TRIBHUVAN UNIVERSITY

LUMBINI ICT CAMPUS

Gaindakot-4, Nawalpur

Lab report on:

Artificial Intelligence

Submitted by:

Name: Bipin Paudel
Faculty: BCA
Semester: VII

Submitted to:

Er. Bibek Chalise

Table of Contents

Implementation of PROLOG

Introduction

Prolog (Programming in Logic) is a general-purpose programming language that is based on the principles of logic programming. It is widely used for tasks such as natural language processing, automated reasoning, and symbolic computation.

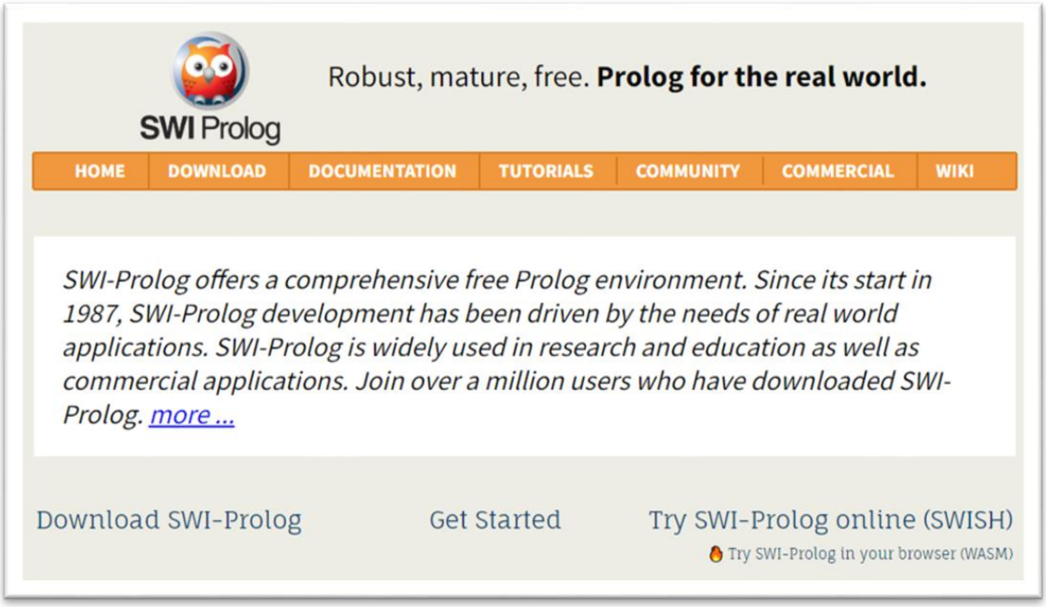
Prolog has a declarative programming style, which means that you specify what you want the program to do, rather than how to do it. The Prolog interpreter then uses a set of logical rules and facts to deduce how to achieve the desired result. This approach is different from most imperative programming languages, which require you to specify how the program should execute each step-in detail.

Prolog is particularly well-suited for tasks that involve searching for solutions to complex problems or queries, such as solving puzzles or answering questions based on a set of facts. It is also often used in artificial intelligence and expert systems applications.

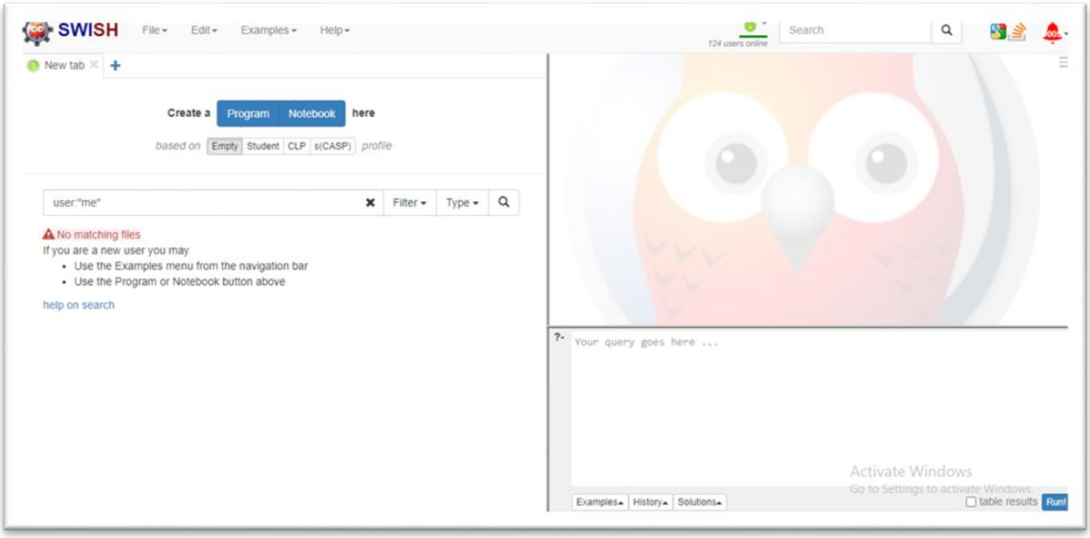
Prolog has a syntax that is based on logical statements and operators, and it includes built-in support for data types such as atoms, numbers, and lists. It also includes a range of built-in predicates and control structures that can be used to manipulate and reason about data.

