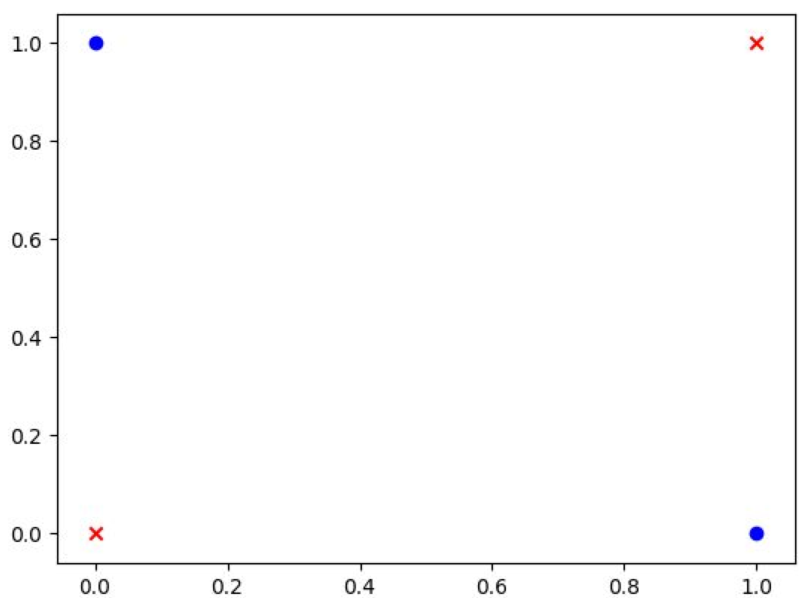
Code:

```
import numpy as np

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
input_not = np.array([[0], [1]])

def threshold(x, t):
    return 1 if x >= t else 0

def get_weights_threshold(gate):
    if gate == 'or':
        return np.array([1, 1]), 1
    elif gate == 'nor':
        return np.array([-1, -1]), 0
    elif gate == 'and':
        return np.array([1, 1]), 2
    elif gate == 'nand':
        return np.array([-1, -1]), -1
    elif gate == 'not':
        return np.array([-1]), 0

def get_sum(inp, weights):
    return inp.dot(weights.T)

gate = input('Gate: ')
weights, t = get_weights_threshold(gate.strip().lower())

if gate == 'not':
    print('X Output')
    for i in range(0, 2):
        ip = input_not[i, :]
        print(ip[0], threshold(get_sum(ip, weights)), t)
else:
    print('X1 X2 Output')
    for i in range(0, 4):
        ip = inputs[i, :]
        print(ip[0], ip[1], threshold(get_sum(ip, weights), t))
```

Output:

```
rudra@hogwarts:~/Soft Computing$ python3 gates_nn.py
Gate: or
X1      X2      Output
0       0       0
0       1       1
1       0       1
1       1       1
rudra@hogwarts:~/Soft Computing$ python3 gates_nn.py
Gate: and
X1      X2      Output
0       0       0
0       1       0
1       0       0
1       1       1
rudra@hogwarts:~/Soft Computing$ python3 gates_nn.py
Gate: nor
X1      X2      Output
0       0       1
0       1       0
1       0       0
1       1       0
rudra@hogwarts:~/Soft Computing$ python3 gates_nn.py
Gate: nand
X1      X2      Output
0       0       1
0       1       1
1       0       1
1       1       0
rudra@hogwarts:~/Soft Computing$ python3 gates_nn.py
Gate: not
X Output
0       1
1       0
rudra@hogwarts:~/Soft Computing$
```



Code:

```
import numpy as np

x_train = np.array([[0,0],[0,1],[1,0],[1,1]])
x_not = np.array([[0],[1]])

def get_output(gate):
    if gate == "and":
        return np.array([0,0,0,1])
    elif gate == "or":
        return np.array([0,1,1,1])
    elif gate == "not":
        return np.array([1,0])
    elif gate == "nand":
        return np.array([1,1,1,0])
    elif gate == "nor":
        return np.array([1,0,0,0])
    else:
        pass
def tlu(net, threshold):
    return 1 if net >= threshold else 0
def init(x_type):
    x = np.hstack((np.ones((x_type.shape[0],1)), x_type))
    w = np.random.rand(x.shape[1])
    print("Weights before training: " + str(w))
    return x,w

def sgd(y, n_iter, c, x_type=x_train):
    x, w = init(x_type)
    y_temp = np.copy(y)
    for _ in range(n_iter):
        dataset = np.hstack((x, y_temp.reshape((x.shape[0], 1))))
        np.random.shuffle(dataset)
        x = dataset[:, :-1]
        y_temp = dataset[:, -1]
        for i in range(x.shape[0]):
            ex = x[i]
            net = np.sum(w*ex)
            out = tlu(net,0)
            w = w + c*(y_temp[i] - out)*ex          #Perceptron learning rule
    print_report(x, y_temp.astype(np.int), w)
def print_report(x, y, w):
    print("Weights after training : " + str(w) + "\n")
```

```
        print("After training :")
        print("Actual Values : " + str(list(y)))
        print("Calculated Values : " + str([tlu(net, 0) for net in x.dot(w.reshape((w.shape[0],1)))]))

def logic_function(fn_name, n_iter = 10, c=1):
        if fn_name not in ['not', 'or', 'and', 'nand', 'nor']:
                print("Invalid function name")
        if fn_name == "not":
                print(20*"-" + fn_name.upper() + 20*"-")
                sgd(get_output(fn_name), n_iter, c, x_not)
        else:
                print(20*"-" + fn_name.upper() + 20*"-")
                sgd(get_output(fn_name), n_iter, c)
logic_function(input(“Gate : ”).strip().lower())
```

