||JAI SRI GURUDEV||

Adichunchanagiri Shikshana Trust ®

ADICHUNCHANAGIRI INSTITUTE OF TECHNOLOGY

CHIKKAMAGALURU

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

LAB MANUAL

SEVENTH SEMESTER

DEEP LEARNING AND REINFORCEMENT LEARNING

(BAI701)

(Academic Year 2025-2026)

[As per Choice Based Credit System (CBCS) scheme]

When all castes and creeds live together with a sense of equality and brotherhood, they will be truly happy and country strong. Collective organization of society has its rewards.

--Sri Sri Sri Dr. Balagangadharanatha Mahaswamiji

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belagavi, Karnataka –590018

SEVENTH SEMESTER

Deep learning and reinforcement learning

(BAI701)

(Academic Year 2024-2025)

[As per Choice Based Credit System (CBCS) scheme]

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Adichunchanagiri institute of technology

Beekanahalli Rural, Chikkamagaluru, Karnataka - 577101

(2025-2026)

Prepared By,

Dr. Sunitha M R, Professor & Head,
Ms. Sathyabhama R, Assistant Professor
Mr. Purushothama B A Assistant Professor
Mrs. Sana Ara, Assistant Professor
Mrs. Bhavya A, Assistant Professor
Ms. Sonia Fathima, Assistant Professor
Mrs. Sindhu B N ,Assistant Professor
Mr. Suhas K P, Assistant Instructor
Mrs. Vedashri S G, Assistant Instructor

COLLEGE VISION AND MISSION

VISION

To develop Adichunchanagiri Institute of Technology as a center of excellence and to strive for continuous improvement of technical education and human resource advancement.

MISSION

To achieve Excellence in Education, Entrepreneurship, and Innovation by producing Engineers with high Ethical Standards, Integrity, and Credibility.

DEPARMENT VISION AND MISSION

The department was started in the year 2021-22 with an intake of 60. Department comprises of highly qualified and experienced teaching staff and research scholars.

Department is performing well by securing VTU ranks in academics. With support from the institution, the department is constantly providing career guidance for the students in order to achieve good results and it has excellent placement records.

Department regularly conducts technical talks, seminars, short-term courses, hands-on training, and workshops for students and faculty to bridge the technical gap between educational institutions and industries.

VISION

To be recognized as a centre of excellence in information technology and allied areas with quality learning and research environment.

MISSION

- Provide intellectual & professional leadership in ethical and social areas pertaining to information in contemporary society.
- Advancing the state of knowledge of information studies through research and development.

COURSE OBJECTIVES:

- To introduce fundamental and advanced concepts of deep learning by exploring neural network architectures for text, image, and time-series data.
- To provide hands-on experience in designing, implementing, and evaluating deep learning models, including CNNs, autoencoders, and recurrent architectures for various real-world tasks.

- To develop the ability to preprocess, represent, and analyze data effectively using embeddings, feature extraction, and dimensionality reduction techniques.

- To enable students to apply transfer learning and pre-trained models for efficient training and accurate classification on complex datasets.

- To foster problem-solving and research-oriented thinking through the implementation of reinforcement learning (grid world) and advanced deep learning applications.

COURSE OUTCOME:

- To demonstrate the implementation of deep learning techniques.
- To examine various deep learning techniques for solving the real-world problems.
- To Design and implement research-oriented scenario using deep learning techniques in a team.

CIE for the practical component of the IC:

- 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.

- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.

- The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.

- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks

- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

Do's

- Do wear ID card and follow dress code.
- Do log off the computers when you finish.
- Do ask the staff for assistance if you need help.

- Do keep your voice low when speaking to others in the LAB.
- Do ask for assistance in downloading any software.
- Do make suggestions as to how we can improve the LAB.
- In case of any hardware related problem, ask LAB in charge for solution.
- If you are the last one leaving the LAB, make sure that the staffs in charge of the LAB are informed to close the LAB.
- Be on time to LAB sessions.
- Do keep the LAB as clean as possible

Dont's

- Don't do anything that can make the LAB dirty (like eating, throwing waste papers etc).
- Do not carry any external devices without permission.
- Don't move the chairs of the LAB.
- Don't interchange any part of one computer with another.
- Don't leave the computers of the LAB turned on while leaving the LAB.
- Do not install or download any software or modify or delete any system files on any lab computers.
- Do not damage, remove, or disconnect any labels, parts, cables, or equipment.
- Don't attempt to bypass the computer security system.
- Do not read or modify other user's file.
- If you leave the lab, do not leave your personal belongings unattended. We are not responsible for any theft.
- Do not use mobile phone inside the lab.

SPECIAL THANKS TO

MR. MALLESH V S &

MR. SHIVARAJ V S

ABEYAAANTRIX

TABLE OF CONTENTS

COURSE OVERVIEW

Course Code: BAI701 | **Semester:** 7 | **Credits:** 4

Teaching Hours: 3:0:2:0 (Theory: 3, Practical: 2)

This lab manual provides comprehensive hands-on experience with deep learning and reinforcement learning technologies. Students will implement various neural network architectures, understand their theoretical foundations, and apply them to real-world problems.

A. Course Structure

- **Total Hours:** 40 Theory + 10 Lab Sessions
- **Examination:** CIE (50 marks) + SEE (50 marks) = 100 marks

B. Technologies Used

- **Primary:** Python, TensorFlow/Keras, PyTorch
- **Supporting:** NumPy, Pandas, Matplotlib, scikit-learn
- **Environment:** Jupyter Notebook, Google Colab (recommended)
- **RL Framework:** OpenAI Gym, Stable Baselines3

C. Learning Objectives

Upon completion of this lab, students will be able to:

1. **Design and implement** various neural network architectures from scratch
2. **Apply deep learning** techniques to computer vision and natural language processing tasks
3. **Understand and utilize** convolutional neural networks for image classification
4. **Implement autoencoders** for dimensionality reduction and feature learning
5. **Develop RNN/LSTM models** for sequential data processing
6. **Apply transfer learning** techniques using pre-trained models
7. **Understand reinforcement learning** fundamentals and implement basic RL algorithms
8. **Evaluate model performance** using appropriate metrics and visualization techniques

D. Lab Setup and Prerequisites

Software Requirements

Python 3.8+ with essential packages

- pip install tensorflow>=2.8.0
- pip install torch torchvision torchaudio
- pip install numpy pandas matplotlib seaborn
- pip install scikit-learn jupyter notebook
- pip install gym stable-baselines3
- pip install nltk gensim wordcloud

Hardware Recommendations

- **Minimum:** 8GB RAM, CPU with 4+ cores
- **Recommended:** 16GB RAM, GPU (GTX 1060 or better)
- **Cloud Alternative:** Google Colab Pro (recommended for students without GPU)

E. Dataset Preparation

All datasets will be downloaded programmatically during experiments. Ensure stable internet connection for first-time setup.

Assessment Guidelines

A. Continuous Internal Evaluation (CIE) - 50 Marks

1. **Theory Component (25 marks):**
    - Internal Test 1: 15 marks
    - Assignments/Quiz: 10 marks
2. **Practical Component (25 marks):**
    - Lab Records: 15 marks (8 experiments × ~2 marks each)
    - Lab Test: 10 marks

B. Semester End Examination (SEE) - 50 Marks

- 10 questions (2 per module)
- Students answer 5 questions (one from each module)
- Duration: 3 hours

C. Lab Record Evaluation Criteria

- **Code Quality (30%):** Clean, well-commented, reproducible code
- **Understanding (25%):** Clear explanations of methodology and results
- **Analysis (25%):** Proper interpretation of results and performance metrics
- **Innovation (10%):** Creative approaches or improvements to basic requirements
- **Documentation (10%):** Professional presentation and organization

D. Lab Report Format

Each experiment report should contain:

1. **Title and Objective** (1-2 pages)
2. **Theory and Literature Review** (2-3 pages)
3. **Methodology and Implementation** (3-4 pages)
4. **Results and Analysis** (2-3 pages)
5. **Conclusion and Future Work** (1 page)

E. Submission Guidelines

- Submit both digital (PDF) and printed versions
- Include complete code with detailed comments
- All graphs and visualizations must be properly labeled
- Submit within one week of lab completion

Experiment 1: Neural Network for Word Embedding Generation

➢ Objective

Design and implement a neural network-based approach for generating word embeddings from text corpus, understanding semantic relationships in vector space.

➢ Prerequisites

- Basic understanding of neural networks
- Text preprocessing concepts
- Vector space mathematics

➢ Theory Background

Word2Vec is a widely used word representation technique that uses neural networks under the hood. The resulting word representation or embeddings can be used to infer semantic similarity between words and phrases, expand queries, surface related concepts and more. The sky is the limit when it comes to how you can use these embeddings for different NLP tasks. Word embeddings represent words as dense vectors in continuous space where semantically similar words are positioned closer together. Word2Vec, developed by Mikolov et al., uses neural networks to learn these representations through two main architectures:

1. **CBOW (Continuous Bag of Words):** Predicts target word from context
2. **Skip-gram:** Predicts context words from target word

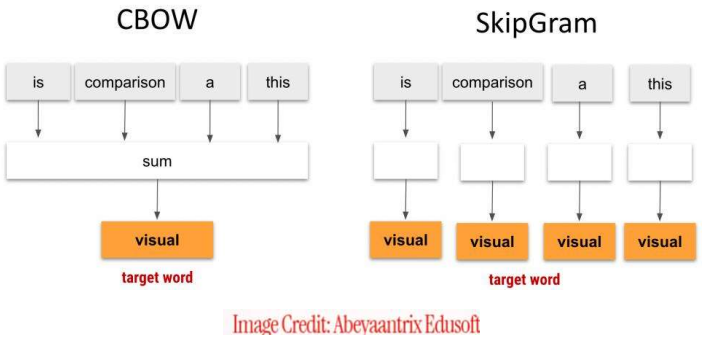


Figure 1: Difference between SkipGram and CBOW training architectures.

1. CBOW (CONTINUOUS BAG OF WORDS):

CBOW is one of the two main architectures of Word2Vec. CBOW tries to predict a target word given its surrounding context word.

Example:

Suppose you have a sentence:

"The cat sat on the mat"

If we set a context window = 2, then for the target word “sat”, the context words are:

["The", "cat", "on", "the"]

- Input: Context words (The, cat, on, the)
- Output: Predict the target word (sat)

How it Works

1. Input Representation
    - Each context word is represented as a one-hot vector.
    - Suppose vocabulary size is 10,000. Each word is a 10,000-dim vector with just one “1”.
2. Projection / Embedding Layer
    - These one-hot vectors are projected into a lower-dimensional embedding space (say 300 dimensions).
    - CBOW averages (or sums) the embeddings of all context words.
3. Hidden Layer → Output
    - The averaged context embedding is passed through a hidden layer and then a softmax classifier.
    - The softmax outputs probabilities for all words in the vocabulary.
    - The model is trained to maximize the probability of the true target word.

