Algorithm

A Matlab program was implemented to train a perceptron to be able to recognise if a random given boolean function in given n dimensions is linearly separable or not. The program also counts how many boolean functions are linearly separable for given dimensions. The network learns iteratively over 20 epochs for each random boolean function. Each boolean function is only used only once for training. The program runs for 10^4 trials. The algorithm used to train the network was an iterative gradient descent (see chapter 5.2 in the book *Machine Learning for Neural Networks* by Bernard Mehlig.

Results

Table 1: Matlab results of linearly separable boolean functions in relation to n dimensions

n dimensions	Number of linearly separable boolean functions from Matlab results	Actual number of linear separable boolean functions	Number of boolean functions in n dimensions
2	14	14	16
3	104	104	256
4	262	1882	65536
5	0	94572	4294967296

Discussion and conclussion

The perceptron computes the right number of linearly separable functions in 2 and 3 dimensions. Though, not for 4 and 5. The number of boolean functions in 4 and 5 dimensions are 65536 respectively 4294967296. It's impossible for the perceptron to be trained for 65536 functions and beyond when the network is only capable of being trained for 10000 boolean functions (10^4 trials). The number of linearly separable boolean functions in 5 dimensions are 94572, the probability of finding *one* linearly separable function in 5 dimensions in one trial is therefore 0.00002 and 0.2 for 10^4 trials. That means that in one batch run, the perceptron will probably not find *any* linearly separable boolean functions in 5 dimensions. Similarly goes for 4 dimensions, the number of linearly separable boolean functions in 4 dimensions are 1882. The probability of finding *one* linearly separable function in 4 dimensions in one trial is therefore 0.0287 and 287 in 10^4 trials; the program should find about 287 linearly separable boolean functions for 4 dimensions which is roughly what the results show (see table 1). For 2 and 3 dimensions, the number of boolean functions are 16 respectively 256 which is well below 10^4 trials. This means that the perceptron will be trained for all boolean functions and likely find all linearly separable boolean functions for these dimensions, which the results shows as well. To conclude, in order to achieve accurate results from a trained network, the selected parameters involved must be evaluated and chosen carefully so that one doesn't blindly believe results from a network just because it's ”trained”.

Boolean functions 2022

Erik Norlin

All equations are taken from the course book Machine Learning With Neural Networks.

$$O = \mathrm{sgn}(w_1 x_1 + w_2 x_2 - \theta) = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{x} - \theta). \qquad (5.9)$$

$$\delta w_{ij}^{(\mu)} = \eta(t_i^{(\mu)} - O_i^{(\mu)})x_j^{(\mu)}. \qquad (5.18)$$

```
nDimensions = 5;
counter = zeros(nDimensions,1);

for iDimension = 2:nDimensions
    nTrials = 10^4;
    eta = 0.05;
    nEpochs = 20;
    booleanInputs = zeros(2^iDimension,iDimension);
    booleanOutputs = zeros(2^iDimension,1);
    usedBoolean = [];

    % Generating boolean inputs based on given dimensions
    booleanInputs(1,:) = -1;
    for i = 1:2^iDimension - 1
        binary = dec2bin(i,iDimension);
        for j = 1:iDimension
            booleanInputs(i+1,j) = str2num(binary(j));
            if booleanInputs(i+1,j) == 0
                booleanInputs(i+1,j) = -1;
            end
        end
    end

    for iTrial = 1:nTrials

        % Sampling a random boolean function based on given dimensions
        for j = 1:2^iDimension
            outputState = rand;
            if outputState < 0.5
                outputState = -1;
            else
                outputState = 1;
            end
            booleanOutputs(j) = outputState;
        end

        % Checking if sampled boolean function already has been used
        validBoolean = true;
        if width(usedBoolean) > 0
            for jCol = 1:width(usedBoolean)
                if isequal(booleanOutputs, usedBoolean(:,jCol))
                    validBoolean = false;
```

```
                    break
                end
            end
        end

        if validBoolean
            weight = randn(1,iDimension)/sqrt(iDimension);
            theta = 0;

            % Training the network for given number of epochs
            for jEpoch = 1:nEpochs
                errorCounter = 0;

                % Updating weight and threshold for each boolean output
                for muPattern = 1:2^iDimension

                    input = booleanInputs(muPattern,:)';
                    output = sign(weight*input - theta);
                    if output == 0
                        output = 1;
                    end

                    target = booleanOutputs(muPattern);
                    xMu = booleanInputs(muPattern,:);

                    deltaWeight = eta*(target - output)*xMu;
                    deltaTheta = -eta*(target - output);

                    weight = weight + deltaWeight;
                    theta = theta + deltaTheta;

                    error = target - output;
                    errorCounter = errorCounter + abs(error);
                end
                if errorCounter == 0
                    counter(iDimension,1) = counter(iDimension,1) + 1;
                    break
                end
            end
            % Adding sampled boolean function to used boolean functions
            iCol = width(usedBoolean) + 1;
            usedBoolean(:,iCol) = booleanOutputs;
        end
    end
end

% Printing results
fprintf("Based on the training of the network. ..." + ...
    "The number of linearly separable boolean functions in 'n' dimensions:");
```

```
Based on the training of the network. ...The number of linearly separable boolean functions in 'n' dimensions:
```

```
for iDimension = 2:nDimensions
    fprintf("%d dimensions: %d\n", iDimension, counter(iDimension));
```

```
end
```

```
2 dimensions: 14
3 dimensions: 104
4 dimensions: 262
5 dimensions: 0
```

