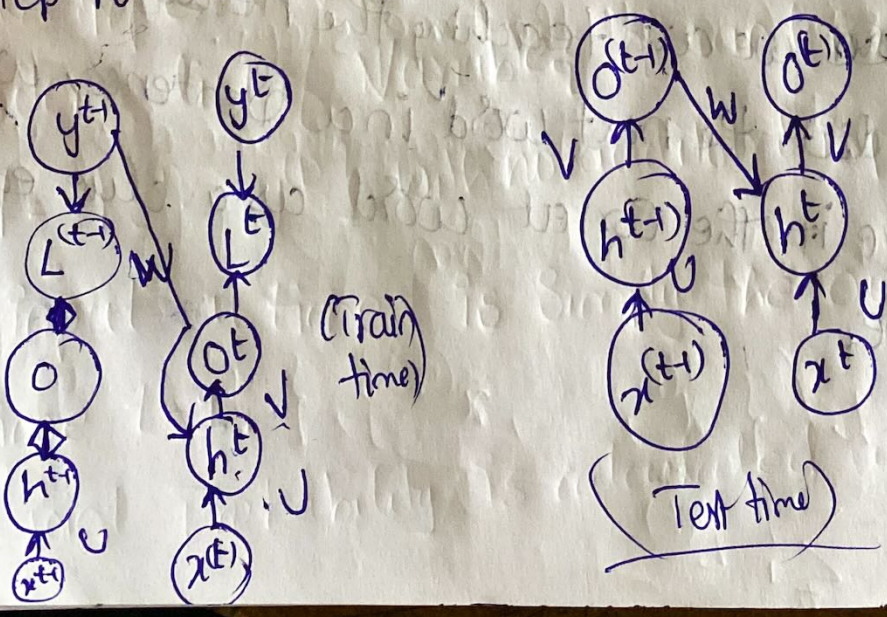
The condition maximum likelihood is about training the RNN to make the most likely Predictions. It's criterion is:

$$\log P(y^{(1)}, y^{(2)} \mid x^{(1)}, x^{(2)})$$

$$= \log P(y^{(2)} \mid y^{(1)}, x^{(1)}, x^{(2)}) + \log P(y^{(1)} \mid x^{(1)}, x^{(2)})$$

When a network is being trained, it's given the correct answer to learn form, But when it's actually being used, it doesn't have the Correct answer anymore and has to make Predictions on it's own.

The Back-Propagation Through time (BPTT) in models that lack hidden-to-hidden connections. Teacher forcing may still be appiled to models that have hidden-to-hidden connections as long as they have connection from the output at one time step to values computed in the next time step.



The disadvantage of teacher learning is that if the network is trained with T.F, it might become too reliant on the correct answer that is Open-loop it sees during training and might not learn to make good predictions on its own.

To overcome this, we can mix the correct answers with its own Predictions during training.

Recurrent Networks as Directed Graphical model:

*) In RNNs, losses are calculated between the Predicted outputs and the actual training targets, Similar to feedforward networks.

*) There are Different loss functions which can be used in RNNs, such as cross-Entropy or mean Squared Error, depending on the task and how the output distribution is modeled.

*) Training an RNN using a predictive log-likelihood objective involves estimating the conditional probability of the next sequence element $y^{(t)}$ given based on the past inputs,

aiming to maximize this likelihood.

$$\log P(y^{(t)} | x^{(1)}, \ldots, x^{(t)})$$

& if the model includes connections from the outputs at one time step to the next time step,

$$\log P(y^{(t)} | x^{(1)}, \ldots, x^{(t)}, y^{(1)}, \ldots, y^{(t-1)}).$$

The joint probability of a sequence can be decomposed into the product of conditional probabilities, capturing the full distribution across the sequence.

An RNN can be viewed as a graphical model, where the directed edges indicate dependencies b/w variables.

~~as an example, RNN model~~

Consider a list of variables of random $Y = \{y^{(1)} \ldots y^{(t)}\}$ with no additional inputs x.

The input at time t is simply the output at time t-1.

