One-layer perceptron

Jonatan Hellgren

October 2021

1 Training perceptron

```
using Plots
using Distributions
using DelimitedFiles

# The main function initalizes by loading the the data and constructing the model.
# Then it trains the model for 1000 epochs and with a mini-batch of size 10. The
# traingin will be stopped when the desired accuracy is reached. When it is done
# we will plot the results and save the matrices containing the parameters.
function main()
    X_train, X_val = load_data()
    m1 = 100
    g = tanh
    function g_prime(x) return 1 - tanh(x)^2 end
    pelle = init_perceptron([2, m1, 1], g, g_prime, 0.01)

    train(pelle, X_train, X_val, 1000, 10)

    scatter(X_val[:,1], X_val[:,2], color = heavy_side.(predict(pelle, X_val)))
    # writedlm("/home/jona/NN/homework2/w1.csv", pelle.W[1], ',')
    # writedlm("/home/jona/NN/homework2/w2.csv", pelle.W[2]', ',')
    # writedlm("/home/jona/NN/homework2/t1.csv", pelle.[1], ',')
    # writedlm("/home/jona/NN/homework2/t2.csv", pelle.[2], ',')
end

# We load the data here and also normalizes it before returning.
function load_data()
    training_path = "/home/jona/NN/homework2/training_set.csv"
    X_train = read_csv(training_path)

    validation_path = "/home/jona/NN/homework2/validation_set.csv"
    X_val = read_csv(validation_path)

    normalize_data(X_train, X_val)
```

```
    return X_train, X_val
end

# This is a helper function which normalizes the data, it takes as input two
# matrices and normalizes both of them acording to the the mean and standard
# deviation in the first matrix. Hard coded to only work with two matrices.
function normalize_data(df1, df2)
    dim = size(df1)[2] - 1
    for it in 1:dim
         = mean(df1[:,it])
         = std(df1[:,it])
        df1[:,it] = df1[:,it] .-
        df2[:,it] = df2[:,it] .-
        df1[:,it] = df1[:,it] /
        df2[:,it] = df2[:,it] /
    end
end

# I couldn't find a good csv reader in Julia so I did my own, it is however
# hard-coded for this problem specificly. It takes a file path as input and
# returns a matrix containging the values in that file.
function read_csv(path)
    open(path) do f
        lines = readlines(f)
        sz = length(lines)
        M = zeros(sz, 3)
        for (ind, line) in enumerate(lines)
            x1, x2, y = split(line, ',')
            M[ind, :] = [parse(Float64, x1), parse(Float64, x2), parse(Int, y)]
        end
        return M
    end
end

# A function used in homework 1 that found its place in this script since it
# made the plotting possible
function heavy_side(n)
    return n > 0 ? 1 : 0
end

main()
```

2 Perceptron

```
using Distributions
using Plots

# This struct object is all the parameters in the perceptron model, so we can
# consider it to be the model
mutable struct perceptron
    W::Any # 3-d weight vector
    W::Any
    V::Any # 2-d neuron values matrix
    B::Any # 2-d local field matrix
    ::Any # 2-d bias matrix
    ::Any
    ::Any # 2-d matrix where the error term is stored
    ::Any # learning rate
    g::Any # activation function
    g_prime::Any
    dim::Any
end

# This function makes it easier to create a perceptron struct. It is even
# possible to create multi-layered perceptrons here. It takes a vector with the
# dimensions, a activation function, the activation functions derivative and a
# learning rate as input and outputs the desired struct.
function init_perceptron(dimensions, activation_function, derivative, )
    dim = length(dimensions)

    W = [randn(dimensions[2], dimensions[1])] #./ dimensions[2]
    W = [randn(dimensions[2], dimensions[1])] #./ dimensions[2]
     = [zeros(dimensions[2])]
     = [zeros(dimensions[2])]
     = [zeros(dimensions[2])]
    for ind in 2:dim-1
        push!(W, randn(dimensions[ind+1], dimensions[ind])) #./ dimensions[ind+1]
        push!(W, randn(dimensions[ind+1], dimensions[ind])) #./ dimensions[ind+1]
        push!(, zeros(dimensions[ind+1]))
        push!(, zeros(dimensions[ind+1]))
        push!(, zeros(dimensions[ind+1]))
    end

    V = [zeros(dimensions[1])]
    B = [zeros(dimensions[1])]
    for ind in 2:dim
        push!(V, zeros(dimensions[ind]))
        push!(B, zeros(dimensions[ind]))
```

```
    end

    g = activation_function
    g_prime = derivative
    model = perceptron(W, W, V, B, , , , , g, g_prime, dim)
    return model
end

# This function trains the perceptron on the data in X_train for a certain
# number of epochs and batch_size, after every epoch we evaluate our model on
# the validation data and stop the training if the accuracy is less then 0.118
function train(this, X_train, X_val, epochs, batch_size)
    for it in 1:epochs
        fit(this, X_train, batch_size)
        train_score = score(this, X_train)
        val_score = score(this, X_val)
        println(it, "; C_train = ", train_score, ",C_val = ", val_score)
        if val_score < 0.118
            break
        end
    end
end

# This function takes a perceptron struct a matrix X and an integer batch_size
# as input and fits the perceptron to the data in X.
function fit(this, X, batch_size)
    _max = size(X)[1]
    for batch in 1:(_max/batch_size)
        for _ in 1:batch_size
             = sample(1:_max, 1)[1]
            t = X[, 3]
            forward_propegate(this, X[, 1:2])
            back_propegate(this, t)
        end
        update_network(this)
    end
end

# This functions performs a forward propegation for a given coordinate, updates
# all the local fields and neurons
function forward_propegate(this, X)
    this.V[1] = X#[1:this.dim]
    for (ind, w) in enumerate(this.W)
        this.B[ind+1] = w * this.V[ind] .- this.[ind]
        this.V[ind+1] = this.g.(this.B[ind+1])
    end
```

```
end

# This function performs a backwards propegation, i.e it computes all the  for
# all layers and updates the weight increments
function back_propegate(this, t)
    this.[end] = (t .- this.V[end]) .* this.g_prime.(this.B[end])
    for ind in (this.dim-1):2
        this.[ind-1] = (this.W[ind]' * this.[ind]) .* this.g_prime.(this.B[ind])
    end
    for (ind, ) in enumerate(this.)
        this.W[ind] += this. .* ( * this.V[ind]')
        this.[ind] -= this. .*
    end
end

# This function adds the weight increments to the parameter matrices and resets
# the increment matrices afterwards
function update_network(this)
    for ind in 1:this.dim-1
        this.W[ind] += this.W[ind]
        this.[ind] += this.[ind]
    end
    this.W -= this.W
    this. -= this.
end

# Here we take a matrix as input and let the model predict on it. We later
# return the predictions.
function predict(this, X)
    output = []
    sz = size(X)[1]
    for ind in 1:sz
        forward_propegate(this, X[ind, 1:2])
        push!(output, sign.(this.V[end][1]))   # !push appends
    end
    return output
end

# This function we use to score our model, it returns the accuracy for the
# predictions
function score(this, X)
    output = predict(this, X)
    target = X[:,end]
    p_val = length(output)
    total = 0
    for (o, t) in zip(output, target)
```

```
        total += abs(o - t)
    end
    return total / (2 * p_val)
end
```