**Experiment: -6**

**AIM:** Implementing Deep CNN

Code:

import numpy as np

# Convolutional Layer

class ConvLayer:

def \_\_init\_\_(self, num\_filters, filter\_size):

self.num\_filters = num\_filters

self.filter\_size = filter\_size

self.filters = np.random.randn(num\_filters, filter\_size, filter\_size) / (filter\_size \* filter\_size)

def iterate\_regions(self, image):

h, w = image.shape[:2]

for i in range(h - self.filter\_size + 1):

for j in range(w - self.filter\_size + 1):

yield i, j, image[i:(i + self.filter\_size), j:(j + self.filter\_size)]

def forward(self, input):

self.last\_input = input

h, w = input.shape[:2]

output = np.zeros((h - self.filter\_size + 1, w - self.filter\_size + 1, self.num\_filters))

for i, j, region in self.iterate\_regions(input):

output[i, j] = np.sum(region \* self.filters, axis=(1, 2))

return output

def backward(self, d\_L\_d\_out, learning\_rate):

d\_L\_d\_filters = np.zeros(self.filters.shape)

for i, j, region 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] \* region

self.filters -= learning\_rate \* d\_L\_d\_filters

return None

# Max Pooling Layer

class MaxPoolLayer:

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

self.size = size

def iterate\_regions(self, image):

h, w, \_ = image.shape

new\_h = h // self.size

new\_w = w // self.size

for i in range(new\_h):

for j in range(new\_w):

yield i, j, image[(i \* self.size):(i \* self.size + self.size), (j \* self.size):(j \* self.size + self.size)]

def forward(self, input):

self.last\_input = input

h, w, num\_filters = input.shape

output = np.zeros((h // self.size, w // self.size, num\_filters))

for i, j, region in self.iterate\_regions(input):

output[i, j] = np.amax(region, axis=(0, 1))

return output

def backward(self, d\_L\_d\_out):

d\_L\_d\_input = np.zeros(self.last\_input.shape)

for i, j, region in self.iterate\_regions(self.last\_input):

h, w, f = region.shape

amax = np.amax(region, axis=(0, 1))

for i2 in range(h):

for j2 in range(w):

for f2 in range(f):

if region[i2, j2, f2] == amax[f2]:

d\_L\_d\_input[i \* self.size + i2, j \* self.size + j2, f2] = d\_L\_d\_out[i, j, f2]

break

return d\_L\_d\_input

# Fully Connected (Softmax) Layer

class SoftmaxLayer:

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 - np.max(totals, axis=0))

return exp / np.sum(exp, axis=0)

def backward(self, d\_L\_d\_out, learning\_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 -= learning\_rate \* d\_L\_d\_w

self.biases -= learning\_rate \* d\_L\_d\_b

return d\_L\_d\_inputs.reshape(self.last\_input\_shape)

# Deep CNN

class DeepCNN:

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

self.conv1 = ConvLayer(8, 3) # First conv layer with 8 filters, each 3x3

self.pool1 = MaxPoolLayer(2) # First max pool layer with pool size 2

self.conv2 = ConvLayer(16, 3) # Second conv layer with 16 filters, each 3x3

self.pool2 = MaxPoolLayer(2) # Second max pool layer with pool size 2

self.softmax = SoftmaxLayer(5 \* 5 \* 16, 10) # Fully connected layer to 10 outputs (assuming input size 28x28)

def forward(self, image, label):

out = self.conv1.forward(image)

out = self.pool1.forward(out)

out = self.conv2.forward(out)

out = self.pool2.forward(out)

out = self.softmax.forward(out)

loss = -np.log(out[label])

acc = 1 if np.argmax(out) == label else 0

return out, loss, acc

def train(self, image, label, learning\_rate=0.005):

out, loss, acc = self.forward(image, label)

gradient = np.zeros(10)

gradient[label] = -1 / out[label]

grad\_back = self.softmax.backward(gradient, learning\_rate)

grad\_back = self.pool2.backward(grad\_back)

grad\_back = self.conv2.backward(grad\_back, learning\_rate)

grad\_back = self.pool1.backward(grad\_back)

self.conv1.backward(grad\_back, learning\_rate)

return loss, acc

# Example usage (using MNIST dataset as a placeholder)

def preprocess\_data():

from sklearn.datasets import fetch\_openml

mnist = fetch\_openml('mnist\_784')

X, y = mnist.data, mnist.target

X = X.reshape(-1, 28, 28)

y = y.astype(int)

return X, y

X, y = preprocess\_data()

cnn = DeepCNN()

losses, accuracies = [], []

for epoch in range(1000):

idx = np.random.randint(0, len(X))

image, label = X[idx], y[idx]

loss, acc = cnn.train(image, label)

losses.append(loss)

accuracies.append(acc)

if epoch % 100 == 0:

print(f'Epoch {epoch}, Loss: {np.mean(losses):.4f}, Accuracy: {np.mean(accuracies):.4f}')

losses, accuracies = [], []