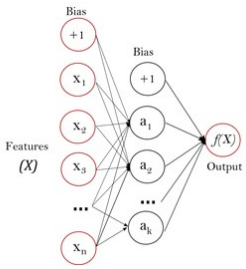

Multilayer Perceptron

by Jesse P. Gutierrez Jr UHD, Data Science

A multilayer perceptrons(MLP) is a finite directed acyclic graph. The nodes are logistically activated. The nodes are neurons with logistic activation. A multilayer perceptron (MLP) is a deep, artificial neural network. It is composed of more than one perceptron. They are composed of an input layer to receive the signal, an output layer that makes a decision or prediction about the input, and in between those two, an arbitrary number of hidden layers that are the true computational engine of the MLP. MLPs with one hidden layer are capable of approximating any continuous function.

Multilayer perceptrons are often applied to supervised learning problems. They train on a set of input-output pairs and learn to model the correlation (or dependencies) between those inputs and outputs. Training involves adjusting the parameters, or the weights and biases, of the model in order to minimize error. Backpropagation is used to make those weigh and bias adjustments relative to the error, and the error itself can be measured in a variety of ways, including by root mean squared error (RMSE).

In the forward pass, the signal flow moves from the input layer through the hidden layers to the output layer, and the decision of the output layer is measured against the ground truth labels. In the backward pass, using backpropagation and the chain rule of calculus, partial derivatives of the error function w.r.t. the various weights and biases are back-propagated through the MLP. That act of differentiation gives us a gradient, or a landscape of error, along which the parameters may be adjusted as they move the MLP one step closer to the error minimum. This can be done with any gradient-based optimisation algorithm such as stochastic gradient descent. The network keeps playing that game of tennis until the error can go no lower. This state is known as convergence.

In [31]:
```
import CSV

iris = CSV.read("iris_data.csv")    ## NOTE "CSV" MUST BE CAPITALIZED
println(iris)
```

150×5 DataFrames.DataFrame

Row	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64?	Float64?	Float64?	Float64?	String?
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
17	5.4	3.9	1.3	0.4	setosa
18	5.1	3.5	1.4	0.3	setosa
19	5.7	3.8	1.7	0.3	setosa
20	5.1	3.8	1.5	0.3	setosa
21	5.4	3.4	1.7	0.2	setosa
22	5.1	3.7	1.5	0.4	setosa
23	4.6	3.6	1.0	0.2	setosa
24	5.1	3.3	1.7	0.5	setosa
25	4.8	3.4	1.9	0.2	setosa
26	5.0	3.0	1.6	0.2	setosa
27	5.0	3.4	1.6	0.4	setosa
28	5.2	3.5	1.5	0.2	setosa
29	5.2	3.4	1.4	0.2	setosa
30	4.7	3.2	1.6	0.2	setosa
31	4.8	3.1	1.6	0.2	setosa
32	5.4	3.4	1.5	0.4	setosa
33	5.2	4.1	1.5	0.1	setosa
34	5.5	4.2	1.4	0.2	setosa
35	4.9	3.1	1.5	0.2	setosa
36	5.0	3.2	1.2	0.2	setosa
37	5.5	3.5	1.3	0.2	setosa
38	4.9	3.6	1.4	0.1	setosa
39	4.4	3.0	1.3	0.2	setosa
40	5.1	3.4	1.5	0.2	setosa
41	5.0	3.5	1.3	0.3	setosa
42	4.5	2.3	1.3	0.3	setosa
43	4.4	3.2	1.3	0.2	setosa
44	5.0	3.5	1.6	0.6	setosa
45	5.1	3.8	1.9	0.4	setosa
46	4.8	3.0	1.4	0.3	setosa
47	5.1	3.8	1.6	0.2	setosa
48	4.6	3.2	1.4	0.2	setosa
49	5.3	3.7	1.5	0.2	setosa
50	5.0	3.3	1.4	0.2	setosa
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
53	6.9	3.1	4.9	1.5	versicolor
54	5.5	2.3	4.0	1.3	versicolor
55	6.5	2.8	4.6	1.5	versicolor
56	5.7	2.8	4.5	1.3	versicolor
57	6.3	3.3	4.7	1.6	versicolor
58	4.9	2.4	3.3	1.0	versicolor
59	6.6	2.9	4.6	1.3	versicolor
60	5.2	2.7	3.9	1.4	versicolor

In [37]:
```
X = zeros(4, 150)
Y = zeros(3, 150)

for i = 1:150
    for j = 1:4
        X[j, i] = iris[i, j]
        if iris[i , 5] == "setosa"
            Y[1, i] = 1.0
        elseif iris[i, 5] == "versicolor"
            Y[2, i] = 1.0
        else
            Y[3, i] = 1.0
        end
    end
end
```

In [19]:
```
iris[90,:]
```

Out[19]: DataFrameRow

1 rows × 5 columns

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64	Float64	Float64	Float64	String
90	5.5	2.5	4.0	1.3	versicolor

In [20]:
```
X[:,90]      # going to compare with the 90th column
```

Out[20]:
```
4-element Array{Float64,1}:
 5.5
 2.5
 4.0
 1.3
```

In [21]:
```
Y[:,90]    # look at the 90 row from Y
```

Out[21]:
```
3-element Array{Float64,1}:
 0.0
 1.0
 0.0
```

Activation is the sigmoid function σ(s) = 1/(1+exp(-s))

