PHASE 2 REPORT

EEE 485 TERM PROJECT

Wednesday, April 24, 2019

Ahmad Zafar Khan
21403048

Hassan Ahmed
21500439

Contents

Problem and Dataset Description

The dataset we plan to use is a collection of images of fruits taken from different rotated angles. The total size of the dataset is 65429 images. It is divided into a training and test set of 48905 images and 16421 images respectively. There are 95 different fruits which means the classification algorithm we implement will have 95 classes to choose from. There is also a smaller dataset of 103 images with multiple fruits in the same image. This can be useful to test our algorithm on more complex training sets to see how they would perform in situations more similar to the real world. The dataset also does not combine different varieties of fruit so apples are separated into five different classes corresponding to five varieties.

For the dataset, we conducted some basic processing on it. This was done to standardize the data instances. To remove the variations between different types of data. This was done by subtracting the mean and dividing by the standard deviation. We can also perform PCA on the data to lower the dimensionality of the data. This will help increase the accuracy too and is planned for the next stage.

Review of Methods

1. Neural Network

For the third method, we created a Convolutional Neural Network to train the model to recognize the type of fruit from a given image. This network can be divided into two separate parts to clearly understand how it works and was implemented. The first part of the model works by passing the data into 2 convolution layers and then a maxpool layer. Then it is passed into a fully connected hidden layer that leads to the output layer.

The convolution layers work by applying filters to the image data to extract information from it. This works by multiplying a smaller matrix of numbers to the image dataset and striding through the image one by one both horizontally and vertically. The resulting matrix is a smaller one which reduces dimensionality and makes it faster and more feasible for the fully connected layer to operate. The convolution is done by all the initialized filters. The dimensions of the input to the first convolution layer is number of channels * height * width. The number of channels are usually three for the RGB values of an image but for our case it is the number of methods of normalization for the orientation gradients which are 4 in our case. When passed through the layer the dimensions become, number of filters * height * width. The feed forward layer equations are:

$$z_{x,y}^{l} = w^{l} * \sigma\left(z_{x,y}^{l-1}\right) + b_{x,y}^{l}$$

Where $z = fieldoutput, \sigma = activationfunction$

For the back propagation we use the gradients of the functions.

$$\delta_j^l = -\frac{\partial e(w)}{\partial v_j^l}$$

And

$$\delta_i^{l-1} = \sum_{j=1}^{d^l} -\frac{\partial e(w)}{\partial v_j^l}\frac{\partial v_j^l}{\partial x_i^{l-1}}\frac{\partial x_i^{l-1}}{\partial v_j^{l-1}} = \delta_j^{l-1} \times w_{ij}^l \times \phi'(v_i^{l-1})$$

$$w^l_{i,j_{old}} = w^l_{i,j_{new}} + \delta^l_j x^{l-1}_i$$

Training Phase

For updating our algorithm we used a version of the stochastic gradient descent known as adam gradient descent. This estimates the value of the mean and variance using parameters β_1 and β_2 and uses these to update the value of the weights/filters/biases. The formula used for updating these rules is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \times m_t$$

