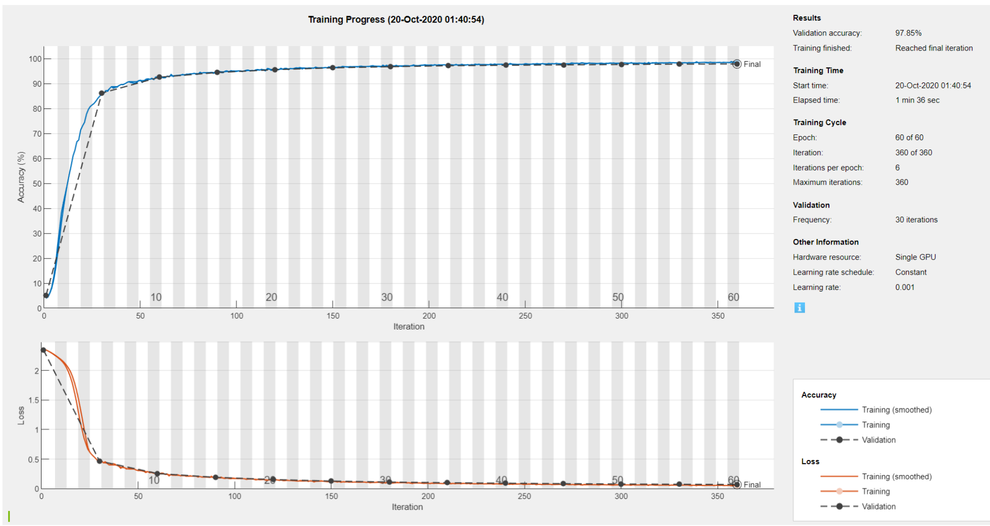
FFR135 Artificial Neural Networks
Homework 3

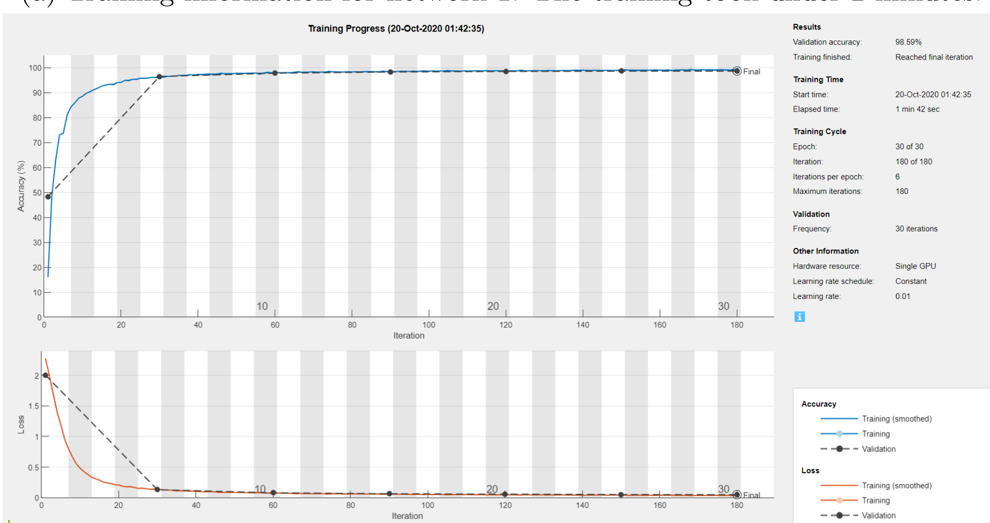
David Tonderski, davton

Convolutional networks

David Tonderski (davton)



(a) Training information for network 1. The training took under 2 minutes.



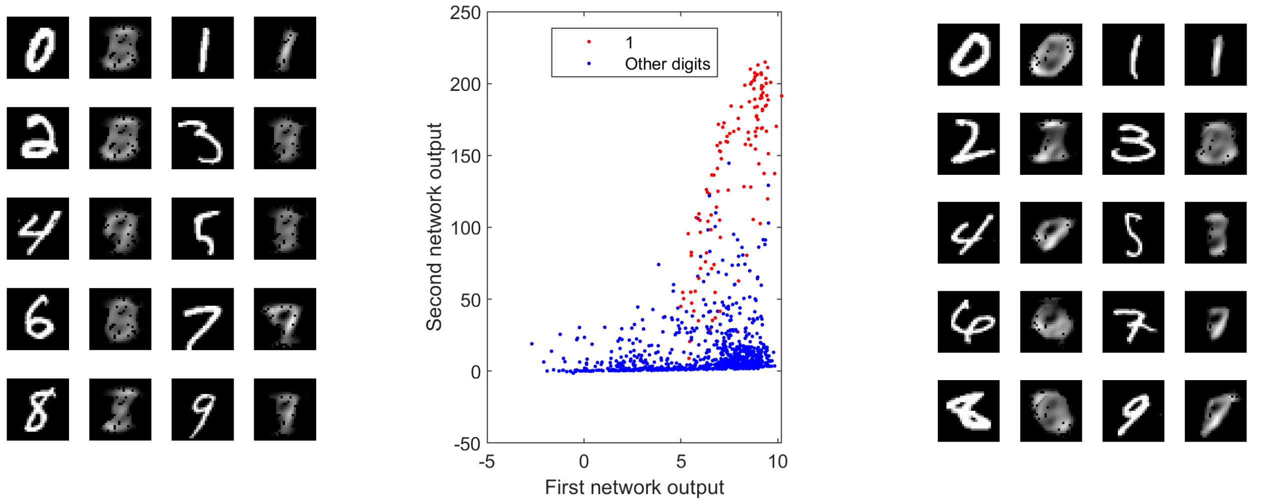
(b) Training information for network 2. The training took under 2 minutes.

