UNIT I DEEP NETWORKS BASICS

SYLLABUS:

Linear Algebra: Scalars -- Vectors -- Matrices and tensors; Probability Distributions -- Gradient-based Optimization – Machine Learning Basics: Capacity -- Overfitting and underfitting -- Hyperparameters and validation sets -- Estimators -- Bias and variance -- Stochastic gradient descent -- Challenges motivating deep learning; Deep Networks: Deep feedforward networks; Regularization -- Optimization.

PART A

1. What is Deep Learning?

- It is the branch of machine learning that is based on artificial neural network architecture.
- Deep learning algorithms are based on building a network of connected computational elements.
- The fundamental unit of such networks is a small bundle of computation called an artificial neuron, often referred to simply as a neuron.

2. Differentiate between machine learning and Deep learning

Machine Learning	Deep Learning
Apply statistical algorithms to learn the hidden patterns and relationships in the dataset.	Uses artificial neural network architecture to learn the hidden patterns and relationships in the dataset.
Can work on the smaller amount of dataset	Requires the larger volume of dataset compared to machine learning
Better for the low-label task.	Better for complex task like image processing, natural language processing, etc.
Takes less time to train the model.	Takes more time to train the model.

3. List the Advantages of Deep Learning.

i. **High accuracy:** Deep Learning algorithms can achieve state-of-the-art performance in various tasks, such as image recognition and natural language processing.

ii. **Automated feature engineering:** Deep Learning algorithms can automatically discover and learn relevant features from data without the need for manual feature engineering.

iii. **Scalability:** Deep Learning models can scale to handle large and complex datasets, and can learn from massive amounts of data.

iv. **Flexibility:** Deep Learning models can be applied to a wide range of tasks and can handle various types of data, such as images, text, and speech.

v. **Continual improvement:** Deep Learning models can continually improve their performance as more data becomes available.

4. List the Disadvantages of Deep Learning.

i. **High computational requirements:** Deep Learning AI models require large amounts of data and computational resources to train and optimize.

ii. **Requires large amounts of labelled data**: Deep Learning models often require a large amount of labelled data for training, which can be expensive and time- consuming to acquire.

iii. **Interpretability:** Deep Learning models can be challenging to interpret, making it difficult to understand how they make decisions.

iv. **Over fitting:** Deep Learning models can sometimes overfit to the training data, resulting in poor performance on new and unseen data.

v. **Black-box nature:** Deep Learning models are often treated as black boxes, making it difficult to understand how they work and how they arrived at their predictions.

5. List the Applications or Uses of Deep Learning.

- **Automatic speech recognition**

Large-scale automatic speech recognition is the first and most convincing successful case of deep learning.

- **Image recognition**

Deep learning-based image recognition has become "superhuman", producing more accurate results than human contestants.

- **Visual art processing**

Identifying the style period of a given painting

Capturing the style of a given artwork

Applying it in a visually pleasing manner to an arbitrary photograph or video

- **Natural language processing**

Natural language processing (NLP) is a subfield of computer science and artificial intelligence (AI) that uses machine learning to enable computers to understand and communicate with human language.

6. Differentiate scalar and vector.

Scalars	**Vector**
• A scalar quantity is defined as the physical quantity with only magnitude and no direction.	• A vector quantity is defined as the physical quantity that has both directions as well as magnitude.
• A scalar is a single real number that is used to measure magnitude (size).	• A vector with a value of magnitude equal to one is called a unit vector and is represented by a lowercase alphabet with a "hat" circumflex, i.e. "û".
• In Data Science, scalars are employed to encapsulate statistical metrics like mean, variance, or correlation coefficients.	• Vectors, within the context of Data Science, represent ordered collections of numerical values endowed with both magnitude and directionality.
• In Python can represent a Scalar like: ***#Scalars can be represented simply as numerical variables*** *scalar = 8.4* *scalar* ***Output:*** *8.4*	• *In Python can represent a Vector like:* *import numpy as np* ***# Vectors can be represented as one-dimensional arrays*** *vector = np.array([2, -3, 1.5])* *vector* ***Output:*** *array([2. , -3. , 1.5])*

7. List out the special kind of matrices.

- Diagonal Matrix- **Diagonal** matrices consist mostly of zeros and have non-zero entries only along the main diagonal.
- Symmetric Matrix- A **symmetric** matrix is any matrix that is equal to its own transpose: $A = A^T$.
- Orthogonal Matrix- An **orthogonal matrix** is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

A Square matrix 'A' is orthogonal if

$$A^T = A^{-1}$$

(OR)

$$AA^T = A^TA = I \text{, where}$$

- A^T = Transpose of A
- A^{-1} = Inverse of A
- I = Identity matrix of same order as 'A'

8. What is Probability theory?

- Probability theory is a mathematical framework for representing uncertain statements.
- It provides a means of quantifying uncertainty and axioms for deriving new uncertain statements.
- In artificial intelligence applications, probability theory is used in two major ways.

9. What is Random Variables?

- A **random variable** is a variable that can take on different values randomly.
- Random variables may be discrete or continuous.
  - A discrete random variable is one that has a finite or countable infinite number of states.
  - A continuous random variable is associated with a real value.

10. What is probability distributions?

- A probability distribution is a statistical function that describes all the possible values and probabilities for a random variable within a given range.

- This range will be bound by the minimum and maximum possible values, but where the possible value would be plotted on the probability distribution will be determined by a number of factors.

11. List the Types of Probability Distribution.

- Discrete Probability Distributions
- Continuous Probability Distributions

12. What are Expectation, Variance and Covariance?

- **Expectation**
  - The **expectation** or **expected value** of some function f(x) with respect to a probability distribution P (x) is the average or mean value that f takes on when x is drawn from P.

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x),$$

- **Variance**
  - The **variance** gives a measure of how much the values of a function of a random variable x vary for different values of x from its probability distribution:

$$\mathrm{Var}(f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right].$$

- **Covariance**
  - The **covariance** defines how much two values are linearly related to each other, as well as the scale of these variables:

$$\mathrm{Cov}(f(x), g(y)) = \mathbb{E}\left[(f(x) - \mathbb{E}\left[f(x)\right])\left(g(y) - \mathbb{E}\left[g(y)\right]\right)\right].$$

13. List some of the Common Probability Distributions.

- Bernoulli Distribution
- Multinoulli Distribution
- Gaussian Distribution
- Exponential and Laplace Distributions
- The Dirac Distribution and Empirical Distribution

14. State the Bayes rule.

Bayes' rule is derived from the definition of conditional Probability

Formula For Bayes' Theorem

$$P(A|B) = \frac{P(A \bigcap B)}{P(B)} = \frac{P(A) \cdot P(B|A)}{P(B)}$$

where:

$P(A) =$ The probability of A occurring

$P(B) =$ The probability of B occurring

$P(A|B) =$ The probability of A given B

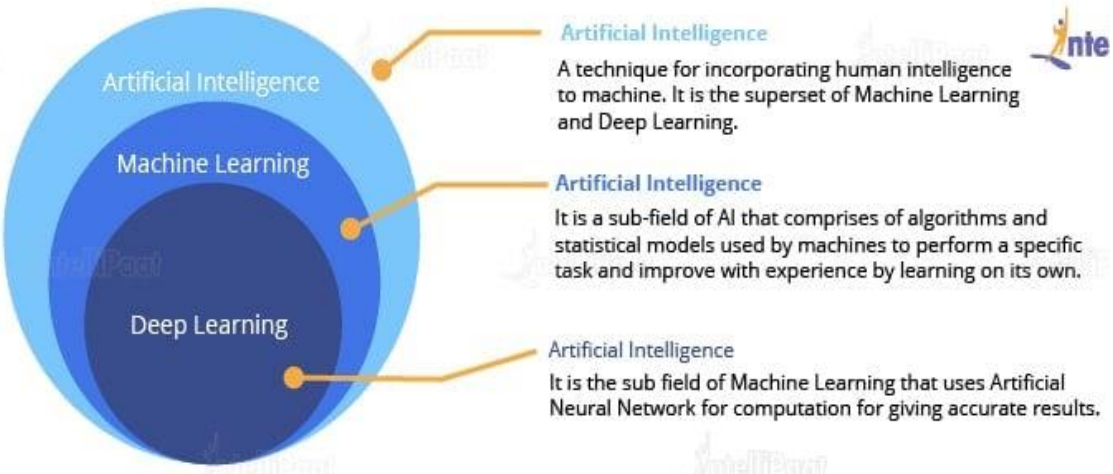
$P(B|A) =$ The probability of B given A

$P\left(A \bigcap B\right)) =$ The probability of both A and B occurri

15. List out some supervised learning algorithms.

- Supervised machine learning is the machine learning technique in which the neural network learns to make predictions or classify data based on the labelled datasets.
- Some of the supervised learning algorithms
  - Regression
  - Classification
    - Logistic Regression
    - K-Nearest Neighbors
    - Support Vector Machines
    - Kernel SVM
    - Naïve Bayes
    - Decision Tree Classification
    - Random Forest Classification

16. Give Venn diagram for Deep Learning.



17. Define Gradient Descent and how it works?

- Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that that aims to find the optimal parameters—

weights and biases— by minimizing the cost function of a neural network model during training.

- It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.

18. What are the types of Gradient Descent?

- Batch Gradient Descent
- Stochastic gradient descent
- MiniBatch Gradient Descent

19. What is Estimator?

- An estimator in statistics is a rule or a formula that helps us to estimate the value of an unknown parameter in a population using sample data.
- An estimator is a function of the sample data that provides an estimate of a population parameter.

20. List the Properties of Good Estimators

Properties of Good Estimators

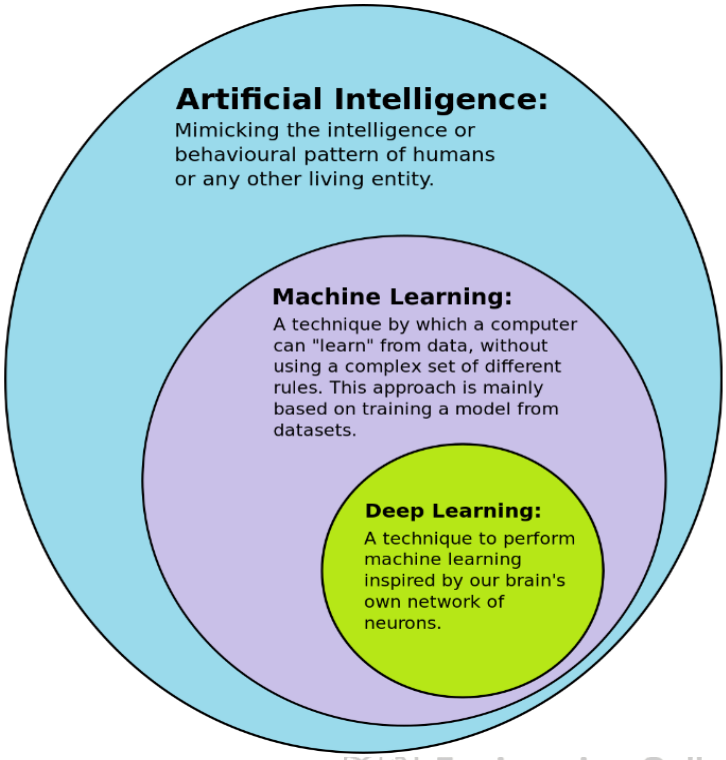
- **Unbiasedness**
  - An estimator is said to be unbiased if its expected value is equal to the true value of the parameter being estimated.
  - An estimator that systematically overestimates or underestimates the parameter is considered biased.
- **Consistency**
  - A consistent estimator is one where the estimates become closer to the true parameter value as the sample size increases.
- **Efficiency**
  - Efficiency refers to the variance of the estimator.
  - An efficient estimator provides more precision and is less spread out around the true parameter value.
- **Sufficiency**
  - A sufficient estimator captures all the information in the sample that is needed to estimate the parameter.

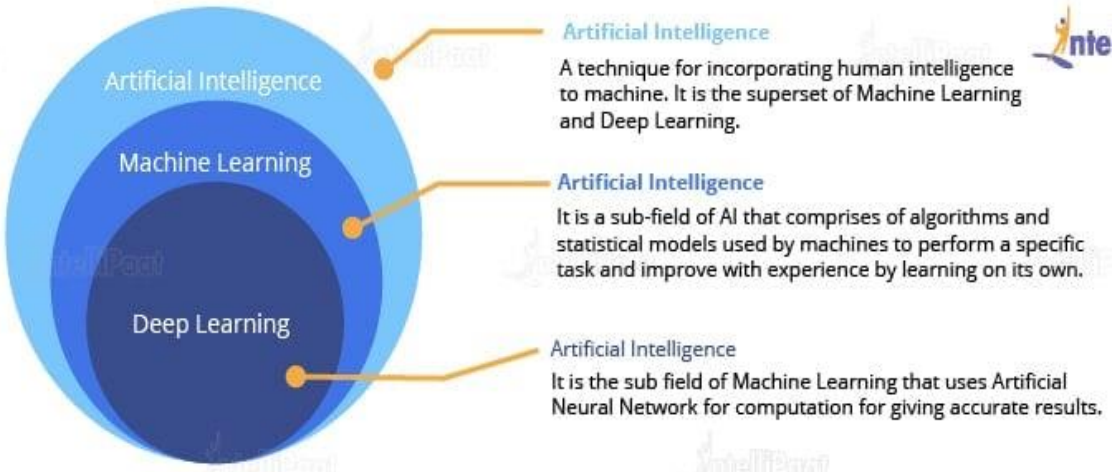
21. Write an algorithm for k -fold cross validation.

Define `KFoldXV`($\mathbb{D}, A, L, k$):
Require: $\mathbb{D}$, the given dataset, with elements $\boldsymbol{z}^{(i)}$
Require: A, the learning algorithm, seen as a function that takes a dataset as input and outputs a learned function
Require: L, the loss function, seen as a function from a learned function f and an example $\boldsymbol{z}^{(i)} \in \mathbb{D}$ to a scalar $\in \mathbb{R}$
Require: k, the number of folds
Split $\mathbb{D}$ into k mutually exclusive subsets $\mathbb{D}_i$, whose union is $\mathbb{D}$.
for i from 1 to k **do**
  $f_i = A(\mathbb{D} \backslash \mathbb{D}_i)$ Train A on dataset without D_i
  for $\boldsymbol{z}^{(j)}$ in $\mathbb{D}_i$ **do**
    $e_j = L(f_i, \boldsymbol{z}^{(j)})$ Determine errors for samples in D_i
  end for
end for
Return $\boldsymbol{e}$ Return vector of errors $\boldsymbol{e}$ for samples in D

PART B

1. **Discuss in detail about Deep Learning? Differentiate between AI, ML and DL.**





- **Artificial intelligence**
  - Artificial intelligence, or AI, is technology that enables computers and machines to simulate human intelligence and problem-solving capabilities.
  - Artificial intelligence encompasses machine learning and deep learning.

- Digital assistants, GPS guidance, autonomous vehicles, and generative AI tools (like Open AI's Chat GPT) are just a few examples of AI in the daily news and our daily lives.

- **Types of artificial intelligence: weak AI vs. strong AI**
  - **Weak AI**—also known as *narrow AI* or *artificial narrow intelligence* (ANI)—is AI trained and focused to perform specific tasks. It enables some very robust applications, such as Apple's Siri, Amazon's Alexa, IBM watsonx™, and self-driving vehicles.
  - **Strong AI** is made up of *artificial general intelligence* (AGI) and *artificial super intelligence* (ASI). AGI, or general AI, is a theoretical form of AI where a machine would have an intelligence equal to humans; ASI—also known as super intelligence—would surpass the intelligence and ability of the human brain.

➢ **Machine Learning**
  - Machine learning (ML) is a discipline of artificial intelligence (AI) that provides machines the ability to automatically learn from data and past experiences to identify patterns and make predictions with minimal human intervention.
  - It enables algorithms to uncover hidden patterns within datasets, allowing them to make predictions on new, similar data without explicit programming for each task.
  - **Example:** categorizing images, analyzing data, or predicting price fluctuations.

Supervised Machine Learning:
- Supervised machine learning is the machine learning technique in which the neural network learns to make predictions or classify data based on the labelled datasets.
- Here both input features along with the target variables are the input.
- The neural network learns to make predictions based on the cost or error that comes from the difference between the predicted and the actual target, this process is known as back propagation.
- Deep learning algorithms like Convolutional neural networks, Recurrent neural networks are used for many supervised tasks like image classifications and recognition, sentiment analysis, language translations, etc.

Unsupervised Machine Learning:

- Unsupervised machine learning is the machine learning technique in which the neural network learns to discover the patterns or to cluster the dataset based on unlabelled datasets.
- Here there are no target variables.
- The machine has to self-determined the hidden patterns or relationships within the datasets.
- Deep learning algorithms like auto encoders and generative models are used for unsupervised tasks like clustering, dimensionality reduction, and anomaly detection.

Reinforcement Machine Learning:

- Reinforcement Machine Learning is the machine learning technique in which an agent learns to make decisions in an environment to maximize a reward signal.
- The agent interacts with the environment by taking action and observing the resulting rewards.
- Deep learning can be used to learn policies, or a set of actions, that maximizes the cumulative reward over time.
- Deep reinforcement learning algorithms like Deep Q networks and Deep Deterministic Policy Gradient (DDPG) are used to reinforce tasks like robotics and game playing etc.

➢ **Deep Learning**

- It is the branch of machine learning that is based on artificial neural network architecture.
- Deep learning algorithms are based on building a network of connected computational elements.
- The fundamental unit of such networks is a small bundle of computation called an artificial neuron, often referred to simply as a neuron.
- An artificial neural network or ANN uses layers of interconnected nodes called neurons that work together to process and learn from the input data. Refer Figure 1.1.
- In a fully connected Deep neural network, there is an input layer and one or more hidden layers connected one after the other.
- Each neuron receives input from the previous layer neurons or the input layer.
- The output of one neuron becomes the input to other neurons in the next layer of the network, and this process continues until the final layer produces the output of the network.

- The layers of the neural network transform the input data through a series of nonlinear transformations, allowing the network to learn complex representations of the input data.

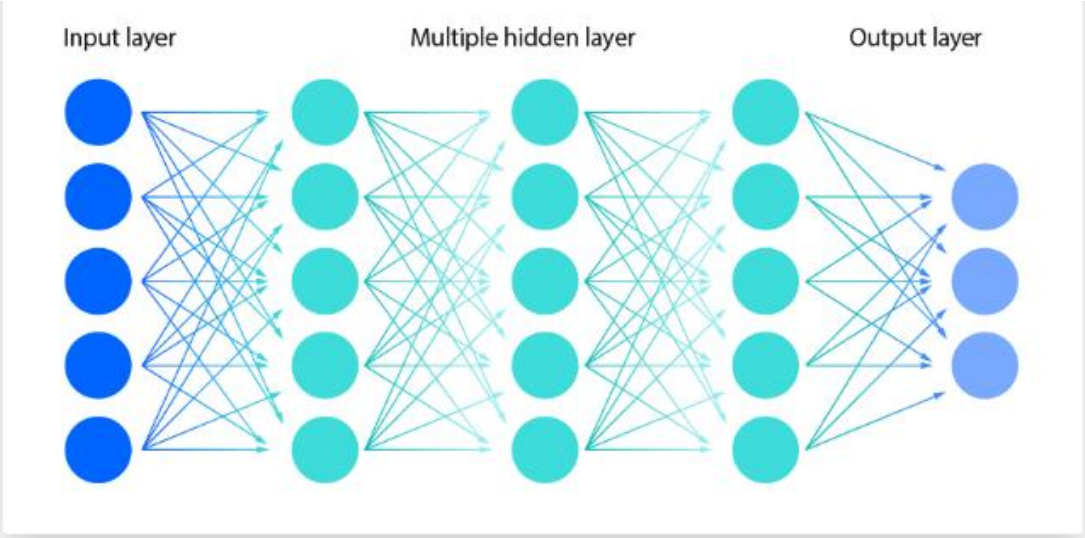


Figure 1.1 Deep Neural Network

- Deep learning AI can be used for supervised, unsupervised as well as reinforcement machine learning. it uses a variety of ways to process these.

Deep learning vs. machine learning

Machine Learning	**Deep Learning**
Apply statistical algorithms to learn the hidden patterns and relationships in the dataset.	Uses artificial neural network architecture to learn the hidden patterns and relationships in the dataset.
Can work on the smaller amount of dataset	Requires the larger volume of dataset compared to machine learning
Better for the low-label task.	Better for complex task like image processing, natural language processing, etc.
Takes less time to train the model.	Takes more time to train the model.

Machine Learning	Deep Learning
A model is created by relevant features which are manually extracted from images to detect an object in the image.	Relevant features are automatically extracted from images. It is an end-to-end learning process.
Less complex and easy to interpret the result.	More complex, it works like the black box interpretations of the result are not easy.
It can work on the CPU or requires less computing power as compared to deep learning.	It requires a high-performance computer with GPU.

Advantages of Deep Learning:

High accuracy: Deep Learning algorithms can achieve state-of-the-art performance in various tasks, such as image recognition and natural language processing.

Automated feature engineering: Deep Learning algorithms can automatically discover and learn relevant features from data without the need for manual feature engineering.

Scalability: Deep Learning models can scale to handle large and complex datasets, and can learn from massive amounts of data.

Flexibility: Deep Learning models can be applied to a wide range of tasks and can handle various types of data, such as images, text, and speech.

Continual improvement: Deep Learning models can continually improve their performance as more data becomes available.

Disadvantages of Deep Learning:

High computational requirements: Deep Learning AI models require large amounts of data and computational resources to train and optimize.

Requires large amounts of labelled data: Deep Learning models often require a large amount of labelled data for training, which can be expensive and time- consuming to acquire.

Interpretability: Deep Learning models can be challenging to interpret, making it difficult to understand how they make decisions.

Over fitting: Deep Learning models can sometimes overfit to the training data, resulting in poor performance on new and unseen data.

Black-box nature: Deep Learning models are often treated as black boxes, making it difficult to understand how they work and how they arrived at their predictions.

Applications or Uses of Deep Learning

- **Automatic speech recognition**
  Large-scale automatic speech recognition is the first and most convincing successful case of deep learning.
- **Image recognition**
  Deep learning-based image recognition has become "superhuman", producing more accurate results than human contestants.
- **Visual art processing**
  Identifying the style period of a given painting
  Capturing the style of a given artwork
  Applying it in a visually pleasing manner to an arbitrary photograph or video
- **Natural language processing**
  Natural language processing (NLP) is a subfield of computer science and artificial intelligence (AI) that uses machine learning to enable computers to understand and communicate with human language.
- **Drug discovery and toxicology**
  AI is utilized in drug discovery for target identification, lead optimization, and clinical trial design. It analyzes vast datasets to predict drug properties, identifies potential drug targets, and accelerates overall drug development, contributing to more efficient and effective drug discovery.
- **Customer relationship management**
  CRM AI helps businesses better organize customer information and access that information more easily.
- **Recommendation systems**
  A recommendation system is an artificial intelligence or AI algorithm, usually associated with machine learning, that uses Big Data to suggest or recommend additional products to consumers.
- **Bioinformatics**
  Bioinformatics, as related to genetics and genomics, is a scientific subdiscipline that involves using computer technology to collect, store, analyze and disseminate biological data and information, such as DNA and amino acid sequences or annotations about those sequences.
- **Deep Neural Network Estimations**
  Deep neural networks can be used to estimate the entropy of a stochastic process and called Neural Joint Entropy Estimator (NJEE).

- **Medical image analysis**
  Deep learning has been shown to produce competitive results in medical application such as cancer cell classification, lesion detection, organ segmentation and image enhancement.
- **Mobile advertising**
  Finding the appropriate mobile audience for mobile advertising is always challenging, since many data points must be considered and analyzed before a target segment can be created and used in ad serving by any ad server.
- **Image restoration**
  AI Image restoration tool helps repair damaged and scratched old photos, enhance details, fix colors issues and reduce blur, by making images crisper.
- **Financial fraud detection**
  Deep learning is being successfully applied to financial fraud detection, tax evasion detection, and anti-money laundering.
- **Military**
  The United States Department of Defense applied deep learning to train robots in new tasks through observation.
- **Partial differential equations**
  Physics informed neural networks have been used to solve partial differential equations in both forward and inverse problems in a data driven manner.
- **Image reconstruction**
  Image reconstruction is the reconstruction of the underlying images from the image-related measurements.
- **Weather prediction**
  GraphCast is a deep learning based model, trained on a long history of weather data to predict how weather patterns change over time.
- **Epigenetic clock**
  An epigenetic clock is a biochemical test that can be used to measure age.

2. List and explain the historical trends in Deep Learning.

Historical Trends in Deep Learning

Deep Learning have been three waves of development as shown in figure 1.2:

1. The first wave started with cybernetics in the 1940s-1960s, with the development of theories of biological learning and implementations of the first models such as the perceptron allowing the training of a single neuron.
2. The second wave started with the connectionist approach of the 1980-1995 period, with back-propagation to train a neural network with one or two hidden layers.
3. The current and third wave, deep learning, started around 2006.

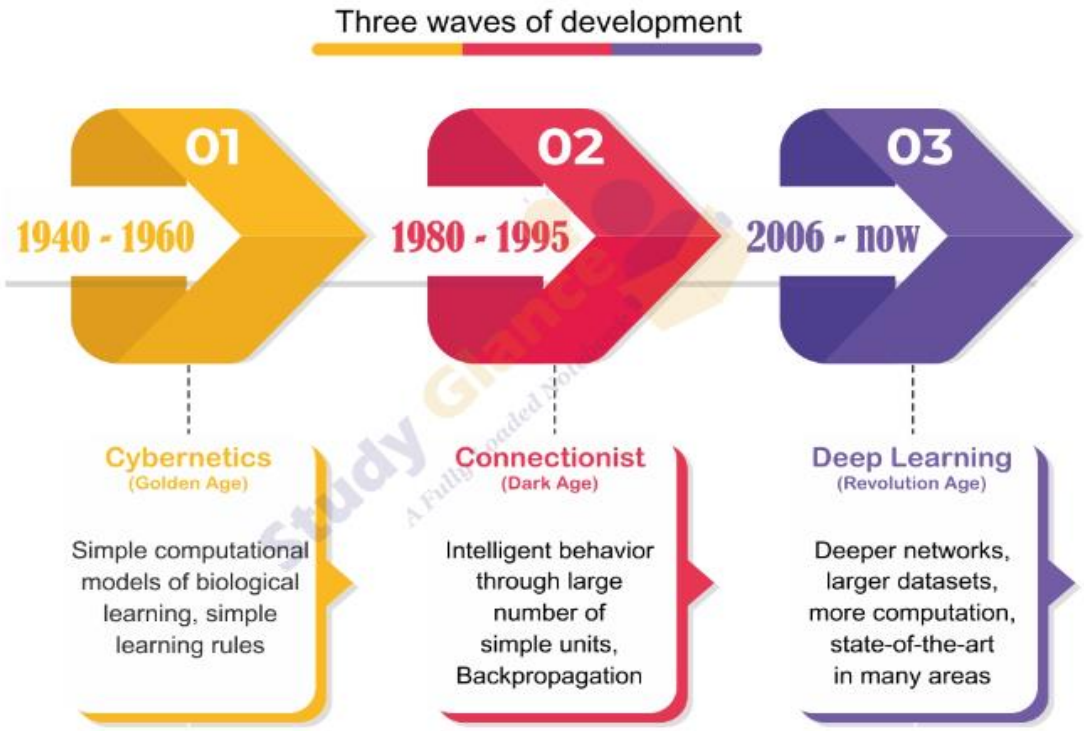


Figure 1.2 Deep Neural Network

Brief History

- **1943:** Warren McCulloch and Walter Pitts create a computational model for neural networks based on mathematics and algorithms called threshold logic.
- **1958:** Frank Rosenblatt creates the perceptron, an algorithm for pattern recognition based on a two-layer computer neural network using simple addition and subtraction.
- **1980:** Kunihiko Fukushima proposes the Neoconitron, a hierarchical, multilayered artificial neural network that has been used for handwriting recognition and other pattern recognition problems.
- **1989:** Scientists were able to create algorithms that used deep neural networks, but training times for the systems were measured in days, making them impractical for real-world use.

- **1992:** Juyang Weng publishes Cresceptron, a method for performing 3-D object recognition automatically from cluttered scenes.
- **Mid-2000s:** The term "deep learning" begins to gain popularity after a paper by Geoffrey Hinton and Ruslan Salakhutdinov showed how a many-layered neural network could be pre-trained one layer at a time.
- **2009:** NIPS Workshop on Deep Learning for Speech Recognition discovers that with a large enough data set, the neural networks don't need pre-training, and the error rates drop significantly.
- **2012:** Artificial pattern-recognition algorithms achieve human-level performance on certain tasks. And Google's deep learning algorithm discovers cats.
- **2014:** Google buys UK artificial intelligence startup Deepmind for £400m
- **2015:** Facebook puts deep learning technology – called DeepFace – into operations to automatically tag and identify Facebook users in photographs. Algorithms perform superior face recognition tasks using deep networks that take into account 120 million parameters.
- **2016:** Google DeepMind's algorithm AlphaGo masters the art of the complex board game Go and beats the professional go player Lee Sedol at a highly publicized tournament in Seoul.

3. Discuss in detail about Scalars, Vectors, Matrices and Tensors in Linear Algebra.

- **Scalars:**
  - A scalar quantity is defined as the physical quantity with only magnitude and no direction.
  - A scalar is a single real number that is used to measure magnitude (size).
  - For example 10, -999 and ½ are scalars.
  - They serve as the elemental components utilized in mathematical computations and algorithmic frameworks.
  - Scalars often represent fundamental quantities such as constants, probabilities, or error metrics.
  - In Data Science, scalars are employed to encapsulate statistical metrics like mean, variance, or correlation coefficients.
  - In Python can represent a Scalar like:

```
#Scalars can be represented simply as numerical variables
scalar = 8.4
scalar
```

Output:

8.4

Vectors:

- A vector quantity is defined as the physical quantity that has both directions as well as magnitude.
- A vector with a value of magnitude equal to one is called a unit vector and is represented by a lowercase alphabet with a "hat" circumflex, i.e. "û".
- Vectors, within the context of Data Science, represent ordered collections of numerical values endowed with both magnitude and directionality.
- In Artificial Intelligence, vectors find application in feature representation, where each dimension corresponds to a distinct feature of the dataset.
- In Machine Learning, vectors play a pivotal role in encapsulating data points, model parameters, and gradient computations during the training process.
- Moreover, within DS, vectors facilitate tasks like data visualization, clustering, and dimensionality reduction.

In Python can represent a Vector like:

```
import numpy as np
# Vectors can be represented as one-dimensional arrays
vector = np.array([2, -3, 1.5])
vector
```


Output:

```
array([ 2. , -3. ,  1.5])
```

Matrices:

- Matrices, as two-dimensional arrays of numerical values, enjoy widespread utility across AI-ML-DS endeavours.
- They serve as foundational structures for organizing and manipulating tabular data, wherein rows typically represent observations and columns denote features or variables.
- Matrices facilitate a plethora of statistical operations, including matrix multiplication, determinant calculation, and singular value decomposition.
- In the domain of AI, matrices find application in representing weight matrices within neural networks, with each element signifying the synaptic connection strength between neurons.
- Similarly, within ML, matrices serve as repositories for datasets, building kernel matrices for support vector machines, and implementing

dimensionality reduction techniques such as principal component analysis.

- Within DS, matrices are indispensable for data pre-processing, transformation, and model assessment tasks.

In Python can represent a Matrix like:

```
import numpy as np
# Matrices can be represented as two-dimensional arrays
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

Output:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Matrix Operations

✓ **Transpose**.

- The transpose of a matrix is the mirror image of the matrix across a diagonal line, called the **main diagonal**, running down and to the right, starting from its upper left corner.

$$(\boldsymbol{A}^\top)_{i,j} = A_{j,i}.$$

- A graphical depiction of this operation.

$$\boldsymbol{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} \Rightarrow \boldsymbol{A}^\top = \begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} \\ A_{1,2} & A_{2,2} & A_{3,2} \end{bmatrix}$$

The transpose of the matrix is a mirror image across the main diagonal.

✓ **Broadcasting**

- The addition of matrix and a vector, yields another matrix:

$$C = A + b,$$

where

$$C_{i,j} = A_{i,j} + b_j.$$

- The vector b is added to each row of the matrix.

- This eliminates the need to define a matrix with b copied into each row before doing the addition.
- This implicit copying of b to many locations is called **broadcasting**.

Types of Matrices

✓ **Diagonal Matrix**

- **Diagonal** matrices consist mostly of zeros and have non-zero entries only along the main diagonal.
- Formally, a matrix D is diagonal if and only if $D_{i,j} = 0$ for all $i \neq j$.

$$A = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix} \quad n \times n$$

Examples $\begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix}$ $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -4 \end{bmatrix}$

✓ **Symmetric Matrix**

- A **symmetric** matrix is any matrix that is equal to its own transpose:

$$A = A^T.$$

Symmetric Matrix

$$A^T = A$$

Skew Symmetric Matrix

$$A^T = -A$$

Example

Symmetric Matrix

$$B = \begin{bmatrix} 2 & 3 & 6 \\ 3 & 4 & 5 \\ 6 & 5 & 9 \end{bmatrix}$$

$$B^T = \begin{bmatrix} 2 & 3 & 6 \\ 3 & 4 & 5 \\ 6 & 5 & 9 \end{bmatrix}$$

Skew Symmetric Matrix

$$A = \begin{bmatrix} 0 & 5 \\ -5 & 0 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 0 & -5 \\ 5 & 0 \end{bmatrix}$$

$$-A = -\begin{bmatrix} 0 & 5 \\ -5 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -5 \\ 5 & 0 \end{bmatrix}$$

✓ **Orthogonal Matrix**

- An **orthogonal matrix** is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

A Square matrix 'A' is orthogonal if

$$A^T = A^{-1}$$

(OR)

$$AA^T = A^TA = I \text{, where}$$

- A^T = Transpose of A
- A^{-1} = Inverse of A
- I = Identity matrix of same order as 'A'

Example

Let us consider a matrix $A = \begin{bmatrix} \cos x & \sin x \\ -\sin x & \cos x \end{bmatrix}$. Its transpose is, $A^T = \begin{bmatrix} \cos x & -\sin x \\ \sin x & \cos x \end{bmatrix}$. We will find the product of these two matrices.

$$AA^T = \begin{bmatrix} \cos x & \sin x \\ -\sin x & \cos x \end{bmatrix}\begin{bmatrix} \cos x & -\sin x \\ \sin x & \cos x \end{bmatrix}$$

$$= \begin{bmatrix} \cos^2 + \sin^2 x & -\cos x \sin x + \sin x \cos x \\ -\sin x \cos x + \cos x \sin x & \sin^2 x + \cos^2 x \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$= I$$

Similarly, we can prove $A^TA = I$. Therefore, A is an orthogonal matrix example of order 2x2.

Tensors:

- Tensors in Data Science generalize the concept of vectors and matrices to better dimensions.

- They are multidimensional arrays of numerical values which can constitute complex facts structures and relationship
- Tensors are integral in deep mastering frameworks like TensorFlow and PyTorch, in which they may be used to shop and control multi-dimensional information which include snap shots, movies, and sequences.
- In AI, tensors are employed for representing enter statistics, version parameters, and intermediate activations in neural networks.
- In ML, tensors facilitate operations in convolutional neural networks, recurrent neural networks, and transformer architectures.
- Moreover, in DS, tensors are utilized for multi-dimensional facts analysis, time-series forecasting, and natural language processing duties.

- ***In Python can represent a Tensor like:***

```
import numpy as np

# Tensors can be represented as multi-dimensional arrays
tensor = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
tensor
```

Output:


```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

Scalar Vs Vector Vs Matrix Vs Tensor

Aspect	Scalar	Vector	Matrix	Tensor
Dimensionality	0	1	2	≥ 3
Representation	Single numerical value	Ordered array of values	Two-dimensional array of values	Multidimensional array of values
Usage	Represent basic quantities	Represent features, observations	Organize data in tabular format	Handle complex data structures

Aspect	Scalar	Vector	Matrix	Tensor
Examples	Error metrics, probabilities	Feature vectors, gradients	Data matrices, weight matrices	Image tensors, sequence tensors
Manipulation	Simple arithmetic operations	Linear algebra operations	Matrix operations, linear transformations	Tensor operations, deep learning operations
Data Representation	Point in space	Direction and magnitude in space	Rows and columns in tabular format	Multi-dimensional relationships
Applications	Basic calculations, statistical measures	Machine learning models, data representation	Data manipulation, statistical analysis	Deep learning, natural language processing
Notation	Lowercase letters or symbols	Boldface letters or arrows	Uppercase boldface letters	Boldface uppercase letters or indices

4. **Discuss in detail about probability distributions. Extrapolate conditional probability and develop a summary of various common probability distribution.**

Probability theory

- Probability theory is a mathematical framework for representing uncertain statements.
- It provides a means of quantifying uncertainty and axioms for deriving new uncertain statements.
- In artificial intelligence applications, probability theory is used in two major ways.
  - First, the laws of probability tells how AI systems should reason, so should design algorithms to compute or approximate various expressions derived using probability theory.

- Second, can use probability and statistics to theoretically analyse the behaviour of proposed AI systems.

- There are three possible sources of uncertainty:
  1. Inherent stochasticity in the system being modelled.
  2. Incomplete observability.
  3. Incomplete modelling.

Random Variables

- A **random variable** is a variable that can take on different values randomly.
- Random variables may be discrete or continuous.
  - A discrete random variable is one that has a finite or countable infinite number of states.
  - A continuous random variable is associated with a real value.

Probability Distributions

- A probability distribution is a statistical function that describes all the possible values and probabilities for a random variable within a given range.
- This range will be bound by the minimum and maximum possible values, but where the possible value would be plotted on the probability distribution will be determined by a number of factors.
- The mean (average), standard deviation, skewness, and kurtosis of the distribution are among these factors.

Types of Probability Distribution

- Discrete Probability Distributions
- Continuous Probability Distributions

Discrete Variables and Probability Mass Functions

- A probability distribution over discrete variables may be described using a **probability mass function** (PMF) denoted with a capital **P** .
- The probability mass function maps from a state of a random variable to the probability of that random variable taking on that state.
- The probability that x = x is denoted as P (x), with a probability of 1 indicating that x = x is certain and a probability of 0 indicating that x = x is impossible.
- Probability mass functions can act on many variables at the same time.
- Such a probability distribution over many variables is known as a **joint probability distribution**.
- P (x = x, y = y) denotes the probability that x = x and y = y

- To be a probability mass function on a random variable x, a function P must satisfy the following properties:
  - The domain of P must be the set of all possible states of x.
  - ∀x ∈ x, 0 ≤ P(x) ≤ 1.
    - An impossible event has probability 0 and no state can be less probable than that.
    - Likewise, an event that is guaranteed to happen has probability 1, and no state can have a greater chance of occurring.
  - $\sum_{x \in \mathrm{x}} P(x) = 1.$ this property is called as **normalized**.

Continuous Variables and Probability Density Functions

- A probability distribution over continuous random variables, describe a **probability density function (PDF)**
- To be a probability density function, a function p must satisfy the Following properties:
  - The domain of p must be the set of p all possible states of x.
  - ∀x ∈ x, p(x) ≥ 0.
- $\int p(x)dx = 1.$

Marginal Probability

- The probability distribution over the subset is known as the **marginal probability** distribution.
- For example, suppose for discrete random variables x and y, find P(x) with the sum rule:

$$\forall x \in \mathrm{x}, P(\mathrm{x} = x) = \sum_{y} P(\mathrm{x} = x, \mathrm{y} = y).$$

- For continuous variables, use integration instead of summation:

$$p(x) = \int p(x, y)dy.$$

Conditional Probability

- The probability of some event, given that some other event has happened is called a **conditional probability**.
- The conditional probability can be computed with the formula

$$P(\mathrm{y} = y \mid \mathrm{x} = x) = \frac{P(\mathrm{y} = y, \mathrm{x} = x)}{P(\mathrm{x} = x)}.$$

The Chain Rule of Conditional Probabilities

- Any joint probability distribution over many random variables may be decomposed into conditional distributions over only one variable.

$$P(\mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)}) \Pi_{i=2}^{n} P(\mathrm{x}^{(i)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(i-1)}).$$

- This observation is known as the **chain rule** or **product rule** of probability.

Independence and Conditional Independence

- Two random variables x and y are **independent** if their probability distribution can be expressed as a product of two factors, one involving only x and one involving only y:

$$\forall x \in \mathrm{x}, y \in \mathrm{y},\ p(\mathrm{x} = x, \mathrm{y} = y) = p(\mathrm{x} = x)p(\mathrm{y} = y).$$

- Two random variables x and y are **conditionally independent** given a random variable z if the conditional probability distribution over x and y factorizes in this way for every value of z:

$$\forall x \in \mathrm{x}, y \in \mathrm{y}, z \in \mathrm{z},\ p(\mathrm{x} = x, \mathrm{y} = y \mid \mathrm{z} = z) = p(\mathrm{x} = x \mid \mathrm{z} = z)p(\mathrm{y} = y \mid \mathrm{z} = z).$$

- Can denote independence and conditional independence with compact Notation:

$\mathrm{x} \perp \mathrm{y}$ means that x and y are independent,

$\mathrm{x} \perp \mathrm{y} \mid \mathrm{z}$ means that x and y are conditionally independent given z.

Expectation, Variance and Covariance

- **Expectation**
  - The **expectation** or **expected value** of some function f(x) with respect to a probability distribution P (x) is the average or mean value that f takes on when x is drawn from P.
  - For discrete variables it is computed with a summation:

$$\mathbb{E}_{\mathrm{x} \sim P}[f(x)] = \sum_{x} P(x) f(x),$$

  - For continuous variables, it is computed with an integral:

$$\mathbb{E}_{\mathrm{x} \sim p}[f(x)] = \int p(x) f(x) dx.$$

Expectations are linear,

$$\mathbb{E}_{\mathrm{x}}[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}_{\mathrm{x}}[f(x)] + \beta \mathbb{E}_{\mathrm{x}}[g(x)],$$

when α and β are not dependent on x.

- **Variance**
  - The **variance** gives a measure of how much the values of a function of a random variable x vary for different values of x from its probability distribution:

$$\mathrm{Var}(f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right].$$

  - When the variance is low, the values of f (x) cluster near their expected value.
  - The square root of the variance is known as the **standard deviation**.

- **Covariance**
  - The **covariance** defines how much two values are linearly related to each other, as well as the scale of these variables:

$$\mathrm{Cov}(f(x), g(y)) = \mathbb{E}\left[(f(x) - \mathbb{E}\left[f(x)\right])\left(g(y) - \mathbb{E}\left[g(y)\right]\right)\right].$$

  - The **covariance matrix** of a random vector $\boldsymbol{x} \in \mathbb{R}^n$ is an n × n matrix, such that

$$\mathrm{Cov}(\mathbf{x})_{i,j} = \mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_j).$$

  - The diagonal elements of the covariance give the variance:

$$\mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_i) = \mathrm{Var}(\mathrm{x}_i).$$

Common Probability Distributions

a. Bernoulli Distribution

- The **Bernoulli** distribution is a distribution over a single binary random variable.
- It is controlled by a single parameter φ ∈ [0, 1], which gives the probability of the random variable being equal to 1.
- It has the following properties:

$$P(\mathrm{x}=1)=\phi$$

$$P(\mathrm{x}=0)=1-\phi$$

$$P(\mathrm{x}=x)=\phi^{x}(1-\phi)^{1-x}$$

$$\mathbb{E}_{\mathrm{x}}[\mathrm{x}]=\phi$$

$$\mathrm{Var}_{\mathrm{x}}(\mathrm{x})=\phi(1-\phi)$$

b. Multinoulli Distribution

- The **multinoulli** or **categorical** distribution is a distribution over a single discrete variable with k different states, where k is finite.
- The multinoulli distribution is a special case of the **multinomial** distribution.
- The multinoulli distribution is parametrized by a vector $\boldsymbol{p} \in [0,1]^{k-1}$, where p_i gives the probability of the i-th state.
- The final, k -th state's probability is given by $1-\mathbf{1}^{\top}\boldsymbol{p}$.
  The Bernoulli and multinoulli distributions are sufficient to describe any distribution over their domain.

c. Gaussian Distribution

a. The most commonly used distribution over real numbers is the **normal distribution**, also known as the **Gaussian distribution**:

$$\mathcal{N}(x;\mu,\sigma^2)=\sqrt{\frac{1}{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

b. Refer figure 1.3 for a plot of the density function.

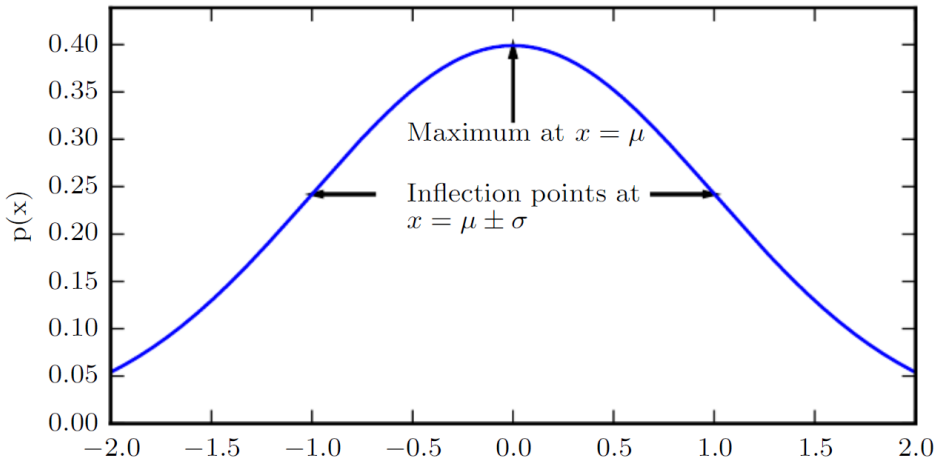


Figure 1.3: The normal distribution: The normal distribution N (x ; μ, σ²) exhibits a classic "bell curve" shape, with the x coordinate of its central peak given by μ, and the width of its peak controlled by σ. This example, depicts the standard normal distribution, with μ = 0 and σ = 1.

i. The isotropic Gaussian distribution, whose covariance matrix is a scalar times the identity matrix

d. Exponential and Laplace Distributions

- **Exponential distribution**
  - To have a probability distribution with a sharp point at x = 0 use the **exponential distribution**:

$$p(x;\lambda) = \lambda \mathbf{1}_{x\geq 0} \exp\left(-\lambda x\right).$$

  - The exponential distribution uses the indicator function $1_{x\geq 0}$ to assign probability zero to all negative values of x.

- **Laplace Distributions**
  - To place a sharp peak of probability mass at an arbitrary point μ is the **Laplace distribution**

$$\text{Laplace}(x;\mu,\gamma) = \frac{1}{2\gamma}\exp\left(-\frac{|x-\mu|}{\gamma}\right).$$

e The Dirac Distribution and Empirical Distribution

- **Dirac Distribution**
  - To specify that all of the mass in a probability distribution cluster around a single point can be accomplished by defining a PDF using the Dirac delta function, δ(x):

$$p(x) = \delta(x-\mu).$$

  - The Dirac delta function is defined such that it is zero-valued everywhere except 0, yet integrates to 1.
- **Empirical distribution**,
  - A common use of the Dirac delta distribution is as a component of an **empirical distribution**,

$$\hat{p}(\boldsymbol{x}) = \frac{1}{m}\sum_{i=1}^{m}\delta(\boldsymbol{x}-\boldsymbol{x}^{(i)})$$

Useful Properties of Common Functions

- **Logistic sigmoid (Refer Figure 1.4)**:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

  - The logistic sigmoid is commonly used to produce the φ parameter of a Bernoulli distribution because its range is (0,1), which lies within the valid range of values for the φ parameter.

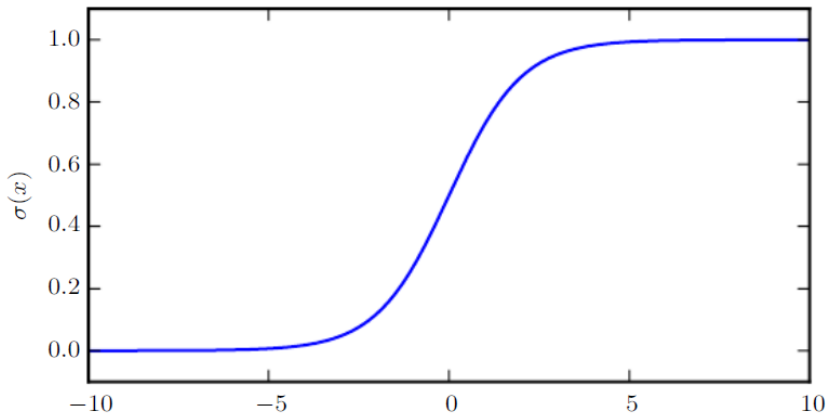


Figure 1.4: The logistic sigmoid function.