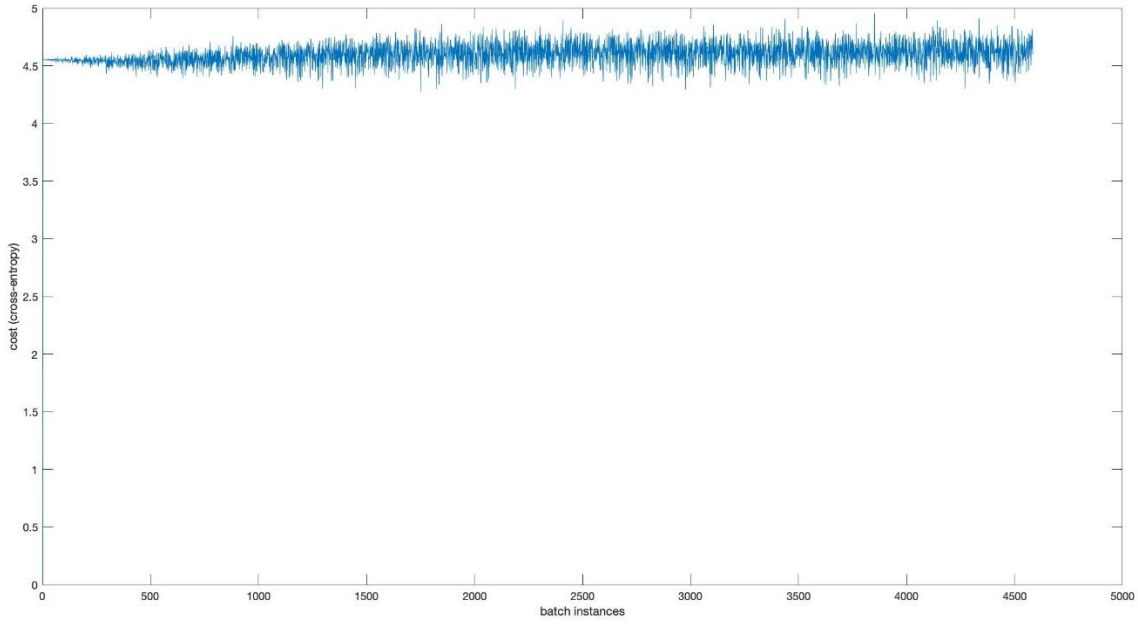
Where $m_t = mean, v_t = variance$

The activation functions for the convolution, maxpooling and first hidden layers were ReLu while the output was the softmax function since we wanted the probabilities to sum to one for the output. The hidden layer had 100 nodes. They were all fully connected.

The time taken to train the dataset is 11.6 hours for 2 epochs. The batch size was taken as 32 datapoints from a training dataset of 48,905 images.

Testing Phase and Results

When the accuracy of the model was tested it came to 2% which implied that the model could not train properly. It will require a lot more epochs than two to train completely. This is because of high number of classes. This will need a much more powerful computer to train but we can still show some results.



2. Naïve Bayes Classifier

The Naïve Bayes classifier is based on the bayes rule given below:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

The object of this classifier is to learn the probability $P(Y|X)$ for each class. Such a classifier can be extended to multiple classes therefore it was chosen.

Given that the feature set of the fruit data, which was a 20x20x4 matrix of HOG values for each label, was composed of continuous values we modeled the feature set on the gaussian distribution as follows:

$$P(Y = y_k|X_1, X_2, \ldots, X_n) = \frac{1}{\sqrt{2\pi\sigma_x^2}} exp(\frac{-(y_k - \mu_x)^2}{2\sigma_x^2})$$

Training Phase

The mean and standard deviations were learned for each class. This was done by first isolating the each label of class k $k = 1,2,\ldots,95$. The corresponding feature vectors in the train set were also isolated and the mean and standard deviation for each variable in the training feature set was learned from these isolated sets. This was continued until we had a mean and standard deviation of each of 1600 features for each class. So in total we had $1600 * 95 = 152000$ values for the mean and the standard deviation. These values were used to form the likelihood function for each class:

$$L(\mu, \sigma^2; x_1, \ldots x_1 600) = \prod_{i=1}^{1600} (2\pi\sigma_i^2)^{-1/2} exp(\frac{-(x_i - \mu_i)^2}{2\sigma_i^2})$$

These observation are independent as per the naive assumption therefore we multiply these probabilities. And we do so for each class. The value for x_i will be each feature vector of the test set in the testing phase.

We also had to learn the prior probabilities for each class $y_k = 1, \ldots 95$. This was done by simply finding each instance of the class y_k in the training label set and then dividing it by the total number of labels in the class. i.e.:

$$P(Y = y_k) = \frac{n}{\Sigma_{i=1}^{95} y_i}$$

Where n is the number of instances of label y_k in the label set.

Testing Phase

After the aforementioned parameters were learned, it was time to test. For this part We used the log likelihood function iteratively over each class of label. This was done for each feature vector. Then whichever value of y_k for the likelihood function gave us the maximum value, the feature vector was attributed to belonging to that set. i.e.:

$$y_k = \underset{y_k}{\text{argmax}}(logP(Y = y_k) + \Sigma_{i=1}^{1600} logP(X_i|Y = y_k))$$

Where the probability of X given Y is the gaussian distribution given above, with each variable in the feature set occupying the x_i variable in the gaussian distribution with its corresponding mean and standard deviation. The probability of y is the prior probability given above.

Results

The accuracy is roughly 23%. We believe that for naive bayes classifier, a low accuracy is to be expected, especially in cases which involve continuous features. Moreover, our data set is not strictly balanced as we are making multi class classifier therefore we are expected to lose a bit of accuracy.

However, for the final testing period, we believe that we can increase this accuracy to the mid 30s by adding additive smoothing which would regularize the naive bayes classifier to a certain extent. Therein, our new likelihood probabilities will be:

$$P(X_i|Y = y_k) = \frac{T_{i,Y=y_k} + \alpha}{\Sigma_{i=1}^{W} T_{i,Y=y_k} + \alpha * W}$$

However, we will only apply this additive smoothing if we discretize the values of the features. Otherwise, we might apply Bessel's correction to the gaussian distribution or try different distributions such as gamma distribution. But it is important to note that we will try all these methods and present the one which gives us the highest accuracy.