Key Points

- **Fast Training:** Works well with large corpora.
- **Better for Frequent Words:** Since common words appear often, CBOW learns stable embeddings for them.
- **Context to Target:** Always predicts the center word from surrounding context.

2. SKIP-GRAM IN WORD2VEC

Skip-gram is the reverse of CBOW. It tries to **predict the surrounding context words given a target word**.

Example

Sentence:

"The cat sat on the mat"

If window size = 2, and target word = **"sat"**, then context words are:

["The", "cat", "on", "the"]

- **Input:** Target word = "sat"
- **Output:** Predict its context words ("The", "cat", "on", "the")

How it Works

1. **Input Representation**
   - The target word (e.g., "sat") is represented as a **one-hot vector**.
2. **Projection / Embedding Layer**
   - That one-hot vector is projected into a dense embedding (e.g., 300-dimensional vector).
3. **Prediction of Context Words**
   - The embedding is then used to predict **each surrounding context word**.
   - This is done using a **softmax classifier** (or hierarchical softmax / negative sampling for efficiency).
4. **Training Objective**
   - Maximize the probability of predicting the true context words around the target word.

- **Key Differences: CBOW vs Skip-gram**

Aspect	CBOW	Skip-gram
Input	Context words	Target word
Output	Target word	Context words
Training Speed	Faster (fewer predictions)	Slower (multiple predictions)
Performance on Frequent Words	Better	Good
Performance on Rare Words	Good	Better
Memory Usage	Lower	Higher
Use Case	Large datasets, frequent words	Small datasets, rare words

Key Points

- **Slower to Train** than CBOW (because it predicts multiple words per target).
- **Better for Rare Words** since each occurrence of a rare word gives multiple training samples (target → many contexts).
- **Captures More Detailed Semantic Relationships** than CBOW.

- **Implementation Steps**

Step 1: Install & Import Libraries

```
# Install gensim (only first time)
# !pip install gensim

from gensim.models import Word2Vec
```

- **gensim**: A Python library widely used for NLP tasks, especially word embeddings.
- **Word2Vec**: Model for learning vector representations of words.

Step 2: Prepare the Dataset

```
Tokenized dataset: each sentence is a list of words
    sentences = [
        ["natural", "language", "processing", "is", "fun"],
        ["word2vec", "is", "a", "powerful", "technique"],
        ["we", "use", "word2vec", "for", "nlp", "tasks"],
        ["deep", "learning", "improves", "nlp"],
         ["language", "models", "are", "useful"]
```

The dataset is a list of tokenized sentences.

- Each word is a token → the model learns embeddings from this context.

Step 3: Train Word2Vec Model

```
model = Word2Vec(
    sentences=sentences,
    vector_size=50,   # embedding dimensions
    window=3,         # context window size
    min_count=1,      # ignore words with frequency < 1
    sg=1,             # 0 = CBOW, 1 = Skip-gram
    workers=4         # parallel training threads
)
```

- **vector_size**: Size of the word vector (50-dim here).
- **window**: Number of surrounding words considered as context.
- **min_count**: Ignores words appearing fewer times than this threshold.
- **sg**: Defines model type: sg=0 → CBOW, sg=1 → Skip-gram.
- **workers**: Number of CPU cores for faster training.

Step 4: Explore Word Embeddings

```
# Retrieve vector representation of a word
    print(model.wv['nlp'])

    # Find most similar words
    print(model.wv.most_similar('nlp'))
```

- model.wv['nlp'] → gives a **50-dim embedding vector** of the word "nlp".
- most_similar → finds semantically similar words based on embeddings.

Step 5: Save & Reuse Model

```
    # Save the trained model
    model.save("word2vec.model")

    # Load the model back
    loaded_model = Word2Vec.load("word2vec.model")
```

- Allows reusing embeddings without retraining.

*This code demonstrates **Skip-gram Word2Vec** (since sg=1) on a small dataset.*

- ***Input:** Sentences (words in context).*
- ***Output:** Dense vectors representing semantic relationships.*
- *You can switch to **CBOW** by setting sg=0.*

➢ Expected Results

Command	Expected Output
model.wv['nlp']	50-dim vector like [0.02, -0.11, 0.05, ...]
model.wv.most_similar('nlp')	[('processing', 0.87), ('language', 0.84), ...]
model.wv.similarity('language', 'processing')	~0.7 – 0.8 (high similarity)

Experiment 1: Word Embeddings (Word2Vec)

```
# Install gensim (if needed)
# !pip install gensim

from gensim.models import Word2Vec

# Tokenized dataset
sentences = [
    ["natural", "language", "processing", "is", "fun"],
    ["word2vec", "is", "a", "powerful", "technique"],
    ["we", "use", "word2vec", "for", "nlp", "tasks"],
    ["deep", "learning", "improves", "nlp"],
    ["language", "models", "are", "useful"]
]

# Train Skip-gram model
model = Word2Vec(sentences=sentences, vector_size=50, window=3, min_count=1, sg=1, workers=4)

# Explore
print(model.wv['nlp'])
print(model.wv.most_similar('nlp'))
```

➢ Viva Questions

1. What is the difference between CBOW and Skip-gram architectures?
2. Why do we use embeddings instead of one-hot encoding in NLP?
3. Explain the role of negative sampling in Word2Vec training.
4. Give two real-world applications of word embeddings.
5. How can we evaluate the quality of word embeddings?
6. Compare CBOW vs Skip-gram architectures – what are their advantages?
7. What is the significance of embedding dimensions?
8. How does the context window size affect Word2Vec performance?
9. What is hierarchical softmax and when is it used?
10. What are the limitations of Word2Vec compared to modern embeddings like BERT or GPT?

Experiment 2: Deep Learning for Classification

➢ Objective

Design and implement a deep neural network for classification tasks using popular datasets (MNIST, Iris). Understand how neural networks learn to classify inputs into multiple classes.

➢ Prerequisites

- Basics of supervised learning
- Concept of neural networks (layers, activation functions)
- Familiarity with Python & TensorFlow/Keras

➢ Theory Background

Deep Neural Networks (DNNs) are artificial neural networks with multiple hidden layers between input and output layers. They can model complex non-linear relationships and form the foundation of modern machine learning applications.

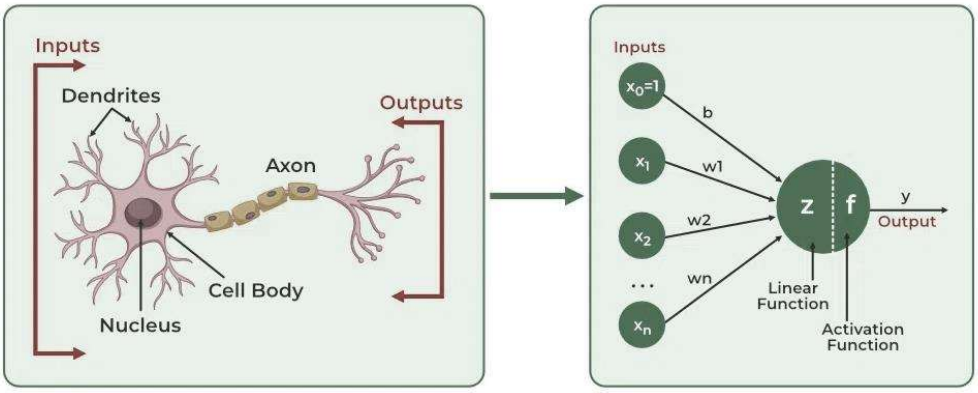


Image Credit: Abevaantrix Edusoft

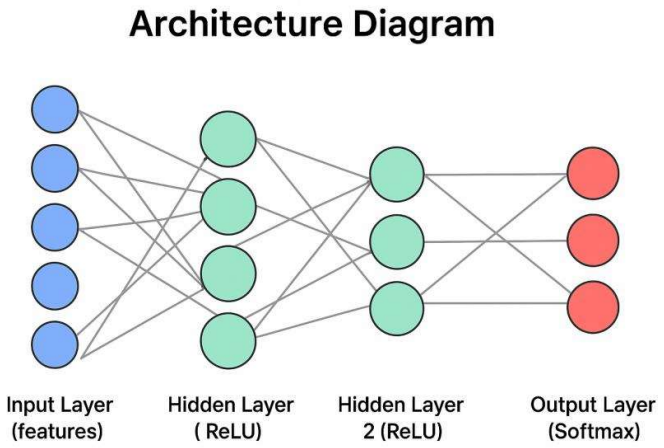
Biological neurons to Artificial neurons

- Each neuron applies a **linear transformation** followed by a **non-linear activation** (e.g., ReLU, sigmoid, softmax).
- For classification:
  - **Input layer** receives features (e.g., pixels in images or measurements in Iris dataset).
  - **Hidden layers** extract higher-level patterns.
  - **Output layer** (with softmax) assigns class probabilities.

A. KEY COMPONENTS OF ANN

1. **Activation Functions** (introduce non-linearity):

ReLU (Rectified Linear Unit) Activation Function



ANN block diagram: Image Credit: Abevaantrix Edusoft

Definition:

f(x)=max⁡(0,x)f(x) = \max(0, x)f(x)=max(0,x)

This means:

- If input x is positive → output is x.
- If input x is negative or zero → output is 0.

How it works (Step by Step):

1. Suppose a neuron computes a weighted sum:

z=w1x1+w2x2+bz = w_1x_1 + w_2x_2 + bz=w1x1+w2x2+b

2. Before passing to the next layer, we apply ReLU:

a=f(z)=max⁡(0,z)a = f(z) = \max(0, z)a=f(z)=max(0,z)

3. So only positive signals are passed forward, negatives are **blocked as zero**.

Why ReLU?

- **Fast to compute** (just compare with 0).
- **Prevents vanishing gradients** (unlike sigmoid/tanh).
- **Sparse activation:** Many neurons output zero, which makes the network more efficient and easier to learn.

Example:

Input values: −3,−1,0,2,5-3, -1, 0, 2, 5−3,−1,0,2,5

Apply ReLU: 0,0,0,2,50, 0, 0, 2, 50,0,0,2,5

👉 Negative inputs become **0**, positives remain unchanged.

Visualization (Mental Picture):

- Draw a graph:
  - X-axis: input
  - Y-axis: output
  - For $x < 0$ → line along $y = 0$
  - For $x \geq 0$ → line $y = x$ (a diagonal)

⚡ In practice: If we feed an image to a neural network → pixel values that are positive (important features) are **highlighted and passed on**, while irrelevant signals (negative values) are **ignored**.

