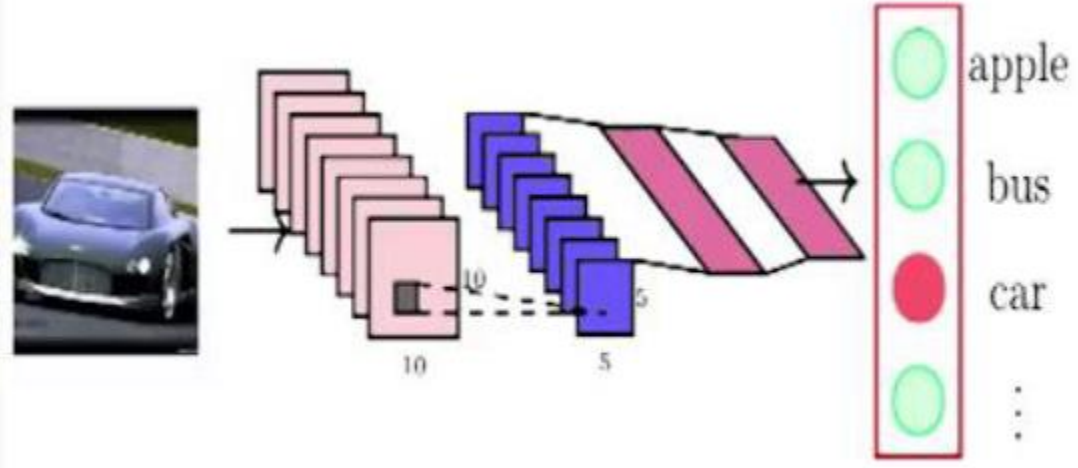
What are Sequence Learning Problem?

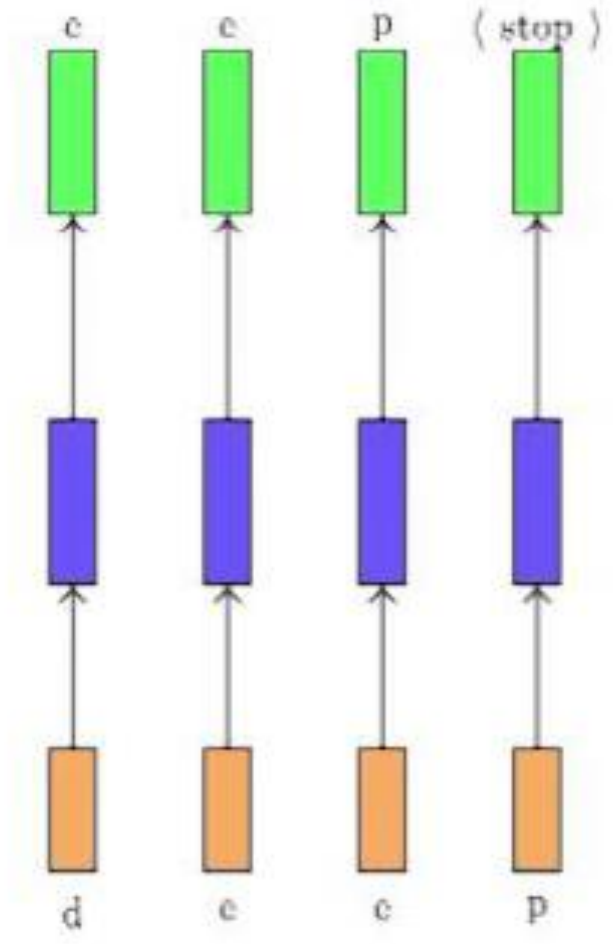
Introduction

- We have dealt with two types of networks:
- feedforward neural networks and
- convolution neural networks

- In both these networks the input was always of a fixed size.

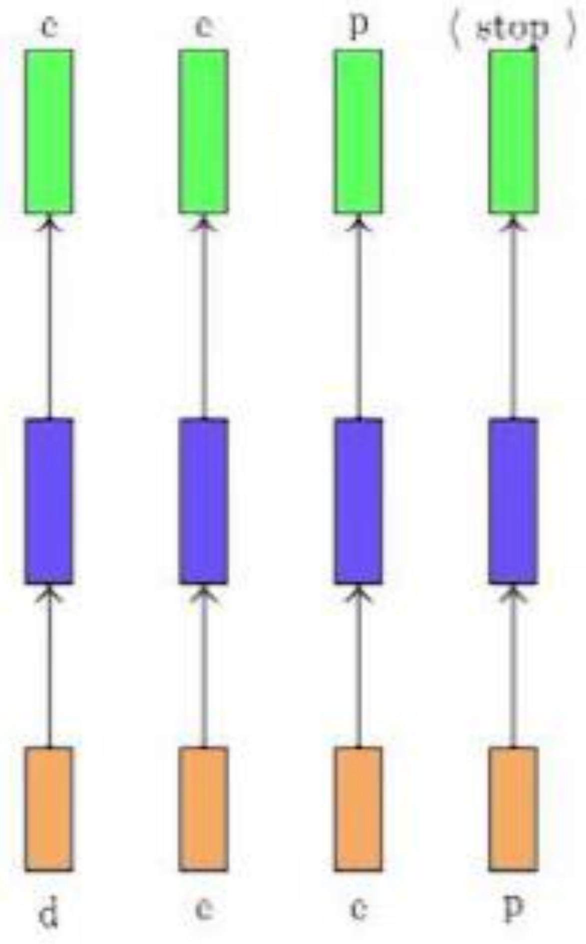


- In feedforward and convolutional neural networks, the size of the input was always fixed.
- Further, each input to the network was independent of the previous or future inputs.
- For example, the computations, outputs and decisions for two successive images are completely independent of each other.