The joint probability is calculated using a

chain Rule: the break down the probability of Entire Sequence into prob of each item given the previous ones.

$$P(Y) = P(y^{(1)}, \dots y^{(t)}) = \prod_{t=1}^{T} P(y^t \mid y^{(t-1)}, y^{(t-2)}, \dots y^{(1)}),$$

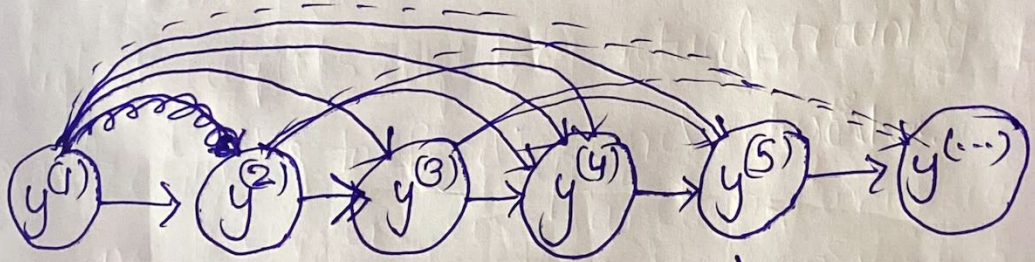
$$L = \sum_{t} L^{(t)},$$

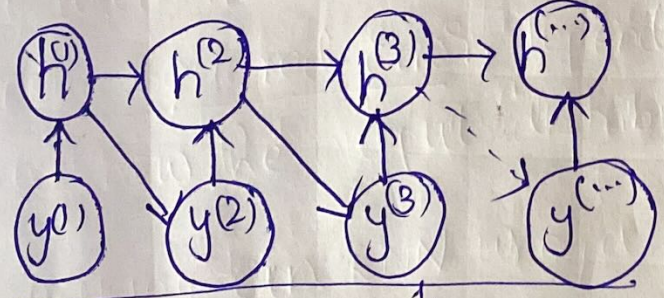
where

$$L^{(t)} = -\log P(y^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \dots y^{(1)}).$$

The Edges in a graphical model indicates which variable depends directly on other Variable.

→ RNN use parameters Sharing across time steps for Efficiency, meaning the same parameters are used for diff parts of the input Sequence.





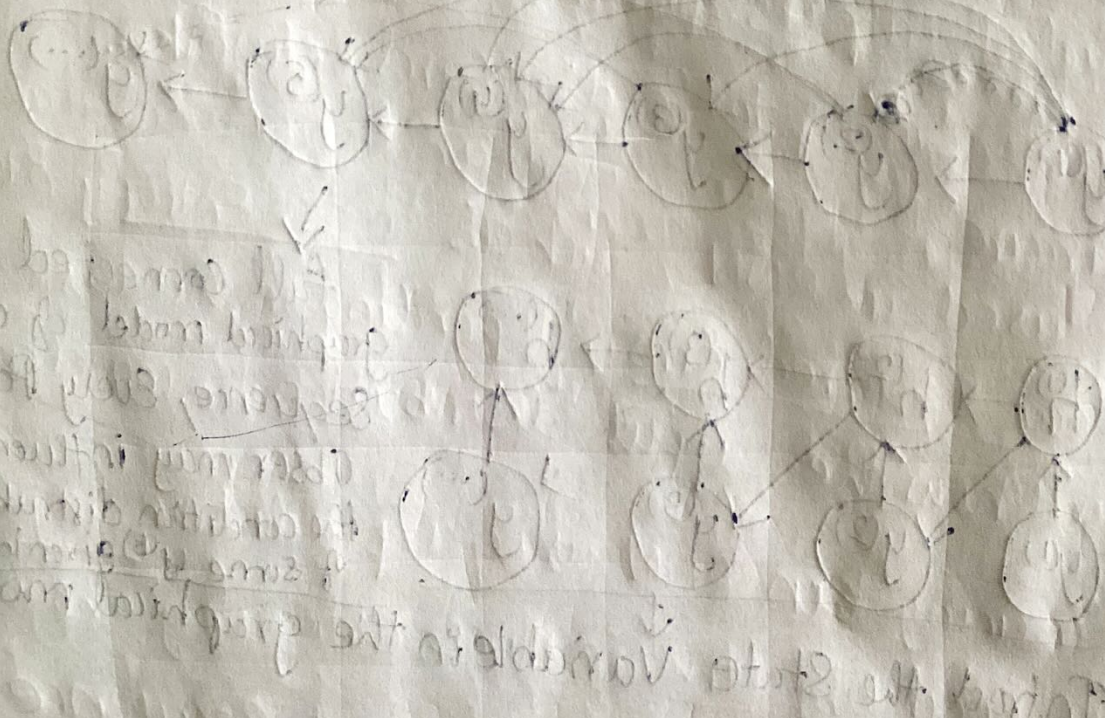
Full connected graphical model of a Sequence, Every Past Obser may influence the condition distribution of some $y^{(t)}$ given its Previous

Introd the State Variable in the graphical model of the RNN.

Some models assume only recent info is needed for the next prediction, but RNNs can in principal consider longer histories.

Despite their effeciency, RNNs may find it challenging to predict values that are missing in the middle of a sequence.

Different strategies are mentioned for deciding when to stop generating a sequence, such as using a special symbol, a Bernoulli output or predicting the sequence length.



5) Bidirectional RNN:

The Bi-RNN are an advanced type of Neural network designed for Sequence processing. They improve upon standard Recurrent Neural Networks by processing data in both directions with two separate layers, which allows them to have both backward and forward information about the sequence at every point in time.

→ Unlike the standard RNNs that only process data from Past to Present, Bi RNN add another layer the process data from future to past.

→ The Bi RNNs are designed to capture info from the Entire input Sequence, which is not possible with Unidirectional RNNs that only process inputs from Past to Present.

→ By considering all available input info, Bi RNNs can make more accurate predictions.

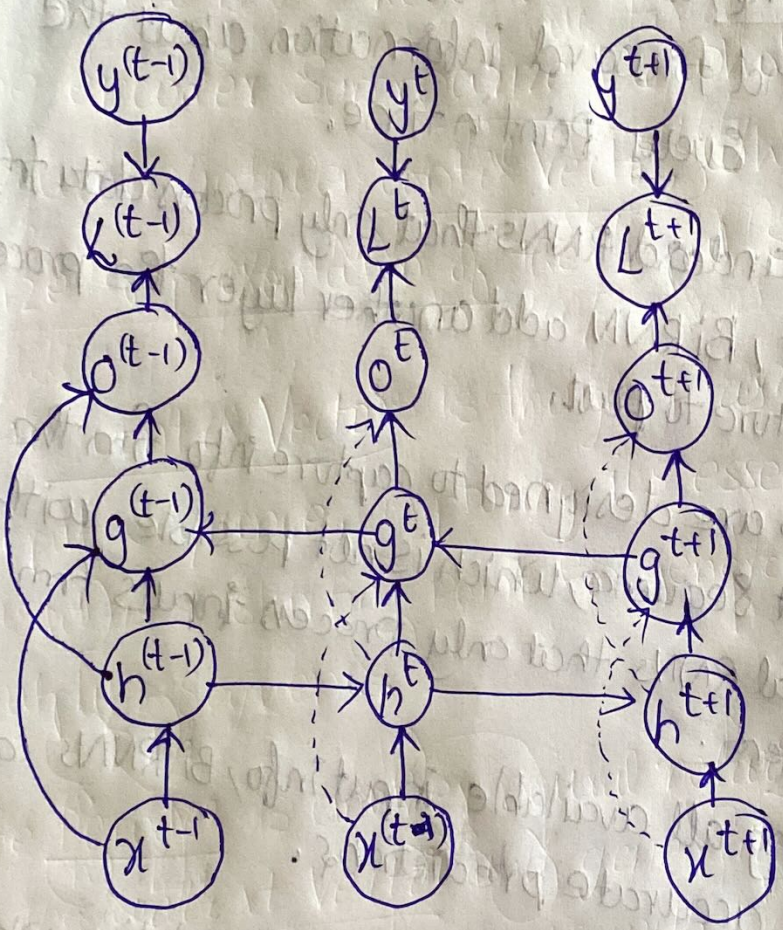
→ They are mostly useful in language translation, where meaning can depend on surrounding words.

→ It has 2 sub networks → one processes the Sequence from start to end and another from end to start.

→ The concept of Bi RNN can also be Extended to multi-dimensional data, like image processed in multiple directions to capture patterns over different parts of the input.

Training Bi-RNNs can be more complex due to the bidirectional info flow, ~~but they~~

Bi RNN maintain 2 hidden States for each time step, one for forward processing and one for backward processing, which are then combined.