Like such we have other activation functions also.

a. **ReLU (Rectified Linear Unit):**
   Most widely used; outputs 0 if input < 0, otherwise outputs input. Fast and avoids vanishing gradient issues.

b. **Sigmoid:**
   Squashes values between 0 and 1. Good for probabilities but can cause vanishing gradients.

c. **Tanh:**
   Squashes values between -1 and 1. Centered around zero, better than sigmoid in some cases.

🔑 **Why needed?** Without them, the network would just behave like a linear model and fail to capture complex relationships.

2. **Forward Propagation**

   - Input data passes layer by layer → multiplied by weights → added with bias → activation function applied.
   - Finally, the output layer produces prediction.
     👉 Think of it as **data flowing forward**.

3. **Backpropagation**
    - After prediction, we calculate **error (loss)**.
    - Backpropagation uses **gradients** (derivatives) to update weights and reduce error.
    - Works with optimization algorithms like **Gradient Descent**.

☞ Think of it as **error flowing backward** to adjust weights.

4. **Loss Functions**
    - Measure how far predictions are from the actual values.
    - **Cross-Entropy Loss:** Used in classification problems (e.g., cat vs dog).
    - **MSE (Mean Squared Error):** Used in regression tasks (predicting numbers).
5. **Regularization** (to avoid overfitting – when model memorizes instead of generalizing):
    - **Dropout:** Randomly ignores some neurons during training to force generalization.
    - **L1/L2 Regularization:** Adds penalty terms to weight values to keep them small and avoid complexity.

Here's a simplified view of a **fully connected neural network for classification**:
Input Layer (features) → Hidden Layer 1 (ReLU) → Hidden Layer 2 (ReLU) → Output Layer (Softmax classes)

- **MNIST**: Input = 784 pixels → Hidden layers → Output = 10 digits (0–9).
- **Iris**: Input = 4 features → Hidden layers → Output = 3 flower types.

➢ Implementation Steps

Step 1: Import Libraries

```
# !pip install tensorflow  # only once if not installed

from tensorflow import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.utils import to_categorical
print("Libraries imported successfully")
```

Step 2: Load Dataset (Iris examp

```
from sklearn.datasets import load_iris
import pandas as pd

iris = load_iris(as_frame=True)
Iris = iris.frame
Iris.head()
```

Step 3: Preprocess Data

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X = Iris.iloc[:, :-1]   # features
y = Iris.iloc[:, -1]    # labels

# Normalize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# One-hot encode labels
y_categorical = to_categorical(y)

# Split into train/test
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
y_categorical, test_size=0.2, random_state=42)
```

Step 4: Build the Neural Network

```
model = Sequential()
model.add(Dense(10, input_dim=4, activation='relu'))   # hidden layer
model.add(Dense(8, activation='relu'))                 # hidden layer
model.add(Dense(3, activation='softmax'))              # output (3
classes)

model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

Step 5: Train the Model

```
history = model.fit(X_train, y_train, epochs=50, batch_size=5,
validation_split=0.1)
```

Step 6: Evaluate the Model

```
loss, acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {acc:.2f}")
```

➢ **Expected Results**

- **Accuracy**: ~95%–98% for **Iris dataset**.
- **Training graph**: Loss decreases, accuracy increases with epochs.
- For **MNIST (digits dataset)**, DNN should achieve ~97% accuracy.

Experiment 2: Deep Neural Network (Iris Classification)

```
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load Iris dataset
iris = load_iris(as_frame=True)
Iris = iris.frame
X = Iris.iloc[:, :-1]
y = Iris.iloc[:, -1]

# Preprocess
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
y_categorical = to_categorical(y)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_categorical, test_size=0.2, random_state=42)

# Build Model
model = Sequential()
model.add(Dense(10, input_dim=4, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(3, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=50, batch_size=5, validation_split=0.1)

loss, acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {acc:.2f}")
```

- **Viva Questions**

1. What is the difference between shallow and deep neural networks?
2. Why do we use activation functions like ReLU instead of linear?
3. Explain the role of softmax in classification tasks.
4. What is the purpose of normalization in preprocessing?
5. Why do we split data into train and test sets?
6. What are the advantages of using more hidden layers?
7. Compare Iris classification vs MNIST classification challenges.
8. What is overfitting and how can we prevent it in DNNs?
9. How does batch size affect training?
10. What are real-world applications of DNN classification?

Experiment 3: Convolutional Neural Network (CNN) for Image Classification

➢ Objective

To design and implement a Convolutional Neural Network (CNN) for image classification on the Fashion-MNIST dataset and understand how convolutional layers extract spatial features from images.

➢ Prerequisites

- Basics of Neural Networks
- Concept of images as pixel matrices
- Familiarity with Keras/TensorFlow

➢ Theory Background

A **Convolutional Neural Network (CNN)** is designed mainly for **images**. Instead of connecting every pixel to every neuron (like in ANN), CNNs use **filters (kernels)** that look at small patches of the image at a time.

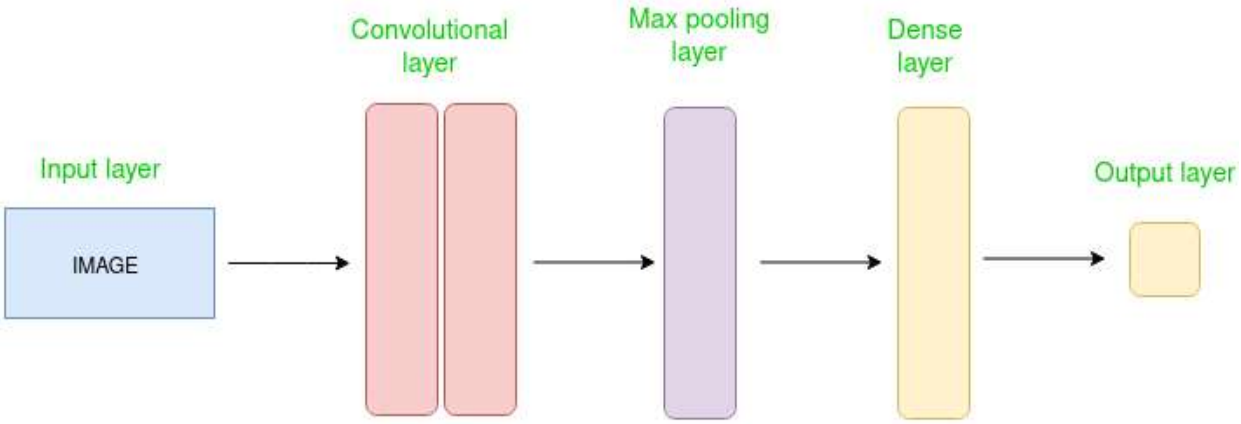


Image Credit: Abevaantrix Edusoft

Step-by-Step Explanation

1. **Input Layer**
   - An image is given as input.
   - Example: A **32 × 32 × 3** image (width × height × channels → RGB).

2. **Convolutional Layer**
    - A **filter (small matrix, e.g., 3×3 or 5×5)** slides over the image.
    - At each position, it calculates a **dot product** between filter values and pixel values.
    - The result is a **feature map** that highlights important features (edges, corners, textures).
    - If we use 12 filters → we get **12 feature maps**.

☞ Think of filters as "feature detectors." One filter may detect vertical edges, another horizontal lines, another corners.

3. **Activation Layer**
    - After convolution, we apply a function like **ReLU** → turns negative values into 0, keeps positives.
    - This adds **non-linearity** so the model can learn complex patterns.
4. **Pooling Layer**
    - Reduces the size of feature maps to make computations faster and avoid overfitting.
    - Example: **Max Pooling (2×2)** → keeps only the maximum value in each patch.
    - If input was 32×32×12, after pooling it becomes 16×16×12.

☞ Pooling = "shrinking the image while keeping important info."

5. **Flattening**
    - After several convolution + pooling steps, the feature maps are converted into a **1D vector**.
    - This prepares the data for the fully connected layers.
6. **Fully Connected Layer (Dense Layer)**
    - Works like a traditional ANN.
    - Uses the extracted features to classify or predict.

7. **Output Layer**

   - Produces the final prediction.
   - Example:
     - **Softmax** → probabilities for multiple classes (e.g., cat = 0.8, dog = 0.2).
     - **Sigmoid** → probability for binary classification (yes/no).

