3 Recurrent Neural Networks

Unit - III

Syllabus

At the end of this unit, you should be able to understand and comprehend the following syllabus topics:

- 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

3.1 Recurrent Neural Networks (RNNs)

You have already learnt about Recurrent Neural Networks in "Neural Network (NN) architecture" in Unit 1, Section 1.3 . You are good!

3.1.1 Types of Recurrent Neural Networks

- Earlier you had learnt about simple RNN and deep RNN. Let's learn about some more types of RNNs.

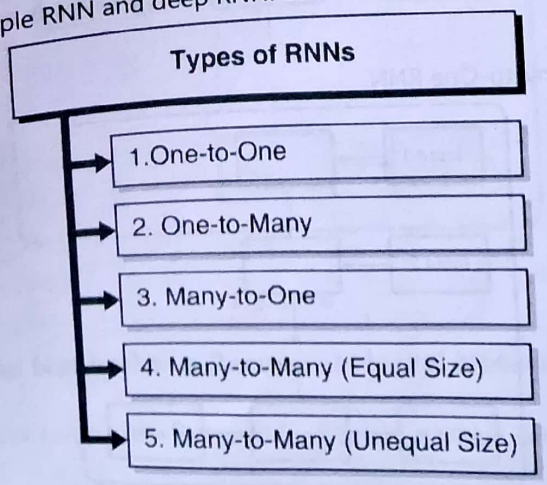


Fig. 3.1.1

1. One-to-One RNN

- The simplest type of RNN is One-to-One, which allows a single input and a single output. It has fixed input and output sizes and acts as a traditional neural network. An example of One-to-One RNN application is image classification.
- The Fig. 3.1.2 illustrates One-to-One RNN.

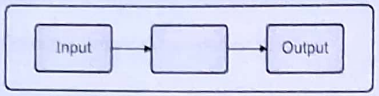


Fig. 3.1.2

2. One-to-Many RNN

- One-to-Many is a type of RNN that gives multiple outputs when given a single input. It takes a fixed input size and gives a sequence of data outputs. Its applications can be found in music generation and image captioning.
- The Fig. 3.1.3 illustrates One-to-Many RNN.

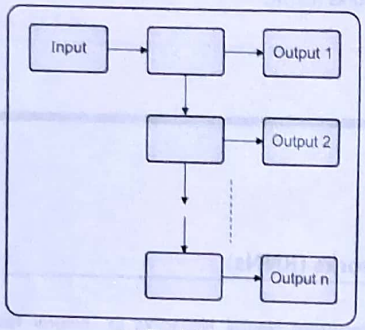


Fig. 3.1.3

3. Many-to-One RNN

- Many-to-One RNN is used when a single output is required from multiple input units or a sequence of them. It takes a sequence of inputs to give a fixed output. Sentiment analysis is a common example of Many-to-One RNN.
- The Fig. 3.1.4 illustrates Many-to-One RNN.

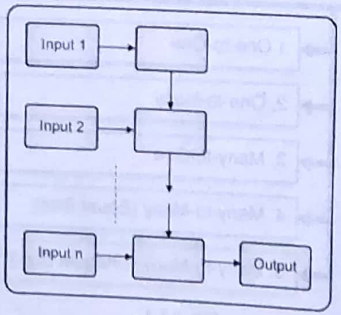


Fig. 3.1.4

Many-to-Many RNN with Equal Unit Size

- Many-to-Many RNN is used to generate a sequence of output data from a sequence of input units. In the case of equal unit size, the number of both the input and output units is the same. A common application of Many-to-Many RNN with equal unit size is Name-Entity Recognition.
- The Fig. 3.1.5 illustrates Many-to-Many RNN with equal unit size.

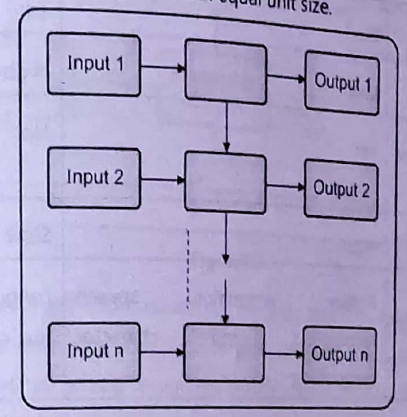


Fig. 3.1.5

Many-to-Many RNN with Unequal Unit Size

- Many-to-Many RNN is used to generate a sequence of output data from a sequence of input units. In the case of unequal unit size, inputs and outputs have different numbers of units. A common application of Many-to-Many RNN with unequal unit size is machine translation.
- The Fig. 3.1.6 illustrates Many-to-Many RNN with unequal unit size.

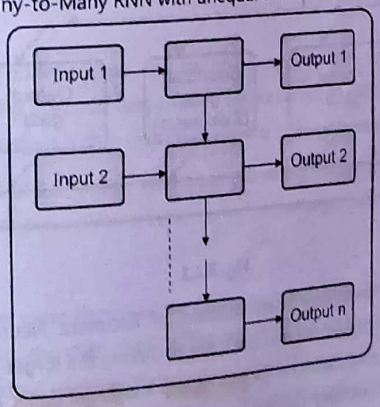


Fig. 3.1.6

3.1.2 Feed-Forward Neural Networks vs Recurrent Neural Networks

- The Table 3.1.1 provides a quick comparison between feed-forward neural networks and recurrent neural networks.

Table 3.1.1

Comparison Attribute	Feed-forward Neural Networks	Recurrent Neural Networks
Signal flow direction	Forward only	Bidirectional
Delay introduced	No	Yes
Complexity	Low	High
Neuron independence in same layer	Yes	No
Speed	High	Slow
Commonly used for	Pattern recognition, speech recognition, and character recognition	Language translation, speech to text conversion, and robotic control

3.2 Long Short-Term Memory Networks (LSTM)

- You previously got a brief about LSTM. Let's dive a little deeper in this section.
- Fig. 3.2.1 shows a simple diagram of a LSTM cell.

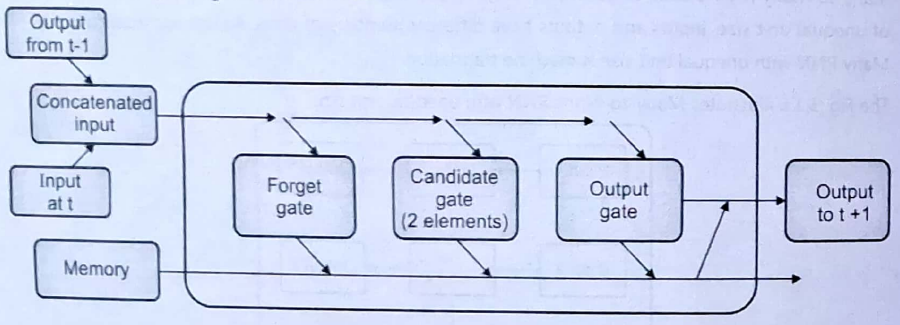


Fig. 3.2.1

- LSTM networks are the most commonly used variation of Recurrent Neural Networks (RNNs). The critical component of the LSTM is the memory cell and the gates (including the forget gate but also the input gate). The contents of the memory cell are modulated by the input gates and forget gates. Assuming that both of these gates are closed, the contents of the memory cell will remain unmodified between one time-step and the next. The gating structure allows information to be retained across many time-steps, and consequently also allows gradients to flow across many time-steps. This allows the LSTM model to overcome the vanishing gradient problem that occurs with most Recurrent Neural Network models.
- The LSTM uses three internal neural networks as respective gates.

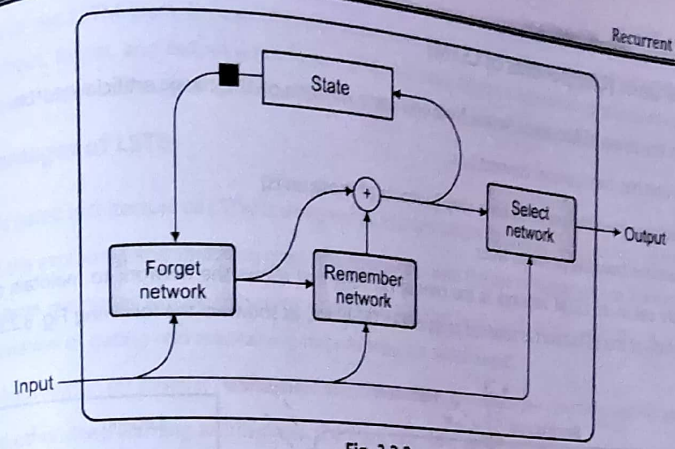


Fig. 3.2.2

The first is used to remove (or forget) information from the state that is no longer needed. The second inserts new information the cell wants to remember. The third network presents a version of the internal state as the cell's output. The convention is that "forgetting" a number simply means moving it toward zero, and remembering a new number means adding it in to the appropriate location in the state memory. The LSTM doesn't require repeated copies of itself, like the basic recurrent cell. So it avoids the problems of vanishing and exploding gradients. You can place this LSTM cell on a layer and train the neural networks inside it using normal backpropagation and optimisation.

3.2.1 LSTM Gates

As you understand, there are three gates used in LSTM.

1. **Forget Gate**

- At forget gate the input is combined with the previous output to generate a fraction between 0 and 1 that determines how much of the previous state need to be preserved (or in other words, how much of the state should be forgotten). This output is then multiplied with the previous state.
- Note: An activation output of 1.0 means "remember everything" and activation output of 0.0 means "forget everything." From a different perspective, a better name for the forget gate might be the "remember gate"!

2. **Input Gate**

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

3. **Output Gate**

- At output gate, the input and previous state are gated as before to generate another scaling fraction that is combined with the output of tanh block that brings the current state. This output is then given out. The output and state are fed back into the LSTM block.

3.2.2 LSTM Units (Components of LSTM)

- The units in the layers of Recurrent Neural Networks are a variation on the classic artificial neuron.
- Each LSTM unit has two types of connections:
  1. Connections from the previous time-step (outputs of those units)
  2. Connections from the previous layer
- The memory cell in an LSTM network is the central concept that allows the network to maintain state over time. The main body of the LSTM unit is referred to as the LSTM block, as shown in the following Fig. 3.2.3.

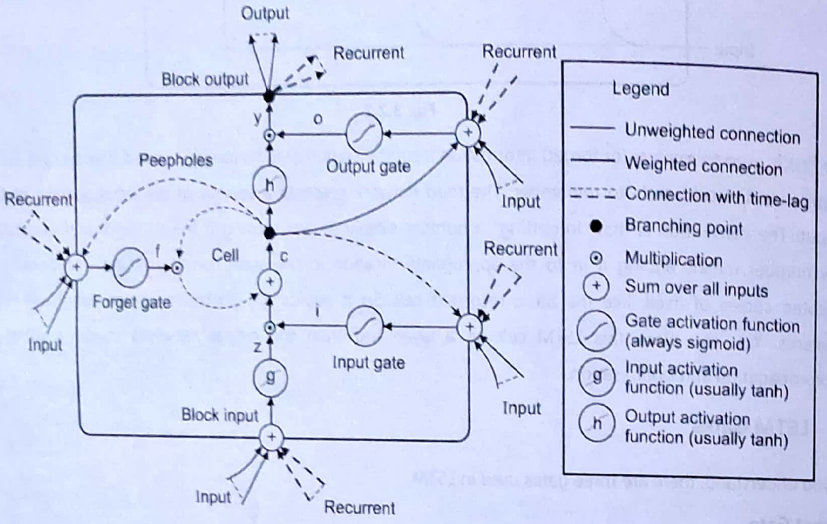


Fig. 3.2.3

- The components in an LSTM unit are as following.
- Three gates
  - Input gate (input modulation gate)
  - Forget gate
  - Output gate
- Block input
- Memory cell
- Output activation function
- Peephole connections
- There are three gate units, which learn to protect the linear unit from misleading signals.
  1. The input gate protects the unit from irrelevant input events.
  2. The forget gate helps the unit forget previous memory contents.
  3. The output gate exposes the contents of the memory cell (or not) at the output of the LSTM unit.

the output of the LSTM block is recurrent connected back to the block input and all of the gates for the LSTM block. The input, forget, and output gates in an LSTM unit have sigmoid activation functions for [0, 1] restriction. the LSTM block input and output activation function (usually) is a tan h activation function.

Advantages of LSTM

The specific gated architecture of LSTM is designed to improve all the following shortcomings of the classical RNN.

1. Avoid the exploding and vanishing gradients, specifically with the use of forget gate at the beginning.
2. Long term memories can be preserved along with learning new trends in the data. This is achieved through combination of gating and maintaining state as separate signal.
3. Prior information on states is not required and the model is capable of learning from default values.
4. Unlike other deep learning architectures, there are not many hyperparameters needed to be tuned for model optimisation.

Encoder Decoder Architectures

Encoder-Decoder models are a family of models which learn to map datapoints from an input domain to an output domain via a two-stage network.

1. The encoder, represented by an encoding function z = f(x), compresses the input into a latent-space representation.
2. The decoder, y = g(z), aims to predict the output from the latent space representation. The latent representation here essentially refers to a feature (vector) representation, which is able to capture the underlying semantic information of the input that is useful for predicting the output.

These models are extremely popular in image-to-image translation problems, as well as for sequence-to-sequence models in natural language processing (NLP) where you can translate say English to French. The Fig. 3.3.1 illustrates the block-diagram of a simple encoder-decoder model.

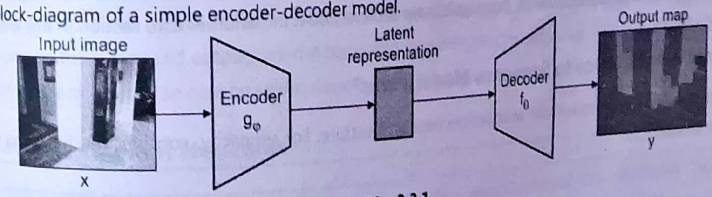
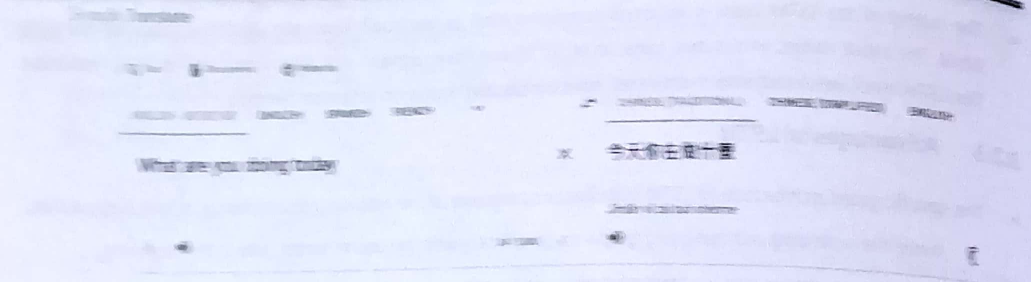


Fig. 3.3.1

These models are usually trained by minimising the re construction loss, L(y, y), which measures the differences between the ground-truth output y and the subsequent reconstruction y'. The output here could be an enhanced version of the image (such as in image de-blurring, or super-resolution), or a segmentation map. Autoencoders are special case of encoder-decoder models in which the input and output are the same. You will learn about autoencoders later in the book.