Output:

```
rudra@hogwarts:~/Soft Computing$ python3 perceptron.py
--------------------AND--------------------
Weights before training: [ 0.89909047  0.01977794  0.25340875]
Weights after training : [-2.10090953  1.01977794  1.25340875]

After training :
Actual Values : [0, 0, 1, 0]
Calculated Values : [0, 0, 1, 0]
--------------------OR--------------------
Weights before training: [ 0.34603535  0.78380161  0.24202832]
Weights after training : [-0.65396465  0.78380161  1.24202832]

After training :
Actual Values : [1, 1, 1, 0]
Calculated Values : [1, 1, 1, 0]
--------------------NAND--------------------
Weights before training: [ 0.16345287  0.38397698  0.14404643]
Weights after training : [ 2.16345287 -0.61602302 -1.85595357]

After training :
Actual Values : [1, 1, 1, 0]
Calculated Values : [1, 1, 1, 0]
--------------------NOR--------------------
Weights before training: [ 0.68592094  0.27795546  0.80745318]
Weights after training : [ 0.68592094 -0.72204454 -1.19254682]

After training :
Actual Values : [1, 0, 0, 0]
Calculated Values : [1, 0, 0, 0]
--------------------NOT--------------------
Weights before training: [ 0.63950924  0.55066904]
Weights after training : [ 0.63950924 -1.44933096]

After training :
Actual Values : [0, 1]
Calculated Values : [0, 1]
rudra@hogwarts:~/Soft Computing$
```

Code:

```
import numpy as np
from math import exp

x_train = np.array([[0,0],[0,1],[1,0],[1,1]])
x_not = np.array([[0],[1]])

def get_output(gate):
    if gate == "and":
        return np.array([0,0,0,1])
    elif gate == "or":
        return np.array([0,1,1,1])
    elif gate == "not":
        return np.array([1,0])
    elif gate == "nand":
        return np.array([1,1,1,0])
    elif gate == "nor":
        return np.array([1,0,0,0])
    else:
        pass

def activation(net):
    return 1/(1 + exp(-net))
def derivative(out):
    return (out - out**2)
def init(x_type):
    x = np.hstack((np.ones((x_type.shape[0],1)), x_type))
    w = np.random.rand(x.shape[1])
    print("Weights before training: " + str(w))
    return x,w

def sgd(y, n_iter, c, x_type=x_train):
    x, w = init(x_type)
    y_temp = np.copy(y)
    for _ in range(n_iter):
        dataset = np.hstack((x, y_temp.reshape((x.shape[0], 1))))
        np.random.shuffle(dataset)
        x = dataset[:, :-1]
        y_temp = dataset[:, -1]
        for i in range(x.shape[0]):
            ex = x[i]
            net = np.sum(w*ex)
            out = activation(net)
```

```
                w = w + c*(y_temp[i] - out)*derivative(out)*ex          #Delta learning
    print_report(x, y_temp.astype(np.int), w)
def print_report(x, y, w):
    print("Weights after training : " + str(w) + "\n")
    print("After training :")
    print("Actual Values : " + str(list(y)))
    out = np.array([activation(net) for net in x.dot(w.reshape((w.shape[0],1)))])
    print("Activation Values : " + str(out))
    print("Calculated Values : " + str((out > 0.5).astype(np.int)))

def logic_function(fn_name, n_iter = 100, c=1):
    if fn_name not in ['not', 'or', 'and', 'nand', 'nor']:
        print("Invalid function name")
    if fn_name == "not":
        print(20*"-" + fn_name.upper() + 20*"-")
        sgd(get_output(fn_name), n_iter, c, x_not)
    else:
        print(20*"-" + fn_name.upper() + 20*"-")
        sgd(get_output(fn_name), n_iter, c)

logic_function(input(“Gate : ”).strip().lower())
```

Output:

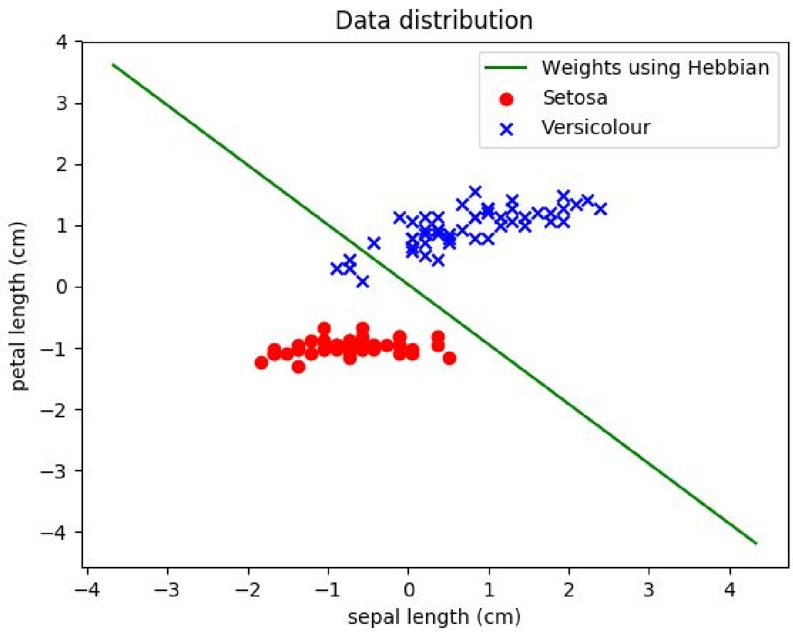
```
rudra@hogwarts:~/Soft Computing$ python3 delta\ rule.py
--------------------AND--------------------
Weights before training: [ 0.14628133  0.08169978  0.97577748]
Weights after training : [-4.21747994  2.73030499  2.73723617]

After training :
Actual Values : [1, 0, 0, 0]
Activation Values : [ 0.77731046  0.01452174  0.18434613  0.1853906 ]
Calculated Values : [1 0 0 0]
--------------------OR--------------------
Weights before training: [ 0.93987599  0.3172606   0.0600553 ]
Weights after training : [-1.33160342  3.29352239  3.28972942]

After training :
Actual Values : [0, 1, 1, 1]
Activation Values : [ 0.20889426  0.99478843  0.87633     0.87674048]
Calculated Values : [0 1 1 1]
--------------------NAND--------------------
Weights before training: [ 0.15469491  0.56921393  0.61112897]
Weights after training : [ 4.04977875 -2.63761337 -2.6273654 ]

After training :
Actual Values : [1, 1, 1, 0]
Activation Values : [ 0.80410726  0.98287224  0.80571647  0.22878226]
Calculated Values : [1 1 1 0]
--------------------NOR--------------------
Weights before training: [ 0.95870889  0.49956029  0.066974  ]
Weights after training : [ 1.38247678 -3.37388619 -3.38308052]

After training :
Actual Values : [0, 0, 0, 1]
Activation Values : [ 0.12010783  0.00461191  0.11913955  0.79938849]
Calculated Values : [0 0 0 1]
--------------------NOT--------------------
Weights before training: [ 0.09534359  0.21549647]
Weights after training : [ 1.70359087 -3.67319978]

After training :
Actual Values : [1, 0]
Activation Values : [ 0.84600314  0.1224309 ]
Calculated Values : [1 0]
rudra@hogwarts:~/Soft Computing$
```

Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import *
import matplotlib.pyplot as plt

data, target = load_iris(True)
X = data[:100, [0, 2]]
X = StandardScaler().fit_transform(X)
x = np.hstack((np.ones((X.shape[0], 1)), X))
y = target[:100]
y = y.reshape((x.shape[0], 1))
y[y == 0] = -1
class NN():
    def __init__(self, eta=0.01, n_iter=2000):
        self.eta = eta
        self.n_iter = n_iter
        self.w = np.random.sample(x.shape[1])
        self.w = normalize(self.w, norm='l1')
    def __f(self, net):
        return 2/(1 + np.exp(-net)) - 1
    def __binary(self, out):
        return np.sign(out)
    def train(self):
        for _ in range(self.n_iter):
            net = np.dot(x, self.w.reshape((x.shape[1], 1)))
            out = self.__f(net)
            error = y - out
            self.w += self.eta * (x.T.dot(out)).reshape(x.shape[1])
            self.w = normalize(self.w, norm='l1')
        self.w = self.w.reshape(x.shape[1])
        return self
    def predict(self, x_test):
        net = np.dot(x_test, self.w.reshape((x.shape[1], 1)))
        out = self.__f(net)
        return out.reshape(x.shape[0])
    def plot(self):
        plt.ion()
        plt.title("Data distribution")
        plt.xlabel("sepal length (cm)")
        plt.ylabel("petal length (cm)")
        plt.scatter(x[:50, 1], x[:50, 2], color='r', label='Setosa')
        plt.scatter(x[50:, 1], x[50:, 2], color='b', label='Versicolour', marker='x')
```

