Unit III

3 Recurrent Neural Networks

Syllabus

Recurrent Neural Networks : Types of Recurrent Neural Networks, Feed-Forward Neural Networks vs Recurrent Neural Networks, Long Short-Term Memory Networks (LSTM), Encoder Decoder architectures, Recursive Neural Networks

Contents

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 :

$$s^{(t)} = f(s^{(t-1)}; \theta) \quad \ldots(3.1.1)$$

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.

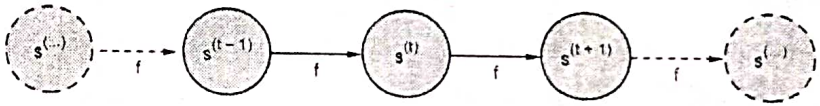

Fig. 3.1.1 A computational graph that has been unfurled serves as an illustration of the classical dynamical system given by Eq. 3.1.1

- 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.

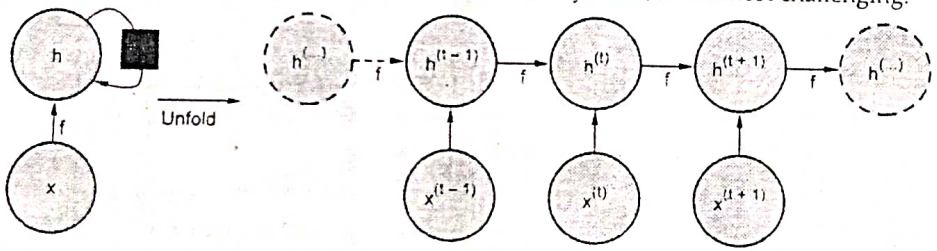

Fig. 3.1.2 An output less recurrent network

- 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 ($x^{(t)}$, $x^{(t1)}$, $x^{(t2)}$, ..., $x^{(2)}$, $x^{(1)}$) 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.

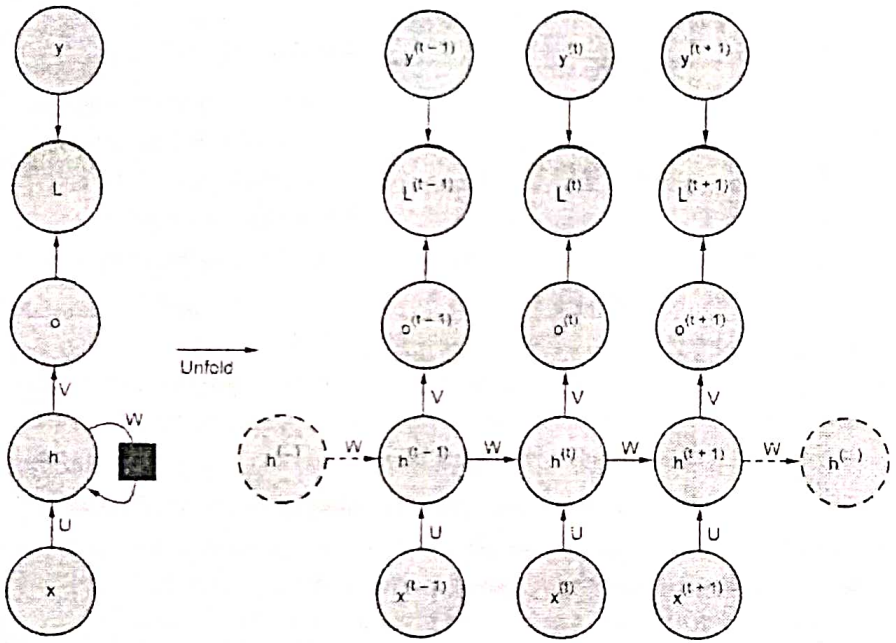

Fig. 3.2.1

- 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 :

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \quad \ldots(3.2.1)$$

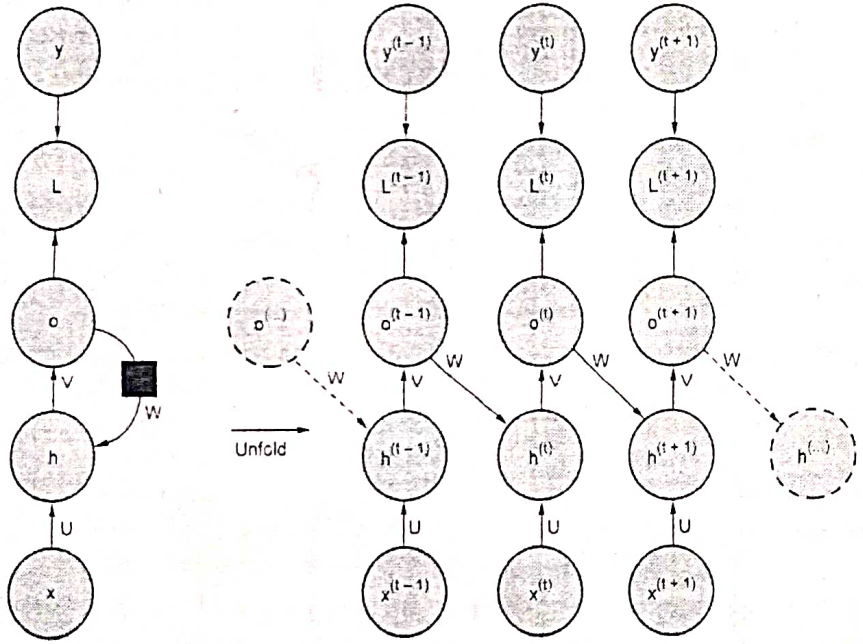

Fig. 3.2.2

- 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.

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

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

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

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

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

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

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

where $P_{\text{model}}\,(y^{(t)} | \{x^{(1)}, \ldots, x^{(t)}\})$ 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 ?

3.2.1 Teacher Forcing and Networks with Output Recurrence

- Because it lacks hidden-to-hidden recurrent connections, the network (illustrated in Fig. 3.2.2) with recurrent connections simply from the output at one time step to the hidden units at the following time step is strictly less effective. It cannot imitate a general-purpose Turing machine, for instance. The output units must record all of the past data that the network will use to make predictions about the future since this network lacks hidden-to-hidden recurrence. If the user does not know how to define the whole state of the system and does not include it as part of the training set goals, the output units are unlikely to catch the essential information about the prior history of the input.
- The benefit of removing hidden-to-hidden recurrence is that all time steps are decoupled for any loss function based on comparing the prediction at time t to the training objective at time t. Thus, training may be done in parallel while computing the gradient for each step t separately. The training set already contains the optimal value of the output for the previous time step, thus there is no need to compute it first.

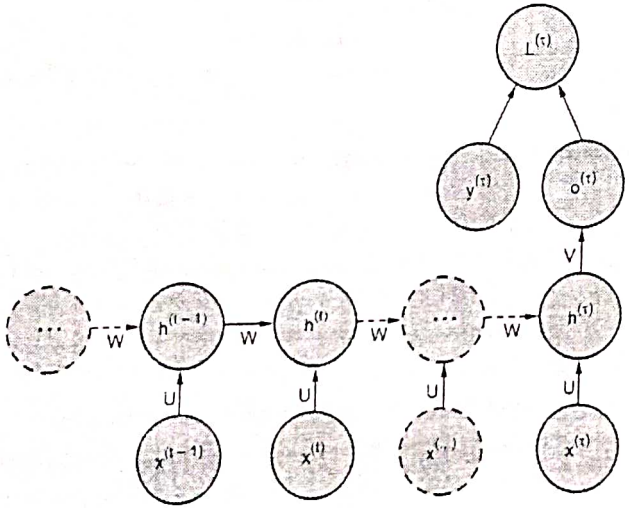

Fig. 3.2.3

- Fig. 3.2.3 Single output, time-unfolded recurrent neural network at the conclusion of the sequence. A network of this type can be used to condense a sequence into a fixed-size representation that can be used as input for additional processing. The gradient on the output o(t) may have a goal right at the end (as shown above), or it may be possible to retrieve it through back-propagating from other downstream modules.
- Trainer forcing can be used to train models that contain recurrent connections from their outputs that go back into the model. Using a technique called "teacher forcing," which is derived from the maximum likelihood criteria, the model is trained with the ground truth output $y^{(t)}$ as input at time t + 1. By looking at a sequence with two time steps, we can see this.
- The conditional maximum likelihood criterion is

$$\log p(y^{(1)}, y^{(2)} | x^{(1)}, x^{(2)}) \quad \ldots(3.2.8)$$

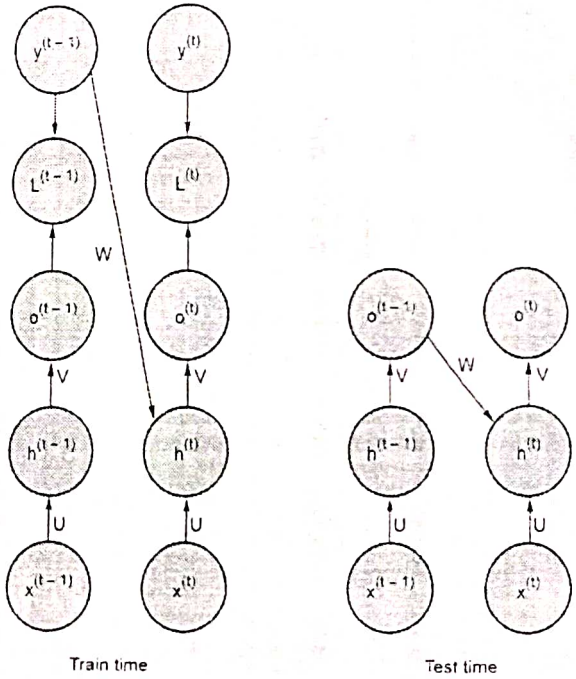

Fig. 3.2.4 An RNN in which the feedback link from the output to the hidden layer is the sole repetition

- RNNs that have links between their output and their hidden states at the following time step might benefit from the training strategy known as teacher forcing. At train time, we supply $h^{(t+1)}$ with the proper output $y^{(t)}$ taken from the train set. (Right) The real output is typically unknown when the model is installed. In this instance, we use the model's output $o^{(t)}$ to estimate the proper output $y^{(t)}$ and then feed that output back into the model.

$$= \log p(y^{(2)}|y^{(1)}, x^{(1)}, x^{(2)}) + \log p(y^{(1)}|x^{(1)}, x^{(2)}) \quad \ldots(3.2.9)$$

- In this illustration, we can see that the model is trained to maximize the conditional probability of $y^{(2)}$ at time t = 2, given the current x sequence and the previous y value from the training set. Thus, maximum likelihood recommends that during training, these connections be provided with target values that indicate what the desired output should be, rather than feeding the model's own output back into itself. Fig. 3.2.4 provides an illustration of this.
- Initially, our motivation for teacher forcing was to prevent time-back propagation in models without hidden-to-hidden links. Models with hidden-to-hidden connections may still use teacher forcing as long as they have connections going from the output of one time step to values computed at the following time step. The BPTT method is required if the hidden units start to depend on previous time steps, though. Thus, certain models may be trained using both BPTT and teacher forcing.
- If the network is later employed in an open-loop mode, with samples from the output distribution supplied back as input, the drawback of severe instructor forcing becomes apparent. The inputs the network receives during training may not be the same as the inputs it receives during testing in this situation. Training using both teacher-forced and free-running inputs can help to alleviate this issue, for instance by anticipating the proper goal a number of steps in the future using the unfolding recurrent output-to-input routes.
- In this manner, the network may learn to account for input circumstances that were not seen during training and how to map the state back towards one that will cause the network to create appropriate outputs after a few steps, such as those it generates on its own in the free-running mode. Another method to reduce the discrepancy between the inputs observed during training and the inputs seen during testing randomly selects produced values or actual data values as input. The goal of this technique is to gradually employ more of the produced values as input by utilizing a curriculum learning mechanism.

3.2.2 Computing the Gradient in a Recurrent Neural Network

- It is simple to calculate the gradient using a recurrent neural network. The unrolled computational network is simply subjected to the generalized back-propagation method. No particular algorithms are required. The back-propagation through time (BPTT) technique applies back-propagation to the unrolled graph. Then, to train an RNN, gradients generated from back-propagation can be employed with any general-purpose gradient-based approach.
- We give an example of how to compute gradients via BPTT for the aforementioned RNN equations in order to give the reader an understanding of how the BPTT method functions (Equation (3.2.1) and Equation (3.2.4)). The parameters U, V, W, b and c, as well as the series of nodes indexed by t for $x^{(t)}$, $h^{(t)}$, $o^{(t)}$ and $L^{(t)}$ are all nodes in our computational graph. Based on the gradient calculated at nodes that follow it in the graph, we must recursively compute the gradient ∇NL for each node N.
- We start the recursion with the nodes immediately preceding the final loss

$$\frac{\partial L}{\partial L^{(t)}} = 1 \quad \ldots(3.2.10)$$

- In this derivation, we assume that the vector $\hat{y}$ of probabilities over the output is obtained by passing the outputs $o^{(t)}$ as the argument to the softmax function. Additionally, we presume that given the current input, the loss is the negative log-likelihood of the real goal $y^{(t)}$.
- The gradient $\nabla o^{(t)} L$ on the outputs at time step t, for all i, t, is as follows :

$$(\nabla o^{(t)} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - 1_{i,y}{}^{(t)} \quad \ldots(3.2.11)$$

- Starting at the conclusion of the series, we work our way backward. At the final time step τ, $h^{(\tau)}$ only has $o^{(\tau)}$ as a descendent, so its gradient is simple :

$$\nabla h^{(\tau)} L = V^T \nabla o^{(\tau)} L \quad \ldots(3.2.12)$$

- 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 as descendants both $o^{(t)}$ and $h^{(t+1)}$.

- Its gradient is thus given by,

$$\nabla h^{(t)}L = \left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)(\nabla h^{(t+1)}L) + \left(\frac{\partial o^{(t)}}{\partial h^{(t)}}\right)T(\nabla o^{(t)}L) \quad \ldots(3.2.13)$$

$$= W^T(\nabla h^{(t+1)}L)\,\mathrm{diag}(1 - h^{(t+1)})^2) + V^T(Vo^{(t)}L) \quad \ldots(3.2.14)$$

where 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.

- We may acquire the gradients on the parameter nodes once we have the gradients on the internal nodes of the computational network. We must be careful when designating calculus operations using these variables since they are shared over several time steps. The bprop approach, which computes the contribution of a single edge in the computational graph to the gradient, is used in the equations we want to employ.
- The calculus ∇W f operator, on the other hand, accounts for the contribution of W to the value of f resulting from each edge in the computational graph. In order to clear up this uncertainty, we construct dummy variables W(t), which are duplicates of W that are only utilized at time step t. The weights' contribution to the gradient at time step t is then shown by the symbol ∇W(t).
- The gradient on the remaining parameters is represented by the following notation:

$$\nabla cL = \sum_t \left(\frac{\partial o^{(t)}}{\partial c}\right)^T \nabla o^{(t)}L = \sum_t \nabla o^{(t)}L \quad \ldots(3.2.15)$$

$$\nabla bL = \sum_t \left(\frac{\partial h^{(t)}}{\partial b^{(t)}}\right)^T \nabla h^{(t)}L = \sum_t \mathrm{diag}(1 - h^{(t)})^2)\,\nabla h^{(t)}L \quad \ldots(3.2.16)$$

$$\nabla WL = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla Vo_i^{(t)} = \sum_t \nabla o^{(t)}Lh(t)^T \quad \ldots(3.2.17)$$

$$\nabla WL = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla W^{(t)}h_i^{(t)} \quad \ldots(3.2.18)$$

$$= \sum_t \mathrm{diag}(1 - (h^{(t)})^2)\,(\nabla h^{(t)}L)h^{(t-1)T} \quad \ldots(3.2.19)$$

$$\nabla UL = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla U^{(t)}h_i^{(t)} \quad \ldots(3.2.20)$$

$$= \sum_t \mathrm{diag}(1 - (h^{(t)})^2)\,(\nabla h^{(t)}L)x^{(t)T} \quad \ldots(3.2.21)$$

- Since it has no parameters as ancestors in the computational graph defining the loss, we do not need to compute the gradient with respect to x(t) for training.

3.2.3 Recurrent Networks as Directed Graphical Models

- Cross-entropies between training objectives $y^{(t)}$ and outputs o were the losses $L^{(t)}$ in the example recurrent network we have created so far $o^{(t)}$. In theory, practically any loss may be used with a recurrent network, much like with a feedforward network. The job should guide the loss selection.
- We often want to interpret the output of the RNN as a probability distribution, similar to how we would with a feedforward network, and we typically use the cross-entropy of that distribution to quantify the loss.
- The cross-entropy loss associated, for instance, with a feedforward network and a unit Gaussian output distribution is called mean squared error.
- When we choose a training target for predictive log-likelihood, as equation (3.2.4). Using the previous inputs, we train the RNN to estimate the conditional distribution of the following sequence element, y(t).
- This may mean that we maximize the log-likelihood

$$\log p(y^{(t)} | x^{(1)}, \ldots, x^{(t)}) \quad \ldots(3.2.22)$$

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

$$\log p(y^{(t)} | x^{(1)}, \ldots, x^{(t)}, y^{(1)}, \ldots, y^{(t-1)}) \quad \ldots(3.2.23)$$

- One method to capture the whole joint distribution across the entire sequence is to decompose the joint probability over the sequence of y values into a series of one-step probabilistic predictions. The directed graphical model does not contain any edges from any $y^{(i)}$ in the past to any $y^{(t)}$ in the present when we do not feed previous y values as inputs that condition the next step prediction. Given the sequence of x values in this instance, the outputs y are conditionally independent. The directed graphical model has edges from all previous $y^{(i)}$ values to the present $y^{(t)}$ value when we feed the network the real y values (not their prediction, but the actual observed or created values) back.

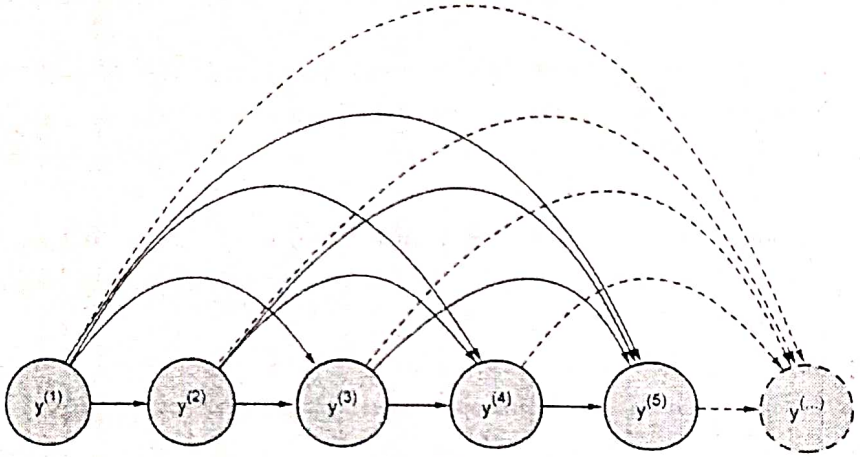

Fig. 3.2.5

- Fig. 3.2.5 A fully connected graphical model for a sequence of values $y^{(1)}$, $y^{(2)}$, ..., $y^{(t)}$, ... shows that each prior observation $y^{(t)}$ may have an impact on the conditional distribution of some $y^{(t)}$ (for t > I given the values from before. It may be exceedingly wasteful to parameterize the graphical model directly using this graph (as in Equation 6), as there are an increasing number of inputs and parameters for each member of the series. As seen in Fig. 3.2.6, RNNs achieve the same complete connectivity but with efficient parameterization.
- Let's take the scenario when the RNN models merely a series of scalar random variables $Y = \{y^{(1)}, ..., y^{(T)}\}$ with no extra inputs x as a basic illustration. Simply expressed, the output at time step t 1 is the input at time step t. The directed graphical model over the y variables is thus defined by the RNN. Using the chain rule for conditional probabilities, we parameterize the joint distribution of these observations :

$$P(Y) = P(y^{(1)}, ..., y^{(T)}) = \prod_{t=1} P(y^{(t)} | y^{(t-1)}, y^{(t-2)}, ..., y^{(1)}) \quad ...(3.2.24)$$

- Thus, naturally, for t = 1, the right-hand side of the bar is empty. Because of this, the negative log-likelihood of a set of values $y^{(1)}$, ... and $y^{(T)}$ in this model is

$$L = \sum_{t} L^{(t)} \quad ...(3.2.25)$$

Where

$$L(t) = -\log P(y^{(t)} = y | y^{(t-1)}, y^{(t-2)}, ..., y^{(1)}) \quad ...(3.3.26)$$

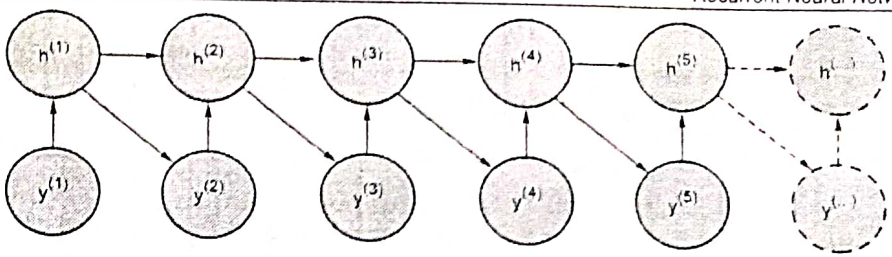

Fig. 3.2.6

- Fig. 3.2.6 Even though the state variable in the RNN's graphical model is a deterministic function of its inputs, its introduction aids in understanding how we may derive a very effective parameterization based on equation (3.1.5). The for $h^{(t)}$ and $y^{(t)}$ stages of the sequence all have the same structure (the same amount of inputs for each node) and they can share parameters with one another.
- Which variables have direct relationships with other variables are indicated by their edges in a graphical model. A common strategy used in graphical models is to eliminate edges that don't correspond to strong interactions in order to increase statistical and computational efficiency. The Markov assumption, for instance, states that the graphical model should only include edges from $\{y^{(t-k)}, ..., y^{(t-1)}\}$ to $y^{(t)}$, rather than include edges from the full previous history. However, in some circumstances, we think that every prior input should have an impact on the sequence's subsequent element.
- When we think that the relationship between the distribution over $y^{(t)}$ and a value of $y^{(i)}$ from the distant past may be more complex than the relationship between $y^{(i)}$ and $y^{(t-1)}$, RNNs can be helpful.
- An RNN may be seen as establishing a graphical model whose structure is the whole graph, capable of representing direct relationships between any pair of y values. This is one method to understand an RNN as a graphical model. Fig. 3.2.5 depicts the whole graph structure and the graphical model over the y values. The hidden units $h^{(t)}$ are marginalized out of the model in the full graph interpretation of the RNN.
- The graphical model structure of RNNs that arises from thinking of the hidden units $h^{(t)}$ as random variables is more fascinating to investigate. It becomes clear

that the RNN offers a highly effective parameterization of the joint distribution over the observations when the hidden units are included in the graphical model. Consider how a tabular representation of an arbitrary joint distribution over discrete values might look : An array with a single entry for each conceivable assignment of values, with the value of that entry indicating the likelihood that that assignment would take place.