3.3.1 Sequence to Sequence Model (seq2seq)

A sequence to sequence (seq2seq) model aims to map a fixed-length input with a fixed-length output where the length of the input and output may differ. For example, translating "What are you doing today" from English to Chinese has input of 5 words and output of 7 symbols (今天你在做什麼?).



- Clearly, you cannot use a regular LSTM network to map each word from the English sentence to the Chinese sentence. This is why the sequence to sequence model is used to address problems like this one.
- A sequence to sequence model lies behind numerous systems which you use on a daily basis. For instance, seq2seq model powers applications like Google Translate, voice-enabled devices and online chatbots. Generally speaking, seq2seq has the following applications.

1. **Machine translation :** Machine translation is the task of automatically converting source text in one language to text in another language. Given a sequence of text in a source language, there is no one single best translation of that text to another language. This is because of the natural ambiguity and flexibility of human language. This makes the challenge of automatic machine translation difficult, perhaps one of the most difficult in machine learning.
2. **Speech recognition :** You must have used speech recognition in products such as Amazon Alexa, Apple Siri, and Google Home. Speech recognition helps you to carry out voice-assisted tasks and the machine typically "understands" what you need or are asking for.
3. **Video captioning :** You can automatically generate video captioning based on what is happening in the video or what is being said in various languages. For example, the video could be in English, but you can caption it in Hindi.
4. **Question-Answer problems :** These could be chatbot applications or voice assistants that respond to your questions with probable answers. You would have experienced it in customer care section of any e-commerce application.

3.3.2 How the Sequence to Sequence Model works?

- Sequence to Sequence models use encoder-decoder architecture. For simplicity, consider the following encoder-decoder architecture.

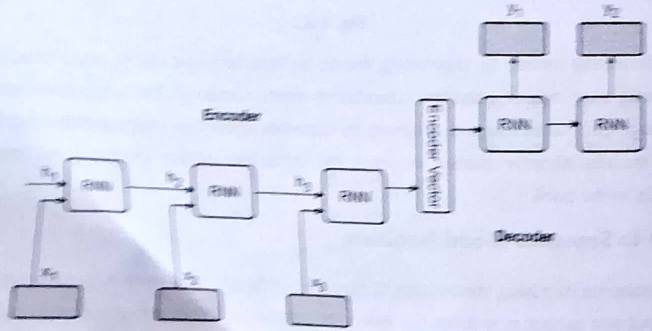


Fig. 3.3.2

...model consists of 3 parts as following.

Encoder : Encoder is a network of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence. It collects information for that element and propagates it forward. In question-answering problem, the input sequence is a collection of all words from the question. Each word is represented as x_i where i is the order of that word. The hidden states h_i are computed using the following formula.

$$h_t = f(W^{hh} h_{t-1} + W^{hx} x_t)$$

Intermediate (encoder) vector : This is the final hidden state produced from the encoder part of the model. It is calculated using the formula above. This vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions. It acts as the initial hidden state of the decoder part of the model.

Decoder : Like encoder, decoder is a also a network of several recurrent units where each predicts an output y_t at a time step t. Each recurrent unit accepts a hidden state from the previous unit and produces and output as well as its own hidden state. In the question-answering problem, the output sequence is a collection of all words from the answer. Each word is represented as y_i where i is the order of that word. Any hidden state h_i is computed using the following formula.

$$h_t = f(W^{hh} h_{t-1})$$

As you can see, you are just using the previous hidden state to compute the next one. The output y_t at time step t is computed using the following formula.

$$y_t = \text{softmax}(W^{hy} h_t)$$

You calculate the outputs using the hidden state at the current time step together with the respective weight $W(S)$. Softmax is used to create a probability vector which will help you determine the final output (e.g. word in the question-answering problem).

The power of this model lies in the fact that it can map sequences of different lengths to each other. As you can see, the inputs and outputs are not correlated, and their lengths can differ. This opens a whole new range of problems which can now be solved using such architecture.

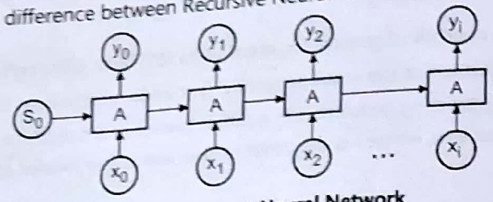
3.4 Recursive Neural Networks

Note: Do not get confused between Recursive Neural Networks and Recurrent Neural Networks (RNN). They have the same abbreviation but are different. To avoid confusion, I would use the Recursive Neural Networks in full for all my references.

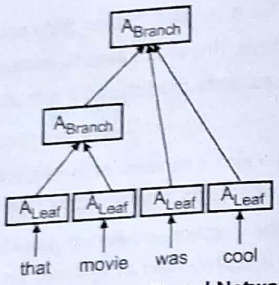
Recursive Neural Networks, like Recurrent Neural Networks, can deal with variable length input. The primary difference is that Recursive Neural Networks have the ability to model the hierarchical structures in the training dataset. It is constructed in such a way that it includes applying the same set of weights recursively over the different tree-like structures. Recursive neural networks generalise the recurrent neural networks from a chain-like structure to a tree-like structure.

For example, images commonly have a scene composed of many objects. Deconstructing scenes is often a problem domain of interest that is nontrivial. The recursive nature of this deconstruction challenges us to not only identify the objects in the scene, but also how the objects relate to form the scene.

- The Fig. 3.4.1 illustrates the difference between Recursive Neural Networks and Recurrent Neural Networks.



(a) Recurrent Neural Network



(b) Recursive Neural Network

Fig. 3.4.1

3.4.1 Network Architecture

- A Recursive Neural Network architecture is composed of a shared-weight matrix and a binary tree structure that allows the recursive network to learn varying sequences of words or parts of an image. It is useful as a sentence and scene parser. Recursive Neural Networks use a variation of backpropagation called backpropagation through structure (BPTS). The feed-forward pass happens bottom-up, and backpropagation is top-down. Think of the objective as the top of the tree, whereas the inputs are the bottom.
- The Fig. 3.4.2 illustrates the network architecture of Recursive Neural Networks.

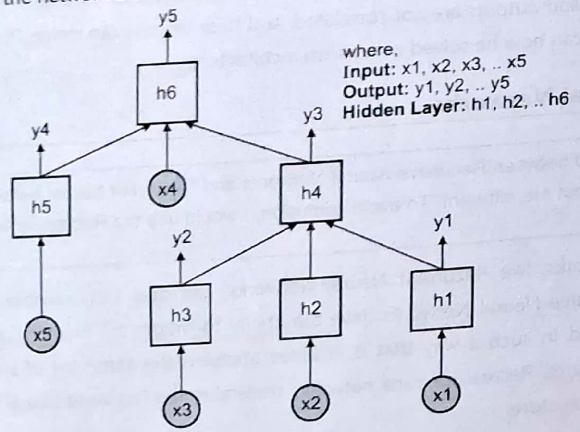


Fig. 3.4.2

- Let's take a simple architecture, such as the following, to understand better.

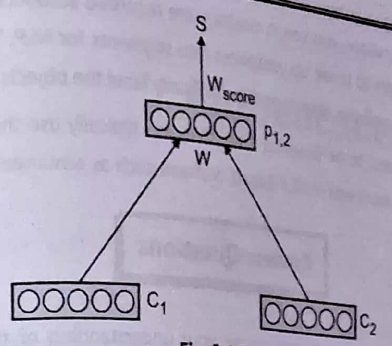


Fig. 3.4.3

In the simplest architecture, nodes are combined into parents using a weight matrix that is shared across the whole network and using a non-linear activation function such as tanh. If c_1 and c_2 are n-dimensional vector representation of nodes, their parent will also be an n-dimensional vector, calculated as following.

$$P_{1,2} = \tanh(W[C_1, C_2])$$

Where W is a learned $n \times 2$ n weight matrix.

This architecture, with a few improvements, has been used for successfully parsing natural scenes, syntactic parsing of natural language sentences, and recursive autoencoding and generative modelling of 3D shape structures in the form of cuboid abstractions.

3.4.2 Types of Recursive Neural Networks

cursive Neural Networks have a few variations

Recursive Autoencoder : Recursive autoencoders learn how to reconstruct the input. In the case of Natural Language Processing (NLP), it learns how to reconstruct contexts. A semi-supervised recursive autoencoder learns the likelihood of certain labels in each context.

Recursive Neural Tensor Network : It computes a supervised objective at each node of the tree. The tensor part of this means that it calculates the gradient a little differently, factoring in more information at each node by taking advantage of another dimension of information using a tensor (a matrix of three or more dimensions).

3.4.3 Applications of Recursive Neural Networks

Both Recursive and Recurrent Neural Networks share many of the same use cases. Recurrent Neural Networks are traditionally used in NLP because of their ties to binary trees, contexts, and natural-language-based parsers. For example, constituency parsers are able to break up a sentence into a binary tree, segmenting it by the linguistic properties of the sentence. In the case of Recursive Neural Networks, it is a constraint that you use a parser that builds the tree structure (typically constituency parsing).

Recursive Neural Networks can recover both granular structure and higher-level hierarchical structure in datasets such as images or sentences. Applications for recursive neural networks include the following.

1. Image scene decomposition
2. Natural Language Processing (NLP)
3. Audio-to-text transcription

- Two specific network configurations you see in practice are recursive autoencoders and recursive neural tensors. You use recursive autoencoders to break up sentences into segments for NLP. You use recursive neural tensors to break up an image into its composing objects and semantically label the objects in the scene.
- Recurrent Neural Networks tend to be faster to train, thus you typically use them in more temporal applications, but they have been shown to work well in NLP-based domains such as sentiment analysis, as well.

Review Questions

Here are a few review questions to help you gauge your understanding of this chapter. Try to attempt these questions and ensure that you can recall the points mentioned in the chapter.

Recurrent Neural Networks

Q. 1 Compare Feed-Forward Neural Networks and Recurrent Neural Networks. **(4 Marks)**

Q. 2 Describe the types of RNNs. **(6 Marks)**

Long Short-Term Memory Networks (LSTM)

Q. 3 With a diagram, explain the general layout of a LSTM cell. **(6 Marks)**

Q. 4 Describe LSTM gates. **(4 Marks)**

Q. 5 Explain the components of a LSTM unit. **(6 Marks)**

Q. 6 Describe the advantages of LSTM. **(4 Marks)**

Encoder Decoder Architectures

Q. 7 Explain encoder decoder architecture. **(4 Marks)**

Q. 8 Write a short note on Sequence to Sequence Model. **(4 Marks)**

Q. 9 Where would you use seq2seq and why? **(4 Marks)**

Q. 10 Explain how seq2seq model works. **(6 Marks)**

Recursive Neural Networks

Q. 11 Explain Recursive Neural Networks. **(6 Marks)**

Q. 12 Explain the network architecture of Recursive Neural Networks. **(6 Marks)**

Q. 13 With a diagram, describe the difference between Recurrent Neural Network and Recursive Neural Network. **(4 Marks)**

Q. 14 Describe the types of Recursive Neural Networks. **(4 Marks)**

Q. 15 Explain the applications of Recursive Neural Networks. **(4 Marks)**

□□□

4 Autoencoders

Unit - IV

Syllabus

At the end of this unit, you should be able to understand and comprehend the following syllabus topics:

- Types of Autoencoders
  - Undercomplete Autoencoders
  - Regularised Autoencoders
  - Sparse Autoencoders
  - Denoising Autoencoders
  - Stochastic Encoders and Decoders
  - Contractive Autoencoders
- Applications of Autoencoders

4.1 Autoencoder

Definition : Autoencoders are artificial neural networks capable of learning efficient representations of the input data, called codings, without any supervision.

These codings typically have a much lower dimensionality (depth of information) than the input data, making autoencoders useful for dimensionality reduction and compression. These codings, or simply, the code is a compact "summary" or "compression" of the input, also called the latent-space representation. For example, MP3 format encodes the audio file to a much smaller size than the raw .wav file. Similarly, JPG image format, when compressing images, preserves the look and feel of the images without compromising a lot on the quality.

More importantly, autoencoders act as powerful feature detectors, and they can be used for unsupervised pre-training of deep neural networks Lastly, they are capable of randomly generating new data that looks very similar to the training data; this is called a generative model. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces.

4.1.1 Basic Concept of Efficient Data Representation

Suppose that I ask you, which of the following number sequences do you find the easiest to memorise?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

- At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you may notice that it follows two simple rules: even numbers are followed by their half, and odd numbers are followed by their triple plus one (this is a famous sequence known as the hailstone sequence). Once you notice this pattern, the second sequence becomes much easier to memorise than the first because you only need to memorise the two rules, the first number, and the length of the sequence. Isn't it better than to memorise the entire series of numbers?
- Now you could argue that that you are an expert in memorising very large sequence of numbers and hence you don't care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be it. But, generally speaking it is the fact that it is hard to memorise long sequences and it is useful to recognize patterns. That's the basic concept behind how autoencoder works to recognise the "hidden" patterns and then later use them to reconstruct the input.

4.2 Components and Architecture of Autoencoders

- An autoencoder consists of 3 components.
  1. **Encoder :** It compresses the input into a latent space representation (the code). The encoder layer encodes the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.
  2. **Code :** This is the compressed input (from encoder) which is fed to the decoder for reconstructing the original input later.
  3. **Decoder :** It decodes the encoded output, in the form of code, back to the original input. The decoded output is a lossy reconstruction of the original input, and it is reconstructed from the latent space representation (the code).The goal is to get an output as identical as was the input.
- Simply speaking, first the encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.
- The Fig. 4.2.1 depicts these components or layers of the neural network in an autoencoder.