Prolog Code Editor:

1. Go to Offical Website of Prolog https://www.swi-prolog.org/



2. Click to the Try swi-prolog online. Which will redirect to prolog web editor page.



3. Click on create a program and it will show textarea to write prolog code of facts and rules. The query code should be written in bottom right text editor.

Demonstration of Fact Table

Facts

A fact is a statement that represents a piece of information or a relationship between different pieces of information. Facts are written in the form of logical statements and are stored in a Prolog program or database.

For example, a fact might be "John is a father," which represents the relationship between the person "John" and the role "father”. In Prolog syntax, this fact would be written as: father(john).

Rules

A rule is a logical statement that defines a relationship or pattern between different facts or variables. Rules are written in the form of a head and a body, separated by a colon. The head is a logical statement that represents the conclusion or result of the rule, and the body is a list of logical statements that represent the conditions or premises that must be satisfied in order for the rule to be applied.

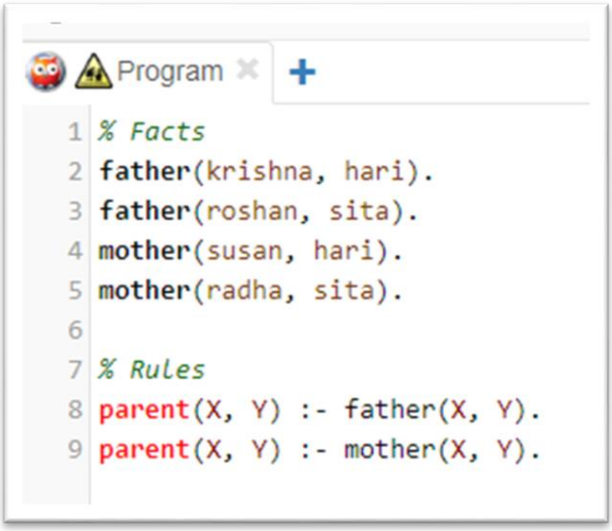
For example, a rule might be "if X is a parent and Y is a child of X, then X is a mother or father of Y." In Prolog syntax, this rule would be written as:

parent_of(X, Y) :- mother(X, Y).

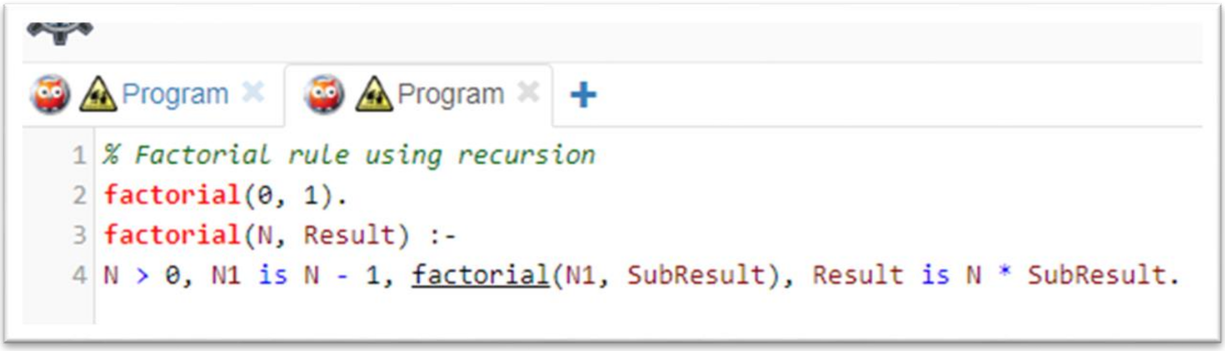
parent_of(X, Y) :- father(X, Y).