One-step error probability (unweighted diagonal)

Erik Norlin

All equations are taken from the course book *Machine Learning With Neural Networks.*

```
pPatterns = [12,24,48,70,100,120];
nNeurons = 120;
nTrials = 10^5;
mRowWeightMatrix = zeros(1,nNeurons); % Row "m" from weight matrix "W" based on random
                                      % neuron "m" and all patterns for each "p"
tempRowWeightMatrix = zeros(1,nNeurons); % Row "m" from a temporary weight
                                         % matrix for each pattern
nSetOfPatterns = length(pPatterns);
oneStepErrorProb = zeros(1,nSetOfPatterns);

for iSetOfPatterns = 1:nSetOfPatterns
    nPatterns = pPatterns(iSetOfPatterns);
    patternsMatrix = zeros(nPatterns,nNeurons); % Matrix to store "p" number of patterns

    errorCounter = 0;
    for jTrial = 1:nTrials
        % Creating "p" number of random patterns
        for muPattern = 1:nPatterns
            for jNeuron = 1:nNeurons
                rand = randi(2); % Generating a random number between 1 and 2 as an
                                 % activation state of given neuron
                if rand == 2
                    rand = -1;
                end
                patternsMatrix(muPattern,jNeuron) = rand;
            end
        end
```

Hebb's rule to compute the weight matrix:

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} x_i^{(\mu)} x_j^{(\mu)} \quad \text{for} \quad i \neq j, \quad w_{ii} = 0, \quad \text{and} \quad \theta_i = 0. \qquad (2.26)$$

```
        % Creating only the necessary row "m" from the weight matrix "W"
        % based on all generated patterns and a random chosen neuron "m"
        mRandomNeuron = randi(nNeurons);
        for muPattern = 1:nPatterns
            for jNeuron = 1:nNeurons
                tempRowWeightMatrix(jNeuron) = patternsMatrix(muPattern,mRandomNeuron)...
                * patternsMatrix(muPattern,jNeuron);
                if jNeuron == mRandomNeuron % wii = 0
                    tempRowWeightMatrix(jNeuron) = 0;
                end
            end
            mRowWeightMatrix = mRowWeightMatrix + tempRowWeightMatrix;
        end
```

```
        mRowWeightMatrix = (1/nNeurons) * mRowWeightMatrix;
```

To compute activation state of chosen neuron:

$$s_i(t+1) = \begin{cases} g\left(\sum_j w_{mj} s_j(t) - \theta_m\right) & \text{for } i = m\,, \\ s_i(t) & \text{otherwise}\,. \end{cases} \qquad (1.9)$$

```
        % Choosing one random pattern to feed pattern to neuron "m" once, as one
        % asynchronous update (equation 1.9)
        randomPattern = randi(nPatterns);
        sInputPattern = patternsMatrix(randomPattern,:);
        sOutput = mRowWeightMatrix * sInputPattern';

        if sOutput < 0
            sOutput = -1;
        else % sOutput >= 0
            sOutput = 1;
        end

        % If the neuron is updated
        if sOutput ~= sInputPattern(mRandomNeuron)
            errorCounter = errorCounter + 1;
        end
    end
    oneStepErrorProb(iSetOfPatterns) = errorCounter / nTrials;
end

% Printing final result
disp("One-step error probability for each value of p:")
```

```
One-step error probability for each value of p:
```

```
for i = 1:nSetOfPatterns
    fprintf("p%d: %.4f\n", i, oneStepErrorProb(i));
end
```

```
p1: 0.0004
p2: 0.0110
p3: 0.0551
p4: 0.0934
p5: 0.1364
p6: 0.1581
```

One-step error probability (weighted diagonal)

