Chapter-5

More Deep Learning Models, Architectures & Applications

Instructor Name: B N V Narasimha Raju

5.1 Alexnet

AlexNet is a classic convolutional neural network (CNN) that gained prominence after its success in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It consists of eight layers, including five convolutional layers and three fully connected layers, using traditional stacked convolutional layers with max-pooling in between. This deep network structure allows AlexNet to effectively extract complex features from images.

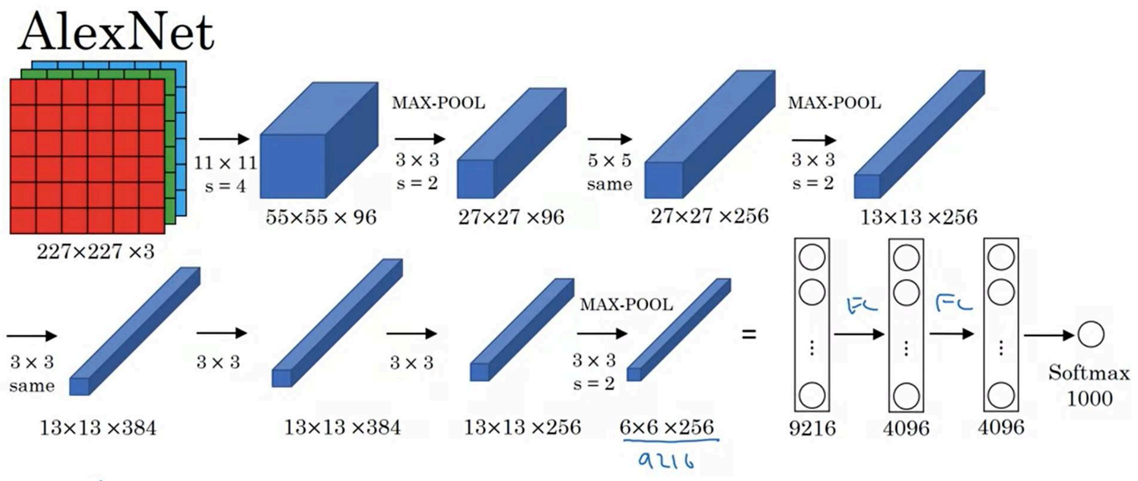
A key feature of its architecture is the use of overlapping pooling layers, which help reduce spatial dimensions while retaining spatial relationships among neighboring features. AlexNet also employs the ReLU (Rectified Linear Unit) activation function, which captures nonlinear relationships within the data and speeds up training by avoiding the vanishing gradient problem. The ReLU activation function is used after every convolutional and fully connected layer, except for the final output layer which uses softmax. Dropout regularization is applied to the fully connected layers to prevent overfitting, which is crucial due to the large number of parameters in the network.

One of the notable innovations of AlexNet was its computational efficiency; it introduced the use of parallel computing by utilizing two GPUs during training, enabling faster processing and handling large-scale data. In the 2012 ImageNet challenge, AlexNet produced groundbreaking results, outpacing previous CNN architectures and setting the stage for the resurgence of deep learning in computer vision.

Several architectural improvements, including the use of ReLU, overlapping pooling, and dropout, were key to its success, enhancing the model's performance and generalization ability. The network architecture is given below:

Model Explanation: The model described uses an input image of size 227x227x3, representing an RGB image with a height and width of 227 pixels. The first layer is a convolutional layer with 96 filters of size 11x11, applied with 'valid' padding and a stride of 4. The formula for the output size is given by the equation:

*floor(((n + 2*padding -filter)/stride) + 1) * floor(((n + 2*padding —filter)/stride) + 1)*



This formula is for square input with *height = width = n*. Explaining the first Layer with input 227x227x3 and Convolutional layer with 96 filters of 11x11 , 'valid' padding and stride = 4 , output dims will be

*= floor(((227 + 0–11)/4) + 1) * floor(((227 + 0–11)/4) + 1)*

*= floor((216/4) + 1) * floor((216/4) + 1)*

*= floor(54 + 1) * floor(54 + 1)*

*= 55 * 55*

For an input size of 227 and filter size of 11, the output dimensions become 55x55. Since there are 96 filters, the output of this convolutional layer is 55x55x96. This is followed by a max-pooling layer with a 3x3 filter and a stride of 2, reducing the output dimensions to 27x27x96. The next convolutional layer uses 256 filters of size 5x5 with 'same' padding, keeping the dimensions at 27x27x256.

Another max-pooling layer follows, reducing the size further to 13x13x256. The model then applies two convolutional layers sequentially, both using 384 filters of size 3x3 with 'same' padding, producing an output of 13x13x384. The model then applies a convolutional layer using 256 filters resulting in an output of 13x13x256.

A final max-pooling layer reduces the size to 6x6x256. The output is then flattened into a vector of 9216 units (6x6x256). This is followed by two fully connected layers with 4096 units each, and finally, a softmax layer with 6 units that classifies the input image into one of 6 classes, providing probabilities for each class.

5.2 VGGNet

The Visual Geometry Group (VGG) models, particularly VGG-16 and VGG-19, have significantly influenced the field of computer vision since their inception. These models stood out for their deep convolutional neural networks (CNNs) with a uniform architecture. VGG-19, the deeper variant of the VGG models, has garnered considerable attention due to its simplicity and effectiveness.

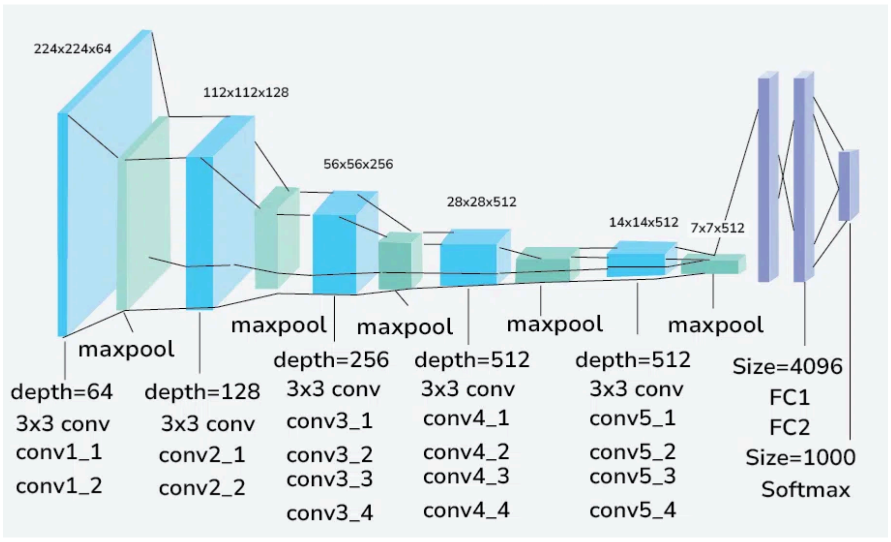
5.2.1 VGG-19 Architecture

VGG-19 is a deep convolutional neural network with 19 weight layers, comprising 16 convolutional layers and 3 fully connected layers. The architecture follows a straightforward and repetitive pattern, making it easier to understand and implement.

The key components of the VGG-19 architecture are:

- Convolutional Layers: 3x3 filters with a stride of 1 and padding of 1 to preserve spatial resolution.
- Activation Function: ReLU (Rectified Linear Unit) applied after each convolutional layer to introduce non-linearity.
- Pooling Layers: Max pooling with a 2x2 filter and a stride of 2 to reduce the spatial dimensions.
- Fully Connected Layers: Three fully connected layers at the end of the network for classification.
- Softmax Layer: Final layer for outputting class probabilities.