In [38]:
```
# Define sigmoid function and its derivative
σ(s) = 1/(1+exp(-s))
dσ(s) = σ(s)*(1 - σ(s))

# Define softmax function
softmax(a, i) = exp(a[i])/(sum(exp(a[j]) for j = 1:length(a)))

# Define cross-entropy loss function
L(O, y) = -sum(y[i]*log(O[i]) for i = 1:length(y))

# Define Hadamard Product
hadamard(x,y) = [x[i]*y[i] for i = 1:length(x)];
```

In [39]:
```
function forward_propagation(x, y, W, b)
    a1 = copy(x)
    z2 = W[1]*a1 + b[1]
    a2 = σ.(z2)

    z3 = W[2]*a2 + b[2]
    a3 = σ.(z3)

    z4 = W[3]*a3 + b[3]
    a4 = σ.(z4)

    a = [a1, a2, a3, a4]
    z = [[0.0], z2, z3, z4]
    O = [softmax(a4, i) for i = 1:length(a4)]
    loss = L(O, y)
    return a, z, O, loss
end
```

Out[39]:
```
forward_propagation (generic function with 1 method)
```

In [19]:
```
forward_propagation(X[:,1], Y[:,1], W, b)
```

Out[19]:
```
(Array{Float64,1}[[5.1, 3.5, 1.4, 0.2], [0.518936, 0.999229, 0.998986, 0.981069,
0.975397], [0.810219, 0.820909, 0.658624, 0.736097, 0.732109], [0.828315, 0.5670
58, 0.699447]], Array{Float64,1}[[0.0], [0.0757783, 7.16726, 6.89309, 3.94783, 3
.67996], [1.45143, 1.52251, 0.657169, 1.02578, 1.00535], [1.57374, 0.269857, 0.8
44666]], [0.377476, 0.290688, 0.331836], 0.9742471683443287)
```

In [40]:
```
function backpropagation(x, y, W, b)
    a, z, O, loss = forward_propagation(x, y, W, b)
    δ4 = a[4] - y
    δ3 = hadamard(W[3]'*δ4, dσ.(z[3]))
    δ2 = hadamard(W[2]'*δ3, dσ.(z[2]))
    δ = [[0.0], δ2, δ3, δ4]
    return a, δ
end

function ∇L(x, y, W, b)

    a, δ = backpropagation(x, y, W, b)

    db1 = copy(δ[2])
    db2 = copy(δ[3])
    db3 = copy(δ[4])

    dW1 = δ[2]*a[1]'
    dW2 = δ[3]*a[2]'
    dW3 = δ[4]*a[3]'
    return [db1, db2, db3], [dW1, dW2, dW3]
end


function gradient_descent!(x, y, W, b, α)
    db, dW = ∇L(x, y, W, b)
    for i = 1:length(W)
        W[i] -= α*dW[i]
        b[i] -= α*b[i]
    end
end
```

Out[40]:
```
gradient_descent! (generic function with 1 method)
```

In [41]:
```
function mini_batch_∇L(train_data, train_label, W, b, m)

    i = rand(1:100)
    a, δ = backpropagation(train_data[:,i], train_label[:,i], W, b)

    db1 = δ[2]
    db2 = δ[3]
    db3 = δ[4]

    dW1 = δ[2]*a[1]'
    dW2 = δ[3]*a[2]'
    dW3 = δ[4]*a[3]'

    for _ in 1:m
        j = rand(1:100)
        a, δ = backpropagation(train_data[:,j], train_label[:,j], W, b)

        db1 += copy(δ[2])
        db2 += copy(δ[3])
        db3 += copy(δ[4])

        dW1 += δ[2]*a[1]'
        dW2 += δ[3]*a[2]'
        dW3 += δ[4]*a[3]'
    end

    return [db1/m, db2/m, db3/m], [dW1/m, dW2/m, dW3/m]
end
```

Out[41]:
```
mini_batch_∇L (generic function with 1 method)
```

In [42]:
```
function stochastic_gradient_descent!(train_data, train_label, W, b, α, m)
    db , dW = mini_batch_∇L(train_data, train_label, W, b, m)
    for i = 1:length(W)
        W[i] -= α*dW[i]
        b[i] -= α*b[i]
    end
end
```

Out[42]:
```
stochastic_gradient_descent! (generic function with 1 method)
```

In [43]:
```
# Initialize weight matrices
W1 = rand(5, 4)
W2 = rand(5, 5)
W3 = rand(3, 5)
W = [W1, W2, W3]

# Initialize bias
b1 = -1*ones(5)
b2 = -1*ones(5)
b3 = -1*ones(3)
b = [b1, b2, b3]
```

Out[43]:
```
3-element Array{Array{Float64,1},1}:
 [-1.0, -1.0, -1.0, -1.0, -1.0]
 [-1.0, -1.0, -1.0, -1.0, -1.0]
 [-1.0, -1.0, -1.0]
```

In [44]:
```
function make_prediction(i)
    output = forward_propagation(X[:,i], Y[:,i], W, b)[3]
    println("      setosa       |     versicolor       |     virginica")
    println("-----------------------------------------------------------------")
    println(output[1]," | ", output[2], "  |  ", output[3])
end
```

Out[44]:
```
make_prediction (generic function with 1 method)
```

In [45]:
```
for _ in 1:1000000
    j = rand(1:150)
    gradient_descent!(X[:,j], Y[:,j], W, b, 0.37)
end
#verified up to 35:39
```

In [46]:
```
make_prediction(12)
```

```
      setosa       |     versicolor       |     virginica
-----------------------------------------------------------------
0.5761150276183369 | 0.21194258042827416  |  0.21194239195338885
```