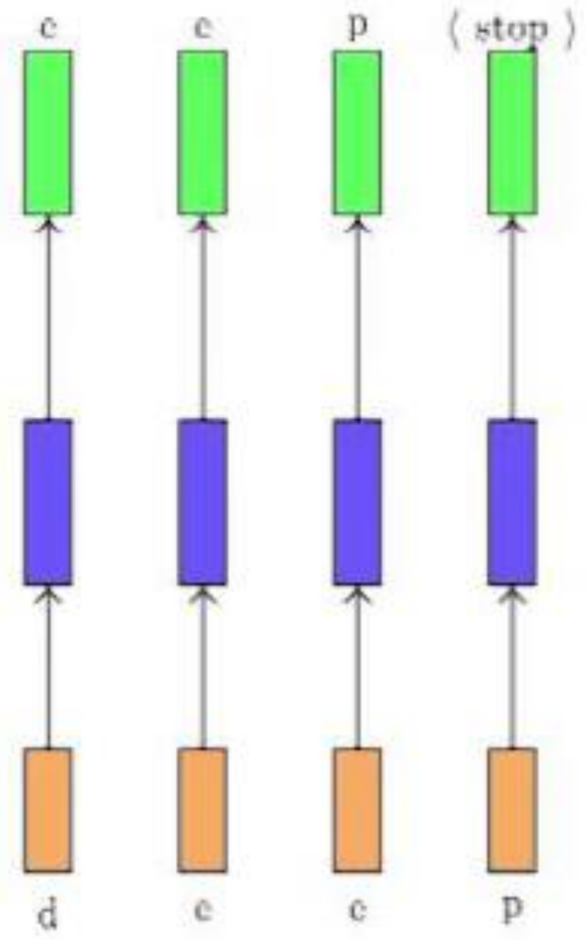


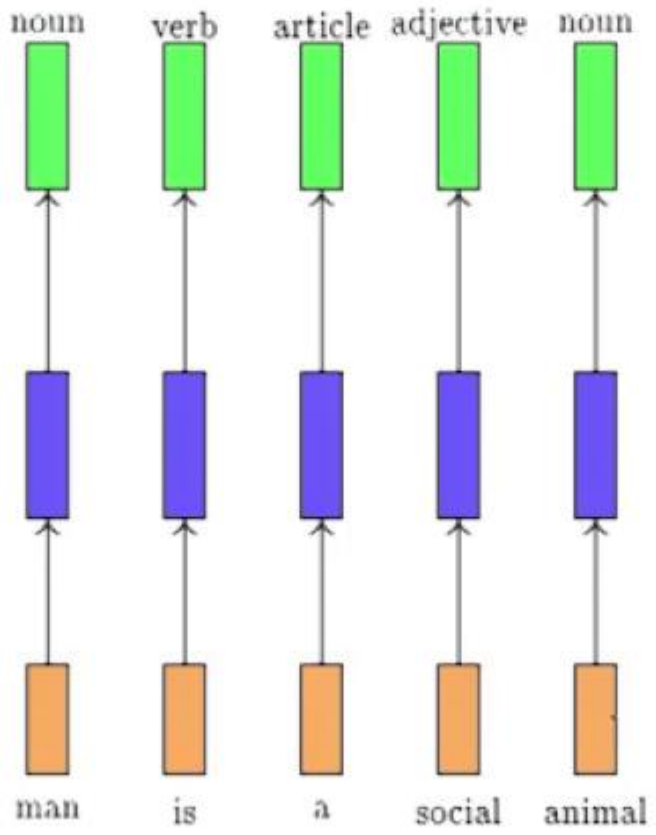
- In many applications the input is not of a fixed size.
- Further successive inputs may not be independent of each other.
- Consider the example of auto completion
- Given the first character ‘d’ you want to predict the next character ‘e’ and so on.

- Notice a few things
- First, successive inputs are no longer independent (while predicting 'e' you would want to know what the previous input was in addition to the current input).
- Second, the length of the inputs and the number of predictions you need to make is not fixed (for example, "learn", "deep", "machine" have different number of characters)
- Third, each network (orange-blue-green structure) is performing the same task (input: character , output : character)

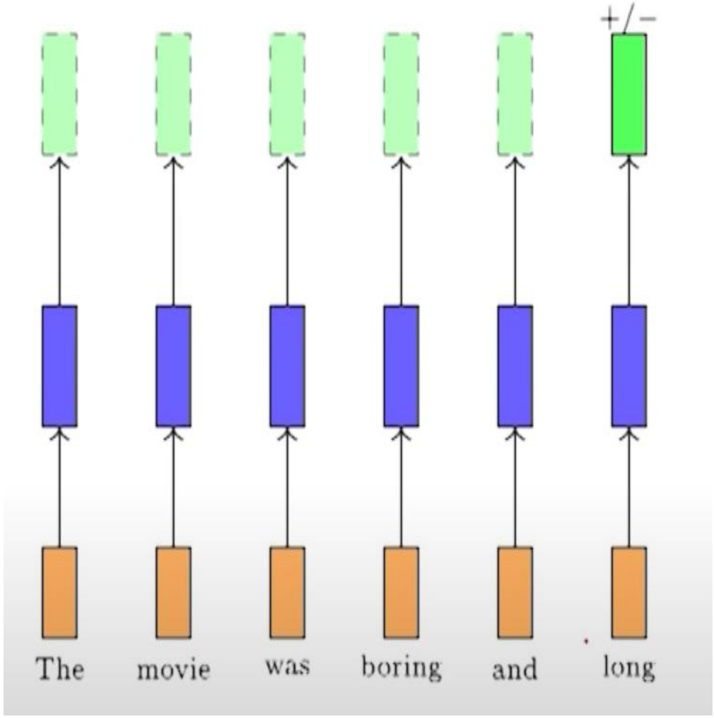


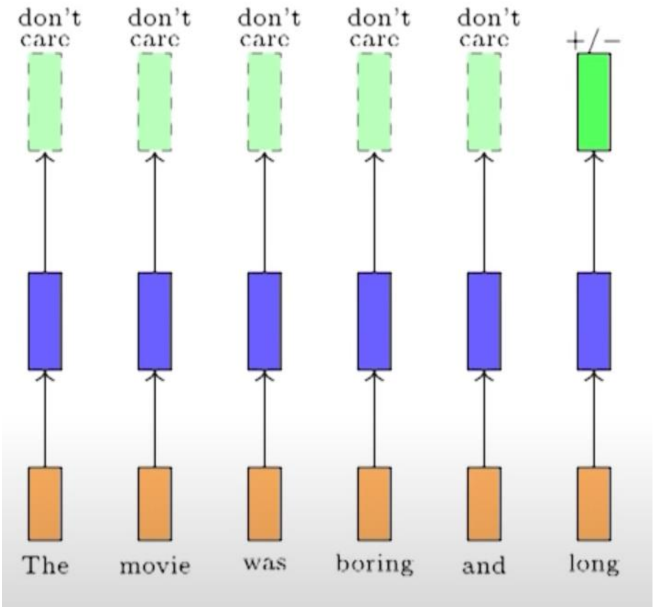
- These are known as sequence learning problems.
- We need to look at a sequence of (dependent) inputs and produce an output (or outputs).
- Each input corresponds to one time step.