Figure 1: Training information for the two networks.

After training, network 1 has a training accuracy of about 98.43%, a validation accuracy of 97.85%, and a test accuracy of 98.18%, while network 2 has a training accuracy of about 99.01%, a validation accuracy of 98.6%, and a test accuracy of 98.88%. As expected, the deeper second network performs slightly better. Interestingly, the training times using the GPU were quite similar, which I did not expect from the description (2 hours vs 6 hours).

Fully connected autoencoder

David Tonderski (davton)



(a) Montage for the first network.

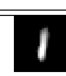
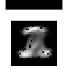
(b) Scatter plot of the encoded values.

(c) Montage for the second network.

Figure 1: Montage for the two networks and scatter plot for the first.

As shown in figure 1a, the first encoder only convincingly reproduces the digit 1. This is also visible in figure 1b. The second encoder seems to reproduce the digits 1, 2, and 7, as shown in figure 1c. Examining scatter plots of the values encoded in the second network shows the approximate regions for each of the reproduced digits: ones are in the region $x_1 \in [15, 40], x_2 \in [-5, 10], x_3 \in [0, 100], x_4 \in [20, 350]$, twos are in $x_1 \in [3, 15], x_2 \in [3, 15], x_3 \in [0, 4], x_4 \in [0, 10]$, while sevens are in $x_1 \in [20, 30], x_2 \in [10, 40], x_3 \in [80, 140], x_4 \in [20, 50]$. This is visualised in table 1. Fives and threes often resemble each other, while the other digits are usually visually similar to zeros. This was also seen in the scatter plots, as all the other digits where scrambled together.

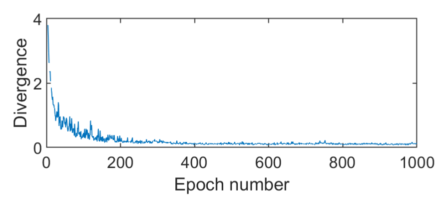
Table 1: Shows generated images for chosen x_1, x_2, x_3, x_4.

Digit	x_1	x_2	x_3	x_4	Generated digit
1	25	3	50	175	
2	9	9	2	5	
7	25	25	110	35	

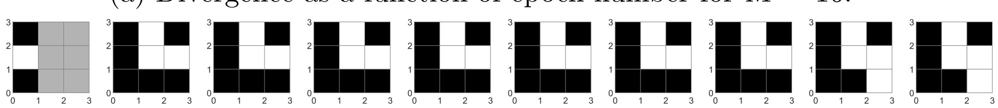
Restricted Boltzmann machine

David Tonderski (davton)

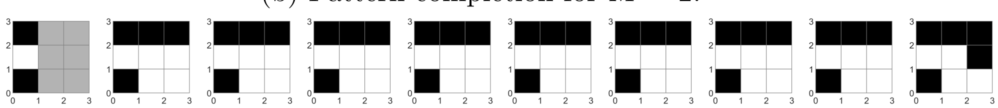
The divergence was calculated as follows: in each epoch, each *possible* pattern was fed to the Boltzmann machine. The dynamics were then run for 100 iterations, and the frequencies at which the different *possible* patterns occured were counted. This was then used as an approximation for the model distribution P_B for each epoch, and the Kullback-Leibler divergence was calculated using $d = \sum_\mu P_D(\mu) \log(P_D(\mu)/P_B(i_\mu))$, where $P_D(\mu) = \frac{1}{14}$ for all data set patterns μ, and i_μ is the index of the data set pattern μ in the set of all possible patterns. If $P_B(\mu) = 0$, I set $d = \infty$. For $M = 2, 4, 8$, the divergence was infinity for (almost) all epochs, so the plots are not shown. For $M = 16$, the divergence is shown in figure 1a. The first 10 produced patterns after feeding the first column are shown in figures 1b, 1c, 1d, 1e.



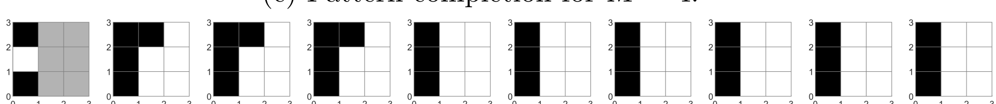
(a) Divergence as a function of epoch number for $M = 16$.



