WEEK-6

Aim : Write a program to implement Delta learning

Description :

Delta learning rule:

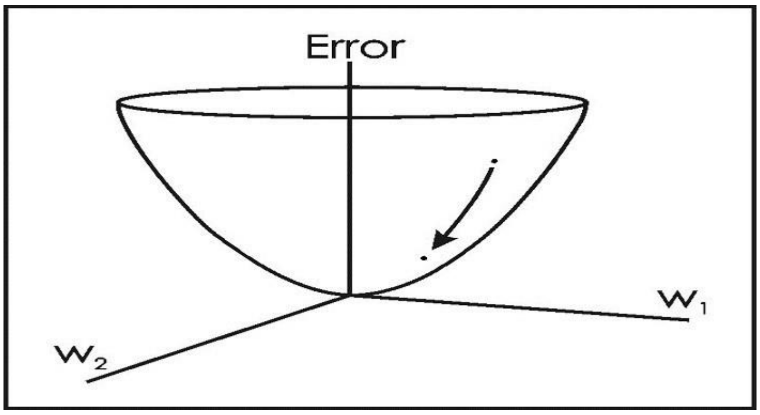
The Delta rule in an artificial neural network is a specific kind of backpropagation that assists in refining the machine learning/artificial intelligence network, making associations among input and outputs with different layers of artificial neurons.

The Delta Rule uses the difference between target activation (i.e., target output values) and obtained activation to drive learning. For reasons discussed below, the use of a threshold activation function (as used in both the McCulloch-Pitts network and the perceptron) is dropped & instead a linear sum of products is used to calculate the activation of the output neuron (alternative activation functions can also be applied). Thus, the activation function is called a Linear Activation function, in which the output node's activation is simply equal to the sum of the network's respective input/weight products. The strength of network connections (i.e., the values of the weights) are adjusted to reduce the difference between target and actual output activation (i.e., error). A graphical depiction of a simple two-layer network capable of deploying the Delta Rule is given in the figure below

During forward propagation through a network, the output (activation) of a given node is a function of its inputs. The inputs to a node, which are simply the products of the output of preceding nodes with their associated weights, are summed and then passed through an activation function before being sent out from the node. Thus, we have the following:

$$S_j = \sum_i w_{ij} a_i$$

$$a_j = f(S_i)$$



where ‘Sj’ is the sum of all relevant products of weights and outputs from the previous layer i, ‘wij’ represents the relevant weights connecting layer i with layer j, ‘ai’ represents the activation of nodes in the previous layer i, ‘aj’ is the activation of the node at hand, and ‘f’is the activation function.

For any given set of input data and weights, there will be an associated magnitude of error, which is measured by an error function (also known as a cost function). The Delta Rule employs the error function for what is known as Gradient Descent learning, which involves the ‘modification of weights along the most direct path in weight-space to minimize error’, so change applied to a given weight is proportional to the negative of the derivative of the error with respect to that weight The Error/Cost function is commonly given as the sum of the squares of the differences between all target and actual node activation for the output layer. For a particular training pattern (i.e., training case), error is thus given by:

$$E_p = \frac{1}{2} \sum_n (t_{j_n} - a_{j_n})^2$$

where ‘E’ is total error, and ‘p’ represents all training patterns. An equivalent term for E in earlier equation is Sum-of-squares error. A normalized version of this equation is given by the Mean Squared Error (MSE) equation:

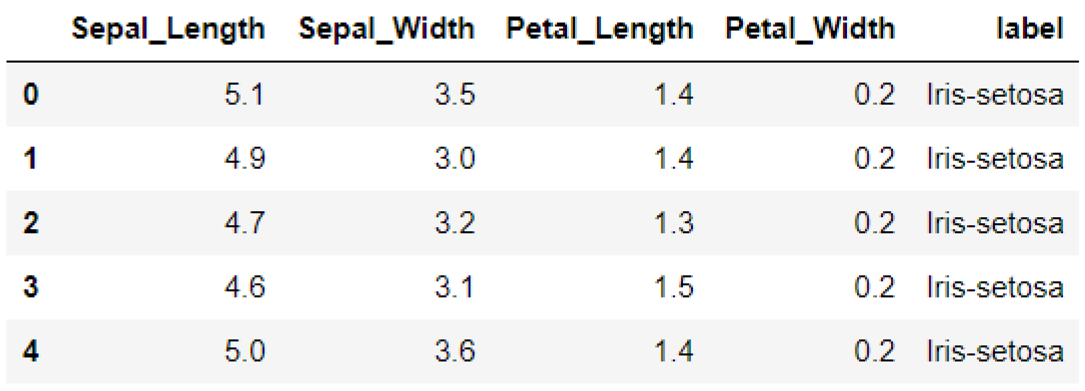
$$MSE = \frac{1}{2PN} \sum_p \sum_n (t_{j_n} - a_{j_n})^2$$

where ‘P’ and ’N’ are the total number of training patterns and output nodes, respectively. It is the error of both previous equations, that gradient descent attempts to minimize (not strictly true if weights are changed after each input pattern is submitted to the network. Error over a given training pattern is commonly expressed in terms of the Total Sum of Squares (‘tss’) error, which is simply equal to the sum of all squared errors over all output nodes and all training patterns. ‘The negative of the derivative of the error function is required in order to perform Gradient Descent Learning’.

$$\frac{\delta E_p}{\delta w_{ij_x}} = -(t_{j_z} - a_{j_z})(a_{i_x})$$

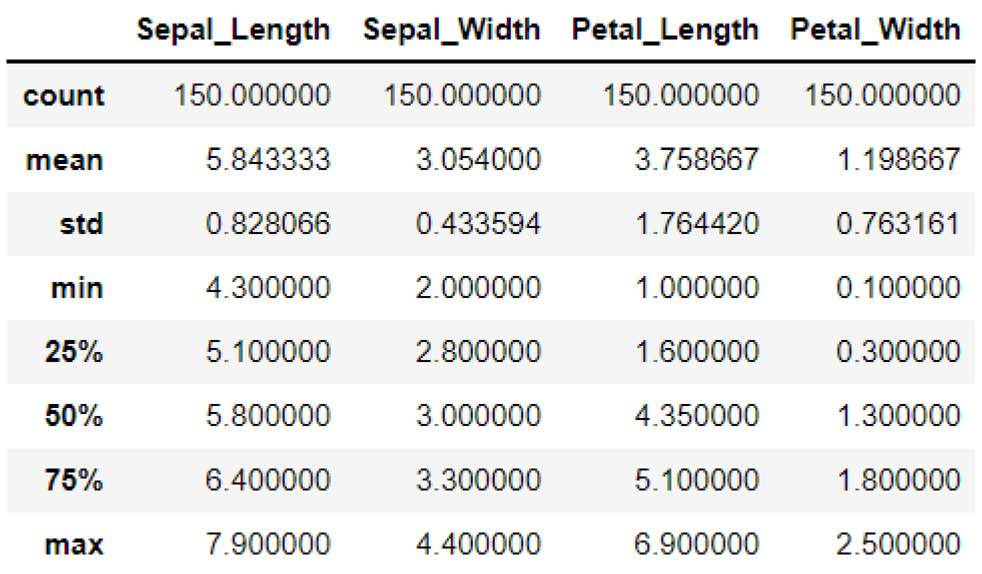
CODE :

```
# importing libraies
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# reaading dataset
df = pd.read_csv("/content/drive/MyDrive/isl/Iris/iris.csv",
names=['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width', 'label'])

df.head()
```



	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	label
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
df.describe()
```



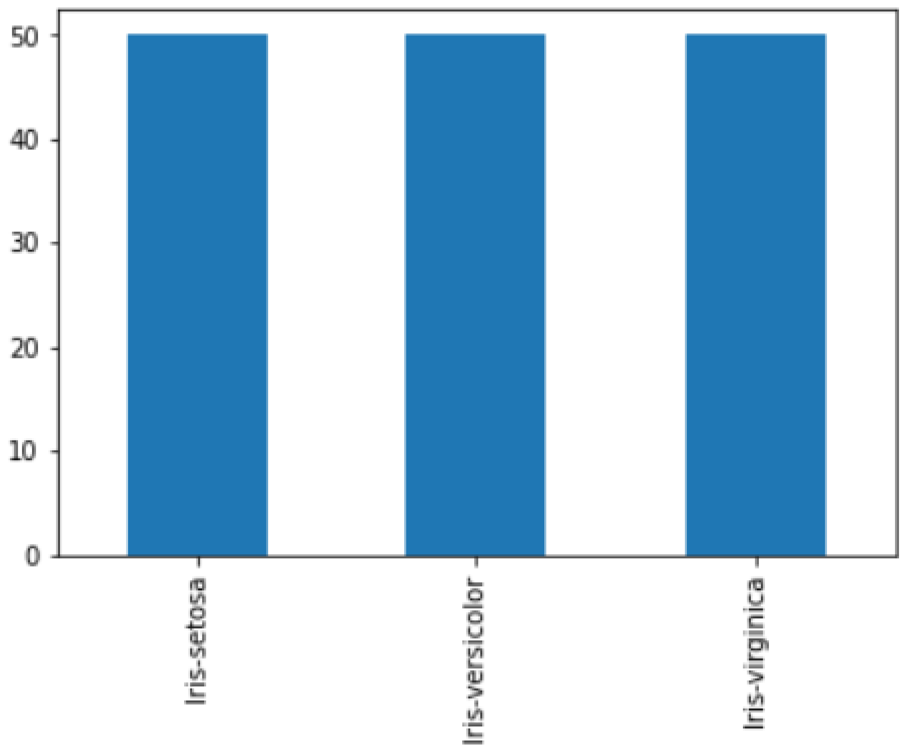
	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

```
count = df['label'].value_counts()
count
```

```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: label, dtype: int64
```

```
count.plot(kind = 'bar', x=count.index)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f62216e3890>
```



```
# Delta Learning on iris dataset
X = df.drop(['label'], axis = 1)
X.head()
```

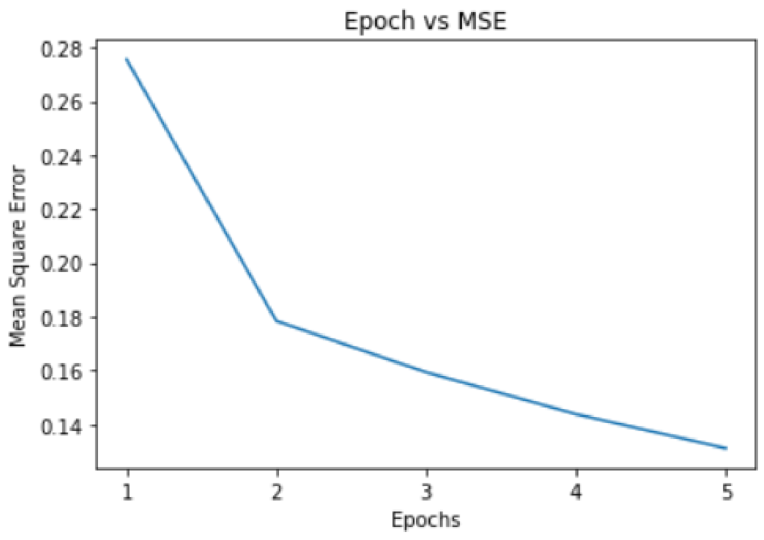
	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

```
y = df['label']
# coverting Categorical to numeric
y = y.map({'Iris-setosa' : 1,'Iris-versicolor' : 1, 'Iris-virginica' : 3})

--------------------------------------------------------------------------------

# Equation of the model or classifier
# W1.X1 + W2.X2 + W3.X3 + W4.X4 + b = Y

# initialising the Guassian random weights to model
w1, w2, w3, w4, b = tuple(np.random.normal(size=5))
print("Initial Weights + bias : ", (w1, w2, w3, w4, b))
# w1 = w2 = w3 = w4 = b = 0.1

# Number of epochs and learning rate
epochs = 5
alpha = 0.01

# delta learning
MSE = []
for i in range(epochs):
   error = []
   for i in range(len(X)):
      # Predicting target label with current weights
      y_pred = w1*X.iloc[i, 0] + w2*X.iloc[i, 1] + w3*X.iloc[i, 2] + w4*X.iloc[i, 3] +
b
      # actual target label
      t = y[i]
      # difference between actual and predicted
      diff = t - y_pred
      # error
      err = diff**2
      error.append(err)
      # --- Updation ---
      # gradients
      delta_w1 = alpha*diff*X.iloc[i, 0]
      delta_w2 = alpha*diff*X.iloc[i, 1]
      delta_w3 = alpha*diff*X.iloc[i, 2]
      delta_w4 = alpha*diff*X.iloc[i, 3]
      delta_b = alpha*diff
```

```
        # New weights
        w1 = w1 + delta_w1
        w2 = w2 + delta_w2
        w3 = w3 + delta_w3
        w4 = w4 + delta_w4
        b = b + delta_b

    # Mean square error
    MSE.append(np.array(error).mean())

for i in range(epochs):
    print("Mean Square error at epoch -", str(i+1)+" :", MSE[i])
```

```
Initial Weights + bias :  (-0.895832491489811, 0.6275785879548321, 0.3845075708854882, 0.7529748667934127, 0.010668801756195686)
Mean Square error at epoch - 1 : 0.2755811572060846
Mean Square error at epoch - 2 : 0.17841891435163415
Mean Square error at epoch - 3 : 0.15934806476436794
Mean Square error at epoch - 4 : 0.1438010736241405
Mean Square error at epoch - 5 : 0.13110813873071814
```

```
plt.plot(np.array(MSE))
plt.xticks(np.arange(epochs), [str(i+1) for i in range(epochs)])
plt.xlabel('Epochs')
plt.ylabel("Mean Square Error")
plt.title("Epoch vs MSE")
plt.show()
```


Epoch vs MSE
Mean Square Error
Epochs