Sandip University, Nashik (MS), India
At Post Mahiravani, Trimbak Road, Nashik – 422 213, Maharashtra
Phone: +91 9545453092 / Toll- Free: 1800-212-2714
https:// www.sandipuniversity.edu.in /info@sandipuniversity.edu.in

COURSE CODE: 17YCM503

COURSE NAME: ADVANCED NEURAL NETWORK

UNIT-4 SYLLABUS

LECTURE: 3HRS/WEEK

MARKS: 100

OBJECTIVES:

Understand neural network architecture and use the neural networks for classification and regression.
Understand training, verification and validation of neural network models.
Able to demonstrate neural network applications on real-world tasks.
Understand Feed forward networks and Back propagation algorithms.
Able to acquire the knowledge on Deep Learning Concepts.

OUTCOMES

On completion of the course, student will be able to–	
1	Learn neural network architecture to introduce the neural networks for classification and regression.
2	Learn training, verification and validation of neural network models.
3	To demonstrate neural network applications on real-world tasks.
4	Explain Feed forward, multi-layer feed forward networks and Back propagation algorithms.
5	Explain the concept of Deep Learning.

TEXTBOOKS

Text Books
1. Kosko, B, "Neural Networks and Fuzzy Systems: A Dynamical Approach to Machine Intelligence", Prentice Hall, New Delhi, 2004. 2. Neural Networks and Learning Machines. Simon Hykin, 2009.Pearson Education Asia
Reference Books

1. William J. Lawrence. A Pathway to Deep Learning, Machine Intelligence, and Machine Learning. (2023)
2. Vishal Rajput. Ultimate Neural Network Programming with Python. Orange AVA (2023)
3. Neural Networks and Deep Learning: A Textbook by Charu C. Aggarwal. Springer. (2018)
4. Klir G. J. and Folger T.A., "Fuzzy sets, Uncertainty and Information", Prentice–Hall of India Pvt. Ltd., New Delhi, 2008.

SYLLABUS:

Module 1: Autoencoders, Types of Autoencoders: undercomplete autoencoders, contractive autoencoders, sparse autoencoders, denoising autoencoders, Use Cases of Autoencoders

Module 2: Deep Belief Networks, Stacking RBM to create Deep Belief Network, Wake Sleep Algorithm

Sandip University, Nashik (MS), India
At Post Mahiravani, Trimbak Road, Nashik – 422 213, Maharashtra
Phone: +91 9545453092 / Toll- Free: 1800-212-2714
https:// www.sandipuniversity.edu.in /info@sandipuniversity.edu.in

MODULE 1

AUTOENCODERS

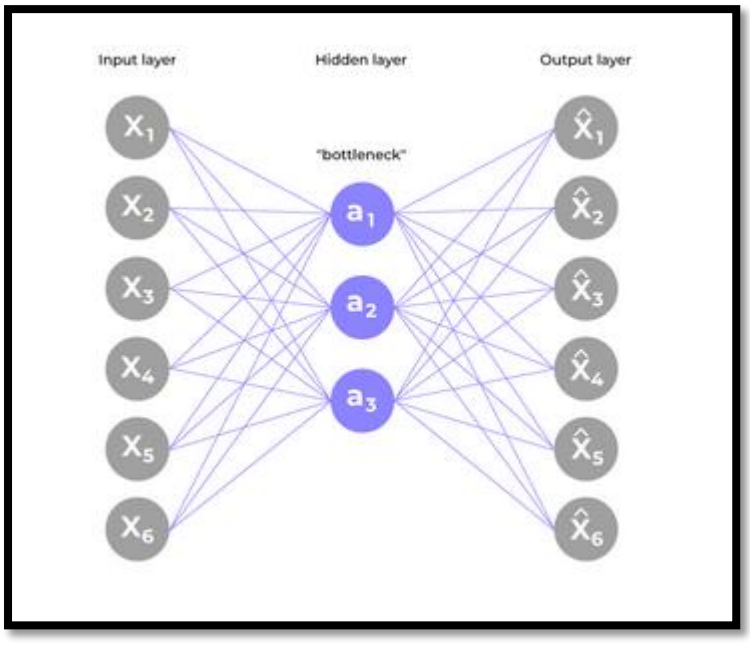
At the heart of **deep learning** lies the neural network, an intricate interconnected system of nodes that mimics the human brain's neural architecture. Neural networks excel at discerning intricate patterns and representations within vast datasets, allowing them to make predictions, classify information, and generate novel insights. **Autoencoders** emerge as a fascinating subset of neural networks, offering a unique approach to unsupervised learning. Autoencoders are an adaptable and strong class of architectures for the dynamic field of deep learning, where neural networks develop constantly to identify complicated patterns and representations. With their ability to learn effective representations of data, these unsupervised learning models have received considerable attention and are useful in a wide variety of areas, from image processing to anomaly detection.

What are Autoencoders?

Autoencoders are a specialized class of algorithms that can learn efficient representations of input data with no need for labels. It is a class of artificial neural networks designed for unsupervised learning. Learning to compress and effectively represent input data without specific labels is the essential principle of an automatic decoder. This is accomplished using a two-fold structure that consists of an encoder and a decoder. The encoder transforms the input data into a reduced-dimensional representation, which is often referred to as "latent space" or "encoding". From that representation, a decoder rebuilds the initial input. For the network to gain meaningful patterns in data, a process of encoding and decoding facilitates the definition of essential features.

Architecture of Auto encoder in Deep Learning

The general architecture of an autoencoder includes an encoder, decoder, and bottleneck layer.



1. **Encoder**
   - Input layer take raw input data
   - The hidden layers progressively reduce the dimensionality of the input, capturing important features and patterns. These layer compose the encoder.
   - The bottleneck layer (latent space) is the final hidden layer, where the dimensionality is significantly reduced. This layer represents the compressed encoding of the input data.

2. **Decoder**
   - The bottleneck layer takes the encoded representation and expands it back to the dimensionality of the original input.
   - The hidden layers progressively increase the dimensionality and aim to reconstruct the original input.
   - The output layer produces the reconstructed output, which ideally should be as close as possible to the input data.

3. The loss function used during training is typically a reconstruction loss, measuring the difference between the input and the reconstructed output. Common choices include mean squared error (MSE) for continuous data or binary cross-entropy for binary data.

4. During training, the autoencoder learns to minimize the reconstruction loss, forcing the network to capture the most important features of the input data in the bottleneck layer.

After the training process, only the encoder part of the autoencoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are: –

- **Keep small Hidden Layers:** If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.

- **Regularization:** In this method, a loss term is added to the cost function which encourages the network to train in ways other than copying the input.

- **Denoising:** Another way of constraining the network is to add noise to the input and teach the network how to remove the noise from the data.