- The tabular representation would have $O(\tau)$ parameters if y could have k alternative values. In contrast, the RNN has O(1) parameters as a function of sequence length because of parameter sharing. The RNN's parameter count can be altered to regulate model performance; however, it is not required to scale with sequence length. Equation 3.1.5 demonstrates that the RNN efficiently parameterizes long-term relationships between variables by applying the same function f and parameters repeatedly at each time step.
- The interpretation of the graphical model is shown in Fig. 3.2.6. By serving as an intermediary quantity between the past and the future, the $h^{(t)}$ nodes enable the graphical model to decouple the past and the future. Through its impact on h, a variable $h^{(i)}$ from the remote past may have an impact on a variable $y^{(t)}$. By applying the same conditional probability distributions at each time step, the model may be efficiently parameterized and when all of the variables are observed, it is possible to evaluate the likelihood of the joint assignment of all variables in an effective manner.
- Some procedures are still computationally difficult even after the graphical model has been efficiently parameterized. Predicting missing numbers in the midst of the sequence, for instance, is challenging.
- Recurrent networks trade off their fewer parameters for the potential difficulty in parameter optimization.
- Recurrent networks' use of parameter sharing is predicated on the idea that the same parameters may be applied to many time steps. The equivalent assumption is that the relationship between the previous time step and the future time step does not rely on t and that the conditional probability distribution over the variables at time t + 1 given the variables at time t is stationary. Theoretically, it would be

conceivable to use t as an additional input at each time step and allow the learner to identify any time-dependence while distributing as much information as it can over various time stages.

- The network would then have to extrapolate when faced with new values of t, which would already be much better than using a different conditional probability distribution for each t.
- We must explain how to extract samples from the model in order to fully depict an RNN as a graphical model. Simply taking a sample from the conditional distribution at each time step is the key action we must take. There is one more difficulty, though. The length of the sequence must be determined by the RNN somehow. There are several ways to accomplish this.
- One can add a special symbol that corresponds to the conclusion of a sequence when the output is a symbol selected from a vocabulary. The sampling procedure ends when that symbol is produced. We add this symbol as a second component of the sequence immediately following $x^{(T)}$ in each training example in the training set.
- The addition of an additional Bernoulli output to the model to indicate the choice to either continue or stop creation at each time step provides an alternative. This method is more inclusive than the method of including a new symbol in the vocabulary since it may be used with any RNN, not only those that produce a string of symbols. It could be used, for instance, with an RNN that emits a series of real numbers. Most often, a sigmoid unit trained with the cross-entropy loss serves as the new output unit. By using this method, the sigmoid is trained to make the best possible guess as to whether the sequence will finish or continue at each time step.
- Adding an additional output to the model that predicts the integer τ may also be used to calculate the sequence length . The model is capable of sampling a value for and then sampling data for steps. In order for the recurrent update to be aware of whether it is getting close to the end of the created sequence, this method necessitates introducing an additional input to it at each time step. This additional input may either be the value of τ or $\tau - t$, the number of uncompleted time steps. Without this additional information, the RNN could produce sequences that come to a sudden stop, like a phrase that finishes before it is finished.

- This approach is based on the decomposition

$$P(x^{(1)}, \ldots, x^{(T)}) = P^{(T)}P(x^{(1)}, \ldots, x^{(T)}|T) \quad \ldots(3.2.27)$$

3.2.4 Modeling Sequences Conditioned on Context with RNNs

- We discussed how an RNN may equate to a directed graphical model over a series of random variables $y^{(t)}$ without any inputs x in the preceding section. Of course, the inputs $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$ were incorporated in the creation of our RNNs as in equation (3.2.1). RNNs, in general, enable the expansion of the graphical model view to depict both a conditional distribution over y given x and a joint distribution over the y variables.
- A conditional distribution $P(y|\omega)$ with = can be represented by any model that represents a variable P (y; θ). By employing the same $P(y|\omega)$ as previously but making a function of x, we may extend such a model to describe a distribution $P(y|x)$. This may be done in several ways when using an RNN. Here, we go over the most popular and obvious options.
- RNNs that accept a series of vectors $x^{(t)}$ for t = 1, ..., as input have already been covered. Another choice is to accept x as the sole input vector. We can easily add x as an additional input to the RNN that creates the y sequence when x is a fixed-size vector. Typical methods for adding more input to an RNN include :
  1. As an extra input at each time step or
  2. As the initial state $h^{(0)}$ or
  3. Both.
- Fig. 3.2.7 presents the first and most typical method. A newly inserted weight matrix R, which was missing from the model using simply the sequence of y values, parameterizes the interaction between each hidden unit vector and the input x. At each time step, the concealed units get an extra input of the same product .
- The value of x^TR, which serves as a new bias parameter for each of the hidden units, is essentially determined by the choice of x. The input has no impact on the weights. This model may be compared to changing the non-conditional model's parameters θ into ω, where the bias parameters within ω are now a function of the input.

- 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 equation (3.2.8) corresponds to a conditional distribution $P(y^{(1)}, \ldots, y^{(T)}|x^{(1)}, \ldots, x^{(T)})$ that makes a conditional independence assumption that this distribution factorizes as

$$\prod_t P(y^{(t)}|x^{(1)}, \ldots, x^{(t)}) \quad \ldots(3.2.28)$$

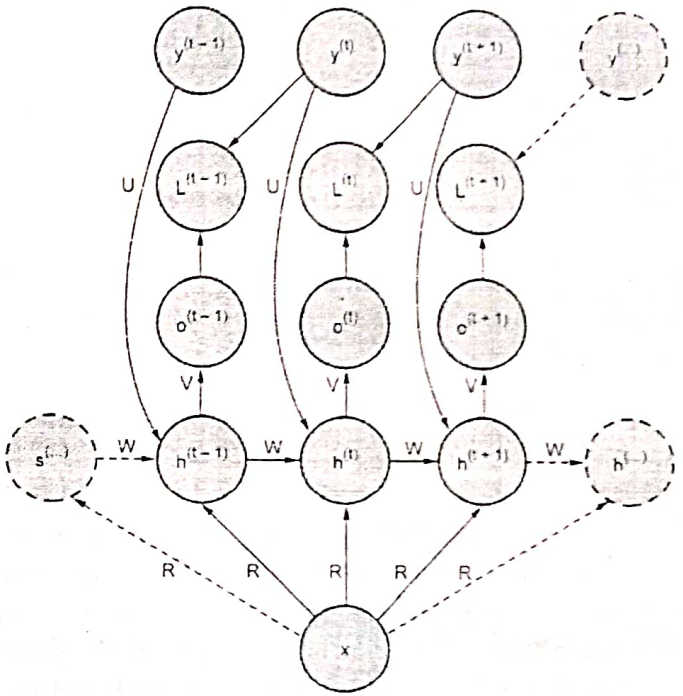

Fig. 3.2.7

- Fig. 3.2.7 an RNN that converts a vector of fixed length x into a distribution across sequences Y. This RNN is suitable for jobs like captioning images, where a single picture serves as the input to a model, which then outputs a string of words that describe the image. Each component of the observed output sequence, denoted by y(t), serves as both an input (for the current time step) and a goal during training (for the previous time step).

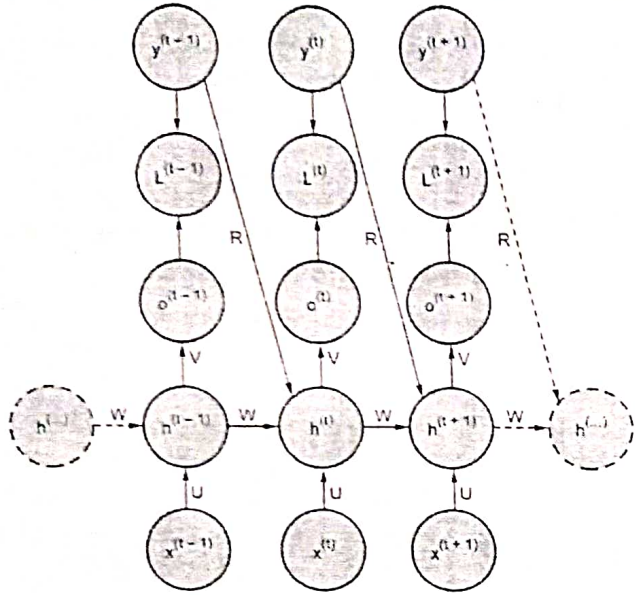

Fig. 3.2.8

- Fig. 3.2.8 a conditional recurrent neural network that converts a distribution over sequences of the same length of y values from a variable-length sequence of x values. This RNN has linkages from the prior output to the current state as opposed to Fig. 3.2.1, which does not. This RNN can simulate any distribution over sequences of y given sequences of x that have the same length thanks to these connections. Only distributions in which the y values are conditionally independent from one another given the x values may be represented by the RNN of Fig. 3.2.1.
- We can add connections from the output at time t to the hidden unit at time t+1 as illustrated in Fig. 3.2.8 to eliminate the conditional independence assumption. This allows the model to depict any given probability distribution across the y series. The length of both sequences must match for this type of model to accurately describe a distribution across a sequence given another sequence.

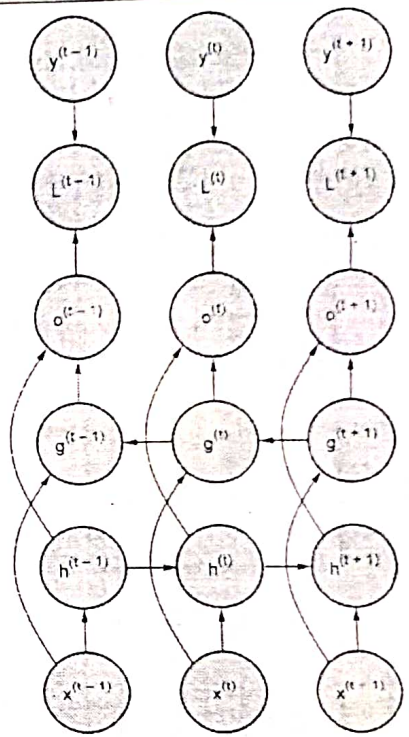

Fig. 3.2.9

- Fig. 3.2.9 : Computation of a typical bidirectional recurrent neural network with loss L(t) at each step t, designed to learn to map input sequences x to target sequences y. Information is propagated backward in time by the g recurrence and forward in time (to the right) by the h recurrence (towards the left). As a result, the output units o(t) can profit from a pertinent summation of the past in its h(t) input and a pertinent summation of the future in its g(t) input at each point t.

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.

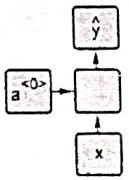

Fig. 3.3.1 One to One RNN

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.

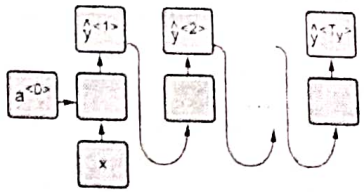

Fig. 3.3.2 One to Many RNN

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 the positive or negative sentiment.
- Fig. 3.3.3 shows the Many-to-One RNN.

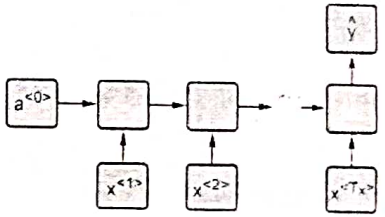

Fig. 3.3.3 Many to One RNN

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.

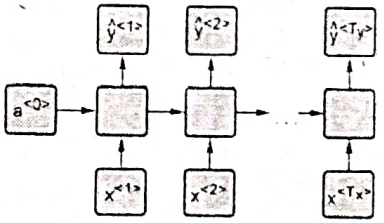

Fig. 3.3.4 Many to Many RNN

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.

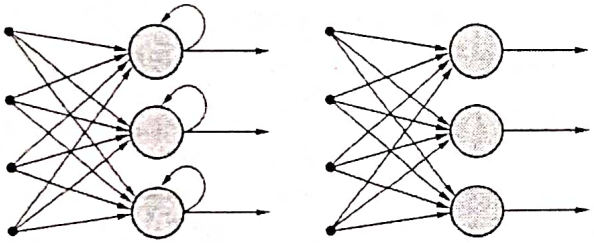

Fig. 3.4.1 Feedforward NN Vs RNN

- 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).

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.

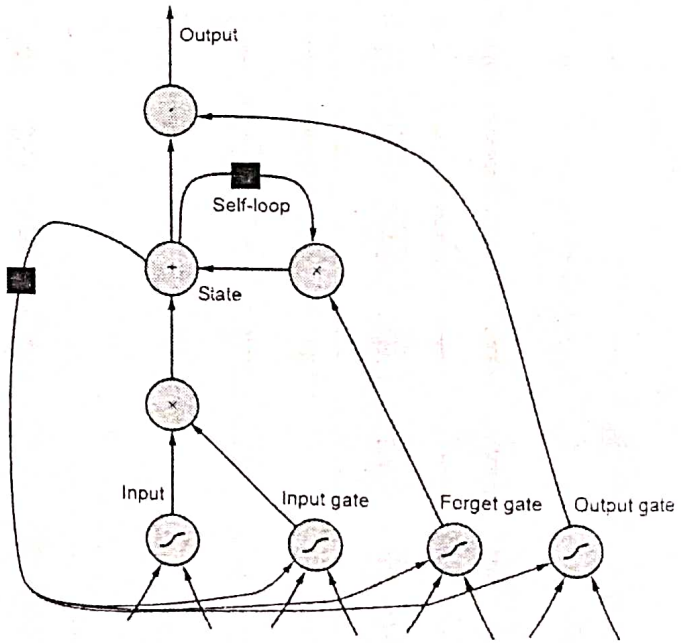

Fig. 3.5.1

- 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 :

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_i^{(t-1)}\right) \quad \ldots(3.5.1)$$

- 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 $f_i^{(t)}$:

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

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) :

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_i^{(t-1)}\right) \quad \ldots(3.5.3)$$

- 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 :

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

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

- 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.

3.6 Encoder Decoder Architectures

- Fig. 3.2.3 illustrates how an RNN may convert an input sequence into a fixed-size vector. Eig. 3.2.7 illustrates how an RNN may convert a fixed-size vector into a sequence. Fig. 3.2.1, 3.2.2, 3.2.8, and 3.2.9 show how an RNN may translate an input sequence into an output sequence of the same length.
- In this article, we'll go over how an RNN may be trained to translate input 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 = ($y^{(1)}$, ..., $y^{(ny)}$), 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.

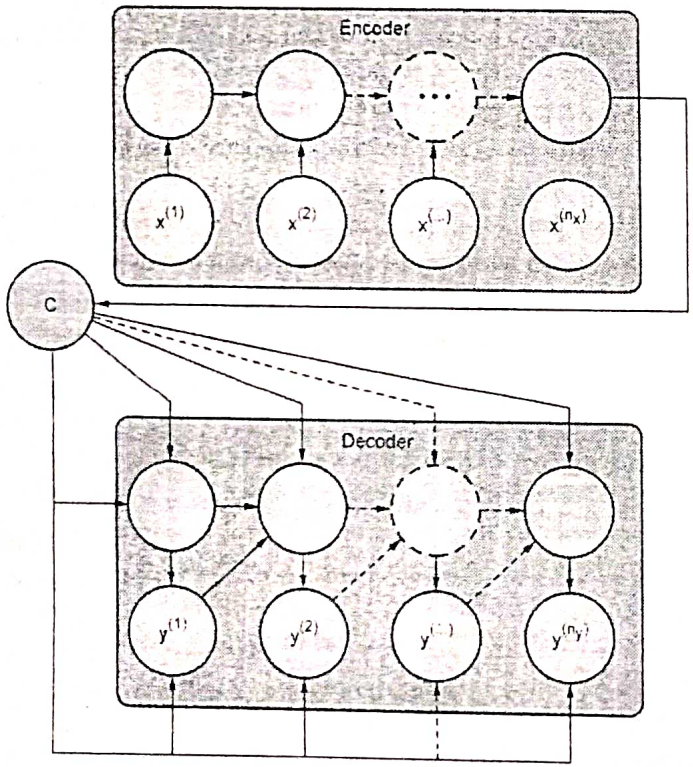

Fig. 3.6.1

- Fig. 3.6.1 An example of an encoder-decoder or sequence-to-sequence RNN architecture that can learn to produce an output sequence ($y^{(1)}$, ..., $y^{(ny)}$) from an input sequence ($x^{(1)}$, $x^{(2)}$, ..., $y^{(nx)}$). 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 $P(y^{(1)}, ..., y^{(ny)} | x^{(1)}, ..., x^{(nx)})$ 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 $h_{\tau x}$ 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.

3.7 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 $O(\log \tau)$ 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.

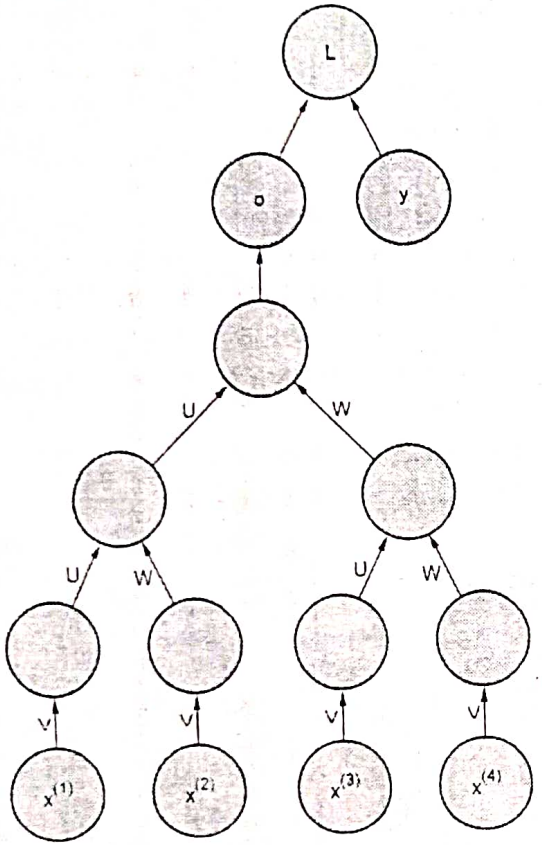

Fig. 3.7.1

- 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).

Review Questions

1. *Explain the concept of recurrent neural networks with example.*
2. *Explain different types of RNN.*
3. *Differentiate feedforward neural network Vs RNN.*
4. *Explain encoder decoder architectures.*
5. *Explain the concept and role of LSTM in RNN.*
6. *Explain recursive neural networks.*

❑❑❑

Unit IV

4 Autoencoders

Syllabus

Undercomplete Autoencoders, Regulraized Autoencoders - Sparse Autoencoders, Stochastic Encoders and Decoders, Denoising Autoencoders, Contractive Autoencoders, Applications of Autoencoders.

Contents

4.1 Autoencoders

- An autoencoder is unsupervised learning technique. It is a artificial neural network used to learn data encodings of unlabeled data or the task of representation learning.
- Autoencoders are a specific type of feedforward neural networks trained to copy its input to output. A bottleneck is imposed in the network to represent a compressed knowledge of the original input. The input is compressed into a lower-dimensional code and then the output is reconstructed from this representation. The code is also called as latent-space representation which is a compact "summary" or "compression" of the input.
- Fig. 4.1.1 shows three components of the autoencoder

  1. Encoder, 2. Code and 3. Decoder.

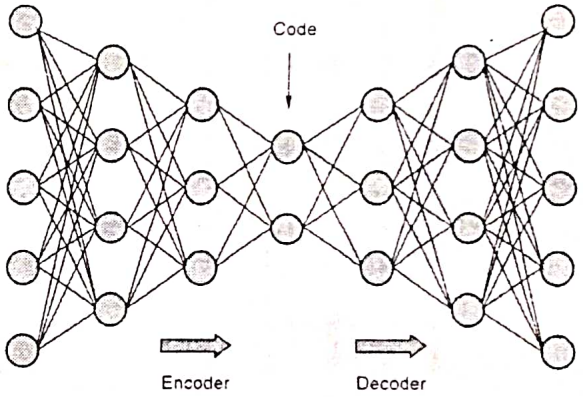

Fig. 4.1.1 Components of Autoencoder

The input is compressed by the encoder to produces the code. The decoder uses this code to reconstructs the input.