- **Softplus function (Refer Figure 1.5)**
  - The softplus function can be useful for producing the β or σ parameter of a normal distribution because its range is (0, ∞).
  - It also arises commonly when manipulating expressions involving sigmoids.

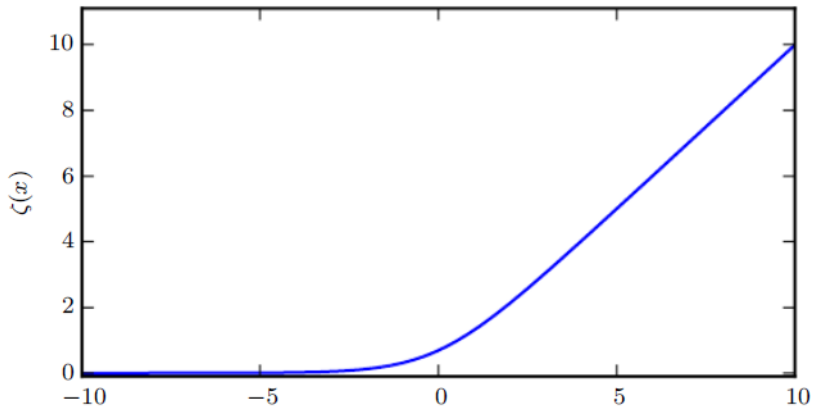


Figure 1.5: The softplus function.

- **Bayes' Rule**
  - Bayes' rule is derived from the definition of conditional Probability

Formula For Bayes' Theorem

$$P(A|B) = \frac{P(A \bigcap B)}{P(B)} = \frac{P(A) \cdot P(B|A)}{P(B)}$$

where:

$P(A) =$ The probability of A occurring

$P(B) =$ The probability of B occurring

$P(A|B) =$ The probability of A given B

$P(B|A) =$ The probability of B given A

$P\left(A \bigcap B\right)) =$ The probability of both A and B occurri

5. Explain in detail about Gradient Based Optimization. Discuss in detail about Stochastic Gradient Descent in Deep Learning.

Gradient

- A gradient is a derivative that defines the effects on outputs of the function with a little bit of variation in inputs.

Gradient Descent

- Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that that aims to find the optimal parameters—weights and biases— by minimizing the cost function of a neural network model during training.
- It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.
- The learning happens during the back propagation while training the neural network-based model.
- There is a term known as Gradient Descent, which is used to optimize the weight and biases based on the cost function.
- The cost function evaluates the difference between the actual and predicted outputs.
- The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.
- These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.
- Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks.

Learning Rate:

- It is defined as the step size taken to reach the minimum or lowest point.
- This is typically a small value that is evaluated and updated based on the behaviour of the cost function.
- If the learning rate is high, it results in larger steps but also leads to risks of overshooting the minimum.
- At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.
- Refer Figure 1.6

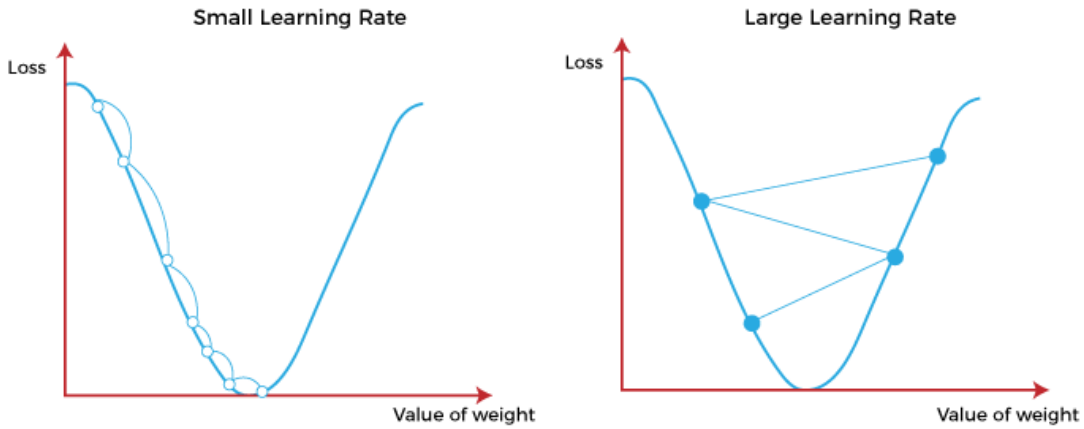


Figure 1.6: Learning Rate.

➢ **Working of Gradient Descent Algorithm**

- Loss function for the single row is

$$J(w,b) = \frac{1}{n}(y_p - y)^2$$

- In the above function x and y are our input data i.e constant.
- To find the optimal value of weight w and bias b. partially differentiate with respect to w and b.

Gradient of J(w,b) with respect to w

$$
\begin{aligned}
J'_w &= \frac{\partial J(w,b)}{\partial w} \\
&= \frac{\partial}{\partial w}\left[\frac{1}{n}(y_p - y)^2\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial w}\left[(y_p - y)\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial w}\left[((xW^T + b) - y)\right] \\
&= \frac{2(y_p - y)}{n}\left[\frac{\partial(xW^T + b)}{\partial w} - \frac{\partial(y)}{\partial w}\right] \\
&= \frac{2(y_p - y)}{n}\left[x - 0\right] \\
&= \frac{1}{n}(y_p - y)[2x]
\end{aligned}
$$

i.e

$$
\boxed{\begin{aligned} J'_w &= \frac{\partial J(w,b)}{\partial w} \\ &= J(w,b)[2x] \end{aligned}}
$$

Gradient of J(w,b) with respect to b

$$
\begin{aligned}
J'_b &= \frac{\partial J(w,b)}{\partial b} \\
&= \frac{\partial}{\partial b}\left[\frac{1}{n}(y_p - y)^2\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial b}\left[(y_p - y)\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial b}\left[((xW^T + b) - y)\right] \\
&= \frac{2(y_p - y)}{n}\left[\frac{\partial(xW^T + b)}{\partial b} - \frac{\partial(y)}{\partial b}\right] \\
&= \frac{2(y_p - y)}{n}\left[1 - 0\right] \\
&= \frac{1}{n}(y_p - y)[2]
\end{aligned}
$$

i.e

$$J'_b = \frac{\partial J(w,b)}{\partial b} = J(w,b)[2]$$

In a fully connected neural network model there can be multiple layers and multiple parameters

$$Param = Param - \gamma \nabla J$$

Here,

- γ = Learning rate
- J = Loss function
- ∇ = Gradient symbol denotes the derivative of loss function J
- Param = weight and bias
- There can be multiple weight and bias values depending upon the complexity of the model and features in the dataset

- **Implementations of the Gradient Descent algorithm for the above model**

Steps:

- Find the gradient using loss.backward()
- Get the parameter using model.linear.weight and model.linear.bias
- Update the parameter using the above-defined equation.
- Again assign the model parameter to our model

```
# Find the gradient using
        loss.backward()
# Learning Rate
        learning_rate = 0.001
# Model Parameter
        w = model.linear.weight
        b = model.linear.bias
# Update the model parameter
        w = w - learning_rate * w.grad
        b = b - learning_rate * b.grad
```

```
# assign the weight & bias parameter to the linear layer
model.linear.weight = nn.Parameter(w)
model.linear.bias   = nn.Parameter(b)
```

- **Types of Gradient Descent**

1. Batch Gradient Descent:

- Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples.
- This procedure is known as the training epoch..

Advantages of Batch gradient descent:

- It produces less noise in comparison to other gradient descent.
- It produces stable gradient descent convergence.
- It is computationally efficient as all resources are used for all training samples.

2. Stochastic gradient descent

- Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration.

Advantages of Stochastic gradient descent:

- It is easier to allocate in desired memory.
- It is relatively fast to compute than batch gradient descent.
- It is more efficient for large datasets.

3. MiniBatch Gradient Descent:

- Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent.
- It divides the training datasets into small batch sizes then performs the updates on those batches separately.

Advantages of Mini Batch gradient descent:

- It is easier to fit in allocated memory.
- It is computationally efficient.
- It produces stable gradient descent convergence.

- **Stochastic Gradient Descent (SGD):**

- Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm that is used for optimizing machine learning models.
- It addresses the computational inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.

- In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters.
- This random selection introduces randomness into the optimization process, hence the term "stochastic" in stochastic Gradient Descent
- The advantage of using SGD is its computational efficiency, especially when dealing with large datasets.

Stochastic Gradient Descent Algorithm

- **Initialization:** Randomly initialize the parameters of the model.
- **Set Parameters**: Determine the number of iterations and the learning rate (alpha) for updating the parameters.
- **Stochastic Gradient Descent Loop:** Repeat the following steps until the model converges or reaches the maximum number of iterations:
- **Shuffle the training dataset** to introduce randomness.
- **Iterate over each training example** (or a small batch) in the shuffled order.
- **Compute the gradient of the cost function** with respect to the model parameters using the current training example (or batch).
- **Update the model parameters** by taking a step in the direction of the negative gradient, scaled by the learning rate.
- **Evaluate the convergence criteria**, such as the difference in the cost function between iterations of the gradient.
- **Return Optimized Parameters:** Once the convergence criteria are met or the maximum number of iterations is reached, return the optimized model parameters.

- In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier with a significantly shorter training time.
- A path taken by Stochastic Gradient Descent is shown in figure 1.7.

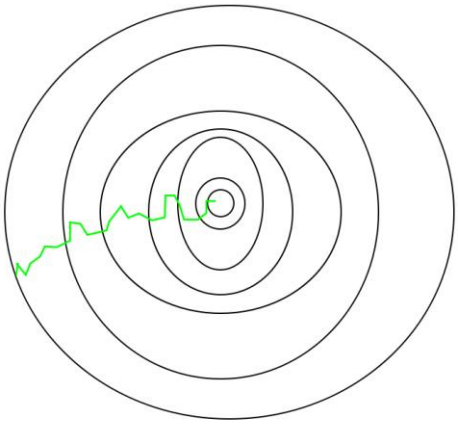


Figure 1.7: Path taken by Stochastic Gradient Descent.

Stochastic Gradient Descent (SGD)

- Nearly all deep learning is powered by SGD, SGD extends the gradient descent algorithm
- Suppose $y = f(x)$ where both x and y are real nos.
- Derivative is a function denoted as $f'(x)$ or dy/dx
- It gives the slope of $f(x)$ at the point x, i.e., it specifies how to make a small change in the input to make a corresponding change in the output

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$$

For multiple inputs need partial derivatives $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ is how f changes as only x_i increases

- Gradient of f is a vector of partial derivatives $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$
- Gradient descent proposes a new point

$$\boldsymbol{x}' = \boldsymbol{x} - \varepsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

Where ε is the learning rate, a positive scalar.

6. Define Machine Learning and elaborate the basic concepts of Machine Learning in detail.

➢ **Machine Learning**

- Machine learning (ML) is a discipline of artificial intelligence (AI) that provides machines the ability to automatically learn from data and past experiences to identify patterns and make predictions with minimal human intervention.

- It enables algorithms to uncover hidden patterns within datasets, allowing them to make predictions on new, similar data without explicit programming for each task.
- Figure 1.8 depicts the operation of Machine Learning algorithms
- **Example:** categorizing images, analyzing data, or predicting price fluctuations.

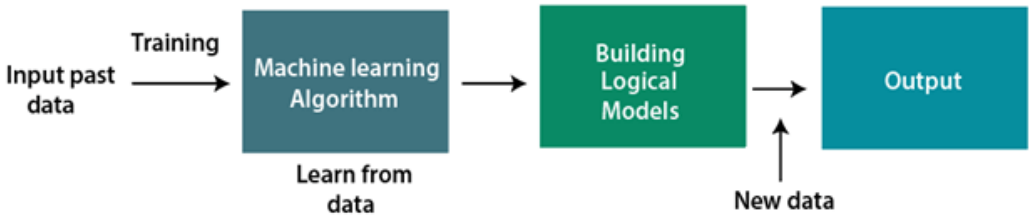


Figure 1.8: Operation of Machine Learning Algorithms

- **Mitchell (1997) provides the definition**

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E."

The Task, T

- Machine learning allows to tackle tasks that are too difficult to solve.
- For example, if want a robot to be able to walk, then walking is the task.
- Some of the most common machine learning tasks include the following:

i. **Classification**:

- In this type of task, the computer program is asked to specify which of k categories some input belongs to.
- An example of a classification task is object recognition, where the input is an image, and the output is a numeric code identifying the object in the image.
- Modern object recognition is best accomplished with deep learning.
- Object recognition is the same basic technology that allows computers to recognize faces, which can be used to

automatically tag people in photo collections and allow computers to interact more naturally with their users.

ii. **Classification with missing inputs**:

- Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided.
- In order to solve the classification task, the learning algorithm only has to define a *single* function mapping from a vector input to a categorical output.
- When some of the inputs may be missing, rather than providing a single classification function, the learning algorithm must learn a *set* of functions.
- This kind of situation arises frequently in medical diagnosis, because many kinds of medical tests are expensive or invasive.

iii. **Regression**:

- In this type of task, the computer program is asked to predict a numerical value given some input.
- An example of a regression task is the prediction of the expected claim amount that an insured person will make, or the prediction of future prices of securities.
- These kinds of predictions are also used for algorithmic trading.

iv Transcription:

- In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe it into discrete, textual form.
- For example, in optical character recognition, the computer program is shown a photograph containing an image of text and is asked to return this text in the form of a sequence of characters (e.g., in ASCII or Unicode format).
- Another example is speech recognition.

v **Machine translation**:

- In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language.
- This is commonly applied to natural languages, such as translating from English to French.

Vi **Anomaly detection**:

- In this type of task, the computer program sifts through a set of events or objects, and flags some of them as being unusual or atypical.
- An example of an anomaly detection task is credit card fraud detection.
- By modelling your purchasing habits, a credit card company can detect misuse of your cards.

The Performance Measure, P

- In order to evaluate the abilities of a machine learning algorithm, must design a quantitative measure of its performance.
- Usually this performance measure P is specific to the task T being carried out by the system.
- For tasks such as classification, classification with missing inputs, and transcription, it is often measured the **accuracy** of the model.
- Accuracy is just the proportion of examples for which the model produces the correct output.
- Can also obtain equivalent information by measuring the **error rate** ,
- The error rate is often referred as the expected 0-1 loss.
- The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not.

The Experience, E

- Refer Supervised, Unsupervised and Reinforcement Learning from Part B – Q. 1**.**

7. Discuss in detail about Capacity, Over fitting and Under fitting

➢ **Generalization**

- Central challenge of ML is that the algorithm must perform well on new, previously unseen inputs and not just those on which the model has been trained
- Ability to perform well on previously unobserved inputs is called generalization
- The train and test data are generated by a probability distribution over datasets called the data generating process.

➢ **Generalization Error**

- The generalization error also called the test error of a machine learning model is estimated by measuring its performance on a test set of examples that were collected separately from the training set.

$$\frac{1}{m^{(\text{test})}}\left\|X^{(\text{test})}\boldsymbol{w}-\boldsymbol{y}^{(\text{test})}\right\|_2^2$$

- When training a machine learning model, using a training set, can compute some error measure on the training set called the training error.

$$\frac{1}{m^{(\text{train})}}\left\|X^{(\text{train})}\boldsymbol{w}-\boldsymbol{y}^{(\text{train})}\right\|_2^2$$

➢ **Under- and Over-fitting**

- Factors determining how well an ML algorithm will perform are its ability to:
  - Make the training error small
  - Make gap between training and test errors small
- They correspond to two ML challenges
  - Underfitting - Inability to obtain low enough error rate on the training set
  - Overfitting - Gap between training error and testing error is too large
- Whether a model is more likely to overfit or underfit can be controlled by altering its capacity

➢ **Capacity of a model**

- Capacity of a model refers to the ability of the model to fit a wide range of possible functions.
  – Models with high capacity tend to overfit.
  – Models with low capacity tend to underfit.

Hypothesis space

- One way to control capacity of a learning algorithm is by choosing the hypothesis space.
- A set of functions that the learning algorithm is allowed to select as being the solution
- Example - the linear regression algorithm has the set of all linear functions of its input as the hypothesis space.

➢ **Appropriate Capacity**

- Machine Learning algorithms will perform well when their capacity is appropriate.
- Models with insufficient capacity are unable to solve complex tasks

- Models with high capacity can solve complex tasks, they may overfit

Principle of Capacity in action

- Figure 1.9a,1.9b and 1,9c, fits three models to a training set
- Data generated synthetically sampling x values and choosing y deterministically (a quadratic function)

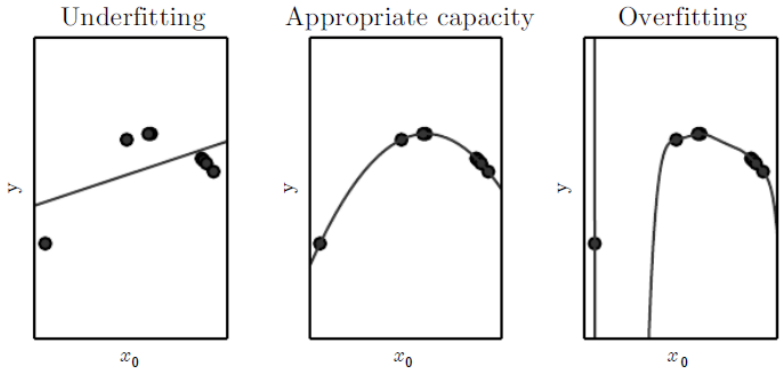


Figure 1.9a, 1.9b, 1.9c: Three models to this example training set

Figure 1.9a:

- A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data.

Figure 1.9b:

- A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting.

Figure 1.9c:

- A polynomial of degree 9 fit to the data suffers from overfitting.
- The solution passes through all of the training points exactly, but not able to extract the correct structure.

➢ **Representational and Effective Capacity**

- **Representational capacity:**
  - The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective.
  - It refers to the extent to which for a wide range of possible functions, some model in the class approximates that function well.
  - **Example:** Deep neural networks have very high representational capacity.

- **Effective capacity:**
  - Given a specific learning algorithm with a specific amount of data refers to the extent to which for a wide range of possible functions, the model

in the class produced by the learning algorithm can approximate that function well.

Typical relationship between capacity and error.

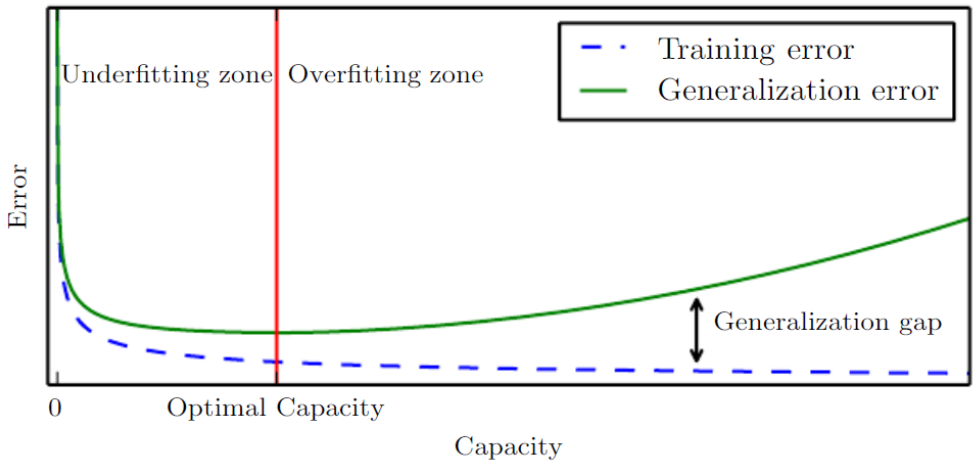


Figure 1.10: Typical relationship between capacity and error.

- In figure 1.10 Training and test error behave differently.
- At the left end of the graph, training error and generalization error are both high. This is the **underfitting regime**.
- As the increase in capacity, training error decreases, but the gap between training and generalization error increases.
- Eventually, the size of this gap outweighs the decrease in training error, and we enter the **overfitting regime**, where capacity is too large, above the **optimal capacity**.

8. Discuss in detail about Hyperparameters and Validation sets.

➢ **Hyperparameters**

- Hyperparameters are those parameters that are explicitly defined by the user to control the learning process.
- These hyperparameters are used to improve the learning of the model, and their values are set before starting the learning process of the model.
- These settings are called **hyperparameters**.
- **Examples** of Hyperparameters
  - The k in kNN or K-Nearest Neighbour algorithm
  - Train-test split ratio
  - Branches in Decision Tree
  - Number of clusters in Clustering Algorithm

➢ **Validation Set**

- Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because it is difficult to optimize.
- To solve this problem, a **validation set** of examples that the training algorithm does not observe is required.
- Specifically, the training data is split into two disjoint subsets.
  - The subset of data used to learn the parameters is called the training set
  - The subset of data used to guide the selection of hyperparameters is called the validation set.
- Typically, one uses about 80% of the training data for training and 20% for validation.

Cross validation

- Cross validation is a technique used in machine learning to evaluate the performance of a model on unseen data.
- It involves dividing the available data into multiple folds or subsets, using one of these folds as a validation set, and training the model on the remaining folds.
- This process is repeated multiple times, each time using a different fold as the validation set.
- Finally, the results from each validation step are averaged to produce a more robust estimate of the model's performance.
- The main purpose of cross validation is to prevent overfitting, which occurs when a model is trained too well on the training data and performs poorly on new, unseen data.

k-fold cross-validation

Procedure

- The dataset is divided into k folds while maintaining the proportion of classes in each fold.
- During each iteration, one-fold is used for testing, and the remaining folds are used for training.
- The process is repeated k times, with each fold serving as the test set exactly once.

k -fold cross validation algorithm

Define KFoldXV($\mathbb{D}, A, L, k$):
Require: $\mathbb{D}$, the given dataset, with elements $\boldsymbol{z}^{(i)}$
Require: A, the learning algorithm, seen as a function that takes a dataset as input and outputs a learned function
Require: L, the loss function, seen as a function from a learned function f and an example $\boldsymbol{z}^{(i)} \in \mathbb{D}$ to a scalar $\in \mathbb{R}$
Require: k, the number of folds
Split $\mathbb{D}$ into k mutually exclusive subsets $\mathbb{D}_i$, whose union is $\mathbb{D}$.
for i from 1 to k **do**
  $f_i = A(\mathbb{D} \backslash \mathbb{D}_i)$ — Train A on dataset without D_i
  for $\boldsymbol{z}^{(j)}$ in $\mathbb{D}_i$ **do**
    $e_j = L(f_i, \boldsymbol{z}^{(j)})$ — Determine errors for samples in D_i
  end for
end for
Return $\boldsymbol{e}$ — Return vector of errors $\boldsymbol{e}$ for samples in D

9. Discuss in detail about Estimators, Bias and Variance

- Statistics provides many tools to achieve the ML goal of solving a task and also to notions of generalization, over and under-fitting
- Tools are
  - Parameter estimation
  - Bias
  - Variance

Estimator

- An estimator in statistics is a rule or a formula that helps us to estimate the value of an unknown parameter in a population using sample data.
- An estimator is a function of the sample data that provides an estimate of a population parameter.

Properties of Good Estimators

- **Unbiasedness**
  - An estimator is said to be unbiased if its expected value is equal to the true value of the parameter being estimated.
  - An estimator that systematically overestimates or underestimates the parameter is considered biased.
- **Consistency**
  - A consistent estimator is one where the estimates become closer to the true parameter value as the sample size increases.
- **Efficiency**
  - Efficiency refers to the variance of the estimator.
  - An efficient estimator provides more precision and is less spread out around the true parameter value.

- **Sufficiency**
  - A sufficient estimator captures all the information in the sample that is needed to estimate the parameter.

Types of Estimators

- **Point Estimators**
  - A point estimator provides a single value as the estimate of the parameter. For instance, the sample mean is a point estimator for the population mean.
  - To distinguish estimates of parameters from their true value, a point estimate of a parameter θ is represented by $\hat{\theta}$
  - Let $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)},..\boldsymbol{x}^{(m)}\}$ be m independent and identically distributed data points, then a point estimator or statistic is any function of the data

$$\hat{\theta}_m = g(\boldsymbol{x}^{(1)},...\boldsymbol{x}^{(m)})$$

- **Interval Estimators**
  - An interval estimator, or an interval estimate, provides a range of values within which the parameter is expected to lie. Confidence intervals are a common example of interval estimation as shown in figure 1.11.

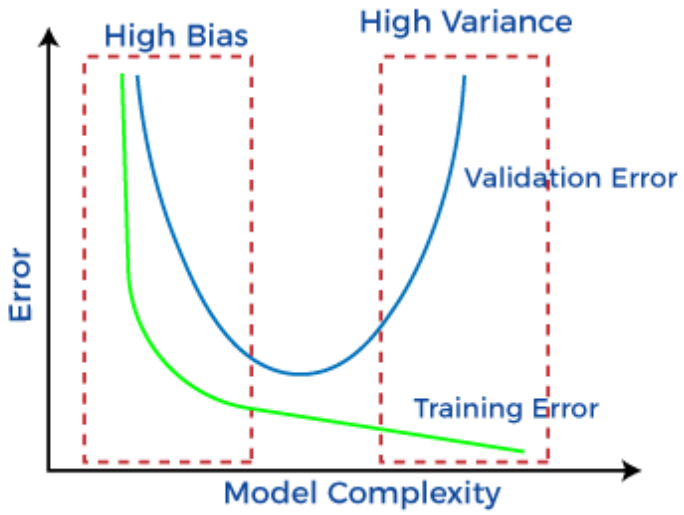


Figure 1.11 interval estimator

Bias

- Bias is a systematic error that occurs due to wrong assumptions in the machine learning process.

- While making predictions, a difference occurs between prediction values made by the model and actual values/expected values, and this difference is known as bias errors or Errors due to bias.
- **Low Bias:** Low bias value means fewer assumptions are taken to build the target function. In this case, the model will closely match the training dataset.

  Examples - Decision Trees, k-Nearest Neighbor's and Support Vector Machines.
- **High Bias:** High bias value means more assumptions are taken to build the target function. In this case, the model will not match the training dataset closely.
- **Examples** - Linear Regression, Linear Discriminant Analysis and Logistic Regression.

Ways to reduce high bias in Machine Learning:

- **Use a more complex model:** One of the main reasons for high bias is the very simplified model. In such cases, make the model more complex by increasing the number of hidden layers in the case of a deep neural network..
- **Increase the number of features:** By adding more features to train the dataset will increase the complexity of the model..
- **Increase the size of the training data:** Increasing the size of the training data can help to reduce bias by providing the model with more examples to learn from the dataset.

Bias of an estimator

- The bias of an estimator $\hat{\theta}_m = g(\boldsymbol{x}^{(1)}, \ldots \boldsymbol{x}^{(m)})$ for parameter θ is defined as

$$\text{bias}(\hat{\theta}_m) = E[\hat{\theta}_m] - \theta$$

- The estimator is unbiased if $\text{bias}(\hat{\theta}_m) = 0$

Example of Estimator Bias - Gaussian distribution: mean μ

Estimator of Gaussian mean

Sample mean is an estimator of the mean parameter

$$\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

To determine bias of the sample mean:

$$\begin{aligned}
\text{bias}(\hat{\mu}_m) &= \mathbb{E}[\hat{\mu}_m] - \mu \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right]\right) - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mu\right) - \mu \\
&= \mu - \mu = 0
\end{aligned}$$

Thus the sample mean is an unbiased estimator of the Gaussian Mean.

- **Variance**
  - Variance is the amount by which the performance of a predictive model changes when it is trained on different subsets of the training data.
  - Variance errors are either low or high-variance errors.
  - **Low variance** means there is a small variation in the prediction of the target function with changes in the training data set.
  - **Example:** Linear Regression, Logistic Regression, and Linear discriminant analysis
  - **High variance** shows a large variation in the prediction of the target function with changes in the training dataset.
  - A model that shows high variance learns a lot and perform well with the training dataset, and does not generalize well with the unseen dataset.
  - **Example:** decision tree, Support Vector Machine, and K-nearest neighbor's.

Variance of an estimator

- The variance of an estimator is simply $\mathrm{Var}(\hat{\theta})$ where the random variable is the training set
- The square root of the variance is called the standard error, denoted $\mathrm{SE}(\hat{\theta})$
- The standard error of the mean is given by

$$SE(\hat{\mu}_m) = \sqrt{Var\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}$$

Example: Bernoulli Distribution

Consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ drawn independently and identically from a Bernoulli distribution

Computing the variance of the estimator

$$\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}.$$

$$\begin{aligned}
\mathrm{Var}\left(\hat{\theta}_m\right) &= \mathrm{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \mathrm{Var}\left(x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \theta(1-\theta) \\
&= \frac{1}{m^2} m\theta(1-\theta) \\
&= \frac{1}{m}\theta(1-\theta)
\end{aligned}$$

- The variance of the estimator decreases as a function of m, the number of examples in the dataset.

Ways to Reduce High Variance:

- Reduce the input features or number of parameters.
- Do not use a much complex model.

- Increase the training data.

10. List and describe the challenges motivating Deep Learning.

➢ **Challenges Motivating DL**

1. Curse of dimensionality

- Many machine learning problems become exceedingly difficult when the number of dimensions in the data is high.
- This phenomenon is known as the curse of dimensionality.
- Refer figure 1.11

Figure 1.11: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially.

2. Local Constancy & Smoothness Regularization

- In order to generalize well, machine learning algorithms need to be guided by prior beliefs about what kind of function they should learn.
- Among the most widely used priors is the smoothness or local constancy prior. A
- function is said to have local constancy if it does not change much within a small region of space.
- As the machine learning algorithm becomes simpler, it tends to rely extensively on this prior.
- **Example:** K nearest neighbors.
- Among the most widely used of these implicit "priors" is the smoothness prior or local constancy prior.
- This prior states that the function should not change very much within a small region.
- Many simpler algorithms rely exclusively on this prior to generalize well, and as a result they fail to scale to the statistical challenges involved in solving AI- level tasks.

3. Manifold Learning

- A manifold is a mathematical abstraction used to describe complex geometric objects, such as curved surfaces or folded structures, in terms of local coordinates and intrinsic dimensions.

- A manifold is a connected region, it is a set of points, associated with a neighbourhood around each point.
- **Manifold Learning**
  - Manifold learning is a technique for dimensionality reduction that seeks to preserve the underlying structure of high-dimensional data while representing it in a lower-dimensional environment.
  - In machine learning, manifold learning is crucial in order to overcome the challenges posed by high-dimensional and non-linear data.
- Reducing the amount of features in a dataset is done using the dimensionality reduction technique.
- Refer figure 1.12 for Manifold Learning

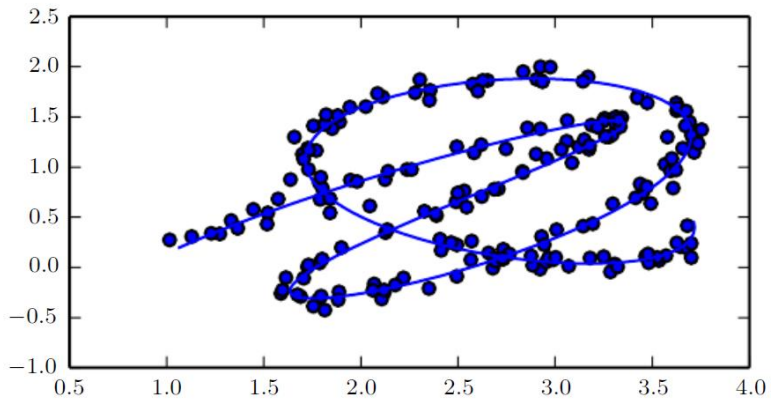


Figure 1.12: Data sampled from a distribution in a two-dimensional space that is actually concentrated near a one-dimensional manifold, like a twisted string. The solid line indicates the underlying manifold that the learner should infer.

4. **Data Availability and Quality:**
   - Deep learning models require large amounts of labelled data for training.
   - Acquiring and annotating such data can be expensive and time-consuming.
   - Additionally, the quality and representativeness of the data are crucial for model performance.
5. **Overfitting:**
   - Deep neural networks are prone to overfitting, especially when dealing with complex models and small datasets.
6. **Computation and Resource Intensiveness:**
   - Training deep learning models can be computationally expensive.
7. **Interpretability:**

- Deep neural networks are often considered "black-box" models, making it challenging to interpret their decisions.
- Understanding why a model makes a particular prediction is important in applications such as healthcare and finance.

8. **Hyperparameter Tuning:**
   - Selecting the right architecture and hyperparameters for a deep learning model can be a time-consuming and iterative process.
   - Grid search, random search, or Bayesian optimization techniques are used to address this challenge.
9. **Transfer Learning and Pretraining:**
   - Fine-tuning pretrained models can be challenging, as selecting the right layers to freeze and update, and adapting the model to a specific task requires expertise.
10. **Data Privacy and Security:**
    - Deep learning models can inadvertently leak sensitive information contained in training data, posing privacy and security concerns. Techniques like differential privacy are being explored to address this challenge.
11. **Ethical and Bias Concerns:**
    - Deep learning models can inherit biases present in training data, leading to biased predictions.
    - Addressing bias and ensuring fairness in AI systems is a critical challenge.
12. **Robustness:**
    - Deep learning models can be sensitive to adversarial attacks, where small, carefully crafted perturbations to input data can lead to incorrect predictions.
    - Ensuring model robustness is an ongoing research area.
13. **Scalability:**
    - While deep learning has achieved remarkable results, scaling models to handle even larger datasets and more complex tasks can be challenging.
    - Novel architectures and distributed training techniques are being developed to address this issue.
14. **Energy Consumption:**
    - Training large deep learning models can consume significant amounts of energy, contributing to environmental concerns.
    - Research into more energy-efficient algorithms and hardware is ongoing.
15. **Generalization:**

- Deep learning models may not always generalize well to new, unseen data, especially if the distribution of the test data differs from the training data.
- Techniques like domain adaptation are used to improve generalization.

11. Discuss in detail about Deep Network and Deep Feedforward Network

- A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain.
- **Deep feedforward networks DFN**, also often called **feedforward neural networks**, or **multilayer perceptrons** (MLPs), are such neural networks which only uses input to feed forward through a function. There is no feedback mechanism in **DFN.**
- The goal of a feedforward network is to approximate some function f^*.
- For example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input x to a category y.
- A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the parameters θ that result in the best function approximation.
- These models are called **feedforward** because information flows through the function being evaluated from x, through the intermediate computations used to define f , and finally to the output y.
- There are no **feedback** connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks**
- For example, the convolutional networks used for object recognition from photos are a specialized kind of feedforward network.
- Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together.
- For example, Consider three functions $f^{(1)}, f^{(2)}, \text{and } f^{(3)}$) connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the **first layer** of the network, $f^{(2)}$ is called the **second layer**, and so on. The overall length of the chain gives the **depth** of the model. The final layer of a feedforward network is called the **output layer**.
- The training examples specify directly what the output layer must do at each point x; it must produce a value that is close to y.

- The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f∗. Because the training data does not show the desired output for each of these layers, these layers are called **hidden layers**.
- Finally, these networks are called *neural* because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the **width** of the model. Refer 1.13.

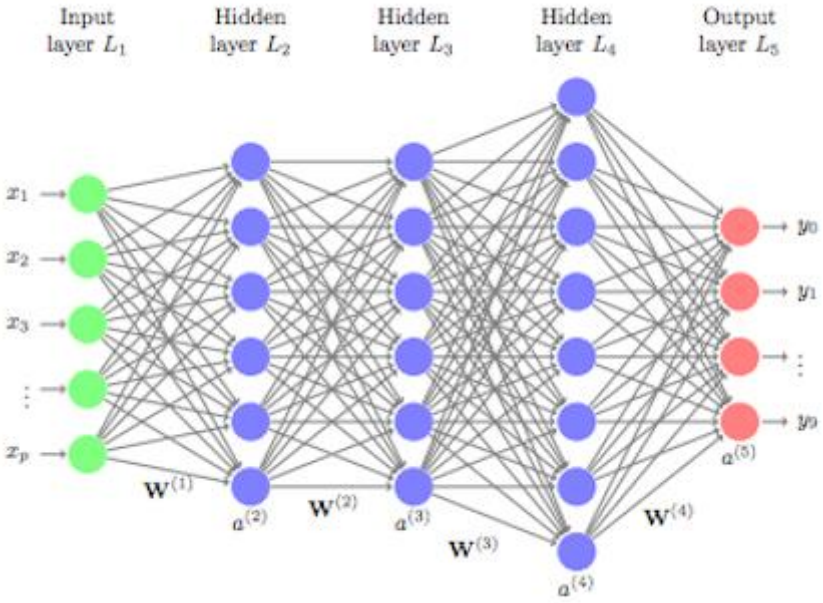
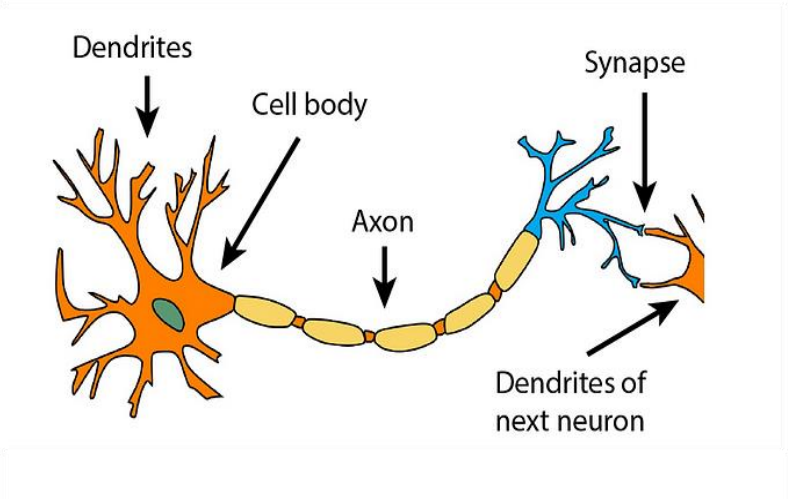
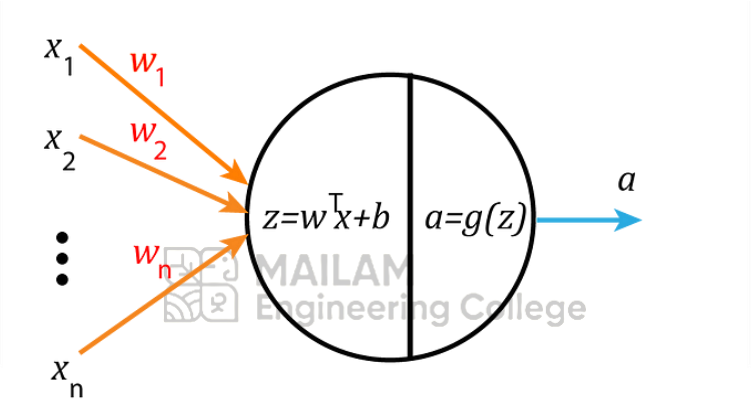


Figure 1.13: structure of Neural network

- Think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The idea of using many layers of vector-valued representation is drawn from neuroscience.
- An example of a feedforward network, drawn in two different styles. Refer figure 1.14.





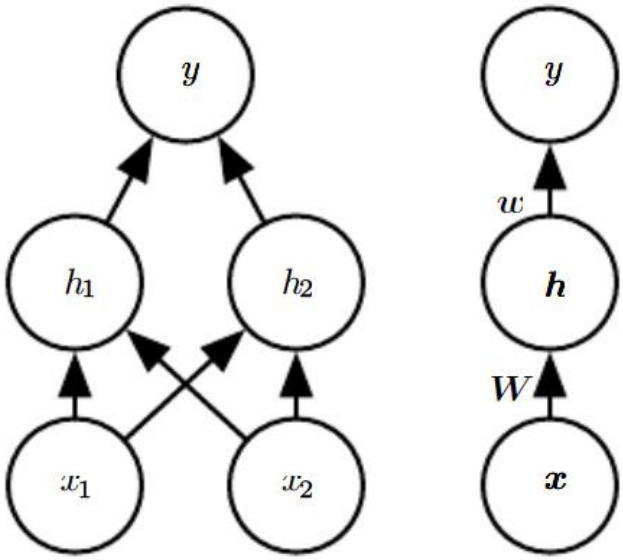


Figure 1.14: An example of a feedforward network, drawn in two different styles.

- Specifically, this is the feedforward network we use to solve the XOR example. It has a single hidden layer containing two units. *(Left)*In this style, we draw every unit as a node in the graph.
- This style is very explicit and unambiguous but for networks larger than this example it can consume too much space. *(Right)*In this style, we draw a node in the graph for each entire vector representing a layer's activations. This style is much more compact.
- Sometimes we annotate the edges in this graph with the name of the parameters that describe the relationship between two layers. Here, we indicate that a matrixW describes the mapping from x to h, and a vector w describes the mapping from h to y. We typically omit the intercept parameters associated with each layer when labeling this kind of drawing.

Activation Functions

Activation functions play a crucial role in feedforward neural networks. They introduce non-linear properties to the network, which allows the model to learn more complex patterns. Common activation functions include the sigmoid, tanh, and ReLU (Rectified Linear Unit).

1). **Linear Function**: –

- Equation: The equation for a linear function is y = ax, which is very much similar to the equation for a straight line.
- -inf to +inf range
- Applications: The linear activation function is only used once, in the output layer.

2) **The sigmoid function**:

- It's a function that is being plotted in the form of 'S' Shape.
- Formula: A = 1/(1 + e-x)
- ***Non-linear in nature. The values of X ranges from -2 to 2, but the Y values are highly steep. This indicates that slight changes*** in x will result in massive changes in Y's value. Refer figure 1.15.

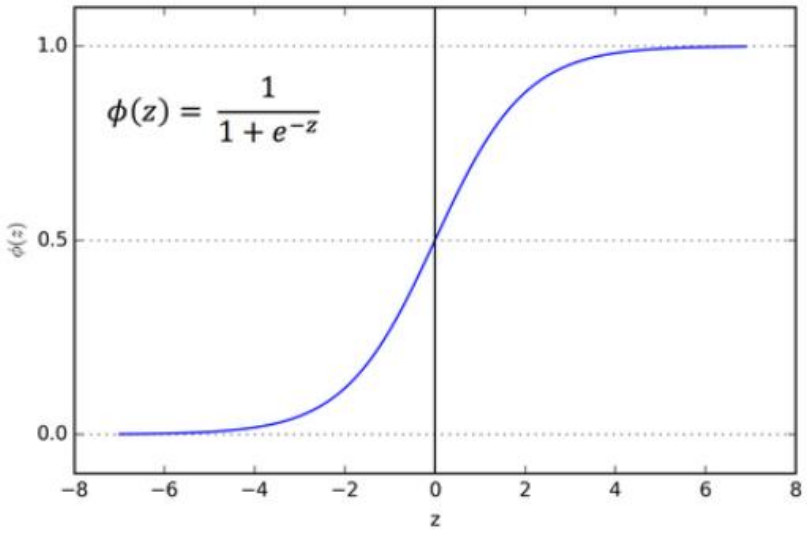


Figure 1.15:Representation of sigmoid function

3). **Tanh Function**:

- Tanh function, also identified as Tangent Hyperbolic function, is an activation that almost always works better than sigmoid function. ***It's simply a sigmoid function that has been adjusted. Both are related and can be deduced from one another.*** *Refer 1.16.*
- Equation: f(x) = tanh(x) = 2/(1 + e-2x) – 1 OR tanh(x) = 2 * sigmoid(2x) – 1 OR tanh(x) = 2 * sigmoid(2x) – 1

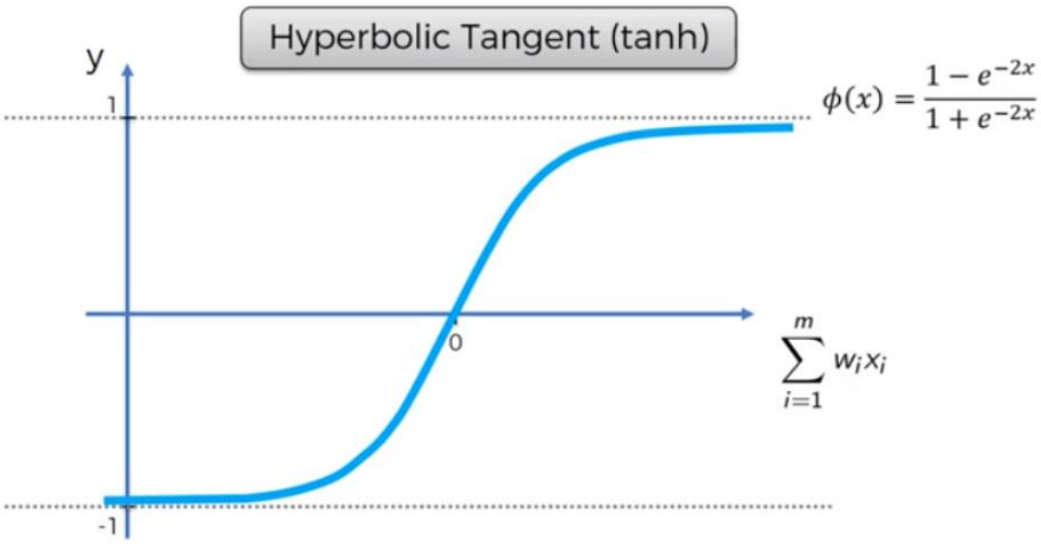


Figure 1.16: Representation of Tanh function

- **Range of values: -1 to +1**
- **Uses:-** ***Usually employed in hidden layers of a Neural Network since its values change from -1 to 1***, causing the hidden layer's mean to be 0

or very close to it, which aids in data centring by bringing the mean close to 0. This makes learning the next layer much more straight.

4). **RELU**

- Rectified linear unit(RELU) is the fourth letter in the alphabet. It's the most used activation method. Hidden layers of neural networks are primarily used.
- Formula: A(x) = maximum (0,x). If x is positive, it returns x; else, it returns 0.
- Value Range: (inf, 0)
- Non-linear in nature, which means simply backpropagating errors and also having the ReLU function activating many layers of neurons.
- **Applications:** ***Because it includes fewer mathematical operations, ReLu is less computationally expensive than tanh and sigmoid***. Only a few neurons are active at a time, making the network scarce and efficient for computation. Refer figure 1.17.
- Simply put, the RELU function learns much faster than the sigmoid and Tanh functions.

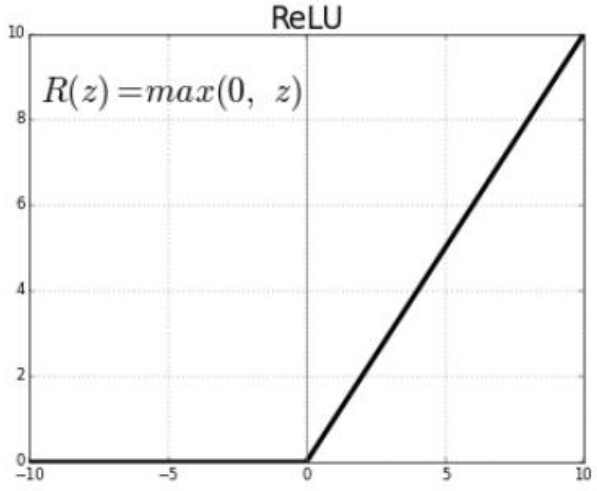


Figure 1.17:Representation of ReLU function

5). **Softmax Function**:

- The softmax function is a type of sigmoid function that comes in handy when dealing with categorization issues.
- Non-linearity in nature.
- Uses: ***Typically utilised when dealing with many classes. The softmax function would divide by the sum of the outputs and squeeze*** the outputs for each class between 0 and 1.

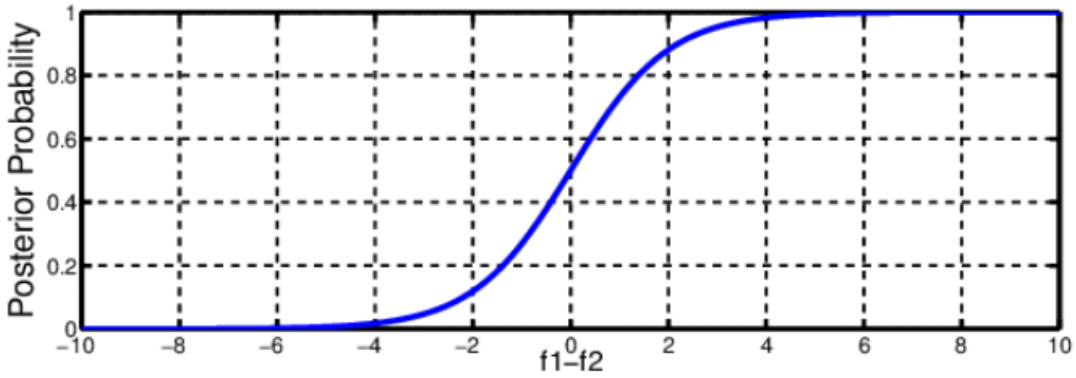


Figure 1.18:Representation of softmax function

- Output: The softmax function is best used in the classifier's output layer, where we're trying to define the class of each input using probabilities. Refer figure 1.18.

Selecting The Right Activation Function

- If one is unsure about the activation function to utilise, just select RELU, which is a broad activation function that is used in most circumstances these days.
- If our output layer is meant to be used for binary identification/detection, the sigmoid function is an obvious choice.

Applications of Feedforward Neural Networks

Feedforward neural networks are used in a variety of machine learning tasks including:

- Pattern recognition
- Classification tasks
- Regression analysis
- Image recognition
- Time series prediction

Challenges and Limitations

While feedforward neural networks are powerful, they come with their own set of challenges and limitations.

- One of the main challenges is the choice of the number of hidden layers and the number of neurons in each layer, which can significantly affect the performance of the network.
- Overfitting is another common issue where the network learns the training data too well, including the noise, and performs poorly on new, unseen data.

12. Explain in detail about Regularization.

Overfitting:

- When a model performs well on the training data and does not perform well on the testing data, then the model is said to have high generalization error. In other words, in such a scenario, the model has low bias and high variance and is too complex. This is called overfitting.
- Overfitting means that the model is a good fit on the train data compared to the data. Overfitting is also a result of the model being too complex

Regularization:

- Regularization is one of the most important concepts of machine learning. It is a technique to prevent the model from overfitting by adding extra information to it. Regularization helps choose a simple model rather than a complex one.
- Generalization error is "a measure of how accurately an algorithm can predict outcome values for previously unseen data." Regularization refers to the modifications that can be made to a leaming algorithm that helps to reduce this generalization error and not the training error.

The commonly used regularization techniques are:

1. Hints

- *Hints* are properties of the target function that are known to us independent of the training examples

Figure 1.19: **HINTS**

- The identity of the object does not change when it is translated, rotated, or scaled. Note that this may not always be true, or may be true up to a point: 'b' and 'q' are rotated versions of each other. These are hints that can be incorporated into the learning process to make learning easier.
- In image recognition, there are invariance hints: The identity of an object does not change when it is rotated, translated, or scaled (see **Figure 1.19).** Hints are auxiliary information that can be used to guide the learning process and are especially useful when the training set is limited.
- There are different ways in which hints can be used:
  - ✓ Hints can be used to create *virtual examples*.
  - ✓ The hint may be incorporated into the network structure.

2. Weight decay:

- Incentivize the network to use smaller weights by adding a penalty to the loss function.
- Even if we start with a weight close to zero, because of some noisy instances, it may move away from zero; the idea in *weight decay* is to add some small constant background force that always pulls a weight toward zero, unless it is absolutely necessary that it be large (in magnitude) to decrease error. For any weight *wi*, the update rule is

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} - \lambda' w_i$$

3. Ridge Regression

- The Ridge regression technique is used to analyze the model where the variables may be having multicollinearity.
- It reduces the insignificant independent variables though it does not remove them completely. This type of regularization uses the L_2 norm for regularization.

$$E' = E + \frac{\lambda}{2} \sum_i w_i^2$$

4. Lasso Regression

- Least Absolute Shrinkage and Selection Operator (or LASSO) Regression penalizes the coefficients to the extent that it becomes zero. It eliminates the insignificant independent variables. This regularization technique uses the L1 norm for regularization.

$$E' = E + \frac{\lambda}{2} \sum_i |w_i|$$

5. Dropout

- "**Dropout**" in machine learning refers to the process of randomly ignoring certain nodes in a layer during training.
- In the **Figure 1.20**, the neural network on the left represents a typical neural network where all units are activated. On the right, the red units have been dropped out of the model- the values of their weights and biases are not considered during training.

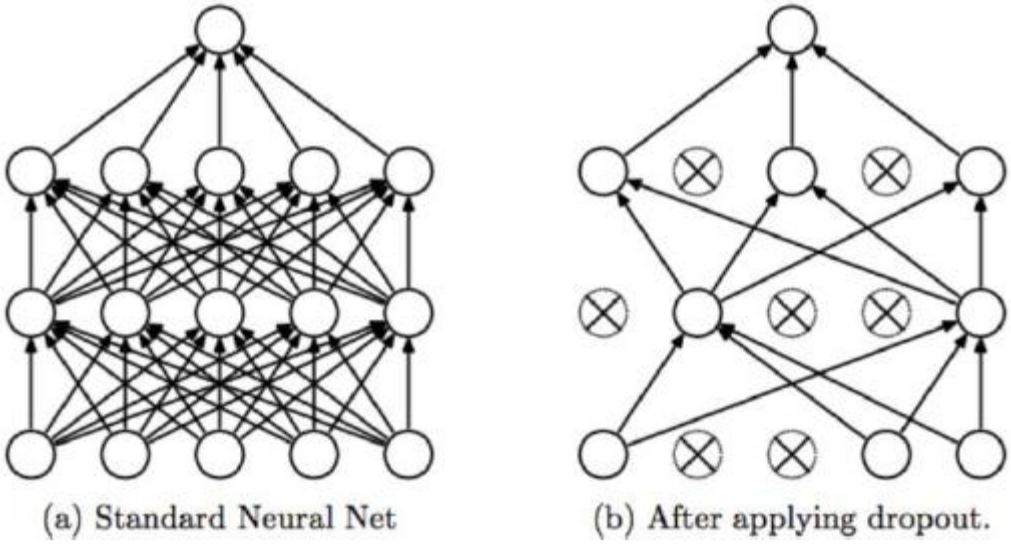


Figure 1.20: **Dropout**

- Dropout is used as a regularization technique - it prevents overfitting by ensuring that no units are codependent.
- In *dropout*, we have a hyperparameter p, and we drop the input or hidden unit with probability p, that is, set its output to zero, or keep it with probability $1 - p$.
- p is adjusted using cross-validation; with more inputs or hidden units in a layer, we can afford a larger p (see **Figure 1.21**).

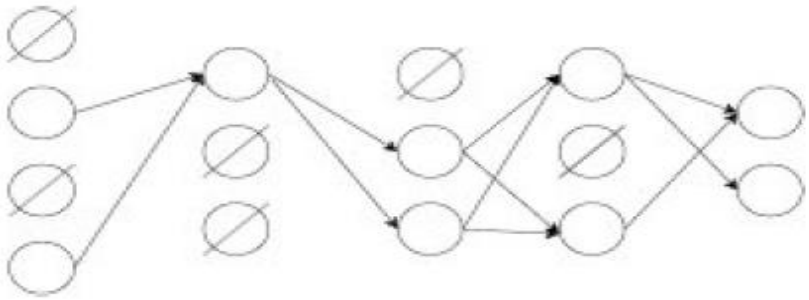


Figure 1.21 In dropout, the output of a random subset of the units are set to zero, and backpropagation is done on the remaining smaller network.

- In each batch or minibatch, for each unit independently we decide randomly to keep it or not. Let us say that $p = 0.25$. So, on average, we remove a quarter of the units and we do backpropagation as usual on the remaining network for that batch or minibatch. We need to make up for the loss of units, though: In each layer, we divide the activation of the remaining units by $1 - p$ to make sure that they provide a vector of similar magnitude to the next layer. There is no dropout during testing.
- In each batch or minibatch, a smaller network (with smaller variance) is trained. Thus dropout is effectively sampling from a pool of possible networks of different depths and widths.

- There is a version called *dropconnect* that drops out or not connections independently, which allows a larger set of possible networks to sample from, and this may be preferable in smaller networks.

UNIT II CONVOLUTIONAL NEURAL NETWORKS

SYLLABUS:

Convolution Operation -- Sparse Interactions -- Parameter Sharing -- Equivariance -- Pooling -- Convolution Variants: Strided -- Tiled -- Transposed and dilated convolutions; CNN Learning: Nonlinearity Functions -- Loss Functions -- Regularization -- Optimizers --Gradient Computation.

PART A

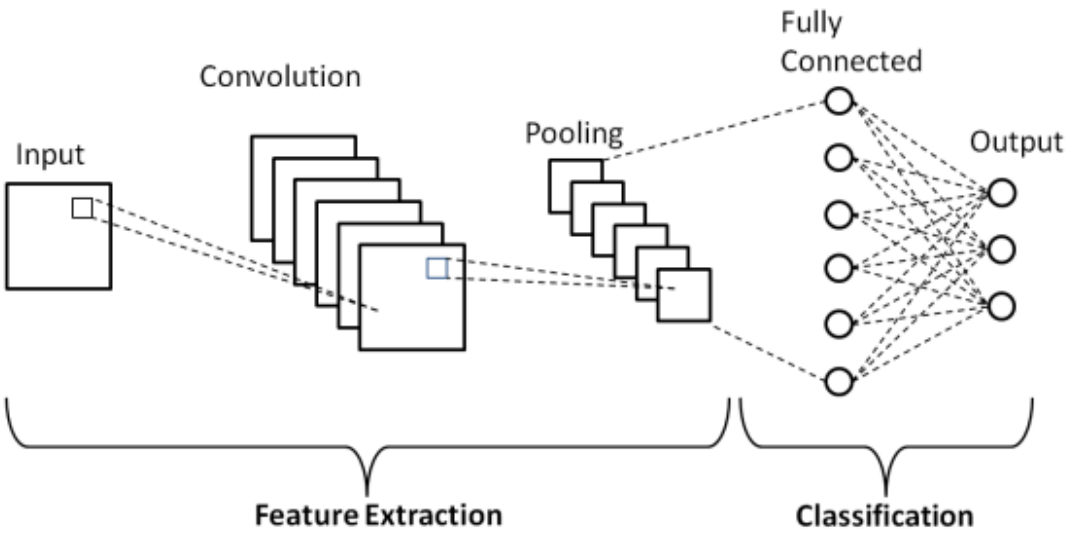
1. What is Convolution?

- Convolution is the process in which each element of the image is added to its local neighbours, and then it is weighted by the kernel.
- It is related to a form of mathematical convolution.
- In Convolution, the matrix does not perform traditional matrix multiplication but it is denoted by *.

2. What is convolution network?

- Convolutional neural networks are designed to work with grid-structured inputs, which have strong spatial dependencies in local regions of the grid.
- Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

3. Draw the architecture of CNN.



4. What is Convolution Operation?

The convolution operation is the process of implying a combination of two functions that produce the third function as a result, employing filters across the entire input image allows the filter to discover that feature of the image. Which is also called a feature map.

5. **Define Sparse Interactions.**
   - Sparse interaction or sparse weights is implemented by using kernels or feature detector smaller than the input image.
   - In traditional Neural Networks, **every output unit interacts with every input unit.**

6. **Define Parameter Sharing.**
   - Parameter sharing refers to **using same parameter for more than one function in a model.**
   - In a traditional neural net, each element of the neural network has its own weight, i.e. each element of the weight matrix is used only once while computing the output of a layer. It is multiplied by one element of the input and then never revisited.

7. **Define Equivariance Representation.**
   - **Parameter sharing in a convolutional network provides equivariance to translation.**
   - Translation of image results in corresponding translation in the output map.
   - Convolution operation by itself is **not equivariant to changes in scale or rotation.**

8. **Write the function for Equivariance Representation.**

> *A function* **f** *is said to be equivariant to a function* **g** *if*
>
> $f(g(x)) = g(f(x))$
>
> *i.e. if input changes, the output changes in the same way.*

9. **What are the 3 layers convolution layer?**

A convolution layer consists of **3 layers -**

a. Convolution
b. Activation (Detector Stage)
c. Pooling

10. What is Pooling?

A pooling function **replaces the output** of net at a certain location with **summary statistic of nearby outputs.**

- Pooling **reduces input size to the next layer** in turn reducing the number of computations required upstream.
- Pooling involves down sampling feature maps, reducing spatial dimensions and computational load.

11. What are the Parameters that define a convolutional layer?

Kernel Size: The kernel size defines the field of view of the convolution. A common choice for 2D is 3 — that is 3x3 pixels.

Stride: The stride defines the step size of the kernel when traversing the image. While its default is usually 1, we can use a stride of 2 for downsampling an image similar to MaxPooling.

Padding: The padding defines how the border of a sample is handled. A (half) padded convolution will keep the spatial output dimensions equal to the input, whereas unpadded convolutions will crop away some of the borders if the kernel is larger than 1.

Input & Output Channels: A convolutional layer takes a certain number of input channels (I) and calculates a specific number of output channels (O). The needed parameters for such a layer can be calculated by I*O*K, where K equals the number of values in the kernel.

12. What is Tiled convolution?

Tiled convolution is a sort of middle step between locally connected layer and traditional convolution. It uses a set of kernels that are cycled through. This reduces the number of parameters in the model while allowing for some freedom provided by unshared convolution.

13. What is Dilated Convolutions (a.k.a. atrous convolutions)?

Dilated convolutions introduce another parameter to convolutional layers called the dilation rate. This defines a spacing between the values in a kernel.

A 3x3 kernel with a dilation rate of 2 will have the same field of view as a 5x5 kernel, while only using 9 parameters. Imagine taking a 5x5 kernel and deleting every second column and row.

14. What is Activation function?

Artificial neurons are elementary units in an artificial neural network. The artificial neuron receives one or more inputs and sums them to produce an output. Each input is separately weighted, and the sum is passed through a function known as an activation function or transfer function.

15. What is the Need for Non-linear activation function?

A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

16. What is Loss Functions in CNN learning?

- Some loss functions are utilized in the output layer to calculate the predicted error created across the training samples in the CNN model.
- This error reveals the difference between the actual output and the predicted one.

17. List some of the loss function types.

- Cross-Entropy or Softmax Loss Function
- Euclidean Loss Function
- Hinge Loss Function

18. How to Improve the performance of CNN?

Improving performance of CNN

- Based on our experiments in different DL applications can conclude the most active solutions that may improve the performance of CNN are:
    - Expand the dataset with data augmentation or use transfer learning
    - Increase the training time.
    - Increase the depth (or width) of the model.
    - Add regularization.

- Increase hyperparameters tuning.

PART B

1. Explain in detail about Convolution Operation.

Convolution

- Convolution is the process in which each element of the image is added to its local neighbours, and then it is weighted by the kernel.
- It is related to a form of mathematical convolution.
- In Convolution, the matrix does not perform traditional matrix multiplication but it is denoted by *****.

Convolutional networks

- Convolutional neural networks are designed to work with grid-structured inputs, which have strong spatial dependencies in local regions of the grid.
- Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.
- Refer figure 2.1 for CNN architecture.

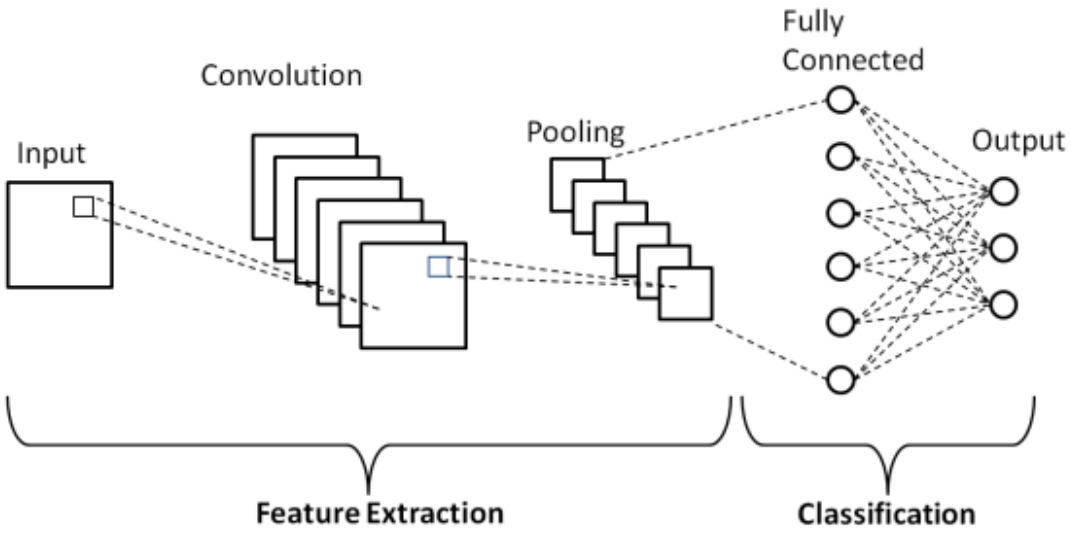


Figure 2.1 CNN Architecture

Layers used to build ConvNets

A complete Convolution Neural Networks architecture is also known as covnets. A covnets is a sequence of layers, and every layer transforms one volume to another through a differentiable function.

Types of layers: datasets Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

- **Input Layers:** It's the layer in which we give input to our model. In CNN, Generally, the input will be an image or a sequence of images. This layer holds the raw input of the image with width 32, height 32, and depth 3.
- **Convolutional Layers:** This is the layer, which is used to extract the feature from the input dataset. It applies a set of learnable filters known as the kernels to the input images. The filters/kernels are smaller matrices usually 2×2, 3×3, or 5×5 shape. it slides over the input image data and computes the dot product between kernel weight and the corresponding input image patch. The output of this layer is referred as feature maps. Suppose we use a total of 12 filters for this layer we'll get an output volume of dimension 32 x 32 x 12.
- **Activation Layer:** By adding an activation function to the output of the preceding layer, activation layers add nonlinearity to the network. it will apply an element-wise activation function to the output of the convolution layer. Some common activation functions are **RELU**: max(0, x), **Tanh**, **Leaky RELU**, etc. The volume remains unchanged hence output volume will have dimensions 32 x 32 x 12.
- **Pooling layer:** This layer is periodically inserted in the covnets and its main function is to reduce the size of volume which makes the computation fast reduces memory and also prevents overfitting. Two common types of pooling layers are **max pooling** and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.
- **Flattening:** The resulting feature maps are flattened into a one-dimensional vector after the convolution and pooling layers so they can be passed into a completely linked layer for categorization or regression.
- **Fully Connected Layers:** It takes the input from the previous layer and computes the final classification or regression task.

Convolution Operation

- The convolution operation is the process of implying a combination of two functions that produce the third function as a result, employing filters across the entire input image allows the filter to discover that feature of the image. Which is also called a feature map.
- Convolution is an operation on two function of a real valued argument.
- The conv operation is usually denoted with an asterisk
- In the case of indiscrete value

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

- In the case of dsicrete value

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Note: w needs to be 0 for all negative arguments

x: input

w: kernel

s: feature map

In convolutional network terminology, the **first argument** to the convolution is often referred to as the **input** and the **second argument** as the **kernel**. The output is sometimes referred to as the **feature map. Refer figure 2.2.**

For 2D input and 2D kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

It is commutative, so we can also right

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Cross-coorelation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Many machine learning libraries implement cross correlations but call it convolution.

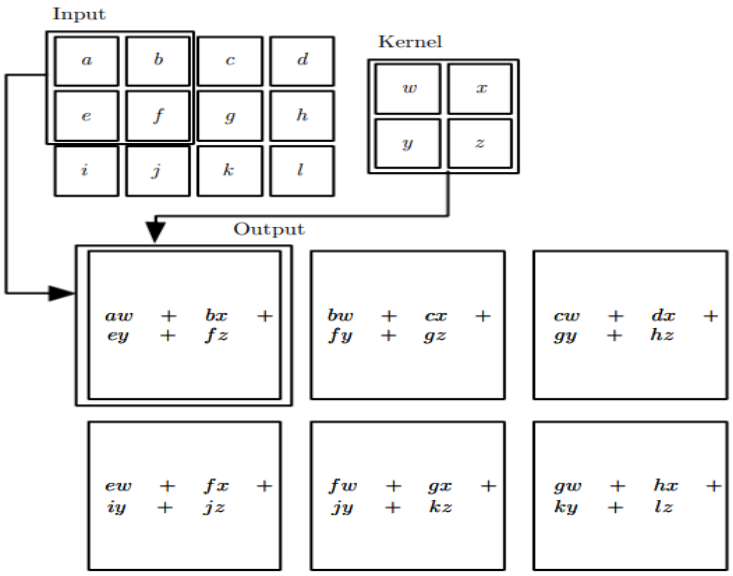


Figure 2.2 process of convolution operation

Example:

```
#Importing Some Relevant Libraries
import NumPy as np
%matplotlib inline
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import TensorFlow as tf
tf.compat.v1.set_random_seed(2019)
#Loading the MNIST Dataset
(X_train,Y_train),(X_test,Y_test) = keras.datasets.mnist.load_data()
#Scaling our Data
X_train = X_train / 255
X_test = X_test / 255
#flatenning
X_train_flattened = X_train.reshape(len(X_train), 28*28)
X_test_flattened = X_test.reshape(len(X_test), 28*28)
#Designing Neural Network
model = keras.Sequential([
keras.layers.Dense(10, input_shape=(784,), activation='sigmoid')])
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(X_train_flattened, Y_train, epochs=5)
```

Output

```
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.7187 -
accuracy: 0.8141
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3122 -
accuracy: 0.9128
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2908 -
accuracy: 0.9187
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2783 -
accuracy: 0.9229
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2643 -
accuracy: 0.9262
```

2. **Describe in detail about Sparse Interactions, Parameter Sharing, Equivariance, Pooling.**

The convolution neural networks are suitable for image processing as they take into account the neighboring pixels, as for images the neighboring pixels of a pixel get a say in defining it. Convolution neural networks leverage important ideas that can help improve machine learning systems:

i. **Sparse interactions**
ii. **Parameter sharing**
iii. **Equivariant representations**.
iv. **Pooling**

Sparse Interactions

- Sparse interaction or sparse weights is implemented by using kernels or feature detector smaller than the input image.
- In traditional Neural Networks, **every output unit interacts with every input unit.**
- Convolutional networks, however, typically have **sparse interactions,** by making **kernel smaller than input.**
    i. Reduces memory requirements
    ii. Improves statistical efficiency
- In a deep convolutional network, units in the deeper layers may **indirectly interact** with a larger portion of the input.

- For example, if we have an input image of the size 256 by 256 then it becomes difficult to detect edges in the image may occupy only a smaller subset of pixels in the image. If we use smaller feature detectors then we can easily identify the edges as we focus on the local feature identification. Refer figure 2.3.

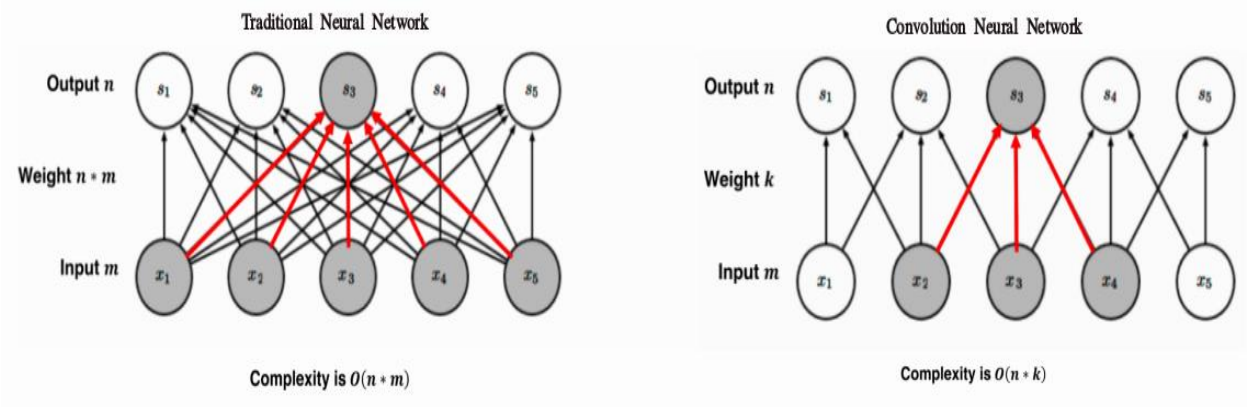


Figure 2.3 a. When s is formed by convolution with a kernel of width 3, only three inputs affect s3. b. When s is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect s3.

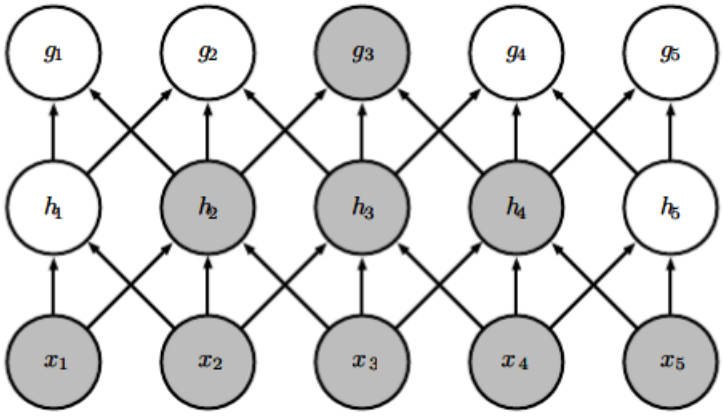


Figure 2.4 The receptive field of the units in the deeper layers

- The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. Refer figure 2.3.
- This effect increases if the network includes architectural features like strided convolution (figure 2.4) or pooling.
- This means that even though direct connections in a convolutional net are very sparse, units in the deeper layers can be indirectly connected to all or most of the input image.

Parameter Sharing

- Parameter sharing refers to **using same parameter for more than one function in a model.**
- In a traditional neural net, each element of the neural network has its own weight, i.e. each element of the weight matrix is used only once while computing the output of a layer. It is multiplied by one element of the input and then never revisited.
- It is multiplied by one element of the input and then never revisited.
- As a synonym for parameter sharing, one can say that a network has **tied weights**, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere.
- In convolutional neural net, **each member of kernel** is used at **every position of input** i.e. parameters used to compute different output units are **tied together** (all times their values are same).
- **Sparse interactions and parameter sharing combined** can improve efficiency of a linear function for **detecting edges** in an image.
- The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation—it is still $O(k \times n)$ but it does further reduce the storage requirements of the model to k parameters. Refer figure 2.6.
- Recall that k is usually several orders of magnitude less than m. Since m and n are usually roughly the same size, k is practically insignificant compared to m ×n.
- Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, see figure 2.5

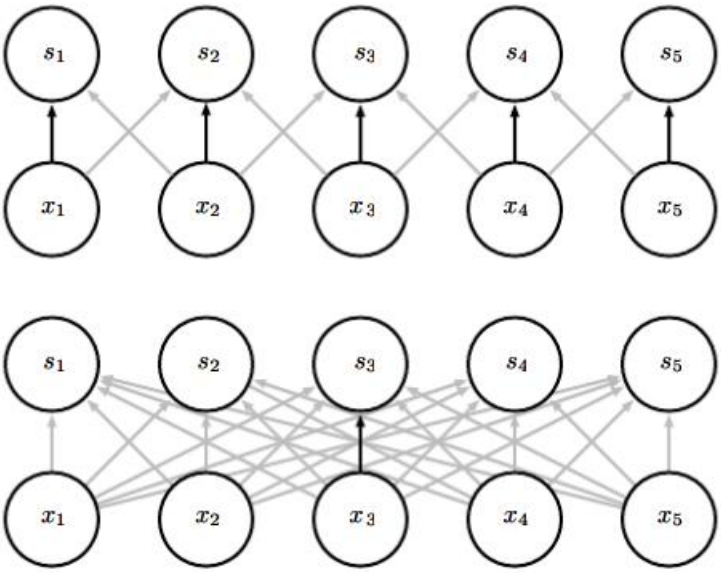


Figure 2.5 Parameter sharing

- Black arrows indicate the connections that use a particular parameter in two different models.
- Top-The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations.
- Bottom-The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

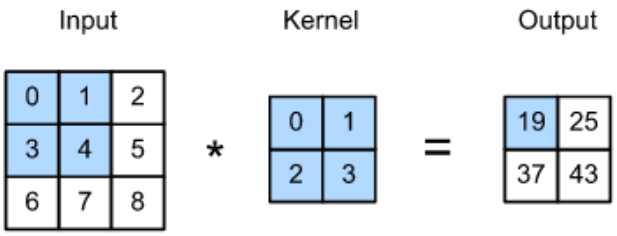


Figure 2.6 2*2 Parameter sharing

- We can also say that the convolution networks have tied weights because the weights applied to one input are tied to the value of the weight applied elsewhere. This means that rather than learning a different set of parameters for each location, we use only one set.

Equivariance Representation

- **Parameter sharing in a convolutional network provides equivariance to translation.**
- Translation of image results in corresponding translation in the output map.
- Convolution operation by itself is **not equivariant to changes in scale or rotation.**

> *A function* **f** *is said to be equivariant to a function* **g** *if*
>
> $f(g(x)) = g(f(x))$
>
> *i.e. if input changes, the output changes in the same way.*

- In convolutions, the particular case of parameter sharing causes the networks to have a property called equivariance. To say a function is equivariant means that if the input changes, the output changes in the same way.
- When processing time series data, this means that convolution produces a sort of timeline of when different features appear in the input. If we move an event later in time, the exact same representation of it will appear in the output, just later in time.

- Similarly, for images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, it will move by the same amount in the output. This is useful when we know that some function of the same number of pixels is useful when applied to multiple input locations. Refer figure 2.7.

b. Example: represent(rose) = represent(transform(rose)).

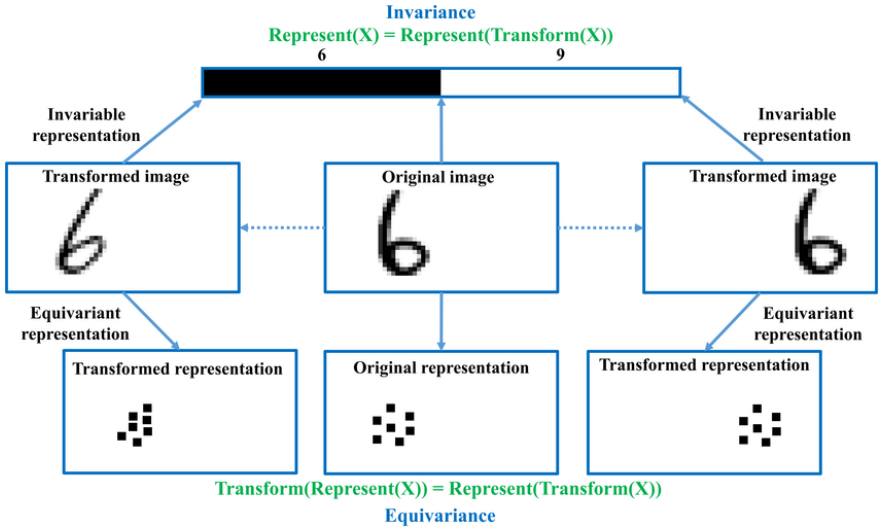


Figure 2.7 represent(X) = represent(transform(X)).

Pooling:

- A convolution layer consists of **3 layers -**
  - Convolution
  - Activation (Detector Stage)
  - Pooling
- A pooling function **replaces the output** of net at a certain location with **summary statistic of nearby outputs.**
- Common summary statistics are: **mean, median, weighted average.**
- Pooling makes the representation slightly **translation invariant**, in that **small translations** in the input **do not cause large changes in output map.**
- It allows detection of a particular feature **if we only care about its existence**, not its position in an image.
- Pooling **reduces input size to the next layer** in turn reducing the number of computations required upstream.
  - Pooling involves down sampling feature maps, reducing spatial dimensions and computational load.
  - Max pooling, for instance, extracts the maximum value from local regions, preserving dominant features.
  - Pooling is crucial for retaining essential information while enhancing computational efficiency.
  - In image classification, max pooling reduces the dimensionality, focusing on the most relevant features. Refer figure 2.8.

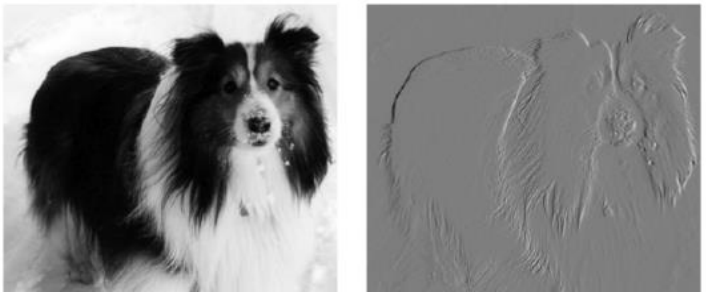


Figure 2.8 Efficiency of edge detection.

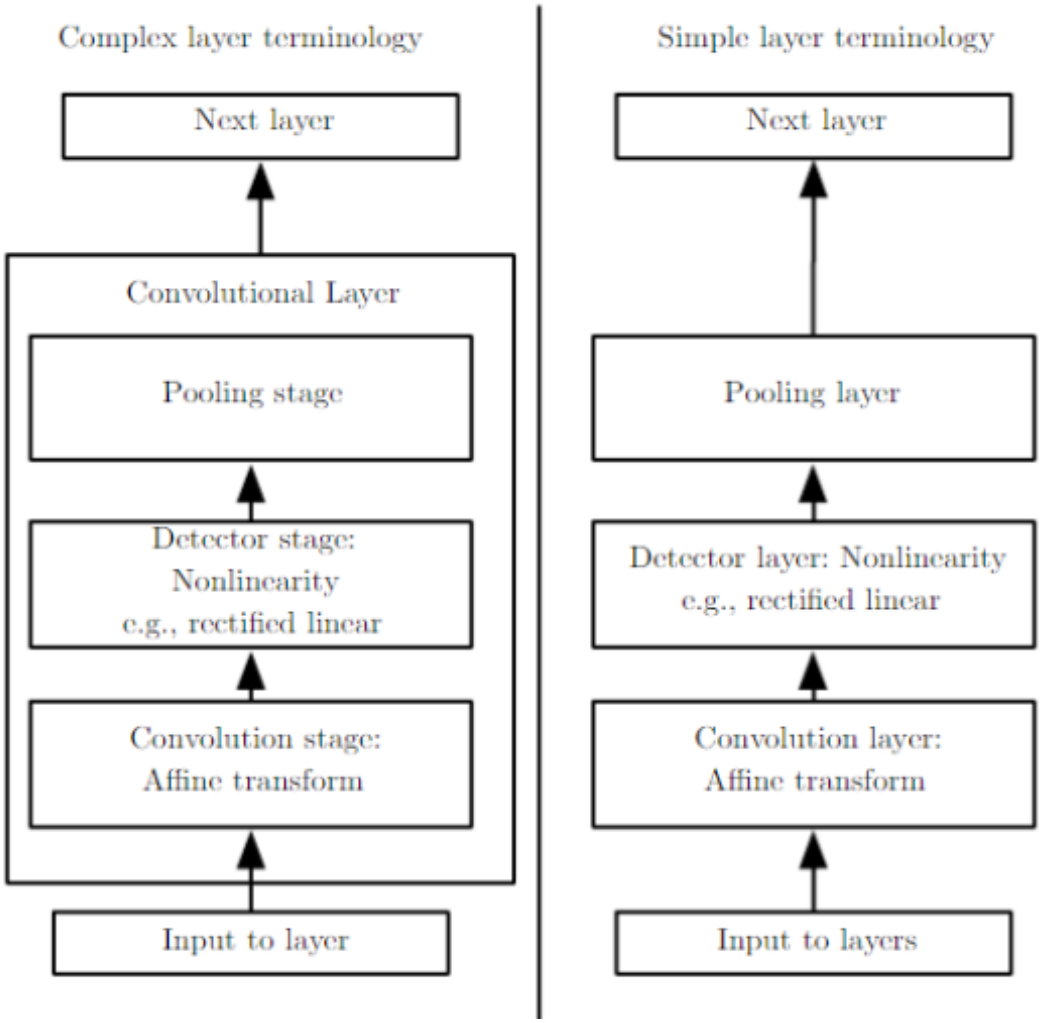


Figure 2.9 The components of a typical convolution Neural Network layer

- In all cases, pooling helps to make the representation become approximately **invariant** to small translations of the input. **Invariance** to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Example of invariance refer figure 2.10.
- **Invariance** to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.
- The components of a typical convolution Neural Network layer. Refer figure 2.9.

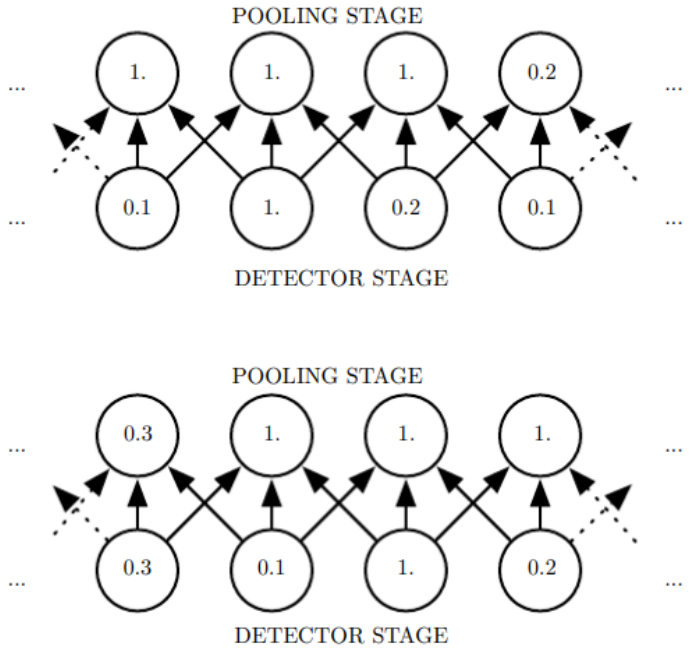


Figure 2.10 Max pooling introduces invariance.

- The top row shows the outputs of **max pooling**, with a stride of one pixel between pooling regions and a pooling region width of three pixels. Refer figure 2.10.
- (Bottom)A view of the same network, after the input has been shifted to the right by one pixel.
- Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.
- Example of learned invariances refer figure 2.11.

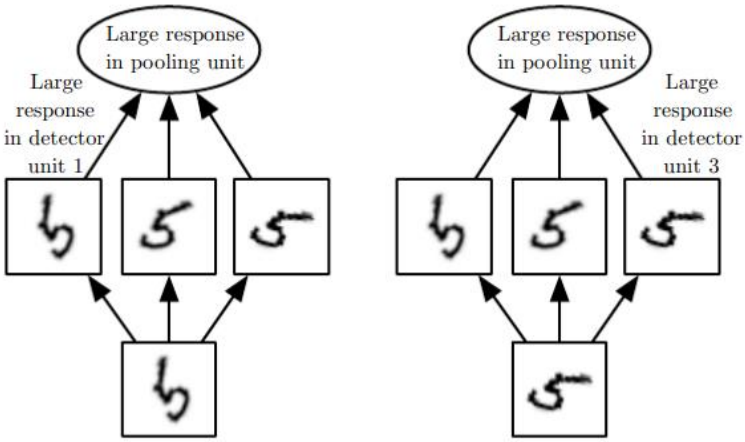


Figure 2.11 Example of learned invariances

- All three filters are intended to detect a hand-written 5.
- Each filter attempts to match a slightly different orientation of the 5.
- When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit.
- The max pooling unit then has a large activation regardless of which detector unit was activated. Refer figure 2.12.

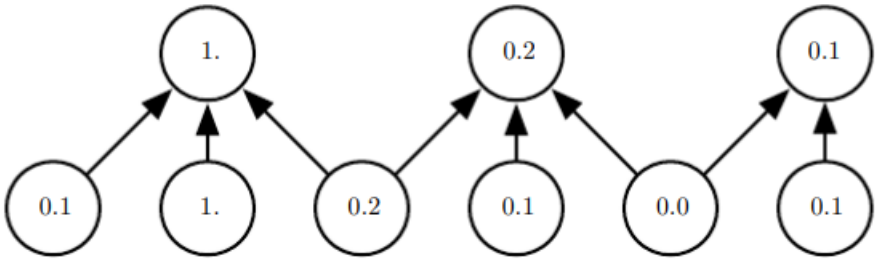


Figure 2.12 Pooling with downsampling.

- Here we use **max-pooling** with a pool width of three and a stride between pools of two.
- This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer.
- Note that the rightmost pooling region has a smaller size, but must be included if we do not want to ignore some of the detector units.

Convolution and Pooling as an Infinity Strong Prior:

Weight Prior

Assumptions about weights (before learning) in terms of acceptable values and range are encoded into the **prior distribution** of weights.

- Convolution imposes an **infinitely strong prior** by making the following **restrictions on weights:**
  - **Adjacent units** must have the **same weight** but shifted in space.
  - Except for a **small spatially connected** region, all **other weights** must be **zero.**
- Features should be **translation invariant.**
- If tasks relies on preserving specific spatial information, then pooling can cause on all features can increase training error.
- Refer Figure 2.13 for Examples of architectures for classification with convolutional networks.

S.No.	Prior Type	Variance/Confidence Type
1.	Weak	High Variance, Low Confidence
2.	Strong	**Narrow range** of values about which we are **confident** before learning begins.
3.	Infinitely strong	Demarkates certain values as **forbidden** completely assigning them **zero probability.**

Table 2.1 Types of Variance and its prior type

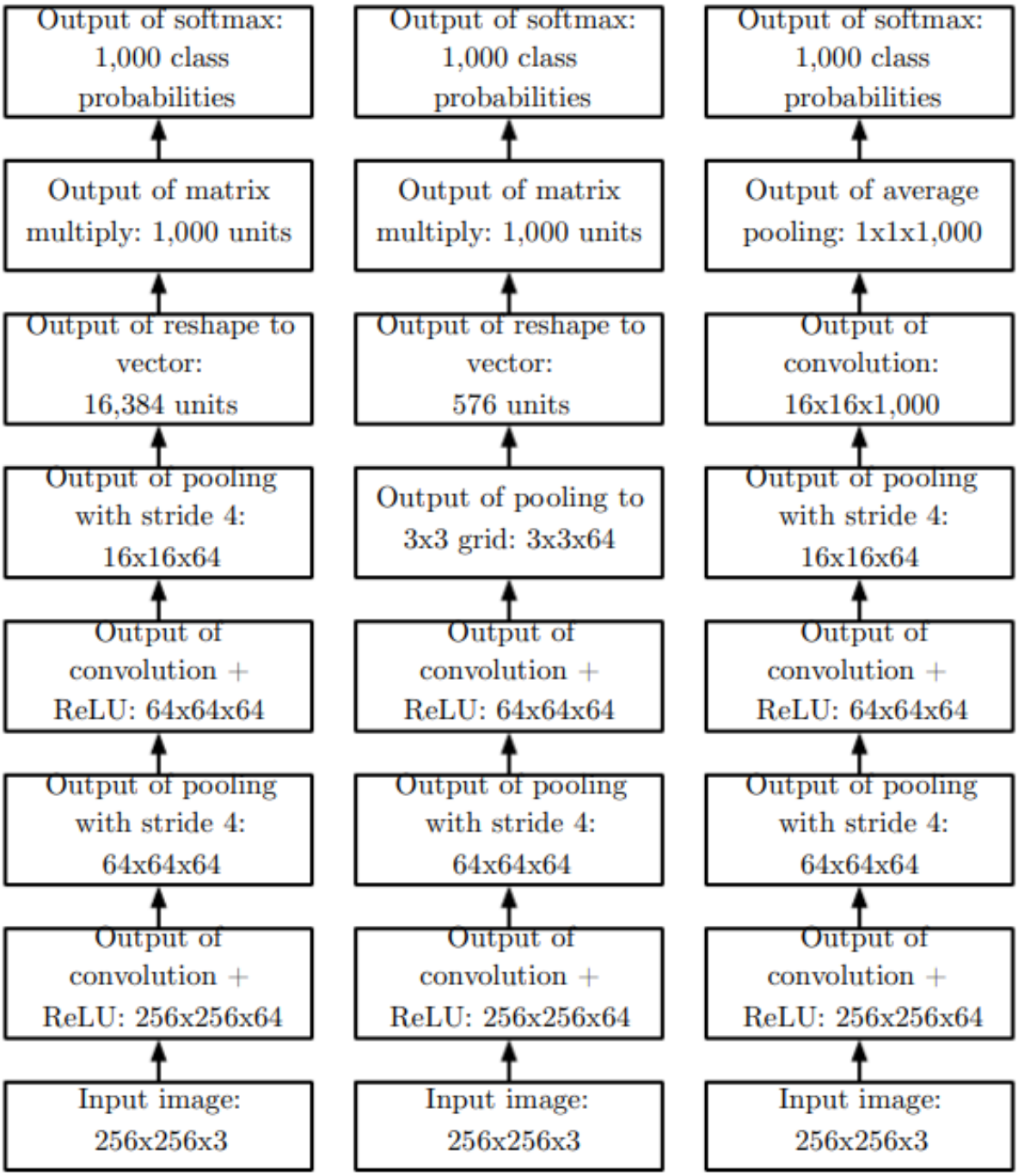


Figure 2.13 Examples of architectures for classification with convolutional networks.

Variants of Basic Convolution Function:

In practical implementations of the convolution operation, certain modifications are made which deviate from standard discrete convolution operation -

- In general, a convolution layer consists of application of **several different kernels** to the input. Since, convolution with a **single kernel can extract only one kind of feature.**
- The input is generally not real-valued but instead **vector valued.**
- Multi-channel convolutions are commutative if **number of output and input channels is the same.**

- **K=** 4D kernel tensor with elements $K_{i,l,m,n}$ giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the input unit.
- **V=** Assume our input consists of observed data V with element $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k.
- **Z=** Assume our output consists of Z with the same format as V.
- If **Z** is produced by convolving **K across V without flipping K**, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n} \qquad (9.7)$$

Efficient Convolution Algorithms:

Efficient algorithms, such as the **Fast Fourier Transform (FFT)** for convolution, optimize computation. FFT reduces the complexity of convolution operations, accelerating training and inference. In audio signal processing, efficient convolution algorithms enhance the speed and effectiveness of feature extraction.

Example:

Consider an image classification task using a convolutional neural network (CNN). Apply parameter sharing and pooling to efficiently capture features while reducing computation. Visualize the impact on model efficiency and performance by comparing with a non-convolutional approach.

The convolution operation stands as a pillar in deep learning, wielding sparse interactions, parameter sharing, pooling, and efficient algorithms. Understanding these components and their applications illuminates the path to optimized feature extraction and model efficiency, revolutionizing tasks across various domains. As we

continue to unravel the complexities, convolution remains a driving force in the evolution of deep learning.

3. Discuss in detail about Convolution Neural Network Variants. (or) What is Convolution Neural Network? Outline transposed and dilated convolution with example. (NOV/DEC 2023)

Parameters that define a convolutional layer:

Kernel Size: The kernel size defines the field of view of the convolution. A common choice for 2D is 3 — that is 3x3 pixels.

Stride: The stride defines the step size of the kernel when traversing the image. While its default is usually 1, we can use a stride of 2 for downsampling an image similar to MaxPooling.

Padding: The padding defines how the border of a sample is handled. A (half) padded convolution will keep the spatial output dimensions equal to the input, whereas unpadded convolutions will crop away some of the borders if the kernel is larger than 1.

Input & Output Channels: A convolutional layer takes a certain number of input channels (I) and calculates a specific number of output channels (O). The needed parameters for such a layer can be calculated by I*O*K, where K equals the number of values in the kernel.

Stride

- When computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right.
- The number of rows and columns traversed per slide as *stride*. So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride.
- A two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally.
- The shaded portions are the output elements as well as the input and kernel tensor elements used for the output computation:

  0×0+0×1+1×2+2×3=8, 0×0+6×1+0×2+0×3=6.
- We can see that when the second element of the first column is generated, the convolution window slides down three rows.
- The convolution window slides two columns to the right when the second element of the first row is generated.

- When the convolution window continues to slide two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

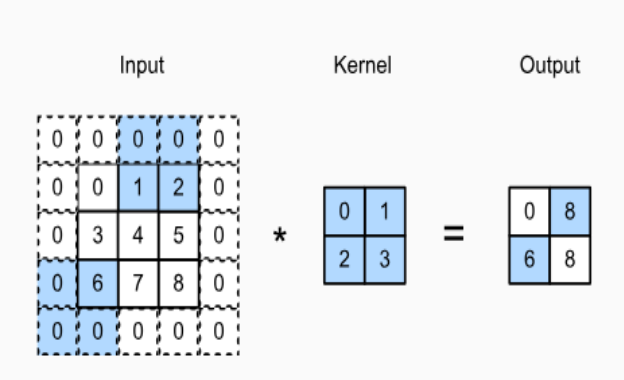


Figure 2.7 Cross-connection with strides of 3 and 2 for height and width, respectively.