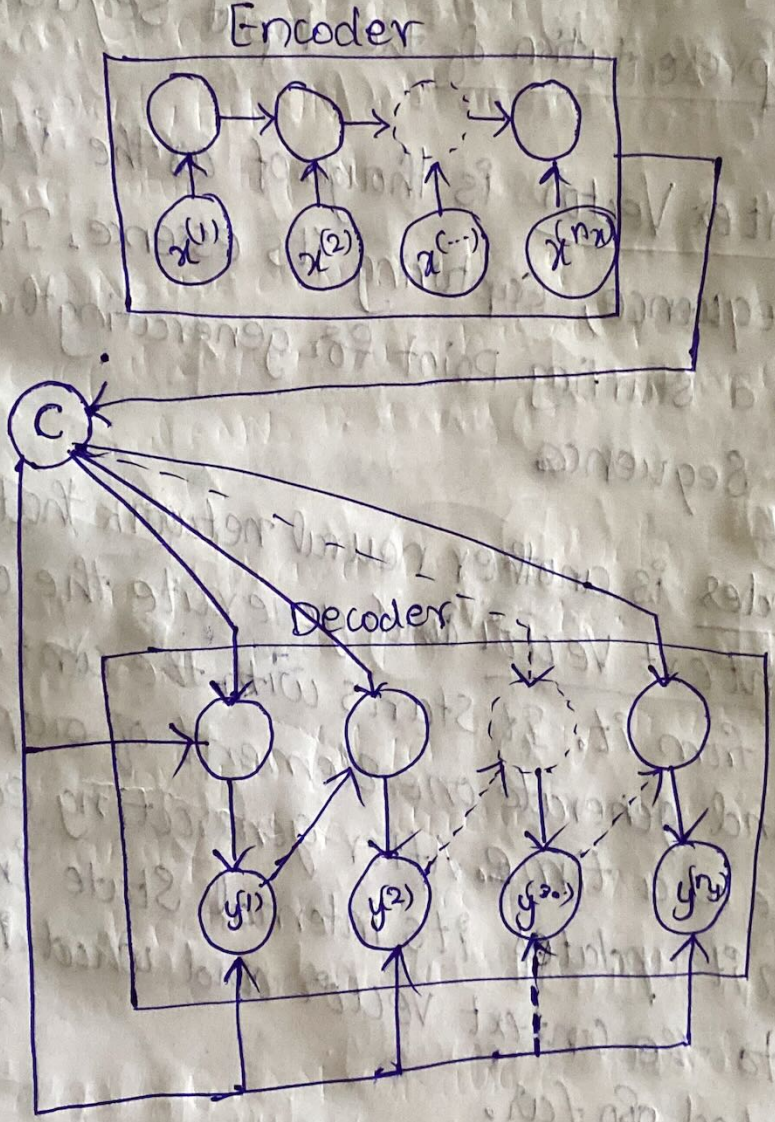


h → Procen info forward in time

g → Procen info backward in time.

Encoder-Decoder Sequence-to-Sequence architecture

The Encoder-Decoder (or) Sequence to Sequence architecture is a design pattern in neural networks that is especially suited for transforming sequence from one domain to another, like translating a sentence from one language to another.



$x^{(1)}, x^{(2)} \ldots \rightarrow$ input sequence.

$y^1, y^2, \ldots \rightarrow$ output sequence.

The encoder is a neural network that process the input sequence. It reads each element x of the input sequence one by one, and with each element, it updates its internal state to capture info about the sequence so far. Once it has Read the Entire input sequence, the encoder summarize the whole input sequence into a context vector, which is fixed-size representation of the input sequence.

The Context Vector is though of as the info of the input sequence, capturing its essence. It's used as a starting point for generating the output sequence

The Decoder is another neural network that takes the context vector and generate the output sequence from it. It starts with the context vector and generate one element of output sequence at a time. After generating each element, it updates its internal state in Response to the context vector and what it has generated so far.

The Sequence Generation continues until the decoder generate the Entire output sequence. The length of the output sequence can be diff

from the input sequence.

The Advanced model may include an ATTENTION MECHANISM that allows the decoder to focus on diff part of the context vector at diff steps of the generation process.

Deep Recurrent Network

The Deep Recurrent Neural Networks enhace the standard RNN architecture by adding more complexity and depth to better capture info in sequence. Here's how to Exp

A Basic RNN process sequence through 3 main stages → it takes an input to update its hidden states, process this hidden state through time to maintain memory, and finally generate an output from the hidden state.

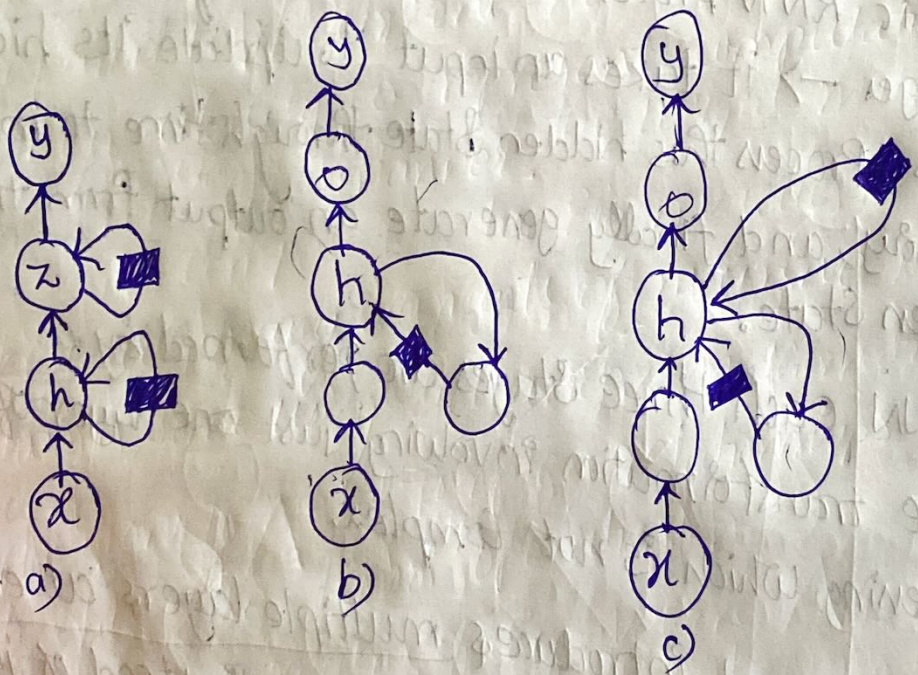
In RNN each three stages are performed by a simple transformation involving just one layer of processing, which is not complex.

A Deep RNN introduces multiple layers at each stage, not just one. This means instead of just having a single step from input to hidden state or from one hidden state to the next, there are multiple layers making these transitions, adding depth.

*) These additional layers allows the network to learn more complex patterns and perform more sophisticated transformation of the input data.

*) However adding depth can make the network harder to train. It can also make it harder for the changes in the input to hidden state and the output when they are far apart in time.

*) To help with the difficulty in training, some Deep RNN use skip connections that allows information to jump across layers, helping to address the issue of long term dependencies.

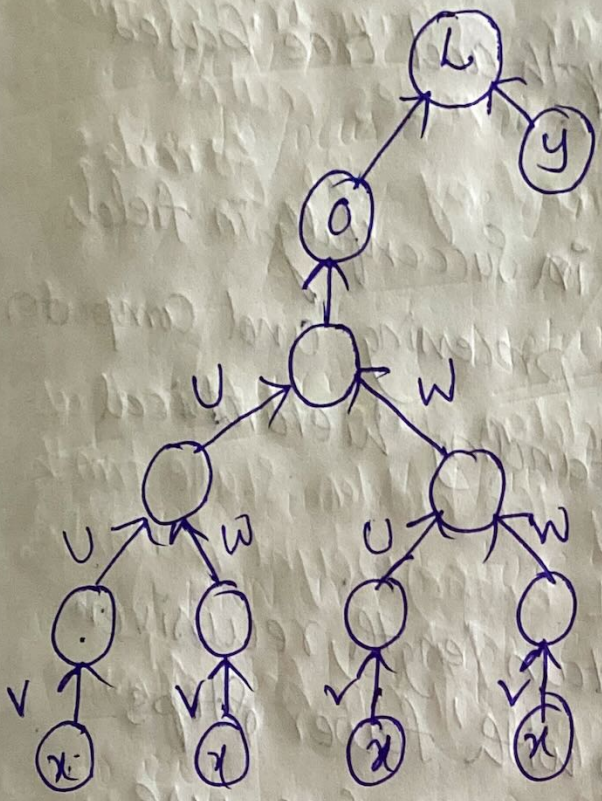


a → a hidden recurrent state that can broken down into groups organized hierarchally.

b → Deep computation can be introduced in input-to-hidden, hidden-to-hidden, hidden-to-output parts.

c → Path-lengthening effect can be mitigated by introducing skip connection.