- Traditional use of autoencoders is dimensionality reduction or feature learning. Recently, they have been used in generative modeling also.
- As autoencoder is a special case of feedforward networks , training techniques similar to feedforward neural network such as minibatch gradient descent following gradients computed by back-propagation can be used for training.

- In contrast to feedforward network, autoencoders can also use recirculation learning algorithm. This algorithm is based on comparing the activations of the network on the original input to the activations on the reconstructed input.

4.1.1 Architecture of Autoencoder

- The architecture of autoencoder is shown in Fig. 4.1.2. It consists of a hidden layer 'h'. This hidden layer 'h' describes a code used to represent the input.
- The network has two parts :
  i) Encoder function h = f (x) and
  ii) A decoder that produces a reconstruction r = g (h).

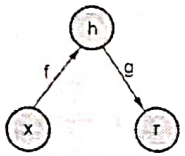

Fig. 4.1.2 Architecture of autoencoder

- The detail visualization of autoencoder is as shown in Fig. 4.1.3.

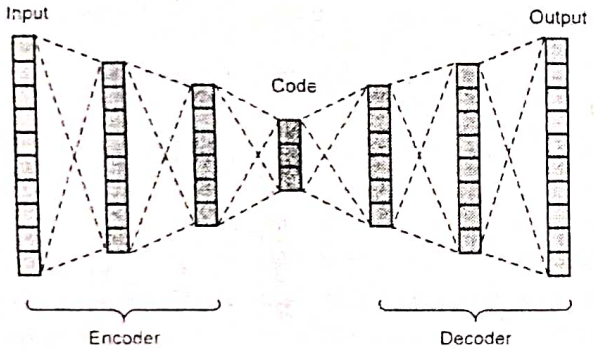

Fig. 4.1.3 Autoencoder structure

- Encoder and decoder both are fully-connected feedforward neural networks. The artificial neural network structure of decoder is same as that of encoder. Code is single layer artificial neural network having dimension of our choice. Code size i.e the number of nodes in code layer is a *hyperparameter* which need to be set before training the autoencoder.

- The input is passed through the encoder to produces the code. The decoder produces the output only using the code. Though it is not required, typically the architecture of the decoder is the mirror image of the encoder.
- The main objective is to get an output that is identical with the input. The dimensionality of the input and output must be same.
- There are four hyperparameters that must be set before training the autoencoders. They are as follows.
  1. **Code size :** It is the number of nodes in the middle layer. Smaller the size more is the compression.
  2. **Number of layers :** The autoencoder can be as deep as we like without considering the input and output. e.g the autoencoder shown in Fig. 4.1.2 have 2 layers each in both the encoder and decoder, without considering the input and output.
  3. **Number of nodes per layer :** The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the layer structure of decoder is symmetric to the encoder.
  4. **Loss function :** *Mean squared error (mse) or binary crossentropy* can be used as loss function. Crossentropy is used if the input values are in the range [0, 1] else mean squared error is used.
- The different types of autoencoders are as follows :
  1. Undercomplete autoencoders
  2. Sparse autoencoders
  3. Contractive autoencoders
  4. Denoising autoencoders
  5. Variational autoencoders (for generative modelling)

4.1.2 Undercomplete Autoencoders

- An autoencoder in which the dimension of the code is less than the dimension of the input is called undercomplete autoencoder. Fig. 4.1.4 shows structure of undercomplete autoencoder.Most salient features of the training data can be captured by learning an undercomplete representation.

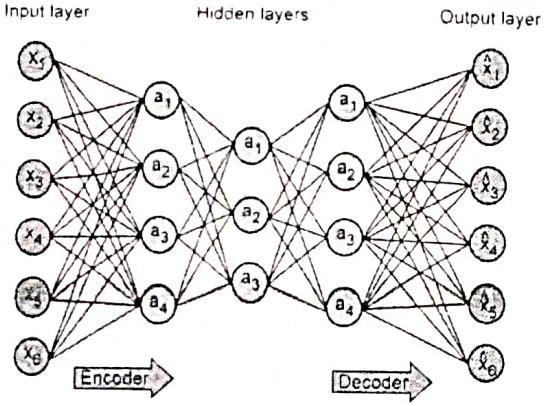

Fig. 4.1.4 Undercomplete autoencoder

- Undercomplete autoencoder limits the amount of information that can flow through the network by constraining the number of hidden layer(s) node.
- Ideally, this encoding will learn and describe latent attributes of the input data.
- The learning process is to minimizing a loss function given by equation (4.1.1)

$$L(x, g(f(x))) \tag{4.1.1}$$

- where L is a loss function. The loss function L penalize $g(f(x))$ so that it is different from x. e.g. (mse) mean squared error.
- When the decoder is linear and the mean squared error is used as loss function L, then an undercomplete autoencoder generates the same subspace as PCA. Here the autoencoder is trained for copying task but it also learns the principal subspace of the training data as a side effect.
- When encoder function f and decoder function g are non linear, autoencoders can learn more powerful nonlinear generalization of PCA.
- The difference between linear and nonlinear dimensionality reduction is shown in Fig. 4.1.5. As neural networks can learn nonlinear relationship, autoencoders can be considered as more powerful nonlinear generalization of PCA. PCA discovers a lower dimensional hyper plane that describes the original data. On the other hand autoencoders are able to learn nonlinear manifolds (a manifold is a continuous, non-intersecting surface).

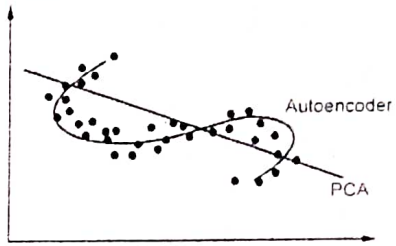

Fig. 4.1.5 Linear vs nonlinear Dimensionality reduction

- If encoder and decoder has too much capacity then such autoencoder will not extract any useful information about data distribution while it learns the copying task. By using powerful nonlinear encoder, each training example $x^{(i)}$ can be represented using code i where code is only one dimensional. Decoder learns to map these integer indices back to the values of specific training examples.
- This is the reason why an autoencoder trained to do the copying task cannot learn anything useful about the dataset if the capacity of the autoencoder becomes too great.

4.2 Regulraized Autoencoders

- In undercomplete autoencoders the code dimension is less than the input dimension and they learn to capture most salient features of the data distribution. These autoencoders fails in learning anything useful about the dataset if the capacity of the encoder and decoder is too great.
- A similar problem occurs if the hidden code has dimension equal to the input and also in case of overcomplete autoencoder in which the hidden code has dimension greater than the input.
- In these cases, even a linear encoder and a linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.
- Any architecture of autoencoder should be able to trained successfully by choosing the dimension of code and the capacity of the encoder and decoder based on the complexity of distribution to be modeled.
- Regularized autoencoders provides this ability. Instead of limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties in addition to the ability to copy its input to its output.

- These other properties include sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or to missing inputs.
- A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn a trivial identity function.

4.2.1 Sparse Autoencoders

- Autoencoders can learn useful features in two ways,
  i) By keeping the code size small and
  ii) Denoising autoencoders.
- One more method that can be used is regularization. Autoencoder can be regularized by using a sparsity constraint. Sparsity constraint is introduced on the hidden layer such that only a fraction of the nodes would have nonzero values, called active nodes. So only a reduced number of hidden nodes are used at a time. Fig. 4.2.1 shows sparse autoencoder where the opacity of the node denotes activation level of node.

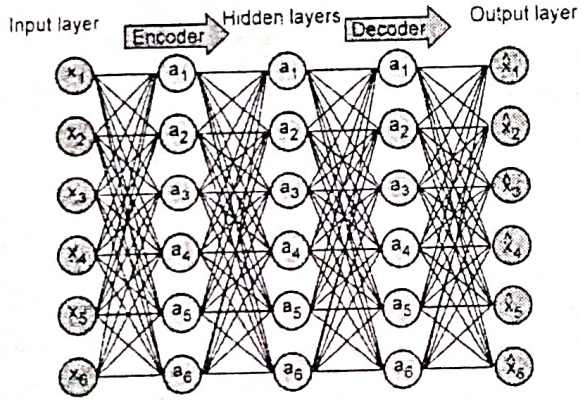

Fig. 4.2.1 Sparse autoencoders

- A penalty term is added to the loss function such that only a fraction of the nodes become active. So the autoencoder is forced to represent each input as a combination of small number of nodes, and to discover salient features in the data.
- The value of Sparsity penalty, $\Omega(h)$ is close to zero but not zero. In addition to the reconstruction error, sparsity penalty is applied on the hidden layer which prevents overfitting.

$$L(x, g(f(x))) + \Omega(h) \quad \ldots(4.2.1)$$

where $g(h)$ is the decoder output, and typically we have $h = f(x)$, the encoder output.

- In sparse autoencoders hidden nodes are greater than input nodes. As only a small subset of the nodes will be active at any time this method works even if the code size is large.
- Sparse autoencoders are typically used to learn features for another task, such as classification.
- Instead of serving as just an identity function the sparse autoencoder should respond to unique statistical features of the dataset.
- The model yield by training the autoencoder to perform the copying task with a sparsity penalty learns the useful features as by product. The penalty Ω(h) can be simply treated as a regularizer term added to a feedforward network whose primary task is to copy the input to the output (unsupervised learning objective). This network can also perform some supervised task (with a supervised learning objective) that depends on these sparse features.

4.2.2 Denoising Autoencoders

- Autoencoder can learn useful representations by changing the reconstruction error term of the cost function rather than adding a penalty Ω to the cost function.
- In contrast to traditional autoencoder that minimize some loss function as given in equation (4.2.2) the denoising autoencoder (DAE) minimizes the loss function given by equation (4.2.3)

$$L(x, g(f(x))) \quad \ldots(4.2.2)$$

  - *L is a loss function that penalize $g(f(x))$ for being dissimilar from x, e.g. L^2 norm of their difference.*

$$L(x, g(f(\tilde{x}))) \quad \ldots(4.2.3)$$

  - *$\tilde{x}$ is a corrupted copy of x by adding some form of noise.*
- This helps to avoid the autoencoders to copy the input to the output without learning features about the data.
- Denoising autoencoders thus provide yet another example of how useful properties can emerge as a byproduct of minimizing reconstruction error.
- They are also an example of how overcomplete, high-capacity models may be used as autoencoders as long as care is taken to prevent them from learning the identity function.

4.2.3 Regularizing by Penalizing Derivatives

- Another way of regularizing an autoencoder is by using a penalty Ω, similar to sparse autoencoders

$$L(x, g(f(x))) + \Omega(h, x) \quad \ldots (4.2.4)$$

but with a different form of Ω :

$$\Omega(h, x) = \lambda \sum_i \| \nabla_x h_i \|^2 \quad \ldots (4.2.5)$$

- Because of this the model learns a function which will not change much when there is slight change in x.
- Also as the penalty is applied only at training examples, the autoencoder is forced to learn features about the structure of training distribution. An autoencoder regularized in this way is called a contractive autoencoder, or CAE.

4.3 Stochastic Encoders and Decoders

- Autoencoders are feedforward networks and use the same loss functions and output unit that are used in traditional feedforward networks
- For designing the output units and the loss function of a feedforward network, an output distribution $p(y \mid x)$ is defined and the negative log-likelihood $-\log p(y \mid x)$ is minimized where y is a vector of targets, e.g. class labels .
- But in an autoencoder, target as well as the input is x and still the same strategy can be applied. So by using same strategy as in feedforward network, we can assume that for a given code h, decoder is providing a conditional distribution $p_{decoder}(x \mid h)$. Autoencoder can then be trained by minimizing $-\log p_{decoder}(x \mid h)$ where the exact form of the loss function depends on the form of $p_{decoder}$.
- Similar to traditional feedforward networks, linear output units are used to parameterize the mean of a Gaussian distribution for real valued x. The negative log-likelihood yields a mean squared error criterion in this case. Binary x values correspond to a Bernoulli distribution whose parameters are given by a sigmoid output unit, discrete x values correspond to a softmax distribution and so on.
- Given h, the output variables are treated as conditionally independent so that evaluation of probability distribution is inexpensive. For modeling outputs with correlations, mixture density outputs can be used.
- Encoding function f(x) can be generalized to an encoding distribution $p_{encoder}(h \mid x)$, as shown in Fig. 4.3.1. Fig. 4.3.1 shows the structure of stochastic autoencoder. Here both encoder and decoder involve some noise injection. The output of encoder

and decoder can be seen as sampled from a distribution, $p_{encoder}(h \mid x)$ for encoder and $p_{decoder}(x \mid h)$ for the decoder.

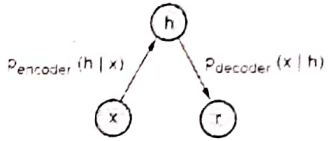

Fig. 4.3.1 The structure of a stochastic autoencoder

- Any latent variable model $p_{model}(h, x)$ defines a stochastic encoder

$$p_{encoder}(h \mid x) = p_{model}(h \mid x) \quad \ldots (4.2.6)$$

and a stochastic decoder

$$p_{decoder}(x \mid h) = p_{model}(x \mid h) \quad \ldots (4.2.7)$$

- In general, the encoder and decoder distributions need not be necessarily conditional distributions compatible with a unique joint distribution $p_{model}(x, h)$.

4.4 Denoising Autoencoders

- Fig. 4.4.1 shows denoising autoencoder.

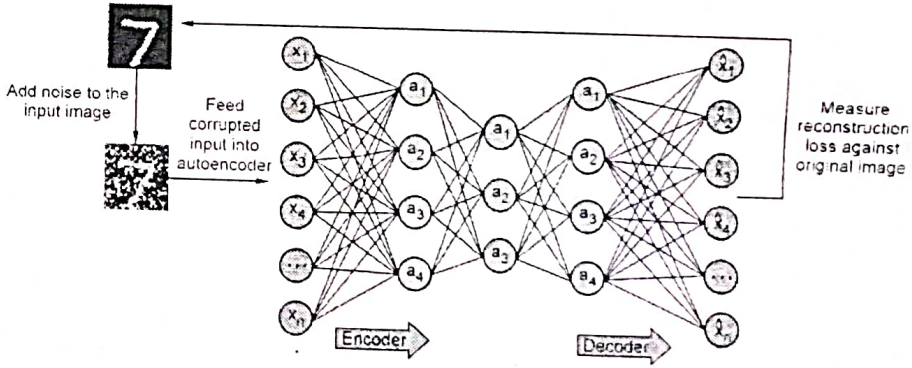

Fig. 4.4.1 Denoising autoencoder

- The denoising autoencoder (DAE) receives a corrupted data point as input. It is trained to predict the uncorrupted original, data point as the output.
- Fig. 4.4.2 illustrates the training procedure of DAE. A corruption process $C(\tilde{x} \mid x)$ is introduced. It represents a conditional distribution over corrupted samples $\tilde{x}$, given a data sample x.

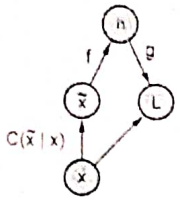

Fig. 4.4.2 The computational graph of the cost function for a denoising autoencoder

- The autoencoder then learns a reconstruction distribution $p_{reconstruct}(x|\tilde{x})$ estimated from training pairs $(x,\tilde{x})$ as follows:
  1. Sample a training example x from the training data.
  2. Sample a corrupted version $\tilde{x}$ from $C(\tilde{x} \mid x = x)$.
  3. Use $(x,\tilde{x})$ as a training example for estimating the autoencoder reconstruction distribution $p_{reconstruct}(x|\tilde{x}) = p_{decoder}(x|h)$ with h the output of encoder $f(\tilde{x})$ and $p_{decoder}$ typically defined by a decoder g(h).
- Gradient-based approximate minimization (such as minibatch gradient descent) can be performed on the negative log-likelihood $-\log p_{decoder}(x \mid h)$.
- As long as the encoder is deterministic, the denoising autoencoder is a feedforward network. So it can be trained using exactly the same techniques as that of any other feedforward network. DAE performs stochastic gradient descent on the following expectation:-

$$- E_{\tilde{x}\, \hat{p}_{data}(x)} E_{\tilde{x} \sim C(\tilde{x}|x)} \log p_{decoder}(x \mid h = f(\tilde{x})) \quad \ldots(4.2.8)$$

where $\hat{p}_{data}(x)$ is the training distribution.

4.4.1 Estimating the Score

- Score matching provides estimator of probability distributions so that the model can have the same score as the data distribution at every training point x. It is as an alternative to maximum likelihood.
- The score is a particular gradient field given by :

$$\nabla_x \log p(x) \quad \ldots(4.2.9)$$

- Learning the gradient field of $\log p_{data}$ is one way to learn the structure of p_{data} itself.

- Score matching works by fitting the slope (score) of the model density to the slope of the true underlying density at the data points
- DAE, with conditionally Gaussian $p(x|h)$, makes the autoencoder learn a vector field $(g(f(x) - x)$ that estimates this score of the data distribution. The DAE is trained to minimize $\| g(f(\tilde{x}) - x) \|^2$
- Fig. 4.4.3 illustrates how DAE estimates a vector field.

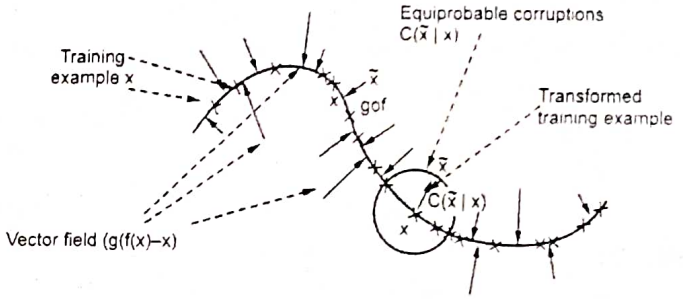

Fig. 4.4.3 Vector field learned by DAE

- A denoising autoencoder is trained to map a corrupted data point $\tilde{x}$ back to the original data point x.
- Training examples x are indicated by crosses. Training examples x lie on a low-dimensional manifold which is indicated by bold black line.
- Circle indicates equiprobable corruptions $C(\tilde{x} \mid x)$
- Solid arrows indicates the vector field $(g(f(x) - x)$.This vector field estimates the score $\nabla_x \log p_{data}(x)$ which is the slope of the density of data.
- A arrow inside the circle show one training example is transformed into one sample from this corruption process.
- Fig. 4.4.4 shows the vector field learned by a denoising autoencoder around a 1-D curved manifold near which the data concentrate in a 2-D space.
- Each arrow is proportional to reconstruction minus input vector of DAE and points towards higher probability
- Length of the arrow shows the norm of reconstruction error. When the length of the arrow i.e norm of reconstruction error is large, probability can be significantly increased by moving in the direction of the arrow. Mostly these are the places of low probability. These low probability points are mapped to higher probability reconstructions by the autoencoder.

- Arrows shrinks where probability is maximum because the reconstruction becomes more accurate.

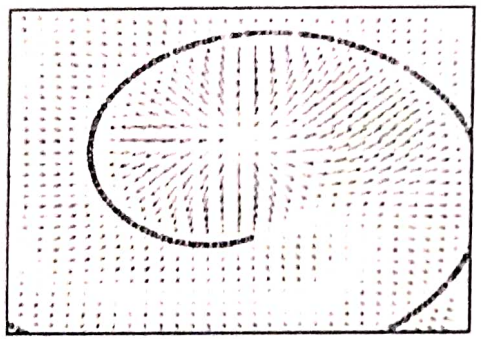

Fig. 4.4.4 Vector field learned by a denoising autoencoder around a 1-D curved manifold

4.5 Contractive Autoencoders

- The main goal of Contractive Autoencoder (CAE) is to have a robust learned representation that is less sensitive to small variation in the data.
- A penalty term is applied to the loss function so as to make the representation robust.
- In order to make the derivatives of f to be as small as possible, contractive autoencoder introduces an explicit regularizer on the code h = f(x)
- The penalty term is Frobenius norm of the Jacobian matrix which is calculated with respect to input for the hidden layer. Frobenius norm of the Jacobian matrix is the sum of square of all elements. This is shown in Fig. 4.5.1.

$$L = \| x - g(f(x)) \| + \lambda \| J_f(x) \|_F^2$$

$$\| J_f(x) \|_F^2 = \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2$$

Fig. 4.5.1 Loss function with penalty term - Frobenius norm of the Jacobian matrix

- Contractive autoencoder is similar to denoising autoencoder in a sense that in presence of small Gaussian noise the denoising reconstruction error is equivalent to a contractive penalty on the reconstruction function that maps x to r = g(f(x)).

- This means, in case of denoising autoencoders the reconstruction function resist small but finite sized perturbations of the input, while in contractive autoencoders the feature extraction function resist infinitesimal perturbations of the input.
- CAE surpasses results obtained by regularizing autoencoder using weight decay or by denoising. CAE is a better as compare to denoising autoencoder to learn useful feature extraction.
- The model is encouraged to learn an encoding where similar inputs have similar encodings. So the model is forced to learn how to **contract** a neighborhood of inputs into a smaller neighborhood of outputs.
- Fig. 4.5.2 indicates how the derivative of the reconstructed data (i.e slope) is essentially zero for local neighborhoods of input data.

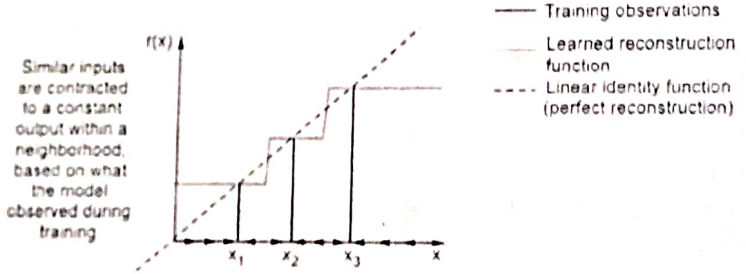

Fig. 4.5.2 Slope of the reconstructed data

- This can be achieved by penalizing the instances where a small change in the input leads to a large change in the encoding space. For this the loss term should penalizes large *derivatives* of *hidden* layer activations w.r.t. the input training instances.
- Regularization loss term used is the squared Frobenius norm $\|A\|_F$ of the Jacobian matrix J for the hidden layer activations w.r.t. the input observations. A Frobenius norm is an L_2 norm for a matrix. The Jacobian matrix represents all first-order partial derivatives of a vector valued function.
- These values can be calculated using equation (4.5.1) and (4.5.2). where

  m : Observations

  n : Hidden layer nodes, and

Loss function can be defined by equation (4.5.3)

$$\| A \|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{i,j}|^2} \quad \ldots (4.5.1)$$

$$J = \begin{bmatrix} \frac{\delta a_1^{(h)}(x)}{\delta x_i} & \cdots & \frac{\delta a_1^{(h)}(x)}{\delta x_m} \\ \vdots & \ddots & \vdots \\ \frac{\delta a_n^{(h)}(x)}{\delta x_i} & \cdots & \frac{\delta a_n^{(h)}(x)}{\delta x_m} \end{bmatrix} \quad \ldots (4.5.2)$$

$$L(x, \hat{x} + \lambda \sum_i || \nabla_x a_i^{(h)}(x) ||^2 \quad \ldots (4.5.3)$$

where

$\nabla_x a_i^{(h)}(x)$: Gradient field of hidden layer activations w.r.t. the input x, summed over all i training examples.

4.6 Applications of Autoencoders

Following are the most common applications of the autoencoders :

1. . Dimensionality reduction
2. Image denoising
3. Generation of image and time series data
4. Data compression
5. Feature extraction
6. Image colourisation
7. Anomaly detection
8. Data denoising (ex. images, audio)
9. Image inpainting
10. Information retrieval
11. Sequence to sequence prediction
12. Recommendation system

1. Dimensionality reduction

- Undercomplete autoencoders can be used as preprocessing step for dimensionality reduction.
- In undercomplete autoencoder the size of hidden layer is smaller than input layer. Network is forced to learn important features by reducing the size of hidden layer. The reduced representation of input data is stored in middle layer of the autoencoder called as code. So at the code layer of the autoencoder, compressed form of information from the input is present. By extracting this layer, each node of

this layer can be treated as a feature variable or a dimension. Thus the autoencoder can be used for dimensionality reduction by deleting the decoder and taking the output at code layer.

- Network may not learn anything useful if it is deep and highly nonlinaer. As the number of layers in an autoencoder increases, the hidden layer size will have to be decreased. There will be loss of information if the size of the hidden layer becomes smaller than the intrinsic dimension of the data.
- Autoencoders are more powerful as compare to PCA in dimensionality reduction because PCA can only perform linear dimensionality reductions. Whereas undercomplete autoencoders can perform large-scale non-linear dimensionality reductions.

2. Image denoising

- Image denoising is performed to obtain the proper information about the content of an image. Denoising autoencoders can be used for this.
- Denoising autoencoders do not search for noise in image, instead they extract the image from noisy data fed as input to them by learning their representations. Then the noise free image is formed by decompressing these representations
- Denoising autoencoders can perform efficient and highly accurate image denoising and can be used for denoising complex images that could not be denoised using traditional methods.

3 . Image and time series data generation

- Variational Autoencoder (VAE) is a Generative Model. It can be used to generate the images that are yet unseen by the model. This is useful in applications like
  - New animated character generation
  - Fake human images generation.
- Discrete values for latent attributes can be generated by sampling the parameterized distribution at the bottleneck of the autoencoder. Then it can be forwarded to decoder that leads to image generation.
- Variational Autoencoders (VAE) are also useful in modeling time series data like music.

4 . Data compression

- Autoencoders are basically designed for data compression. But in practical situations they are hardly used for data compression because of lossy compression and data specific nature.

a. **Lossy compression :** The output of the autoencoder is close to the input and not exactly same as input. It is a degraded representation. So not suitable for lossless compression.

b. **Data specific :** Autoencoders learns the representation specific to the given training data. Because of this they can meaningfully compress the data which is similar to the training data used for training the autoencoder. Hence, an autoencoder trained on handwritten digits may not able to compress landscape photos.

5. Feature extraction

- Autoencoder consists of encoder, code and decoder.
- The encoder learns the important hidden features of the input data while reducing the reconstruction error.
- During this encoding process new subset of features is formed which contains unique combinations of original features.

6. Image colourisation

- Autoencoders are useful for converting colour image to grayscale image and also for converting black and white image to colour image

7. Anomaly detection

- A neural network trained with specific dataset learns the data it commonly "sees" and represents the input dataset. This network can also represent the difference between input and output and tell us when it "sees" unusual data.
- Autoencoders can be used in such systems where it is difficult to describe unusual or anomalous data. Undercomplete autoencoders are used for anomaly detection.
- If autoencoder is trained on specific image dataset say "D", then it is supposed to reconstruct the image as it is with low reconstruction loss.
- But autoencoder cannot perform reconstruction task for image which is not present in the training dataset because latent attributes will not be adapted by network for any unseen image. As a result the reconstruction loss for such image will be very high. So by setting appropriate threshold it can be easily identified as anomaly or unusual image.
- Due to this autoencoders are good at powering anomaly detection systems.

8. Information retrieval

- Information retrieval is the task of searching the entries in a database that resemble a query entry.
- Dimensionality reduction benefits the task of information retrieval. In case of certain type of low dimensional data search can become more efficient due to dimensionality reduction. As one of the application of autoencoder is dimensionality reduction, they can be applied to information retrieval using semantic hashing.
- By training the dimensionality reduction algorithm to produce low dimensional and binary code , all database entries can be stored in a hash table that maps binary code vectors to entries.
- Using this hash table information retrieval can be performed by returning all database entries having same binary code as the query. By flipping some bits from encoding of the query, slightly less similar entries can also be searched very efficiently.
- This approach of information retrieval is suitable for both textual and image data.
- Typically encoding function with sigmoids is used on final layer for producing binary codes for semantic hashing. The sigmoid unit must be saturated to nearly 0 or nearly 1. For this additive noise is injected before sigmoid function during training and its magnitude should increase over time. So in order to preserve as much information as possible against this noise , the network must increase the magnitude of the inputs to the sigmoid function, until saturation occurs.

9. Sequence to sequence prediction

- LSTMs-based autoencoders, can capture temporal structure.
- Such autoencoders are suitable for Machine translation applications such as,
  - Fake video generation
  - Next video frame prediction.

10. Recommendation systems

- Deep autoencoders can be employed to learn user choices to recommend movies, books, music or other items. The concept is as below. Consider the example of YouTube
- Clusters of the users based on similar interest is given as the input data.

- Videos watched, watch time for each video, commenting with videos etc denotes the user interests. This data is captured by clustering content.
- The interests of the user will be captured by Encoder part of the autoencoder.
- Decoder part of the autoencoder projects on two parts :
  - New content from content creators
  - Existing unseen content.

Review Questions

1. *Explain architecture of autoencoder with neat diagram.*
2. *Explain the hyperparameters that must be set before training of autoencoders.*
3. *List the different types of autoencoders.*
4. *What is undercomplete autoencoder ? Explain its application ?*
5. *What is the drawback of undercomplete autoencoder ? How it can be overcomed using regularized autoencoder ?*
6. *What happens in case of undercomplete autoencoder, if the capacity of encoder and decoder is too great ?*
7. *Write note on Sparse autoencoder.*
8. *State and explain different ways of regularizing autoencoders.*
9. *Describe stochastic autoencoder.*
10. *How the denoising autoencoder estimates the score and learns the vector field ?*
11. *Explain the concept of contractive autoencoder.*
12. *State applications of autoencoders and explain any two of them.*
13. *Why undercomplete autoencoders are suitable for dimensionality reduction.*
14. *Which type of autoencoder is suitable for image denoising ? Why ?*
15. *Though autoencoders are designed for data compression they are hardly used for it. Explain.*
16. *Why autoencoders are good at anomaly detection ?*
17. *Explain how the dimensionality reduction feature of autoencoder is useful in information retrieval task.*

❑❑❑

Notes

Unit V

5 Representation Learning

Syllabus

Greedy Layerwise Pre-training, Transfer Learning and Domain Adaption, Distributed Representation, Variants of CNN : DenseNet.

Contents

5.1 Introduction to Representation Learning

- Depending on how the information is represented, many jobs involving the processing of information can be either extremely simple or highly complex. This is an overarching idea that applies to both machine learning and general computer science. To divide 210 by 6, for instance, one can do it easily using long division. If the job is instead asked using the Roman numeral representation of the numbers, it becomes much more difficult.
- When asked to divide CCX by VI, most contemporary individuals would start by changing the numbers' representation to Arabic numerals, which allows for lengthy division operations that utilize the place value system. More specifically, by employing suitable or unsuitable representations, we may measure the asymptotic runtime of particular operations.
- For instance, if the list is represented as a linked list, inserting a number into the proper place in a sorted list of numbers is an O(n) operation, but only O(log n) if the list is represented as a red-black tree.
- What distinguishes one representation from another in machine learning? In general, an effective representation is one that facilitates subsequent learning tasks. Typically, the selection of the ensuing learning problem will determine the representation to be used.
- Feedforward networks that have been taught through supervised learning are capable of learning representations. A linear classifier, such as a softmax regression classifier, is often used as the network's last layer. The remainder of the network gains knowledge to provide this classifier a representation. Every hidden layer (but especially those close to the top hidden layer) naturally acquires features that facilitate the classification job after training with a supervised criteria.
- For instance, classes that were not linearly separable in the input characteristics may change in the final hidden layer to become linearly separable. The last layer may theoretically be another type of model, such a closest neighbor classifier. Depending on the nature of the final layer, the penultimate layer's features should learn various qualities.
- It is not necessary to explicitly impose any conditions on the learnt intermediate features during supervised training of feedforward networks. Many times, other types of representation learning algorithms are expressly created to form the representation in a specific manner. Think about learning a representation that

makes density estimation simpler, for instance. We may create an objective function that promotes the components of the representation vector h to be independent since distributions with more independences are simpler to describe.

- Unsupervised deep learning algorithms, like supervised networks, have a primary training goal but also discover a representation incidentally. No matter how a representation was created, it may be applied to different tasks. Alternately, a common internal representation can be used to learn a variety of tasks (some supervised, some unsupervised).
- In most representation learning situations, there is a trade-off between maintaining as much of the input's information as feasible and obtaining desirable features (such as independence).
- Since representation learning offers a method for doing unsupervised and semi-supervised learning, it is particularly intriguing. We frequently have a lot more training data that isn't labelled than training data that is. Severe overfitting frequently occurs when supervised learning techniques are used to train on the labelled subset.
- By including learning from unlabeled data, semi-supervised learning provides a way to overcome the overfitting issue. To tackle the supervised learning challenge, we can develop effective representations for the unlabeled data and apply those representations.
- Very few labelled instances are sufficient for both humans and animals to learn from. We are still unsure of how this is possible. Improvements in human performance might be attributed to a variety of causes, such as the brain's use of Bayesian inference methods or very large ensembles of classifiers.
- The idea that the brain might benefit from unsupervised or semi-supervised learning is one that is frequently put out. Unlabeled data may be used in many different ways. In this chapter, we concentrate on the claim that a decent representation may be learned from unlabeled data.

5.2 Greedy Layer - Wise Unsupervised Pretraining

- Deep neural networks were revived in part because to unsupervised learning, which for the first time eliminated the need for architectural specialties like convolution or recurrence while training a deep supervised network. We refer to this process as greedy layer-wise unsupervised pretraining, or unsupervised pretraining without supervision.

- This process is a classic illustration of how a representation learnt for one job unsupervised learning, which aims to capture the structure of the input distribution can occasionally be helpful for another task (supervised learning with the same input domain).
- A single-layer representation learning technique, such as an RBM, a single-layer autoencoder, a sparse coding model, or any model that learns latent representations, provides the foundation of greedy layer-wise unsupervised pretraining.
- Each layer is pretrained using unsupervised learning, taking the output of the previous layer as input and producing as an output a new representation of the data, whose distribution (or its relation to other variables such as categories to predict) is produced in the hopes of being more straightforward.
- See Algorithm for 5.1 a formal description.
- The challenge of concurrently training the layers of a deep neural network for a supervised job has long been avoided by greedy layer-wise training approaches based on unsupervised criteria. This strategy has been around at least since the Neocognitron.
- The discovery that this greedy learning procedure could be used to find a good initialization for a joint learning procedure over all the layers and that this method could be used to successfully train even fully connected architectures marked the beginning of the deep learning renaissance in 2006.
- Only convolutional deep networks or networks with depth due to recurrence were thought to be trainable until this discovery. Although the unsupervised pretraining method was the first to be successful, we now know that greedy layer-wise pretraining is not necessary to train fully linked deep architectures.
- The reason greedy layer-wise pretraining is so named is because it uses a greedy algorithm, which individually optimizes each component of the solution rather than doing it collectively. The reason it is named layer-wise is because these separate components serve as the network's layers. More specifically, greedy layer-wise pretraining trains the k-th layer while fixing the preceding layers one at a time.
- In particular, with the introduction of the top layers, the lower layers which are taught first do not adjust. Because each layer is taught using an unsupervised representation learning technique, it is known as unsupervised learning. The fact that it is intended to be simply the initial step before a combined training algorithm is used to fine-tune all the layers simultaneously is why it is also known as

pretraining. It may be seen as a regularizer and a type of parameter initialization in the context of a supervised learning task (in certain trials, pretraining reduces test error without reducing training error).

- Pretraining is a term that is frequently used to describe both the pretraining phase and the full two phase technique, which incorporates the pretraining phase and a supervised learning phase. The supervised learning step may require fine-tuning the whole network that was trained in the pretraining phase, or it may involve training a straightforward classifier on top of the characteristics discovered during pretraining.
- The general training approach is typically very similar regardless of the sort of unsupervised learning algorithm or model that is used. Although the unsupervised learning algorithm used will undoubtedly affect the specifics, this fundamental procedure is followed by the majority of unsupervised pretraining applications.
- Other unsupervised learning techniques, such deep autoencoders and probabilistic models with several layers of latent variables, can be initialized using greedy layer-wise unsupervised pretraining as well. Deep belief networks and deep Boltzmann machines are two examples of these models.
- Pretraining with greedy layer-wise supervision is another option. This is based on the idea that it is simpler to train a shallow network than a deep one, which appears to have gained support in a number of situations.

5.2.1 When and Why Does Unsupervised Pretraining Work ?

- Greedy layer-wise unsupervised pretraining can significantly reduce test error for a variety of tasks, including classification tasks. Beginning in 2006, this finding led to a resurgence of interest in deep neural.
- **Algorithm 5.1 :**
  - Given the following: Unsupervised feature learning method L produces an encoder or feature function f from a training set of instances. With one row per example, the raw input data is X, and the dataset utilized by the second level unsupervised feature learner is $f^{(1)}(X)$, which is the output of the first stage encoder on X. In the instance of supervised fine-tuning, we employ a learner T that accepts an initial function f, input examples X and related targets and outputs a tuned function Y. There are m phases in all.

f ← Identity function

$\tilde{X} = X$

for k = 1, ..., m **do**

$f^{(k)} = L(\tilde{X})$

$f \leftarrow f^{(k)} \circ f$

$\tilde{X} \leftarrow f^{(k)} \tilde{X}$

end for

If fine-tuning **then**

f ← T(f, X, Y)

end if

Return f

- Unsupervised pretraining, however, either does not give an advantage or even clearly harms many other activities how pretraining affected machine learning models for predicting chemical activity and discovered that, overall, pretraining was marginally damaging but highly beneficial for certain tasks. Unsupervised pretraining can occasionally be beneficial but is frequently damaging, therefore it's crucial to understand when and why it works in order to decide if it's appropriate for a certain activity.
- It is crucial to state upfront that the majority of this debate is focused on certain greedy unsupervised pretraining. For using neural networks to do semi-supervised learning, there are alternative, entirely separate paradigms. Along with the supervised model, an autoencoder or generative model may also be trained. The discriminative RBM and the ladder network are examples of this single-stage technique, where the overall target is a clear sum of the two terms (one using the labels and one only using the input).
- Pretraining without supervision combines two distinct concepts. First, it utilises the notion that the model may be significantly regularised depending on the initial parameters chosen for a deep neural network (and, to a lesser extent, that it can improve optimization). Second, it takes advantage of the more general notion that comprehension of the input distribution can aid in comprehension of the relationship between inputs and outcomes.
- Both of these concepts entail several intricate interactions between numerous unrecognized components of the machine learning process.
- The first hypothesis - that a deep neural network's initial settings can strongly regularise its performance - is the one that is least known. When pretraining first

gained popularity, it was thought of as initialising the model at a place where it would be more likely to reach one local minimum than another.

- Local minima are no longer seen as a significant obstacle to neural network optimization. We now understand that there is typically no critical point encountered during our typical neural network training operations. It is still possible that pretraining initializes the model in an area that would otherwise be inaccessible, such as a region surrounded by regions where the cost function varies significantly between examples and minibatches only provide noisy estimates of the gradient, or a region surrounded by regions where the Hessian matrix is so poorly conditioned that gradient descent methods must use very small steps.
- However, it is difficult for us to pinpoint precisely which properties of the pretrained parameters are kept after the supervised training step. This is one of the reasons why contemporary methods frequently combine supervised and unsupervised learning simultaneously rather than in two stages. By simply freezing the parameters for the feature extractors and using supervised learning to only add a classifier on top of the learned features, one can avoid struggling with these difficult concepts about how optimization in the supervised learning stage preserves information from the unsupervised learning stage.
- The alternative hypothesis, according to which a learning system can employ data obtained during the unsupervised phase to improve performance during the supervised learning stage, is more clearly known. The fundamental tenet is that some traits may be advantageous for both the supervised learning task and the unsupervised one. For instance, if we train a generative model using photos of automobiles and motorbikes, the model will need to understand what wheels are and how many wheels should be present in an image.
- If we're lucky, the wheel representation will take on a shape that the supervised learner can easily access. It is not always feasible to forecast which activities would benefit from unsupervised learning in this way because this is still not fully understood mathematically and theoretically. The specific models employed greatly influence a number of facets of this technique.
- For instance, the features must enable linear class separation between the underlying classes if we want to layer a linear classifier on top of pretrained features. These characteristics are frequently seen in nature, but not always. Another reason why concurrent supervised and unsupervised learning may be preferred is because the limitations imposed by the output layer are already there.

- We may anticipate that unsupervised pretraining will be more successful when the original representation is weak from the perspective of learning a representation through unsupervised pretraining. The usage of word embeddings is a prime illustration of this. Because every two different one-hot vectors are the same distance from one another (squared L2 distance of), words represented by one-hot vectors are not particularly informative. The distance between words in learned word embeddings naturally encodes the similarity 2 between them. Unsupervised pretraining is therefore particularly helpful for word processing. When processing pictures, it is less helpful, probably because distances only offer a poor measure of similarity because vector spaces are rich and full of information.
- When considering unsupervised pretraining as a regularizer, we may anticipate that it will be most beneficial when there are relatively few labelled instances. Unsupervised pretraining should work at its best when there are a lot of unlabeled instances since unlabeled data is the source of new knowledge that it adds. Unsupervised pretraining won two international transfer learning competitions in 2011 in situations where the number of labelled examples in the target task was low, demonstrating the benefit of semi-supervised learning via unsupervised pretraining with many unlabeled examples and few labelled examples (from a handful to dozens of examples per class).
- There may be more elements at play. Unsupervised pretraining, for instance, is likely to be most beneficial when the function that has to be taught is quite complex. Unsupervised learning differs from regularizers like weight decay in that it encourages the learner to find feature functions that are beneficial for the unsupervised learning job rather than favouring the discovery of simple functions.
- Unsupervised learning may be a better regularizer if the real underlying functions are complex and affected by regularities in the input distribution.
- Keeping these warnings in mind, we will now examine a few success stories in which unsupervised pretraining is known to result in an improvement and explain what is known about the reasons why this improvement happens. Unsupervised pretraining is typically used to enhance classifiers and is especially intriguing when it comes to lowering test set error.

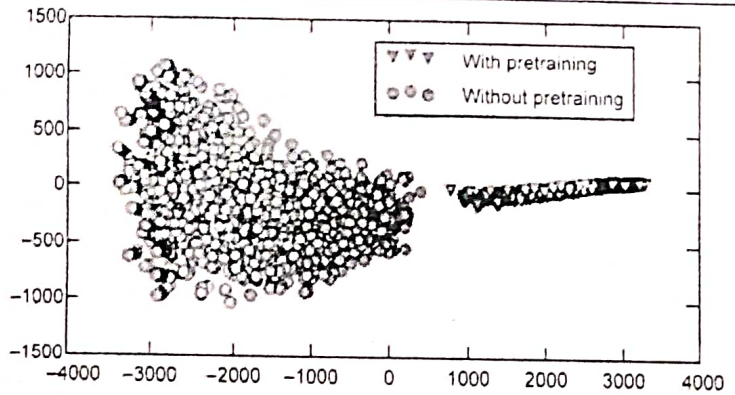

Fig. 5.2.1

- Fig. 5.2.1 : visualization of learning trajectories of distinct neural networks using nonlinear projection in function space (rather than parameter space, to overcome the problem of many-to-one mappings from parameter vectors to functions), with or without unsupervised pretraining. Each point represents a distinct neural network at a certain stage in its training. With permission, this image has been modified from ( ). An infinite dimensional vector that links each input x with each output y is referred to as a coordinate in the function. In order to create a linear projection to high-dimensional space, concatenated the y values for several distinct x positions. They next performed a second Isomap nonlinear projection to 2-D. Colors represent time. Near the middle of the figure, which corresponds to the area of functions that generates roughly uniform distributions over the class y for the majority of inputs, all networks are initialised. Learning expands the function over time to include regions that can reliably anticipate events. When pretraining is used, training consistently ends in one location and when pretraining is not used, it ends in a different, non-overlapping zone. The narrow region corresponding to the pretrained models may be an indication that the pretraining-based estimator has lower variance since Isomap strives to preserve global relative distances (and consequently volumes).
- Unsupervised pretraining, however, can aid in tasks outside classification and increase optimization as opposed to only serving as a regularizer. For deep autoencoders, it can decrease both the train and test reconstruction error.

- Many studies were conducted to illustrate the many benefits of unsupervised pretraining. Unsupervised pretraining may be used to explain both improvements in training error and improvements in test error by moving the parameters into an area that would not otherwise be accessible. The training of neural networks is non-deterministic and converges to a new function each time. Training may stop when the gradient becomes modest, when early stopping prevents overfitting, or when the gradient is significant but it is challenging to locate a downhill step because of issues like stochasticity or weak Hessian conditioning.
- Unsupervised pretraining causes neural networks to repeatedly halt in one area of the function space, whereas pretraining causes neural networks to repeatedly halt in a different area. For a representation of this occurrence, see Fig. 5.2.1. It is possible that pretraining lowers the estimate process' variance, which can lower the probability of severe over-fitting, because the region where pretrained networks arrive is smaller. To put it another way, unsupervised pretraining initialises neural network parameters into an area from which they cannot escape, and the results obtained as a consequence are more reliable and less likely to provide very poor outcomes than those obtained without initialization.
- When pretraining is most effective, the mean and variation of the test error were most significantly decreased by pretraining for deeper networks. Remember that these experiments were conducted prior to the development and widespread adoption of contemporary methods for training very deep networks (rectified linear units, dropout, and batch normalisation). As a result, less is known about the outcomes of combining unsupervised pretraining with these methods.
- How unsupervised pretraining may function as a regularizer is a crucial topic. Pretraining, according to one theory, promotes the learning algorithm to find characteristics connected to the fundamental factors causing the observed data. In addition to unsupervised pretraining, numerous additional algorithms are motivated by this crucial principle.
- Unsupervised pretraining has the drawback of operating with two distinct training stages, which makes it less effective than other methods of adopting this idea utilizing unsupervised learning. The weakness of the regularization resulting from the unsupervised pretraining cannot be predicted reduced or increased by any one hyperparameter, which makes these two training stages unfavorable. Instead, there are a large number of hyperparameters, whose impact may be observed after the fact but is frequently hard to foresee.

- A single hperparameter, often a coefficient related to the unsupervised cost, controls how strongly the unsupervised goal will regularize the supervised model when we do supervised and unsupervised learning concurrently instead of utilizing the pretraining technique. By lowering this parameter, less regularity may always be obtained consistently. There is no method to adjust the regularization's intensity in the case of unsupervised pretraining; either the supervised model is started with pretrained parameters or it is not.
- The fact that each phase has its own set of hyperparameters is another drawback of having two distinct training stages. It often takes a long time between suggesting hyperparameters for the first phase and being able to adjust them using input from the second phase since the performance of the second phase cannot be predicted during the first phase. Utilizing validation set error in the supervised phase to choose the pretraining phase hyperparameters is the most ethical strategy.
- Using early stopping on the unsupervised goal, which is less ideal but computationally considerably less expensive than using the supervised objective, makes it easier to configure specific hyperparameters during the pretraining phase in practise, such as the number of pretraining iterations.
- With the exception of natural language processing, where the natural representation of words as one-hot vectors provides little similarity information and where extremely large unlabeled sets are accessible, unsupervised pretraining has been largely abandoned in the modern era. The benefit of pretraining in this scenario is that one can pretrain once on a sizable unlabeled set (for instance, with a corpus containing billions of words), learn a good representation (typically of words, but also of sentences), and then use this representation or fine-tune it for a supervised task for which the training set contains significantly fewer examples.
- Only with extremely large labelled datasets can deep learning approaches that are based on supervised learning and regularised with dropout or batch normalisation reach human-level performance on a wide range of tasks. On medium-sized datasets like CIFAR-10 and MNIST, which have around 5,000 labelled samples per class, the same strategies outperform unsupervised pretraining. In comparison to approaches based on unsupervised pretraining, Bayesian methods perform better on extremely tiny datasets, such as the alternative splicing dataset. These factors have reduced the appeal of unsupervised pretraining.

- Unsupervised pretraining is still regarded as a significant turning point in the history of deep learning research and continues to have an impact on current methods. Pretraining has been broadly used to supervised pretraining, which is a popular transfer learning strategy. It is common to employ supervised pretraining for transfer learning using convolutional networks that have been pretrained on the ImageNet dataset. The parameters of these trained networks are made public by practitioners for this purpose, just like pretrained word vectors are made public for natural language tasks.

5.3 Transfer Learning and Domain Adaptation

- When what has been learnt in one environment (such as distribution P1) is used to increase generalization in another area, this is referred to as transfer learning and domain adaptation (say distribution P2). This expands on the concept that was introduced in the section before, when we showed how representations may be moved between an unsupervised learning task and a supervised learning task.
- The learner is required to complete two or more distinct tasks during transfer learning, but we anticipate that many of the variables that affect P1 will also affect P2 learning. This is commonly understood in the context of supervised learning, when the input is constant but the target may take multiple forms.
- For instance, in the first environment, we may learn about one set of visual categories, like cats and dogs, while in the second setting, we might learn about a another set of visual categories, like ants and wasps. If the initial setting (sampled from P1) has substantially more data, this may make it easier to acquire representations that can be easily generalized from just a small number of instances taken from P2.
- Numerous visual categories share fundamental ideas about edges, forms, the effects of geometry, lighting, etc. If there are traits that are helpful for the many settings or tasks, correlating to underlying factors that emerge in more than one context, transfer learning, multi-task learning, and domain adaptation may generally be accomplished via representation learning. Fig. 5.3.1 uses common bottom layers and task-dependent higher levels to show this.
- The semantics of the output, rather than the semantics of the input, may occasionally be shared by many activities. For instance, a voice recognition system must create correct phrases at the output layer, but depending on the speaker, the earlier layers close to the input layer may also need to identify highly diverse phonetic or sub-phonetic vocalisations. In these situations, it makes more sense to

combine the neural network's higher layers, which are located close to the output, with task-specific preprocessing, as shown in Fig. 5.3.1.

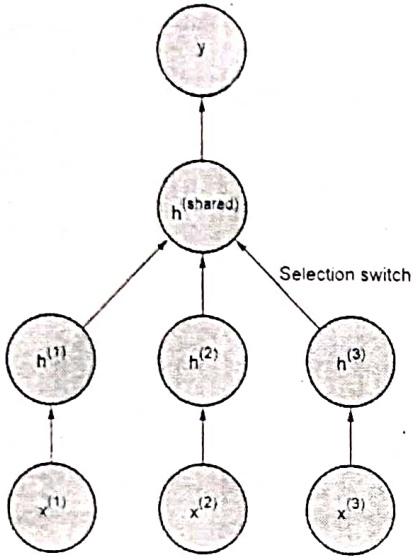

Fig. 5.3.1

- Example of a multi-task or transfer learning architecture, termed $x^{(1)}$, $x^{(2)}$, and $x^{(3)}$ for three tasks, where the output variable y has the same semantics for all tasks whereas the input variable x has a distinct meaning (and perhaps a new dimension) for each task (or, for example, each user). The top levels are shared, but the lower ones (up to the selection switch) are task-specific. The lower levels acquire the ability to convert their task-specific information into a collection of characteristics that are general.
- The objective (and the best input-to-output mapping) in the related scenario of domain adaptation is the same in each environment, but the input distribution varies significantly. Think about sentiment analysis, for instance, which involves figuring out if a comment indicates good or negative attitude. There are many different sorts of comments made online. When a sentiment predictor that was trained on customer evaluations of media material, such as books, films, and music, is later used to assess comments about consumer products, such as televisions or smartphones, a domain adaption scenario may occur.

- While it is conceivable that a function determines whether a statement is positive, neutral, or negative, the terminology and writing style will undoubtedly differ from one area to the next, making it more challenging to generalize across them. For sentiment analysis with domain adaptation, straightforward unsupervised pretraining (with denoising autoencoders) has been proven to be quite effective.
- Concept drift is an issue that is connected to transfer learning since it results from progressive changes in the data distribution over time. It is possible to think about idea drift and transfer learning as specific types of multi-task learning.
- While "multi-task learning" often refers to supervised learning activities, unsupervised learning and reinforcement learning can also fall under the more general category of transfer learning.
- The goal in each of these scenarios is to utilise the data from the first environment to draw conclusions that may be applied to learning or even direct prediction in the second context. The fundamental tenet of representation learning is that a single representation may be effective in a variety of contexts. The representation can take use of the training data that is available for both tasks by using the same representation in both scenarios.
- As previously indicated, certain machine learning contests have shown success using unsupervised deep learning for transfer learning. The experimental setting for the first of these tournaments is as follows. A dataset from the first setting (from distribution P1), including instances of a particular set of categories, is initially provided to each participant. In order to apply this learnt transformation to inputs from the transfer setting (distribution P2) and train a linear classifier effectively from a small number of labelled instances, participants must utilize this to learn a decent feature space (mapping the raw input to some representation).
- One of the competition's most notable findings is that the learning curve on the additional categories of the second (transfer) setting P2 improves significantly when an architecture uses deeper and deeper representations (learned entirely unsupervised from data acquired in the first setting, P1). The apparent asymptotic generalization performance of the transfer tasks requires fewer labelled instances for deep representations.
- One-shot learning and zero-shot learning, sometimes known as zero-data learning, are two extreme examples of transfer learning. In contrast to the zero-shot learning task, which provides no labelled examples at all, the one-shot learning task only provides one labelled example of the transfer task.

- Because the representation learns to clearly differentiate the underlying classes in the first stage, one-shot learning is feasible. Only one labelled example is required during the transfer learning phase in order to infer the label of several potential test cases that all congregate at the same location in the representation space. This is effective to the degree that the learnt representation space allows for a clear separation of the elements of variation corresponding to these invariances from the other factors, and we have figured out which factors are relevant for differentiating objects from other categories.
- Consider the issue of having a student read a sizable amount of material before having them perform object identification questions as an illustration of a zero-shot learning environment. If the language adequately describes the thing, it could be able to identify a certain object class even without having seen an image of that object. For instance, a student could be able to identify a picture as a cat without having ever seen one because they know that cats have four legs and pointed ears.
- Zero-shot learning and zero-data learning are only achievable when extra information was utilized during training. The conventional inputs x, the traditional outputs or objectives y, and an additional random variable representing the job, T, can all be thought of as random variables in the zero-data learning scenario. The job we want the model to complete is described by T, and the model is trained to predict the conditional distribution p(y | x, T). In our case, the result is a binary variable y with y = 1 signifying "yes" and y = 0 indicating "no" for the task of recognizing cats after reading about them.
- The task variable T thus stands for inquiries like "Is there a cat in this picture?" We might be able to infer the significance of unseen instances of T if we have a training set with unsupervised examples of items that coexist with T in that area. It's crucial that we used unlabeled text data with statements like "cats have four legs" and "cats have pointy ears" in our example of identifying cats without having seen an image of the cat.
- T must be expressed in a fashion that enables some level of generalization in order for zero-shot learning to work. T, for instance, cannot just be a single-hot code designating an object category. Instead, using a learnt word embedding for the word connected to each category.

- In machine translation, a similar situation occurs : on the one hand, we have words in a single language and can learn the connections between words from unilingual corpora; on the other hand, we have translated sentences that connect words in a single language with words in a different language. Because we have learned a distributed representation for words in language X, a distributed representation for words in language Y, and we have established a link (possibly two-way) relating the two spaces through training examples consisting of matched pairs of sentences in both languages, we can generalise and guess a translation for word A even though we may not have labelled examples translating word A in language X to word B in language Y.
- The three components (the two representations and the relationships between them) must be understood collectively for this transfer to be most effective.
- Fig. 5.3.2 Zero-shot learning is made possible by transfer learning between two domains, x and y. Examples of x, whether labelled or unlabeled, can be used to learn a representation function called f_x, and examples of y can be used to learn f_y. Each use of the f_x and f_y functions is represented by an upward arrow, with the arrow's style specifying which function is being used. A similarity measure between any two points in x space is provided by distance in hx space, which may be more significant than distance in x space. A similarity measure between any two locations in y space is provided by distance in hy space. Dotted bidirectional arrows are used to denote both of these similarity functions. Labeled instances (dashed horizontal lines) are pairs of examples (x, y) that enable learning of a one- or two-way map (solid bidirectional arrow) between the representations $f_x(x)$ and $f_y(y)$ and anchoring of these representations to one another. The following then enables zero-data learning. Simply because word-representations $f_y(y_{test})$ and image-representations $f_x(x_{test})$ may be associated to one another via the mappings between representation spaces, one can identify an image with a word even if that word has never been represented by an image. Although that word and that image were never matched, it works because their corresponding feature vectors, $f_x(x_{test})$ and $f_y(y_{test})$, have been connected to one another. Hrant Khachatrian's proposal was the inspiration for the Fig. 5.3.2.

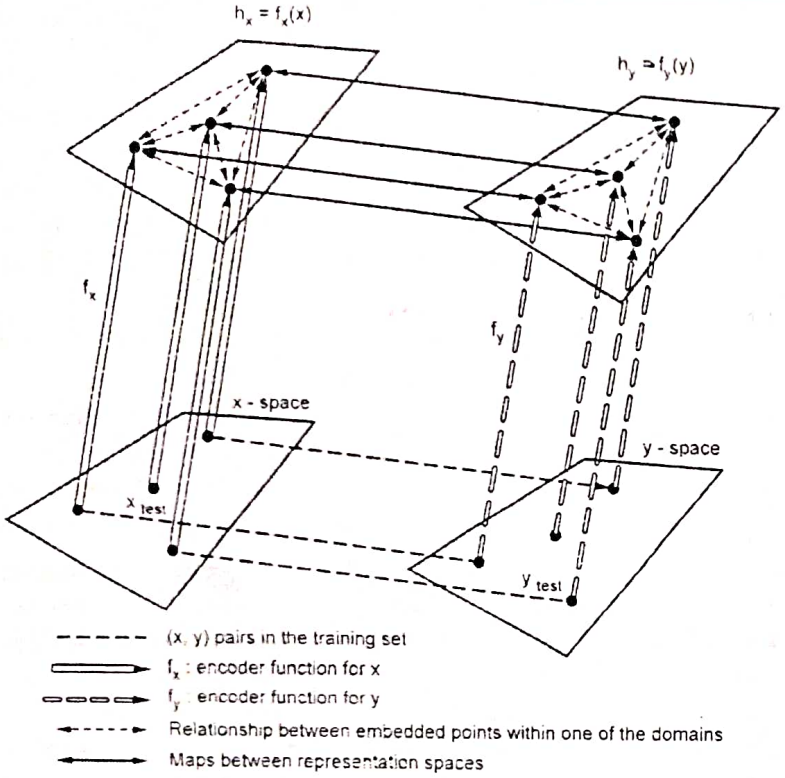

Fig. 5.3.2

- One type of transfer learning is known as zero-shot learning. The same principle explains how one can perform multi-modal learning, which involves capturing a representation in one modality, a representation in the other, and the relationship (generally a joint distribution) between pairs of data (x, y) made up of an observation in one modality (x) and an observation in another modality (y). Concepts in one representation are anchored in the other, and vice versa, allowing one to meaningfully generalize to new pairs. This is accomplished by learning all three sets of parameters (from x to its representation, from y to its representation, and the relationship between the two representations). Fig. 5.3.2 provides an illustration of the process.

5.4 Distributed Representation

- One of the most vital tools for representation learning is distributed representations of concepts, which are representations made up of several pieces that may be set

apart from one another. Because they may be used to express k n various concepts, distributed representations are strong because they can employ n features with k values. The method of distributed representation is used by probabilistic models with numerous latent variables and neural networks with many hidden units.

- We will now make another observation. The idea that the hidden units might learn to reflect the underlying causal elements that explain the data is what drives many deep learning algorithms. Because each direction in the representation space might correspond to the value of a different underlying configuration variable, distributed representations are a logical fit for this strategy.
- A vector of n binary features, as shown in Fig. 5.4.1, is an example of a distributed representation. This vector can assume 2n configurations, each of which may correspond to a distinct area in the input space. When the input is connected to a single symbol or category, it may be compared to a symbolic representation. One can envision n feature detectors, each of which corresponds to the detection of the existence of the related category, if the dictionary contains n symbols.
- In such scenario, only n distinct representation space configurations are feasible, slicing n distinct input space areas, as shown in Fig. 5.4.2. Due to the fact that such a symbolic representation may be recorded by a binary vector with n bits that are mutually exclusive, it is also known as a one-hot representation (only one of them can be active). A specific example of the wider class of non-distributed representations, which includes representations with several elements but no discernible meaningful independent control over each entry, is a symbolic representation.
- Non-distributed representation-based learning algorithms include :
  - There are other clustering algorithms, such as the k-means algorithm, which assigns each input point to precisely one cluster.
  - One or more templates or prototype samples are connected to a given input in k-nearest neighbors algorithms. Since each input in the situation of k > 1 has numerous values that describe it but cannot be controlled independently of one another, this is not a genuine distributed representation.
  - Decision trees: When an input is supplied, just one leaf (and the nodes on the route from root to leaf) are active.
  - The templates (cluster centers) or experts are now connected with a level of activation in both Gaussian mixtures and mixtures of experts. Similar to the

k-nearest neighbors technique, each input is represented by a number of values, however those values are difficult to regulate independently.

- Even if each "support vector" or template example's degree of activation is now continuous-valued, the same problem still exists with kernel machines with a Gaussian kernel (or another similarly local kernel).
- Models for language or translation are based on n-grams. A tree structure of suffixes is used to divide up the collection of contexts (sequences of symbols). For instance, w_1 and w_2 may be the final two words that make up a leaf. Each tree leaf's properties are calculated separately (with some sharing being possible).

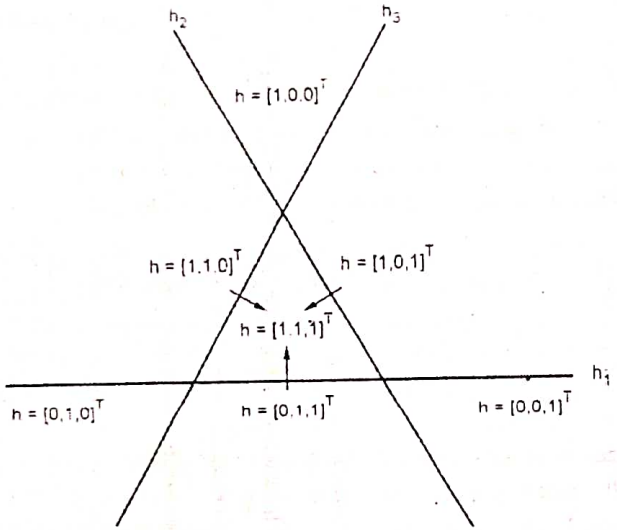

Fig. 5.4.1

- Fig. 5.4.1 : An example of how the input space is divided into regions by a learning algorithm using a distributed representation. There are three binary features in this example : h_1, h_2 and h_3. The output of a learnt, linear transformation is thresholded to determine each feature's characteristics. R^2 is split into two half-planes by each feature. Let the set of input points for which $h_i = 1$ be h_i^+ and the set for which $h_i = 0$ be h_i^-. Each line in this diagram depicts the one h_i decision border and each arrow points to the h_i^+ side of the boundary. For example, the representation value

$[1, 1, 1]^T$ corresponds to the region $h_1^+ \cap h_2^+ \cap h_3^+$. To the non-distributed representations in Fig. 5.4.2, compare this. A distributed representation divides R^d by intersecting half-spaces rather than half-planes in the typical case of d input dimensions. While the closest neighbor technique with n instances assigns unique codes to just n areas, the distributed representation with n features assigns unique codes to $O(n^d)$ various regions. Thus, compared to the non-distributed representation, the distributed representation can differentiate exponentially more areas. A linear classifier built on top of the distributed representation is unable to assign different class identities to every neighboring region and even a deep linear-threshold network only has a VC dimension of O(w log w), where w is the number of weights. Keep in mind that not all h values are feasible (h = 0 is not present in this example) (Sontag, 1998). A classifier trying to learn the concept of "person" versus "not a person" does not need to assign a different class to an input represented as "woman with glasses" than it does to an input represented as "man without glasses." Instead, the combination of a powerful representation layer and a weak classifier layer can be a powerful regularizer. This capacity restriction motivates each classifier to concentrate on a few hi and motivates classes to learn to represent the classes in a linearly separable manner.

- Some of these non-distributed algorithms provide output that interpolates between nearby areas rather than being consistent throughout portions. The number of parameters (or examples) and the number of areas they can describe continue to follow a linear relationship.
- The idea that generalization results from common qualities across several ideas is a crucial one that separates a distributed representation from a symbolic one. "Cat" and "dog" are as unlike to one another as any other two symbols are when seen as pure symbols. However, many of the things that can be said about cats and dogs may be applied to each other provided they are associated with a meaningful distributed representation.
- For instance, our distributed representation may include terms like "has fur" or "number of legs" that are equal for both "cat" and "dog" embedding. Compared to previous models that directly work on one-hot representations of words, neural language models that operate on distributed representations of words generalize significantly more well. In contrast to merely symbolic representations, distributed representations produce a rich similarity space in which ideas (or inputs) that are semantically related are physically adjacent to one another.

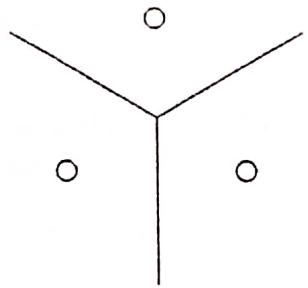

Fig. 5.4.2

- Fig. 5.4.2 : Illustration of the input space being divided into various sections using the closest neighbor method. An illustration of a learning algorithm built on a non-distributed representation is the closest neighbor method. The geometry of various non-distributed algorithms may vary, but they all usually divide the input space into regions and have a unique set of parameters for each zone. Given sufficient parameters, a non-distributed technique can match the training set without having to solve a challenging optimization procedure since it is simple to select a different output individually for each region. The drawback of such non-distributed models is that they only generalize locally through the smoothness prior, which makes it challenging to learn a complex function with more peaks and troughs than the number of samples. Contrast this with a distributed representation, Fig. 5.4.2.
- When and why may adopting a distributed representation as a component of a learning process be advantageous statistically? When an ostensibly complex structure can be compactly described using a few parameters, distributed representations can have a statistical benefit. The smoothness assumption, which asserts that if $u \approx v$, then the target function f to be learnt has the characteristic that $f(u) \approx f(v)$, in general, is what allows certain classic non-distributed learning methods to generalize.
- There are several methods to formalize this assumption, but the end result is that if we have an example (x, y) for which we know that $f(x) \approx y$, we select an estimator f that roughly meets these restrictions while changing as little as possible when we shift to an adjacent input $x + \varepsilon$. This presumption is undoubtedly very helpful, but it is plagued by the problem of dimensionality: we may require a number of examples that is at least as large as the number of distinguishable regions in order to learn a target function that increases and decreases many times in many different regions.

- You might think of each of these regions as a category or a symbol. By giving each symbol (or region) its own degree of freedom, we can learn any decoder that maps a symbol to a value. This prevents us from extrapolating to new symbols for new regions, though.
- If we're lucky, the goal function could not only be smooth but also have some regularity. For instance, even if an item is moved spatially and does not necessarily correlate to smooth changes in the input space, a convolutional network with max-pooling may still detect the object regardless of where it is in the image.
- Let's look at a specific instance of a distributed representation learning technique that thresholds linear functions of the input to extract binary features. As seen in Fig. 5.4.1, each binary feature in this form separates R^d into two half-spaces. How many regions this distributed representation learner can differentiate depends on how many intersections of n of the related half-spaces there are.
- How many regions are generated by an arrangement of n hyperplanes in R^d ? By applying a general result concerning the intersection of hyperplanes, one can show that the number of regions this binary feature representation can distinguish is

$$\sum_{j=0}^{d} \binom{n}{j} = O(n^d) \qquad \ldots(5.4.1)$$

- As a result, we see exponential rise in the input size and polynomial growth in the number of hidden units.
- This offers a geometric justification for the generalization ability of distributed representation : for n linear-threshold features in R^d, we can clearly represent $O(n^d)$ areas in input space with $O(n^d)$ parameters. Instead, if we utilized a representation with one distinct symbol for each area and separate arguments for each symbol to identify its associated section of R^d, we would need $O(n^d)$ instances to indicate $O(n^d)$ regions. In a broader sense, the scenario where nonlinear, perhaps continuous, feature extractors are used for each of the characteristics in the distributed representation rather than linear threshold units might be used to support the distributed representation.
- In this instance, it is argued that if a parametric transformation with k parameters can learn about r regions in input space, with $k \ll r$, and if obtaining such a representation was beneficial to the task at hand, then doing so could enable us to generalize much more effectively than in a non-distributed setting, where we would

require O (r) examples to obtain the same features and the associated partitioning of the input space into r regions. We have fewer parameters to fit and hence need far fewer training examples when the model is represented by fewer parameters.

- The fact that distributed representation-based models can only represent a finite number of separate areas further supports the claim that these models generalize effectively. A neural network with linear threshold units, for instance, only has a VC dimension of O(w log w), where w is the total number of weights. This constraint results from the fact that we cannot use the entire code space, nor can we learn arbitrary functions mapping from the representation space h to the output y using a linear classifier, despite the fact that we may assign very many unique codes to the representation space.
- A priori knowledge that the classes to be identified are linearly separable as a function of the underlying causal variables described by h is expressed by the use of a distributed representation together with a linear classifier. We won't generally wish to learn categories that involve nonlinear, XOR logic, such as the set of all photographs of all green items or the set of all images of vehicles. For instance, we normally do not want to divide the data into a class for all red and green vehicles and another class for all green and red vehicles.
- Although the concepts up to this point have been abstract, they may be empirically verified. Hidden units trained on the ImageNet and Places benchmark datasets learn characteristics that are often interpretable and correlate to a label that people would ordinarily provide. Although it is exciting to see this emerge towards the top levels of the greatest computer vision deep networks, it is clear that hidden units do not always learn things that have a straightforward language name. The fact that one may envision learning about one of these aspects without needing to observe all of the configurations of the others is what these traits have in common.

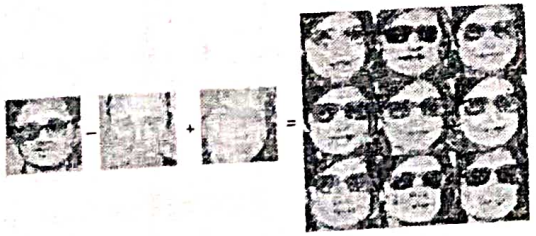

Fig. 5.4.3

- Fig. 5.4.3 : A distributed representation that separates the idea of gender from the idea of wearing glasses has been learned by a generative model. The vector representing the notion of a woman wearing glasses may be created by starting with the vector representing the concept of a man wearing glasses, then subtracting the vector for a man without glasses, and lastly adding the vector for a woman without glasses. All of these representation vectors are successfully decoded by the generative model into pictures that can be identified as being in the right class. Images taken from.
- A generative model may learn a representation of photographs of faces, as shown by, with distinct orientations in the representation space reflecting various underlying sources of variation. Fig. 5.4.3 illustrates how a person's gender and whether or not they are wearing glasses are represented by different directions in representation space. These characteristics weren't predetermined; they were automatically found.
- As long as the task calls for such features, gradient descent on an objective function of interest automatically learns semantically useful features for the hidden unit classifiers. Without having to characterize all of the configurations of the n + 1 additional traits, we may learn about the differentiation between male and female, or about the presence or absence of spectacles, via instances that cover all of these combinations of values. One can generalize to novel combinations of a person's characteristics that have never been encountered during training thanks to this type of statistical separability.

5.5 Variants of CNN : DenseNet

- There are number of variants of CNN including ResNet, LeNet, AlexNet, DenseNet etc. Here, we will discuss the DenseNet in detail.
- Compared to ResNet and Pre-Activation ResNet, dense connection achieves great accuracy with fewer parameters.
- DenseNet is discussed with the following points :
  - Dense block
  - DenseNet architecture
  - Advantages of DenseNet.

5.5.1 Dense Block

- Standard ConvNet uses several convolutions to extract high-level characteristics from the input picture.
- Identity mapping is suggested in ResNet to promote gradient propagation. It uses element-by-element addition. It may be thought of as an algorithm that is handed a state from one ResNet module to another.
- Each layer in DenseNet receives extra inputs from all levels that came before it and transmits its own feature-maps to all layers that came after it. You utilize concatenation. Each layer receives "collective knowledge" from the levels that came before it.
- Fig. 5.5.1 shows the DenseNet block.

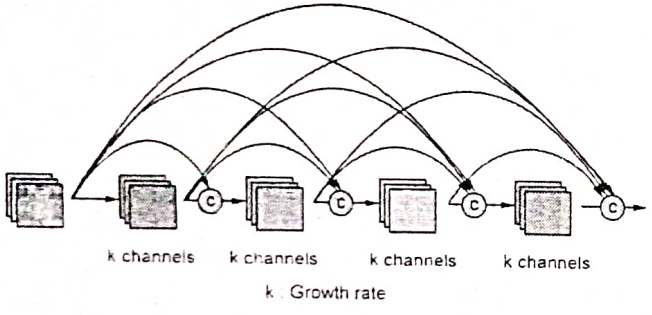

Fig. 5.5.1 Dense Block in DenseNet with growth rate k

- Each layer receives feature maps from all layers that came before it, allowing for a more compact and thin network with fewer channels. The extra number of channels for each layer is the growth rate k.
- Therefore, it has greater memory and processing efficiency.

5.5.2 DenseNet Architecture

- Basic DenseNet composition layer :

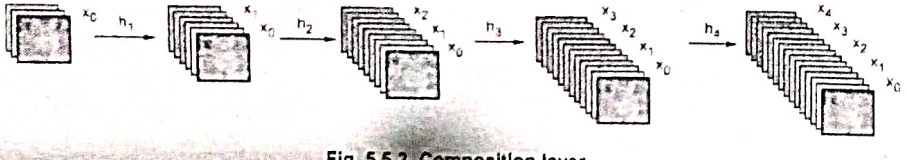

Fig. 5.5.2 Composition layer

- Pre-Activation Batch Norm (BN), ReLU and 3 × 3 Conv are performed for each composition layer with output feature maps of k channels, for example, to transform x_0, x_1, x_2 and x_3 to x_4. The Pre-Activation ResNet came up with this concept.
- **DenseNet-B (Bottleneck layers)**
  - BN-ReLU-1 × 1 Conv is carried out prior to BN-ReLU-3 × 3 Conv to lessen the complexity and size of the model.

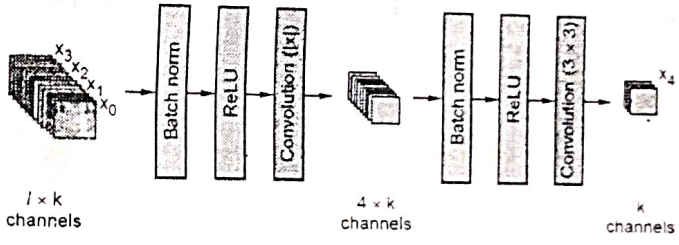

Fig. 5.5.3 DenseNet-B

- **Multiple Dense Blocks with Transition Layers**
  - The transition layers between two adjacent dense blocks are 1 × 1 Conv and 2 × 2 average pooling.
  - Within the dense block, feature map sizes are uniform, making it simple to concatenate them.
  - A softmax classifier is applied once a global average pooling is completed at the conclusion of the final dense block. (Refer Fig. 5.4.4 on next page)
- **DenseNet-BC (Further Compression)**
  - The transition layer produces m output feature maps if a dense block has m feature-maps, where $0 < \theta \leq 1$ is referred to as the compression factor.
  - The quantity of feature-maps across transition layers is constant when $\theta = 1$. DenseNet-C, or DenseNet with a value of $\theta < 1$, and $\theta = 0.5$ in the experiment.
  - The model is known as DenseNet-BC when both the bottleneck and transition layers with $\theta < 1$ are implemented.
  - DenseNets with/without B/C, various L layers, and k growth rates are also trained at this point.

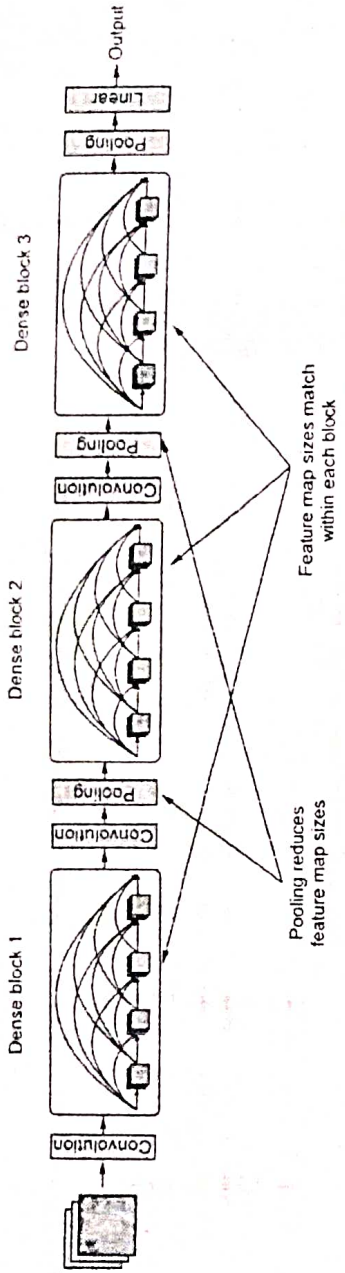

Fig. 5.5.4 Multiple dense blocks

5.5.3 Advantages of DenseNet

- **Strong Gradient Flow**
  - Direct propagation of the error signal to prior levels is simple. As previous layers can get direct supervision from the final classification layer, this is a form of implicit deep supervision.

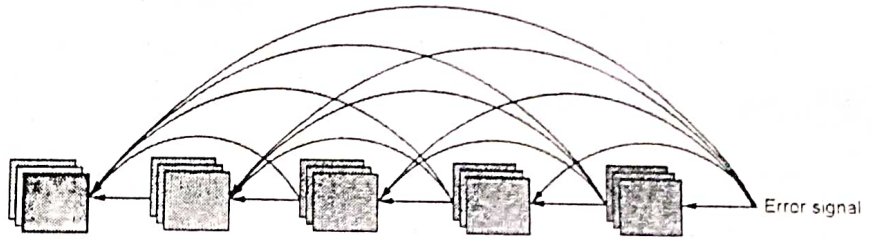

Fig. 5.5.5 Implicit "Deep Supervision"

- **Parameter and Computational Efficiency**
  - For each layer, number of parameters in ResNet is directly proportional to $C \times C$ while Number of parameters in DenseNet is directly proportional to $l \times k \times k$.
  - Since $k << C$, DenseNet has much smaller size than ResNet.

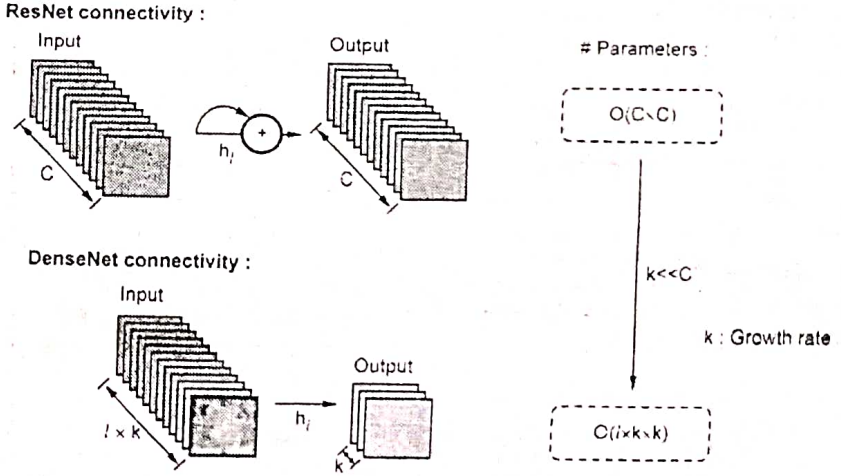

Fig. 5.5.6 Number of Parameters for ResNet and DenseNet

- **More Diversified Features**
  - DenseNet has more diverse characteristics and generally has richer patterns since each layer receives input from all preceding levels.

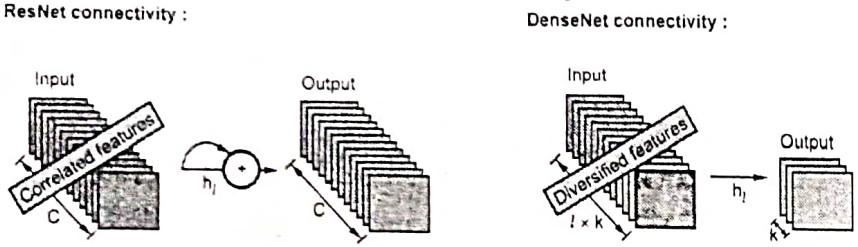

Fig. 5.5.7 More diversified features in DenseNet

- **Maintains Low Complexity Features**
  - The classifier in standard ConvNet employs the most intricate features.
  - The classifier in DenseNet incorporates characteristics of all degrees of complexity. It often provides more supple decision boundaries. It also explains why DenseNet functions effectively in the absence of sufficient training data. (Refer Fig. 5.5.8 on next page)

Review Questions

1. *Explain the concept of representation learning.*
2. *Explain the Greedy layer-wise unsupervised pretraining.*
3. *What is transfer learning. Elaborate transfer learning and domain adaption.*
4. *Write short note on distributed representation.*
5. *Explain the CNN variant DenseNet with block structure, architecture and advantages.*

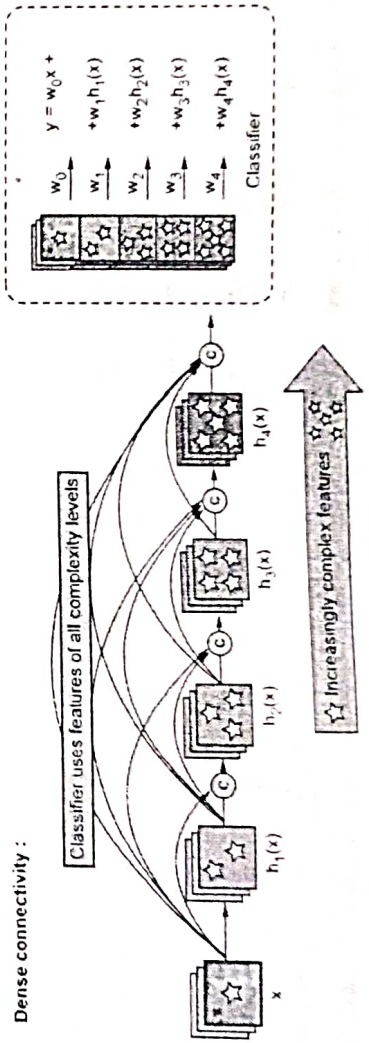

Fig. 5.5.8 DenseNet

Unit VI

6 Applications of Deep Learning

Syllabus

Overview of Deep Learning Applications : Image Classification, Social N/w/ analysis, Speech Recognition, Recommender system, Natural Language Processing.

Contents

6.1 Overview of Deep Learning Applications

- Deep Learning a subset of Machine Learning has gained great popularity in various AI applications. Deep learning algorithms excels the conventional algorithms in terms of accuracy and performance. There is substantial enhancement in machine translation and computer vision tasks using deep networks.
- Deep learning uses learning methods with multiple intermediate layers of representation. The non linear modules in these layers modify the representation into a more conceptual representation at next level. As deep learning learns the features from data using a general purpose learning procedure instead of being designed by human experts it has outperformed the conventional algorithms in terms of accuracy.
- Machine learning models can achieve good generalization and high accuracy with deep learning.
- Spoken word recognition task can achieve near good performance to human being by using deep learning techniques .
- It is also used for decision making in various fields like finance, medicine, e-commerce, manufacturing and many more.
- Deep learning technology is playing a key role in various applications like Object detection, business analytics, robotic automation, video analytics, speech recognition, social media analysis, gaming etc. Fig. 6.1.1 shows different application areas of deep learning.

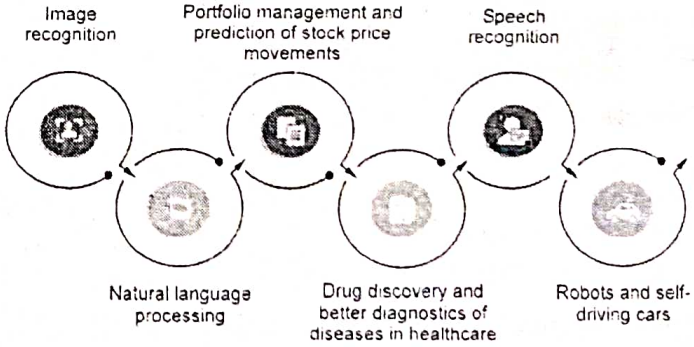

Fig. 6.1.1 Examples of deep learning application

- Deep learning techniques are making valuable contribution in strong and accurate decision making .

