Course Code:	CSA-4002	Course Name:	Artificial Neural Networks
Course Type:	LTP	Class Number:	BL2025260100463
Credits:	4	Semester/Year:	Fall Semester 2025-26

Lab Report

Submitted by

Name: Pranjal Kumar Shukla
Reg No: 23BAI10411

Bachelor in Technology
in
CSE (AIML)

Submitted to: **Dr Praveen Kumar Tyagi Sir**

School of Computer Science and Engineering and Artificial Intelligence, VIT Bhopal University

Bhopal-Indore Highway, Kotrikalan, Sehore MP-466114, India

Date of Submission: 31st July, 2025

Contents

Experiment 1: Study of Artificial Neural Networks (ANN)

1.1 Objective:

To study and understand the fundamental concepts, architecture, components, working principles, and various types of Artificial Neural Networks (ANN). This experiment also aims to explore the significance of each component and how they interact within a neural architecture.

1.2 Theory:

An Artificial Neural Network (ANN) is a computational model inspired by the biological neural networks found in the human brain. It is designed to simulate the way humans learn, generalize, and recognize patterns. ANNs consist of interconnected processing units called neurons, which operate collectively to solve complex problems.

The structure of an ANN typically includes multiple layers of nodes: an input layer, one or more hidden layers, and an output layer. Each connection between the nodes has an associated weight, which is adjusted during training to minimize the error in predictions. The network also employs activation functions that introduce non-linearity, allowing it to learn more complex patterns.

The primary components of an ANN are as follows:

- **Neuron:** The basic processing unit that receives one or more inputs, applies weights and a bias, and then passes the result through an activation function to produce an output.
- **Weights:** Numerical parameters assigned to connections between neurons. They determine the strength and influence of a given input on the neuron's output. During training, weights are updated to reduce the error using optimization techniques like gradient descent.
- **Activation Functions:** Mathematical functions that decide whether a neuron should be activated or not. They help introduce non-linearities

into the network, enabling it to approximate complex functions. Common activation functions include:

  - **ReLU (Rectified Linear Unit):** $f(x) = \max(0, x)$
  - **Sigmoid:** $f(x) = \frac{1}{1+e^{-x}}$
  - **Tanh:** $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

- **Layers:** These are organized groups of neurons. The input layer receives the initial data, hidden layers process the data, and the output layer produces the final prediction. The number and size of hidden layers can significantly influence the performance of the network.

ANNs can be categorized into several types based on their architecture and learning strategy:

- **Feedforward Neural Network (FNN):** The simplest type of ANN where connections do not form cycles. Data flows in one direction—from input to output.
- **Convolutional Neural Network (CNN):** Specially designed for processing grid-like data such as images. CNNs use convolutional layers to detect spatial features.
- **Recurrent Neural Network (RNN):** A type of ANN where connections form cycles, allowing information to persist across time steps. RNNs are suitable for sequential data such as time series or text.
- **Radial Basis Function Network (RBFN):** Uses radial basis functions as activation functions. It is mainly used for function approximation and classification tasks.

1.3 Mathematical Formula:

The output of a single artificial neuron can be mathematically represented as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where:

- x_i: Input features or signals.
- w_i: Corresponding weights assigned to each input.
- b: Bias term, used to shift the activation function.
- f: Activation function applied to the weighted sum.

This formula reflects the core operation of a neuron, where inputs are weighted, summed, and then transformed. The output y is the neuron's response to the inputs, and it determines how strongly the neuron should activate.

1.4 Procedure:

1. Understand the structure of a single artificial neuron, including how it processes input signals to produce an output.
2. Study the role of weights and biases in controlling the influence of different inputs and shifting the decision boundary.
3. Explore different types of activation functions and their mathematical behavior, especially how they affect learning and convergence.
4. Investigate various ANN architectures such as FNN, CNN, and RNN. Focus on how each type is suited to different problem domains and data types.
5. Implement a basic neural network model to observe how inputs are transformed into outputs using code.

1.5 Code:

The following Python code implements a simple artificial neuron using the sigmoid activation function. It demonstrates how inputs, weights, and bias interact to produce an output.

```
import numpy as np
```

```
# Define the sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Inputs to the neuron
inputs = np.array([0.5, 0.3, 0.2])

# Corresponding weights for each input
weights = np.array([0.4, 0.7, 0.2])

# Bias term
bias = 0.1

# Weighted sum + bias passed through sigmoid function
output = sigmoid(np.dot(inputs, weights) + bias)

print("Output:", output)
```

This code helps visualize the computation performed by a single neuron. The ‘np.dot()‘ function calculates the weighted sum, and the result is passed through the sigmoid function to produce a non-linear output.