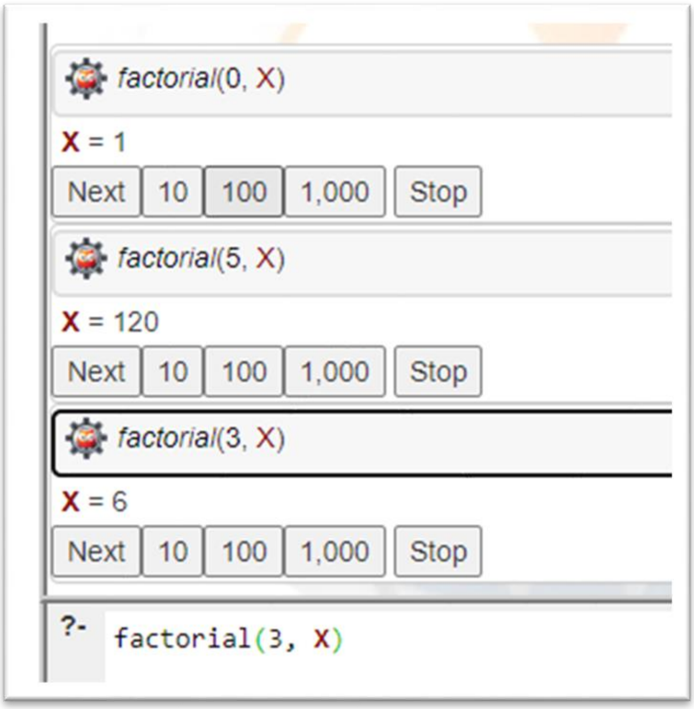
Facts and rules are used together in Prolog programs to represent and reason about knowledge. Prolog's built-in search and unification algorithms can be used to query and manipulate the facts and rules in a program to find solutions to problems or answer questions.

Example 1 (Family Relationship) :



Example 2 (Factorial using recursion):





Example 3 (Family Relationships) :

```
Program  Program  Program  +
% Facts
father(john, jim).
father(bob, ann).
father(jim, sarah).
mother(susan, jim).
mother(lisa, ann).
mother(ann, sarah).

% Rules
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

```
grandparent(john, sarah).
true
Next 10 100 1,000 Stop
grandparent(bob, sarah).
true
Next 10 100 1,000 Stop
grandparent(jim, sarah).
false
grandparent(john, ann).
false
?- grandparent(john, ann).
```

Example 4 (List Operations) :

```
Program  Program
% Rules for list operations
head([X|_], X).
tail([_,_|X], X).
isEmpty([]).
```