The VGG-19 model consists of five blocks of convolutional layers, followed by three fully connected layers. Here is a detailed breakdown of each block:



- Block 1
  - Conv1_1: 64 filters, 3x3 kernel, ReLU activation
  - Conv1_2: 64 filters, 3x3 kernel, ReLU activation
  - Max Pooling: 2x2 filter, stride 2
- Block 2
  - Conv2_1: 128 filters, 3x3 kernel, ReLU activation
  - Conv2_2: 128 filters, 3x3 kernel, ReLU activation
  - Max Pooling: 2x2 filter, stride 2
- Block 3
  - Conv3_1: 256 filters, 3x3 kernel, ReLU activation
  - Conv3_2: 256 filters, 3x3 kernel, ReLU activation
  - Conv3_3: 256 filters, 3x3 kernel, ReLU activation
  - Conv3_4: 256 filters, 3x3 kernel, ReLU activation
  - Max Pooling: 2x2 filter, stride 2
- Block 4
  - Conv4_1: 512 filters, 3x3 kernel, ReLU activation
  - Conv4_2: 512 filters, 3x3 kernel, ReLU activation
  - Conv4_3: 512 filters, 3x3 kernel, ReLU activation
  - Conv4_4: 512 filters, 3x3 kernel, ReLU activation
  - Max Pooling: 2x2 filter, stride 2
- Block 5
  - Conv5_1: 512 filters, 3x3 kernel, ReLU activation
  - Conv5_2: 512 filters, 3x3 kernel, ReLU activation
  - Conv5_3: 512 filters, 3x3 kernel, ReLU activation
  - Conv5_4: 512 filters, 3x3 kernel, ReLU activation

    - Max Pooling: 2x2 filter, stride 2
- Fully Connected Layers
    - FC1: 4096 neurons, ReLU activation
    - FC2: 4096 neurons, ReLU activation
    - FC3: 1000 neurons, softmax activation (for 1000-class classification)

5.2.2 Architectural Design Principles

The VGG-19 architecture follows several key design principles:

- Uniform Convolution Filters: Consistently using 3x3 convolution filters simplifies the architecture and helps maintain uniformity.
- Deep Architecture: Increasing the depth of the network enables learning more complex features.
- ReLU Activation: Introducing non-linearity helps in learning complex patterns.
- Max Pooling: Reduces the spatial dimensions while preserving important features.
- Fully Connected Layers: Combines the learned features for classification.

5.2.3 Impact and Legacy of VGG-19

VGG-19's primary legacy is demonstrating that increasing network depth with a simple, uniform architecture significantly boosts image recognition performance.

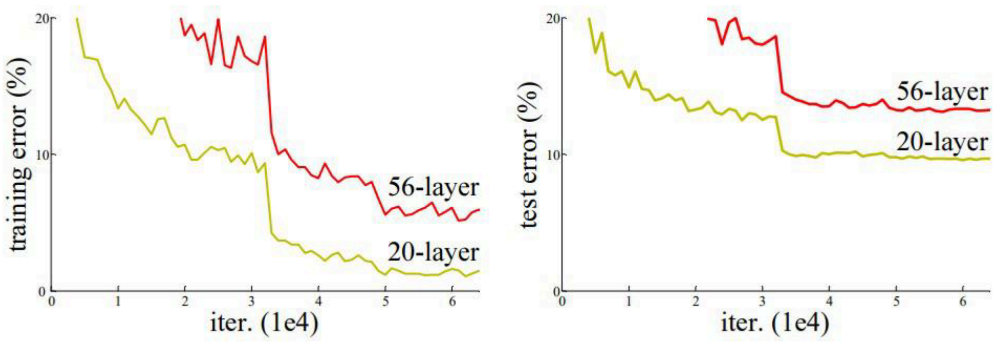
- Architectural Influence: Its design, which exclusively uses small 3x3 convolutional filters, became a foundational principle for subsequent advanced models like ResNet and Inception.
- Transfer Learning Powerhouse: It is extensively used as a pre-trained feature extractor. Models trained on ImageNet are fine-tuned for diverse tasks like object detection, medical imaging, and style transfer.
- Research Benchmark: It serves as a crucial baseline in academic and industry research for comparing and validating new architectures.

5.3 ResNet

ResNet, short for Residual Network, is a deep neural network architecture introduced by Kaiming He et al. in 2015. It became highly successful due to its use of residual (or skip) connections, which help mitigate the vanishing gradient problem and enable training of much deeper networks. In the context of Faster R-CNN, replacing the VGG-16 backbone with ResNet-101 resulted in a significant improvement, with detection accuracy increasing by approximately 28%. Additionally, ResNet demonstrated the feasibility of training networks with exceptionally deep architectures, such as 100-layer and even 1000-layer models, without suffering from the typical issues encountered in deep learning, like degradation in performance. The incorporation of residual connections allowed ResNet to maintain manageable levels of training error even in very deep networks.

5.3.1 Need for ResNet

In deep learning, increasing the depth of a network by adding more layers is often considered a strategy for addressing complex problems, as it allows the network to learn more complex and abstract features. For example, in image recognition tasks, initial layers may detect simple features like edges, while deeper layers may identify more complex structures like textures and objects. This approach typically improves accuracy and performance. However, in traditional Convolutional Neural Networks (CNNs), there is a threshold beyond which adding more layers can lead to performance degradation. This phenomenon is illustrated in the plot, which compares the error percentages of a 20-layer network to a 56-layer network.

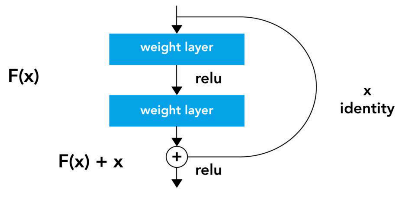


As shown in the plot, the error percentage for the 56-layer network is higher than that of the 20-layer network for both training and testing data. This suggests that increasing the depth of the network beyond a certain point may not necessarily lead to better performance. Several factors could contribute to this decline, including issues with the optimization function, improper network initialization, and the vanishing gradient problem. While overfitting might be a potential concern in some cases, it is not the cause in this scenario. Overfitting typically results in low training error but higher testing error, while the 56-layer network shows higher error percentages for both training and testing data, indicating that the performance degradation is likely due to the network's excessive depth rather than overfitting. ResNet addresses these issues by using residual connections or skip connections, which allow gradients to flow more easily through the network during training.

5.3.2 Residual Block

The challenge of training very deep networks has been significantly alleviated with the introduction of ResNet (Residual Networks), which are composed of residual blocks. The key feature of ResNet is the presence of a "skip connection" that bypasses one or more layers, allowing the output of a layer to be directly added to the output of a deeper layer. This skip connection is a core component of residual blocks. In traditional networks, the output of a layer is calculated by multiplying the input *x* by the layer's weights, adding a bias term, and passing the result through an activation function *f*() to obtain the output, *H*(*x*). This can be represented as:

$$H(x) = f(wx + b) \text{ or } H(x) = f(x)$$



However, with the skip connection in ResNet, the output becomes $H(x)=f(x)+x$, where the input *x* is added directly to the output of the activation function. A potential issue with this approach arises when the dimensions of the input *x* differ from those of the output, which can occur due to operations like convolution or pooling. In such cases, two solutions can address the dimensional mismatch:

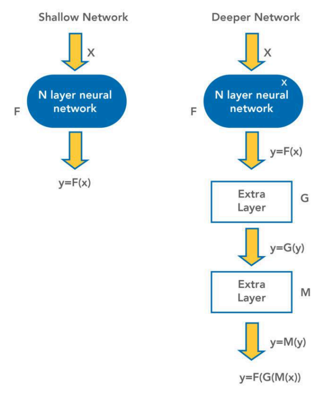
- **Padding**, where the skip connection is padded with zeros to match the dimensions of the output.
- **Projection**, where a 1×1 convolutional layer is used to match the dimensions, introducing an additional parameter *w1*. In this case, the output becomes: $H(x) = f(x) + w1.x$

The projection method differs from padding in that it introduces extra parameters, providing more flexibility for dimension matching without losing information.

5.3.3 How ResNet helps

ResNet helps address the vanishing gradient problem in deep neural networks through the use of skip connections. These connections provide an alternate shortcut path for the gradient to flow through, allowing gradients to bypass certain layers and reach deeper layers more easily. This ensures that the network can continue to learn effectively, even with a deeper architecture. Additionally, skip connections enable the model to learn identity functions, meaning that higher layers can perform at least as well as lower layers, preventing performance degradation that typically occurs in plain neural networks without residual blocks.

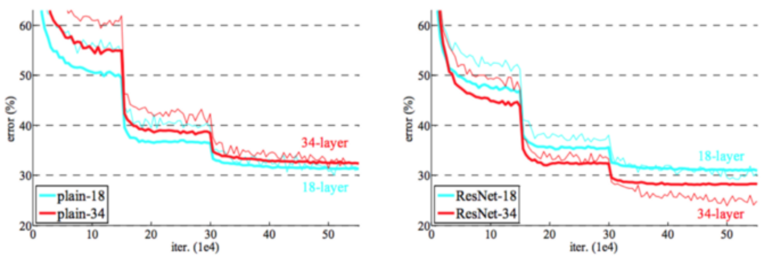
To understand this more clearly, consider a shallow network and a deep network, both mapping an input '*x*' to an output '*y*' using the function $H(x)$. Ideally, the deep network should perform at least as well as the shallow network, without any degradation in performance. One way to achieve this is for the additional layers in the deep network to learn the identity function, where their outputs equal their inputs, ensuring no loss in performance despite the added depth. In traditional networks, learning the identity function is difficult, as the output is $H(x) = f(x)$, and for the identity function to hold, $f(x)$ must equal x. In contrast, ResNet simplifies this by introducing skip connections, where the output becomes $H(x) = f(x) + x$. Now, for the network to learn the identity function, all that is needed is for $f(x)$ to equal zero, making it easier to achieve $H(x) = x$.