Erik Norlin

All equations are taken from the course book *Machine Learning With Neural Networks.*

```
pPatterns = [12,24,48,70,100,120];
nNeurons = 120;
nTrials = 10^5;
mRowWeightMatrix = zeros(1,nNeurons); % Row "m" from weight matrix "W" based on random
                                      % neuron "m" and all patterns for each "p"
tempRowWeightMatrix = zeros(1,nNeurons); % Row "m" from a temporary weight
                                         % matrix for each pattern
nSetOfPatterns = length(pPatterns);
oneStepErrorProb = zeros(1,nSetOfPatterns);

for iSetOfPatterns = 1:nSetOfPatterns
    nPatterns = pPatterns(iSetOfPatterns);
    patternsMatrix = zeros(nPatterns,nNeurons); % Matrix to store "p" number of patterns

    errorCounter = 0;
    for jTrial = 1:nTrials
        % Creating "p" number of random patterns
        for muPattern = 1:nPatterns
            for jNeuron = 1:nNeurons
                rand = randi(2); % Generating a random number between 1 and 2 as an
                                 % activation state of given neuron
                if rand == 2
                    rand = -1;
                end
                patternsMatrix(muPattern,jNeuron) = rand;
            end
        end
```

Hebb's rule to compute the weight matrix:

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} x_i^{(\mu)} x_j^{(\mu)} \quad \text{for} \quad i \neq j, \quad w_{ii} = 0, \quad \text{and} \quad \theta_i = 0. \qquad (2.26)$$

```
        % Creating only the necessary row "m" from the weight matrix "W"
        % based on all generated patterns and a random chosen neuron "m"
        mRandomNeuron = randi(nNeurons);
        for muPattern = 1:nPatterns
            for jNeuron = 1:nNeurons
                tempRowWeightMatrix(jNeuron) = patternsMatrix(muPattern,mRandomNeuron)...
                * patternsMatrix(muPattern,jNeuron);
%                 if jNeuron == mRandomNeuron % wii = 0
%                     tempRowWeightMatrix(jNeuron) = 0;
%                 end
            end
            mRowWeightMatrix = mRowWeightMatrix + tempRowWeightMatrix;
        end
```

```
        mRowWeightMatrix = (1/nNeurons) * mRowWeightMatrix;
```

To compute activation state of chosen neuron:

$$s_i(t+1) = \begin{cases} g\left(\sum_j w_{mj} s_j(t) - \theta_m\right) & \text{for } i = m\,, \\ s_i(t) & \text{otherwise}. \end{cases} \qquad (1.9)$$

```
        % Choosing one random pattern to feed pattern to neuron "m" once, as one
        % asynchronous update (equation 1.9)
        randomPattern = randi(nPatterns);
        sInputPattern = patternsMatrix(randomPattern,:);
        sOutput = mRowWeightMatrix * sInputPattern';

        if sOutput < 0
            sOutput = -1;
        else % sOutput >= 0
            sOutput = 1;
        end

        % If the neuron is updated
        if sOutput ~= sInputPattern(mRandomNeuron)
            errorCounter = errorCounter + 1;
        end
    end
    oneStepErrorProb(iSetOfPatterns) = errorCounter / nTrials;
end

% Printing final result
disp("One-step error probability for each value of p:")
```

```
One-step error probability for each value of p:
```

```
for i = 1:nSetOfPatterns
    fprintf("p%d: %.4f\n", i, oneStepErrorProb(i));
end
```

```
p1: 0.0002
p2: 0.0032
p3: 0.0128
p4: 0.0182
p5: 0.0223
p6: 0.0231
```

Recognising digits

Erik Norlin

All equations are taken from the course book *Machine Learning With Neural Networks.*

```
x1=[ [ -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,
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,
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, -
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

sInputPattern1 = [[1, -1, -1, 1, 1, 1, 1, -1, -1, 1], [-1, 1, 1, -1, -1, -1, -1, 1, 1, -1], [-1
sInputPattern2 = [[1, -1, -1, 1, -1, 1, -1, 1, 1, -1], [1, -1, -1, 1, -1, 1, -1, 1, -1, -1], [1
sInputPattern3 = [[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1

storedPatterns = [x1;x2;x3;x4;x5];
sInputPatterns = [sInputPattern1;sInputPattern2;sInputPattern3];

nNeurons = length(storedPatterns);
tempWeightMatrix = zeros(nNeurons); % Temporary weight matrix for each pattern
weightMatrix = zeros(nNeurons); % Weight matrix based on all patterns
nPatterns = height(storedPatterns);
```

Hebb's rule to compute the weight matrix:

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} x_i^{(\mu)} x_j^{(\mu)} \quad \text{for} \quad i \neq j, \quad w_{ii} = 0, \quad \text{and} \quad \theta_i = 0. \qquad (2.26)$$

```
% Creating weighted matrix "W"
for muPattern = 1:nPatterns
    for iNeuron = 1:nNeurons
        for jNeuron = 1:nNeurons
            tempWeightMatrix(iNeuron,jNeuron) = storedPatterns(muPattern,iNeuron)...
            * storedPatterns(muPattern,jNeuron);
            if iNeuron == jNeuron % wii = 0
                tempWeightMatrix(iNeuron,jNeuron) = 0;
            end
        end
    end
    weightMatrix = weightMatrix + tempWeightMatrix;
end
weightMatrix = (1/nNeurons) * weightMatrix;

% Looping through the three questions
for iQuestion = 1:height(sInputPatterns)
    counter = 0;
    valid = false;

    % Looping until a pattern is recognised or number of iteration exceeds 1000
    while ~valid
```

```
        % Inverting a copy of the pattern we're feeding
        invertedInputPattern = sInputPatterns(iQuestion,:);
        for iRow = 1:length(invertedInputPattern)
            if invertedInputPattern(iRow) == 1
                invertedInputPattern(iRow) = -1;
            else % invertedInputPattern(i) == -1;
                invertedInputPattern(iRow) = 1;
            end
        end

        % Checking if the network is stable i.e. if the feeding pattern
        % is equal to any of the stored patterns
        for muPattern = 1:nPatterns
            inputPattern = sInputPatterns(iQuestion,:);
            identifyPattern = storedPatterns(muPattern,:);

            if isequal(inputPattern, identifyPattern) || ...
                isequal(invertedInputPattern, identifyPattern)

                if isequal(inputPattern, identifyPattern)
                    pattern = muPattern;
                elseif isequal(invertedInputPattern, identifyPattern)
                    pattern = -muPattern;
                end

                % Converting the steady pattern to OpenTA format, starting
                % with creating a matrix for the steady state pattern
                steadyStatePatternArr = zeros(16,10);
                nRows = height(steadyStatePatternArr);
                nCols = width(steadyStatePatternArr);
                iCounter = 1;

                for iRow = 1:nRows
                    for jCol = 1:nCols
                        steadyStatePatternArr(iRow,jCol) = inputPattern(iCounter);
                        iCounter = iCounter + 1;
                    end
                end

                % Converting the matrix to a string
                steadyStatePatternString = "";
                commaCounter = 1;

                for iRow = 1:nRows
                    rowString = join(string(steadyStatePatternArr(iRow,:)),", ");
                    steadyStatePatternString = steadyStatePatternString + "["...
                    + rowString + "]";
                    if commaCounter < nRows
                        steadyStatePatternString = steadyStatePatternString + ", ";
                    end
                    commaCounter = commaCounter + 1;
                end
                steadyStatePatternString = "[" + steadyStatePatternString + "]";
```

```
                % Printing final result
                fprintf("In Q%d, the network recognized pattern %d as digit %d\n", ...
                    iQuestion, pattern, muPattern-1);
                fprintf("Steady pattern for Q%d:", iQuestion);
                fprintf("%s", steadyStatePatternString);
                valid = true;
                break
            end
        end
```

Updating the network:

$$s_i(t+1) = \begin{cases} g\left(\sum_j w_{mj} s_j(t) - \theta_m\right) & \text{for } i = m, \\ s_i(t) & \text{otherwise.} \end{cases} \qquad (1.9)$$

If the ouput of the activation function is < 0, the state is -1 else; the state is +1.

$$\operatorname{sgn}(b) = \begin{cases} -1, & b < 0, \\ +1, & b \geq 0. \end{cases} \qquad (1.3)$$

$$b_i(t) = \sum_{j=1}^{N} w_{ij} s_j(t) - \theta_i, \qquad (1.4)$$

$\theta = 0$ *(as stated in 2.26)*

```
        % Updating the network using typewriter scheme if the pattern wasn't recognised
        if valid == false
            for iNeuron = 1:nNeurons
                sOutput = weightMatrix(iNeuron,:) * sInputPatterns(iQuestion,:)';
                if sOutput < 0
                    sOutput = -1;
                else % sOutput >= 0
                    sOutput = 1;
                end

                if sOutput ~= sInputPatterns(iQuestion, iNeuron)
                    sInputPatterns(iQuestion, iNeuron) = sOutput;
                end
            end
        end

        % Breaking the while loop if the no patterns are recognized
        counter = counter + 1;
        if counter > 1000
            fprintf("No patterns were recognised in Q%d within 1000 iterations.", iQuestion)
            break
        end
    end
end
```

```
In Q1, the network recognized pattern 5 as digit 4
```

```
Steady pattern for Q1:
[[-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, -1], [
In Q2, the network recognized pattern 5 as digit 4
Steady pattern for Q2:
[[-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, -1], [
In Q3, the network recognized pattern 1 as digit 0
Steady pattern for Q3:
[[-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, -1],
```