3. Logistic Regression

Logistic regression is based on the sigmoid function below:

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$P(Y|X) = \frac{1}{1 + exp(w_o + \sum_{i=1}^{n} w_i Xi)}$$

As stated in the previous report, we decided to use one-vs-all logistic regression. We defined our loss function as the one below:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} [-y^{(i)} log(h_\theta(x^{(i)})) - (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))]$$

Where h_θ was the product of our sigmoid function for the dot product of the weights with the feature vector. We used gradient descent to optimize our weights. The formula for it is given below (b(n+1) is our bias term in the weights):

$$w(n+1) \leftarrow w(n) - \frac{\alpha}{r} \cdot \frac{\partial \mathrm{y}}{\partial \mathrm{w}}$$

$$b(n+1) \leftarrow b(n) - \frac{\alpha}{r} \cdot \frac{\partial \mathrm{y}}{\partial \mathrm{b}}$$

Where α is our learning rate, which we will fine tune by trial and error, and b is what ever the bias is, found by gradient descent.

In order to actually implement this formula in the code, we had to apply chain rule to get the jacobian matrix in the correct format:

$$dL/dw = dL/dh * dh/dz * dz/dw$$

$$dz/dw = X$$

$$dh/dz = h(1-h)$$

$$dL/dh = y * (1/h) - (1-y)/(1-h)$$

Therefore, our update rule for the gradient descent algorithm becomes:

$$dL/dw = X(h-y); h = \frac{1}{1 + exp(-b - X^T W)}$$

Before the training could commence, the HOGValuesTrain and test cell datasets were transformed into matrices of size 48905x1600 and 16421x1600 to make the computations easier. Moreover, a bias column of 1s was added to the sets.

Training Phase

The data given to the logisticRegressionWeights2 function which would compute the weights. As we were low on time and the weight calculation for the desired 100000 iterations would take at least 12 hours, we tested the algorithm for only 5 epochs with a greater learning of 0.1 to compensate. Ideally we would like to give the algorithm enough iterations with a smaller learning rate to learn.

Testing Phase

For the testing phase we used the logisticRegressionClassifyforN function which takes the test feature set and the weights as the input. For one-vs-all we had to run this algorithm for each feature vector, a total of 95 times with a different weight each time. This algorithm input the dot product of the weights and the feature vector into the sigmoid function. Then the weight corresponding to maximum of the outputs of the

sigmoid function was chosen. And then the feature vector was predicted to belong to the class of the weight with the highest output of the sigmoid function as the sigmoid function represents the probability $P(Y = 1|X)$.

Results

The accuracy of this algorithm was around 1% which is extremely low, however we did not give the algorithm enough time to train as we did not anticipate that the logistic regression algorithm would take so long to train. Right now we were training for 5 iterations, with an elapsed time of around 30 minutes, but we expect that with 100000 iterations we would get a very high accuracy. Moreover, will also then be willing to implement the mini-batch gradient function instead of the gradient function but as we are getting our highest accuracy for the gradient descent function, we decided to include that in our report.

Before, the next report we will spend more effort on the pre-processing of the dataset with PCA algorithm too to salvage more time and make eliminate the instances in the dataset with very low variances. Another, problem we encountered was the precision of the calculated probabilities. In order to eliminate false 0 probabilities, we will increase the precision of the probabilities to around 20 digits, which is likely to be computationally exhaustive.