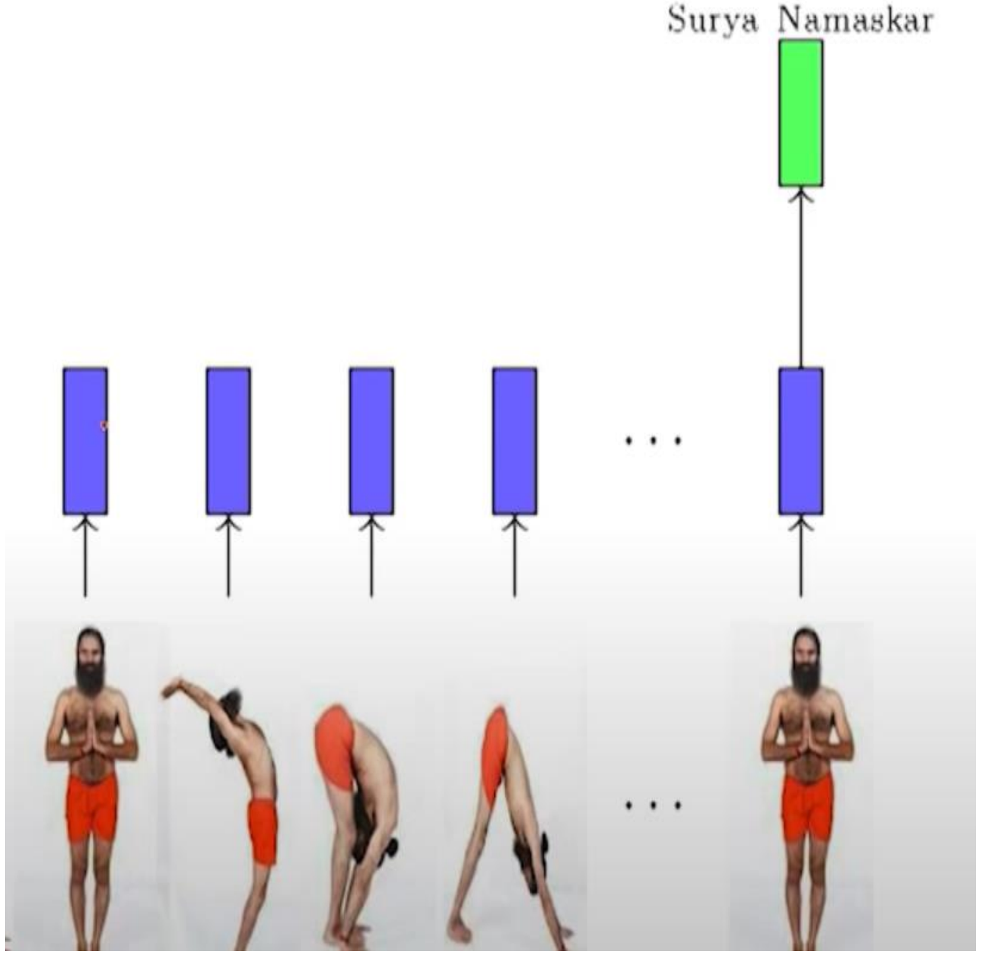


- Let us consider another example.
- Consider the task of predicting the part of speech tag (noun, adverb, adjective verb) of each word in a sentence.
- Once we see an adjective (social) we are almost sure that the next word should be a noun (man).
- Thus the current output (noun) depends on the current input as well as the previous input.
- Further the size of the input is not fixed (sentences could have arbitrary number of words)
- Notice that here we are interested in producing an output at each time step.
- Each network is performing the same task (input: word , output: tag).





- Sometimes we may not be interested in producing an output at every stage.
- Instead, we would look at the full sequence and then produce an output.
- For example, consider the task of predicting the polarity of a movie review.
- The prediction clearly doesn't depend only on the last word but also on some words which appear before.
- Here again we could think that the network is performing the same task at each step (input : word, output: +/-) but its just that we don't care about intermediate outputs.



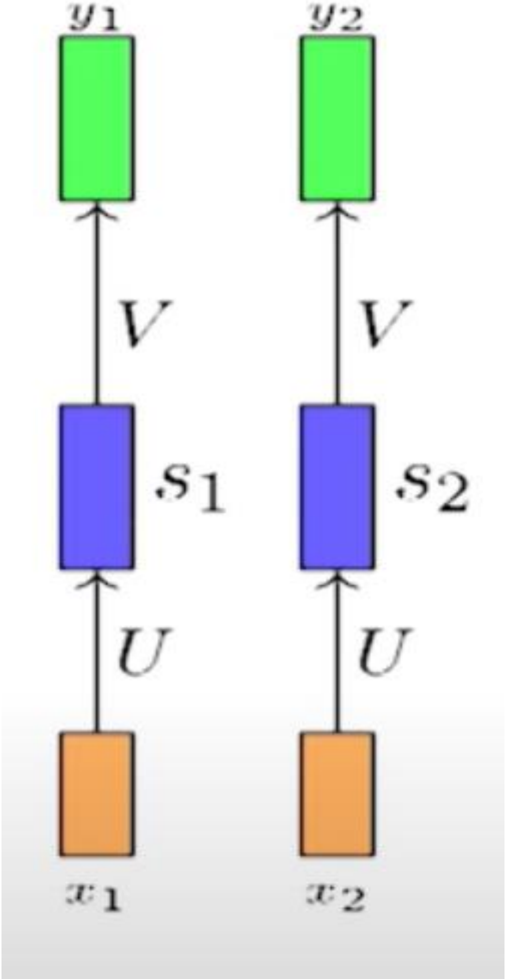
- Sequences could be composed of anything (not just words)
- For example, a video could be treated as a sequence of images.

