3.1 Basics of Recurrent Neural Networks

- A class of neural networks for processing sequential data is known as recurrent neural networks, or RNNs.
- Recurrent neural networks are neural networks that are specialized for processing a series of values x(1),..., x (τ), just like convolutional networks are neural networks that are specialized for processing a grid of values X, such as an image.
- Recurrent networks can scale to far longer sequences than would be possible for networks without sequence-based specialization, much as convolutional networks can easily scale to pictures with vast width and height and certain convolutional networks can handle images of varied size.
- Sequences of different lengths may generally be processed by recurrent networks.
- We need to use one of the early concepts from machine learning and statistical models from the 1980s: sharing parameters across various regions of a model—to transition from multilayer networks to recurrent networks.
- The model may be expanded and used to instances of other forms (different lengths in this case) and generalized across them, thanks to parameter sharing.
- We could not share statistical strength across different sequence lengths and across various places in time if we had distinct parameters for each value of the time index or generalize to sequence lengths not encountered during training.
- When a given piece of information might occur numerous times during the sequence, such sharing is very crucial.
- Take the phrases "I travelled to Nepal in 2009" and "In 2009, I went to Nepal," for instance. We want the year 2009 to be recognized as the pertinent piece of information, whether it comes in the sixth word or the second of the phrase, if we ask a machine learning model to scan each line and extract the year in which the narrator travelled to Nepal.
- Let's say we developed a feedforward network that analyses texts of a specific length. A conventional fully linked feedforward network would need to learn each language rule independently for each point in the sentence since it would have unique parameters for each input characteristic. A recurrent neural network, in contrast, uses the same weights over a number of time steps.
- Convolution across a 1-D temporal sequence is a similar concept. Time-delay neural networks are built using this convolutional method. Although shallow, the

convolution technique enables a network to communicate parameters across time. A sequence is produced via convolution, and each element of the sequence is a function of a few nearby input elements. The use of the same convolution kernel at each time step is how the concept of parameter sharing is put into practice. Differently from other networks, recurrent ones exchange parameters. Each component of the output is a function of the components that came before it. The same updating rule that was used to create the previous outputs is used to construct each component of the output. With this recurrent approach, parameters are shared via an extremely complex computational graph.

- RNNs are said to operate on a sequence that has vectors $x^{(t)}$ with a time step index t ranging from 1 to τ for the sake of clarity. Recurrent networks often function with minibatch sizes of these sequences, each of which has a unique sequence length. To make the notation simpler, the minibatch indices have been removed. Furthermore, the time step index need not correspond to actual time passing in the real world. At times, it just relates to the place in the sequence. RNNs may be used in two dimensions spanning spatial data, such as photographs, and even when applied to time-related data, the network may contain connections that reach back in time, given that the complete sequence has been viewed before it is given to the network.

3.1.1 Unfolding Computational Graphs

- The structure of a number of calculations, such as those involved in mapping inputs and parameters to outputs and loss, can be formalized using a computational graph. The concept of unfolding a recursive or recurrent computation into a computational network with a repeated structure, often corresponding to a series of occurrences, is explained in this section. The sharing of parameters across a deep network structure is the outcome of unfolding this graph.
- Take the traditional form of a dynamical system, for instance :

$$s^{(t)} = f(s^{(t-1)}; \theta) \qquad \text{...(3.1.1)}$$

where s(t)is called the state of the system.

- Because the definition of s at time t goes back to the identical definition at time t – 1, equation (3.1.1) is recurring.

- The graph can be unfolded for a limited number of time steps τ by using the definition $\tau - 1$ times. For example, if we unfold equation (3.1.1) for $\tau = 3$ time steps, we obtain

$$s^{(3)} = f(s^{(2)}; \theta) \quad \ldots(3.1.2)$$

$$= f(f(s^{(1)}; \theta); \theta) \quad \ldots(3.1.3)$$

- By continually using the definition in this manner to unfold the equation, an expression that does not involve recurrence has been produced. In the present, a conventional directed acyclic computational network can represent such an expression. Fig. 3.1.1 shows the unfolded computational graph of equations 3.1.1 and 3.1.3.

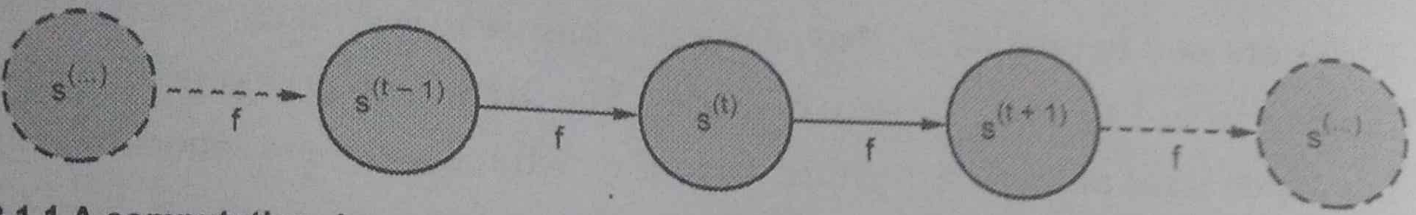


Fig. 3.1.1 A computational graph that has been unfurled serves as an illustration of the classical dynamical system given by Eq. 3.1.1

- The state at each node at time t is represented and the function f translates the state at time t to the state at time t + 1. For each time step, the same parameters (i.e., the same value of used to parameterize f) are applied.
- As another example, let us consider a dynamical system driven by an external signal $x^{(t)}$,

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta) \quad \ldots(3.1.4)$$

where we see that the state now contains information about the whole past sequence.

- There are several methods for constructing recurrent neural networks. Any function that involves recurrence may be seen as a recurrent neural network, much like practically any function can be regarded as a feedforward neural network.
- Equation (3.1.5) or a related equation is frequently used by recurrent neural networks to specify the values of its hidden units. We now rewrite equation (3.1.4) using the variable h as the state to show that the state is the network's hidden units:

$$h^{(t)} = fh(h^{(t-1)}, x^{(t)}; \theta) \quad \ldots(3.1.5)$$

- Typical RNNs will include additional architectural features, like output layers that read data from the state h to make predictions, as shown in Fig. 3.1.2.

- The recurrent network often learns to utilize $h^{(t)}$ as a type of lossy summary of the task-relevant elements of the previous sequence of inputs up to t when it is trained to execute a task that involves forecasting the future from the past. Since it converts an arbitrary length sequence $(x^{(t)}, x^{(t1)}, x^{(t2)},..., x^{(2)}, x^{(1)})$ to a fixed length vector h, this summary is inherently lossy. This summary may retain some former sequence elements with greater precision than others depending on the training criterion. For instance, it might not be necessary to store all of the data in the input sequence up to time t, only enough to predict the rest of the sentence if the RNN is used in statistical language modelling, which typically predicts the next word given previous words.
- The circumstance when we require $h^{(t)}$ to be rich enough to allow one to roughly recover the input sequence, like in autoencoder systems, is the most challenging.

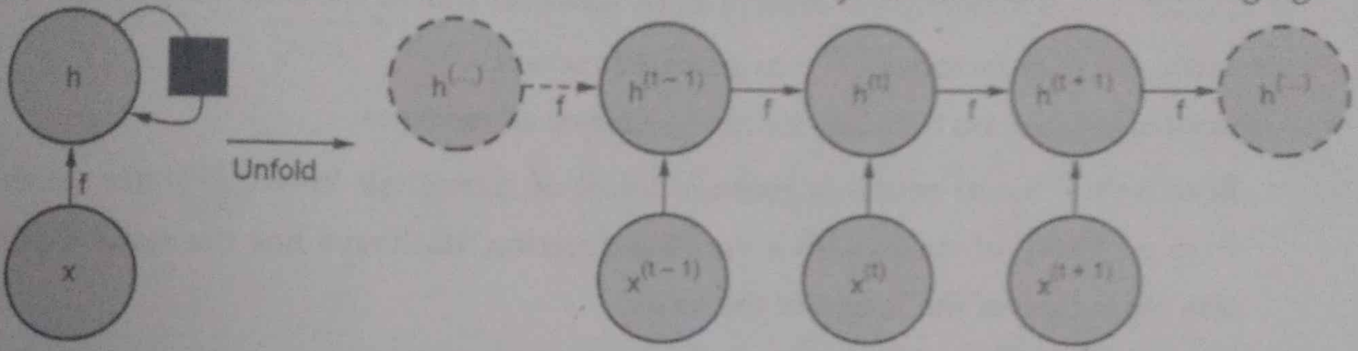


Fig. 3.1.2 An output less recurrent network

- Simply by combining information from the input x into the state h that is transmitted forward over time, this recurrent network processes information from the input x. Circuit schematic (left). A onetime step delay is shown by the black square. (Right) The same network as a computational graph that has been unfolded, where each node is now connected to a specific time occurrence.
- There are two possible methods to draw equation (3.1.5) A diagram with one node for each element that may be present in a real-world application of the model, like a biological neural network, is one approach to represent an RNN. In this approach, the network establishes a real-time circuit made up of physical components, as shown on the left of Fig. 3.1.2, whose present condition might affect their future state.
- In each circuit diagram in this chapter, a black square denotes an interaction that occurs one time step later, from the state at time t to the state at time t + 1. The RNN may also be represented as an unfolding computational graph, where each component is represented by a variety of distinct variables, one variable per time

step, each indicating the component's state at that instant in time. As seen in the right of Fig. 3.1.2, each variable for each time step is represented as a distinct node of the computational graph. The technique that converts a circuit, shown on the left side of the picture, into a computational graph with repeated elements, shown on the right side, is what we refer to as unfolding. The size of the unfolded graph now varies with the length of the series.

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

$$h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)}) \quad \ldots(3.1.6)$$

$$= f(h^{(t-1)}, x^{(t)}; \theta) \quad \ldots(3.1.7)$$

- The function $g^{(t)}$ applies a function f repeatedly to the entire past sequence $(x^{(t)}, x^{(t1)}, x^{(t2)}, \ldots, x^{(2)}, x^{(1)})$ to obtain the present state, however due to the unfolding recurrent structure, we may factorize $g^{(t)}$ into a single function.
- The unfolding process thus introduces two major advantages:
  - Because the learnt model is given in terms of transitions between states rather than a history of states with a variable duration, it always has the same input size regardless of the length of the series.
  - Every time step can employ the same transition function f with the same inputs.
- Instead of having to train a different model $g^{(t)}$ for each potential time step, these two components allow us to learn a single model f that works on all time steps and all sequence lengths. A single, shared model may be learned, enabling generalization to sequence lengths not included in the training set and allowing the model to be estimated with a much less number of training samples than would otherwise be necessary.
- There are applications for both the recurrent graph and the unrolled graph. Recurrent graph is clear and concise. The unfolded graph gives a clear explanation of the computations that need be run. By explicitly displaying the path along which this information travels, the unfolded graph also contributes to the illustration of the notion of information flow both forward in time (computing outputs and losses) and backward in time (computing gradients).

3.2 Recurrent Neural Networks

- With the concepts of parameter sharing and graph unrolling from the previous section, we may create a wide range of recurrent neural networks.

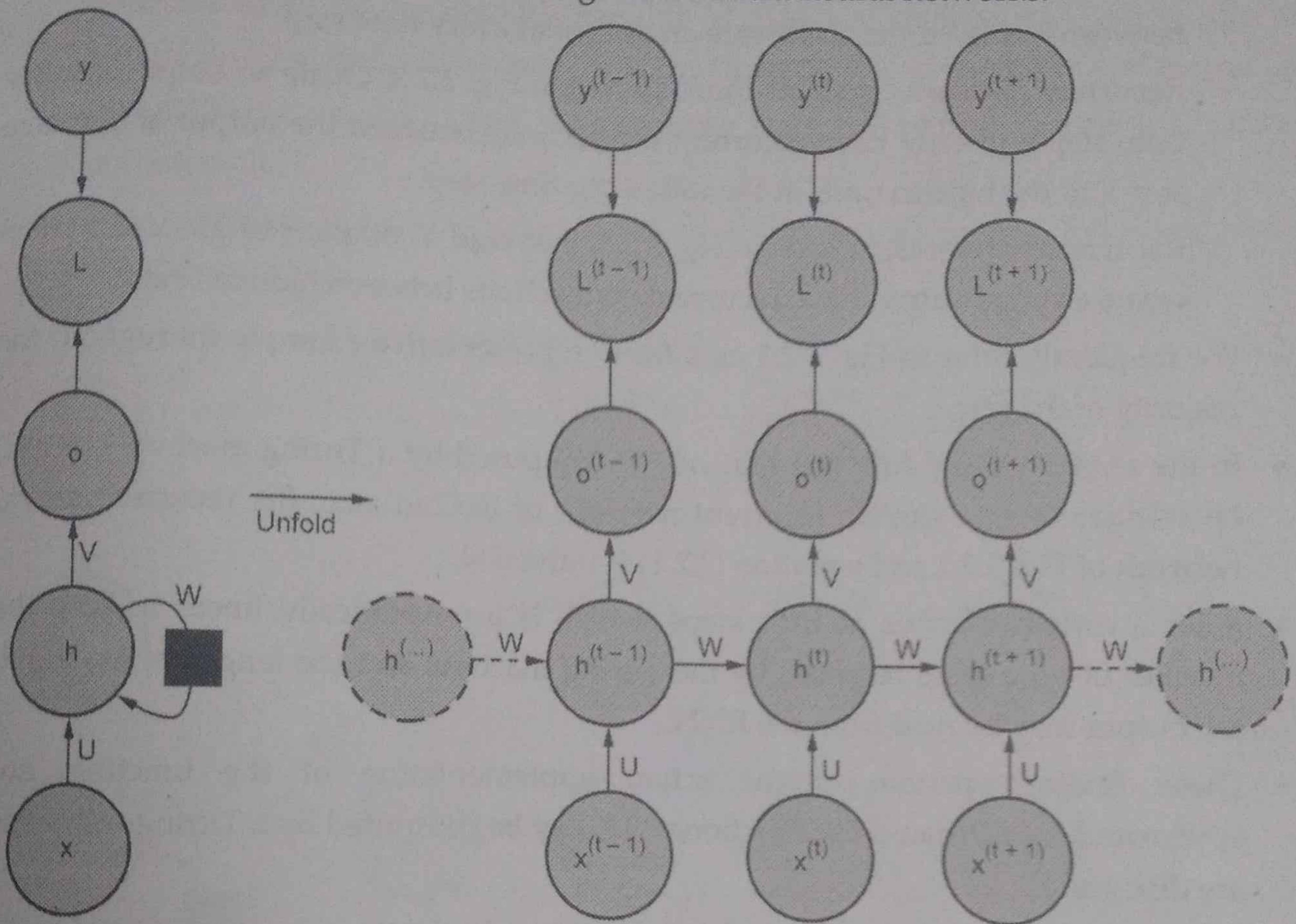


Fig. 3.2.1

- Fig. 3.2.1 The graph used to calculate a recurrent network's training loss, which converts a sequence of input x values into a sequence of output o values. Each o's distance from the matching training objective y is indicated by a loss L. We assume that o is the unnormalized log probabilities when utilizing softmax outputs. Internally, the loss L calculates $\hat{y}$ = softmax(o) and contrasts it with the desired y. The RNN has hidden-to-hidden recurrent connections, hidden-to-hidden connections, and hidden-to-output connections, all of which are parameterized by weight matrices U, W, and V, respectively. In this paradigm, forward propagation is defined by equation (3.2.1). (Left) Recurrent connections are used to draw the RNN and its loss. (Right) The same is shown as a computational network that has been time-unfolded, where each node is now connected to a specific time occurrence.

- The following are some instances of significant design patterns for recurrent neural networks :
  - Recurrent networks, as seen in Fig. 3.2.1, that feature recurrent connections between hidden units and create an output at every time step.
  - Recurrent networks, such as those shown in Fig. 3.2.2, create an output at every time step and only have recurrent connections between the output at one time step and the hidden units at the following time step.
  - Recurrent networks, as seen in Fig. 3.2.3, that read a complete sequence and then create a single output have recurrent connections between hidden units.
- We frequently refer to Fig. 3.2.1 as a fairly representative example throughout the majority of the text.
- In the sense that any function that can be computed by a Turing machine can also be calculated by a similar recurrent network of limited size, the recurrent neural network of Fig. 3.2.1 and equation (3.2.1) is universal.
- After a certain number of time steps, which is asymptotically linear in both the number of time steps required by the Turing machine and the length of the input, the output may be read from the RNN.
- These findings pertain to the actual implementation of the function, not approximations, because the functions that may be computed by a Turing computer are discrete.
- When utilized as a Turing machine, the RNN requires discretization of its outputs in order to produce a binary output from an input binary sequence.
- Using a single unique RNN of limited size, it is feasible to calculate all functions in this environment.
- The Turing machine's "input" is a description of the function that has to be calculated, hence the same network that replicates this Turing machine is enough for all issues.
- By expressing its activations and weights with rational values of unlimited precision, the theoretical RNN utilized for the proof may imitate an unbounded stack.
- The forward propagation equations for the RNN shown in Fig. 3.2.1 are now developed. The choice of activation function for the concealed units is not indicated in the illustration. Here, the activation function of the hyperbolic tangent is assumed. Additionally, the output and loss functions' actual shapes are not

described in the image. Since the RNN is being used to predict words or characters, we'll assume that the output is discrete in this case.

- Regarding the output o as providing the unnormalized log probabilities of each potential value of the discrete variable is a logical method to describe discrete variables. After that, we can use the softmax method to get a vector $\hat{y}$ of normalized probabilities over the output. The starting state $h^{(0)}$ is first specified in forward propagation.
- Then, for each time step from t = 1 to t = ?, we apply the following update equations :

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \qquad \text{...(3.2.1)}$$

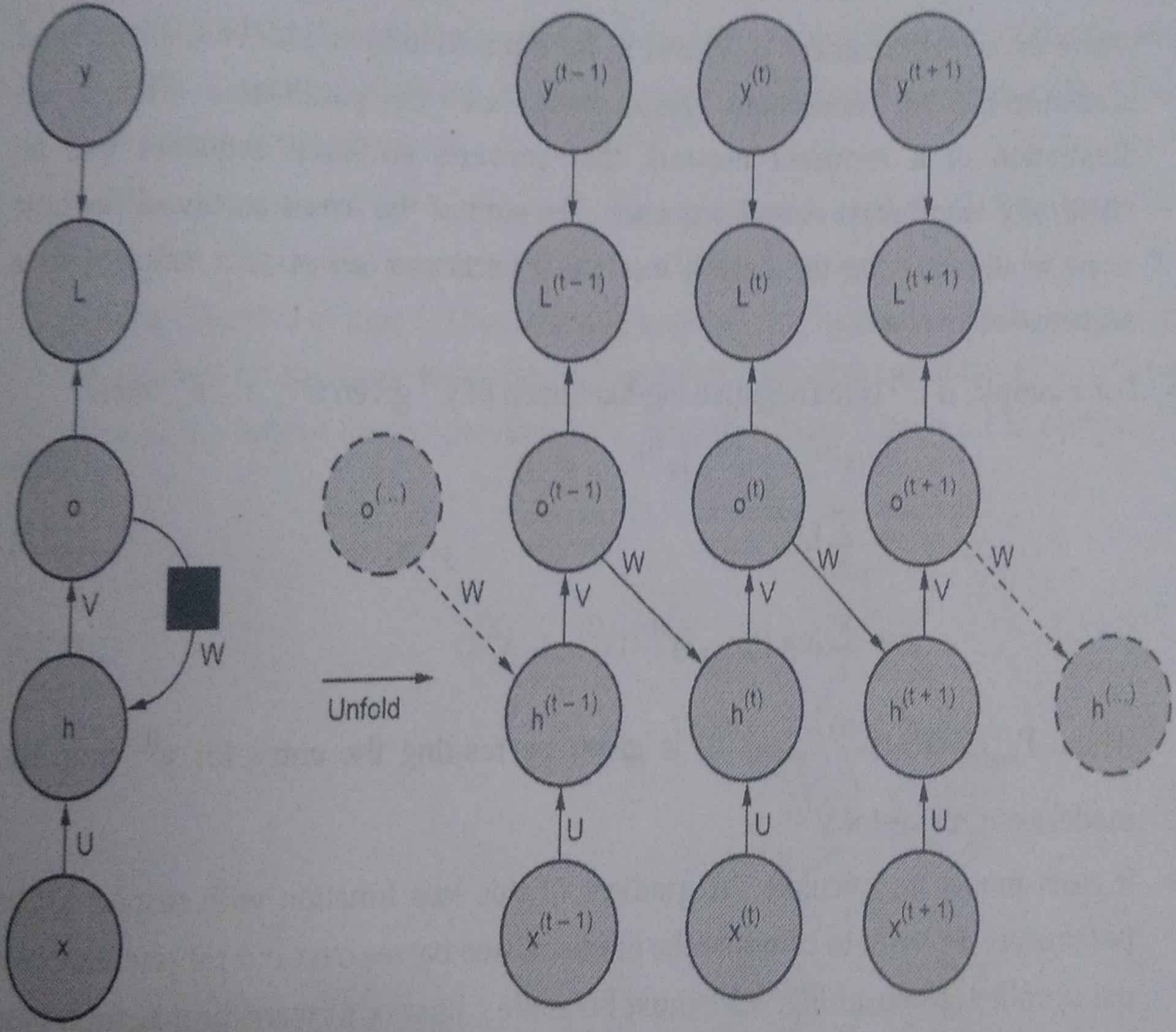


Fig. 3.2.2

- Fig. 3.2.2 An RNN in which the feedback link from the output to the hidden layer is the sole repetition. Input is x t, hidden layer activations are $h^{(t)}$, outputs are $o^{(t)}$, targets are $y^{(t)}$, and loss is $L^{(t)}$ at each time step t. (Left) circuit schematic. (Right) computed graph that has been unfolded. Such an RNN is less capable than those in the family depicted in Fig. 3.2.1 (can express a narrower number of functions). The

RNN in Fig. 3.2.1 is free to pick what data it wishes to broadcast from the past to the future as part of its hidden representation h. Only indirectly, by the predictions it generated, is the past related to the present. Unless o is extremely high-dimensional and rich, it typically lacks crucial historical data. As a result, the RNN in this figure is less effective, but it might be simpler to train because each time step can be trained independently of the others, allowing for more parallel training as explained in the following section.

$$h^{(t)} = \tanh(a^{(t)}) \quad \text{...(3.2.2)}$$

$$o^{(t)} = c + Vh^{(t)} \quad \text{...(3.2.3)}$$

$$\hat{y}^{(t)} = \text{softmax}\,(o^{(t)}) \quad \text{...(3.2.4)}$$

where the weight matrices U, V, and W for input-to-hidden, hidden-to-output and hidden-to-hidden connections, respectively, are the parameters. This is an illustration of a recurrent network that converts an input sequence into an identically lengthened output sequence. The sum of the losses across all the time steps would thus be the overall loss for a particular series of x values and a sequence of y values.

- For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(t)}$, then

$$L(\{x^{(1)}, \ldots, x^{(\tau)}\}, \{y^{(1)}, \ldots, y^{(\tau)}\}) \quad \text{...(3.2.5)}$$

$$= \sum_t L^{(t)} \quad \text{...(3.2.6)}$$

$$= \sum_t \log P_{\text{model}}\,(y^{(t)} \mid \{x^{(1)}, \ldots, x^{(t)}\}) \quad \text{...(3.2.7)}$$

where $P_{\text{model}}\,(y^{(t)} \mid \{x^{(1)}, \ldots, x^{(t)}\})$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{y}^{(t)}$.

- It costs money to calculate the gradient of this loss function with respect to the parameters. In order to compute the gradient, two passes over our representation of the unrolled graph in Fig. 3.2.1 must be made : First, a forward propagation pass from left to right and then a backward propagation pass from right to left. Because the forward propagation graph is intrinsically sequential and each time step can only be computed after the preceding one, the runtime is $O(\tau)$ and cannot be decreased by parallelization.
- The memory cost is also $O(\tau)$, as calculated states in the forward pass must be maintained until they are utilized in the backward round. Back-propagation through time, or BPTT, is an $O(\tau)$ cost back-propagation method that is used to

unroll a graph; it is further detailed in Sec. 3.2.2. As a result, although incredibly strong, the network with recurrence between hidden units is also costly to train. Is there a substitute ?

Teacher Forcing and Networks with Output Recurrence

- The network with recurrent connections only from the output at one time step to the hidden units at the next time step is strictly less powerful because it lacks hidden-to-hidden recurrent connections. It requires that the output units capture all of the information about the past the network will use to predict the future.
- The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled.
- Training can thus be parallelized, with the gradient for each step t computed in isolation. There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.

Models that have recurrent connections from their outputs leading back into the model may be trained with *teacher forcing*. Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input time $t+1$. We can see this by examining a sequence with two time steps. The conditional maximum likelihood criterion is

$$\log p\left(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) \tag{10.15}$$

$$= \log p\left(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) + \log p\left(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) \tag{10.16}$$

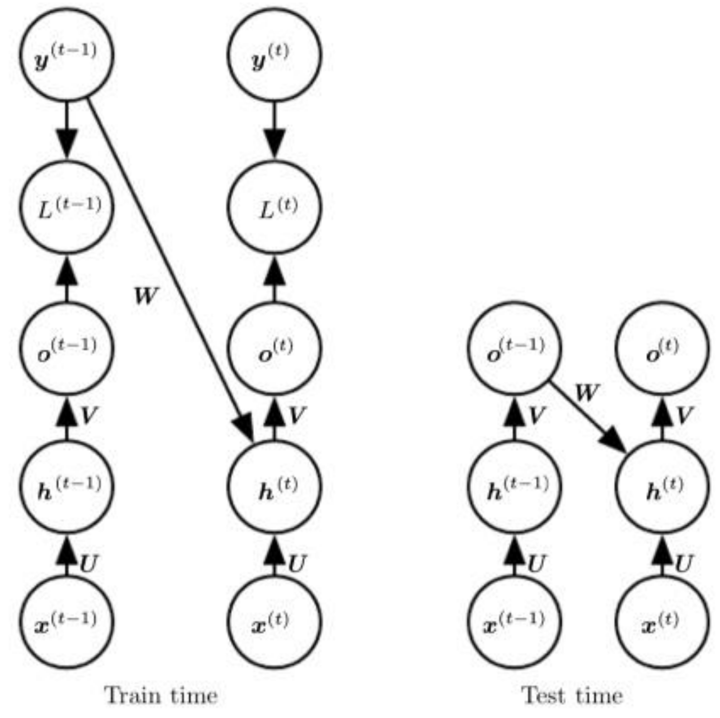


Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. *(Left)* At train time, we feed the *correct output* $\boldsymbol{y}^{(t)}$ drawn from the train set as input to $\boldsymbol{h}^{(t+1)}$. *(Right)* When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $\boldsymbol{y}^{(t)}$ with the model's output $\boldsymbol{o}^{(t)}$, and feed the output back into the model.

We originally motivated teacher forcing as allowing us to avoid back-propagation through time in models that lack hidden-to-hidden connections.

Teacher forcing may still be applied to models that have hidden-to-hidden connections so long as they have connections from the output at one time step to values computed in the next time step.

However, as soon as the hidden units become a function of earlier time steps, the BPTT algorithm is necessary. Some models may thus be trained with both teacher forcing and BPTT.

The disadvantage of strict teacher forcing arises if the network is going to be later used in an *open-loop* mode, with the network outputs fed back as input.

Computing the Gradient in a Recurrent Neural Network

To gain some intuition for how the BPTT algorithm behaves, we provide an example of how to compute gradients by BPTT for the RNN equations above. The nodes of our computational graph include the parameters U, V, W, b, and c as well as the sequence of nodes indexed by t for $x^{(t)}$, $h^{(t)}$, $o^{(t)}$ and $L^{(t)}$.

We start the recursion with the nodes immediately preceding the final loss

$$\frac{\partial L}{\partial L^{(t)}} = 1. \tag{10.17}$$

In this derivation we assume that the outputs $o^{(t)}$ are used at the argument to the softmax function to obtain the vector $\hat{y}$ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target $y^{(t)}$ given the input so far.

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}. \tag{10.18}$$

We work our way backwards, starting from the end of the sequence. At the final time step τ , $h^{(\tau)}$ only has $o^{(\tau)}$ as a descendent, so its gradient is simple:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L. \tag{10.19}$$

We can then iterate backwards in time to back-propagate gradients through time, from $t = \tau - 1$ down to $t = 1$, noting that $h^{(t)}$ (for $t < \tau$) has a descendents both $o^{(t)}$ and $h^{(t+1)}$. Its gradient is thus given by

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} \right)^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \right)^\top (\nabla_{\boldsymbol{o}^{(t)}} L) \tag{10.20}$$

$$= \boldsymbol{W}^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag} \left(1 - \left(\boldsymbol{h}^{(t+1)} \right)^2 \right) + \boldsymbol{V}^\top (\nabla_{\boldsymbol{o}^{(t)}} L) \tag{10.21}$$

where $\texttt{diag}(1 - (h^{(t+1)})^2)$ indicates the diagonal matrix containing the elements $1 - (h_i^{(t+1)})^2$. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t + 1$.

Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes. Because the parameters are shared across many time stamps, we must take some care when denoting calculus operations involving these variables.

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^\top \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L \tag{10.22}$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^\top \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L \tag{10.23}$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top} \tag{10.24}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \tag{10.25}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)\top} \tag{10.26}$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \tag{10.27}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)\top} \tag{10.28}$$

We do not need to compute the gradient with respect to $x^{(t)}$ for training because it does not have any parameters as ancestors in the computational graph defining the loss.

Recurrent Networks as Directed Graphical Models

As with a feedforward network, we usually wish to interpret the output of the RNN as a probability distribution, and we usually use the cross-entropy associated with the distribution to define the loss. Mean squared error is the cross-entropy loss associated with an output distribution that is a unit Gaussian, for example, just as with a feedforward network.

This may mean that we maximize the log-likelihood

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}), \tag{10.29}$$

or, if the model includes connections from the output at one time step to the next time step,

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t-1)}). \tag{10.30}$$

Decomposing the joint probability over the sequence of y values as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence.

As a simple example, let us consider the case where the RNN models only a sequence of scalar random variables $\mathbb{Y} = \{y^{(1)}, \ldots, y^{(\tau)}\}$, with no additional inputs x.

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \ldots, \mathbf{y}^{(1)}) \qquad (10.31)$$

where the right-hand side of the bar is empty for $t = 1$, of course. Hence the negative log-likelihood of a set of values $\{y^{(1)}, \ldots, y^{(\tau)}\}$ according to such a model is

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

where

$L^{(t)} = -\log P(\mathrm{y}^{(t)} = y^{(t)} | y^{(t-1)}, y^{(t-2)}, \ldots, y^{(1)})$

One way to interpret an RNN as a graphical model is to view the RNN as defining a graphical model whose structure is the complete graph, able to represent direct dependencies between any pair of y values. The graphical model over the y values with the complete graph structure is shown in Fig. 10.7. The complete graph interpretation of the RNN is based on ignoring the hidden units $h^{(t)}$ by marginalizing them out of the model.

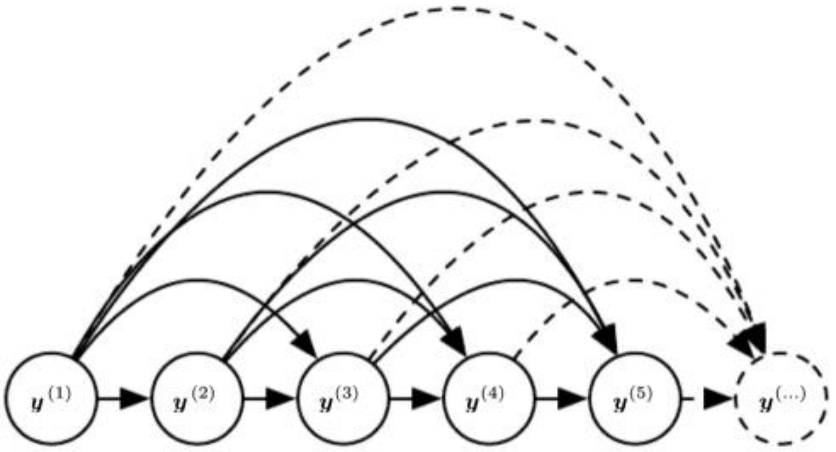


Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \ldots, y^{(t)}, \ldots$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this graph (as in Eq. 10.6) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in Fig. 10.8.

It is more interesting to consider the graphical model structure of RNNs that results from regarding the hidden units $h^{(t)}$ as a random variables. Including the hidden units in the graphical model reveals that the RNN provides a very efficient parametrization of the joint distribution over discrete values with a tabular representation — an array containing a separate entry for each possible assignment of value, with the value of that entry giving the probability of that assignment occurring. If y can take on k different values, the tabular representation would have $O(k^{\tau})$ parameters.

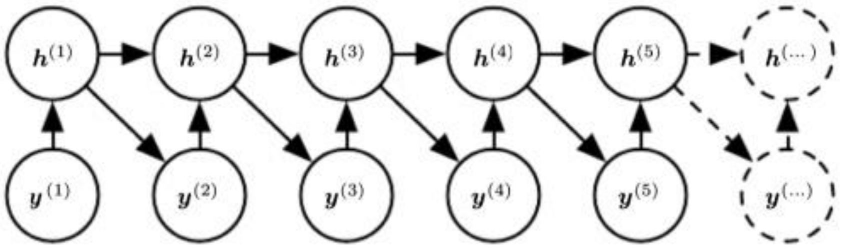


Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on Eq. 10.5. Every stage in the sequence (for $\boldsymbol{h}^{(t)}$ and $\boldsymbol{y}^{(t)}$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

The price recurrent networks pay for their reduced number of parameters is that **optimizing** the parameters may be difficult.

The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps. Equivalently, the assumption is that the conditional probability distribution over the variables at time $t+1$ given the variables at time t is *stationary*, meaning that the relationship between the previous time step and the next step does not depend on t.

To complete our view of an RNN as a graphical model, we must describe how to draw samples from the model. The main operation that we need to perform is simply to sample from the conditional distribution at each time step. However, there is one additional complication. The RNN must have some mechanism for determining the length of the sequence. This can be achieved in various ways.

- In the case when the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence. When that symbol is generated, the sampling process stops. In the training set, we insert this symbol as an extra member of the sequence, immediately after $x^{(\tau)}$ in each training example.
- Another option is to introduce an extra Bernoulli output to the model that represents the decision to either continue generation or halt generation at each time step. This approach is more general than the approach of adding an extra symbol to the vocabulary, because it may be applied to any RNN, rather than only RNNs that output a sequence of symbols.
- Another way to determine the sequence length τ is to add an extra output to the model that predicts the integer τ itself. The model can sample a value of τ and then sample τ steps worth of data. This approach requires adding an extra input to the recurrent update at each time step so that the recurrent update is aware of whether it is near the end of the generated sequence.

$$P(\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}) = P(\tau)P(\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)} \mid \tau). \qquad (10.34)$$

The strategy of predicting τ directly is used for example by Goodfellow *et al.*

Modeling Sequences Conditioned on Context with RNNs

In the previous section we described how an RNN could correspond to a directed graphical model over a sequence of random variables $y^{(t)}$ with no inputs x. Of course, our development of RNNs as in Eq. 10.8 included as a sequence of inputs $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$.

In general, RNNs allow the extension of the graphical model view to represent not only a joint distribution over the y variables but also a conditional distribution over y given $\mathbf{x}$.

As discussed in the context of feedforward networks, any model representing a variable $P(y; \theta)$ can be reinterpreted as a model representing a conditional distribution $P(y|\omega)$ with $\omega = \theta$.

We can extend such a model to represent a distribution $P(y|x)$ by using the same $P(y|\omega)$ as before, but making ω a function of x. In the case of an RNN, this can be achieved in different ways. We review here the most common and obvious choices.

Previously, we have discussed RNNs that take a sequence of vectors $P(y; \theta)$ for $t = 1, \ldots, \tau$ as input. Another option is to take only a single vector x as input. When x is a fixed-size vector, we can simply make it an extra input of the RNN that generates the y sequence. Some common ways of providing an extra input to an RNN are:

1. as an extra input at each time step, or
2. as the initial state $h^{(0)}$, or
3. both.

The first and most common approach is illustrated in Fig. 10.9.

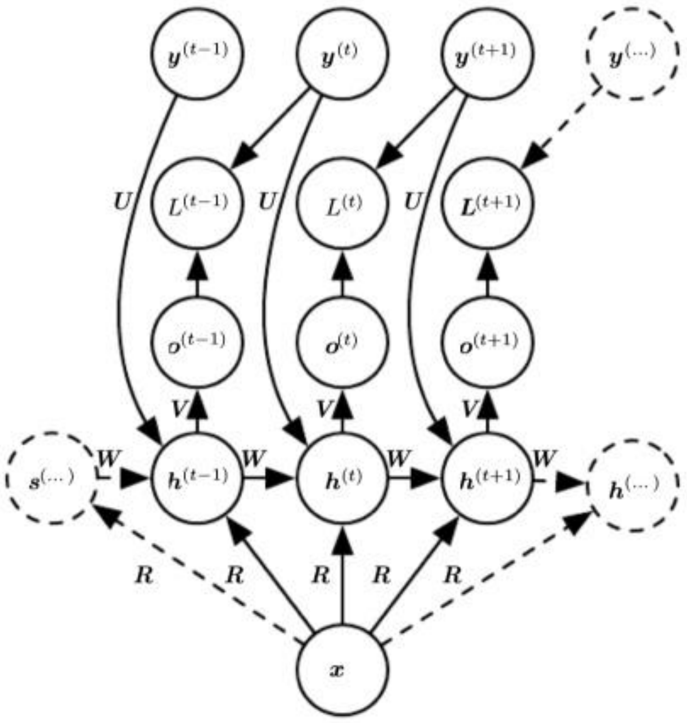


Figure 10.9: An RNN that maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element $\boldsymbol{y}^{(t)}$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).

The interaction between the input x and each hidden unit vector $h^{(t)}$ is parameterized by a newly introduced weight matrix $\mathbf{R}$ that was absent from the model of only the sequence of y values. The same product $x^T R$ is added as additional input to the hidden units every time step.

Rather than receiving only a single vector x as input, the RNN may receive a sequence of vectors $x^{(t)}$ as input. The RNN described in Eq. 10.8 corresponds to a conditional distribution $P(y^{(1)}, \dots, y^{(\tau)} | x^{(1)}, \dots, x^{(\tau)})$ that makes a conditional independence assumption that this distribution factorizes as

$$\prod_t P(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}). \tag{10.35}$$

To remove the conditional independence assumption, we can add connections from the output at time t to the hidden unit at time $t+1$, as shown in Fig. 10.10. (?)

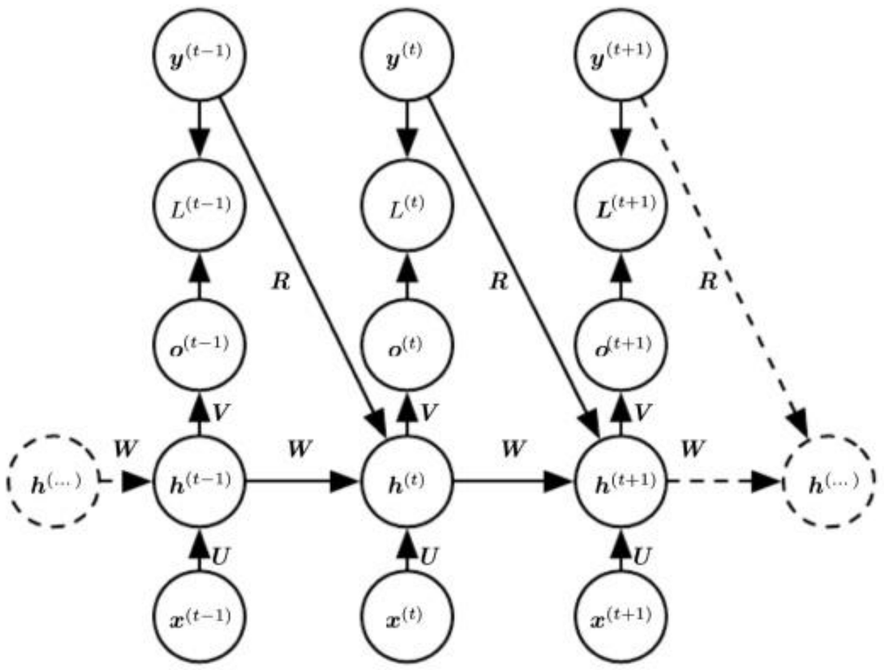
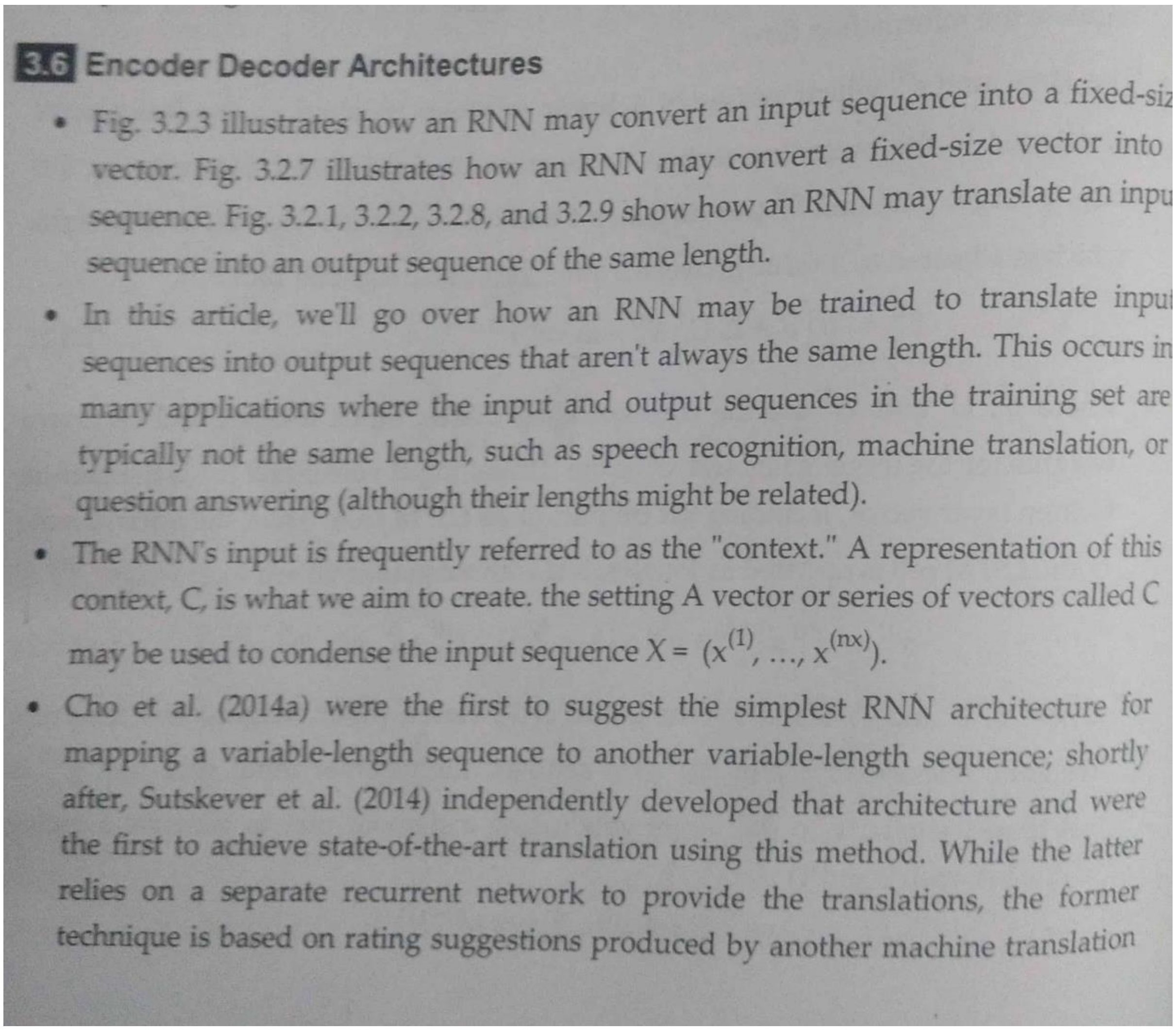


Figure 10.10: A conditional recurrent neural network mapping a variable-length sequence of $\boldsymbol{x}$ values into a distribution over sequences of $\boldsymbol{y}$ values of the same length. Compared to Fig. 10.3, this RNN contains connections from the previous output to the current state. These connections allow this RNN to model an arbitrary distribution over sequences of $\boldsymbol{y}$ given sequences of $\boldsymbol{x}$ of the same length. The RNN of Fig. 10.3 is only able to represent distributions in which the $\boldsymbol{y}$ values are conditionally independent from each other given the $\boldsymbol{x}$ values.

The model can then represent arbitrary probability distributions over the y sequence. This kind of model representing a distribution over a sequence given another sequence still has one restriction, which is that the length of both sequences must be the same.

Encoder-Decoder Sequence-to-Sequence Architectures



3.6 Encoder Decoder Architectures

- Fig. 3.2.3 illustrates how an RNN may convert an input sequence into a fixed-siz vector. Fig. 3.2.7 illustrates how an RNN may convert a fixed-size vector into sequence. Fig. 3.2.1, 3.2.2, 3.2.8, and 3.2.9 show how an RNN may translate an inpu sequence into an output sequence of the same length.
- In this article, we'll go over how an RNN may be trained to translate inpu sequences into output sequences that aren't always the same length. This occurs in many applications where the input and output sequences in the training set are typically not the same length, such as speech recognition, machine translation, or question answering (although their lengths might be related).
- The RNN's input is frequently referred to as the "context." A representation of this context, C, is what we aim to create. the setting A vector or series of vectors called C may be used to condense the input sequence $X = (x^{(1)}, \ldots, x^{(nx)})$.
- Cho et al. (2014a) were the first to suggest the simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence; shortly after, Sutskever et al. (2014) independently developed that architecture and were the first to achieve state-of-the-art translation using this method. While the latter relies on a separate recurrent network to provide the translations, the former technique is based on rating suggestions produced by another machine translation

system. These writers referred to the encoder-decoder or sequence-to-sequence architecture, which is shown in Fig. 3.6.1.

- The concept is relatively straightforward : (1) The input sequence is processed by an encoder, reader, or input RNN. The context C is generally simply a function of the encoder's final concealed state. (2) To produce the output sequence Y = $(y^{(1)}, \ldots, y^{(ny)})$, a decoder, writer or output RNN is conditioned on the fixed-length vector (exactly like in Fig. 3.2.7). This type of design differs from those described in earlier sections of the chapter in that the lengths nx and ny are no longer bound to be equal to one another.

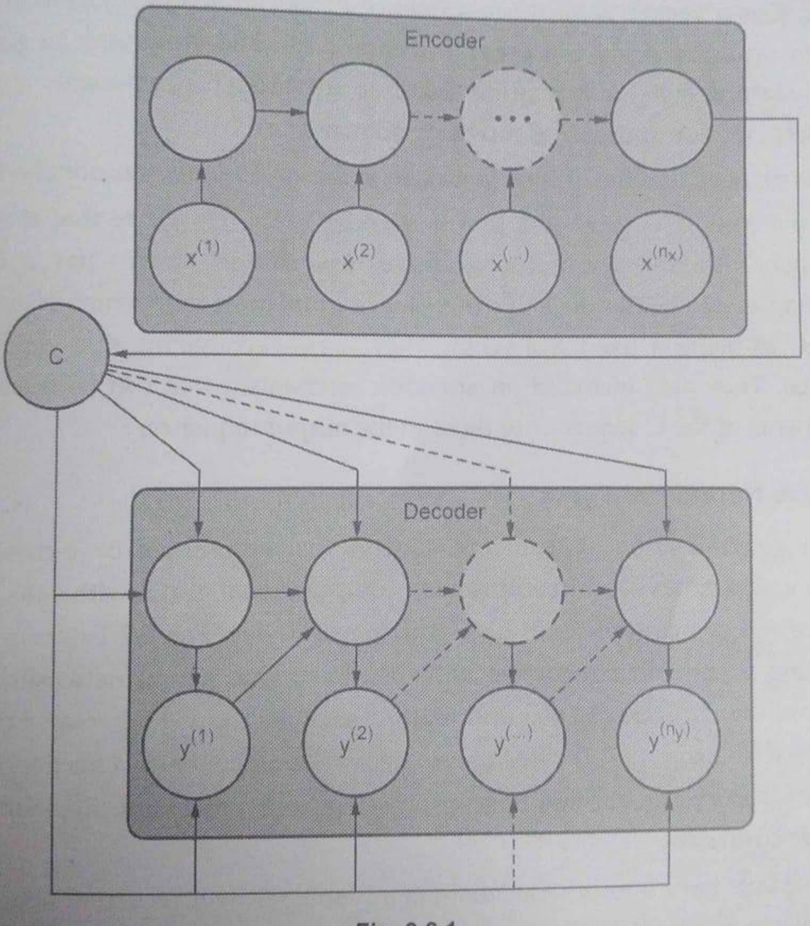
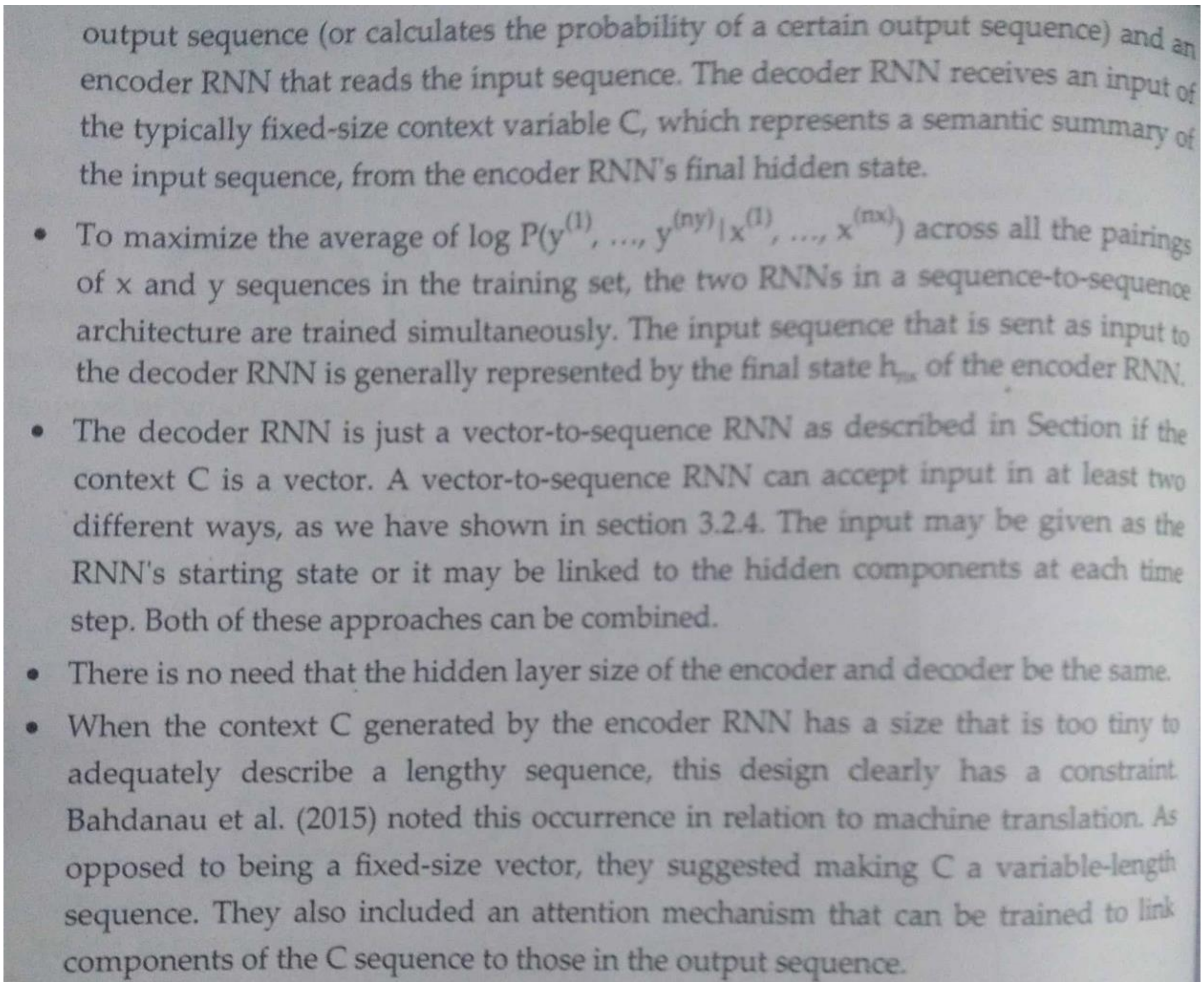


Fig. 3.6.1

- Fig. 3.6.1 An example of an encoder-decoder or sequence-to-sequence RNN architecture that can learn to produce an output sequence $(y^{(1)}, \ldots, y^{(ny)})$ from an input sequence $(x^{(1)}, x^{(2)}, \ldots, y^{(nx)})$. It is made up of a decoder RNN that creates the


output sequence (or calculates the probability of a certain output sequence) and an encoder RNN that reads the input sequence. The decoder RNN receives an input of the typically fixed-size context variable C, which represents a semantic summary of the input sequence, from the encoder RNN's final hidden state.

- To maximize the average of log $P(y^{(1)}, \ldots, y^{(ny)} | x^{(1)}, \ldots, x^{(nx)})$ across all the pairings of x and y sequences in the training set, the two RNNs in a sequence-to-sequence architecture are trained simultaneously. The input sequence that is sent as input to the decoder RNN is generally represented by the final state h_{nx} of the encoder RNN.
- The decoder RNN is just a vector-to-sequence RNN as described in Section if the context C is a vector. A vector-to-sequence RNN can accept input in at least two different ways, as we have shown in section 3.2.4. The input may be given as the RNN's starting state or it may be linked to the hidden components at each time step. Both of these approaches can be combined.
- There is no need that the hidden layer size of the encoder and decoder be the same.
- When the context C generated by the encoder RNN has a size that is too tiny to adequately describe a lengthy sequence, this design clearly has a constraint. Bahdanau et al. (2015) noted this occurrence in relation to machine translation. As opposed to being a fixed-size vector, they suggested making C a variable-length sequence. They also included an attention mechanism that can be trained to link components of the C sequence to those in the output sequence.

Deep Recurrent Networks

The computation in most recurrent neural networks can be decomposed into three blocks of parameters and associated transformations:

1. From the input to the hidden state

2. From the previous hidden state to the next hidden state

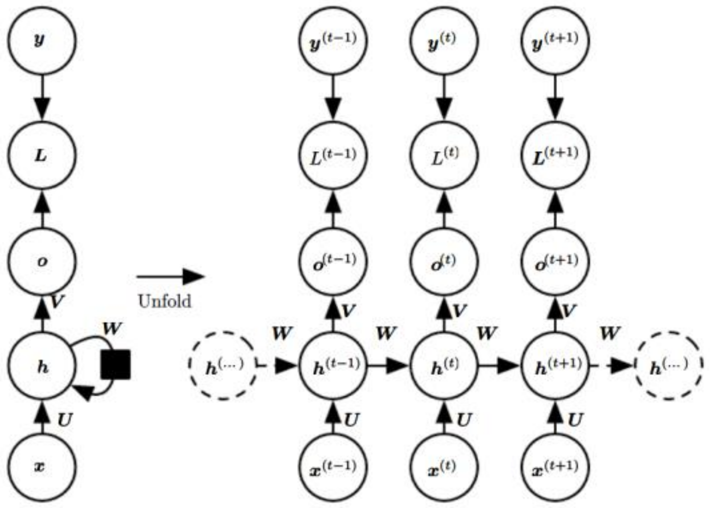
3. From the hidden state to the output

With the RNN architecture shown each of these three blocks is associated with a single weight matrix, i.e.,

• When the network is unfolded, each of these corresponds to a shallow transformation.

• By a shallow Transformation we mean a transformation that would be represented a single layer within a deep MLP.

• Typically, this is a transformation represented by a learned affine transformation followed by a fixed nonlinearity.

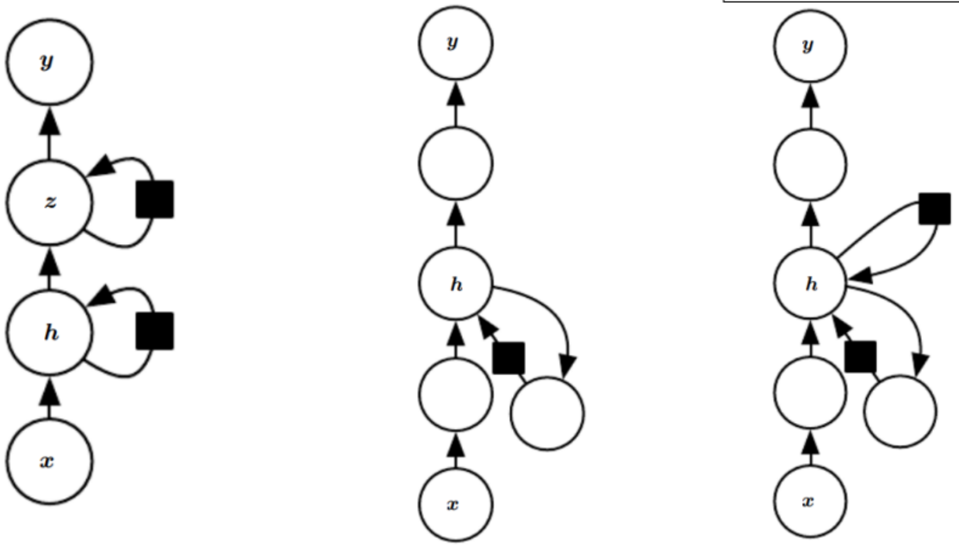


Ways of making an RNN deep

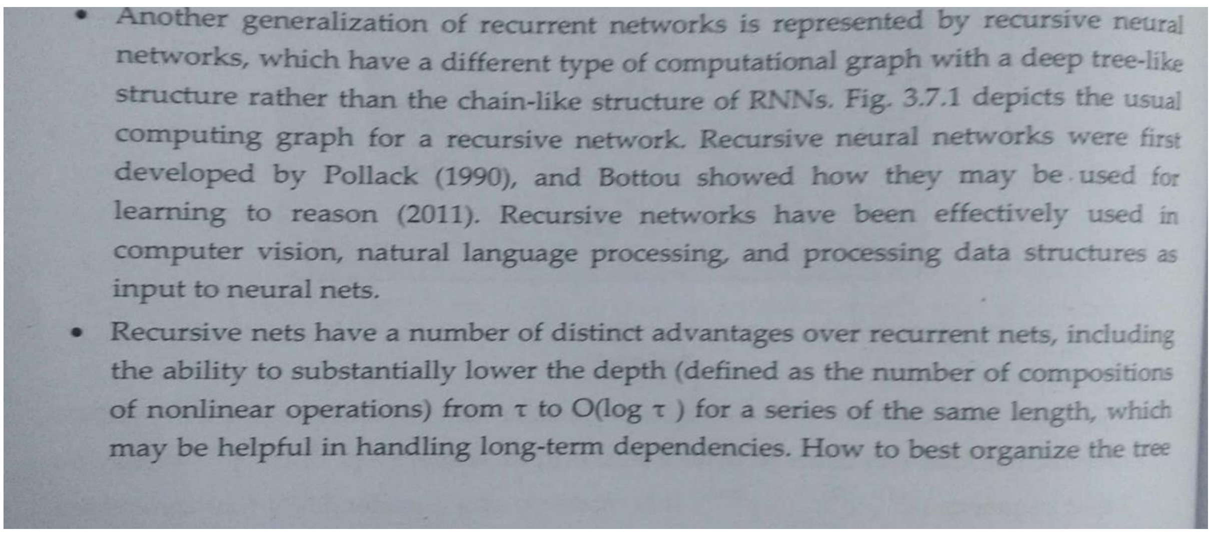
1. Hidden recurrent state can be broken down into groups organized hierarchically

2. Deeper computation can be introduced in the input-hidden, hidden-hidden and hidden-output parts. This may lengthen the shortest path linking different time steps

3. The path-lengthening effect can be mitigated by introducing skip connections.



Recursive Neural Networks



- Another generalization of recurrent networks is represented by recursive neural networks, which have a different type of computational graph with a deep tree-like structure rather than the chain-like structure of RNNs. Fig. 3.7.1 depicts the usual computing graph for a recursive network. Recursive neural networks were first developed by Pollack (1990), and Bottou showed how they may be used for learning to reason (2011). Recursive networks have been effectively used in computer vision, natural language processing, and processing data structures as input to neural nets.
- Recursive nets have a number of distinct advantages over recurrent nets, including the ability to substantially lower the depth (defined as the number of compositions of nonlinear operations) from τ to O(log τ) for a series of the same length, which may be helpful in handling long-term dependencies. How to best organize the tree

is still up for debate. A balanced binary tree is an example of a tree structure that is independent of the data.

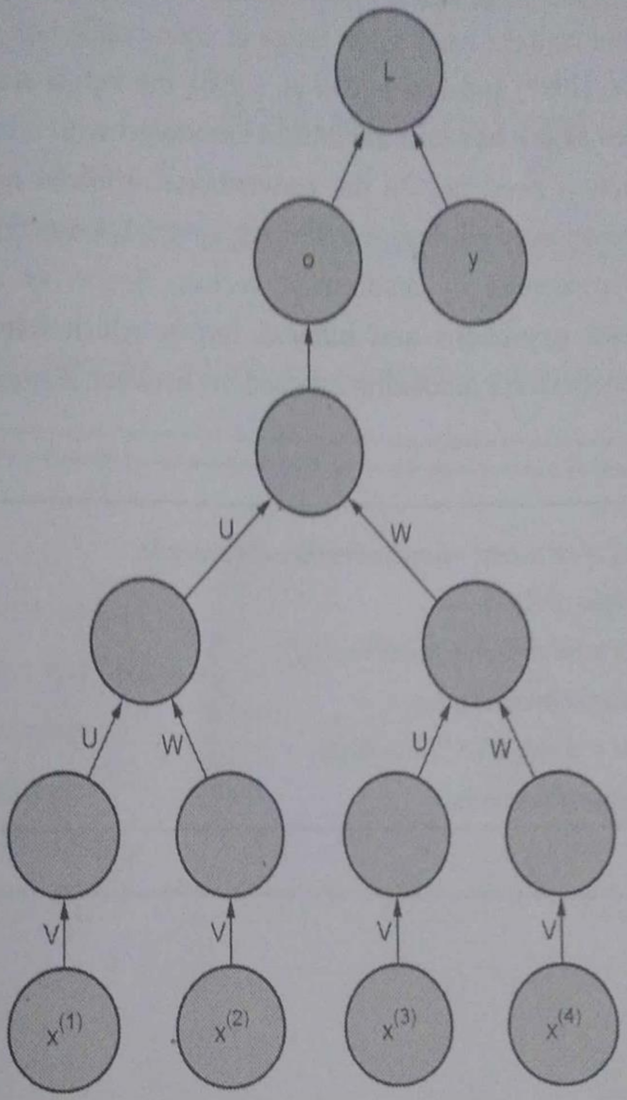
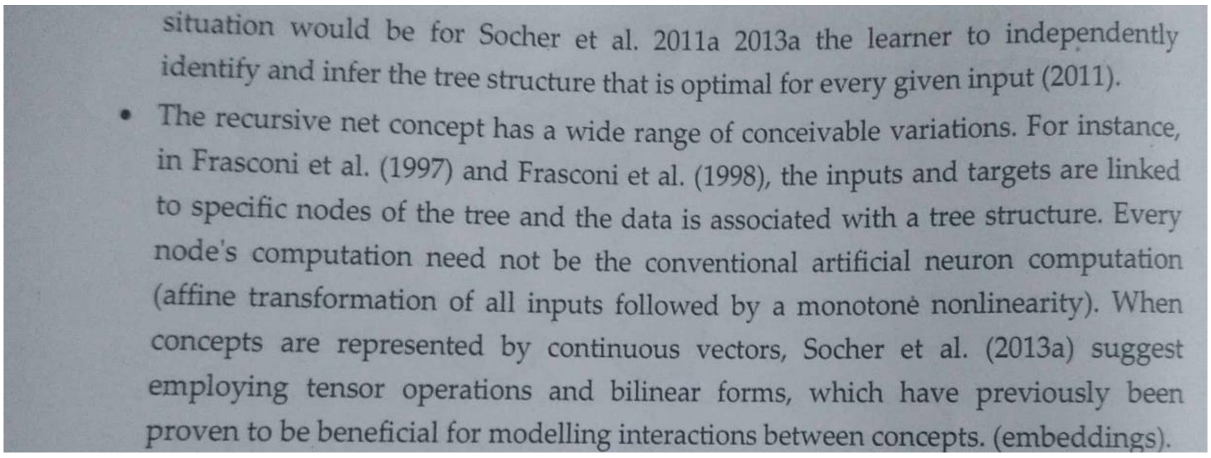


Fig. 3.7.1

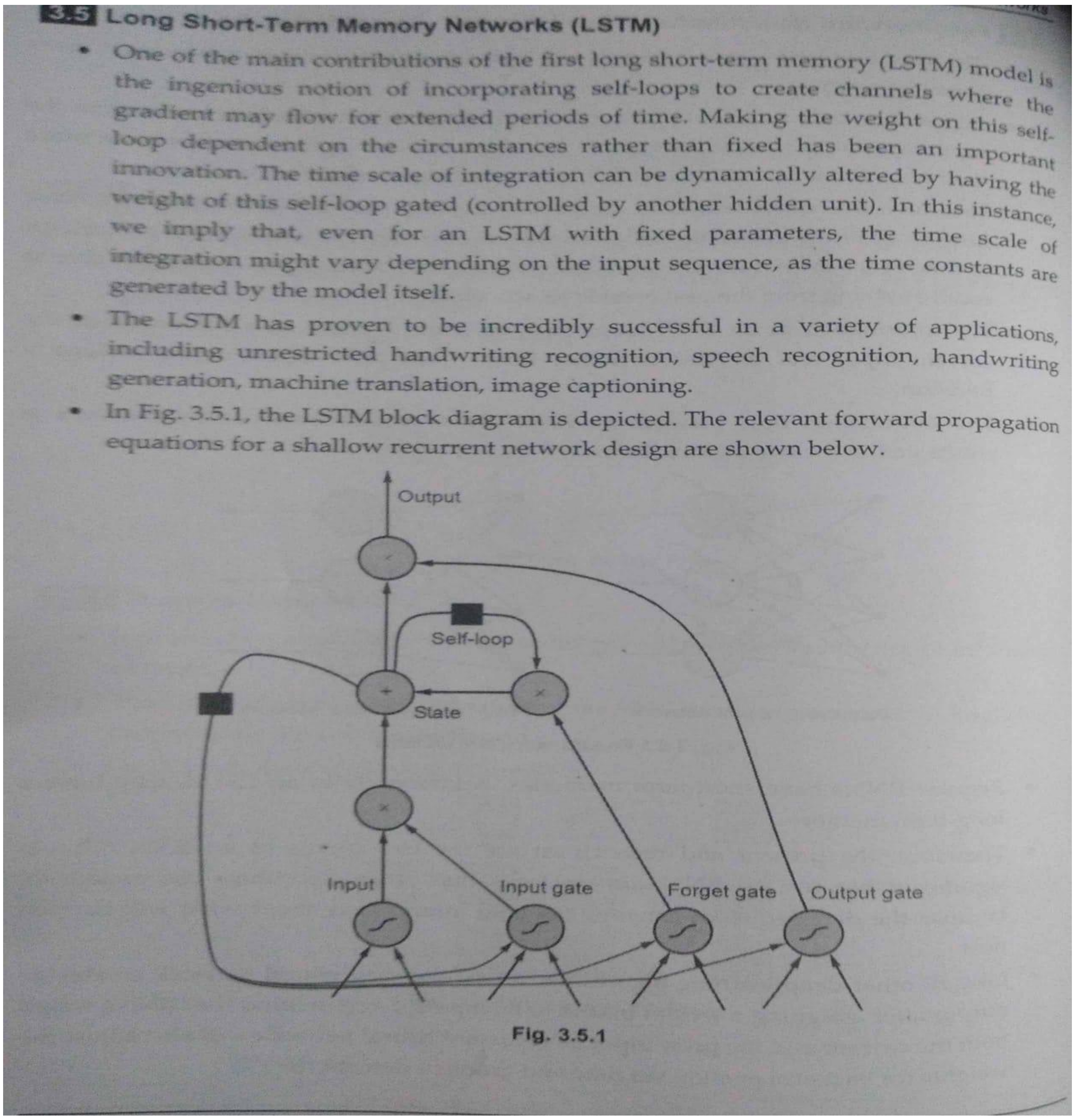
- Fig. 3.7.1 The computational graph of a recursive network is a generalization of the recurrent network from a chain to a tree. A fixed-size representation (the output o) with a fixed set of parameters can be created from a variable-size sequence ($x^{(1)}$, $x^{(2)}$, ..., $x^{(t)}$) (the weight matrices U, V, W). The image shows a scenario of supervised learning where a target y is given and is connected to the entire sequence.
- In some application fields, outside approaches can offer recommendations for the best tree structure. For instance, while processing sentences in natural language, the recursive network's tree structure can be adjusted to match the sentence's parse tree as supplied by the natural language parser. As mentioned by Bottou, the ideal



situation would be for Socher et al. 2011a 2013a the learner to independently identify and infer the tree structure that is optimal for every given input (2011).

- The recursive net concept has a wide range of conceivable variations. For instance, in Frasconi et al. (1997) and Frasconi et al. (1998), the inputs and targets are linked to specific nodes of the tree and the data is associated with a tree structure. Every node's computation need not be the conventional artificial neuron computation (affine transformation of all inputs followed by a monotone nonlinearity). When concepts are represented by continuous vectors, Socher et al. (2013a) suggest employing tensor operations and bilinear forms, which have previously been proven to be beneficial for modelling interactions between concepts. (embeddings).

The Long Short-Term Memory and Other Gated RNNs



3.5 Long Short-Term Memory Networks (LSTM)

- One of the main contributions of the first long short-term memory (LSTM) model is the ingenious notion of incorporating self-loops to create channels where the gradient may flow for extended periods of time. Making the weight on this self-loop dependent on the circumstances rather than fixed has been an important innovation. The time scale of integration can be dynamically altered by having the weight of this self-loop gated (controlled by another hidden unit). In this instance, we imply that, even for an LSTM with fixed parameters, the time scale of integration might vary depending on the input sequence, as the time constants are generated by the model itself.
- The LSTM has proven to be incredibly successful in a variety of applications, including unrestricted handwriting recognition, speech recognition, handwriting generation, machine translation, image captioning.
- In Fig. 3.5.1, the LSTM block diagram is depicted. The relevant forward propagation equations for a shallow recurrent network design are shown below.

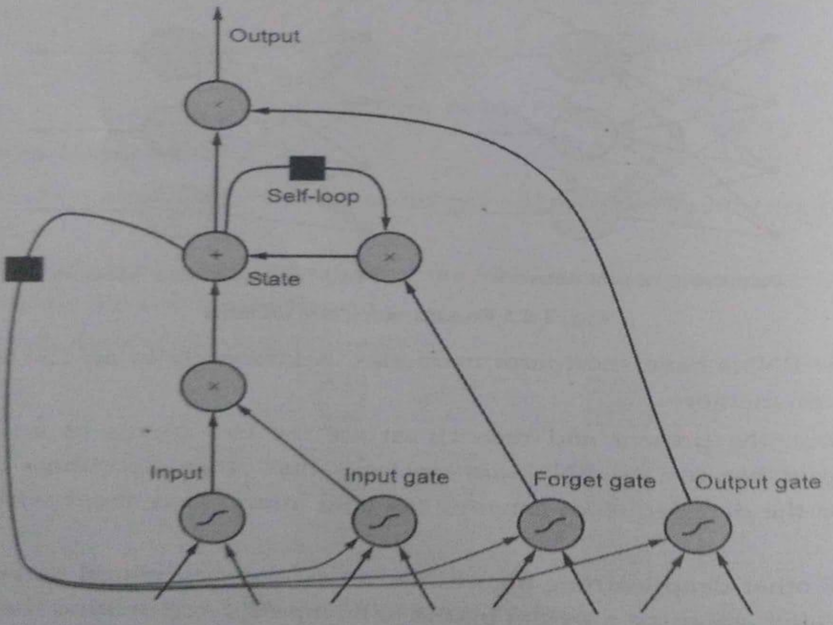


Fig. 3.5.1

- Fig. 3.5.1 LSTM recurrent network "cell" block diagram. Recurrent connections between cells take the place of the typical hidden units seen in conventional recurrent networks. A typical artificial neuron unit computes an input characteristic. In the event that the sigmoidal input gate permits it, its value may be accumulated into the state. The forget gate regulates the weight of the state unit's linear self-loop. The output gate has the ability to disable the cell's output. While the input unit may have any squashing nonlinearity, all gating units have a sigmoid nonlinearity. An additional input to the gating units may also be provided by the state unit. A single time step's worth of delay is represented by the black square.
- The usage of deeper architectures has also proved fruitful. LSTM recurrent networks contain "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN, as opposed to a unit that just applies an elementwise nonlinearity to the affine transformation of inputs and recurrent units. The inputs and outputs of each cell are identical to those of a regular recurrent network, but there are extra parameters and a set of gating units to regulate the information flow.
- The state unit $s_i^{(t)}$ which possesses a linear self-loop identical to the leaky units mentioned in the preceding section, is the most crucial element. However, in this case, a forget gate unit $f_i^{(t)}$ (for time step t and cell i regulates the self-loop weight, which is adjusted to a value between 0 and 1 through a sigmoid unit :

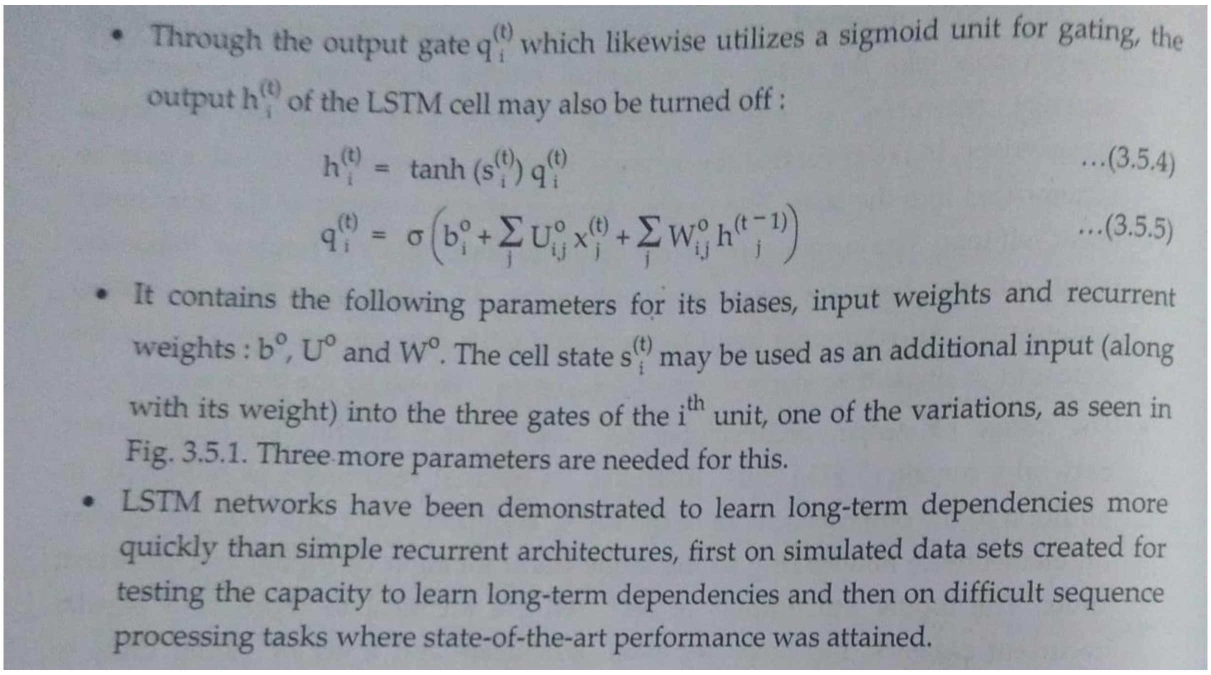
$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \qquad \text{...(3.5.1)}$$

- where b^f, U^f and W^f are the corresponding biases, input weights and recurrent weights for the forget gates and $x^{(t)}$ is the current input vector and $h^{(t)}$ is the current hidden layer vector, including the outputs of all LSTM cells. Thus, the internal state of the LSTM cell is updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right) \qquad \text{...(3.5.2)}$$

where b, U and W stand for the LSTM cell's biases, input weights, and recurrent weights, respectively. With its own settings, the external input gate unit $g_i^{(t)}$ is calculated similarly to the forget gate (using a sigmoid unit to generate a gating value between 0 and 1) :

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{ij}^g x_j^{(t)} + \sum_j W_{ij}^g h_j^{(t-1)}\right) \qquad \text{...(3.5.3)}$$



- Through the output gate $q_i^{(t)}$ which likewise utilizes a sigmoid unit for gating, the output $h_i^{(t)}$ of the LSTM cell may also be turned off :

$$h_i^{(t)} = \tanh(s_i^{(t)})\, q_i^{(t)} \qquad \ldots(3.5.4)$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right) \qquad \ldots(3.5.5)$$

- It contains the following parameters for its biases, input weights and recurrent weights : b^o, U^o and W^o. The cell state $s_i^{(t)}$ may be used as an additional input (along with its weight) into the three gates of the i^{th} unit, one of the variations, as seen in Fig. 3.5.1. Three more parameters are needed for this.
- LSTM networks have been demonstrated to learn long-term dependencies more quickly than simple recurrent architectures, first on simulated data sets created for testing the capacity to learn long-term dependencies and then on difficult sequence processing tasks where state-of-the-art performance was attained.

Applications of Recurrent Neural Networks

Language Modelling and Prediction:

In this method, the likelihood of a word in a sentence is considered. The probability of the output of a particular time-step is used to sample the words in the next iteration(memory). In Language Modelling, input is usually a sequence of words from the data and output will be a sequence of predicted word by the model. While training we set ***xt+1 = ot***, the output of the previous time step will be the input of the present time step.

Speech Recognition:

A set of inputs containing phoneme(acoustic signals) from an audio is used as an input. This network will compute the phonemes and produce a phonetic segments with the likelihood of output.

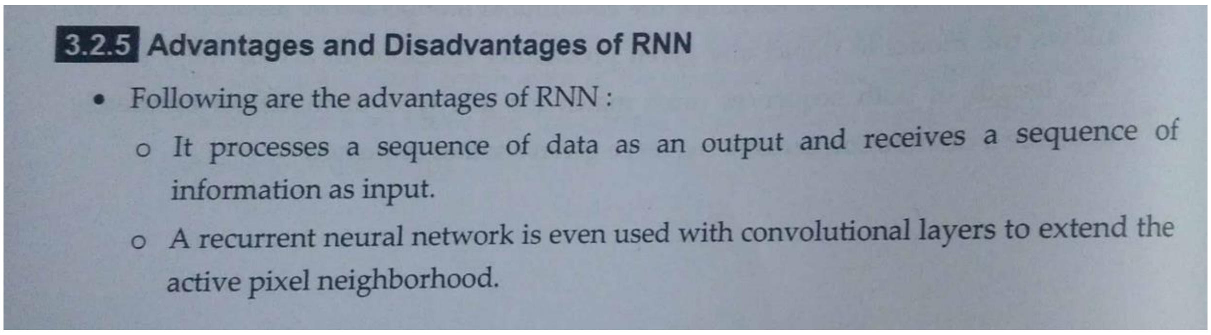
Machine Translation:

In Machine Translation, the input is will be the source language(e.g. Hindi) and the output will be in the target language(e.g. English). The main difference between Machine Translation and Language modelling is that the output starts only after the complete input has been fed into the network.

Image recognition and characterization:

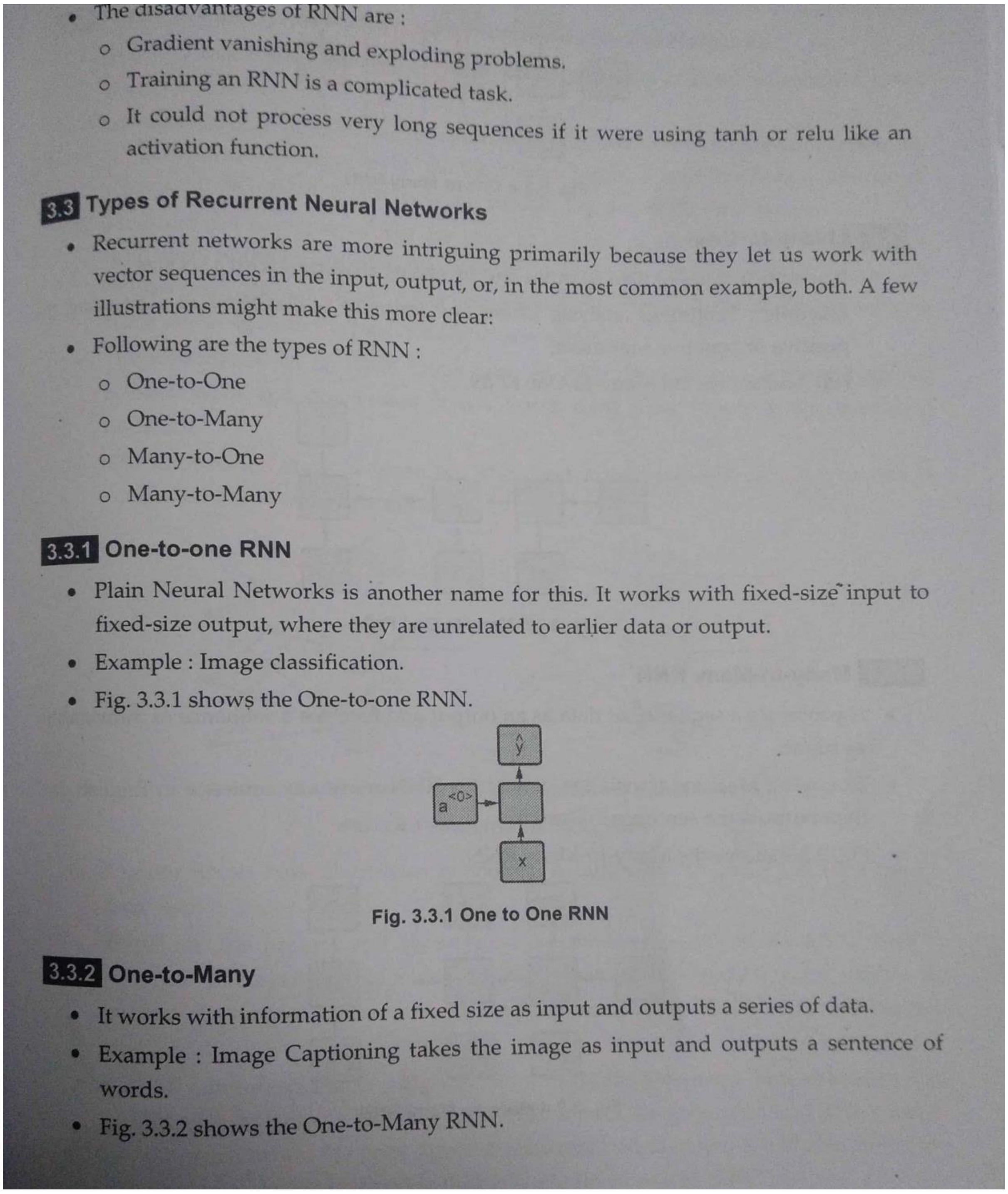
Recurrent Neural Network along with a ConvNet work together to recognize an image and give a description about it if it is unnamed. This combination of neural network works in a beautiful and it produces fascinating results. Here is a visual description about how it goes on doing this, the combined model even aligns the generated words with features found in the images.

Advantages and Disadvantages of RNN



3.2.5 Advantages and Disadvantages of RNN

- Following are the advantages of RNN :
  - It processes a sequence of data as an output and receives a sequence of information as input.
  - A recurrent neural network is even used with convolutional layers to extend the active pixel neighborhood.



- The disadvantages of RNN are :
  - Gradient vanishing and exploding problems.
  - Training an RNN is a complicated task.
  - It could not process very long sequences if it were using tanh or relu like an activation function.

3.3 Types of Recurrent Neural Networks

- Recurrent networks are more intriguing primarily because they let us work with vector sequences in the input, output, or, in the most common example, both. A few illustrations might make this more clear:
- Following are the types of RNN :
  - One-to-One
  - One-to-Many
  - Many-to-One
  - Many-to-Many

3.3.1 One-to-one RNN

- Plain Neural Networks is another name for this. It works with fixed-size input to fixed-size output, where they are unrelated to earlier data or output.
- Example : Image classification.
- Fig. 3.3.1 shows the One-to-one RNN.

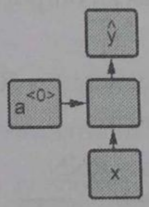


Fig. 3.3.1 One to One RNN

3.3.2 One-to-Many

- It works with information of a fixed size as input and outputs a series of data.
- Example : Image Captioning takes the image as input and outputs a sentence of words.
- Fig. 3.3.2 shows the One-to-Many RNN.

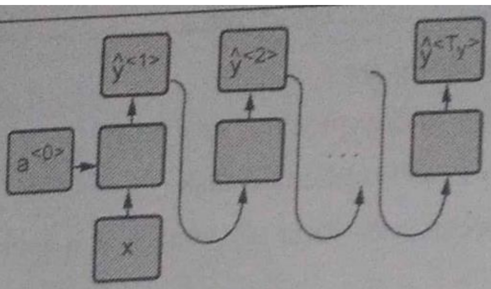


Fig. 3.3.2 One to Many RNN

3.3.3 Many-to-One

- It accepts a series of data as input and produces an output with a set size.
- Example : Sentiment analysis where any sentence is classified as expressing th positive or negative sentiment.
- Fig. 3.3.3 shows the Many-to-One RNN.

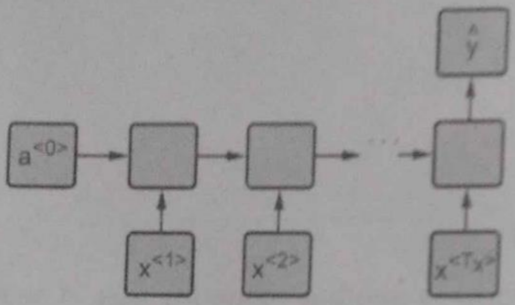


Fig. 3.3.3 Many to One RNN

3.3.4 Many-to-Many RNN

- It processes a sequence of data as an output and receives a sequence of information as input.
- Example : Machine translation, where the RNN reads any sentence in English and then outputs the sentence in French.
- Fig. 3.3.4 shows the Many-to-Many RNN.

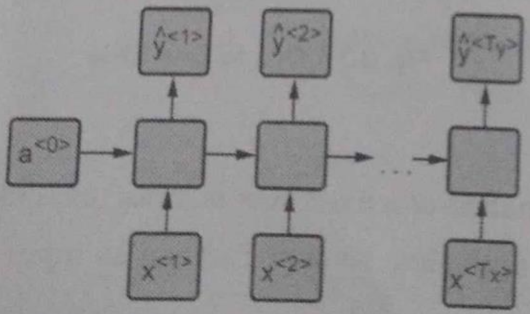


Fig. 3.3.4 Many to Many RNN

3.4 Feed-Forward Neural Networks vs Recurrent Neural Networks

- The way that RNNs and feed-forward neural networks channel information gives them their names.
- Information only flows in one direction in a feed-forward neural network - from the input layer to the output layer, via the hidden layers. Never touching a node more than once, the information travels in a straight line across the network.
- Feed-forward neural networks are poor at making predictions because they have little recall of the information they receive. A feed-forward network has no concept of time order since it simply takes into account the current input. It just isn't able to recall anything from the past outside its schooling.
- The information in an RNN loops back on itself. It takes into account both the current input and the lessons it has learnt from prior inputs when making a decision.
- The information flow between an RNN and a feed-forward neural network is contrasted in the two pictures below.

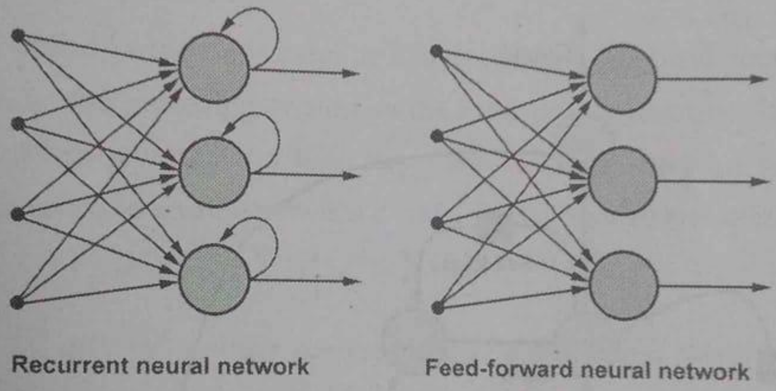


Fig. 3.4.1 Feedforward NN Vs RNN

- Regular RNNs have short-term memories. Additionally to an LSTM, they have a long-term memory.
- Therefore, the present and recent past are the two inputs of an RNN. This is significant because an RNN can do tasks that other algorithms are unable to, because the data sequence conveys essential information about what will happen next.
- Like all other deep learning algorithms, a feed-forward neural network creates an output after assigning a weight matrix to its inputs. Keep in mind that RNNs weigh both the current and the prior input. A recurrent neural network will also adjust the weights for backpropagation via time and gradient descent (BPTT).