Autoencoder

- An autoencoder is a type of unsupervised learning where neural networks are subject to the task of representation learning.
- Its main purpose is to learn efficient representations of data, typically for the purpose of dimensionality reduction or feature extraction.
- Here, we use unlabelled data and still use backpropagation algorithm to represent the input data into encoded form so that we can decode the encoded data to reconstruct the original data from the encoded representation.

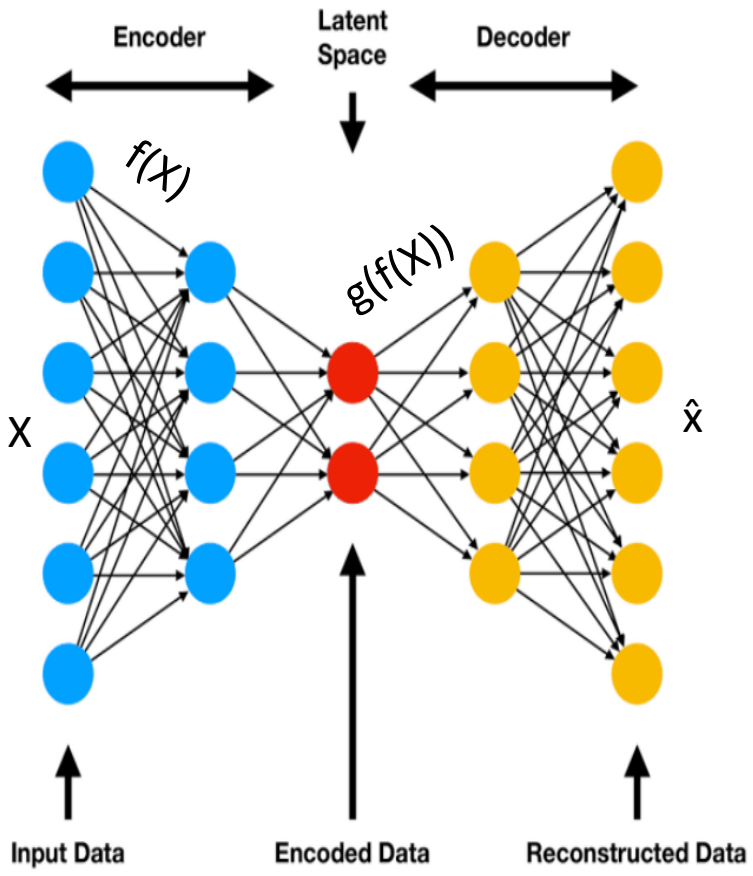
Structure : Autoencoder

- An autoencoder consists of two main parts:

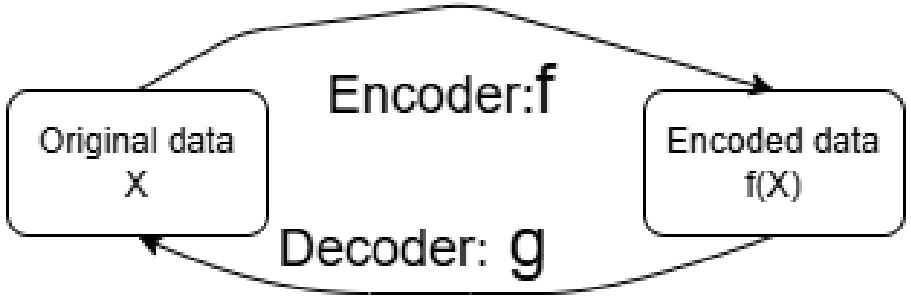
(i) Encoder: This part compresses the input data into a lower-dimensional representation, often referred to as the "latent space" or "bottleneck."

(ii) Decoder: This part reconstructs the original input from the compressed representation.

- So, for an input feature X, there is a mapping f(X) that gives the encoded data, and is another mapping g(f(X)) that transform the encoded data to reconstructed data which should be identical to the original input.



Training



- Autoencoders are trained to minimize the difference between the original input and its reconstructed output.
- This is usually done using a loss function, such as mean squared error.
- The goal is to learn an efficient encoding of the data that captures its essential features.
- For efficient encoding, the task of the neural network is to go for representation learning, i.e., how to encode the input data.

Autoencoder

- For efficient encoding or representation learning, bottleneck is introduced in the neural network.
- As mentioned earlier, for an input feature X, there is an encoder f(X) that gives the encoded data, and is a decoder g(f(X)) that transform the encoded data to reconstructed output.
- The reconstructed output should be identical or almost identical to the original input.
- So, there is a possibility that the network may eventually learn an identity mapping, as g(f(X))=X ➔$g_o f(X)=X$, where $g_o f$ is a composite identity mapping in input feature space.

Autoencoder

- A network learns an identity mapping means it does not learn the representation, i.e., does not learn the inner structure of the data.
- To learn the inner structure, we need a bottleneck layer in the network.
- This bottleneck layer forces a compressed knowledge representation of the input.
- That is, if input feature vectors are of dimension m, in the compressed knowledge representation, the input vector will be mapped to a vector of dimension, say d, where d <<m.

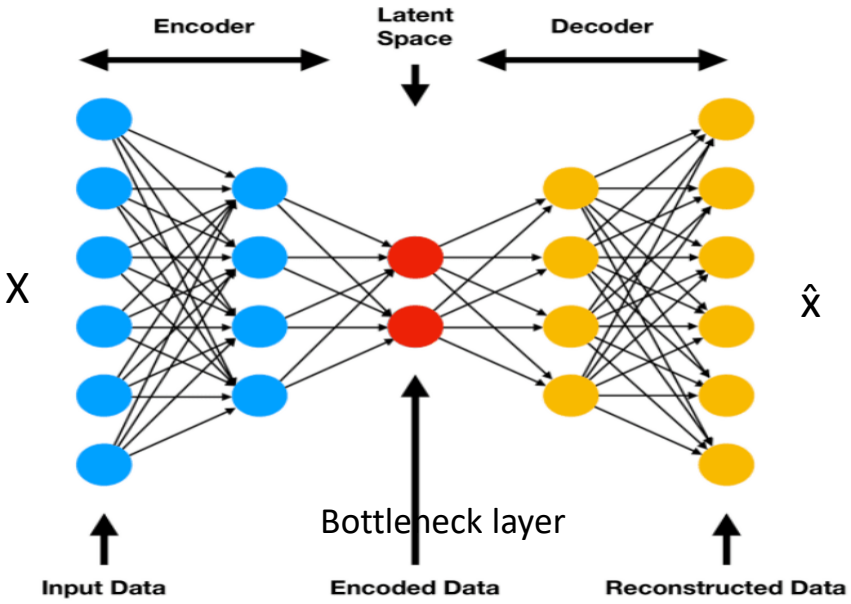
Assumptions: Autoencoder

- Autoencoder is designed based on the assumption that there exists high degree of correlation in the input data.
- If they are uncorrelated, i.e., if the features of the input data are independent to each other then the compressed domain representation and subsequent reconstruction of the original input will be difficult, in fact may not be possible at all, because during compression they will loose salient features of the input.
- So, when the neural network goes for representation learning, it basically transforms the input data to the compressed domain by removing the correlation or redundancy present in the data.
- Thus it preserves only the uncorrelated part and from this uncorrelated part it should subsequently possible to reconstruct the original input data.

Autoencoder: Summary

- Autoencoder encodes the data, i.e., it codes the data in its own.
- This is an unsupervised learning, as we don't need the class label of the data during training.
- Whatever is fed to the input of the autoencoder, it outputs the same thing.
- For this, we need two different functions: encoder and decoder.
- The encoder will encode the input data to a compressed domain knowledge representation using one or many hidden layers, where last hidden layer is called the bottleneck or latent layer.
- The decoder will decode the data from that compressed representation available at the bottleneck layer to the original input or closer to original input at the output layer. The decoder also may contain many hidden layers.
- Encoder part is from input layer to bottleneck layer and decoder part is from bottleneck layer to output layer.

Base Architecture of an Autoencoder

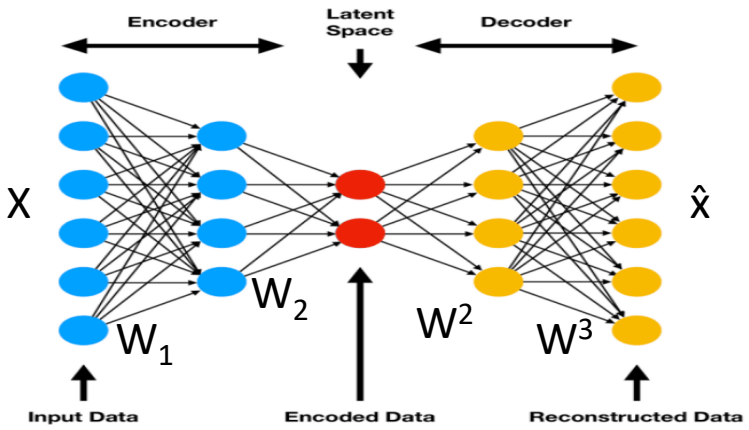


- If input is X and autoencoder reconstructed x̂ then the error between X and x̂ should be minimum.
- We have encoder half and decoder half in our base autoencoder.
- It has one input layer, one output layer and one or more hidden layer with a bottleneck layer or latent layer.
- In the bottleneck layer, we are compressing the data and get the compressed domain knowledge representation of the input data.
- The number of nodes in the bottleneck layer is much less than the number of nodes in the input layer.

Base Architecture of an Autoencoder

- So we have to reconstruct X from $\hat{x}$.

That is why input layer and output layer should have same number of nodes. But, input layer has a bias input so it contains one more node than output layer.



- But if the input is an image, say of size M×N, then we have MN number of pixels, represented by a vector.
- Each pixel is represented by a node. So we need MN+1 nodes, as one node is required for bias.
- But we don't need the bias at output layer, so the number of nodes at output layer is MN.
- The hidden layer is the compressed domain knowledge representation of the input image, so the representation space contains vectors of dimension d << MN. But here also we need a bias node, so number of nodes in bottleneck is d+1

Loss Function

- Whatever may be the type of autoencoder, we have to encode the input data and again decode the encoded data for faithful reconstruction of the input.
- At the encoder side, the bottleneck layer compress the input data and at decoder side the compressed data is reconstructed.
- So autoencoder should perform two tasks:

(i) Autoencoder should be sensitive to input for accurate reconstruction.

(ii) It should not be sensitive enough to memorize or overfit the training data.

Loss Function

(i) Sensitive to input for accurate reconstruction =>

- Reconstructed vector $\hat{x}$ should be as close as possible to the input vector X, i.e., autoencoder should accurately reconstruct the input vector.
- But if this is the only aim of the autoencoder, then it might be possible that autoencoder simply learns the identity mapping.
- Then the learnt identity mapping will faithfully reconstruct the input data.
- But this should not be the only objective function of an autoencoder, as the autoencoder than memorize the input data.

Loss Function

(ii) Insensitive enough to memorize or overfit the training data =>

- The main interest in an autoencoder is that how the data is represented in the compressed domain. Because this encoded data is useful for some other applications.
- So we have two conflicting requirements or expectations from an autoencoder: it should be sensitive and at the same time should not be sensitive enough.

Loss Function

• Both the conflicting requirements are satisfied by defining an appropriate loss function.

Loss Function: L(X, x̂) + Regularizer

- The **loss function** measures how well the network can reconstruct its input after passing through a lower-dimensional representation (latent space).
- This loss function has two components. First part, L(X, x̂) gives the error between original and reconstructed input.

$$L(\mathrm{X}, \hat{\mathrm{x}}) = \frac{1}{2} \sum_{N} ||\mathrm{X} - \hat{\mathrm{x}}||^2$$

- This should be minimized. That is autoencoder is sensitive for faithful reconstruction of the input.