APPENDIX

1. Naïve Bayes

```
%% naive bayes classifier

%calculating mean and std dev of features
%total no. of features for each image: 400x4
%features are x1, x2, x3, x4

piRatios = zeros(95, 1);
featIsolated = {};
mu = {};
stdDev = {};
featCentered = [];

for k = 1:95
   piRatios(k) = sum(yTrain == k);
end

clear i j k;
tempMean = 0;

for fruit = 1:95

   v = find(yTrain == fruit);

   for u = 1:length(v)
     featIsolated{u, 1} = HOGValuesTRAIN{v(u), 1};
   end

   clear v;

   for i = 1:20
      for j = 1:20
         for k = 1:4
            for u = 1: length(featIsolated)
               tempMean = tempMean +...
                  featIsolated{u, 1}(i, j, k);
               featCentered(u) = featIsolated{u, 1}(i, j, k);
            end
            mu{fruit, 1}(i, j, k) = tempMean/length(featIsolated);
            featCentered = (abs(featCentered - mu{fruit, 1}(i, j, k))).^2;
            stdDev{fruit, 1}(i, j, k) = sqrt((sum(featCentered)/length(featIsolated)));
%              disp(stdDev{fruit, 1}(i, j, k))
            tempMean = 0;
            featCentered = [];

         end
      end
   end

end
```

```
clear u;
disp("learning means and standard deviations are done")

%% Testing phase

disp("Testing phase")
y_mle = zeros(95, 1);
y_hat = zeros(95, 1);
probs = 0;
theta_fun = @(x, u, sigma) 1/(sqrt(2*pi)*sigma)*exp(-((x-u)^2)*(1/(2*sigma^2)));
theta = [];
clear fruit;
count = 1;
for u = 1: length(HOGValuesTEST)
   for fruit = 1:95
      for i = 1:20
         for j = 1:20
            for k = 1:4
               sigma =  stdDev{fruit, 1}(i, j, k);
               muu = mu{fruit, 1}(i, j, k);
               probs = probs + log(double(1/(sqrt(2*pi)*sigma)*exp(-((HOGValuesTEST{u, 1}(i, j, k)-
muu).^2)./(2*sigma.^2))));
            end
         end
      end
      y_mle(fruit) = log(piRatios(fruit)) + (probs);
%        disp(probs)
      probs = 0;
   end
   [D I] = max(y_mle);
   y_hat(u) = I;
%    disp(y_hat(u))
   y_mle = zeros(95, 1);
end

disp("accuracy is: ")
accuracy = sum(yTest == y_hat)/length(yTest)

%% confusion matrix plot
plotconfusion(yTest,y_hat)

%logistic regression script
weight = zeros(1601, 95);
for N = 1:95
   disp(N)
   y = zeros(size(yTrain));
   k = find(yTrain == N);
   y(k) = 1;


   XTrain = trainF(:, 2:end);
```

```
    XTest = testF(:, 2:end);

    [n, p] = size(XTrain);
    w0 = rand(p + 1, 1);
    weight(:, N) = logisticRegressionWeights2( XTrain, yTrain, w0, 20, 0.1);

end

%%
y_hat = ClassifyforN( XTest, weight/100000000 );
disp("done")

%%
err = sum(yTest ~= y_hat');
accuracy_log = 1 - err / size(XTest, 1) ;
```

2. Logistic Regression

```
function [w] = logisticRegressionWeights2( XTrain, yTrain, w_o, epoch, lr)

    [n, m] = size(XTrain);
    w = w_o;
    previous = 0;
    for j = 1:epoch
        disp("iteration number: " + j)
        temp = zeros(m + 1,1);
        for k = 1:n
            temp = temp + (sigmoid([1.0 XTrain(k,:)] * w) - yTrain(k)) * [1.0 XTrain(k,:)]';
        end
        w = w - lr * temp;
        cost = CostFunc(XTrain, yTrain, w);
        if j~=0 && abs(cost - precost) / cost <= 0.0001
            break;
        end
        precost = cost;
    end
% w = w/1601;
end

function [ res ] = ClassifyforN( XTest, w )

[m n] = size(w);
y_mle = [];
y_hat = [];

    %this w should be of size(1601, N)
nTest = size(XTest,1);
res = zeros(nTest,1);

for i = 1:nTest
    disp(i)
```

```
    for j = 1:n
    w0 = w(:, j);
%   y_mle = vpa([ y_mle sigmoid([1.0 XTest(i,:)] * w0) ], 20);
    y_mle = ([ y_mle sigmoid([1.0 XTest(i,:)] * w0) ]);
    end
    [D I] = max(y_mle);
    disp([D I]);
    y_hat(i) = I;
    y_mle=[];

end
res = y_hat;
end

function g = sigmoid(z)

% z = dot(x, B);

h = 1*exp(-z)/(1 + exp(-z));


g = h;
end
```