```
plt.legend()
domain_min, domain_max = x[:, 1:].min(), x[:, 1:].max()
domain = np.arange(domain_min*2, domain_max*2, 0.5)
plt.plot(domain, (-self.w[1]*domain - self.w[0])/self.w[2], color='g')
plt.pause(0.0001)
```

Output:



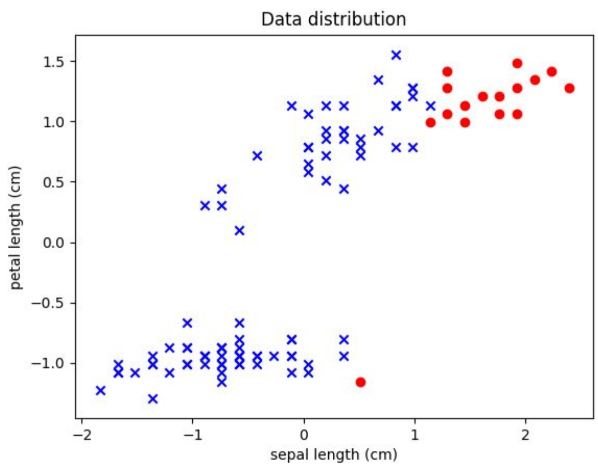
Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import *
import matplotlib.pyplot as plt

data, target = load_iris(True)
X = data[:100, [0, 2]]
X = StandardScaler().fit_transform(X)
x = np.hstack((np.ones((X.shape[0], 1)), X))
y = target[:100]
y = y.reshape((x.shape[0], 1))
class NN():
    def __init__(self, eta=0.01, n_iter=2000, n_clusters=2):
        self.eta = eta
        self.n_iter = n_iter
        self.w = np.random.rand(n_clusters, x.shape[1])
        self.w = normalize(self.w, norm='l2', axis=1)
    def __f(self, net):
        return 1/(1 + np.exp(-net))
    def cluster(self):
        self.plot(x)
        for _ in range(self.n_iter):
            np.random.shuffle(x)
            for i in range(x.shape[0]):
                sample = x[i, :]
                out = self.__f(sample.dot(self.w.T))
                m = np.argmax(out)
                self.w[m, :] += self.eta * (sample - self.w[m, :])
                self.w = normalize(self.w, norm='l2', axis=1)
        self.plot(x)
        return self
    def predict(self, x_test):
        net = x_test.dot(self.w.T)
        out = self.__f(net)
        return np.argmax(out, axis=1)
    def plot(self, x):
        labels = self.predict(x)
        plt.title("Data distribution")
        plt.xlabel("sepal length (cm)")
        plt.ylabel("petal length (cm)")
        plt.scatter(x[labels == 0, 1], x[labels == 0, 2], color='r')
        plt.scatter(x[labels == 1, 1], x[labels == 1, 2], color='b', marker='x')
```

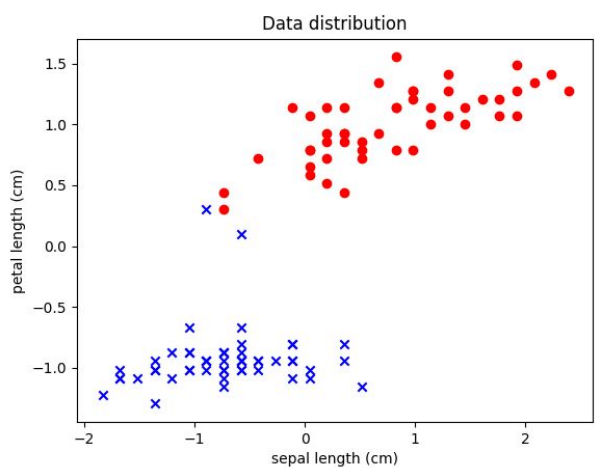
```
if self.w.shape[0] == 3:
        plt.scatter(x[labels == 2, 1], x[labels == 2, 2], color='g', marker='v')
plt.show()
```

Output:

Initial:



Final:



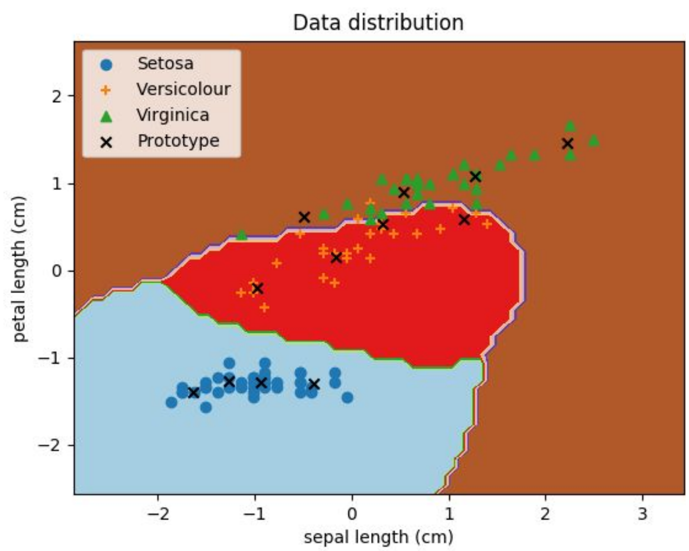
Code:

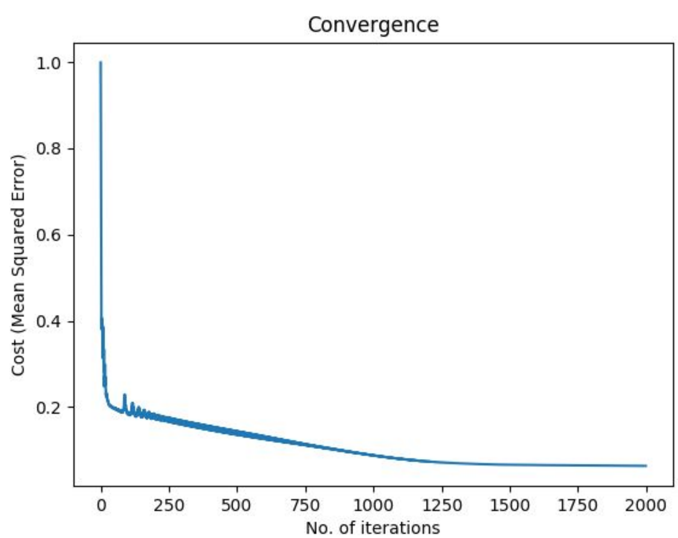
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import *
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from scipy.linalg import *

data, target = load_iris(True)
X = data[:, [0,2]]
X = StandardScaler().fit_transform(X)
x = np.hstack((np.ones((X.shape[0], 1)), X))
y = target[:]
y = y.reshape((x.shape[0], 1))
y_matrix = OneHotEncoder().fit_transform(y).todense()
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.33)
y_train_matrix = OneHotEncoder().fit_transform(y_train).todense()
y_test_matrix = OneHotEncoder().fit_transform(y_test).todense()
class NN():
        def __init__(self, eta=0.01, n_iter=2000, rbf_nodes=12):
                self.eta = eta
                self.n_iter = n_iter
                self.w = np.random.randn(3, rbf_nodes + 1)
                self.rbf_nodes = rbf_nodes
                self.centers = None
                self.betas = None
                self.costs = []
        def __getPrototypes(self, x_train=x_train, y_train=y_train_matrix):
                labels_train = np.array(np.argmax(y_train, axis=1).flatten()).reshape(x_train.shape[0])
                n_clusters = int(self.rbf_nodes/3)
                centers, betas = [], []
                for label in range(3):
                        temp_data = x_train[labels_train == label]
                        km = KMeans(n_clusters=n_clusters, n_jobs=-1).fit(temp_data)
                        centers.extend(km.cluster_centers_)
                        betas.extend(np.random.uniform(size=n_clusters))
                self.centers, self.betas = np.array(centers), np.array(betas)
                return self.centers, self.betas
        def __f(self, net):
                return 1/(1 + np.exp(-net))
        def __fdash(self, out):
                return (out - np.square(out))
```

```
    def __RBFactivation(self, x):
        return np.exp(-self.betas * ((x - self.centers)**2).sum(axis=1))
    def train(self, x_train=x_train, y_train=y_train_matrix):
        centers, betas = self.__getPrototypes(x_train, y_train)
        x_activation = np.hstack((np.ones((x_train.shape[0], 1)),
np.array([self.__RBFactivation(ex) for ex in x_train])))
        for _ in range(self.n_iter):
            net = x_activation.dot(self.w.T)
            out = self.__f(net)
            error = y_train - out
            self.w += self.eta * (np.multiply(error, self.__fdash(out))).T.dot(x_activation)
            self.costs.append(np.square(error).sum(axis=1).mean())
        return self
    def predict(self, x_test=x_test, y_test=y_test, error=True):
        x_activation = np.hstack((np.ones((x_test.shape[0], 1)),
np.array([self.__RBFactivation(ex) for ex in x_test])))
        net = x_activation.dot(self.w.T)
        out = self.__f(net)
        predictions = np.argmax(out, axis=1)
        if error:
            print("No. of misclassified test examples :
{0}".format(np.count_nonzero(predictions - y_test.flatten())))
        return predictions
    def convergence(self):
        plt.plot(np.arange(len(self.costs)), self.costs)
        plt.title("Convergence")
        plt.xlabel("No. of iterations")
        plt.ylabel("Cost (Mean Squared Error)")
        plt.show()
    def plot(self, x_train=x_train, y_train=y_train):
        plt.ion()
        plt.title("Data distribution")
        plt.xlabel("sepal length (cm)")
        plt.ylabel("petal length (cm)")
        x1_min, x1_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
        x2_min, x2_max = x_train[:, 2].min() - 1, x_train[:, 2].max() + 1
        xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, 0.1), np.arange(x2_min, x2_max,
0.1))
        z = self.predict(np.hstack((np.ones((xx1.shape[0] * xx1.shape[1],1)),
np.array([xx1.ravel(), xx2.ravel()]).T)), error=False).reshape(xx1.shape)
        plt.contourf(xx1, xx2, z, cmap=plt.cm.Paired)