- **Tuning the Activation Functions:** This method involves changing the activation functions of various nodes so that a majority of the nodes are dormant thus, effectively reducing the size of the hidden layers.

If an Autoencoder is provided with a set of input features completely independent of each other, then it would be really difficult for the model to find a good lower-dimensional representation without losing a great deal of information (lossy compression).

Autoencoders can, therefore, also be considered a dimensionality reduction technique, which compared to traditional techniques such as Principal Component Analysis (PCA), can make use of non-linear transformations to project data in a lower dimensional space. If you are interested in learning more about other Feature Extraction techniques, additional information is available in this feature extraction tutorial..

Additionally, compared to standard data compression algorithms like gzpi, Autoencoders can not be used as general-purpose compression algorithms but are handcrafted to work best just on similar data on which they have been trained on.

Some of the most common hyperparameters that can be tuned when optimizing your Autoencoder are:

- The number of layers for the Encoder and Decoder neural networks
- The number of nodes for each of these layers
- The loss function to use for the optimization process (e.g., binary cross-entropy or mean squared error)
- The size of the latent space (the smaller, the higher the compression, acting, therefore as a regularization mechanism)

Finally, Autoencoders can be designed to work with different types of data, such as tabular, time-series, or image data, and can, therefore, be designed to use a variety of layers, such as convolutional layers, for image analysis.

Ideally, a well-trained Autoencoder should be responsive enough to adapt to the input data in order to provide a tailor-made response but not so much as to just mimic the input data and not be able to generalize with unseen data (therefore overfitting).

UNDERCOMPLETE AUTOENCODER

An undercomplete autoencoder is a type of autoencoder of aims to learn a compressed representation of its input data. It is termed "undercomplete" because it forces the representation to have a lower dimensionality than the input itself, thereby learning to capture only the most essential features.

Architectural Overview

An undercomplete autoencoder is typically structured into two main components:

1. **Encoder**: This part compresses the input into a smaller, dense representation. Mathematically, it transforms the input x using weights W and biases b, and an activation function \sigma : h = \sigma(Wx + b) The encoder reduces the dimensionality of the input, preparing a compressed version that retains critical data characteristics.

2. **Decoder**: The decoder part aims to reconstruct the original input from the compressed code as accurately as possible. It mirrors the encoder's structure but in reverse, using potentially different weights W' and biases b': \hat{x} = \sigma'(W'h + b')

Objective Function

The primary goal of training an undercomplete autoencoder is minimizing the difference between the original input x and its reconstruction \hat{x}. This is generally achieved using the mean squared error (MSE) loss function:

L(x, \hat{x}) = \|x - \hat{x}\|^2

Minimizing this loss encourages the autoencoder to learn efficient data representations and reconstruction mappings.

How Undercomplete Autoencoders Work

The operation of an undercomplete autoencoder involves several key steps:

1. **Compression**: The encoder processes the input data to form a condensed representation, focusing on the most significant attributes of the data.

2. **Reconstruction**: The decoder then attempts to reconstruct the original data from this compressed form, aiming to minimize discrepancies between the original and reconstructed data.

3. **Optimization**: Through iterative training and backpropagation, the network optimizes the weights and biases to reduce the reconstruction error, refining the model's ability to compress and reconstruct data accurately.

Difference between Vanilla Autoencoders and Undercomplete Autoencoders

1. **Dimensionality Reduction**: An undercomplete autoencoder specifically focuses on reducing the dimensionality of the input data, while a vanilla autoencoder does not necessarily do so.

2. **Bottleneck Layer Size**: In an undercomplete autoencoder, the bottleneck layer has fewer neurons than the input layer, enforcing a compressed representation. In contrast, a vanilla autoencoder might have a bottleneck layer with an equal or greater number of neurons compared to the input layer.

3. **Generalization**: Undercomplete autoencoders, by reducing dimensionality, often capture the most important features of the data, leading to better generalization. Vanilla autoencoders, without a reduced bottleneck, might overfit the data, capturing noise as well as signal.

Example:

- **Vanilla Autoencoder**: Input size = 784, Hidden layer 1 size = 128, Bottleneck size = 64, Output size = 784.

- **Undercomplete Autoencoder**: Input size = 784, Hidden layer 1 size = 128, Bottleneck size = 32, Output size = 784.

In the undercomplete version, the bottleneck size (32) is significantly smaller than the input size (784), forcing the network to learn a compressed representation of the data.

Applications of Undercomplete Autoencoders

Undercomplete autoencoders are versatile and find applications across various domains:

- **Feature Extraction:** They efficiently identify and encode significant features from data, useful for preprocessing in other analytical tasks.

- **Dimensionality Reduction:** They help in reducing the dimensionality of data, analogous to PCA but with the ability to capture non-linear dependencies.

- **Anomaly Detection:** By learning to reconstruct normal data efficiently, they can identify anomalies as data points that significantly deviate from expected reconstructions.

Benefits of Undercomplete Autoencoders

- **Non-linear Capability:** Unlike linear methods like PCA, undercomplete autoencoders can learn non-linear transformations, making them more effective in capturing complex patterns.
- **Customization:** They offer flexibility in architecture, activation functions, and optimizations to cater to specific data types and tasks.

Challenges of Undercomplete Autoencoders

Despite their advantages, undercomplete autoencoders face several challenges:

- **Overfitting:** They can memorize the training data rather than learning to generalize, particularly in cases with small datasets.
- **Optimization Difficulties:** The presence of multiple local minima and the inherent non-linearity can complicate the training process, requiring careful tuning of parameters and initialization.

CONTRACTIVE AUTOENCODER

Contractive Autoencoder was proposed by researchers at the University of Toronto in 2011 in the paper Contractive auto-encoders: Explicit invariance during feature extraction. The idea behind that is to make the autoencoders robust to small changes in the training dataset.

To deal with the above challenge that is posed by basic autoencoders, the authors proposed adding another penalty term to the loss function of autoencoders. We will discuss this loss function in detail.

Loss Function of Contactive AutoEncoder

Contractive autoencoder adds an extra term in the loss function of autoencoder, it is given as:

$$\|J_h(X)\|_F^2 = \sum_{ij}\left(\frac{\partial h_j(X)}{\partial X_i}\right)^2$$

i.e. the above penalty term is the Frobenius Norm of the encoder, the Frobenius norm is just a generalization of the Euclidean norm.