```
head([1, 2, 3], X).
X = 1
tail([1, 2, 3], Tail).
Tail = [3]
isEmpty([]).
true
isEmpty([1, 2, 3]).
false
?- isEmpty([1, 2, 3]).
```

Implementation of Breadth-first search algorithm.

Theory:

The Breadth First Search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

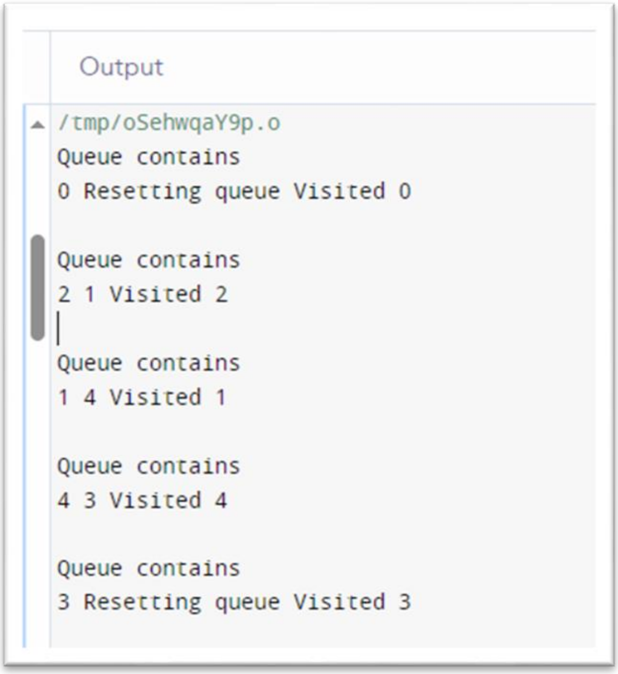
Code:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue {
int items[SIZE];
int front;
int rear;
};
struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);
struct node {
int vertex;
struct node* next;
};
struct node* createNode(int);
struct Graph {
int numVertices;
struct node** adjLists;
int* visited;
};
// BFS algorithm
void bfs(struct Graph* graph, int
startVertex) {
struct queue* q = createQueue();
graph->visited[startVertex] = 1;
enqueue(q, startVertex);
while (!isEmpty(q)) {
printQueue(q);
int currentVertex = dequeue(q);
printf("Visited %d\n", currentVertex);
struct node* temp = graph-
>adjLists[currentVertex];
while (temp) {
int adjVertex = temp->vertex;
if (graph->visited[adjVertex] == 0) {
graph->visited[adjVertex] = 1;
enqueue(q, adjVertex);
}
temp = temp->next;
}
}
}
// Creating a node
struct node* createNode(int v) {
struct node* newNode =
malloc(sizeof(struct node));
newNode->vertex = v;
newNode->next = NULL;
return newNode;
```

```
}
// Creating a graph
struct Graph* createGraph(int vertices) {
struct Graph* graph = malloc(sizeof(struct
Graph));
graph->numVertices = vertices;
graph->adjLists = malloc(vertices *
sizeof(struct node*));
graph->visited = malloc(vertices *
sizeof(int));
int i;
for (i = 0; i < vertices; i++) {
graph->adjLists[i] = NULL;
graph->visited[i] = 0;
}
return graph;
}
// Add edge
void addEdge(struct Graph* graph, int src,
int dest) {
// Add edge from src to dest
struct node* newNode = createNode(dest);
newNode->next = graph->adjLists[src];
graph->adjLists[src] = newNode;
// Add edge from dest to src
newNode = createNode(src);
newNode->next = graph->adjLists[dest];
graph->adjLists[dest] = newNode;
}
// Create a queue
struct queue* createQueue() {
struct queue* q = malloc(sizeof(struct
queue));
q->front = -1;
q->rear = -1;
return q;
}
// Check if the queue is empty
int isEmpty(struct queue* q) {
if (q->rear == -1)
return 1;
else
return 0;
}
// Adding elements into queue
void enqueue(struct queue* q, int value) {
if (q->rear == SIZE - 1)
printf("\nQueue is Full!!");
else {
if (q->front == -1)
q->front = 0;
q->rear++;
q->items[q->rear] = value;
}
}
// Removing elements from queue
int dequeue(struct queue* q) {
int item;
if (isEmpty(q)) {
printf("Queue is empty");
item = -1;
} else {
item = q->items[q->front];
q->front++;
```

```
if (q->front > q->rear) {
printf("Resetting queue ");
q->front = q->rear = -1;
}
}
return item;
}
// Print the queue
void printQueue(struct queue* q) {
int i = q->front;
if (isEmpty(q)) {
printf("Queue is empty");
} else {
printf("\nQueue contains \n");
for (i = q->front; i < q->rear + 1; i++) {
printf("%d ", q->items[i]);
}
}
}
int main() {
struct Graph* graph = createGraph(6);
addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 1, 2);
addEdge(graph, 1, 4);
addEdge(graph, 1, 3);
addEdge(graph, 2, 4);
addEdge(graph, 3, 4);
bfs(graph, 0);
return 0;
}
```

Output:



Implementation of Depth-first search algorithm.

Theory:

DFS (Depth-first search) is a technique used for traversing trees or graphs. Here backtracking is used for traversal. In this traversal first, the deepest node is visited and then backtracks to its parent node if no sibling of that node exists

Code:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int vertex;
   struct node* next;
};
struct Graph {
   int numVertices;
   int* visited;
   struct node** adjLists; // Corrected syntax
here
};
void DFS(struct Graph* graph, int vertex);
struct node* createNode(int v);
struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int
dest);
void printGraph(struct Graph* graph);
// DFS algorithm
void DFS(struct Graph* graph, int vertex) {
   struct node* adjList = graph-
>adjLists[vertex];
   struct node* temp = adjList;
   graph->visited[vertex] = 1;
   printf("Visited %d \n", vertex);
   while (temp != NULL) {
      int connectedVertex = temp->vertex;
      if (graph->visited[connectedVertex] ==
0) {
         DFS(graph, connectedVertex);
      }
      temp = temp->next;
   }
}
// Create a node
struct node* createNode(int v) {
   struct node* newNode =
malloc(sizeof(struct node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}
// Create graph
struct Graph* createGraph(int vertices) {
   struct Graph* graph = malloc(sizeof(struct
Graph));
   graph->numVertices = vertices;
   graph->adjLists = malloc(vertices *
sizeof(struct node*));
   graph->visited = malloc(vertices *
sizeof(int));
   int i;
   for (i = 0; i < vertices; i++) {
      graph->adjLists[i] = NULL;
      graph->visited[i] = 0;
```