3. Neural Network

```
function cost=trainNN(numClasses, learningRate, beta1, beta2, imgDepth, filterSize, numFilter1,
numFilter2, batchSize, numEpoch)
tic
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Data preprocessing and normalisation
%
load Dataset.mat;

X=HOGValuesTRAIN;

% Normalize and standardize over N

X=permute(X,[4,3,2,1]);
X2=reshape(X,size(X,1),[]);
X=(X2' - floor(mean(X2,'omitnan'))')';
% X=(X / floor(std(X,1,1,'omitnan')));
B=floor(std(X,1,1,'omitnan'));

% Start of standard deviation normalisation
ispos = B>0;
C = X;
```

```
C(:,ispos) = X(:,ispos) ./ B(ispos);
X=C;
clear C B ispos;

X=reshape(X,size(X,1),imgDepth,20,20);

y=yTrain;
y=ones(length(y),1,20,20).*y;
dataTrain=horzcat(X,y);

%%%%%%%%%%%%%%%%
dataTrain=dataTrain(randperm(size(dataTrain,1)),:,:,:);

f1=[numFilter1,imgDepth,filterSize,filterSize];
f2=[numFilter2,numFilter2,filterSize,filterSize];

w3=[200,288];
w4=[95,200];

f1=initFilter(f1);
f2=initFilter(f2);
w3=initWeights(w3);
w4=initWeights(w4);

b1=zeros(size(f1,1),1);
b2=zeros(size(f2,1),1);
b3=zeros(size(w3,1),1);
b4=zeros(size(w4,1),1);

% params={f1,f2,w3,w4,b1,b2,b3,b4};

load param.mat


% cost=0;

learningRate
batchSize
cnt=1;

for i=1:numEpoch
   dataTrain=dataTrain(randperm(size(dataTrain,1)),:,:,:);
   for j=1:batchSize:size(dataTrain,1)-batchSize
      batches{cnt}=dataTrain([j:j + batchSize-1],:,:,:);
      cnt=cnt+1;
   end
```

```
    length(batches)
    toc
    for j=1:length(batches)
    [params,cost]=adamGD(batches{j},numClasses,learningRate,beta1,beta2,params,cost);
    disp(cost(end));
    end
end
save('param.mat','params','cost');

toc
end

function [params1,rCost]=adamGD(batch, numClasses, learningRate, beta1, beta2, params,
rCost)
%
% Update parameters using Adams gradient descent
%

filter1 = params{1};
filter2 = params{2};
weight3 = params{3};
weight4 = params{4};
bias1 = params{5};
bias2 = params{6};
bias3 = params{7};
bias4 = params{8};


A=batch(:,[1:end-1],:,:);
Y=batch(:,end,1,1);

cost=0;

batchSize=size(A,1);

df1 = zeros(size(filter1));
df2 = zeros(size(filter2));
dw3 = zeros(size(weight3));
dw4 = zeros(size(weight4));
db1 = zeros(size(bias1));
db2 = zeros(size(bias2));
db3 = zeros(size(bias3));
db4 = zeros(size(bias4));

v1 = zeros(size(filter1));
v2 = zeros(size(filter2));
```

```
v3 = zeros(size(weight3));
v4 = zeros(size(weight4));
bv1 = zeros(size(bias1));
bv2 = zeros(size(bias2));
bv3 = zeros(size(bias3));
bv4 = zeros(size(bias4));

s1 = zeros(size(filter1));
s2 = zeros(size(filter2));
s3 = zeros(size(weight3));
s4 = zeros(size(weight4));
bs1 = zeros(size(bias1));
bs2 = zeros(size(bias2));
bs3 = zeros(size(bias3));
bs4 = zeros(size(bias4));

for i=1:batchSize
    x=A(i,:,:,:);
    x=reshape(x,[4,20,20]);
    y=eye(numClasses);
    y=y(Y(i),:);
    y=reshape(y,[numClasses,1]);

    [grads,loss]=ConvolutionMain(x,y,params,1,2,2);
    dF1=grads{1};
    dF2=grads{2};
    dW3=grads{3};
    dW4=grads{4};
    dB1=grads{5};
    dB2=grads{6};
    dB3=grads{7};
    dB4=grads{8};

    df1=df1+dF1;
    db1=db1+dB1;
    df2=df2+dF2;
    db2=db2+dB2;
    dw3=dw3+dW3;
    db3=db3+dB3;
    dw4=dw4+dW4;
    db4=db4+dB4;

    cost=cost+loss;

end
```

```
v1=(beta1.*v1) + (1-beta1).*df1./batchSize;
s1=(beta2.*s1) + (1-beta2).*((df1./batchSize).^2);
filter1=filter1-(learningRate*(v1./sqrt(s1+(1*(10^(-7))))));

bv1=(beta1.*bv1) + (1-beta1).*db1./batchSize;
bs1=(beta2.*bs1) + (1-beta2).*((db1./batchSize).^2);
bias1=bias1-(learningRate.*(bv1./sqrt(bs1+(1*(10^(-7))))));

v2=(beta1.*v2) + (1-beta1).*df2./batchSize;
s2=(beta2.*s2) + (1-beta2).*((df2./batchSize).^2);
filter2=filter2-(learningRate.*(v2./sqrt(s2+(1*(10^(-7))))));

bv2=(beta1.*bv2) + (1-beta1).*db2/batchSize;
bs2=(beta2.*bs2) + (1-beta2).*((db2/batchSize).^2);
bias2=bias2-(learningRate.*(bv2./sqrt(bs2+(1*(10^(-7))))));

v3=(beta1.*v3) + (1-beta1).*dw3/batchSize;
s3=(beta2.*s3) + (1-beta2).*((dw3/batchSize).^2);
weight3=weight3-(learningRate.*(v3./sqrt(s3+(1*(10^(-7))))));

bv3=(beta1.*bv3) + (1-beta1).*db3/batchSize;
bs3=(beta2.*bs3) + (1-beta2).*((db3./batchSize).^2);
bias3=bias3-(learningRate.*(bv3./sqrt(bs3+(1*(10^(-7))))));

v4=(beta1.*v4) + (1-beta1).*dw4/batchSize;
s4=(beta2.*s4) + (1-beta2).*((dw4./batchSize).^2);
weight4=weight4-(learningRate.*(v4./sqrt(s4+(1*(10^(-7))))));

bv4=(beta1.*bv4) + (1-beta1).*db4./batchSize;
bs4=(beta2.*bs4) + (1-beta2).*((db4./batchSize).^2);
bias4=bias4-(learningRate*(bv4./sqrt(bs4+(1*(10^(-7))))));

cost= cost/batchSize;
rCost=[rCost cost];

params1={filter1, filter2, weight3, weight4, bias1, bias2, bias3, bias4};

end


function [gradients, loss]=ConvolutionMain(image, label, params, sConv, fPool, sPool)

filter1 = params{1};
filter2 = params{2};
weight3 = params{3};
weight4 = params{4};
```

```
bias1 = params{5};
bias2 = params{6};
bias3 = params{7};
bias4 = params{8};


%
% Forward operation
%

conv1=conv(image,filter1,bias1,sConv);
conv1(conv1<=0)=0; % ReLU filter

conv2=conv(conv1,filter2,bias2,sConv);
conv2(conv2<=0)=0;  % ReLU filter

pooled=maxpool(conv2,fPool,sPool);

[numberFilters2,yDimPool,xDimPool]=size(pooled);
newPooled=reshape(pooled,[xDimPool*yDimPool*numberFilters2,1]); %Flattened vector

l=weight3*newPooled + bias3; % dot product
l(l<=0)=0; % ReLU filter

output=weight4*l+bias4; % dot product


classProb=softMax(output);

%
% Loss
%

loss=crossEntopy(classProb,label);

%
% Backward Operation
%

delOut= classProb - label;

delWeight4=delOut*l';

delBias4=reshape(sum(delOut,2), size(bias4));
```

```
del_l=weight4'*delOut;

del_l(del_l<=0) = 0; %BackPropagate through ReLU

delWeight3=del_l*newPooled';

delBias3=reshape(sum(del_l,2),size(bias3));


delNewPooled=weight3'*del_l;

delPool=reshape(delNewPooled,size(pooled));


delConv2=backwardMaxpool(delPool,conv2,fPool,sPool);

delConv2(delConv2<=0)=0;

[delConv1,delFilter2,delBias2]= backwardConvolution(delConv2,conv1,filter2,sConv);

delConv1(delConv1<=0)=0;

[delImage, delFilter1, delBias1]=backwardConvolution(delConv1, image, filter1, sConv);

gradients={delFilter1, delFilter2, delWeight3, delWeight4, delBias1, delBias2, delBias3,
delBias4};

end


function output=conv(img,filterLayer,bias,stride)
%
% conv(image,filterLayer,bias,stride)
%
% FilterLayer comes from initializeFilter. It should be a 4 dimensional
% matrix where 1st dim is the number of filters, 2nd dim is the number of z
% layers of the images or dataset and 3rd and 4th dim is the x or y dim.
%


%
%
%
%
s=stride;
[nFilters,nChannelsF,yDimF,xDimF]=size(filterLayer);
```

```
% We find the dimensions of the dataset
[nChannels, x_img, y_img]=size(img);

% We find the output dimensions after the convolution
outDim=floor((y_img - yDimF)/s) + 1;

% We carry out a basic check for dimension mismatch
if nChannels~=nChannelsF
   warning('Dimensions should match');
else
   % we initialize our output
   output=zeros(nFilters,outDim,outDim);
   % We go through all the filters
   for currFilter=1:nFilters
      output_y=1;
      current_y=1;
      % We convolve the image with the filter vertically
      while current_y+yDimF <= y_img+s
         output_x=1;
         current_x=1;
         % We convolve the image with the filter horizontally
         while current_x+yDimF <= x_img+s