Loss Function

Loss Function: L(X, $\hat{x}$) + Regularizer

- The second part of the loss function is the regularizer which conflicts the first part.
- Regularizer term tries to make the autoencoder insensitive to the input and forces it to learn the low dimensional representation.
- Thus autoencoder learns the salient features of the input. Using this salient feature, the decoder can reconstruct the input data.
- So, it does not simply learn the identity function.
- This loss function is used in backpropagation learning for training of autoencoder

Loss Function

- The most common loss functions for autoencoders are:

1. Mean Squared Error (MSE) Loss

This is the most commonly used loss function for standard autoencoders. It computes the squared difference between the original input (x) and the reconstructed output ($\hat{x}$) for each input sample, then takes the mean over all samples and dimensions.

$$\text{MSE Loss} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$$

Where:

- N is the total number of features in the input.
- x_i is the original input.
- $\hat{x}_i$ is the reconstructed output.

This loss function tries to minimize the pixel-wise or feature-wise reconstruction error. It is particularly useful when the input data is continuous.

Loss Function

2. Binary Cross-Entropy (BCE) Loss

If the input data is binary or normalized between 0 and 1 (like in the case of images), **binary cross-entropy** can be used as the loss function:

$$\text{BCE Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left(x_i \log(\hat{x}_i) + (1 - x_i)\log(1 - \hat{x}_i)\right)$$

This loss measures the difference between the original binary input and the reconstructed output using a probabilistic approach, which is useful for binary or normalized data.

Loss Function

3. Custom Loss Functions

Depending on the task, autoencoders might use more sophisticated loss functions. Some examples are:

- **KL Divergence (for Variational Autoencoders)**: This term is added to the loss function to measure how closely the learned latent space follows a standard normal distribution.
- **Perceptual Loss**: For autoencoders that deal with high-quality images, a perceptual loss may be used to compare higher-level features from pre-trained networks (e.g., VGG) rather than just pixel values.
- **Total Variation Loss**: Used in some applications to penalize large variations in the reconstructed image, encouraging smoother reconstructions.

Regularization

- In autoencoders, **regularization** is often applied to the loss function to improve the learned latent space representation or to prevent overfitting.
- Various types of regularization encourage the autoencoder to learn meaningful and efficient representations.

L2 Regularization (Ridge)

L2 regularization adds a penalty proportional to the sum of the squared weights in the network. It helps prevent overfitting by discouraging large weights in the encoder and decoder.

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reconstruction}} + \lambda \sum_{i,j} W_{ij}^2$$

Where:

- $\mathcal{L}_{\text{reconstruction}}$ is the reconstruction loss (like MSE or Binary Cross-Entropy).
- W_{ij} are the weights of the network.
- λ is the regularization strength (hyperparameter).

Regularization

L1 Regularization (Lasso)

L1 regularization encourages sparsity in the weight matrix by adding a penalty proportional to the sum of the absolute values of the weights:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reconstruction}} + \lambda \sum_{i,j} |W_{ij}|$$

L1 regularization can be useful when you want to force certain weights to be zero, leading to a more compact model.

Regularization

2. Sparse Regularization (KL Divergence)

Sparse autoencoders impose sparsity on the activations of the hidden units. This is often done by constraining the activations of the neurons to be close to zero for most inputs. A common approach is to minimize the Kullback-Leibler (KL) divergence between the average activation of a hidden neuron and a small value ρ (sparsity parameter).

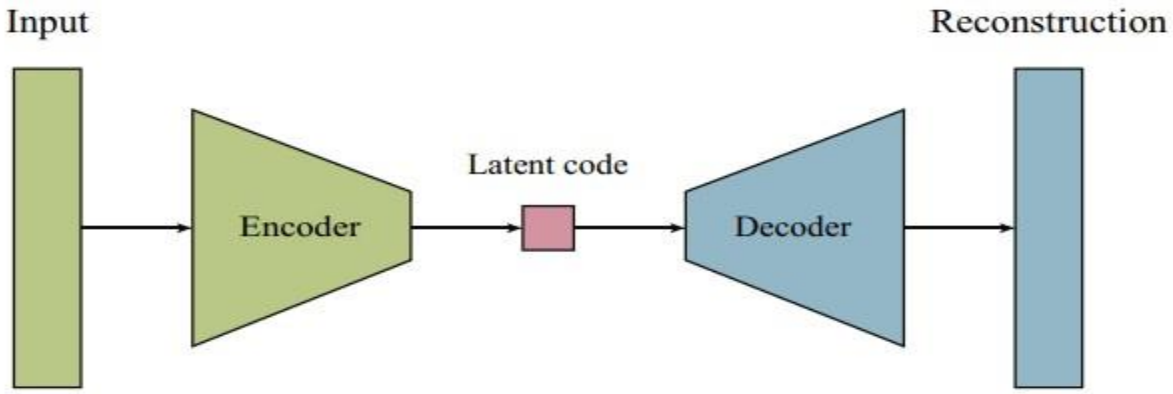
$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reconstruction}} + \beta \sum_{j} \text{KL}(\rho \parallel \hat{\rho}_j)$$

Where:

- $\hat{\rho}_j$ is the average activation of hidden neuron j.
- ρ is the desired sparsity level (a small value, usually between 0 and 1).
- β is a regularization hyperparameter that controls the importance of the sparsity constraint.

The KL divergence term ensures that the average activation $\hat{\rho}_j$ is close to ρ, thereby encouraging sparsity.