G and M act as Identity Functions. Both the Network Give same output

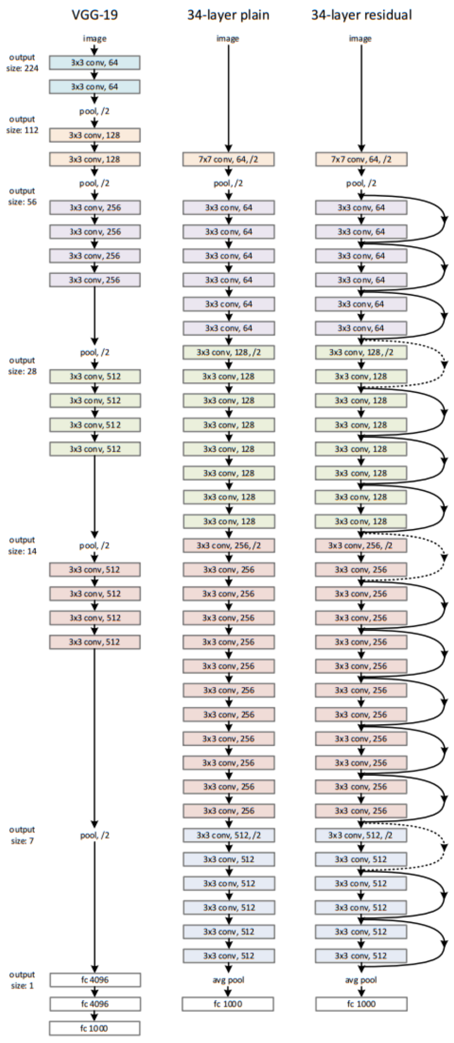
Skip connections and identity functions are crucial components of ResNet, allowing the network to train deeper architectures without suffering from the vanishing gradient problem. The use of residual blocks helps gradients flow more effectively through the network, enabling it to learn from information at different depths. Furthermore, the introduction of identity functions allows the network to focus on learning residual functions, the difference between the input and the desired output, rather than the entire mapping, which is easier to optimize. In the best-case scenario, deeper networks in ResNet can better approximate the mapping from '*x*' to '*y*' than shallower networks, reducing error rates significantly.

As demonstrated in a plot comparing error rates, ResNet-34 (with 34 layers) achieves a much lower error rate compared to a plain 34-layer network. For 18-layer networks, the error rates of plain-18 and ResNet-18 are nearly identical, indicating that ResNet's benefits are more pronounced in deeper networks. This illustrates how ResNet has dramatically improved the performance of deeper networks by effectively addressing the vanishing gradient problem and enabling the optimization of much deeper architectures.



5.3.4 ResNet Architecture

The ResNet network uses a 34-layer plain network architecture inspired by VGG-19 in which then the shortcut connection is added. These shortcut connections then convert the architecture into the residual network as shown in the figure below:



5.4 Transfer learning

Traditional machine learning involves isolated, single-task learning, where each model is trained independently on a specific dataset for a specific task. The knowledge acquired during training is not retained or reused for future tasks, meaning each new task requires training a model from scratch—often demanding large amounts of labeled data. This can become inefficient and ineffective, especially when the new task lacks sufficient data.

Transfer learning, on the other hand, is a machine learning technique that allows a model trained on one task to be repurposed for a second, related task. The core idea is to leverage the knowledge—such as learned features or model weights—from a previously trained model to improve learning on a new but related task. For instance, if a model has been trained to identify objects in restaurant images (Task T1), its learned features—like edges, shapes, and textures—can be transferred to help recognize objects in park or café images (Task T2), even when there is less data available for T2. This process can lead to faster training, improved accuracy, and reduced data requirements for the new task.

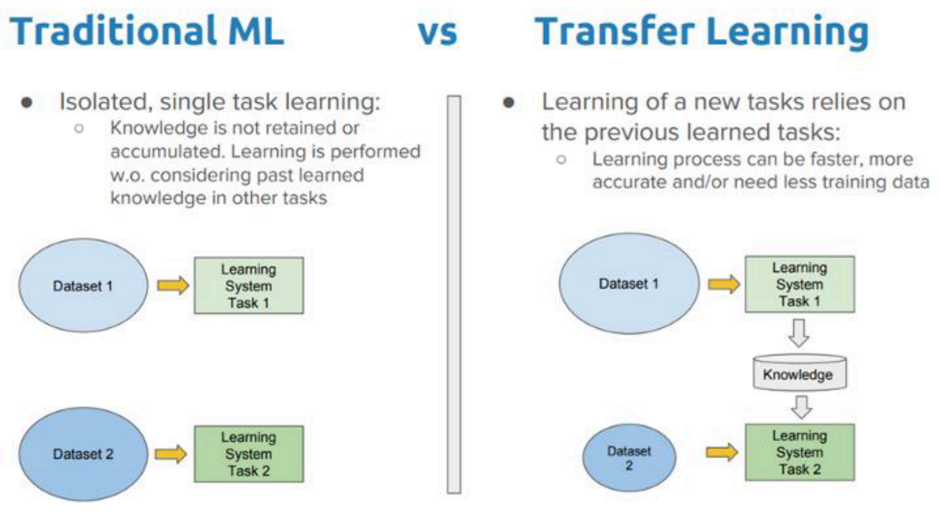

Traditional ML
vs
Transfer Learning
Isolated, single task learning:
Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks
Learning of a new tasks relies on the previous learned tasks:
Learning process can be faster, more accurate and/or need less training data
Dataset 1
Learning System Task 1
Dataset 2
Learning System Task 2
Dataset 1
Learning System Task 1
Knowledge
Dataset 2
Learning System Task 2

Figure: Traditional ML vs Transfer Learning

The above figure illustrates the key differences between the two approaches to machine learning. Traditional machine learning is shown as an isolated learning process. Each task—represented by Dataset 1 and Dataset 2—is learned separately using independent models for Task 1 and Task 2. There is no sharing or retention of knowledge between tasks. The model trained on Dataset 1 cannot assist or enhance the training of Dataset 2. This approach becomes inefficient when data is limited, as every new task starts from scratch without leveraging prior experience.

In contrast, transfer learning enables the reuse of knowledge from a previously learned task to enhance performance on a new, related task. The model is first trained on Dataset 1 for Task 1, gaining valuable knowledge—such as learned features or network weights. This knowledge is then transferred to assist in learning Task 2 using Dataset 2. As a result, the second task benefits from a more informed starting point, leading to faster training, reduced data needs, and better accuracy.

In the realm of deep learning, transfer learning is typically implemented by pretraining a neural network on a large benchmark dataset (e.g., ImageNet) and then fine-tuning it on a target dataset. This method is widely used because training deep networks from scratch is often costly in terms of time and computational resources. Transfer learning has proven to be highly effective for rapid prototyping, performance improvement, and resource-efficient training—especially when labeled data is limited. By mimicking the human ability to apply prior knowledge to new contexts, transfer learning brings a more adaptive and scalable approach to machine learning.

5.4.1 Transfer Learning Base Models

To design an efficient neural network using transfer learning, it's important to understand the various base models available. These base models serve as the foundation from which knowledge—referring to the network architecture and learned weights—is transferred to a new task-specific model. Broadly, base models can be categorized into two groups: the first is based on convolutional neural network (CNN) architectures commonly used in computer vision and classification tasks; the second is based on models designed for sequential data, primarily used in natural language processing (NLP) tasks.