1.6 Result:

The result of the code execution is a numerical value between 0 and 1, representing the activated output of the neuron. This value can be interpreted as a probability or a decision signal in a classification problem.

1.7 Conclusion:

Artificial Neural Networks are a fundamental building block of modern AI systems. They are capable of learning patterns and making decisions by adjusting their internal parameters (weights and biases). Through this experiment, we understood the architecture of ANNs, including neurons, layers, and activation functions, and observed how they function together to process inputs and generate outputs. This foundational understanding is crucial for

further exploration in deep learning, computer vision, and natural language processing.

Experiment 2: Matrix Addition in ANN Computations

2.1 Objective:

To perform matrix addition, which is a foundational operation in Artificial Neural Network (ANN) computations. This includes its application in processes such as forward propagation, updating weights and biases, and combining feature maps in deep learning.

2.2 Theory:

Matrix addition is an element-wise operation where corresponding elements from two matrices are added together to produce a new matrix of the same dimensions. It is essential in linear algebra and frequently used in the computation of neural network layers. For two matrices A and B of the same shape $m \times n$, their sum C is also an $m \times n$ matrix, with each element calculated as $C_{ij} = A_{ij} + B_{ij}$.

In ANN computations, matrix addition often appears when:

- Adding the bias vector to the weighted sum during forward propagation.
- Merging feature maps in convolutional layers.
- Accumulating gradients during backpropagation.

Both operands in matrix addition must have the same shape. If they do not, the operation is undefined, and an error will occur in most programming environments.

2.3 Mathematical Formula:

The general formula for matrix addition is given by:

$$C = A + B \quad \text{where } C_{ij} = A_{ij} + B_{ij}$$

Here:

- A and B are matrices of the same dimension $m \times n$.
- C is the resulting matrix.
- i and j denote the row and column indices, respectively.

Each element C_{ij} is computed by simply adding the elements A_{ij} and B_{ij}.

2.4 Procedure:

1. Define two matrices A and B of equal dimensions using an appropriate data structure (such as a 2D array in Python).
2. Verify that both matrices have the same number of rows and columns to ensure that element-wise addition is valid.
3. Add the corresponding elements of matrix A and matrix B using the formula $C_{ij} = A_{ij} + B_{ij}$.
4. Store the result in a new matrix C, which will contain the element-wise sum of A and B.
5. Display or analyze the resulting matrix C to confirm correctness.

2.5 Code:

Below is a Python program using NumPy to perform matrix addition. This demonstrates how easily such operations can be implemented using high-level numerical libraries.

```
import numpy as np

# Define two matrices of shape 2x2
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Perform element-wise matrix addition
C = A + B
```

```
# Print the resulting matrix
print("Matrix Addition Result:\n", C)
```

This code utilizes NumPy's internal broadcasting and vectorized operations for fast and efficient matrix computation. Matrix C will hold the sum of corresponding elements from A and B.

2.6 Result:

After executing the code, the resulting matrix from the addition operation is:

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Each value in the result is obtained by adding the corresponding elements in matrices A and B:

$$C_{11} = 1 + 5 = 6, \quad C_{12} = 2 + 6 = 8, \quad C_{21} = 3 + 7 = 10, \quad C_{22} = 4 + 8 = 12$$

2.7 Conclusion:

Matrix addition is a fundamental yet crucial operation in ANN computations. It is often used during forward propagation when biases are added to weighted sums, and during training when multiple gradients are combined. Understanding how matrix addition works helps build a solid foundation for more complex neural network operations involving linear algebra. Mastery of such operations is essential for effectively implementing and debugging deep learning algorithms.

Experiment 3: Matrix Multiplication in ANN Computations

3.1 Objective:

To perform matrix multiplication, a crucial operation in Artificial Neural Network (ANN) computations. Matrix multiplication is extensively used in processes such as forward propagation, backpropagation, and updating weight matrices during the training of neural networks.

3.2 Theory:

Matrix multiplication, also known as the dot product of matrices, involves the combination of rows from the first matrix with columns of the second matrix. The resulting matrix represents a transformation of input data via the weight matrix, a process that occurs repeatedly in neural networks during computations between layers.

For two matrices A and B, where the number of columns in A equals the number of rows in B, their product $C = A \cdot B$ results in a new matrix C. Each element C_{ij} of this matrix is computed by taking the dot product of the i-th row of A and the j-th column of B.

Matrix multiplication plays a central role in:

- Forward propagation: calculating the weighted sum of inputs.
- Representing transformations from one layer to the next.
- Gradient computations during backpropagation.
- Efficient implementation of deep learning layers (especially with large batches of data).

