DL QB

UNIT-4

1. Explain the difference between ML , ANN , RNN , CNN & LSTM.

1. Machine Learning (ML):

- **Definition**: Machine Learning is a subset of Artificial Intelligence (AI) that enables systems to learn from data and improve performance without being explicitly programmed.
- **Goal**: To develop models that can make predictions or decisions based on input data.
- **Types**:
  - **Supervised Learning**: Uses labeled data (e.g., classification, regression).
  - **Unsupervised Learning**: Uses unlabeled data (e.g., clustering).
  - **Reinforcement Learning**: Agents learn through rewards and penalties.
- **Example**: Email spam detection, price prediction, recommendation systems.

---

2. Artificial Neural Networks (ANN):

- **Definition**: ANN is a type of ML model inspired by the structure of the human brain, composed of layers of interconnected "neurons".
- **Architecture**:
  - **Input Layer**: Takes the features.
  - **Hidden Layers**: Perform computations through weighted connections.
  - **Output Layer**: Gives the prediction.
- **Use Cases**: Pattern recognition, image classification, simple time series prediction.

---

3. Convolutional Neural Networks (CNN):

- **Definition**: CNN is a specialized ANN designed for **spatial data**, especially **image and video** processing.

- **Key Features**:
  - **Convolutional Layers**: Apply filters to extract local features.
  - **Pooling Layers**: Reduce dimensionality (e.g., max pooling).
  - **Fully Connected Layers**: Used for final prediction.
- **Advantages**: Captures spatial hierarchy in data, reduces number of parameters.
- **Use Cases**: Face recognition, object detection, medical imaging.

---

4. Recurrent Neural Networks (RNN):

- **Definition**: RNN is a type of ANN designed for **sequential data** (e.g., time series, language).
- **Key Feature**: Maintains a **memory** of previous inputs through **feedback loops**.
- **Limitation**: Suffers from **vanishing gradient problem**, limiting learning over long sequences.
- **Use Cases**: Speech recognition, text generation, stock price forecasting.

---

5. Long Short-Term Memory (LSTM):

- **Definition**: LSTM is an **advanced type of RNN** specifically designed to overcome the vanishing gradient problem and **remember long-term dependencies**.
- **Architecture**:
  - Contains **memory cells**, and **gates** (input, forget, output) to control the flow of information.
- **Advantages**: Can learn long-term patterns and context better than RNN.
- **Use Cases**: Language translation, chatbots, music generation, sentiment analysis.

2. Explain RNN in details with real life examples.

Recurrent Neural Networks (RNN) –

Definition:

Recurrent Neural Network (RNN) is a type of Artificial Neural Network designed for **sequential data** where the output depends not only on the current input but also on the **previous inputs**. It has a **memory** that captures information about what has been calculated so far.

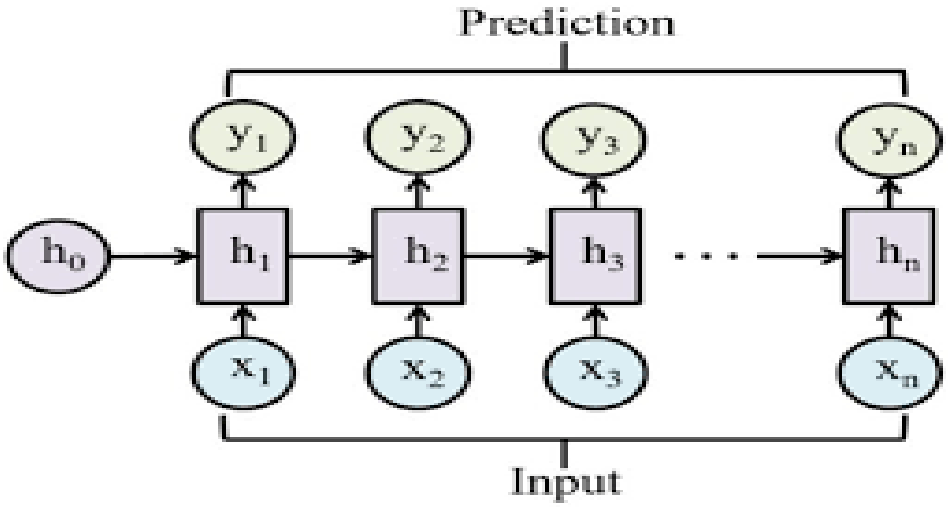
Architecture:

- In RNNs, neurons are connected in a loop.
- Each RNN unit takes **current input (x_t)** and **previous hidden state (h_{t-1})** to produce the **current hidden state (h_t)**.
- This loop mechanism allows the network to **retain information across time steps**.

$$h_t = f(W_{hx} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$$

Where:

- x_t: input at time t
- h_{t-1}: previous hidden state
- f: activation function (like tanh or ReLU)
- W: weight matrices
- b: bias term



Working:

RNNs process sequences one element at a time while maintaining an internal memory of the past. Unlike feedforward networks, RNNs are **stateful**, meaning past data influences current predictions.

Real-Life Examples:

1. Text Prediction (Typing Assistant):

When typing on a smartphone, RNNs can predict the next word based on previous words. Example:
Input: "I want to drink ___"
Prediction: "water", "juice", etc.

2. Speech Recognition:

RNNs are used to convert audio signals into text by analyzing the temporal relationship between sound waves.

3. Music Generation:

Given a sequence of notes, RNNs can generate the next notes in the melody, creating music automatically.

4. Language Translation:

RNNs (especially with LSTM/GRU) can translate one language into another by understanding the sequence and context of words.

5. Sentiment Analysis:

In reviews or tweets, RNNs capture the sequence of words to determine the sentiment (positive/negative).

Advantages:

- Captures **temporal dynamics** and **dependencies** in sequence data.
- Can work with **variable-length input/output**.
- Useful for **time-series forecasting**, such as weather or stock prediction.

Limitations:

- **Vanishing gradient problem**: Difficult to learn long-term dependencies.
- **Training is slow** due to sequential processing.
- **Not efficient** for very long sequences.

Solution to Limitation:

To overcome the above issues, advanced versions like:

- **LSTM (Long Short-Term Memory)** and
- **GRU (Gated Recurrent Unit)**
  are used for better memory retention and performance.

3.Explain Bidirectional RNN in details with Example?

Bidirectional Recurrent Neural Network (Bi-RNN)

---

1. Definition:

A **Bidirectional Recurrent Neural Network (Bi-RNN)** is an advanced type of RNN that improves performance on sequence tasks by processing the input data in **both forward and backward directions**. This allows the network to have access to **past (previous)** and **future (next)** context at every point in the sequence.

---

2. Key Idea:

- Standard RNNs only use **past context** (previous inputs).
- Bi-RNNs use **two RNNs**: one processes the sequence **from start to end** (forward), and another processes it **from end to start** (backward).
- The final output at each time step is a **combination** of both directions, allowing the model to understand full context.

3. Working:

1. Input sequence is fed **forward** to one RNN and **backward** to another.
2. Both RNNs independently compute hidden states.
3. Outputs from both RNNs are **merged** to get the final output at each time step.

This helps the model understand context from **before and after** each word, frame, or data point.

---

3. Architecture:

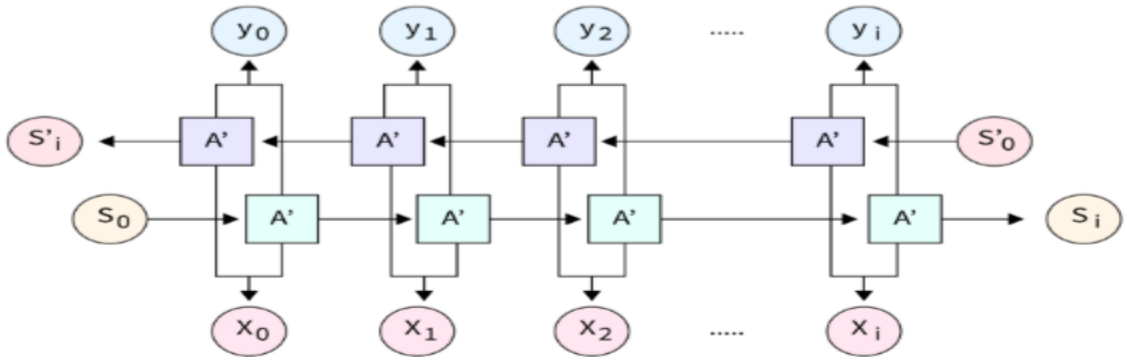
Each input is passed through:

- **Forward RNN:** $\overrightarrow{h_t} = f(W_x^f x_t + W_h^f \overrightarrow{h_{t-1}} + b^f)$
- **Backward RNN:** $\overleftarrow{h_t} = f(W_x^b x_t + W_h^b \overleftarrow{h_{t+1}} + b^b)$

Final output at time `t`:

$$y_t = g(\overrightarrow{h_t}, \overleftarrow{h_t})$$

Where `g` is a function to combine both hidden states (e.g., concatenation or summation).



4. Real-Life Example:

Named Entity Recognition (NER):

- Task: Identify names of people, places, organizations in text.
- Input: "Sachin Tendulkar played for India."
- For the word "played", Bi-RNN considers:
  - **Past**: "Sachin Tendulkar"
  - **Future**: "for India"
- This full context helps in correctly tagging "Sachin Tendulkar" as a **person** and "India" as a **location**.

---

5. Applications:

- **Natural Language Processing (NLP)**:
  - Part-of-speech tagging
  - Machine translation
  - Question answering
- **Speech Recognition**:
  - Understands complete audio context for better transcription.
- **Bioinformatics**:
  - Gene sequence analysis.

---

6. Advantages:

- Understands **both past and future** context.
- Improves accuracy in tasks where **contextual meaning** is important.
- Enhances performance over unidirectional RNNs in many sequence-based applications.

---

7. Limitations:

- Requires **full input sequence** beforehand, so not ideal for real-time streaming tasks.
- Computationally **more expensive** than standard RNNs.
- Cannot be used for causal tasks (e.g., real-time prediction).

4. Explain sequence-to-sequence architecture with example in details?

Sequence-to-Sequence (Seq2Seq) Architecture –

Definition:

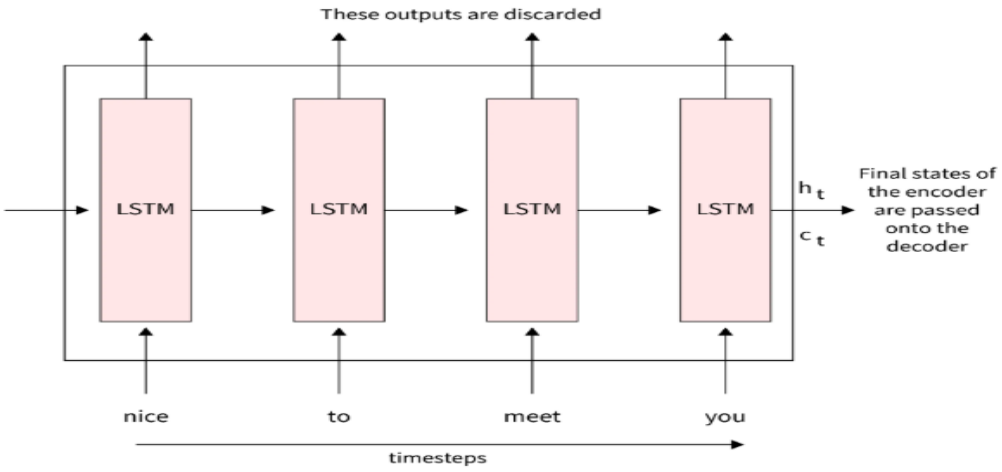
Sequence-to-Sequence (Seq2Seq) is a **neural network architecture** designed to **convert one sequence into another**.
It is widely used in **Natural Language Processing (NLP)** tasks such as **machine translation, chatbot systems, text summarization, and speech recognition**.

Core Components:

Seq2Seq is composed of **two main parts**:

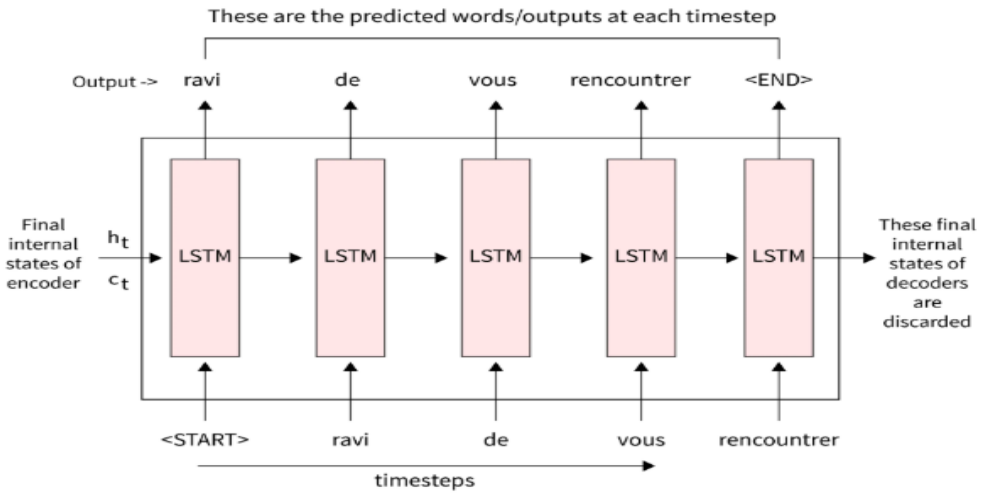
1. Encoder:

- Takes the input sequence.
- Processes each element and compresses it into a **fixed-size context vector** (also called thought vector).
- The final hidden state summarizes all input information.



2. Decoder:

- Takes the context vector from the encoder.
- Generates the output sequence **one step at a time**.
- Each output is fed as input to the next decoding step.



Working Steps:

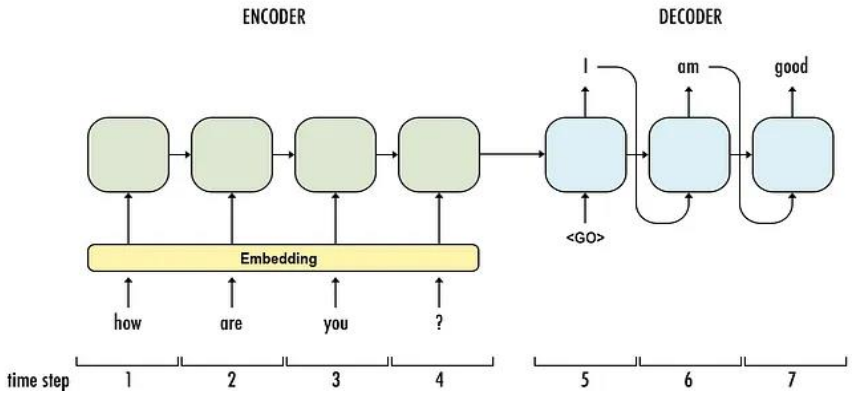
1. **Encoder** reads the input sequence and updates its hidden state at each step.
2. After reading the entire input, the **final hidden state** becomes the **context vector**.
3. **Decoder** takes this context vector and starts producing the output tokens **step-by-step**, using its own previous output and hidden state.

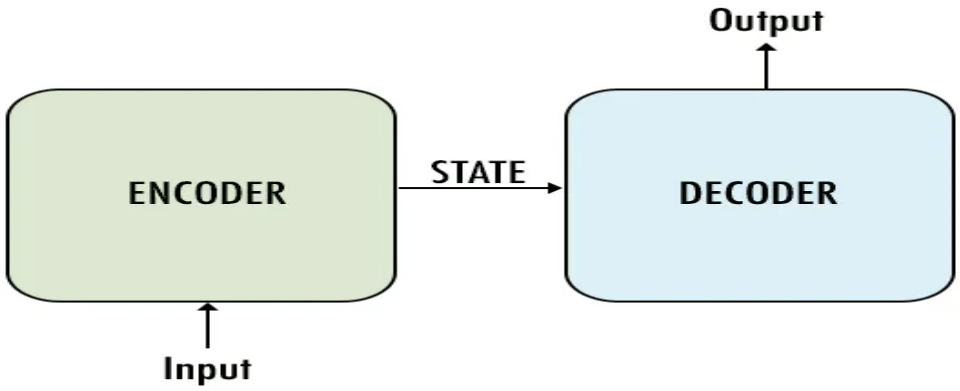
Example:

Input (English): "I love programming"

Output (French): "J'aime la programmation"

- **Encoder** processes: ["I", "love", "programming"]
- **Context Vector**: A hidden state summarizing the sentence.
- **Decoder** generates: ["J'", "aime", "la", "programmation"]





Real-Life Applications:

1. **Machine Translation**: Translating sentences between languages (Google Translate).
2. **Text Summarization**: Generating a short summary from a long article.
3. **Chatbots**: Generating human-like responses in conversation.
4. **Speech Recognition**: Converting audio (sequence of signals) into text.
5. **Question Answering Systems**: Generating appropriate responses based on input queries.

Enhancements:

- **Attention Mechanism**: Solves the issue of long input sequences by allowing the decoder to focus on specific parts of the input at each step.
- **Bidirectional Encoder**: Improves context representation.

- **LSTM/GRU Units**: Used to handle long-term dependencies and avoid vanishing gradients.

Advantages:

- Can handle input and output sequences of **different lengths**.
- Effective for many NLP and sequence tasks.
- Flexible and powerful for complex sequence mapping problems.

Limitations:

- Simple Seq2Seq may struggle with **very long sequences**.
- Entire input must be processed before output begins (not ideal for real-time).

5. Explain Deep recurrent networks with examples?

Deep Recurrent Neural Networks (DRNN)

Definition:

A **Deep Recurrent Neural Network (DRNN)** is an advanced version of a standard Recurrent Neural Network (RNN) that has **multiple hidden recurrent layers stacked on top of each other**.

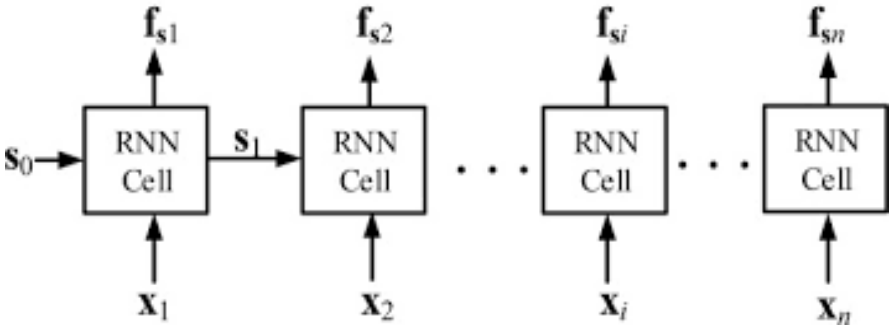
- Unlike simple RNNs (which have only **one hidden layer**), DRNNs can learn more **complex patterns** and **hierarchical representations**.
- Each recurrent layer passes information to the **next layer in depth**, as well as to the **next time step**.

Why Use Deep RNNs?

- Captures **low-level features** in lower layers and **high-level abstract features** in upper layers.
- Better suited for **long sequences**, **complex dependencies**, and **structured outputs**.
- Useful when a **single-layer RNN** is not enough for good performance.

Working Mechanism:

1. The input sequence is passed to the **first RNN layer**.
2. The output of each time step is passed to the **next recurrent layer** at the same time step.
3. This is repeated for **multiple layers**, allowing the model to understand **deep sequential features**.



Real-Life Examples:

1. Speech Recognition:

- Lower layers detect **phonemes** or sound patterns.
- Middle layers capture **syllables** or word-level features.
- Higher layers understand **phrases** or context.

2. Text-to-Speech (TTS):

- DRNNs model both the **text sequence** and the **temporal nature** of human speech for more natural synthesis.

3. Music Generation:

- Learns complex structure in musical sequences — rhythm, harmony, and melody patterns at different layers.

4. Video Captioning:

- Processes temporal frames and generates descriptive sentences using deeper temporal understanding.

5. Financial Time Series Prediction:

- Multiple layers capture both **short-term fluctuations** and **long-term trends** in stock or currency prices.

Advantages:

- **Better learning capacity** for complex and long sequences.
- Improves generalization by capturing **multiple levels of abstraction**.
- Useful in tasks involving **language**, **audio**, or **video sequences**.

Limitations:

- **Training is slower** due to multiple layers and longer backpropagation paths.
- **Risk of overfitting** on small datasets.
- Still suffers (though less than shallow RNNs) from **vanishing gradient problems** in long sequences.

Solution: Use **LSTM** or **GRU** units within each recurrent layer to handle long dependencies.

6. Difference Between BRNN and Recurrent Neural Network?

Difference Between RNN and BRNN

No.	Feature	RNN (Recurrent Neural Network)	BRNN (Bidirectional RNN)
1.	**Data Flow Direction**	Processes data in **one direction** (past → future)	Processes data in **two directions** (past ↔ future)

No.	Feature	RNN (Recurrent Neural Network)	BRNN (Bidirectional RNN)
2.	**Context Captured**	Captures only **past context**	Captures **past and future context**
3.	**Architecture**	Single RNN layer with forward connections	Two RNNs: one forward, one backward
4.	**Understanding of Sequences**	**Limited context** understanding	**Deeper context** understanding
5.	**Suitability for Real-Time**	Suitable for **real-time** processing	Not suitable for real-time (requires full input)
6.	**Computation & Memory**	**Less computationally** intensive	Requires **more computation and memory**
7.	**Application Area**	Time-series forecasting, sensor data	NLP tasks: NER, POS tagging, machine translation
8.	**Learning Long Dependencies**	May struggle with long dependencies	**Better at learning** long-range dependencies
9.	**Training Complexity**	**Simpler** to implement and train	**More complex** due to dual networks
10.	**Information Loss Risk**	May lose info at end of long sequences	Recovers info with **backward RNN**
11.	**Example Use Case**	Stock prediction, weather forecasting	Sentiment analysis, language translation
12.	**Variants**	Often uses **LSTM** or **GRU** for improvements	Uses **BiLSTM** or **BiGRU** for richer modeling

UNIT-5

1. Explain GAN in detail with example?

Generative Adversarial Network (GAN) –

Definition:

A **Generative Adversarial Network (GAN)** is a type of **deep learning architecture** where **two neural networks** — a **Generator** and a **Discriminator** — compete with each other in a **game-theoretic** scenario.

Introduced by **Ian Goodfellow in 2014**, GANs are widely used for generating **realistic synthetic data** such as images, music, text, and videos.

Architecture Overview:

GAN consists of two core components:

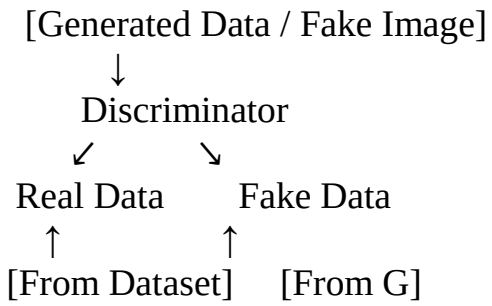
Component	Role
Generator (G)	Learns to generate **fake data** that resembles real data
Discriminator (D)	Learns to **distinguish between real and fake** data

These two networks are trained **simultaneously** in a **minimax game**:

- **Generator tries to fool the Discriminator**.
- **Discriminator tries to catch the Generator** producing fakes.

Diagram (Conceptual):

[Random Noise z]
↓
Generator (G)
↓



Training Process:

1. **Input random noise vector (z)** into the **Generator (G)**.
2. Generator produces **fake data** (e.g., an image).
3. **Discriminator (D)** receives both **real data** and **fake data** and tries to classify them correctly.
4. Based on D's feedback:
   - G improves to produce **more realistic data**.
   - D improves to become **better at spotting fakes**.
5. This continues until **Generator produces data indistinguishable from real data**.

✅ **Mathematical Objective:**

The minimax loss function:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Example: Image Generation

- **Real dataset**: Photos of human faces.
- **Generator**: Learns to generate new **realistic-looking faces** from noise.
- **Discriminator**: Learns to tell whether a face is **real** (from dataset) or **fake** (from Generator).

Eventually, the Generator becomes so good that **humans or even the Discriminator can't distinguish** between real and fake faces.

Real-Life Applications:

1. **Image Generation** – Generate realistic human faces (thispersondoesnotexist.com).
2. **Game Character Design** – Create custom game characters, textures.
3. **Deepfake Videos** – Superimposing faces in videos.
4. **Image-to-Image Translation** – Turn sketches into colored images (Pix2Pix).
5. **Drug Discovery** – Generate novel molecular structures.
6. **Photo Restoration** – Remove noise or restore damaged images.

Types of GANs:

Type	Use Case
DCGAN (Deep Convolutional GAN)	Used for image generation
Conditional GAN (cGAN)	Generate images based on labels
CycleGAN	Image-to-image translation without paired data
StyleGAN	High-quality face generation
Pix2Pix	Convert images from one domain to another

Advantages:

- Generates **high-quality synthetic data**.
- Learns **unsupervised** — no label needed.
- Flexible for **creative and scientific tasks**.

Limitations:

- **Difficult to train** (training instability).
- **Mode collapse** – Generator may produce limited varieties.
- Requires **careful tuning of loss functions** and architectures.

2. Explain the working of Generative Adversarial Networks (GANs) with the help of TensorFlow code. Also, write a program to generate handwritten digits using GAN trained on the MNIST dataset.

Generative Adversarial Networks (GANs):

GANs consist of two neural networks:

- **Generator (G):** Takes random noise and generates fake data/images.
- **Discriminator (D):** Classifies data as real (from dataset) or fake (from Generator).

They are trained together:

- The generator tries to "fool" the discriminator by generating realistic data.
- The discriminator tries to detect fake data.

This is a **min-max game**:

- Discriminator maximizes the ability to classify real vs fake.
- Generator minimizes the chance of being caught (tries to generate realistic outputs).

Real-Life Application:

Image generation: GANs can be trained to generate handwritten digits by learning from the MNIST dataset.

Features of GAN:

Feature	Description
Type	Unsupervised Learning
Components	Generator & Discriminator

Feature	Description
Training Strategy	Adversarial MinMax Game
Dataset	MNIST (28x28 grayscale images)
Output	Fake digits that resemble real handwritten digits
Loss Function	Binary Crossentropy
Generator Activation	LeakyReLU, Tanh
Discriminator Act.	LeakyReLU, Sigmoid
Latent Space (z)	100-Dimensional random noise

Program: Generating Handwritten Digits using GAN

 # Import required libraries

```
import numpy as np

import matplotlib.pyplot as plt

from keras.datasets import mnist

from keras.models import Sequential, Model

from keras.layers import Dense, LeakyReLU, BatchNormalization, Reshape, Flatten, Input

from keras.optimizers import SGD

from keras import backend as K
```

 # Load and preprocess MNIST data

```
(x_train, _), (_, _) = mnist.load_data()
```

```
x_train = (x_train / 127.5) - 1.0  # Normalize to [-1, 1]
x_train = np.expand_dims(x_train, axis=3)

img_shape = (28, 28, 1)
z_dim = 100

# Build Generator
def build_generator():
    model = Sequential()
    model.add(Dense(256, input_dim=z_dim))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(0.8))
    model.add(Dense(1024))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(0.8))
    model.add(Dense(np.prod(img_shape), activation='tanh'))
    model.add(Reshape(img_shape))
    noise = Input(shape=(z_dim,))
    img = model(noise)
    return Model(noise, img)
```

```
# Build Discriminator
def build_discriminator():
    model = Sequential()
    model.add(Flatten(input_shape=img_shape))
    model.add(Dense(512))
    model.add(LeakyReLU(0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(0.2))
    model.add(Dense(1, activation='sigmoid'))
    img = Input(shape=img_shape)
    validity = model(img)
    return Model(img, validity)

# Compile models
optimizer = SGD(0.0002, 0.5)
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer=optimizer,
metrics=['accuracy'])


generator = build_generator()
z = Input(shape=(z_dim,))
img = generator(z)
```

```
discriminator.trainable = False
validity = discriminator(img)
combined = Model(z, validity)
combined.compile(loss='binary_crossentropy', optimizer=optimizer)

# Training
def train(epochs, batch_size=32, sample_interval=1000):
    valid = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

    for epoch in range(epochs):
        # Train Discriminator
        idx = np.random.randint(0, x_train.shape[0], batch_size)
        real_imgs = x_train[idx]
        noise = np.random.normal(0, 1, (batch_size, z_dim))
        gen_imgs = generator.predict(noise)
        d_loss_real = discriminator.train_on_batch(real_imgs, valid)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train Generator
        noise = np.random.normal(0, 1, (batch_size, z_dim))
        g_loss = combined.train_on_batch(noise, valid)
```

```
        # Print progress
        if epoch % 100 == 0:
            print(f"{epoch} [D loss: {d_loss[0]}, acc.: {100*d_loss[1]}%] [G loss: {g_loss}]")

        # Save samples
        if epoch % sample_interval == 0:
            sample_images(epoch)

# Generate images
def sample_images(epoch):
    r, c = 5, 5
    noise = np.random.normal(0, 1, (r * c, z_dim))
    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5  # Rescale to [0, 1]

    fig, axs = plt.subplots(r, c)
    cnt = 0
    for i in range(r):
        for j in range(c):
            axs[i,j].imshow(gen_imgs[cnt, :, :, 0], cmap='gray')
            axs[i,j].axis('off')
```

```
            cnt += 1
    plt.savefig(f"gan_digit_{epoch}.png")
    plt.close()


# Run training
train(epochs=10000, batch_size=32, sample_interval=2000)
```

3. Explain in details Restricted Boltzmann Machines with examples?

Restricted Boltzmann Machines (RBMs) —

1. What is a Restricted Boltzmann Machine?

An RBM is a type of **stochastic neural network** that can learn a probability distribution over its set of inputs. It is an **unsupervised learning model** used mainly for dimensionality reduction, feature learning, and as a building block for deep learning architectures like Deep Belief Networks.

---

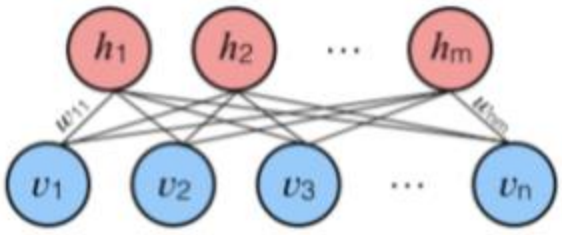
2. Structure of RBM

RBM is a **two-layer network** consisting of:

- **Visible layer (v):** Represents the observed data (input).
- **Hidden layer (h):** Learns to capture features or representations of the input data.

Key restriction: There are **no connections between nodes within the same layer** (i.e., no visible-visible or hidden-hidden connections), only **visible-to-hidden connections**.

