誤差逆伝播法の応用

2017年6月9日

今回の内容

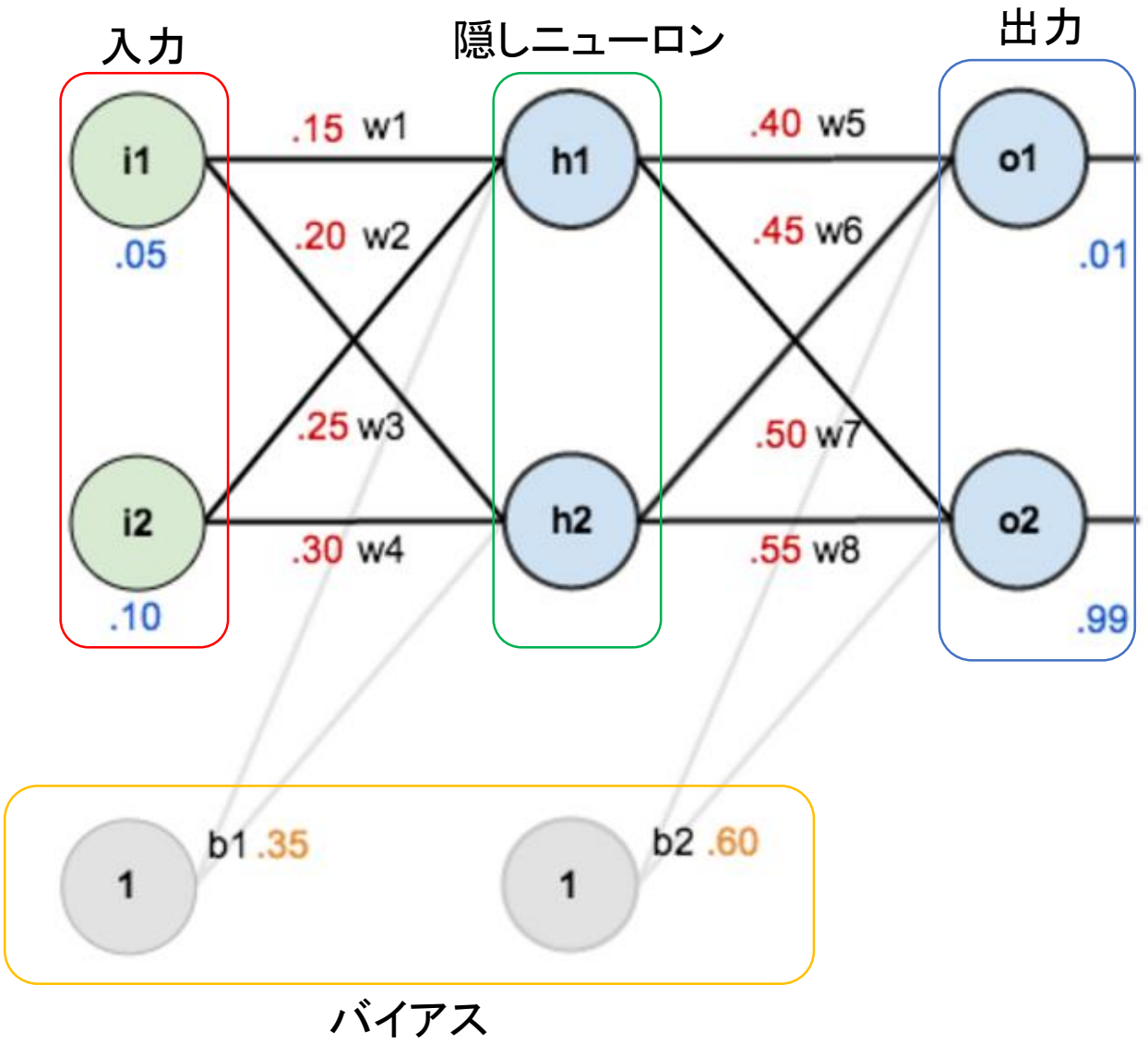
- 誤差逆伝播法応用。
- 誤差逆伝播法コード解説。
- 輪読(4章)。

class NeuralNetwork(init)

```
def __init__(self, num_inputs, num_hidden, num_outputs, hidden_layer_weights = None,
             hidden_layer_bias = None, output_layer_weights = None, output_layer_bias = None):
    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_layer_weights)
    self.init_weights_from_hidden_layer_neurons_to_output_layer_neurons(output_layer_weights)
```

- 入力層、隠れ層、出力層のノード数（今回はどれも2）、隠れ層の重み（w1~w4）、隠れ層のバイアス（b1）、出力層の重み(w5~w8)、出力層のバイアス(b2)を受け取る。
- さらに、hidden_layerとoutput_layerを定義し、その中にはそれぞれに対応するノードとバイアスを入れる。
  層を構成する意味合い。

class NeuralNetwork(init)のイメージ



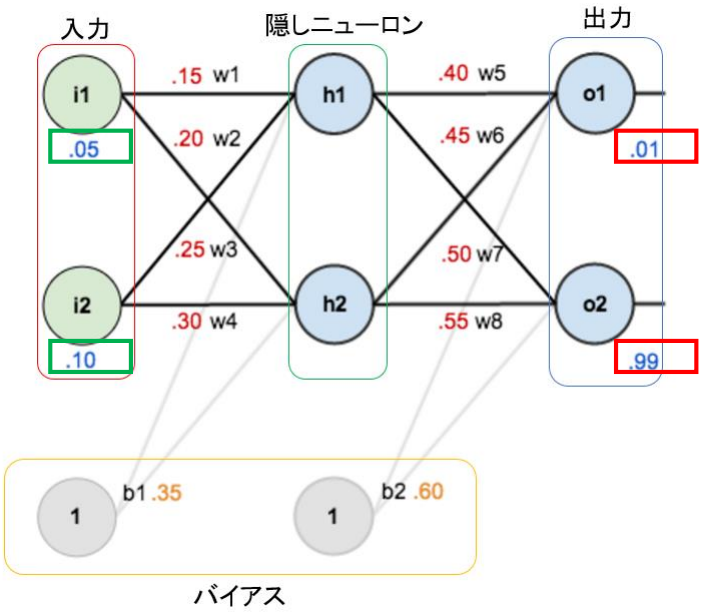
class NeuralNetwork(init_weight関数)

```
def init_weights_from_inputs_to_hidden_layer_neurons(self, hidden_layer_weights):
    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.random())
            else:
                self.hidden_layer.neurons[h].weights.append(hidden_layer_weights[weight_num])
            weight_num += 1

def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self, output_layer_weights):
    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.random())
            else:
                self.output_layer.neurons[o].weights.append(output_layer_weights[weight_num])
            weight_num += 1
```

- 入力層と隠れ層の間の重み(w1~w4)、隠れ層と出力層の間の重み(w5~w8)をそれぞれの関数でinitする。

class NeuralNetwork

```
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)
```



- inspectは呼び出されていない。
- feed_forwardは、出力層における出力（推定値）を出す。
  この値が、学習を重ねるごとに、目標値に近づく。

```
[0.7513650695523157, 0.7729284653214625]
```

⋮

```
[0.015913620443550 68, 0.9840642735146238]
```

```
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]]]), 9))
```

← 入力値と目標値

class NeuralNetwork(train関数)

```
# Uses online learning, ie updating the weights after each training case
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.output_layer.neurons)
    for o in range(len(self.output_layer.neurons)):

        # ∂E/∂zⱼ
        pd_errors_wrt_output_neuron_total_net_input[o] = self.output_layer.neurons[o].calculate_pd_error_wrt_total_net_input(training_outputs[o])

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

        # We need to calculate the derivative of the error with respect to the output of each hidden layer neuron
        # 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_neuron_total_net_input[o] * self.output_layer.neurons[o].weights[h]

        # ∂E/∂zⱼ = dE/dyⱼ * ∂zⱼ/∂
        pd_errors_wrt_hidden_neuron_total_net_input[h] = d_error_wrt_hidden_neuron_output * self.hidden_layer.neurons[h].calculate_pd_total_net_input_wrt_input()
```

- 重み更新の準備。
- 出力層部分と、隠れ層部分とで別の関数を使っている。
- Neuronクラスの関数を呼び出して計算しているため、ここを見ただけでは何をしているか分かりにくい。

class NeuralNetwork(train関数)

```
# Uses online learning, ie updating the weights after each training case
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.output_layer.neurons)
    for o in range(len(self.output_layer.neurons)):

        # ∂E/∂z_j
        pd_errors_wrt_output_neuron_total_net_input[o] = self.output_layer.neurons[o].calculate_pd_error_wrt_total_net_input(training_outputs[o])

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

        # We need to calculate the derivative of the error with respect to the output of each hidden layer neuron
        # dE/dy_j = Σ ∂E/∂z_j * ∂z/∂y_j = Σ ∂E/∂z_j * w_ij
        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_neuron_total_net_input[o] * self.output_layer.neurons[o].weights[h]

        # ∂E/∂z_j = dE/dy_j * ∂z_j/∂
        pd_errors_wrt_hidden_neuron_total_net_input[h] = d_error_wrt_hidden_neuron_output * self.hidden_layer.neurons[h].calculate_pd_total_net_input_wrt_input()
```

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

output

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

hidden

- これら上2つの式の分割部分それぞれを求める。
  一部分は今回スライドp.14、p.15にある関数に投げている。
  (neuronクラスにおける処理)

class NeuralNetwork(train関数続き)

```
# 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)):

        # ∂E/∂w_ho = ∂E/∂z_j * ∂z_j/∂w_ho
        pd_error_wrt_weight = pd_errors_wrt_output_neuron_total_net_input[o] * self.output_layer.neurons[o].calculate_pd_total_net_input_wrt_weight(w_ho)

        # Δw = α * ∂E/∂w_i
        self.output_layer.neurons[o].weights[w_ho] -= self.LEARNING_RATE * pd_error_wrt_weight

# 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)):

        # ∂E/∂w_i = ∂E/∂z_j * ∂z_j/∂w_i
        pd_error_wrt_weight = pd_errors_wrt_hidden_neuron_total_net_input[h] * self.hidden_layer.neurons[h].calculate_pd_total_net_input_wrt_weight(w_ih)

        # Δw = α * ∂E_j/∂w_i
        self.hidden_layer.neurons[h].weights[w_ih] -= self.LEARNING_RATE * pd_error_wrt_weight
```

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1}$$

- 重みの更新に関する部分。
- 出力層部分と、隠れ層部分とで分かれている。

class NeuralNetwork (calculate_total_error関数)

```
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_error(training_outputs[o])
    return total_error
```

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

- 総誤差計算。
- 前回スライドp.15を参照。また、部分ごとの誤差は今回スライドp.13に示すcalculate_errorで行う。

class NeuronLayer

```
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
```

- 層におけるバイアスとノードの設定に関する部分。
- これらによりネットワークが形成される。

class Neuron

```
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))
```

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

- ニューロンにおける入出力計算部分。
- 該当部分は前回スライドp.8(netやoutの計算)。
- Squashは、outputの計算で使用するシグモイド関数。

class Neuron

```
# Determine how much the neuron's total input has to change to move closer to the expected output
#
# Now that we have the partial derivative of the error with respect to the output (∂E/∂yⱼ) and
# the derivative of the output with respect to the total net input (dyⱼ/dzⱼ) we can calculate
# the partial derivative of the error with respect to the total net input.
# 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.calculate_pd_total_net_input_wrt_input();

# The error for each neuron is calculated by the Mean Square Error method:
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 then is calculated by:
# = 2 * 0.5 * (target output - actual output) ^ (2 - 1) * -1
# = -(target output - actual output)
#
# The Wikipedia article on backpropagation [1] simplifies to the following, but most other learning material does not [2]
# = actual output - target output
#
# Alternative, you can use (target - output), but then need to add it during backpropagation [3]
#
# Note that the actual output of the output neuron is often written as yⱼ and target output as tⱼ so:
# = ∂E/∂yⱼ = -(tⱼ - yⱼ)
def calculate_pd_error_wrt_output(self, target_output):
    return -(target_output - self.output)
```

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 =$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1})$$

- 誤差計算。重み更新の材料の計算。今回スライドp.8で呼び出されているもの。
  前回スライドp.15~p.19等が該当部分。

class Neuron

```
# The total net input into the neuron is squashed using logistic function to calculate the neuron's output:
# y_j = φ = 1 / (1 + e^(-z_j))
# Note that where j represents the output of the neurons in whatever layer we're looking at and i represents the layer below it
#
# The derivative (not partial derivative since there is only one variable) of the output then is:
# dy_j/dz_j = y_j * (1 - y_j)
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 neuron and their respective weights:
# = z_j = net_j = x_1w_1 + x_2w_2 ...
#
# The partial derivative of the total net input with respective to a given weight (with everything else held constant) then is:
# = ∂z_j/∂w_i = some constant + 1 * x_iw_1^(1-0) + some constant ... = x_i
def calculate_pd_total_net_input_wrt_weight(self, index):
    return self.inputs[index]
```

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1})$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1}$$

- 重み更新の材料の計算。前回スライドp.20,p.21参照。

$$\varphi(z) = \frac{1}{1 + e^{-z}}$$

which has a nice derivative of:

$$\frac{\partial \varphi}{\partial z} = \varphi(1 - \varphi)$$

⇐

φ = 1/(e^(-z) + 1) = (e^(-z) + 1)^(-1)

∂φ/∂z = e^(-z)/(e^(-z) + 1)^2
= 1/(e^(-z) + 1)(e^(-z)+1-1)/(e^(-z) + 1)
= φ (1- φ)

演習

- ユニットの持つ初期値、重み、バイアス、目標値を変更して検証を行ってみる。
  誤差の変動具合等が考察できる。
- さらに、ユニット数を増やして考察してみる。

3年生輪読予定表改訂版

担当箇所	名前	発表予定日
1-1	保科	5/12
1-3	猪狩	5/12
2-1	村田 西澤	5/19
2-2 2-3	西ヶ谷 坂中	5/19
3-1	米倉 田之井	5/26
3-2	松田 清水	5/26

3年生輪読予定表改訂版

担当箇所	名前	発表予定日
3-3	宿野 前原 村田	6/2
4章	**門木** **山下** **上野** **數見**	**6/9** **6/16**
5章	下窪 黒川 関根 坂本	6/16

次回

- 輪読(4章、5章)
- RNN？
- ２年生は、どのアルゴリズムを選択するか考えてみる。
  作成したい内容の実装上の質問も受け付けます。