        labels_train = y_train.flatten()
```

```
plt.scatter(x_train[labels_train == 0, 1], x_train[labels_train == 0, 2], label='Setosa')
plt.scatter(x_train[labels_train == 1, 1], x_train[labels_train == 1, 2], label='Versicolour',
marker='+')
plt.scatter(x_train[labels_train == 2, 1], x_train[labels_train == 2, 2], label='Virginica',
marker='x')
for centre in self.centers:
    plt.scatter(centre[1], centre[2], marker='x', color='k', label='Prototype')
plt.legend()
plt.pause(0.0001)
```

Output:



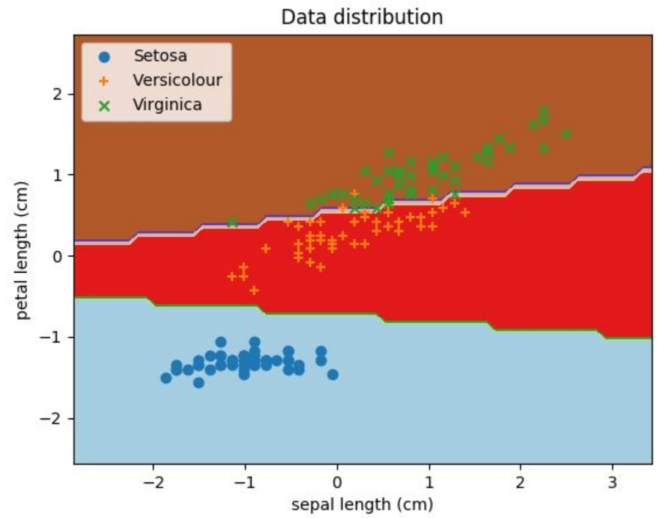


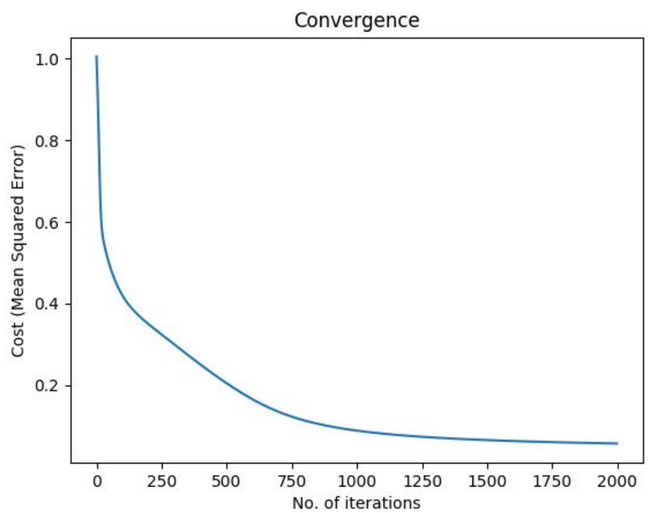
Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import *
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

data, target = load_iris(True)
X = data[:, [0,2]]
X = StandardScaler().fit_transform(X)
x = np.hstack((np.ones((X.shape[0], 1)), X))
y = target[:]
y = y.reshape((x.shape[0], 1))
y_matrix = OneHotEncoder().fit_transform(y).todense()
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.33)
y_train_matrix = OneHotEncoder().fit_transform(y_train).todense()
y_test_matrix = OneHotEncoder().fit_transform(y_test).todense()

class NN():
	def __init__(self, eta=0.01, n_iter=2000, hidden_nodes=4):
		self.eta = eta
		self.n_iter = n_iter
		self.w0 = np.random.randn(hidden_nodes, x.shape[1])
		self.w1 = np.random.randn(3, hidden_nodes + 1)
		self.costs = []
	def __f(self, net):
		return 1/(1 + np.exp(-net))
	def __fdash(self, out):
		return (out - np.square(out))
	def train(self, x_train=x_train, y_train=y_train_matrix):
		for _ in range(self.n_iter):
			net1 = x_train.dot(self.w0.T)
			hidden_out = self.__f(net1)
			augmented_hidden_out = np.hstack((np.ones((hidden_out.shape[0], 1)),
hidden_out))
			net2 = np.dot(augmented_hidden_out, self.w1.T)
			out = self.__f(net2)
			error = y_train - out
			del3 = np.multiply(error, self.__fdash(out))
			del2 = np.multiply(del3.dot(self.w1), np.hstack((np.ones((hidden_out.shape[0],
1)), self.__fdash(hidden_out))))
			d2 = del3.T.dot(augmented_hidden_out)
			d1 = del2[:, 1:].T.dot(x_train)
```

```
                self.w0 += d1/x_train.shape[0]
                self.w1 += d2/x_train.shape[0]
                self.costs.append(np.square(error).sum(axis=1).mean())
            return self
    def predict(self, x_test=x_test, y_test=y_test, error=True):
            net1 = x_test.dot(self.w0.T)
            hidden_out = self.__f(net1)
            augmented_hidden_out = np.hstack((np.ones((hidden_out.shape[0], 1)), hidden_out))
            net2 = np.dot(augmented_hidden_out, self.w1.T)
            out = self.__f(net2)
            predictions = np.argmax(out, axis=1)
            if error:
                print("No. of misclassified test examples :
{0}".format(np.count_nonzero(predictions - y_test.flatten())))
            return predictions
    def convergence(self):
            plt.plot(np.arange(len(self.costs)), self.costs)
            plt.title("Convergence")
            plt.xlabel("No. of iterations")
            plt.ylabel("Cost (Mean Squared Error)")
            plt.show()
    def plot(self, x_train=x_train, y_train=y_train):
            plt.ion()
            plt.title("Data distribution")
            plt.xlabel("sepal length (cm)")
            plt.ylabel("petal length (cm)")
            x1_min, x1_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
            x2_min, x2_max = x_train[:, 2].min() - 1, x_train[:, 2].max() + 1
            xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, 0.1), np.arange(x2_min, x2_max,
0.1))
            z = self.predict(np.hstack((np.ones((xx1.shape[0] * xx1.shape[1],1)),
np.array([xx1.ravel(), xx2.ravel()]).T)), error=False).reshape(xx1.shape)
            plt.contourf(xx1, xx2, z, cmap=plt.cm.Paired)
            labels_train = y_train.flatten()
            plt.scatter(x_train[labels_train == 0, 1], x_train[labels_train == 0, 2], label='Setosa')
            plt.scatter(x_train[labels_train == 1, 1], x_train[labels_train == 1, 2], label='Versicolour',
marker='+')
            plt.scatter(x_train[labels_train == 2, 1], x_train[labels_train == 2, 2], label='Virginica',
marker='x')
            plt.legend()
```