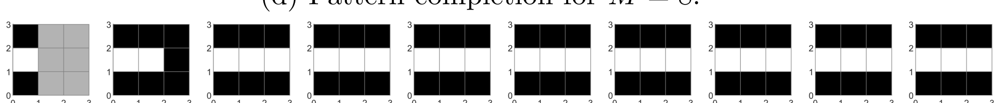
(b) Pattern completion for $M = 2$.



(c) Pattern completion for $M = 4$.



(d) Pattern completion for $M = 8$.



(e) Pattern completion for $M = 16$.

Figure 1: Training information for the two networks.

As expected, the model works well for $M = 16$, and not at all for $M = 2, 4$. Interestingly, the model converges to a data set pattern for $M = 8$, despite the divergence being ∞, but it is not the correct one. This probably means that the model has learned a few of the data set patterns, but not all of them.

Tic tac toe

David Tonderski (davton)

To shorten training, a hash function was created as follows:

$$H(S) = 1 + \sum_{i=1}^{i=9} (S(10-i)+1) \cdot 3^{i-1}, \tag{1}$$

where $S(j)$ is the state of the j-th field of board S, where an "X" is represented by a "1", an "O" by a "-1", and an empty field by a "0". We have $H([-1,-1,-1,-1,-1,-1,-1,,-1,-1]) = 1, H([-1,-1,-1,-1,-1,-1,-1,-1,0]) = 2, ..., H([1,1,1,1,1,1,1,1,1]) = 19683$. Then, the Q-table was initialized as a 2 by 19683 matrix (there are 19683 possible tic tac toe board states), where each field was a struct with the field 'entry' containing an empty matrix. Whenever a state was encountered, the program checked whether the H-th column of the Q-table consisted of structs containing empty matrices. If it did, the program initialized those matrices, where row 1 represented the board state, and row 2 represented the Q values. If the structs didn't contain empty matrices, the program had already encountered those board states before, and the existing Q-values were used. Then, after each iteration, the Q-values were updated according to the rules described in the lecture notes.

The upside of this approach is that the training is much faster, as the program doesn't have to loop through the Q-table to find the existing entry, but instead knows almost instantly where to look. However, the program is less memory-efficient (as the Q-table must be initialized to be able to contain every possible value), and less intuitive. The program was run for 1e5 iterations, as visualised in figure 1. In the beginning, ϵ was set to 1 (completely random play), but after every $1e4$ iterations, ϵ was reduced by a factor 0.9 ($\epsilon \leftarrow \epsilon/10$).

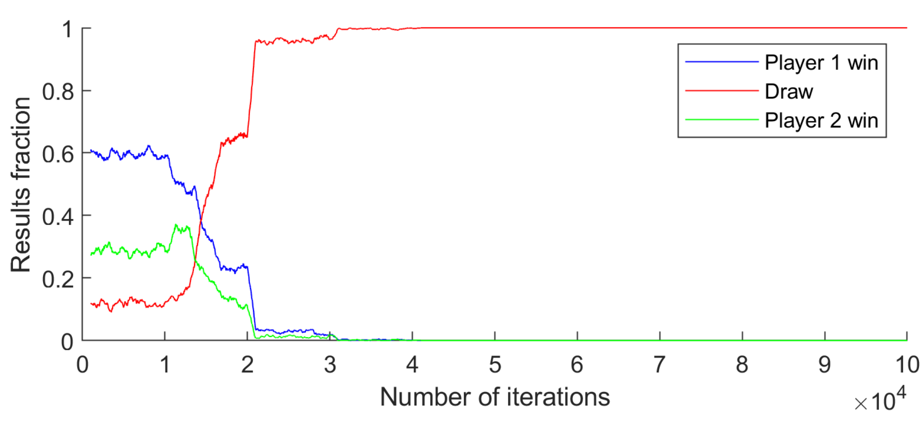


Figure 1: Game results as a function of the number of training iterations.

Convolutional networks

Table of Contents

Initialization

```
clear; clc;
loadNetworks = true;
```

Load and convert data to the desired format

```
[xTrain, tTrain, xValid, tValid, xTest, tTest] = LoadMNIST(3);
imageSize = [28 28 1];
```

Network 1

```
layers1 = [
    imageInputLayer(imageSize)

    convolution2dLayer(5, 20, 'Stride', 1, 'Padding',
 1, 'WeightsInitializer', 'narrow-normal');
    reluLayer

    maxPooling2dLayer(2, 'Stride', 2);

    fullyConnectedLayer(100, 'WeightsInitializer', 'narrow-normal')
    reluLayer

    fullyConnectedLayer(10, 'WeightsInitializer', 'narrow-normal')
    softmaxLayer

    classificationLayer];

options1 = trainingOptions('sgdm', ...
    'Momentum', 0.9, ...
    'MaxEpochs', 60, ...
    'MiniBatchSize', 8192, ...
```

```
    'InitialLearnRate',0.001, ...
    'ValidationPatience', 5, ...
    'ValidationFrequency', 30, ...
    'Shuffle', 'every-epoch', ...
    'ValidationData', {xValid, tValid}, ...
    'Plots','training-progress');

if ~loadNetworks
    [net1, trainingInfo1] = trainNetwork(xTrain, tTrain, layers1,
 options1);
end
```