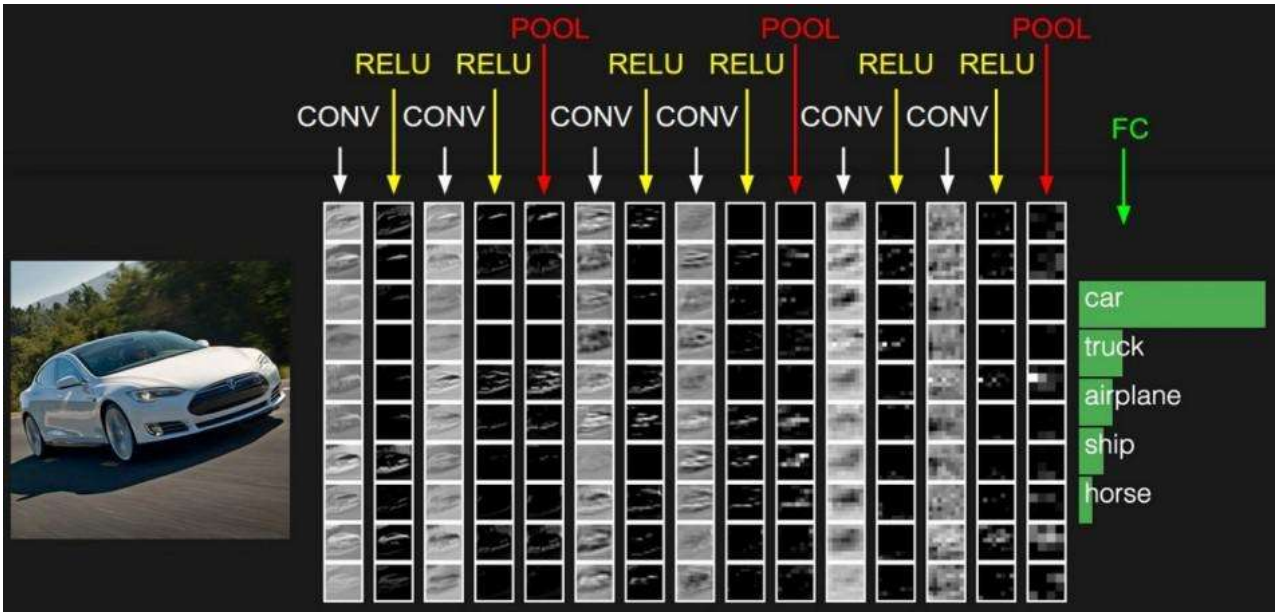


Figure: Sample CNN filter visualization Image Credit: Abeyaantrix Edusoft

➢ CNN Architecture Diagram

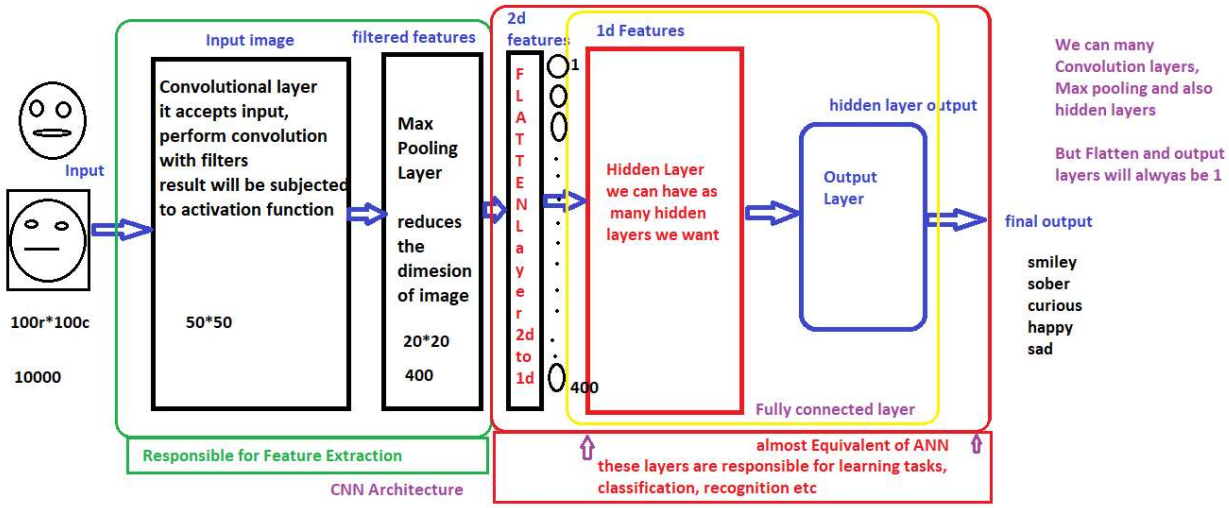


Image Credit: Abeyaantrix Edusoft

➢ Implementation Steps

Step 1: Import Libraries

```
from keras.models import Sequential
from keras.layers import Dense, Convolution2D, MaxPooling2D, Flatten
from keras.datasets import fashion_mnist
import numpy as np
import matplotlib.pyplot as plt
from keras.utils import to_categorical

print("Libraries imported successfully")
```

Step 2: Load and Explore Dataset

```
(X_train, Y_train), (X_test, Y_test) = fashion_mnist.load_data()


print(X_train.shape, Y_train.shape)
print(X_test.shape, Y_test.shape)
```

- **Fashion-MNIST** has 60,000 training and 10,000 test images (28×28 grayscale).
- 10 classes: T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot.

Step 3: Preprocess Data

```
# Reshape and normalize
X_train = X_train.reshape(-1, 28, 28, 1) / 255.0
X_test = X_test.reshape(-1, 28, 28, 1) / 255.0

# One-hot encode labels
Y_train = to_categorical(Y_train, 10)
Y_test = to_categorical(Y_test, 10)
```

Step 4: Build CNN Model

```
model = Sequential()
model.add(Convolution2D(32, (3,3), activation='relu',
input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

Step 5: Train the Model

```
history = model.fit(X_train, Y_train, epochs=10, batch_size=128,
validation_split=0.1)
```

Step 6: Evaluate the Model

```
loss, acc = model.evaluate(X_test, Y_test)
print(f"Test Accuracy: {acc:.2f}")
```

➢ **Expected Results**

- **Training Accuracy**: ~90–92%
- **Test Accuracy**: ~88–90% (Fashion-MNIST is harder than MNIST).
- **Loss curve**: Decreases over epochs.
- **Accuracy curve**: Increases steadily, validation accuracy stabilizes around 0.88–0.90.

Experiment 3: CNN (Fashion-MNIST Classification)

```
from keras.models import Sequential
from keras.layers import Dense, Convolution2D, MaxPooling2D, Flatten
from keras.datasets import fashion_mnist
from keras.utils import to_categorical

# Load dataset
(X_train, Y_train), (X_test, Y_test) = fashion_mnist.load_data()

# Preprocess
X_train = X_train.reshape(-1, 28, 28, 1) / 255.0
X_test = X_test.reshape(-1, 28, 28, 1) / 255.0
Y_train = to_categorical(Y_train, 10)
Y_test = to_categorical(Y_test, 10)

# Build CNN
model = Sequential()
model.add(Convolution2D(32, (3,3), activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, Y_train, epochs=10, batch_size=128, validation_split=0.1)
print(model.evaluate(X_test, Y_test))
```

- **Viva Questions**

1. What is the role of convolution layers in CNN?
2. Why do we use pooling layers?
3. What is the difference between fully connected (Dense) and convolution layers?
4. Why do we normalize image pixel values?
5. What activation function is commonly used in CNN hidden layers and why?
6. Explain why softmax is used in the output layer.
7. What is overfitting in CNN and how can it be reduced?
8. How does CNN differ from a traditional feedforward neural network?
9. Why is Fashion-MNIST considered more challenging than MNIST?
10. Mention real-world applications of CNNs.

Experiment 4: Autoencoder for Data Compression

➢ **Objective**

To design and implement an **autoencoder neural network** for compressing image data (MNIST digits) and reconstructing it back, understanding unsupervised feature learning.

➢ **Prerequisites**

- Basics of neural networks
- Concept of dimensionality reduction
- Familiarity with Keras/TensorFlow

➢ **Theory Background**

Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning. Specifically, we'll design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of the original input.

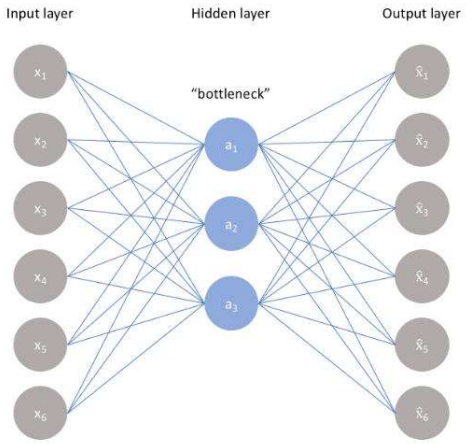


Image Credit: Abevaantrix Edusoft

An **Autoencoder** is a type of neural network used for **unsupervised learning of efficient codings**.
It consists of two parts:

1. **Encoder**: Compresses the input into a smaller representation (latent space).
2. **Decoder**: Reconstructs the input from the compressed form.

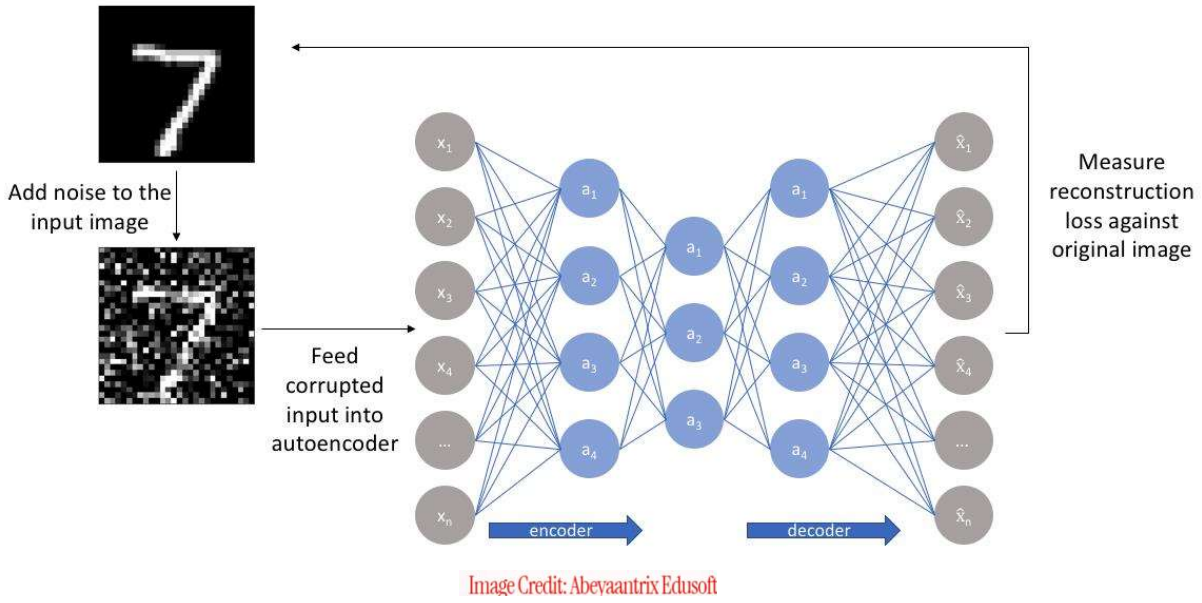


Image Credit: Abeyaantrix Edusoft

1. Encoder

It compresses 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.

Why use Autoencoders?

- Data compression
- Denoising
- Dimensionality reduction (like PCA, but non-linear)
- Feature extraction for downstream tasks

➢ **Implementation Steps**

Step 1: Import Libraries & Load Data

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Load data
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize
x_train = x_train / 255.0
x_test = x_test / 255.0

# Flatten inputs
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)
```

Step 2: Build Autoencoder

```
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(32, activation='relu'),   # bottleneck (compressed)
    Dense(64, activation='relu'),
    Dense(784, activation='sigmoid')  # output same size as input
])

model.compile(optimizer='adam', loss='mse')
model.summary()
```

Step 3: Train Autoencoder

```
model.fit(x_train, x_train, epochs=5, batch_size=256, validation_split=0.1)
```

- Note: Input = Output (autoencoder learns to reconstruct).
- Loss = reconstruction error (Mean Squared Error).

Step 4: Test Reconstruction

```
decoded = model.predict(x_test[:5])

import matplotlib.pyplot as plt
for i in range(5):
    plt.subplot(2, 5, i+1)
    plt.imshow(x_test[i].reshape(28,28), cmap='gray')
    plt.title("Original")
    plt.axis("off")

    plt.subplot(2, 5, i+6)
    plt.imshow(decoded[i].reshape(28,28), cmap='gray')
    plt.title("Reconstructed")
    plt.axis("off")
plt.show()
```

➢ **Expected Results**

- **Loss curve** decreases across epochs.
- Reconstructed images should be **like** originals, though slightly blurred.
- Compressed layer (32 neurons) stores the essential features of 784-pixel input.