```
    }
    return graph;
}
// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
// Print the graph
void printGraph(struct Graph* graph) {
    int v;
    for (v = 0; v < graph->numVertices; v++) {
        struct node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
```

```
int main() {
    struct Graph* graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    printGraph(graph);
    DFS(graph, 2);
    return 0;
}
```

Output

```
Output
/tmp/oSehwqaY9p.o
Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```

Implemention of Machine Learning

Machine Learning

Machine learning is a subset of artificial intelligence that gives systems the ability to learn and optimize processes without having to be consistently programmed. Simply put, machine learning uses data, statistics and trial and error to "learn" a specific task without ever having to be specifically coded for the task.

1. **Supervised Learning**: Learns from labeled data to make predictions.
2. **Unsupervised Learning:** Finds patterns in unlabeled data.
3. **Reinforcement Learning:** Learns by interacting with an environment and receiving rewards or penalties.

Machine learning is used in various fields to make computers improve their performance on tasks by learning from data.

Supervised Learning:

Supervised learning is a category of machine learning that uses labeled datasets to train algorithms to predict outcomes and recognize patterns.

Types of supervised learning

Supervised learning in machine learning is generally divided into two categories: classification and regression.

Classification

Classification algorithms are used to group data by predicting a categorical label or output variable based on the input data. Classification is used when output variables are categorical, meaning there are two or more classes.

Program 1: Classification Algorithim using Random Forest Classifier

```
# Import necessary libraries

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score, classification_report

from sklearn.datasets import make_classification

# Generate dummy data

X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,
n_clusters_per_class=2, random_state=42)

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a RandomForestClassifier

classifier = RandomForestClassifier(random_state=42)
```

```
# Train the classifier on the training data
classifier.fit(X_train, y_train)
# Make predictions on the test data
y_pred = classifier.predict(X_test)
# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
# Print the results
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_rep)
```

Output:

```
Accuracy: 0.92
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.89      0.92       104
           1       0.89      0.95      0.92        96

    accuracy                           0.92       200
   macro avg       0.92      0.92      0.92       200
weighted avg       0.92      0.92      0.92       200
```

Program 2: New Data Input For Classification

```
# Assuming 'new_data' is your new input data for classification
new_data = X_test[:5]  # Example: Use the first 5 samples from the test set
# Make predictions using the trained classifier
classification_predictions = classifier.predict(new_data)
# Print the classification predictions
print("Classification Predictions:", classification_predictions)
```

Output:

```
Classification Predictions: [1 0 0 0 0]
```

Regression

Regression algorithms are used to predict a real or continuous value, where the algorithm detects a relationship between two or more variables.

A common example of a regression task might be predicting a salary based on work experience. For instance, a supervised learning algorithm would be fed inputs related to work experience (e.g., length of time, the industry or field, location, etc.) and the corresponding assigned salary amount. After the model is trained, it could be used to predict the average salary based on work experience.

Program 3: Regression Algorithm using Random Forest Regressor

```
# Import necessary libraries

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean_squared_error

from sklearn.datasets import make_regression

# Generate dummy data for regression

X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=42)

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a RandomForestRegressor

regressor = RandomForestRegressor(random_state=42)

# Train the regressor on the training data

regressor.fit(X_train, y_train)

# Make predictions on the test data

y_pred = regressor.predict(X_test)

# Evaluate the regressor

mse = mean_squared_error(y_test, y_pred)

# Print the results

print("Mean Squared Error:", mse)
```

Output:

```
Mean Squared Error: 7055.507694741972
```

Program 4: New Data Input for Regression

```
# Assuming 'new_data' is your new input data for regression
new_data = X_test[:5]  # Example: Use the first 5 samples from the test set
# Make predictions using the trained regressor
regression_predictions = regressor.predict(new_data)
# Print the regression predictions
print("Regression Predictions:", regression_predictions)
```

Output:

```
Regression Predictions: [-254.10089711 -223.93887587  259.52388486  -28.86765312  170.17294842]
```

Implemtation of Neutral Networks

Neutrol Network

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. Neural networks can help computers make intelligent decisions with limited human assistance. This is because they can learn and model the relationships between input and output data that are nonlinear and complex.