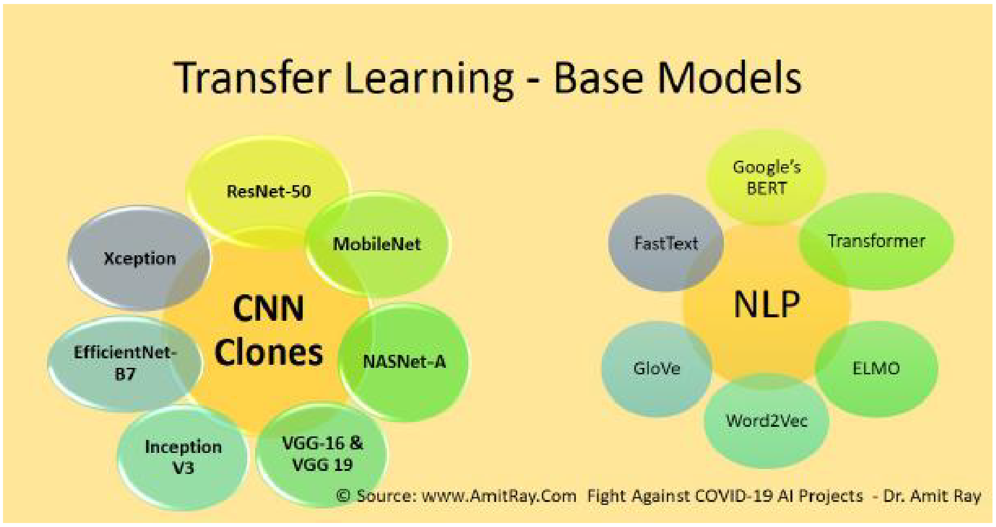
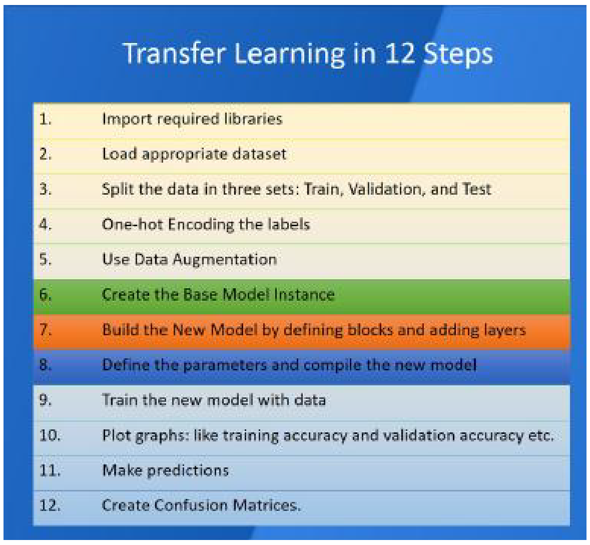


Figure: Transfer Learning - Base Models

In computer vision, ImageNet pre-trained CNN models are widely used as the standard base models for transfer learning. ImageNet is a large-scale database containing over 1.2 million labeled natural images across more than 1000 classes, with each class having at least 1000 images. CNN models trained on this dataset have demonstrated remarkable performance and generalization, making them ideal candidates for transfer learning in various vision tasks. Early architectures such as LeNet-5, AlexNet, VGGNet, GoogLeNet, and ResNet form the foundation of this approach. More advanced and widely adopted base models include ResNet-50, MobileNet, NASNet-A, VGG-16, VGG-19, Inception V3, EfficientNet-B7, and Xception.

For sequential transfer learning, especially in NLP tasks, popular base models include ULMFiT, Word2Vec, GloVe, FastText, and more recent transformer-based models like BERT, ELMo, and Transformer. These models have been trained on large text corpora and can capture rich semantic representations, making them highly effective for downstream tasks like sentiment analysis, question answering, and text classification.



Transfer Learning in 12 Steps

1.	Import required libraries
2.	Load appropriate dataset
3.	Split the data in three sets: Train, Validation, and Test
4.	One-hot Encoding the labels
5.	Use Data Augmentation
6.	Create the Base Model Instance
7.	Build the New Model by defining blocks and adding layers
8.	Define the parameters and compile the new model
9.	Train the new model with data
10.	Plot graphs: like training accuracy and validation accuracy etc.
11.	Make predictions
12.	Create Confusion Matrices.

Figure: Steps in Transfer Learning

Among the steps in transfer learning, Step 7 – Building the New Model – is the most critical. This step involves integrating the pre-trained base model into a new architecture tailored to the target task, often by adding new layers and fine-tuning them as needed. The success of transfer learning heavily relies on how well this step is executed, ensuring that the transferred knowledge is effectively adapted to the new domain.

5.4.2 Model Building Strategies in Transfer Learning

Feature Extraction: In this strategy, the model uses a pre-trained network as a feature extractor. The weights of the feature extraction layers (typically the initial layers) are frozen, meaning they are not updated during training. The output from these frozen layers is then passed to a new model or a traditional machine learning algorithm like random forest (RF), support vector machine (SVM), k-nearest neighbors (kNN), decision tree (DT), or naive Bayes (NB) for further processing. This approach is computationally efficient since feature extraction is done once and reused for similar tasks, saving on resources.

Fine-Tuning: Unlike feature extraction, fine-tuning involves unfreezing some of the layers of the pre-trained model and adjusting the weights for the specific target task. This allows the model to adapt more specifically to the new task, leveraging the general features learned from the original data. Fine-tuning speeds up convergence compared to training a model from scratch. While all the weights are typically updated during fine-tuning, only the final layers are adjusted in feature extraction, and only these weights are updated in the training process for the target task. Depending on the nature of the task, a combination of both approaches might be beneficial.

5.4.3 Steps for Fine-tuning the New Model

- Build the new network model on top of an already trained base model.
- Freeze the base network.
- Train the part you added.
- Unfreeze some layers in the base network.
- Jointly train both these layers and the part you added.

5.4.4 Basic Building Blocks of CNN Based Transfer Learning Models

In CNN-based transfer learning models, there are several key building blocks, which are crucial for feature extraction and classification tasks. The figure explains the basic building blocks of transfer learning.

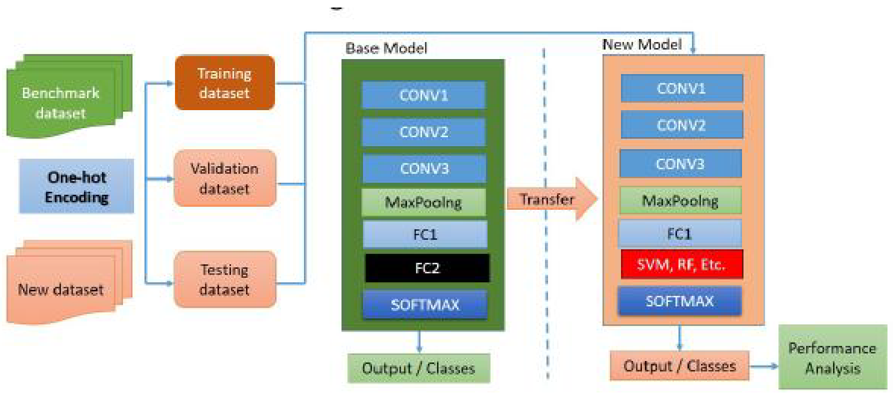


Figure: basic building blocks of transfer learning

5.4.4.1 Convolution, ReLU and Pooling Blocks

The first set of blocks in the CNN architecture typically includes convolutional layers (CONV), ReLU (Rectified Linear Unit) activation functions, and pooling layers. These blocks collectively perform feature extraction, which is similar to feature extraction in traditional machine learning models. Each convolutional layer uses a set of convolutional kernels to detect various patterns or features in the input data.

5.4.4.2 Pooling Layers

The pooling layer serves two primary purposes: it performs feature selection and reduces the dimensionality of the data while maintaining the most important features.

Common pooling methods include maximum pooling, mean pooling, and random pooling. Maximum pooling selects the largest value in a specified region.

5.4.4.3 Fully connected layers

The fully connected (FC) layers typically sit near the end of a CNN architecture. These layers integrate the information from the earlier layers and are used for classification tasks. Output layer is also called the softmax layer, which maps the output of the fully connected layer to (0, 1) using the softmax function.

5.5 Object Detection

Object detection is a computer vision technique used to find and identify objects within an image or video. It has wide-ranging applications across many different industries. At its core, object detection answers two main questions about an image: "Which objects are present?" and "Where are these objects located?". This involves two simultaneous processes:

- Classification: Figuring out what the object is (e.g., a dog, a car, or a person).
- Localization: Pinpointing the exact position of the object in the image, usually by drawing a box around it.

5.5.1 Key Components of Object Detection

Object detection combines the following key ideas:

- Image Classification: This is the process of assigning a single label to an entire image. For example, labeling a picture as "a day at the park." It tells you what's in the image but not where.
- Object Localization: This step goes further by finding the position of an object and drawing a bounding box around it.
- Object Detection: This merges both classification and localization. It finds multiple different objects in an image, identifies what they are, and draws a box around each one to show its location.

5.5.2 Working of Object Detection

The process generally follows these steps:

- Input Image: It all starts with an image or a video that needs to be analyzed.

- Pre-processing: The image is prepared and formatted so the detection model can use it properly.
- Feature Extraction: A model, often a Convolutional Neural Network (CNN), scans the image to find interesting regions and pull out key features (like shapes, colors, and textures) that help identify objects.
- Classification: Based on the extracted features, each region is classified into a category, like "cat" or "dog". This is often done using a neural network that calculates the probability of an object being in that region.
- Localization: At the same time, the model calculates the coordinates for a bounding box to draw around each object it finds.
- Non-max Suppression: Sometimes the model detects the same object multiple times with overlapping boxes. This technique cleans up the extras by keeping only the box with the highest confidence score and removing the rest.
- Output: The final result is the original image with labeled bounding boxes showing all the detected objects.

5.5.3 Deep Learning Methods for Object Detection

Deep learning has revolutionized object detection. The methods are primarily divided into two main types:

- Two-Stage Detectors: These methods first propose potential regions where objects might be and then classify those regions in a second step. Examples include R-CNN, Fast R-CNN, and Faster R-CNN.
- Single-Stage Detectors: These models are more direct. They predict the bounding boxes and the object classes in a single pass, making them very fast. Popular examples are YOLO and SSD.

5.5.4 Two-Stage Detectors for Object Detection

These detectors are known for their high accuracy and work in two steps. The main types are:

- R-CNN (Regions with Convolutional Neural Networks): This technique first generates about 2000 "region proposals" from an image using a selective search algorithm. Each proposed region is then passed through a CNN model to classify the object inside it.
- Fast R-CNN: An improvement on R-CNN, this method processes the entire image with a CNN first to create a feature map. It then uses this map to extract features for each region, making the process much faster.

- Faster R-CNN: This version introduces a Region Proposal Network (RPN), which allows the model to learn where to look for objects directly from the feature maps. This makes the proposal step much more efficient than the selective search used in the original R-CNN.

5.5.5 Single-Stage Detectors for Object Detection

These detectors are built for speed and combine localization and classification into a single step. The two most popular models are:

- SSD (Single Shot MultiBox Detector): SSD examines the image once and predicts bounding boxes and classes using feature maps of different sizes. This allows it to detect objects of various scales efficiently in a single neural network pass.
- YOLO (You Only Look Once): YOLO also processes the entire image in one go. It divides the image into a grid and predicts bounding boxes and class probabilities for each grid cell. This approach is extremely fast, making it ideal for real-time object detection.

5.5.6 Applications of Object Detection

Object detection is a powerful technology that drives innovation in many fields. Some key applications include:

- Autonomous Vehicles: Self-driving cars like those from Tesla and Waymo use object detection to "see" their surroundings. They can identify other cars, pedestrians, and road signs to navigate safely.
- Security and Surveillance: Smart cameras can automatically detect intruders or suspicious activities, and facial recognition systems can identify individuals for security purposes.
- Healthcare: In medical imaging, object detection helps doctors find tumors in X-rays and MRIs, leading to earlier and more accurate diagnoses.
- Retail: Stores like Amazon Go use object detection to create a cashier-less shopping experience by tracking which items you pick up. It's also used to monitor shelf stock in real-time.
- Robotics: Warehouse robots can identify and pick items for orders, while service robots can recognize and interact with people to provide assistance.

5.6 Natural Language Classification

Natural Language Classification (NLC) is a fundamental task in the field of Natural Language Processing (NLP). Its primary goal is to assign a piece of text to one or more predefined categories or labels. It's a form of supervised machine learning, which means we train a computer model on a large set of examples that have already been correctly labeled by humans. By analyzing these examples, the model learns the underlying patterns and can then make accurate predictions on new, unseen text.

NLC is crucial because it allows us to structure the vast amount of unstructured text data in the world. Emails, social media posts, articles, and customer reviews are all unstructured. NLC helps businesses and individuals to:

- Automate Processes: Instead of manually sorting thousands of documents, a model can do it instantly.
- Gain Insights at Scale: Analyze millions of customer reviews to understand public sentiment about a product in minutes, not months.
- Improve User Experience: Automatically route a support query to the correct department, leading to faster response times.

5.6.1 Step-by-Step Process

The journey from raw text to an accurate classification involves several critical stages.

Step 1: Data Collection and Preparation

- This is the foundation of any classification task. The quality of your model is directly dependent on the quality of your data.
- Gathering a Labeled Dataset: You need a substantial collection of text documents, where each document is already associated with its correct category.
- Ensuring Data Quality: The data should be clean and the labels accurate.
- Creating a Balanced Dataset: Ideally, you should have a roughly equal number of examples for each category. If you're building a positive/negative classifier with 95% positive examples and only 5% negative, the model might become lazy and just predict positive all the time.

Step 2: Text Preprocessing and Feature Engineering

Computers don't understand words and sentences; they only understand numbers. This step, also called Feature Engineering, involves cleaning the text and converting it into a numerical format that a machine learning model can process.

- Cleaning the Text:
    - Lowercasing: Converts all text to lowercase to treat words like "Apple" and "apple" as the same.
    - Removing Punctuation and Special Characters: Characters like !, ?, @, # are often removed as they can be noisy.
    - Removing Stop Words: These are very common words that usually add little meaning to the classification task, such as "a", "an", "the", "in", "is", "at".
- Tokenization: This is the process of breaking down a string of text into smaller pieces, or "tokens." Most commonly, this means splitting a sentence into a list of individual words.
    - Example: "The quick brown fox jumps." becomes ['the', 'quick', 'brown', 'fox', 'jumps'].
- Stemming and Lemmatization: These techniques reduce words to their root form to group related words together.
    - Stemming: A crude, rule-based process of chopping off the end of words. For example, "studies", "studying", and "studied" might all become "studi".
    - Lemmatization: A more intelligent process that uses a dictionary to reduce words to their actual root form (lemma). For example, "studies", "studying", and "studied" would all become "study". Lemmatization is often preferred because it results in real words.
- Vectorization: Turning Text into Numbers: This is the most crucial part of preprocessing. There are several popular methods like Bag-of-Words, TF-IDF, and Word Embeddings.

Step 3: Model Training

Once the text is converted into numerical vectors, it's fed into a machine learning algorithm. Before training, the dataset is typically split into three parts:

- Training Set (e.g., 70% of the data): The model learns the patterns from this data.

- Validation Set (e.g., 15%): Used to tune the model's parameters during training and prevent it from "overfitting" (memorizing the training data instead of learning general patterns).
- Test Set (e.g., 15%): This data is kept completely separate and is used only once at the very end to evaluate the final performance of the trained model on unseen data.

Step 4: Model Evaluation

After training, you must check how well the model performs. This is done using the test set. Key metrics include Accuracy, Precision, Recall and F1-Score.

Step 5: Deployment and Inference

Once you have a trained and evaluated model that meets your performance criteria, it can be deployed. This means integrating it into an application where it can take new, unseen text as input and output a predicted category in real-time. This process is called inference.

5.6.2 Real-World Examples

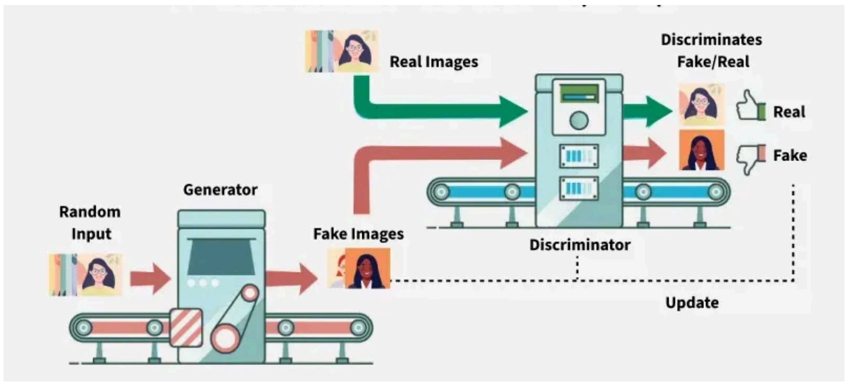
- Email Spam Detection: Your email service (like Gmail) automatically classifies incoming emails as 'Inbox' or 'Spam'. It has been trained on billions of emails to recognize patterns typical of spam.
- Sentiment Analysis: Companies use this to understand customer opinions. They analyze social media posts, reviews, and survey responses to classify them as 'positive', 'negative', or 'neutral'. This helps them gauge public reaction to a new product.
- Topic Categorization: News websites automatically categorize articles into sections like 'Sports', 'Technology', 'Politics', or 'Business'. This is done by a model that has learned to identify the key words and phrases for each topic.
- Language Detection: Tools like Google Translate first classify which language the input text is written in before attempting to translate it.

5.7 Generative Adversarial Networks

Generative Adversarial Networks (GAN) help machines to create new, realistic data by learning from existing examples. It was introduced by Ian Goodfellow and his team in 2014 and they have transformed how computers generate images, videos, music and more. Unlike traditional models that only recognize or classify data, they take a creative way by generating entirely new content that closely resembles real-world data. This ability helped various fields such as art, gaming, healthcare and data science. In this article, we will see more about GANs and its core concepts.

5.7.1 Architecture of GAN

GAN consist of two main models that work together to create realistic synthetic data which are as follows:



5.7.1.1 Generator Model

The generator is a deep neural network that takes random noise as input to generate realistic data samples like images or text. It learns the underlying data patterns by adjusting its internal parameters during training through backpropagation. Its objective is to produce samples that the discriminator classifies as real.

