60004210210

Amartya Mishra

COMPS – C31

ML Experiment 6

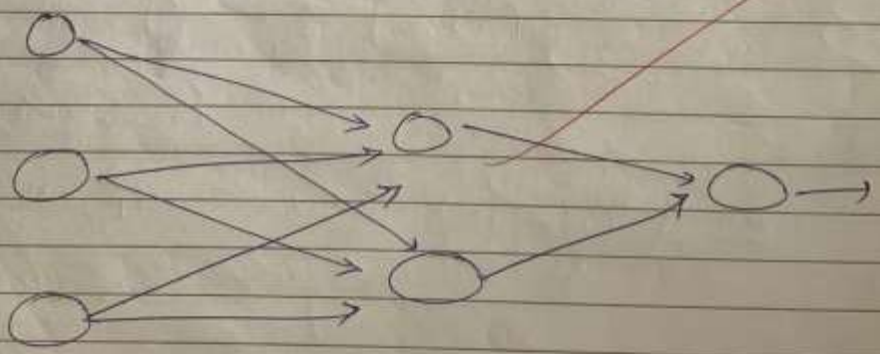
60004210210
Amartya Mishra
COMPS - C31

ML - Experiment 6

Aim To implement Back propogation

Theory
Back propogt^n is an algorithm that Back Propogates the errors from the output nodes to unput nodes
It is widely used Algorithm for training feed forward neural networks
It computes the gradient of loss function with respect to the network weights.

It is very efficient, rather than naively directly computing the gradient concerning each weight



Parameters:
x = inputs training Vector (x_1 x_2 ... x_n)
t = target of vector (t_1 t_2 ... t_k)
δ_k = error at output unit

δ_j : error at hidden layer
α : learning rate
V_{oj} : biase of hidden unit j

conclusion. Thus, we implemented back Propogation

Implementation:

```
 import numpy as np
class NeuralNetwork:
def __init__(self, input_size, hidden_size, output_size):
self.input_size = input_size
self.hidden_size = hidden_size
self.output_size = output_size
# Initialize weights and biases
self.weights_input_hidden = np.random.randn(self.input_size,
self.hidden_size)
self.bias_input_hidden = np.random.randn(1, self.hidden_size)
self.weights_hidden_output = np.random.randn(self.hidden_size,
self.output_size)
self.bias_hidden_output = np.random.randn(1, self.output_size)
def sigmoid(self, x):
return 1 / (1 + np.exp(-x))
def sigmoid_derivative(self, x):
return x * (1 - x)
def forward(self, inputs):
self.hidden_input = np.dot(inputs, self.weights_input_hidden) +
self.bias_input_hidden
self.hidden_output = self.sigmoid(self.hidden_input)
self.final_input = np.dot(self.hidden_output,
self.weights_hidden_output) + self.bias_hidden_output
self.final_output = self.sigmoid(self.final_input)
return self.hidden_output, self.final_output
def backward(self, inputs, targets, learning_rate):
error = targets - self.final_output
delta_output = error * self.sigmoid_derivative(self.final_output)
delta_hidden = np.dot(delta_output, self.weights_hidden_output.T) *
self.sigmoid_derivative(self.hidden_output)
self.weights_hidden_output += np.dot(self.hidden_output.T,
delta_output) * learning_rate
self.bias_hidden_output += np.sum(delta_output, axis=0,
keepdims=True) * learning_rate
self.weights_input_hidden += np.dot(inputs.T, delta_hidden) *
learning_rate
self.bias_input_hidden += np.sum(delta_hidden, axis=0,
keepdims=True) * learning_rate
return error
def train(self, inputs, targets, learning_rate):
hidden_output, final_output = self.forward(inputs)
error = self.backward(inputs, targets, learning_rate)
print("Output of hidden layer:")
print(hidden_output)
print("Output of output layer:")

print(final_output)
print("Error found:")
print(error)
print("Updated weights after 1 iteration:")
print("Weights from input to hidden layer:")
print(self.weights_input_hidden)
print("Weights from hidden to output layer:")
print(self.weights_hidden_output)
```

```
dataset = pd.read_csv('reduced_digits_dataset.csv')
inputs = dataset.drop(columns=['target']).values
targets = dataset['target'].values.reshape(-1, 1)
input_size = inputs.shape[1]
output_size = len(np.unique(targets))
hidden_size = 3
nn = NeuralNetwork(input_size, hidden_size, output_size)
nn.train(inputs, targets, learning_rate=0.1)
```