Experiment 4: Autoencoder (MNIST Compression)

```python
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt

# Load data
(x_train, _), (x_test, _) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

# Build Autoencoder
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(32, activation='relu'),  # bottleneck
    Dense(64, activation='relu'),
    Dense(784, activation='sigmoid')
])

model.compile(optimizer='adam', loss='mse')
model.fit(x_train, x_train, epochs=5, batch_size=256, validation_split=0.1)

# Reconstruct
decoded = model.predict(x_test[:5])
for i in range(5):
    plt.subplot(2,5,i+1); plt.imshow(x_test[i].reshape(28,28), cmap='gray'); plt.axis("off")
    plt.subplot(2,5,i+6); plt.imshow(decoded[i].reshape(28,28), cmap='gray'); plt.axis("off")
plt.show()
```

➢ Viva Questions

1. What is an autoencoder?
2. How does an autoencoder differ from PCA?
3. What is the role of the encoder and decoder in an autoencoder?
4. Why do we use a smaller hidden layer (bottleneck)?
5. What loss function is used in autoencoders and why?
6. Can autoencoders be used for supervised tasks?
7. Explain undercomplete and overcomplete autoencoders.
8. What are denoising autoencoders?
9. Why is sigmoid often used in the output layer for images?
10. Give two real-world applications of autoencoders.

Experiment 5: RNN for Text Classification

➢ Objective

Design and implement a deep learning network for classification of textual documents using LSTM. Understand how recurrent neural networks capture sequential dependencies in text data.

➢ Prerequisites

- Basics of Natural Language Processing (NLP)
- Understanding of RNN and LSTM architectures
- Familiarity with Keras/TensorFlow
- Knowledge of tokenization, padding, and label encoding

➢ Theory Background

Traditional neural networks struggle with sequential data like text, where word order and context matter. **Recurrent Neural Networks (RNNs)** address this by maintaining a hidden state that captures dependencies across time steps. However, standard RNNs face **vanishing gradient problems**, making them ineffective for long sequences.

Long Short-Term Memory (LSTM) networks overcome this by introducing **gates** (input, forget, output) that control the flow of information, allowing the model to capture long-range dependencies.

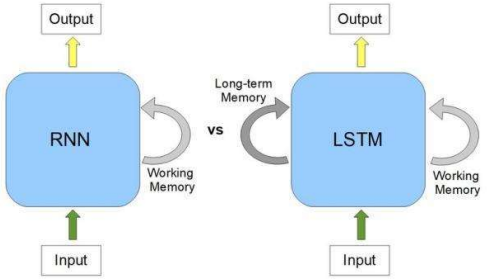


Image Credit: Abevaantrix Edusoft

❖ What is LSTM?

LSTM (Long Short-Term Memory) is a type of neural network that can remember important information for long periods of time. Think of it like a smart notebook that can decide what to remember, what to forget, and what to share.

❖ **The Main Idea**

Imagine you're reading a long story. As you read:

- You remember important characters and plot points
- You forget irrelevant details
- You use what you remember to understand new parts of the story

LSTM works similarly with data sequences!

❖ **The Memory Cell**

At the heart of LSTM is a memory cell - like a storage box that holds important information. This cell is controlled by three "gates" (think of them as smart filters):

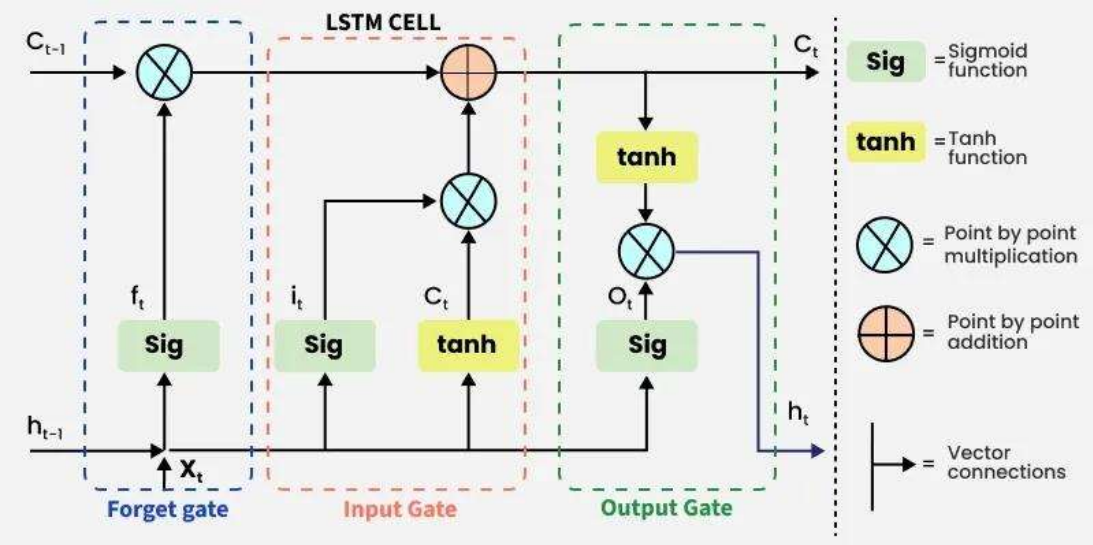


Image Credit: Abevaantrix Edusoft

- **The Three Gates**

1. Forget Gate - "What should I forget?"

- Job: Decides what old information is no longer needed
- How it works: Looks at new input and previous memory, then decides what to throw away
- Example: When reading "The cat was black. The dog was brown.", it might forget the cat's color when focusing on the dog

2. 📥 Input Gate - "What new information should I remember?"

- Job: Decides what new information is worth storing
- How it works: Filters incoming information and decides what's important enough to remember
- Example: When you read "John is the main character", it decides this is important to remember

3. 📤 Output Gate - "What should I share right now?"

- Job: Decides what information from memory to use for the current output
- How it works: Looks at what's stored in memory and decides what's relevant for the current situation
- Example: When asked "Who is the main character?", it retrieves "John" from memory

❖ **How It All Works Together**

1. New information arrives (like a new word in a sentence)
2. Forget Gate says: "Do I need to forget anything old?"
3. Input Gate says: "Should I remember this new information?"
4. Memory gets updated with the new important stuff
5. Output Gate says: "What should I output based on what I remember?"
6. Output is produced and sent to the next step

❖ **Why is LSTM Special?**

- Regular neural networks have short memory (like goldfish)
- LSTM networks can remember things from much earlier in the sequence (like elephants)
- This makes them great for tasks like:
    - Language translation
    - Speech recognition
    - Stock price prediction
    - Any task where context from the past matters

❖ **Simple Analogy**

Think of LSTM like a smart assistant reading your emails:

- **Forget Gate:** "This old meeting info is outdated, I'll forget it"
- **Input Gate:** "This new project deadline is important, I'll remember it"
- **Output Gate:** "When asked about upcoming deadlines, I'll mention the project"

Why LSTM for Text Classification?

- Captures word order and semantic context
- Effective for sentiment analysis, topic classification, and document categorization
- Works with embeddings → learns meaningful feature representations of words

Choosing RNN Type:

- **Start with LSTM**: Good default choice
- **Try GRU**: If you need speed or have limited data
- **Use Bidirectional**: For better context understanding
- **Add Attention**: For interpretability and performance

Hyperparameter Guidelines:

- **Sequence Length**: Start with 200-500 for most text tasks
- **Embedding Dimension**: 128-300 typically works well
- **Hidden Units**: 64-256 depending on complexity
- **Dropout**: 0.2-0.5 to prevent overfitting

Common Pitfalls to Avoid:

1. **Too long sequences**: Leads to memory issues and slower training
2. **No dropout**: Model will overfit on training data
3. **Wrong padding**: Can confuse the model
4. **Ignoring class imbalance**: Use weighted loss or resampling
5. **No validation set**: Can't detect overfitting

➢ Implementation Steps

Step 1: Import Libraries

```
import pandas as pd
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

Step 2: Load and Preprocess Dataset

```
# Load dataset
data = pd.read_csv("/content/drive/MyDrive/sentiment.csv")
print(data.head())

# Encode labels
le = LabelEncoder()
data['Sentiment'] = le.fit_transform(data['Sentiment'])  # e.g.,
positive=2, negative=0, neutral=1

# Tokenize text
tokenizer = Tokenizer(num_words=5000, oov_token="<OOV>")
tokenizer.fit_on_texts(data['Sentence'])
sequences = tokenizer.texts_to_sequences(data['Sentence'])

# Pad sequences to fixed length
padded = pad_sequences(sequences, maxlen=50, padding='post')

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(padded,
data['Sentiment'], test_size=0.2, random_state=42)
```

Step 3: Build LSTM Model

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=5000, output_dim=16,
input_length=50),
    tf.keras.layers.LSTM(32),
    tf.keras.layers.Dense(16, activation='relu'),
 tf.keras.layers.Dense(3, activation='softmax')  # 3 sentiment
classes
])
model.compile(loss='sparse_categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
```

Step 4: Train the Model

```
history = model.fit(X_train, y_train, epochs=5, batch_size=16,
validation_data=(X_test, y_test))
```

Step 5: Evaluate the Model

```
loss, acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {acc:.2f}")
```

Step 6: Sample Predictions

```
print(data[
print("Model predictions =", 'Sentence'][:2])
model.predict(padded[:2]))
print("Actual Sentiments =", data['Sentiment'][:2]
```

➢ Expected Results

- Training accuracy improves with epochs (~85–90% on sentiment dataset).
- Validation accuracy stabilizes around 80–85%.
- Predictions return probabilities across 3 sentiment classes.

Experiment 5: Text Classification (LSTM)

```
import pandas as pd
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load dataset
data = pd.read_csv("/content/drive/MyDrive/sentiment.csv")

# Encode labels
le = LabelEncoder()
data['Sentiment'] = le.fit_transform(data['Sentiment'])

# Tokenize & Pad
tokenizer = Tokenizer(num_words=5000, oov_token="<OOV>")
tokenizer.fit_on_texts(data['Sentence'])
sequences = tokenizer.texts_to_sequences(data['Sentence'])
padded = pad_sequences(sequences, maxlen=50, padding='post')

X_train, X_test, y_train, y_test = train_test_split(padded, data['Sentiment'], test_size=0.2, random_state=42)

# Build LSTM
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=5000, output_dim=16, input_length=50),
    tf.keras.layers.LSTM(32),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=5, batch_size=16, validation_data=(X_test, y_test))
print(model.evaluate(X_test, y_test))
```

➢ Viva Questions

1. What is the difference between RNN and LSTM?
2. Why do we use padding in text preprocessing?
3. Explain the role of word embeddings in text classification.
4. What are vanishing gradients, and how does LSTM solve them?
5. Compare Bag-of-Words vs LSTM approaches for text classification.
6. Why is softmax used in the output layer for multi-class classification?
7. How would performance change if we used GRU instead of LSTM?
8. Explain the importance of sequence length (maxlen) in padding.
9. How can we prevent overfitting in LSTM models?
10. Suggest real-world applications of LSTM-based text classification.

