Homework 2

Jonatan Hellgren

October 2021

1 Three dimensional Boolean function

To begin answering this question we need to first define three relations that an arbitrary point can have to the other 7 points in a cube. The first one I will call *directly connected*, three of these exists and it is the closest points. The second one is *diagonally connected*, there are also three of these and they can be found by crossing a the cubes sides. The last and final one I will call *long diagonally connected*, now with this one it only exists one and it is the one where the connection goes through the cubes centre.

To enumerate all linearly separable functions we only need to figure out the amount of linearly separable functions for $k = \{0, 1, 2, 3, 4\}$, where k is the amount of points that is labeled as 1. We may reduce the problem in such a way due to symmetry, the amount of linearly inseparable functions for $k = 1$ will be the same for $k = 7$ since we get the same pattern but only reversed.

k=0 This case only yields one possibility and that pattern is linearly separable.

k=1 In this case we have 8 possible patterns, of which all are linearly separable.

k=2 Now we have $8!/6!2! = 28$ possible patterns. If the two positive points in a cube are *directly connected* the pattern will be linearly separable. However if they are *diagonally connected* or *long diagonally connected* the pattern is not linearly separable. So in this case we have 12 linearly separable patterns since a cube consists of 12 straight lines.

k=3 Here we get $8!/5!3! = 56$ possible patterns. Of these we can find three different symmetries. The first one we get by choosing to connect an arbitrary point to two of it's *directly connected* neighbors. This pattern is linearly separable and we can calculate that there exists $3 \cdot 8 = 24$ since each of the eight points has three possibilities for creating this symmetry. The second one we get by choosing to connect an arbitrary point with two other points that are *diagonally connected*, this symmetry is not linearly separable. The third and final one we get by connecting an arbitrary point with the point which it is *long diagonally connected* with and then choosing the third point arbitrarily, the third one doesn't really matter. The third symmetry is also not linearly separable. So we get 24 linearly separable for $k = 3$.

k=4 In this situation we have $8!/4!4! = 70$ possible functions. There are multiple symmetries which is quite hard to define without having two different symmetries overlapping each other, so I will only explain the two symmetries that gives us linearly separable functions. The first one we get by connecting an arbitrary point to all of its *directly connected* points, this pattern basically forms an edge in the cube. The second one we get by first choosing an arbitrary point and the choosing two *directly connected* to the first one and lastly choosing the point that is directly connected to both of the points chosen in the second step. This symmetry becomes the side of a cube. Since a cube have six edges and eight sides we can conclude that we have $6 + 8 = 14$ linearly separable functions when $k = 4$.

Conclusion

Now we may conclude that there exists:

$$2 \cdot 1 + 2 \cdot 8 + 2 \cdot 12 + 2 \cdot 24 + 14 = 104$$

linearly separable Boolean functions in three dimensions.

2 Restricted Boltzmann machine

2.1 Results

In Figure 1 we can see the resulting estimate of the Kullback-Leibler divergence after training a Boltzmann machine using the CD-K algorithm with k=100. The results are created by performing the CD-K algorithm a thousand times then computing the divergence, repeating this procedure ten times for each value of M and then averaging the divergence.

To estimate the Boltzmann probabilities well I sampled a random pattern containing a three Boolean variables and the letting the Boltzmann Machine iterate over this pattern a thousand times and recording the frequencies. The random sampling for the initial pattern was also performed a thousand times.

The batch sizes used in the CD-K algorithm was the following: $[4, 4, 20, 25]$, where the first value corresponds to the first value in M, the second one corresponding to the second value in M and so on.

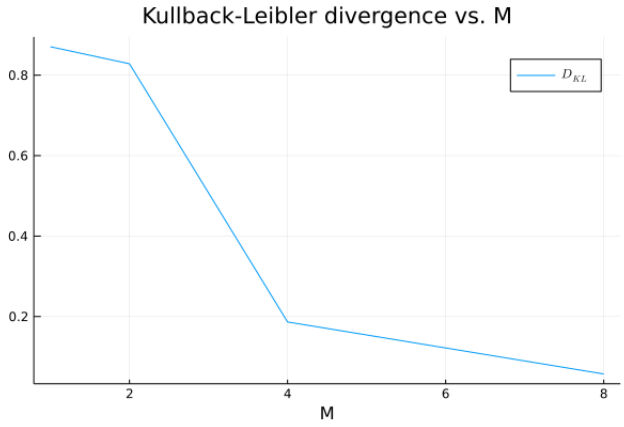

Figure 1: Here we can see the KL divergence for different values for M.

2.2 Discussion

As we can see in the figure the KL-divergence gets lower when increasing the amount of hidden neurons. The reason why it is quite high for $M = 1, 2$ is that the theory states that a Boltzamann machine needs $2^N/2 - 1$ hidden neurons to be able to approximate a N-dimensional distribution. In our case the sufficient amount of neurons would be 3 which is higher then both 1 and 2. We can also see that the Boltzmann machine is able to approximate the data distribution well when using 4 or 8 hidden neurons.

When training the Boltzmann machine we noticed that the mini-batch size used during CD-K algorithm had a high influence on the results. So the results could be considered arbitrary since we didn't get any clear information what batch sizes to use.