- In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor.$$

- If we set $p_h = k_h - 1$ and $p_w = k_w - 1$, then the output shape can be simplified to $\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$.
- Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_h/s_h) \times (n_w/s_w)$.
- Below, we set the strides on both the height and width to 2, thus halving the input height and width.

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape

#Sytax:
torch.Size([4, 4])
```

Tiled convolution

- Tiled convolution is a sort of middle step between locally connected layer and traditional convolution. It uses a set of kernels that are cycled through. This reduces the number of parameters in the model while allowing for some freedom provided by unshared convolution.
- Learn a set of kernels that we rotate through as we move through space. Immediately neighboring locations will have different filters, but the memory

requirement for storing the parameters will increase by a factor of the size of this set of kernels. Comparison on locally connected layers, tiled convolution and standard convolution:

K: 6-D tensor, t different choice of kernel stack

$$Z_{i,j,k} = \sum_{l,m,n} [V_{i,i+m-1,j+n-1} K_{i,l,m,n,j\%t+1,k\%t+1}]$$

- Local connected layers and tiled convolutional layer with max pooling: the detector units of these layers are driven by different filters. If the filters learn to detect different transformed version of the same underlying features, then the max-pooled units become invariant to the learned transformation.

- Refer Figure 2.8 for Comparison of locally connected layers, tiled convolution and standard convolution

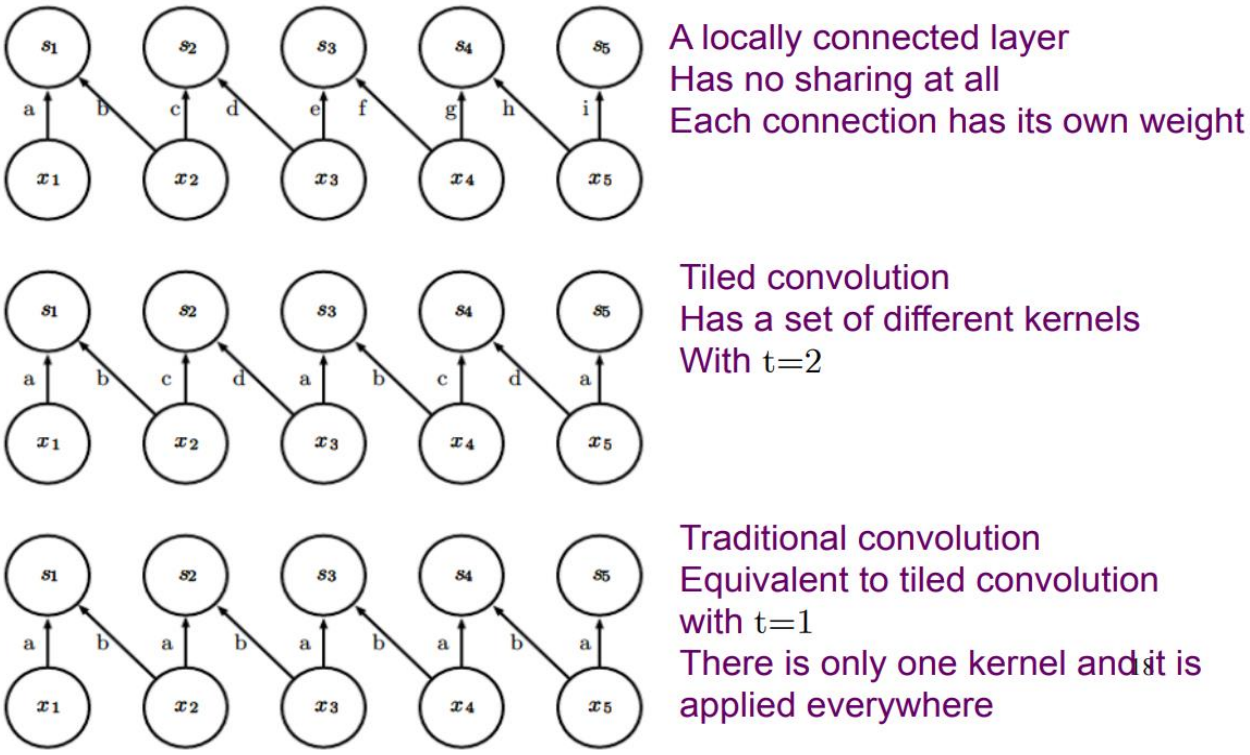


Figure 2.8 Comparison of locally connected layers, tiled convolution and standard convolution

- The parameter complexity and computation complexity can be obtained as below. Note that:
- ***m* = number of input units**
- ***n* = number of output units**
- ***k* = kernel size**

- ***l* = number of kernels in the set (for tiled convolution)**

Type	Computations	Parameters
Fully connected	***O(mn)***	***O(mn)***
Locally connected	***O(kn)***	***O(kn)***
Tiled	***O(kn)***	***O(kl)***
Traditional	***O(kn)***	***O(k)***

- You can see now that the quantity of ~451 thousand parameters correspond to the locally connected convolution operation.
- If we use a set of 200 kernels, the number of parameters for tiled convolution is 1.8 thousand. For a traditional convolution operation, this number is 9 parameters.

Dilated Convolutions (a.k.a. atrous convolutions)

- Dilated convolutions introduce another parameter to convolutional layers called the dilation rate. This defines a spacing between the values in a kernel.
- A 3x3 kernel with a dilation rate of 2 will have the same field of view as a 5x5 kernel, while only using 9 parameters. Imagine taking a 5x5 kernel and deleting every second column and row.
- This delivers a wider field of view at the same computational cost.
- Dilated convolutions are particularly popular in the field of real-time segmentation.
- Use them if you need a wide field of view and cannot afford multiple convolutions or larger kernels.
- Refer figure 2.9 2D convolution using a 3 kernel with a dilation rate of 2 and no padding

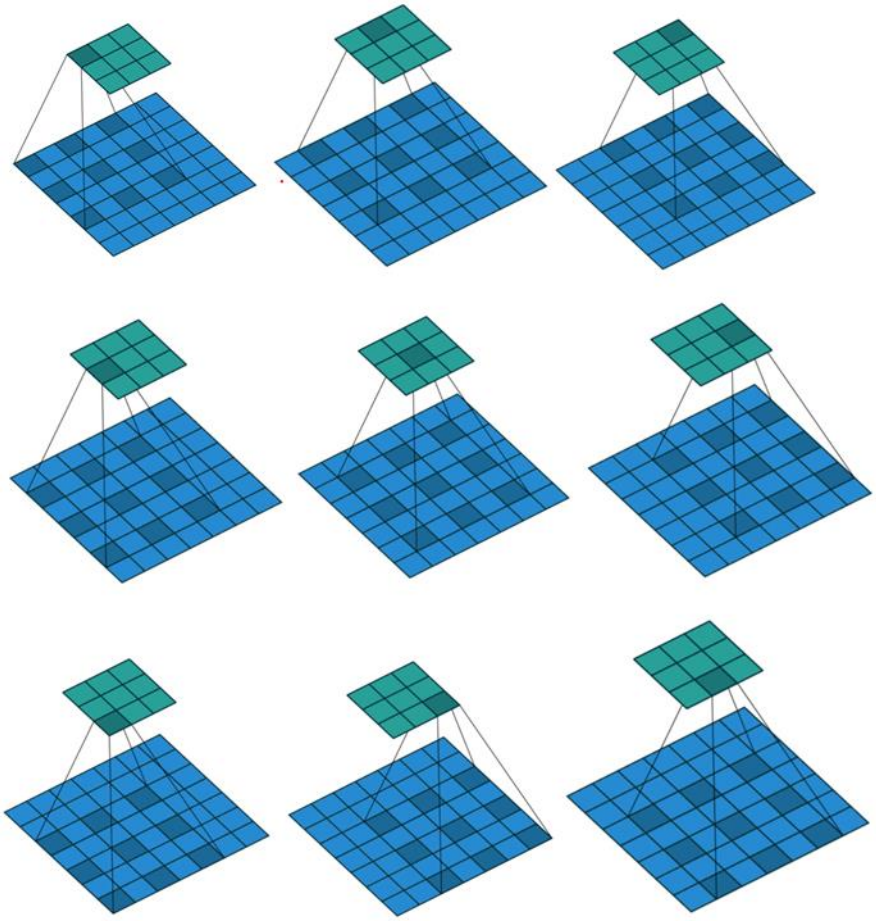


Figure 2.9(a) 2D convolution using a 3 kernel with a dilation rate of 2 and no padding

Transposed Convolutions (deconvolutions or fractionally strided convolutions)

- Some sources use the name deconvolution, which is inappropriate because it's not a deconvolution.
- To make things worse deconvolutions do exists, but they're not common in the field of deep learning.
- An actual deconvolution reverts the process of a convolution. Imagine inputting an image into a single convolutional layer.
- Now take the output, throw it into a black box and out comes your original image again.
- This black box does a deconvolution. It is the mathematical inverse of what a convolutional layer does.
- A transposed convolution is somewhat similar because it produces the same spatial resolution a hypothetical deconvolutional layer would.
- However, the actual mathematical operation that's being performed on the values is different.

- A transposed convolutional layer carries out a regular convolution but reverts its spatial transformation.

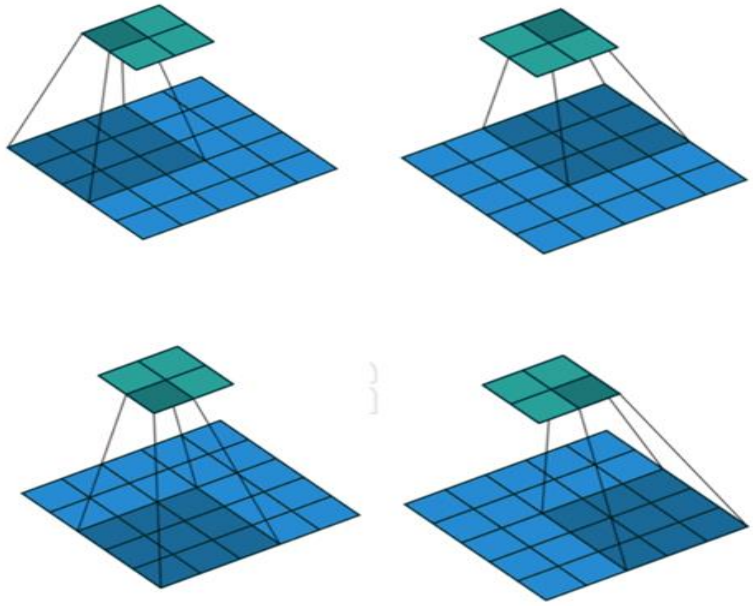


Figure 2.9(b) 2D convolution with no padding, stride of 2 and kernel of 3

- At this point you should be pretty confused, so let's look at a concrete example. An image of 5x5 is fed into a convolutional layer. The stride is set to 2, the padding is deactivated and the kernel is 3x3. This results in a 2x2 image.
- Refer figure 2.9(b) 2D convolution with no padding, stride of 2 and kernel of 3

- If we wanted to reverse this process, we'd need the inverse mathematical operation so that 9 values are generated from each pixel we input. Afterward, we traverse the output image with a stride of 2. This would be a deconvolution.

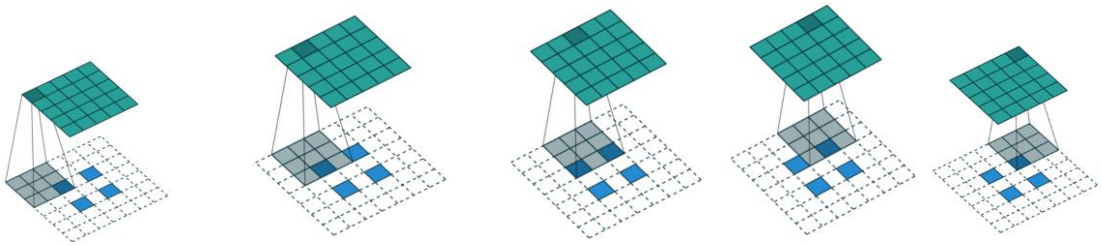


Figure 2.10 2D convolution with no padding, stride of 2 and kernel of 3

- A transposed convolution does not do that. The only thing in common is it guarantees that the output will be a 5x5 image as well, while still performing a

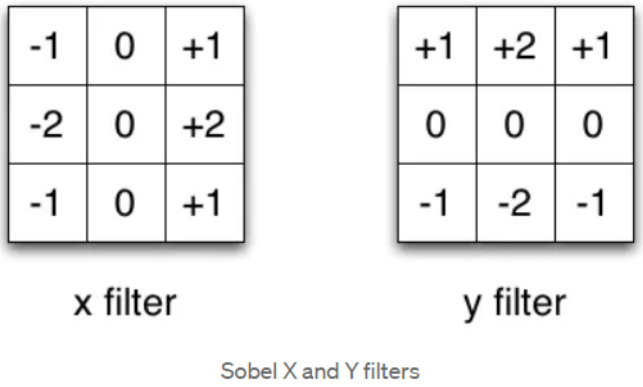
normal convolution operation. To achieve this, we need to perform some fancy padding on the input.

- As you can imagine now, this step will not reverse the process from above. At least not concerning the numeric values.
- It merely reconstructs the spatial resolution from before and performs a convolution. This may not be the mathematical inverse, but for Encoder-Decoder architectures, it's still very helpful. This way we can combine the upscaling of an image with a convolution, instead of doing two separate processes.

Separable Convolutions

- In a separable convolution, we can split the kernel operation into multiple steps. Let's express a convolution as **y = conv (x, k)** where **y** is the output image, **x** is the input image, and **k** is the kernel.
- Easy. Next, let's assume k can be calculated by: **k = k1.dot(k2)**. This would make it a separable convolution because instead of doing a 2D convolution with k, we could get to the same result by doing 2 1D convolutions with k1 and k2.

Refer Figure 2.11.



Sobel X and Y filters

Figure 2.11 Sobel X and Y filter

- Take the Sobel kernel for example, which is often used in image processing. You could get the same kernel by multiplying the vector [1, 0, -1] and [1,2,1].T.
- This would require 6 instead of 9 parameters while doing the same operation. The example above shows what's called a **spatial separable convolution**, which to my knowledge isn't used in deep learning.

- *Edit: Actually, one can create something very similar to a spatial separable convolution by stacking a 1xN and a Nx1 kernel layer. This was recently used in an architecture called EffNet showing promising results.*
- In neural networks, we commonly use something called a **depth wise separable convolution.**
- This will perform a spatial convolution while keeping the channels separate and then follow with a depth wise convolution. In my opinion, it can be best
- understood with an example.
- Let's say we have a 3x3 convolutional layer on 16 input channels and 32 output channels. What happens in detail is that every of the 16 channels is traversed by 32 3x3 kernels resulting in 512 (16x32) feature maps.
- Next, we merge 1 feature map out of every input channel by adding them up. Since we can do those 32 times, we get the 32 output channels we wanted.
- For a depth wise separable convolution on the same example, we traverse the 16 channels with 1 3x3 kernel each, giving us 16 feature maps. Now, before merging anything, we traverse these 16 feature maps with 32 1x1 convolutions each and only then start to them add together. This results in 656 (16x3x3 + 16x32x1x1) parameters opposed to the 4608 (16x32x3x3) parameters from above.
- The example is a specific implementation of a depth wise separable convolution where the so-called **depth multiplier** is 1. This is by far the most common setup for such layers.
- We do this because of the hypothesis that spatial and depth wise information can be decoupled.
- Looking at the performance of the Xception model this theory seems to work. Depth wise separable convolutions are also used for mobile devices because of their efficient use of parameters.

4. Describe in detail about non linearity functions in CNN Learning. (or) How to introduce non linearity in convolutional neural networks. Explain with an example. (NOV/DEC 2023)

Activation function:

- Artificial neurons are elementary units in an artificial neural network. The artificial neuron receives one or more inputs and sums them to produce an output. Each input is separately weighted, and the sum is passed through a function known as an activation function or transfer function.

- In an artificial neural network, the function which takes the incoming signals as input and produces the output signal is known as the activation function.
- Neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.
- The neural network has neurons that work in correspondence with *weight, bias,* and their respective activation function. In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as **Backpropagation**.
- Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.

Need for Non-linear activation function

- A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.
- The two main categories of activation functions are:
  1. Linear Activation Function
  2. Non-linear Activation Functions

Linear Activation Function

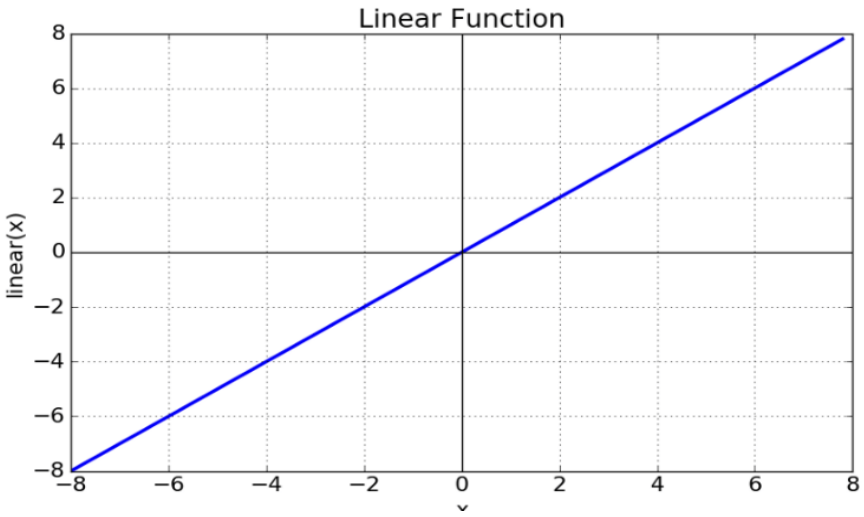


Figure 2.12 Linear Activation Function

Non-linear Activation Functions

- **Linear Function**
  - Linear function has the equation similar to as of a straight line i.e. **y = x** (see in Figure 2.12).
  - No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.
  - **Range :** -inf to +inf
  - **Uses : Linear activation function** is used at just one place i.e. output layer.
- **Sigmoid Function**
  - It is a function which is plotted as **'S'** shaped graph (Refer Figure 2.13).
  - **Equation:** $A = 1/(1 + e^{-x})$
  - **Nature:** Non-linear. Notice that X values lies between -2 to 2, Y values are very steep. This means, small changes in x would also bring about large changes in the value of Y.
  - **Value Range:** 0 to 1
  - **Uses:** Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be ***1*** if value is greater than **0.5** and ***0*** otherwise.

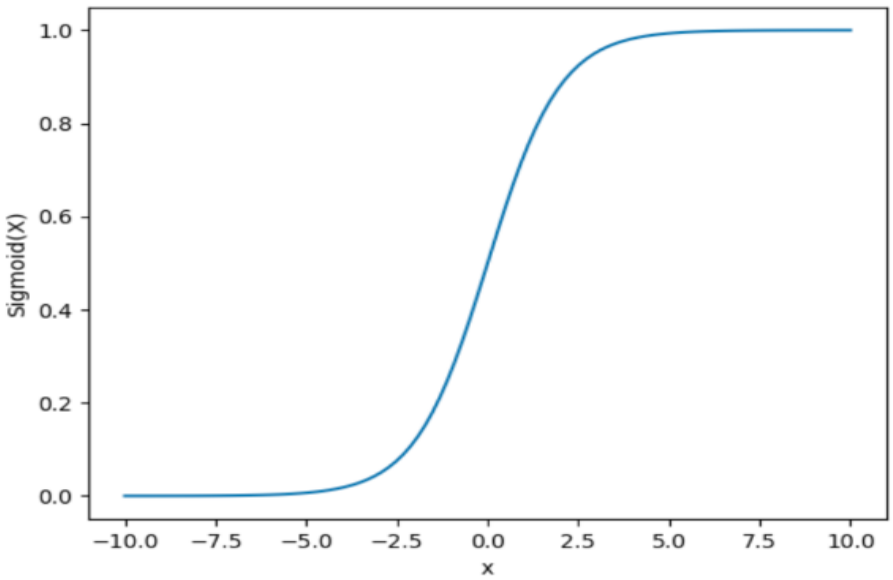


Figure 2.13 Sigmoid Function

- **Tanh Function**
  - The activation that works almost always better than sigmoid function is Tanh function also known as **Tangent Hyperbolic function**. It's actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other **(see in Figure 2.14).**

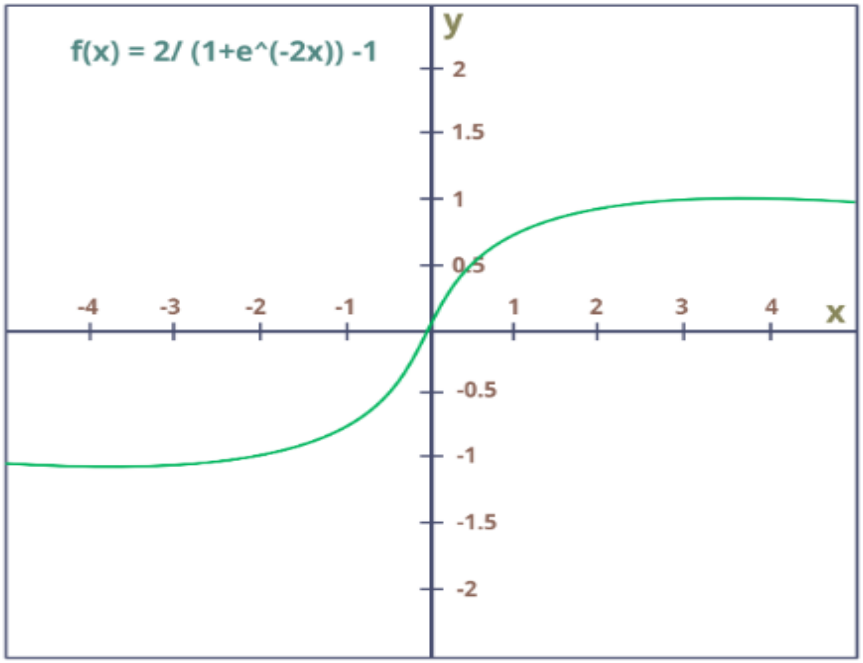


Figure:2.14 Tanh Function

- **Equation :-**

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

- **Value Range :-** -1 to +1
- **Nature :-** non-linear
- **Uses :-** Usually used in hidden layers of a neural network as it's values lies between **-1 to 1** hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in *centering the data* by bringing mean close to 0. This makes learning for the next layer much easier.

- **ReLU Function**
  - It Stands for *Rectified linear unit*. It is the most widely used activation function. Chiefly implemented in *hidden layers* of Neural network.

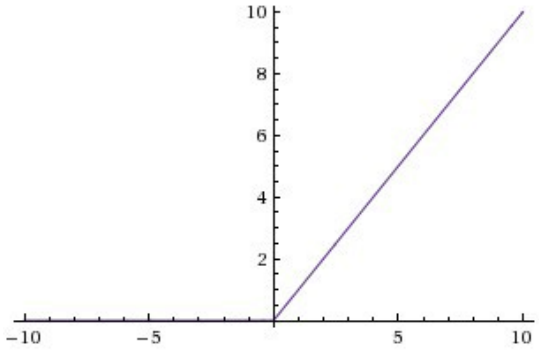


Figure 2.15 ReLU FUNCTION

- **Equation :-** ***A(x) = max(0,x)***. It gives an output x if x is positive and 0 otherwise **(see in Figure 2.15).**
- **Value Range :-** [0, inf]
- **Nature :-** non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.
- **Uses :-** ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.
- In simple words, RELU learns *much faster* than sigmoid and Tanh function.

- **Softmax Function**
  - ✓ It is a subclass of the sigmoid function, the softmax function comes in handy when dealing with multiclass classification issues.
  - ✓ Used frequently when managing several classes. In the output nodes of image classification issues, the softmax was typically present. The softmax function would split by the sum of the outputs and squeeze all outputs for each category between 0 and 1.
  - ✓ The output unit of the classifier, where we are actually attempting to obtain the probabilities to determine the class of each input, is where the softmax function is best applied.

5. Discuss about Loss Functions in CNN learning.

Loss Functions:

- The previous section has presented various layer-types of CNN architecture.
- In addition, the final classification is achieved from the output layer, which represents the last layer of the CNN architecture.
- Some loss functions are utilized in the output layer to calculate the predicted error created across the training samples in the CNN model.
- This error reveals the difference between the actual output and the predicted one.
- Next, it will be optimized through the CNN learning process.
- However, two parameters are used by the loss function to calculate the error.
- The CNN estimated output (referred to as the prediction) is the first parameter.
- The actual output (referred to as the label) is the second parameter. Several types of loss function are employed in various problem types.
- The following concisely explains some of the loss function types.

Cross-Entropy or Softmax Loss Function:

- This function is commonly employed for measuring the CNN model performance.
- It is also referred to as the log loss function. Its output is the probability p∈{0,1}.
- In addition, it is usually employed as a substitution of the square error loss function in multi-class classification problems.
- In the output layer, it employs the softmax activations to generate the output within a probability distribution.
- The mathematical representation of the output class probability is

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a}$$

- Here, e^{ai} represents the non-normalized output from the preceding layer, while *N* represents the number of neurons in the output layer. Finally, the mathematical representation of cross-entropy loss function is

$$H(p, y) = -\sum_i y_i \log(p_i) \quad where \quad i \in [1, N]$$

Euclidean Loss Function:

- This function is widely used in regression problems. In addition, it is also the so-called mean square error. The mathematical expression of the estimated Euclidean loss is

$$H(p,y) = \frac{1}{2N}\sum_{i=1}^{N}(p_i - y_i)^2$$

Hinge Loss Function

- This function is commonly employed in problems related to binary classification. This problem relates to maximum-margin-based classification; this is mostly important for SVMs, which use the hinge loss function, wherein the optimizer attempts to maximize the margin around dual objective classes.
- Its mathematical formula is

$$H(p,y) = \sum_{i=1}^{N} max(0, m - (2y_i - 1)p_i)$$

- The margin m is commonly set to 1. Moreover, the predicted output is denoted as pi, while the desired output is denoted as yi.

6. Explain in detail about Regularization to CNN.

Regularization to CNN:

For CNN models, over-fitting represents the central issue associated with obtaining well-behaved generalization. The model is entitled over-fitted in cases where the model executes especially well on training data and does not succeed on test data (unseen data) which is more explained in the latter section. An under-fitted model is the opposite; this case occurs when the model does not learn a sufficient amount from the training data. The model is referred to as "just-fitted" if it executes well on both training and testing data. These three types are illustrated in Figure 2.16. Various intuitive concepts are used to help the regularization to avoid over-fitting; more details about over-fitting and under-fitting are discussed in latter sections.

- **Dropout:** This is a widely utilized technique for generalization. During each training epoch, neurons are randomly dropped. In doing this, the feature selection power is distributed equally across the whole group of neurons, as

well as forcing the model to learn different independent features. During the training process, the dropped neuron will not be a part of back-propagation or forward-propagation. By contrast, the full-scale network is utilized to perform prediction during the testing process.

- **Drop-Weights:** This method is highly similar to dropout. In each training epoch, the connections between neurons (weights) are dropped rather than dropping the neurons; this represents the only difference between drop-weights and dropout.
- **Data Augmentation:** Training the model on a sizeable amount of data is the easiest way to avoid over-fitting. To achieve this, data augmentation is used. Several techniques are utilized to artificially expand the size of the training dataset. More details can be found in the latter section, which describes the data augmentation techniques.
- **Batch Normalization:** This method ensures the performance of the output activations. This performance follows a unit Gaussian distribution. Subtracting the mean and dividing by the standard deviation will normalize the output at each layer. While it is possible to consider this as a pre-processing task at each layer in the network, it is also possible to differentiate and to integrate it with other networks. In addition, it is employed to reduce the "internal covariance shift" of the activation layers. In each layer, the variation in the activation distribution defines the internal covariance shift. This shift becomes very high due to the continuous weight updating through training, which may occur if the samples of the training data are gathered from numerous dissimilar sources (for example, day and night images). Thus, the model will consume extra time for convergence, and in turn, the time required for training will also increase. To resolve this issue, a layer representing the operation of batch normalization is applied in the CNN architecture.

  The advantages of utilizing batch normalization are as follows:

  - It prevents the problem of vanishing gradient from arising.
  - It can effectively control the poor weight initialization.
  - It significantly reduces the time required for network convergence (for large-scale datasets, this will be extremely useful).
  - It struggles to decrease training dependency across hyper-parameters.

- Chances of over-fitting are reduced, since it has a minor influence on regularization.

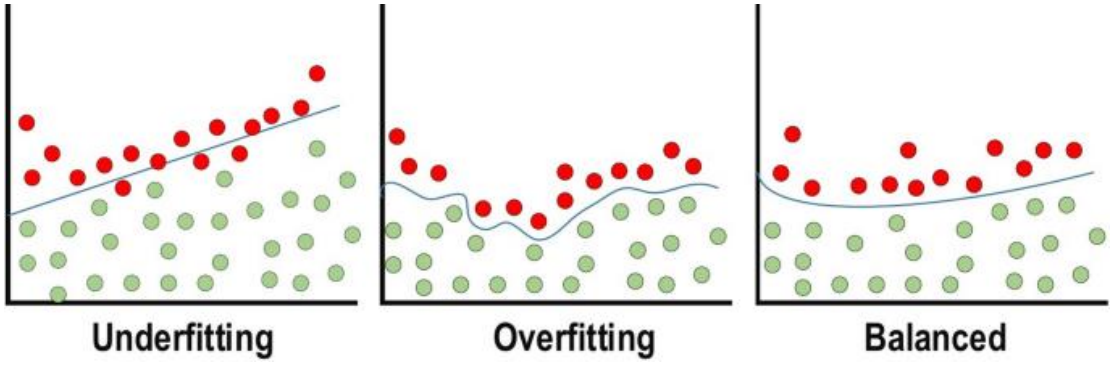


Figure 2.16 underfitting vs overfitting vs balanced or goodness of fit

7. Explain in detail about Optimizer.

Optimizer

- Two major issues are included in the learning process: the first issue is the learning algorithm selection (optimizer), while the second issue is the use of many enhancements (such as AdaDelta, Adagrad, and momentum) along with the learning algorithm to enhance the output.
- Loss functions, which are founded on numerous learnable parameters (e.g. biases, weights, etc.) or minimizing the error (variation between actual and predicted output), are the core purpose of all supervised learning algorithms.
- The techniques of gradient-based learning for a CNN network appear as the usual selection.
- The network parameters should always update though all training epochs, while the network should also look for the locally optimized answer in all training epochs in order to minimize the error.
- The learning rate is defined as the step size of the parameter updating. The training epoch represents a complete repetition of the parameter update that involves the complete training dataset at one time. Note that it needs to select the learning rate wisely so that it does not influence the learning process imperfectly, although it is a hyper-parameter.
- Gradient Descent or Gradient-based learning algorithm: To minimize the training error, this algorithm repetitively updates the network parameters through every training epoch.

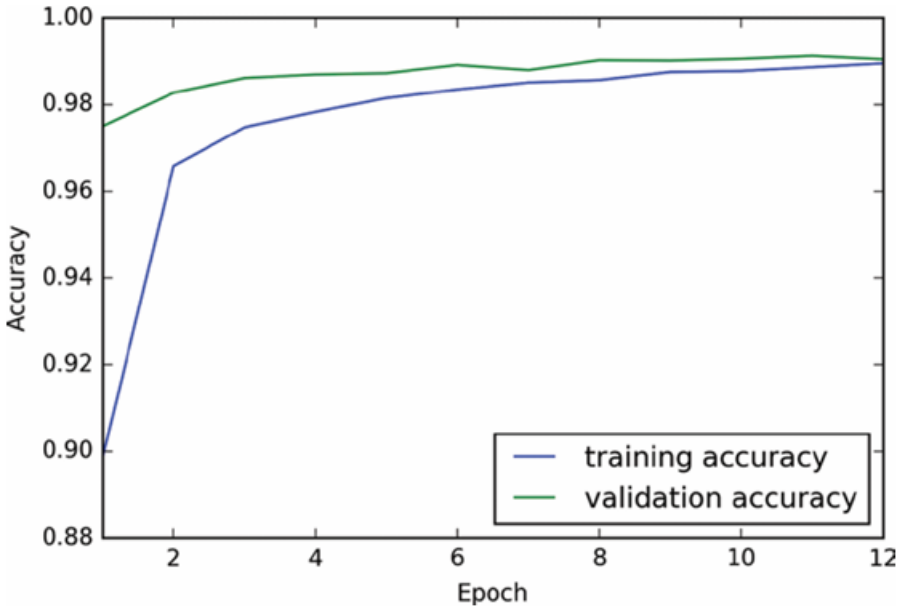


Figure 17-3: The training performance of our convnet in Figure 17-2. We trained for 12 epochs, and since the training and validation curves are not diverging, we've successfully avoided overfitting, while reaching about 99 percent accuracy on both data sets.

- More specifically, to update the parameters correctly, it needs to compute the objective function gradient (slope) by applying a first-order derivative with respect to the network parameters.
- Next, the parameter is updated in the reverse direction of the gradient to reduce the error.
- The parameter updating process is performed though network back-propagation, in which the gradient at every neuron is back-propagated to all neurons in the preceding layer. The mathematical representation of this operation is as

$$w_{ij^t} = w_{ij^{t-1}} - \Delta w_{ij^t}, \quad \Delta w_{ij^t} = \eta * \frac{\partial E}{\partial w_{ij}}$$

- The final weight in the current training epoch is denoted by wijt, while the weight in the preceding (t−1) training epoch is denoted wijt−1. The learning rate is η and the prediction error is *E*.

8. Explain in detail about Gradient Computation.

Different alternatives of the gradient-based learning algorithm are available and commonly employed; these include the following:

- **Batch Gradient Descent:** During the execution of this technique, the network parameters are updated merely one time behind considering all training datasets via the network. In more depth, it calculates the gradient of the whole training set and subsequently uses this gradient to update the parameters. For a small-sized dataset, the CNN model converges faster and creates an extra-stable gradient using BGD. Since the parameters are changed only once for every training epoch, it requires a substantial amount of resources. By contrast, for a large training dataset, additional time is required for converging, and it could converge to a local optimum (for non-convex instances).
- **Stochastic Gradient Descent:** The parameters are updated at each training sample in this technique. It is preferred to arbitrarily sample the training samples in every epoch in advance of training. For a large-sized training dataset, this technique is both more memory-effective and much faster than BGD. However, because it is frequently updated, it takes extremely noisy steps in the direction of the answer, which in turn causes the convergence behavior to become highly unstable.
- **Mini-batch Gradient Descent:** In this approach, the training samples are partitioned into several mini-batches, in which every mini-batch can be considered an under-sized collection of samples with no overlap between them. Next, parameter updating is performed following gradient computation on every mini-batch. The advantage of this method comes from combining the advantages of both BGD and SGD techniques. Thus, it has a steady convergence, more computational efficiency and extra memory effectiveness. The following describes several enhancement techniques in gradient-based learning algorithms (usually in SGD), which further powerfully enhance the CNN training process.
- **Momentum:** For neural networks, this technique is employed in the objective function. It enhances both the accuracy and the training speed by summing the computed gradient at the preceding training step, which is weighted via a factor λ (known as the momentum factor). However, it therefore simply becomes stuck in a local minimum rather than a global minimum. This represents the main disadvantage of gradient-based learning algorithms. Issues of this kind frequently occur if the issue has no convex surface (or solution space).

Together with the learning algorithm, momentum is used to solve this issue, which can be expressed mathematically as in

$$\Delta w_{ij^t} = \left(\eta * \frac{\partial E}{\partial w_{ij}}\right) + (\lambda * \Delta w_{ij^{t-1}})$$

- The weight increment in the current t′th training epoch is denoted as Δwijt, while η is the learning rate, and the weight increment in the preceding (t−1)′th training epoch.
- The momentum factor value is maintained within the range 0 to 1; in turn, the step size of the weight updating increases in the direction of the bare minimum to minimize the error.
- As the value of the momentum factor becomes very low, the model loses its ability to avoid the local bare minimum.
- By contrast, as the momentum factor value becomes high, the model develops the ability to converge much more rapidly. If a high value of momentum factor is used together with LR, then the model could miss the global bare minimum by crossing over it.
- However, when the gradient varies its direction continually throughout the training process, then the suitable value of the momentum factor (which is a hyper-parameter) causes a smoothening of the weight updating variations.

Adaptive Moment Estimation (Adam):

- It is another optimization technique or learning algorithm that is widely used. Adam represents the latest trends in deep learning optimization.
- This is represented by the Hessian matrix, which employs a second-order derivative. Adam is a learning strategy that has been designed specifically for training deep neural networks.
- More memory efficient and less computational power are two advantages of Adam. The mechanism of Adam is to calculate adaptive LR for each parameter in the model. It integrates the pros of both Momentum and RMSprop. It utilizes the squared gradients to scale the learning rate as RMSprop and it is similar to the momentum by using the moving average of the gradient. The equation of Adam is represented in

$$w_{ij^t} = w_{ij^{t-1}} - \frac{\eta}{\sqrt{\widehat{E[\delta^2]^t} + \in}} * \widehat{E[\delta^2]^t}$$

Improving performance of CNN

Based on experiments in different DL applications can conclude the most active solutions that may improve the performance of CNN are:

- Expand the dataset with data augmentation or use transfer learning
- Increase the training time.
- Increase the depth (or width) of the model.
- Add regularization.
- Increase hyperparameters tuning.

Develop a table with examples of different formats of data that can be used with convolutional networks.

Data Format	Description	Example	Dimensionality	Typical Applications
Images	2D grids of pixel values	Handwritten digits (MNIST)	$H \times W \times C$ (e.g., $28 \times 28 \times 1$ for grayscale, $224 \times 224 \times 3$ for RGB)	Image classification, object detection, segmentation
Video	Sequence of frames over time	Sports video clips	$T \times H \times W \times C$ (e.g., 16 frames, $224 \times 224 \times 3$ for RGB)	Action recognition, video classification
1D Time Series	Sequential data over time	Stock prices, ECG signals	$T \times 1$ or T	Time series analysis, anomaly detection
Spectrograms	Visual representation of audio frequencies over time	Speech data, bird sounds	$H \times W$ or $H \times W \times 1$	Speech recognition, audio classification
Text (Word Embeddings)	Sequences converted to numerical vectors	Sentences from a news article	$L \times E$ (e.g., 100 words, 300 embedding dimensions)	Sentiment analysis, text classification

3D Volumetric Data	3D objects with values in each voxel	CT scans, MRI scans	$D \times H \times W$ (e.g., $128 \times 128 \times 128$)	Medical imaging, 3D object recognition
Multi-Spectral Images	Images with more than three color channels	Satellite images (e.g., LANDSAT)	$H \times W \times C$ (e.g., $512 \times 512 \times 7$)	Remote sensing, agricultural monitoring
Depth Maps	Images with depth information for each pixel	Kinect camera outputs	$H \times W \times 1$	3D reconstruction, human pose estimation
Point Clouds	Sets of points representing 3D objects	LiDAR data for self-driving cars	$N \times 3$ (N points with x, y, z coordinates)	3D object detection, autonomous driving
Graphs	Data represented as nodes and edges	Social networks, molecular structures	Variable (nodes with features and edges)	Graph classification, chemical property prediction

- **Images** are the most common data format for CNNs, but CNNs can be adapted to handle many other data types, such as audio, video, text, and 3D data.
- Dimensionality depends on the data format: H×W×CH \times W \times CH×W×C for 2D images, T×H×W×CT \times H \times W \times CT×H×W×C for videos, and D×H×WD \times H \times WD×H×W for 3D volumetric data.
- Each data format is used in various applications, demonstrating the versatility of convolutional networks in handling diverse data types.

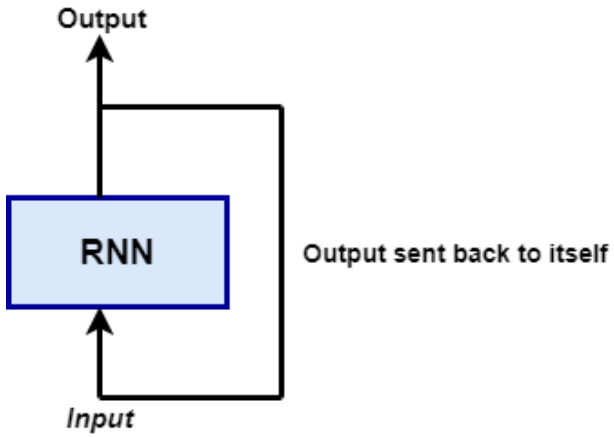
UNIT III RECURRENT NEURAL NETWORKS

Unfolding Graphs -- RNN Design Patterns: Acceptor -- Encoder --Transducer; Gradient Computation -- Sequence Modeling Conditioned on Contexts -- Bidirectional RNN -- Sequence to Sequence RNN – Deep Recurrent Networks -- Recursive Neural Networks -- Long Term Dependencies; Leaky Units: Skip connections and dropouts; Gated Architecture: LSTM.

PART A

1. **What is Recurrent neural network?**

A recurrent neural network (**RNN**) is a kind of artificial neural network mainly used in **speech recognition** and **natural language processing** (NLP). RNN is used in deep learning and in the development of models that imitate the activity of neurons in the human **brain**.



2. **What are the types of Recurrent Neural Networks?**

There are four types of Recurrent Neural Networks:

1. One to One
2. One to Many
3. Many to One
4. Many to Many

3. **List the advantages of Recurrent Neural Networks.**

Advantages of RNNs

- RNN architecture is designed it such a way than it can process inputs of any length. Even with the input size growing larger, the model size does not increase.
- An RNN model is modeled to remember each information throughout the time which is very helpful in any time series predictor.

- The weights of all the dependent hidden layers in between can be shared across the time steps.
- The internal memory of Recurrent Neural Networks is an inherent property that is used for processing the arbitrary series of inputs which is not the case with feedforward neural networks.
- RNNs when combined with traditional Convolutinal Neural Networks gives an effective pixel neighborhood prediction.

4. List the advantages and disadvantages of Recurrent Neural Networks.

Disadvantages of RNNs

- Due to its recurrent nature, the computation becomes slow.
- Training of RNN models can be very difficult and time-consuming as compared to other Artificial Neural Networks.
- It becomes very difficult to process sequences that are very long if the activation functions used are ReLu or tanh as activation functions,
- Prone to problems such as *exploding* and *gradient* vanishing.
- RNNs cannot be stacked into very deep models
- RNNs are not able to keep track of long-term dependencies

5. List the Applications of RNN.

1. Machine Translation
2. Speech Recognition
3. Sentiment Analysis
4. Automatic Image Tagger

6. What are the Common RNN usage-patterns?

Acceptor

Encoder

Transducers

7. Define Gradient Descent.

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.

The general idea is to tweak parameters iteratively in order to minimize the cost function.

8. Define Gradient computation.

Gradient computation is central to training neural networks, enabling the model to learn from data by adjusting its parameters in a way that minimizes the error between its predictions and the actual target values. Backpropagation, powered by the chain rule, and gradient descent are key components in this process.

9. List the types of Gradient Descent.

1. Batch Gradient Descent

Batch Gradient Descent involves calculations over the full training set at each step as a result of which it is very slow on very large training data. Thus, it becomes very computationally expensive to do Batch GD.

2. Stochastic Gradient Descent

In SGD, only one training example is used to compute the gradient and update the parameters at each iteration. This can be faster than batch gradient descent but may lead to more noise in the updates.

3. Mini-batch Gradient Descent

In mini-batch gradient descent, a small batch of training examples is used to compute the gradient and update the parameters at each iteration. This can be a good compromise between batch gradient descent and SGD, as it can be faster than batch gradient descent and less noisy than SGD.

10. Define Backpropagation.

Backpropagation is the process of computing the gradient of the loss function with respect to each of the model's parameters (weights and biases). This involves moving backwards through the network, from the output layer to the input layer.

11. Define Chain Rule.

The chain rule of calculus is used to compute the gradient of the loss function with respect to each parameter. The gradient is computed by multiplying the derivative of the loss function with respect to the output of a layer by the derivative of that layer's output with respect to its input.

12. Define Bidirectional Recurrent Neural Network.

An architecture of a neural network called a bidirectional recurrent neural network (BRNN) is made to process sequential data. In order for the network to use information from both the past and future context in its predictions, BRNNs process input sequences in both the forward and backward directions. This is the main distinction between BRNNs and conventional recurrent neural networks.

13. What is Sequence to Sequence RNN?

- Seq2Seq model or Sequence-to-Sequence model, is a machine learning architecture designed for tasks involving sequential data.
- It takes an input sequence, processes it, and generates an output sequence.
- The architecture consists of two fundamental components: an encoder and a decoder. Seq2Seq models have significantly improved the quality of machine translation systems making them an important technology.

14. Define Deep Recurrent Network.

- Deep RNN (Recurrent Neural Network) refers to a neural network architecture that has multiple layers of recurrent units.
- Recurrent Neural Networks are a type of neural network that is designed to handle sequential data, such as time series or natural language, by maintaining an internal memory of previous inputs.

15. Difference between Recursive Neural Network and CNN

RvNN	CNN
It is designed to process hierarchical or tree-structured data, capturing dependencies within recursively structured information.	It primarily used for grid-structured data like images, where convolutional layers are applied to local receptive fields.
It processes data in a recursive manner, combining information from child nodes to form representations for parent nodes.	It uses convolutional layers to extract local features and spatial hierarchies from input data.

It is suitable for tasks involving nested structures like natural language parsing or molecular structure analysis.	It excels in tasks related to computer vision, image recognition and other grid-based data.

16. Difference between RvNN and RNN

RvNN	RNN
It is designed for sequential data like time series or sequences of words.	It is specially designed to tailor for hierarchical data structures like trees.
It processes sequences by maintaining hidden states which capture temporal dependencies.	It processes hierarchical structures by recursively combining information from child nodes.
Commonly used in tasks like natural language processing, speech recognition and time series prediction.	Commonly used in applications which involves hierarchical relationships like parsing in NLP, analyzing molecular structures or image segmentation.
It has linear topology where information flows sequentially through time steps.	It has a tree-like or hierarchical topology which allows them to capture nested relationships within the data.

PART B

1. Describe in detail Unfolding Graphs.

Unfolding Computational Graphs

- A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss.
- The idea of unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure. For example, consider the classical form of a dynamical system:

$$s^{(t)} = f(s^{(t-1)}; \theta), \qquad (10.1)$$

- Where s(t) is called the state of the system. Equation 10.1 is recurrent because the definition of s at time t refers back to the same definition at time t − 1.
- For a finite number of time steps τ , the graph can be unfolded by applying the definition τ − 1 times. For example, if we unfold equation 10.1 for τ = 3 time steps, we obtain

$$s^{(3)} = f(s^{(2)}; \theta) \qquad (10.2)$$

$$= f(f(s^{(1)}; \theta); \theta) \qquad (10.3)$$

- Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence.
- Such an expression can now be represented by a traditional directed acyclic computational graph.
- The unfolded computational graph of equation 10.1 and equation 10.3 is illustrated in figure 10.1. s(t−1) s(t) s(t+1) f s(... as shown figure 3.1.

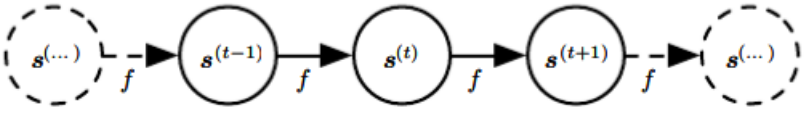


Figure 3.1: The classical dynamical system described by equation 10.1

- Each node represents the state at some time t and the function f maps the state at t to the state at t + 1. The same parameters (the same value of θ used to parametrize f) are used for all time steps.
- As another example, let us consider a dynamical system driven by an external signal x(t),

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta), \qquad (10.4)$$

- where we see that the state now contains information about the whole past sequence.
- Many recurrent neural networks use equation 10.5 or a similar equation to define the values of their hidden units. To indicate that the state is the hidden units of the network, we now rewrite equation 10.4 using the variable h to represent the state:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta), \qquad (10.5)$$

- As illustrated in figure 10.2, typical RNNs will add extra architectural features such as output layers that read information out of the state h to make predictions.

Predicting the Future from the Past

- When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use h(t) as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.
- This summary is in general necessarily lossy, since it maps an arbitrary length sequence $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)})$ to a fixed length vector h(t) .
- Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects.

Examples:

- RNN used in statistical language modeling, typically to predict next word from past words.
- It may not be necessary to store all information upto time t but only enough information to predict rest of sentence.
- Most demanding situation: we ask h(t) to be rich enough to allow one to approximately recover the input sequence as in autoencoders.

Unfolding: from circuit diagram to computational graph

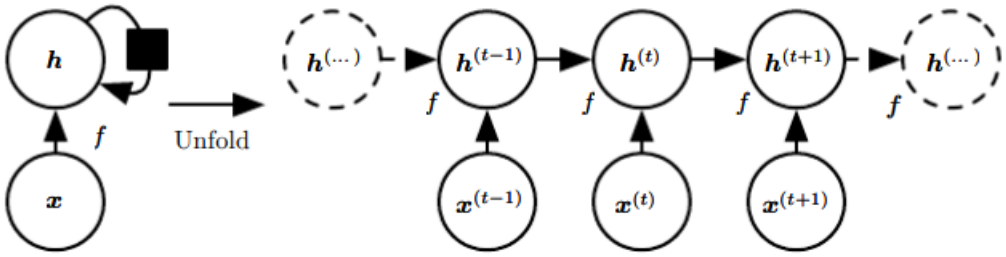


Figure 3.2 A recurrent network with no outputs.

- This recurrent network just processes information from the input x by incorporating it into the state h that is passed forward through time.
- as shown in figure 3.2 (Left)Circuit diagram. The black square indicates a delay of a single time step. (Right)The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.
- We can represent the unfolded recurrence after t steps with a function g(t) :

$$\boldsymbol{h}^{(t)} = g^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right) \quad (10.6)$$

$$= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \quad (10.7)$$

- The function g(t) takes the whole past sequence ($x^{(t)}$, $x^{(t-1)}$, $x^{(t-2)}$, . . . , $x^{(2)}$, $x^{(1)}$) as input and produces the current state, but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f .

The unfolding process thus introduces two major advantages:

- Regardless of sequence length, learned model has same input size because it is specified in terms of transition from one state to another state rather than specified in terms of a variable length history of states.
- Possible to use same function f with same parameters at every step.
- These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths rather than needing separate model g(t) for all possible time steps Learning a single shared model allows:
  - Generalization to sequence lengths that did not appear in the training.
  - Allows model to be estimated with far fewer training examples than would be required without parameter sharing.
- Both the recurrent graph and the unrolled graph have their uses.
- The recurrent graph is succinct.
- The unfolded graph provides an explicit description of which computations to perform.

- The unfolded graph also helps to illustrate the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients) by explicitly showing the path along which this information flows.

2. Explain in detail about RNN Design Patterns.

Recurrent Neural Network (RNN)

A recurrent neural network (**RNN**) is a kind of artificial neural network mainly used in **speech recognition** and **natural language processing** (NLP). RNN is used in deep learning and in the development of models that imitate the activity of neurons in the human **brain**.

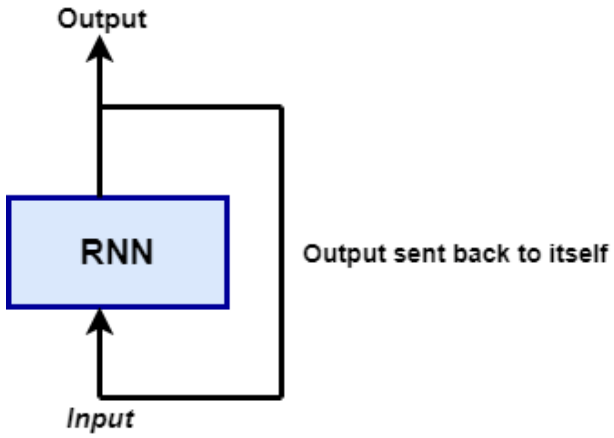


Figure 3.3 working of recurrent neural network

Types of Recurrent Neural Networks

There are four types of Recurrent Neural Networks:

1. One to One
2. One to Many
3. Many to One
4. Many to Many

One to One RNN

- This type of neural network is known as the Vanilla Neural Network. It's used for general machine learning problems, which has a single input and a single output. Refer figure 3.4.

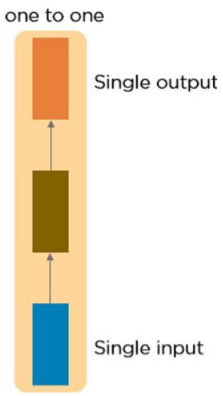


Figure 3.4 one to one RNN

One to Many RNN

- This type of neural network has a single input and multiple outputs. Refer figure 3.5.

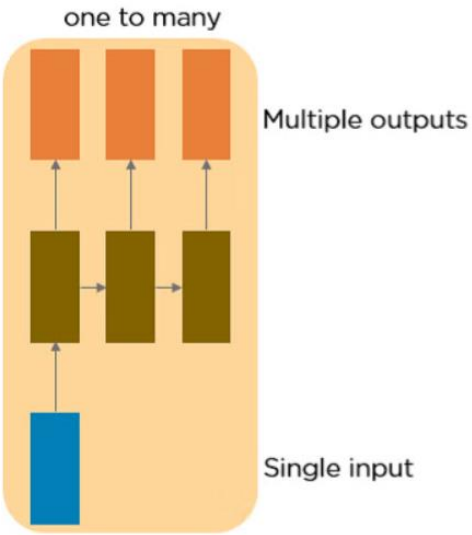


Figure 3.5 one to many RNN

Many to One RNN

- This RNN takes a sequence of inputs and generates a single output. Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments. Refer figure 3.6.

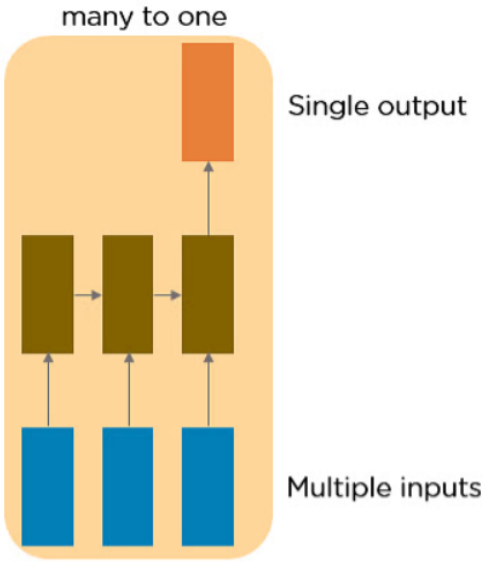


Figure 3.6 many to one RNN

Many to Many RNN

- This RNN takes a sequence of inputs and generates a sequence of outputs. Machine translation is one of the examples. Refer figure 3.7.

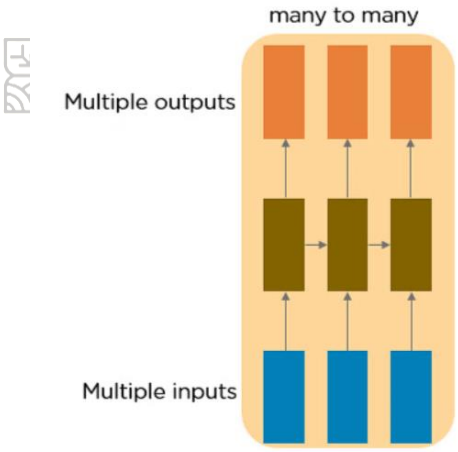


Figure 3.7 many to many RNN

Advantages of RNNs

- RNN architecture is designed it such a way than it can process inputs of any length. Even with the input size growing larger, the model size does not increase.
- An RNN model is modeled to remember each information throughout the time which is very helpful in any time series predictor.
- The weights of all the dependent hidden layers in between can be shared across the time steps.

- The internal memory of Recurrent Neural Networks is an inherent property that is used for processing the arbitrary series of inputs which is not the case with feedforward neural networks.
- RNNs when combined with traditional Convolutinal Neural Networks gives an effective pixel neighborhood prediction.

Disadvantages of RNNs

- Due to its recurrent nature, the computation becomes slow.
- Training of RNN models can be very difficult and time-consuming as compared to other Artificial Neural Networks.
- It becomes very difficult to process sequences that are very long if the activation functions used are ReLu or tanh as activation functions,
- Prone to problems such as *exploding* and *gradient* vanishing.
- RNNs cannot be stacked into very deep models
- RNNs are not able to keep track of long-term dependencies

Application of RNN

1. Machine Translation
2. Speech Recognition
3. Sentiment Analysis
4. Automatic Image Tagger

Limitations of RNN

- RNN is supposed to carry the information in time. However, it is quite challenging to propagate all this information when the time step is too long. When a network has too many deep layers, it becomes untrainable. This problem is called: vanishing gradient problem.

Common RNN usage-patterns

Acceptor

- One option is to base the supervision signal only at the final output vector, y_n. Viewed this way, the RNN is trained as an acceptor. Observe the final state, and then decide on an outcome. Refer figure 3.8.

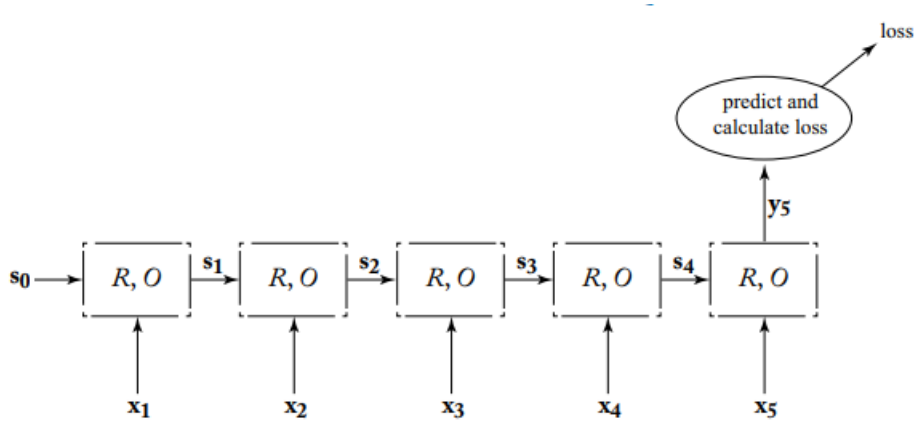


Figure 3.8 RNN is trained as an acceptor

- For example, consider training an RNN to read the characters of a word one by one and then use the final state to predict the part-of-speech of that word, an RNN that reads in a sentence and, based on the final state decides if it conveys positive or negative sentiment or an RNN that reads in a sequence of words and decides whether it is a valid noun-phrase.
- The loss in such cases is defined in terms of a function of $y_n = O(s_n)$.
- Typically, the RNN's output vector y_n is fed into a fully connected layer or an MLP, which produce a prediction. The error gradients are then backpropagated through the rest of the sequence (see Figure3.8).
- The loss can take any familiar form: cross entropy, hinge, margin, etc.

Encoders

- Similar to the acceptor case, an encoder supervision uses only the final output vector, y_n.
- However, unlike the acceptor, where a prediction is made solely on the basis of the final vector, here the final vector is treated as an encoding of the information in the sequence, and is used as additional information together with other signals.
- For example, an extractive document summarization system may first run over the document with an RNN, resulting in a vector y_n summarizing the entire document.
- Then, y_n will be used together with other features in order to select the sentences to be included in the summarization.

Transducers

- RNN-Transducer are a form of sequence-to-sequence models that do not employ attention mechanisms.

- Unlike most sequence-to-sequence models, which typically need to process the entire input sequence (the waveform in our case) to produce an output (the sentence), the RNN-T continuously processes input samples and streams output symbols, a property that is welcome for speech dictation.

```
import torch
import torch.nn as nn
from rnnt import RNNTransducer

batch_size, sequence_length, dim = 3, 12345, 80

cuda = torch.cuda.is_available()
device = torch.device('cuda' if cuda else 'cpu')

inputs = torch.rand(batch_size, sequence_length, dim).to(device)
input_lengths = torch.IntTensor([12345, 12300, 12000])
targets = torch.LongTensor([[1, 3, 3, 3, 3, 3, 4, 5, 6, 2],
                            [1, 3, 3, 3, 3, 3, 4, 5, 2, 0],
                            [1, 3, 3, 3, 3, 3, 4, 2, 0, 0]]).to(device)
target_lengths = torch.LongTensor([9, 8, 7])

model = nn.DataParallel(RNNTransducer(num_classes=10)).to(device)

# Forward propagate
outputs = model(inputs, input_lengths, targets, target_lengths)

# Recognize input speech
outputs = model.module.recognize(inputs, input_lengths)
```

- Another option is to treat the RNN as a transducer, producing an output $\hat{t}_i$ for each input it reads in. Modeled this way, we can compute a local loss signal $L_{\text{local}}(\hat{t}_i, t_i)$ for each of the output $\hat{t}_i$ i based on a true label t_i.
- The loss for unrolled sequence will then be:

$L(\hat{t}_{1:n}, t_{1:n}) = \sum_{i=1}^{n} L_{\text{local}}(\hat{t}_i, t_i),$ or using another combination rather than a sum such as an average or a weighted average.

RNN: Transducer Architecture

- Refer figure 3.9.

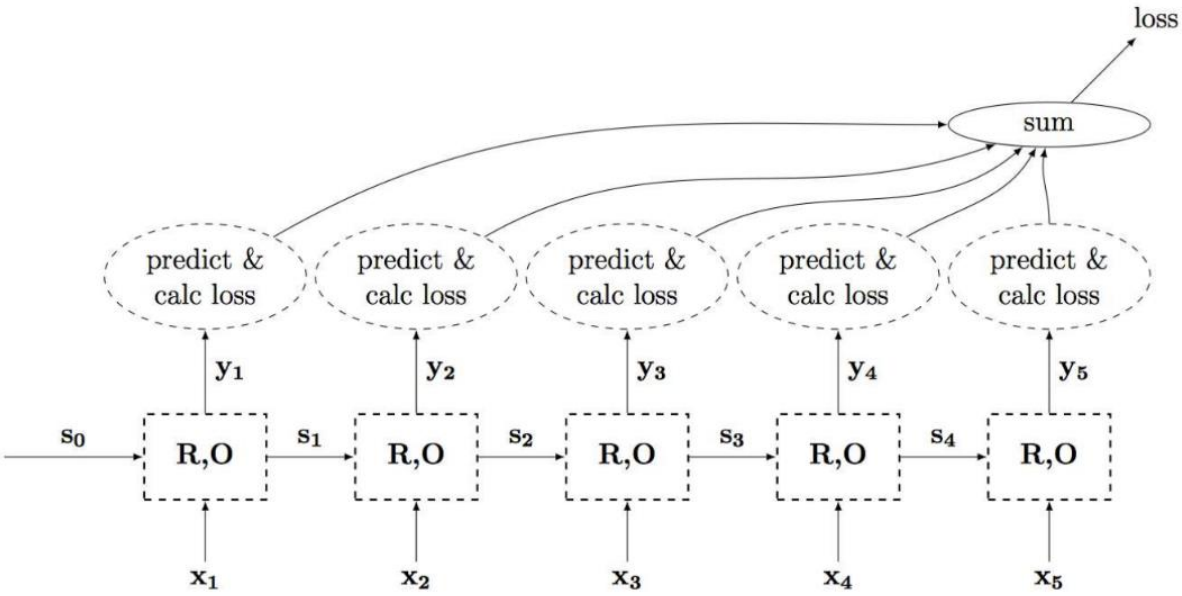


Figure 3.9 RNN: Transducer Architecture

3. Define is gradient descent and explain in detail about gradient computation.

- Gradient computation is central to training neural networks, enabling the model to learn from data by adjusting its parameters in a way that minimizes the error between its predictions and the actual target values. Backpropagation, powered by the chain rule, and gradient descent are key components in this process.

Gradient Descent

- Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea is to tweak parameters iteratively in order to minimize the cost function.
- An important parameter of Gradient Descent (GD) is the size of the steps, determined by the learning rate hyperparameters. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time, and if it is too high we may jump the optimal value.

Types of Gradient Descent:

Typically, there are three types of Gradient Descent:

1. **Batch Gradient Descent**

   Batch Gradient Descent involves calculations over the full training set at each step as a result of which it is very slow on very large training data. Thus, it becomes very computationally expensive to do Batch GD.

2. **Stochastic Gradient Descent**

   In SGD, only one training example is used to compute the gradient and update the parameters at each iteration. This can be faster than batch gradient descent but may lead to more noise in the updates.

3. **Mini-batch Gradient Descent**

   In mini-batch gradient descent, a small batch of training examples is used to compute the gradient and update the parameters at each iteration. This can be a good compromise between batch gradient descent and SGD, as it can be faster than batch gradient descent and less noisy than SGD.

Forward Pass

- During the forward pass, the input data is passed through the network layer by layer. Each layer performs a mathematical operation (like matrix multiplication followed by a non-linear activation function) to produce an output, eventually leading to a prediction.
- The loss function compares the model's predictions with the actual target values to compute the loss.

Backpropagation and Gradient Computation

- **Backpropagation**: Backpropagation is the process of computing the gradient of the loss function with respect to each of the model's parameters (weights and biases). This involves moving backwards through the network, from the output layer to the input layer.

- **Chain Rule**: The chain rule of calculus is used to compute the gradient of the loss function with respect to each parameter. The gradient is computed by multiplying the derivative of the loss function with respect to the output of a layer by the derivative of that layer's output with respect to its input.

- **Gradient for a Layer**: For a layer l, if the output is denoted by a^l and the weights by W^l, the gradient with respect to the weights is calculated as:

$$\frac{\partial \text{Loss}}{\partial W^l} = \delta^l \cdot (a^{l-1})^T$$

where δ^l is the error term (gradient of the loss with respect to the input of layer l), and a^{l-1} is the output of the previous layer.

- **Gradient Descent**: Once the gradients are computed, they are used to update the model's parameters in the direction that minimizes the loss function. This is typically done using a variant of gradient descent, such as stochastic gradient descent (SGD), Adam, or RMSprop.

Updating Parameters

- **Learning Rate**: The parameters are updated by subtracting the product of the gradient and a learning rate (a small positive scalar) from the current parameter values:

$$W^l \leftarrow W^l - \eta \cdot \frac{\partial \text{Loss}}{\partial W^l}$$

where η(etaη) is the learning rate.

Iterative Process

- This process of forward pass, backpropagation, and parameter update is repeated iteratively over many epochs, allowing the model to gradually learn and minimize the loss function.

Automatic Differentiation

- Deep learning frameworks like TensorFlow and PyTorch use a technique called automatic differentiation to efficiently compute gradients. This eliminates the need for manually deriving the gradients and significantly speeds up the training process.

4. Explain in detail Sequence Modeling Conditioned on Contexts.

Recurrent Networks as Directed Graphical Models

- As with a feedforward network, it is in principle possible to use almost any loss with a recurrent network. The loss should be chosen based on the task.

- As with a feedforward network, usually wish to interpret the output of the RNN as a probability distribution, and we usually use the cross-entropy associated with that distribution to define the loss. Mean squared error is the cross-entropy loss associated with an output distribution that is a unit Gaussian, for example, just as with a feedforward network.

- Using a predictive log-likelihood training objective, such as equation, we train the RNN to estimate the conditional distribution of the next sequence element $y^{(t)}$ given the past inputs. This may mean that we maximize the log-likelihood

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}), \qquad (10.29)$$

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

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t-1)}). \qquad (10.30)$$

- Decomposing the joint probability over the sequence of y values as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence.

- When we do not feed past **y** values as inputs that condition the next step prediction, the directed graphical model contains no edges from any $\mathbf{y^{(i)}}$ in the past to the current $\mathbf{y^{(t)}}$.

- In this case, the outputs **y** is conditionally independent given the sequence of **x** values. When we do feed the actual **y** values (not their prediction, but the actual observed or generated values) back into the network, the directed graphical model contains edges from all $\mathbf{y^{(i)}}$values in the past to the current $\mathbf{y^{(t)}}$ value.

- As a simple example, let us consider the case where the RNN models only a sequence of scalar random variables $\mathbb{Y} = \{y^{(1)}, \ldots, y^{(\tau)}\}$, with no additional inputs **x**. The input at time step t is simply the output at time step **t−1**.

- The RNN then defines a directed graphical model over the y variables. We parametrize the joint distribution of these observations using the chain rule for conditional probabilities:

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \ldots, \mathbf{y}^{(1)}) \qquad (10.31)$$

where the right-hand side of the bar is empty for t = 1, of course. Hence the negative log-likelihood of a set of values $\{y^{(1)},\ldots,y^{(t)}\}$ according to such a model.

is

$$L = \sum_t L^{(t)} \qquad (10.32)$$

where

$$L^{(t)} = -\log P(\mathrm{y}^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \ldots, y^{(1)}). \qquad (10.33)$$

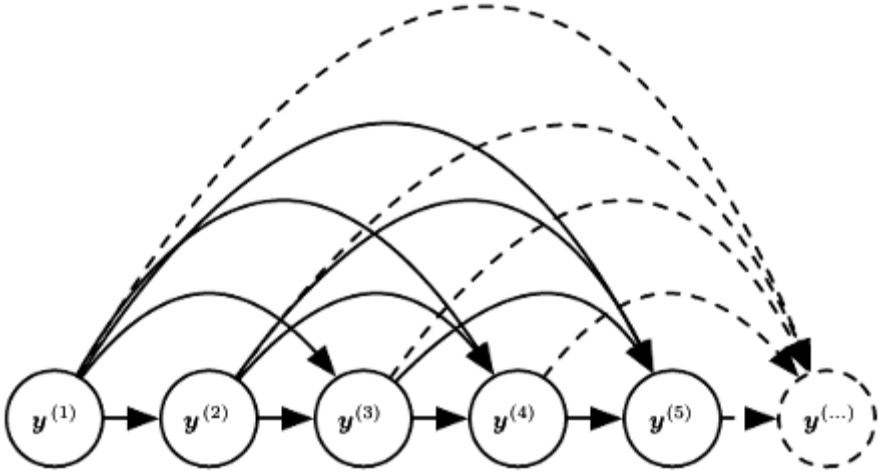


Figure 3.10 Fully connected graphical model for a sequence $y^{(1)},y^{(2)},\ldots,y(t),\ldots$:

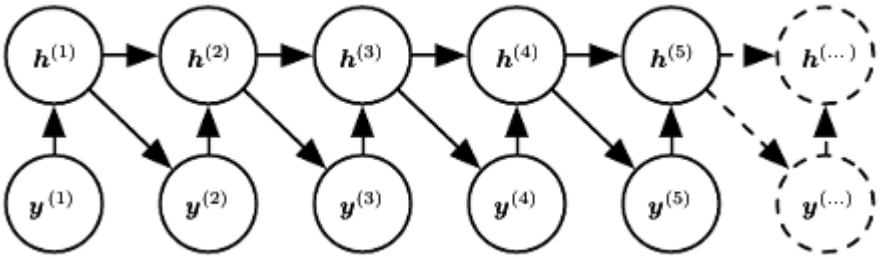


Figure 3.11 Introducing the state variable in the graphical model of the RNN

- The edges in a graphical model indicate which variables depend directly on other variables. Many graphical models aim to achieve statistical and computational efficiency by omitting edges that do not correspond to strong interactions.

- For example, it is common to make the Markov assumption that the graphical model should only contain edges from $\{y^{(t-k)}, \ldots, y^{(t-1)}\}$ to $y^{(t)}$, , rather than containing edges from the entire past history.
- One way to interpret an RNN as a graphical model is to view the RNN as defining a graphical model whose structure is the complete graph, able to represent direct dependencies between any pair of **y** values.
- The graphical model over the **y** values with the complete graph structure is shown in figure 3.10.
- The complete graph interpretation of the RNN is based on ignoring the hidden units $h^{(t)}$ by marginalizing them out of the model.
- The complete graph interpretation of the RNN is based on ignoring the hidden units $\mathbf{h}^{(t)}$ by marginalizing them out of the model.
- The price recurrent networks pay for their reduced number of parameters is that optimizing the parameters may be difficult.
- The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps.
- Equivalently, the assumption is that the conditional probability distribution over the variables at time t+1 given the variables at time t is stationary, meaning that the relationship between the previous time step and the next time step does not depend on t.
- In principle, it would be possible to use **t** as an extra input at each time step and let the learner discover any time-dependence while sharing as much as it can between different time steps.
- This would already be much better than using a different conditional probability distribution for each **t**, but the network would then have to extrapolate when faced with new values of **t**.
- To complete the view of an **RNN as a graphical model**, we must describe how to draw samples from the model.

- The main operation that we need to perform is simply to sample from the conditional distribution at each time step.
- However, there is one additional complication. The RNN must have some mechanism for determining the length of the sequence. This can be achieved in various ways.
- In the case when the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence.
- When that symbol is generated, the sampling process stops.
- In the training set, we insert this symbol as an extra member of the sequence, immediately after $\mathbf{x}^{(\tau)}$ in each training example.
- Another option is to introduce an **extra Bernoulli output** to the model that represents the decision to either continue generation or halt generation at each time step.
- This approach is more general than the approach of adding an extra symbol to the vocabulary, because it may be applied to any RNN, rather than only RNNs that output a sequence of symbols.

- Another way to determine the sequence length $\boldsymbol{\tau}$ is to add an extra output to the model that predicts the integer $\boldsymbol{\tau}$ itself.
- The model can sample a value of τ and then sample $\boldsymbol{\tau}$ steps worth of data.
- This approach requires adding an extra input to the recurrent update at each time step so that the recurrent update is aware of whether it is near the end of the generated sequence.
- This extra input can either consist of the value of τ or can consist of $\boldsymbol{\tau - t}$, the number of remaining time steps.
- Without this extra input, the RNN might generate sequences that end abruptly, such as a sentence that ends before it is complete.
- This approach is based on the decomposition.

$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}) = P(\tau) P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)} \mid \tau). \qquad (10.34)$$

5. Discuss in detail about Bidirectional Recurrent Neural Network.

Recurrent Neural Networks (RNNs)

- Recurrent Neural Networks (RNNs) are a particular class of neural networks that was created with the express ***purpose of processing sequential input***, including speech, text, and time series data.
- RNNs process data as a ***sequence of vectors*** rather than feedforward neural networks, which process data as a fixed-length vector. Each vector is processed depending on the hidden state from the previous phase.
- The network can store data from earlier steps in the sequence in a type of memory by computing the hidden state by taking into account both the current input and the hidden state from the previous phase. RNNs are thus well suited for jobs that call for knowledge of the context and connections among sequence elements.
- Even though conventional RNNs can handle variable-length sequences, they sometimes have trouble with the vanishing gradient problem. Gradients during backpropagation become extremely small at this point, making it challenging for the network to learn from the data. Many RNN versions, such LSTMs, and GRUs, which use gating methods to regulate the flow of information and enhance learning, have been created to address this problem.

Bi-directional Recurrent Neural Network

- An architecture of a neural network called a bidirectional recurrent neural network (BRNN) is made to process sequential data.
- In order for the network to use information from both the past and future context in its predictions, BRNNs process input sequences in both the forward and backward directions.
- This is the main distinction between BRNNs and conventional recurrent neural networks.
- A BRNN has ***two distinct recurrent hidden layers***, one of which processes the ***input sequence forward*** and the other of which ***processes it backward***.
- After that, the results from these hidden layers are collected and input into a prediction-making final layer. Any recurrent neural network cell, such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit, can be used to create the recurrent hidden layers.
- Compared to *conventional unidirectional recurrent neural networks*, the accuracy of the BRNN is improved since it can process information in both directions and account for both past and future contexts. Because the two hidden layers can

complement one another and give the final prediction layer more data, using two distinct hidden layers also offers a type of model regularisation.

- In order to update the model parameters, the gradients are computed for both the forward and backward passes of the backpropagation through the time technique that is typically used to train BRNNs.
- The input sequence is processed by the BRNN in a single forward pass at inference time, and predictions are made based on the combined outputs of the two hidden layers. layers. Refer figure 3.12.

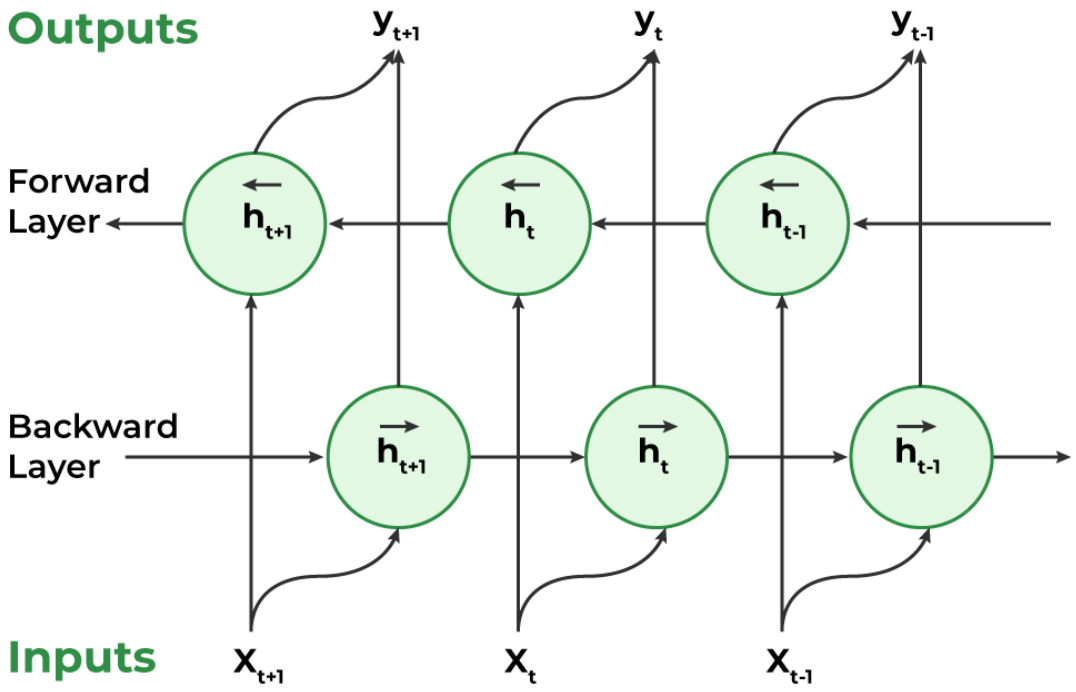


Figure 3.12 Bidirectional Recurrent Neural Network

Working of Bidirectional Recurrent Neural Network

1. **Inputting a sequence:** A sequence of data points, each represented as a vector with the same dimensionality, are fed into a BRNN. The sequence might have different lengths.
2. **Dual Processing:** Both the forward and backward directions are used to process the data. On the basis of the input at that step and the hidden state at step t-1, the hidden state at time step t is determined in the forward direction. The input at step t and the hidden state at step t+1 are used to calculate the hidden state at step t in a reverse way.
3. **Computing the hidden state:** A non-linear activation function on the weighted sum of the input and previous hidden state is used to calculate the hidden state

at each step. This creates a memory mechanism that enables the network to remember data from earlier steps in the process.

4. **Determining the output:** A non-linear activation function is used to determine the output at each step from the weighted sum of the hidden state and a number of output weights. This output has two options: it can be the final output or input for another layer in the network.
5. **Training:** The network is trained through a supervised learning approach where the goal is to minimize the discrepancy between the predicted output and the actual output. The network adjusts its weights in the input-to-hidden and hidden-to-output connections during training through backpropagation.

To calculate the output from an RNN unit, we use the following formula:

$$H_t \text{ (Forward)} = A(X_t * W_{XH} \text{ (forward)} + H_{t-1} \text{ (Forward)} * W_{HH} \text{ (Forward)} + b_H \text{ (Forward)}$$
$$H_t \text{ (Backward)} = A(X_t * W_{XH} \text{ (Backward)} + H_{t+1} \text{ (Backward)} * W_{HH} \text{ (Backward)} + b_H \text{ (Backward)}$$

where,
A = activation function,
W = weight matrix
b = bias

The hidden state at time t is given by a combination of H_t (Forward) and H_t (Backward). The output at any given hidden state is :

$$Y_t = H_t * W_{AY} + b_y$$

Applications of Bidirectional Recurrent Neural Network

Bi-RNNs have been applied to various natural language processing (NLP) tasks, including:

1. **Sentiment Analysis:** By taking into account both the prior and subsequent context, BRNNs can be utilized to categorize the sentiment of a particular sentence.
2. **Named Entity Recognition:** By considering the context both before and after the stated thing, BRNNs can be utilized to identify those entities in a sentence.
3. **Part-of-Speech Tagging:** The classification of words in a phrase into their corresponding parts of speech, such as nouns, verbs, adjectives, etc., can be done using BRNNs.
4. **Machine Translation:** BRNNs can be used in encoder-decoder models for machine translation, where the decoder creates the target sentence and the encoder analyses the source sentence in both directions to capture its context.

5. **Speech Recognition:** When the input voice signal is processed in both directions to capture the contextual information, BRNNs can be used in automatic speech recognition systems.

Advantages of Bidirectional RNN

- **Context from both past and future:** With the ability to process sequential input both forward and backward, BRNNs provide a thorough grasp of the full context of a sequence. Because of this, BRNNs are effective at tasks like sentiment analysis and speech recognition.
- **Enhanced accuracy:** BRNNs frequently yield more precise answers since they take both historical and upcoming data into account.
- **Efficient handling of variable-length sequences:** When compared to conventional RNNs, which require padding to have a constant length, BRNNs are better equipped to handle variable-length sequences.
- **Resilience to noise and irrelevant information:** BRNNs may be resistant to noise and irrelevant data that are present in the data. This is so because both the forward and backward paths offer useful information that supports the predictions made by the network.
- **Ability to handle sequential dependencies:** BRNNs can capture long-term links between sequence pieces, making them extremely adept at handling complicated sequential dependencies.

Disadvantages of Bidirectional RNN

- **Computational complexity:** Given that they analyze data both forward and backward, BRNNs can be computationally expensive due to the increased amount of calculations needed.
- **Long training time**: BRNNs can also take a while to train because there are many parameters to optimize, especially when using huge datasets.
- **Difficulty in parallelization:** Due to the requirement for sequential processing in both the forward and backward directions, BRNNs can be challenging to parallelize.
- **Overfitting:** BRNNs are prone to overfitting since they include many parameters that might result in too complicated models, especially when trained on short datasets.
- **Interpretability:** Due to the processing of data in both forward and backward directions, BRNNs can be tricky to interpret since it can be difficult to comprehend what the model is doing and how it is producing predictions.

Implementation of Bi-directional Recurrent Neural Network on NLP dataset

- There are multiple processes involved in training a bidirectional RNN on an NLP dataset, including data preprocessing, model development, and model training.
- Here is an illustration of a Python implementation using Keras and TensorFlow. Utilize the IMDb movie review sentiment classification dataset from Keras in this example. The data must first be loaded and preprocessed.

```
import warnings
warnings.filterwarnings('ignore')
from keras.datasets import imdb
from keras_preprocessing.sequence import pad_sequences

# let's load the dataset and then split
# it into training and testing sets
features = 2000
len = 50
(X_train, y_train),\
(X_test, y_test) = imdb.load_data(num_words=features)

# we are using pad sequences to a fixed length
X_train = pad_sequences(X_train, maxlen=len)
X_test = pad_sequences(X_test, maxlen=len)
```

Model Architecture

By using the high-level API of the Keras we will implement a Bidirectional Recurrent Neural Network model. This model will have 64 hidden units and 128 as the size of the embedding layer. While compiling a model we provide these three essential parameters:

- **optimizer** – This is the method that helps to optimize the cost function by using gradient descent.
- **loss** – The loss function by which we monitor whether the model is improving with training or not.
- **metrics** – This helps to evaluate the model by predicting the training and the validation data.

```
# Import the necessary modules from Keras:
from keras.models import Sequential
from keras.layers import Embedding,\
    Bidirectional, SimpleRNN, Dense

# Set the values for the embedding size and
# number of hidden units in the LSTM layer
embedding = 128
hidden = 64

# Create a Sequential model object
model = Sequential()
model.add(Embedding(features, embedding,
                    input_length=len))
model.add(Bidirectional(SimpleRNN(hidden)))
model.add(Dense(1, activation='sigmoid'))
model.compile('adam', 'binary_crossentropy',
              metrics=['accuracy'])
```

Model Training

As we have compiled our model successfully and the data pipeline is also ready so, we can move forward toward the process of training our BRNN.

```
#set batch size and number of epochs you want
batch_size = 32
epochs = 5

model.fit(X_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(X_test, y_test))
```

Output:

```
Epoch 1/5
782/782 [==============================] - 30s 36ms/step - loss: 0.5625 - accuracy: 0.6900 - val_loss: 0.4535 - val_accuracy: 0.7872
Epoch 2/5
782/782 [==============================] - 30s 38ms/step - loss: 0.4030 - accuracy: 0.8164 - val_loss: 0.4738 - val_accuracy: 0.7793
Epoch 3/5
782/782 [==============================] - 27s 34ms/step - loss: 0.3236 - accuracy: 0.8631 - val_loss: 0.5178 - val_accuracy: 0.7594
Epoch 4/5
782/782 [==============================] - 27s 35ms/step - loss: 0.2278 - accuracy: 0.9102 - val_loss: 0.6023 - val_accuracy: 0.7662
Epoch 5/5
782/782 [==============================] - 26s 33ms/step - loss: 0.1477 - accuracy: 0.9430 - val_loss: 0.7477 - val_accuracy: 0.7522
<keras.callbacks.History at 0x7f49f431c730>
```

Training progress of the BRNN epoch-by-epoch

Evaluate the Model

Model ready let's evaluate its performance on the validation data using different evaluation metrics. For this purpose, we will first predict the class for the validation data using this model and then compare the output with the true labels.

```
loss, accuracy = model.evaluate(X_test, y_test)
print('Test accuracy:', accuracy)
```

6. Explain in detail about Sequence to Sequence RNN.

Sequence to Sequence RNN

- Seq2Seq model or Sequence-to-Sequence model, is a machine learning architecture designed for tasks involving sequential data.
- It takes an input sequence, processes it, and generates an output sequence.
- The architecture consists of two fundamental components: an encoder and a decoder.
- Seq2Seq models have significantly improved the quality of machine translation systems making them an important technology.
- Before the arrival of Seq2Seq models, the machine translation systems relied on statistical methods and phrase-based approaches.
- The most popular approach was the use of phrase-based statistical machine translation (SMT) systems.
- That was not able to handle long-distance dependencies and capture global context.
- The seq2seq models are ***encoder-decoder models***.
- The ***encoder*** processes the input sequence and transforms it into a fixed-size hidden representation.
- The ***decoder*** uses the hidden representation to generate output sequence.
- The encoder-decoder structure allows them to handle input and output sequences of different lengths, making them capable to handle sequential data.
- Seq2Seq models are trained using a dataset of input-output pairs, where the input is a sequence of tokens, and the output is also a sequence of tokens.
- The model is trained to maximize the likelihood of the correct output sequence given the input sequence.
- Seq2Seq models have been widely used in NLP tasks due to their ability to handle variable-length input and output sequences.

Encoder and Decoder in Seq2Seq model

In the seq2seq model, the Encoder and the Decoder architecture plays a vital role in converting input sequences into output sequences. Let's explore about each block refer figure 3.13.

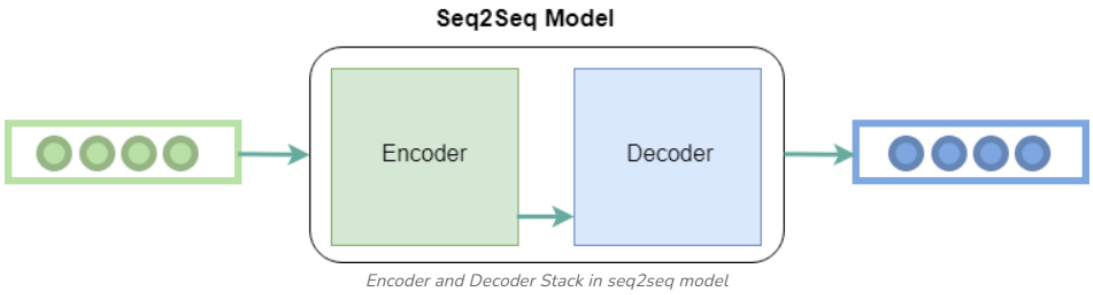


Figure 3.13 Encoder and Decoder in Seq2Seq model

Encoder

- Both the encoder and the decoder are Long short-term memory (LSTM) models (or sometimes GRU models)
- The encoder reads the input sequence and summarizes the information in something called the internal state vectors or context vectors.
- Discard the outputs of the encoder and only preserve the internal states. This context vector aims to encapsulate the information for all input elements to help the decoder make accurate predictions. Refer figure 3.14.
- The hidden states h_i are computed using the formula:

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

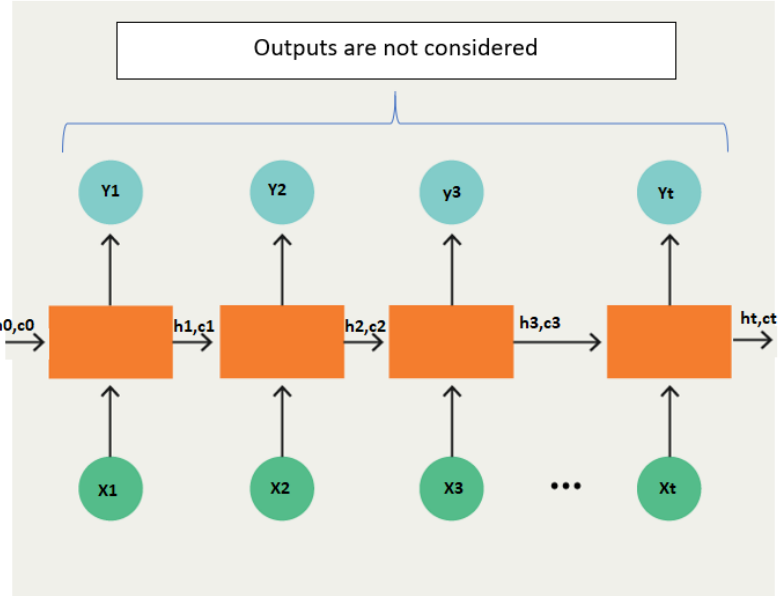


Figure 3.14 Encoder in Seq2Seq model

- The LSTM reads the data, one sequence after the other. Thus if the input is a sequence of length 't', we say that LSTM reads it in 't' time steps.

1. Xi = Input sequence at time step i.

2. hi and ci = The LSTM maintains two states ('h' for hidden state and 'c' for cell state) at each time step. Combined, these are the internal states of the LSTM at time step i.

3. Yi = Output sequence at time step i. Yi is actually a probability distribution over the entire vocabulary generated by using a softmax activation. Thus, each Yi is a vector of size "vocab_size" representing a probability distribution.

Decoder Block

- The decoder is a long-short-term memory (LSTM) whose initial states are initialized to the final states of the Encoder LSTM. In other words, the encoder sector of the encoder's final cell is input to the first cell of the decoder network. Using these initial states, the decoder starts generating the output sequence, and these outputs are also considered for future outputs.
- A stack of several LSTM units where each predicts an output y_t at a time step t.
- Each recurrent unit accepts a hidden state from the previous unit and produces an output and its hidden state.
- Any hidden state *h_i* is computed using the formula:

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

- The output *y_t* at time step *t* is computed using the formula:

$$y_t = softmax(W^S h_t)$$

- We calculate the outputs using the hidden state at the current time step and the respective weight W(S). Softmax creates a probability vector to help us determine the final output (e.g., word in the question-answering problem).

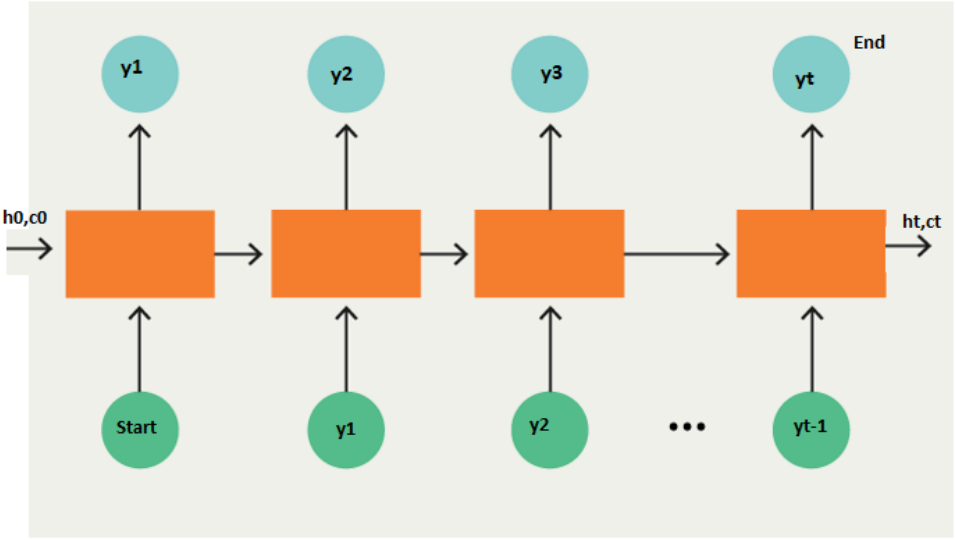


Figure 3.15 Decoder Block

Drawbacks of Encoder-Decoder Models

There are two primary drawbacks to this architecture, both related to length.

1. Firstly, as with humans, this architecture has ***minimal memory***. Thinking of neural networks in terms of the "lossy compression" they must perform is sometimes quite useful.

2. Second, as a general rule of thumb, the deeper a neural network is, the harder it is to train. For recurrent neural networks, the longer the sequence is, the deeper the neural network is along the time dimension. This results in vanishing gradients, where the gradient signal from the objective that the recurrent neural network learns from disappears as it travels backward.

7. Explain in detail about Deep Recurrent Networks.

Deep RNN (Recurrent Neural Network)

- Deep RNN (Recurrent Neural Network) refers to a neural network architecture that has multiple layers of recurrent units.
- A Deep RNN takes the output from one layer of recurrent units and feeds it into the next layer, allowing the network to capture more complex relationships between the input and output sequences.
- The number of layers in a deep RNN can vary depending on the complexity of the problem being solved, and the number of hidden units in each layer can also be adjusted.

- The standard method for building this sort of deep RNN is strikingly simple: we stack the RNNs on top of each other. Given a sequence of length T, the first RNN produces a sequence of outputs, also of length T.
- Illustrate a deep RNN with L hidden layers. Each hidden state operates on a sequential input and produces a sequential output. Moreover, any RNN cell (white box in figure 3.16) at each time step depends on both the same layer's value at the previous time step and the previous layer's value at the same time step.

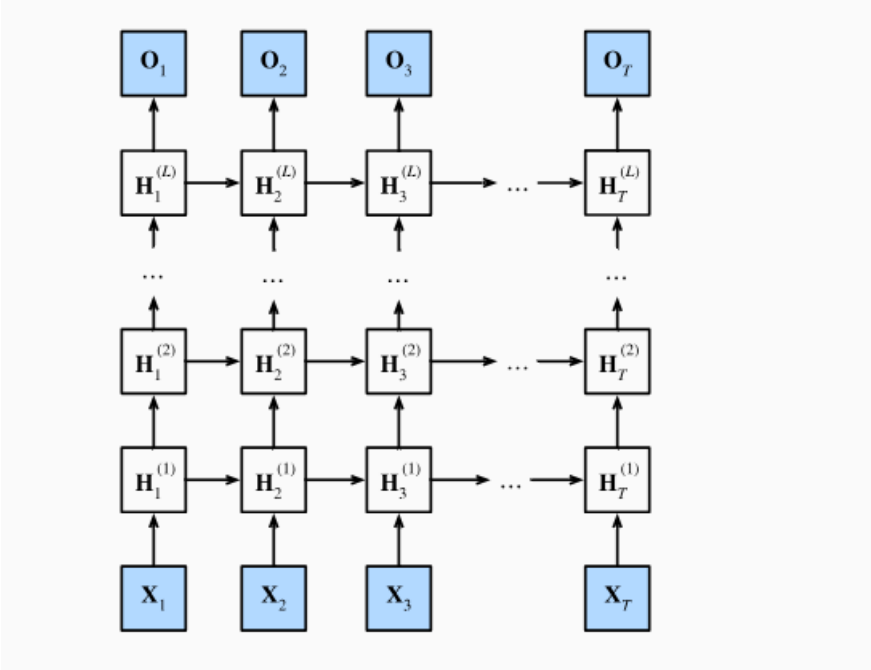


Figure 3.16 Architecture of a deep RNN

- Normally, suppose that we have a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples =n; number of inputs in each example =d) at time step t.
- At the same time step, let the hidden state of the lth hidden layer $(l = 1, \ldots, L)$ be $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (number of hidden units =h) and the output layer variable be $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q).
- Setting $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, the hidden state of the lth hidden layer that uses the activation function ϕl is calculated as follows:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{\text{xh}}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{\text{hh}}^{(l)} + \mathbf{b}_{\text{h}}^{(l)}), \quad (10.3.1)$$

where the weights $\mathbf{W}_{\text{xh}}^{(l)} \in \mathbb{R}^{h \times h}$ and $\mathbf{W}_{\text{hh}}^{(l)} \in \mathbb{R}^{h \times h}$, together with the bias $\mathbf{b}_{\text{h}}^{(l)} \in \mathbb{R}^{1 \times h}$, are the model parameters of the lth hidden layer.

- At the end, the calculation of the output layer is only based on the hidden state of the final Lth hidden layer:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}, \tag{10.3.2}$$

where the weight $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times q}$ and the bias $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer.

- Just as with MLPs, the number of hidden layers L and the number of hidden units h are hyperparameters that we can tune.
- Common RNN layer widths (h) are in the range (64,2056), and common depths (L) are in the range (1,8). In addition, we can easily get a deep-gated RNN by replacing the hidden state computation in figure 3.14 with that from an LSTM or a GRU.

Examples

Deep RNNs have been successfully applied in various real-life applications. Here are a few examples:

1. Speech Recognition: Deep RNNs have been used to build speech recognition systems, such as Google's Speech API, Amazon's Alexa, and Apple's Siri. These systems use deep RNNs to convert speech signals into text.
2. Natural Language Processing (NLP): Deep RNNs are used in various NLP applications, such as language translation, sentiment analysis, and text classification. For example, Google Translate uses a deep RNN to translate text from one language to another.
3. Music Generation: Deep RNNs have been used to generate music, such as Magenta's MusicVAE, which uses a deep RNN to generate melodies and harmonies.
4. Image Captioning: Deep RNNs are used in image captioning systems, such as Google's Show and Tell, which uses a deep RNN to generate captions for images.
5. Autonomous Driving: Deep RNNs have been used in autonomous driving systems to predict the behaviour of other vehicles on the road, such as the work done by Waymo.

Recursive Neural Networks

- Recursive Neural Networks (RvNNs) are a class of deep neural networks that can learn detailed and structured information.
- With RvNN, you can get a structured prediction by recursively applying the same set of weights on structured inputs.
- The word recursive indicates that the neural network is applied to its output.
- Recursive neural networks are capable of handling hierarchical data because of their ***indepth tree-like structure***.
- In a tree structure, parent nodes are created by joining child nodes.
- There is a ***weight matrix*** for every child-parent bond, and comparable children have the same weights.
- To allow for recursive operations and the use of the same weights, the number of children for each node in the tree is fixed.
- When it's necessary to parse a whole sentence, RvNNs are employed.

- We add the ***weight matrices'*** **(W i)** and ***children's*** **(C i)** products and use the transformation f to determine the parent node's representation.

$$h = f\left(\sum_{i=1}^{i=c} W_i C_i\right)$$

, refers to the number of children.

Working Principals of RvNN

Some of the key-working principals of RvNN is discussed below:

1. **Recursive Structure Handling:** RvNN is designed to handle recursive structures which means it can naturally process hierarchical relationships in data by combining information from child nodes to form representations for parent nodes.
2. **Parameter Sharing:** RvNN often uses shared parameters across different levels of the hierarchy which enables the model to generalize well and learn from various parts of the input structure.

3. **Tree Traversal:** RvNN traverses the tree structure in a bottom-up or top-down manner by simultaneously updating node representations based on the information gathered from their children.
4. **Composition Function:** The composition function in RvNN combines information from child nodes to create a representation for the parent node. This function is crucial in capturing the hierarchical relationships within the data.

Best suited neural network models for recursive data

There are several popular neural network models which are widely used for handling recursive data. Some models are discussed below:

- **Recursive Neural Network (RvNN):** This is already discussed in the article. This model is specifically designed for processing tree-structured data, capturing dependencies within nested or hierarchical relationships.
- **Tree-LSTM (Long Short-Term Memory):** It is an extension of the traditional LSTM architecture, adapted to handle tree structures more effectively.
- **Graph Neural Networks (GNNs):** This model is particularly designed for processing graph-structured data, which includes recursive structures. GNNs are particularly useful for tasks involving relationships between entities.

Difference between Recursive Neural Network and CNN

RvNN	CNN
It is designed to process hierarchical or tree-structured data, capturing dependencies within recursively structured information.	It primarily used for grid-structured data like images, where convolutional layers are applied to local receptive fields.
It processes data in a recursive manner, combining information from child nodes to form representations for parent nodes.	It uses convolutional layers to extract local features and spatial hierarchies from input data.
It is suitable for tasks involving nested structures like natural language parsing or molecular structure analysis.	It excels in tasks related to computer vision, image recognition and other grid-based data.

Difference between RvNN and RNN

RvNN	RNN
It is designed for sequential data like time series or sequences of words.	It is specially designed to tailor for hierarchical data structures like trees.
It processes sequences by maintaining hidden states which capture temporal dependencies.	It processes hierarchical structures by recursively combining information from child nodes.
Commonly used in tasks like natural language processing, speech recognition and time series prediction.	Commonly used in applications which involves hierarchical relationships like parsing in NLP, analyzing molecular structures or image segmentation.
It has linear topology where information flows sequentially through time steps.	It has a tree-like or hierarchical topology which allows them to capture nested relationships within the data.

8. Discuss in detail about Long Term Dependencies problem in RNN.

- Long-term dependencies are the situations where the output of an RNN depends on the input that occurred many time steps ago.
- For instance, consider the sentence ***"The cat, which was very hungry, ate the mouse".***
- To understand the meaning of this sentence, need to remember that the cat is the subject of the verb ate, even though they are separated by a long clause.
- This is a long-term dependency, and it can affect the performance of an RNN that tries to generate or analyze such sentences.
- Long-term dependencies are hard to learn in RNNs because of the ***vanishing or exploding gradient problem.***
- During ***backpropagation***, gradients can become extremely small or large, making it difficult for the network to update the weights effectively.
- One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform

the understanding of the present frame. If RNNs could do this, they'd be extremely useful.

- But can they? It depends. For some tasks, we only need to look at recent information.
- For example, consider a language model trying to predict the next word in a sentence based on the previous words and sentences.
- If we are trying to predict the last word in ***"the clouds are in the sky,"*** we don't need any previous sentences, – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information. Refer figure 3.17.

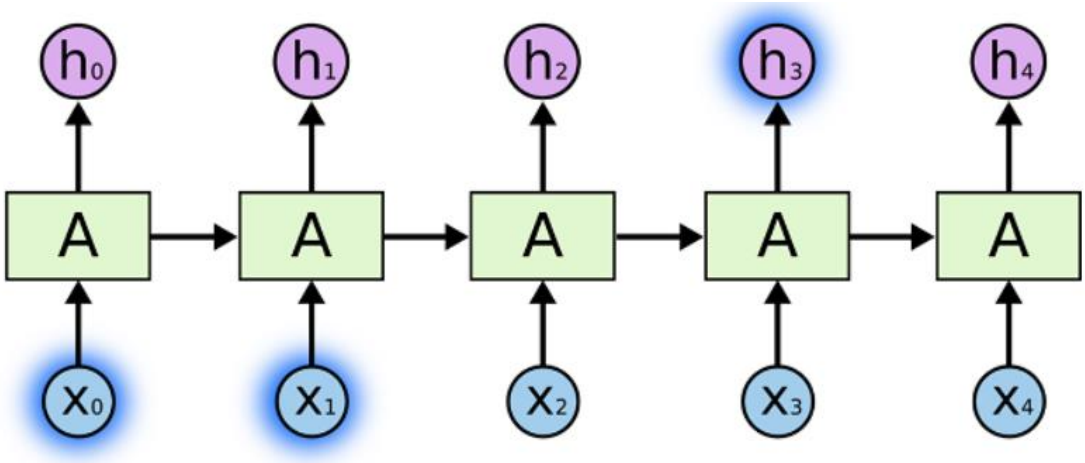


Figure 3.17 RNNs learn to use the past information.

- There are also cases where we need more context. Consider trying to predict the last word in the text ***"I speak fluent French."***
- Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context the sentence, like a reference to France, from sentences further back.
- It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- Unfortunately, as that gap grows, RNNs become unable to learn to connect the information. Refer figure 3.18.

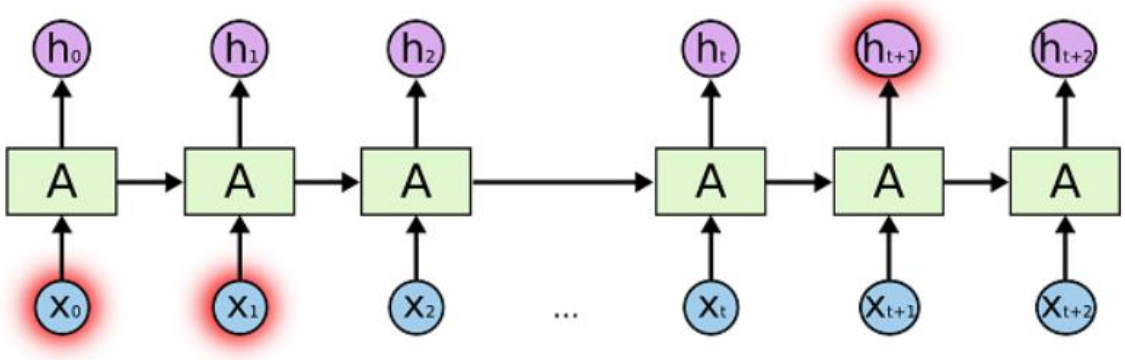


Figure 3.18 Understanding the long-term dependency situation in RNN

- In theory, RNNs are absolutely capable of handling such "long-term dependencies."
- A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them.
- To put it simple, when we train RNN, we will also use backpropagation algorithm to adjust error of the parameters. Let's take gradient at t=3 as an example, the derivative will look like this:

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \left(\prod_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W}$$

- In RNN, we need to sum up the contributions of each time step to the gradient. In other words, because W is used in every step up to the output we care about, we need to backpropagate gradients from t=3 through the network all the way to t=0 in this case as shown in figure 3.19.

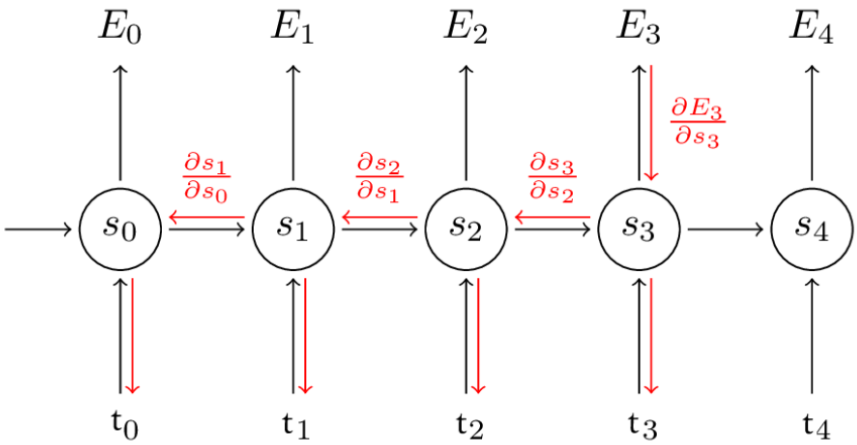


Figure 3.19 Backpropagate gradients through the network

- Note that this is exactly the same as the standard backpropagation algorithm that we use in deep Feedforward Neural Networks.
- The key difference is that we sum up the gradients for W across each time step. This should also give you an idea of why standard RNNs are hard to train: Sequences (sentences) can be quite long, perhaps 20 words or more, and thus you need to back-propagate through many layers. It is natural for this error signal to get muddied up by noise and imprecision. In practice many people truncate the backpropagation to a few steps.

9. Define Leaky Units. Explain in detail about Skip connections and dropouts.

Leaky Units

- A Leaky Rectified Linear Unit (Leaky ReLU) is an activation function where the negative section allows a small gradient instead of being completely zero, helping to reduce the risk of overfitting in neural networks.

Skip connections

- The need for deeper networks emerges while handling ***complex tasks***.
- However, training a deep neural net has a lot of complications not only limited to overfitting and high computation costs but also has some non-trivial problems. In this article, we will solve some complex deep learning problems using skip connections.
- The beauty of *deep* neural networks is that they can learn complex functions more efficiently than their shallow counterparts.
- While training deep neural nets, the performance of the model drops down with the increase in depth of the architecture. This is known as the **degradation problem**. But, what could be the reasons for the saturation inaccuracy with the increase in network depth? Let us try to understand the reasons behind the degradation problem.

Deeper Network Performance Analysis: Overfitting Discarded

- One of the possible reasons could be overfitting. The model tends to overfit with the increase in depth but that's not the case here. As you can infer from the

below figure, the deeper network with 56 layers has more training error than the shallow one with 20 layers. **The deeper model doesn't perform as well as the shallow one.** Clearly, overfitting is not the problem here.

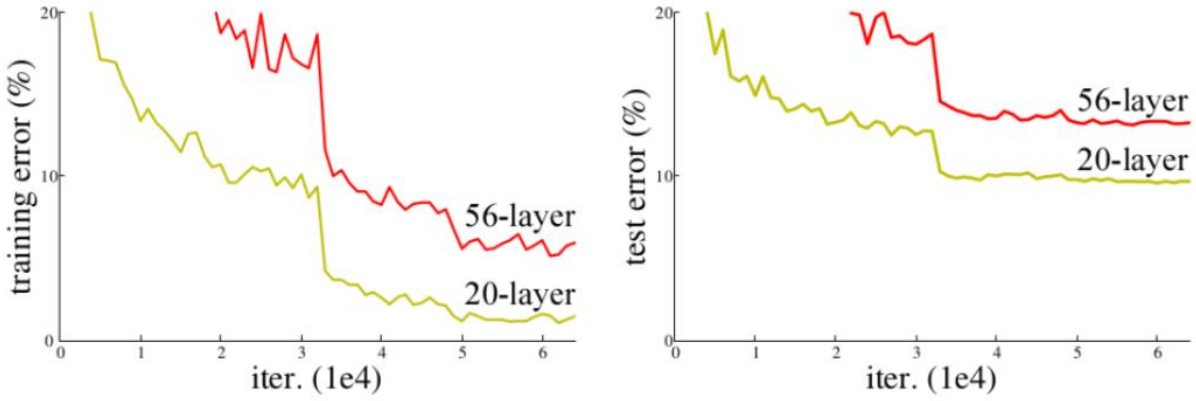


Figure 3.20 Train and test error for 20-layer and 56-layer NN

Gradient Issues in ResNet Construction

- Another possible reason can be vanishing gradient and/or exploding gradient problems.

- However, the authors of ResNet. argued that the use of Batch Normalization and proper initialization of weights through normalization ensures that the gradients have healthy norms.
- Consider a shallow neural network that was trained on a dataset.
- Also, consider a deeper one in which the initial layers have the same weight matrices as the shallow network (the blue colored layers in the below figure 3.19) with added some extra layers (green colored layers). We set the weight matrices of the added layers as *identity matrices* (identity mappings).

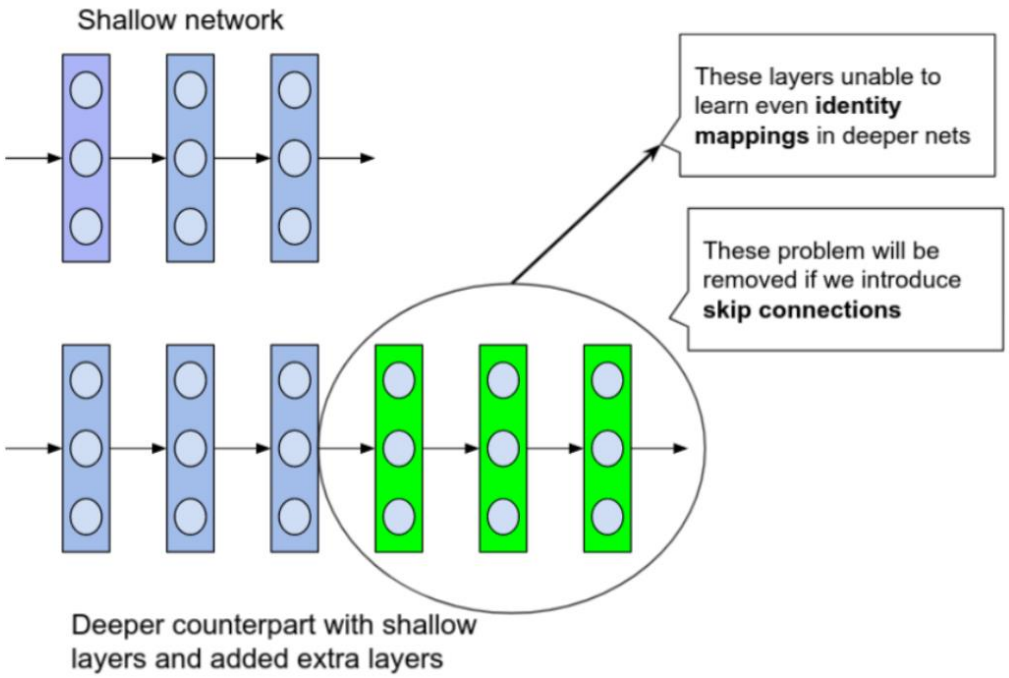


Figure 3.21 Diagram explaining the construction

- From this construction, the deeper network should not produce any higher training error than its shallow counterpart because we are actually using the shallow model's weight in the deeper network with added identity layers.

- But experiments prove that the deeper network produces high training error comparing to the shallow one. This states the **inability of deeper layers to learn even identity mappings**.

- The degradation of training accuracy indicates that *not all systems are similarly easy to optimize.*

- One of the primary reasons is due to random initialization of weights with a mean around zero, L1, and L2 regularization. As a result, the weights in the model would always be around zero and thus the deeper layers can't learn identity mappings as well.

Skip Connections

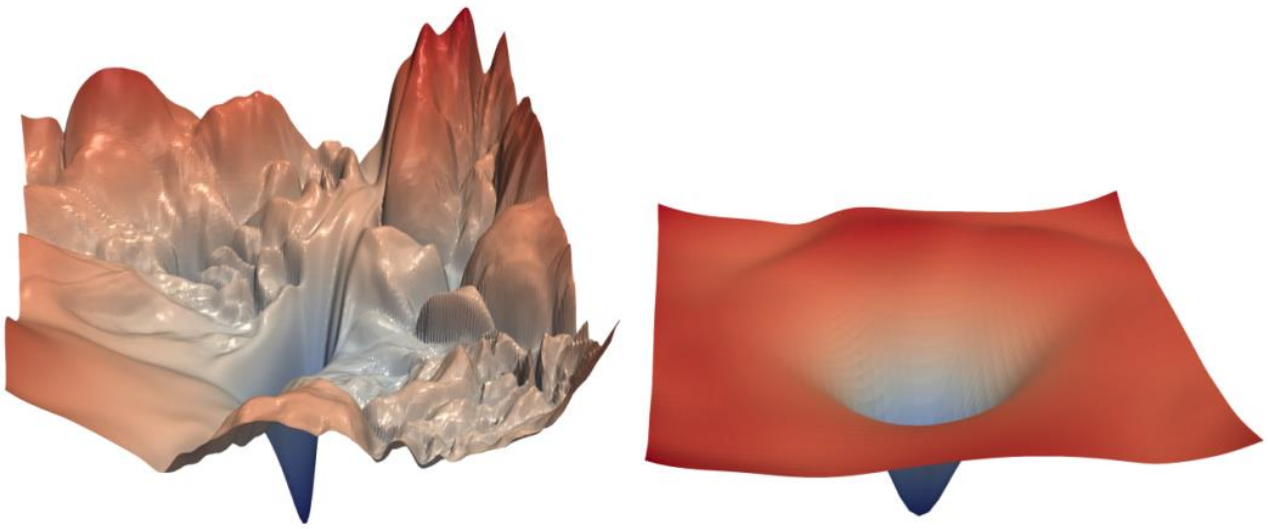
- *Skip Connections* (or Shortcut Connections) as the name suggests *skips some of the layers in the neural network and feeds the output of one layer as the input to the next layers.*

- Skip Connections were introduced to solve different problems in different architectures. In the case of ResNets, skip connections solved the *degradation*

problem that we addressed earlier whereas, in the case of DenseNets, it ensured **feature reusability**.

How do Skip Connections Work?

- Skip connections were introduced in literature even before residual networks. For example, **Highway Networks** had skip connections with *gates* that controlled and learned the flow of information to deeper layers.
- This concept is similar to the gating mechanism in LSTM. Although ResNets is actually a special case of Highway networks, the performance isn't up to the mark comparing to ResNets.
- This suggests that it's better to keep the gradient highways *clear* than to go for any gates – simplicity wins here!
- Neural networks can learn any functions of arbitrary complexity, which could be high-dimensional and non-convex.
- Visualizations have the potential to help us answer several important questions about why neural networks work. And there is actually some nice work done by Li et al. which enables us to visualize the complex loss surfaces.
- The results from the networks with skip connections are even more surprising! Take a look at them.



(a) without skip connections (b) with skip connections

Figure 3.22 The loss surfaces of ResNet-56 with and without skip connections

- The loss surface of the neural network with skip connections is smoother and thus leading to faster convergence than the network without any skip connections. Let's see the variants of skip connections in the next section.

Variants of Skip Connections

- Skip Connections can be used in 2 fundamental ways in Neural Networks: ***Addition and Concatenation.***

Residual Networks (ResNets)

- In ResNets, the information from the initial layers is passed to deeper layers by matrix addition.
- This operation doesn't have any additional parameters as the output from the previous layer is added to the layer ahead. A single residual block with skip connection as shown in figure 3.23.

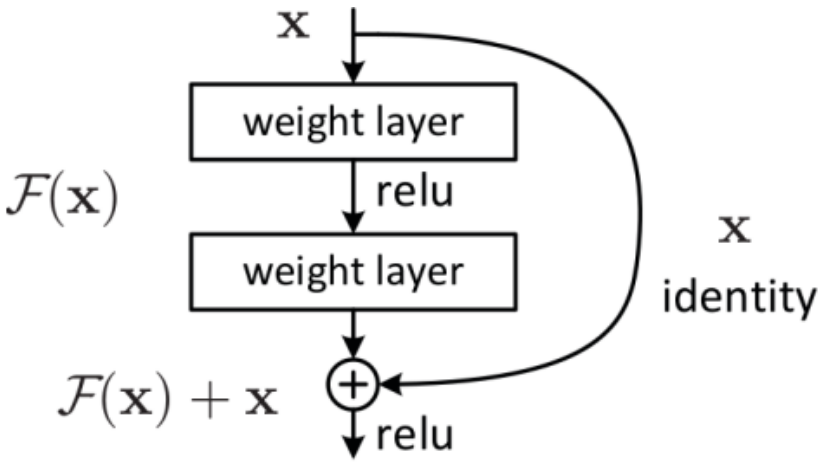


Figure 3.23 A residual block

Dropout in Neural Networks

- The concept of Neural Networks is inspired by the neurons in the human brain and scientists wanted a machine to replicate the same process.
- This craved a path to one of the most important topics in Artificial Intelligence.
- A Neural Network (NN) is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain.

- Since such a network is created artificially in machines, we refer to that as Artificial Neural Networks (ANN

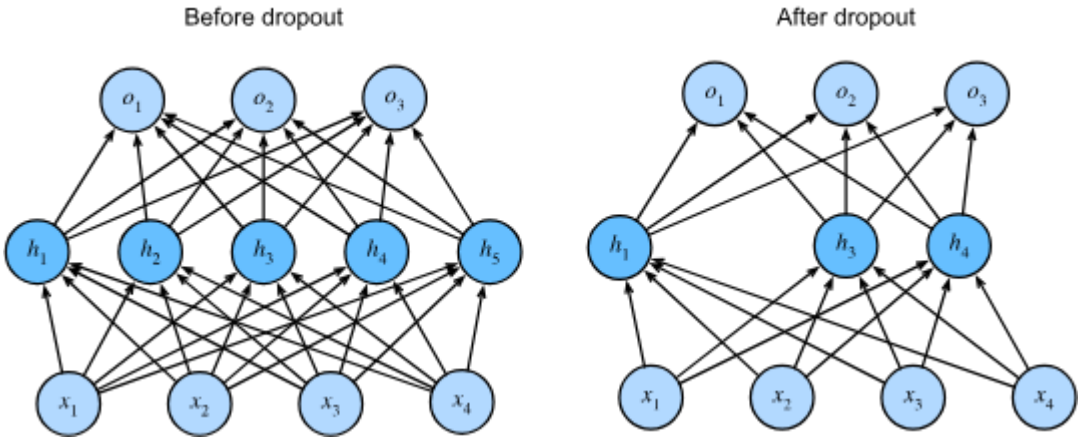
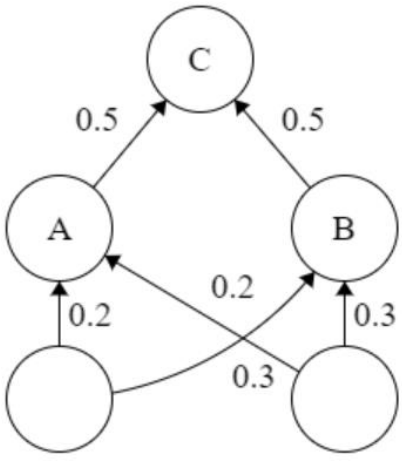


Figure 3.24 **MLP before and after dropout.**

```
tf.keras.layers.Dropout(
    rate
)

# rate: Float between 0 and 1.
# The fraction of the input units to drop.
```


Problem: When a fully-connected layer has a large number of neurons, co-adaptation is more likely to happen. Co-adaptation refers to when multiple neurons in a layer extract the same, or very similar, hidden features from the input data. This can happen when the connection weights for two different neurons are nearly identical.

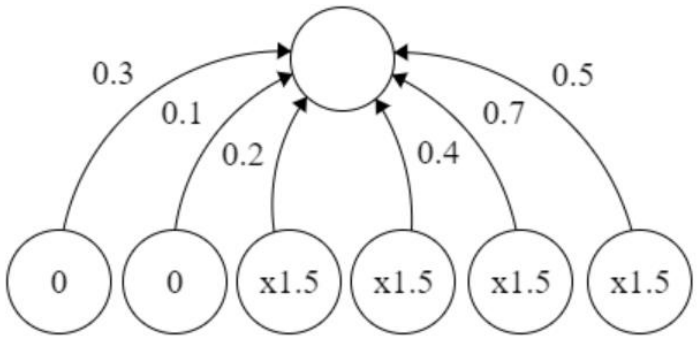


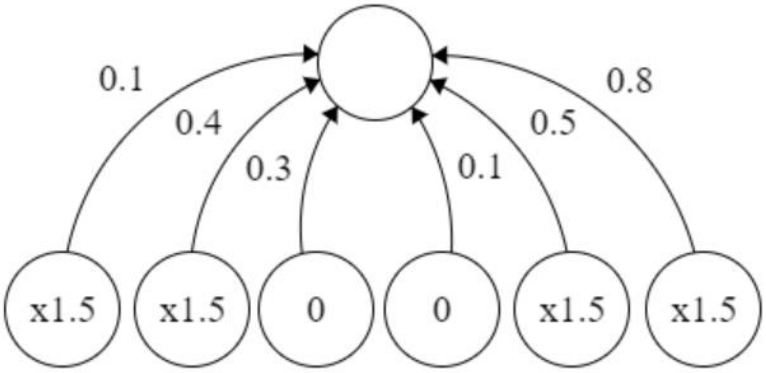
This poses two different problems to our model:

- Wastage of machine's resources when computing the same output.

- If many neurons are extracting the same features, it adds more significance to those features for our model. This leads to overfitting if the duplicate extracted features are specific to only the training set.

Solution to the problem: As the title suggests, we use dropout while training the NN to minimize co-adaptation. In dropout, we randomly shut down some fraction of a layer's neurons at each training step by zeroing out the neuron values. The fraction of neurons to be zeroed out is known as the dropout rate, The remaining neurons have their values multiplied by so that the overall sum of the neuron values remains the same.





- The two images represent dropout applied to a layer of 6 units, shown at multiple training steps.
- The dropout rate is 1/3, and the remaining 4 neurons at each training step have their value scaled by x1.5.
- Thereby, we are choosing a random sample of neurons rather than training the whole network at once.
- This ensures that the co-adaptation is solved and they learn the hidden features better.

Why dropout works?

- By using dropout, in every iteration, you will work on a smaller neural network than the previous one and therefore, it approaches regularization.
- Dropout helps in shrinking the squared norm of the weights and this tends to a reduction in overfitting.

Dropout can be applied to a network using TensorFlow APIs as follows:

```
tf.keras.layers.Dropout(
    rate
)

# rate: Float between 0 and 1.
# The fraction of the input units to drop.
```

9. Discuss in detail about Gated Architecture: LSTM.

Long Short-Term Memory

- **Long Short-Term Memory** is an improved version of recurrent neural network designed by Hochreiter & Schmidhuber.
- A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies.
- **LSTMs model** address this problem by introducing a memory cell, which is a container that can hold information for an extended period.
- LSTM architectures are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, and time series forecasting.

The architecture of LSTM

LSTMs architecture deal with both Long Term Memory (LTM) and Short Term Memory (STM) and for making the calculations simple and effective it uses the concept of gates.

- **Forget Gate:** LTM goes to forget gate and it forgets information that is not useful.
- **Learn Gate:** Event (current input) and STM are combined together so that necessary information that we have recently learned from STM can be applied to the current input.

- **Remember Gate:** LTM information that we haven't forget and STM and Event are combined together in Remember gate which works as updated LTM.
- **Use Gate:** This gate also uses LTM, STM, and Event to predict the output of the current event which works as an updated STM.

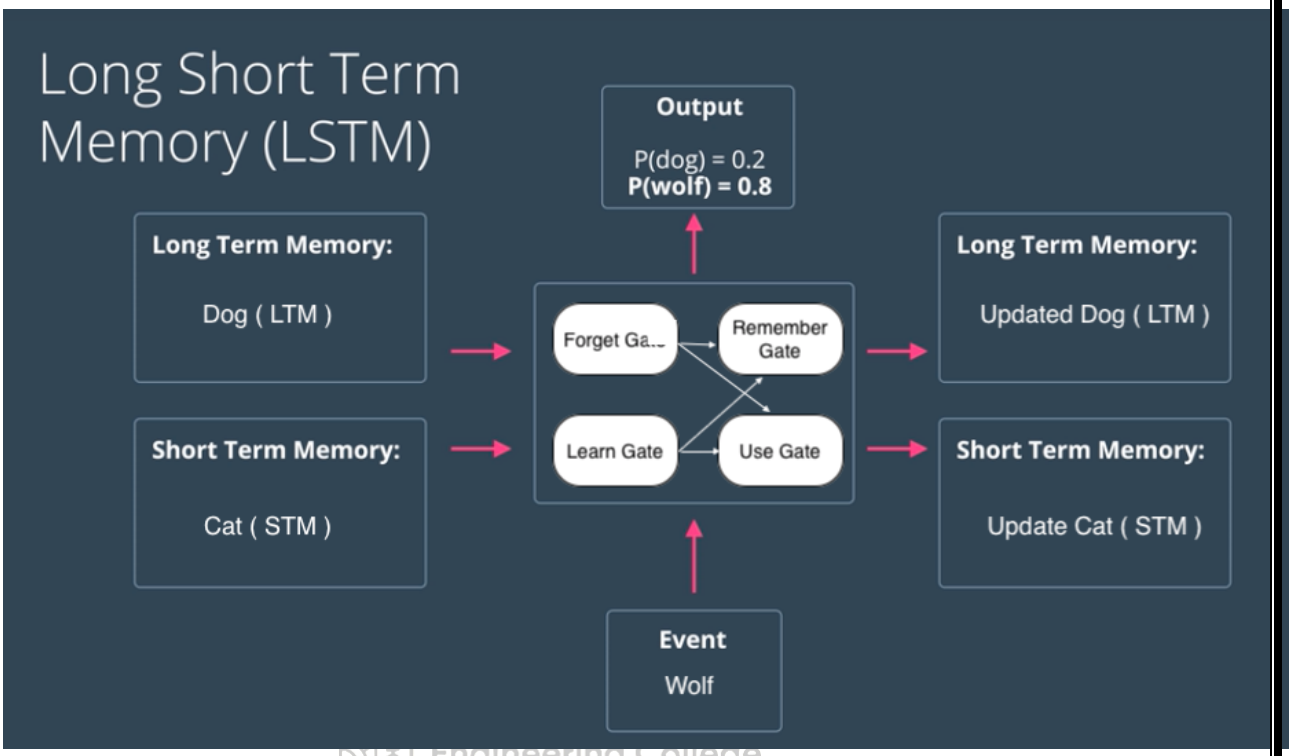


Figure 3.23: Remember GateSource

The figure 3.23 shows the simplified architecture of LSTMs. The actual mathematical architecture of LSTM is represented using the following figure:

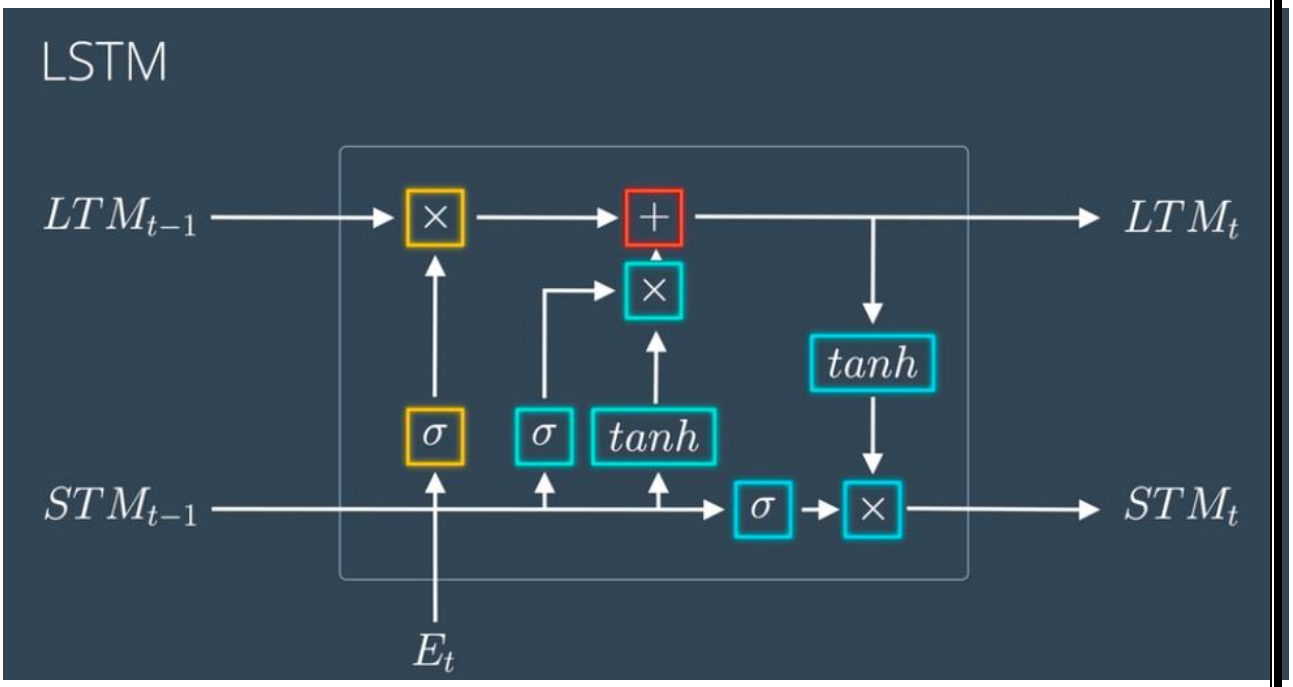


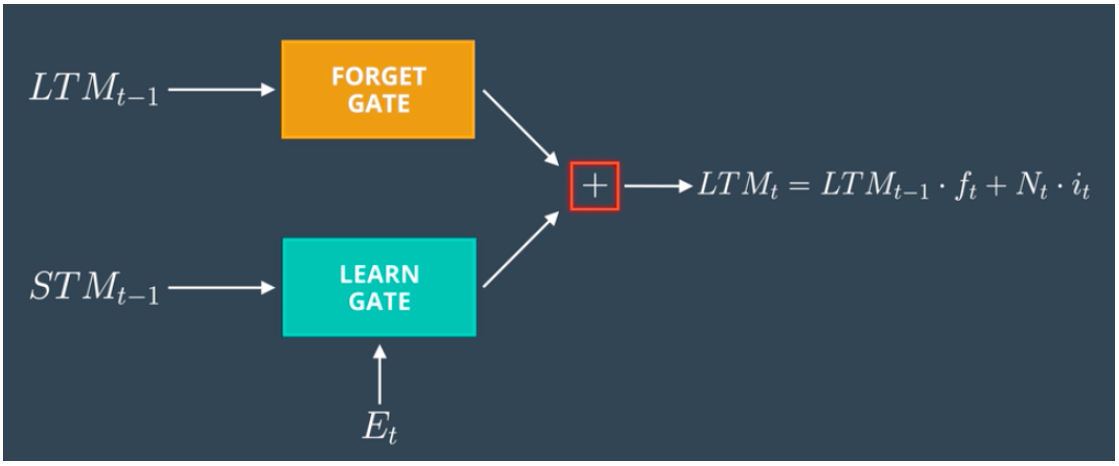
Figure 3.26: LSTM architecture diagram

don't go haywire with this architecture we will break it down into simpler steps which will make this a piece of cake to grab.

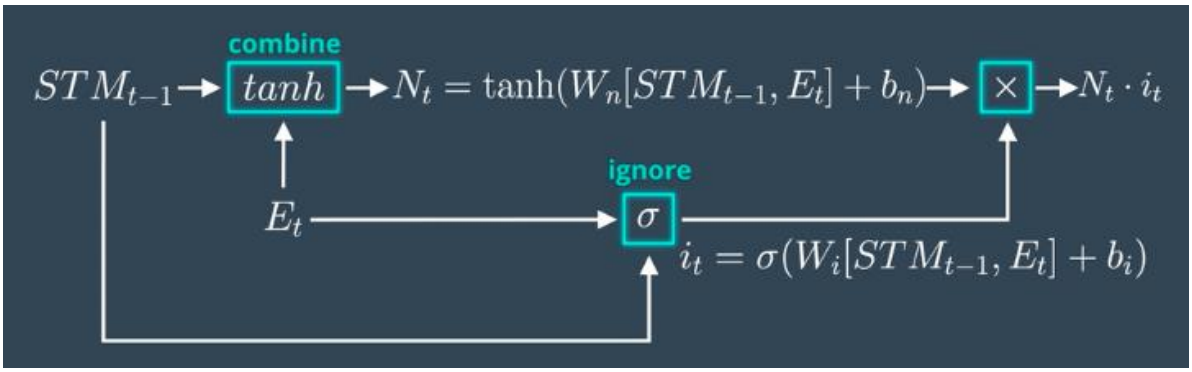
Breaking Down the Architecture of LSTM

1. Learn Gate

Takes Event (E_t) and Previous Short Term Memory (STM_{t-1}) as input and keeps only relevant information for prediction.



Calculation



- The model joins the Previous Short Term Memory STMt-1 and the Current Event vector Et together [STMt-1, Et], multiplies the joined vector with the weight matrix Wn, adds some bias to the result, passes it through the tanh (hyperbolic tangent) function to introduce non-linearity, and finally creates a matrix Nt.
- For ignoring insignificant information we calculate one Ignore Factor i_t, for which we join Short Term Memory STM_{t-1} and Current Event vector E_t and multiply with weight matrix W_i and passed through Sigmoid activation function with some bias.
- Learn Matrix N_t and Ignore Factor i_t is multiplied together to produce learn gate result.

2. The Forget Gate

Takes Previous Long Term Memory (LTM_{t-1}) as input and decides on which information should be kept and which to forget.

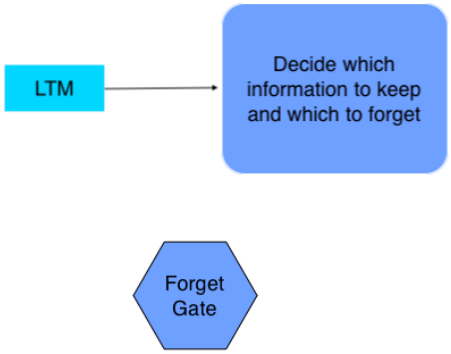
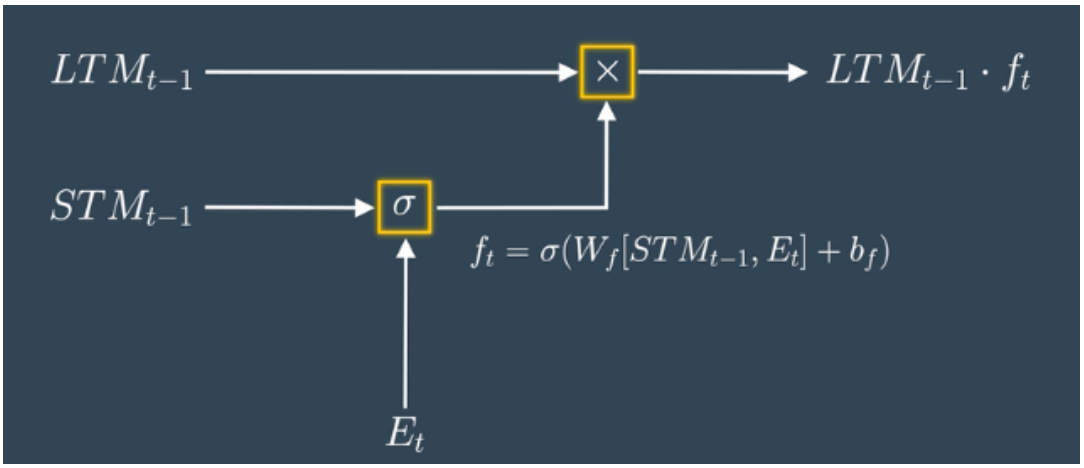


Figure 3.26: Forget Gate

Calculation



- The model joins the Previous Short Term Memory STMt-1 and the Current Event vector Et together [STMt-1, Et], multiplies them with the weight matrix Wf, and passes the result through the Sigmoid activation function with some bias to form the Forget Factor ft.
- The model multiplies the Forget Factor ft with the Previous Long Term Memory (LTMt-1) to produce the forget gate output.

3. The Remember Gate

Combine Previous Short Term Memory (STM_{t-1}) and Current Event (E_t) to produce output.

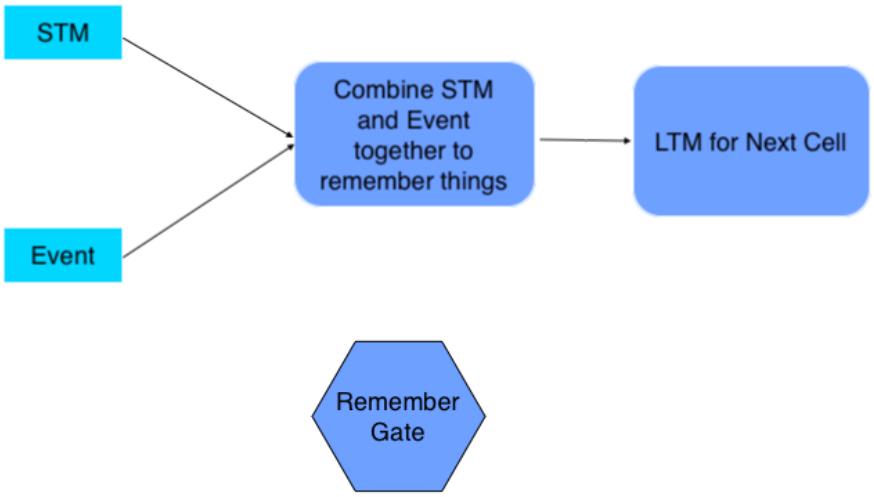
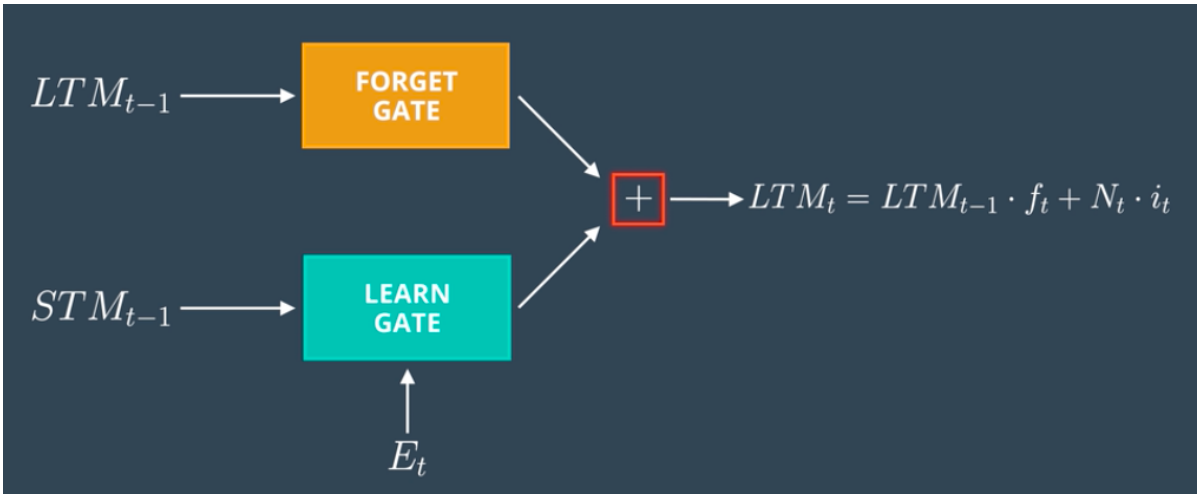


Figure: 3.27 Remember Gate

Calculation



- The output of Forget Gate and Learn Gate are added together to produce an output of Remember Gate which would be LTM for the next cell.

4. The Use Gate

Combine important information from Previous Long Term Memory and Previous Short Term Memory to create STM for next and cell and produce output for the current event.

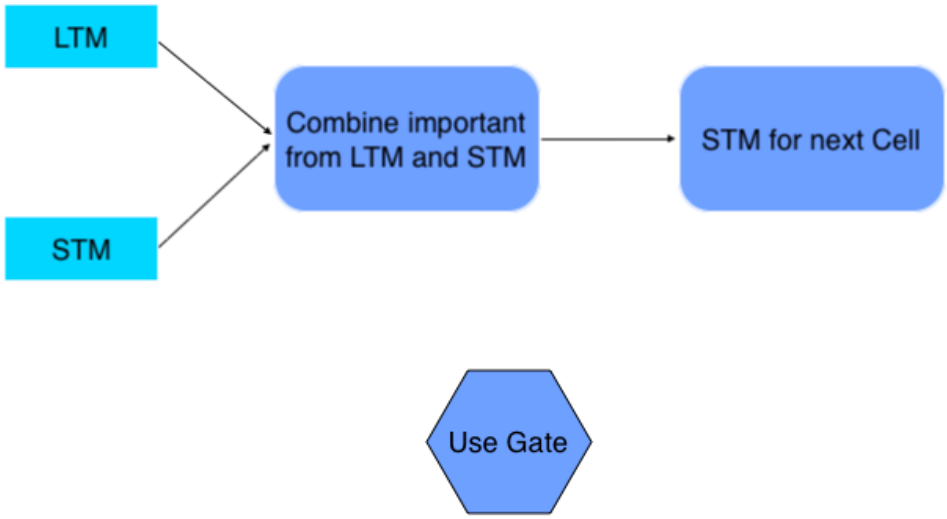
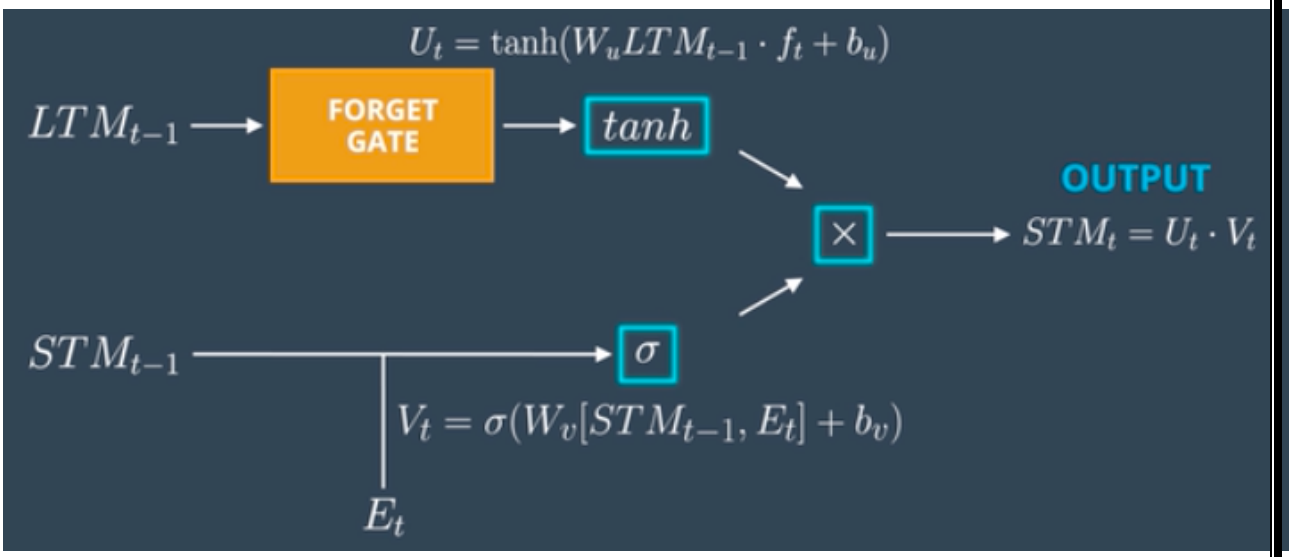


Figure 3.28 Use Gate

Calculation



- The model passes the Previous Long Term Memory (LTM-1) through the Tangent activation function with some bias to produce Ut.
- The model joins the Previous Short Term Memory (STMt-1) and the Current Event (Et) together and passes them through the Sigmoid activation function with some bias to produce Vt.
- The model multiplies the Output Ut and Vt together to produce the output of the use gate, which also serves as the STM for the next cell.
- Training LSTMs with their lstm model architecture removes the vanishing gradient problem but faces the exploding gradient issue. The vanishing gradient causes weights to become too small, underfitting the model. The exploding gradient makes weights too large, overfitting the model.

- LSTMs can be trained using Python frameworks like TensorFlow, PyTorch, and Theano. However, training deeper LSTM networks with the architecture of lstm in deep learning requires GPU hardware, similar to RNNs.
- The lstm model architecture enables LSTMs to handle long-term dependencies effectively. This makes them widely used for language generation, voice recognition, image OCR, and other tasks leveraging the lstm model architecture. Additionally, the architecture of lstm in deep learning is gaining traction in object detection, especially scene text detection.
- LSTMs find crucial applications in language generation, voice recognition, and image OCR tasks. Their expanding role in object detection heralds a new era of AI innovation. Both the lstm model architecture and architecture of lstm in deep learning enable these capabilities. Despite being complex, LSTMs represent a significant advancement in deep learning models.

SYLLABUS:

UNIT IV MODEL EVALUATION **8**

Performance metrics -- Baseline Models -- Hyperparameters: Manual Hyperparameter -- Automatic Hyperparameter -- Grid search -- Random search -- Debugging strategies.

PART A

1. **How to evaluate the performance of a classification model?**

The different metrics are used, and some of them are as follows:

- **Accuracy**
- **Confusion Matrix**
- **Precision**
- **Recall**
- **F-Score**
- **AUC(Area Under the Curve)-ROC**

2. **What is Confusion Matrix?**

A confusion matrix is a tabular representation of prediction outcomes of any binary classifier, which is used to describe the performance of the classification model on a set of test data when true values are known.

n=165	Predicted: NO	Predicted: YES
Actual: NO	50	10
Actual: YES	5	100

3. **What is F- Score?**

F-score or F1 Score is a metric to evaluate a binary classification model on the basis of predictions that are made for the positive class. It is calculated with the help of Precision and Recall. It is a type of single score that represents both Precision and Recall. So, ***the F1 Score can be calculated as the harmonic mean of both precision and Recall, assigning equal weight to each of them.***

The formula for calculating the F1 score is given below:

$$F1 - score = 2 * \frac{precision * recall}{precision + recall}$$

4. Define Baseline Model.

A baseline model is a simple model used to predict the outcome of data. It serves as a starting point for analysis, allowing us to assess the performance of more complex models and the impact of additional features.

5. List the importance of Baseline Models in Machine Learning.

- Performance comparison-
- Minimum performance requirement
- Decision-making

6. What are the types of baseline models?

Random Baseline Models: Data in the actual world isn't always reliable. A dummy classifier or regressor is the optimal baseline model for these issues. This baseline model will inform you if your machine learning model is learning or not.

ML Baseline Modes: Now, if the data is predictable, you can create a baseline model which helps us analyze which features are critical for prediction and which are not. The baseline models are commonly used with feature engineering.

Automated ML Baseline Models: It is the ultimate baseline model. It's an excellent model for comparing your ML model. If your ML model outperforms the automated baseline model, it's a strong indication that the model has the potential to become a product.

7. Define Hyperparameters? List some examples of hyperparameter in machine learning.

- "***Hyperparameters are defined as the parameters that are explicitly defined by the user to control the learning process.***"
- Some examples of Hyperparameters in Machine Learning
  - The k in kNN or K-Nearest Neighbour algorithm
  - Learning rate for training a neural network
  - Train-test split ratio
  - Batch Size
  - Number of Epochs
  - Branches in Decision Tree
  - Number of clusters in Clustering Algorithm

8. What are the two basic approaches to choosing these hyperparameters?

There are two basic approaches to choosing these hyperparameters:

- Manual Hyperparameter
- Automatic Hyperparameter

Manual Hyperparameter

To set hyperparameters manually, must understand the relationship between hyperparameters, training error, generalization error and computational resources (memory and runtime).

Automatic Hyperparameter

Neural networks can sometimes perform well with only a small number of tuned hyperparameters, but often benefit significantly from tuning of forty or more hyperparameters.

9. Difference between Hyperparameters and Parameters.

- **Parameters** are the variables that are used by the Machine Learning algorithm for predicting the results based on the input historic data. Example of best

Parameters: **Coefficient** of independent variables Linear Regression and Logistic Regression.

- Hyperparameters are the variables that the user specifies, usually when building the Machine Learning model.

10. Explain in about Random search.

- Random search is a technique where random combinations of the hyperparameters are used to find the best solution for the built model.
- It tries random combinations of a range of values.
- To optimize with random search, the function is evaluated at some number of random configurations in the parameter space.

11. What is Grid search?

- **Grid Search** is an optimization algorithm that allows us to select the best parameters to optimize the issue from a list of parameter choices we are providing, thus automating the 'trial-and-error' method.
- Although applying it to multiple optimization issues; however, it is most commonly known for its utilization in machine learning in order to obtain the parameters at which the model provides the best accuracy.

PART B

1. Explain in detail about Measuring classifier performance and explain the performance metrics for Classification. (or) Discuss the various performance metrics to evaluate a deep learning model with example.

Measuring classifier performance

- To evaluate the performance or quality of the model, different metrics are used, and these metrics are known as performance metrics or evaluation metrics.
- These performance metrics help us understand how well our model has performed for the given data. In this way, we can improve the model's performance by tuning the hyper-parameters.
- Each ML model aims to generalize well on unseen/new data, and performance metrics help determine how well the model generalizes on the new dataset.
- In machine learning, each task or problem is divided into **classification** and **Regression**. Not all metrics can be used for all types of problems; hence, it is important to know and understand which metrics should be used. Different evaluation metrics are used for both Regression and Classification tasks. In this topic, we will discuss metrics used for classification and regression tasks.

Performance Metrics for Classification

In a classification problem, the category or classes of data is identified based on training data. The model learns from the given dataset and then classifies the new data into classes or groups based on the training. It predicts class labels as the output, such as *Yes or No, 0 or 1, Spam or Not Spam,* etc. To evaluate the performance of a classification model, different metrics are used, and some of them are as follows:

- **Accuracy**
- **Confusion Matrix**
- **Precision**
- **Recall**
- **F-Score**
- **AUC(Area Under the Curve)-ROC**

I. Accuracy

The accuracy metric is one of the simplest Classification metrics to implement, and it can be determined as the number of correct predictions to the total number of predictions.

- It can be formulated as:

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ number\ of\ predictions}$$

To implement an accuracy metric, we can compare ground truth and predicted values in a loop, or we can also use the scikit-learn module for this.

Firstly, we need to import the *accuracy_score* function of the scikit-learn library as follows:

```
from sklearn.metrics import accuracy_score
```

Here, metrics is a class of sklearn.

Then we need to pass the ground truth and predicted values in the function to calculate the accuracy.

```
print(f'Accuracy Score is {accuracy_score(y_test,y_hat)}')
```


II. Confusion Matrix

A confusion matrix is a tabular representation of prediction outcomes of any binary classifier, which is used to describe the performance of the classification model on a set of test data when true values are known.

The confusion matrix is simple to implement, but the terminologies used in this matrix might be confusing for beginners.

A typical confusion matrix for a binary classifier looks like the below image(However, it can be extended to use for classifiers with more than two classes).

n=165	Predicted: NO	Predicted: YES
Actual: NO	50	10
Actual: YES	5	100

We can determine the following from the above matrix:

- In the matrix, columns are for the prediction values, and rows specify the Actual values. Here Actual and prediction give two possible classes, Yes or No. So, if we are predicting the presence of a disease in a patient, the Prediction column with Yes means, Patient has the disease, and for NO, the Patient doesn't have the disease.
- In this example, the total number of predictions are 165, out of which 110 time predicted yes, whereas 55 times predicted No.
- However, in reality, 60 cases in which patients don't have the disease, whereas 105 cases in which patients have the disease.

In general, the table is divided into four terminologies, which are as follows:

1. **True Positive(TP):** In this case, the prediction outcome is true, and it is true in reality, also.
2. **True Negative(TN):** in this case, the prediction outcome is false, and it is false in reality, also.
3. **False Positive(FP):** In this case, prediction outcomes are true, but they are false in actuality.
4. **False Negative(FN):** In this case, predictions are false, and they are true in actuality.

III. Precision

The precision metric is used to overcome the limitation of Accuracy. The precision determines the proportion of positive prediction that was actually correct. It can be calculated as the True Positive or predictions that are actually true to the total positive predictions (True Positive and False Positive).

$$Precision = \frac{TP}{(TP + FP)}$$

IV. Recall or Sensitivity

It is also similar to the Precision metric; however, it aims to calculate the proportion of actual positive that was identified incorrectly. It can be calculated as True Positive or predictions that are actually true to the total number of positives, either correctly predicted as positive or incorrectly predicted as negative (true Positive and false negative).

The formula for calculating Recall is given below:

$$\text{Recall} = \frac{TP}{TP+FN}$$

If we maximize precision, it will minimize the FP errors, and if we maximize recall, it will minimize the FN error.

V. F-Scores

F-score or F1 Score is a metric to evaluate a binary classification model on the basis of predictions that are made for the positive class. It is calculated with the help of Precision and Recall. It is a type of single score that represents both Precision and Recall. So, ***the F1 Score can be calculated as the harmonic mean of both precision and Recall, assigning equal weight to each of them.***

The formula for calculating the F1 score is given below:

$$F1 - score = 2 * \frac{precision * recall}{precision + recall}$$

VI. AUC-ROC

Sometimes we need to visualize the performance of the classification model on charts; then, we can use the AUC-ROC curve. It is one of the popular and important metrics for evaluating the performance of the classification model.

Firstly, let's understand ROC (Receiver Operating Characteristic curve) curve. ***ROC represents a graph to show the performance of a classification model at different threshold levels***. The curve is plotted between two parameters, which are:

- **True Positive Rate**
- **False Positive Rate**

TPR or true Positive rate is a synonym for Recall, hence can be calculated as:

$$TPR = \frac{TP}{TP + FN}$$

FPR or False Positive Rate can be calculated as:

$$TPR = \frac{FP}{FP + TN}$$

AUC: Area Under the ROC curve

AUC is known for **Area Under the ROC curve**. As its name suggests, AUC calculates the two-dimensional area under the entire ROC curve, as shown below image:

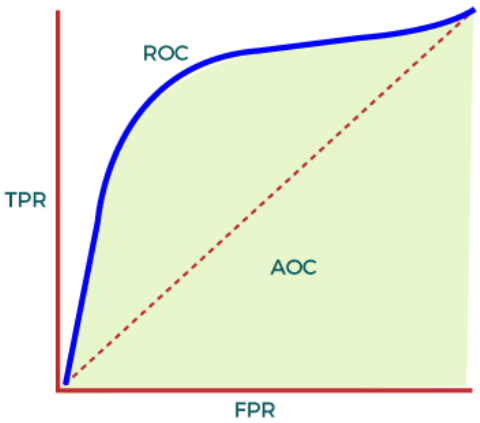


Figure 4. 1 AUC calculates the performance across all the thresholds and provides an aggregate measure. The value of AUC ranges from 0 to 1. It means a model with 100% wrong prediction will have an AUC of 0.0, whereas models with 100% correct predictions will have an AUC of 1.0.

2. Explain in detail about Baseline Models and importance of Baseline Models in Machine Learning.

Introduction to Baseline Models

- After choosing performance metrics and goals, the next step in any practical application is to establish a reasonable end-to-end system as soon as possible.
- A baseline model is a simple model used to predict the outcome of data. It serves as a starting point for analysis, allowing us to assess the performance of more complex models and the impact of additional features.

Figure 4.2 Baseline Models overview

- There are two common approaches to creating a baseline model for classification.
  - **Majority class classifier:** where the most frequent class in the data is predicted for all observations. For instance, if we have 80% of observations in class A and

20% in class B for a binary classification problem, the baseline model would predict Class A for all instances.

- **Random classifier:** Randomly assigning class labels based on the class distribution in the data. In the aforementioned binary classification scenario, we would assign class A to 80% of the observations and class B to 20% of the observations randomly. The random classifier is particularly useful when there is no specific guidance or knowledge available to make informed predictions.

Importance of Baseline Models in Machine Learning

Baseline models serve as a reference point in machine learning tasks and offer several benefits. Here are the key reasons for using baseline models:

1. **Performance comparison**: Baseline models provide a basis for comparing the performance of more advanced models. They help determine if the complexity of a model translates into improved performance. If a model fails to outperform the baseline, it suggests issues with the approach or data.
2. **Minimum performance requirement**: Baseline models set a minimum performance requirement for any useful model. If a complex model cannot surpass the baseline's performance, it may not be worthwhile to implement it practically.
3. **Decision-making**: Baseline models aid in resource allocation, model selection, and further model improvement. If the baseline model already achieves satisfactory performance, additional time and resources may not be necessary for building more complex models.

Types of baseline models

Baseline models are divided into three main categories:

- **Random Baseline Models:** Data in the actual world isn't always reliable. A dummy classifier or regressor is the optimal baseline model for these issues. This baseline model will inform you if your machine learning model is learning or not.
- **ML Baseline Modes:** Now, if the data is predictable, you can create a baseline model which helps us analyze which features are critical for prediction and which are not. The baseline models are commonly used with feature engineering.
- **Automated ML Baseline Models:** It is the ultimate baseline model. It's an excellent model for comparing your ML model. If your ML model outperforms the automated baseline model, it's a strong indication that the model has the potential to become a product.

Building a baseline model

There are a few different ways to create a baseline for your models:

Rule-based models

As the name implies, rule-based models produce predictions based on basic rules. The sort of rule that a model can use is determined by its purpose. You can also build a model that makes random or constant predictions of your choice, although this method is less common because it does not take advantage of domain expertise. This strategy is popular due to its prediction delivery, but it has a few drawbacks, such as the fact that it ignores input data, which might have an influence on your problem statement.

Baseline regression models

Let's look at a few baselines that may be utilized to solve regression difficulties.

- **Mean or median:** You can use mean or median as baseline for your outcome.
- **Business or Conditional Logic:** In this you consider 1-2 factors for your problem. For example, if you are working on a model that analyzes the height and weight of a child. In this you can take a base value like an average 5-7 year old child's height is around 39 to 48 inches. This can help you achieve your problem statement or business goals.
- **Linear regression:** If you're using a sophisticated model with a large dataset as your primary model, a simple linear regression model with a few parameters might be a good baseline model.

Baseline classification models

Let's look at a few baselines that may be utilized to solve classification difficulties.

- **Mode**: The simplest baseline model for binary classification problems is just predicting the mode of the outcome variable for all data.
- **Business or Conditional Logic:** You take into account a few elements in order to solve your problem. For example, if you want to know how much milk a cat drinks in a day. So, you can classify into two groups with more than two liters but fewer than two liters . So, to work on this model, you may divide them by cat size. Large cats, for example, have been known to consume more than two liters.
- **Logistic regression:** If your classification model contains a lot of features, a basic model like a logistic regression model might be used as a good starting point.

Baseline Models for Imbalanced Classes

- A baseline model, like a dummy classifier, is useful for detecting imbalanced classes by providing a comparison point. It allows us to assess the performance of more advanced models in the context of imbalanced data.
- Imbalanced classes often lead to the majority class dominating predictions, resulting in high accuracy but poor identification of the minority class.
- A baseline model helps establish the expected performance level using a random or simplistic approach. Say a new kind of class emerges in your data pipeline, this new class would go unnoticed as the ratio to the existing class differs by a huge margin.
- Creating a baseline model helps determine its accuracy in predicting the majority class and serves as a starting point for evaluating complex models. If an advanced model fails to outperform the baseline, it suggests ineffective handling of the imbalanced class issue.
- The baseline classifier, such as a dummy classifier with the 'most_frequent' strategy, is suitable for detecting imbalanced classes in binary classification. It predicts the most frequent class for all instances, essentially ignoring the minority class and introducing bias towards the majority class.
- When evaluating the baseline classifier's performance on imbalanced classes, relying solely on accuracy may not provide an accurate representation.

Strategies in Dummy Classifier: Exploring Different Approaches

The scikit-learn library's DummyClassifier class offers various strategies for generating predictions. These strategies are designed to create simple baseline models for comparison with more advanced models. Here are some commonly used strategies:

1. "**stratified**": This strategy randomly selects class labels based on the class distribution in the training set. It aims to maintain the same class distribution as the training data, making it useful for imbalanced classes.
2. "**most_frequent**": This strategy always predicts the most frequent class in the training set. It is suitable for imbalanced datasets where the majority class dominates the distribution. It provides a baseline performance level based on the most common class, without considering input features.
3. "**uniform**": This strategy assigns class labels randomly and uniformly, without considering the class distribution in the training data. It is useful when there is no specific pattern or information to guide the predictions.
4. "**constant**": This strategy always predicts a constant class label specified by a constant parameter. It helps create a baseline model that consistently predicts a

particular class. It aids in evaluating the impact of class imbalance and comparing model performance against a fixed prediction.

By selecting an appropriate strategy, you can create a baseline classifier that reflects different aspects of the data, such as class distribution, majority class dominance, randomness, or a fixed prediction.

#A sample baseline classifier for Breast Cancer dataset

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the breast cancer dataset
data = load_breast_cancer()
X = data.data
y = data.target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```



```
# Create a baseline random classifier
dummy_clf = DummyClassifier(strategy='stratified', random_state=42)

# Fit the baseline classifier on the training data
dummy_clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = dummy_clf.predict(X_test)

# Calculate accuracy and other metrics
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

# Print the results
print("Baseline Classifier Accuracy:", accuracy)
print("Classification Report:")
print(report)
```

3. Why are Hyperparameters? Discuss the steps to perform hyperparameter tuning.

Hyperparameters

- "***Hyperparameters are defined as the parameters that are explicitly defined by the user to control the learning process.***"
- Some examples of Hyperparameters in Machine Learning
  - The k in kNN or K-Nearest Neighbour algorithm
  - Learning rate for training a neural network
  - Train-test split ratio
  - Batch Size
  - Number of Epochs
  - Branches in Decision Tree
  - Number of clusters in Clustering Algorithm
- Most deep learning algorithms come with many hyperparameters that control many aspects of the algorithm's behavior.
- Some of these hyperparameters affect the time and memory cost of running the algorithm.
- Some of these hyperparameters affect the quality of the model recovered by the training process and its ability to infer correct results when deployed on new inputs.
- There are two basic approaches to choosing these hyperparameters:
  - Manual Hyperparameter
  - Automatic Hyperparameter

Manual Hyperparameter

- To set hyperparameters manually, must understand the relationship between hyperparameters, training error, generalization error and computational resources (memory and runtime).

Goals of manual hyperparameter

- To find the lowest generalization error subject to some runtime and memory budget.
- The primary goal of manual hyperparameter search is to adjust the effective capacity of the model to match the complexity of the task.

Effective capacity is constrained by three factors:

- The representational capacity of the model,
- The ability of the learning algorithm to successfully minimize the cost function used to train the model and
- The degree to which the cost function and training procedure regularize the model.

- A model with more layers and more hidden units per layer has higher representational capacity, it is capable of representing more complicated functions.
- It can not necessarily actually learn all of these functions though, if the training algorithm cannot discover that certain functions do a good job of minimizing the training cost, or if regularization terms such as weight decay forbid some of these functions.
- The generalization error typically follows a U-shaped curve when plotted as a function of one of the hyperparameters.
- At one extreme, the hyperparameter value corresponds to low capacity, and generalization error is high because training error is high. This is the ***underfitting*** regime.
- At the other extreme, the hyperparameter value corresponds to high capacity, and the generalization error is high because the gap between training and test error is high.
- Somewhere in the middle lies the optimal model capacity, which achieves the lowest possible generalization error, by adding a medium generalization gap to a medium amount of training error.
- For some hyperparameters, overfitting occurs when the value of the hyper parameter is large.

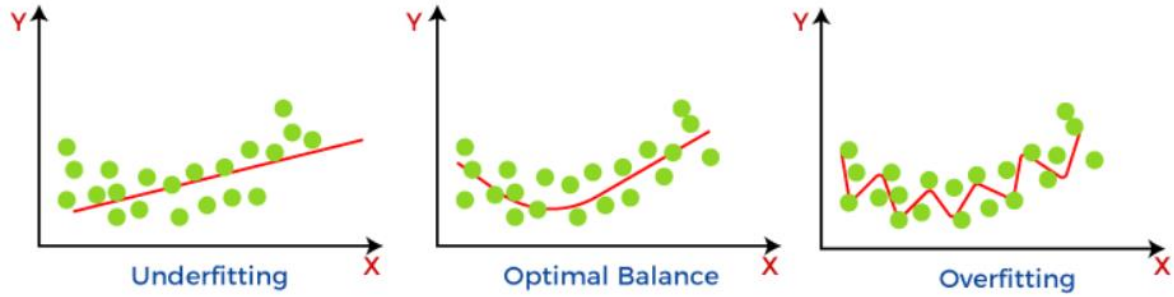


Figure 4.3 Underfitting, optimal balance and Overfitting

- The number of hidden units increases the capacity of the model.
- The learning rate is perhaps the most important hyperparameter.
- If you have time to tune only one hyperparameter, tune the learning rate.
- It controls the effective capacity of the model in a more complicated way than other hyperparameters—the effective capacity of the model is highest when the learning rate is correct for the optimization problem, not when the learning rate is especially large or especially small. The learning rate has a U-shaped curve for training error as shown in figure 4.3.

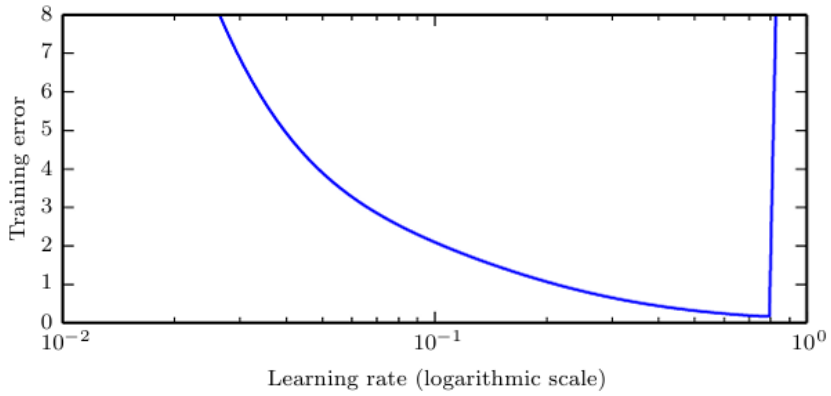


Figure 4.4 Relationship between the learning rate and the training error

- When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error.
- When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error. This effect is poorly understood.
- Tuning the parameters other than the learning rate requires monitoring both training and test error to diagnose whether your model is overfitting or underfitting, then adjusting its capacity appropriately.

Hyperparameter	Increases capacity when...	Reason	Caveats
Number of hidden units	increased	Increasing the number of hidden units increases the representational capacity of the model.	Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model.
Learning rate	tuned optimally	An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure	
Convolution kernel width	increased	Increasing the kernel width increases the number of parameters in the model	A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost.
Implicit zero padding	increased	Adding implicit zeros before convolution keeps the representation size large	Increased time and memory cost of most operations.
Weight decay coefficient	decreased	Decreasing the weight decay coefficient frees the model parameters to become larger	
Dropout rate	decreased	Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set	

Table 11.1: The effect of various hyperparameters on model capacity.

Automatic Hyperparameter

- Neural networks can sometimes perform well with only a small number of tuned hyperparameters, but often benefit significantly from tuning of forty or more hyperparameters.
- Manual hyperparameter tuning can work very well when the user has a good starting point, such as one determined by others having worked on the same type of application and architecture, or when the user has months or years of experience in exploring hyperparameter values for neural networks applied to similar tasks.
- However, for many applications, these starting points are not available.

- In these cases, automated algorithms can find useful values of the hyperparameters.
- Develop hyperparameter optimization algorithms that wrap a learning algorithm and choose its hyperparameters, thus hiding the hyperparameters of the learning algorithm from the user.
- Unfortunately, hyperparameter optimization algorithms often have their own hyperparameters, such as the range of values that should be explored for each of the learning algorithm's hyperparameters.
- However, these secondary hyperparameters are usually easier to choose, in the sense that acceptable performance may be achieved on a wide range of tasks using the same secondary hyperparameters for all tasks.

4. Explain in detail about Grid search (or) Discuss the steps involved in Grid search.

Grid search

- **Grid Search** is an optimization algorithm that allows us to select the best parameters to optimize the issue from a list of parameter choices we are providing, thus automating the 'trial-and-error' method.
- Although applying it to multiple optimization issues; however, it is most commonly known for its utilization in machine learning in order to obtain the parameters at which the model provides the best accuracy.

Hyperparameters vs Parameters

- **Parameters** are the variables that are used by the Machine Learning algorithm for predicting the results based on the input historic data. Example of best Parameters: **Coefficient** of independent variables Linear Regression and Logistic Regression.
- Hyperparameters are the variables that the user specifies, usually when building the Machine Learning model.
- They are distinct from parameters and play a crucial role in determining the model's performance.
- Hyperparameters are chosen prior to defining the parameters, and they are instrumental in finding the optimal parameter combinations.
- One common approach to finding the best hyperparameters is through methods like grid search, where a parameter grid is defined, and various combinations of Specified parameter values are evaluated against a specified evaluation metric.

Understanding Grid Search

- Grid Search employs an exhaustive search strategy, systematically exploring various combinations of specified hyperparameters and their Default values.
- This approach involves tuning parameters, such as learning rate, through a cross-validated model, which assesses performance across different parameter settings.
- However, due to its exhaustive nature, Grid Search can become time-consuming and resource-intensive, particularly as the number of hyperparameters increases as shown in figure 4.5.

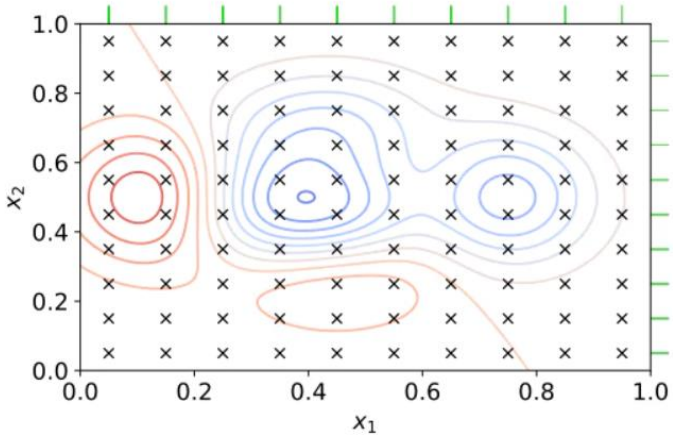


Figure 4.5 grid search

Cross-Validation and GridSearchCV

- In GridSearchCV, along with Grid Search, cross-validation is also performed. Cross-Validation is used while training the model. As we know that before training the model with data, we divide the data into two parts – **train data** and **test data**. In cross-validation, the process divides the train data further into two parts – the **train data** and the **validation data.**

- The most popular type of Cross-validation is K-fold Cross-Validation. It is an iterative process that divides the train data into k partitions. Each iteration keeps one partition for testing and the remaining **k-1** partitions for training the model. The next iteration will set the next partition as test data and the remaining k-1 as train data and so on. In each iteration, it will record the performance of the model and at the end give the average of all the performance. Thus, it is also a time-consuming process.

- Thus, GridSearch along with cross-validation takes huge time cumulatively to evaluate the best hyperparameters. Now we will see how to use GridSearchCV in our Machine Learning problem.

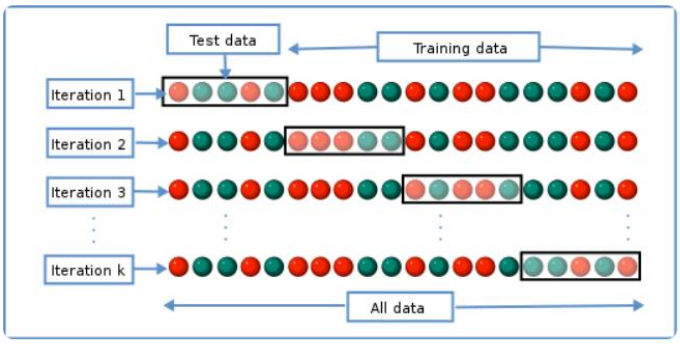


Figure 4. 6 Visualization of Grid Search

How to Apply GridSearchCV?

GridSearchCV() method is available in the scikit-learn class **model_selection.** It can be initiated by creating **an object** of GridSearchCV():

clf = GridSearchCv(estimator, param_grid, cv, scoring)

Primarily, it takes 4 arguments i.e. **estimator**, **param_grid**, **cv**, and **scoring**. The description of the arguments is as follows:

1. **estimator** – A scikit-learn model
2. **param_grid** – A dictionary with parameter names as keys and lists of parameter values.
3. **scoring** – The performance measure. For example, '**r2**' for regression models, '**precision**' for classification models.
4. **cv** – An integer that is the number of folds for K-fold cross-validation.

GridSearchCV can be used on several hyperparameters to get the best values for the specified hyperparameters.

Now let's apply GridSearchCV with a sample dataset:

Importing the Libraries & the Dataset

Python Code:

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
import warnings
warnings.filterwarnings('ignore')
df = pd.read_csv('heart.csv')
print(df.head())
```

Here we are going to use the HeartDiseaseUCI dataset.

Specifying Independent and Dependent Variables

```
X = df.drop('target', axis = 1)
y = df['target']
```

Splitting the data into train and test set

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

Building Random Forest Classifier

```
rfc = RandomForestClassifier()
```

Here, we created the object rfc of RandomForestClassifier().

Initializing GridSearchCV() object and fitting it with hyperparameters

```
forest_params = [{'max_depth': list(range(10, 15)), 'max_features': list(range(0,14))}]
clf = GridSearchCV(rfc, forest_params, cv = 10, scoring='accuracy')
clf.fit(X_train, y_train)
```

Here, we passed the estimator object **rfc**, param_grid as **forest_params**, cv = **5** and scoring method as **accuracy** in to GridSearchCV() as arguments.

Getting the Best Hyperparameters

```
print(clf.best_params_)
```

This will give the combination of hyperparameters along with values that give the best performance of our estimate specified.

Putting it all together

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
import warnings
warnings.filterwarnings('ignore')
df = pd.read_csv('heart.csv')
X = df.drop('target', axis = 1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
rfc = RandomForestClassifier()
forest_params = [{'max_depth': list(range(10, 15)), 'max_features': list(range(0,14))}]
clf = GridSearchCV(rfc, forest_params, cv = 10, scoring='accuracy')
clf.fit(X_train, y_train)
```

```
print(clf.best_params_)
print(clf.best_score_)
```

On executing the above code, we get:

```
{'max_depth': 13, 'max_features': 3}
0.8584415584415584
```

Best Params and Best Score of the Random Forest Classifier

Thus, **clf.best_params_** gives the best combination of tuned hyperparameters, and **clf.best_score_** gives the average cross-validated score of our Random Forest Classifier.

5. Explain in detail about Random search.

Random search

- Random search is a technique where random combinations of the hyperparameters are used to find the best solution for the built model.
- It tries random combinations of a range of values.
- To optimise with random search, the function is evaluated at some number of random configurations in the parameter space.

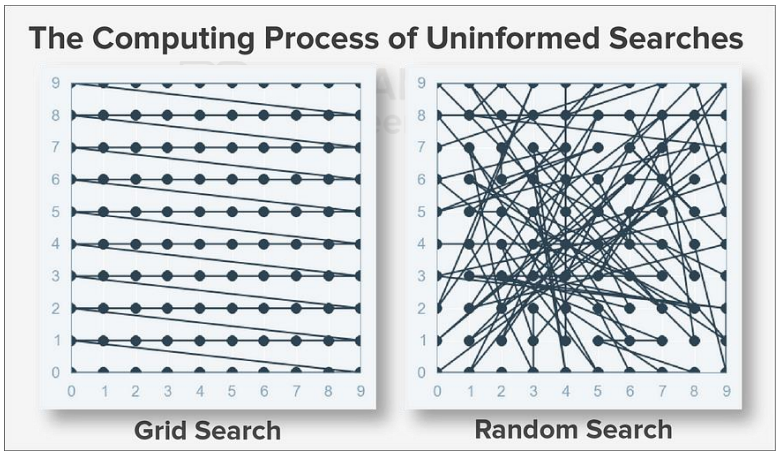


Figure 4.7 Comparison of grid search and random search.

- First define a marginal distribution for each hyperparameter, e.g., a Bernoulli or multinoulli for binary or discrete hyperparameters, or a uniform distribution on a log-scale for positive real-valued hyperparameters. For example,

$$\texttt{log\_learning\_rate} \sim u(-1, -5) \tag{11.2}$$

$$\texttt{learning\_rate} = 10^{\texttt{log\_learning\_rate}}. \tag{11.3}$$

where u(a,b) indicates a sample of the uniform distribution in the interval (a,b). Similarly the log number of hidden units _ _ _ maybesampledfromu(log(50), log(2000))

- The chances of finding the optimal parameter are comparatively higher in random search because of the random search pattern where the model might end up being trained on the optimised parameters without any aliasing.
- Random search works best for lower dimensional data since the time taken to find the right set is less with less number of iterations.
- Random search is the best parameter search technique when there are less number of dimensions.

Defining the Hyperparameter Space

- **Max_depth:** The "max_depth" hyperparameter defines the maximum depth or level of each decision tree in the random forest. A deeper tree can capture intricate relationships and details in the training data, potentially leading to high accuracy. However, deep trees are prone to overfitting and may not generalize well to unseen test data. The default value of "max_depth" in Scikit-Learn is set to "None," allowing the trees to expand until all leaves are pure or contain minimum samples.
- **Max_features:** "Max_features" determines the maximum number of features considered for each split in the random forest model. At each node, the algorithm evaluates a subset of features and selects the best one for splitting. The default value of "max_features" in Scikit-Learn is the square root of the total number of features in the dataset. This default setting promotes diversity among the trees and helps prevent overfitting.
- **N_estimators:** The "n_estimators" hyperparameter specifies the number of decision trees in the random forest ensemble. Increasing the number of estimators generally improves model performance up to a certain point, where additional trees may provide diminishing returns. The default value of "n_estimators" in Scikit-Learn is 10, but it is often recommended to increase this value to achieve better model performance.
- **Min_samples_leaf:** "Min_samples_leaf" sets the minimum number of samples required to be present at a leaf node. If the number of samples at a node falls below this threshold, further splitting is halted, and the node becomes a leaf. A higher value of "min_samples_leaf" can prevent overfitting by ensuring that each leaf contains sufficient samples to make reliable predictions. The default value in Scikit-Learn is 1, meaning that a leaf can contain a single sample.
- **Min_samples_split:** The "min_samples_split" hyperparameter determines the minimum number of samples required to split an internal node further. If the number of samples at a node is below this threshold, no further splitting is performed, and the node becomes a leaf. Setting a higher value for "min_samples_split" can prevent the tree from creating small branches with insufficient data. The default value in Scikit-Learn is 2, meaning that a node must have at least two samples to be eligible for splitting.

6. What are the Debugging strategies used in Machine Learning?

- When a machine learning system performs poorly, it is usually difficult to tell whether the poor performance is intrinsic to the algorithm itself or whether there is a bug in the implementation of the algorithm.
- Machine learning systems are difficult to debug for a variety of reasons.
- In most cases, we do not know a priori what the intended behavior of the algorithm is.
- In fact, the entire point of using machine learning is that it will discover useful behavior that we were not able to specify ourselves.
- If train a neural network on a new classification task and it achieves 5% test error, we have no straightforward way of knowing if this is the expected behavior or sub-optimal behavior.
- A further difficulty is that most machine learning models have multiple parts that are each adaptive.
- If one part is broken, the other parts can adapt and still achieve roughly acceptable performance.
- For example, suppose that we are training a neural net with several layers parametrized by weights **W** and biases **b**.
- Suppose further that have manually implemented the gradient descent rule for each parameter separately, and we made an error in the update for the biases:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \qquad (11.4)$$

where α is the learning rate.

- This erroneous update does not use the gradient at all. It causes the biases to constantly become negative throughout learning, which is clearly not a correct implementation of any reasonable learning algorithm.
- The bug may not be apparent just from examining the output of the model though. Depending on the distribution of the input, the weights may be able to adapt to compensate for the negative biases.

  Some important debugging tests include:
- **Visualize the model in action:**
  - When training a model to detect objects in images, view some images with the detections proposed by the model displayed superimposed on the image. When training a generative model of speech, listen to some of the speech samples it produces. This may seem obvious, but it is easy to fall into the

practice of only looking at quantitative performance measurements like accuracy or log-likelihood.

- Directly observing the machine learning model performing its task will help to determine whether the quantitative performance numbers it achieves seem reasonable.
- Evaluation bugs can be some of the most devastating bugs because they can mislead you into believing your system is performing well when it is not.

- **Visualize the worst mistakes:**
  - Most models are able to output some sort of confidence measure for the task they perform.
  - For example, classifiers based on a softmax output layer assign a probability to each class. The probability assigned to the most likely class thus gives an estimate of the confidence the model has in its classification decision. Typically, maximum likelihood training results in these values being overestimates rather than accurate probabilities of correct prediction, but they are somewhat useful in the sense that examples that are actually less likely to be correctly labeled receive smaller probabilities under the model.
- **Reasoning about software using train and test error:**
  - It is often difficult to determine whether the underlying software is correctly implemented.
  - Some clues can be obtained from the train and test error.
  - If training error is low but test error is high, then it is likely that that the training procedure works correctly, and the model is overfitting for fundamental algorithmic reasons.
  - An alternative possibility is that the test error is measured incorrectly due to a problem with saving the model after training then reloading it for test set evaluation, or if the test data was prepared differently from the training data.
- **Fit a tiny dataset:**
  - If you have high error on the training set, determine whether it is due to genuine underfitting or due to a software defect. Usually even small models can be guaranteed to be able fit a sufficiently small dataset.
- **Compare back-propagated derivatives to numerical derivatives:**
  - If you are using a software framework that requires you to implement your own gradient computations, or if you are adding a new operation to a differentiation library and must define its bprop method, then a common source of error is implementing this gradient expression incorrectly.

- One way to verify that these derivatives are correct is to compare the derivatives computed by your implementation of automatic differentiation to the derivatives computed by **finite differences.** Because

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}, \tag{11.5}$$

approximate the derivative by using a small, finite ε:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{11.6}$$

improve the accuracy of the approximation by using the centered difference:

$$f'(x) \approx \frac{f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)}{\epsilon}. \tag{11.7}$$

- The perturbation size ε must chosen to be large enough to ensure that the perturbation is not rounded down too much by finite-precision numerical computations.
- If one has access to numerical computation on complex numbers, then there is a very efficient way to numerically estimate the gradient by using complex numbers as input to the function. The method is based on the observation that

$$f(x+i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2) \tag{11.8}$$

$$\text{real}(f(x+i\epsilon)) = f(x) + O(\epsilon^2), \quad \text{imag}(\frac{f(x+i\epsilon)}{\epsilon}) = f'(x) + O(\epsilon^2), \tag{11.9}$$

Where $i = \sqrt{-1}$. Unlike in the real-valued case above, there is no cancellation effect due to taking the difference between the value of f at different points. This allows the use of tiny values of ε like $\epsilon = 10^{-150}$, error insignificant for all practical purposes.

- **Monitor histograms of activations and gradient:**
  - It is often useful to visualize statistics of neural network activations and gradients, collected over a large amount of training iterations (maybe one epoch).
  - The pre-activation value of hidden units can tell us if the units saturate, or how often they do.
  - For example, for rectifiers, how often are they off? Are there units that are always off? For tanh units, the average of the absolute value of the pre-activations tells us how saturated the unit is.

- In a deep network where the propagated gradients quickly grow or quickly vanish, optimization may be hampered.
- Finally, it is useful to compare the magnitude of parameter gradients to the magnitude of the parameters themselves.

SYLLABUS:

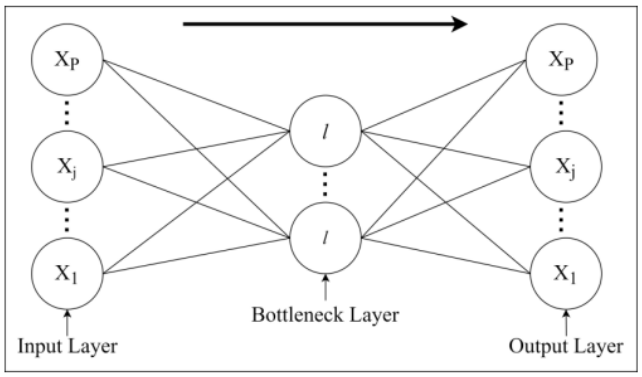
UNIT V AUTOENCODERS AND GENERATIVE MODELS 9

Autoencoders: Undercomplete autoencoders -- Regularized autoencoders -- Stochastic encoders and decoders -- Learning with autoencoders; Deep Generative Models: Variational autoencoders – Generative adversarial networks.

PART A

1. **Define Autoencoders.**

Autoencoder, also called auto-associative neural network, is a multi-layer perceptron having the same number of outputs as inputs, designed to learn an approximation to the identity function, so as the output is as similar to the input as possible.



2. **List the types of autoencoders.**
   - Undercomplete Autoencoders
   - Regularized Autoencoders
   - Sparse Autoencoder
   - Stochastic Encoders and Decoders
   - Variational Autoencoder
   - Denoising Autoencoders
   - Contractive Autoencoders

3. **What is Dimensionality Reduction?**

Dimensionality reduction is a technique of reducing the feature space to obtain a stable and statistically sound machine learning model avoiding the Curse of dimensionality. There are mainly two approaches to perform dimensionality reduction: Feature Selection and Feature Transformation.

4. **What is feature selection and feature transformation?**

Feature Selection approach tries to subset important features and remove collinear or not-so-important features. One can read more about it, here.

Feature Transformation also is known as Feature Extraction tries to project the high-dimensional data into lower dimensions. Some Feature Transformation techniques are PCA, Matrix-Factorisation, Autoencoders, t-Sne, UMAP, etc.

5. Define Principal Component Analysis (PCA).

PCA can be defined as the orthogonal projection of data onto a lower dimensional linear space, such that the variance of the projected data is maximized. To derive the form of PC's, suppose x is a vector of p variables with a covariance matrix S. The first step is to search for a linear function $\alpha 1\ ' x$ of the elements of x having maximum variance:

$$\alpha_1' x = \alpha_{11}x_1 + \alpha_{12}x_2 + \ldots + \alpha_{1p}x_p = \sum_{j=1}^{p} \alpha_{1j}x_j \qquad (1)$$

6. What is Undercomplete Autoencoders?

- An autoencoder whose code dimension is less than the input dimension is called undercomplete.
- Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.
- The learning process is described simply as minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) \qquad (14.1)$$

7. Define Regularization.

- Regularization helps with the effects of out-of-control parameters by using different methods to minimize parameter size over time.
- Regularization coefficients L1 and L2 help fight overfitting by making certain weights smaller. Smaller-valued weights lead to simpler hypotheses, which are the most generalizable. Unregularized weights with several higher-order polynomials in the feature sets tend to overfit the training set.

8. List and explain the Types of Regularization.

There are several types of regularization that can be used with autoencoders, including:

- **L1 Regularization:** This method adds a penalty to the loss function for the sum of the absolute values of the model weights. This encourages the model to learn sparse representations, where many of the weights are set to zero.
- **L2 Regularization:** This method adds a penalty to the loss function for the sum of the squares of the model weights. This encourages the model to learn small, non-zero weights.
- **Dropout:** This method randomly sets a fraction of the model's activations to zero during each training iteration. This helps prevent the model from relying too heavily on any one set of activations.

9. Define Regularized autoencoders.

Regularized Autoencoders introduce penalties, like dropout or weight constraints, to prevent overfitting and encourage robust feature extraction. The regularized loss function, incorporating regularization terms, prevents the network from memorizing data.

10. What are the Two generative modeling approaches?

The Two generative modeling approaches that emphasize this connection with autoencoders are the descendants of the Helmholtz machine, such as the ***variational autoencoder*** and ***the generative stochastic networks***.

11. What is Generative Density Estimation?

One way to make an autoencoder generative is to use a technique called ***ex-post* density estimation**.

This involves fitting a **Mixture of Gaussian distribution** to the embeddings of the training data after the model has been trained. The embedding is the low-dimensional representation of the data that the autoencoder learns.

12. List the Applications of RAE.

RAE can be useful in a variety of applications, including:

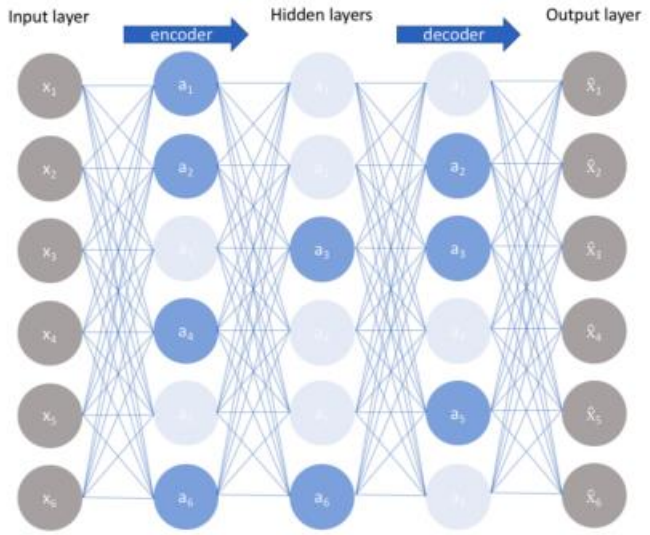
Dimensionality Reduction: RAE can be used to learn a compressed representation of high-dimensional data, making it easier to work with and visualize.

Feature Extraction: RAE can be used to extract meaningful features from the input data, which can then be used as inputs to other machine learning models.

Generative Modeling: RAE can be used to generate new data points that are similar to the training data, which can be useful in image and text generation tasks.

13. Define Sparse Autoencoders.

Another way of regularizing the autoencoder is by using a sparsity constraint. In this way of regularization, only fraction nodes are allowed to do forward and backward propagation. These nodes have non-zero values and are called active nodes.



14. What are Stochastic Encoders and Decoders?

Stochastic Autoencoders employ randomness in the encoding and decoding processes. Sampling from a distribution introduces diversity, aiding generative tasks. Variational Autoencoders (VAEs) exemplify stochasticity, with a probabilistic interpretation of latent space.

15. Give the General strategy for designing the output units and loss function of a feedforward network.

- Define the output distribution p(y|x)
- Minimize the negative log-likelihood –log p(y|x)
- In this setting y is a vector of targets such as class labels

16. Define Generative Models.

A ***Generative Model*** is a powerful way of learning any kind of data distribution using unsupervised learning and it has achieved tremendous success in just few years.

Two of the most commonly used and efficient approaches are

- Variational Autoencoders (VAE)

- Generative Adversarial Networks (GAN).

17. Define Variational Autoencoder.

Autoencoder is used to encode an input image to a much smaller dimensional representation which can store latent information about the input data distribution. But in a vanilla autoencoder, the encoded vector can only be mapped to the corresponding input using a decoder. It certainly can't be used to generate similar images with some variability.

PART B

1. **Discuss in detail about Autoencoders.(or) Justify your answer, that how autoencoders are suitable compared to Principal Compact analysis (PCA) for dimension reduction. (Nov 2023)**

Dimensionality Reduction

- Dimensionality reduction is a technique of reducing the feature space to obtain a stable and statistically sound machine learning model avoiding the Curse of dimensionality. There are mainly two approaches to perform dimensionality reduction: Feature Selection and Feature Transformation.
    - **Feature Selection** approach tries to subset important features and remove collinear or not-so-important features. One can read more about it, here.
    - **Feature Transformation** also is known as Feature Extraction tries to project the high-dimensional data into lower dimensions. Some Feature Transformation techniques are PCA, Matrix-Factorisation, Autoencoders, t-Sne, UMAP, etc.
- Traditionally, dimensionality reduction is performed using linear techniques such as Principal Components Analysis (PCA), which is one of the most used statistical techniques in behavioral sciences and is a standard part of measure development.

Principal Component Analysis (PCA)

PCA can be defined as the orthogonal projection of data onto a lower dimensional linear space, such that the variance of the projected data is maximized. To derive the form of PC's, suppose x is a vector of p variables with a covariance matrix S. The first step is to search for a linear function $\alpha 1\ '\ x$ of the elements of x having maximum variance:

$$\alpha_1' x = \alpha_{11}x_1 + \alpha_{12}x_2 + \ldots + \alpha_{1p}x_p = \sum_{j=1}^{p} \alpha_{1j}x_j \qquad (1)$$

where $\alpha 1\ '$ is a vector of p constants $\alpha 11x1, \alpha 12x2, \ldots, \alpha 1pxp$ and ' denotes transpose. The variance of the projected data is given by:

$$var[\alpha_1' x] = \alpha_1' S \alpha_1 \qquad (2)$$

and is maximized under the normalization constraints $\alpha 1\ '\alpha 1 = 1$ using the techinque of Lagrange multipliers. It follows that:

$$\alpha_1' S \alpha_1 = \lambda(\alpha_1' a_1 - 1) \qquad (3)$$

where λ is the Lagrange multipliers.

- By setting differentiation with respect $\alpha 1$ equal to zero, the solution of this problem can be obtained as a unit eigenvector of the covariance matrix S corresponding to the largest eigenvalue. Thus, $\alpha 1$ is the eigenvector corresponding to the largest eigenvalue of S, and $var(a1x) = \alpha 1 \ ' \ S\alpha 1 = \lambda 1$ the largest eigenvalue.
- In general, the kth PC of x is αkx and its variance is λ , where λk is the largest eigenvalue of S and αk is the corresponding eigenvector or, also, the vector of loadings for the kth component.
- Define additional principal components in an incremental fashion by choosing each new direction to be that which maximizes the projected variance amongst all possible directions orthogonal to those already considered.
- To summarize, principal component analysis involves evaluating the covariance matrix S of the dataset and then finding the k eigenvectors of S corresponding to the k largest eigenvalues.
- PCA can be also viewed as a linear projection of data points into a lower dimensional space such that the squared reconstruction loss is minimized. In general, a dimension reduction technique provides an approximation $\hat{x}(t)$ to $x(t)$ which is the composition of two functions f and g:

$$x(t) = \hat{x}(t) + \epsilon(t) = g\left(f(x(t))\right) + \epsilon(t) \qquad (4)$$

- The projection function $f : R\ p \rightarrow R\ z$ projects the original P-dimensional data $x(t)$ onto a Zdimensional subspace, while the expansion function $g : R\ Z \rightarrow R\ P$defines a mapping from the Zdimensional space back into the original P-dimensional space with $\epsilon(t)$ as the residue.
- The feature extraction problem may involve the determination of functions f and g. The mean square error (MSE) in reconstructing the original data is:

$$MSE = E\left[\|x - g(f(x))\|^2\right] \qquad (5)$$

- It can be shown that PCA is the algorithm which obtains the smallest MSE among all techniques with linear projection and expansion functions f and g.

Autoencoders

- Autoencoder, also called auto-associative neural network, is a multi-layer perceptron having the same number of outputs as inputs, designed to learn an approximation to the identity function, so as the output is as similar to the input as possible.

- This is achieved by minimizing an error function which captures the degree of mismatch between the input vectors and their reconstructions, typically a sum of-squares error of the form:

$$E(w) = \frac{1}{2}\sum_{j=1}^{\cdot} \|y(x_j, w) - x_j\|^2 \qquad (6)$$

- When used with a hidden layer smaller than the input/output layers and linear activations only, as represented in Figure 5.1, the autoencoder performs a compression scheme which was shown to be equivalent to PCA. In fact, both principal component analysis and the neural network are using linear dimensionality reduction and are minimizing the same sum-of-squares error function.

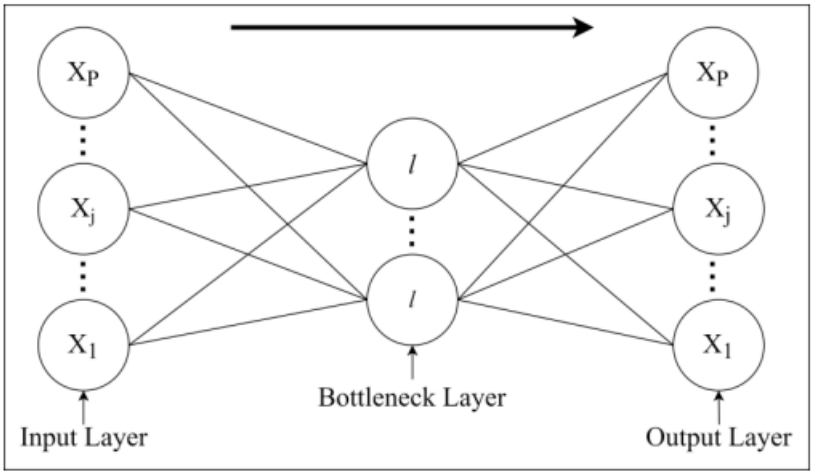


Figure 5.1 Linear Autoencoder with a single hidden layer(l=linear activation function)

- In 1991, an interesting non-linear generalization was introduced by Kramer. The network described by Kramer is again trained by minimization of the error function. We can view this network as two successive functional mappings f and g as indicated in Figure 5.2. The first mapping f projects the original P-dimensional data into a Z-dimensional subspace S defined by the activations of the units in the second hidden layer. Because of the presence of the first hidden layer of nonlinear units, this mapping is very general, and is not restricted to being linear

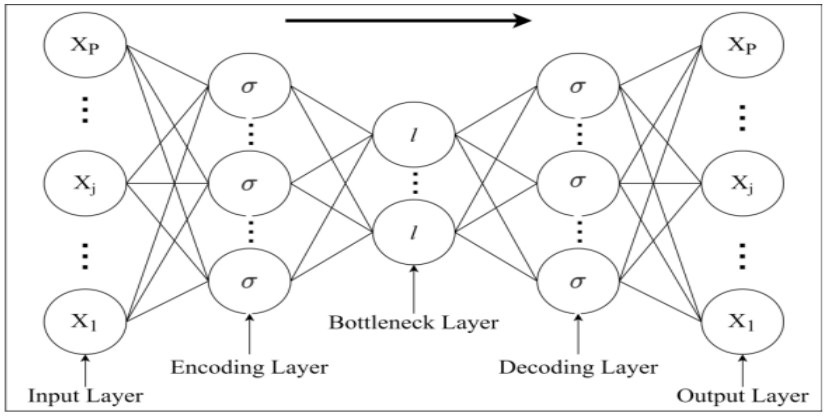


Figure 2: Non-linear autoencoder introduced by Kramer (σ = sigmoidal activation function, l = linear activation function)

Figure 5.2 Non- Linear Autoencoder introduced by Kramer (θ=sigmoidal activation function, l=linear activation function)

- Let consider a network with p neurons in the input and output layers, k neurons in the mapping and demapping layers and a single neuron in the bottleneck layer. Without biases, the projection functions f has the form:

$$f(x) = \sum_{i=1}^{K} w_{1i}^{(2)} \sigma \sum_{j=1}^{P} w_{ij}^{(1)} x_j \qquad (7)$$

- Similarly, the second half of the network defines an arbitrary functional mapping g from the Zdimensional space back into the original P-dimensional input space and takes the form:

$$g(y) = [g_1(y) \dots g_p(y)]^T = \left[\sum_{i=1}^{k} w_{1i}^{(4)} \sigma\left(w_{i1}^{(3)} y\right) \sum_{i=1}^{k} w_{ji}^{(4)} \sigma\left(w_{i1}^{(3)} y\right) \right] \qquad (8)$$

where wij (m) is the weight between the i-th neuron of layer m + 1 and the j-th neuron of layer m, and σ is a non-linear function, usually a sigmoid or a hyperbolic tangent function.

- This process has a simple geometrical interpretation, as indicated for the case P = 3 and Z = 2 in Figure 3. The function f defines a projection of points from the original P-dimensional space into the Z-dimensional subspace S; then, the function g maps from a Z-dimensional space S back into a Pdimensional space and therefore defines the way in which the space S is embedded within the original x-space. Since the mapping g can be nonlinear, the embedding of S can be nonplanar, as indicated in the figure 5.3.

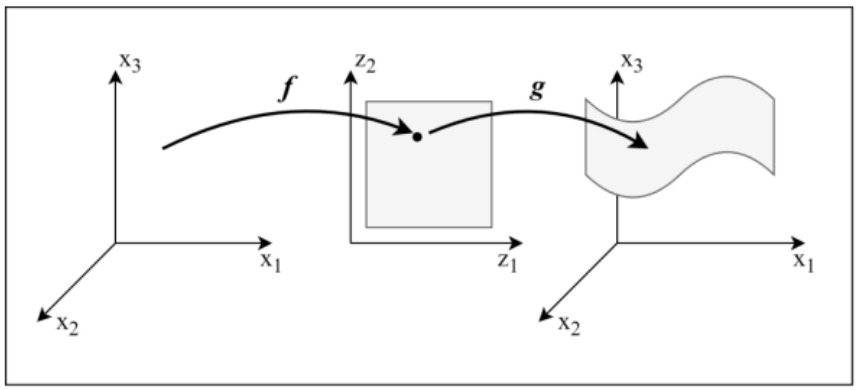


Figure 3. Geometrical interpretation of the mappings performed by the network in Figure 2 for the case of $P = 3$ inputs and $Z = 2$ units in the middle hidden layer.

Figure 5.3 Geometrical interpretation of the mappings performed by the network

- Autoencoder has the advantage of not being limited to linear transformations and can learn more complicated relations between visible and hidden units, although it contains standard principal component analysis as a special case.
- However, unlike PCA, the coordinates of the output of the bottleneck are correlated and are not sorted in descending order of variance.
- Moreover, computationally intensive nonlinear optimization techniques must be used, and there is the risk of finding a suboptimal local minimum of the error function.
- One solution to mitigate this problem was introduced by Hinton in 2006, who proposed a "layer-wise pretraining" procedure for binary data using restricted Boltzmann machines.
- V Autoencoders can be a preferable choice over Principal Component Analysis (PCA) or Singular Value Decomposition (SVD) for dimensionality reduction in several scenarios:

1. **Non-Linear Relationships**

   Use Autoencoders: If the data exhibits non-linear relationships, autoencoders can capture complex patterns through their non-linear activation functions. PCA and SVD, being linear methods, may not perform well in such cases.

2. **High-Dimensional Data**

   Use Autoencoders: For very high-dimensional data, such as images or text embeddings, autoencoders can efficiently learn lower-dimensional representations. PCA can become computationally expensive and may not scale well with very high dimensions.

3. **Feature Learning**

   Use Autoencoders: Autoencoders can learn meaningful features in an unsupervised manner, which can be beneficial for downstream tasks. PCA focuses solely on variance and may not capture features that are useful for specific applications.

4. **Data Types**

   Use Autoencoders: For data types like images, audio, or complex structured data, autoencoders are designed to handle these types effectively. PCA and SVD may not be as effective for these data types.

5. **Flexibility in Architecture**

   Use Autoencoders: Autoencoders allow for various architectures (e.g., convolutional autoencoders for images, recurrent autoencoders for sequences), enabling tailored approaches for specific data types and tasks.

6. **Regularization and Noise Reduction**

   Use Autoencoders: Autoencoders can incorporate regularization techniques (like dropout or weight decay) and denoising capabilities, enhancing their robustness against noise in the input data.

7. **Interpretability**

   Use PCA/SVD: If interpretability is a key concern, PCA provides principal components that can be easily interpreted in terms of the original features. Autoencoders, being neural networks, may produce less interpretable results.

- In summary, choose autoencoders when dealing with non-linear, high-dimensional data, when you need feature learning and flexibility in architecture, and when regularization is important. Use PCA/SVD for simpler linear relationships, when interpretability is crucial, or when computational efficiency is a concern for lower-dimensional datasets.

2. What are the types of autoencoders? Explain in detail about Undercomplete Autoencoders with neat diagram.

Types of autoencoders

- Undercomplete Autoencoders
- Regularized Autoencoders
- Sparse Autoencoder
- Stochastic Encoders and Decoders
- Variational Autoencoder
- Denoising Autoencoders
- Contractive Autoencoders

Undercomplete Autoencoders

- Copying the input to the output may sound useless, but we are typically not interested in the output of the decoder.
- Instead, we hope that training the autoencoder to perform the input copying task will result in h taking on useful properties.
- One way to obtain useful features from the autoencoder is to constrain h to have smaller dimension than x.
- An autoencoder whose code dimension is less than the input dimension is called undercomplete.

- Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.
- The learning process is described simply as minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) \qquad (14.1)$$

Where,

L is a loss function penalizing

g(f(x)) for being dissimilar from x, such as the mean squared error.

- When the decoder is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA. In this case, an autoencoder trained to perform the copying task has learned the principal subspace of the training data as a side-effect. Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can thus learn a more powerful nonlinear generalization of PCA.
- Unfortunately, if the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information about the distribution of the data.

- Theoretically, one could imagine that an autoencoder with a one-dimensional code but a very powerful nonlinear encoder could learn to represent each training example $x^{(i)}$ with the code i.
- The decoder could learn to map these integer indices back to the values of specific training examples.
- This specific scenario does not occur in practice, but it illustrates clearly that an autoencoder trained to perform the copying task can fail to learn anything useful about the dataset if the capacity of the autoencoder is allowed to become too great.

3. Explain in detail about Regularized autoencoders and sparse autoencoders.

Regularization

- Regularization helps with the effects of out-of-control parameters by using different methods to minimize parameter size over time.
- Regularization coefficients L1 and L2 help fight overfitting by making certain weights smaller. Smaller-valued weights lead to simpler hypotheses, which are the most generalizable. Unregularized weights with several higher-order polynomials in the feature sets tend to overfit the training set.
- As the input training set size grows, the effect of regularization decreases, and the parameters tend to increase in magnitude. This is appropriate because an excess of features relative to training set examples leads to overfitting in the first place. Bigger data is the ultimate regularizer.

Types of Regularization

There are several types of regularization that can be used with autoencoders, including:

- **L1 Regularization:** This method adds a penalty to the loss function for the sum of the absolute values of the model weights. This encourages the model to learn sparse representations, where many of the weights are set to zero.
- **L2 Regularization:** This method adds a penalty to the loss function for the sum of the squares of the model weights. This encourages the model to learn small, non-zero weights.
- **Dropout:** This method randomly sets a fraction of the model's activations to zero during each training iteration. This helps prevent the model from relying too heavily on any one set of activations.

Regularized autoencoders

Regularized Autoencoders introduce penalties, like dropout or weight constraints, to prevent overfitting and encourage robust feature extraction. The regularized loss function, incorporating regularization terms, prevents the network from memorizing data.

- **Undercomplete autoencoders**, with code dimension less than the input dimension, can learn the most salient features of the data distribution. We have seen that these autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity.
- A similar problem occurs if the hidden code is allowed to have dimension equal to the input, and in the overcomplete case in which the hidden code has dimension greater than the input.
- In these cases, even a **linear encoder** and **linear decoder** can learn to copy the input to the output without learning anything useful about the data distribution.
- The properties of **RAE** include **sparsity of the representation**, smallness of the derivative of the representation, and robustness to noise or to missing inputs.
- A regularized autoencoder can be **nonlinear** and **overcomplete** but still learn something useful about the data distribution even if the model capacity is great enough to learn a **trivial identity function**.
- **Two generative modeling approaches** that emphasize this connection with autoencoders are the descendants of the Helmholtz machine, such as the ***variational autoencoder*** and ***the generative stochastic networks***.
- These models naturally learn high-capacity, overcomplete encodings of the input and do not require regularization for these encodings to be useful.
- Their encodings are naturally useful because the models were trained to approximately maximize the probability of the training data rather than to copy the input to the output.

Generative Density Estimation

- One way to make an autoencoder generative is to use a technique called ***ex-post* density estimation**.
- This involves fitting a **Mixture of Gaussian distribution** to the embeddings of the training data after the model has been trained. The embedding is the low-dimensional representation of the data that the autoencoder learns.
- The **Mixture of Gaussian distribution** consists of a weighted sum of different Gaussian distributions, each with its own mean and variance.

- By fitting this distribution to the embeddings, the model can generate new data points by randomly sampling from the distribution. This allows us to generate new data that is similar to the training data, but not identical.

Applications of RAE

RAE can be useful in a variety of applications, including:

- **Dimensionality Reduction:** RAE can be used to learn a compressed representation of high-dimensional data, making it easier to work with and visualize.
- **Feature Extraction:** RAE can be used to extract meaningful features from the input data, which can then be used as inputs to other machine learning models.
- **Generative Modeling:** RAE can be used to generate new data points that are similar to the training data, which can be useful in image and text generation tasks.

Sparse Autoencoders

- Another way of regularizing the autoencoder is by using a sparsity constraint. In this way of regularization, only fraction nodes are allowed to do forward and backward propagation. These nodes have non-zero values and are called active nodes as shown in figure 5.4.

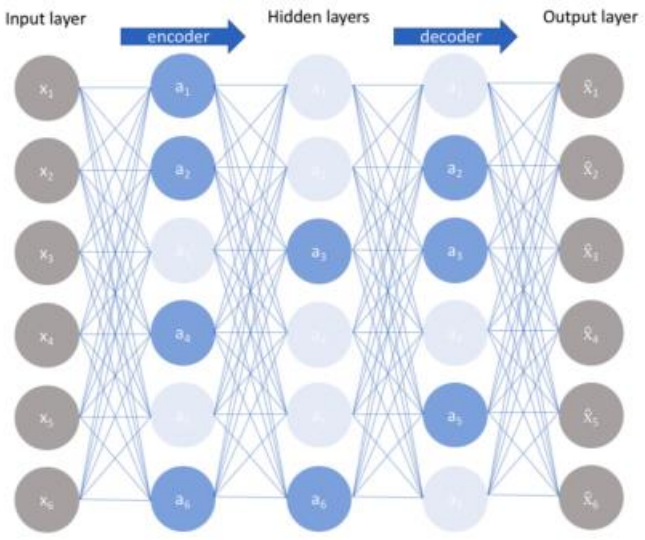


Figure 5.4 regularizing the autoencoder is by using a sparsity constraint

- To do so, add a penalty term to the loss function, which helps to activate the fraction of nodes. This forces the autoencoder to represent each input as a combination of a small number of nodes and demands it to discover interesting

structures in the data. This method is efficient even if the code size is large because only a small subset of the nodes will be active.

- **For example**, add a regularization term in the loss function. Doing this will make our autoencoder learn the sparse representation of data.

```
input_size = 256
hidden_size = 32
output_size = 256
l1 = Input(shape=(input_size,))
# Encoder
h1 = Dense(hidden_size ,activity_regularizer=regularizers.l1(10e-6),
activation='relu')(l1)
# Decoder
l2 = Dense(output_size, activation='sigmoid')(h1)
autoencoder = Model(input=l1, output=l2)
autoencoder.compile(loss='mse', optimizer='adam')
```

In the above code, we have added L1 regularization to the hidden layer of the encoder, which adds the penalty to the loss function.

4. Explain in detail about Stochastic encoders and decoders.

Stochastic Encoders and Decoders:

- Stochastic Autoencoders employ randomness in the encoding and decoding processes. Sampling from a distribution introduces diversity, aiding generative tasks. Variational Autoencoders (VAEs) exemplify stochasticity, with a probabilistic interpretation of latent space.
- General strategy for designing the output units and loss function of a feedforward network is to
  - Define the output distribution p(y|x)
  - Minimize the negative log-likelihood –log p(y|x)
  - In this setting y is a vector of targets such as class labels
- In an autoencoder x is the target as well as the input

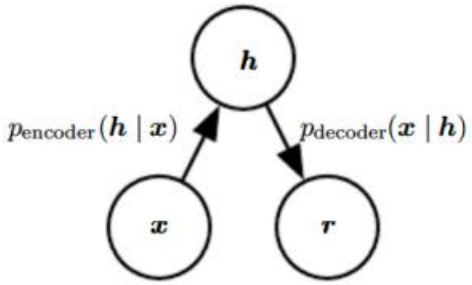
Loss function for Stochastic Decoder

- Given a hidden code h, we may think of the decoder as providing a conditional distribution pdecoder(x|h)
- We train the autoencoder by minimizing –log pdecoder(x|h)

- The exact form of this loss function will change depending on the form of pdecoder(x|h)
- As with feedforward networks we use linear output units to parameterize the mean of the Gaussian distribution if x is real
- In this case negative log-likelihood is the mean-squared error
- With binary x correspond to a Bernoulli with parameters given by a sigmoid
- Discrete x values correspond to a softmax
- The output variables are treated as being conditionally independent given h.

Stochastic encoder

- We can also generalize the notion of an encoding function f(x) to an encoding distribution pencoder(h|x)



Structure of stochastic autoencoder

- Both the encoder and decoder are not simple functions but involve a distribution.
- The output is sampled from a distribution pencoder(h|x) for the encoder and pdecoder(x|h) for the decoder.

Relationship to joint distribution

- Any latent variable model pmodel(h|x) defines a stochastic encoder pencoder(h|x)=pmodel(h|x) and a stochastic decoder pdecoder(x|h)=pmodel(x|h)
- In general the encoder and decoder distributions are not conditional distributions compatible with a unique joint distribution pmodel(x,h)
- Training the autoencoder as a denoising autoencoder will tend to make them compatible asymptotically with enough capacity and examples as shown figure 5.5.

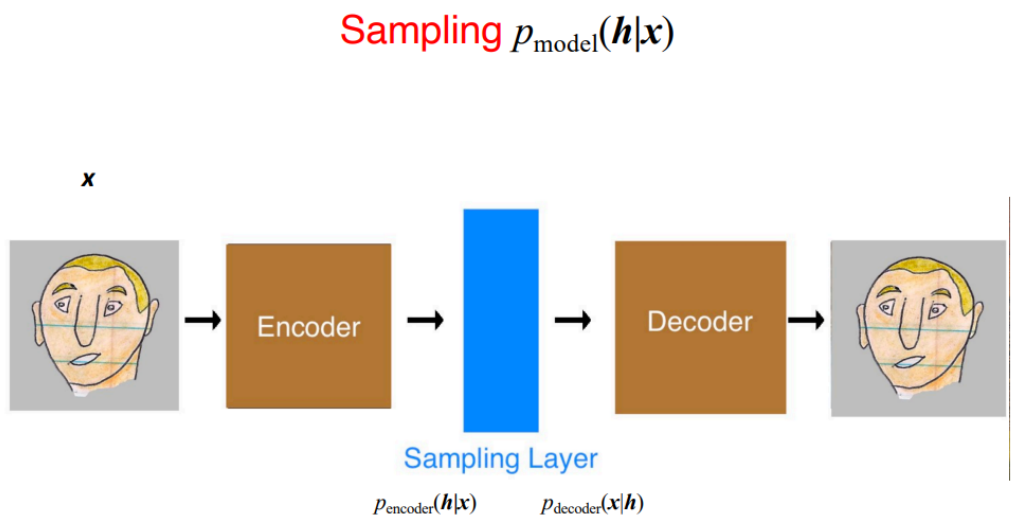


Figure 5.5 example for stochastic autoencoder

Ex: Sampling p(x|h): Deepstyle

- Boil down to a representation which relates to style by iterating neural network through a set of images learn efficient representations.
- Choosing a random numerical description in encoded space will generate new images of styles not seen.
- Using one input image and changing values along different dimensions of feature space you can see how the generated image changes (patterning, color texture) in style space.

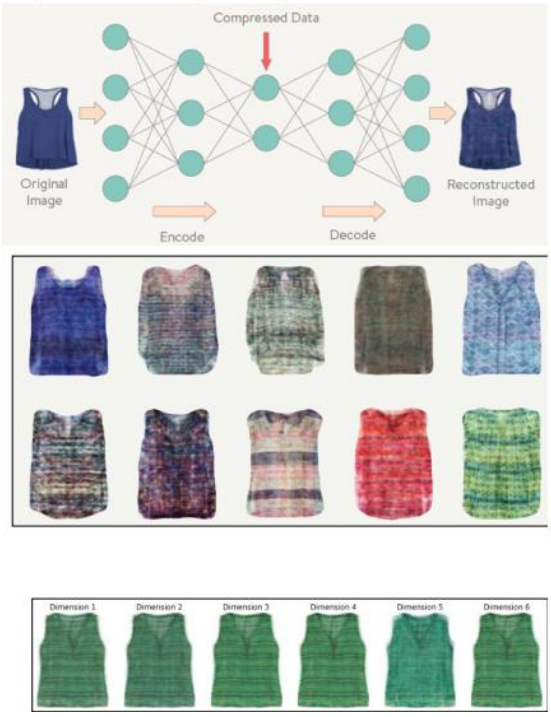


Figure 5.6 example for stochastic autoencoder

5. Explain in detail about Deep Generative Models.

Deep Generative Models

- A ***Generative Model*** is a powerful way of learning any kind of data distribution using unsupervised learning and it has achieved tremendous success in just few years.
- All types of generative models aim at learning the true data distribution of the training set so as to generate new data points with some variations. But it is not always possible to learn the exact distribution of our data either implicitly or explicitly and so we try to model a distribution which is as similar as possible to the true data distribution.
- For this, we can leverage the power of neural networks to learn a function which can approximate the model distribution to the true distribution.
- Two of the most commonly used and efficient approaches are
    - Variational Autoencoders (VAE)
    - Generative Adversarial Networks (GAN).
- VAE aims at maximizing the lower bound of the data log-likelihood and GAN aims at achieving an equilibrium between Generator and Discriminator.

Variational Autoencoder

- Autoencoder is used to encode an input image to a much smaller dimensional representation which can store latent information about the input data distribution. But in a vanilla autoencoder, the encoded vector can only be mapped to the corresponding input using a decoder. It certainly can't be used to generate similar images with some variability.
- To achieve this, the model needs to learn the probability distribution of the training data. VAE is one of the most popular approach to learn the complicated data distribution such as images using neural networks in an unsupervised fashion. It is a probabilistic graphical model rooted in Bayesian inference i.e., the model aims to learn the underlying probability distribution of the training data so that it could easily sample new data from that learned distribution.
- The idea is to learn a low-dimensional latent representation of the training data called latent variables (variables which are not directly observed but are rather inferred through a mathematical model) which we assume to have generated our actual training data.
- These latent variables can store useful information about the type of output the model needs to generate.

- The probability distribution of latent variables z is denoted by P(z). A Gaussian distribution is selected as a prior to learn the distribution P(z) so as to easily sample new data points during inference time.
- Now the primary objective is to model the data with some parameters which maximizes the likelihood of training data X. In short, we are assuming that a low-dimensional latent vector has generated our data x ($x \in X$) and we can map this latent vector to data x using a deterministic function $f(z;\theta)$ parameterized by theta which we need to evaluate (see fig. 1[1]).
- Under this generative process, our aim is to maximize the probability of each data in X which is given as,

$$P\theta(X) = \int P\theta(X, z)dz = \int P\theta(X|z)P\theta(z)dz \quad (1)$$

Here, $f(z;\theta)$has been replaced by a distribution $P\theta(X|z)$.

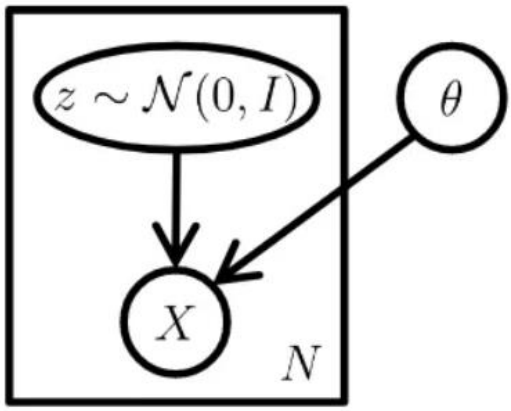


Figure 5.7 Latent vector mapped to data distribution using parameter e[1]

- The intuition behind this maximum likelihood estimation is that if the model can generate training samples from these latent variables then it can also generate similar samples with some variations. In other words, if we sample a large number of latent variables from P(z) and generate x from these variables then the generated x should match the data distribution Pdata(x). Now we have two questions which we need to answer. How to capture the distribution of latent variables and how to integrate Equation 1 over all the dimensions of *z*?
- Obviously it is a tedious task to manually specify the relevant information we would like to encode in latent vector to generate the output image. Rather we rely on neural networks to compute z just with an assumption that this latent vector can be well approximated as a normal distribution so as to sample easily at inference time. If we have a normal distribution of z in n dimensional space

then it is always possible to generate any kind of distribution using a sufficiently complicated function and the inverse of this function can be used to learn the latent variables itself.

- In equation 1, integration is carried over all the dimensions of z and is therefore intractable.
- However, it can be calculated using methods of Monte-Carlo integration which is something not easy to implement.
- So follow an another approach to approximately maximize Pθ(X) in equation 1.
- The idea of VAE is to infer P(z) using P(z|X) which we don't know. We infer P(z|X) using a method called variational inference which is basically an optimization problem in Bayesian statistics.
- First model P(z|X) using simpler distribution Q(z|X) which is easy to find and we try to minimize the difference between P(z|X) and Q(z|X) using KL-divergence metric approach so that our hypothesis is close to the true distribution.
- This is followed by a lot of mathematical equations which I will not be explaining here but you can find it in the original paper. But I must say that those equations are not very difficult to understand once you get the intuition behind VAE.

The final objective function of VAE is :-

$$\log P(X) - D_{KL}[Q(z|X)\|P(z|X)] = E[\log P(X|z)] - D_{KL}[Q(z|X)\|P(z)]$$

- The above equation has a very nice interpretation. The term Q(z|X) is basically our encoder net, z is our encoded representation of data x(x ∈ X) and P(X|z) is our decoder net. So in the above equation our goal is to maximize the log-likelihood of our data distribution under some error given by D_KL[Q(z|X) || P(z|X)]. It can easily seen that VAE is trying to minimize the lower bound of log(P(X)) since P(z|X) is not tractable but the KL-divergence term is >=0. This is same as maximizing E[logP(X|z)] and minimizing D_KL[Q(z|X) || P(z|X)]. We know that maximizing E[logP(X|z)] is a maximum likelihood estimation and is modeled using a decoder net. As I said earlier that we want our latent representation to be close to Gaussian and hence we assume P(z) as N(0, 1). Following this assumption, Q(z|X) should also be close to this distribution. If we assume that it is a Gaussian with parameters μ(X) and Σ(X), the error due to the difference between these two distributions i.e., P(z) and Q(z|X) given by KL-divergence results in a closed form solution given below.

$$D_{KL}[N(\mu(X), \Sigma(X)) \| N(0,1)] = \frac{1}{2} \sum_{k} \left(\exp(\Sigma(X)) + \mu^2(X) - 1 - \Sigma(X) \right)$$

Considering we are optimizing the lower variational bound, our optimization function is :

$\log(P(X|z)) - D\_KL[Q(z|X)||P(z)]$, where the solution of the second is shown above.

- Hence, our loss function will contain two terms. First one is reconstruction loss of the input to output and the second loss is KL-divergence term. Now we can train the network using backpropagation algorithm.
- But there is a problem and that is the first term doesn't only depend on the parameters of P but also on the parameters of Q but this dependency doesn't appear in the above equation.
- So how to backpropagate through the layer where we are sampling z randomly from the distribution Q(z|X) or N[μ(X), Σ(X)] so that P can decode.
- Gradients can't flow through random nodes. We use reparameterization trick (see fig) to make the network differentiable.
- Sample from N(μ(X), Σ(X)) by first sampling ε ~ N(0, I), then computing z=μ(X) + Σ1/2(X)∗ε.
- This has been very beautifully shown in the figure 2[1]? . It should be noted that the feedforward step is identical for both of these networks (left & right) but gradients can only backpropagate through right network.

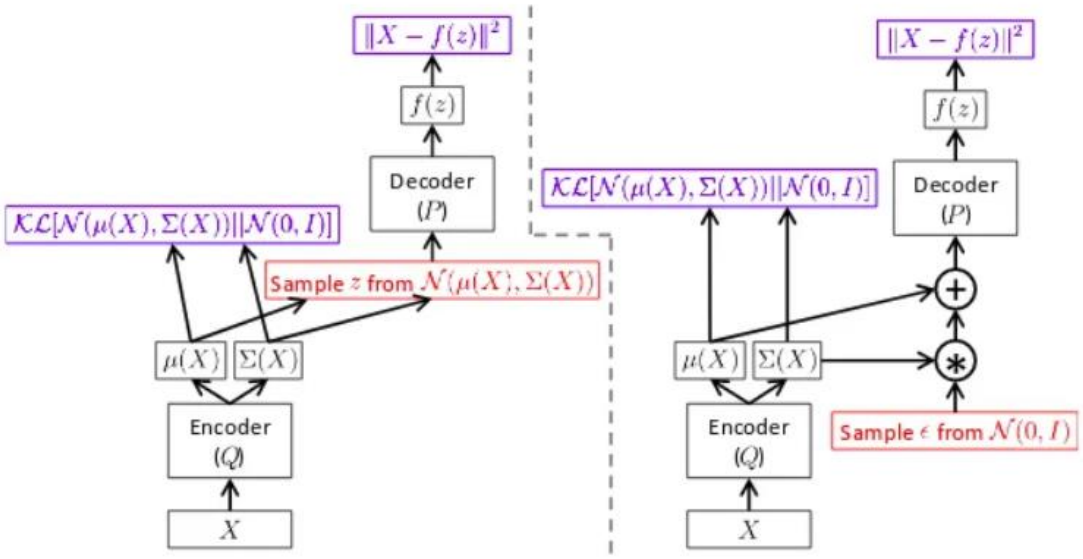


Fig.2. Reparameterization trick used to backpropagate through random nodes [1]

Figure 5.8 Repramenterization trick used to backpropagate through random nodes

- At inference time, we can simply sample z from N(0, 1) and feed it to decoder net to generate new data point. Since we are optimizing the lower variational bound, the quality of the generated image is somewhat poor as compared to state-of-the art techniques like Generative Adversarial Networks.
- The best thing of VAE is that it learns both the generative model and an inference model.
- Although both VAE and GANs are very exciting approaches to learn the underlying data distribution using unsupervised learning but GANs yield better results as compared to VAE.
- In VAE, we optimize the lower variational bound whereas in GAN, there is no such assumption. In fact, GANs don't deal with any explicit probability density estimation.
- The failure of VAE in generating sharp images implies that the model is not able to learn the true posterior distribution. VAE and GAN mainly differ in the way of training. Let's now dive into Generative Adversarial Networks.

6. Explain in detail about Generative adversarial networks.

Generative Adversarial Networks

- Yann LeCun says that adversarial training is the coolest thing since sliced bread. Seeing the popularity of Generative Adversarial Networks and the quality of the results they produce.
- Adversarial training has completely changed the way we teach the neural networks to do a specific task. Generative Adversarial Networks don't work with any explicit density estimation like Variational Autoencoders.
- Instead, it is based on game theory approach with an objective to find Nash equilibrium between the two networks, Generator and Discriminator.
- The idea is to sample from a simple distribution like Gaussian and then learn to transform this noise to data distribution using universal function approximators such as neural networks.
- This is achieved by adversarial training of these two networks. A generator model G learns to capture the data distribution and a discriminator model D estimates the probability that a sample came from the data distribution rather than model distribution.
- Basically, the task of the Generator is to generate natural looking images and the task of the Discriminator is to decide whether the image is fake or real. This can be thought of as a mini-max two player game where the performance of both the networks improves over time.
- In this game, the generator tries to fool the discriminator by generating real images as far as possible and the discriminator tries not to get fooled by the generator by improving its discriminative capability. Below figure 5.9 shows the basic architecture of GAN.

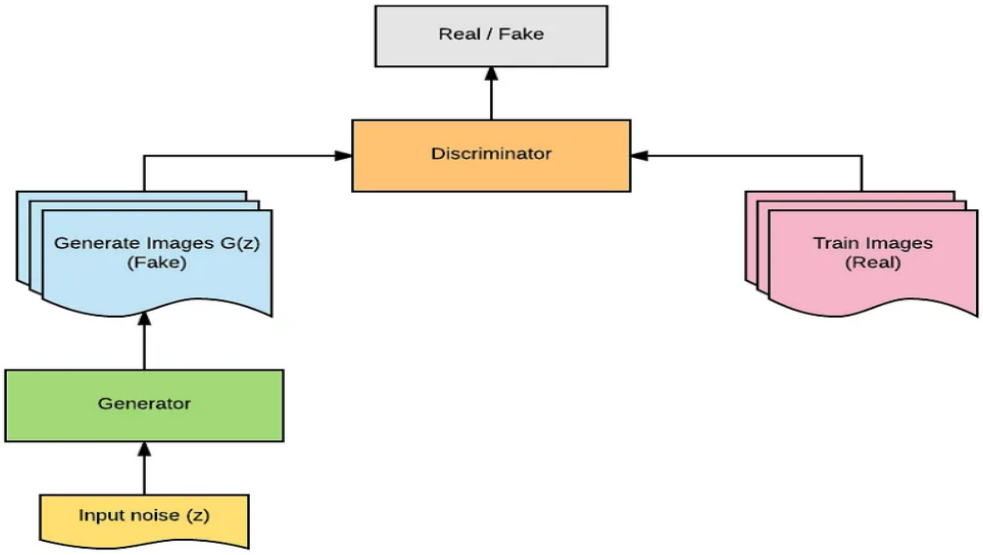


Figure 5.9 Budling block of Generative Adversarial Network

- We define a prior on input noise variables P(z) and then the generator maps this to data distribution using a complex differentiable function with parameters ɵg. In addition to this, we have another network called Discriminator which takes in input x and using another differentiable function with parameters ɵd outputs a single scalar value denoting the probability that x comes from the true data distribution Pdata(x).
- The objective function of the GAN is defined as

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

- In the above equation, if the input to the Discriminator comes from true data distribution then D(x) should output 1 to maximize the above objective function w.r.t D whereas if the image has been generated from the Generator then D(G(z)) should output 1 to minimize the objective function w.r.t G.
- The latter basically implies that G should generate such realistic images which can fool D.
- Maximize the above function w.r.t parameters of Discriminator using Gradient Ascent and minimize the same w.r.t parameters of Generator using Gradient Descent. But there is a problem in optimizing generator objective.
- At the start of the game when the generator hasn't learned anything, the gradient is usually very small and when it is doing very well, the gradients are very high (see Figure 5.10). But we want the opposite behaviour. We therefore maximize E[log(D(G(z))] rather than minimizing E[log(1-D(G(z))]

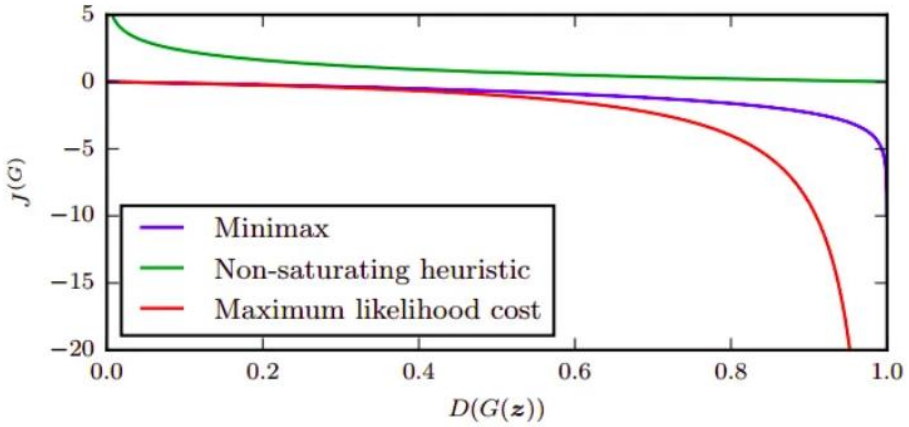


Figure 5.10 Cost for the Generator as a function of Discriminator response on the generated image

- The training process consists of simultaneous application of Stochastic Gradient Descent on Discriminator and Generator.
- While training, we alternate between k steps of optimizing D and one step of optimizing G on the mini-batch.
- The process of training stops when the Discriminator is unable to distinguish ρg and ρdata i.e. D(x, θd) = ½ or when ρg = ρdata.
- One of the earliest model on GAN employing Convolutional Neural Network was **DCGAN** which stands for Deep Convolutional Generative Adversarial Networks.
- This network takes as input 100 random numbers drawn from a uniform distribution and outputs an image of desired shape.
- The network consists of many convolutional, deconvolutional and fully connected layers.
- The network uses many deconvolutional layers to map the input noise to the desired output image.
- Batch Normalization is used to stabilize the training of the network. ReLU activation is used in generator for all layers except the output layer which uses tanh layer and Leaky ReLU is used for all layers in the Discriminator.
- This network was trained using mini-batch stochastic gradient descent and Adam optimizer was used to accelerate training with tuned hyperparameters.
- The results of the paper were quite interesting.
- figure 5.11 showes that the generators have interesting vector arithmetic properties using which we can manipulate images in the way we want.

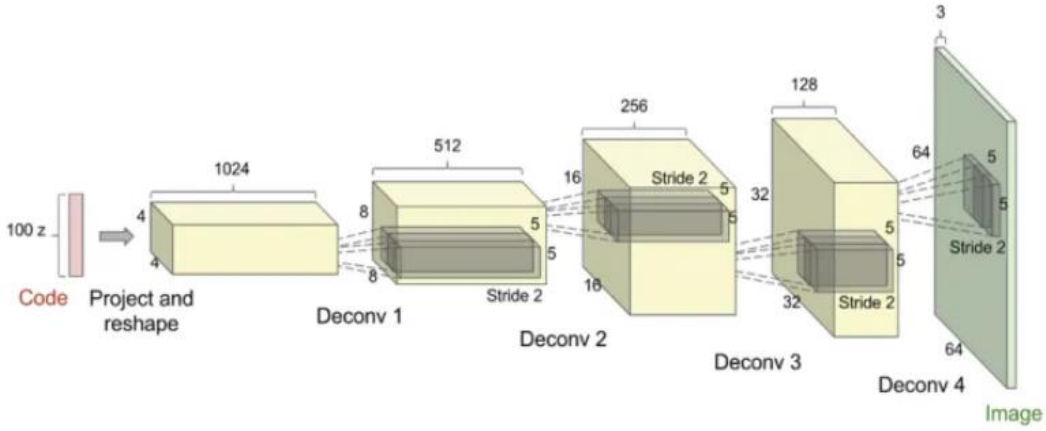


Figure 5.11 Generator of DCGAN

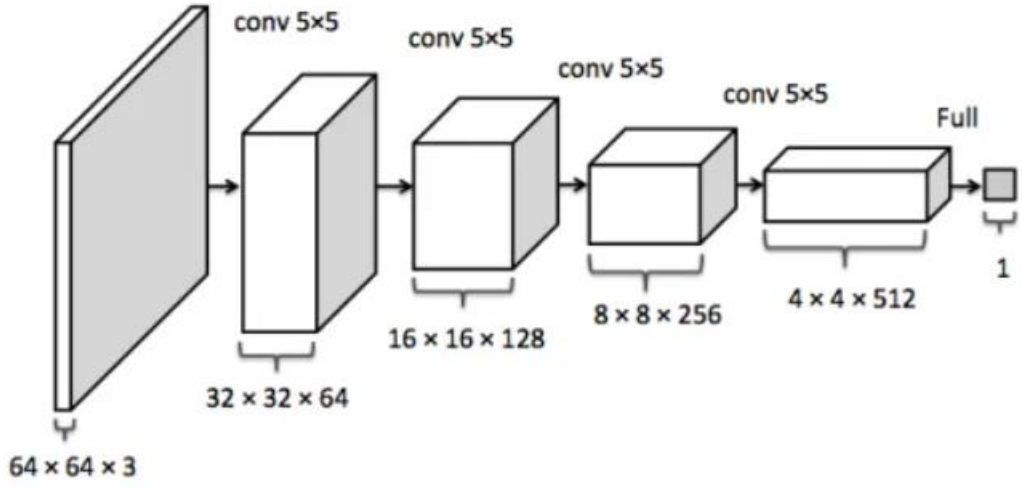


Figure 5.12 Discriminator of DCGAN

- One of the most widely used variation of GANs is conditional GAN which is constructed by simply adding conditional vector along with the noise vector (see Figure 5.13).
- Prior to cGAN, we were generating images randomly from random samples of noise z.
- What if we want to generate an image with some desired features. Is there any way to provide this extra information to the model anyhow about what type of image we want to generate?
- The answer is yes and Conditional GAN is the way to do that. By conditioning the model on additional information which is provided to both generator and discriminator, it is possible to direct the data generation process.

- Conditional GANs are used in a variety of tasks such as text to image generation, image to image translation, automated image tagging etc. A unified structure of both the networks has been shown in the figure 5.13.

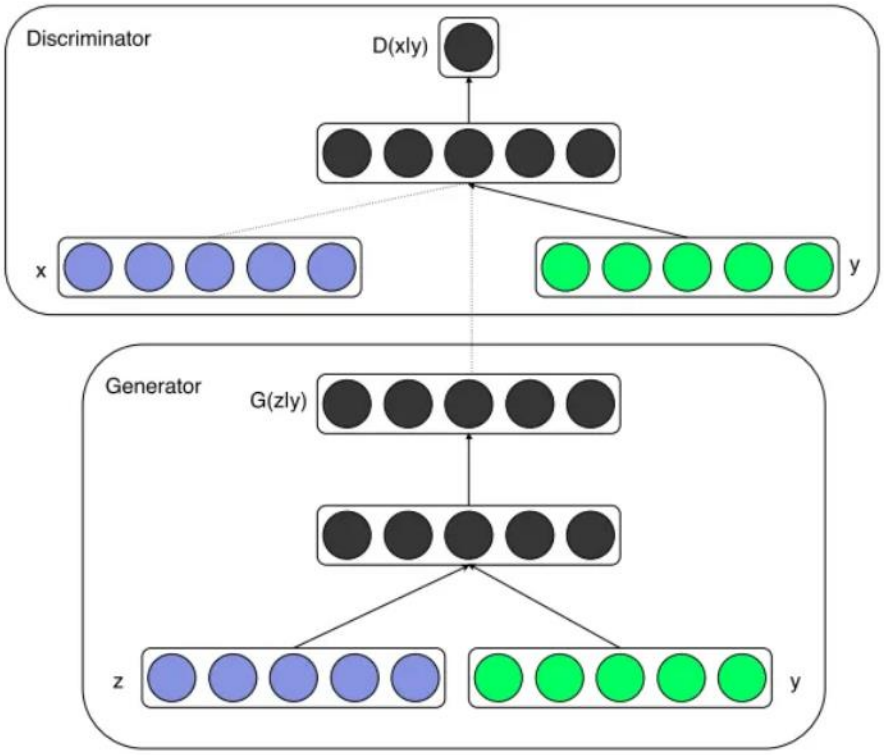


Figure 5.13 A basic example of cGAN with y as the conditioning vector

- One of the cool thing about GANs is that they can be trained even with small training data.
- Indeed, the results of GANs are promising but the training procedure is not trivial especially setting up the hyperparameters of the network.
- Moreover, GANs are difficult to optimize as they don't converge easily.
- Of course, there are some tips and tricks to hack GANs but they may not always help. You can find some of these tips here.
- Also, we don't have any criteria for the quantitative evaluation of the results except to check whether the generated images are perceptually realistic or not.