CNN Backpropagation

Shorthand: "pd_" as a variable prefix means "partial derivative" "d_" as a variable prefix means "derivative" "_wrt_" is shorthand for "with respect to" "w_ho" and "w_ih" are the index of weights from hidden to output layer neurons and input to hidden layer neurons respectively

Comment references:

[1] Wikipedia article on Backpropagation http://en.wikipedia.org/wiki/Backpropagation#Finding_the_derivative_of_the_error (http://en.wikipedia.org/wiki/Backpropagation#Finding_the_derivative_of_the_error) [2] Neural Networks for Machine Learning course on Coursera by Geoffrey Hinton https://class.coursera.org/neuralnets-2012-001/lecture/39 (https://class.coursera.org/neuralnets-2012-001/lecture/39) [3] The Back Propagation Algorithm https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf (https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf)

Autor: Hermann Völlinger, DHBW Stuttgart; Date: 7.09.2024

In [8]:

```python
# Importing everything we need
import random
import math

# Import libary time to check execution date+time
import time
# print the date & time of the notebook
print('*******************************************************************
print("Actual date & time of the notebook:",time.strftime("%d.%m.%Y %H:%M
print('*******************************************************************

#check versions of libraries
#print('random version is: {}'.format(random.__version__))
#print('math version is: {}'.format(math.__version__))

```

```
************************************************************************
Actual date & time of the notebook: 08.09.2024 17:22:26
************************************************************************
```

In [9]:

```python
# Definition von einme Neuronalen Netz (NN)
class NeuralNetwork:
    LEARNING_RATE = 0.5

    def __init__(self, num_inputs, num_hidden, num_outputs, hidden_layer_
        self.num_inputs = num_inputs

        self.hidden_layer = NeuronLayer(num_hidden, hidden_layer_bias)
        self.output_layer = NeuronLayer(num_outputs, output_layer_bias)

        self.init_weights_from_inputs_to_hidden_layer_neurons(hidden_laye
        self.init_weights_from_hidden_layer_neurons_to_output_layer_neur

    def init_weights_from_inputs_to_hidden_layer_neurons(self, hidden_lay
        weight_num = 0
        for h in range(len(self.hidden_layer.neurons)):
            for i in range(self.num_inputs):
                if not hidden_layer_weights:
                    self.hidden_layer.neurons[h].weights.append(random.ra
                else:
                    self.hidden_layer.neurons[h].weights.append(hidden_la
                weight_num += 1

    def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(se
        weight_num = 0
        for o in range(len(self.output_layer.neurons)):
            for h in range(len(self.hidden_layer.neurons)):
                if not output_layer_weights:
                    self.output_layer.neurons[o].weights.append(random.ra
                else:
                    self.output_layer.neurons[o].weights.append(output_la
                weight_num += 1

    def inspect(self):
        print('------')
        print('* Inputs: {}'.format(self.num_inputs))
        print('------')
        print('Hidden Layer')
        self.hidden_layer.inspect()
        print('------')
        print('* Output Layer')
        self.output_layer.inspect()
        print('------')

    def feed_forward(self, inputs):
        hidden_layer_outputs = self.hidden_layer.feed_forward(inputs)
        return self.output_layer.feed_forward(hidden_layer_outputs)

    # Uses online learning, ie updating the weights after each training
    def train(self, training_inputs, training_outputs):
        self.feed_forward(training_inputs)

        # 1. Output neuron deltas
        pd_errors_wrt_output_neuron_total_net_input = [0] * len(self.out
        for o in range(len(self.output_layer.neurons)):

            # ∂E/∂zⱼ
            pd_errors_wrt_output_neuron_total_net_input[o] = self.output_

        # 2. Hidden neuron deltas
        pd_errors_wrt_hidden_neuron_total_net_input = [0] * len(self.hid
        for h in range(len(self.hidden_layer.neurons)):
```

```

            # We need to calculate the derivative of the error with respe
            # dE/dyⱼ = Σ ∂E/∂zⱼ * ∂z/∂yⱼ = Σ ∂E/∂zⱼ * wᵢⱼ
            d_error_wrt_hidden_neuron_output = 0
            for o in range(len(self.output_layer.neurons)):
                d_error_wrt_hidden_neuron_output += pd_errors_wrt_output_

            # ∂E/∂zⱼ = dE/dyⱼ * ∂zⱼ/∂
            pd_errors_wrt_hidden_neuron_total_net_input[h] = d_error_wrt_

        # 3. Update output neuron weights
        for o in range(len(self.output_layer.neurons)):
            for w_ho in range(len(self.output_layer.neurons[o].weights))

                # ∂Eⱼ/∂wᵢⱼ = ∂E/∂zⱼ * ∂zⱼ/∂wᵢⱼ
                pd_error_wrt_weight = pd_errors_wrt_output_neuron_total_r

                # Δw = α * ∂Eⱼ/∂wᵢ
                self.output_layer.neurons[o].weights[w_ho] -= self.LEARNI

        # 4. Update hidden neuron weights
        for h in range(len(self.hidden_layer.neurons)):
            for w_ih in range(len(self.hidden_layer.neurons[h].weights))

                # ∂Eⱼ/∂wᵢ = ∂E/∂zⱼ * ∂zⱼ/∂wᵢ
                pd_error_wrt_weight = pd_errors_wrt_hidden_neuron_total_r

                # Δw = α * ∂Eⱼ/∂wᵢ
                self.hidden_layer.neurons[h].weights[w_ih] -= self.LEARNI

    def calculate_total_error(self, training_sets):
        total_error = 0
        for t in range(len(training_sets)):
            training_inputs, training_outputs = training_sets[t]
            self.feed_forward(training_inputs)
            for o in range(len(training_outputs)):
                total_error += self.output_layer.neurons[o].calculate_err
        return total_error