Artifical Neural Network

An Artificial Neural Network (ANN) is a computational model inspired by the human brain's neural structure. It consists of interconnected nodes (neurons) organized into layers. Information flows through these nodes, and the network adjusts the connection strengths (weights) during training to learn from data, enabling it to recognize patterns, make predictions, and solve various tasks in machine learning and artificial intelligence.

Program 1: Simple ANN Program

```
# Import necessary libraries

import numpy as np

import tensorflow as tf

from tensorflow import keras

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.datasets import make_classification

# Generate dummy data for classification

X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,
n_clusters_per_class=2, random_state=42)

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the data

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)

# Build the ANN model

model = keras.Sequential([

  keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
```

```
    keras.layers.Dense(1, activation='sigmoid')
])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, y_test)
print("Test Accuracy:", accuracy)
# Make predictions on new data
new_data = X_test[:5]
predictions = model.predict(new_data)
print("Predictions for the first 5 samples:", predictions)
```

Output:

```
Epoch 1/10
20/20 [==============================] - 1s 11ms/step - loss: 0.7408 - accuracy: 0.5219 - val_loss: 0.7020 - val_accuracy: 0.6062
Epoch 2/10
20/20 [==============================] - 0s 4ms/step - loss: 0.6302 - accuracy: 0.6359 - val_loss: 0.6051 - val_accuracy: 0.6625
Epoch 3/10
20/20 [==============================] - 0s 4ms/step - loss: 0.5510 - accuracy: 0.7344 - val_loss: 0.5333 - val_accuracy: 0.7437
Epoch 4/10
20/20 [==============================] - 0s 3ms/step - loss: 0.4888 - accuracy: 0.8094 - val_loss: 0.4774 - val_accuracy: 0.8313
Epoch 5/10
20/20 [==============================] - 0s 3ms/step - loss: 0.4380 - accuracy: 0.8531 - val_loss: 0.4309 - val_accuracy: 0.8687
Epoch 6/10
20/20 [==============================] - 0s 3ms/step - loss: 0.3963 - accuracy: 0.8828 - val_loss: 0.3917 - val_accuracy: 0.8813
Epoch 7/10
20/20 [==============================] - 0s 4ms/step - loss: 0.3620 - accuracy: 0.8922 - val_loss: 0.3605 - val_accuracy: 0.9000
Epoch 8/10
20/20 [==============================] - 0s 3ms/step - loss: 0.3339 - accuracy: 0.9016 - val_loss: 0.3344 - val_accuracy: 0.9062
Epoch 9/10
20/20 [==============================] - 0s 3ms/step - loss: 0.3109 - accuracy: 0.9047 - val_loss: 0.3136 - val_accuracy: 0.9062
Epoch 10/10
20/20 [==============================] - 0s 4ms/step - loss: 0.2912 - accuracy: 0.9016 - val_loss: 0.2929 - val_accuracy: 0.9125
7/7 [==============================] - 0s 2ms/step - loss: 0.3293 - accuracy: 0.8900
Test Accuracy: 0.8899999856948853
1/1 [==============================] - 0s 57ms/step
Predictions for the first 5 samples: [[0.952115  ]
 [0.02607499]
 [0.06479347]
 [0.34940735]
 [0.10493085]]
```

Program 2: Program of ANN with Hidden Layers

```
# Import necessary libraries
import numpy as np
import tensorflow as tf
```

```python
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification
# Generate dummy data for classification
X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,
n_clusters_per_class=2, random_state=42)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Build a more advanced ANN model
model = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
    keras.layers.Dropout(0.5),  # Dropout layer for regularization
    keras.layers.Dense(64, activation='relu'),
    keras.layers.BatchNormalization(),  # Batch normalization for better convergence
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dropout(0.3),  # Another dropout layer
    keras.layers.Dense(1, activation='sigmoid')
])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model
model.fit(X_train, y_train, epochs=20, batch_size=64, validation_split=0.2)
# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, y_test)
print("Test Accuracy:", accuracy)
# Make predictions on new data
```

new_data = X_test[:5]