Undercomplete Autoencoder



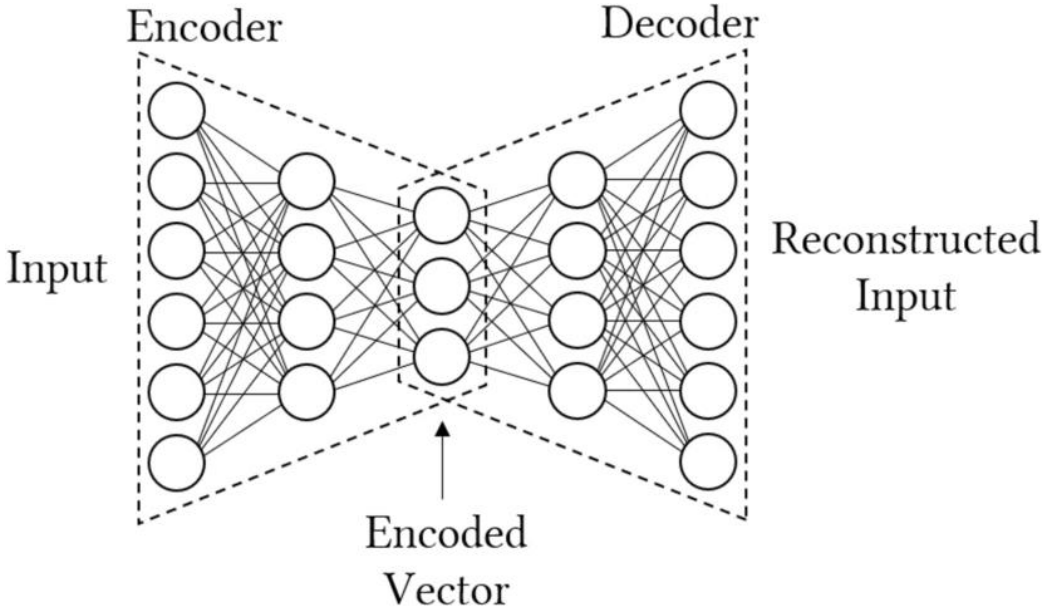
• An **undercomplete autoencoder** is a type of autoencoder where the size of the latent space (also called the bottleneck or hidden representation) is smaller than the input space.

• Here, the network is made insensitive to the input by restricting number of nodes in the hidden layer. Vanilla autoencoders can have a latent space that is equal to or larger than the input, while undercomplete autoencoders always have a smaller latent space.

•This forces the network to learn a more compact and efficient encoding of the input data, often capturing the most important features or patterns.

• For training such an autoencoder, we simply minimize the loss function, and we do not use any regularization function separately in the loss function

Undercomplete Autoencoder



1. Structure

- **Encoder**: The encoder compresses the input x into a lower-dimensional latent representation z (the bottleneck), such that $z = f_{\text{encoder}}(x)$.
- **Latent Space (Bottleneck)**: The latent representation z has a smaller dimension than the input, which constrains the capacity of the autoencoder and forces it to learn useful, compressed features.
- **Decoder**: The decoder reconstructs the input x from the latent representation z, such that $\hat{x} = f_{\text{decoder}}(z)$.

Because the latent space is smaller, the autoencoder cannot simply memorize the input, and instead must learn to capture the essential structure of the data.

Undercomplete Autoencoder

2. **Objective**

The goal of an undercomplete autoencoder is to minimize the **reconstruction error**, which measures how well the output $\hat{x}$ (reconstructed input) matches the original input x. The most common loss functions for this purpose are:

- **Mean Squared Error (MSE):** $\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$
- **Binary Cross-Entropy (BCE):** $\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left(x_i \log(\hat{x}_i) + (1 - x_i)\log(1 - \hat{x}_i)\right)$

3. **Key Properties**

- **Compression**: The latent space has fewer dimensions than the input, so the model has to compress the information. This can lead to the discovery of meaningful patterns or features in the data.
- **No Explicit Regularization Required**: Unlike other autoencoder variants (such as sparse autoencoders or denoising autoencoders), an undercomplete autoencoder relies on the bottleneck architecture itself for regularization. The network is constrained by its architecture to avoid learning trivial identity mappings because it doesn't have enough capacity to copy the input directly.

Undercomplete Autoencoder

4. Why Use an Undercomplete Autoencoder?

The purpose of using an undercomplete autoencoder is to learn compact and informative representations of data. This is useful in scenarios such as:

- **Dimensionality Reduction**: By reducing the dimensionality of the data in the latent space, undercomplete autoencoders can serve as an alternative to techniques like Principal Component Analysis (PCA).
- **Feature Learning**: The latent space may capture important features or patterns in the data, which can be useful for downstream tasks such as classification, clustering, or visualization.
- **Noise Reduction**: When the input contains noise, an undercomplete autoencoder is forced to focus on the most important features, often ignoring irrelevant noise in the process.

5. Relationship to PCA

In the case of a linear autoencoder (i.e., where the encoder and decoder are simple linear transformations), an undercomplete autoencoder can perform a similar function to Principal Component Analysis (PCA). It projects the input data into a lower-dimensional subspace that captures the maximum variance, assuming a linear relationship between features. However, undercomplete autoencoders can learn more complex, nonlinear mappings if non-linear activation functions (like ReLU or Sigmoid) are used in the encoder and decoder.

Undercomplete Autoencoder

6. Limitations

- **Loss of Information**: Since the latent space is smaller than the input, some information might be lost during the encoding process. This can be problematic if the latent space is too small to represent the necessary features for reconstruction.
- **Trivial Features in Nonlinear Models**: If the autoencoder is too powerful (i.e., has many layers or neurons), even an undercomplete autoencoder might still find ways to memorize the input. This can be mitigated by further regularization techniques or simpler models.

Example Architecture

Let's say we have an input image of size 28×28 pixels (784 input units). An undercomplete autoencoder might compress this input into a latent space of size 32 units. The architecture could look like:

- **Input Layer**: 784 units (28x28 image)
- **Encoder Layer**: Compress to 128 units
- **Latent Layer (Bottleneck)**: Compress to 32 units
- **Decoder Layer**: Reconstruct to 128 units
- **Output Layer**: Reconstruct back to 784 units (same as input size)

The small latent space (32 units) forces the network to learn a compact representation of the input image.

Undercomplete Autoencoder

- **Applications:**

  i) Dimensionality Reduction: An undercomplete autoencoder can be used for dimensionality reduction in large datasets, allowing for faster processing in subsequent tasks.