Experiment 6: LSTM for Time Series Forecasting

➢ Objective

Design and implement a deep learning network (LSTM and ANN) for forecasting time series data. Understand how sequential dependencies in temporal data are captured for future predictions.

➢ Prerequisites

- Basics of sequential data and time series
- Concepts of Artificial Neural Networks (ANN), RNNs, and LSTM
- Familiarity with Keras/TensorFlow
- Knowledge of normalization and sliding windows in time series

➢ Theory Background

Time series forecasting aims to predict future values of a variable using its past behavior. Unlike static datasets, time series data has **temporal structure**, making it essential to capture dependencies across time.

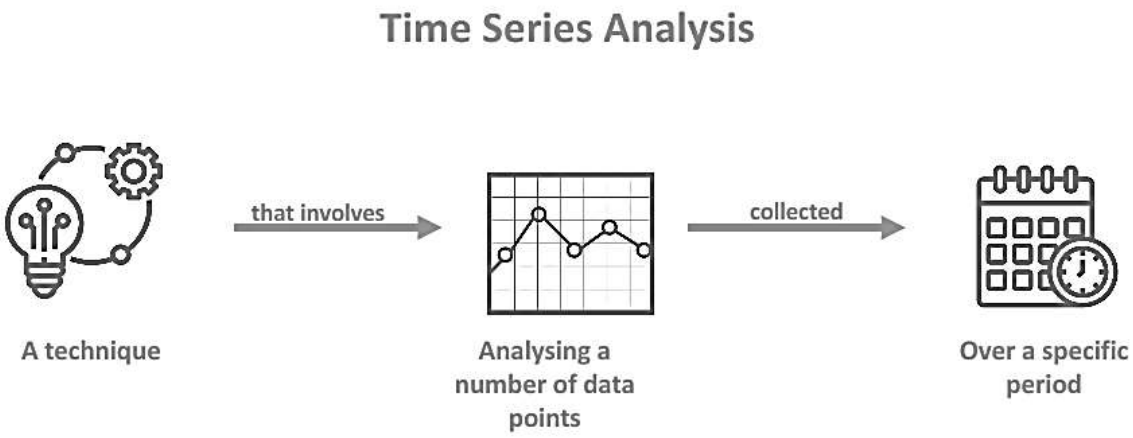


Image Credit: Abeyaantrix Edusoft

Key Characteristics of Time Series

1. **Temporal Dependencies –** The current value is influenced by past observations.
2. **Trend –** Long-term upward or downward movements (e.g., population growth, stock index).
3. **Seasonality –** Repeated patterns at fixed intervals (e.g., holiday sales, monthly airline passengers).
4. **Cyclic Behavior –** Fluctuations without a fixed period, often linked to economic/business cycles.
5. **Noise/Irregularity –** Random variations that add uncertainty and must be filtered.

Traditional Forecasting Approaches

1. **Statistical Models**
    - **AR (Auto-Regressive):** Current value depends linearly on past values.
    - **MA (Moving Average):** Current value depends on past error terms.
    - **ARIMA:** Combination of AR and MA with differencing to handle non-stationarity.
    - **SARIMA:** Extends ARIMA to model seasonality explicitly.
2. **Machine Learning Approaches**
    - **Regression Models (Linear/Polynomial):** Fit trend/seasonality using features like time index.
    - **Decision Trees / Random Forests:** Handle non-linear dependencies but struggle with sequential continuity.
    - **Support Vector Regression (SVR):** Works well for short-term patterns.
3. **Deep Learning Approaches**
    - **Feedforward ANN:** Uses a sliding window (past n values → next value). Captures local dependencies but not long sequences.
    - **Convolutional Neural Networks (1D-CNN):** Capture local temporal patterns efficiently; used in hybrid models.
    - **Recurrent Models (LSTM/GRU):** Best suited for long-term sequential dependencies, robust against vanishing gradients.
    - **Hybrid Models:** Combine CNN (feature extraction) + LSTM (temporal learning) for better forecasting accuracy.

Why Deep Learning for Forecasting?

- Can capture **non-linear patterns** missed by ARIMA-like models.
- Adaptable to multivariate time series (multiple features such as temperature + demand).
- Effective for complex domains: stock price prediction, weather, energy demand, healthcare signals.

☞ **Experiment Focus:**

In this experiment, we implement **two models**:

- **Feedforward ANN** – baseline for short-term dependencies.
- **LSTM** – advanced method for sequential modeling.

This allows comparison between **basic neural networks vs sequence-aware models** and helps understand why recurrent models are often preferred for time series forecasting.

➢ Implementation Steps

Version 1: LSTM-based Forecasting

Step 1: Import Libraries & Load Data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler

# Load dataset
df = pd.read_csv("/content/drive/MyDrive/AirPassenger.csv")
df['Year-Month'] = pd.to_datetime(df['Year-Month'])
df.set_index('Year-Month', inplace=True)

# Extract values
data = df[['Pax']].values

# Normalize
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
```

Step 2: Create Sequences

```
def create_sequences(dataset, time_step=12):
    X, y = [], []
    for i in range(len(dataset)-time_step):
        X.append(dataset[i:i+time_step])
        y.append(dataset[i+time_step])
    return np.array(X), np.array(y)

time_step = 12
X, y = create_sequences(scaled_data, time_step)
```

Step 3: Train-Test Split & Reshape

```
train_size = int(len(X)*0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Reshape for LSTM: [samples, timesteps, features]
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_test  = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))
```

Step 4: Model building and training

Build L
STM Model

```
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(time_step, 1)),
    LSTM(50, return_sequences=False),
    Dense(25, activation='relu'),
    Dense(1)
])

model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=50, batch_size=16,
validation_data=(X_test, y_test))
```

Step 5: Predictions & Visualization

```
train_predict = scaler.inverse_transform(model.predict(X_train))
test_predict  = scaler.inverse_transform(model.predict(X_test))
y_train_actual = scaler.inverse_transform(y_train.reshape(-1,1))
y_test_actual  = scaler.inverse_transform(y_test.reshape(-1,1))


plt.figure(figsize=(10,6))
plt.plot(df.index, data, label="Actual Data")
plt.plot(df.index[time_step:len(train_predict)+time_step],
train_predict, label="Train Predict")
plt.plot(df.index[len(train_predict)+(time_step*2):], test_predict,
label="Test Predict")
plt.legend()
plt.show()
```

Version 2: ANN-based Forecasting

```
from tensorflow.keras.layers import Dense

# Flattened sequences for ANN
def create_sequences(dataset, time_step=12):
    X, y = [], []
    for i in range(len(dataset)-time_step):
        X.append(dataset[i:i+time_step, 0])
        y.append(dataset[i+time_step, 0])
    return np.array(X), np.array(y)

X, y = create_sequences(scaled_data, 12)

# Train-Test split
train_size = int(len(X) * 0.8)
X_train, X_test, y_train, y_test = X[:train_size], X[train_size:],
y[:train_size], y[train_size:]

# ANN model
model = Sequential
    Dense(64, activation='relu', input_shape=(12,)),
    Dense(32, activation='relu'),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=100, batch_size=16,
validation_data=(X_test, y_test))
```

➢ Expected Results

- Training & Validation loss decreases across epochs.
- Predictions follow trend of original data (though slightly smoothed).
- LSTM generally outperforms ANN for sequential dependencies.

Experiment 6: Time Series Forecasting (LSTM & ANN)

```
import pandas as pd, numpy as np, matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler

# Load dataset
df = pd.read_csv("/content/drive/MyDrive/AirPassenger.csv")
df['Year-Month'] = pd.to_datetime(df['Year-Month'])
df.set_index('Year-Month', inplace=True)
data = df[['Pax']].values

scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)

# Create sequences
def create_sequences(dataset, time_step=12):
    X, y = [], []
    for i in range(len(dataset)-time_step):
        X.append(dataset[i:i+time_step])
        y.append(dataset[i+time_step])
    return np.array(X), np.array(y)

time_step=12
X, y = create_sequences(scaled_data, time_step)
train_size = int(len(X)*0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))

# LSTM Model
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(time_step,1)),
    LSTM(50, return_sequences=False),
    Dense(25, activation='relu'),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=50, batch_size=16, validation_data=(X_test,y_test))

