Computer Vision – Programming Project 2

王順興 0210184

Structure:

Linear Softmax Regression

1. LinearSoftmaxRegression.m: The learning process
2. gradientDescentWD.m: The gradient descent for one single iteration
3. softmax.m: The Softmax function I'm using

Dual Softmax Regression

4. DualSoftmaxRegression.m: The learning process
5. gradientDescent.m: The gradient descent for one single iteration
6. dualSoftmax.m: The Dual Softmax function I'm using

Sigmoid Neural Network

7. SigmoidNN.m: The Neural Network learning process
8. sigmoid.m: The Sigmoid function I'm using

Files for reading dataset and testing are omitted.
Full project: https://github.com/vicodin1123/CV-project/tree/master/p2

Linear Softmax Regression - LinearSoftmaxRegression.m

```
% Linear Softmax Regression

MAX_ITERATION = 200;        % Stop the iteration when this is reached
LEARNING_RATE = 1;          % Learning rate
WEIGHT_DECAY  = 0.0000;     % Weight decay parameter

% DxK model parameter, +1 for bias. Randomly intialized
theta = 0.0005 * randn(DIMENSION + 1, MAX_CLASS);

% Train until maximum iterations is reached
% Can be modified to stop when gradient change and avg. loss is small
for i = 1 : MAX_ITERATION

  % Gradient Descent, get the Loss and Gradient
   [L, g] = gradientDescentWD(TrainSet, theta, WEIGHT_DECAY);

  % Update theta
   theta = theta - LEARNING_RATE.*g./TrainCount;
   disp(L/TrainCount);
end
```

Linear Softmax Regression - gradientDescentWD.m

```
function [L, g] = gradientDescentWD(X, Theta, WD)
% Batch Gradient Descent with Weight Decay
% INPT: x: DxI matrix. The dataset with D dimensions and I samples
%       Theta: DxK matrix. The model parameters with D dimensions of K classes
%       WD: Scalar. The weight of Weight Decay term
% OUPT: L: scalar. The cost function of Softmax Regression calculated using Theta
%       g: DxK matrix. The gradient of parameter matrix

L = 0;  % Loss
g = zeros(size(Theta)); % Gradient matrix, matches the size of Theta
K = size(Theta, 2);         % # of Class



% For each class in training set
for i = 1 : K
    target = i;    % The target of this subset
    xi = X{i}.';   % The samples of this subset
   % Softmax function, the reuslt is KxI
    yi = softmax(xi, oldTheta);

   % Update the loss
    L = L - (sum(log( yi(target, :) )) + WD*(sum(sum(oldTheta.^2))));

    % Update the graident
    for n = 1 : K
        for r = 1 : size(X{i}, 1)
            if target == n
                g(:, n) = g(:, n) + (yi(n, r) - 1)*xi(:, r) + WD*oldTheta(:, n);
            else
                g(:, n) = g(:, n) + yi(n, r)*xi(:, r) + WD*oldTheta(:, n);
            end
        end
    end
end
end
```

Linear Softmax Regression - softmax.m

```
function Y = softmax(X, Theta)
% Softmax function
% INPT: X: DxI matrix. The dataset with D dimensions and I samples
%                  theta: DxK matrix. The model parameters with D dimensions of K
classes
% OUPT: Y: KxI matrix. The result of Softmax with K classes and I samples

K = size(Theta, 2); % # of Class

weight = exp(Theta.' * X);
Y = weight ./ repmat(sum(weight), K, 1);

end
```

Dual Softmax Regression - DualSoftmaxRegression.m

```
% Dual Softmax Regression Learning

MAX_ITERATION = 2000;       % Stop the iteration when this is reached
LEARNING_RATE = 0.002;      % Learning rate
psi = cell(1, MAX_CLASS);   % 1xK Ix1 The psi for K class, each with I samples

% Initialize the psi with random number
for i = 1 : MAX_CLASS
    psi{i} = 0.005 * randn(TrainCounts(i), 1);
end

% Gradient Descent process
for i = 1 : MAX_ITERATION

         % Gradient Descent, get the Loss and Gradient
    [L, g] = gradientDescent(TrainSet, psi);

         % Update psi
    for r = 1 : MAX_CLASS
        psi{r} = psi{r} - LEARNING_RATE.*g{r}./TrainCount;
    end
    disp(sprintf('%d %.3f', i, L/TrainCount));
end

disp(sprintf('Iterations: %d Final loss: %.2f', i, L/TrainCount));
```

Dual Softmax Regression - gradientDescent.m

```
function [L, g] = gradientDescent(X, psi)
% Gradient Descent using algorithm 9.8
% INPT: X: DxI matrix. The dataset with D dimensions and I samples
%       psi: 1xKxIx1 matrix. The psi for K class, each with I samples
% OUPT: L: scalar. The cost function of Softmax Regression calculated using psi
%       g: DxK matrix. The gradient of parameter matrix

K = size(psi, 2);

L = 0;    % Loss
g = cell(1, size(psi, 2));  % Gradient matrix, matches the size of psi
% Initialize the gradient
for i = 1 : size(psi, 2)
    g{i} = zeros(size(psi{i}));
end

% For each class in training set
for i = 1 : K
    target = i;    % The target of this subset

    xi = X{i}.';   % The samples of this subset

         % Dual Softmax function, the reuslt is KxI
    yi = dualSoftmax(X, psi, xi);

         % Update the loss
    L = L - sum(log(yi(target, :)));

         % Update the graident
    for n = 1 : K
        for r = 1 : size(X{i}, 1)
            if target == n
                g{n} = g{n} + (yi(n, r) - 1)*X{n}*xi(:, r);
            else
                g{n} = g{n} + yi(n, r)*X{n}*xi(:, r);
            end
```

```
        end
    end
end

end
```