Network 2

```
layers2 = [
    imageInputLayer(imageSize)
    convolution2dLayer(3, 20, 'Stride', 1, 'Padding',
 1, 'WeightsInitializer', 'narrow-normal');
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2, 'Stride', 2);

    convolution2dLayer(3, 30, 'Stride', 1, 'Padding',
 1, 'WeightsInitializer', 'narrow-normal');
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2, 'Stride', 2);

    convolution2dLayer(3, 50, 'Stride', 1, 'Padding',
 1, 'WeightsInitializer', 'narrow-normal');
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10, 'WeightsInitializer', 'narrow-normal')
    softmaxLayer
    classificationLayer];

options2 = trainingOptions('sgdm', ...
    'Momentum', 0.9, ...
    'MaxEpochs', 30, ...
    'MiniBatchSize', 8192, ...
    'InitialLearnRate',0.01, ...
    'ValidationPatience', 5, ...
    'ValidationFrequency', 30, ...
    'Shuffle', 'every-epoch', ...
    'ValidationData', {xValid, tValid}, ...
    'Plots','training-progress');
if ~loadNetworks
    [net2, trainingInfo2] = trainNetwork(xTrain, tTrain, layers2,
 options2);
end
```

Save networks

```
if ~loadNetworks

 save('Homework3_1_networks.mat', 'net1', 'trainingInfo1', 'net2', 'trainingInfo2'
end
```

Load networks

```
if loadNetworks
    load('Homework3_1_networks.mat')
end
```

Test network 1

```
trainAccuracy1 = trainingInfo1.TrainingAccuracy(end);
validAccuracy1 = trainingInfo1.ValidationAccuracy(end);
testAccuracy1 = GetTestAccuracy(net1, xTest, tTest)*100;

fprintf('Network 1 results: training accuracy = %.4f, validation
 accuracy = %.4f, and test accuracy = %.4f.\n', ...
    trainAccuracy1, validAccuracy1, testAccuracy1);
```

Test network 2

```
trainAccuracy2 = trainingInfo2.TrainingAccuracy(end);
validAccuracy2 = trainingInfo2.ValidationAccuracy(end);
testAccuracy2 = GetTestAccuracy(net2, xTest, tTest)*100;

fprintf('Network 2 results: training accuracy = %.4f, validation
 accuracy = %.4f, and test accuracy = %.4f.\n', ...
    trainAccuracy2, validAccuracy2, testAccuracy2);
```

Functions

```
function testAccuracy = GetTestAccuracy(network, xTest, tTest)
    yTest = classify(network, xTest);
    testAccuracy = sum(tTest == yTest)/numel(yTest);
end

function [trainAccuracy, validAccuracy, testAccuracy] =
 testNetwork(network, xTrain, tTrain, xValid, tValid, xTest, tTest)
    yTrain = classify(network, xTrain);
    trainAccuracy = sum(tTrain == yTrain)/numel(yTrain);

    yValid = classify(network, xValid);
    validAccuracy = sum(tValid == yValid)/numel(yValid);

    yTest = classify(network, xTest);
    testAccuracy = sum(tTest == yTest)/numel(yTest);
```

```
end
```

Published with MATLAB® R2020b

Fully connected Autoencoder

Table of Contents

Initialization

```
clear; clc;
loadNetworks = true;
```

Load and convert data to the desired format

```
[xTrain, tTrain, xValid, tValid, xTest, tTest] = LoadMNIST(1);
```

Autoencoder 1

```
layers1 = [
    sequenceInputLayer(784)

    fullyConnectedLayer(50, 'WeightsInitializer', 'glorot')
    reluLayer

    fullyConnectedLayer(2, 'WeightsInitializer', 'glorot')
    reluLayer

    fullyConnectedLayer(784, 'WeightsInitializer', 'glorot')
    reluLayer

    regressionLayer];

options = trainingOptions('adam', ...
    'MiniBatchSize', 8192, ...
    'InitialLearnRate',0.001, ...
    'Shuffle', 'every-epoch', ...
    'MaxEpochs', 10000, ...
```

```
    'Plots','training-progress');

if ~loadNetworks
    [net1, trainingInfo1] = trainNetwork(xTrain, xTrain, layers1,
 options);
end
```

Autoencoder 2

```
layers2 = [
    sequenceInputLayer(784)

    fullyConnectedLayer(50, 'WeightsInitializer', 'glorot')
    reluLayer

    fullyConnectedLayer(4, 'WeightsInitializer', 'glorot')
    reluLayer

    fullyConnectedLayer(784, 'WeightsInitializer', 'glorot')
    reluLayer

    regressionLayer];

if ~loadNetworks
    [net2, trainingInfo2] = trainNetwork(xTrain, xTrain, layers2,
 options);
end

if ~loadNetworks

 save('Homework3_2_networks.mat', 'net1', 'trainingInfo1', 'net2', 'trainingInfo2'
end

if loadNetworks
    load('Homework3_2_networks.mat');
end
```