In the above penalty term, we first need to calculate the Jacobian matrix of the hidden layer, calculating a Jacobian of the hidden layer with respect to input is similar to gradient calculation. Let's first calculate the Jacobian of the hidden layer:

$$Z_j = W_i X_i$$
$$h_j = \phi(Z_j)$$

where \phi is non-linearity. Now, to get the jth hidden unit, we need to get the dot product of the i^{th} feature vector and the corresponding weight. For this, we need to apply the chain rule.

$$\begin{aligned}\frac{\partial h_j}{\partial X_i} &= \frac{\partial \phi(Z_j)}{\partial X_i} \\ &= \frac{\partial \phi(W_i X_i)}{\partial W_i X_i}\frac{\partial W_i X_i}{\partial X_i} \\ &= [\phi(W_i X_i)(1-\phi(W_i X_i))]\, W_i \\ &= [h_j(1-h_j)]\, W_i\end{aligned}$$

The above method is similar to how we calculate the gradient descent, but there is one major difference, that is we take h(X) as a vector-valued function, each as a separate output. Intuitively, For example, we have 64 hidden units, then we have 64 function outputs, and so we will have a gradient vector for each of that 64 hidden units.

Applications of Contractive Autoencoders

1. **Feature extraction**: They help extract robust, low-dimensional representations of complex datasets.
2. **Anomaly detection**: By learning typical patterns, they can identify deviations as anomalies.
3. **Data denoising**: The representations are less sensitive to input noise, making them useful for denoising tasks.
4. **Dimensionality reduction**: Useful as a preprocessing step for tasks like visualization or clustering.

SPARCE AUTOENCODERS

What are Sparse Autoencoders?

Sparse Autoencoders are a variant of autoencoders, which are neural networks trained to reconstruct their input data. However, unlike traditional autoencoders, sparse autoencoders are designed to be sensitive to specific types of high-level features in the data, while being insensitive to most other features. This is achieved by imposing a sparsity constraint on the hidden units during training, which forces the autoencoder to respond to unique statistical features of the dataset it is trained on.

How do Sparse Autoencoders work?

Sparse Autoencoders consist of an encoder, a decoder, and a loss function. The encoder is used to compress the input into a latent-space representation, and the decoder is used to reconstruct the input from this representation. The sparsity constraint is typically enforced by adding a penalty term to the loss function that encourages the activations of the hidden units to be sparse.

The sparsity constraint can be implemented in various ways, such as by using a sparsity penalty, a sparsity regularizer, or a sparsity proportion. The sparsity penalty is a term added to the loss function that penalizes

the network for having non-sparse activations. The sparsity regularizer is a function that encourages the network to have sparse activations. The sparsity proportion is a hyperparameter that determines the desired level of sparsity in the activations.

Why are Sparse Autoencoders important?

Sparse Autoencoders are important because they can learn useful features from unlabeled data, which can be used for tasks such as anomaly detection, denoising, and dimensionality reduction. They are particularly useful when the dimensionality of the input data is high, as they can learn a lower-dimensional representation that captures the most important features of the data.

Furthermore, Sparse Autoencoders can be used to pretrain deep neural networks. Pretraining a deep neural network with a sparse autoencoder can help the network learn a good initial set of weights, which can improve the performance of the network on a subsequent supervised learning task.

Applications of Sparse Autoencoders

Sparse Autoencoders have been used in a variety of applications, including:

- **Anomaly detection**: Sparse autoencoders can be used to learn a normal representation of the data, and then detect anomalies as data points that have a high reconstruction error.
- **Denoising**: Sparse autoencoders can be used to learn a clean representation of the data, and then reconstruct the clean data from a noisy input.
- **Dimensionality reduction**: Sparse autoencoders can be used to learn a lower-dimensional representation of the data, which can be used for visualization or to reduce the computational complexity of subsequent tasks.
- **Pretraining deep neural networks**: Sparse autoencoders can be used to pretrain the weights of a deep neural network, which can improve the performance of the network on a subsequent supervised learning task.

DENOISING AUTOENCODERS

Autoencoders are types of **neural network architecture** used for **unsupervised learning**. The architecture consists of an encoder and a decoder. The encoder encodes the input data into a lower dimensional space while the decode decodes the encoded data back to the original input. The network is trained to minimize the difference between decoded data and input. Autoencoders have the risk of becoming an Identify function meaning the output equals the input which makes the whole neural network of autoecoders useless. This generally happens when there are more nodes in the hidden layer than there are inputs.

Denoising Autoencoder (DAE)

Now, a **denoising autoencoder** is a modification of the original autoencoder in which instead of giving the original input we give a corrupted or noisy version of input to the encoder while decoder loss is calculated concerning original input only. This results in efficient learning of autoencoders and the risk of autoencoder becoming an identity function is significantly reduced.

Architecture of DAE

The denoising autoencoder (DAE) architecture resembles a standard autoencoder and consists of two main components:

Encoder:

- The encoder is a neural network with one or more hidden layers.
- It receives noisy input data instead of the original input and generates an encoding in a low-dimensional space.
- There are several ways to generate a corrupted input. The most common being adding a Gaussian noise or randomly masking some of the inputs.

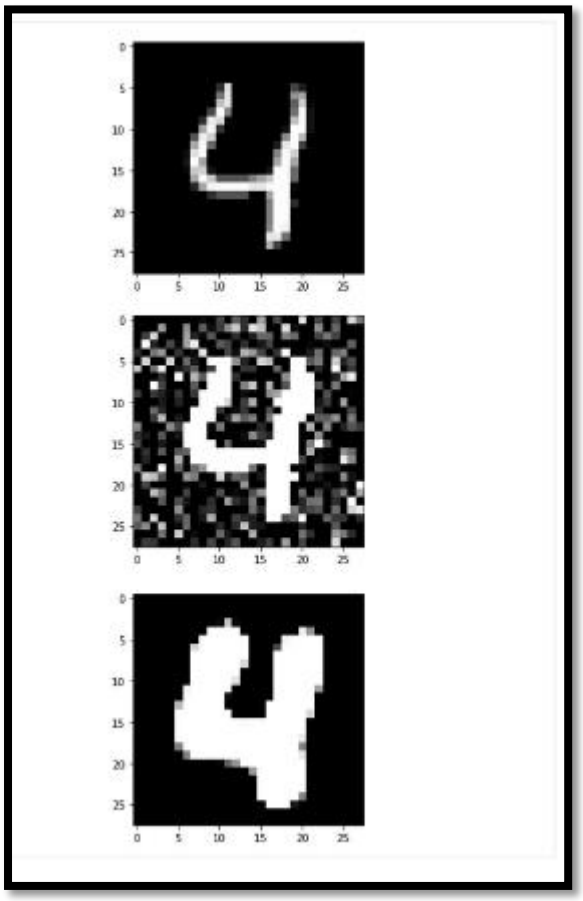
Decoder:

- Similar to encoders, decoders are implemented as neural networks with one or more hidden layers.
- It takes the encoding generated by the encoder as input and reconstructs the original data.
- When calculating the Loss function it compares the output values with the original input, not with the corrupted input.

What DAE Learns?

The above architecture of using a corrupted input helps decrease the risk of overfitting and prevents the DAE from becoming an identity function.

