**Experiment No: 01 Calculate the output of a simple neuron**

import numpy as np

import pandas as pd

import random

lr = 1 #learning rate

bias = 1 #value of bias

weights = [random.random(),random.random(),random.random()]

#weights generated in a list (3 weights in total for 2 neurons and the bias)

lr is the learning rate, which determines the step size at each iteration when adjusting the weights.

bias is the bias term, a constant added to the weighted sum of inputs.

weights is a list containing three random initial weights. The first two weights correspond to the inputs, and the third weight corresponds to the bias.

Perceptron is a function that takes two inputs (input1 and input2) and a target output (output).

outputP is the weighted sum of inputs plus the bias.

def Perceptron(input1, input2, output) :

outputP = input1\*weights[0]+input2\*weights[1]+bias\*weights[2]

if outputP > 0 : #activation function (here Heaviside)

outputP = 1

print(outputP)

else :

outputP = 0

error = output-outputP

weights[0] += error \* input1 \* lr

weights[1] += error \* input2 \* lr

weights[2] += error \* bias \* lr

outputP=1/(1+numpy.exp(-outputP))

print(outputP)

The activation function used here is a simple step function (Heaviside step function). If the weighted sum is greater than 0, the output is set to 1; otherwise, it is set to 0.

If the predicted output does not match the target output, weights are adjusted using the perceptron learning rule.

for i in range(50) :

Perceptron(1,1,1) #True or true

Perceptron(1,0,1) #True or false

Perceptron(0,1,1) #False or true

Perceptron(0,0,0) #False or false

The perceptron is trained for 50 iterations with different input combinations.

The target outputs are set based on logical OR operation: true or true is 1, true or false is 1, false or true is 1, and false or false is 0.

THEORY:

1. **Neural Network Architecture:**
   * A neural network consists of layers, including an input layer, one or more hidden layers, and an output layer.
   * Each layer contains neurons (or nodes), and connections between neurons are represented by weights.
2. **Activation Functions:**
   * Activation functions introduce non-linearity into the network, allowing it to learn complex relationships.
   * Common activation functions include the sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU).
3. **Forward Pass:**
   * During the forward pass, input data is propagated through the network to generate predictions.
   * Neurons in each layer calculate a weighted sum of inputs, apply an activation function, and pass the result to the next layer.
4. **Loss Function:**
   * The loss function measures the difference between the predicted output and the actual target.
   * The goal is to minimize the loss during training.
5. **Backpropagation:**
   * Backpropagation is a supervised learning algorithm used to train neural networks.
   * It involves calculating the gradient of the loss function with respect to the weights and biases, and then adjusting them using gradient descent.
6. **Gradient Descent:**
   * Gradient descent is an optimization algorithm used to minimize the loss function.
   * It involves iteratively adjusting the weights and biases in the direction opposite to the gradient.
7. **Learning Rate:**
   * The learning rate determines the step size in weight and bias updates during gradient descent.
   * Choosing an appropriate learning rate is crucial for training stability and convergence.
8. **Epochs:**
   * An epoch is a single pass through the entire training dataset.
   * Training involves multiple epochs to allow the network to learn from the data over multiple iterations.
9. **Overfitting and Regularization:**
   * Overfitting occurs when a model performs well on the training data but poorly on new, unseen data.
   * Regularization techniques, such as dropout and weight decay, help prevent overfitting.
10. **Hyperparameters:**
    * Hyperparameters are parameters not learned during training, such as the learning rate, number of hidden layers, and number of neurons in each layer.
    * Tuning hyperparameters is essential for achieving optimal network performance.
11. **Testing and Evaluation:**
    * After training, the model should be evaluated on a separate test dataset to assess its generalization performance.
    * Metrics such as accuracy, precision, recall, and F1 score are commonly used for evaluation.
12. **Deep Learning and Deep Neural Networks:**
    * Deep learning involves training neural networks with multiple hidden layers, enabling the learning of hierarchical representations.
    * Deep neural networks have demonstrated success in various complex tasks, including image recognition, natural language processing, and more.

**Experiment No: 02 Create and view custom neural networks**

MATLAB

**Neural Networks Overview:** Neural networks are algorithmic models inspired by the human brain, designed to recognize and interpret patterns in various forms of data. They work by clustering and classifying input data through interconnected nodes or perceptrons.

**Key Points:**

1. **Adaptability:**
   * Neural networks adapt to changing input, generating optimal results without redesigning output criteria.
2. **Organic and Artificial Neurons:**
   * Neural networks can consist of organic (biological) or artificial neurons.
3. **Pattern Recognition:**
   * Neural networks excel at recognizing numerical patterns in vectors derived from real-world data like images, sound, text, or time series.
4. **Interconnected Nodes:**
   * Neural networks comprise layers of interconnected nodes, each representing a perceptron similar to multiple linear regression.
5. **Activation Function:**
   * Nodes feed signals through activation functions, often nonlinear, enhancing the network's ability to capture complex relationships.
6. **Multi-Layered Perceptron (MLP):**
   * In MLP, perceptrons are organized in layers, with input layers collecting patterns, hidden layers fine-tuning weightings, and output layers providing classifications or signals.
7. **Hidden Layers' Role:**
   * Hidden layers are believed to extract salient features from input data, enhancing predictive power for outputs.

**Procedure:**

1. **Step-1 - Network Definition:**
   * Define a custom neural network, specifying input and output/target variables.
   * Assign input and target data to be used in training.
2. **Step-2 - Topology and Transfer Function:**
   * Define the network's topology, including the arrangement of layers.
   * Specify the transfer function, often nonlinear, to enhance the learning capacity of the network.
3. **Step-3 - Configure and View:**
   * Configure the network parameters and view the architecture.
4. **Step-4 - Training and Output Calculation:**
   * Train the network using the provided data.
   * Calculate neuron outputs during training to optimize the network's performance.

**Experiment No: 03 Vacuum Cleaner World Agent using Python**

import random

class Environment(object):

def \_\_init\_\_(self):

self.locationcondition={'A' : '1' , 'B' :'1'}

self.locationcondition['A']=random.randint(0,1)

self.locationcondition['B']=random.randint(0,1)

class Sreflexagent(Environment):

def \_\_init\_\_(self,Environment):

#place vacuum at random location

print(Environment.locationcondition)

vacuumlocation=random.randint(0,1)

if vacuumlocation==0:

print("vacuum is randomly placed at locationn A")

#and if location A is dirty

if Environment.locationcondition['A']==1:

print("Location A is dirty")

#suck the dirt and mark it clean

Environment.locationcondition['A']=0

print("Location A has been cleaned")

vacuumlocation=1

else:

print("Location A is clean") #move to B

print("moving to location B")

vacuumlocation=1

if Environment.locationcondition['B']==1:

#suck the dirt and mark it clean

Environment.locationcondition['B']=0

print("Location B has been cleaned")

elif vacuumlocation==1:

print("vacuum is randomly placed at locationn B")

#and if location B is dirty

else:

print("Location B is clean")

#and if location B is dirty

if Environment.locationcondition['B']==1:

print("Location B is dirty")

#suck the dirt and mark it clean Environment.

Environment.locationcondition['B']=0;

print("Location B has been cleaned")

print("moving to location A")

vacuumlocation=0

else:

print("Location B is Clean")

#Move to A

print("moving to location A")

vacuumlocation

if vacuumlocation==0:

if Environment.locationcondition['A']==1:

print("Location A is dirty")

#suck the dirt and mark it clean

Environment.locationcondition['A']=0;

print("Location A has been cleaned")

else:

print("Location A is Clean")

obj= Environment()

agent=Sreflexagent(obj)

1. **Environment Class:**
   * The **Environment** class initializes a two-location environment (A and B) and assigns a random dirt condition (0 for clean, 1 for dirty) to each location.
2. **Sreflexagent Class:**
   * The **Sreflexagent** class inherits from the **Environment** class, representing a simple reflex agent.
   * The agent is placed randomly at either location A or B.
   * If the agent is at location A, it checks if A is dirty, cleans it, and moves to B. If B is dirty, it cleans it; otherwise, it moves back to A.
   * If the agent is at location B, the process is similar, checking cleanliness, cleaning if necessary, and moving between locations.
3. **Key Points:**
   * The agent's vacuum location is determined randomly (0 for A, 1 for B).
   * The agent checks the dirt condition at its current location and cleans if necessary.
   * The agent then moves to the other location, repeating the process.
   * The code prints information about the agent's actions and the cleanliness status of locations.
4. **Problem:**
   * AI search problem featuring a goal-based vacuum cleaner agent.
5. **Agent Objective:**
   * Clean both rooms in a two-room environment.
6. **Environment:**
   * Consists of two rooms, initially dirty.
   * Single vacuum cleaner present in either room.
7. **Agent's Goal:**
   * Reach a state where both rooms are clean.
8. **Program Components:**
   * **States:** Different environment configurations.
   * **Goal State:** Both rooms clean.
   * **Goal Test:** Condition to check goal achievement.
   * **Actions:** Possible agent movements (e.g., move left, move right, suck dirt).
   * **Transition Model:** Describes action effects.
   * **Path Cost:** Cost of action sequence leading to the goal.
9. **Input for Test Cases:**
   * **Room Locations:** 'A' or 'B' in capitals.
   * **Status:** '0' (CLEAN) or '1' (DIRTY).
   * **Environment Sensing:** Vacuum cleaner checks other room's status.
10. **Output:**
    * Returns action sequence for each initial state.
    * Includes path cost.
11. **Test Case Generation:**
    * Program generates two test cases.
12. **Sensing Mechanism:**
    * Vacuum cleaner senses other room's status.
13. **Goal Achievement:**
    * Goal reached when both rooms are clean.