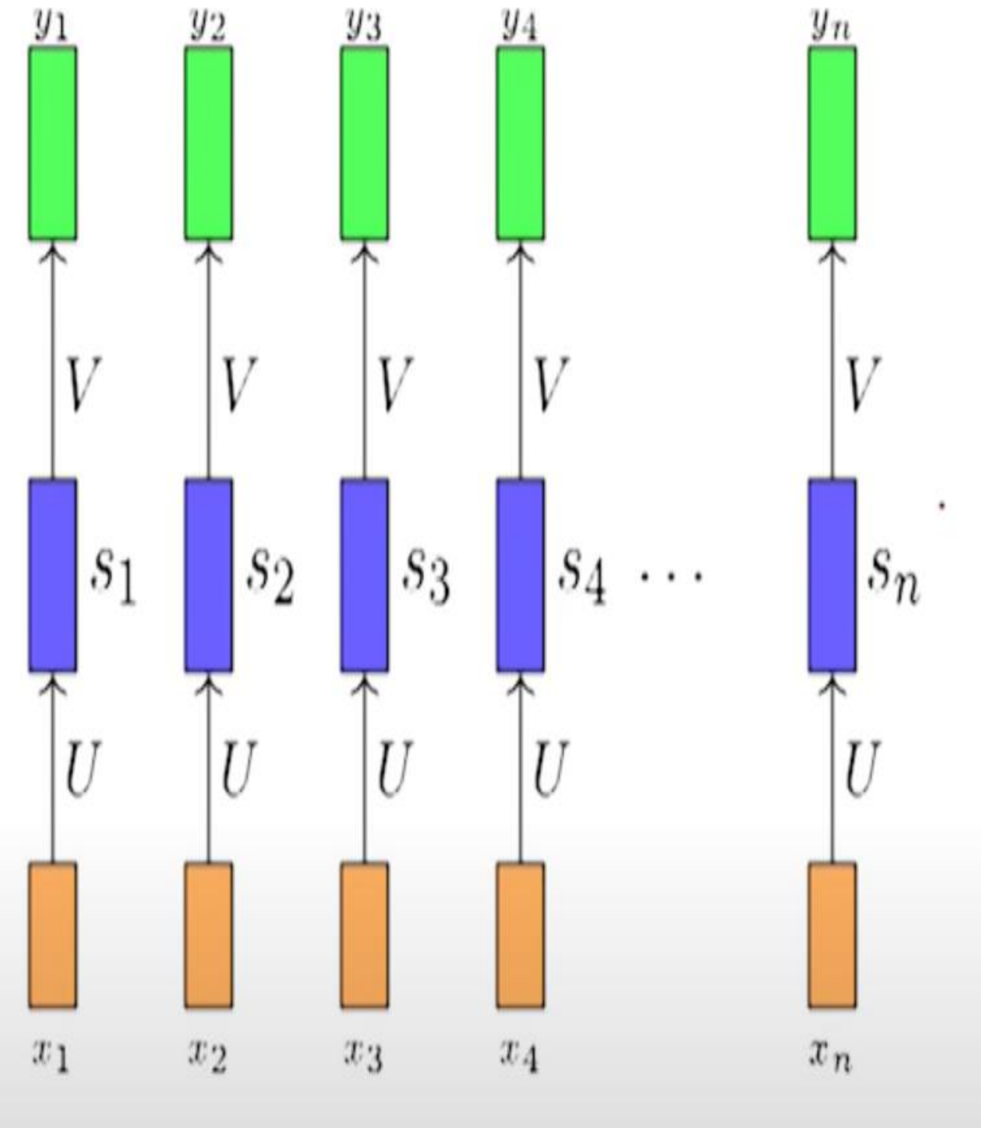
How to model these sequence learning problems?

- We look at something known as RNN : Recurrent Neural Networks
- Let's look at the function that is being executed at each time step:

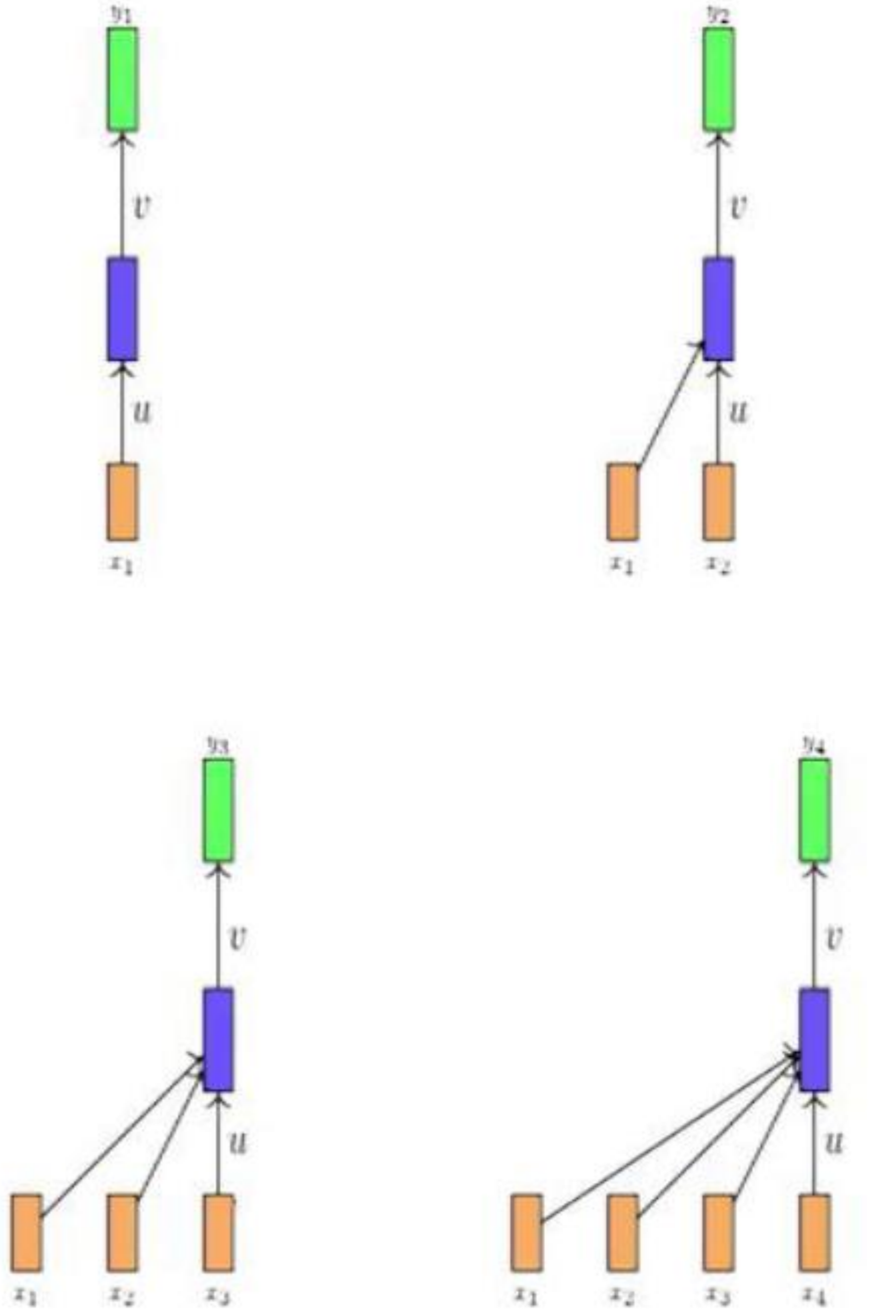
$$s_i = \sigma(Ux_i + b)$$
$$y_i = \sigma(Vs_i + c)$$
$$i = \text{timestep}$$