Generator Loss Function: The generator tries to minimize this loss:

$$J_G = -\frac{1}{m}\sum_{i=1}^{m} logD(G(z_i))$$

where

- J_G measures how well the generator is fooling the discriminator.
- $G(z_i)$ is the generated sample from random noise
- $D(G(z_i))$ is the discriminator's estimated probability that the generated sample is real.

The generator aims to maximize $D(G(z_i))$ meaning it wants the discriminator to classify its fake data as real (probability close to 1).

5.7.1.2. Discriminator Model

The discriminator acts as a binary classifier and helps in distinguishing between real and generated data. It learns to improve its classification ability through training, refining its parameters to detect fake samples more accurately. When dealing with image data, the discriminator uses convolutional layers or other relevant architectures which help to extract features and enhance the model's ability.

Discriminator Loss Function: The discriminator tries to minimize this loss:

$$J_D = -\frac{1}{m}\sum_{i=1}^{m} logD(x_i) - \frac{1}{m}\sum_{i=1}^{m} log(1 - D(G(z_i)))$$

- J_D measures how well the discriminator classifies real and fake samples.
- x_i is a real data sample.
- $G(z_i)$ is a fake sample from the generator.
- $D(x_i)$ is the discriminator's probability that x_i is real.
- $D(G(z_i))$ is the discriminator's probability that the fake sample is real.

The discriminator wants to correctly classify real data as real (maximize log $D(x_i)$) and fake data as fake (maximize log(1 - $D(G(z_i))$))

5.7.1.3 MinMax Loss

GANs are trained using a MinMax Loss between the generator and discriminator:

$$min_G\, max_D(G, D) = [E_{x \sim Pdata}[log\, D(x) + E_{z \sim Pz(z)}[log(1 - D(G(z)))]$$

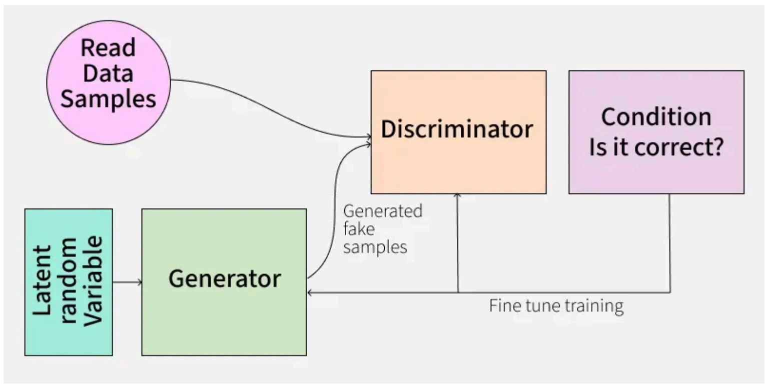
where,

- G is generator network and D is the discriminator network
- $Pdata(x)$ = true data distribution
- $Pz(z)$ = distribution of random noise (usually normal or uniform)
- $D(x)$ = discriminator's estimate of real data
- $D(G(z))$ = discriminator's estimate of generated data

The generator tries to minimize this loss (to fool the discriminator) and the discriminator tries to maximize it (to detect fakes accurately).

5.7.2 Working of GAN

GAN trains by having two networks the Generator (G) and the Discriminator (D) compete and improve together. Here's the step-by-step process



5.7.2.1. Generator's First Move

The generator starts with a random noise vector like random numbers. It uses this noise as a starting point to create a fake data sample such as a generated image. The generator's internal layers transform this noise into something that looks like real data.

5.7.2.2. Discriminator's Turn

The discriminator receives two types of data:

- Real samples from the actual training dataset.
- Fake samples created by the generator.

D's job is to analyze each input and find whether it's real data or something G cooked up. It outputs a probability score between 0 and 1. A score of 1 shows the data is likely real and 0 suggests it's fake.

5.7.2.3. Adversarial Learning

- If the discriminator correctly classifies real and fake data it gets better at its job.
- If the generator fools the discriminator by creating realistic fake data, it receives a positive update and the discriminator is penalized for making a wrong decision.

5.7.2.4. Generator's Improvement

- Each time the discriminator mistakes fake data for real, the generator learns from this success.

- Through many iterations, the generator improves and creates more convincing fake samples.

5.7.2.5. Discriminator's Adaptation

- The discriminator also learns continuously by updating itself to better spot fake data.
- This constant back-and-forth makes both networks stronger over time.

5.7.2.6. Training Progression

- As training continues, the generator becomes highly proficient at producing realistic data.
- Eventually the discriminator struggles to distinguish real from fake shows that the GAN has reached a well-trained state.
- At this point, the generator can produce high-quality synthetic data that can be used for different applications.

5.7.3 Advantages

- Synthetic Data Generation: GANs produce new, synthetic data resembling real data distributions which is useful for augmentation, anomaly detection and creative tasks.
- High-Quality Results: They can generate photorealistic images, videos, music and other media with high quality.
- Unsupervised Learning: They don't require labeled data helps in making them effective in scenarios where labeling is expensive or difficult.
- Versatility: They can be applied across many tasks including image synthesis, text-to-image generation, style transfer, anomaly detection and more.

GANs are evolving and shaping the future of artificial intelligence. As the technology improves, we can expect even more innovative applications that will change how we create, work and interact with digital content.

5.8 Boltzmann Machines

A Boltzmann Machine (BM) is a type of unsupervised deep learning model. They are distinct from other neural network architectures such as Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs). Boltzmann Machines have several unique features that set them apart:

- Undirected Network: BMs are undirected, or bidirectional, networks. This means that every node is connected to every other node, unlike typical models that have directed connections. This structure is effective for representing complex dependencies between nodes.
- Unsupervised Model: They are well-suited for modeling data distributions in an unsupervised way.
- Stochastic Nature: BMs are stochastic, or generative, models, which means they are not deterministic. They work by exploring different possible states to learn the underlying distribution of the data.

The nodes within a Boltzmann Machine are divided into two main categories:

- Visible Nodes (v): These nodes represent the observable or measurable features of a system, like the pixels in an image or specific features in a dataset.
- Hidden Nodes (h): These nodes represent unobservable factors or latent variables that influence the system's behavior.

As seen in the provided figure, all nodes (both visible and hidden) are interconnected.

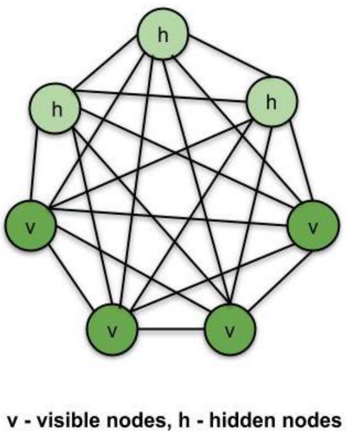


Figure: Boltzmann Machine

5.8.1 Energy-Based Models

A Boltzmann Machine functions as an energy-based model, using the Boltzmann distribution to sample from the system. The probability of the system being in a specific state is related to the energy of that state. The formula for the Boltzmann distribution is given by:

$$P_i = \frac{e^{-\frac{E_i}{kT}}}{Z}$$

P_i - probability of system being in state i
E_i - Energy of system in state i
T - Temperature of the system
k - Boltzmann constant
Z - partition function, which normalizes the probabilities over all possible states of the system

A key principle here is that lower-energy states are more probable. This is because as the energy (E_i) decreases, the exponent in the formula becomes less negative, making the resulting probability larger.

The training of a Boltzmann Machine focuses on adjusting the weights of the connections between all nodes. The primary goal is to minimize the system's energy so it reaches an equilibrium state. In this state, the model's learned distribution closely matches the distribution of the input data. Because direct sampling from the Boltzmann distribution is computationally very expensive, the training process uses approximation methods like contrastive divergence to estimate the gradient of the log-likelihood function. Ultimately, the objective of training is to adjust the system's weights so the machine can generate new samples that are similar to the original training data, making it a powerful generative model.

5.9 Restricted Boltzmann Machines (RBMs)

In a full Boltzmann Machine, every node is connected to every other node, resulting in a complex and computationally expensive structure as the number of nodes increases. Restricted Boltzmann Machines address this issue by simplifying the network architecture. In RBMs, connections are only allowed between visible and hidden nodes—visible-to-visible and hidden-to-hidden connections are not permitted. This structure makes learning and inference more efficient and manageable.

The energy function E(v, h) captures the interaction between the visible units (v_i), the hidden units (h_j), the associated weights (w_{ij}), and their biases (a_i for visible units, b_j for hidden units). The energy function is defined as:

$$E(v, h) = -\Sigma\, a_i v_i - \Sigma\, b_j h_j - \Sigma\Sigma\, v_i w_{i,j} h_j$$
a, b - biases in the system - constants

v_i, h_j - visible node, hidden node
P(v, h) is the probability of being in a certain state, where $P(v, h) = e^{(-E(v, h))/Z}$
Z - sum if values for all possible states

The probability of the network being in a specific state P(v, h) is determined by this energy function. A lower energy value indicates a more favorable or stable state for the network.

5.9.1 Training RBMs with Contrastive Divergence

RBMs are trained using an algorithm called Contrastive Divergence (CD), which adjusts the weights between the nodes. The goal of training is to update the weights so that the model can effectively reconstruct the original input data.

The CD process works as follows:

- Initialization: The weights of the network are initialized with random values.
- Forward Pass: An input vector (e.g., user movie ratings) is fed into the visible layer. The RBM then calculates the activation states of the hidden nodes based on this input.
- Backward Pass (Reconstruction): The activations of the hidden nodes are used to reconstruct the visible layer, creating a synthetic version of the original input. Initially, this reconstruction will differ significantly from the original input because the weights are random.
- Weight Update: The weights are updated to minimize the difference between the original input data and the reconstructed data. The formula for updating the weights is guided by the gradient of the log-likelihood of the data:

$d/dw_{ij}(\log(P(v^0))) = \langle v_i^0 * h_j^0 \rangle - \langle v_i^\infty * h_j^\infty \rangle$
v - visible state, h- hidden state
$\langle v_i^0 * h_j^0 \rangle$ - initial state of the system
$\langle v_i^\infty * h_j^\infty \rangle$ - final state of the system
$P(v^0)$ - probability that the system is in state v^0
w_{ij} - weights of the system

This iterative, back-and-forth process between the visible and hidden layers is a form of Gibbs Sampling. However, running Gibbs Sampling until full convergence is computationally expensive. Contrastive Divergence, leveraging a concept known as Hinton's shortcut, takes only a few steps. This approach approximates the gradient descent, allowing the system to reach a low-energy, stable state much faster while still producing high-quality results.

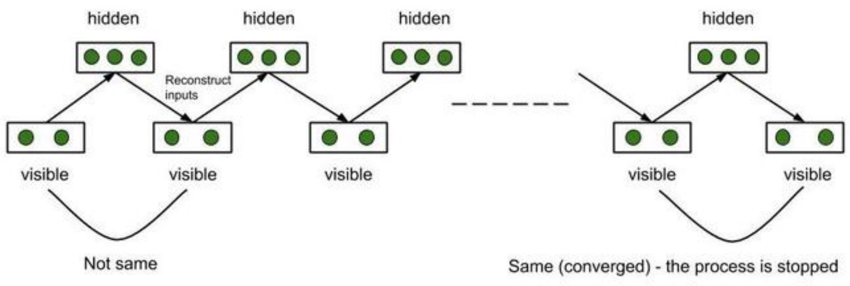


Figure: Gibbs Sampling

5.9.2 Working of RBM – Illustrative Example

RBMs are highly effective for collaborative filtering, especially with sparse datasets where users have rated only a few items. They work by discovering latent features (like genre or actor preferences) from user ratings. Consider a user, Mary, who has rated four out of six available movies. Her preferences are binary: 1 for liked, 0 for disliked.

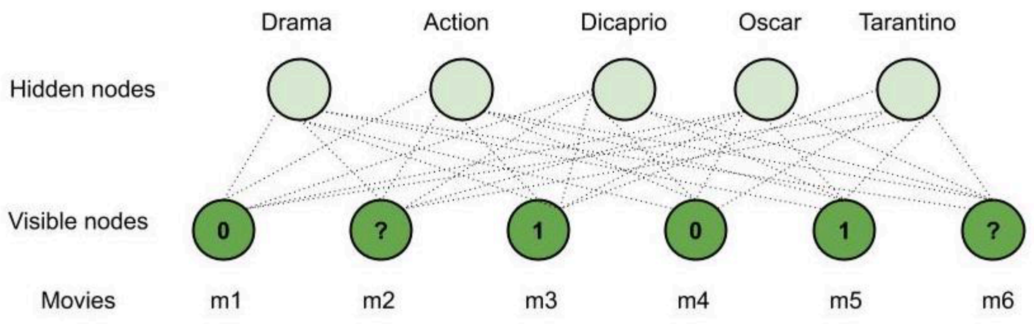


Figure: Working of RBM

Consider – Mary watches four movies out of the six available movies and rates four of them. Say, she watched m1, m3, m4 and m5 and likes m3, m5 (rated 1) and dislikes the other two, that is m1, m4 (rated 0) whereas the other two movies – m2, m6 are unrated. Now, using our RBM, we will recommend one of these movies for her to watch next. Say –

- m3, m5 are of the ‘Drama’ genre.
- m1, m4 are of the ‘Action’ genre.
- ‘Dicaprio’ played a role in m5.
- m3, m5 have won ‘Oscar.’
- ‘Tarantino’ directed m4.
- m2 is of the ‘Action’ genre.

- m6 is of both the genres ‘Action’ and ‘Drama’, ‘Dicaprio’ acted in it and it has won an ‘Oscar’.

We have the following observations

- Mary likes m3, m5 and they are of the genre 'Drama', she probably likes 'Drama' movies.
- Mary dislikes m1, m4 and they are of the action genre, she probably dislikes 'Action' movies.
- Mary likes m3, m5 and they have won an 'Oscar', she probably likes an 'Oscar' movie.
- Since 'Dicaprio' acted in m5 and Mary likes it, she will probably like a movie in which 'Dicaprio' acted.
- Mary does not like m4 which is directed by Tarantino, she probably dislikes any movie directed by 'Tarantino'.

Therefore, based on the observations and the details of m2, m6; our RBM recommends m6 to Mary (‘Drama’, ‘Dicaprio’ and ‘Oscar’ matches both Mary’s interests and m6). This is how an RBM works and hence is used in recommender systems. Thus, RBMs are used to build Recommender Systems.

5.10 Deep Belief Networks

Deep Belief Networks (DBNs) are a type of neural network that are made up of multiple layers of "restricted Boltzmann machines" (RBMs). Think of them as a stack of intelligent filters that learn to understand complex information step by step.

DBNs are particularly good at unsupervised learning. This means they can learn from data that hasn't been labeled or categorized by humans. Imagine showing a computer thousands of pictures of animals without telling it which one is a cat or a dog. A DBN can start to figure out patterns and similarities on its own. They are excellent for tasks like:

- Image Recognition: Identifying objects, faces, or scenes in pictures.
- Speech Recognition: Understanding spoken language.
- Feature Learning: Discovering important characteristics in data.

5.10.1 Building a Deep Belief Network (DBN)

A Deep Belief Network is essentially a stack of these RBMs, one on top of the other. Here's how it works:

- First RBM: The first RBM takes your raw input data (e.g., pixels of an image) as its visible layer. It learns to find basic features in this data (like edges or simple shapes).
- Second RBM: Once the first RBM is trained, its hidden layer (which now represents the basic features) becomes the visible layer for the second RBM. This second RBM then learns more complex features by combining the basic features found by the first RBM (e.g., combining edges to form a nose or an eye).
- Stacking Up: You can keep adding more RBMs. Each new RBM's visible layer is the hidden layer of the R one below it. This allows the DBN to learn increasingly abstract and complex features at each successive layer.

Imagine training a DBN to recognize animals:

- Layer 1 (RBM 1): Takes raw pixels of animal images. It might learn to detect simple things like horizontal lines, vertical lines, curves, and blobs.
- Layer 2 (RBM 2): Takes the "activations" from Layer 1 (the detected lines and curves). It might combine these to recognize more complex shapes like eyes, ears, tails, or paws.
- Layer 3 (RBM 3): Takes the "activations" from Layer 2 (the detected eyes, ears, etc.). It might combine these features to recognize a whole face or body shape, leading to the identification of an actual animal like "cat" or "dog."

5.10.2 Learning of DBN

DBNs learn in two main phases:

- Greedy Layer-wise Pre-training
- Fine-tuning

5.10.2.1 Greedy Layer-wise Pre-training (Unsupervised):

- Each RBM in the stack is trained one at a time, from bottom to top.
- The first RBM is trained on the raw input data.
- Once trained, its hidden layer's output is used as the input to train the second RBM.

- This process continues until all RBMs in the stack are trained.
- The goal here is for each RBM to learn useful features from its input without needing any labels. It's like building up a general understanding of the data.