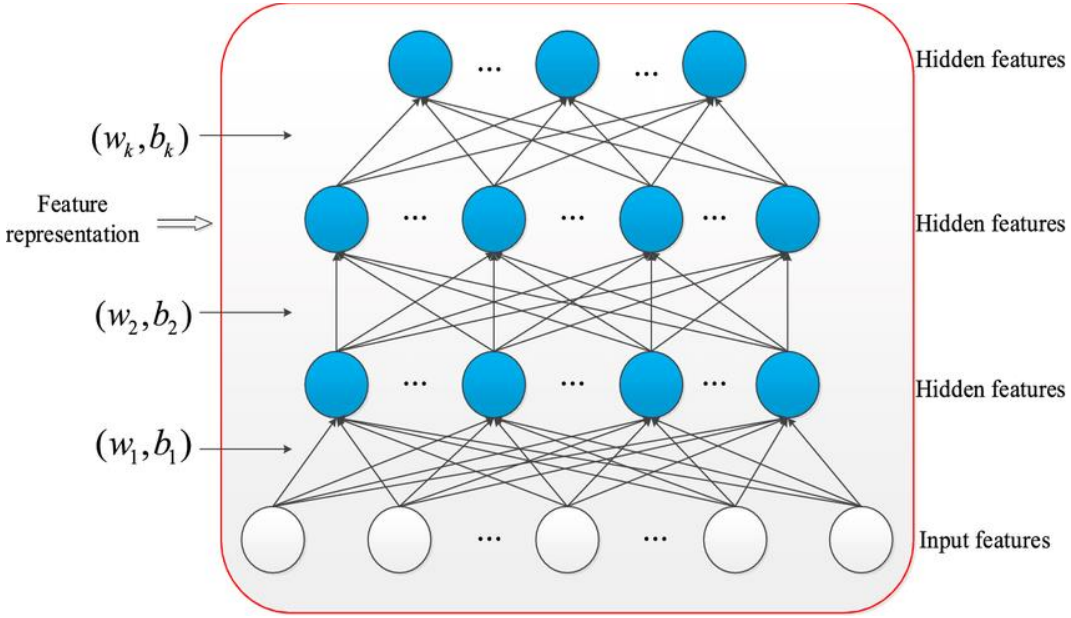
  ii) Anomaly Detection: If trained on normal data, the autoencoder will have trouble reconstructing anomalous inputs, making it a good tool for detecting outliers.

  iii) Pretraining for Deep Networks: The learned representations in the bottleneck can be used as feature representations for initializing deep neural networks (transfer learning).

- **Summary:**

- An **undercomplete autoencoder** is a neural network model designed to learn compact, efficient representations by restricting the size of the latent space.
- It achieves dimensionality reduction, feature extraction, and noise reduction by forcing the network to encode only the most important aspects of the input data.
- This constraint makes it an effective tool for many machine learning tasks, particularly when the goal is to capture the essential structure of the data.

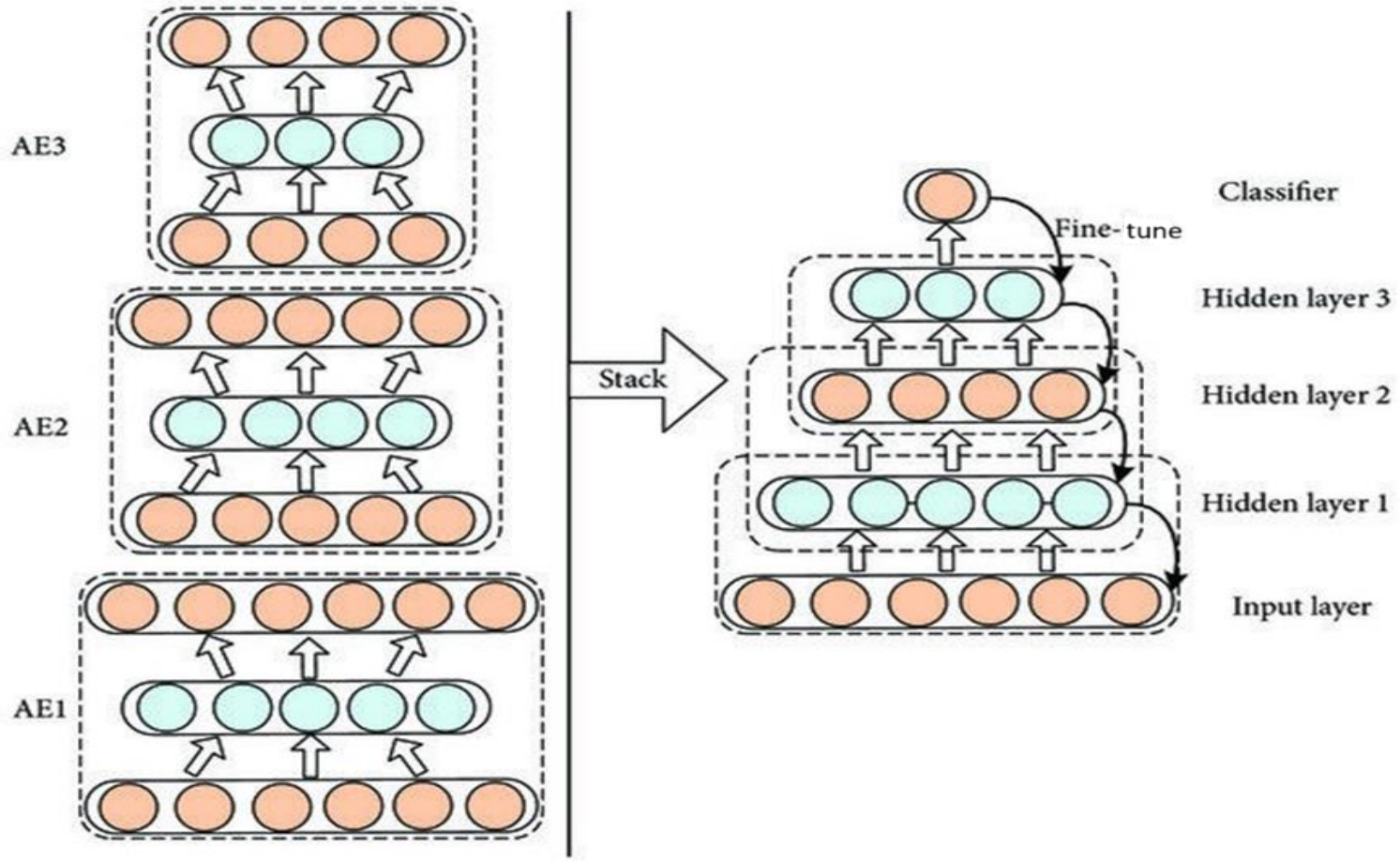
Stacked Autoencoder



- Stacked Autoencoders are a type of artificial neural network architecture used in unsupervised learning.
- They are designed to learn efficient data coding in an unsupervised manner, with the goal of reducing the dimensionality of the input data, and are particularly effective in dealing with large, high-dimensional datasets.

