3/5/23

EXPERIMENT - 9A

AIM

To implement the Support Vector Machine (SVM) algorithm in Python

ALGORITHM

* To choose the support vectors that are closer to the decision boundary initially.
* Each vector is then augmented with a 1 as a bias input to form augmented vector.
* 3 linear equations are formed to solve the values of $\alpha_1, \alpha_2, \alpha_3$.
* Then, the weight vector is computed using $\bar{w} = \sum_i \alpha_i \bar{s}_i$ resulting in 3 values.
* 1^{st} 2 values represent 'w' and last value is 'b' $\Rightarrow$ Hyperplane equation: $y = wx + b$

THEORY

It is a supervised learning algorithm used to create the best line / decision boundary that can segregate n-dimensional space into classes. This boundary is called a hyperplane.

OUTPUT

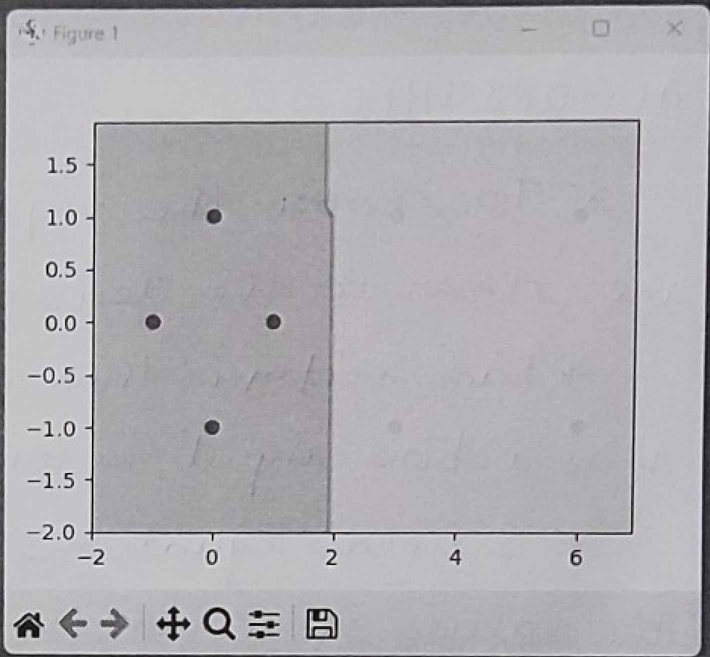
```
Data read from dataset.csv :

   Feature 1  Feature 2  Target
0          3          1       1
1          1          0      -1
2          3         -1       1
3          0          1      -1
4          6          1       1
5          0         -1      -1
6          6         -1       1
7         -1          0      -1

X :
[[ 3  1]
 [ 1  0]
 [ 3 -1]
 [ 0  1]
 [ 6  1]
 [ 0 -1]
 [ 6 -1]
 [-1  0]]

y :
[ 1 -1  1 -1  1 -1  1 -1]

Weight vector :  [1.24209993 0.03858835]
Bias term :  2.400000000000001
```



CODE

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv ( "svm_dataset.csv" )
print ( "\n Data read from dataset.csv :\n\n" , data )

arr = np.array ( data )
X = arr [ : , : -1 ]
y = arr [ : , -1 ]
print ( "\n X :\n" , X , "\n\n y :\n" , y )

w = np.zeros ( len ( X [ 0 ] ) )
b , lr , epochs = 0 , 0.1 , 1000

for epoch in range ( epochs ) :
    for i , x in enumerate ( X ) :
        if y [ i ] * ( np.dot ( X [ i ] , w ) - b ) >= 1 :
            w -= lr * ( 2 * 1 / epochs * w )
        else :
            w -= lr * (2 * 1 / epochs * w - np.dot ( X [ i ] , y [ i ] ) )
            b -= lr * y [ i ]

print ( "\n Weight vector : " , w )
print ( "Bias term : " , b )

plt.scatter ( X [ : , 0 ] , X [ : , 1 ] , c = y )
x_min, x_max = X [ : , 0 ].min () - 1 , X [ : , 0 ].max () + 1
y_min, y_max = X [ : , 1 ].min () - 1 , X [ : , 1 ].max () + 1
xx , yy = np.meshgrid ( np.arange ( x_min , x_max , 0.1 ) , np.arange ( y_min , y_max , 0.1 ) )
Z = np.dot ( np.c_ [ xx.ravel () , yy.ravel () ], w ) - b
Z = np.sign ( Z ).reshape ( xx.shape )
plt.contourf ( xx , yy , Z , alpha = 0.3 )
plt.show ()
```

RESULT

Hence, the SVM algorithm has been implemented successfully.

3/5/23

EXPERIMENT - 9B

AIM

To implement the Convolutional Neural Network (CNN) algorithm in Python.

ALGORITHM

* The data is fed into the model and output from each layer is obtained from the above step is called feedforward.

* We then calculate the error using an error function, some common error functions are cross-entropy, square loss error, etc.,

* After that, we backpropagate into the model by calculating the derivatives. This step is called backpropagation which basically is used to minimize the loss.

THEORY

A CNN is a deep learning neural network consisting of multiple layers like the input layer, convolutional layer, pooling layer and fully connected layers. The network learns the optimal filters through backpropagation and gradient descent.

OUTPUT

```
MNIST CNN initialized
----EPOCH 1 ---
[Step 100] Past 100 steps: Average Loss 2.213 | Accuracy: 17%
[Step 200] Past 100 steps: Average Loss 2.089 | Accuracy: 33%
[Step 300] Past 100 steps: Average Loss 1.982 | Accuracy: 38%
[Step 400] Past 100 steps: Average Loss 1.802 | Accuracy: 59%
[Step 500] Past 100 steps: Average Loss 1.622 | Accuracy: 63%
[Step 600] Past 100 steps: Average Loss 1.339 | Accuracy: 74%
[Step 700] Past 100 steps: Average Loss 1.190 | Accuracy: 69%
[Step 800] Past 100 steps: Average Loss 1.099 | Accuracy: 72%
[Step 900] Past 100 steps: Average Loss 0.891 | Accuracy: 81%
[Step 1000] Past 100 steps: Average Loss 0.664 | Accuracy: 82%
----EPOCH 2 ---
[Step 100] Past 100 steps: Average Loss 0.670 | Accuracy: 83%
[Step 200] Past 100 steps: Average Loss 0.750 | Accuracy: 82%
[Step 300] Past 100 steps: Average Loss 0.554 | Accuracy: 84%
[Step 400] Past 100 steps: Average Loss 0.485 | Accuracy: 87%
[Step 500] Past 100 steps: Average Loss 0.637 | Accuracy: 85%
[Step 600] Past 100 steps: Average Loss 0.547 | Accuracy: 86%
[Step 700] Past 100 steps: Average Loss 0.553 | Accuracy: 80%
[Step 800] Past 100 steps: Average Loss 0.510 | Accuracy: 87%
[Step 900] Past 100 steps: Average Loss 0.766 | Accuracy: 78%
[Step 1000] Past 100 steps: Average Loss 0.518 | Accuracy: 85%
----EPOCH 3 ---
[Step 100] Past 100 steps: Average Loss 0.516 | Accuracy: 84%
[Step 200] Past 100 steps: Average Loss 0.471 | Accuracy: 85%
[Step 300] Past 100 steps: Average Loss 0.452 | Accuracy: 85%
[Step 400] Past 100 steps: Average Loss 0.366 | Accuracy: 88%
[Step 500] Past 100 steps: Average Loss 0.439 | Accuracy: 90%
[Step 600] Past 100 steps: Average Loss 0.497 | Accuracy: 81%
[Step 700] Past 100 steps: Average Loss 0.441 | Accuracy: 87%
[Step 800] Past 100 steps: Average Loss 0.479 | Accuracy: 84%
[Step 900] Past 100 steps: Average Loss 0.287 | Accuracy: 90%
[Step 1000] Past 100 steps: Average Loss 0.450 | Accuracy: 86%
```

CODE

```
import numpy as np
from keras.datasets import mnist

class Conv:

  def __init__(self, num_filters):
    self.num_filters = num_filters
    self.filters = np.random.randn(num_filters, 3, 3)/9

  def iterate_regions(self, image):
    h,w = image.shape
    for i in range(h-2):
      for j in range(w-2):
        im_region = image[i:(i+3), j:(j+3)]
        yield im_region, i, j

  def forward(self, input):
    self.last_input = input
    h,w = input.shape
    output = np.zeros((h-2, w-2, self.num_filters))
    for im_regions, i, j in self.iterate_regions(input):
      output[i, j] = np.sum(im_regions * self.filters, axis=(1,2))
    return output

  def backprop(self, d_l_d_out, learn_rate):
    d_l_d_filters = np.zeros(self.filters.shape)
    for im_region, i, j in self.iterate_regions(self.last_input):
      for f in range(self.num_filters):
        d_l_d_filters[f] += d_l_d_out[i,j,f] * im_region
    self.filters -= learn_rate * d_l_d_filters
    return None

class MaxPool:

  def iterate_regions(self, image):
    h, w, _ = image.shape
    new_h = h // 2
    new_w = w // 2
    for i in range(new_h):
      for j in range(new_w):
        im_region = image[(i*2):(i*2+2), (j*2):(j*2+2)]
        yield im_region, i, j

  def forward(self, input):
    self.last_input = input
    h, w, num_filters = input.shape
    output = np.zeros((h//2, w//2, num_filters))
    for im_region, i, j in self.iterate_regions(input):
      output[i,j] = np.amax(im_region,axis=(0,1))
    return output
```

```
def backprop(self, d_l_d_out):
        d_l_d_input = np.zeros(self.last_input.shape)
        for im_region, i, j in self.iterate_regions(self.last_input):
            h, w, f = im_region.shape
            amax = np.amax(im_region, axis=(0,1))
            for i2 in range(h):
                for j2 in range(w):
                    for f2 in range(f):
                        if(im_region[i2,j2,f2] == amax[f2]):
                            d_l_d_input[i*2+i2, j*2+j2 ,f2] = d_l_d_out[i, j, f2]
                            break
        return d_l_d_input

class Softmax:

    def __init__(self, input_len, nodes):
        self.weights = np.random.randn(input_len, nodes)/input_len
        self.biases = np.zeros(nodes)

    def forward(self, input):
        self.last_input_shape = input.shape
        input = input.flatten()
        self.last_input = input
        input_len, nodes = self.weights.shape
        totals = np.dot(input, self.weights) + self.biases
        self.last_totals = totals
        exp = np.exp(totals)
        return(exp/np.sum(exp, axis=0))

    def backprop(self, d_l_d_out, learn_rate):
        for i, gradient in enumerate(d_l_d_out):
            if(gradient == 0):
                continue
            t_exp = np.exp(self.last_totals)
            S = np.sum(t_exp)
            d_out_d_t = -t_exp[i] * t_exp/ (S**2)
            d_out_d_t[i] = t_exp[i] * (S-t_exp[i]) /(S**2)
            d_t_d_w = self.last_input
            d_t_d_b = 1
            d_t_d_inputs = self.weights
            d_l_d_t = gradient * d_out_d_t
            d_l_d_w = d_t_d_w[np.newaxis].T @ d_l_d_t[np.newaxis]
            d_l_d_b = d_l_d_t * d_t_d_b
            d_l_d_inputs = d_t_d_inputs @ d_l_d_t
            self.weights -= learn_rate * d_l_d_w
            self.biases -= learn_rate * d_l_d_b
            return d_l_d_inputs.reshape(self.last_input_shape)

(train_images,train_labels),(test_images, test_labels) = mnist.load_data()[:1000]
train_images = train_images[:1000]
train_labels = train_labels[:1000]
test_images = test_images[:1000]
test_labels = test_labels[:1000]
```

```
conv = Conv(8)
pool = MaxPool()
softmax = Softmax(13 * 13 * 8, 10)

def forward(image, label):
   out = conv.forward((image/255) - 0.5)
   out = pool.forward(out)
   out = softmax.forward(out)
   loss = -np.log(out[label])
   acc = 1 if(np.argmax(out) == label) else 0
   return out, loss, acc

def train(im, label, lr=0.005):
   out,loss,acc = forward(im, label)
   gradient = np.zeros(10)
   gradient[label] = -1/out[label]
   gradient = softmax.backprop(gradient, lr)
   gradient = pool.backprop(gradient)
   gradient = conv.backprop(gradient, lr)
   return loss, acc

print('MNIST CNN initialized')
for epoch in range(3):
   print('----EPOCH %d ---'%(epoch+1))
   permutation = np.random.permutation(len(train_images))
   train_images = train_images[permutation]
   train_labels = train_labels[permutation]
   loss = 0
   num_correct = 0
   for i, (im, label) in enumerate(zip(train_images, train_labels)):
      if(i>0 and i %100 == 99):
         print('[Step %d] Past 100 steps: Average Loss %.3f | Accuracy: %d%%' %(i + 1, loss / 100,
num_correct))
         loss = 0
         num_correct = 0
      l, acc = train(im, label)
      loss += l
      num_correct += acc
```

RESULT

Hence, the CNN algorithm has been implemented successfully.