Recursive Neural Networks



The Recursive neural network are a type of neural network that process data with a hierachical, tree-like structure. unlike traditional recurrent neural network that process data in sequence with a chain-like structure, recursive network operate on data that naturally forms a hierachy or tree.

The Recursive neural network organize and process data in a tree structure, which allows them to work with data where relationships are not strictly sequential but hierarchical.

They can Reduce the complexity of dealing with long-term dependice in Sequences. Because of their depth and tree based Processing.

They have been used succesfully in fields like natural language procesing and computer vision, where understanding hierarchical structure is imp.

for Sequence of a certain length, recursive nets can process info with fewer steps compared to recurrent nets because of their ability to represent computations in a tree-like manner.

There are many variants of recursive neural networks, some using advanced mathematical operations to model complex relationships with data like tensor operations.

The challenges of long term Dependicies:

Long-term dependicies in Recurrent neural networks(RNNs) is challenging due to issues with gradients, which are the mechanisms that neural networks use to learn.

RNN are designed to remember info fm the previous inputs by using their internal state.

RNN struggles to learn fm data points that are far apart in time sequences (long term dependicies). This is because the info fm past either fades away or becomes too dominant over time due to the Repeated multiplication of gradients in the training process.

Vanishing & Exploding gradients:

The gradients can become to small (vanish) or too large (explode) as they Propagate back through each step of the RNN. When gradients vanish, the network can't learn long term dependencies. When they explode, they can cause learning to be unstable.

An RNN composes the same function many times, once per time step, which can lead to extremely nonlinear behaviour. The function composition in RNNs resembles matrix multiplication, described by the formula:

$$h^{(t)} = W^T h^{(t-1)}$$

This represents the hidden state at time t as a result of transforming the hidden state from the previous time step $t-1$ by the weight matrix W. Without nonlinear of input it can be Represented as $h^{(t)} = (W^t) h^{(0)}$

Eigen Decomposition: If the weight matrix W can be composed into Eigen values and Eigen vectors, the Equation becomes:

$$h^{(t)} = Q \Lambda^t Q^T h^{(0)}$$

where Q is a matrix of EigenVectors, Λ is a diagonal matrix of Eigen values, t is the time step.

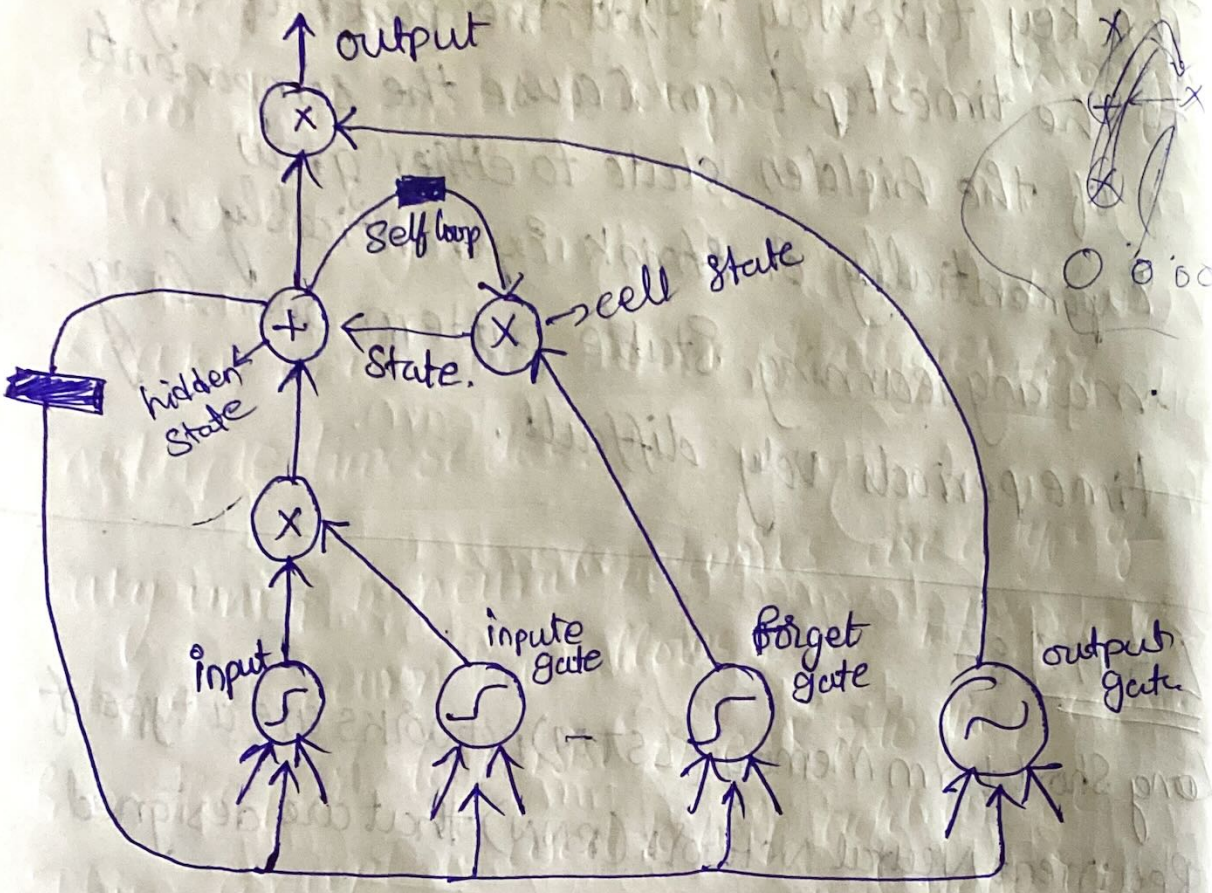
The key takeway is that the power of Eigen values to the timestep t can cause the components of the hidden state to either grow Exponentially or shrink Exponentially, making learning Stable pattern over long time periods very diffall.