Output:


Data distribution
Setosa
Versicolour
Virginica
petal length (cm)
sepal length (cm)
2
1
0
−1
−2
−2
−1
0
1
2
3

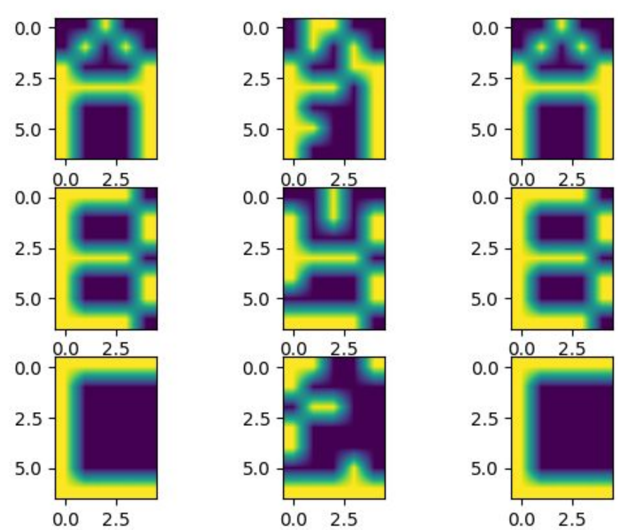

Convergence
Cost (Mean Squared Error)
No. of iterations
1.0
0.8
0.6
0.4
0.2
0
250
500
750
1000
1250
1500
1750
2000

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from math import ceil

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

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

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

rows, cols = a_pattern.shape  #Same for b and c
dims = rows * cols

def convertToVector(pattern):
	pattern[pattern == 0] = -1
	return pattern.reshape(1, dims)
def calculateWeight(patterns):
	return sum([(pattern.T.dot(pattern) - np.eye(dims)) for pattern in patterns])
def step_function(out, inp):
	out[out > 0] = 1
	out[out < 0] = -1
	ind = np.where(out == 0)
	out[ind] = inp[ind]
	return out
def calculateOutput(inp, weight):
	return step_function(inp.dot(weight), inp)
def addSaltAndPepperNoise(pattern, s_vs_p = 0.5, amount=0.4):
	outPattern = pattern.copy()
	num_of_salts = ceil(s_vs_p * amount * dims)
	num_of_pepper = ceil((1 - s_vs_p) * amount * dims)
	num_of_salt_and_pepper = num_of_salts + num_of_pepper
	coords = [np.random.randint(0, i-1, num_of_salt_and_pepper) for i in pattern.shape]
	outPattern[coords[0][:num_of_salts], coords[1][:num_of_salts]] = 1
	outPattern[coords[0][num_of_salts:], coords[1][num_of_salts:]] = 0
	return outPattern
def getNoisyProbes(patterns):
	return [addSaltAndPepperNoise(pattern) for pattern in patterns]
def plotResults(actualPatterns, inPatterns, outPatterns):
	if len(inPatterns) == len(actualPatterns) and len(inPatterns) == len(outPatterns):
```

```
        n_patterns = len(inPatterns)
        for i in range(n_patterns):
            plt.subplot(n_patterns, 3, 3*i+1)
            plt.imshow(actualPatterns[i].reshape(rows, cols), interpolation="bilinear")
            plt.subplot(n_patterns, 3, 3*i+2)
            plt.imshow(inPatterns[i].reshape(rows, cols), interpolation="bilinear")
            plt.subplot(n_patterns, 3, 3*i+3)
            plt.imshow(outPatterns[i].reshape(rows, cols), interpolation="bilinear")
        plt.show()
a_input, b_input, c_input = tuple([convertToVector(pattern) for pattern in [a_pattern, b_pattern,
c_pattern]])
a_noisy, b_noisy, c_noisy = tuple([convertToVector(noisy) for noisy in getNoisyProbes([a_pattern,
b_pattern, c_pattern])])
w = calculateWeight([a_input, b_input, c_input])
print("Network works correctly for provided input : {0}".format(verifyNetwork([a_input, b_input,
c_input], w)))
plotResults([a_input, b_input, c_input], [a_noisy, b_noisy, c_noisy], [calculateOutput(noisy, w) for noisy
in [a_noisy, b_noisy, c_noisy]])
```

Output:



Code:

```
import numpy as np
import matplotlib.pyplot as plt
from math import ceil
a_pattern = np.array([[0, 0, 1, 0, 0],[0, 1, 0, 1, 0],[1, 0, 0, 0, 1],[1, 1, 1, 1, 1],[1, 0, 0, 0, 1],[1, 0, 0, 0, 1],[1,
0, 0, 0, 1]])
b_pattern = np.array([[1, 1, 1, 1, 0],[1, 0, 0, 0, 1],[1, 0, 0, 0, 1],[1, 1, 1, 1, 0],[1, 0, 0, 0, 1],[1, 0, 0, 0, 1],[1,
1, 1, 1, 0]])
c_pattern = np.array([[1, 1, 1, 1, 1],[1, 0, 0, 0, 0],[1, 0, 0, 0, 0],[1, 0, 0, 0, 0],[1, 0, 0, 0, 0],[1, 0, 0, 0, 0],[1,
1, 1, 1, 1]])
rows, cols, dims = a_pattern.shape[0], a_pattern.shape[1], a_pattern.shape[0] * a_pattern.shape[1]
A_output, b_output, c_output = np.array([[1, -1, -1]]), np.array([[-1, 1, -1]]), np.array([[-1, -1, 1]])
def convertToVector(pattern):
	pattern[pattern == 0] = -1
	return pattern.reshape(1, dims)
def calculateWeight(inpatterns, outpatterns):
	return sum([inp.T.dot(out) for inp, out in zip(inpatterns, outpatterns)])
def step_function(out, inp):
	out[out > 0] = 1
	out[out < 0] = -1
	ind = np.where(out == 0)
	out[ind] = inp[ind]
	return out
def calculateOutput(inp, weight):
	return step_function(inp.dot(weight), inp)
def forwardPass(inp, weight):
	return calculateOutput(inp, weight)
def backwardPass(out, weight):
	return calculateOutput(out, weight.T)
def addSaltAndPepperNoise(pattern, s_vs_p = 0.5, amount=0.4):
	outPattern = pattern.copy()
	num_of_salts = ceil(s_vs_p * amount * dims)
	num_of_pepper = ceil((1 - s_vs_p) * amount * dims)
	num_of_salt_and_pepper = num_of_salts + num_of_pepper
	coords = [np.random.randint(0, i-1, num_of_salt_and_pepper) for i in pattern.shape]
	outPattern[coords[0][:num_of_salts], coords[1][:num_of_salts]] = 1
	outPattern[coords[0][num_of_salts:], coords[1][num_of_salts:]] = 0
	return outPattern
def plotImages(expectedPatterns, actualPatterns):
	if len(expectedPatterns) == len(actualPatterns):
		n_patterns = len(expectedPatterns)
		fig, axes = plt.subplots(nrows=3, ncols=2)
		for i in range(n_patterns):
```

```
                ax = plt.subplot(n_patterns, 2, 2*i+1)
                if i == 0:
                        ax.set_title("Expected")
                plt.imshow(expectedPatterns[i].reshape(rows, cols), interpolation="bilinear")
                ax = plt.subplot(n_patterns, 2, 2*i+2)
                if i == 0:
                        ax.set_title("Actual")
                plt.imshow(actualPatterns[i].reshape(rows, cols), interpolation="bilinear")
        plt.show()
def getNoisyProbes(patterns):
        return [addSaltAndPepperNoise(pattern) for pattern in patterns]
a_input, b_input, c_input = tuple([convertToVector(pattern) for pattern in [a_pattern, b_pattern, c_pattern]])
a_noisy, b_noisy, c_noisy = tuple([convertToVector(noisy) for noisy in getNoisyProbes([a_pattern, b_pattern, c_pattern])])
w = calculateWeight([a_input, b_input, c_input], [a_output, b_output, c_output])
actual_outs = [forwardPass(inp, w) for inp in [a_noisy, b_noisy, c_noisy]]
print(20*"-" + "Noisy images (Forward Pass)" + 20*"-" + "\nRequired outputs : {0}\n Actual outputs : {1}".format([a_output, b_output, c_output], actual_outs))
actual_ins = [backwardPass(out, w) for out in [a_output, b_output, c_output]]
plotImages([a_input, b_input, c_input], actual_ins)
```

Output:

Result of forward pass:

```
rudra@hogwarts:~/Soft Computing$ python3 bam.py
Forward pass verification : True
Backward Pass verification : True
--------------------Noisy images (Forward Pass)--------------------
Required outputs : [array([[ 1, -1, -1]]), array([[-1,  1, -1]]), array([[-1, -1
,  1]])]
 Actual outputs : [array([[ 1, -1, -1]]), array([[-1,  1, -1]]), array([[-1, -1,
  1]])]
```

Result of backward pass: