Homework 1 FIM720

Jonatan Hellgren

September 14, 2021

1 One-step error probability

```
#= This script answers the first question "One-step error probability" of Homework 1 =#

#= All the scrips are written in julia, since I didn’t want to be bothered with the  =#
#=   slow programing languages such as python or matlab. I will do my best to make the =#
#=   scripts understandable for someone with no experience with Julia. The performance =#
#=   in this scrips are outstanding compared to the equivalent scripts that are written =#
#=   with python or matlab, it only takes 5 seconds to execute this script on my laptop =#

using Distributions # used for random sampling

#= The main script computes the error probability for different values for P =#
function main()
  P = [12,24,48,70,100,120]
  trials = 1e5
  for pt in P
    P_error(pt, trials)
  end
end

#= This function computes the error probability for a given value for P patterns =#
#=   and a given value for trials =#
function P_error(P, trials)
  N = 120

  #= generate the patterns X =#
  X = generate_patterns(N, P)

  #= Compute the weights W =#
  zero_dig = false
  W = hebbs_rule(X, zero_dig)

  total_errors = 0
  for t in 1:trials
    # sample a random pixle from a random pattern
    i = sample(1:N)
     = sample(1:P)

    # perform a asynchronous update
    S_i = async_update(W, X[], i)

    # check wheter the pixle changed or not
    if S_i != X[][i]
      total_errors += 1
    end
  end

  # compute P_error and print it
  total_errors /= trials
```

```
  print(total_errors, ", ")
end


#= Generates a pattern of length N, where each element is = 1 with p=1/2 and -1 otherwise =#
function generate_pattern(N)
  sample([-1, 1], N)
end

#= Generates a list of P patterns and returns it =#
function generate_patterns(N, P)
  #= Initalize a empty list and append(push!) each pattern to it =#
  X = []
  for it in 1:P
    push!(X, generate_pattern(N))
  end

  return X
end

#= A function that computes the weights for a hopfield network using Hebb's rule =#
#= Inputs the patterns and a boolean that says wheter the diagonal should be zero or not =#
function hebbs_rule(X, zero_dig)
  P = length(X)
  N = length(X[1])
  W = zeros(Int, N, N)
  for it in 1:N
    for jt in 1:N
      for pt in 1:P
        W[it,jt] += X[pt][it] * X[pt][jt]
      end
    end
  end
  W /= N

  # if we want the diagonal to equal zero we need this extra loop
  if zero_dig
    for it in 1:N
      W[it,it] = 0
    end
  end

  return W
end

#= This function performs a asynchronous update on the pattern S at index i  =#
#=   using the weights W. Then returns the value, if the local field is equal =#
#=   to 0, then it returns 1 =#
function async_update(W, S, i)
  N = length(S)

  #= Compute the localfield =#
  local_field = 0
  for j in 1:N
    local_field += W[i, j] * S[j]
  end

  #= Return the sign of the local field =#
  s_i = sign(local_field)
  if s_i == 0
    return 1
```

```
  else
    return s_i
  end
end


main()
```

2 Recognising digits

```
#= This script answers the first question "One-step error probability" of Homework 1 =#

#= All the scrips are written in julia, since I didn't want to be bothered with the  =#
#=   slow programing languages such as python or matlab. I will do my best to make the =#
#=   scripts understandable for someone with no experience with Julia. The performance =#
#=   in this scrips are outstanding compared to the equivalent scripts that are written =#
#=   with python or matlab, it only takes 5 seconds to execute this script on my laptop =#

using Distributions # used for random sampling

#= The main script computes the error probability for different values for P =#
function main()
  P = [12,24,48,70,100,120]
  trials = 1e5
  for pt in P
    P_error(pt, trials)
  end
end

#= This function computes the error probability for a given value for P patterns =#
#=   and a given value for trials =#
function P_error(P, trials)
  N = 120

  #= generate the patterns X =#
  X = generate_patterns(N, P)

  #= Compute the weights W =#
  zero_dig = false
  W = hebbs_rule(X, zero_dig)

  total_errors = 0
  for t in 1:trials
    # sample a random pixle from a random pattern
    i = sample(1:N)
     = sample(1:P)

    # perform a asynchronous update
    S_i = async_update(W, X[], i)

    # check wheter the pixle changed or not
    if S_i != X[][i]
      total_errors += 1
    end
  end

  # compute P_error and print it
  total_errors /= trials
  print(total_errors, ", ")
```

```
end


#= Generates a pattern of length N, where each element is = 1 with p=1/2 and -1 otherwise =#
function generate_pattern(N)
  sample([-1, 1], N)
end

#= Generates a list of P patterns and returns it =#
function generate_patterns(N, P)
  #= Initalize a empty list and append(push!) each pattern to it =#
  X = []
  for it in 1:P
    push!(X, generate_pattern(N))
  end

  return X
end

#= A function that computes the weights for a hopfield network using Hebb's rule =#
#= Inputs the patterns and a boolean that says wheter the diagonal should be zero or not =#
function hebbs_rule(X, zero_dig)
  P = length(X)
  N = length(X[1])
  W = zeros(Int, N, N)
  for it in 1:N
    for jt in 1:N
      for pt in 1:P
        W[it,jt] += X[pt][it] * X[pt][jt]
      end
    end
  end
  W /= N

  # if we want the diagonal to equal zero we need this extra loop
  if zero_dig
    for it in 1:N
      W[it,it] = 0
    end
  end

  return W
end

#= This function performs a asynchronous update on the pattern S at index i  =#
#=   using the weights W. Then returns the value, if the local field is equal =#
#=   to 0, then it returns 1 =#
function async_update(W, S, i)
  N = length(S)

  #= Compute the localfield =#
  local_field = 0
  for j in 1:N
    local_field += W[i, j] * S[j]
  end

  #= Return the sign of the local field =#
  s_i = sign(local_field)
  if s_i == 0
    return 1
  else
```

```
    return s_i
  end
end


main()

(base) jona@marmor:~/NN$ cat recognizing_digits.jl
#= This script answers the second question "Recognising digits" in Homework 1 =#

#= In the main script we load the data, fit it to our model and later use the model =#
#=   to classify a pattern =#
function main()
  X, P = load_data()

  patterns = length(X)
  N = length(X[1])

  zero_dig = true
  W = hebbs_rule(X, zero_dig)

  #= Here we choose which pattern we want to try our model on =#
  S = copy(P[1])
  #= Then we print it, just so that we can get a visual understanding of how the =#
  #=   model operates =#
  print_pattern(S)

  #= This while-loop performs asynchronous updates until convergence =#
  while true
    S_t = update_pattern(W, S)
    #= Again here we print the current pattern just for fun =#
    print_pattern(S_t)
    if S == S_t
      #= Output the value when the model converges and outputs it to stdout =#
      println(reshape(S_t, (10,16))')
      break
    end
    S = S_t
  end
end

#= This function takes as input the weight matrix and a pattern, then it returns a  =#
#=   updated version of the pattern =#
function update_pattern(W, S)
  N = length(S)
  S_t = copy(S)
  for i in 1:N
    S_t[i] = async_update(W, S_t, i)
  end
  return S_t
end

#= This function performs a asynchronous update on the pattern S at index i  =#
#=   using the weights W. Then returns the value, if the local field is equal =#
#=   to 0, then it returns 1 =#
function async_update(W, S, i)
  N = length(S)

  #= Compute the localfield =#
  local_field = 0
  for j in 1:N
```

```
    local_field += W[i, j] * S[j]
  end

  #= Return the sign of the local field =#
  s_i = sign(local_field)
  if s_i == 0
    return 1
  else
    return s_i
  end
end


#= A function that computes the weights for a hopfield network using Hebb's rule =#
#= Inputs the patterns and a boolean that says wheter the diagonal should be zero or not =#
function hebbs_rule(X, zero_dig)
  P = length(X)
  N = length(X[1])
  W = zeros(Int, N, N)
  for it in 1:N
    for jt in 1:N
      for pt in 1:P
        W[it,jt] += X[pt][it] * X[pt][jt]
      end
    end
  end
  W /= N

  # if we want the diagonal to equal zero we need this extra loop
  if zero_dig
    for it in 1:N
      W[it,it] = 0
    end
  end

  return W
end

#= To make the script a bit more fun and also esier to troubleshoot this function where =#
#=   created. It is just a very simple visualization of the patterns =#
function print_pattern(X, m=16, n=10)
  for it in 1:length(X)
    if X[it] == 1
      print('X')
    else
      print(' ')
    end
    if it % 10 == 0
      print('\n')
    end
  end
  print('\n')
end

#= Here are the data that I got from OpenTA =#
function load_data()
  # 0
  x1=[ [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, 1, 1, 1, 1, 1,
  # 1
  x2=[ [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1
  # 2
```

```
  x3=[ [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],[ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],[ -1, -1, -1, -1, -1, 1, 1, 1, -1, -
  # 3
  x4=[ [ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1
  # 4
  x5=[ [ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 

  # combinined
  xs=[x1, x2, x3, x4, x5]
  X = transform_data(xs)


  # distorted 3
  p1 = [[1, 1, -1, -1, -1, -1, -1, -1, 1, 1], [1, 1, -1, -1, -1, -1, -1, -1, -1, 1], [-1, -1, -1, -1, -1, -1, 1
  # distorted 4
  p2 = [[1, -1, -1, 1, 1, 1, 1, -1, -1, 1], [-1, 1, 1, -1, -1, -1, -1, 1, 1, -1], [-1, 1, 1, -1, -1, -1, -1, 1,
  # distorted 1
  p3 = [[-1, -1, -1, 1, 1, 1, 1, -1, -1, -1], [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1], [-1, -1, -1, 1, 1, 1, 1, -1

  # combinind
  ps = [p1, p2, p3]
  P = transform_data(ps)

  return X, P
end

#= I ran in to a problem with the way the patterns where stored, so this helper =#
#=   function was necesary to make use of the functions created previously.  =#
#=   It basically turns a matrix in to a long vector. =#
function transform_data(ts, m=16)
  T = []
  for t in ts
    tmp = []
    for it in 1:m
      tmp = vcat(tmp, t[it])
    end
    push!(T, tmp)
  end
  return T
end

main()
```

3 Stochastic Hopfield network

```
#= This script computes the order parameter using a asynchronous stochastic update, =#
#=   it answers the third question "Stochastic Hopfield network" in homework 1 =#

using Distributions

function main()
  #= initalizing paramers =#
   = 2
  N = 200
  P = 7
  T = 2e5
  order_parameter = 0
  iterations = 100

  #= Compute the order parameter 100 times and compute the average =#
  for it in 1:iterations
```

```
    println(it)
    order_parameter += compute_order_parameter(, N, P, T)
  end
  order_parameter /= iterations

  #= print the results  =#
  println(order_parameter)
end

#= A function that computes the weights for a hopfield network using Hebb's rule =#
#= Inputs the patterns and a boolean that says wheter the diagonal should be zero or not =#
function hebbs_rule(X, zero_dig)
  P = length(X)
  N = length(X[1])
  W = zeros(Int, N, N)
  for it in 1:N
    for jt in 1:N
      for pt in 1:P
        W[it,jt] += X[pt][it] * X[pt][jt]
      end
    end
  end
  W /= N

  # if we want the diagonal to equal zero we need this extra loop
  if zero_dig
    for it in 1:N
      W[it,it] = 0
    end
  end

  return W
end

#= Generates a pattern of length N, where each element is = 1 with p=1/2 and -1 otherwise =#
function generate_pattern(N)
  sample([-1, 1], N)
end

#= Generates a list of P patterns and returns it =#
function generate_patterns(N, P)
  #= Initalize a empty list and append(push!) each pattern to it =#
  X = []
  for it in 1:P
    push!(X, generate_pattern(N))
  end

  return X
end

#= This function performs a stochastig asynchronous update and returns the value  =#
#=   for that pixle =#
#= The inputs are the weight matrix, the patterns we want to update, and the index =#
#=   in that pattern and  the noise parameter =#
function stoch_async_update(W, S, i, )
  N = length(S)
  local_field = 0
  for j in 1:N
    local_field += W[i, j] * S[j]
  end
  if rand() < p(local_field, )
```

```
    return 1
  else
    return -1
  end
end

#= This function computes the probability for the new state being equal to 1, given =#
#=   the local field and noise parameter =#
function p(b, )
  return 1 / (1 + exp(-2**b))
end

#= This function computes the equality measure for two patterns, it equals 1 if the are =#
#=   the same and -1 if inverse =#
function equality_measure(S, p_t)
  count = 0
  for it in 1:length(S)
    count += S[it] * p_t[it]
  end
  return count / length(S)
end

function compute_order_parameter(, N, P, T)
  # Matrix for the random patterns
  X = generate_patterns(N, P)
  S = copy(X[1])

  # Get weights with zero diagonal
  zero_dig = true
  W = hebbs_rule(X, zero_dig)

  #= Perform a asynchronous update and compute the equality measure, =#
  #=   repeat for T times and lastly computing the average, which is to be =#
  #=   returned =#
  m = 0
  for it in 1:T
    ind::Int = it % 200 + 1
    S[ind] = stoch_async_update(W, S, ind, )
    m += equality_measure(S, X[1])
  end
  m /= T

  return m
end

main()
```