**Experiment No: 04 Classify species of Iris flower using MLP**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import sklearn.datasets

Import necessary libraries including NumPy, Pandas, Matplotlib, Seaborn, and the Iris dataset from Scikit-learn.

iris = sklearn.datasets.load\_iris()

iris\_data = pd.DataFrame(iris.data, columns=iris.feature\_names)

iris\_data['class'] = iris.target

iris\_data.head()

**Load and Explore the Iris Dataset:**

* Load the Iris dataset using Scikit-learn's **load\_iris()** function.
* Create a Pandas DataFrame (**iris\_data**) from the dataset, including feature names and target values.

iris\_data.describe()

iris\_data.info()

**Dataset Summary:**

* Display summary statistics of the dataset using **describe()** and information about data types using **info()**.

iris\_data.isnull().sum()

**Check for Missing Values:**

* Check for missing values in the dataset using **isnull().sum()**.

iris\_data.corr()

**Correlation Matrix:**

* Calculate the correlation matrix of features using **corr()**.

X = iris\_data.drop('class',axis = 1)

Y = iris\_data['class']

from sklearn.model\_selection import train\_test\_split

X\_train,X\_test,Y\_train,Y\_test = train\_test\_split(X,Y, test\_size=0.30)

**Prepare Data for Training:**

* Separate the features (**X**) and target variable (**Y**) from the dataset.
* Split the dataset into training and testing sets using **train\_test\_split()**.

from sklearn.linear\_model import LogisticRegression

model = LogisticRegression()

model.fit(X\_train, Y\_train) #model training

**Train a Logistic Regression Model:**

* Import the Logistic Regression model from Scikit-learn.
* Initialize the model and train it using the training data.

#print metric to get performance

print('Acuracy:', model.score(X\_test, Y\_test)\*100)

**Evaluate Model Performance:**

* Print the accuracy of the trained model on the test set.

**THEORy**

1. **Objective:**
   * A hobby botanist aims to distinguish the species of iris flowers based on measurements.
   * Measurements include petal and sepal length and width in centimeters.
2. **Data Collection:**
   * The botanist has measurements for irises, including known species (setosa, versicolor, virginica).
3. **Supervised Learning Problem:**
   * It's a supervised learning problem since there are measurements with known species labels.
   * The task is to predict the species of new irises based on measurements.
4. **Classification Problem:**
   * The goal is to classify irises into one of three species.
   * It's a three-class classification problem.
5. **Model Building:**
   * The machine learning model is created to learn from known iris species measurements.
   * The model aims to predict the species of new irises.
6. **Features and Labels:**
   * Measurements (petal and sepal dimensions) are features.
   * The species label is the desired output for a given iris.
7. **Three Species Classes:**
   * The dataset consists of irises belonging to three species: setosa, versicolor, and virginica.
8. **Supervised Learning Approach:**
   * The model learns from labeled data where species labels are already known.
   * It generalizes to predict species for new, unlabeled irises.
9. **Classification Output:**
   * For each iris, the model's output is the predicted species label.
10. **Desired Outcome:**
    * The botanist can use the model to predict iris species based on measurements for the newly discovered irises.