 # Apply a threshold to convert probabilities to binary predictions

binary_predictions = (predictions > 0.5).astype(int)

print("Binary Predictions for the first 5 samples:", binary_predictions)

Output:

```
Epoch 1/20
10/10 [==============================] - 3s 75ms/step - loss: 0.7301 - accuracy: 0.5578 - val_loss: 0.6486 - val_accuracy: 0.6313
Epoch 2/20
10/10 [==============================] - 0s 25ms/step - loss: 0.6361 - accuracy: 0.6344 - val_loss: 0.6014 - val_accuracy: 0.7563
Epoch 3/20
10/10 [==============================] - 0s 20ms/step - loss: 0.5562 - accuracy: 0.7063 - val_loss: 0.5576 - val_accuracy: 0.8188
Epoch 4/20
10/10 [==============================] - 0s 9ms/step - loss: 0.5335 - accuracy: 0.7391 - val_loss: 0.5217 - val_accuracy: 0.8375
Epoch 5/20
10/10 [==============================] - 0s 9ms/step - loss: 0.4934 - accuracy: 0.7688 - val_loss: 0.4858 - val_accuracy: 0.8625
Epoch 6/20
10/10 [==============================] - 0s 8ms/step - loss: 0.4556 - accuracy: 0.7859 - val_loss: 0.4484 - val_accuracy: 0.8813
Epoch 7/20
10/10 [==============================] - 0s 7ms/step - loss: 0.4331 - accuracy: 0.8016 - val_loss: 0.4186 - val_accuracy: 0.8813
Epoch 8/20
10/10 [==============================] - 0s 8ms/step - loss: 0.3952 - accuracy: 0.8219 - val_loss: 0.3908 - val_accuracy: 0.8938
Epoch 9/20
10/10 [==============================] - 0s 11ms/step - loss: 0.3716 - accuracy: 0.8344 - val_loss: 0.3635 - val_accuracy: 0.9062
Epoch 10/20
10/10 [==============================] - 0s 9ms/step - loss: 0.3554 - accuracy: 0.8531 - val_loss: 0.3415 - val_accuracy: 0.9125
Epoch 11/20
10/10 [==============================] - 0s 9ms/step - loss: 0.3353 - accuracy: 0.8594 - val_loss: 0.3191 - val_accuracy: 0.9187
Epoch 12/20
10/10 [==============================] - 0s 9ms/step - loss: 0.3093 - accuracy: 0.8609 - val_loss: 0.2946 - val_accuracy: 0.9312
Epoch 13/20
10/10 [==============================] - 0s 9ms/step - loss: 0.2837 - accuracy: 0.8891 - val_loss: 0.2693 - val_accuracy: 0.9312
Epoch 14/20
10/10 [==============================] - 0s 10ms/step - loss: 0.3055 - accuracy: 0.8797 - val_loss: 0.2513 - val_accuracy: 0.9375
Epoch 15/20
10/10 [==============================] - 0s 9ms/step - loss: 0.2671 - accuracy: 0.8828 - val_loss: 0.2383 - val_accuracy: 0.9375
Epoch 16/20
10/10 [==============================] - 0s 8ms/step - loss: 0.2587 - accuracy: 0.8813 - val_loss: 0.2284 - val_accuracy: 0.9375
Epoch 17/20
10/10 [==============================] - 0s 8ms/step - loss: 0.2519 - accuracy: 0.8953 - val_loss: 0.2173 - val_accuracy: 0.9375
Epoch 18/20
10/10 [==============================] - 0s 9ms/step - loss: 0.2844 - accuracy: 0.8922 - val_loss: 0.2097 - val_accuracy: 0.9500
Epoch 19/20
10/10 [==============================] - 0s 9ms/step - loss: 0.2491 - accuracy: 0.9047 - val_loss: 0.2034 - val_accuracy: 0.9438
Epoch 20/20
10/10 [==============================] - 0s 9ms/step - loss: 0.2446 - accuracy: 0.8875 - val_loss: 0.1927 - val_accuracy: 0.9375
7/7 [==============================] - 0s 2ms/step - loss: 0.2147 - accuracy: 0.9350
Test Accuracy: 0.9350000023841858
Binary Predictions for the first 5 samples: [[1]
 [0]
 [0]
 [0]
```