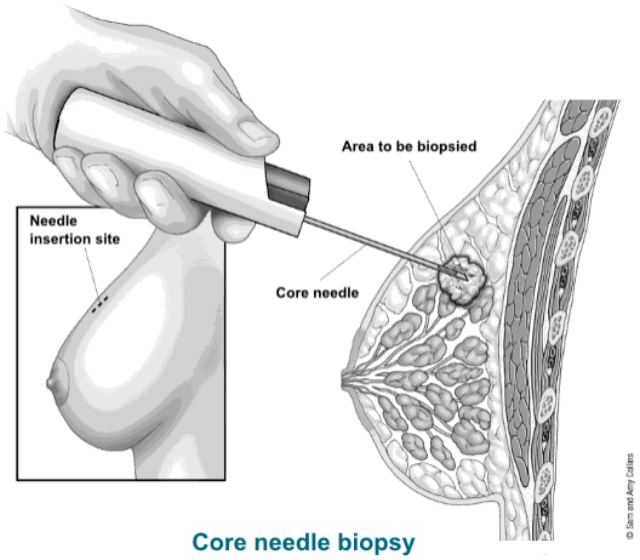
Breast Cancer Classification using MLP Algorithm
By Yuta Nguyen

Introduction

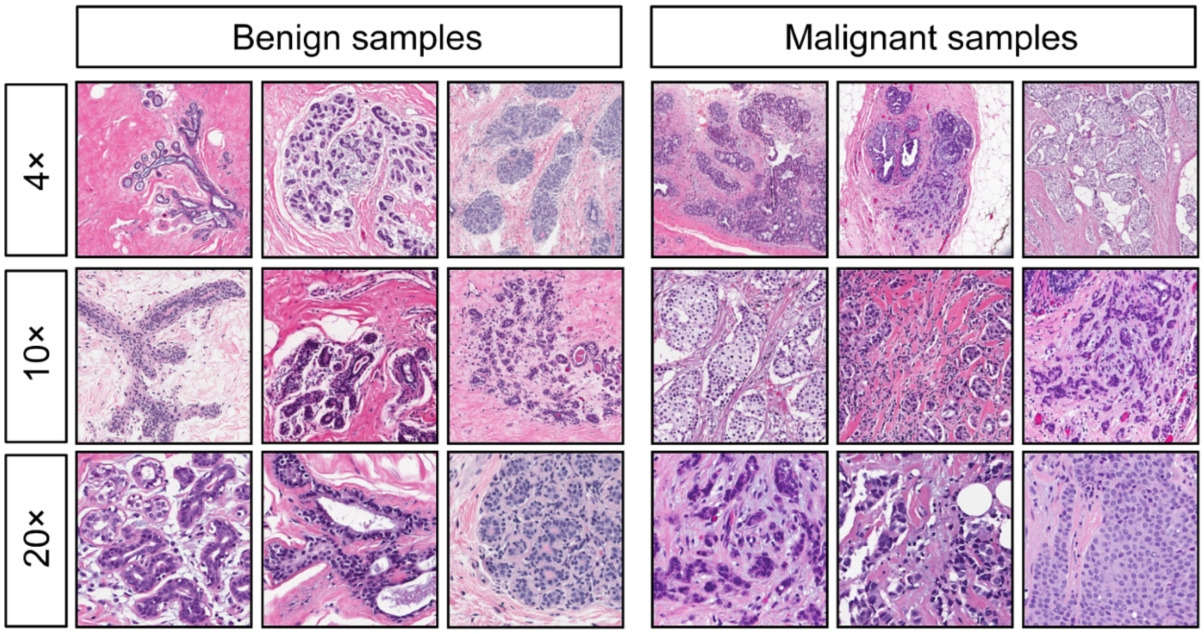
For my Machine Learning Project, I'll be writing and coding about Breast Cancer Classification using the Multilayer Perceptron Learning Algorithm with back propagation. The medical dataset I'll be using will be from UCI's page given: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