# Predictions
train_predict = scaler.inverse_transform(model.predict(X_train))
test_predict  = scaler.inverse_transform(model.predict(X_test))
```

➢ Viva Questions

1. Why do we normalize data before training?
2. What is the difference between ANN and LSTM in time series forecasting?
3. Explain the role of the sliding window (time_step).
4. Why is reshaping required before feeding data into LSTM?
5. What are vanishing gradients, and how do LSTMs handle them?
6. Compare performance of ANN vs LSTM on time series data.
7. What loss function is typically used in forecasting tasks?
8. How do we evaluate forecasting accuracy?
9. Explain overfitting in time series models and ways to prevent it.
10. Give real-world applications of LSTM-based forecasting.

Experiment 7: Transfer Learning with Pre-trained Models

➢ Objective

To design and implement a transfer learning approach using a pre-trained CNN (ResNet50) model for image classification. Understand how pre-trained models on large datasets like ImageNet can be applied to new tasks with minimal training.

➢ Prerequisites

- Basics of Convolutional Neural Networks (CNNs)
- Familiarity with Keras/TensorFlow
- Understanding of concepts: feature extraction, fine-tuning, ImageNet dataset
- Knowledge of image preprocessing (resizing, normalization)

➢ Theory Background

Training deep neural networks from scratch requires large datasets and high computational power. **Transfer Learning** addresses this by reusing a model pre-trained on a large dataset (like ImageNet with 1.2M images across 1000 classes).

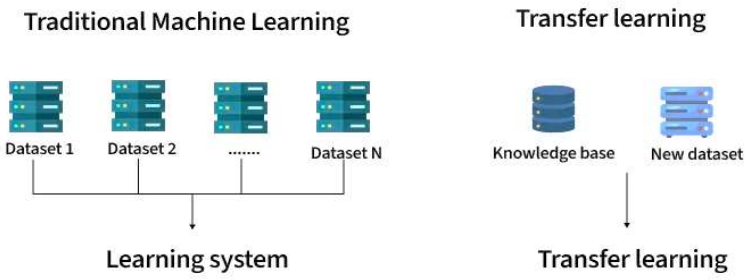


Image Credit: Abevaantrix Edusoft

Why Transfer Learning?

- Saves training time and resources
- Leverages rich feature representations already learned (edges, textures, object parts)
- Effective for small datasets
- Achieves state-of-the-art performance with minimal fine-tuning

Types of Transfer Learning

1. **Feature Extraction**
    - Use pre-trained model as a fixed feature extractor
    - Replace final classification layer to suit the new dataset
    - Freeze earlier layers, train only the classifier
2. **Fine-Tuning**
    - Unfreeze some deeper layers of the pre-trained model
    - Retrain with smaller learning rate on the new dataset
    - Allows adaptation to domain-specific features

Pre-trained Models (Keras Applications)

- ResNet50, VGG16, InceptionV3, MobileNet, EfficientNet etc.
- Each has different depth, parameter size, and performance trade-offs.

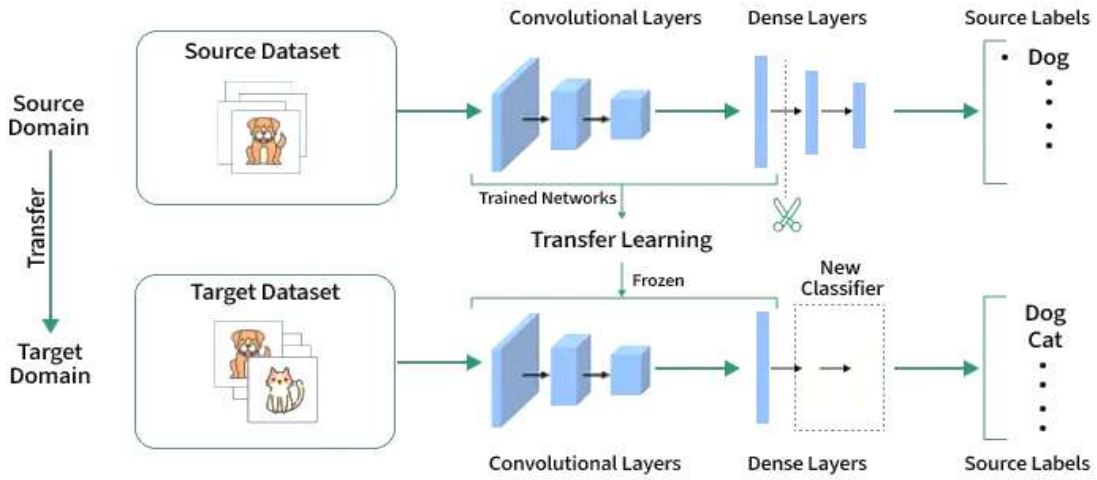


Image Credit: Abevaantrix Edusoft

How Does Transfer Learning Work?

Transfer learning involves a structured process to use existing knowledge from a pre-trained model for new tasks:

1. **Pre-trained Model:** Start with a model already trained on a large dataset for a specific task. This pre-trained model has learned general features and patterns that are relevant across related tasks.
2. **Base Model:** This pre-trained model, known as the base model, includes layers that have processed data to learn hierarchical representations, capturing low-level to complex features.

3. **Transfer Layers:** Identify layers within the base model that hold generic information applicable to both the original and new tasks. These layers often near the top of the network capture broad, reusable features.

4. **Fine-tuning:** Fine-tune these selected layers with data from the new task. This process helps retain the pre-trained knowledge while adjusting parameters to meet the specific requirements of the new task, improving accuracy and adaptability.

➢ Implementation Steps

Step 1: Import Libraries

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.applications.resnet50 import ResNet50,
preprocess_input, decode_predictions
from tensorflow.keras.preprocessing import image
import matplotlib.pyplot as plt
```

Step 2: Load and Preprocess Image

```
img_path = "/content/drive/MyDrive/dog.jpg"   # replace with your
image path
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
```

Step 3: Load Pre-trained Model

```
model = ResNet50(weights='imagenet')
preds = model.predict(x)
```

Step 4: Decode and Display Prediction

```
label = decode_predictions(preds, top=1)[0][0]
print(f"Predicted: {label[1]} ({label[2]*100:.2f}%)")

# Plot image with prediction
pred_text = f"Predicted: {label[1]} ({label[2]*100:.2f}%)"
plt.imshow(image.load_img(img_path))
plt.axis('off')
plt.title(pred_text)
plt.show()
```

➢ Expected Results

- The model outputs the **predicted label** of the input image along with probability (confidence score).
- Example: Predicted: Labrador_retriever (87.34%) for a dog image.
- Visualization shows input image with predicted label as title.

Experiment 7: Transfer Learning (ResNet50)

```
import tensorflow as tf, numpy as np, matplotlib.pyplot as plt
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input, decode_predictions
from tensorflow.keras.preprocessing import image

# Load & preprocess image
img_path="/content/drive/MyDrive/dog.jpg"
img=image.load_img(img_path, target_size=(224,224))
x=image.img_to_array(img)
x=np.expand_dims(x,axis=0)
x=preprocess_input(x)

# Load ResNet50
model=ResNet50(weights='imagenet')
preds=model.predict(x)

# Decode
label=decode_predictions(preds,top=1)[0][0]
print(f"Predicted: {label[1]} ({label[2]*100:.2f}%)")

plt.imshow(image.load_img(img_path)); plt.axis('off')
plt.title(f"{label[1]} ({label[2]*100:.2f}%)")
plt.show()
```

Viva Questions

1. What is transfer learning and why is it useful?
2. Differentiate between feature extraction and fine-tuning.
3. What is ImageNet and why is it widely used in transfer learning?
4. Why do we preprocess images before passing them to CNNs?
5. Explain the architecture of ResNet and why skip connections are important.
6. How does transfer learning help when datasets are small?
7. What are other popular pre-trained models apart from ResNet50?
8. Why do we freeze layers in transfer learning?
9. Compare transfer learning with training from scratch.
10. Give two real-world applications of transfer learning in AI.

Experiment 8: Reinforcement Learning Fundamentals (Q-Learning in Grid World)

➢ Objective

To design and implement a reinforcement learning agent using **Q-Learning** in a custom 2D grid world where the agent learns to navigate from a start position to a goal while avoiding obstacles.

➢ Prerequisites

- Basics of Reinforcement Learning (RL)
- Understanding of agent–environment interaction
- Knowledge of Q-Learning concepts: state, action, reward, Q-table, Bellman equation
- Familiarity with Python and NumPy

➢ Theory Background

Reinforcement Learning is a branch of machine learning where an **agent learns to take actions in an environment** to maximize cumulative rewards. Unlike supervised learning, RL does not require labeled input-output pairs.

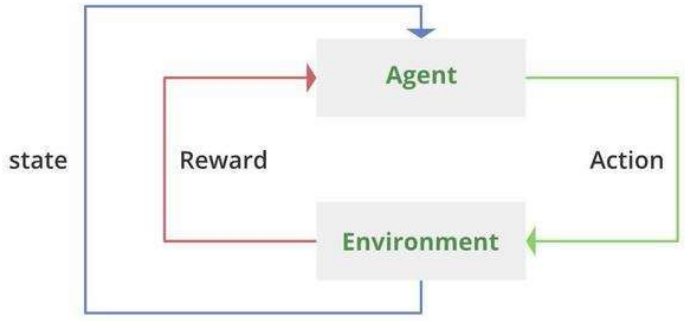


Image Credit: Abevaantrix Edusoft

Key Concepts

I. Agent (Learner/Decision-maker)

- The agent is the entity that interacts with the environment, makes decisions, and learns from feedback.
- In a grid world, the agent could be a robot trying to reach a goal cell.
- Its goal is to maximize cumulative rewards over time by choosing the best actions.

II. Environment (World)

- The environment is the system with which the agent interacts.
- It provides the agent with the state and returns a reward after each action.
- Example: In a self-driving car simulation, the environment includes roads, traffic signals, and pedestrians.

III. State (s)

- A state is a snapshot of the environment at a given time.
- It describes the agent's current situation.
- Example: In grid world, the state is the agent's current position (x,y).
- States can be discrete (e.g., grid cells) or continuous (e.g., robot's exact position in 3D).

IV. Action (a)

- An action is any possible move the agent can take from its current state.
- Example: In a grid, actions = {Up, Down, Left, Right}.
- The set of all possible actions is called the action space.

V. Reward (r)

- A reward is the feedback signal the agent receives after taking an action.
- Positive rewards encourage good behavior, negative rewards discourage poor actions.
- Example: +10 for reaching the goal, –1 for each step, –100 for hitting an obstacle.

VI. Policy (π)

- A policy defines the strategy the agent follows to select actions.
- Formally, it is a mapping from states → actions.
- Example: π(s) = always move right if the goal is on the right.
- Policies can be deterministic (fixed action per state) or stochastic (probability distribution over actions).

VII. Q-Table (Action-Value Function)

- A Q-table stores values (Q-values) for each state–action pair.
- Q(s,a) = expected future reward if the agent takes action *a* in state *s*, and follows the best policy afterward.
- Example: In grid world, Q[(2,3), "Right"] might equal +7, meaning moving right from (2,3) is likely good.
- The table gets updated over time as the agent learns.

VIII. Bellman Equation (Update Rule)

- The Bellman equation provides a way to iteratively update Q-values based on rewards and future predictions.