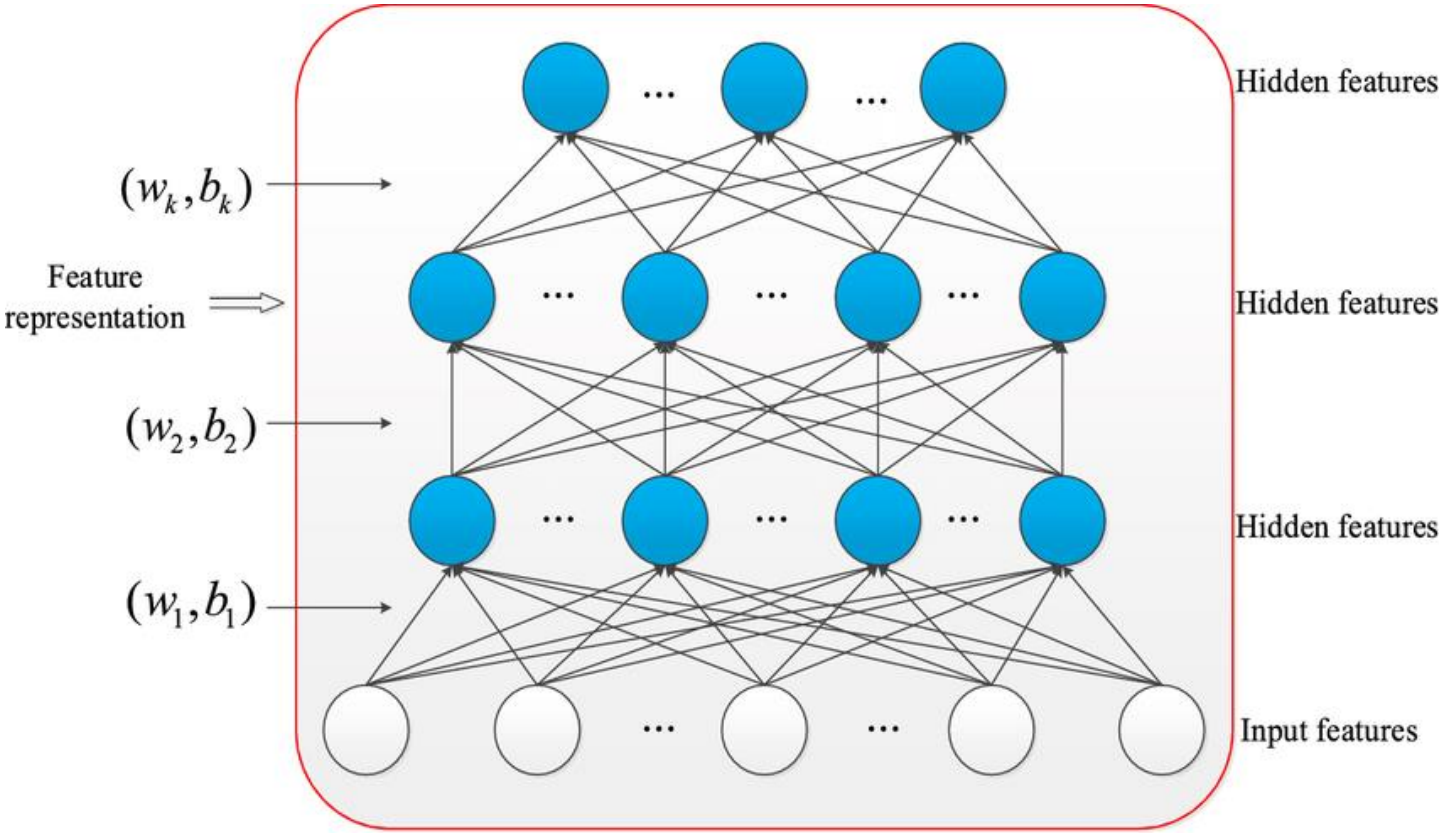
Stacked Autoencoder

• A Stacked Autoencoder (SAE) is a neural network that is composed of multiple layers of autoencoders, where each layer is trained on the output of the previous one.



SAE

• This “stacking” of autoencoders allows the network to learn more complex representations of the input data.



Structure of SAE

- **Layered Architecture**: A stacked autoencoder consists of several layers of autoencoders, where the output of one autoencoder serves as the input for the next. Each autoencoder typically consists of:
  - **Encoder**: Compresses the input data into a lower-dimensional representation.
  - **Decoder**: Reconstructs the original data from the compressed representation.
- **Training**: The layers can be

(i) pre-trained individually using unsupervised learning (typically by minimizing the reconstruction error), and then

(ii) fine-tuned together with supervised learning if labels are available.

How it works?

- In a stacked autoencoder, the output of the **latent layer** (the encoded representation) of one autoencoder is passed as the input to the next autoencoder. Here's how it works:
  - **Encoder Phase**: The first autoencoder takes the input data and encodes it into a lower-dimensional representation (the latent layer).
  - **Stacking**: This encoded representation is then used as the input for the next autoencoder, which has its own encoder and decoder.
  - **Decoding**: Each autoencoder reconstructs its input from its latent representation, but for stacking, we typically only use the latent outputs for the subsequent layers.
- As a summary, the latent layer output (not the decoder output) of the current autoencoder is used as the input for the next autoencoder. This allows each subsequent autoencoder to learn increasingly abstract representations of the data.

Benefits of SAE

- **Hierarchical Feature Learning**: Each layer can learn increasingly abstract features of the input data, similar to deep learning models.
- **Dimensionality Reduction**: The lower-dimensional representation can help reduce noise and improve performance for downstream tasks like classification or regression.
- **Improved Performance**: Stacking autoencoders can lead to better performance on tasks compared to a single autoencoder, especially with complex datasets.