So, for the dataset Information, it features a computerized digitized image of a fine needle aspirate of a breast mass where it describes the characteristics of the cell nuclei present in the image for example below



Core needle biopsy

As for the main attributes within the file, the dataset is formatted in a csv file where it lists:
1) ID number of patient
2) Diagnosis (M = malignant(Cancerous), B = benign(Noncancerous)



3) Radius (mean of distances from center to points on the perimeter)
4) Texture (standard deviation of gray-scale values
5) Perimeter
6) Area
7) Smoothness (local variation in radius lengths
8) Compactness (perimeter^2/aera - 1.0)
9) Concavity (severity of concave portions of the contour)
10) Concave points (number of concave portions of the contour
11) Symmetry
12) Fractal dimension: which is the ratio providing a statistical index of complexity comparing how detail in a pattern changes with the scale which it's measured

Solution Method:

So, for the solution method to detect cancerous and noncancerous cells from the breast, I'll be using the Multilayer Perceptron method with back propagation. I'll be using a simple two layer

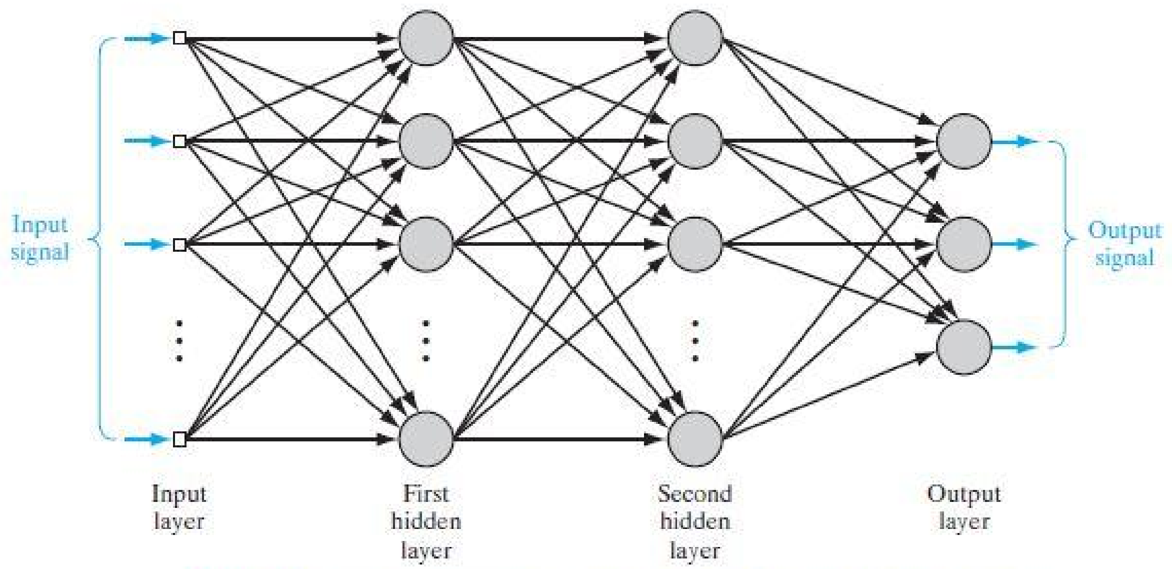
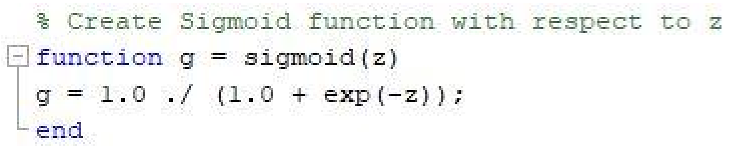


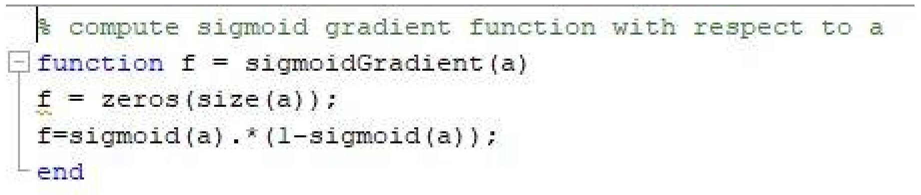
FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

Activation Function

First, I'll use the sigmoid function and sigmoid gradient function. The sigmoid function below will be our reference to our sigmoid gradient function



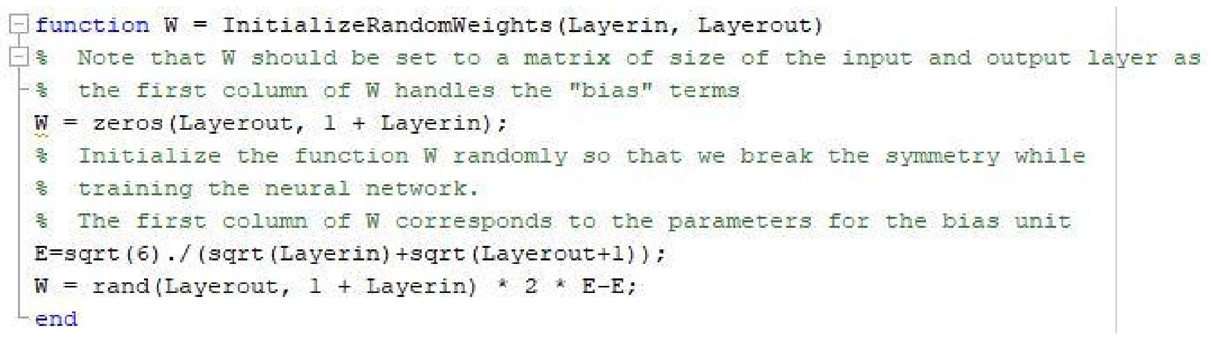
```
% Create Sigmoid function with respect to z
function g = sigmoid(z)
g = 1.0 ./ (1.0 + exp(-z));
end
```



```
% compute sigmoid gradient function with respect to a
function f = sigmoidGradient(a)
f = zeros(size(a));
f=sigmoid(a).*(1-sigmoid(a));
end
```

Weights

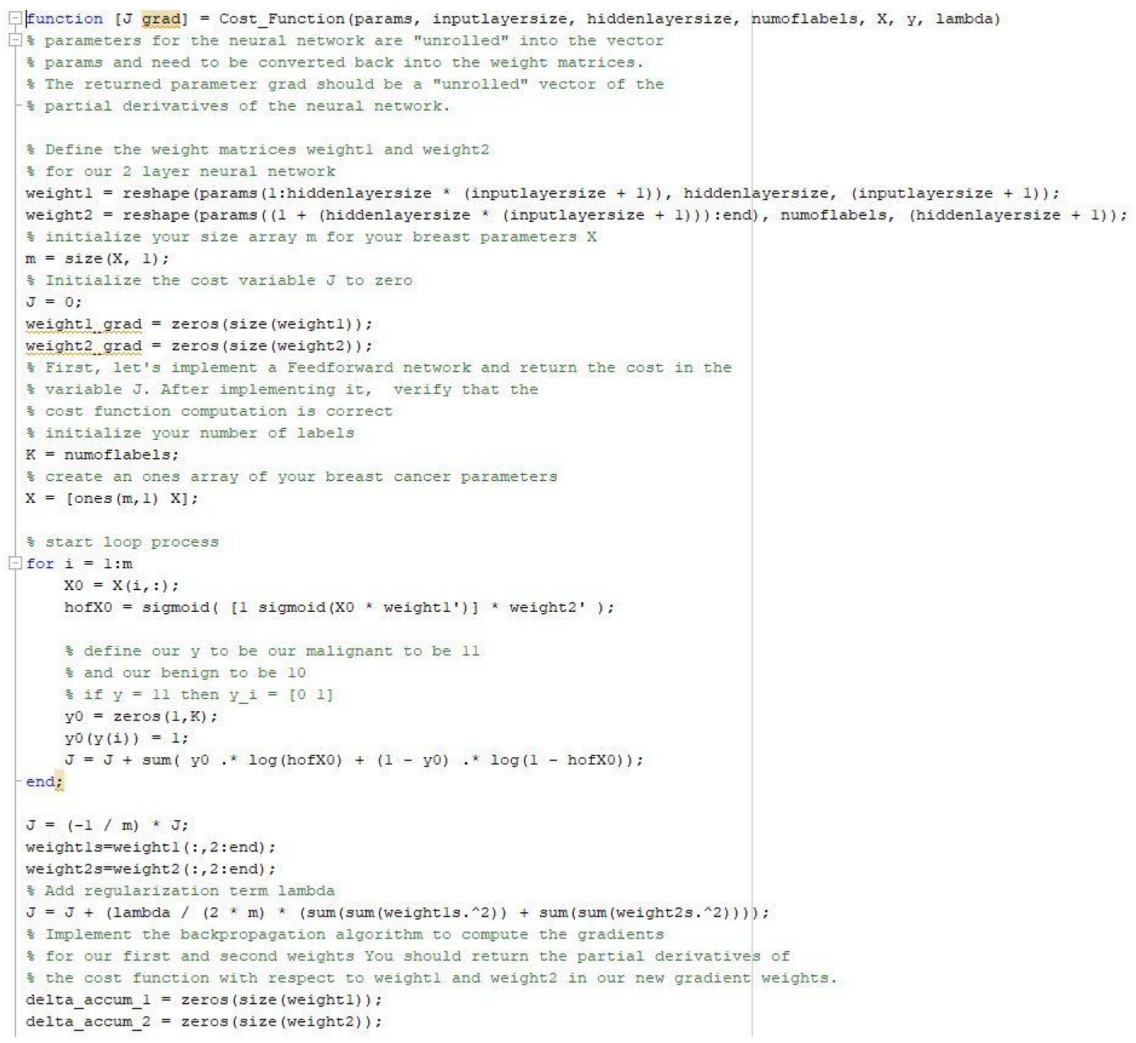
Next, I need to initialize the weights randomly into the input layer for the incoming connections and the output layer of the output connections



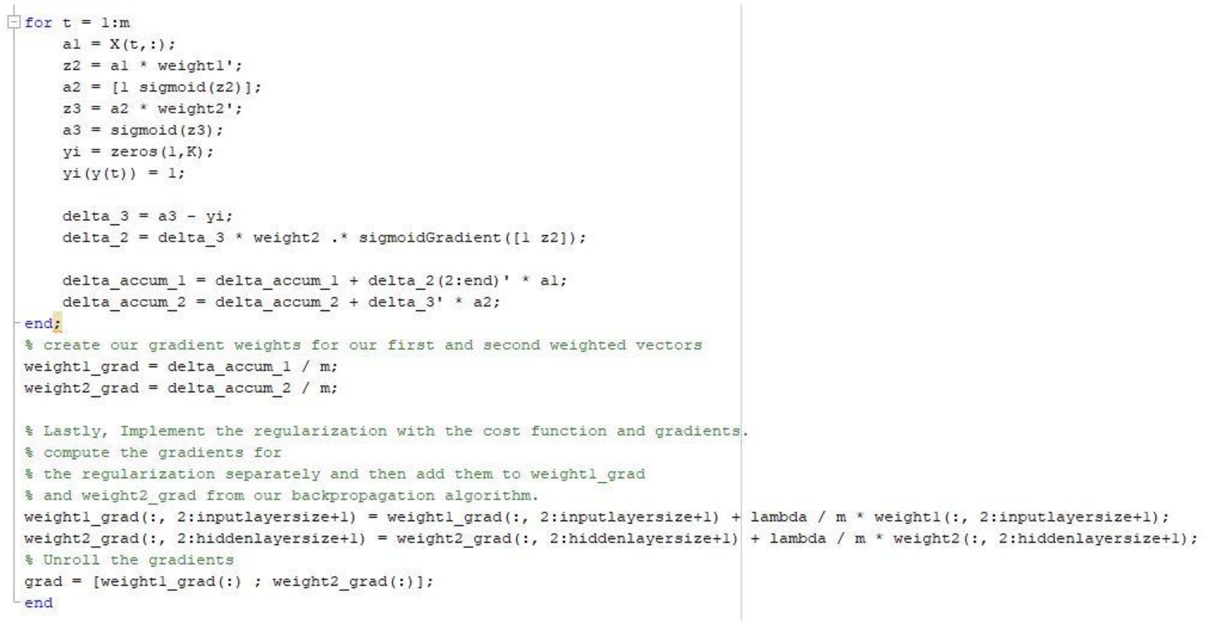
```
function W = InitializeRandomWeights(Layerin, Layerout)
%  Note that W should be set to a matrix of size of the input and output layer as
%  the first column of W handles the "bias" terms
W = zeros(Layerout, 1 + Layerin);
%  Initialize the function W randomly so that we break the symmetry while
%  training the neural network.
%  The first column of W corresponds to the parameters for the bias unit
E=sqrt(6)./(sqrt(Layerin)+sqrt(Layerout+1));
W = rand(Layerout, 1 + Layerin) * 2 * E-E;
end
```

<u>Cost Function</u>

Next, I'll implement a simple two layer cost function which will perform our classification and compute our cost and gradient of the neural network. Our important parameters we need to consider are input layer size, hidden layer size, the number of labels, and our regularization parameter **λ**. Next we need to initialize a variable **X** which will be our dataset parameters above from 3-10. Then we'll implement another variable **y** which will represent our diagnosis to determine if it's malignant or benign.



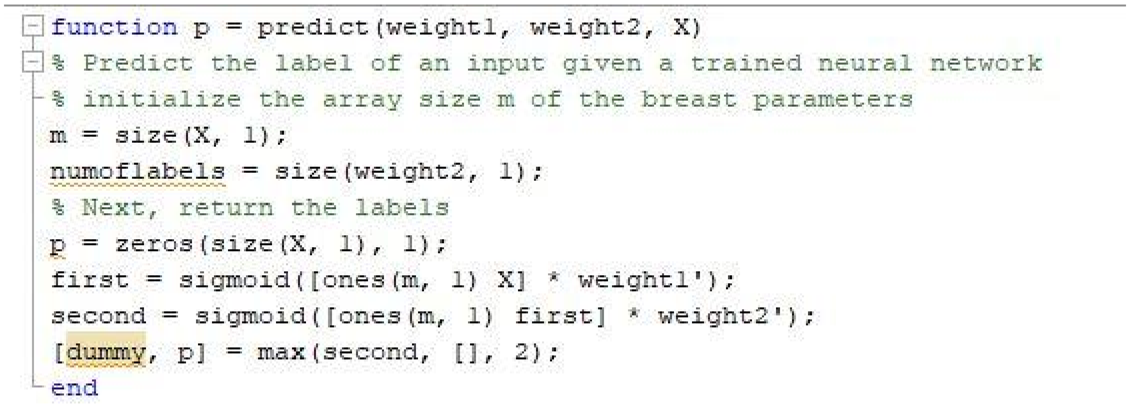
```
function [J grad] = Cost_Function(params, inputlayersize, hiddenlayersize, numoflabels, X, y, lambda)
% parameters for the neural network are "unrolled" into the vector
% params and need to be converted back into the weight matrices.
% The returned parameter grad should be a "unrolled" vector of the
% partial derivatives of the neural network.

% Define the weight matrices weight1 and weight2
% for our 2 layer neural network
weight1 = reshape(params(1:hiddenlayersize * (inputlayersize + 1)), hiddenlayersize, (inputlayersize + 1));
weight2 = reshape(params((1 + (hiddenlayersize * (inputlayersize + 1))):end), numoflabels, (hiddenlayersize + 1));
% initialize your size array m for your breast parameters X
m = size(X, 1);
% Initialize the cost variable J to zero
J = 0;
weight1_grad = zeros(size(weight1));
weight2_grad = zeros(size(weight2));
% First, let's implement a Feedforward network and return the cost in the
% variable J. After implementing it,  verify that the
% cost function computation is correct
% initialize your number of labels
K = numoflabels;
% create an ones array of your breast cancer parameters
X = [ones(m,1) X];

% start loop process
for i = 1:m
    X0 = X(i,:);
    hofX0 = sigmoid( [1 sigmoid(X0 * weight1')] * weight2' );

    % define our y to be our malignant to be 11
    % and our benign to be 10
    % if y = 11 then y_i = [0 1]
    y0 = zeros(1,K);
    y0(y(i)) = 1;
    J = J + sum( y0 .* log(hofX0) + (1 - y0) .* log(1 - hofX0));
end;

J = (-1 / m) * J;
weight1s=weight1(:,2:end);
weight2s=weight2(:,2:end);
% Add regularization term lambda
J = J + (lambda / (2 * m) * (sum(sum(weight1s.^2)) + sum(sum(weight2s.^2))));
% Implement the backpropagation algorithm to compute the gradients
% for our first and second weights You should return the partial derivatives of
% the cost function with respect to weight1 and weight2 in our new gradient weights.
delta_accum_1 = zeros(size(weight1));
delta_accum_2 = zeros(size(weight2));
```



```
for t = 1:m
    a1 = X(t,:);
    z2 = a1 * weight1';
    a2 = [1 sigmoid(z2)];
    z3 = a2 * weight2';
    a3 = sigmoid(z3);
    yi = zeros(1,K);
    yi(y(t)) = 1;

    delta_3 = a3 - yi;
    delta_2 = delta_3 * weight2 .* sigmoidGradient([1 z2]);

    delta_accum_1 = delta_accum_1 + delta_2(2:end)' * a1;
    delta_accum_2 = delta_accum_2 + delta_3' * a2;
end;
% create our gradient weights for our first and second weighted vectors
weight1_grad = delta_accum_1 / m;
weight2_grad = delta_accum_2 / m;

% Lastly, Implement the regularization with the cost function and gradients.
% compute the gradients for
% the regularization separately and then add them to weight1_grad
% and weight2_grad from our backpropagation algorithm.
weight1_grad(:, 2:inputlayersize+1) = weight1_grad(:, 2:inputlayersize+1) + lambda / m * weight1(:, 2:inputlayersize+1);
weight2_grad(:, 2:hiddenlayersize+1) = weight2_grad(:, 2:hiddenlayersize+1) + lambda / m * weight2(:, 2:hiddenlayersize+1);
% Unroll the gradients
grad = [weight1_grad(:) ; weight2_grad(:)];
end
```