```

In [10]:

```
# Definition eines Neuronnalen Layers
class NeuronLayer:
    def __init__(self, num_neurons, bias):

        # Every neuron in a layer shares the same bias
        self.bias = bias if bias else random.random()

        self.neurons = []
        for i in range(num_neurons):
            self.neurons.append(Neuron(self.bias))

    def inspect(self):
        print('Neurons:', len(self.neurons))
        for n in range(len(self.neurons)):
            print(' Neuron', n)
            for w in range(len(self.neurons[n].weights)):
                print('  Weight:', self.neurons[n].weights[w])
            print('  Bias:', self.bias)

    def feed_forward(self, inputs):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.calculate_output(inputs))
        return outputs

    def get_outputs(self):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.output)
        return outputs
```

In [11]:

```
class Neuron:
    def __init__(self, bias):
        self.bias = bias
        self.weights = []

    def calculate_output(self, inputs):
        self.inputs = inputs
        self.output = self.squash(self.calculate_total_net_input())
        return self.output

    def calculate_total_net_input(self):
        total = 0
        for i in range(len(self.inputs)):
            total += self.inputs[i] * self.weights[i]
        return total + self.bias

    # Apply the logistic function to squash the output of the neuron
    # The result is sometimes referred to as 'net' [2] or 'net' [1]
    def squash(self, total_net_input):
        return 1 / (1 + math.exp(-total_net_input))

    # Determine how much the neuron's total input has to change to move c
    #
    # Now that we have the partial derivative of the error with respect t
    # the derivative of the output with respect to the total net input (d
    # the partial derivative of the error with respect to the total net i
    # This value is also known as the delta (δ) [1]
    # δ = ∂E/∂zⱼ = ∂E/∂yⱼ * dyⱼ/dzⱼ
    #
    def calculate_pd_error_wrt_total_net_input(self, target_output):
        return self.calculate_pd_error_wrt_output(target_output) * self.c

    # The error for each neuron is calculated by the Mean Square Error me
    def calculate_error(self, target_output):
        return 0.5 * (target_output - self.output) ** 2

    # The partial derivate of the error with respect to actual output the
    # = 2 * 0.5 * (target output - actual output) ^ (2 - 1) * -1
    # = -(target output - actual output)
    #
    # The Wikipedia article on backpropagation [1] simplifies to the foll
    # = actual output - target output
    #
    # Alternative, you can use (target - output), but then need to add it
    #
    # Note that the actual output of the output neuron is often written a
    # = ∂E/∂yⱼ = -(tⱼ - yⱼ)
    def calculate_pd_error_wrt_output(self, target_output):
        return -(target_output - self.output)

    # The total net input into the neuron is squashed using logistic func
    # yⱼ = φ = 1 / (1 + e^(-zⱼ))
    # Note that where ⱼ represents the output of the neurons in whatever
    #
    # The derivative (not partial derivative since there is only one vari
    # dyⱼ/dzⱼ = yⱼ * (1 - yⱼ)
    def calculate_pd_total_net_input_wrt_input(self):
        return self.output * (1 - self.output)

    # The total net input is the weighted sum of all the inputs to the ne
    # = zⱼ = netⱼ = x₁w₁ + x₂w₂ ...
    #
```

```python
    # The partial derivative of the total net input with respective to a
    # = ∂zⱼ/∂wᵢ = some constant + 1 * xᵢw₁^(1-0) + some constant ... = xᵢ
    def calculate_pd_total_net_input_wrt_weight(self, index):
        return self.inputs[index]
```

In [12]:

```python
##

# Blog post example:

nn = NeuralNetwork(2, 2, 2, hidden_layer_weights=[0.15, 0.2, 0.25, 0.3],
for i in range(10000):
    nn.train([0.05, 0.1], [0.01, 0.99])
    print(i, round(nn.calculate_total_error([[[0.05, 0.1], [0.01, 0.99]]]

# XOR example:

# training_sets = [
#     [[0, 0], [0]],
#     [[0, 1], [1]],
#     [[1, 0], [1]],
#     [[1, 1], [0]]
# ]

# nn = NeuralNetwork(len(training_sets[0][0]), 5, len(training_sets[0][1]
# for i in range(10000):
#     training_inputs, training_outputs = random.choice(training_sets)
#     nn.train(training_inputs, training_outputs)
#     print(i, nn.calculate_total_error(training_sets))

```

```
0 0.291027774
1 0.283547133
2 0.275943289
3 0.268232761
4 0.260434393
5 0.252569176
6 0.244659999
7 0.236731316
8 0.228808741
9 0.220918592
10 0.213087389
11 0.205341328
12 0.197705769
13 0.190204742
14 0.182860503
15 0.175693166
16 0.168720403
17 0.16195725
18 0.155415989
```

In [ ]:

```
#Abschluss
# print current date and time

print("Date & Time:",time.strftime("%d.%m.%Y  %H:%M:%S"))
# end of import test
print ("*** End of Program Backpropagation ***")



```