EEC 289Q Data Analytics for Computer Engineers
Homework 4

Ahmed Mahmoud

May, 25th 2018

Convolution and Pooling:

The following code shows the implementation convolution operation

```
function convolvedFeatures = cnnConvolve(filterDim, numFilters, images, ...
   W, b)

    numImages = size(images, 3);
    imageDim = size(images, 1);
    convDim = imageDim - filterDim + 1;
    convolvedFeatures = zeros(convDim, convDim, numFilters, numImages);

    for imageNum = 1:numImages
      for filterNum = 1:numFilters
        convolvedImage = zeros(convDim, convDim);

        %%% YOUR CODE HERE %%%
        filter = W(:,:,filterNum);
        filter = rot90(squeeze(filter),2);
        im = squeeze(images(:, :, imageNum));

        %%% YOUR CODE HERE %%%
        convolvedImage = conv2(im, filter,'valid');

        %%% YOUR CODE HERE %%%
        convolvedImage = convolvedImage + b(filterNum);
        convolvedImage  = sigmoid(convolvedImage);
        convolvedFeatures(:, :, filterNum, imageNum) = convolvedImage;

      end
    end
end
```

The following code shows the implementation average pooling operation

```
function pooledFeatures = cnnPool(poolDim, convolvedFeatures)

    numImages = size(convolvedFeatures, 4);
    numFilters = size(convolvedFeatures, 3);
    convolvedDim = size(convolvedFeatures, 1);

    pooledFeatures = zeros(convolvedDim / poolDim, ...
            convolvedDim / poolDim, numFilters, numImages);

    %%% YOUR CODE HERE %%%
    pool_window = ones(poolDim)/poolDim^2;
    out_size = 1:poolDim:convolvedDim;
    for imageNum = 1:numImages
        for filterNum = 1:numFilters
            conv_out = conv2(convolvedFeatures(:,:,filterNum,imageNum),...
                pool_window,'valid');
            pooledFeatures(:,:,filterNum,imageNum) = ...
               conv_out(out_size,out_size);
        end
    end
end
```

These two pieces of codes have passed the tests in cnnExercise.m successfully.

Convolutional Neural Network:

Our implementation for the CNN is shown in the following four codes/functions. Our implementation passed the gradient check with a difference between numerical gradient and ours of 2.3679×10^{-10} which is less than the tolerance (i.e., 10^{-9}). After training, the accuracy of our implementation after three epochs was 97.26% and the training took in average 4 minutes.

```
function [cost, grad, preds] = cnnCost(theta,images,labels,numClasses,...
                                filterDim,numFilters,poolDim,pred)
    if ¬exist('pred','var')
        pred = false;
    end;


    imageDim = size(images,1); % height/width of image
    numImages = size(images,3); % number of images

    %% Reshape parameters and setup gradient matrices
    [Wc, Wd, bc, bd] = ...
       cnnParamsToStack(theta,imageDim,filterDim,numFilters,...
                            poolDim,numClasses);

    % Same sizes as Wc,Wd,bc,bd. Used to hold gradient w.r.t above params.
    Wc_grad = zeros(size(Wc));
    Wd_grad = zeros(size(Wd));
    bc_grad = zeros(size(bc));
    bd_grad = zeros(size(bd));

    convDim = imageDim-filterDim+1; % dimension of convolved output
    outputDim = (convDim)/poolDim; % dimension of subsampled output

    % convDim x convDim x numFilters x numImages tensor for storing ...
       activations
    activations = zeros(convDim,convDim,numFilters,numImages);

    % outputDim x outputDim x numFilters x numImages tensor for storing
    % subsampled activations
    activationsPooled = zeros(outputDim,outputDim,numFilters,numImages);

    %%% YOUR CODE HERE %%%
    activations = cnnConvolve(filterDim, numFilters, images, Wc, bc);
    activationsPooled = cnnPool(poolDim,activations);

    % Reshape activations into 2-d matrix, hiddenSize x numImages,
    % for Softmax layer
    activationsPooled = reshape(activationsPooled,[],numImages);

    probs = zeros(numClasses,numImages);

    %%% YOUR CODE HERE %%%
    soft = Wd*activationsPooled + repmat(bd,1,numImages);
    soft = exp(soft - max(soft,[],1));
    probs = soft./sum(soft);

    cost = 0; % save objective into cost

    %%% YOUR CODE HERE %%%
    %one-hot encoding
    hot = full(sparse(labels,1:numImages,1));
    cost = -hot(:)'*log(probs(:))/numImages;
    % Makes predictions given probs and returns without backproagating ...
       errors.
    if pred
        [¬,preds] = max(probs,[],1);
        preds = preds';
        grad = 0;
        return;
    end;

    %%% YOUR CODE HERE %%%
    dalta = probs-hot;
    term1 = Wd'*((1/numImages).*dalta);
    err_tmp = reshape(term1,outputDim,outputDim,numFilters,numImages);
    for I=1:numImages
        for F=1:numFilters
            err(:,:,F,I) = ...
                (1/(poolDim^2)).*kron(squeeze(err_tmp(:,:,F,I))...
                        ,ones(poolDim,poolDim));
        end
    end
    err = activations.*(1.0 -activations).*err;

    %%% YOUR CODE HERE %%%
    for I=1:numImages
        for F=1:numFilters
            Wc_grad(:,:,F) = Wc_grad(:,:,F)+conv2(images(:,:,I),...
                    rot90(err(:,:,F,I),2),'valid');
            bc_grad(F) = bc_grad(F) + sum(sum(err(:,:,F,I)));
        end
    end
    Wd_grad= dalta*(activationsPooled)'/numImages;
    bd_grad = (1/numImages).*dalta*ones(numImages,1);

    %% Unroll gradient into grad vector for minFunc
    grad = [Wc_grad(:) ; Wd_grad(:) ; bc_grad(:) ; bd_grad(:)];
end
```

```
function [opttheta] = minFuncSGD(funObj,theta,data,labels,...
                        options)
    assert(all(isfield(options,{'epochs','alpha','minibatch'})),...
            'Some options not defined');
    if ¬isfield(options,'momentum')
        options.momentum = 0.9;
    end
    epochs = options.epochs;
    alpha = options.alpha;
    minibatch = options.minibatch;
    m = length(labels); % training set size
    % Setup for momentum
    mom = 0.5;
    momIncrease = 20;
    velocity = zeros(size(theta));

    %%=======================================================
    %% SGD loop
    it = 0;
    for e = 1:epochs

        % randomly permute indices of data for quick minibatch sampling
        rp = randperm(m);

        for s=1:minibatch:(m-minibatch+1)
            it = it + 1;
            % increase momentum after momIncrease iterations
            if it == momIncrease
                mom = options.momentum;
            end
            % get next randomly selected minibatch
            mb_data = data(:,:,rp(s:s+minibatch-1));
            mb_labels = labels(rp(s:s+minibatch-1));
            % evaluate the objective function on the next minibatch
            [cost, grad] = funObj(theta,mb_data,mb_labels);

            %%% YOUR CODE HERE %%%
            velocity =(mom.*velocity)+(alpha.*grad);
            theta = theta - velocity;

            fprintf('Epoch %d: Cost on iteration %d is %f\n',e,it,cost);
        end

        % aneal learning rate by factor of two after each epoch
        alpha = alpha/2.0;
    end
    opttheta = theta;
end
```