3.3 Mathematical Formula:

The general formula for matrix multiplication is given by:

$$C = A \cdot B \quad \text{where } C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Here:

- A is an $m \times n$ matrix.
- B is an $n \times p$ matrix.
- C is the resulting $m \times p$ matrix.
- i: row index from matrix A
- j: column index from matrix B
- k: summation index over the shared dimension.

This means that to compute each element C_{ij}, we take the dot product of the i-th row from A and the j-th column from B.

3.4 Procedure:

1. Define two matrices A and B such that the number of columns in matrix A is equal to the number of rows in matrix B. This condition is necessary for matrix multiplication to be valid.
2. For each element C_{ij} in the resulting matrix C, compute the sum of products between the corresponding elements of the i-th row of A and the j-th column of B.
3. Store each computed value in its respective position C_{ij} in the result matrix.
4. Repeat this for all valid combinations of rows from A and columns from B to complete the matrix C.
5. Display or verify the resulting matrix to ensure correctness.

3.5 Code:

The following Python code demonstrates matrix multiplication using the NumPy library. NumPy's `dot()` function simplifies the operation and is optimized for performance.

```
import numpy as np

# Define two matrices of compatible dimensions
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Perform matrix multiplication
C = np.dot(A, B)

# Print the result
print("Matrix Multiplication Result:\n", C)
```

In this example, matrix A has dimensions 2×2, and matrix B also has dimensions 2×2. The result matrix C will be 2×2, with each element computed as described in the mathematical formula.

3.6 Result:

After executing the above code, the resulting matrix C is:

$$\begin{bmatrix} (1 \cdot 5 + 2 \cdot 7) & (1 \cdot 6 + 2 \cdot 8) \\ (3 \cdot 5 + 4 \cdot 7) & (3 \cdot 6 + 4 \cdot 8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

This result is obtained by computing the dot product of rows from matrix A and columns from matrix B, as per the rules of matrix multiplication.

3.7 Conclusion:

Matrix multiplication is a core computation in neural networks, used in every layer to calculate the weighted input to neurons. It enables the transformation of input data into higher-level features and is central to the training

and inference processes of deep learning models. Mastering matrix multiplication is essential for understanding how ANNs function internally and for implementing custom network layers or optimization routines.

Experiment 4: Matrix Transposition in ANN Computations

4.1 Objective:

To perform matrix transposition, which is a fundamental operation in linear algebra and plays an important role in neural network training, especially in the context of weight manipulation, backpropagation, and optimization techniques.

4.2 Theory:

Matrix transposition is the process of converting a matrix by flipping it over its main diagonal. This operation turns rows into columns and columns into rows. If a matrix A has dimensions $m \times n$, its transpose A^T will have dimensions $n \times m$.

In Artificial Neural Networks (ANNs), matrix transposition is commonly required when:

- Aligning matrix dimensions for valid multiplications.
- Transferring gradients during backpropagation.
- Implementing weight updates when input and output vectors must be reorganized.

For example, during backpropagation, gradients of the loss with respect to weights often involve the transpose of the input matrix to ensure dimension compatibility during matrix multiplication.

4.3 Mathematical Formula:

The formula for transposing a matrix is:

$$A_{ij}^T = A_{ji}$$

This implies that the element at the i-th row and j-th column of the original matrix A will become the element at the j-th row and i-th column in the transposed matrix A^T.

4.4 Procedure:

1. Define a matrix A with dimensions $m \times n$.
2. Swap the rows with the columns such that the element at position (i, j) becomes positioned at (j, i).
3. Store the result in a new matrix A^T, which will have dimensions $n \times m$.
4. Display the transposed matrix to verify the correctness of the operation.

4.5 Code:

The following Python code uses NumPy to transpose a matrix. NumPy makes it easy to apply transposition using the '.T' attribute.

```
import numpy as np

# Define a 2x3 matrix
A = np.array([[1, 2, 3], [4, 5, 6]])

# Perform matrix transposition
A_T = A.T

# Display the transposed matrix
print("Transposed Matrix:\n", A_T)
```

Here, matrix A has dimensions 2×3, and the transposed matrix A^T will have dimensions 3×2. The 'np.array().T' command effectively and efficiently performs the transposition.

4.6 Result:

The transposed matrix A^T is:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

This result shows that each row in the original matrix becomes a column in the transposed matrix, and vice versa.

4.7 Conclusion:

Matrix transposition is a simple yet powerful operation that is frequently utilized in ANN computations. It is particularly useful in weight updates, reshaping data for matrix multiplication, and ensuring dimensional compatibility in learning algorithms. A clear understanding of transposition helps in both implementing neural network operations and debugging issues related to shape mismatches in matrix-based computations.