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.

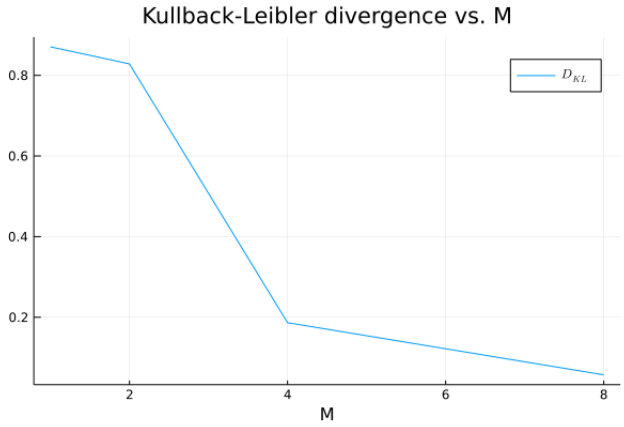

Figure 1: Here we can see the KL divergence for different values for M.

2 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.

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()
```