- If DAEs are trained with partially corrupted inputs (e.g., with masking values), they learn to impute or fill in missing information during the reconstruction process. This makes them useful for tasks involving incomplete datasets.
- If DAEs are trained with partially noisy inputs (gaussian noise) DAEs tend to generalize well to unseen, real-world data with different levels of noise or corruption as they learn to extract robust features. This is beneficial in various applications where data quality is compromised, such as image denoising or signal processing.



Objective Function of DAE

The objective of DAE is to minimize the difference between the original input (clean input without the notice) and the reconstructed output. This is quantified using a reconstruction loss function. Two types of loss function are generally used depending on the type of input data.

Applications of DAE

- Image Denoising: DAEs are widely employed for cleaning and enhancing images by removing noise.
- Audio Denoising: DAEs can be applied to denoise audio signals, making them valuable in speech-enhancement tasks.
- Sensor Data Processing: DAEs are valuable in processing sensor data, removing noise, and extracting relevant information from sensor readings.
- Data Compression: Autoencoders, including DAEs, can be utilized for data compression by learning compact representations of input data.
- Feature Learning: DAEs are effective in unsupervised feature learning, capturing relevant features in the data without explicit labels.

Autoencoders have a wide range of applications in machine learning and data science, primarily because of their ability to learn compressed, meaningful representations of data in an unsupervised manner. Here are some prominent **use cases**:

---

1. Dimensionality Reduction

- Autoencoders can reduce the number of features in high-dimensional data while preserving important information, similar to **Principal Component Analysis (PCA)** but capable of capturing nonlinear relationships.
- **Example**: Reducing the dimensionality of image datasets (e.g., MNIST, CIFAR-10) for visualization or preprocessing.

---

2. Data Denoising

- Autoencoders can remove noise from corrupted data by learning to reconstruct clean versions.
- **Example**: Removing noise from images, audio signals, or text data (e.g., denoising noisy scanned documents or audio recordings).

---

3. Anomaly Detection

- By learning typical patterns in data, autoencoders can identify anomalies as inputs with high reconstruction error (i.e., when the input differs significantly from the learned patterns).
- **Example**:
  - Detecting fraudulent transactions in financial data.
  - Identifying defective products in manufacturing.
  - Spotting unusual network traffic for cybersecurity.

---

4. Feature Extraction

- The encoder part of the autoencoder can be used to extract compact, meaningful features for downstream tasks like classification, clustering, or regression.
- **Example**:
  - Extracting features from medical imaging for disease prediction.
  - Learning embeddings for word or sentence representations in natural language processing.

---

5. Generative Modeling

- Variants like **Variational Autoencoders (VAEs)** are used to generate new data samples similar to the training data.
- **Example**:
  - Creating realistic synthetic images, such as faces or artwork.
  - Generating plausible medical data for research while preserving patient privacy.

---

6. Image Compression

- Autoencoders can encode images into smaller representations, serving as a method for lossy compression.
- **Example**: Compressing large images to save storage or reduce transmission time in web applications.

---

7. Image Reconstruction and Super-Resolution

- Autoencoders can reconstruct missing parts of an image or enhance the resolution of low-quality images.
- **Example**: Enhancing satellite images or improving the quality of old photographs.

---

8. Recommendation Systems

- Autoencoders can capture latent factors in user-item interactions, enabling better recommendations.
- **Example**: Learning user preferences for recommending movies, books, or products.

---

9. Pretraining for Deep Networks

- Autoencoders can be used to pretrain deep neural networks in a layer-wise fashion, initializing weights before fine-tuning on supervised tasks.
- **Example**: Pretraining models for complex tasks like speech recognition or sentiment analysis.

---

10. Biomedical Applications

- Autoencoders are applied to analyze high-dimensional biomedical data.
- **Example**:
  - Identifying biomarkers from gene expression data.
  - Compressing and analyzing medical imaging like MRI or CT scans.

---

11. Time Series Forecasting and Anomaly Detection

- Autoencoders are used to model temporal patterns and detect unusual time series behaviors.
- **Example**: Monitoring equipment in predictive maintenance or detecting irregularities in stock market trends.

---

12. Transfer Learning

- The encoder of a pre-trained autoencoder can be used as a feature extractor for other tasks on similar datasets.
- **Example**: Using features learned from a general image dataset for domain-specific image classification.

Why Autoencoders Are Useful

- They work in an **unsupervised** manner, which means they do not require labeled data.
- They capture **nonlinear relationships** in data, unlike linear models such as PCA.
- They can be tailored for specific tasks using variations like **sparse autoencoders**, **denoising autoencoders**, or **variational autoencoders**.

Sandip University, Nashik (MS), India
At Post Mahiravani, Trimbak Road, Nashik – 422 213, Maharashtra
Phone: +91 9545453092 / Toll- Free: 1800-212-2714
https:// www.sandipuniversity.edu.in /info@sandipuniversity.edu.in

MODULE 2

DEEP BELIEF NETWORK

What is a Deep Belief Network?

Deep Belief Networks (DBNs) are sophisticated artificial neural networks used in the field of deep learning, a subset of machine learning. They are designed to discover and learn patterns within large sets of data automatically. Imagine them as multi-layered networks, where each layer is capable of making sense of the information received from the previous one, gradually building up a complex understanding of the overall data.

DBNs are composed of multiple layers of stochastic, or randomly determined, units. These units are known as Restricted Boltzmann Machines (RBMs) or other similar structures. Each layer in a DBN aims to extract different features from the input data, with lower layers identifying basic patterns and higher layers recognizing more abstract concepts. This structure allows DBNs to effectively learn complex representations of data, which makes them particularly useful for tasks like image and speech recognition, where the input data is high-dimensional and requires a deep level of understanding.

The architecture of DBNs also makes them good at unsupervised learning, where the goal is to understand and label input data without explicit guidance. This characteristic is particularly useful in scenarios where labelled data is scarce or when the goal is to explore the structure of the data without any preconceived labels.

How Deep Belief Networks Work?

DBNs work in two main phases: pre-training and fine-tuning. In the pre-training phase, the network learns to represent the input data layer by layer. Each layer is trained independently as an RBM, which allows the network to learn complex data representations efficiently. During this phase, the network learns the probability distribution of the inputs, which helps it understand the underlying structure of the data.

In the fine-tuning phase, the DBN adjusts its parameters for a specific task, like classification or regression. This is typically done using a technique known as backpropagation, where the network's performance on a task is evaluated, and the errors are used to update the network's parameters. This phase often involves supervised learning, where the network is trained with labelled data.

Concepts Related to Deep Belief Networks (DBNs)

- **Restricted Boltzmann Machines (RBMs):** These are the building blocks of DBNs. An RBM is a two-layered neural network that learns the probability distribution of the input data. Each layer in a DBN is typically an RBM.