Testing parameters

```
pauseTime = 1;
numberOfImages = 5;
```

Test network 1

```
figure(1)
index = randi([1 length(xTrain) - numberOfImages]);
for iImage = index:index+numberOfImages
    imageData = xTrain(:,iImage);

    subplot(1,2,1)
    image = reshape(imageData, 28, 28, 1);
    imshow(image);
```

```
    subplot(1,2,2)
    predictedImageData = predict(net1, imageData);
    predictedImage = reshape(predictedImageData, 28, 28, 1);
    imshow(predictedImage);

    pause(pauseTime);
end
```

Test network 2

```
figure(2)
for iImage = index:index+numberOfImages
    imageData = xTrain(:,iImage);

    subplot(1,2,1)
    image = reshape(imageData, 28, 28, 1);
    imshow(image);

    subplot(1,2,2)
    predictedImageData = predict(net2, imageData);
    predictedImage = reshape(predictedImageData, 28, 28, 1);
    imshow(predictedImage);
    a = iImage;
    pause(pauseTime);
end
```

Divide networks

```
net1_layers_encode(1:5) = [net1.Layers(1:4); regressionLayer];
net1_layers_decode(1:5) = [sequenceInputLayer(2); net1.Layers(5:8)];
net1_encode = assembleNetwork(net1_layers_encode);
net1_decode = assembleNetwork(net1_layers_decode);

net2_layers_encode(1:5) = [net2.Layers(1:4); regressionLayer];
net2_layers_decode(1:5) = [sequenceInputLayer(4); net2.Layers(5:8)];
net2_encode = assembleNetwork(net2_layers_encode);
net2_decode = assembleNetwork(net2_layers_decode);
```

Network 1 montage

```
currentDigit = 0;
iImage = 20;
figure(3)
clf
while currentDigit < 10
    if find(tTrain(:,iImage))-1 == currentDigit
        imageData = xTrain(:,iImage);

        subplot(5,4,currentDigit*2+1)
        image = reshape(imageData, 28, 28, 1);
        imshow(image);
```

```
        subplot(5,4,currentDigit*2+2)
        predictedImageData = predict(net1, imageData);
        predictedImage = reshape(predictedImageData, 28, 28, 1);
        imshow(predictedImage);
        currentDigit = currentDigit+1;
    end
    iImage = iImage+1;
end
```

Network 1 scatter plot

```
successfulDigits = [1];
colors = {'red'};
b = [];

figure(4)
clf
plotDigit = 1;

for iImage = 1:1000
    digit = find(tTrain(:,iImage))-1;
    a = predict(net1_encode, xTrain(:,iImage));
    if ismember(digit, successfulDigits)
        hold on
        if plotDigit < length(successfulDigits)+1
            if digit == successfulDigits(plotDigit)
                b(plotDigit) = plot(a(1), a(2), '.', 'color',
 char(colors(successfulDigits == digit)));
                plotDigit = plotDigit + 1;
                continue
            end
        end
        plot(a(1), a(2), '.', 'color', char(colors(successfulDigits ==
 digit)));
    else
        plot(a(1), a(2), '.blue')
    end
end
b(plotDigit) = plot(a(1), a(2), '.', 'color', 'blue');
legend(b, '1', 'Other digits','Location', 'north')
xlabel('First network output')
ylabel('Second network output')
```

Network 2 montage

```
currentDigit = 0;
iImage = 300;
figure(5)
clf
while currentDigit < 10
    if find(tTrain(:,iImage))-1 == currentDigit
        imageData = xTrain(:,iImage);
```

```
        subplot(5,4,currentDigit*2+1)
        image = reshape(imageData, 28, 28, 1);
        imshow(image);

        subplot(5,4,currentDigit*2+2)
        predictedImageData = predict(net2, imageData);
        predictedImage = reshape(predictedImageData, 28, 28, 1);
        imshow(predictedImage);
        currentDigit = currentDigit+1;
    end
    iImage = iImage+1;
end
```

Network 2 scatter

```
successfulDigits = [1,2,7];
colors = {'red', 'green', 'cyan'};
b = [];

figure(6)
clf
plotDigit = 1;

for iImage = 1:1000
    digit = find(tTrain(:,iImage))-1;
    a = predict(net2_encode, xTrain(:,iImage));
    if ismember(digit, successfulDigits)
        hold on
        if plotDigit < length(successfulDigits)+1
            if digit == successfulDigits(plotDigit)
                b(plotDigit) = plot(a(1), a(2), '.', 'color',
 char(colors(successfulDigits == digit)));
                plotDigit = plotDigit + 1;
                continue
            end
        end
        plot(a(1), a(2), '.', 'color', char(colors(successfulDigits ==
 digit)));
    else
        plot(a(1), a(2), '.blue')
    end
end
b(plotDigit) = plot(a(1), a(2), '.', 'color', 'blue');
legend(b, '1','2','7', 'Other digits')
xlabel('First network output')
ylabel('Second network output')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
b = [];

figure(7)
clf
```

```
plotDigit = 1;

for iImage = 1:1000
    digit = find(tTrain(:,iImage))-1;
    a = predict(net2_encode, xTrain(:,iImage));
    if ismember(digit, successfulDigits)
        hold on
        if plotDigit < length(successfulDigits)+1
            if digit == successfulDigits(plotDigit)
                b(plotDigit) = plot(a(3), a(4), '.', 'color',
 char(colors(successfulDigits == digit)));
                plotDigit = plotDigit + 1;
                continue
            end
        end
        plot(a(3), a(4), '.', 'color', char(colors(successfulDigits ==
 digit)));
    else
        plot(a(3), a(4), '.blue')
    end
end
b(plotDigit) = plot(a(3), a(4), '.', 'color', 'blue');
legend(b, '1','2','7', 'Other digits')
xlabel('Third network output')
ylabel('Fourth network output')
```

Check rule

```
figure(8)
a1 = [25; 3; 50; 175];
imageData = predict(net2_decode, a1);
image = reshape(imageData, 28, 28, 1);
imshow(image);

figure(9)
a2 = [9; 9; 2; 5];
imageData = predict(net2_decode, a2);
image = reshape(imageData, 28, 28, 1);
imshow(image);

figure(10)
a3 = [25; 25; 110; 35];
imageData = predict(net2_decode, a3);
image = reshape(imageData, 28, 28, 1);
imshow(image);
```

Published with MATLAB® R2020b

Restricted Boltzmann machine

Table of Contents

Initialization

```
clear; close all
```

Patterns

```
patterns = [[-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;-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; 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], ...
            [-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;-1;-1;-1; 1; 1; 1]];

possiblePatterns = zeros(9,512);

for i = 1:512
    possiblePatterns(:,i) = DecimalToState(i);
end

patternsDecimal = zeros(1, 14);
for i = 1:14
    patternsDecimal(i) = StateToDecimal(patterns(:,i));
end
```

Parameters

```
MArray                      =   [2 4 8 16];            % Hidden
 Neurons
N                           =   9;              % Input
max_epochs                  =   1000;
```

```
p                           =   14;
beta                        =   1;
iterationLength             =   100;
eta                         =   0.01;
verbose                     =   1;
frequencySum                =   1400;
PData                       =   1/14;
repeatsPerPattern           =   1;
divergenceIterationLength   =   100;
```

Training and plotting

```
iFigure = 1;
for M = MArray
    weightMatrix                =   -1+2*rand(M,N);
    thetaV                      =   -1+2*rand(1,N);
    thetaH                      =   -1+2*rand(M,1);
    divergenceArray             =   zeros(1, max_epochs);
    for iEpoch = 1:max_epochs
        frequency = zeros(512, 1);
        if(verbose && mod(iEpoch, max_epochs/10) == 0)
            fprintf('M is %d, Epoch is %d, %d percent done.\n', M,
 iEpoch, round(iEpoch/max_epochs*100));
        end
        deltaWeight = zeros([size(weightMatrix), 14]);
        deltaThetaV = zeros([size(thetaV), 14]);
        deltaThetaH = zeros([size(thetaH), 14]);
        for mu = 1:14
            pattern = patterns(:,mu);
            correctDecimal = StateToDecimal(pattern);
            v = pattern;
            for t = 1:iterationLength
                v = RunIteration(v, weightMatrix, beta, thetaV,
 thetaH);
            end
            deltaWeight(:,:,mu)  =  eta*(tanh(weightMatrix*pattern -
 thetaH)*pattern' - tanh(weightMatrix*v - thetaH)*v');
            deltaThetaV (:,:,mu) = -eta*(pattern - v);
            deltaThetaH (:,:,mu) = -eta*(tanh(weightMatrix*pattern -
 thetaH) - tanh(weightMatrix*v - thetaH));
        end
        weightMatrix = weightMatrix + sum(deltaWeight,3);
        thetaV       = thetaV +       sum(deltaThetaV,3);
        thetaH       = thetaH +       sum(deltaThetaH,3);

        divergenceArray(iEpoch) =
 GetKullbackLeiblerDivergence(weightMatrix, ...
            repeatsPerPattern, divergenceIterationLength,
 possiblePatterns, ...
            patternsDecimal, beta, thetaV, thetaH);
    end

    figure(iFigure)
```

```
    clf
    plot(divergenceArray);
    xlabel('Epoch number')
    ylabel('Divergence')
    iFigure = iFigure + 1;

    figure(iFigure)
    clf
    middleAndRightColumnIndices = [2,3,5,6,8,9];
    v = patterns(:,14);
    v(middleAndRightColumnIndices) = 0;
    for iteration = 1:10
        subplot(1,10,iteration);
        PlotPattern(v);
        v = RunIteration(v, weightMatrix, beta, thetaV, thetaH);
    end
    iFigure = iFigure + 1;
end
```