5.10.2.2 Fine-tuning (Supervised):

- After pre-training, an additional "output layer" (like a simple logistic regression classifier) is usually added on top of the last RBM's hidden layer.
- Now, the entire DBN (all the stacked RBMs plus the output layer) is trained together using labeled data (e.g., "this image is a cat," "this image is a dog").
- This "fine-tuning" step uses a supervised learning algorithm (like backpropagation) to slightly adjust all the weights in the entire network. This helps the network to perform specific tasks like classification very accurately.

The pre-training step is very powerful because it helps the DBN overcome a common problem in deep networks called the "vanishing gradient problem." In simple terms, without pre-training, deep networks can be very hard to train effectively from scratch because the learning signals get weaker as they pass through many layers. Pre-training gives the network a good starting point, making fine-tuning much easier and more effective.

5.10.3 Example: Recognizing Handwritten Digits

Let's say you want to train a DBN to recognize handwritten digits (0-9) from images.

- Input: Each image of a handwritten digit (e.g., 28x28 pixels) would be fed into the visible layer of the first RBM.
- RBM 1 (Feature Learner 1): This RBM learns to detect basic strokes, curves, and corners present in the digits. Its hidden layer activates when it sees these simple features.
- RBM 2 (Feature Learner 2): The activations from RBM 1 (the basic strokes) become the input for RBM 2. This RBM learns to combine these strokes into more complex patterns that form parts of digits, like the top loop of an '8' or the vertical bar of a '1'.
- RBM 3 (Feature Learner 3): The activations from RBM 2 (the complex patterns) become the input for RBM 3. This RBM might learn to recognize whole digit structures, like the full shape of a '0' or a '7'.
- Output Layer (Classifier): After these layers are pre-trained, a final output layer is added. Now, with labeled examples (images of '0's, '1's, etc.), the entire network is fine-tuned to accurately classify which digit is in the image.

5.10.4 Key Benefits of DBNs

- Effective Unsupervised Learning: Can learn useful representations from unlabeled data. This is crucial as labeled data can be expensive and time-consuming to obtain.
- Good for Feature Extraction: Automatically discovers hierarchical features in data, from simple to complex.
- Initialization for Deep Networks: The pre-training phase provides excellent starting weights for deep neural networks, helping them train more effectively and avoid common issues like vanishing gradients.

While more modern architectures like Convolutional Neural Networks (CNNs) have surpassed DBNs in many computer vision tasks, DBNs were pioneering in demonstrating the power of deep learning and unsupervised pre-training, laying important groundwork for the deep learning revolution.

5.11 Transformer Model

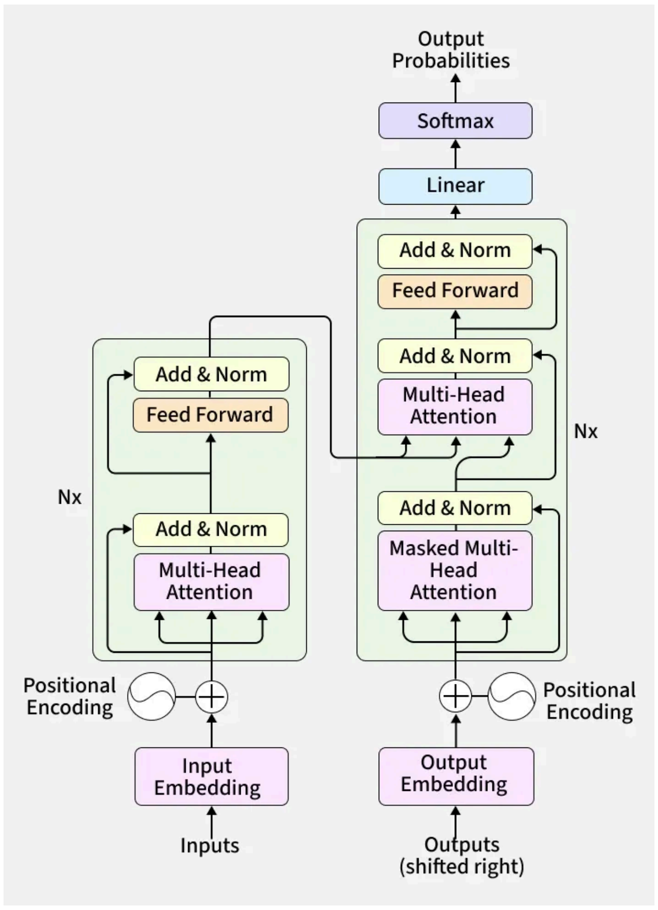
The Transformer is a powerful neural network architecture that has revolutionized Natural Language Processing (NLP). Unlike older models that process words one by one in order, the Transformer can process all words in a sentence at the same time, making it much faster and more effective at understanding context.

5.11.1 An Encoder-Decoder Structure

At its core, the Transformer has two main parts: an Encoder and a Decoder.

- The Encoder: Its job is to read the input sentence (e.g., an English sentence) and understand its meaning and context. It creates a rich numerical representation of the sentence.
- The Decoder: Its job is to take the encoder's representation and generate the output sentence (e.g., the French translation).

Imagine a human translator. They first read and understand the entire English sentence (the encoder's job) before starting to write the translation in French (the decoder's job). The Transformer works similarly. In practice, a Transformer uses a stack of several identical encoders and decoders on top of each other to refine the representations. The architecture of the Transformer model is shown in the figure below. Let's break down the key components that make the Transformer work.



5.11.1.1 Input Processing: Embeddings and Positional Encodings

Computers don't understand words, they understand numbers. So, the first step is to turn each word into a vector of numbers. This is called word embedding. However, the Transformer processes all words at once, so it has no natural sense of word order. To fix this, we add another vector called Positional Encoding to the embedding of each word. This vector gives the model information about the position of each word in the sentence (e.g., is it the first word, second word, etc.).

5.11.1.2 The Attention Mechanism: The Secret Sauce

This is the most important idea in the Transformer. The attention mechanism allows the model to weigh the importance of different words in the input sentence when processing a specific word. For example, in the sentence "The cat sat on the mat, and it was asleep," when the model processes the word "it," attention helps it realize that "it" refers to the "cat," not the "mat." To do this, the Transformer uses Scaled Dot-Product Attention. For every word, we create three vectors:

- Query (Q): Represents the current word you are focusing on. It's like asking a question: "What should I pay attention to?"
- Key (K): Represents all the words in the sentence. It's like a label or a keyword for a word.
- Value (V): Also represents all the words in the sentence. It contains the actual meaning or content of the word.

The model calculates an attention score by taking the dot product of the Query of the current word with the Key of every other word. These scores are then scaled down, passed through a softmax function to turn them into probabilities, and finally used to create a weighted sum of all the Value vectors. The core equation for attention is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

- QK^T: This calculates the similarity score between the Query and every Key.
- $\sqrt{d_k}$: This is a scaling factor to keep the numbers stable during training. d_k is the dimension of the Key vector.
- softmax: This function converts the scores into weights that all add up to 1. A word with a high score gets a high weight.
- V: The final output is the sum of all Value vectors, weighted by their attention scores.

5.11.1.3 Multi-Head Attention

Instead of just calculating attention once, the Transformer does it multiple times in parallel. This is called Multi-Head Attention. Each "head" learns to pay attention to different things. For example, one head might focus on grammatical relationships, while another focuses on semantic meaning. This allows the model to capture a richer understanding of the language. The outputs from all the heads are then combined to produce the final result.

5.11.1.4 The Encoder Block in Detail

A single encoder block has two main layers:

- Multi-Head Attention Layer: This is the self-attention mechanism described above, where the sentence looks at itself to understand context.

- Feed-Forward Neural Network: A simple, fully connected neural network that processes the output from the attention layer. It is applied to each word's representation independently.

Each of these two layers also has a residual connection (the input to the layer is added to its output) followed by layer normalization. This helps the model train more effectively.

5.11.1.5 The Decoder Block in Detail

A single decoder block is similar to an encoder block but has three main layers:

- Masked Multi-Head Attention Layer: This is a self-attention layer for the output sentence. It is "masked" to prevent the model from cheating. When predicting the fourth word, the model should only be able to see the first three words it has already generated, not the words that come after.
- Encoder-Decoder Attention Layer: This is where the decoder pays attention to the output of the encoder. The Queries come from the decoder, while the Keys and Values come from the encoder's output. This is the crucial step where the decoder looks at the input sentence to generate the correct output word.
- Feed-Forward Neural Network: Same as in the encoder.

These layers also have residual connections and layer normalization.

5.11.2 The Final Output

After the input passes through the final decoder block, the resulting vector is fed into two final layers:

- A Linear Layer: A fully connected neural network that projects the vector into a much larger vector, with a size equal to the number of words in the vocabulary.
- A Softmax Layer: This turns the scores from the linear layer into probabilities. The word with the highest probability is chosen as the next word in the output sentence.

This process is repeated one word at a time until the decoder generates a special "end-of-sentence" token.