            %Main convolution operation
            output(currFilter,output_y,output_x)=sum(sum(sum(reshape(filterLayer(currFil-
ter,:,:,:),nChannelsF,yDimF,xDimF).*img(:,[current_y:current_y+yDimF-1],[current_x:cur-
rent_x+yDimF-1]),'omitnan'),'omitnan'),'omitnan') + bias(currFilter);
            current_x=current_x + s;
            output_x=output_x+1;
         end
      current_y=current_y+s;
      output_y=output_y+1;
      end
   end
end

end
```

```
function out=maxpool(image,f,s)

% f=2; % filter dimension size
% s=2; % step/stride size
```

```
[nChannels,yImg,xImg]=size(image);

outputHeight = floor((yImg - f)/s) +1;
outputWidth  = floor((xImg - f)/s) +1;

% initialize matrix for output
% downSampled = zeros(nChannels,outputHeight,outputWidth);

%For each z dimension

for i=1:nChannels
   yOutput=1;
   yCurrent=1;

   while (yCurrent +f <= yImg+s)
      % Sweep through Height
      xOutput=1;
      xCurrent=1;
      while (xCurrent+f<=xImg+s)
         % Sweep through width
         downSampled(i,yOutput,xOutput)=max(max(max(image(i,[yCurrent:yCurrent+f-
1],[xCurrent:xCurrent+f-1]),[],'omitnan'),[],'omitnan'),[],'omitnan');

         % Update xCurrent
         xCurrent=xCurrent+s;
         xOutput=xOutput+1;
      end
      % Update yCurrent
      yCurrent=yCurrent +s;
      yOutput=yOutput+1;
   end
end
out=downSampled;
end


function delOutput=backwardMaxpool(delPool,orig,filterSize,s)
%
% Maxpool backward convolution
%

[nChannels,origDim,~]=size(orig);

delOutput=zeros(size(orig));

for zCurrent=1:nChannels
```

```
    yOutput=1;
    yCurrent=1;

    while yCurrent+filterSize<=origDim+1
        xOutput=1;
        xCurrent=1;

        while xCurrent+filterSize<=origDim+1
            A=orig(zCurrent,[yCurrent:yCurrent+filterSize-1],[xCurrent:xCurrent+filterSize-1]);
            %%%%%%%%%%%%%%%%
            %
            % This is the problematic code
            %
            A=reshape(A,2,2);
            [a,b]=find(A==max(max(max(A,[],'omitnan'),[],'omitnan'),[],'omitnan'),1);
            %%%%%%%%%%%%%%%%
            delOutput(zCurrent,yCurrent+a-1,xCurrent+b-1)=delPool(zCurrent,yOutput,xOutput);

            xCurrent=xCurrent+s;
            xOutput=xOutput+1;
        end
        yCurrent=yCurrent+s;
        yOutput=yOutput+1;
    end
end
end


function [delOutput, delFilter, delBias]=backwardConvolution(delPrevConv, inputConv, filter,
s)
%
% Backward convolution
%

[numberFilters,nChannelsF,filtSize,filtSize]=size(filter); % after the filter is initialized
[nChannelsConv,yInputConv,xInputConv]=size(inputConv);

delOutput=zeros(size(inputConv));
delFilter=zeros(size(filter));
delBias=zeros(numberFilters,1);

for currentFilt=1:numberFilters
    %Through all filters
    yOutput=1;
    yCurrent=1;
```

```
    while yCurrent+filtSize<=yInputConv+s
        xOutput=1;
        xCurrent=1;
        while xCurrent+filtSize<=yInputConv+s %y and x are equal for input conv
            %gradient used to update the filter
            delFilter(currentFilt,:,:,:)=reshape(delFilter(currentFilt,:,:,:),nChan-
nelsF,filtSize,filtSize)+(delPrevConv(currentFilt,yOutput,xOutput)*inputConv(:,[yCurrent:yCur-
rent+filtSize-1],[xCurrent:xCurrent+filtSize-1]));
            %loss gradient of input to conv layer
            delOutput(:,[yCurrent:yCurrent+filtSize-1],[xCurrent:xCurrent+filtSize-1]) =delOut-
put(:,[yCurrent:yCurrent+filtSize-1],[xCurrent:xCurrent+filtSize-1]) + (delPrevConv(current-
Filt,yOutput,xOutput)*reshape(delFilter(currentFilt,:,:,:),nChannelsF,filtSize,filtSize));
            xCurrent=xCurrent + s;
            xOutput=xOutput+1;
        end
