10. Fuzzy Arithmetic operations

```
def fu_add(a,b):
    res={}
    for k_a,v_a in a.items():
        for k_b,v_b in b.items():
            if k_a == k_b:
                res[k_a]=max(v_a,v_b)
                break
    return res

def fu_sub(a,b):
    res={}
    for k_a,v_a in a.items():
        for k_b,v_b in b.items():
            if k_a == k_b:
                res[k_a]=max(v_a - v_b,0)
                break
    return res

def fu_mul(a,b):
    res={}
    for k_a,v_a in a.items():
        for k_b,v_b in b.items():
            if k_a == k_b:
                res[k_a]=v_a * v_b
                break
    return res

def fu_div(a,b):
    res={}
    for k_a,v_a in a.items():
        for k_b,v_b in b.items():
            if k_a == k_b and v_b!=0:
                res[k_a]=v_a/v_b
```

```
                break
    return res

A={"a":0.2,"b":0.3,"c":0.6,"d":0.6}
B={"a":0.9,"b":0.9,"c":0.4,"d":0.5}

c=fu_add(A,B)
print("Fuzzy Addition:",c)

d=fu_sub(A,B)
print("Fuzzy Subtraction:",d)

e=fu_mul(A,B)
print("Fuzzy Multiplication:",e)

f=fu_div(A,B)
```

9.Fuzzy Logical Operations:

```
A=dict()
B=dict()
Y=dict()

A={"a":0.2,"b":0.3,"c":0.6,"d":0.6}
B={"a":0.9,"b":0.9,"c":0.4,"d":0.5}

print("The First Fuzzy Set is :",A)
print("The Second Fuzzy Set is :",B)
for k_a,k_b in zip(A,B):
    a_val=A[k_a]
    b_val=B[k_a]
    if a_val > b_val:
        Y[k_a]=a_val
    else:
        Y[k_b]=b_val
print("The Fuzzy set union is :",Y)

print("\nThe First Fuzzy Set is :",A)
print("The Second Fuzzy Set is :",B)
for k_a,k_b in zip(A,B):
    a_val=A[k_a]
    b_val=B[k_a]
    if a_val < b_val:
        Y[k_a]=a_val
    else:
        Y[k_b]=b_val
print("The Fuzzy set intersection is :",Y)

print("\nThe Fuzzy Set is :",A)
for k_a in A:
    Y[k_a]=1-A[k_a]
print("The Fuzzy set complement is :",Y)
```

6. De_Morgan's Law

```
def demorgan_law(statement):

    if 'not (' in statement and ' and ' in statement:
        A, B = statement.split('not (')[1].split(' and ')
        return f'(not {A}) or (not {B})'

    elif 'not (' in statement and ' or ' in statement:
        A, B = statement.split('not (')[1].split(' or ')
        return f'(not {A}) and (not {B})'

    else:
        print("Invalid Statement")
        return statement

result = demorgan_law('not (A and B)')
print("De Morgan's Law to 'not (A and B)' is",result)

result = demorgan_law('not (A or B)')
print("De Morgan's Law to 'not (A or B)' is",result)
```

4.SOM

```
import math


class   SOM:
    def winner(self,weight,sample):
        D0,D1=0,0
        for i in range(len(sample)):
            D0+=math.pow((sample[i]-weight[0][i]),2)
            D1+=math.pow((sample[i]-weight[1][i]),2)
        return 0 if D0>D1 else 0
    def update(self,weight,sample,J,alpha):
        for i in range(len(weight)):
            weight[J][i]+=alpha*(sample[i]-weight[J][i])
        return weight


def main():
    T = [[1,1,0,0],[0,0,0,1],[1,0,0,0],[0,0,1,1]]
    weight=[[0.2,0.6,0.8,0.9],[0.4,0.5,0.3,0.6]]
    som=SOM()
    alpha=0.5
    epoch=3000
    for i in range(epoch):
        for j in range(len(T)):
            sample=T[j]
            J=som.winner(weight,sample)
            weight=som.update(weight,sample,J,alpha)
    t_s=[0,0,0,1]
    J=som.winner(weight,t_s)
    print("The sample cluster is:",J)
    print("Train weight is:",weight)


if __name__=="__main__":
    main()
```

2.PERCEPTRON:

```
import numpy as np

def unitStep(v):
    return 1 if v>=0 else 0

def perceptrionModel(x,w,b):
    v=np.dot(w,x)+b
    y=unitStep(v)
    return y

def OR_logic(x):
    w=np.array([1,1])
    b=0.5
    return perceptrionModel(x,w,b)

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

for test in tests:
    x,expected=test
    result=OR_logic(x)
    print(f"OR({x[0]},{x[1]})={result} (expected {expected})")
```

1.ANN

```
import numpy as np
from matplotlib import pyplot as plt

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

def inPa(inFe,neHid,ouFe):
    w1 = np.random.randn(neHid,inFe)
    w2 = np.random.randn(ouFe,neHid)
    b1 = np.zeros((neHid,1))
    b2 = np.zeros((ouFe,1))
    parameters = {"w1":w1,"w2":w2,"b1":b1,"b2":b2}
    return parameters

def foPa(X,Y,parameters):
    m=X.shape[1]
    w1=parameters["w1"]
    w2=parameters["w2"]
    b1=parameters["b1"]
    b2=parameters["b2"]
    z1=np.dot(w1,X)+b1
    a1=sigmoid(z1)
    z2=np.dot(w2,a1)+b2
    a2=sigmoid(z2)
    cahce=(z1,a1,w1,b1,z2,a2,w2,b2)
    logprobs=np.multiply(np.log(a2),Y)+np.multiply(np.log(1-a2),(1-Y))
    cost=-np.sum(logprobs)/m
    return cost,cahce,a2

def baPa(X,Y,cache):
    m=X.shape[1]
```

```
    (z1,a1,w1,b1,z2,a2,w2,b2)=cache
    dz2=a2-Y
    dw2=np.dot(dz2,a1.T)/m
    db2=np.sum(dz2,axis=1,keepdims=True)
    da1=np.dot(w2.T,dz2)
    dz1=np.multiply(da1,a1*(1-a1))
    dw1=np.dot(dz1,X.T)/m
    db1=np.sum(dz1,axis=1,keepdims=True)/m
    gradients ={"dz2":dz2,"dw2":dw2,"db2":db2,"dz1":dz1,"dw1":dw1,"db1":db1}
    return gradients

def upPa(gradients,learningRate,parameters):
    parameters["w1"]=parameters["w1"]-learningRate*gradients["dw1"]
    parameters["w2"]=parameters["w2"]-learningRate*gradients["dw2"]
    parameters["b1"]=parameters["b1"]-learningRate*gradients["db1"]
    parameters["b2"]=parameters["b2"]-learningRate*gradients["db2"]
    return parameters

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

neHid=2
inFe=X.shape[0]
ouFe=Y.shape[0]
parameters = inPa(inFe,neHid,ouFe)
learningRate=0.01
epoch=100000
losses =np.zeros((epoch,1))
for i in range(epoch):
    losses[i,0],cache,a2=foPa(X,Y,parameters)
    gradients=baPa(X,Y,cache)
    parameters=upPa(gradients,learningRate,parameters)
```

```
plt.figure()
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Loss Value")
plt.show()

X=np.array([[1,1,0,0,],[0,1,0,1]])
cost,_,a1=foPa(X,Y,parameters)
predict=(a2>0.5)*1.0

print(predict)
```