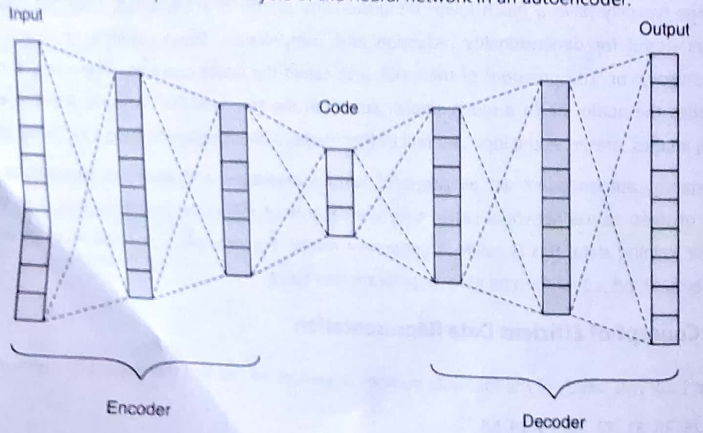


Fig. 4.2.1

simple example of autoencoding in action is illustrated as shown in Fig. 4.2.2

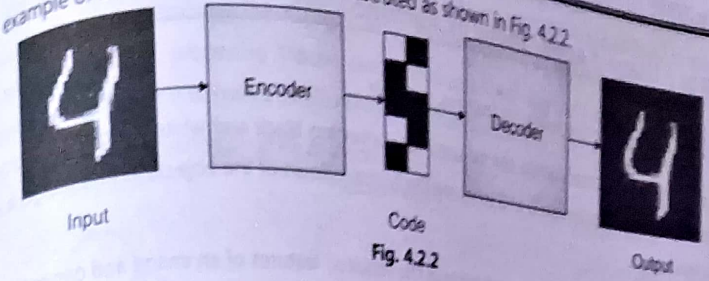


Fig. 4.2.2

the layer between the encoder and decoder, i.e. the code is also known as Bottleneck. This is a well-designed approach to decide which aspects of the observed data are relevant information and what aspects can be discarded.

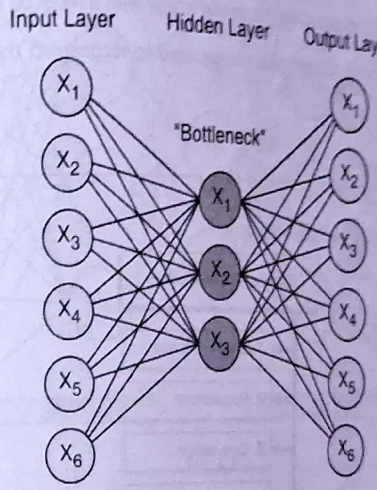


Fig. 4.2.3

3 Training an Autoencoder

Autoencoders are trained the same way as ANNs are trained. You need to set the following four parameters before training an autoencoder.

1. **Code size :** It is the number of nodes in the middle (bottleneck) layer. Smaller size results in more compression and it may be difficult to make the size smaller beyond a certain limit to get satisfactory results.
2. **Number of layers :** As you understand, the autoencoder can be as deep as you like. Very similar to training an ANN, you need to decide how many layers autoencoder should have.
3. **Number of nodes per layer :** You also need to deicide the number of nodes per layer of the autoencoder. Typically, the number of nodes per layer decreases with each subsequent layer of the encoder and increases back in the decoder. Also the decoder is usually symmetric to the encoder in terms of layer structure.
4. **Loss (Cost) function :** You either use mean squared error (MSE) or binary cross-entropy as the loss function. If the input values are in the range [0, 1] then you typically use cross-entropy, otherwise you use the mean squared error.

4.4 Features / Usage / Applications of Autoencoders

- Autoencoders typically have the following features or usage.

1. **Image colouring :** Autoencoders can be used for converting black and white images into coloured images. Depending on what the image is and what are the typical colours of the objects in that image, it is possible to colour the image.
2. **Feature extraction :** Autoencoders extract only the required features of an image and generate the output by removing any noise or unnecessary interruption. They can also be used for compression.
3. **Dimensionality Reduction :** The reconstructed image is similar to the input image but with reduced dimensions (features). It helps in providing the similar image with a reduced number of pixels.
4. **Denoising Image :** A denoising autoencoder can be used to reconstruct the image by eliminating the noise from the input image.

4.5 Types of Autoencoders

- At a high-level, autoencoders are of the following types.

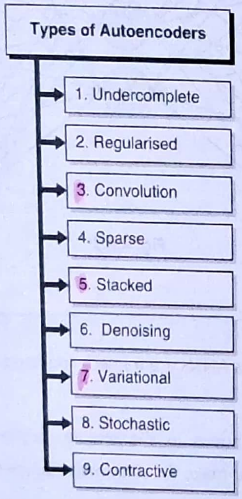


Fig. 4.5.1

4.5.1 Undercomplete Autoencoder

Definition: An autoencoder whose code dimension is less than the input dimension is called undercomplete autoencoder.

- The simplest architecture for constructing an autoencoder is to constrain the number of nodes present in the hidden layer(s) of the network, limiting the amount of information that can flow through the network. By penalising the network according to the reconstruction error, the model can then learn the most important attributes of the input data and how to best reconstruct the original input from an "encoded" state. Ideally, this encoding learns and describes latent attributes of the input data.

...pying the input to the output may sound useless, but you are typically not interested in the output of the ...coder. Instead, you are training the autoencoder to perform the input copying task that will result in hidden ...er h taking on useful properties (obtain useful features). One way to obtain useful features from the ...toencoder is to constrain h to have a smaller dimension than the input x.

...Fig. 4.5.2 illustrates an undercomplete autoencoder.

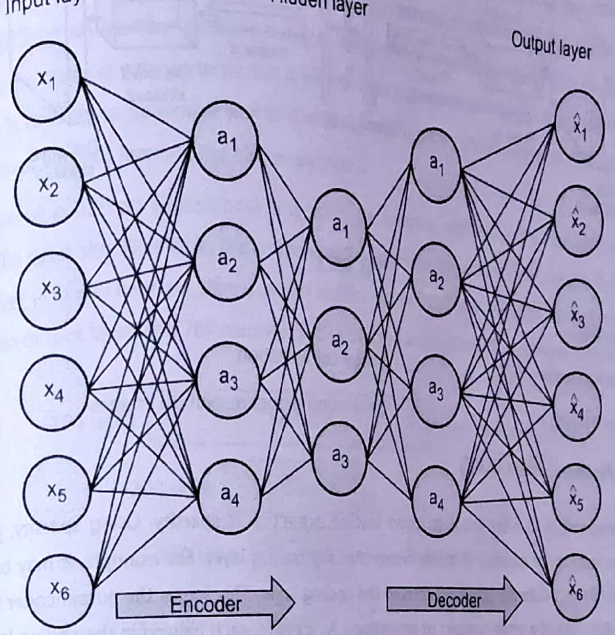


Fig. 4.5.2

4.5.2 Regularised Autoencoder

Ideally, you could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modelled. Regularised autoencoders provide the ability to do so. Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularised autoencoders use a loss function that encourages the model to have other properties besides 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 regularised 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. You will learn about a few regularised autoencoders such as sparse autoencoders and denoising autoencoders subsequently.

4.5.3 Convolution Autoencoders (CAE)

As you know, a signal can be formed as a sum of other signals. Convolutional Autoencoders use the convolution operator to encode the input into a set of simple signals and then try to reconstruct the input from those signals using the convolution network.

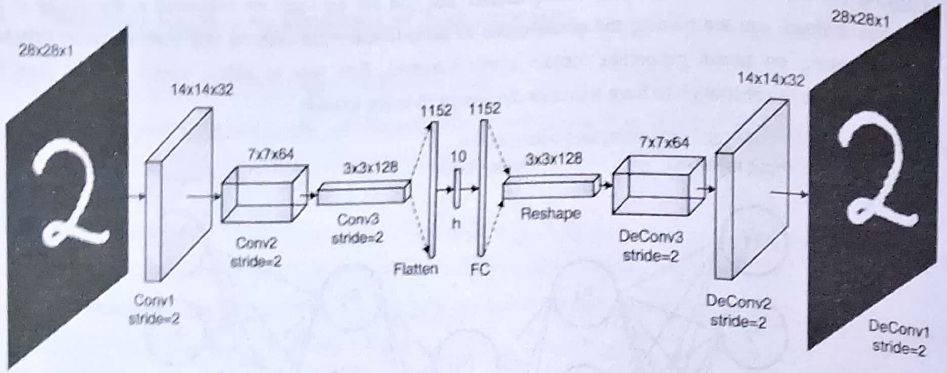


Fig. 4.5.3

- It is typically used for
  1. Image reconstruction
  2. Image colourization
  3. Latent space clustering
  4. Generating higher resolution images

4.5.4 Sparse Autoencoders (SAE)

- One of the constraints that often leads to good feature extraction is sparsity. Using sparsity, you can push the autoencoder to reduce the number of active neurons in the coding layer. For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer. This forces the autoencoder to represent each input as a combination of a small number of activations. As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).

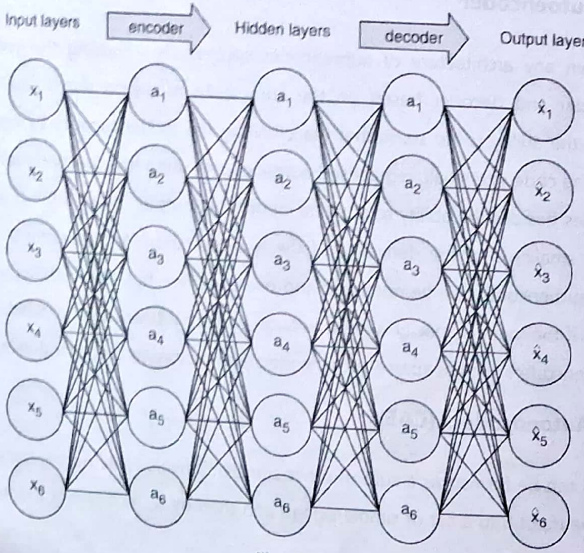


Fig. 4.5.4

Stacked Autoencoders (Deep Autoencoders)

Stacked autoencoders are the typical and the most basic form of autoencoders. Just like other neural networks, autoencoders can have multiple hidden layers. Such autoencoders are called stacked autoencoders or deep autoencoders. Adding more layers helps the autoencoder learn more complex codings. However, you must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have actually learned any useful data representation in the process (and it is unlikely to generalise well to new instances). It is very much similar to you memorising the entire stream of numbers without learning the hidden pattern.

The architecture of a stacked autoencoder is typically symmetrical with regards to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder may have 784 inputs, followed by a hidden layer with 300 neurons, then a central hidden layer of 150 neurons, then another hidden layer with 300 neurons, and an output layer with 784 neurons. Such a stacked autoencoder is illustrated as following.

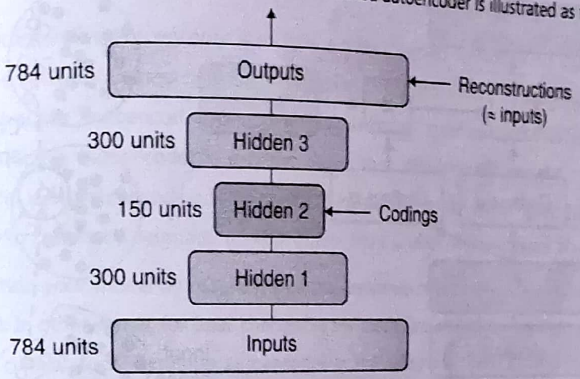


Fig. 4.5.5

4.5.6 Denoising Autoencoders (DAE)

- You can force the autoencoder to learn useful features by adding noise to its inputs and then training it to recover the original noise-free inputs. This prevents the autoencoder from trivially copying its inputs to its outputs and so it ends up having to find patterns in the data. So, you are forcing the autoencoder to subtract the noise and produce the underlying meaningful data. Such an autoencoder is called a denoising autoencoder.
- The Fig. 4.5.6 illustrates a denoising autoencoder.

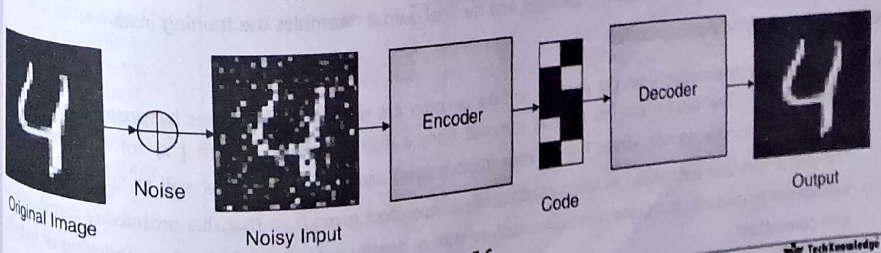


Fig. 4.5.6

4.5.7 Variational Autoencoder (VAE)

- Variational autoencoders are more modern and complex. They are quite different from all the autoencoders that you have learnt so far in the following respect.
  1. They are probabilistic autoencoders, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
  2. Most importantly, they are generative autoencoders, meaning that they can generate new instances that look like they were sampled from the training set.
- Let's understand how it works with the help of a Fig. 4.5.7.

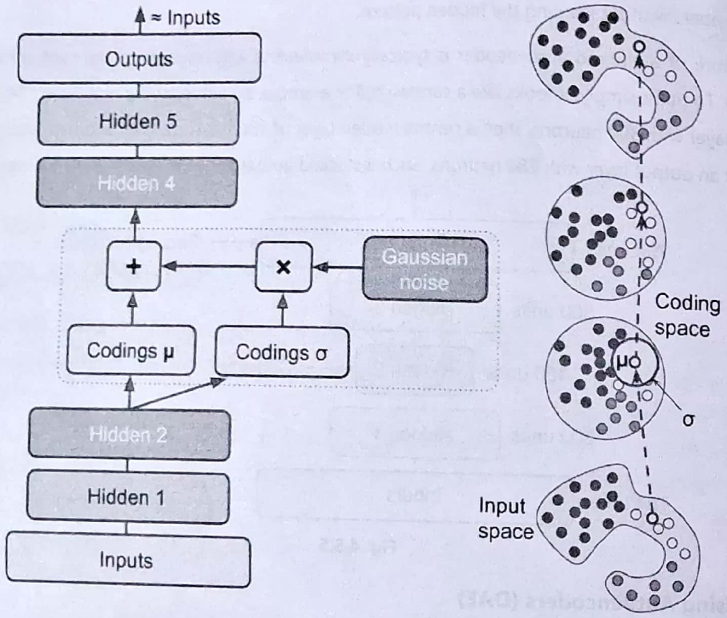


Fig. 4.5.7

- You see the typical structure of an autoencoder but this time with a twist. Instead of directly producing a coding for a given input, the encoder produces a mean coding μ and a standard deviation σ. The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ. After that, the decoder just decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces μ and σ, then a coding is sampled randomly (notice that it is not exactly located at μ), and finally this coding is decoded, and the final output resembles the training instance.

4.5.8 Stochastic Autoencoder

- In stochastic autoencoder, both the encoder and the decoder are not simple functions but instead involve some noise injection. The output can be seen as sampled from a distribution, $P_{encoder}(h \mid x)$ for the encoder and $P_{encoder}(x \mid h)$ for the decoder where h is the code (hidden layer) and x is the input (as well as target for decoder). The output variables are treated as being conditionally independent given h so that this probability distribution is inexpensive to evaluate, but some techniques, such as mixture density outputs, allow flexible modelling of outputs with correlations.

Fig. 4.5.8 illustrates a stochastic autoencoder.

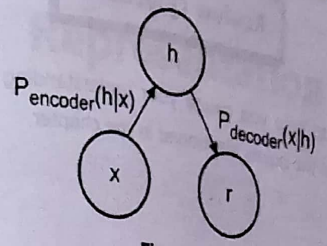


Fig. 4.5.8

So mathematically, you can say that

$$\text{Stochastic } P_{encoder}(h \mid x) = P_{model}(h \mid x)$$

$$\text{Stochastic } P_{decoder}(x \mid h) = P_{model}(x \mid h)$$

Contractive Autoencoder

The objective of a contractive autoencoder is to have a robust learned representation which is less sensitive to small variation in the data. Robustness of the representation for the data is done by applying a penalty term to the loss function. Contractive autoencoder is another regularisation technique just like sparse and denoising autoencoders. Contractive autoencoder is a better choice than denoising autoencoder to learn useful feature extraction. The model learns an encoding in which similar inputs have similar encodings. Hence, you are forcing the model to learn how to contract a neighbourhood of inputs into a smaller neighbourhood of outputs.

You can explicitly train your model by requiring that the derivative of the hidden layer activations are small with respect to the input. In other words, for small changes to the input, you should still maintain a very similar encoded state. This is quite similar to a denoising autoencoder in the sense that these small changes to the input are essentially considered noise and that you would like your model to be robust against noise. Simply speaking, denoising autoencoders make the reconstruction function (decoder) resist small but finite-sized changes in the input, while contractive autoencoders make the feature extraction function (encoder) resist small changes in the input.

The Fig. 4.5.9 illustrates contractive autoencoder.

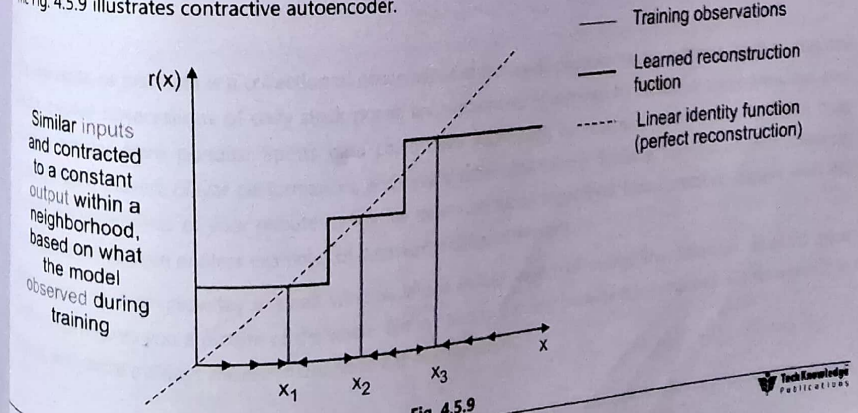


Fig. 4.5.9

Review Questions

Here are a few review questions to help you gauge your understanding of this chapter. Try to attempt these questions and ensure that you can recall the points mentioned in the chapter.

Autoencoder

Q. 1	Write a short note on autoencoder.	(4 Marks)
Q. 2	With a diagram, explain the general architecture of autoencoders.	(6 Marks)
Q. 3	What are some of the parameters that you need to define while training an autoencoder?	(4 Marks)
Q. 4	Explain some of the applications of autoencoders.	(4 Marks)
Q. 5	Write a short note on Undercomplete Autoencoders.	(4 Marks)
Q. 6	Write a short note on Regularised Autoencoders.	(4 Marks)
Q. 7	Describe Convolution Autoencoders.	(6 Marks)
Q. 8	Describe Sparse Autoencoders.	(6 Marks)
Q. 9	Describe Stacked Autoencoders.	(6 Marks)
Q. 10	Describe Deep Autoencoders.	(6 Marks)
Q. 11	Describe Denoising Autoencoders.	(6 Marks)
Q. 12	Describe Variational Autoencoders.	(6 Marks)
Q. 13	Write a short note on Stochastic Autoencoders.	(4 Marks)
Q. 14	Write a short note on Contractive Autoencoders.	(4 Marks)

□□□

5

Unit - V

Representation Learning

Syllabus

At the end of this unit, you should be able to understand and comprehend the following syllabus topics:

- Greedy Layer wise Pre-training
- Transfer Learning and Domain Adaption
- Distributed Representation
- Variants of CNN : DenseNet

1 Representation Learning (Feature Leaning)

Definition : Feature learning or representation learning is a set of techniques that allows a system to automatically discover the representations needed for feature detection or classification from raw data.

It replaces manual feature engineering and allows a machine to both learn the features and use them to perform a specific task. Before you learn more about representation learning, let's quickly learn (recap) about feature engineering so that you understand why representation learning is extremely useful.

1.1 Feature and Feature Engineering

Before diving into feature engineering, let's take a moment and do a recap of what you have learnt so far. Let's take a look at the overall machine learning pipeline.

Data

The raw data, or just data is a collection of observations of real-world phenomena. For instance, stock market data might involve observations of daily stock prices, announcements of earnings by individual companies, and even opinion articles from pundits. Sports data could have information on matches, environment in which those matches were played, player performances, and several other observations. Similarly, personal biometric data can include measurements of your minute-by-minute heart rate, blood sugar level, blood pressure, oxygen level, etc. You can come up with endless examples of data across different domains.

Each piece of data provides a small window into a limited aspect of reality. The collection of all of these observations gives you a picture of the whole. But the picture is messy because it is composed of a thousand little pieces, and there's always measurement noise and missing pieces.

Tasks

Why do you collect data? I am sure you would say that there are several questions that data can help you answer (or predict). Some of the popular questions are

- How likely is that a customer buying product A will also buy product B?
- Which team is likely to win?
- How will be the weather next month?
- What food you should eat to get healthier?
- What is the risk of getting diabetes based on your biometric data?

The path from data to answers is full of false starts and dead ends.

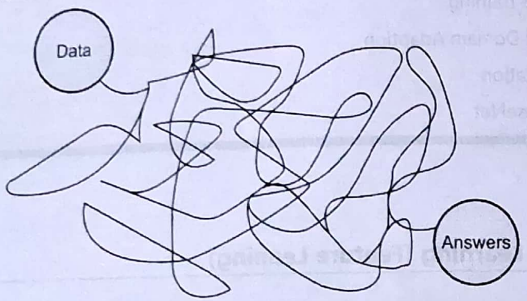


Fig. 5.1.1

- What starts out as a promising approach may not work in reality. What was originally just a hunch may end up leading to the best solution. Workflows with data are frequently multistage and iterative processes. For instance, stock prices are observed at the exchange, aggregated by an intermediary like Thomson Reuters, stored in a database, bought by a company, converted into a Hive store on a Hadoop cluster, pulled out of the store by a script, subsampled, massaged, and cleaned by another script, dumped to a file, and converted to a format that you can try out in your favourite modelling library in R, Python, or Scala. The predictions are then dumped back out to a CSV file and parsed by an evaluator, and the model is iterated multiple times, rewritten in C++ or Java by your production team, and run on all of the data before the final predictions are pumped out to another database.
- However, if you disregard the mess of tools and systems for a moment, you might see that the process involves two mathematical entities that are at the centre of machine learning – models and features.

Models

- Trying to understand the world through data is like trying to piece together reality using a noisy, incomplete jigsaw puzzle with a bunch of extra pieces. This is where mathematical modelling—in particular statistical modelling—comes in. The language of statistics contains concepts for many frequent characteristics of data, such as wrong, redundant, or missing. Wrong data is the result of a mistake in measurement.
- Redundant data contains multiple aspects that convey exactly the same information. For instance, the day of week may be present as a categorical variable with values of "Monday," "Tuesday," ... "Sunday," and again included as an integer value between 0 and 6. If this day-of-week information is not present for some data points, then you have got missing data on your hands.

mathematical model of data describes the relationships between different aspects of the data. For instance, a odel that predicts stock prices might be a formula that maps a company's earning history, past stock prices, and ustry to the predicted stock price. A model that recommends music might measure the similarity between users ased on their listening habits) and recommend the same artists to users who have listened to a lot of the same ngs.

athematical formulas relate numeric quantities to each other. But raw data is often not numeric. For example, the ction "Rohit bought Motorola G9 on Friday", is not numeric. Similarly, product reviews may not be numeric. This is here features come in.

res

arlier, you read that feature is anything that you can measure and build data for. For example, the typical length f various animals. Feature could be numeric, set of characters, Boolean values, or anything else that describes the ata.

ut, for most machine learning mathematic models, features are required to be numeric so that they can be used n various computation.

o, let's redefine features as,

Definition : *A feature is a numeric representation of raw data.*

s you know, the features in a data set are also called its dimensions. So a data set having *n* features is called an n - imensional data set. For example, consider the following data set.

Gender	Marks
Girl	65
Girl	46
Boy	56
Boy	43
Boy	53
Boy	49
Girl	42
Boy	84
Boy	44
Girl	42
Girl	40

How many features or dimensions does it have? Two, right? Yes, this is a two-dimensional data set, or you could also say that this data set has 2 features. I know you are saying out loud that "hey look, the gender field is not numeric". I understand, that is where feature engineering would come in where you would understand how you could convert this raw data set into something more meaningful and computationally more appropriate. For example, you could assign a value of "0" for boys and a value of "1" for girls. Now that is numeric, isn't it?

5.1.2 Feature Engineering

- There are many ways to turn raw data into numeric measurements (I just showed you one earlier), which is why features can end up looking like a lot of things. Naturally, features must derive from the type of data that is available. Features are also tied to the model. Some models are more appropriate for some types of features, and vice versa. The right features are relevant to the task at hand and should be easy for the model to ingest.

✍ ***Definition :*** *Feature engineering is the process of formulating the most appropriate features given the data, the model, and the task.*

- The Fig.5.1.2 depicts where feature engineering sits in the machine learning pipeline.

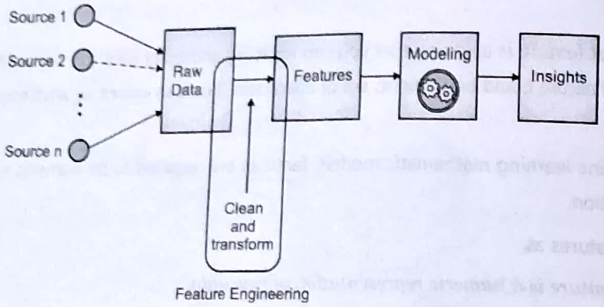


Fig. 5.1.2

- Features and models sit between raw data and the desired insights. In a machine learning workflow, you pick not only the model, but also the features. This is a double-jointed lever, and the choice of one affects the other. Good features make the subsequent modelling step easy and the resulting model more capable of completing the desired task. Bad features may require a much more complicated model to achieve the same level of performance.
- The number of features is also important. If there are not enough informative features, then the model will be unable to perform the ultimate task. If there are too many features, or if most of them are irrelevant, then the model will be more expensive and trickier to train. Something might go wrong in the training process that impacts the model's performance.
- Feature engineering typically includes feature creation, feature transformation, feature extraction, and feature selection.

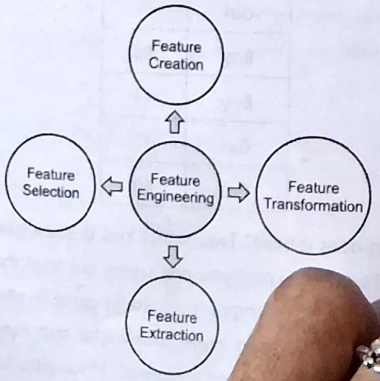


Fig. 5.1.3

Feature creation identifies the features in the dataset that are relevant to the problem at hand.

Feature transformation manages replacing missing features or features that are not valid.

Feature extraction is the process of creating new features from existing features, typically with the goal of reducing the dimensionality of the features.

Feature selection is the filtering of irrelevant or redundant features from your dataset. This is usually done by observing variance or correlation thresholds to determine which features to remove.

At a high-level, the feature engineering process looks like shown in Fig. 5.1.4.

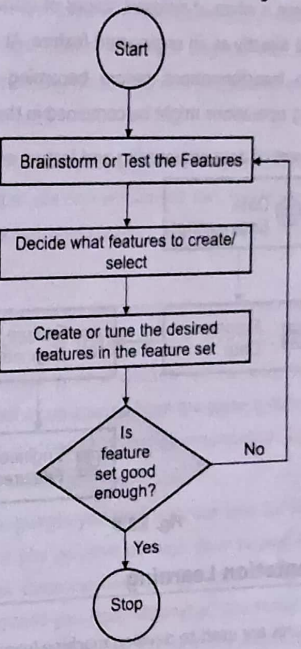


Fig. 5.1.4

Data Engineering -vs- Feature Engineering

Often raw data engineering (data pre-processing) is confused with feature engineering. Data engineering is the process of converting raw data into prepared data. Feature engineering then tunes the prepared data to create the features expected by the machine learning model. These terms have specific meanings as outlined here.

- **Raw data (or just data) :** This refers to the data in its source form, without any prior preparation for machine learning. Note that in this context, the data might be in its raw form (in a data lake) or in a transformed form (in a data warehouse). Transformed data in a data warehouse might have been converted from its original raw form to be used for analytics, but in this context, it means that the data was not prepared specifically for your machine learning task. In addition, data sent from streaming systems that eventually call machine learning models for predictions is considered to be data in its raw form.
- **Prepared data :** This refers to the dataset in the form ready for your machine learning task. Data sources have been parsed, joined, and put into a tabular form. Data has been aggregated and summarized to the right granularity-for example, each row in the dataset represents a unique customer, and each column represents

summary information for the customer, like the total spent in the last six weeks. In the case of supervised learning tasks, the target feature is present. Irrelevant columns have been dropped, and invalid records have been filtered out.

- **Engineered features :** This refers to the dataset with the tuned features expected by the model—that is, performing certain machine learning specific operations on the columns in the prepared dataset, and creating new features for your model during training and prediction. Some of the common examples are scaling numerical columns to a value between 0 and 1, clipping values, and one-hot-encoding categorical features.
- In practice, data from the same source is often at different stages of readiness. For example, a field from a table in your data warehouse could be used directly as an engineered feature. At the same time, another field in the same table might need to go through transformations before becoming an engineered feature. Similarly, data engineering and feature engineering operations might be combined in the same data pre-processing step.

The Fig. 5.2.5, highlights the placement of data engineering and feature engineering tasks.

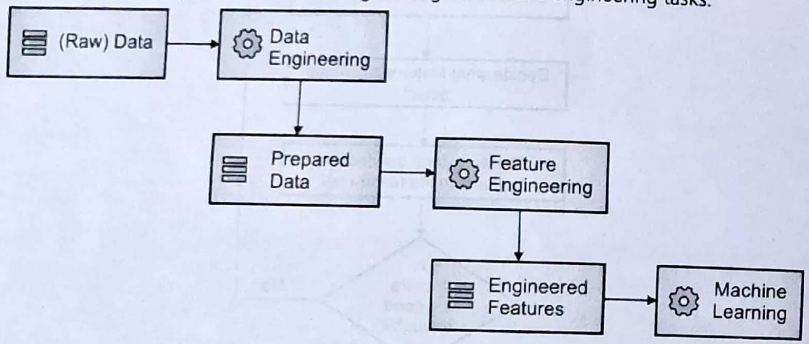


Fig. 5.1.5

5.2 Introduction to Representation Learning

- Now that you understand how features are used to develop machine learning models and why feature engineering is required, let's resume our discussion on representation learning.

What is a Representation?

- Assume that you come across a dish by the name "chicken tortilla soup". As soon as you read the dish name, in your mind, you had some sort of representation about what the words chicken tortilla soup could mean, even though there was no mention of the soup's particular taste, texture, toppings, serving size, appearance, temperature, ingredients, molecular composition, or any other specifics. The 21 letters and spaces making up "chicken tortilla soup" conveyed sufficient information for you to get a fair idea about what could it look and possibly taste like.
- Obviously, those 21 characters are not the soup itself that you can consume and tell what it tastes like. They are a representation of the abstract concept of a chicken tortilla soup. The representation itself consists only of meaningless symbols like "c" and "h" from the Latin alphabet, organised according to the conventions of written English. The concept is an abstract one because it does not contain information specific to concrete examples of the soup -- taste, texture, toppings, etc.

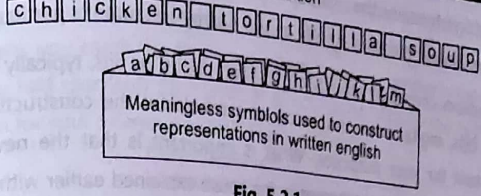


Fig. 5.2.1

The representation makes sense to you only because it is coherent with the patterns of organisation of other representations constructed from the same symbols and conventions. Combinations of the same symbols that are not coherent with the patterns of organisation of written English, such as "ickchenk orttaall pous" and "nekcihc allrrot puos" are meaningless in written English as they don't represent anything (even though they have the exact same letters that chicken tortilla soup had).

If you search online for "chicken tortilla soup," you will find descriptions of and recipes for the soup written in English. Those descriptions and recipes are representations too, made up of the same meaningless symbols from the Latin alphabet, organised into complicated patterns of paragraphs, sentences, phrases, and words according to the conventions of the language.

You could use a different set of conventions for constructing representations (e.g., the conventions of French, German, or Spanish) or even a different set of symbols (e.g., logograms from Chinese Hànzi, or dots and dashes from Morse Code), but the representation of a chicken tortilla soup must be coherent with the patterns of organisation of all other representations constructed from the same symbols and conventions. The entire structure of written language, as it were, rests on those patterns of organisation. Indeed, those patterns must be coherent with each other for us to understand written language.

This is true not only for all written languages you read with our eyes (or fingertips, in the case of Braille), but also for all other kinds of representations you perceive through your senses. For example, the waves of changing air pressure that reach your ears when someone nearby says "chicken tortilla soup" must be coherent with the patterns of organisation of other sounds you hear; otherwise, you would not understand the spoken language. Similarly, the many millions of photons that hit your retinas every second you look at a chicken tortilla soup must be coherent with the patterns of organisation of the many other millions of photons that reach your eyes after bouncing off other visible objects; otherwise, you would not understand what you see.

It must be true for everything you think too. The neuron activations in your brain that represent the thought "chicken tortilla soup" must be coherent with the patterns of organisation of other neuron activations in your brain; otherwise, you would not be able to think of the soup. Hence, you should be able to use other symbols instead of neuron activations to represent your thoughts. The notion might seem far-fetched, but in fact, that is precisely what you do with the meaningless symbols of written language. You represent thoughts with them!

What is Representation Learning?

When you say, "representation learning" (deep or shallow), it means machine learning in which the goal is to learn to transform data from its original representation to a new representation that retains information (features) essential to objects that are of interest to you, while discarding other information (features). The transformation is analogous to the manner in which you transformed neuron activations in your brains, representing a nearby restaurant's chicken tortilla soup, into a new representation consisting of 21 characters in written English. The new

representation in written English retains the essential features of a chicken tortilla soup (the object of interest), but discards other information, including details specific to the nearby restaurant's particular version of the soup.

- Representations in a computer consist of sequences of binary digits, or bits, typically organised into floating-point numbers. The choice of bits, digital ones and zeros, as symbols for the construction of representations inside computers is not important for your purposes. What is important is that the new representations, each one consisting of a sequence of bits, must be coherent in the sense explained earlier with the patterns of organisation of all other bit representations of input data transformed in the same manner.

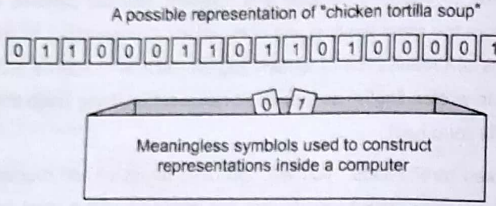


Fig. 5.2.2

- The entire structure of the new representation scheme, as it were, rests on the patterns of organisation of all possible sequences of bits representing different objects in the new scheme. Indeed, those patterns must be coherent with each other for there to be a representation scheme.

Need for Representation Learning

- The performance of machine learning methods is heavily dependent on the choice of data representation (or features) on which they are applied. For that reason, much of the actual effort in deploying machine learning algorithms goes into the design of pre-processing pipelines and data transformations that result in a representation of the data that can support effective machine learning. Such feature engineering is important but labour-intensive and highlights the weakness of current learning algorithms that is their inability to extract and organise the discriminative and useful information from the data. Feature engineering is a way to take advantage of human creativity and prior knowledge to compensate for that weakness.
- In order to expand the scope and ease of applicability of machine learning, it is highly desirable to make learning algorithms less dependent on feature engineering, so that novel applications could be constructed faster. Wouldn't it be nice if you could automatically discover from the dataset what features are important?
- In representation learning, data is sent into the machine, and it learns the representation on its own. It is a way of determining a data representation of the features, the distance function, and the similarity function that determines how the predictive model will perform. Representation learning works by reducing high-dimensional data to low-dimensional data, making it easier to discover patterns and anomalies while also providing a better understanding of the data's overall behaviour.

How Deep Neural Networks Learn Representations (How Representation Learning Works)

- As you know, you train deep learning neural networks to predict something for a given new input. Any deep neural network learns to make predictions via an iterative training process, during which you repeatedly feed sample input data and gradually adjust the behaviour of all layers of neurons in the neural network so that they jointly learn to transform input data into good predictions. You get to decide what those predictions should be and also specify precisely how to measure their goodness to induce learning.

During training, each layer of neurons learns to transform its inputs into a different representation. This representation is itself an input to subsequent layers, which, in turn, learn to transform it into yet other representations. Eventually, you reach the final layer, which learns to transform the last representation into good predictions (however you had specified). Each layer, in effect, learns to make the next layer's job a little easier. This is simply illustrated again for your reference in the following diagram.

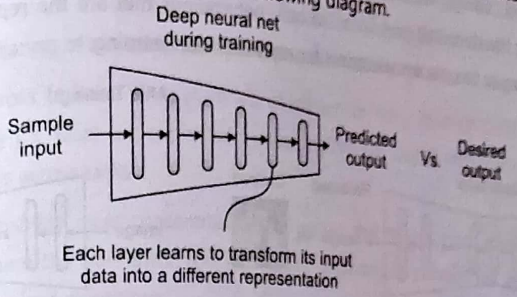


Fig. 5.2.3

So, how can you make a deep neural network learn representations and output that instead of predictions? Simple - you discard the predictions and just keep the representations that it learnt during the course of training. It outputs representations instead of predictions when presented with new input data. For example, let's say that you have a feed forward neural network that has already learned to make good predictions (however you specified them during training). You then slice off its last layer, as shown in the following figure, and that's it! You end up with a slightly shorter neural network, which, when fed input data, outputs an abstract representation of it instead of prediction.

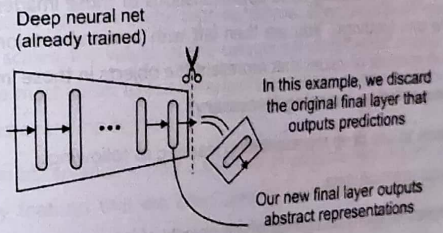


Fig. 5.2.4

You can use the representations produced by this shorter neural network for a purpose different than the prediction objectives you specified to induce learning in the training phase. For example, you could feed the representations as inputs to a second neural network for a different learning task. The second neural network would benefit from the representational knowledge learned by the first one assuming that the prediction objectives you specified to train the first neural network induced learning of representations useful for the second one.

If the main or only purpose of training a neural network is to learn useful representations, then the discarded prediction objectives are more accurately described as training objectives. This is the essence of deep representation learning. The training objectives you specify to induce learning in the training phase, along with the architecture of a neural net (e.g., number, sizes, types, and arrangement of layers), determine what kinds of representations the neural net learns. Your job is to come up with training objectives and suitable architectures that together induce learning of the kinds of representations you want as a by-product. The goal is not to make predictions per se, but to learn useful representations of input data.

Example – Learning Representations of Objects in Images by Predicting Reconstructions

- Let's take a simple example of representation learning to understand how it works. Assume that you have an autoencoder. As you know, autoencoders are artificial neural networks capable of learning efficient representations of the input data, called codings, without any supervision. You train autoencoders as you would do for a regular autoencoder but drop the decoding part so as to only get codings that are the representations. The following diagram illustrates how you can use autoencoders for representation learning.

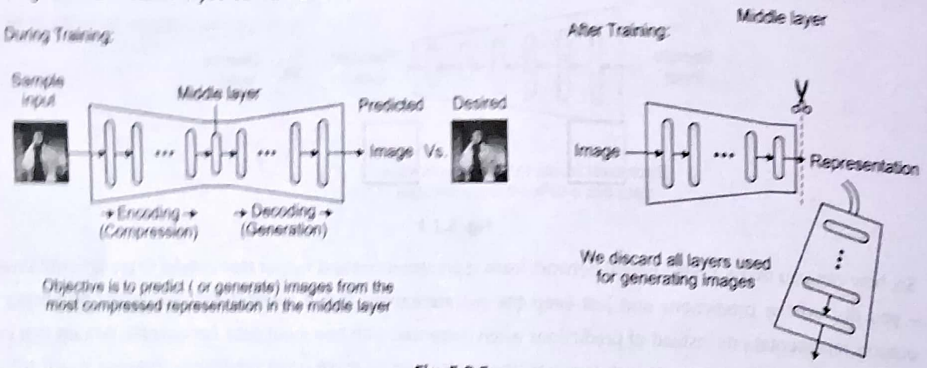


Fig. 5.2.5

- When you train an autoencoder with a large number of examples (say, a few million digital photos), it learns to compress input images into small representations in the middle layer that encode patterns of organisation of the portrayed objects necessary for regenerating close approximations of those images. After training, you discard all layers following the middle one (codings). You are then left with a neural network that transforms images with millions of pixels into a small number of values that represent the objects in those images.

Why and When is Deep Representation Learning Necessary?

- There are three major reasons to use deep representation learning as following.
  1. If you don't have enough training data.
  2. If you have zero examples for many categories of the objects of interest.
  3. If your problem requires a model more computationally complex than feasible.
- In all cases, it is assumed that the problem at hand is complicated enough to require deep learning.

What Makes a Representation Good?

- The following priors (factors that are desired/assumed to be present) play a key role in outputting a good representation by learning a function f that maps input x to output representation y. Models may implement one or more of these priors to learn to output representations suited to a specific task.

1) Smoothness → This is perhaps one of the most basic priors present in machine learning where it is assumed that the learned function f is smooth which means small changes in x lead to small changes in f(x). It can also be equivalently stated as $x \approx y$ which implies $f(x) \approx f(y)$
2) Multiple explanatory features → The input data distribution is generated by combination of several underlying features. A model that learns to compactly represent the input by a combination of these features, could potentially generalise without requiring as many examples as there are variations in the underlying function f. This compactness can only be achieved if the features are reused (achieved by parameters being shared) across examples.

3) A hierarchical organisation of explanatory factors → The concepts that are useful for describing the world around us can be defined in terms of other concepts, in a hierarchy, with more abstract concepts higher in the hierarchy, defined in terms of less abstract ones. This assumption is exploited with deep representations.
4) Semi-supervised learning → With inputs x and y to predict, a subset of the factors explaining x's distribution explain much of y, given x. Hence representations that are useful for p(x) tend to be useful when learning p(y | x), allowing sharing of statistical strength between the unsupervised and supervised learning tasks.
5) Shared factors across tasks → With many y's of interest or many learning tasks in general, tasks (e.g., the corresponding p(y | x, task) are explained by factors that are shared with other tasks, allowing sharing of statistical strengths across tasks.
6) Manifolds → Probability mass concentrates near regions that have a much smaller dimensionality than the original space where the data lives. This is explicitly exploited in some of the autoencoder algorithms and other manifold-inspired algorithms.
7) Natural clustering → Different values of categorical variables such as object classes (e.g. cats, dogs) tend to be associated with separate manifolds. Each manifold is composed of learned representation of an object class (say dog, cat). So moving along a manifold tends to preserve the value of a category (e.g. variations of dog when moving on the "dog" manifold). Interpolating across object classes would require going through a low density region separating the manifolds. In essence, manifolds representing object classes tend not to overlap much. This factor is exploited in machine learning.
8) Temporal and spatial coherence → Identifying slowly moving or changing features in temporal/spatial data could be used as a means to learn useful representations. Even though different features change at different spatial and temporal scales, the values of the categorical variables of interest tend to change slow. So this prior can be used as a mechanism to force the representations to change slowly, penalising change in values of categorical variables over time or space.
9) Sparsity → For a given observation x, only a small set of possible features are relevant. This could captured in the representation by features that are often zero or by the fact the extracted features are insensitive to variations of x. Sparse autoencoders use this prior in the form of a regularisation of the representation.
10) Simplicity of Factor Dependencies → If a representation is abstract enough, the features may relate to each other through simple linear dependencies. This can be seen in many laws of physics, and this is the prior that is assumed when stacking a simple linear predictor on top of a learned representation that is rich and abstract enough.

5.3 Greedy Layer-wise Pre-training

- Unsupervised learning played a key historical role in the revival of deep neural networks, enabling researchers for the first time to train a deep supervised network without requiring architectural specialisations like convolution or recurrence. You can call this procedure unsupervised pre-training, or more precisely, greedy layer-wise unsupervised pre-training. This procedure is an established example of how a representation learned for one task (unsupervised learning, trying to capture the shape of the input distribution) can sometimes be useful for another task (supervised learning with the same input domain).

- Greedy layer-wise unsupervised pre-training relies on a single-layer representation learning algorithm such as an RBM (Restricted Boltzmann machine), a single-layer autoencoder, a sparse coding model, or another model that learns latent representations. Each layer is pretrained using unsupervised learning, taking the output of the previous layer and producing as output a new representation of the data, whose distribution (or its relation to other variables, such as categories to predict) is hopefully simpler.
- Greedy layer-wise pre-training is called greedy because it is a greedy algorithm, meaning that it optimises each piece of the solution independently, one piece at a time, rather than jointly optimising all pieces. It is called layer-wise because these independent pieces are the layers of the network. Specifically, greedy layer-wise pre-training proceeds one layer at a time, training the k^{th} layer while keeping the previous ones fixed. In particular, the lower layers (which are trained first) are not adapted after the upper layers are introduced. It is called unsupervised because each layer is trained with an unsupervised representation learning algorithm. However, it is also called pre-training because it is supposed to be only a first step before a joint training algorithm is applied to fine-tune all the layers together. In the context of a supervised learning task, it can be viewed as a regulariser (in some experiments, pre-training decreases test error without decreasing training error) and a form of parameter initialisation.
- It is common to use the word "pre-training" to refer not only to the pre-training stage itself but to the entire two-phase protocol that combines the pre-training phase and a supervised learning phase. The supervised learning phase may involve training a simple classifier on top of the features learned in the pre-training phase, or it may involve supervised fine-tuning of the entire network learned in the pre-training phase. No matter what kind of unsupervised learning algorithm or what model type is employed, in most cases, the overall training scheme is nearly the same. While the choice of unsupervised learning algorithm will obviously affect the details, most applications of unsupervised pre-training follow this basic protocol.
- Greedy layer-wise unsupervised pre-training can also be used as initialisation for other unsupervised learning algorithms, such as deep auto encoders and probabilistic models with many layers of latent variables. Such models include deep belief networks and deep Boltzmann machines.
- The following diagram illustrates pairs of layers active in each stage of training a 4-layer network.

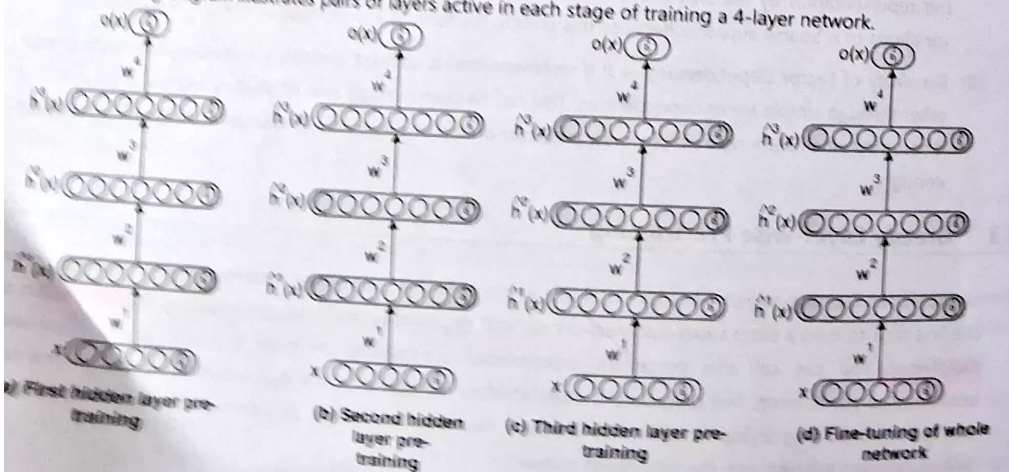


Fig. 5.3.1

Layer-Wise Unsupervised Pre-training Algorithm

The Greedy layer-wise unsupervised pre-training algorithm is given as following.

$f \leftarrow$ Identity function

$\tilde{X} = X$

for $k = 1, \ldots, m$ 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

$f \leftarrow T(f, X, Y)$

end if

Return f

Here,

Unsupervised feature learning algorithm L, which takes a training set of examples and returns an encoder or feature function f.

The raw input data is X, with one row per example, and $f^{(1)}(X)$ is the output of the first stage encoder on X.

In the case where fine-tuning is performed, you use a learner T, which takes an initial function f, input examples X (and in the supervised fine-tuning case, associated targets Y), and returns a tuned function.

The number of stages is m.

Advantages of Unsupervised Pre-training

Following are some of the major advantages of unsupervised pre-training.

1) On many tasks, greedy layer-wise unsupervised pretraining can yield substantial improvements in test error for classification tasks.
2) Unsupervised pre-training is more effective when the initial representation is poor.
3) Unsupervised pre-training is very helpful when the number of labelled examples is very small.
4) Unsupervised pre-training is also useful when the function to be learned is extremely complicated.
5) Neural networks that receive unsupervised pre-training consistently halt in the same region of function space, while neural networks without pre-training consistently halt in another region. The region where pre-trained networks arrive is smaller, suggesting that pre-training reduces the variance of the estimation process, which can in turn reduce the risk of severe over fitting.

Disadvantages of Unsupervised Pre-training

- Following are some of the disadvantages of unsupervised pre-training.

1) It requires two separate training phases and thus each phase requires its own hyperparameters. The performance of the second phase usually cannot be predicted during the first phase, so there is a long delay between proposing hyperparameters for the first phase and being able to update them using feedback from the second phase.

2) It does not offer a clear way to adjust the strength of the regularisation arising from the unsupervised stage. It does not offer a way of flexibly adapting the strength of the regularisation.

3) Today, unsupervised pre-training has been largely abandoned, except in the field of natural language processing, where the natural representation of words as one-hot vectors conveys no similarity information and where very large unlabelled sets are available.

5.4 Transfer Learning and Domain Adaption

Before we proceed, let's understand a few terms.

- Domain – The dictionary meaning of the word "domain" is "a sphere of knowledge, influence, or activity". In mathematics and machine learning, domain is defined as the set of elements to which a mathematical logical variable or function is limited specifically. Technically, given a set of inputs X and corresponding labels Y, a domain D is defined by X and the marginal probability distribution P(X). So, D = {x, P(X)}.
- Task – The dictionary meaning of the word "task" is "assigned piece of work often to be finished within a certain time". In machine learning, task is the work that you need the trained machine learning model to carry out for you. Technically, given a set of inputs X and corresponding labels Y, the task T is defined by Y and the conditional probability distribution P(Y | X). So, T = {y, P(Y|X)}.
- Ok, let me ask you something. If you learn something in maths, do you just use it in maths? Similarly, if you learn English language, do you just use it to attempt your English language exams? No, right ? So, basically, whatever you learn from one domain (set of books), you transfer your learning to other domains (a different set of books or tasks) without you really realising that you have done so. For example, suppose that you learnt to add two numbers in elementary school for a specific addition task that was given to you. Now, once you have mastered the task of addition, you could apply that learning even in your college to solve complex engineering calculations. That is precisely what is transfer learning and domain adaptation is. What you learnt in one domain doing one task, you apply in a different domain doing another task. It saves you time and builds foundation using which you can carry out various tasks across different domains.

Transfer Learning and Domain Adaption in Machine Learning

As you understand, humans have an inherent ability to transfer knowledge across tasks. What you acquire as knowledge while learning about one task, you utilise in the same way to solve related tasks. The more related the tasks, the easier it is for you to transfer, or cross-utilise, knowledge.

However, machine learning and the deep learning algorithms, discussed so far, have been traditionally designed to work in isolation. These algorithms are trained to solve specific tasks. The models have to be rebuilt from scratch once the feature-space distribution changes. Traditional machine learning (ML) trains every model in isolation based on the specific domain, data and task as depicted in the following figure.

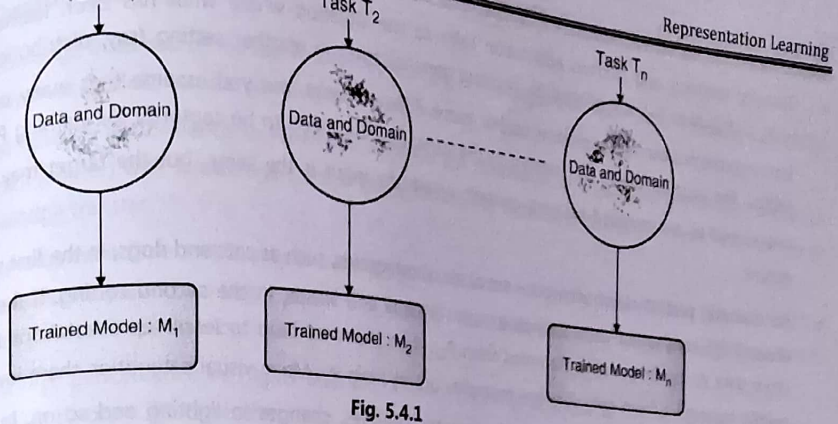


Fig. 5.4.1

Transfer learning is the idea of overcoming the isolated learning paradigm and utilising knowledge acquired for one task to solve related ones. Transfer learning enables you to adapt or transfer the knowledge acquired from one set of tasks and/or domains to a different set of tasks and/or domains. What this means is that a model trained with massive resources, including data, computing power, time, and cost, can be fine-tuned and reused in new settings at a fraction of the original resource requirements. The following diagram illustrates the concept of transfer learning and domain adaptation.

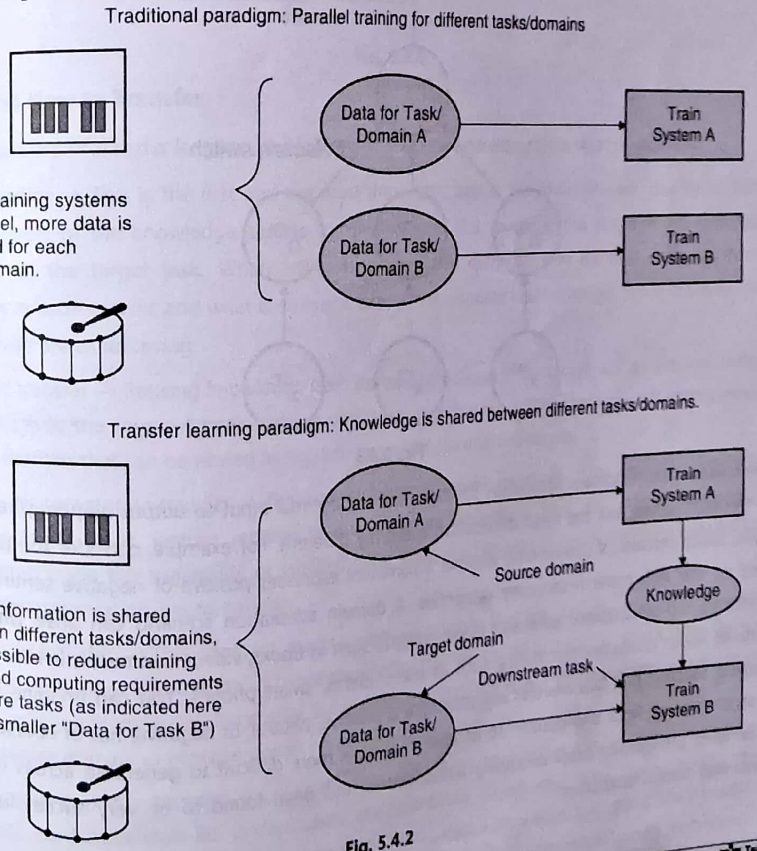


Fig. 5.4.2

- Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (e.g., distribution P_1) is exploited to improve generalisation in another setting (say, distribution P_2). In transfer learning, the learner must perform two or more different tasks, but you assume that many of the factors that explain the variations in P_1 are relevant to the variations that need to be captured for learning P_2. This is typically understood in a supervised learning context, where the input is the same, but the target may be of a different nature.

- For example, you may learn about one set of visual categories, such as cats and dogs, in the first setting, then learn about a different set of visual categories, such as ants and wasps, in the second setting. If there is significantly more data in the first setting (sampled from P_1), then that may help to learn representations that are useful to quickly generalise from only very few examples drawn from P_2. Many visual categories share low-level notions of edges and visual shapes, the effects of geometric changes, changes in lighting and so on. In general, transfer learning, and domain adaptation can be achieved via representation learning when there exist features that are useful for the different settings or tasks, corresponding to underlying factors that appear in more than one setting. This is illustrated in the following figure with shared lower layers and task-dependent upper layers.

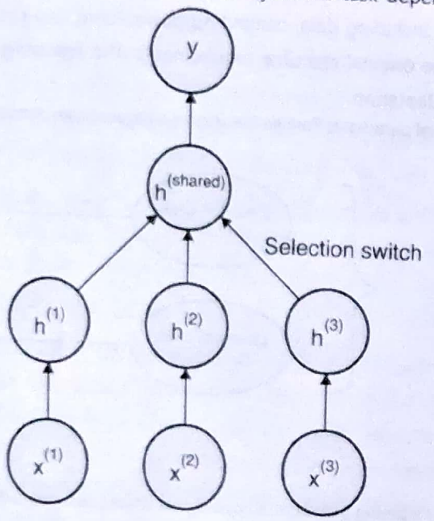


Fig. 5.4.3

- In the related case of domain adaptation, the task (and the optimal input-to-output mapping) remains the same between each setting, but the input distribution is slightly different. For example, consider the task of sentiment analysis, which consists of determining whether a comment expresses positive or negative sentiment. Comments posted on the web come from many categories. A domain adaptation scenario can arise when a sentiment predictor trained on customer reviews of media content, such as books, videos and music, is later used to analyse comments about consumer electronics, such as televisions or smart phones. You can imagine that there is an underlying function that tells whether any statement is positive, neutral, or negative, but of course the vocabulary and style may vary from one domain to another, making it more difficult to generalise across domains. Simple unsupervised pre-training (with denoising autoencoders) has been found to be very successful for sentiment analysis with domain adaptation.

