Artificial Neural Network Lab

Course Code: PE/CSE/84-P

Artificial Neural Network Lab

General Course Information

Course Code: PE/CSE/84-P CourseCredits:1 Type: Professional Core Lab. Course Mode: Lab practice and assignments ContactHours:2 hours/ week	**Course Assessment Methods (internal: 50; external: 50)** The internal and external assessment is based on the level of participation in lab. Sessions and the timely submission of lab experiments/assignments, the quality of solutions designed for the assignments, the performance in VIVA-VOCE, the quality of lab file and ethical practices followed. The internal examination is conducted by the course coordinator. The external examination is conducted by external examiner appointed by the Controller of Examination in association with the internal examiner appointed by the Chairperson of the Department.

Pre-requisites: Knowledge of Artificial Intelligence

About the Course:

In this Course, students learn to design, generate, minimize, and prioritize test cases of a software application using programming language or with the help of software testing tools. The lab experiments involve designing testing datasets by taking case studies and applying software testing techniques on these datasets. The course has a special focus on understanding and implementation of test results of software testing techniques to improve software quality.

Course Outcomes: By the end of the course students will be able to:

CO1. **Implement** different neural network models.(LOTS:Level3:Apply)
CO2. **Apply** optimization techniques. (LOTS: Level 3: Apply)
CO3. **Interpret** the results of various problems solved through neural network. (LOTS: Level 4: Analyze)
CO4. **Plan** patterns to take activities. (LOTS: Level6: Create)
CO5. **Prepare** lab reports for ANN assignments. (LOTS: Level6: Create)
CO6. **Demonstrate** use of ethical practices, self-learning and team spirit. (LOTS: Level3: Apply)

List of experiments/assignments

1. Parallel and distributed processing - I: Interactive activation and competition models
2. Parallel and distributed processing - II: Constraint satisfaction neural network models
3. Perceptron learning
4. Multi layer feed forward neural networks
5. Hopfield model for pattern storage task
6. Hopfield model with stochastic update
7. Competitive learning neural networks for pattern clustering
8. Solution to travelling salesman problem using self organizing maps

9. Solution to optimization problems using Hopfield models.
10. Weighted matching problem: Deterministic, stochastic and mean-field annealing of an Hopfield model

Note:

The actual experiments/assignments will be designed by the course coordinator. One assignment should be designed to be done in groups of two or three students. The assignments must meet the objective of the course and the levels of the given course outcomes. The list of assignments and schedule of submission will be prepared by the course coordinator at the beginning of the semester.

CO-PO Articulation Matrix Artificial Neural Network Lab. Course (PE/CSE/84-P)

Course Outcomes	**PO 1**	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**	**PSO 13**	**PSO 14**	**PSO 15**
CO1	2	2	2	2	3	_	_	_	_	_	_	_	3	_	_
CO2	2	2	2	2	3	_	_	_	_	_	_	_	3	_	_
CO3	3	2	3	3		_	_	_	_	_	_	_	3	_	_
CO4	3	3	3	3	3	_	_	_	_	_	3	_	3	_	_
CO5	2	2	2	2	3	_	_	_	_	_	_	_	3	_	_
CO6	3	2	3	3		_	_	_	_	_	_	_	3	_	_
3-High2-Medium1-Low															

Practical 1: Perceptron Learning Algorithm

Aim

To implement the Perceptron Learning Algorithm for a binary classification problem.

Theory

The perceptron is the simplest type of artificial neural network used for classification. It works on linearly separable data and updates weights using an error correction rule.

Algorithm

1. Initialize weights and bias to zero
2. For each input:
    - Calculate output using weighted sum
    - Apply step activation function
3. Update weights using error
4. Repeat until convergence

Program

```
import numpy as np

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([0,0,0,1])

w = np.zeros(2)
b = 0
lr = 0.1

for epoch in range(10):
    for i in range(len(X)):
        linear = np.dot(X[i], w) + b
        y_pred = 1 if linear > 0 else 0

        error = y[i] - y_pred
        w += lr * error * X[i]
        b += lr * error

print(w, b)
```

Practical 2: Multi-Layer Feed Forward Network

Aim

To implement a multi-layer feed-forward neural network.

Theory

A feed-forward neural network consists of input, hidden, and output layers. It uses activation functions like sigmoid to introduce non-linearity.

Algorithm

1. Initialize weights randomly
2. Perform forward propagation
3. Compute output

Program

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

X = np.array([[0,0],[0,1],[1,0],[1,1]])

W1 = np.random.rand(2,2)
W2 = np.random.rand(2,1)

hidden = sigmoid(np.dot(X, W1))
output = sigmoid(np.dot(hidden, W2))

print(output)
```

Practical 3: Backpropagation Algorithm

Aim

To train a neural network using backpropagation.

Theory

Backpropagation is used to minimize error by adjusting weights using gradient descent.

Algorithm

1. Initialize weights
2. Forward propagate
3. Compute error
4. Backpropagate error
5. Update weights

Program

```
import numpy as np

def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_derivative(x):
    return x*(1-x)

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

W1 = np.random.rand(2,2)
W2 = np.random.rand(2,1)

lr = 0.1

for i in range(10000):
    hidden = sigmoid(np.dot(X, W1))
    output = sigmoid(np.dot(hidden, W2))

    error = y - output

    d_output = error * sigmoid_derivative(output)
    d_hidden = d_output.dot(W2.T) * sigmoid_derivative(hidden)

    W2 += hidden.T.dot(d_output) * lr
    W1 += X.T.dot(d_hidden) * lr
print(output)
```

Practical 4: Hopfield Network

Aim

To store and recall patterns using Hopfield Network.

Theory

Hopfield network is a recurrent neural network used for associative memory.

Algorithm

1. Store patterns in weight matrix
2. Initialize test pattern
3. Update until stable

Program

```
import numpy as np

patterns = np.array([[1, -1, 1], [-1, 1, -1]])

W = np.dot(patterns.T, patterns)
np.fill_diagonal(W, 0)

test = np.array([1, -1, 1])

for i in range(5):
    test = np.sign(np.dot(W, test))

print(test)
```

Practical 5: Competitive Learning

Aim

To implement competitive learning for clustering.

Theory

Neurons compete, and only the winner neuron gets updated.

Algorithm

1. Initialize weights
2. Compute distance
3. Select winner neuron
4. Update winner weights

Program

```
import numpy as np

data = np.array([[1,2],[1,4],[5,6],[6,8]])

weights = np.random.rand(2,2)

lr = 0.5

for epoch in range(10):
    for x in data:
        distances = np.linalg.norm(weights - x, axis=1)
        winner = np.argmin(distances)
        weights[winner] += lr * (x - weights[winner])

print(weights)
```

Practical 6: Self-Organizing Map (SOM)

Aim

To implement SOM for clustering.

Theory

SOM is an unsupervised learning model used for dimensionality reduction.

Algorithm

1. Initialize weights
2. Find Best Matching Unit
3. Update weights

Program

```
import numpy as np

data = np.random.rand(100, 2)
weights = np.random.rand(3,2)

lr = 0.5

for epoch in range(20):
    for x in data:
        d = np.linalg.norm(weights - x, axis=1)
        winner = np.argmin(d)
        weights[winner] += lr * (x - weights[winner])

print(weights)
```

Practical 7: Optimization using Hopfield Network

Aim

To solve optimization problems using Hopfield Network.

Theory

Hopfield network minimizes energy function to reach optimal solution.

Algorithm

1. Initialize weights and state
2. Update state using weight matrix
3. Repeat until stable

Program

```
import numpy as np

W = np.array([[0, -1], [-1, 0]])
state = np.array([1, -1])

for _ in range(5):
    state = np.sign(np.dot(W, state))

print(state)
```

Program 8. Single Neuron Model

```
import numpy as np
x = np.array([1,2,3])
w=np.array([0.5,0.5,0.5]) b = 1
output=np.dot(x,w)+b print("Output:", output)
```

Program 9. **Sigmoid Activation Function**

```
import numpy as np
def sigmoid(x):
return 1/(1+np.exp(-x))
x=np.array([-1,0,1]) print(sigmoid(x))
```

Program 10. **Neural Network using Keras**

```
From tensorflow.keras.modelsimportSequential from tensorflow.keras.layers import
Dense
model = Sequential()
model.add(Dense(4,input_dim=2,activation='relu')) model.add(Dense(1,
activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
 print(model.summary())
```

Program 11. XOR using Neural Network

```
import numpy as np
fromtensorflow.keras.modelsimportSequential from tensorflow.keras.layers import Dense
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([0,1,1,0])
model = Sequential()
model.add(Dense(2,input_dim=2,activation='relu')) model.add(Dense(1,
activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
model.fit(X,y,epochs=100,verbose=0)
print(model.predict(X))
```

Program 12. **ReLU Activation**

```
import numpy as np
def relu(x):
return np.maximum(0,x) print(relu(np.array([-2,0,3])))
```

Program 13. **Binary Classification**

```
from sklearn.neural_network import MLPClassifier
X = [[0,0],[0,1],[1,0],[1,1]]
y = [0,1,1,0]
clf=MLPClassifier(hidden_layer_sizes=(2,)) clf.fit(X,y)
print(clf.predict(X))
```