LSTM :-

Long Short term Memory (LSTM) networks are a type of Recurrent Neural Network (RNN) that are designed to remember info for long Periods.

In Regular RNN have tonuble Learning dependcies b/w Events that are Speaated by long time periods due to the vanishing gradient problem, where the influence of input data fades over time during training.

LSTM introduces Structures called "gates" that Regulate the flow of information. These gates can learn which data in a Sequence is important to keep & throw away, ~~making~~ making it Possible to retain info for over longer periods.



Cells of LSTM replace the hidden units of RNN.

The Gates in LSTM:

Forget Gate: Decides what info. should be thrown away or kept.

$$f^{(t)} = \sigma\left(b_f + U_f x^{(t)} + W_f h^{(t-1)}\right)$$

Input Gate: updates the cell state with new information.

$$i^{(t)} = \sigma\left(b_i + U_i x^{t} + W_i h^{(t-1)}\right)$$

Output gate: Decides what the next hidden state should be.

$$o^{(t)} = \sigma(b_o + U_o x^{(t)} + W_o h^{(t-1)})$$

Cell State: The internal memory of the LSTM that carrier information throughout the Processing of the Sequence.

$$s^t = f^{(t)} * s^{(t-1)} + i^t * g^t$$

Hidden State $h^{(t)}$: The output state of the LSTM, which is used for predictions.

$$h^{(t)} = \tanh(s^{(t)}) * o^{(t)}$$

Here in the above equation "σ" Represents as the Sigmoid function that output a number between 0 to 1, effectively acting as a gatekeeper that allows certain info to pass through or not.

$*$ → Element wise multiplication

b, U, W → Parameters of the model from training data.

Learning and memory:- Because of this gate the LSTM can learn which info is relevant to remember and which can be discarded,

enabling them to maintain a more constant Error that can be backpropagated through time & layers.

Leaky units and other Strategies for multiple timescales.

Leaky units in neural networks are a type of artificial neuron that allows some information from the past to "persist" or "leak" into the present. This concept is used to help address one of the challenges in training neural networks,

Adding Skip Connections through Time:

This involves creating direct connection from Past info to the present within the network By adding these "skip connections", the network can access and leverage relevant info from earlier time steps more effectively, without having to pass through every single intermediate step.

(2) Leaky units and a Spectrum of Different time Scales:

Leaky units are designed to remember inform for varying length of time like short time memory and long time memory. They work by integrating past information at a ~~controlled~~ controlled rate, similar to how running average works. By adjusting the rate at which they "leak" information, these units can maintain a memory of past inputs for a longer or shorter period as needed.

(3) Removing Connections: Instead of maintaining connections b/w all units over all time steps, this strategy involves removing some connections to streamline the networks structure. This can lead to more efficient processing and can help in preventing the network from becoming overwhelmed by less relevant information.