Functions

```
function divergence = GetKullbackLeiblerDivergence(weightMatrix, ...
    repeatsPerPattern, iterationLength, possiblePatterns, ...
    patternsDecimal, beta, thetaV, thetaH)

    frequency = zeros(512, 1);
    for t = 1:512*repeatsPerPattern
        v = possiblePatterns(:,ceil(t/repeatsPerPattern));

        for i = 1:iterationLength
            v = RunIteration(v, weightMatrix, beta, thetaV, thetaH);
            decimalRepresentation = StateToDecimal(v);
            frequency(decimalRepresentation) =
 frequency(decimalRepresentation) + 1;
        end
    end

    frequencySum = 512*repeatsPerPattern*iterationLength;
    PB = frequency/frequencySum;
    PData = 1/14;
    divergence = 0;
    for mu = 1:14
        patternIndex = patternsDecimal(mu);
        divergence = divergence+PData*log(PData/PB(patternIndex));
    end
end

function v = RunIteration(v, weightMatrix, beta, thetaV, thetaH)
    b_h = (weightMatrix*v) - thetaH;
    h   = GetNextNeuronState(b_h,beta);
    b_v = (h*weightMatrix) - thetaV;
    v   = GetNextNeuronState(b_v, beta);
end
```

```
function number = StateToDecimal(state)
    state(state==-1) = 0;
    number = 1;
    for j = 1:9
        number = number + state(j)*2^(9-j);
    end
end

function state = DecimalToState(number)
    number = number-1;
    state = zeros(9, 1);
    for j = 1:9
        state(10-j) = mod(number,2);
        number = floor(number/2);
    end
    state(state == 0) = -1;
end

function s = GetNextNeuronState(b, beta)
    b = b';
    p = 1./(1+exp(-2*b*beta));
    s = zeros(size(b));
    for i = 1:length(p)
        r = rand;
        if r > p(i)
            s(i) = 1;
        else
            s(i) = -1;
        end
    end
end

function PlotPattern(pattern)
    image = reshape(pattern, 3, 3)';
    for x = 1:3
        for y = 1:3
            if image(y,x) == 1
                rectangle('Position', [x-1, 3-y, 1, 1], 'FaceColor',[0
 0 0]);
            elseif image(y,x) == 0
                rectangle('Position', [x-1, 3-y, 1, 1], 'FaceColor',
[0.7 0.7 0.7]);
            elseif image(y,x) == -1
                rectangle('Position', [x-1, 3-y, 1, 1], 'FaceColor',[1
 1 1]);
            end
            rectangle('Position', [x-1, 3-y, 1, 1], 'EdgeColor', [0.5
 0.5 0.5]);
        end
    end
    xlim([0 3])
    ylim([0 3])
```

```
end
```

Published with MATLAB® R2020a

Tic tac toe

Table of Contents

Initialization

```
clear; clc;
```

Parameters

```
initialQValue = 1;
initial_epsilon = 1;
alpha = 1;
max_epochs = 1e5;
verbose = true;
state0 = zeros(3);
player1QTable = InitializeQTable;
player2QTable = InitializeQTable;

epsilon = initial_epsilon;
results = zeros(1,max_epochs);
```

Training

```
for epoch = 1:max_epochs
    if mod(epoch, 10000) == 0
        epsilon = epsilon/10;
    end
    if verbose
        if mod(epoch,max_epochs/50) == 0
            fprintf('Epoch %d: %d%% done.\n', epoch, round(epoch/
max_epochs*100))
        end
    end
    state = state0;
    visitedStateIndices = zeros(1,9);
    moveIndices = zeros(1,9);
    for t = 1:9
        numberOfTurns = t;
```

```
        currentStateIndex = stateToNumber(state);
        visitedStateIndices(t) = currentStateIndex;
        [qValues, player1QTable, player2QTable] =
 GetStateQValues(state, player1QTable, player2QTable, t,
 initialQValue);
        [state, move] = MakeMove(currentStateIndex, t, player1QTable,
 player2QTable, epsilon);
        moveIndices(t) = move;
        if t > 4
            player1reward = GetReward(state);
            if player1reward ~= 0
                break
            end
        end
    end
    results(epoch) = GetReward(state);
    [player1QTable, player2QTable] = updateQTables(state,
 player1QTable, player2QTable, moveIndices, visitedStateIndices,
 numberOfTurns, alpha);
end
```

Plotting

```
figure(1)
clf
hold on
windowSize = 1000;
windowStep = 20;
slidingWindow = [];
win = [];
draw = [];
lose = [];
for epoch = windowSize:windowStep:max_epochs
    results_window = results(epoch-windowSize+1:epoch);
    slidingWindow = [slidingWindow, epoch];
    win = [win, numel(find(results_window==1))/(windowSize)];
    draw = [draw, numel(find(results_window == 0))/(windowSize)];
    lose = [lose, numel(find(results_window == -1))/(windowSize)];
end
plot(slidingWindow, win, 'b')
plot(slidingWindow, draw, 'r')
plot(slidingWindow, lose, 'g')
xlabel('Number of iterations')
legend('Player 1 win', 'Draw', 'Player 2 win')
ylabel('Results fraction')
```

Save Q Tables

```
saveQTable(player1QTable, 'player1.csv');
saveQTable(player2QTable, 'player2.csv');
```

Functions

```
function [player1QTable, player2QTable] = updateQTables(state,
 player1QTable, player2QTable, moveIndices, visitedStateIndices,
 numberOfTurns, alpha)
    [player1reward, player2reward] = GetReward(state);

    % Player 1
    player1LastTurn = numberOfTurns-mod(numberOfTurns+1,2);
    t = player1LastTurn;
    move = moveIndices(t);
    visitedStateIndex = visitedStateIndices(t);
    player1QTable(2, visitedStateIndex).entry(move) = ...
        player1QTable(2, visitedStateIndex).entry(move) + alpha*(...
            player1reward - player1QTable(2,
 visitedStateIndex).entry(move));

    for t = player1LastTurn-2:-2:1
        visitedStateIndex = visitedStateIndices(t);
        nextVisitedStateIndex = visitedStateIndices(t+2);
        move = moveIndices(t);

        player1QTable(2, visitedStateIndex).entry(move) = ...
            player1QTable(2, visitedStateIndex).entry(move) +
 alpha*(...
                -player1QTable(2, visitedStateIndex).entry(move) + ...
                max(max(player1QTable(2,
 nextVisitedStateIndex).entry)));
    end

    % Player 2
    player2LastTurn = numberOfTurns-mod(numberOfTurns,2);
    t = player2LastTurn;
    move = moveIndices(t);
    visitedStateIndex = visitedStateIndices(t);
    player2QTable(2, visitedStateIndex).entry(move) = ...
        player2QTable(2, visitedStateIndex).entry(move) + alpha*(...
            player2reward - player2QTable(2,
 visitedStateIndex).entry(move));

    for t = player2LastTurn-2:-2:2
        visitedStateIndex = visitedStateIndices(t);
        nextVisitedStateIndex = visitedStateIndices(t+2);
        move = moveIndices(t);

        player2QTable(2, visitedStateIndex).entry(move) = ...
            player2QTable(2, visitedStateIndex).entry(move) +
 alpha*(...
                -player2QTable(2, visitedStateIndex).entry(move) + ...
                max(max(player2QTable(2,
 nextVisitedStateIndex).entry)));
    end
end
```

```
function [player1reward, player2reward] = GetReward(state)
    for i = 1:3
        if all(state(i, :) == 1) || all(state(:,i) == 1)
            player1reward = 1; player2reward = -1;
            return
        elseif all(state(i,:) == -1) || all(state(:,i) == -1)
            player1reward = -1; player2reward = 1;
            return
        end
    end
    i = [1,2,3];
    if all(diag(state) == 1) || all(diag(flip(state)) == 1)
        player1reward = 1; player2reward = -1;
    elseif all(diag(state) == -1) || all(diag(flip(state) == -1))
        player1reward = -1; player2reward = 1;
    else
        player1reward = 0; player2reward = 0;
    end
end


function [qValues, player1QTable, player2QTable] =
 GetStateQValues(state, player1QTable, player2QTable, t,
 initialQValue);
    if mod(t,2) == 1
        qTable = player1QTable;
    else
        qTable = player2QTable;
    end
    currentStateIndex = stateToNumber(state);
    if(isempty(qTable(2, currentStateIndex).entry))
        qTable(1, currentStateIndex).entry = state;
        qValues = ones(3)*initialQValue;
        qValues(state ~= 0) = NaN;
        qTable(2, currentStateIndex).entry = qValues;
    else
        qValues = qTable(2, currentStateIndex).entry;
    end
    if mod(t,2) == 1
        player1QTable = qTable;
    else
        player2QTable = qTable;
    end
end

function [newState, move] = MakeMove(currentStateIndex, t,
 player1QTable, player2QTable, epsilon)
    if mod(t,2) == 1
        qTable = player1QTable;
    else
        qTable = player2QTable;
    end
    state   = qTable(1, currentStateIndex).entry;
```

```
    qValues = qTable(2, currentStateIndex).entry;

    newState = state;
    playerMove = mod(t,2)*2-1;
    r = rand;

    if r < epsilon
        possibleMoves = find(~isnan(qValues));
    else
        possibleMoves = find(qValues == max(max(qValues)));
    end
    move = possibleMoves(randi(numel(possibleMoves)));
    newState(move) = playerMove;
end

function qTable = InitializeQTable
    qTable(2, 19683) = struct('entry',[]);
end

function number = stateToNumber(state)
    state = state+1;
    number = 1;
    for i = 1:9
        number = number+state(10-i)*3^(i-1);
    end
end

function testStateToNumber()
    for i = 0:19682
        stringArray = double(string(num2cell(dec2base(i,3))));
        state = reshape([zeros(1, 9-length(stringArray)),
 stringArray],3,3)-1;
        assert(stateToNumber(state) == i+1);
    end
end

function newQTable = saveQTable(qTable, filename)
    newQTable = cell(2, 19683);
    j = 1;
    for i = 1:19683
        state   = qTable(1, i).entry;
        qValues = qTable(2, i).entry;
        if ~isempty(state)
            newQTable(1,j) = {state};
            newQTable(2,j) = {qValues};
            j = j+1;
        end
    end
    newQTable(:, j:end) = [];
    newQTable2 = zeros(size(newQTable)*3);
    for j = 1:size(newQTable,2)
        for i = 1:2
            newQTable2(i*3-2:i*3,j*3-2:j*3) =
 cell2mat(newQTable(i,j));
```

```
        end
    end
    writematrix(newQTable2, filename);
end
```

Published with MATLAB® R2020b