- More and more applications are relying on deep learning techniques from diverse fields such as self driving cars, human resources, healthcare, finance retail, earthquake detection etc.
- Most of these AI applications, the large-scale neural network implementations are required.

6.2 Image Classification

- Image classification is one of the computer vision task that identify what an image represents. It assigns the labels to a given image. A machine/ computer analyze the image and identify its class / label like 'Car', 'bird', 'staircase' 'building' etc. Image classification task may give the probability of the class of image it belongs.
- e.g. if video frame as shown in Fig. 6.2.1 is given as input then image classification is process in which computer will analyze the image and assigns labels as 'person' and 'bicycle'

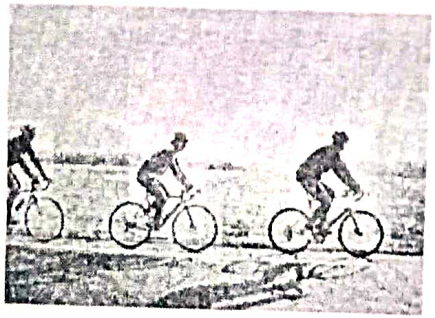

Fig. 6.2.1 Video frame with object detection to recognize the pre-trained classes "person" and "bicycle"

- Image classification finds applications in various areas, such as medical imaging, traffic controlling systems, plant diseases identification in agriculture, object detection in satellite image, machine vision and many more.

6.2.1 Image Classification Techniques

There are two categories of image classification techniques :

i. **Supervised classification**

In supervised image classification technique previously classified reference samples (the ground truth) are used to train the classifier for classifying new, unknown data.

ii. **Unsupervised image classification**

In unsupervised classification technique user need not provide the sample classes. It is fully automatic process where machine learning algorithms analyze and cluster the unlabeled datasets by finding hidden patterns in the image without any human intervention.

6.2.2 How does Image Classification Work ?

- Image is analyzed in the form of pixels. Image is an array of pixels where size of the matrix depends on resolution of an image. Image classification task is done by grouping pixels into specified categories referred to as classes.

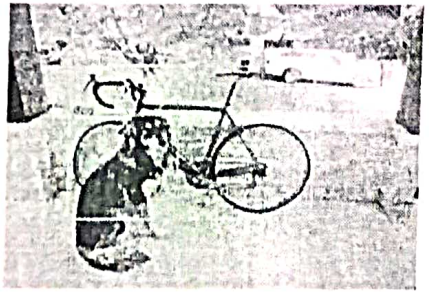

Fig. 6.2.2 Grouping of pixels into specified classes

- The image is segregated into its most prominent features using the algorithm giving the classifier an idea about the class of the image it may belong. Thus the feature extraction process is most important step in image classification. Also data fed to the algorithm plays important role particularly in supervised image classification technique.

6.2.3 Advantages of using Deep Learning in Image Classification

- Early image classification techniques depends on raw pixel data. The image of same object can have different background, poses, angles etc. So there is a problem that two images of same object may look very different.
- Machine learning algorithms can also be used for image classification task. They have the potential to learn the hidden patterns from training dataset of samples. Deep learning is most popular machine learning technique in which model can use many hidden layers.

- Real time object detection and human level performance in image classification task can be achieved by combining deep learning with GPUs and robust AI hardware.
- The main advantage of deep learning over traditional image processing approach is that machines are allowed to identify and extract the features from images. Programmers need not apply the filters to create handcrafted features.

6.2.4 Application Areas of Image Classification

- As far as humans and animals are considered vision is a effortless task but it is challenging for computers. Image classification forms basis of computer vision problems.
- In self driving cars image classification can be used to detect traffic lights, trees, vehicles, peoples around etc.
- In healthcare it can be used to analyze medical images and depict the symptoms of illness.
- With the ubiquitous technologies such as AI and IOT huge amount of data in the form of images, video, speech is generated. Image and video data posted by persons can be used in recommendation system in online shopping or places to visit etc.

6.2.5 Image Classification using Deep Neural Networks

- There are three or more layers of Artificial Neural Networks (ANN) in deep learning. Each layer learns to extracts one or more features of the image. Using ANN for the image classification task is computationally very expensive.

  e.g for image of size 50 × 50 , the trainable parameters becomes -

  [(50 * 50) * 100 image pixels multiplied by hidden layer] + 100 bias

  + [2 * 100 output neurons] + 2 bias = 2,50,302
- On the other hand CNN just requires some preprocessing. CNN develop and adapt their own image filters. CNN uses filters to produce 'Activation Map' or 'Feature Map' which greatly reduced the dimensionality of the image. So Convolutional Neural Network (CNN) are most widely used in computer vision applications.

6.2.6 CNN for Image Classification

- Fig. 6.2.3 illustrates the concept of image classification using CNN. The first part is feature extractor which consists of convolutional layers and max pooling layers. The second part is classifier which consists of fully connected layer .

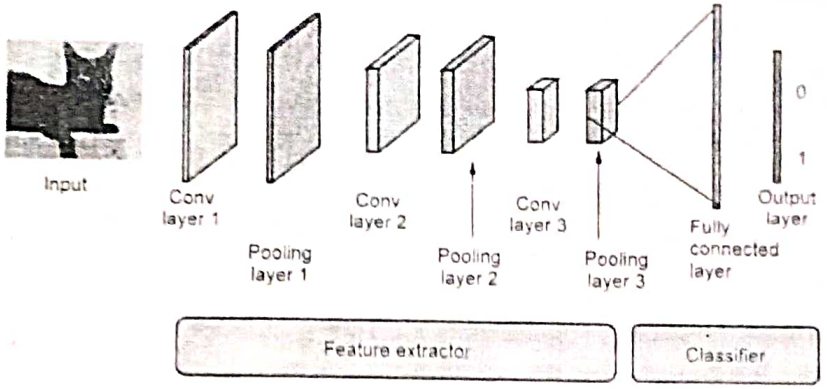

Fig. 6.2.3 Image classification with CNN

- **Convolutional layer :** The convolutional layer acts as the eyes of the CNN. In convolution a simple filter is applied to input that results in activation. The neurons in this layer search for specific features. High activation is produced for the feature they are looking for. Activation map is obtained by sliding the convolution kernel/filter over the entire matrix. Convolution operation slides the kernel on input, perform the pair-wise matrix multiplication and results are summed up. The activation map of one layer is applied as the input to the next layer. Same weights and biases are shared by all neurons in a layer corresponding to one filter. Output reduces in size after each convolution. So zero padding is done to the boundary of input such that the size of the output and input layer is same.
- CNN can learn hierarchical features. Initial layers learns simple features like corners/edges etc. by looking at smaller regions of the image. Neurons in the deeper layers gets the information from larger parts of the image thus learning more complicated features. Fig. 6.2.4 illustrates this.

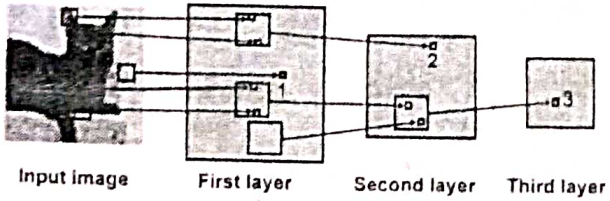

Fig. 6.2.4 Hierarchical feature learning by CNN

- **Pooling layer :** Mostly pooling layer is used immediately after the convolutional layer. The result of all neurons in the preceding layers is collected by pooling layer. The main task of pooling layer is to reduce the spatial size i.e only height and width

and not the depth. So the number of parameters are reduced thus reducing the computations. Due to fewer parameters overfitting is also avoided.

- **Fully-connected layer :** Final output layer of CNN model is the fully connected layer. It gives the results by flattening the input received from the layer before it .

6.3 Social Network Analysis

- Social Network Analysis (SNA) is a field of data analytics. It uses network and graph theory to understand social structures. SNA studies the relationship, interactions and communications in a network with the use of several tools and methods. This study helps in administration, operations and problem solving of that network. Two key components of SNA graphs are :
  1. **Actors :** Also called as node or vertex.
  2. **Relationship :** Also called as tie or edge.
- Internet is the most common application of SNA where webpages are linked with other webpages either on their own webpage or another website. These webpages can be considered as nodes/actors and the links between them as edges or ties or relationship. Thus the entire internet can be interpreted as a large graph. This is shown in Fig. 6.3.1.

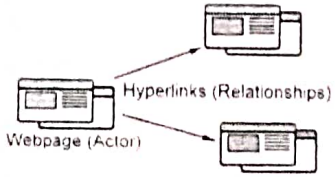

Fig. 6.3.1 Actor and relationship in SNA

6.3.1 SNA Terminologies

i) Nodes and edges

- In network science, actors also called as nodes are denoted by the dots on the graph. Relationships also called as edges denoted by the lines on the graph.

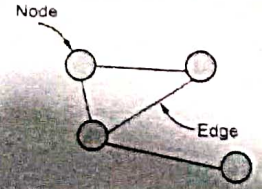

Fig. 6.3.2 Node and edge

- Variety of things can be represented by nodes. e.g. nodes can represent webpages in internet, peoples in social networks, countries in foreign relation network, organizations in supply chain management network etc. Similarly variety of relationships can be represented using edges. e.g. edges can represent hyperlink in case of internet, connections among people in social network, policies in case of foreign relation network and transfer of goods in case of supply chain management network.
- **Edge direction**

  There are two types of edges : directed and undirected.
  - **Directed edge :** The nodes connected by this edge are ordered. This means that there is one way connection between the nodes. They can be either one directional or bidirectional. For example, Twitter, Instagram are predominantly directed edge networks.

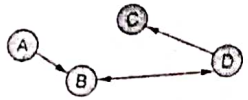

Fig. 6.3.3 Directed edge

  - **Undirected edge :** There is mutual relationship between the nodes connected by this edge. This means that the connection is applicable both ways. Here the relationship means the same thing to both actors E.g., Befriending a person on Facebook, LinkedIn automatically creates a two-way connection.

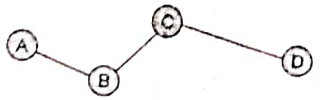

Fig. 6.3.4 Undirected edge

ii) Weight :

- Edge can have label/weight in a weighted network. Weight of the edge is the number of times the particular edge appears between two specific actors/nodes. The number of mutual connections between the nodes connected by particular edge is referred to as weight in case of social media analysis.
- e.g if a person X buys a shirt from some shop 3 times , then the edge connecting Person X and that shop will have a weight of 3. However if another person Y buys the shirt from that shop only once then the edge connecting Person Y and that shop will have a weight of 1.

iii) Centrality measures :

- Centrality measures are applied on specific node within the network to assess how important is that node to the whole network. Several centrality measures are there. Some commonly used are degree, closeness, and betweenness.
- **Degree centrality :** It is the number of direct edges to a node. Most connected node in the group is indicated by it. e.g. consider a network shown by Fig. 6.3.5.

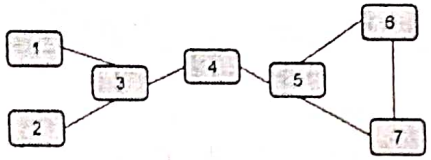

Fig. 6.3.5 Network

Degree centrality score of a network = Sum of edges connected to that node

So, degree centrality score for Node 1 = 1,

Degree centrality score for Node 3 and 5 = 3.

- As nodes 3 and 5 have a high degree centrality score. This means that they are the most well-connected nodes in the network.
- **Closeness centrality :** Closeness measures how well connected a node is to every other node in the network. It shows the node's ability to reach the other nodes in a network. Closeness centrality score is the inverse of the sum of distance between a node and other nodes in network. e.g consider a network shown by Fig. 6.3.5.
- Table below shows the distance from particular node to destination node, sum of distance and closeness score.

Destination Nodes	1	2	3	4	5	6	7	Total	Closeness Score
Distance from node 1 to destination node	-	2	1	2	3	4	4	16	1/16
Distance from node 2 to destination node	2	-	1	2	3	4	4	16	1/16
Distance from node 3 to destination node	1	1	-	1	2	3	3	11	1/11
Distance from node 4 to destination node	2	2	1	-	1	2	2	10	1/10

Distance from node 5 to destination node	3	3	2	1	-	1	1	11	1/11
Distance from node 6 to destination node	4	4	3	2	1	-	1	15	1/15
Distance from node 7 to destination node	4	4	3	2	1	1	-	15	1/15

From above table it is observed that the closeness score of node 4 is highest so it is the central / closest node in a network.

- **Betweenness centrality :** It is a measure of how often a node appears in the shortest path connecting two other nodes. Consider node 5 in Fig. 6.3.5 Node 5 appears in 9 shortest paths between a pair of nodes 1 - 5 ,1 - 6 , 1 - 7, 2 - 5, 2 - 6, 2 - 7 and 3 - 5, 3 - 6, 3 - 7
- Path value is 1 if the node (in this ex node 5) is the only node in the path. Path value is 1/n if it is one of the 'n' nodes in the shortest path. Then the betweenness score is sum of the path values for that node (node 5) for all pairs of nodes. Path values of node 5 for all 9 pairs are shown in following table.

Node pair	Path value of node 5
1 - 5	1/2
1 - 6	1/3
1 - 7	1/3
2 - 5	1/2
2 - 6	1/3
2 - 7	1/3
3 - 5	1
3 - 6	1/2
3 - 7	1/2
Total Score for node 5	**13/3**

Similarly for other nodes the betweenness centrality score are as shown in below table.

Node pair	Score
1	0
2	0
3	16/3
4	13/3
5	13/3
6	0
7	0

Nodes with high betweenness centrality score are critical nodes in the network.

iv) Network-level measures

- Network level measures are calculated for evaluating entire network rather than single node.

a) **Network size :** Number of nodes in the network is called as network size. This measure does not consider the number of edges.
e.g. a network of Fig. 6.3.6 with nodes A, B, and C has a size of 3.

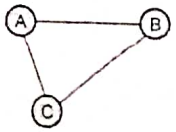

Fig 6.3.6 Network

b) **Network density :** Number of edges divided by the total possible edges gives the network density. e.g. , Consider network of Fig. 6.3.7. Here node A connected to node B and node B connected to node C.

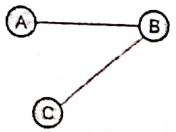

Fig. 6.3.7 Network

So Network density = Total edges / Total possible edges

Total edges = 2

Possible edges = 3

So Network density = 2/3

- These are some of the metrics used in SNA. In addition to this other path level metrics are also used.

6.3.2 Applications of Social Network Analysis

- Applications of SNA are summarized in Fig. 6.3.8

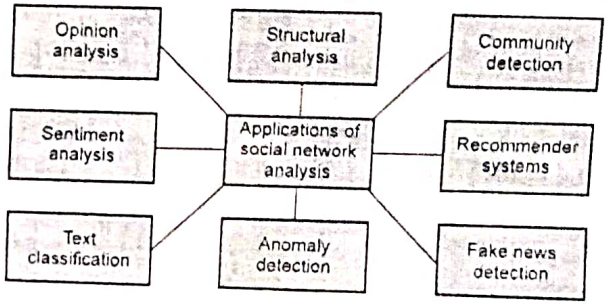

Fig. 6.3.8 Applications of SNA

Some of the applications of social network analysis includes :

1. **Supply chain management :** Retailers, suppliers, warehouses, transporters, regulatory agencies are nodes. SNA helps manufacturers to identify critical nodes and potential sources in order to increase the number of connections to suppliers. Bottlenecks in the supply process and inventory management can also be identified using SNA.
2. **Infectious diseases transmission :** In such cases transmitters of disease can be identified with the help of SNA. Individuals and groups with high betweenness centrality and out degree centrality can be identified with the use of SNA measures so as to isolate them.

6.3.3 Social Network Analysis using Deep Learning

- For social network analysis, the primary step is to encode the network data into network embeddings. Network embeddings are low dimensional representations of network data. Applications like classifications, clustering, link prediction etc. are possible due to network representation learning.

- Deep neural networks can be used to learn representations from network data. Three categories based on type of neural network are :
  1) Look-up table based models
  2) Autoencoder based models
  3) Graph Convolutional Network (GCN) based models.

1. Models with embedding look - Up tables

- Look up tables can be used for network representation learning instead of using multiple layers of nonlinear transformations. Node index is directly mapped with its corresponding representation vector using the look up tables.
- Look up table can be implemented using matrix. Each row of the matrix corresponds to the representation of one node.
- Main building blocks of the model with embedding look-up tables are shown in Fig. 6.3.9. Sampling and modeling are the two key components of this approach.

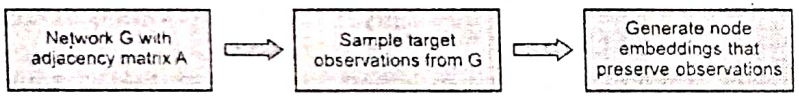

Fig 6.3.9 Building blocks of models with embedding look - up tables

2. Autoencoder based Models

- Two neural network modules of an autoencoder are :
  i) Encoder and ii) Decoder.
- Fig. 6.3.10 shows autoencoder based network representation. The function of encoder is to map the features of each node into a latent space. Information about the network. is then reconstructed by decoder from this latent space.
- The size of hidden representation layer is usually small as compare to input/output layer. The non linear network structure is captured by this compressed representation. The output is decoded from these low dimensional representations. Minimizing the reconstruction error between input and decoded output is the main objective function of autoencoder.
- Autoencoder is fed with the rows of the proximity matrix $S \in R^{|V| \times |V|}$ so as to learn and generate embeddings $Z \in R^{|V| \times D}$ at the hidden layer.

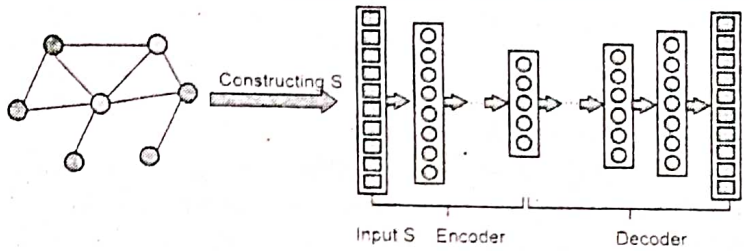

Fig 6.3.10 Autoencoder based network representation

6.3.4 Graph Convolutional Approaches

- As Convolutional Neural Networks (CNN) have improved the performance of image recognition task significantly, CNN modules are also adapted to learn representations of network data. Node embeddings are generated by aggregating the information from its local neighborhood. In contrast to autoencoder based approach, the encoding function of GCN based model uses node's local neighborhood and attribute information.
- This is shown in Fig. 6.3.11. Node attributes are denoted by dashed rectangles. Each individual node's representation is aggregated from its immediate neighbors concatenated with its lower layer representation. e.g in Fig. 6.3.11 node C representation is aggregated from its immediate neighbors i.e. nodes A, B, D, E.

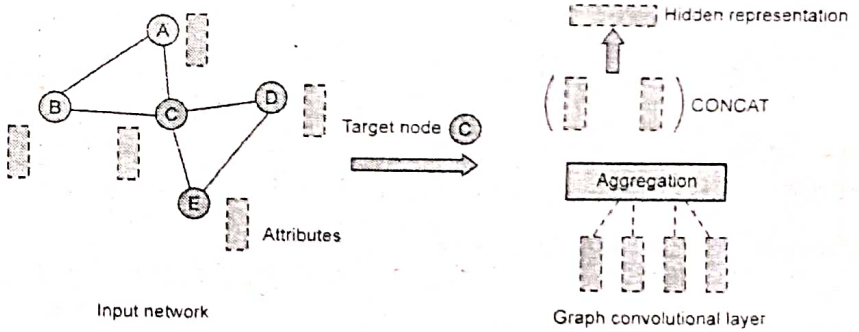

Fig. 6.3.11 Graph Convolutional Networks (GCN)

- Either spatial filter or spectral filters are used as convolutional filters of this approach.

6.4 Speech Recognition

- Automatic Speech Recognition (ASR) plays important role in human to human communications and also in human machine communications. Humans use speech in some language to communicate with each other. Speech recognition is the ability of the machine to recognize human speech and translate it into machine readable format.
- Thus speech recognition is the task of mapping an acoustic signal corresponding to an utterance in spoken natural language to the sequence of words uttered by speaker.
- Let

$$X = (x^{(1)}, x^{(2)}, \ldots, x^{(T)}) \quad : \text{Sequence of acoustic input vectors}$$

$$y = (y_1, y_2, \ldots, y_N) \quad : \text{Target output sequence}$$

where input vector X is produced by splitting the audio signal into frames of 20 ms and output y is sequence of characters or words.

- The Automatic Speech Recognition (ASR) is then defined as the task of finding the function f^*_{ASR} which computes most probable linguistic sequence y from the given input X as given by equation (6.4.1)

$$f^*_{ASR}(X) = \arg\max_y P^*(y \mid X = X) \qquad \ldots(6.4.1)$$

where P^* denotes the true conditional distribution relating the inputs X to the targets y.

6.4.1 Basic Architecture of ASR Systems

- Fig. 6.4.1 shows the architecture of automatic speech recognition system. The main components of ASR are as follows :
  1. Signal processing and feature extraction
  2. Acoustic Model (AM)
  3. Language Model (LM)
  4. Hypothesis search.