- Since we want the same function to be executed at each timestep, we should share the same network (i.e., same parameters at each timestep)





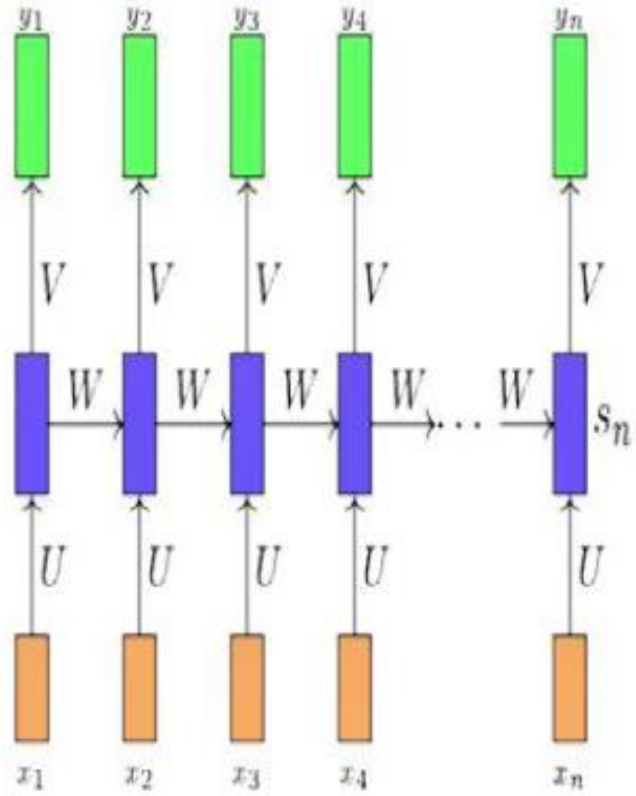
- This parameter sharing also ensures that the network becomes agnostic to the length (size) of the input.
- Since we are simply going to compute the same function (with same parameters) at each timestep, the number of timesteps doesn't matter).
- We just create multiple copies of the network and execute them at each timestep.

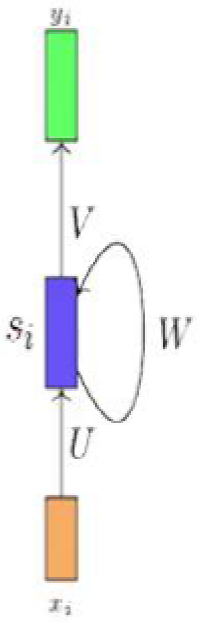


- The function being computed at each time-step is as :

$$
\begin{aligned}
y_1 &= f_1(x_1) \\
y_2 &= f_2(x_1, x_2) \\
y_3 &= f_3(x_1, x_2, x_3)
\end{aligned}
$$

- The network is sensitive to the length of the sequence.
- For example, a sequence of length 10 will require 10 functions (f) whereas a sequence of length 100 will require 100 functions (f).