**Experiment No: 05 Linear Regression Using Python**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split # Add this import

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import r2\_score

import matplotlib.pyplot as plt

data= pd.read\_excel('C:/Users/SHREERANG/Desktop/AIML/CODES/Expt-5 DataSheet.xlsx')

data.describe()

data.head()

x = data.drop(['PE'], axis=1).values

y = data['PE'].values

# Display the features and target variable

print("Features (x):")

print(x)

print("Target variable (y):")

print(y)

# Split the data into training and testing sets

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.30, random\_state=0)

# Create a Linear Regression model

ml = LinearRegression()

# Fit the model on the training data

ml.fit(x\_train, y\_train)

# Make predictions on the test data

y\_pred = ml.predict(x\_test)

print("Predictions on the test data:")

print(y\_pred)

# Make a prediction for a specific input

specific\_input = [[14.96, 41.7, 1024.07, 73.17]]

prediction = ml.predict(specific\_input)

print("Prediction for specific input:")

print(prediction)

# Evaluate the model using R-squared (R2) score

r2 = r2\_score(y\_test, y\_pred)

print("R-squared (R2) score:")

print(r2)

# Create a scatter plot to visualize the model's predictions

plt.figure(figsize=(15, 10))

plt.scatter(y\_test, y\_pred)

plt.xlabel('Actual')

plt.ylabel('Predicted')

plt.title('Actual Vs Predicted')

# Create a DataFrame to display the actual values, predicted values, and the difference

pred\_y\_df = pd.DataFrame({'Actual Value': y\_test, 'Predicted Value': y\_pred, 'Difference': y\_test - y\_pred})

print("Sample of actual vs. predicted values:")

print(pred\_y\_df.head(15))

**Key Points:**

* **Linear Regression:**
  + Models relationship between dependent and independent variables.
  + Simple linear regression for one independent variable, multiple linear regression for more.
  + Mathematical representation: �=�⋅�+�*y*=*m*⋅*x*+*c*.
* **Gradient Descent:**
  + Optimization method for minimizing the sum of squared errors.
  + Iterative process, adjusts model parameters gradually.
  + Moves towards the local minimum of the cost function.
  + Proportional changes in weights based on the derivative of the cost function.
  + Stops when convergence is achieved, and further iterations do not significantly improve the model.

**Top of Form**

**Experiment No: 06 Implement and test MLP trained with back – propagation algorithm**

import numpy as np

# Data Preparation

x = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)

y = np.array(([92], [86], [89]), dtype=float)

# Scale Units

x = x / np.amax(x, axis=0)

# Maximum test score = 100

y = y / 100

print(x)

print(y)

# neural Network Class

class NeuralNetwork(object):

def \_\_init\_\_(self): # Fix the double underscores here

# Parameters

self.inputSize = 2

self.outputSize = 1

self.hiddenSize = 3

# Weights

self.W1 = np.random.randn(self.inputSize, self.hiddenSize)

self.W2 = np.random.randn(self.hiddenSize, self.outputSize)

def feedForward(self, X):

self.z = np.dot(X, self.W1)

self.z2 = self.sigmoid(self.z)

self.z3 = np.dot(self.z2, self.W2)

output = self.sigmoid(self.z3)

return output

def sigmoid(self, s, deriv=False):

if deriv:

return s \* (1 - s)

return 1 / (1 + np.exp(-s))

def backward(self, X, y, output):

self.output\_error = y - output

self.output\_delta = self.output\_error \* self.sigmoid(output, deriv=True)

self.z2\_error = self.output\_delta.dot(self.W2.T)

self.z2\_delta = self.z2\_error \* self.sigmoid(self.z2, deriv=True)

# Update weights

self.W2 += self.z2.T.dot(self.output\_delta)

self.W1 += X.T.dot(self.z2\_delta)

def train(self, X, y):

output = self.feedForward(X) # Fix the typo here

self.backward(X, y, output)

NN = NeuralNetwork()

for i in range(1000): # trains the NN 1000 times

if i % 100 == 0:

print("Loss" + str(np.mean(np.square(y - NN.feedForward(x))))) # Fix the typo here

NN.train(x, y)

## Final RESULT

print("Input: " + str(x))

print("Actual Output: " + str(y))

print("Loss: " + str(np.mean(np.square(y - NN.feedForward(x)))))

print("\n")

print("Predicted Output: " + str(NN.feedForward(x)))

THEORY:

1. **Architecture of MLPs:**
   * MLPs consist of sensory units (input layer), one or more hidden layers, and an output layer.
   * Input signals propagate through the network in a forward direction, layer by layer.
2. **Training with Back-Propagation:**
   * MLPs are trained using the error back-propagation algorithm in a supervised manner.
   * The algorithm involves two passes: a forward pass and a backward pass.
3. **Forward Pass:**
   * During the forward pass, an input pattern is applied to the sensory nodes, and the signal propagates through the network.
   * The network produces a set of outputs as the actual response.
4. **Backward Pass (Error Back-Propagation):**
   * In the backward pass, synaptic weights are adjusted based on an error correction rule.
   * The actual response is subtracted from the desired (target) response to compute an error signal.
   * The error signal is propagated backward against the direction of synaptic weights.
5. **Error Correction Rule:**
   * The error back-propagation algorithm adjusts weights to minimize the difference between the actual and target responses.
   * Synaptic weights are updated in the direction that reduces the error.
6. **Success of MLPs:**
   * MLPs have been successfully applied to solve diverse and challenging problems.
   * Their ability to learn complex patterns and relationships makes them powerful tools in supervised learning.

**Experiment No. 7 Implementation of Simple Genetic Algorithm**

import random

def fitness\_function(individual):

return sum(individual)

def generate\_individual():

return [random.randint(0, 100) for \_ in range(10)]

def mutate(individual):

index\_to\_mutate = random.randint(0, len(individual) - 1)

individual[index\_to\_mutate] = random.randint(0, 100)

return individual

def crossover(parent1, parent2):

midpoint = len(parent1) // 2

child1 = parent1[:midpoint] + parent2[midpoint:]

child2 = parent2[:midpoint] + parent1[midpoint:]

return child1, child2

def select\_parents(population):

fitness\_values = [fitness\_function(ind) for ind in population]

total\_fitness = sum(fitness\_values)

probabilities = [fit / total\_fitness for fit in fitness\_values]

parents = []

for \_ in range(2):

selected\_parents = random.choices(population, probabilities, k=2)

parents.append(max(selected\_parents, key=fitness\_function))

return parents

def genetic\_algorithm():

population\_size = 100

population = [generate\_individual() for \_ in range(population\_size)]

for generation in range(100):

parents = select\_parents(population)

offspring = []

for i in range(population\_size // 2):

parent1, parent2 = parents

child1, child2 = crossover(parent1, parent2)

offspring.append(mutate(child1))

offspring.append(mutate(child2))

population = parents + offspring

population.sort(key=fitness\_function, reverse=True)

population = population[:population\_size]

return population[0]

best\_individual = genetic\_algorithm()

print("Best Individual:", best\_individual)

print("Fitness Value:", fitness\_function(best\_individual))

**THEORY**

\*\*Genetic Algorithm Overview:\*\*

- \*\*Definition:\*\* A genetic algorithm is a search technique used in computing to find true or approximate solutions to optimization and search problems, inspired by Darwin's theory of evolution.

- \*\*Representation:\*\* Solutions are represented as chromosomes in a population. The population undergoes evolution to improve solutions over successive generations.

- \*\*Initialization:\*\*

1. Generate a random population of \(n\) chromosomes, representing potential solutions to the problem.

- \*\*Fitness Evaluation:\*\*

2. Evaluate the fitness \(f(x)\) of each chromosome in the population. Fitness indicates how well a solution addresses the problem.

- \*\*Evolution Process:\*\*

3. Create a new population by repeating the following steps until the new population is complete:

- \*\*Selection:\*\* Choose two parent chromosomes based on their fitness. Higher fitness increases the chance of selection.

- \*\*Crossover:\*\* With a certain probability, combine genetic material from parents to create offspring (children). No crossover results in exact copies.

- \*\*Mutation:\*\* Apply a mutation probability to alter the offspring's genetic makeup at each locus (position in the chromosome).

- \*\*Accepting:\*\* Place the new offspring in the new population.

- \*\*Replace:\*\* Use the new population for the next iteration.

- \*\*Termination:\*\*

6. If the termination condition is met (e.g., a certain number of generations or improvement threshold), stop the algorithm and return the best solution in the current population.

- \*\*Loop:\*\*

7. If the termination condition is not met, go back to step 2 to continue the evolutionary process.

\*\*Important Points:\*\*

- Genetic algorithms simulate the process of natural selection and evolution to optimize solutions.

- Chromosomes represent potential solutions, and their fitness determines their likelihood of being selected for reproduction.

- Crossover combines genetic material, and mutation introduces random changes, promoting diversity in the population.

- The algorithm iteratively refines solutions over generations until a termination condition is satisfied.

- Genetic algorithms are versatile and applicable to a wide range of optimization problems.