- **Stochastic Units:** DBNs use units that make decisions probabilistically. This stochastic nature allows the network to explore and learn more complex patterns in the data.

- **Layer-wise Training:** DBNs are trained one layer at a time, which is efficient and helps in learning deep representations of data.

- **Unsupervised and Supervised Learning:** DBNs are versatile, capable of both unsupervised learning (learning from unlabeled data) and supervised learning (learning from labeled data).

- **Greedy Algorithm:** This is used during the pre-training phase of DBNs. Each layer is trained greedily, meaning it's trained independently of the others, which simplifies the training process.

- **Backpropagation:** In the fine-tuning phase, backpropagation is used for supervised learning tasks. It adjusts the network's parameters to improve its performance on specific tasks.

DBNs, with their deep architecture and efficient learning capabilities, have been pivotal in advancing the field of deep learning, particularly in handling complex tasks like image and speech recognition.

Mathematical Concepts Related to DBN

Deep Belief Networks (DBNs) employ several mathematical concepts, blending probability theory with neural network structures. At their core, they use Restricted Boltzmann Machines (RBMs) for layer-wise learning, which are based on probabilistic graphical models.

1. Energy-Based Model: Each RBM within a DBN is an energy-based model. For an RBM with visible units v and hidden units h, the energy function is defined as:

E(v,h) = -\sum _{i}a_i v_i - \sum _j b_j h_j - \sum _{i,j} v_j h_j w_{ij}

Here, ai and bj are bias terms, and wij represents the weights between units.

2. Probability Distribution: The probability of a given state of the RBM is defined by the Boltzmann distribution:

P(v,h) = \frac{e^{-E(v,h)}}{Z}

where Z is the partition function, a normalization factor calculated as the sum over all possible pairs of visible and hidden units.

3. Training using Contrastive Divergence: RBMs are typically trained using a method called Contrastive Divergence (CD). This method approximates the gradient of the log-likelihood and updates the weights wij, and biases ai,bj to maximize the likelihood of the training data under the model.

In a DBN, these RBMs are stacked. The hidden layer of one RBM serves as the visible layer for the next. After this unsupervised, layer-wise training, the entire network can be fine-tuned using supervised methods like backpropagation, where the goal is to minimize the difference between the predicted output and the actual label of the training data.

Implementation of Deep Belief Networks (DBNs)

Prerequsite:
To implement the Deep Belief Networks (DBNs), first you need to install the numpy, pandas, and scikit-learn

!pip install numpy pandas scikit-learn

The code provided outlines the process of creating a Deep Belief Network (DBN) using Python. Here's a step-by-step explanation:

- **Import Libraries:** Essential Python libraries for data handling (numpy, pandas), machine learning models (scikit-learn), and deep learning (tensorflow) are imported.
- **Load Dataset:** The MNIST dataset, a collection of 28x28 pixel images of handwritten digits, is fetched using fetch_openml from scikit-learn. This dataset is commonly used for benchmarking classification algorithms.
- **Preprocessing:** The dataset is split into training and testing sets with train_test_split. The data is then scaled using StandardScaler to normalize it, which often leads to better performance for neural networks.
- **RBM Layer:** A Restricted Boltzmann Machine (RBM) is initialized with a specified number of components and learning rate. RBMs are unsupervised neural networks that find patterns in data by reconstructing the inputs.
- **Classifier Layer:** A logistic regression classifier is chosen for the final prediction layer. Logistic regression is a simple yet effective linear model for classification tasks.
- **DBN Pipeline:** The RBM and logistic regression model are chained together in a Pipeline. This allows for sequential application of the RBM (for feature extraction) followed by logistic regression (for classification).
- **Training:** The pipeline, which forms the DBN, is trained on the preprocessed training data (X_train_scaled). The RBM learns features that are then used by the logistic regression model to classify the digits.
- **Evaluation:** Finally, the trained DBN's performance is evaluated on the test set. The classification accuracy (dbn_score) is printed to provide a quantitative measure of how well the model performs.

The Architecture of DBN

In the DBN, we have a hierarchy of layers. The top two layers are the associative memory, and the bottom layer is the visible units. The arrows pointing towards the layer closest to the data point to relationships between all lower layers.

Directed acyclic connections in the lower layers translate associative memory to observable variables.

The lowest layer of visible units receives input data as binary or actual data. Like RBM, there are no intralayer connections in DBN. The hidden units represent features that encapsulate the data's correlations.

A matrix of proportional weights W connects two layers. We'll link every unit in each layer to every other unit in the layer above it.

Advantages of DBNs

1. **Unsupervised Learning**:

- DBNs are well-suited for unsupervised learning, making them useful when labeled data is scarce.

2. **Layer-Wise Pretraining**:
   - Pretraining helps initialize deep networks with meaningful weights, addressing issues like vanishing gradients in traditional deep networks.
3. **Feature Extraction**:
   - DBNs extract hierarchical features, capturing complex data structures.
4. **Generative Capability**:
   - As generative models, DBNs can sample new data from the learned distribution.

Limitations of DBNs

1. **Computationally Expensive**:
   - Training multiple RBMs layer-by-layer can be slow, especially for large datasets or deep architectures.
2. **Contrastive Divergence Approximation**:
   - The CD algorithm provides an approximate solution, which may not always yield optimal results.
3. **Obsolescence**:
   - DBNs have largely been replaced by more modern architectures like **Deep Neural Networks (DNNs)**, **Convolutional Neural Networks (CNNs)**, and **Variational Autoencoders (VAEs)** due to advancements in optimization techniques and computational resources.

Applications of DBNs

1. **Feature Learning**:
   - Learning meaningful features for downstream tasks like classification, clustering, and regression.
2. **Dimensionality Reduction**:
   - Compressing high-dimensional data while preserving important information.
3. **Generative Modeling**:
   - Sampling synthetic data from the learned distribution.
4. **Image Recognition**:
   - Early experiments in digit recognition, such as on the MNIST dataset.
5. **Time-Series Analysis**:
   - Modeling sequential data, such as speech and stock market trends.

Comparison with Other Models

Feature	Deep Belief Networks	Deep Neural Networks
Training	Layer-wise pretraining + fine-tuning	End-to-end training
Supervised vs. Unsupervised	Unsupervised pretraining + supervised fine-tuning	Primarily supervised
Generative Capability	Yes	Limited (without specific design)
Popularity	Declining	Highly popular

STACKING RBM TO CREATE DEEP BELIEF NETWORK

Stacking Restricted Boltzmann Machines (RBMs) is the fundamental process of constructing a **Deep Belief Network (DBN)**. This involves training multiple RBMs layer by layer in an unsupervised manner to extract hierarchical features. Here's a step-by-step explanation of how this stacking process works:

Steps to Stack RBMs for a DBN

1. **Start with the First RBM (Input Layer to First Hidden Layer)**
   - Train the first RBM on the input data.
   - This RBM consists of:
     - A **visible layer**: Represents the input data.
     - A **hidden layer**: Learns features from the input.
   - Objective: Minimize the **reconstruction error** between the input data and its reconstruction.

   Training Process:

   - Use an unsupervised learning algorithm like **Contrastive Divergence (CD)** to train the RBM.
   - Update the weights and biases based on the input data.

   Output:

   - The hidden layer activations of this RBM become the input for the next layer.

---

2. **Stack the Second RBM**
   - The output (hidden layer activations) of the first RBM is used as the **visible layer input** for the second RBM.

- Train the second RBM to learn higher-level features from the transformed input provided by the first RBM.

Key Insight:

- The second RBM is effectively modeling the distribution of the features extracted by the first RBM.

---

3. **Repeat for Additional Layers**
   - Continue stacking RBMs, using the hidden layer outputs of the previous RBM as the input for the next one.
   - Each additional RBM captures more abstract and higher-level features.

---

4. **Combine Layers into a DBN**
   - Once all RBMs are trained, they are combined to form the DBN.
   - At this stage, the DBN can be used as:
     - An **unsupervised feature extractor**.
     - A **generative model** capable of sampling new data.

Fine-Tuning the DBN

After stacking RBMs, the DBN can be fine-tuned if labeled data is available. This step transitions the DBN from an unsupervised learning model to a supervised one:

1. **Add a Softmax Layer (or Output Layer)**
   - Attach a softmax classifier or regression head to the final hidden layer of the DBN for supervised tasks.
2. **Supervised Fine-Tuning**
   - Use backpropagation and gradient descent to optimize the entire network for a specific task, such as classification or regression.
   - Fine-tune all layers jointly to improve task performance.

Advantages of Stacking RBMs

1. **Greedy Layer-Wise Training**:
   - By training each RBM individually, the model avoids the vanishing gradient problem common in deep networks.
   - Each RBM focuses on learning meaningful features layer by layer.
2. **Hierarchical Feature Learning**:
   - Each layer learns increasingly abstract and complex representations.
3. **Efficient Initialization**:
   - The unsupervised pretraining initializes the weights, enabling better performance and faster convergence during supervised fine-tuning.

Example Architecture

For an input dataset (e.g., MNIST digit images with 784 pixels):

1. **RBM 1**:
   - Visible layer: 784 neurons (input pixels).
   - Hidden layer: 500 neurons (learn basic features like edges).
2. **RBM 2**:
   - Visible layer: 500 neurons (output of RBM 1).
   - Hidden layer: 300 neurons (learn higher-level patterns like shapes).
3. **RBM 3**:
   - Visible layer: 300 neurons (output of RBM 2).
   - Hidden layer: 100 neurons (capture even more abstract representations).
4. **Classifier**:
   - Add a softmax output layer for classifying digits (e.g., 10 neurons for 10 digits).

Applications of DBNs Built by Stacking RBMs

- **Image Recognition**: Learning hierarchical features for classifying images.
- **Generative Modeling**: Sampling new data by propagating information through the network.
- **Pretraining for Deep Networks**: Initializing weights for deeper neural networks to accelerate and improve training.

WAKE SLEEP ALGORITHM

Neural networks are capable of completing very complicated tasks for their hidden layers. For training these networks, the first step is to pre-train the model using an unsupervised learning approach, and later fine-tune the parameters using a supervised approach.

Unsupervised pre-training makes the deep learning method more effective. It uses a greedy, layer-wise approach for pre-training. The layers of the neural network are trained on unlabeled data as only inputs are used and no output values are needed. The layers are trained separately based on the output of the previous layer.

Supervised methods for neural networks have limited uses. They require feedback in order to train the networks and these networks function in specific circumstances. Based on the feedback, the entire network requires some process to communicate the error information and adjust the weights.

To overcome these problems, the wake-sleep algorithm is suggested. It is an unsupervised learning algorithm for a stochastic multilayer neural network.

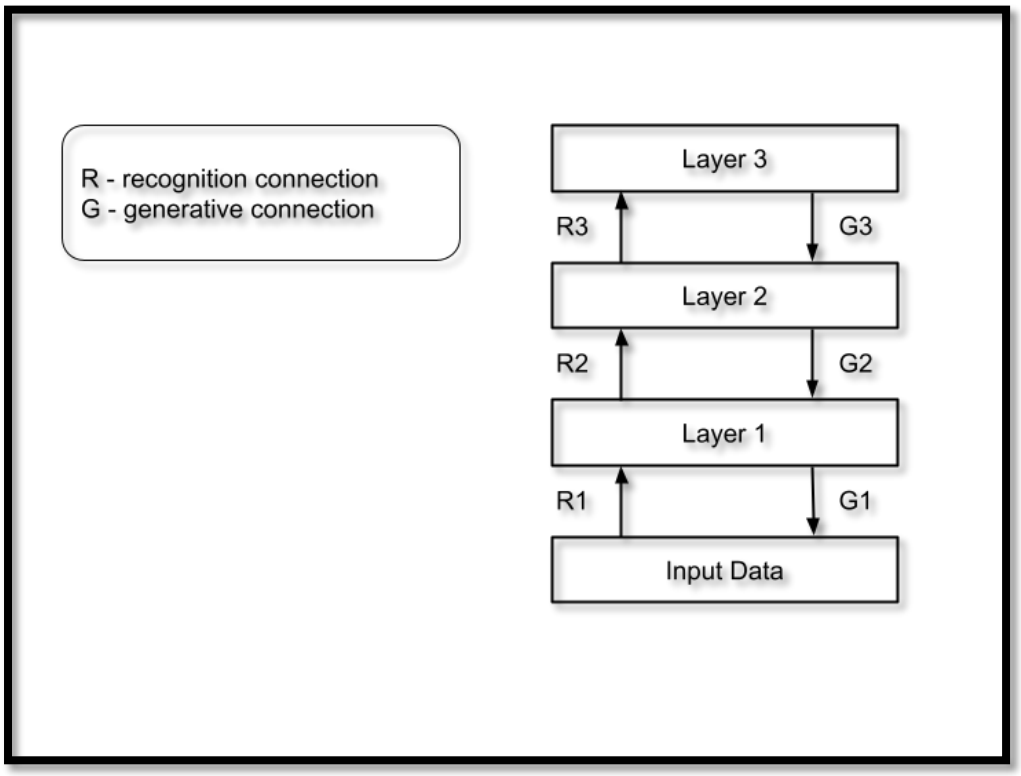
Wake-sleep Algorithm

This algorithm can be explained as a stack of layers that represents the data. In this process, each layer learns to represent the activities of the adjacent hidden layers. This algorithm allows the representations to be efficient and the inputs to be adjusted accurately.