Dual Softmax Regression - dualSoftmax.m

```
function Y = dualSoftmax(X, Psi, x)
% Softmax function with Dual Activation
% INPT: X: 1xKxIxD matrix. The training dataset with D dimensions and I samples
%                   Psi: 1xKxIx1 matrix. The psi for K class, each with I samples
%       x: DxI matrix. The testing dataset with D dimensions and I samples
% OUPT: Y: KxI matrix. The result of Softmax with K classes and I samples

K = size(Psi, 2);
I = size(x, 2);

weights = zeros(K, I); % KxI
powers = zeros(K, I);
for i = 1 : K
    % 1xI * IxD * DxI = 1xI
    powers(i, :) = Psi{i}' * X{i} * x;
end
weights = exp(powers);
Y = weights ./ repmat(sum(weights), K, 1);

end
```

Sigmoid Neural Network - SigmoidNN.m

```
% Sigmoid Neural Network with 1 hidden layer

LEARNING_RATE = 0.5;

LAYER_COUNT = 3;    % Input/Hidden/Output
NEURON_COUNTS = [DIMENSION round(DIMENSION/2) MAX_CLASS]; % # of neurons for each
layer
PARAM_COUNTS = [0 NEURON_COUNTS(1:end-1)];  % # of parameters for neurons in each
layer

bias = cell(1, LAYER_COUNT-1);      % bias of each neuron in each layer
weight = cell(1, LAYER_COUNT-1);    % weight of each neuron in each layer

% Randomly initialize weight and bias
for i = 1 : LAYER_COUNT-1
    weight{i} = unifrnd(-0.5, 0.5, NEURON_COUNTS(i+1), PARAM_COUNTS(i+1));
    bias{i} = unifrnd(-0.5, 0.5, NEURON_COUNTS(i+1), 1);
end

activation = cell(1, LAYER_COUNT);  % Output of each layer
delta = cell(1, LAYER_COUNT);       % Error of each layer
batchGW = cell(1, LAYER_COUNT-1);   % Gradient of weight
batchGB = cell(1, LAYER_COUNT-1);   % Gradient of bias
```

```
for itr = 1 : 500

errorCount = zeros(size(TrainSet));

for i = 1 : LAYER_COUNT-1

    batchGW{i} = zeros(size(weight{i}));

    batchGB{i} = zeros(size(bias{i}));

end


L = 0;

gW = cell(1, LAYER_COUNT-1);

gB = cell(1, LAYER_COUNT-1);


for i = 1 : MAX_CLASS

    Target = zeros(MAX_CLASS, 1); Target(i) = 1;

    for r = 1 : size(TrainSet{i}, 1)  % For each sample


        % Forward propagation

        activation{1} = TrainSet{i}(r, :).';  % Let X be the output of 1st layer


        for l = 1 : LAYER_COUNT - 1

             z = weight{l}*activation{l} + bias{l};

             activation{l+1} = sigmoid(z);

        end

        L = L + 0.5 * norm((activation{LAYER_COUNT} - Target), 2);


        [val ind] = max(activation{LAYER_COUNT}.');

        if ind ~= Target

            errorCount(i) = errorCount(i) + 1;

        end


        % Backpropagation

        delta{LAYER_COUNT} = -(Target - activation{LAYER_COUNT}) .* ...

            (activation{LAYER_COUNT}.*(1-activation{LAYER_COUNT})) ;

        for l = LAYER_COUNT - 1: -1 : 1

            delta{l} = ((weight{l}')*delta{l+1}) .* ...

                (activation{l}.*(1-activation{l})) ;

        end


        % Update gradient
```

```
        for l = 1 : LAYER_COUNT - 1
            gW{l} = delta{l+1} * (activation{l}');
            gB{l} = delta{l+1};
            batchGW{l} = batchGW{l} + gW{l};
            batchGB{l} = batchGB{l} + gB{l};
        end

    end
end

for i = 1 : LAYER_COUNT-1
    weight{i} = weight{i} - LEARNING_RATE * (batchGW{i}./TrainCount);
    bias{i} = bias{i} - LEARNING_RATE * (batchGB{i}./TrainCount);
end
L = L/TrainCount;
disp(L);
disp(sum(errorCount));
end
```

Sigmoid Neural Network - sigmoid.m

```
function Y = sigmoid(X)
% Softmax function
% INPT: X: DxI matrix. The dataset with D dimensions and I samples
% OUPT: Y: Ix1 matrix. The result of Sigmoid of I samples

Y = 1 ./ (1 + exp(-X));


end
```