2.3 Code

```
using Distributions
using Plots


# The main function does all we need to do to answer the question. We compute
# the average Kullback-Leibler divergence for certain values of M. When it is
# done it prints out the reults plot a nice figure and saves that figure.
function main()
    X_train, X_val = load_patterns()
    p_data = [0.25, 0.25, 0.25, 0.25]
    D_KL = zeros(4)
    iterations = 10
    M = [1 2 4 8]
    batch_size = [4 4 20 25]

    for (ind, m) in enumerate(M)
        for it in 1:iterations
            println(m, it)
            this = init_boltzmann(m,3)
            CD_k(this, X_train, 1000, batch_size[ind])
            freq = get_frequencies(this, X_val)
            D_KL[ind] += KL_divergence(p_data, freq) / iterations
        end
    end

    print(D_KL)
    display(plot([1, 2, 4, 8], D_KL, label = "\$D_{KL}\$"))
    plot!(xlabel = "M")
    plot!(title = "Kullback-Leibler divergence vs. M")
    savefig("/home/jona/NN/homework2/DvM.png")
end

# Here we define a struct containg all the necessary values and matrices for a
# Boltzmann machine.
mutable struct boltzmann
    # Hidden neurons
    h::Any      # neurons
    M::Any      # amout of neurons
    _h::Any    # thresholds
    _h::Any   # threshold increments
    b_h::Any    # local field

    # Visible neurons
    v::Any
    N::Any
```

```
    _v::Any
    _v::Any
    b_v::Any

    # weight matrices
    w::Any      # weights
    w::Any     # weight increments
end

# This function initalizes a Boltzmann machine with M hidden neurons and N visible
function init_boltzmann(M, N)
    h = zeros(M)
    _h = zeros(M)
    _h = zeros(M)
    b_h = zeros(M)

    v = zeros(N)
    _v = zeros(N)
    _v = zeros(N)
    b_v = zeros(N)

    w = randn(M, N)
    w = zeros(M, N)

    model = boltzmann(h, M, _h, _h, b_h, v, N, _v, _v, b_v, w, w)
    return model
end

# This function trains a Boltzmann machine on given patterns using the CD_K
# algrithm, with k = 100. We noticed that we got better reults when changing the
# batch size depent on the number of hidden neurons.
function CD_k(this, patterns, _max, batch_size)
    for  in 1:_max
        sub_sample = sample(patterns, batch_size)

        for  in sub_sample
            this.v = copy()'
            update(this, "hidden")
            b_h = copy(this.b_h)

            for t in 1:100
                update(this, "visible")
                update(this, "hidden")
            end
            # compute weight and threshold increments
            update_weights(this, , b_h)
```

```
        end
        # update weights
        this.w += this.w
        this._v += this._v
        this._h += this._h

        # reset increments
        this.w -= this.w
        this._v -= this._v
        this._h -= this._h
    end
end

# This function samples our Boltzmann distribution when giving a random pattern
# of three Boolean values as input.
function get_frequencies(this, X)
    N_out = 1e3      # How many patterns we sample
    N_in = 1e3       # How many times we let the Boltzmann machine iterate
    counts = zeros(8)

    for it in 1:N_out
         = sample([1:8;],1)[1]
        this.v = X[]'
        update(this, "hidden")

        for jt in 1:N_in
            update(this, "visible")
            update(this, "hidden")

            # increment counts vector based on which pattern is currently
            # expressed in the visible neurons
            for (ind, x) in enumerate(X)
                if x' == this.v
                    counts[ind] += 1
                end
            end
        end
    end

    return counts/(N_out*N_in)   # Normalizing before returning
end

# This function updates one layer of neurons based on what argument is passed
# to it.
function update(this, layer)
    if layer == "hidden"
```

```
        this.b_h = this.w * this.v - this._h
        this.h = stochastic_update.(this.b_h)
    elseif layer == "visible"
        this.b_v = (this.h' * this.w)' - this._v
        this.v = stochastic_update.(this.b_v)
    end
end

# Input a local field and this function here will return you a stochastic update.
function stochastic_update(b)
    r = rand()
    return r < 1 / (1 + exp(-2*b)) ? 1 : -1
end

# This function updates the weight increments matrices
function update_weights(this, v0, b_h0)
     = 0.1
    this.w +=  * (tanh.(b_h0) * v0 - tanh.(this.b_h) * this.v')
    this._v -=  * (v0' - this.v)
    this._h -=  * (tanh.(b_h0) - tanh.(this.b_h))
end

# We use this function to store and load the train and validation data
function load_patterns()
    train = [[-1 -1 -1], [1 -1 1], [-1 1 1], [1 1 -1]]
    val = [[-1 -1 -1], [1 -1 1], [-1 1 1], [1 1 -1], [-1 -1 1], [1 -1 -1], [-1 1 -1], [1 1 1]]
    return train, val
end

# This function takes as input a probability for the data and a probability for
# the Boltzmann distribution and returns the Kullback-Leibler divergence score
function KL_divergence(p_data, p_B)
    D_KL = 0

    for  in 1:4
        D_KL += p_data[] * log(p_data[] / p_B[])
    end
    return D_KL
end


main()
```

3 One-layer Perceptron

3.1 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()
```

3.2 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
```