SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES (AUTONOMOUS)

DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS(MCA)

LECTURE NOTES

Course: DEEP LEARNING

Year/Branch: II MCA/II SEMESTER

Regulation: R24

Prepared By: Dr.R.SARASWATHI

II MCA – II SEMESTER			
COURSE CODE:	**24MCA221**	**CREDITS:**	**4**
COURSE TITLE:	**DEEP LEARNING**	**L-T-P:**	**3-1-0**

PREREQUISITES: *A Course on "Machine learning", Artificial Neural Networks and Knowledge on basic linear algebra may be helpful.*

COURSE EDUCATIONAL OBJECTIVES:

CEO 1: ToacquireknowledgeaboutdifferentmathematicaltoolsforDeepLearning.
CEO 2: To understand fundamentals of artificial neural networks.
CEO 3: To Explore various optimizers and Regularization Techniques
CEO 4: To Learn about Convolutional Neural Networks.
CEO 5:Tocomprehend Object Detection, Autoencoder and GAN Architectures

UNIT-1: MATHEMATICAL TOOLS FOR DEEP LEARNING

Linear Algebra :Matrix, Vector,Transpose,Tensor, operations on elements, Systems of Linear equations,Rank, Norm,expressing a Matrix, Determinant, Trace, Eigen values and Eigen Vectors, Singular Value Decomposition(SVD). **Statistics** – Probability, Random Variable, Binomial Distribution, Poisson Distribution, Normal Distribution, Sampling, Central limit Theorem. **Calculus**- Derivatives, rules forderivatives,Partial derivatives.

UNIT-2: FUNDAMENTALSOFARTIFICIALNEURALNETWORK(ANN)

Understanding the Biological Neuron, Exploring the Artificial Neuron, Early Implementation of ANN – RmcCulloch-Pits mosel of Neuron, Rosenblatt's Perceptron,Types of Activation Functions-linear function, non-linear function, softmax function. Architectures of neural network- Single layer feed forward network, Multi-layer feed forward ANN, Recurrent Neural Network, Convolutional Networks. Learning Process in ANN – Weight of Interconnection between neurons, Gradient Descent and Backpropagation.

UNIT-3: TRAININGDEEPNEURALNETWORK

Initializing Weights- He/ Kaiming initialization, Xavier initialization. Batch, Mini-batch and stochastic gradient descent. Regularization-L1/ L2 regularization, Early stopping, Dropout regularization, data augmentation. Normalization of inputs-Batch Normalization, Batch Normas regularizer.

UNIT-4: CONVOLUTIONAL NETWORKS

Building blocks of CNN, Building a Convolution Neural Network, Popular CNN Architectures-LeNet-5, AlexNet, ZFNET, VGG-16, GoogleNet and ResNet Object Detection- one stage deection techniques – YOLO, SSD,Two stage object detection techniques – R-CNN, fast R-CNN, faster R-CNN, Mask R-CNN, Applications of Object Detection

UNIT-5: SEQUENCE-BASEDMODELS ANDOTHERDLARCHITECTURES

Recurrent Neural Network – Data Preparation for RNN Vanishing Gradient problem and RNN, Applications of RNN, Types of RNN, Limitations of RNN, Longshort-termmemory (LSTM), Gated RNNs, Bidirectional RNNs. Other DL Architectures- Autoencoder, Architecture and its applications, GAN, GAN Architecture and its applications,

TEXTBOOKS:

1. *AmitKumar Das,SaptarsiGoswami,PabitraMitra,AmlanChakrabarti,"DeepLearning", Pearson Paperback,First Edition,2021.*
2. *Ian Goodfellow,YoshuaBengio,AaronCourville, "DeepLearning",MITPress,2016.*

REFERENCE BOOKS:

1. *Josh Patterson and Adam Gibson,"Deeplearning:Apractitioner'sapproach",O'ReillyMedia,First Edition,2017.*
2. *Nikhil Buduma, "Fundamentals of Deep Learning, Designing next-generation machine intelligence algorithms",O'Reilly,ShroffPublishers,2019.*

COURSE OUTCOMES: *On successful completion of this course, students will be able to:*		POs related to COs
CO1	Understand the mathematical background required for Deep learning.	**PO1,PO2**
CO2	Analyze the fundamental concepts of Artificial neural networks.	**PO1,PO2**
CO3	Understand and apply the deep neural networks	**PO1,PO2,PO3,PO4**
CO4	Explore the Purpose of Convolution Neural Network, Popular Architectures of CNN	**PO1,PO2,PO3,PO4,PO8**
CO5	To learn about sequence-based models like RNN, LSTM and Gated RNN and other DL architectures and use for real-world problems	**PO1,PO2,PO3,PO4,PO8**

CO-PO MAPPING (DETAILED; HIGH:3; MEDIUM:2; LOW:1)

Course	**POs / COs**	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**
C215: Deep Learning	**CO215.1**	**3**	**3**	**-**	**-**	**-**	**-**	**-**	**-**
	CO215.2	**3**	**3**	**2**	**-**	**-**	**-**	**-**	**-**
	CO215.3	**3**	**3**	**2**	**3**	**-**	**-**	**-**	**-**
	CO215.4	**3**	**3**	**3**	**2**	**-**	**-**	**-**	**3**
	CO215.5	**3**	**3**	**3**	**3**	**-**	**-**	**-**	**3**
	CO215	**3**	**3**	**2.5**	**2.67**				**3**

UNIT 5

SEQUENCE-BASED MODELS AND OTHER DL ARCHITECTURES

"Deep learning allows machines to learn from data." — Yoshua Bengio

UNIT 5: SEQUENCE-BASED MODELS AND OTHER DL ARCHITECTURES

Recurrent Neural Network – Data Preparation for RNN Vanishing Gradient problem and RNN, Applications of RNN, Types of RNN, Limitations of RNN, Long short-term memory (LSTM), Gated RNNs, Bidirectional RNNs. Other DL Architectures- Autoencoder, Architecture and its applications, GAN, GAN Architecture and its applications.

1. Overview

This unit focuses on advanced deep learning architectures designed to process **sequential and complex data**. It begins with **Recurrent Neural Networks (RNNs)**, which are capable of capturing temporal dependencies in data such as text, speech, and time-series signals. It further explores challenges like the **vanishing gradient problem** and introduces improved models such as **LSTM (Long Short-Term Memory)** and **GRU (Gated Recurrent Units)**.

The unit also covers modern deep learning architectures like **Autoencoders** for data compression and **Generative Adversarial Networks (GANs)** for data generation. These models play a critical role in real-world AI applications such as **speech recognition, machine translation, anomaly detection, and image synthesis**.

2. Objectives

After studying this unit, students will be able to:

- Understand the concept of **sequence-based learning models**
- Explain the working of **Recurrent Neural Networks (RNNs)**
- Analyze the **vanishing and exploding gradient problems**
- Differentiate between **RNN, LSTM, and GRU architectures**
- Understand **Autoencoders and GAN architectures**
- Apply deep learning models to **real-world problems**

3. Learning Outcomes

a) Remembering

- Define RNN, LSTM, GRU, Autoencoders, and GANs
- Identify types of RNN architectures

b) Understanding

- Explain how RNN processes sequential data
- Describe vanishing gradient problem

c) Applying

- Use RNN/LSTM models for tasks like **text prediction or time series forecasting**

d) Analyzing

- Compare RNN, LSTM, and GRU
- Analyze advantages and limitations of each model

e) Evaluating

- Evaluate model performance in different applications

f) Creating

- Design deep learning models for real-world applications like **chatbots or image generation**

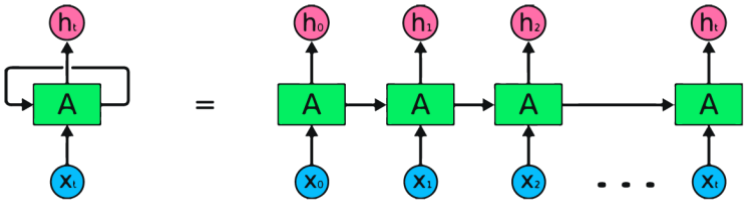
5. Key Terminologies

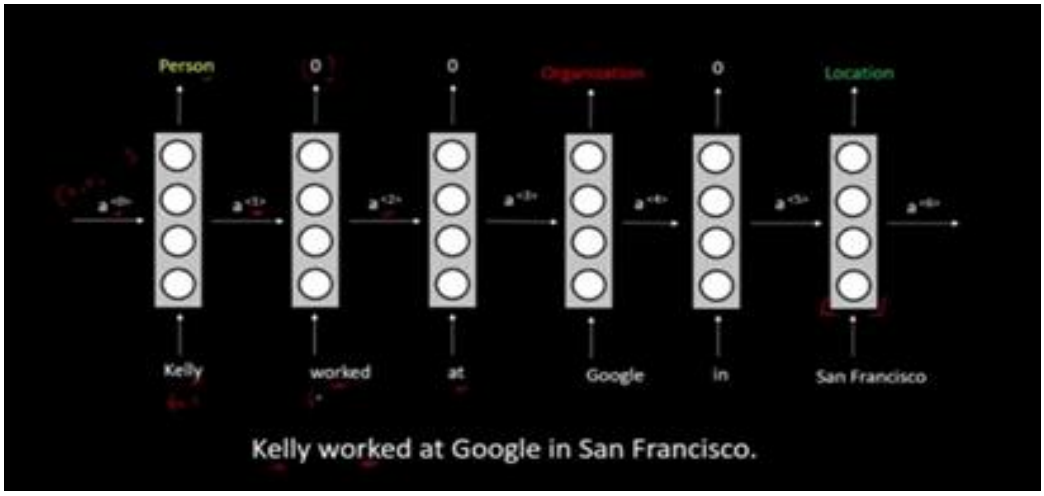
- **Sequential Data**: Data where order matters (e.g., sentences, time series)
- **Hidden State**: Memory of previous inputs in RNN
- **Backpropagation Through Time (BPTT)**: Training method for RNN
- **Vanishing Gradient**: Gradients become too small → learning stops
- **Exploding Gradient**: Gradients become too large → unstable training
- **LSTM**: Advanced RNN with memory gates
- **GRU**: Simplified version of LSTM
- **Autoencoder**: Neural network for data compression
- **GAN**: Model that generates realistic synthetic data

5.1 Recurrent Neural Networks (RNN)

- Recurrent Neural Networks(RNNs) are a type of neural network particularly effective in handling sequential data, such as time series or natural language. Unlike traditional feedforward neural networks, RNNs have a memory that allows them to persist information across different steps in a sequence. This makes them ideal for tasks where the order of input is essential, such as language modelling, translation and speech recognition.
- A recurrent neural network (RNN) is the type of artificial neural network (ANN) that is used in Apple's Siri and Google's voice search.
- Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step.
- In traditional neural networks, all the inputs and outputs are independent of each other.
- But to predict the next word of a sentence, the previous words are required.
- Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer.
- ***Hidden state***, remembers some information about a sequence.
- The state is also referred to as ***Memory State*** since it remembers the previous input to the network.
- It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output.
- This reduces the complexity of parameters, unlike other neural networks.

In the traditional neural network, the inputs and the outputs are independent of each other, whereas the output in RNN is dependent on prior elementals within the sequence. Recurrent networks also share parameters across each layer of the network. In feedforward networks, there are different weights across each node. Whereas RNN shares the same weights within each layer of the network and during gradient descent, the weights and basis are adjusted individually to reduce the loss.

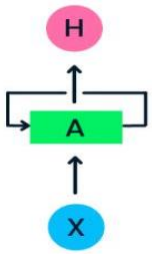




The image above is a simple representation of recurrent neural networks.

How Recurrent Neural Networks Work:

In RNN, the information cycles through the loop, so the output is determined by the current input and previously received inputs.



The input layer X processes the initial input and passes it to the middle layer A. The middle layer consists of multiple hidden layers, each with its activation functions, weights, and biases. These parameters are standardized across the hidden layer so that instead of creating multiple hidden layers, it will create one and loop it over.

Instead of using traditional backpropagation, recurrent neural networks use backpropagation through time (BPTT) algorithms to determine the gradient. In backpropagation, the model adjusts the parameter by calculating errors from the output to the input layer. BPTT sums the error at each time step as RNN shares parameters across each layer.

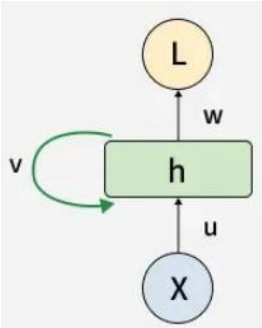
Key Components of RNNs:

There are mainly two components of RNNs:

1. Recurrent Neurons
2. RNN Unfolding

1. Recurrent Neurons

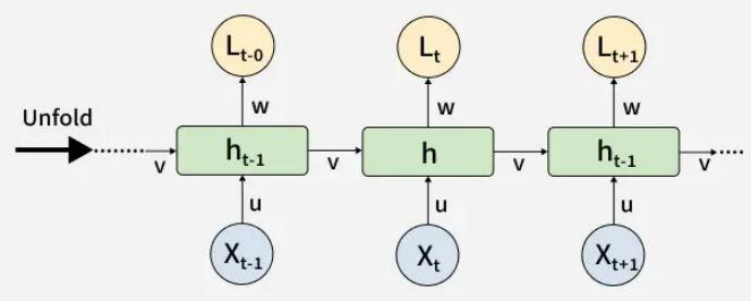
The fundamental processing unit in RNN is a Recurrent Unit. They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

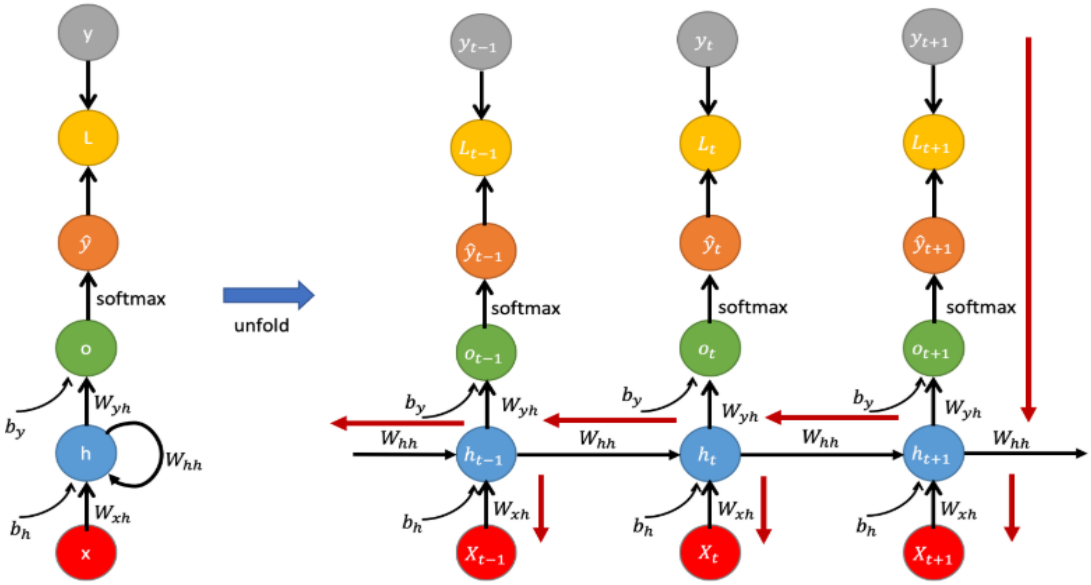


2. RNN Unfolding

RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. During unfolding each step of the sequence is represented as a separate layer in a series illustrating how information flows across each time step.

This unrolling enables back propagation through time (BPTT) a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data.

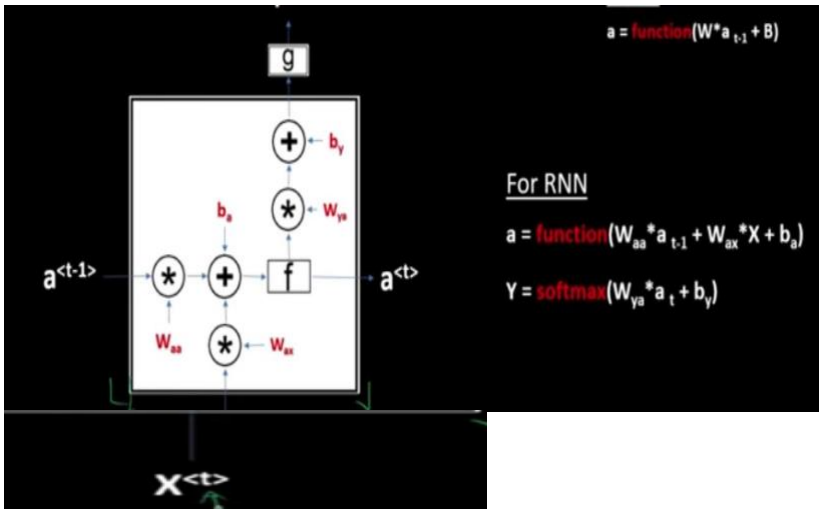




Recurrent Neural Network Architecture

RNNs share similarities in input and output structures with other deep learning architectures but differ significantly in how information flows from input to output. Unlike traditional deep neural networks where each dense layer has distinct weight matrices. RNNs use shared weights across time steps, allowing them to remember information over sequences.

In RNNs the hidden state H_i is calculated for every input X_i to retain sequential dependencies. The computations follow these core formulas:



1. Hidden State Formula (Activation of RNN)

The RNN calculates the new hidden state by combining:

1. Previous hidden state information
2. Current input
3. Bias

Then it applies an activation function.

So the hidden state **stores memory from previous time steps**.

$$a^{\langle t \rangle} = f(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a)$$

Here:

- $a^{\langle t \rangle}$
  Hidden state (activation) at current time step **t**.
- $a^{\langle t-1 \rangle}$
  Hidden state from the **previous time step**.
- $x^{\langle t \rangle}$
  Input at the current time step.
- W_{aa}
  Weight matrix connecting **previous hidden state → current hidden state.**
- W_{ax}
  Weight matrix connecting **input → hidden state.**
- b_a
  Bias vector for the hidden layer.
- f
  Activation function (usually **tanh** or **ReLU**).

2. Output Calculation:

$$y^{\langle t \rangle} = softmax(W_{ya}a^{\langle t \rangle} + b_y)$$

Meaning of Each Term

- $y^{\langle t \rangle}$
  Output at time step **t.**
- W_{ya}
  Weight matrix connecting **hidden state → output layer.**
- b_y
  Bias for the output layer.
- **softmax**
  Activation function used for **classification problems** to convert values into probabilities.

3. Overall Function:

Hidden state:

$$a_t = f(W_{aa}a_{t-1} + W_{ax}x_t + b_a)$$

Output:

$$y_t = softmax(W_{ya}a_t + b_y)$$

5.2 Data Preparation for RNN Vanishing Gradient problem

Data Collection:

- Gather the dataset you want to use for training your RNN. This could be sequential data such as time series data, text data, or any other form of sequential data.

Data Preprocessing:

- Cleaning: Remove any noise or irrelevant information from the data.
- Normalization/Standardization: Scale the data to a smaller range to make it easier for the neural network to learn. This step is particularly important for RNNs because they are sensitive to the scale of input data.
- Feature Engineering: Extract relevant features from the data if necessary. For text data, this could involve tokenization, stemming, or other techniques.

Sequence Generation:

- Convert the sequential data into input-output pairs or sequences.
- Ensure that the sequences are properly aligned and that each input sequence corresponds to the correct output sequence.

Vectorization:

- Convert the sequences into numerical vectors that can be fed into the neural network.
- This often involves encoding categorical variables and converting text data into numerical representations, such as word embeddings or one-hot encodings.

Train-Validation-Test Split:

- Split the dataset into training, validation, and test sets.

Batching:

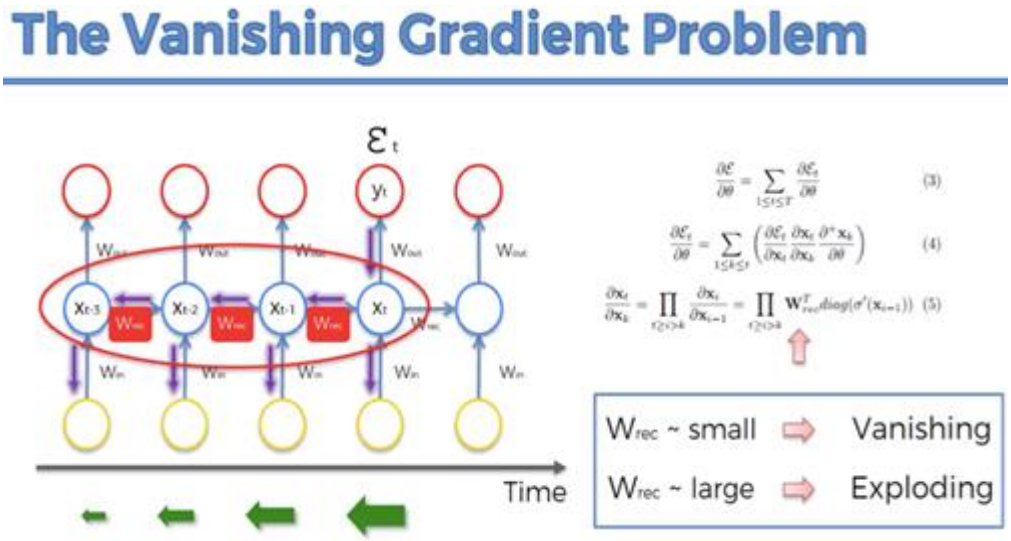
- Group the sequences into batches to speed up the training process. Batching helps the model to generalize better and make use of parallel processing capabilities of modern hardware.

Data Loading:

- Finally, load the pre processed data into a suitable data structure for training the RNN. In Python, this is often done using libraries like TensorFlow or PyTorch, which provide utilities for working with sequential data.

Exploding and Vanishing Gradient Descent

- The **Vanishing Gradient Problem** occurs in deep neural networks and **Recurrent Neural Networks (RNN)** when the gradients become extremely small during backpropagation, preventing the network from learning long-term dependencies.
- The Vanishing Gradient and Exploiding Gradient Problem raises when updating wrec (weight recurring) – the weight that is used to connect the hidden layers to themselves in the unrolled temporal loop.
- if wrec is small, you have vanishing gradient problem, and if wrec is large, you have **exploding gradient problem.**
- when you start with wrec close to zero and multiply xt, xt-1, xt-2, xt-3, … by this value, your gradient becomes less and less with each multiplication.



In **BPTT**, the RNN is **unrolled through time**.

Example:

x1 → a1 → y1
↓
x2 → a2 → y2
↓
x3 → a3 → y3

2. Mathematical Representation

In RNN the hidden state is calculated as

$$a_t = f(W_{rec}a_{t-1} + W_{in}x_t + b)$$

During **Backpropagation Through Time (BPTT)**, the gradient is computed as

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

The gradient depends on repeated multiplication of weights:

$$\prod_{k=1}^{t} W_{rec}$$

If **|Wrec| < 1**, the gradient becomes very small.

Example:

$$0.5 \times 0.5 \times 0.5 \times 0.5 = 0.0625$$

After many multiplications the value approaches **zero**.

Error from **later time steps flows backward** to earlier states.

So gradients depend on **all previous states**.

3. Gradient of Output Layer

Gradient of loss with respect to output weights:

$$\frac{\partial L}{\partial W_{ya}} = \sum_{t=1}^{T} \frac{\partial L^t}{\partial W_{ya}}$$

Since

$$y^t = softmax(W_{ya}a^t + b_y)$$

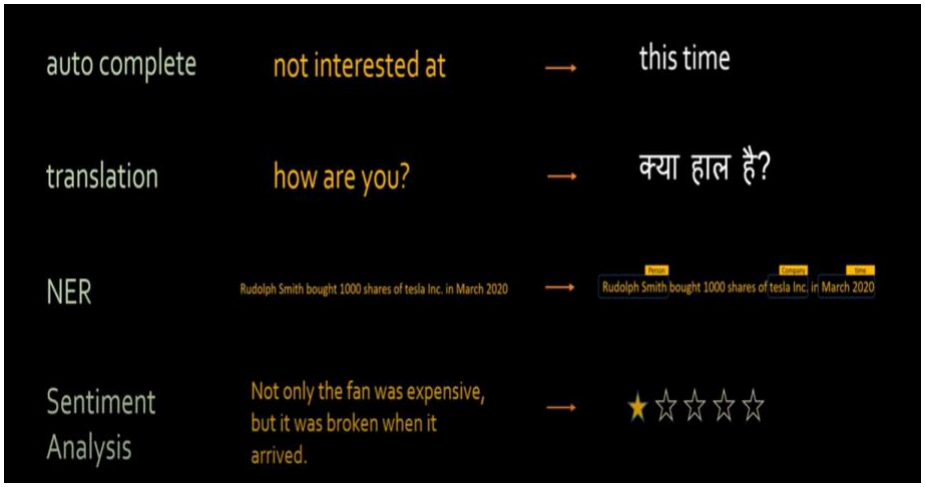
Gradient becomes:

$$\frac{\partial L}{\partial W_{ya}} = \sum_{t=1}^{T} (\hat{y}^t - y^t)a^t$$

Where

- $\hat{y}^t$ = predicted output
- y^t = true output

5.3 Applications of RNN



1. Natural Language Processing (NLP)

Recurrent Neural Networks are widely used in Natural Language Processing to handle sequential text data. RNN remembers previous words in a sentence to understand the context. It is used in applications such as text generation, sentiment analysis, and chatbots. The network processes words one by one while maintaining memory of previous inputs. This helps in understanding language patterns.

2. Machine Translation

RNN is used to translate text from one language to another automatically. It learns the relationship between words in different languages. The model reads the input sentence and generates the translated sentence step by step. Encoder–decoder RNN architecture is commonly used for this task. Examples include English to French or English to Tamil translation systems.

3. Speech Recognition

RNN is used in speech recognition systems to convert spoken language into text. It processes audio signals as sequential data. The network learns patterns in speech and identifies words correctly. This technology is used in voice assistants and voice typing systems. Examples include Google Voice Typing, Siri, and Alexa.

4. Time Series Prediction

RNN is useful for analyzing time-based data and predicting future values. It remembers past data patterns to make predictions. Applications include stock price prediction, weather forecasting, and sales prediction. The model analyzes data collected over time intervals. This helps in identifying trends and future outcomes.

5. Handwriting Recognition

RNN can recognize handwritten characters and words by analyzing sequence patterns. It processes strokes or character sequences to identify the text. This technology is used in digital handwriting recognition systems. It helps convert handwritten notes into digital text. It is also used in signature verification systems.

Video Processing

Recurrent Neural Networks (RNN) are used in video processing to analyze sequences of video frames. The model can understand temporal information between frames in a video. It helps in identifying actions, objects, and activities in videos. RNN is used in applications such as action recognition, video caption generation, and surveillance systems.

Music Generation

RNN is used to generate new music by learning patterns from existing musical sequences. It analyzes notes, rhythm, and melody in the training data. The network then produces new music step by step based on learned patterns. This technology is used in AI music composition and automatic melody generation systems.

5.4 Types of Recurrent Neural Networks:

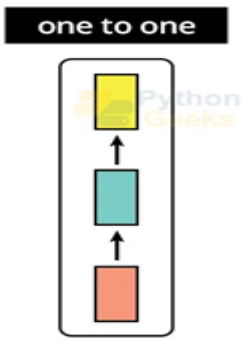
There are **four types of RNNs** based on the number of inputs and outputs in the network:

1. One-to-One RNN

This is the simplest type of neural network architecture where there is a single input and a single output. It is used for straightforward classification tasks such as binary classification where no sequential data is involved.

Use Case:

Image classification (e.g., identifying whether an image contains a cat or dog).



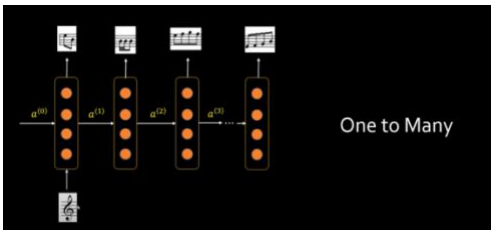
2. One-to-Many RNN

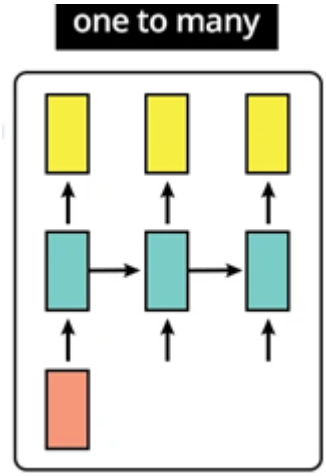
In this model, the RNN receives an **initial input** (for example, gender like *Male* or a start token). After receiving the input, the network generates a **sequence of outputs step by step**. In the diagram, the RNN produces letters **J → O → H → N** to form the name **"John"**. Each step uses the **previous hidden state** to predict the next character.

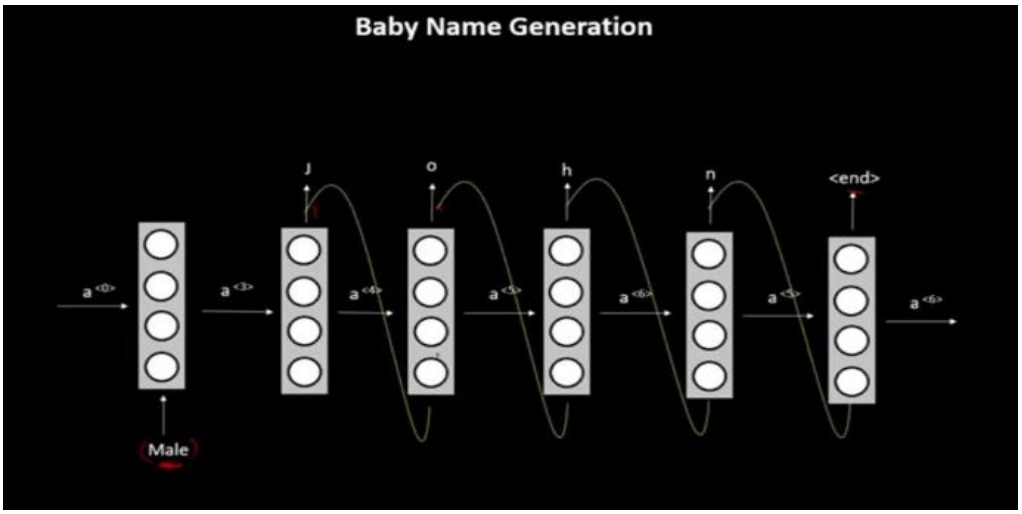
Use Case

This type of RNN is used in:

- **Baby name generation**
- **Text generation**
- **Music generation**
- **Sentence generation**

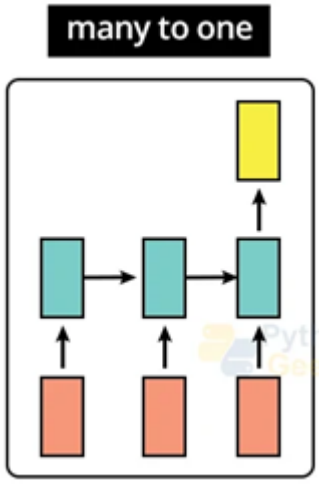


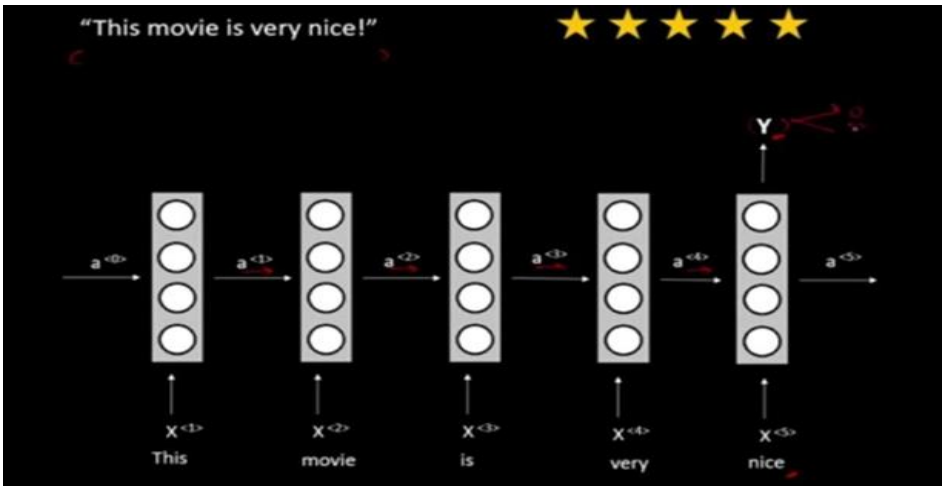




3. Many-to-One RNN

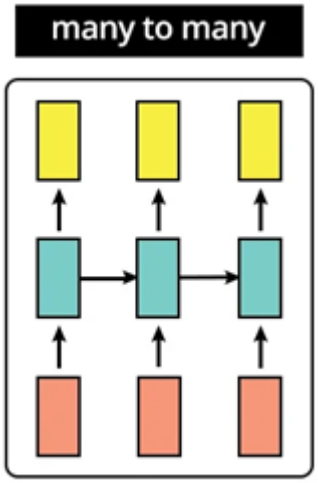
The Many-to-One RNN receives a sequence of inputs and generates a single output. This type is useful when the overall context of the input sequence is needed to make one prediction. In sentiment analysis the model receives a sequence of words (like a sentence) and produces a single output like positive, negative or neutral.

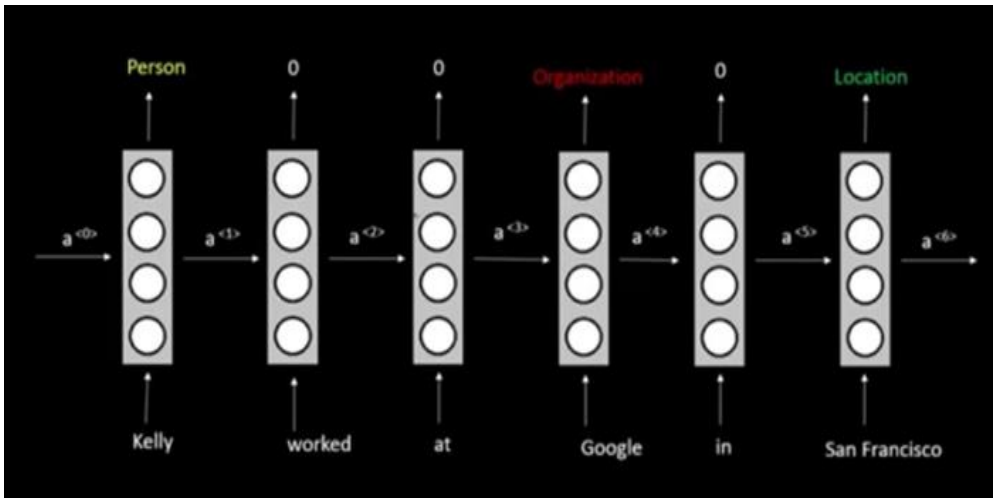




4. Many-to-Many RNN

Many-to-Many RNN is a neural network model that takes a sequence of inputs and generates a sequence of outputs, where each input corresponds to an output.





5.5 Limitations of Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) are widely used for processing sequential data such as text, speech, and time-series information. They maintain a hidden state (memory) that helps capture information from previous inputs. Although RNNs are powerful for sequence modeling, they suffer from several limitations that affect their performance and training efficiency. These challenges led to the development of advanced models such as LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit).

1. Vanishing Gradient Problem

One of the major limitations of RNN is the **vanishing gradient problem**. During training, the gradients are propagated backward through many time steps using the **Backpropagation Through Time (BPTT)** algorithm. When gradients are repeatedly multiplied by small values (less than 1), they become extremely small. As a result, the network fails to update weights effectively.

This prevents the model from learning **long-term dependencies**. For example, when predicting the last word of a long sentence, the model may forget information from earlier words. This problem reduces the learning capability of the network for long sequences.

2. Exploding Gradient Problem

Another limitation is the **exploding gradient problem**. In this case, gradients become extremely large during backpropagation. When weight values are greater than 1, repeated multiplication causes gradients to grow rapidly. This leads to unstable weight updates and makes the training process difficult.

Exploding gradients can cause the model parameters to become very large, resulting in poor convergence or training failure. Techniques such as **gradient clipping** are often used to control this problem.

3. Difficulty in Learning Long-Term Dependencies

RNNs struggle to remember information from earlier time steps when sequences are very long. Although RNNs are designed to maintain memory through hidden states, this memory fades over time due to repeated transformations.

For example, in a long sentence such as:

"The book that I bought yesterday from the story is very interesting."

The network may forget the earlier context while processing later words. Because of this limitation, RNNs cannot effectively capture relationships between distant elements in a sequence.

4. Sequential Processing (Lack of Parallelization)

RNN processes data **step by step in sequence**. Each time step depends on the previous hidden state, which means computations cannot be fully parallelized. Unlike Convolutional Neural Networks (CNNs) or feedforward networks, RNN operations must be performed sequentially.

This increases the training time significantly, especially for large datasets or long sequences. As a result, RNN models are slower compared to modern architectures such as **Transformers**.

5. High Computational Complexity

Training an RNN involves multiple time steps and repeated weight calculations. Each step requires updating hidden states and computing gradients during backpropagation. When the sequence length is large, the computational cost increases significantly.

This makes RNN training expensive in terms of **processing power, memory usage, and time**. For large-scale applications such as language modeling, this becomes a major limitation.

6. Difficulty in Handling Very Long Sequences

When input sequences become extremely long, RNN performance decreases. The network may lose important information from earlier inputs. The hidden state may not effectively store all the relevant information required for prediction.

For example, in document-level text analysis or long speech recordings, RNN models may struggle to capture all dependencies within the sequence.

7. Overfitting Problem

RNN models with many parameters may easily **overfit the training data**. Overfitting occurs when the model learns patterns specific to the training dataset but performs poorly on new data.

This problem can reduce the generalization ability of the model. Techniques such as **dropout, regularization, and early stopping** are used to reduce overfitting.

8. Gradient Instability in Deep RNNs

When RNN networks become deeper or have many layers, gradient instability becomes more severe. The repeated multiplication of gradients across layers and time steps causes unstable learning. This makes training deep RNN models very challenging.

To address this issue, researchers introduced advanced architectures such as **LSTM and GRU**, which use gating mechanisms to control information flow.

Conclusion

Although Recurrent Neural Networks are powerful for sequence modeling tasks such as **speech recognition, machine translation, and text generation**, they have several limitations. Problems like **vanishing gradients, exploding gradients, slow training, difficulty in learning long-term dependencies, and high computational complexity** restrict their performance. To overcome these challenges, improved architectures like **LSTM, GRU, and Transformer models** are widely used in modern deep learning applications.

Variants of Recurrent Neural Networks (RNNs):

There are several variations of RNNs, each designed to address specific challenges or optimize for certain tasks:

1. Vanilla RNN

This simplest form of RNN consists of a single hidden layer where weights are shared across time steps. Vanilla RNNs are suitable for learning short-term dependencies but are limited by the vanishing gradient problem, which hampers long-sequence learning.

2. Bidirectional RNNs

Bidirectional RNNs process inputs in both forward and backward directions, capturing both past and future context for each time step. This architecture is ideal for tasks where the entire sequence is available, such as named entity recognition and question answering.

3. Long Short-Term Memory Networks (LSTMs)

Long Short-Term Memory Networks (LSTMs) introduce a memory mechanism to overcome the vanishing gradient problem. Each LSTM cell has three gates:

Input Gate: Controls how much new information should be added to the cell state.

Forget Gate: Decides what past information should be discarded.

Output Gate: Regulates what information should be output at the current step. This selective memory enables LSTMs to handle long-term dependencies, making them ideal for tasks where earlier context is critical.

4. Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs) simplify LSTMs by combining the input and forget gates into a single update gate and streamlining the output mechanism. This design is computationally efficient, often performing similarly to LSTMs and is useful in tasks where simplicity and faster training are beneficial.

5.6 LSTM (Long Short-Term Memory Networks)

Long Short-Term Memory Networks or LSTM in deep learning, is a sequential neural network that allows information to persist. It is a special type of Recurrent Neural Network which is capable of handling the vanishing gradient problem faced by RNN. LSTM was designed by Hochreiter and Schmidhuber that resolves the problem caused by traditional RNNs and machine learning algorithms.

LSTM Architecture

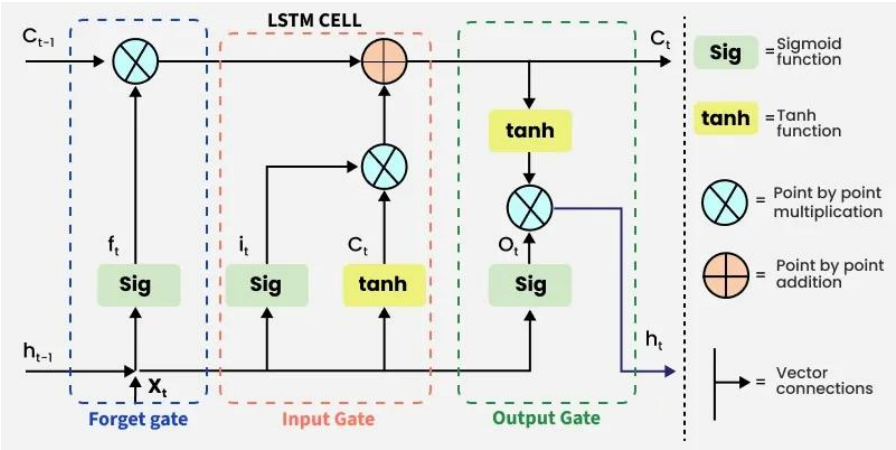
LSTM architectures involves the memory cell which is controlled by three gates:

1. **Input gate**: Controls what information is added to the memory cell.
2. **Forget gate**: Determines what information is removed from the memory cell.
3. **Output gate**: Controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

Working of LSTM

LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells.



Information is retained by the cells and the memory manipulations are done by the gates**.** There are three gates -

1. Forget Gate

The information that is no longer useful in the cell state is removed with the forget gate. Two inputs x_t (input at the particular time) and h_{t-1} (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through sigmoid activation function which gives output in range of [0,1]. If for a particular cell state the output is 0 or near to 0, the piece of information is forgotten and for output of 1 or near to 1, the information is retained for future use.

The equation for the forget gate is:

$$f_t=\sigma(W_f\cdot[h_{t-1},x_t]+b_f)$$

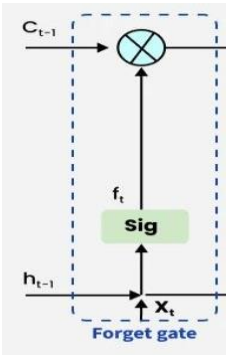
Where:

W_f represents the weight matrix associated with the forget gate.

$[h_{t-1},x_t]$ denotes the concatenation of the current input and the previous hidden state.

b_f is the bias with the forget gate.

σ is the sigmoid activation function.



2. Input gate

The addition of useful information to the cell state is done by the input gate. First the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs $h_{t-1}$and x_t. Then, a vector is created using *tanh* function that gives an output from -1 to +1 which contains all the possible values from h_{t-1} and x_t. At last the values of the vector and the regulated values are multiplied to obtain the useful information. The equation for the input gate is:

$$i_t=\sigma(W_i\cdot[h_{t-1},x_t]+b_i)$$

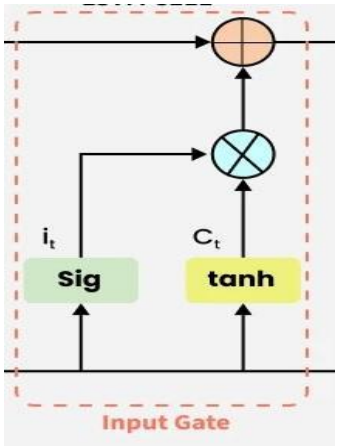
$$\hat{C}_t=\tanh(W_c\cdot[h_{t-1},x_t]+b_c)$$

We multiply the previous state by f_t effectively filtering out the information we had decided to ignore earlier. Then we add i_t*C_t which represents the new candidate values scaled by how much we decided to update each state value.

$$C_t=f_t*C_{t-1}+ i_t*\hat{C}_t$$

where

* denotes element-wise multiplication
tanh is activation function



3. Output gate

The output gate is responsible for deciding what part of the current cell state should be sent as the hidden state (output) for this time step. First, the gate uses a sigmoid function to determine which information from the current cell state will be output. This is done using the previous hidden state h_{t-1} and the current input x_t

$$o_t=\sigma(W_o\cdot[h_{t-1},x_t]+b_o)$$

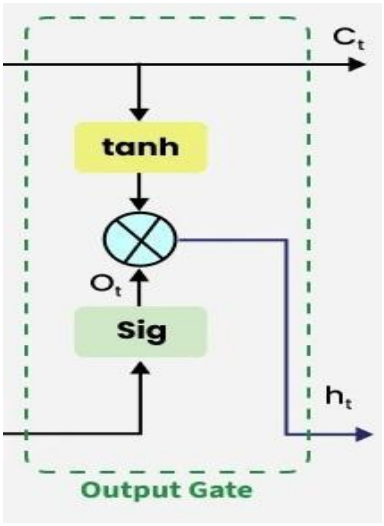
Next, the current cell state C_t is passed through a tanh activation to scale its values between −1 and +1. Finally, this transformed cell state is multiplied element-wise with o_t to produce the hidden state h_t

$$h_t=o_t*\tanh(C_t)$$

Here:

o_t is the output gate activation.
C_t is the current cell state.
* represents element-wise multiplication.
σ is the sigmoid activation function.

This hidden state h_t is then passed to the next time step and can also be used for generating the output of the network.



Applications

Some of the famous applications of LSTM includes:

- **Language Modeling**: Used in tasks like language modeling, machine translation and text summarization. These networks learn the dependencies between words in a sentence to generate coherent and grammatically correct sentences.
- **Speech Recognition**: Used in transcribing speech to text and recognizing spoken commands. By learning speech patterns they can match spoken words to corresponding text.
- **Time Series Forecasting**: Used for predicting stock prices, weather and energy consumption. They learn patterns in time series data to predict future events.
- **Anomaly Detection**: Used for detecting fraud or network intrusions. These networks can identify patterns in data that deviate drastically and flag them as potential anomalies.
- **Recommender Systems**: In recommendation tasks like suggesting movies, music and books. They learn user behavior patterns to provide personalized suggestions.
- **Video Analysis**: Applied in tasks such as object detection, activity recognition and action classification. When combined with Convolutional Neural Networks (CNNs) they help analyze video data and extract useful information.

5.7 Gated Recurrent Unit (GRU)

Gated Recurrent Units (GRUs) are a type of RNN introduced by Cho et al. in 2014. It is a type of recurrent neural network designed to handle sequential data while reducing the complexity of traditional RNNs, hence learning long-term dependencies with faster training and fewer parameters.

The core idea behind GRUs is to use gating mechanisms to selectively update the hidden state at each time step allowing them to remember important information while discarding irrelevant details. GRUs aim to simplify the LSTM architecture by merging some of its components and focusing on just two main gates: the update gate and the reset gate.

How GRU Solve the Limitations of Standard RNN?

- **Gated Mechanisms:** Unlike standard RNNs, GRUs use special gates (Update gate and Reset gate) to control the flow of information within the network. These gates act as filters, deciding what information from the past to keep, forget, or update.
- **Mitigating Vanishing Gradients:** By selectively allowing relevant information through the gates, GRUs prevent gradients from vanishing entirely. This allows the network to learn long-term dependencies even in long sequences.
- **Improved Memory Management:** The gating mechanism allows GRU Activation Function to effectively manage the flow of information. The Reset gate can discard irrelevant past information, and the Update gate controls the balance between keeping past information and incorporating new information. This improves the network's ability to remember important details for longer periods.
- **Faster Training:** Due to the efficient gating mechanisms, GRU Activation Function can often be trained faster than standard RNNs on tasks involving long sequences. The gates help the network learn more effectively, reducing the number of training iterations required.

GRU Architecture:

The GRU architecture consists of the following components:

1. **Input layer:** The input layer takes in sequential data, such as a sequence of words or a time series of values, and feeds it into the GRU.
2. **Reset gate:** The reset gate determines how much of the previous hidden state to forget. It takes as input the previous hidden state and the current input, and produces a vector of numbers that controls the degree to which the previous hidden state is "reset" at the current time step. A value of 0 means the model forgets everything from the previous hidden state, while a value of 1 means it keeps all the information.

   Mathematical Formula:

   $$r_t = \sigma (W_r \cdot [h_{t-1} , x_t] + b_r)$$

   Where:

   - r_t is the reset gate value at time step t
   - W_r is the reset gate weight
   - h_{t-1} is the hidden state from the previous time step
   - x_t is the current input
   - b_r is the bias term
   - σ is the sigmoid activation function that outputs a value between 0 and 1

3. **Update gate:** The update gate determines how much of the candidate activation vector to incorporate into the new hidden state. It takes as input the previous hidden state and the current input, and produces a vector of numbers between 0 and 1 that controls the degree to which the candidate activation vector is incorporated into the new hidden state. A value **of 0 means the model relies entirely on the previous hidden state**, and a value of **1 means it relies entirely on the new input.**

Mathematical Formula:

$$U_t = \sigma (W_u \cdot [h_{t-1} , x_t] + b_u)$$

Where:

- U_t is the update gate value at time step t.
- W_U is the weight for the update gate.
- h_{t-1} is the previous hidden state.
- x_t is the current input.
- b_U is the bias term.

4. **Candidate activation vector(Candidate Hidden State):** The candidate activation vector is a modified version of the previous hidden state that is "reset" by the reset gate and combined with the current input. It is computed using a tanh activation function that squashes its output between -1 and 1.

   Mathematical Formula:

   $$\hat{C}_t = \tanh (W_c \cdot [r_t \cdot h_{t-1} , x_t] + b_c)$$

   Where:

   - $\hat{C}_t$ is the candidate hidden state.
   - W_c is the weight for the candidate hidden state.
   - r_t is the reset gate value.

     The reset gate r_t plays a key role in controlling how much of the previous hidden state should be involved when generating the candidate hidden state.

5. **The Final Hidden State:** Finally, the model combines the candidate hidden state with the previous hidden state, guided by the update gate. The final hidden state is a blend of the previous memory and the new candidate memory. The result is a hidden state that incorporates the relevant parts of both the past and the present.

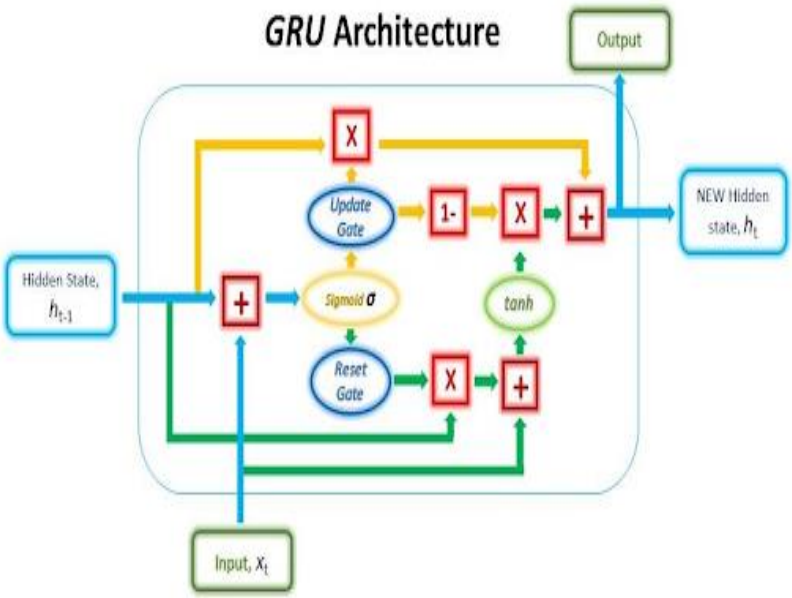
   Mathematical Formula:

   $$h_t = u_t \cdot \hat{C}_t + (1 - u_t) \cdot h_{t-1}$$

   Where:

   - h_t is the final hidden state at time step t.
   - u_t is the update gate value.
   - h_{t-1} is the previous hidden state.
   - $\hat{C}_t$ is the candidate hidden state.

6. **Output layer:** The output layer takes the final hidden state as input and produces the network's output. This could be a single number, a sequence of numbers, or a probability distribution over classes, depending on the task at hand.



$$\hat{C}_t = \tanh\left(W_c \cdot [\, r_t \cdot h_{t-1} , x_t \,] + b_c \right)$$

Advantages of GRU

- **Faster Training and Efficiency**: Compared to LSTMs (Long Short-Term Memory networks), GRUs have a simpler architecture with fewer parameters. This makes them faster to train and computationally less expensive.
- **Effective for Sequential Tasks:** GRUs excel at handling long-term dependencies in sequential data like language or time series. Their gating mechanisms allow them to selectively remember or forget information, leading to better performance on tasks like machine translation or forecasting.
- **Less Prone to Gradient Problems**: The gating mechanisms in GRUs help mitigate the vanishing/exploding gradient problems that plague standard RNNs. This allows for more stable training and better learning in long sequences.

Disadvantages of GRU

- **Less Powerful Gating Mechanism:** While effective, GRUs have a simpler gating mechanism compared to LSTMs which utilize three gates. This can limit their ability to capture very complex relationships or long-term dependencies in certain scenarios.
- **Potential for Overfitting:** With a simpler architecture, LSTM and GRU Architecture might be more susceptible to overfitting, especially on smaller datasets. Careful hyperparameter tuning is crucial to avoid this issue.
- **Limited Interpretability:** Understanding how a GRU Activation Function arrives at its predictions can be challenging due to the complexity of the gating mechanisms. This makes it difficult to analyze or explain the network's decision-making process.

Applications of GRU:

GRUs have several applications in tasks where sequential data is involved, such as:

- Time Series Forecasting: Predicting future values based on past data.
- Natural Language Processing (NLP): Tasks like sentiment analysis, language translation, and text generation.
- Speech Recognition: Converting speech into text by understanding the sequence of sounds.

- Video Processing: Recognizing patterns in sequences of video frames.
- Music Generation: Creating music based on previously heard sequences.

5.8 Bidirectional RNNs

Bidirectional LSTM (BiLSTM) is an advanced type of Recurrent Neural Network (RNN) that processes data in **both** forward and backward directions to capture more context.

Understanding Bidirectional LSTM (BiLSTM)

A Bidirectional LSTM (BiLSTM) consists of two separate LSTM layers:

Forward LSTM: Processes the sequence from start to end

Backward LSTM: Processes the sequence from end to start

The outputs of both LSTMs are then combined to form the final output. Mathematically, the final output at time t is computed as:

At each time step t:

Forward hidden state:

$$\overrightarrow{h_t} = LSTM(x_t, \overrightarrow{h_{t-1}})$$

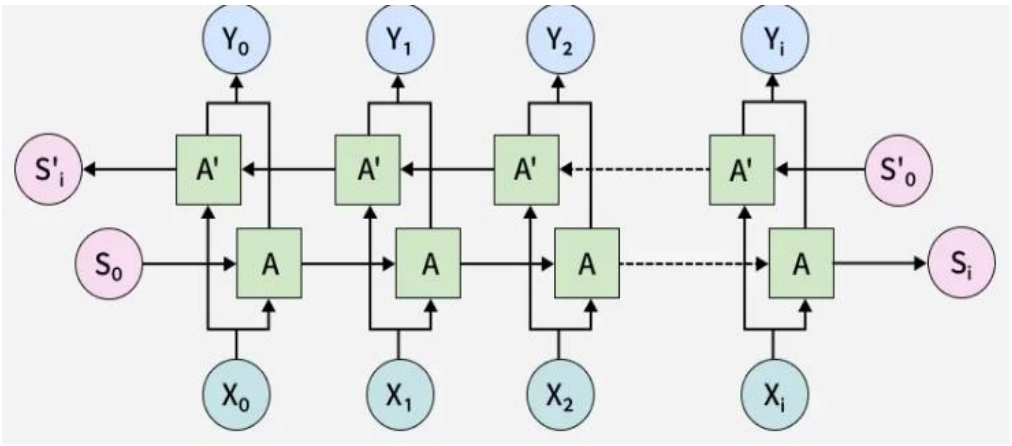
Backward hidden state:

$$\overleftarrow{h_t} = LSTM(x_t, \overleftarrow{h_{t+1}})$$

Final Output:

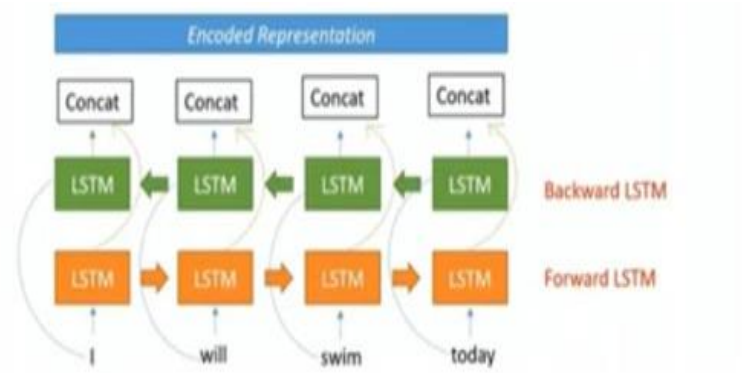
$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$$

The following diagram represents the BiLSTM layer:



Here:

- Xi*Xi* is the input token
- Yi*Yi* is the output token
- A*A* and A′*A′* are Forward and backward LSTM units
- The final output of Yi*Yi* is the combination of A*A* and A′*A′* LSTM nodes.



Working of BiLSTM

1. Input sequence is fed into two LSTMs
2. One processes forward
3. Another processes backward
4. Outputs are combined
5. Final output contains complete context

Advantages

- Captures **past and future context**
- Improves accuracy in sequence learning
- Handles long-term dependencies better

Disadvantages

- High computational cost
- Not suitable for real-time (needs full sequence)

Applications

- Natural Language Processing (NLP)
- Machine Translation
- Sentiment Analysis
- Speech Recognition
- Named Entity Recognition

5.9 Autoencoders

Autoencoders are a special type of unsupervised feed forward neural network. The main application of Autoencoders is to accurately capture the key aspects of the provided data to provide a compressed version of the input data, generate realistic synthetic data, or flag anomalies.

1. **Encoder**:
   - ✓ The encoder compresses ***the input data into a lower-dimensional representation.***
   - ✓ It consists of layers that reduce the dimensionality of the input, effectively performing a type of data compression.
   - ✓ The output of the encoder is often referred to as the "***bottleneck" or "latent space".***
2. **Bottleneck:**

- ✓ The bottleneck is the final layer of the encoder, ***where the dimensionality is significantly reduced.***
- ✓ This compressed representation is the output of the encoder.

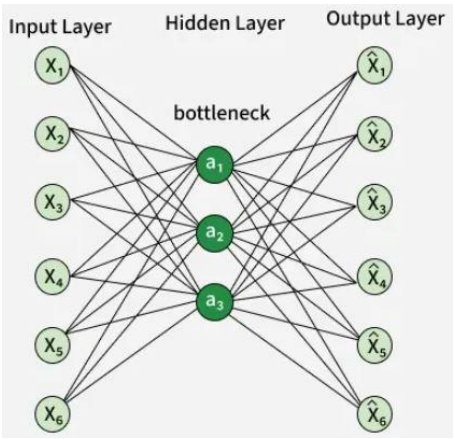
3. **Decoder**:
   - ✓ The decoder reconstructs the input ***data from the lower-dimensional representation produced by the encoder.***
   - ✓ It consists of layers that increase the dimensionality of the data, reconstructing it to match the original input dimensions.

Autoencoders are composed of 2 key fully connected feedforward neural networks:

- **Encoder**: compresses the input data to remove any form of noise and generates a latent space/bottleneck. Therefore, the output neural network dimensions are smaller than the input and can be adjusted as a hyperparameter in order to decide how much lossy our compression should be.
- **Decoder**: making use of only the compressed data representation from the latent space, tries to reconstruct with as much fidelity as possible the original input data (the architecture of this neural network is, therefore, generally a mirror image of the encoder). The "goodness" of the prediction can then be measured by calculating the reconstruction error between the input and output data using a loss function.

Architecture of Autoencoder:

An autoencoder's architecture consists of three main components that work together to compress and then reconstruct



data which are as follows:

1. Encoder

It compress the input data into a smaller, more manageable form by reducing its dimensionality while preserving important information. It has three layers which are:

- Input Layer: This is where the original data enters the network. It can be images, text features or any other structured data.
- Hidden Layers: These layers perform a series of transformations on the input data. Each hidden layer applies weights and activation functions to capture important patterns, progressively reducing the data's size and complexity.
- Output(Latent Space): The encoder outputs a compressed vector known as the latent representation or encoding. This vector captures the important features of the input data in a condensed form helps in filtering out noise and redundancies.

2. Bottleneck (Latent Space)

It is the smallest layer of the network which represents the most compressed version of the input data. It serves as the information bottleneck which force the network to prioritize the most significant features. This compact representation helps the model learn the underlying structure and key patterns of the input helps in enabling better generalization and efficient data encoding.

3. Decoder

It is responsible for taking the compressed representation from the latent space and reconstructing it back into the original data form.

- Hidden Layers: These layers progressively expand the latent vector back into a higher-dimensional space. Through successive transformations decoder attempts to restore the original data shape and details.
- Output Layer: The final layer produces the reconstructed output which aims to closely resemble the original input. The quality of reconstruction depends on how well the encoder-decoder pair can minimize the difference between the input and output during training.

Applications of Autoencoders

1. **Dimensionality Reduction**: Reducing the number of features while preserving important information.
2. **Data Denoising**: Removing noise from data, such as images or signals.
3. **Anomaly Detection**: Identifying unusual patterns that do not fit the learned representation of normal data.

1. Dimensionality Reduction: Reducing the number of features while preserving important information.

- **Undercomplete autoencoders** are those that are used for **dimensionality reduction**.
- These can be used as a pre-processing step for dimensionality reduction as they can perform fast and accurate dimensionality reductions without losing much information.
- Furthermore, while dimensionality reduction procedures like PCA can only perform linear dimensionality reductions, ***undercomplete autoencoders can perform large-scale non-linear dimensionality reductions. i.e for images***

2.Data Denoising: Removing noise from data, such as images or signals.

- Unlike traditional methods of denoising, autoencoders do not search for noise, they extract the image from the noisy data that has been fed to them via learning a representation of it. The representation is then decompressed to form a noise-free image.
- Denoising autoencoders thus can denoise complex images that cannot be denoised via traditional methods.

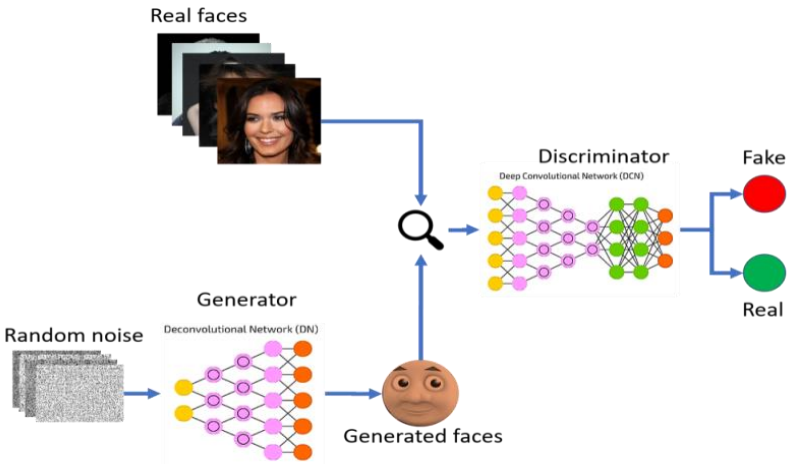
3. Anomaly Detection: Identifying unusual patterns that do not fit the learned representation of normal data.

- Undercomplete autoencoders can also be used for anomaly detection.
- For example—consider an autoencoder that has been trained on a specific dataset *P*. For any image sampled for the training dataset, the autoencoder is bound to give a low reconstruction loss and is supposed to reconstruct the image as is.
- For any image which is not present in the training dataset, however, the autoencoder cannot perform the reconstruction, as the latent attributes are not adapted for the specific image that has never been seen by the network.

- As a result, the outlier image gives off a very high reconstruction loss and can easily be identified as an anomaly with the help of a proper threshold.

5.10 Adversarial Generative Networks

- Generative Adversarial Networks (GANs) were developed in 2014 by Ian Goodfellow and his teammates.
- GAN is basically an approach to generative modeling that generates a new set of data based on training data that look like training data.
- GANs have two main blocks(two neural networks) which compete with each other and are able to capture, copy, and analyze the variations in a dataset.
- A Generative Adversarial Network (GAN) is a deep learning architecture that involves two neural networks, the generator, and the discriminator, competing against each other to generate new data from a given training dataset.
- The ***generator creates synthetic data resembling real data,*** while the ***discriminator evaluates whether the data is real or synthetic.***
- **The goal is for the generator to produce data so realistic that the discriminator cannot reliably tell the difference between real and synthetic data.**
- GANs have various applications like ***Synthetic Data Generation, data augmentation***
- Sample random noise.
- Produce generator output from sampled random noise.
- Get discriminator "Real" or "Fake" classification for generator output.
- Calculate loss from discriminator classification.
- Backpropagate through both the discriminator and generator to obtain gradients.
- Use gradients to change only the generator weights.



Architecture of GAN

GAN consist of two main models that work together to create realistic synthetic data.

1. Generator Model

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

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

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

where

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

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

2. Discriminator Model

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

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

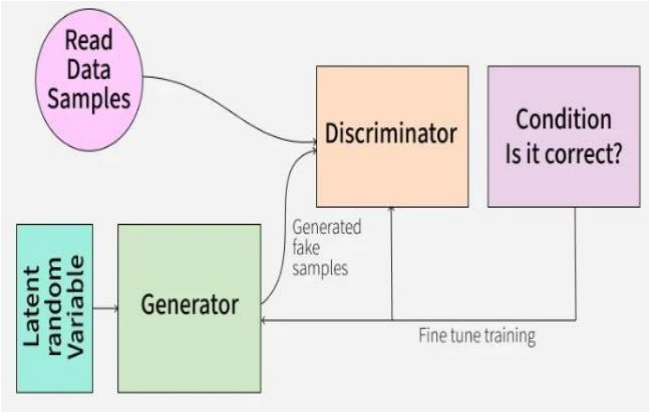
$J_D = -\frac{1}{m}\sum_{i=1}^{m} logD(xi) - \frac{1}{m}\sum_{i=1}^{m} log(1 - D(G(Zi))$

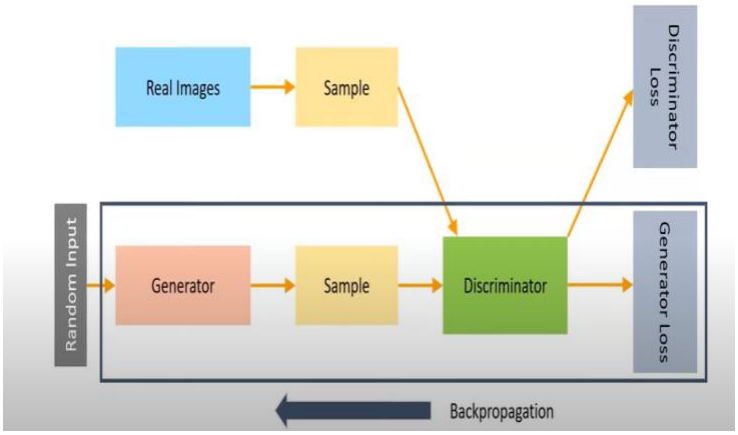
- J_D measures how well the discriminator classifies real and fake samples.
- x_i is a real data sample.
- $G(z_i)$ is a fake sample from the generator.
- $D(x_i)$ is the discriminator's probability that xi is real.
- $D(G(z_i))$ is the discriminator's probability that the fake sample is real.

The discriminator wants to correctly classify real data as real (maximize logD(xi) and fake data as fake (maximize log(1−D(G(zi)))

How does a GAN work?

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





1. Generator's First Move
    - The generator starts with a random noise vector like random numbers. It uses this noise as a starting point to create a fake data sample such as a generated image.
    - The generator's internal layers transform this noise into something that looks like real data.
2. Discriminator's Turn
    - The discriminator receives two types of data:
        - Real samples from the actual training dataset.
        - Fake samples created by the generator.
    - Discriminator's job is to analyze each input and find whether it's real data or something Generator cooked up. It outputs a probability score between 0 and 1. A score of 1 shows the data is likely real and 0 suggests it's fake.
3. Adversarial Learning
    - If the discriminator correctly classifies real and fake data it gets better at its job.
    - If the generator fools the discriminator by creating realistic fake data, it receives a positive update and the discriminator is penalized for making a wrong decision.
4. Generator's Improvement
    - Each time the discriminator mistakes fake data for real, the generator learns from this success.
    - Through many iterations, the generator improves and creates more convincing fake samples.
5. Discriminator's Adaptation
    - The discriminator also learns continuously by updating itself to better spot fake data.
    - This constant back-and-forth makes both networks stronger over time.
6. Training Progression
    - As training continues, the generator becomes highly proficient at producing realistic data.
    - Eventually the discriminator struggles to distinguish real from fake shows that the GAN has reached a well-trained state.
    - At this point, the generator can produce high-quality synthetic data that can be used for different applications.

Types of GAN

There are several types of GANs each designed for different purposes. Here are some important types:

1. Vanilla GAN:

Vanilla GAN is the simplest type of GAN. It consists of:

- A generator and a discriminator

- Both are built using multi-layer perceptrons (MLPs).
- The model optimizes its mathematical formulation using stochastic gradient descent (SGD).

While foundational, Vanilla GAN can face problems like:

- Mode collapse: The generator produces limited types of outputs repeatedly.
- Unstable training: The generator and discriminator may not improve smoothly.

2. Conditional GAN (CGAN):

- Conditional GAN (CGAN) adds an additional conditional parameter to guide the generation process.
- Instead of generating data randomly they allow the model to produce specific types of outputs.

Working of CGANs:

- A conditional variable (y) is fed into both the generator and the discriminator.
- This ensures that the generator creates data corresponding to the given condition (e.g generating images of specific objects).
- The discriminator also receives the labels to help distinguish between real and fake data.

Example: Instead of generating any random image, CGAN can generate a specific object like a dog or a cat based on the label.

3. Deep Convolutional GAN (DCGAN):

Deep Convolutional GAN (DCGAN) are among the most popular types of GANs used for image generation.

They are important because they:

- Uses Convolutional Neural Networks (CNNs) instead of simple multi-layer perceptrons (MLPs).
- Max pooling layers are replaced with convolutional stride helps in making the model more efficient.
- Fully connected layers are removed, which allows for better spatial understanding of images.

DCGANs are successful because they generate high-quality, realistic images.

4. Laplacian Pyramid GAN (LAPGAN):

Laplacian Pyramid GAN (LAPGAN) is designed to generate ultra-high-quality images by using a multi-resolution approach.

Working of LAPGAN:

- Uses multiple generator-discriminator pairs at different levels of the Laplacian pyramid.
- Images are first down sampled at each layer of the pyramid and upscaled again using Conditional GAN (CGAN).
- This process allows the image to gradually refine details and helps in reducing noise and improving clarity.

Due to its ability to generate highly detailed images, LAPGAN is considered a superior approach for photorealistic image generation.

5. Super Resolution GAN (SRGAN):

Super-Resolution GAN (SRGAN) is designed to increase the resolution of low-quality images while preserving details.

Working of SRGAN:

- Uses a deep neural network combined with an adversarial loss function.
- Enhances low-resolution images by adding finer details helps in making them appear sharper and more realistic.
- Helps to reduce common image upscaling errors such as blurriness and pixelation.

6. Cycle GAN:

- It is released in 2017 which performs the task of Image Translation.

- Suppose we have trained it on a horse image dataset and we can translate it into zebra images.

7. Dual Video Discriminator GAN:

- DVD-GAN is a generative adversarial network model for video generation built upon the BigGAN architecture.
- DVD-GAN uses two discriminators: a Spatial Discriminator and a Temporal Discriminator.

Steps for Training GAN:

- Define the problem
- Choose the architecture of GAN
- Train discriminator on real data
- Generate fake inputs for the generator
- Train discriminator on fake data
- Train generator with the output of the discriminator

Applications of Generative Adversarial Networks (GANs):

- Generate new data from available data – It means generating new samples from an available sample that is not similar to a real one.
- Generate realistic pictures of people that have never existed.
- GANs is not limited to Images, It can generate text, articles, songs, poems, etc.
- Generate Music by using some clone Voice – If you provide some voice then GANs can generate a similar clone feature of it.
- Text to Image Generation
- Creation of anime characters in Game Development and animation production.
- Image to Image Translation – We can translate one Image to another without changing the background of the source image. For example, GANs can replace a dog with a cat.
- Low resolution to High resolution – If you pass a low-resolution Image or video, GAN can produce a high-resolution Image version of the same.
- Prediction of Next Frame in the video – By training a neural network on small frames of video, GANs are capable to generate or predict a small next frame of video.
- Interactive Image Generation – This means that GANs can generate images and video footage in an artistic form if they train on the right real dataset.

Unit Highlights

- Focus on **sequence-based deep learning models** such as RNN, LSTM, and GRU
- Understanding of **memory mechanisms in neural networks**
- Detailed study of **vanishing and exploding gradient problems**
- Exploration of **advanced architectures** like Autoencoders and GANs
- Real-world applications in **NLP, speech recognition, and image generation**
- Comparative analysis of **RNN variants and their improvements**
- Introduction to **generative models for synthetic data creation**

Case Study: Speech Recognition using LSTM

Problem Statement

Develop a system that converts spoken language into text accurately.

Approach

- Use **Recurrent Neural Networks (RNN)** and improve performance using **LSTM**
- Input: Audio signals (sequential data)
- Process:
    - Convert audio into features (MFCC)
    - Feed sequence data into LSTM model
    - Model learns temporal dependencies in speech
- Output: Text transcription

Real-World Example

Voice assistants like:

- Google Voice Assistant
- Apple Siri
- Amazon Alexa

Why LSTM is Used

- Handles **long-term dependencies**
- Overcomes **vanishing gradient problem**
- Maintains memory of previous speech context

Outcome

- Improved accuracy in speech recognition
- Better understanding of continuous speech patterns

Case Study 2 (Optional – GAN)

Image Generation using GAN

- GANs are used to generate **realistic human faces**
- Example: AI-generated faces that do not exist
- Used in:
    - Gaming
    - Film industry
    - Data augmentation

Book References

1. *Deep Learning* — Ian Goodfellow, Yoshua Bengio, Aaron Courville ,Publisher: MIT Press, 2016
2. *Neural Networks and Deep Learning: A Textbook* — Charu C Aggarwal, Publisher: Springer, 2018
3. *Pattern Recognition and Machine Learning* — Christopher M Bishop, Publisher: Springer, 2006
4. *Deep Learning with Python* — François Chollet, Publisher: Manning Publications, 2017
5. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* — Aurélien Géron Publisher: O'Reilly Media, 2019
6. *Artificial Intelligence: A Modern Approach* — Stuart Russell, Peter Norvig, Publisher: Pearson, 4th Edition, 2020.

UNIT V - SEQUENCE MODELS & OTHER ARCHITECTURES

PART –A		
1.	Define Recurrent Neural Network (RNN).	L1
2.	Explain how RNN differs from Feedforward Neural Network.	L2
3.	What is vanishing gradient problem in RNN?	L1
4.	Why does vanishing gradient occur during backpropagation?	L2
5.	Define Backpropagation Through Time (BPTT).	L1
6.	What is Long Short-Term Memory (LSTM) network?	L1
7.	State the purpose of forget gate in LSTM.	L2
8.	Define Gated Recurrent Unit (GRU).	L1
9.	Differentiate between LSTM and GRU.	L2
10.	What is Bidirectional RNN?	L1
11.	Define Autoencoder.	L1
12.	Explain the role of encoder and decoder in Autoencoder.	L2
13.	What is Generative Adversarial Network (GAN)?	L1
14.	Explain the roles of Generator and Discriminator in GAN.	L2
15.	Mention two applications of sequence models.	L2
PART –B		
1.	Explain the architecture of Recurrent Neural Network (RNN) with neat diagram. Discuss its working and limitations.	L3
2.	Explain the vanishing gradient problem in RNN and how LSTM overcomes it.	L4
3.	Describe the architecture of LSTM network. Explain the functions of input gate, forget gate and output gate with equations.	L3
4.	Compare LSTM and GRU architectures. Discuss advantages and disadvantages of each.	L4
5.	Explain Backpropagation Through Time (BPTT) algorithm with suitable example.	L3
6.	Discuss Bidirectional RNN and its applications in Natural Language Processing.	L4
7.	Explain Autoencoder architecture. Discuss types of autoencoders and their applications.	L3
8.	Explain Generative Adversarial Networks (GANs). Describe the roles of Generator and Discriminator with loss functions.	L4
9.	Discuss applications of sequence models in Speech Recognition, Machine Translation and Text Generation.	L5
10.	Analyze the importance of LSTM in handling long-term dependencies in sequence data.	L5
11.	Compare RNN, LSTM and GRU in terms of architecture, performance and computational complexity	L4
12.	Design a sequence model for sentiment analysis and explain the steps involved	L6