SAE

- **Applications**

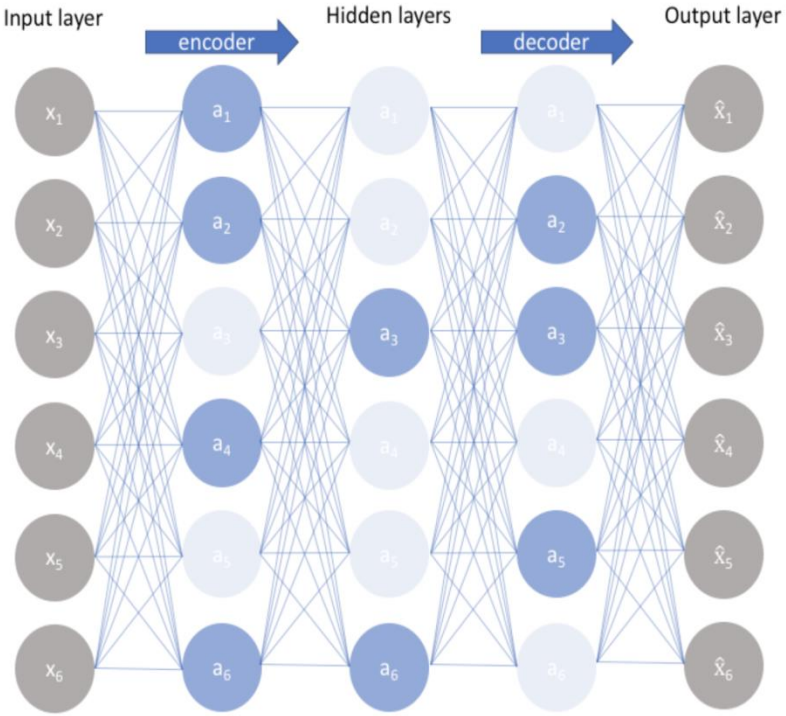
(i) Image Processing: For tasks like denoising or feature extraction.

(ii) Natural Language Processing: To capture semantic representations of words or sentences.

(iii) Anomaly Detection: By training on normal data and identifying deviations in reconstruction.

- Overall, stacked autoencoders are a powerful tool in unsupervised learning and can be leveraged in various applications to uncover meaningful patterns in data.

Sparse Autoencoder



Architecture:

- Like other autoencoders, a sparse autoencoder consists of an encoder and a decoder.
- The encoder compresses the input data into a latent representation, while the decoder reconstructs the input from this representation.

• A **sparse autoencoder** is a type of neural network that encourages sparsity in the latent representation of the data.
• This means that, for a given input, only a small number of neurons in the hidden layer are activated (represented by blue color),
• This leads to a more efficient and meaningful representation of the data.

Sparse Autoencoder: **Key Concepts**

- **Sparsity Constraint**:
  - The primary goal is to ensure that only a small fraction of the neurons are active (non-zero) for any given input. This is usually achieved by adding a sparsity penalty to the loss function.
  - Common methods include using **L1 regularization** on the activations or incorporating a sparsity constraint that compares the average activation of the neurons to a predefined sparsity parameter.

Loss Function

- The overall loss function for a sparse autoencoder typically consists of two parts:
  - **Reconstruction Loss**: Measures how well the autoencoder can reconstruct the input data from the latent representation.
  - **Sparsity Penalty**: Encourages the model to have a certain level of sparsity in the activations. A popular method is to use the Kullback-Leibler (KL) divergence between the average activation of the neurons and a target sparsity value (often close to zero).

$$\text{Total Loss} = \text{Reconstruction Loss} + \lambda \cdot \text{Sparsity Penalty}$$

• Here, λ is a hyperparameter that balances the two loss components.

Sparsity Penalty: Using KL Divergence

- The Kullback-Leibler (KL) divergence is a statistical measure that quantifies how one probability distribution diverges from a second, expected probability distribution.
- In the context of sparse autoencoders, it can be used to encourage sparsity in the activations of the hidden layer.
- **Basic Idea**
  - In a sparse autoencoder, we want to ensure that the average activation of the hidden neurons is close to a predefined target sparsity level, typically denoted as ρ (a small positive value, like 0.05).
  - The KL divergence helps measure how far the actual average activation of the hidden neurons (denoted as $\hat{\rho}$) from this target ρ.

Sparsity Penalty: Using KL Divergence

KL Divergence Formula

For binary variables (which can be a simplification for neurons), the KL divergence from $\hat{\rho}$ to ρ is defined as:

$$D_{KL}(\rho||\hat{\rho}) = \rho \log\left(\frac{\rho}{\hat{\rho}}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}}\right)$$

Where:

- ρ is the target sparsity (desired average activation).
- $\hat{\rho}$ is the actual average activation of the neurons.

Sparsity Penalty: Using KL Divergence

- Example:

Let's go through an example with specific values:

1. **Target Sparsity** (ρ): 0.05 (we want 5% of the neurons to be active on average).
2. **Actual Average Activation** ($\hat{\rho}$): Suppose we observe that the average activation of the hidden neurons during training is 0.1 (10% are active).

Now we can compute the KL divergence:

Step 1: Calculate KL Divergence

$$D_{KL}(0.05||0.1) = 0.05 \log\left(\frac{0.05}{0.1}\right) + (1-0.05)\log\left(\frac{1-0.05}{1-0.1}\right)$$

Sparsity Penalty: Using KL Divergence

- The first term:

$$0.05 \log\left(\frac{0.05}{0.1}\right) = 0.05 \log(0.5) \approx 0.05 \cdot (-0.693) \approx -0.03465$$

- The second term:

$$0.95 \log\left(\frac{0.95}{0.9}\right) = 0.95 \log(1.0556) \approx 0.95 \cdot 0.054 \approx 0.0513$$

Combining these gives:

$$D_{KL}(0.05||0.1) \approx -0.03465 + 0.0513 \approx 0.01665$$

Step 2: Incorporate into Loss Function

This KL divergence value would then be added to the overall loss function of the sparse autoencoder, weighted by a hyperparameter λ to control its influence:

$$\text{Total Loss} = \text{Reconstruction Loss} + \lambda \cdot D_{KL}(\rho||\hat{\rho})$$

Sparsity Penalty: Using KL Divergence

Interpretation

- If $\hat{\rho}$ is close to ρ, the KL divergence will be small, and the penalty will have less impact on the loss function, allowing the autoencoder to focus more on reconstruction.
- If $\hat{\rho}$ deviates significantly from ρ, the KL divergence increases, adding a larger penalty and encouraging the model to adjust the weights to produce a sparser representation.