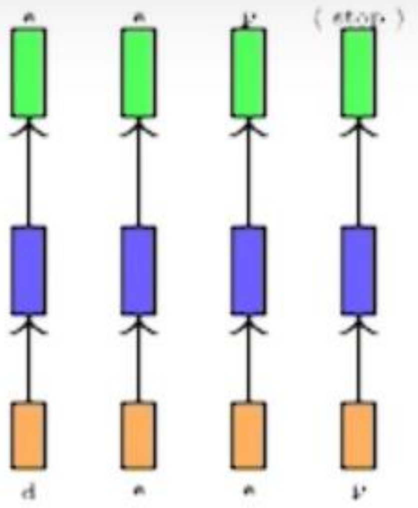


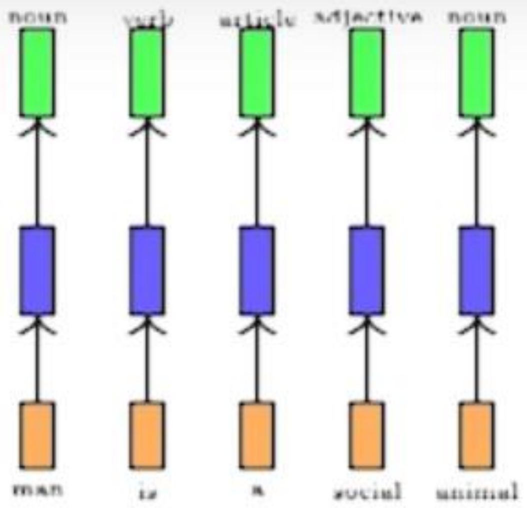


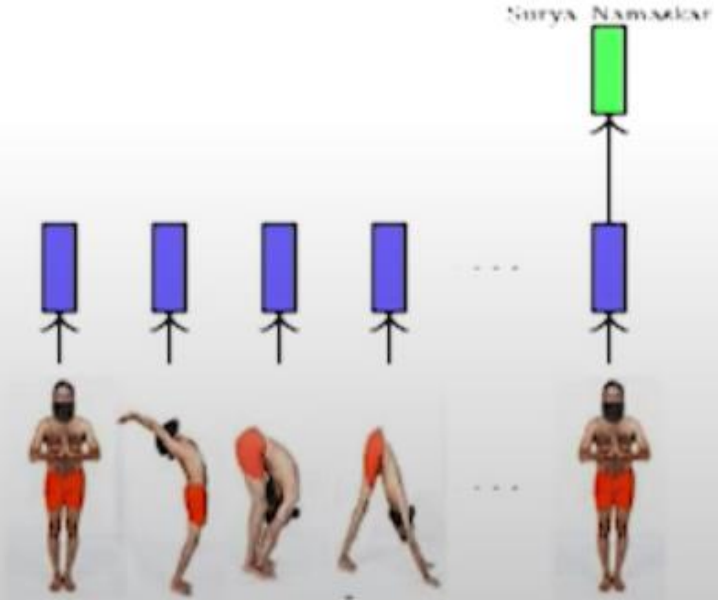
- The solution is to add a recurrent connection in the network.

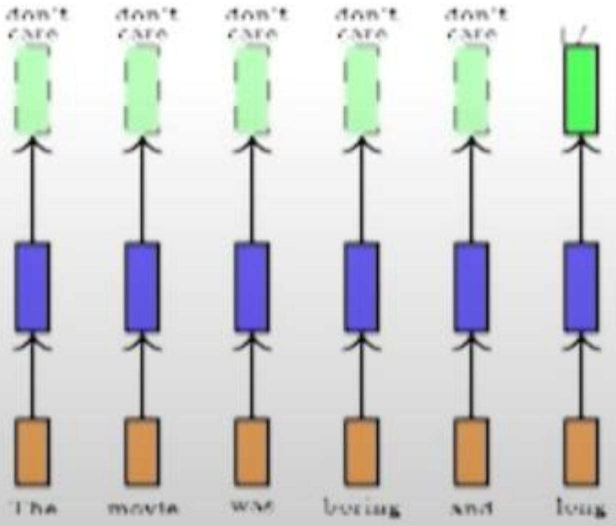
$$s_i = \sigma(Ux + Ws_{i-1} + b)$$
$$y_i = \sigma(Vs_i + c)$$
$$or$$
$$y_i = f(x_i, s_i, W, U, V)$$

- S_i is the state of the network at timestep i.
- The parameters W,U,V, b which are shared across timesteps.
- The same network (and parameters) can be used to compute y1, y2, …y100.


e
e
p
⟨stop⟩
d
e
e
p


noun
verb
article
adjective
noun
man
is
a
social
animal


Surya Namaskar


don't care
don't care
don't care
don't care
don't care
The
movie
was
boring
and
long

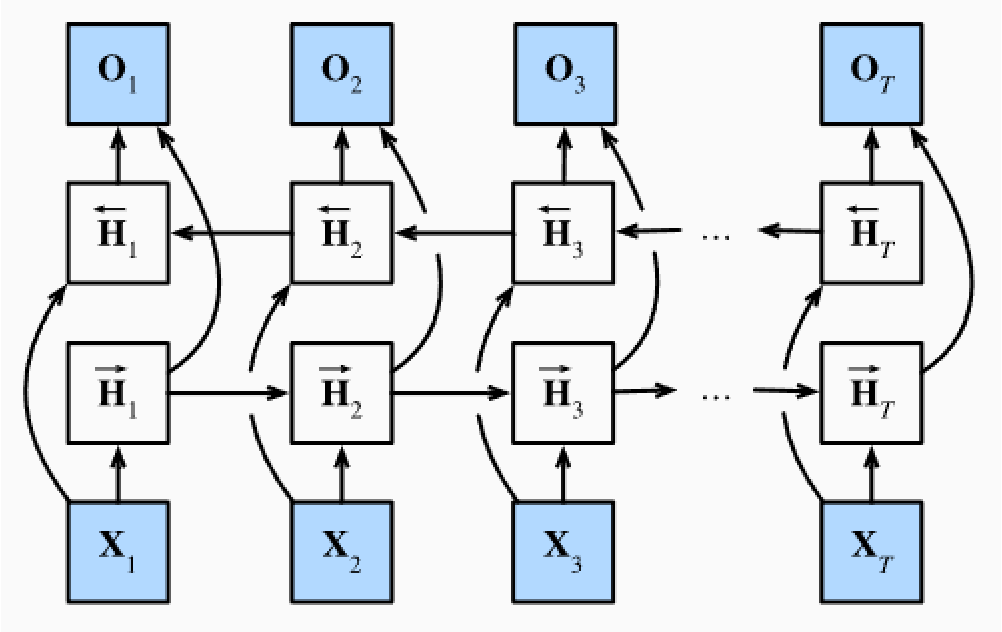
Bidirectional Recurrent Neural Networks

- In sequence learning, so far we assumed that our goal is to model the next output given what we have seen so far, e.g., in the context of a time series or in the context of a language model.
- While this is a typical scenario, it is not the only one we might encounter.
- To illustrate the issue, consider the following three tasks of filling in the blank in a text sequence:

- I am ___.
- I am ___ hungry.
- I am ___ hungry, and I can eat half a cake.

- I am ___.
- I am ___ hungry.
- I am ___ hungry, and I can eat half a cake.
- Depending on the amount of information available, we might fill in the blanks with very different words such as “happy”, “not”, and “very”.
- Clearly the end of the phrase (if available) conveys significant information about which word to pick.

- If we want to have a mechanism in RNNs that offers comparable look-ahead ability, we need to modify the RNN design that we have seen so far.
- Instead of running an RNN only in the forward mode starting from the first token, we start another one from the last token running from back to front.
- *Bidirectional RNNs* add a hidden layer that passes information in a backward direction to more flexibly process such information.



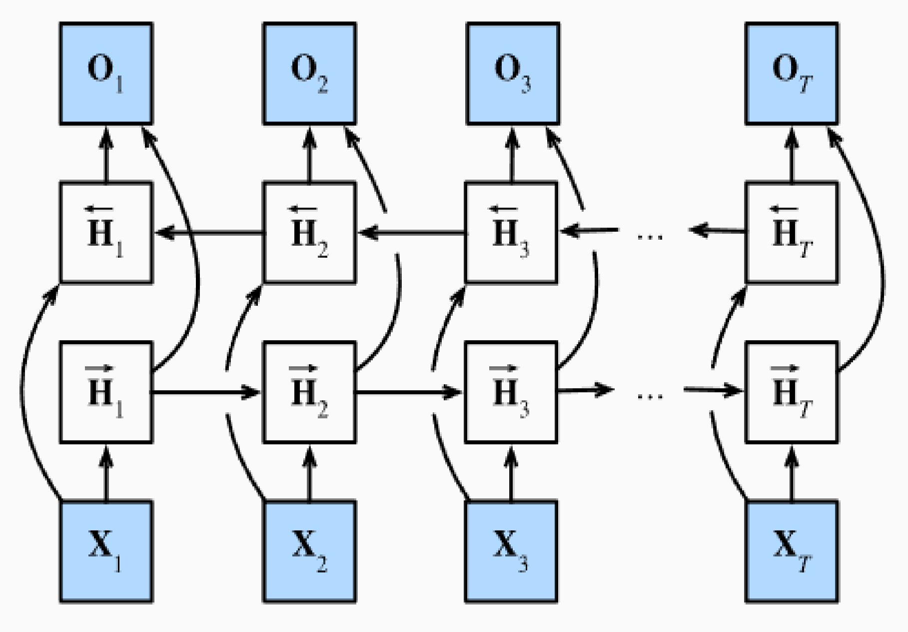
Architecture of a bidirectional RNN.

- For any time step t, given a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples: n, number of inputs in each example: d) and let the hidden layer activation function be φ.
- In the bidirectional architecture, we assume that the forward and backward hidden states for this time step are $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$, respectively, where h is the number of hidden units.
- The forward and backward hidden state updates are as follows:

$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),$$
$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),$$

- where the weights $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d\times h}, \mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h\times h}, \mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d\times h}$

and biases $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1\times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1\times h}$ are all the model parameters.



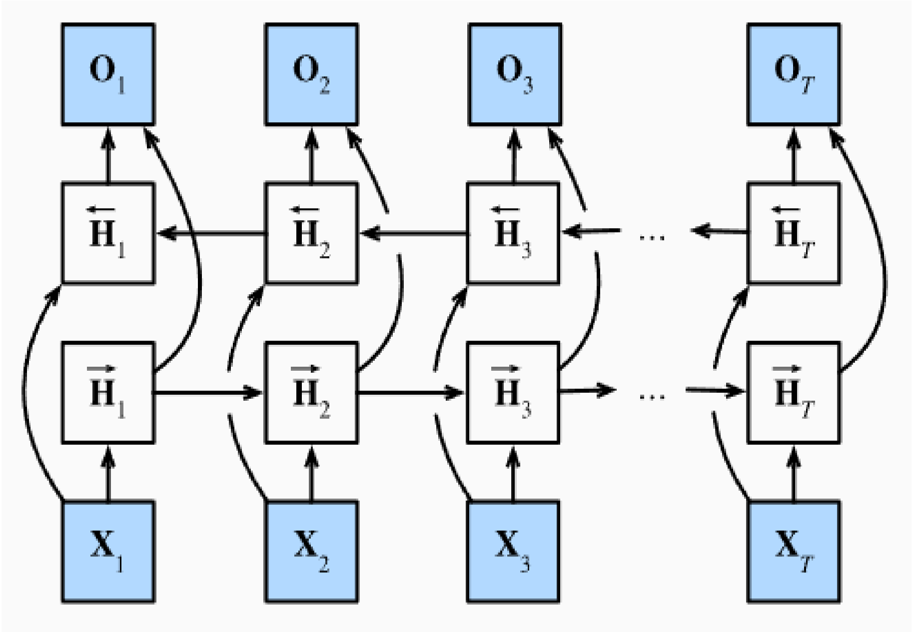
Architecture of a bidirectional RNN.

- Next, we concatenate the forward and backward hidden states $\overrightarrow{\mathbf{H}}_t$ and $\overleftarrow{\mathbf{H}}_t$ to obtain the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ to be fed into the output layer.
- In deep bidirectional RNNs with multiple hidden layers, such information is passed on as input to the next bidirectional layer.
- Last, the output layer computes the output (number of outputs: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

- Here, the weight matrix $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$

and the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer.



Architecture of a bidirectional RNN.