        yCurrent=yCurrent + s;
        yOutput=yOutput+1;
    end
    delBias(currentFilt)=sum(sum(sum(delPrevConv(currentFilt,:,:),'omitnan'),'omitnan'),'omit-
nan');
end
end


function y=crossEntopy(calcProb, label)
%
% We take the expected probablity values we estimated (calcProb) and the labels as the input
% We then return the loss.
%

output=sum(label.*log(calcProb));
y=-output;

end

clear;
clc;

load param.mat

filter1 = params{1};
filter2 = params{2};
weight3 = params{3};
weight4 = params{4};
bias1 = params{5};
bias2 = params{6};
```

```
bias3 = params{7};
bias4 = params{8};

load Dataset.mat;

X=HOGValuesTEST;

% Normalize and standardize over N

X=permute(X,[4,3,2,1]);
X2=reshape(X,size(X,1),[]);
X=(X2' - floor(mean(X2,'omitnan'))')';
% X=(X / floor(std(X,1,1,'omitnan')));
B=floor(std(X,1,1,'omitnan'));

% Start of standard deviation normalisation
ispos = B>0;
C = X;
C(:,ispos) = X(:,ispos) ./ B(ispos);
X=C;
clear C B;

X=reshape(X,size(X,1),4,20,20);

y=yTest;
% y=ones(length(y),1,20,20).*y;
% X=horzcat(X,y);

%%%%%%%%%%%%%%%%

correct=0;

cnt=zeros(95,1);
corrCnt=zeros(95,1);

for i=1:size(X,1)
   x=X(1,:,:,:);
   x=reshape(x,4,20,[]);
   [pred,prob]=predict(x,filter1,filter2,weight3,weight4,bias1,bias2,bias3,bias4);
   cnt(y(i))=cnt(y(i))+1;
   if pred==y(i)
      correct=correct+1;
      corrCnt(pred)=corrCnt(pred)+1;
   end
   correct/(i)*100;
end
```

```
fprintf('Total Accuracy = %.2f', (correct/size(X,1)*100));


function [pred, prob]=predict(data, filter1,filter2,weight3,weight4,bias1,bias2,bias3,bias4);

sConv=1;
fPool=2;
sPool=2;


conv1=conv(data,filter1,bias1,sConv);
conv1(conv1<=0)=0; % ReLU filter

conv2=conv(conv1,filter2,bias2,sConv);
conv2(conv2<=0)=0;  % ReLU filter

pooled=maxpool(conv2,fPool,sPool);

[numberFilters2,yDimPool,xDimPool]=size(pooled);
newPooled=reshape(pooled,[xDimPool*yDimPool*numberFilters2,1]); %Flattened vector

l=weight3*newPooled + bias3; % dot product
l(l<=0)=0; % ReLU filter

output=weight4*l+bias4; % dot product


classProb=softMax(output);

prob=max(classProb,[],'omitnan');
pred=find(classProb==prob,1);

end

function y=initFilter(filterParam)
%
% filter has dims : x-y-z-nF
% x=y dims and nF is the no of Filters
%

scale=1;

standDev=scale/sqrt(prod(prod(prod(filterParam,'omitnan'),'omitnan'),'omitnan'));

y=normrnd(0,standDev,[filterParam]);
```

```
end
```

```
function y=initWeights(weightSize)
%
% weightsize matrix be a vector of two numbers (p,q) where p is the number
% of neurons in the output layer and q is the number of neurons in the
% input layer
%
y=randn(weightSize)*0.01;

end

function output=softMax(predictions)
%
% Codes for the probability class since this is a classification problem
%

num=exp(predictions); % numerator value
output=num/(sum(num)); % denominator whic is sum of numerators equalling 1

end
```

Gantt Chart

ID	Task Name	Duration	Start	Finish	Pred
1	**Phase 1**	**21 days**	**20 Feb '19**	**20 Mar '19**	
2	Literature Research	21 days	20 Feb '19	20 Mar '19	
3	**Phase 2**	**25 days**	**21 Mar '19**	**24 Apr '19**	**1**
4	Research	7 days	21 Mar '19	29 Mar '19	
5	Dataset Preprocessing	3 days	26 Mar '19	28 Mar '19	4SS-day:
6	Logistic Regression	6 days	29 Mar '19	05 Apr '19	5
7	Naïve Bayes Classifier	5 days	06 Apr '19	11 Apr '19	6
8	Neural Network	8 days	12 Apr '19	23 Apr '19	7
9	**Phase 3**	**15 days**	**25 Apr '19**	**15 May '19**	**3**
10	Feedback revision	4 days	25 Apr '19	30 Apr '19	3
11	Logistic Regression	5 days	01 May '19	07 May '19	10
12	Naïve Bayes Classifier	6 days	08 May '19	15 May '19	11
13	Neural Network	10 days	01 May '19	14 May '19	10