The neural network uses two sets of weights and the model is a generative model. It means the model can automatically discover patterns and generate new data. The weights used in this model are generative and recognition weights. The generative weights are the weights of the model and the recognition weights are the estimated representation of the input data.

In the stack of layers of data representations, the above layers represent data from the layer below, and the original data is set below the bottom layer. The recognition and generative weights between two layers are trained to make the algorithm more efficient.

This algorithm uses bottom-up recognition connections to represent the input data into one or more hidden layers and top-down generative connections to approximately reconstruct the actual input data based on the representations in the above layer. The structure is,



Training Phases

In this process, the training takes place in two phases:

WakePhase:
In this phase, the bottom-up connections use recognition weights to represent the original input data on the first hidden layer above and carry on the representation to the second hidden layer. The representations provided from the layers below proceed to the top layer. In this way, it forms a combination of layers of representations which is called the 'total representation' of input data.

After the representation of the input data to the above layer, the reconstruction to the bottom layer begins. In the top-down connections, the layers try to predict the input from the previous layer based on the representation they have learned, and the error in this prediction should be at a minimum.

In the bottom-up process, the input data is feed-forward to the hidden layers, and for each hidden layer, a random binary decision is made. The decisions of all the hidden layers are then used as the sample for training. The learning of the model happens with generative weights and the recognition weights are adjusted accordingly. Because based on these samples, the estimated reconstructions are made in the top-down process and the model learns to reconstruct with minimum error. The system is driven forward with the recognition weights and it learns with the generative weights.

SleepPhase:

The sleep phase works in reverse from the wake phase as the system here drives forward with the generative weights. In this phase, the model closes off the network to the original inputs and works on improving the recognition connections based on the learned generative model. The recognition connections are used for the reconstruction of the activities in the below layer.

The binary decision samples can be extracted from the generative model and these can be used as inputs for the recognition weights to reconstruct. Starting from the highest layer with a random input, a binary decision is generated from the hidden units of the above layer. In the top-down process, a decision is made for each hidden layer and generating a sample. Based on these samples, the recognition weights are trained to reconstruct the activities of the bottom layer.

In this phase, the model learns the recognition weights and the samples are generated using top-down process. Starting with random inputs and alternating between the two phases makes a model quite good.

Example of Wake-Sleep Algorithm

The Helmholtz machine can be an example of this algorithm. This neural network is trained using unsupervised learning algorithms as wake-sleep algorithm. The goal of this network is to estimate the hidden structure with the generative model after learning the representations of the data. It has a bottom-up recognition network that produces a distribution over hidden units, a top-down generative network that generates values of the hidden units and new data.

Limitations

1. The recognition weights are trained to reconstruct the data from the generative model even when there is no data at the beginning and it becomes a waste. Because the generative data differs from the real data.
2. The binary states generated in the hidden layers are conditionally independent and it does not allow the model to represent the ‘explaining-away’ effect. It means the representation of the states of one layer depends on the activation of only one of two units of the adjacent layer.
3. There can be some damage to data representations if any estimation is performed with mistakes.

Why Use the Wake-Sleep Algorithm?

1. **Bidirectional Training**: The algorithm alternates between improving the encoder and decoder, ensuring they complement each other.
2. **Unsupervised Learning**: No labels are required; the model learns to infer latent representations and generate data.
3. **Improved Generative Models**: The sleep phase refines the generative model, enabling it to produce more realistic data.

Challenges

1. **Approximation Errors**: The recognition network may not perfectly approximate the true posterior P(h|x)P(h|x)P(h|x), leading to suboptimal updates in the generative model.
2. **Better Alternatives**: Modern methods like **Variational Autoencoders (VAEs)** have largely replaced the Wake-Sleep Algorithm due to their use of tighter probabilistic bounds and end-to-end training.
3. **Efficiency**: The two-phase approach can be less efficient compared to direct optimization methods used in more recent models.

Applications

- **Deep Belief Networks (DBNs)**: Used to refine generative and recognition models in hierarchical networks.
- **Unsupervised Feature Learning**: Learning compact representations of data.
- **Generative Modeling**: Creating synthetic data or reconstructing missing data.

Summary of the Algorithm

Phase	Goal	Network Being Updated
Wake Phase	Train recognition network (encoder)	Recognition Network
Sleep Phase	Train generative network (decoder)	Generative Network

Algorithm Steps

1. Initialization
   - Randomly initialize the parameters θ (for the generative model) and ϕ (for the recognition model).
2. Repeat for $t = 1, \ldots, T$:
   - Wake Phase (Training the Recognition Model):
     1. Sample a data point x from the dataset X.
     2. Use the recognition model $Q_\phi(h|x)$ to infer the latent variable h for the observed data x.
     3. Update ϕ to maximize the posterior probability $Q_\phi(h|x)$, typically by minimizing the divergence between $Q_\phi(h|x)$ and $P_\theta(h|x)$.
        - Gradient update:
          $$\phi \leftarrow \phi + \eta \cdot \nabla_\phi \log Q_\phi(h|x)$$
   - Sleep Phase (Training the Generative Model):
     1. Sample a latent variable h from the recognition model $Q_\phi(h|x)$.
     2. Generate a data sample $\tilde{x}$ using the generative model $P_\theta(x|h)$.
     3. Update θ to maximize the likelihood of the generated data $\tilde{x}$ under the generative model:
        - Gradient update:
          $$\theta \leftarrow \theta + \eta \cdot \nabla_\theta \log P_\theta(\tilde{x}|h)$$
3. End Repeat

USE CASES OF WAKE SLEEP ALGORITHM

1. Generative Modeling and Data Synthesis

- **Use Case**: Generating new data samples that resemble the training data.
- **Example**:
  - In **image generation**, the Wake-Sleep algorithm could be used to train a generative model capable of generating realistic images of objects or faces. The recognition model would learn the latent representations of images, while the generative model would learn to reconstruct the data from those representations.
  - In **text generation**, the algorithm could generate realistic sentences or paragraphs by learning the distribution of words and phrases.

---

2. Unsupervised Feature Learning

- **Use Case**: Learning useful features from unlabeled data by extracting a hierarchy of features in an unsupervised manner.
- **Example**:

- In **speech recognition**, the Wake-Sleep algorithm could be applied to learn a hierarchical set of features from raw speech data, allowing for better representations of phonemes, words, or phrases without requiring labeled training data.
- For **image recognition**, the algorithm could extract low-level features (e.g., edges, textures) in earlier layers and higher-level features (e.g., shapes, faces) in later layers. These features can then be used for downstream tasks such as classification or clustering.