This restriction simplifies computations and differentiates RBMs from general Boltzmann Machines.



---

3. How RBM Works

- Each visible unit is connected to every hidden unit via a weight wijw_{ij}wij.
- The RBM learns to represent the data distribution by adjusting weights so that the hidden layer learns useful features.
- The units (neurons) are typically **binary stochastic units** (0 or 1).

---

4. Mathematical Formulation

RBM defines a **joint probability distribution** over visible and hidden units based on an **energy function**:

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{ij} h_j$$

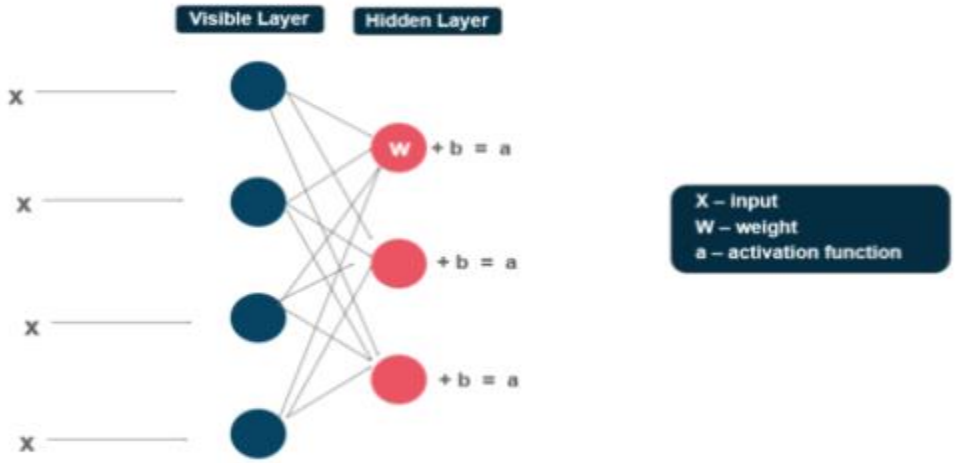
where:

- v_i: state of visible unit i
- h_j: state of hidden unit j
- a_i, b_j: biases of visible and hidden units
- w_{ij}: weight between visible unit i and hidden unit j

The **probability** of a visible vector v is given by marginalizing over all hidden vectors:

$$P(v) = \frac{1}{Z} \sum_h e^{-E(v,h)}$$

where Z is the **partition function** (normalizing constant).



7. Applications of RBMs

- **Dimensionality reduction:** Learn compact representations.
- **Collaborative filtering:** Recommendation systems (e.g., Netflix prize).
- **Feature extraction:** Pretraining layers for deep networks.
- **Classification:** Using learned features in supervised models.

---

8. Simple Illustrative Example

Imagine a dataset of binary images of simple shapes: squares and circles.

- The RBM learns hidden features such as “edges,” “corners,” or “curves” by adjusting weights.
- Given a partially occluded square, the hidden features help reconstruct the missing parts.

3. Explain Deep Belief Network with example?

Deep Belief Network (DBN) —

1. What is a Deep Belief Network?

A **Deep Belief Network (DBN)** is a type of **deep generative probabilistic model** composed of multiple layers of **stochastic latent variables**. It's essentially a stack of **Restricted Boltzmann Machines (RBMs)** or sometimes combines RBMs and other layers.

- DBNs can **learn to represent complex data distributions** by learning hierarchical features layer by layer.
- They were one of the early deep learning models enabling training of deep architectures before modern backpropagation methods took over.

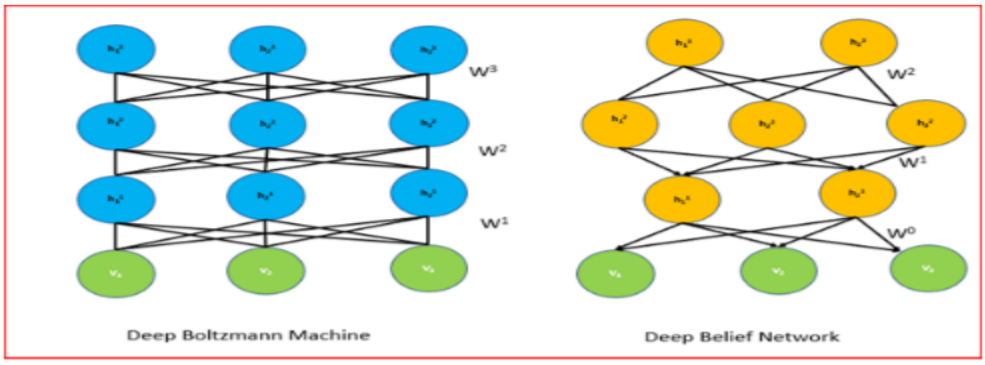
---

2. Structure of a DBN

- Composed of multiple layers of hidden units.
- The **top two layers** form an **undirected bipartite graph (an RBM)**.
- The lower layers form a **directed, generative model** that connects hidden units of one layer to visible units of the next.

Visually:

Input (visible layer) -> Hidden Layer 1 -> Hidden Layer 2 -> ... -> Hidden Layer N



3. How DBN Works

DBNs learn **layer-wise**, using **greedy unsupervised training**:

- Train the first RBM on the raw input data.

- Freeze the first RBM's weights.
- Use the activations of the first RBM's hidden layer as input (visible layer) for the second RBM.
- Train the second RBM on this transformed data.
- Repeat for subsequent layers.

This layer-wise pretraining initializes the deep network's weights well, which can then be **fine-tuned with supervised learning** (e.g., backpropagation) if labels are available.

---

4. Mathematical Intuition

- Each RBM models the joint distribution P(v,h) of its visible and hidden units.
- Stacking RBMs creates a **deep hierarchical generative model** where the top-level RBM captures complex correlations.
- The lower layers can be interpreted as directed sigmoid belief networks.

---

5. Example: DBN for Digit Recognition

Suppose you want to classify handwritten digits from the MNIST dataset.

Step 1: Pretraining

- Train the first RBM on raw pixel values (visible units).
- Train the second RBM on the hidden activations of the first RBM.
- Train a third RBM on the hidden activations of the second RBM.

Each RBM learns features at increasing abstraction:

- First RBM: edges and strokes
- Second RBM: parts of digits (loops, curves)
- Third RBM: digit prototypes or styles

Step 2: Fine-tuning

- Stack all layers into a deep neural network.

- Add a classification layer on top.
- Use labeled data to fine-tune all weights with backpropagation.

---

6. Benefits of DBNs

- **Efficient layer-wise training** avoids problems of training deep networks directly.
- Learns hierarchical representations.
- Can be used for **unsupervised pretraining**, improving performance when labeled data is scarce.
- Can be used as **generative models** (generate samples similar to training data).

4. Difference between Autoencoders & RBMs?

Feature	Autoencoder	Restricted Boltzmann Machine (RBM)
1. **Model Type**	Deterministic neural network	Stochastic generative probabilistic graphical model
2. **Architecture**	Encoder + Decoder (usually feedforward neural nets)	Bipartite undirected graph with visible and hidden layers
3. **Training Objective**	Minimize reconstruction error (e.g., MSE)	Maximize likelihood of data via energy-based model
4. **Learning Method**	Backpropagation with gradient descent	Contrastive Divergence or Gibbs Sampling
5. **Data Representation**	Learns a compressed latent representation	Learns a probabilistic distribution over inputs
6. **Stochasticity**	Generally deterministic (though stochastic variants exist)	Stochastic binary units in hidden and visible layers
7. **Output**	Reconstructed input (deterministic)	Probability distribution over visible units
8. **Generative Ability**	Can be generative if used with Variational Autoencoders	Naturally generative model
9. **Type of Layers**	Typically continuous valued neurons	Binary or probabilistic neurons

Feature	Autoencoder	Restricted Boltzmann Machine (RBM)
10. **Suitability**	Good for dimensionality reduction, denoising, feature learning	Good for modeling distributions, unsupervised pretraining
11. **Training Complexity**	Usually faster to train and scale	Training is more complex due to sampling and energy function
12. **Applications**	Image compression, anomaly detection, denoising	Pretraining deep networks, collaborative filtering, feature extraction

UNIT-6

1. Explain Dynamic Programming with the help of example (Robot in a grid world)?

Dynamic Programming in Reinforcement Learning –

---

1. Definition:

Dynamic Programming (DP) in reinforcement learning is a class of methods used to compute **optimal policies** by breaking a problem into **simpler overlapping sub-problems** and solving them **recursively**. It assumes **complete knowledge** of the environment in the form of a **Markov Decision Process (MDP)**.

---

2. Requirements:

DP methods are applicable **only** when:

- The environment's **transition probabilities** and **reward function** are known.
- The problem is formulated as an **MDP** with well-defined:
  - States (S)
  - Actions (A)

- Rewards (R)
- Transition probabilities (P)

---

3. Key Algorithms in DP:

Policy Evaluation:

- Computes the **value function** V(s) for a **given policy**.
- It estimates the **expected cumulative reward** from each state.
- Done **iteratively** until value estimates **converge**.

Policy Iteration:

- Alternates between two steps:
  1. **Policy Evaluation** – Evaluate current policy.
  2. **Policy Improvement** – Update the policy using current value function.
- Continues until the policy becomes **stable and optimal**.

Example: Robot in a Grid World (3x3)

Scenario:

- A robot is in a 3x3 grid (total of 9 states).
- The goal is to reach the **bottom-right corner**.
- **Reward:**
  - +1 for reaching the goal
  - -1 for every step taken (to encourage efficiency)
- The robot can move **Up, Down, Left, Right**.
- It may **slip** into adjacent cells due to uncertainty.

□ How DP Works in This Example:

Step	Description
1. **Initialize Values**	Assign initial random values to all 9 states, usually 0.
2. **Policy Evaluation**	Use the **Bellman Expectation Equation** to update each state's value based on reward + expected value of next state (weighted by transition probability). Repeat until convergence.
3. **Policy Improvement**	For each state, choose the action that leads to the highest-valued neighbor. Update the policy accordingly.
4. **Repeat**	Iterate steps 2 and 3 until the value function and policy do not change.

 Result:

- The robot learns the **shortest and safest path** to the goal.
- It avoids inefficient routes that take too many steps or involve risk.
- The final policy guides the robot optimally from any cell.

---

4. Advantages of DP in RL:

- Guarantees **optimal policy** if MDP is known.
- **Systematic** and **stable** approach.
- Forms the **foundation** for other RL algorithms like Q-learning.

---

5. Limitations:

- Requires **complete knowledge** of the environment.
- **Computationally expensive** in large state spaces.

---

6. Why DP Matters in DL

- **Reduces redundant computation** in sequence models.
- Helps in building efficient **temporal models** like RNNs and LSTMs.
- Plays a vital role in **policy optimization** in intelligent agents.

2.Difference between Q-Learning and Deep Q-Network (DQN).

Feature	Q-Learning	Deep Q-Network (DQN)
1. Model Type	Tabular Reinforcement Learning algorithm	Deep Reinforcement Learning algorithm
2. Q-Value Representation	Q-values stored in a table (Q-table)	Q-values approximated using a neural network
3. Scalability	Works only for small, discrete state-action spaces	Scales well to large or continuous state spaces using deep learning
4. Function Approximation	No function approximation; uses explicit tables	Uses deep neural networks to approximate Q-function
5. Input Format	Requires discrete state and action representation	Accepts raw or high-dimensional inputs (e.g., images, sensor data)
6. Memory Usage	Memory inefficient for large state-action spaces	Memory efficient with function approximation
7. Generalization	Poor generalization—each state-action pair is learned independently	Good generalization due to shared parameters in neural networks
8. Convergence	Converges with enough visits to all state-action pairs	Convergence is less guaranteed; depends on architecture and hyperparameters
9. Algorithm Complexity	Simple and easy to implement	More complex due to deep learning integration
10. Learning Method	Updates Q-table using Bellman equation	Trains neural network using experience replay and target network
11. Use Case	Educational or low-dimensional environments (e.g., Grid World)	Real-world applications like Atari games, robotics, self-driving cars

Feature	Q-Learning	Deep Q-Network (DQN)
12. Exploration Strategy	Often uses ε-greedy exploration	Uses ε-greedy but can also incorporate advanced techniques (e.g., Double DQN)

3. With the help of Robot in grid world example, elaborate Markov Decision Process.

What is an MDP?

A **Markov Decision Process (MDP)** is a **mathematical framework** used to model **decision-making problems** in environments where outcomes are:

- **Partly random (stochastic)**
- **Partly under the control** of a decision-maker (agent)

MDPs are **widely used in Reinforcement Learning (RL)** to define the environment and interaction mechanism for learning optimal behavior.

Components of MDP

An MDP is typically represented by a 5-tuple:

$$\mathbf{MDP=(S,A,P,R,\pi)}$$

Component	Symbol	Description
States	S	Set of all possible **situations** the agent can be in.
Actions	A	Set of **choices** or actions the agent can perform at each state.
Transition Probabilities	(P(s'	s,a))
Rewards	R(s,a,s′)	Immediate **reward** received for moving from sss to s′ via action aaa.

Component	Symbol	Description
Policy	$\pi(s)$	Strategy or rule that maps each state s to an action a, aiming to **maximize cumulative reward**.

Example: Robot in a Grid World

 Scenario

A robot is placed in a **4×4 grid** (16 cells). Its objective is to **reach the goal cell** (bottom-right corner) while **avoiding inefficient paths**.

 MDP Elements in the Grid World:

Element	Description
States (S)	Each of the 16 cells is a unique state. So, S={s1,s2,...,s16}
Actions (A)	At each state, the robot can take one of 4 actions: UP, DOWN, LEFT, RIGHT
Transition Probability (P)	The robot moves in the intended direction with **80% probability**, and due to **slippage**, ends up in an adjacent cell with **10%** probability each.
Rewards (R)	Reaching the goal state yields a **positive reward (+10)**, while every step incurs a **small penalty (-1)** to encourage quicker solutions.
Policy (π)	A mapping from state to action. Example: $\pi(s_5)$ = RIGHT (if going right is optimal from s_5)

□ Illustrative Transition Example

If the robot is at state s6 and decides to move **RIGHT**:

- 80% chance → it reaches s7
- 10% chance → it slips and ends up at s2
- 10% chance → it slips and ends up at s10

So, transition probabilities are:

$$P(s_7|s_6, \text{RIGHT}) = 0.8, \quad P(s_2|s_6, \text{RIGHT}) = 0.1, \quad P(s_{10}|s_6, \text{RIGHT}) = 0.1$$

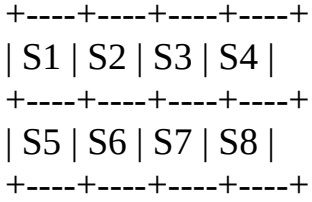
Goal-Oriented Learning

- The robot learns the **optimal policy π^*** through exploration.
- Using algorithms like **Value Iteration** or **Q-Learning**, it identifies the action that **maximizes the expected reward** at each state.
- It avoids unnecessary steps and maximizes rewards by learning from past transitions and outcomes.

Key Benefits of MDP in Grid World

- Helps in **decision-making under uncertainty**
- Models **probabilistic outcomes** of actions
- Encourages the agent to **balance exploration and exploitation**
- Enables the agent to **learn optimal behavior** over time

Grid Diagram



| S9 |S10 |S11 |S12 |
+----+----+----+----+
|S13 |S14 |S15 | G |
+----+----+----+----+

4. Explain Deep Reinforcement Learning (DRL) in detail with its components, working steps, application in Tic-Tac-Toe, and real-world use cases.

1. What is Deep Reinforcement Learning (DRL)?

Deep Reinforcement Learning (DRL) is an advanced AI technique that combines:

- **Reinforcement Learning (RL)**: Learning through interaction with an environment to maximize cumulative rewards.
- **Deep Learning (DL)**: Using neural networks to extract complex patterns from high-dimensional inputs (like images, board states, etc.).

This combination allows agents to learn complex behaviors and make decisions directly from raw, unstructured data.

---

2. Core Components of DRL

Component	Description
Agent	The learner/decision-maker interacting with the environment
Environment	The external system providing observations and rewards
State	The current situation of the environment
Action	A decision or move the agent can take
Reward	Feedback received from the environment for each action
Policy (π)	A strategy that maps states to actions
Value Function	Predicts expected cumulative reward from a state under a policy
Model	Simulates or predicts environment dynamics
Exploration vs	Balancing discovery of new actions vs leveraging

Component	Description
Exploitation	known actions
Learning Algorithm	Updates policy/value functions based on experiences
Deep Neural Networks	Function approximators for value/policy in high-dimensional spaces
Experience Replay	Stores past experiences to train the agent efficiently

---

3. How DRL Works (Step-by-Step)

1. **Initialization**: Set up agent, environment, and neural network.
2. **Interaction**: Agent takes an action in the environment → observes new state & reward.
3. **Storage**: Stores experiences as (state, action, reward, next state).
4. **Training**: Uses neural networks to update the value or policy function.
5. **Policy Update**: Agent's behavior improves by adjusting based on feedback.
6. **Exploration-Exploitation**: Explores new strategies while exploiting known good ones.
7. **Convergence**: Learns a near-optimal or optimal policy.
8. **Generalization**: Applies knowledge to unseen scenarios.
9. **Evaluation**: Performance is tested in unknown or simulated environments.
10. **Deployment**: Agent is used in real-world applications.

---

4. DRL Example: Tic-Tac-Toe Game

Goal: Train an agent to play Tic-Tac-Toe optimally.

- **Agent**: Learns to play as 'X' or 'O'.
- **Environment**: 3x3 grid board.
- **States**: All possible board configurations.
- **Actions**: Valid empty cell positions.
- **Reward**: +1 for win, -1 for loss, 0 for draw.
- **Policy**: Maps board states to best next move.
- **Learning**: Through repeated games using Q-learning or deep Q-networks (DQN).
- **Outcome**: Over time, the agent learns optimal strategies to win or draw every game.

5. Applications of Deep Reinforcement Learning

Domain	Real-World Applications
Gaming	Mastering Go, Chess, Dota 2; smart AI opponents
Robotics	Object manipulation, drone navigation, autonomous driving
Finance	Portfolio optimization, algorithmic trading, risk assessment
Healthcare	Drug discovery, treatment planning, robotic surgery
Energy	Smart grid control, energy usage optimization
NLP	Dialogue systems, translation, summarization
Recommendation	Personalized content suggestions in e-commerce, streaming
Manufacturing	Process automation, predictive maintenance
Agriculture	Smart irrigation, crop monitoring, pest control
Education	Intelligent tutoring systems, adaptive learning platforms