Deep RNN

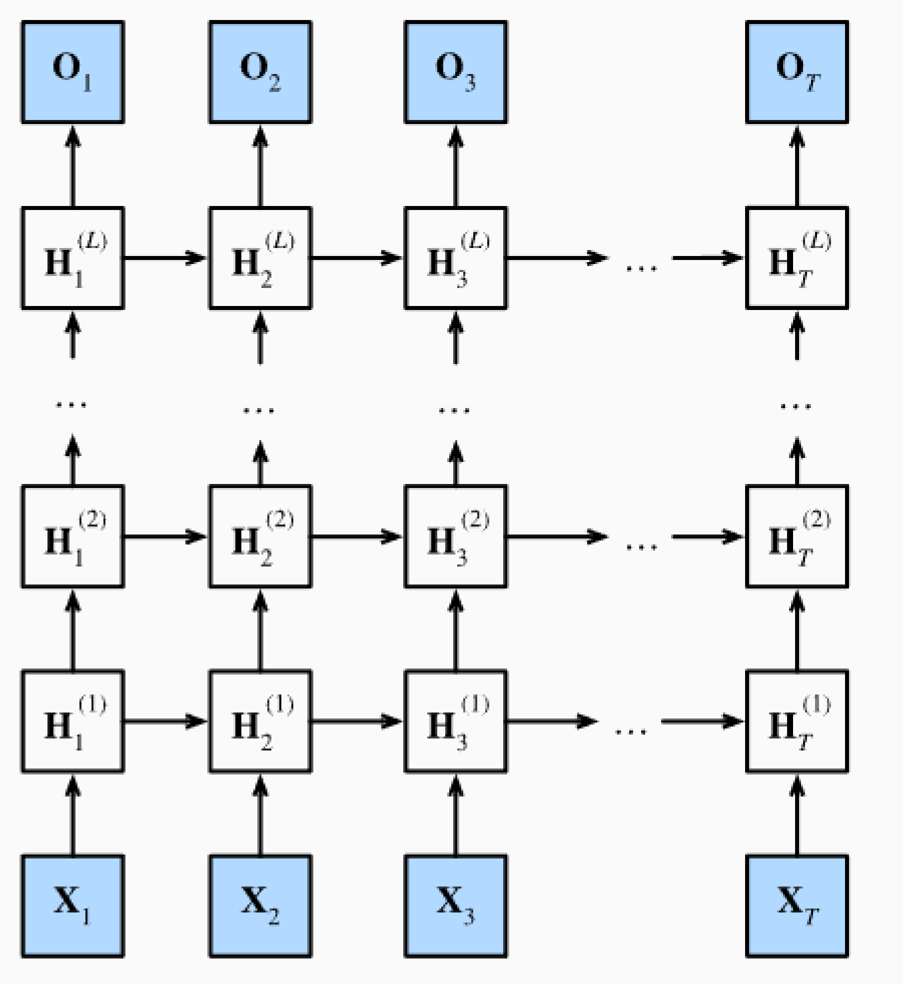


Fig : Architecture of a deep RNN.

Deep RNN Framework is a RNN framework for high-dimensional sequence problems like :

- Video classification
- Video future prediction

The figure shows a deep RNN with L hidden layers.
We could stack multiple layers of RNNs on top of each other.

This results in a flexible mechanism, due to the combination of several simple layers.

In particular, data might be relevant at different levels of the stack.

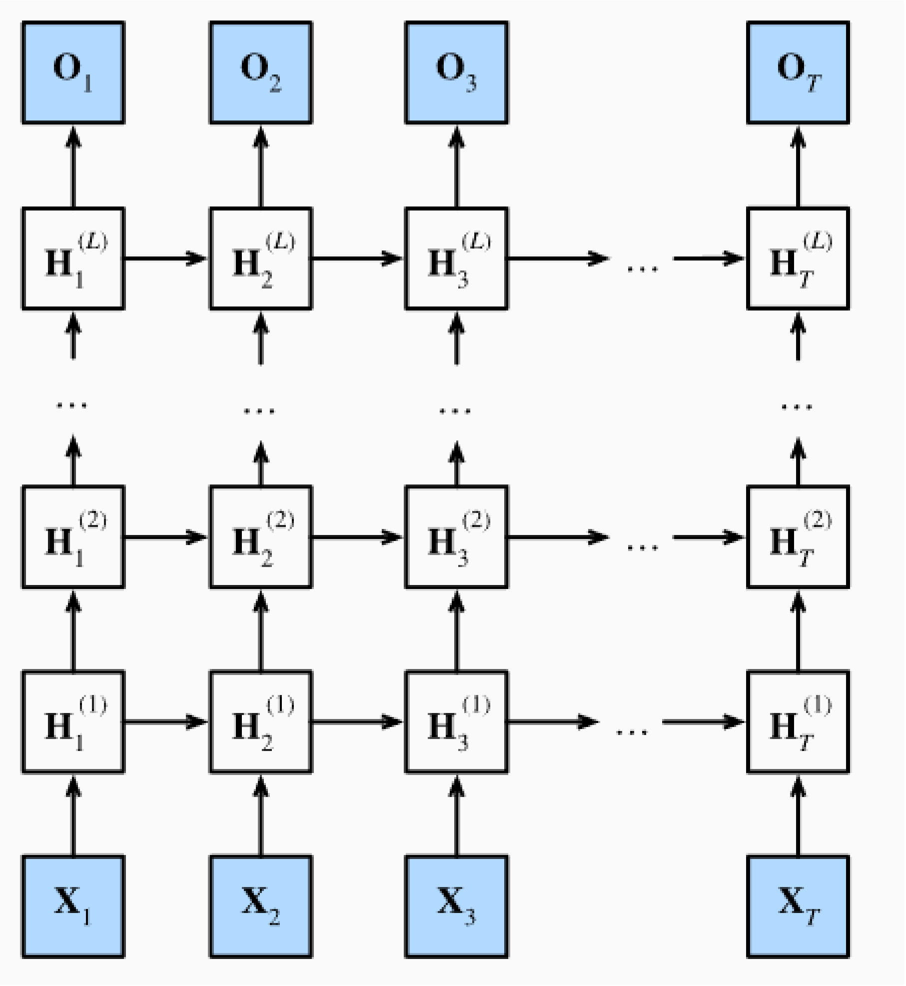
Each hidden state is continuously passed to both the next time step of the current layer and the current time step of the next layer.

Functional Dependencies

- Suppose that we have a minibatch input $X_t \in R^{n \times d}$ (number of examples: n , number of inputs in each example: d) at time step t .
- At the same time step, let the hidden state of the l^{th} hidden layer (l=1,…,L) be $H^{(l)}_t \in R^{n \times h}$ (number of hidden units: h) and the output layer variable be $O_t \in R^{n \times q}$ (number of outputs: q).
- Setting $H^{(0)}_t = X_t$, the hidden state of the lth hidden layer that uses the activation function φ_l is expressed as follows:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

- where the weights $W^{(l)}_{xh} \in R_{h \times h}$ and $W^{(l)}_{hh} \in R_{h \times h}$, together with the bias b(l)h∈R1×h , are the model parameters of the lth hidden layer.



- In the end, the calculation of the output layer is only based on the hidden state of the final L^{th} hidden layer:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{hq} + \mathbf{b}_q$$

- where the weight $W_{hq} \in R_{h\times q}$ and the bias $b_q \in R_{1\times q}$ are the model parameters of the output layer.

- Just as with MLPs, the number of hidden layers L and the number of hidden units h are hyperparameters. In other words, they can be tuned or specified by us.