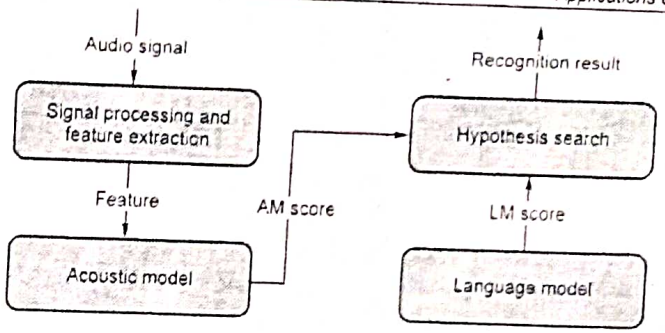

Fig. 6.4.1 Architecture of ASR

- Input audio signal is applied to signal processing and feature extraction unit. Here Channel distortions and noise is removed to enhance the quality of speech signal. Speech signal in time domain is then converted to frequency domain and salient feature vectors suitable for acoustic model are extracted.
- The features generated by signal processing and feature extraction unit are applied as input to Acoustic Model (AM). It integrates the knowledge about the acoustics and phonetics and produce AM score for the variable-length feature sequence.
- Language Model (LM) learns the correlations between words from the given training text corpus. It estimates the LM score i.e the probability of a hypothesized word sequence.
- The AM score of feature vector sequence and LM score of hypothesized word sequence are given as input to hypothesis search unit. Word sequence with highest score is outputted as recognition result.

6.4.2 Traditional ASR Approach

- Traditional speech recognition systems are based on Gaussian Mixture Models (GMMs) and Hidden Markov Models (HMMs). Association between acoustic features and phonemes is modeled by GMM. The phonemes sequence is modeled by HMM. HMM first generates the sequence of phonemes and discrete subphonetic states like beginning, middle, and end of phoneme . Each discrete state is then transformed into segment of audio waveform using GMM.
- Some of the drawbacks of this approach are :
  i) Lower accuracy,
  ii) Time and labor intensive as each model need to be trained independently.
  iii) Requirement of forced aligned data (i.e alignment of text transcription and time of occurring it in speech).
  iv) Need of experts to build phonetic set for boosting accuracy of model.

6.4.3 Deep Learning for ASR

- By using end to end deep learning models sequence of input acoustic features can be directly mapped to sequence of words. Also forced aligned data is not needed.
- Various forms of Artificial Neural Network like CNN, RNN, transformer networks can be used for speech recognition.
- Recognition accuracy improved dramatically by replacing GMM with larger and deeper neural networks and using large datasets.
- In RNN current hidden state depends on all previous hidden states. Hence it is suitable for modeling time series signals. Also long term and short term dependencies from input at different instance of time can be captured using RNN. Due to temporal relationship of speech data and time dependent phonemes RNNs are suitable for speech recognition applications.
- Fig. 6.4.2 shows bidirectional RNN used for automatic speech recognition.

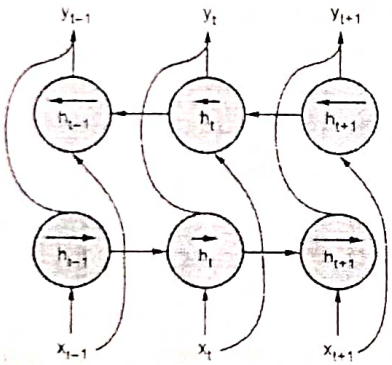

Fig. 6.4.2 Bidirectional RNN for ASR

The input sequence is $x = (x_1, x_2, \ldots x_T)$

Hidden sequence is $h = (h_1, h_2, \ldots h_N)$ and

Output sequence is $y = (y_1, y_2, \ldots y_N)$

Hidden vector h is computed by RNN using eq. (6.4.2)

$$h_t = H(W_x h x_t + W_{hh} h_{t-1} + b_h)$$

$$y_t = W_{hy} h_t + b_y \qquad \ldots(6.4.2)$$

where W : Weights , b : Bias ,H : Nonlinear function

- In speech recognition, information regarding future contex of speech is equally important as the past context. Bidirectional RNN (BiRNNs) process the input vector in both forward and backword directions and finds hidden state vector for each direction. That is why instead of using unidirectional RNN, bidirectional RNN (BiRNNs) are widely used for speech recognition.
- Only framewise classification of audio input signal can be performed using both feed forward and recurrent neural Networks. So force alignment between input audio and corresponding transcribed output is needed. This can be done using Hidden Markov Models (HMM) or Connectionist Temporal Classification (CTC) loss. CTC is an objective function. Alignment between the input speech signal and the output sequence of the words is computed using CTC.
- **RNN-transducer**

  Other commonly used deep learning architecture for speech recognition is RNN transducer. RNN transducer is a combination of one RNN predicting next output given the previous one and another RNN with CTC. It is shown in Fig. 6.4.3.

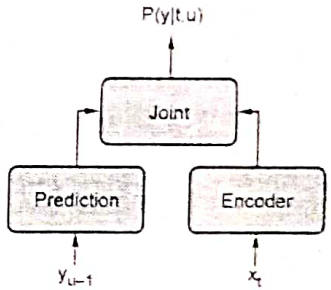

Fig. 6.4.3 RNN transducer overview

- Acoustic feature x^t at time-step t is converted to a representation

  $het = fenc(x_t)$ by the encoder network.
- Prediction network generates a new representation $hp_t = f_p(y_{u-1})$ for the previous label y_{u-1}
- The joint network is fully connected layer. The two representations from encoder and prediction network are combined in joint network to generate the posterior probability P(y|t,u) = f joint(het:hpt)
- Thus by using the information from both encoder and prediction network the next symbol or word is generated depending on whether the predicted label is blank or non blank. When the blank label is emitted for last time step the inference

procedure stops. RNN transducers can also be used for real time speech recognition.

6.5 Recommender System

- In information technology sector, machine learning / deep learning have major role in making recommendations of items to potential users or customers. Two main applications are :

  i) Online advertising and

  ii) Item recommendations.

  For both the applications the association between user and item is required to be predicted.

- A recommendation system is one of subcategory of Information filtering Systems .It attempt to predict the "rating" or "preference" that a user might give to an item. Primarily use of recommender system is in commercial applications. e.g. recommending product to buy in case of online shopping, music on spotify, videos to watch on YouTube etc.

- So the recommender system can be defined as the process of determining the mapping

$$(c, i) \rightarrow R$$

where c : A user,

i : An item, and

R : The utility of the user being recommended with the item.

(The concept of utility means user's action followed e.g. purchase of item, clicking on "not show again" etc.)

Items are sorted by utility and top N items are recommended to user.

6.5.1 Types of Recommender Systems

- Types of recommender systems are :

  1. Content-based filtering
  2. Collaborative filtering
  3. Hybrid systems.

Also recommender systems can be classified depending on whether a model has learned from the underlying data, as

1. Memory - based
2. Model - based.

1. Content-based filtering

- Recommendations of relevant items are made to the user based on the contents of previously searched items by the user. Basically content based recommender system is user specific learning problem. User's utility i.e. likes, dislikes, rating etc. is quantified based on features of the item. There is lack of user's personal information.

- Content means tag or attributes of the product. In this type of system certain keywords are used to tag the products. The system tries to know requirement of user and search in its database to recommend different products to user. Fig. 6.5.1 illustrates this.

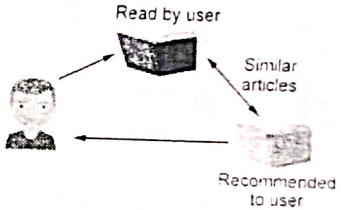

Fig. 6.5.1 Recommender system using content - based filtering

- **Advantages**

  i. As recommendations are specific to single user, data of other user is not needed for the model.

  ii. It can be easily scaled for large number of users.

  iii. Specific interests of the user can be captured by this model.

- **Disadvantages**

  i. As the recommendations are based on user's existing interest it has limited ability to exapand the exisiting interests of user.

  ii. Lot of domain knowledge is required for feature representation of items.

2. Collaborative based filtering

- In this type of recommender system new items are recommended to the user based on the preference and interests of other similar users. e.g online shopping website recommends new products by saying *"Customer who brought this also brought"* .
- Unlike content-based approach, collaborative based filtering systems attempt to predict a user's utility for an item based on other users' previous utility with the item.
- Disadvantage of content based filtering approach is overcomed as it uses the interaction of user instead of content from items used by user.
- **Advantages**
  i. Works well for small data also.
  ii. Domain knowledge is not needed.
- **Disadvantages**
  i. There is a problem of cold start i.e new items can not be handled by the model as it is not trained on newly included items in the database.
  ii. Much importance is not given to side features.

3. Hybrid method

- Hybrid method of recommender system combines the both content based and collaborative methods. Different ways of combining them are as follows
  1. **Model ensemble approach :** In this both content based and collaborative methods are implemented seperatly and their predictions are combinred together.
  2. Content based characteristics are incorporated with collaborative method. This can be done by leveraging user profile to measure similarity between two users and then this similarty can be used as weight during aggregation step of collaborative approach.
  3. Collaborative characteristics are incorporated with content based method.This can be done by applying dimensionality reduction on group of user profiles and give this as collaborative-version profile for the user of interest.
  4. **A cross-user and cross-item model :** In this model is build using both item features and user features . e.g tree model, linear regression model etc.

6.5.2 Deep Learning based Recommender Systems

- The two phases in developing deep learning based recommender system are : Training and Inference. Fig. 6.5.2 and 6.5.3 illustrates this.

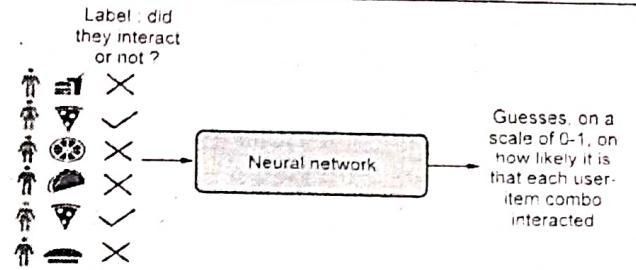

Fig. 6.5.2 Deep learning for recommendation : Training phase

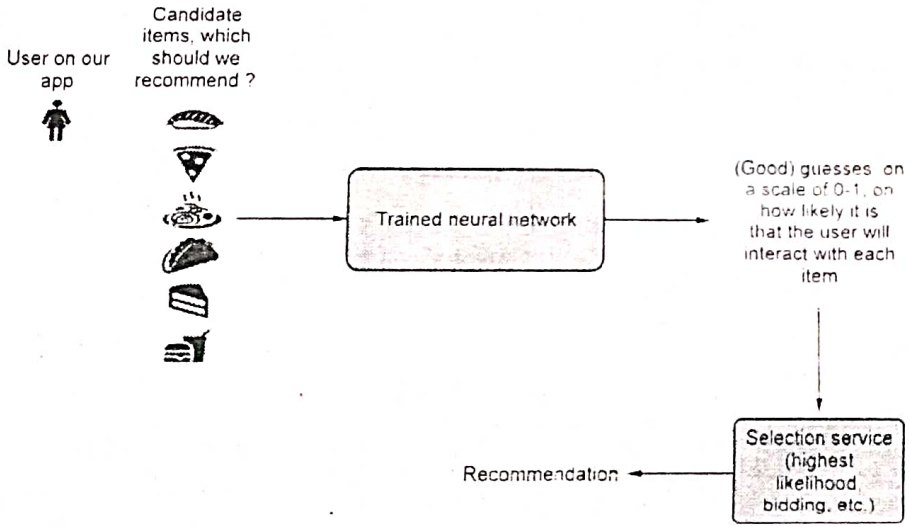

Fig. 6.5.3 Deep learning for recommendation : Inference phase

- During training phase, the system is presented with examples of past interactions or non interactions between users and items. During training model learns to predict probabilities of user-item interactions.
- When the model achieve sufficient level of accuracy of making predictions it is deployed as a service to make inference about the likelihood of new interactions.
- Data utilized at inference phase is different than during the training phase. It is shown in Fig. 6.5.4.
  - **Candidate generation :** Based on the user-item similarity it has learned, user is paired with hundreds or thousands of candidate items.
  - **Candidate ranking :** Rank the likelihood that the user enjoys each item.

- **Filter :** Show the item which user has rated as most likely to enjoy.
- Show the user the item they are rated most likely to enjoy.

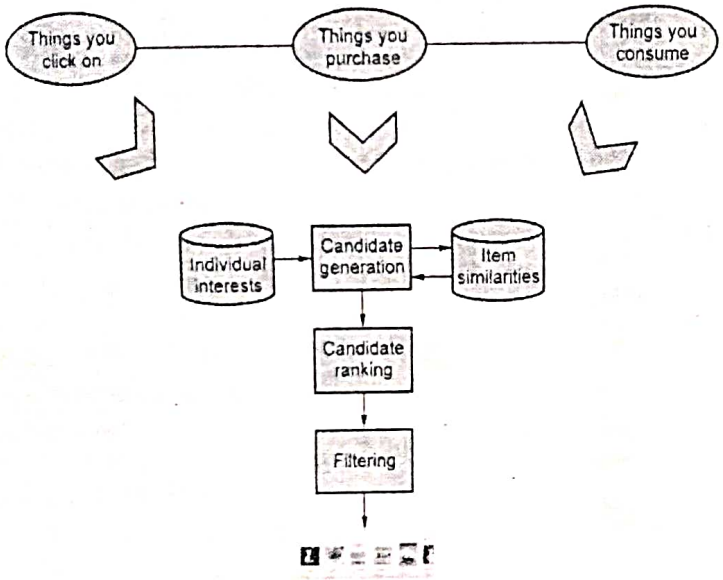

Fig. 6.5.4 Candidate generation, ranking and filtering during inference phase

- Deep learning based recommender systems are build using different variations of ANN such as feed forward network, CNN, RNN, autoencoders etc. Deep learning based recommender systems can cope up with complex interaction patterns and reflect the user's preferences precisely. Such deeper insights cannot be possible with traditional content based and collaborative filtering models as these are relatively linear systems.
- Convolutional neural networks are good at multimedia data like image, text, audio, video processing. Cold start problem in case of collaborative filtering can be overcome by using CNNs. CNN is useful for non Euclidean data e.g. knowledge graph, protein-interaction networks, pinterest recommendations.
- Recurrent neural networks are suitable for sequential data processing. Session based recommendation system without user identification can be build with the help of RNNs. Such systems can also predict what the user may choose to buy next based on their click history.

- Uninformative contents can be filter out by applying an attention mechanism to the recommender system to choose the most representative items. Neural attention models can be integrated with DNNs or CNNs .

6.6 Natural Language Processing

- Natural Language Processing (NLP) is a technology used by computer or a machine to understand , manipulate , analyze and interpret human languages as it is spoken and written. Human languages spoken / written (e.g. English, Hindi etc) are referred as natural language. NLP falls under area of Artificial Intelligence (AI).
- NLP finds applications in machine translation, dialogue generation, automatic summarization, relationship extraction, Named entity recognition (NER). Wide range of applications like Voice assistants Alexa, Siri, powerful search engine of Google are possible due to NLP based systems.
- NLP applications use language models. Probability distribution over sequences of characters, words or bytes in a natural language defines the language model.
- NLP has two phases 1) Data preprocessing and 2) Algorithm development.
  - **Data preprocessing :** In preprocessing text data is cleaned and features in text data are highlighted so as to make it suitable to analyze and process by machine Preprocessing can be done by, Tokenization, Stop word removal, Lemmatization and Speech tagging.
  - **Algorithm development :** After preprocessing the data , NLP algorithm is developed to process it. Two main types of algorithms used for NLP are,
    1. **Rule based system :** It uses carefully designed linguistic rules of a language
    2. **Machine learning based system :** It uses statistical methods. Models learns to perform the task from the training data provided. NLP algorithms can design their own rules by using combination of machine learning, neural network and deep learning through repeated processing and learning.

6.6.1 Deep Learning for Natural Language Processing

- Deep learning offers advantages in learning multiple levels of representation of natural language. Traditional methods use hand crafted features to model NLP tasks. As linguistic information is represented with sparse representation i.e. high dimensional features , there is a problem of curse of dimensionality.

- But with word embeddings that use low dimensional and distributed representations, neural network based NLP models have achieved superior performance as compare to traditional SVM and logistic regression based models.
- Most important advantage of using deep learning for NLP is that it learns multiple levels of representation in increasing order of complexity / abstraction. The lower level representation can be shared across tasks.

6.6.2 Convolutional Neural Network (CNN) based Framework for NLP

- CNN can be used to constitute words or n-grams for extracting high level features. Fig. 6.6.1 shows the CNN based framework. Here the words are transformed into vector representation through look-up table. This results in word embedding approach where weights are learned during training of network.

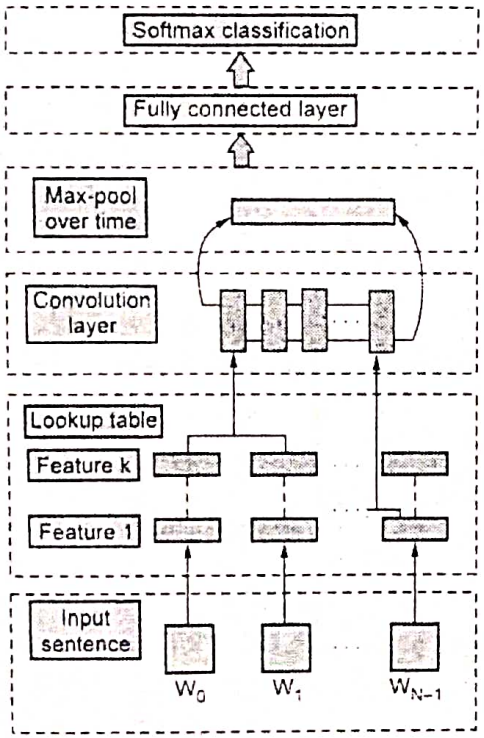

Fig. 6.6.1 CNN based framework for NLP

- The steps to perform sentence modeling with CNN are as follows :
  1. Sentences are tokenized into words. Then it is further transformed into word embedding matrix of dimension 'd' . This forms the input embedding layer.

  2. Convolutional filters are applied to this input layer of word embeddings to produce feature map.
  3. Then max pooling is applied to each filter. This reduces the dimensionality of the output and produce fixed length output. Thus the final sentence representation is created.
- The drawback of CNN is that it cannot handle long distance contextual information and also inefficient in preserving sequential order of context. Therefore Recurrent Neural Networks (RNN) are more suitable for this.

6.6.3 Recurrent Neural Network (RNN) based Framework for NLP

- RNN are effective for sequential data processing. In RNN computation is recursively applied to each instance of input sequence from previous computed results. Recurrent unit is sequentially fed with the sequences represented by fixed size vector of tokens. RNN based framework is shown in Fig. 6.6.2 .

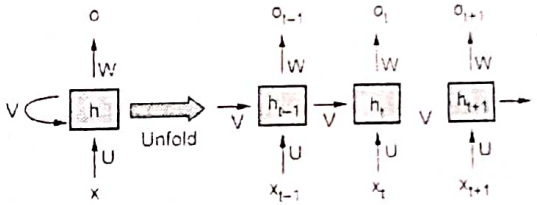

Fig. 6.6.2 RNN based framework for NLP

- The advantage of RNN is that it can memorize the results of previous computation and utilize that information in current computation. So it is possible to model context dependencies in inputs of arbitrary length with RNN and proper composition of input can be created.
- Mainly RNNs are used in different NLP tasks like
  1. Natural language generation (e.g. image captioning, machine translation, visual question answering)
  2. Word - level classification (e.g. Named Entity recognition (NER))
  3. Language modeling
  4. Semantic matching
  5. Sentence-level classification (e.g., sentiment polarity)

6.6.4 Applications of Natural Language Processing

Some common applications of NLP are described below :

1. **Question answering :** Developing the system to answer the questions asked by human beings in a natural language automatically.
2. **Sentiment analysis :** Also called as opinion mining. It analyses the behaviour, attitude and emotional state of the sender on web.
3. **Machine translation :** Translate speech or text in one natural language to another natural language (e.g. Google Translator).
4. **Speech recognition :** It converts spoken words into text. It is useful in applications like dictation to M/S word, voice user interface, voice biometrics etc.
5. **Chatbot :** It is used by many organization and companies to provide chat service to customers.
6. **Automatic document summarization :** It is a process of creating a summary of relevant information from longer text document. The summary is fluent, accurate and short and retains the important contents of the document.

Review Questions

1. *Summarize different application areas where deep learning can be used. What are advantages of using deep learning over conventional methods ?*
2. *Explain what is image classification. What are different image classification techniques ?*
3. *State the advantages of using deep learning for image classification.*
4. *State applications of image processing.*
5. *Explain how CNN can be used for image classification ?*
6. *What are different terminologies used in social network analysis ?*
7. *What are different types of centrality measures used in social network analysis ?*
8. *State and explain network level measures used in social network analysis.*
9. *State applications of social network analysis.*
10. *State different approaches of deep learning based social network analysis. Explain any one of them..*
11. *Explain basic architecture of speech recognition system.*
12. *How traditional speech recognition system works ? What are drawbacks of traditional speech recognition system ?*
13. *Write short note on RNN transducer.*
14. *Why RNN is suitable for speech recognition? How bidirectional RNNs are used in automatic speech recognition ?*

15. *Explain different types of recommender systems.*
16. *Explain deep learning based recommender system.*
17. *What are advantages of using deep learning for NLP ?*
18. *Explain CNN based framework for NLP.*
19. *Explain RNN based framework for NLP.*
20. *State applications of NLP.*

❑❑❑