neural-network-gradient-descent-from-scratch

June 28, 2021

1 Q1 XOR Neural Network with Gradient Descent from Scratch

[1]:
```
import numpy as np
import random as rand
```

[2]:
```
class Layer:
    def __init__(self, layer_neurons, inputs=False, input_bias=False):
        self.layer_neurons = layer_neurons
        self.input_bias = input_bias
        self.inputs = inputs + input_bias  # If bias input would increase by 1
        self.delta_weights = 0
        if inputs:
            self.weights = np.array([[rand.uniform(-1,1) for _ in range(self.
↪layer_neurons)]  for _ in range(self.inputs)])   # (inputs, layer_neurons)␣
↪shape of each layer's weights

    # This method initializes delta_weights
    def initialize_delta_weights(self):
        self.delta_weights = np.array([[0 for _ in range(self.layer_neurons)] ␣
↪for _ in range(self.inputs)])


class NeuralNetwork:
    def __init__(self, inputs):
        self.inputs = inputs
        self.layers = None

    # This method is used to add another layer to neural network, you can use␣
↪it multiple times to add many many layers
    def add_layer(self, layer_neurons, input_bias=False):
        if self.layers is None:  # If first layer
            self.layers = []
            self.layers.append( Layer(layer_neurons, inputs=self.inputs,␣
↪input_bias=input_bias) )
        else:
```

```
            prev_layer_neurons = self.layers[-1].layer_neurons  # New layer's␣
↪input would be from previous layer
            self.layers.append( Layer(layer_neurons, inputs=prev_layer_neurons,␣
↪input_bias=input_bias) )


    # This method returns neuron valyes at a layer
    def neuron_values_at_layer(self, layer_wanted, input_values):
        input_values_copy = input_values.copy()

        if layer_wanted < 0:  # As I have not made a layer object of Input so␣
↪if input layer's neurons are required it would not exist in layers
            if self.layers[0].input_bias:
                input_values_copy.append(1)
            return np.array(input_values_copy)

        for ind_layer, layer in enumerate(self.layers):
            if layer.input_bias:
                input_values_copy.append(1)

            neuron_values = np.matmul(input_values_copy, layer.weights)
#             neuron_values = [self.sigmoid(value) for value in neuron_values] ␣
↪# As this was creating issue and weights were behaving abnormally
            input_values_copy = neuron_values

            if ind_layer == layer_wanted:
                if layer.input_bias:
                    neuron_values = list(neuron_values)
                    neuron_values.append(1)
                return np.array(neuron_values)

    # Calculate sigmoid
    def sigmoid (self, value):
        return 1 / (1 + np.exp(-value))

    # Forward propogate input value to current network with current weights
    def forward_propogate(self, input_values):
        input_values_copy = input_values.copy()
        for layer in self.layers:
            if layer.input_bias:
                input_values_copy.append(1)

            neuron_values = np.matmul(input_values_copy, layer.weights)
            neuron_values = [self.sigmoid(value) for value in neuron_values]
            input_values_copy = neuron_values
        output = self.sigmoid(neuron_values[0])
#         output = neuron_values[0]
```

```
        return output

    # Gradient descent algorithm
    def gradient_descent(self, train_examples, labels, learning_rate, epochs):
        # Step 1: Initialize each weight to small random value
            # Already doing while initalizing layers

        # Step 2: Until Termination condition is met
        for _ in range(epochs):

            # Step 3: Initialize delta_weights to 0
            for layer in self.layers:
                layer.initialize_delta_weights()

            # Step 4: For each individual example in all training examples
            for train_example in train_examples:

                # Step 5: Forward propogate and compute output
                prediction = self.forward_propogate(train_example)

                # Step 6: Compute delta_weights
                for ind, layer in enumerate(self.layers):
                    error_gradient = (labels[ind]-prediction) *␣
 ↪(prediction*(1-prediction)) * self.neuron_values_at_layer(ind-1,␣
 ↪train_example)  # see equation in slides/video lecture for sigmoid
                    layer.delta_weights = ( layer.delta_weights.T +␣
 ↪(learning_rate * error_gradient) ).T

            # Step 7: Update weights using delta_weights
            for layer in self.layers:
#                 print(layer.weights, '-----Updating---')
                layer.weights = layer.weights + layer.delta_weights
#                 print(layer.weights, '\n')
```

[3]:
```
model = NeuralNetwork(inputs=2)
model.add_layer(layer_neurons=2)
model.add_layer(layer_neurons=1)

x = [ [0,0], [0,1], [1,0], [1,1] ]
y = [0, 1, 1, 0]
model.gradient_descent(x, y, 0.01, 3000)
```

[4]:
```
for ind, inp in enumerate(x):
    out = model.forward_propogate(inp)
    print(inp, ' -> ', '{:.9f}'.format(out), '  ', 'actual:', y[ind],␣
 ↪'predicted:', 1 if out>0.5 else 0)
```

```
[0, 0]  ->  0.500000000    actual: 0 predicted: 0
[0, 1]  ->  0.620745754    actual: 1 predicted: 1
[1, 0]  ->  0.621908601    actual: 1 predicted: 1
[1, 1]  ->  0.622459153    actual: 0 predicted: 1
```

[ ]: