Backpropagation

Activation Function

In [1]:
```python
import math
import numpy as np

def linear(x, kwargs=None):
    return x

def sigmoid(x, kwargs=None):
    value = float(1 / (1 + math.exp(x * -1)))
    threshold = kwargs.get("threshold", None)
    if threshold == None:
        return value
    else:
        if value < threshold:
            return 0
        else:
            return 1

def relu(x, kwargs):
    alpha = kwargs.get("alpha", 0.0)
    max_value = kwargs.get("max_value", None)
    threshold = 0
    if x < threshold:
        return max(x, x * alpha)
    else:
        if max_value == None:
            return x
        else:
            return min(x, max_value)

def softmax(arr, kwargs=None):
    arr_exp = np.exp(arr)
    return arr_exp / arr_exp.sum()

def lossDerivative(targetj, oj):
    return oj-targetj

def lossFunction(targetj, oj, lenOutput=1):
    loss = 0
    if lenOutput>1:
        for i in range(len(targetj)):
            for j in range(len(targetj[i])):
#                   print(targetj[i])
                loss += (targetj[i][j]-oj[i][j]) ** 2
    else:
        loss += (targetj-oj) * (targetj-oj)
    return loss/2

def lossSoftmax(pk):
    return -1*math.log(pk)

def reluDerivative(x):
    if x<0:
        return 0
    else:
        return 1

def sigmoidDerivative(x):
    return sigmoid(x)*(1 - sigmoid(x))

def softmaxDerivative(pj, targetClass=False):
    if not targetClass:
        return pj
    else:
        return -1*(1-pj)
```

Chain Rule

In [2]:
```python
def chainRuleOutputSigmoid (target, out_o) :
    return -( target - out_o ) * out_o * ( 1 - out_o )
def chainRuleOutputRelu (target, out_o):
    return -( target - out_o ) * activation.activationFunction.reluDerivative(out_o)
def chainRuleOutput2 (target, out_h, out_o, method) :
    output = chainRuleOutputSigmoid(target, out_o)
    return output * out_h

def chainRuleHidden (arr_target, arr_out_o, arr_hiddenLayer_weight, out_h, vector_i, method)
:
    sum_Output = 0
    if method == "sigmoid":
        for j in range(len(arr_target)):
            arr = []
            output = chainRuleOutputSigmoid(arr_target[j], arr_out_o[j])
            arr.append(output)
            result = np.prod(arr) * arr_hiddenLayer_weight[j]
            sum_Output += result
        return sum_Output * out_h * ( 1 - out_h ) * vector_i

    elif method == "relu":
        for j in range(len(arr_target)):
            arr = []
            output = 1
            arr.append(output)
            result = np.prod(arr) * arr_hiddenLayer_weight[j]
            sum_Output += result
        return sum_Output * out_h * ( 1 - out_h ) * vector_i

    elif method == "relu":
        for j in range(len(arr_target)):
            arr = []
            output = chainRuleOutputRelu(arr_target[j], arr_out_o[j])
            arr.append(output)
            result = np.prod(arr) * arr_hiddenLayer_weight[j]
            sum_Output += result
        return sum_Output * out_h * ( 1 - out_h ) * vector_i

    elif method == "softmax":
        for j in range(len(arr_target)):
            arr = []
            for k in range(len(arr_target[j])):
                output = chainSoftMax(arr_target[j][k], arr_out_o[j][k], activation.activati
onFunction.softmax(arr_out_o[j]), out_h)
                arr.append(output)
            result = np.prod(arr) * arr_hiddenLayer_weight[j]
            sum_Output += result
        return sum_Output * vector_i

def chainSoftMax (target, j, probJ, out_h) :
    if (target == j):
        return -( 1 - probJ ) * out_h
    else:
        return probJ * out_h
```

Parameter Reader

In [3]:
```python
import json

def read_parameter():
    # Opening JSON file
    f = open('parameter.json',)

    # returns JSON object as
    # a dictionary
    parameter = json.load(f)
    # Closing file
    f.close()

    return parameter
```

CSV Reader

In [4]:
```python
import pandas as pd

def read_model():
    df = pd.read_csv('model.csv')
    return df

def read_data():
    df = pd.read_csv('iris.csv')
    df['species'] = df['species'].replace(['setosa'],1)
    df['species'] = df['species'].replace(['versicolor'],2)
    df['species'] = df['species'].replace(['virginica'],3)
    # One hot encoding
    y = pd.get_dummies(df.species, prefix='Class')
#     print(y.head())
    df["Class_1"] = y["Class_1"]
    df["Class_2"] = y["Class_2"]
    df["Class_3"] = y["Class_3"]
#     print(df.head())
    return df
```

Backpropagation + Feed Forward Neural Network

In [5]:
```python
import numpy as np
import math
import random

NEURON_INPUT = 4
# For iterating per item in array. Except for linear function because single parameter is au
tomate to iterate per item
linear = np.vectorize(linear)
sigmoid = np.vectorize(sigmoid)
relu = np.vectorize(relu)

class NeuralNetwork:
    def __init__(self, base_layer, learning_rate=0.001, error_threshold=0.001, max_iter=100,
batch_size=1):
        self.base_layer = base_layer
        self.current_layer = base_layer.copy()
        self.learning_rate = learning_rate
        self.error_threshold = error_threshold
        self.max_iter = max_iter
        self.batch_size = batch_size

    def get_total_layer(self):
        return len(self.layer)

    def enqueue_layer(self, layer):
        self.current_layer.insert(0, layer)

    def deque_layer(self):
        self.current_layer.pop(0)

    def forward_propagation(self):
        for idx in range(len(self.current_layer)):
            if idx != 0:
                self.current_layer[idx].input_value = self.current_layer[idx-1].result
            self.current_layer[idx].compute()

    def draw(self):
        from graphviz import Digraph
        f = Digraph('Feed Forward Neural Network', filename='ann1.gv')
        f.attr('node', shape='circle', fixedsize='true', width='0.9')

        for i in range(len(self.current_layer)):
            if i != 0:
                if i == 1:
                    for j in range(len(self.current_layer[i].weight)):
                        for k in range(len(self.current_layer[i].weight[j])):
                            f.edge(f'x{j}', f'h{i}_{k}', str(
                                self.current_layer[i].weight[j][k]))
                    for j in range(len(self.current_layer[i].bias)):
                        f.edge(f'bx', f'h{i}_{j}', str(
                            self.current_layer[i].bias[j]))
                else:
                    for j in range(len(self.current_layer[i].weight)):
                        for k in range(len(self.current_layer[i].weight[j])):
                            f.edge(f'h{i-1}_{j}', f'h{i}_{k}',
                                   str(self.current_layer[i].weight[j][k]))
                    for j in range(len(self.current_layer[i].bias)):
                        f.edge(f'bhx{i-1}', f'h{i}_{j}',
                               str(self.current_layer[i].bias[j]))

        f.view()

    def learn(self, data):
        # placeholder
        current_iter = 0
        target = []
        result = []
        for _ in range(self.max_iter):
            error = 0.0
            for index, item in data.iterrows():
                # Prepare input
                self.enqueue_layer(InputLayer(
                    [item['sepal_length'], item['sepal_width'], item['petal_length'], item[
'petal_width']]))
                # Forward andd result
                self.forward_propagation()

                target.append(
                    [item['Class_1'], item['Class_2'], item['Class_3']])
                result.append(self.current_layer[-1].result)
                if self.current_layer[-1].activation_function_name == "relu" or self.current
_layer[-1].activation_function_name == "sigmoid" or self.current_layer[-1].activation_functi
on_name == "linear":
                    error += lossFunction(target, result, 3)
                elif self.current_layer[-1].activation_function_name == "softmax":
                    for i in range(len(target)):
                        for j in range(len(target[i])):
                            if target[i][j] != result[i][j]:
                                error += lossSoftmax(result[i][j])
                if error < self.error_threshold:
                    break

                # cleaning layer
                self.deque_layer()

                # Learn with bach_size
                if (index + 1) % self.batch_size == 0 or index == len(data.index):
                    # backpropagation
                    self.back_propagation(target, result)
                    # clearing list and error foreach batch_size
                    target.clear()
                    result.clear()
                    error = 0

            if current_iter < self.max_iter:
                break

    def back_propagation(self, arr_target, arr_out):
        for i in range(len(self.current_layer) - 1, -1, -1):
            if i != len(self.current_layer) - 2 and i > 0:  # Not input or output layer
                for j in range(len(self.current_layer[i].weight)):
                    for k in range(len(self.current_layer[i].weight[j])):
                        self.current_layer[i].weight[j][k] = self.current_layer[i].update_we
ight(arr_target, arr_out,

self.current_layer[i].weight[j], self.current_layer[i].result[j], self.current_layer[i].inpu
t_value[j], self.learning_rate)
                for j in range(len(self.current_layer[i].bias)):
                    self.current_layer[i].bias = self.current_layer[i].update_bias(arr_targe
t, arr_out,
                                                                                      self.curr
ent_layer[i].bias, self.current_layer[i].result[j], np.array([1 for x in range(len(self.curr
ent_layer[i].bias))]), self.learning_rate)
            elif i == len(self.current_layer):
                self.current_layer[i].weight = self.current_layer[i].update_weight_output(ar
r_target, arr_out,
                                                                                           se
lf.current_layer[i].weight, self.current_layer[i].result[j], self.current_layer[i].input_val
ue[j], self.learning_rate)

    def predict(self, data):
        result = []
        target = []
        precise = 0
        total_data = len(data.index)
        for index, item in data.iterrows():
            # Prepare input
            self.enqueue_layer(InputLayer(
                [item['sepal_length'], item['sepal_width'], item['petal_length'], item['peta
l_width']]))

            # Forward andd result
            self.forward_propagation()
            target.append([item['Class_1'], item['Class_2'], item['Class_3']])
            result.append(self.current_layer[-1].result)
            max_index_col_result = np.argmax(result[-1], axis=0)
            max_index_col_data = np.argmax(target[-1], axis=0)
            if(max_index_col_data == max_index_col_result):
                precise = precise + 1
            self.deque_layer()
        accuracy = 0.0
        accuracy = float(precise / total_data)
        print("Accuracy \t: ", accuracy)


class InputLayer:
    def __init__(self, arr=[]):
        self.input_value = np.array(arr)
        self.result = self.input_value

    def compute(self):
        pass


class Layer(InputLayer):
    def __init__(self, neuron_input, neuron_output, activation_function, activation_function
_name, **kwargs):
        super().__init__([])
        self.weight = np.array([[1.5 * (1.0 - random.random())
                                 for x in range(neuron_output)] for j in range(neuron_input
)])
        self.bias = np.array([1.5 * (1.0 - random.random())
                              for x in range(neuron_output)])
        self.result = np.array([])
        self.activation_function = activation_function
        self.activation_function_name = activation_function_name
        self.kwargs = kwargs

    def activate(self):
        self.result = self.activation_function(self.result, self.kwargs)

    def sigma(self):
        # case 1 Dimension
        if(len(self.weight[0]) == 1):
            self.result = np.matmul(
                self.input_value, self.weight.flatten()) + self.bias
        else:
            self.result = np.matmul(self.input_value, self.weight) + self.bias
#         print("Sigma \t: ", self.result)

    def compute(self):
#         print("Input \t: ", self.input_value)
        self.sigma()
        self.activate()
#         print("Weight \t: ", self.weight)
#         print("Result \t: ", self.result)

    def update_weight(self, arr_target, arr_out_o, arr_hiddenLayer_weight, out_h, vector_i,
learning_rate):
        return chainRuleHidden(arr_target, arr_out_o, arr_hiddenLayer_weight, out_h, vector_
i, self.activation_function_name) * learning_rate * -1

    def update_weight_output(self, arr_target, arr_out_o, arr_hiddenLayer_weight, out_h, vec
tor_i, learning_rate):
        return chainRuleOutput2(arr_target, arr_out_o, arr_hiddenLayer_weight, out_h, vector
_i, self.activation_function_name) * learning_rate * -1

    def update_bias(self, arr_target, arr_out_o, arr_hiddenLayer_weight, out_h, vector_i, le
arning_rate):
        return chainRuleHidden(arr_target, arr_out_o, arr_hiddenLayer_weight, out_h, vector_
i, self.activation_function_name) * learning_rate * -1


class OutputLayer(Layer):
    def __init__(self, neuron, activation_function, **kwargs):
        super().__init__(neuron, activation_function, **kwargs)
        self.error([])

def main():
    parameter = read_parameter()
    data = read_data()
    model = read_model()
    layer = []
    learning_rate, error_threshold, max_iter, batch_size = \
        parameter["learning_rate"], parameter["error_threshold"], parameter["max_iter"], par
ameter["batch_size"]

    # Create base layer
#     print("Activation Layer: ")
    for index, item in model.iterrows():
        act = None
        if (item['activation'] == 'sigmoid'):
            act = sigmoid
        elif (item['activation'] == 'linear'):
            act = linear
        elif (item['activation'] == 'relu'):
            act = relu
        elif (item['activation'] == 'softmax'):
            act = softmax

        # Case for near Input Layer or the Output Layer
        if index == 0:
            layer.append(
                Layer(NEURON_INPUT, item['neuron'], act, item['activation'], threshold=0.1))
        elif index > 0 and index != len(model.index):
            layer.append(Layer(
                model.iloc[index - 1, 0], item['neuron'], act, item['activation'], threshold
=0.1))
        elif index == model.index:
            layer.append(OutputLayer(
                model.iloc[index - 1, 0], item['neuron'], act, item['activation'], threshold
=0.1))

    print("")

    # Build ANN model from layer and learn process
    neural_network = NeuralNetwork(
        layer, learning_rate, error_threshold, max_iter, batch_size)
    neural_network.learn(data)

    neural_network.predict(data)
    neural_network.draw()


if __name__ == "__main__":
    print("Learning Iris Dataset using Backpropagation + Feed Forward Neural Network")
    print("")
    print(read_model())
    print("")
    print("-Learning rate : " + str(read_parameter()["learning_rate"]))
    print("-Error threshold : " + str(read_parameter()["error_threshold"]))
    print("-Maximum iteration : " + str(read_parameter()["max_iter"]))
    print("-Batch Size : " + str(read_parameter()["batch_size"]))
    main()
```

```
Learning Iris Dataset using Backpropagation + Feed Forward Neural Network

   neuron activation
0       3    sigmoid
1       3    sigmoid
2       3    sigmoid

-Learning rate : 0.05
-Error threshold : 0.05
-Maximum iteration : 100
-Batch Size : 2

Accuracy        :  0.3333333333333333
```