...se knowledge from source models to improve learning in the target task. Apart from providing capabilities ...se already-built models, transfer learning may assist learning the target task in the following ways.

1. Improved baseline performance → When you augment the knowledge of an isolated learner (also known as an ignorant learner) with knowledge from a source model, the baseline performance might improve due to this knowledge transfer.
2. Model-development time → Utilising knowledge from a source model might also help in fully learning the target task, as compared to a target model that learns from scratch. This, in turn, results in improvements in the overall time taken to develop/learn a model.
3. Improved final performance → Higher final performance might be attained by leveraging transfer learning. The following diagram illustrates this.

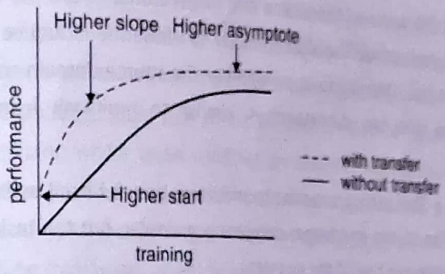


Fig. 5.4.4

...t, When, and How to Transfer

During the process of transfer learning, the following three important questions must be answered.

What to transfer → This is the first and the most important step in the whole process. You try to seek answers about which part of the knowledge can be transferred from the source to the target in order to improve the performance of the target task. When trying to answer this question, you try to identify which portion of knowledge is source-specific and what is common between the source and the target. Some of the approaches on what to transfer are as following.

- Instance transfer → Reusing knowledge from the source domain to the target task is usually an ideal scenario. In most cases, the source domain data cannot be reused directly. Rather, there are certain instances from the source domain that can be reused along with target data to improve results.
- Feature-representation transfer → This approach aims to minimise domain divergence and reduce error rates by identifying good feature representations that can be utilised from the source to target domains. Depending upon the availability of labelled data, supervised or unsupervised methods may be applied for feature-representation-based transfers.
- Parameter transfer → This approach works on the assumption that the models for related tasks share some parameters or prior distribution of hyper parameters. Unlike multitask learning, where both the source and target tasks are learned simultaneously, for transfer learning, you may apply additional weightage to the loss of the target domain to improve overall performance.
- Relational-knowledge transfer → Unlike the preceding three approaches, the relational-knowledge transfer attempts to handle non-IID (Independent and Identically Distributed) data, such as data that is not independent and identically distributed. In other words, data where each data point has a relationship with other data points; for instance, social network data utilises relational-knowledge-transfer techniques.

2) **When to transfer** → There can be scenarios where transferring knowledge for the sake of it may make matters worse than improving anything (also known as negative transfer). You should aim at utilising transfer learning to improve target task performance/results and not degrade them. You need to be careful about when to transfer and when not to.

3) **How to transfer** → Once what and when have been answered, you can proceed toward identifying ways of actually transferring the knowledge across domains/tasks. This involves changes to existing algorithms and using different techniques as you would subsequently learn.

Types of Transfer Learning

- Transfer learning methods can be categorised based on the type of traditional ML algorithms involved as following.

1) Inductive transfer → In this scenario, the source and target domains are the same, yet the source and target tasks are different from each other. The algorithms try to utilise the inductive biases of the source domain to help improve the target task. Depending upon whether the source domain contains labelled data or not, this can be further divided into two subcategories, similar to multitask learning and self-taught learning, respectively.

2) Unsupervised transfer → This setting is similar to inductive transfer itself, with a focus on unsupervised tasks in the target domain. The source and target domains are similar, but the tasks are different. In this scenario, labelled data is unavailable in either of the domains.

3) Transductive transfer → In this scenario, there are similarities between the source and target tasks, but the corresponding domains are different. In this setting, the source domain has a lot of labelled data while the target domain has none. This can be further classified into subcategories, referring to settings where either the feature spaces are different or the marginal probabilities.

Multitask Learning

- Multitask learning is a slightly different flavour of the transfer learning world. In the case of multitask learning, several tasks are learned simultaneously without distinction between the source and targets. In this case, the learner receives information about multiple tasks at once, as compared to transfer learning, where the learner initially has no idea about the target task. It is illustrates in the following figure.

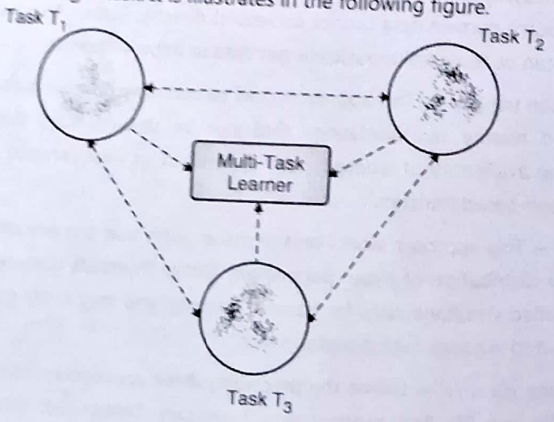


Fig. 5.4.5

...ere are two extreme forms of deep transfer learning.

One-shot learning → Deep learning systems are data hungry by nature, such that they need many training examples to learn the weights. This is one of the limiting aspects of deep neural networks, though such is not the case with human learning. For instance, once a child is shown what an apple looks like, she can easily identify a different variety of apple (with one or a few training examples); this is not the case with ML and deep learning algorithms. One-shot learning is a variant of transfer learning where you try to infer the required output based on just one or a few training examples. This is essentially helpful in real-world scenarios where it is not possible to have labelled data for every possible class (if it is a classification task) and in scenarios where new classes can be added often.

Zero-shot learning → It is another extreme variant of transfer learning, which relies on no labelled examples to learn a task. This might sound unbelievable, especially when learning using examples is what most supervised learning algorithms are about. Zero-data learning, or zero-shot learning, methods make clever adjustments during the training stage itself to exploit additional information to understand unseen data. For example, consider a scenario where three variables are learned, such as the traditional input variable, X, the traditional output variable, y, and the additional random variable that describes the task, T. The model is thus trained to learn the conditional probability distribution of P(y | x, T) Zero-shot learning comes in handy in scenarios such as machine translation, where you may not even have labels in the target language.

Distributed Representation

The concept of distributed representations is often central to deep learning, particularly as it applies to natural language tasks. Those beginning in the field may quickly understand this as simply a vector that represents some piece of data. While this is true, understanding distributed representations at a more conceptual level increases your appreciation of the role they play in making deep learning so effective.

To examine different types of representation, you can do a simple thought exercise. Let's say you have a bunch of "memory units" to store information about shapes. You can choose to represent each individual shape with a single memory unit, as illustrated in the following figure.

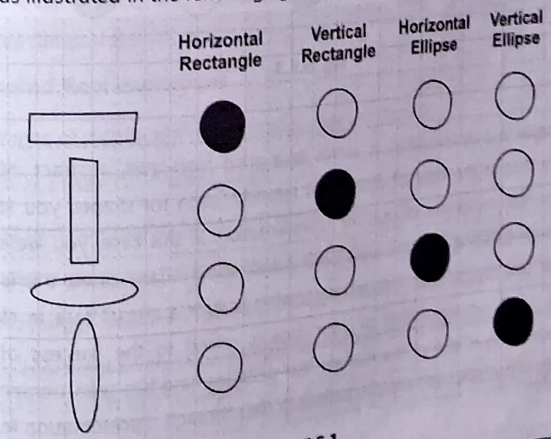


Fig. 5.5.1

- This non-distributed representation, referred to as "sparse" or "local," is inefficient in multiple ways. First, the dimensionality of your representation will grow as the number of shapes that you observe grows. More importantly, it doesn't provide any information about how these shapes relate to each other.
- This is the exactly where distributed representations help. Distributed representations have the ability to capture meaningful "semantic similarity" between data through concepts. The earlier local representation is now distributed as shown in the following figure.

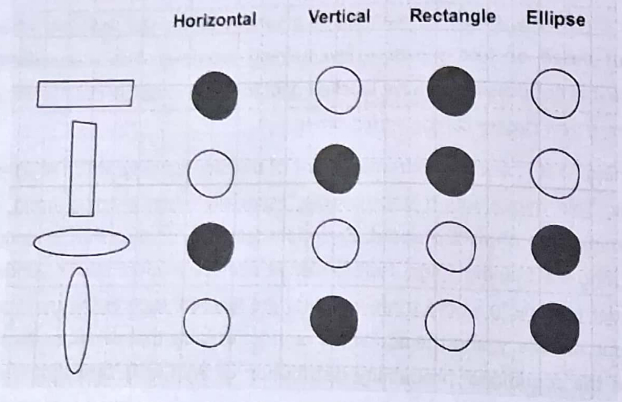


Fig. 5.5.2

- The figure shows a distributed representation of the same set of shapes where information about the shape is represented with multiple "memory units" for concepts related to orientation and shape. Now the "memory units" contain information both about an individual shape and how each shape relates to each other. When you come across a new shape with your distributed representation, such as the circle in the following figure, you don't increase the dimensionality and you also know some information about the circle, as it relates to the other shapes, even though you haven't seen it before.

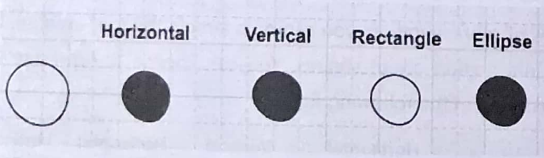


Fig. 5.5.3

Cool, isn't it ?

- While this shape example is oversimplified, it serves as a great high-level, abstract introduction to distributed representations. Notice that in the case of distributed representation for shapes you selected four concepts or features (vertical, horizontal, rectangle, ellipse) for representation. In this case, you were required to know what these important and distinguishing features were beforehand, and in many cases, this is a difficult or impossible thing to know. It is for this reason that feature engineering is such a crucial task in classical machine learning techniques. Finding a good representation of your data is critical to the success of downstream tasks like classification or clustering. This is one of the reasons that deep learning has seen tremendous success is a neural networks' ability to learn rich distributed representations of data through representation learning.

...s take another ... representation for a set of datapoints is as following.

Concept	Representation
Small Red Car	[10000000]
Large Blue SUV	[01000000]
Large Red SUV	[00100000]
Green Apple	[00010000]
Bumble Bee	[00001000]
Tall Building	[00000100]
Small Fish	[00000010]
Banana	[00000001]

As you see, for each data point, you require a separate entry. Learning through these data points have high computational cost as it is a very high dimensionality dataset.

Now, suppose that you could apply distributed representation for the same data point as following.

Concept	Representation
Small Red Car	[0.555 0.761 0.243 0.812]
Large Blue SUV	[0.773 0.309 0.289 0.835]
Large Red SUV	[0.766 0.780 0.294 0.834]
Green Apple	[0.153 0.022 0.654 0.513]
Bumble Bee	[0.045 0.219 0.488 0.647]
Tall Building	[0.955 0.085 0.900 0.773]
Small Fish	[0.118 0.192 0.432 0.618]
Banana	[0.184 0.232 0.671 0.589]

Now, you just have 4 dimensions, and you could have some sort of analysis on resemblance of datapoints with each other on various dimensions.

Advantages of Distributed Representations

Some of the advantages of distributed representations are as following.

1) Continuous values instead of discrete 1's and 0's.
2) Each processing unit contributes to any and all concepts.
3) The representations are dense (vs. localist representations which are sparse).
4) Concepts are no longer localised in one unit (hence are called "distributed").
5) You are able to represent a very large number of concepts using less number of processing units (as opposed to being limited by n units to n concepts).
6) You can learn new concepts without adding new units. All you need is a new configuration of values.

7) Most importantly, you are able to represent similarities better. For example, in the earlier example, Large Red SUV [0.773 0.309 0.289 0.835] and Large Blue SUV [0.766 0.780 0.294 0.834] are much more similar to each other than they are to Small Fish [0.118 0.192 0.432 0.618].

8) Distributed representations provide an efficient way of using parallel hardware to implement best-fit searches

5.6 Variants of CNN : DenseNet

Please Refer Section 2.5 in Chapter 2.

Review Questions

Here are a few review questions to help you gauge your understanding of this chapter. Try to attempt these questions and ensure that you can recall the points mentioned in the chapter.

Representation Learning

Q. 1	What is a representation?	
Q. 2	Write a short note on representation learning.	(4 Marks)
Q. 3	Why do you need representation learning?	(4 Marks)
Q. 4	Explain how representation learning works.	(4 Marks)
Q. 5	How do deep neural networks carry out representation learning?	(6 Marks)
Q. 6	How can you use an auto encoder for representation learning?	(6 Marks)
Q. 7	Explain a few factors that may be used for getting good representations.	(4 Marks)
Q. 8	Write a short note on Greedy Layer-wise Pre-training.	(4 Marks)
Q. 9	Describe the meaning of each word in "Greedy Layer-wise Pre-training".	(4 Marks)
Q. 10	Write Greedy Layer-Wise Unsupervised Pre-training Algorithm.	(4 Marks)
Q. 11	List the advantages of Unsupervised Pre-training.	(4 Marks)
Q. 12	List the disadvantages of Unsupervised Pre-training.	(4 Marks)
Q. 13	What do you mean by transfer learning and domain adaptation?	(4 Marks)
Q. 14	With a diagram, explain the concept of transfer learning in machine learning.	(4 Marks)
Q. 15	With an example, explain the concept of domain adaptation.	(4 Marks)
Q. 16	Describe the advantages of transfer learning.	(4 Marks)
Q. 17	What are the three questions that you must answer during transfer learning?	(4 Marks)
Q. 18	Describe the approaches that you can take to answer "what to transfer" during transfer learning.	(4 Marks)
Q.	Describe the types of transfer learning.	(4 Marks)
Q.	Write a short note on multitask learning.	(4 Marks)
Q.	Explain the types of deep transfer learning.	(4 Marks)
Q.	Write a short note on one-shot learning and zero-shot learning.	(4 Marks)

□□□

6 Applications of Deep Learning

Unit - VI

Syllabus

At the end of this unit, you should be able to understand and comprehend the following syllabus topics:

- Overview of Deep Learning Applications
  - Image Classification
  - Social Network analysis
  - Speech Recognition
  - Recommender system
  - Natural Language Processing

6.1 Applications of Deep Learning

The four broad applications areas of deep learning are as following. Within each area, there are several applications that can be built. Let's learn about them.

6.1.1 Computer Vision

By training deep learning algorithms with millions of labelled images, deep learning neural networks can successfully identify subjects as well or even better than humans, leading to advanced capabilities like rapid facial recognition. Computer vision is the field of deep learning aiming at giving machines the power to analyse and understand images and videos. Computer vision is composed of a broad set of diverse tasks, combined to achieve highly sophisticated applications. The most common tasks in computer vision are image classification, object detection, semantic and instance segmentation, pose estimation and face recognition.

6.1.2 Image Classification

Image Classification is a fundamental computer vision task that attempts to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label automatically just like you would do.

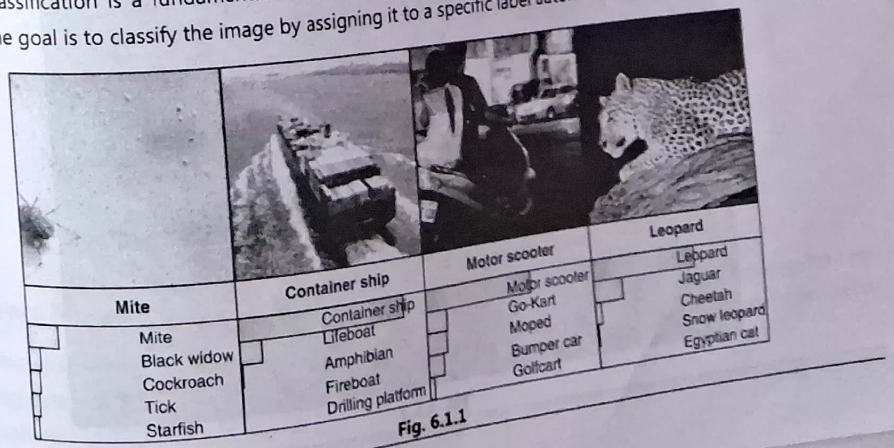


Fig. 6.1.1

- Have you used Google's image search or the modern e-commerce apps that let you point your phone camera to an object, help you identify and search it, and then show you various options for purchasing it? That's precisely what image recognition does.

6.1.3 Object Detection

- Object detection is a computer vision technique for locating instances of objects in images or videos. Object detection algorithms typically leverage deep learning to produce meaningful results. For example, when you look around a street, you can identify traffic lights, various shops, pedestrians walking on the footpath, various types of automobiles, animals, etc. The goal of object detection is to replicate this cognitive task in machines by using artificial neural networks.

Fig. 6.1.2

6.1.4 Semantic and Instance Segmentation

- Semantic segmentation, or image segmentation, is the task of clustering parts of an image together which belong to the same object class.

Fig. 6.1.3

segmentation is the task of detecting each distinct object of interest appearing in an image.

Fig. 6.1.4

6.1.5 Pose Estimation

- Pose Estimation is a general problem in computer vision that attempts to detect the position and orientation of objects within images and videos.

Fig. 6.1.5

6.1.6 Face Recognition

- Face recognition is the task of making a positive identification of a face in an image or video frame against a pre-existing database of faces. You would have seen it at various biometric scanners and automatic attendance register applications.

- Various police departments are also using facial recognition systems at public places such as railway stations, bus stations, toll booths, parks, malls, airports, etc. to detect criminals. Typical facial characteristics are captured through a high definition camera and then those characteristics are matched again a database of faces to make the recognition.

Fig. 6.1.6

6.1.7 *Speech Recognition*

- "Hey Alexa", "Hey Siri", "Ok Google", "Hey Cortana" – you might be very familiar with these terms, aren't you? You are living in an age where you could just speak out to one of the digital voice enabled assistants to get common tasks such as calling someone, playing songs and videos, navigating somewhere, sending text messages, ordering products, and plenty of others.
- Varying speech patterns and accents in humans make speech recognition difficult for computers. Deep learning algorithms can more easily determine what is said. This capability is widely used today in various virtual assistants such as Amazon Alexa, Google Assistant, and Apple Siri.

Fig. 6.1.7

interaction with them with an opportunity to learn more about your voice and accent, thereby providing you a secondary human interaction experience. Virtual assistants use deep learning to know more about their subjects ranging from your dine-out preferences, to your most visited spots or your favourite songs. They learn to understand your commands by evaluating natural human language to execute them. These virtual assistants can also translate your speech to text, make notes for you, and book appointments. Have you used voice to text feature for chat input anytime? It works just great. Virtual assistants are literally at your beck-and-call as they can do everything from running errands to auto-responding to your specific calls to coordinating tasks between you and your team members. With deep learning applications such as text generation and document summarisations, virtual assistants can assist you in creating or sending appropriate email copy as well.

6.1.8 Natural Language Processing

Deep learning helps computers understand regular conversations, where tone and context are critical to communicating unspoken meaning. With algorithms that can detect emotions, automated systems such as customer service bots can understand and respond to users usefully.

Also, a related area of natural language processing is text analysis. Often times, you need to apply analytics on textural features such as product reviews, comments, story line, news reports, etc. For example, you may be developing an application, that when provided an information, can tell whether it is fake or true. How do you relate the information with what is known to be true or fake?

***Definition** : Text analysis, sometimes called text analytics, refers to the representation, processing, and modelling of textual data to derive useful insights.*

An important component of text analysis is text mining, the process of discovering relationships and interesting patterns in large text collections.

6.1.9 Recommendation Engines

An early success of deep learning was the development of systems that track user activity to develop personalised recommendations. By comparing the aggregate activity of numerous users, deep learning systems can even identify totally new items that might interest a user. Various e-commerce websites and apps provide personalised experiences and recommendations based on user's behavioural learning.

6.2 Social Network Analysis (SNA)

The dictionary meaning of graph is "the collection of all points whose coordinates satisfy a given relation". Social networks use graph theory to model pairwise relations between objects.

✍ ***Definition:*** *Social network analysis (SNA) is the process of investigating social structures through the use of networks and graph theory.*

Each graph has nodes (or vertex) and edges.

✍ ***Definition:*** *Vertex or node represents an object in the graph.*

✍ ***Definition:*** *The connection between the nodes is called an edge or a link.*

The nodes in the graph represent the users or objects and the edges represent the relationship between the nodes.

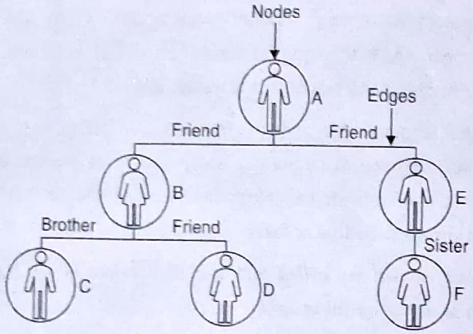


Fig. 6.2.1

Let's understand some of the basic properties of a graph that are used for Social Network Analysis.

1. **Degree of a Node :** Degree of a node is the number of edges it has. For example, degree of node A is 2 whereas degree of node F is 1.
2. **Path Length :** Path length is the distance between two nodes. For example, the path length between node A and node D is 2 (node A→ node B → node D) whereas the path length between node D and node F is 4 (node D → node B → node A → node E → node F).
3. **Centrality :** Centrality refers to the "importance" or "influence" of a particular node within a network. It is often a key node that joins several networks. For example, node A can be treated to have high centrality because it joins the two groups formed by node B and node E.
4. **Density :** Density refers to the proportion of actual direct connections versus total number of direct connections possible in the network. For example, node A is only directly connected to node B and node E. Had it been also directly connected to other nodes (node C, node D and node F), then the network would be considered denser.
5. **Closeness :** Closeness of a node is the average length of the shortest path between the node and all other nodes in the graph.
6. **Betweenness :** Betweenness refers to the number of times a node acts as a bridge along the shortest path between two other nodes. For example, node A is a bridge between node B and node E and also node C and node F.

Need (Applications) of Social Network Analysis

Some of the major applications of SNA are as following.

Identifying the Influencers : In social networks, influencers are people who have the ability to influence potential buyers of a product or service by promoting or recommending the items on social media. Influencer marketing (also known as influence marketing) is a form of social media marketing involving endorsements and product placement from influencers, people and organisations who have a purported expert level of knowledge or social influence in their field.

Influencers are someone with the power to affect the buying habits or quantifiable actions of others by uploading some form of original-often sponsored-content to social media platforms like Instagram, YouTube, Snapchat or other online channels. Influencer marketing is when a brand enrols influencers who have an established credibility and audience on social media platforms to discuss or mention the brand in a social media post. Influencer content may be framed as testimonial advertising.

Social network analysis helps in identifying such influencers based on several criteria such as geography, demographics, topics, etc.

Human Resource Management (HRM) : HRM often strives to identify critical resources and understand their contribution to the organisation flow, collaboration, participation, and information flow. Using SNA, an organisation can optimise the talent connections, productivity, and utilisation. It also helps to identify the reach of an individual, identify accelerators of growth and poorly connected resources, and decide whom to give more opportunity.

Contact tracing : SNA could also be used for contract tracing for infectious diseases (such as Covid-19). SNA could help to identify and isolate individuals and groups with high betweenness and out-degree centrality (transmitters of disease) and implement sound contact tracing activities to reduce the impact and spread.

Identify themes and connections : SNA can also identify dominant themes and relations between keywords and identify the sentiments. For example, the Figs. 6.2.2, from Journal of Medical Internet Search, shows the connection between the top 10 words for COVID-19 themes.

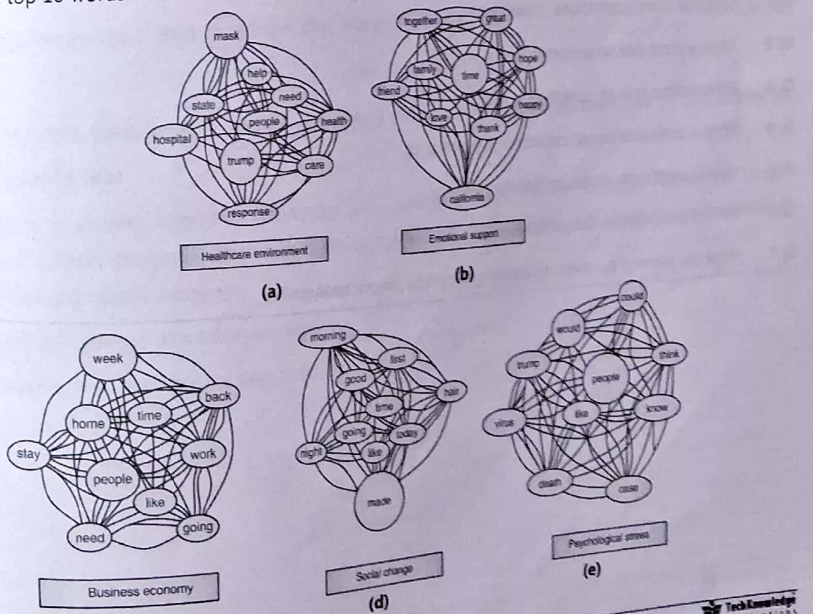


Fig. 6.2.2

Social network centrality measures of the top 10 words on major COVID-19 themes

Table 6.2.1

Themes	Degree	Betweenness	Closeness	Eigenvector
Healthcare environment	18	4.0	0.001885	0.5443
Emotional support	18	3.6	0.009339	0.5824
Business economy	18	1.3	0.000421	0.6495
Social change	18	5.0	0.0002602	0.5315
Psychological stress	18	2.2	0.000656	0.5790

5. **Fraud detection :** Financial organisations can use SNA for fraud detection. Fraud is often organised by groups of people loosely connected to each other. Such a network mapping enables financial institutions to identify customers who may have relations to individuals or organisations on their criminal watchlist (network) and take precautionary measures.

Review Questions

Here are a few review questions to help you gauge your understanding of this chapter. Try to attempt these questions and ensure that you can recall the points mentioned in the chapter.

Applications of Deep Learning

Q. 1 Describe a few applications of deep learning **(6 Marks)**

Q. 2 Write a short note on computer vision. **(4 Marks)**

Q. 3 Write a short note on speech recognition. **(4 Marks)**

Q. 4 Write a short note on natural language processing. **(4 Marks)**

Q. 5 Write a short note on recommendation engines. **(4 Marks)**

Q. 6 Write a short note on Social Network Analysis (SNA). **(4 Marks)**

Q. 7 What are some of the major applications of social network analysis? **(4 Marks)**

❑❑❑

Solved University Question Papers of Oct. 2022 (IN-SEM) and Dec. 2022 (END-SEM)

Oct. 2022 (IN-SEM)

Q. 1(a) Draw and explain the architecture of Multilayered Feedforward Neural network. **(Refer Section 1.3(2))** **(5 Marks)**

Q. 1(b) What is the need of Regularization? Explain Dropout Regularization. **(Refer Sections 1.10.4(B) and 1.10.4(F))** **(5 Marks)**

Q. 1(c) Explain the concept of gradient based Learning. **(Refer Section 1.8)** **(5 Marks)**

OR

Q. 2(a) What is the problem of vanishing Gradient? Describe various solutions to this problem. **(Refer Sections 1.9.4 and 1.9.4(A))** **(7 Marks)**

Q. 2(b) Explain the working of an Artificial neuron. **(Refer Section 1.2)** Also explain the activation functions ReLU and LReLU. **(Refer Sections 1.4.1(5) and 1.4.1(6))** **(8 Marks)**

OR

Q. 3(a) Illustrate Convolution operation in CNN with an example. **(Refer Sections 2.1.1 and 2.1.2)** **(5 Marks)**

Q. 3(b) Explain the use of padding and strides in pooling layers. **(Refer Sections 2.1.2 and 2.1.3)** **(5 Marks)**

Q. 3(c) What is the advantage of weight sharing in CNN. **(Refer Section 2.1.3)** **(5 Marks)**

OR

Q. 4(a) What are pooling layers in CNN? Illustrate Max pooling with an example. **(Refer Section 2.1.4)** **(5 Marks)**

Ans. : Pooling Layers in CNN

Max pooling is a pooling operation commonly used in convolutional neural networks (CNNs) for image processing tasks. It reduces the spatial dimensions of feature maps, providing robustness to small spatial translations and reducing the computational complexity.

Let's illustrate max pooling with a simple example.

Suppose we have a 4x4 input feature map as follows :

Input Feature Map :

2	3	7	5
4	6	2	1
9	8	3	2
5	4	6	7