---

3. Dimensionality Reduction

- **Use Case**: Reducing the dimensionality of high-dimensional data while preserving important structural information.
- **Example**:
  - In **genomic data analysis**, the algorithm could be used to extract lower-dimensional representations of gene expression data, which may be useful for understanding the relationships between genes or discovering disease-related biomarkers.
  - In **image compression**, the Wake-Sleep algorithm could be used to learn a compact representation of high-resolution images, making it possible to compress them without losing important features.

---

4. Semi-Supervised Learning

- **Use Case**: Leveraging unlabeled data to improve the performance of supervised models, particularly in cases with limited labeled data.
- **Example**:
  - In **medical imaging**, where labeled data (e.g., images labeled with disease annotations) may be scarce, the Wake-Sleep algorithm can be used to train a generative model on unlabeled medical images and then fine-tune a classifier with the available labeled data. The generative model can learn useful representations of the images, and the classifier can make predictions based on these representations.

---

5. Anomaly Detection

- **Use Case**: Identifying rare or abnormal data points by learning the typical distribution of data.
- **Example**:
  - In **network security**, the algorithm could be used to model typical network traffic patterns. Once the model is trained, it can detect unusual network behavior (e.g., DDoS attacks, intrusions) that deviates from the learned distribution of normal traffic.
  - In **fraud detection**, the Wake-Sleep algorithm could be applied to detect unusual transactions or patterns that indicate fraudulent activity, by learning from a large set of normal transaction data.

---

6. Image Denoising

- **Use Case**: Removing noise from corrupted or degraded images.
- **Example**:

- The algorithm can be used to train a model capable of reconstructing clean images from noisy versions. The recognition model learns to infer the clean image's latent variables, while the generative model learns to reconstruct the original clean image from these latent variables.

---

7. Hierarchical Learning in Complex Models

- **Use Case**: Learning hierarchical, layered representations in complex, structured data.
- **Example**:
  - In **natural language processing (NLP)**, the Wake-Sleep algorithm could learn hierarchical representations of text, such as understanding the relationships between words, sentences, and paragraphs in a document. The recognition model learns the latent variables (topics, sentiments, etc.), and the generative model learns how to produce text samples based on these latent variables.
  - In **robotics**, it could be applied to learn hierarchical representations of sensory data (e.g., visual, auditory) in a robot, helping the robot to recognize objects and make decisions based on abstract representations of its environment.

---

8. Variational Inference and Bayesian Learning

- **Use Case**: Approximate inference in Bayesian networks or probabilistic graphical models.
- **Example**:
  - The Wake-Sleep algorithm has connections to **variational inference** methods, where the goal is to approximate complex posterior distributions. For instance, in **Bayesian deep learning**, the algorithm can be used to train models in an unsupervised or semi-supervised manner, enabling efficient approximation of posterior distributions for latent variables.

---

9. Training Deep Generative Models

- **Use Case**: Training deep generative models like **Deep Belief Networks (DBNs)**, where multiple layers of latent variables need to be learned.
- **Example**:
  - In **unsupervised pretraining**, the Wake-Sleep algorithm can be applied to a stack of RBMs (as used in DBNs) to pretrain a deep network. This pretraining helps initialize the network with meaningful weights, which can later be fine-tuned with labeled data for a specific supervised task.

---

10. Image Inpainting (Filling Missing Parts of Images)

- **Use Case**: Reconstructing missing or occluded parts of an image by leveraging learned generative models.
- **Example**:
  - In **image inpainting**, the Wake-Sleep algorithm could train a generative model that learns to fill in missing pixels in an image based on the context of surrounding pixels. The recognition model helps infer the latent representation of the image, while the generative model reconstructs the missing parts.

Sandip University, Nashik (MS), India
At Post Mahiravani, Trimbak Road, Nashik – 422 213, Maharashtra
Phone: +91 9545453092 / Toll- Free: 1800-212-2714
https:// www.sandipuniversity.edu.in /info@sandipuniversity.edu.in

QUESTIONS FOR UNIT 4

2 Marks Questions

What is an auto encoder, and how does it work?
Describe an under complete auto encoder.
State is the purpose of a contractive autoencoder?
Explain the concept of sparse autoencoders.
State that How do denoising autoencoders differ from other types of autoencoders?
Provide two use cases of autoencoders in real-world applications.
Define is a Deep Belief Network (DBN)?
State how are Restricted Boltzmann Machines stacked to create a DBN?
State the Wake-Sleep Algorithm, and how does it work in DBNs?

5/7/8/10 Marks Questions

Describe the structure and purpose of a Deep Belief Network in machine learning.
Explain the Wake-Sleep Algorithm and its role in training Deep Belief Networks.
Explain the process of stacking Restricted Boltzmann Machines (RBMs) to create a multi-layered Deep Belief Network.
Discuss the key differences between Deep Belief Networks and standard feedforward neural networks.
Provide a code example demonstrating how to construct and train a DBN on a dataset, highlighting important steps.
Describe how the Wake-Sleep Algorithm can be utilized to train a DBN specifically for image recognition tasks.
Discuss how Deep Belief Networks enhance feature learning capabilities over traditional shallow networks.
Identify and explain potential drawbacks or challenges associated with implementing Deep Belief Networks in real-world scenarios
Compare various training methods (including unsupervised pre-training and fine-tuning) for DBNs regarding efficiency and performance.
Discuss how different hyperparameters (such as learning rate, number of hidden layers, and batch size) influence the training outcome of Deep Belief Networks.
Define an autoencoder and describe its structure, including the encoder and decoder components.
Discuss how the architecture and purpose of these two types of autoencoders differ.
Provide a code example that demonstrates how to create and train a denoising autoencoder on a dataset, explaining each step clearly.
Discuss how sparse autoencoders can be utilized for dimensionality reduction and the potential benefits they offer compared to traditional methods.

Outline the structure of a DBN, detailing how stacked Restricted Boltzmann Machines (RBMs) are used to form the network and their training process.
Define the Wake-Sleep Algorithm and explain its significance in training Deep Belief Networks.
Explain the methodology of denoising autoencoders and their applications in scenarios where data might be corrupted or noisy.
Describe the architecture of Deep Belief Networks (DBNs) and explain how they are constructed using RBMs.
Create a Deep Belief Network using a framework such as TensorFlow or Keras.
Evaluate the Wake-Sleep Algorithm as a training method for Deep Belief Networks.
Analyze the limitations of Deep Belief Networks compared to other deep learning architectures.
Describe the structure of an autoencoder, including the encoder and decoder components, and explain the purpose of each.
Discuss each type of autoencoder, highlighting their unique characteristics and applications in data processing and feature extraction.
Provide a step-by-step code example demonstrating how to create, train, and evaluate a denoising autoencoder on a noisy dataset, explaining key steps throughout the process.
Discuss how sparse autoencoders work to select relevant features and compare their performance with traditional feature selection methods.