- **Formula:**

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

  - α\alphaα: Learning rate (how much we adjust old value with new info)
  - γ\gammaγ: Discount factor (how much we value future rewards)
  - rrr: Reward received after action
  - s′s's′: Next state
- **Intuition:** *Current Q-value = Old Q-value + (New information – Old Q-value).*

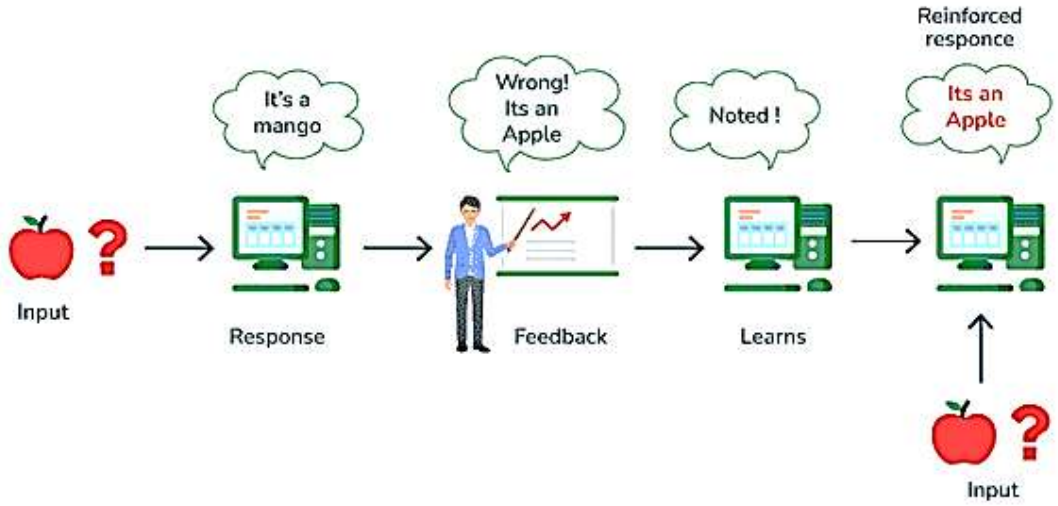


Image Credit: Abevaantrix Edusoft

Q-Learning Process: Step-by-Step

1. **Start at a State (S)**
   - The environment provides the agent with a **starting state**.
   - This state describes the current condition (e.g., position in the grid world).
2. **Agent Selects an Action (A)**
   - The agent chooses an action based on its **policy** and the **Q-table**.
   - Most commonly, the **ε-greedy strategy** is used:
     - With probability **ε**, the agent explores by choosing a random action.
     - With probability **1–ε**, it exploits by choosing the best-known action (highest Q-value).
3. **Action is Executed → Environment Responds**
   - The agent performs the chosen action.
   - The environment then returns:

- A **new state (S′)** → updated situation after the action.
- A **reward (R)** → feedback on the quality of the action.

4. **Q-Table is Updated (Learning Step)**
   - Using the new experience (S, A, R, S′), the agent updates the Q-table.
   - The update considers both the **immediate reward** and the **estimated future reward**.
   - Over many iterations, this helps the agent learn which actions are best in each state.
5. **Policy is Refined**
   - With updated Q-values, the agent improves its **policy** (action strategy).
   - The process of **observe → act → receive reward → update** repeats across episodes.
   - Eventually, the agent converges to the **optimal policy**, consistently yielding the highest possible rewards.

Methods for Determining Q-values

1. **Temporal Difference (TD) Learning**
   - TD methods update value estimates **based on the difference between predicted and actual rewards**.
   - They allow learning directly from raw experience **without needing a full model of the environment**.
   - Update rule:

$$TD\ Error = R + \gamma\, Q(s', a') - Q(s, a)$$

2. **Bellman's Equation**
   - A recursive formula that relates the value of the current state-action pair to:
     - The **immediate reward (R)**
     - The **discounted maximum future reward**
   - Formula:

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

   - Where:
     - Q(s,a)Q(s,a)Q(s,a): Q-value of taking action a in state s
     - R(s,a)R(s,a)R(s,a): Immediate reward
     - γ\gammaγ: Discount factor (importance of future rewards)
     - max⁡a′Q(s′,a′)\max_{a'} Q(s',a')maxa′Q(s′,a′): Best possible future Q-value from next state s'

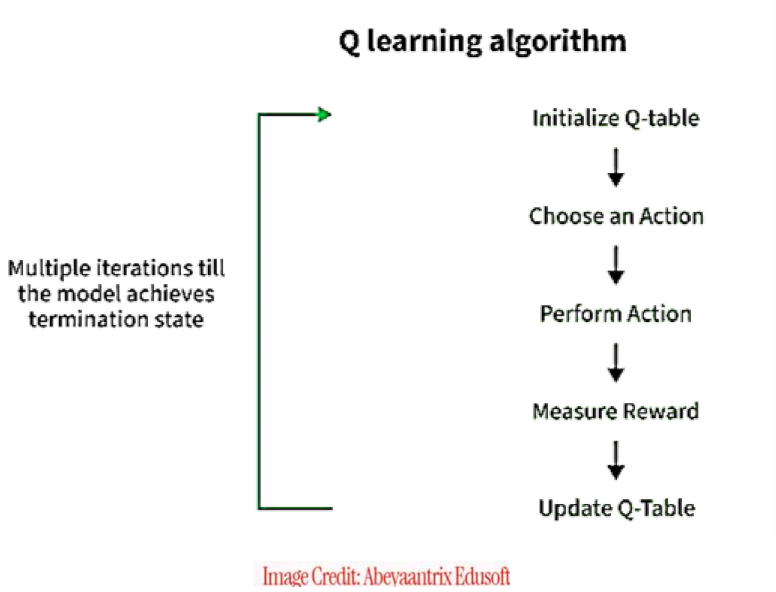


Image Credit: Abevaantrix Edusoft

➢ Implementation Steps

Step 1: Setup Environment

```
import numpy as np, random

size = 5
goal = (4, 4)
obstacles = [(1,2), (2,2), (3,1)]
actions = ['up', 'down', 'left', 'right']
Q = np.zeros((size, size, len(actions)))  # Q-table
```

Step 2: Define Move Function

```
def move(pos, action):
    x, y = pos
    if action == 'up': x -= 1
    if action == 'down': x += 1
    if action == 'left': y -= 1
    if action == 'right': y += 1
    if 0 <= x < size and 0 <= y < size and (x,y) not in obstaci
        return (x, y)
    return pos
```

Step 3: Set Training Parameters

```
episodes = 300
lr = 0.1
gamma = 0.9
eps = 0.2   # exploration rate
```

Step 4: Q-Learning Training Loop

```
for ep in range(episodes):
    state = (0, 0)
    for step in range(100):
        # ε-greedy selection
        if random.random() < eps:
            a = random.randint(0, 3)
        else:
            a = np.argmax(Q[state[0], state[1]])

        next_state = move(state, actions[a])
        reward = 10 if next_state == goal else -1

        # Q-update
        Q[state[0], state[1], a] += lr * (
            reward + gamma * np.max(Q[next_state[0], next_state[1]]) -
Q[state[0], state[1], a]
        )

        state = next_state
        if state == goal:
            break
```

Step 5: Extract Learned Policy

```
policy = [['']*size for _ in range(size)]
for i in range(size):
    for j in range(size):
        if (i,j) in obstacles:
            policy[i][j] = 'X'
        elif (i,j) == goal:
            policy[i][j] = 'G'
        else:
            best = np.argmax(Q[i,j])
            policy[i][j] = actions[best][0].upper()
print("Learned Policy Grid:")
for row in policy:
    print(' '.join(row))
```

➢ Expected Results

- The agent initially explores randomly.
- After training, the Q-table converges.
- The final **policy grid** shows the best action from each cell:
    - U (up), D (down), L (left), R (right)
    - X = obstacle, G = goal

Example Output:

R R R D D
D U X R D
D L X R D
D X D D D
R R R R G

Experiment 8: Reinforcement Learning (Grid World with Q-Learning)

```
import numpy as np, random

size=5
goal=(4,4)
obstacles=[(1,2),(2,2),(3,1)]
actions=['up','down','left','right']
Q=np.zeros((size,size,len(actions)))

def move(pos,action):
    x,y=pos
    if action=='up': x-=1
    if action=='down': x+=1
    if action=='left': y-=1
    if action=='right': y+=1
    if 0<=x<size and 0<=y<size and (x,y) not in obstacles:
        return (x,y)
    return pos

episodes,lr,gamma,eps=300,0.1,0.9,0.2

for ep in range(episodes):
    state=(0,0)
    for step in range(100):
        a = random.randint(0,3) if random.random()<eps else np.argmax(Q[state[0],state[1]])
        next_state=move(state,actions[a])
        reward=10 if next_state==goal else -1
        Q[state[0],state[1],a]+=lr*(reward+gamma*np.max(Q[next_state[0],next_state[1]])-Q[state[0],state[1],a])
        state=next_state
        if state==goal: break

policy=[['']*size for _ in range(size)]
for i in range(size):
    for j in range(size):
        if (i,j) in obstacles: policy[i][j]='X'
        elif (i,j)==goal: policy[i][j]='G'
        else: policy[i][j]=actions[np.argmax(Q[i,j])][0].upper()

print("Learned Policy Grid:")
for row in policy: print(' '.join(row))
```

➢ Viva Questions

1. What are the main components of reinforcement learning?
2. Explain the difference between exploration and exploitation.
3. What role does the discount factor (γ\gammaγ) play in Q-learning?
4. How does the learning rate (α\alphaα) affect training?
5. Why do we use the Bellman equation in RL?
6. What is the ε-greedy strategy?
7. What are the limitations of Q-learning?
8. How can we extend this grid world to continuous environments?
9. Compare Q-learning and Deep Q-Networks (DQN).
10. Mention two real-world applications of reinforcement learning.

Resources and References

- **Textbooks**
  - Ian Goodfellow, Yoshua Bengio, and Aaron Courville – *Deep Learning*, MIT Press, 2016
  - Aurélien Géron – *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*, O'Reilly, 2022
  - François Chollet – *Deep Learning with Python*, Manning, 2021
- **Online Resources**
  - TensorFlow Documentation: https://www.tensorflow.org
  - PyTorch Documentation: https://pytorch.org
  - Keras API Guide: https://keras.io
  - scikit-learn Documentation: https://scikit-learn.org
- **Datasets**
  - UCI Machine Learning Repository: https://archive.ics.uci.edu/ml
  - Kaggle Datasets: https://www.kaggle.com/datasets
  - TensorFlow Datasets (TFDS): https://www.tensorflow.org/datasets

FAQ and Troubleshooting

Q1. My notebook shows "ModuleNotFoundError" (e.g., gensim, keras, etc.). What should I do?

👉 Install the missing library using !pip install <package-name> in Colab or Anaconda Prompt.

Q2. My GPU is not being used in Google Colab.

👉 Go to *Runtime → Change runtime type → Hardware accelerator → GPU*. Re-run the notebook.

Q3. My model accuracy is stuck at very low values.

👉 Check preprocessing (normalization, tokenization, padding). Try adjusting hyperparameters like learning rate, batch size, or number of epochs.

Q4. The loss becomes NaN during training.

👉 Lower the learning rate. Ensure there are no divide-by-zero issues in preprocessing.

Q5. I get an "out of memory" error when training.

👉 Use smaller batch sizes, reduce model complexity, or work with a subset of the dataset first.

Q6. How do I run these notebooks on my local machine?

👉 Install **Anaconda**, then create a new environment:

```
conda create -n dl_lab python=3.9
conda activate dl_lab
pip install tensorflow keras gensim matplotlib scikit-learn pandas
jupyter notebook
```