- **Conclusion:** Using KL divergence for sparsity penalties effectively guides the training of the sparse autoencoder to produce representations that are both compact and meaningful, improving the quality of learned features.

Sparse Autoencoder

The loss function is: $\text{Total Loss} = \text{Reconstruction Loss} + \lambda \cdot \text{Sparsity Penalty}$

- **Implementation Considerations**

(i) Choosing Sparsity Level: Selecting an appropriate sparsity level (the target average activation) is crucial. It often requires experimentation based on the specific dataset and task.

(ii) Regularization Strength: The λ parameter needs to be tuned carefully to balance reconstruction quality and sparsity.

- **Applications**

(i) Image Processing: Useful for tasks like denoising, where the model learns to represent images with minimal active features.

(ii) Anomaly Detection: Since the model learns a compact representation, deviations from the normal pattern (anomalies) can be more easily identified by examining the activations.

(iii) Natural Language Processing: Can be employed to learn meaningful representations of textual data, capturing essential features while ignoring noise.

Sparse Autoencoder

- **Benefits**

(i) Feature Extraction: By promoting sparsity, these autoencoders tend to learn more meaningful and interpretable features, which can be useful for downstream tasks.

(ii) Dimensionality Reduction: Sparse representations can be more efficient in capturing the underlying structure of the data, which is beneficial for reducing dimensionality.

(iii) Robustness: Sparsity can enhance the model's robustness to noise and irrelevant variations in the data.

Denoising Autoencoder

- A **denoising autoencoder (DAE)** is a type of autoencoder specifically designed to learn robust representations of data by reconstructing clean input from noisy versions.
- This approach helps the model learn to filter out noise and can improve its ability to generalize to unseen data.
- **Architecture**:
  - Like a standard autoencoder, a denoising autoencoder consists of an encoder and a decoder.
  - The encoder compresses the noisy input data into a lower-dimensional latent representation, and the decoder reconstructs the original clean input from this representation.

Denoising Autoencoder

- **Input Corruption**: During training, the original input data is intentionally corrupted. Common methods of corruption include:
  - **Adding Gaussian Noise**: Random noise is added to the input features.
  - **Dropout**: Randomly setting a fraction of input units to zero.
  - **Salt-and-Pepper Noise**: Randomly replacing some input pixels with maximum and minimum values (for images).
- The model learns to reconstruct the original clean input from this corrupted version.

Denoising Autoencoder : Loss Function

- The objective is to minimize the reconstruction loss, typically using mean squared error (MSE) or binary cross-entropy, comparing the reconstructed output with the original clean input.

$$\text{Loss} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2$$

Where:

- x_i is the original input.
- $\hat{x}_i$ is the output of the autoencoder.

Denoising Autoencoder : Loss Function

- **Though** the added noise during training acts as a form of regularization to prevent overfitting, but some other regularizers are also helpful.
- **Common Regularization Techniques are:**
- **Dropout**: Randomly dropping units during training can help prevent co-adaptation of neurons, making the model more robust.
- **Weight Regularization**: Adding L1 or L2 penalties to the weights in the loss function can help keep the model from becoming overly complex.
- **Early Stopping**: Monitor validation loss and stop training when it starts to increase can prevent overfitting.

Denoising Autoencoder

- **Benefits**

(i) Robust Feature Learning: By learning to reconstruct from noisy inputs, DAEs can capture more robust features that generalize well to new, unseen data.

(ii) Regularization: The added noise during training acts as a form of regularization, helping to prevent overfitting.

(iii) Improvements in Data Quality: Can be effectively used for tasks such as denoising images, speech signals, or any data that is prone to noise.

Applications of DAE

- **Image Denoising**: Removing noise from images while preserving important features.
- **Speech Processing**: Enhancing speech signals that have been corrupted by noise.
- **Anomaly Detection**: Identifying outliers by comparing the reconstruction error.
- **Data Augmentation**: Improving the robustness of other models by training them on denoised representations (i.e., clean generated data).

Implementation Steps of DAE

1. **Corrupt Input**: Define the method of corruption (e.g., adding noise) and apply it to the training data.
2. **Train the Model**:
   - Feed the corrupted inputs into the encoder.
   - Use the decoder to reconstruct the original clean inputs.
   - Calculate the reconstruction loss and backpropagate the error to update the model weights.
3. **Evaluate Performance**: After training, evaluate the model on clean and noisy data to see how well it denoises new inputs.

Denoising Autoencoder: Example

- **Input Data**: Consider an image of a handwritten digit (like from the MNIST dataset).
- **Corrupted Input**: Add Gaussian noise to the image, resulting in a noisy version that contains random pixel variations.
- **Training**: The model is trained using the noisy image as input and the original clean image as the target output.
- **Reconstruction**: Once trained, when provided with a new noisy image, the model can effectively reconstruct a cleaner version of the original image.

Denoising Autoencoder

- Denoising autoencoders are powerful tools for learning robust representations of data in the presence of noise.
- By reconstructing clean inputs from corrupted versions, they help improve generalization, prevent overfitting, and enhance data quality in various applications.

Applications: Autoencoders

(i) **Dimensionality Reduction**: Similar to PCA, but can capture more complex patterns.

(ii) **Data Denoising**: Removing noise from data by training on noisy inputs and expecting the clean output.

(iii) **Anomaly Detection**: Identifying unusual patterns by analyzing reconstruction errors.

(iv) **Generative Modeling**: Some variants (like variational autoencoders) can generate new data similar to the training set.

- Autoencoders can be simple feedforward networks, convolutional networks, or even recurrent networks, depending on the type of data being processed.

Thank You!