<u>Predict the weights</u>

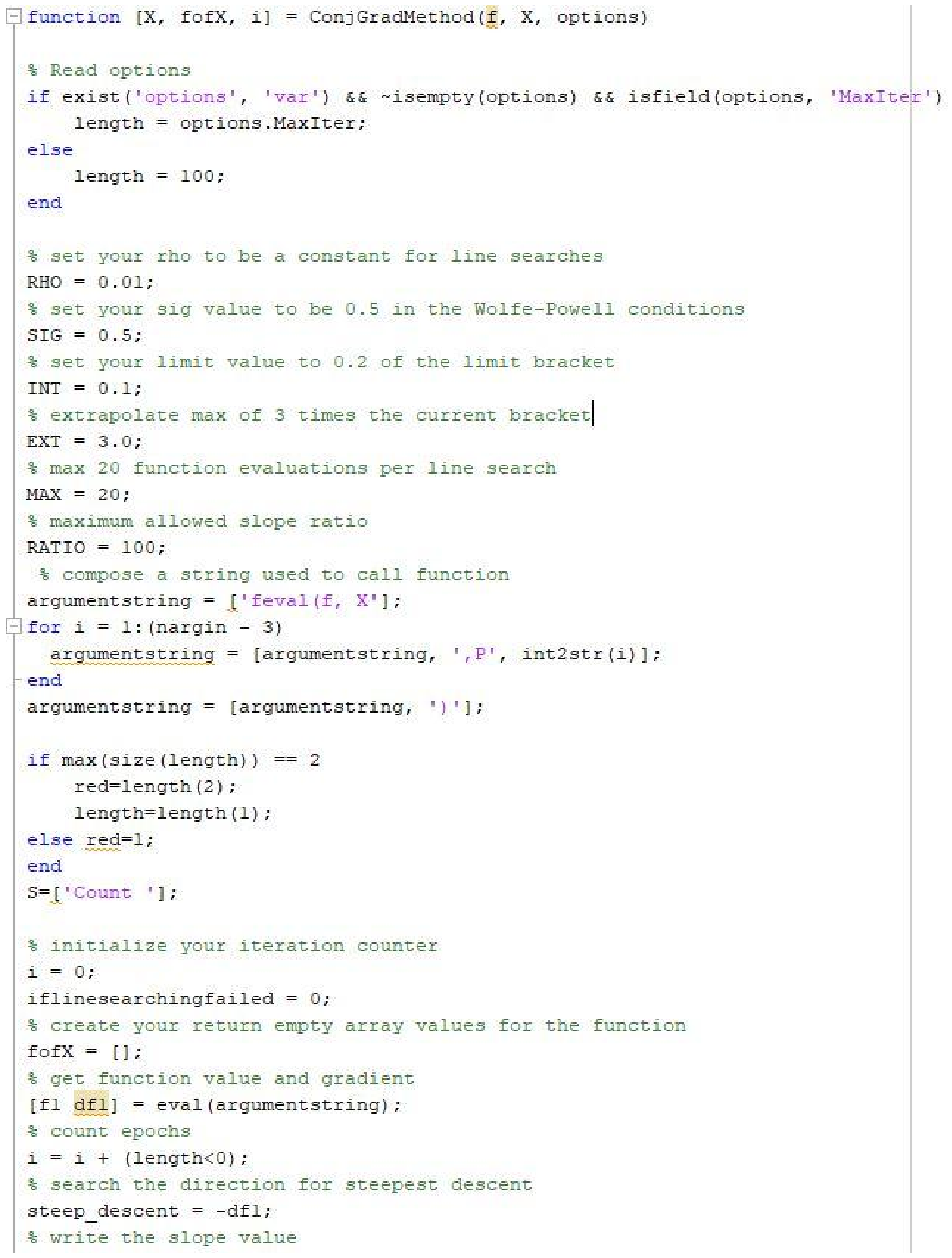
Next, we'll predict the weights which should output the trained weights of our network:



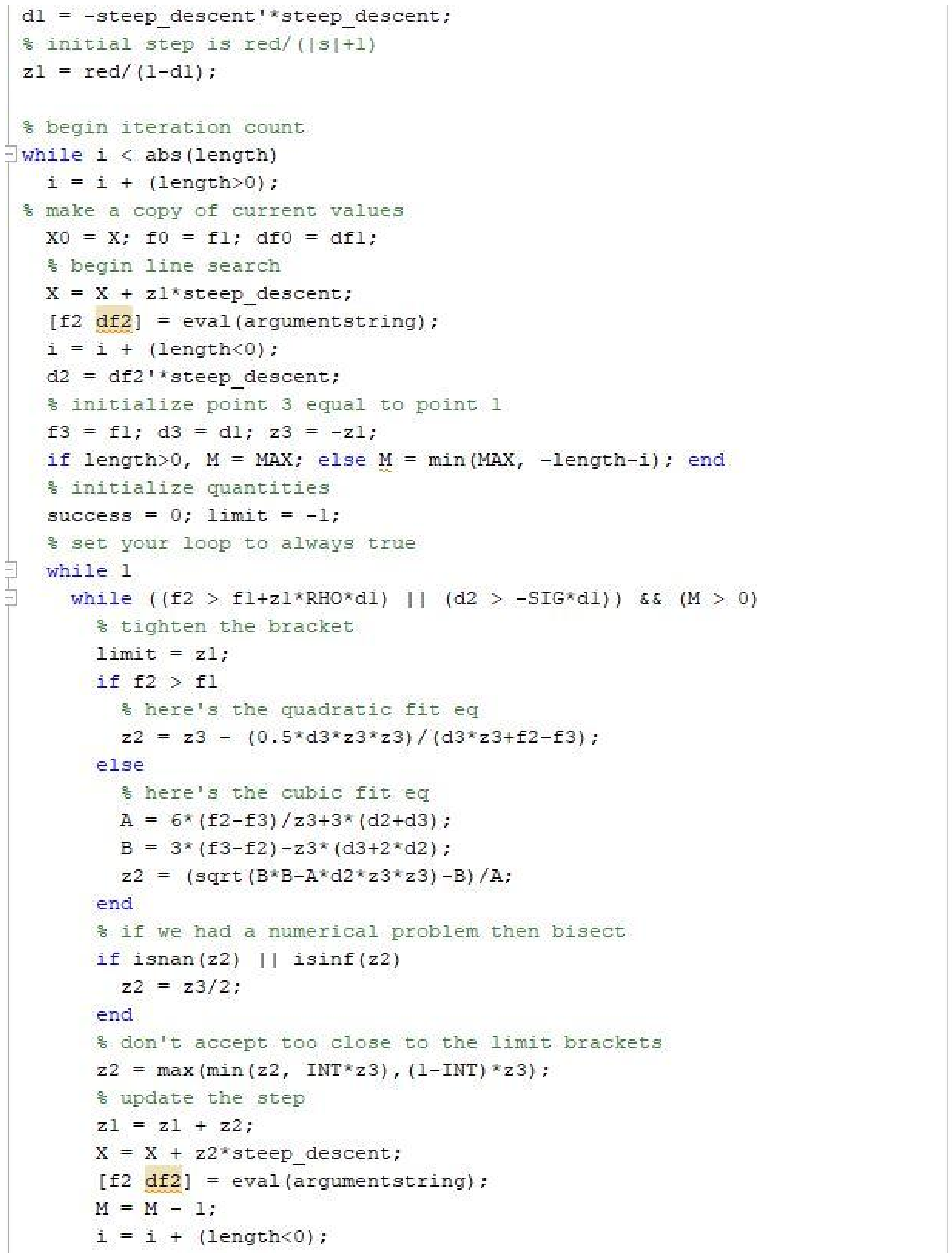
```
function p = predict(weight1, weight2, X)
% Predict the label of an input given a trained neural network
% initialize the array size m of the breast parameters
m = size(X, 1);
numoflabels = size(weight2, 1);
% Next, return the labels
p = zeros(size(X, 1), 1);
first = sigmoid([ones(m, 1) X] * weight1');
second = sigmoid([ones(m, 1) first] * weight2');
[dummy, p] = max(second, [], 2);
end
```

<u>Conjugate Gradient Method</u>

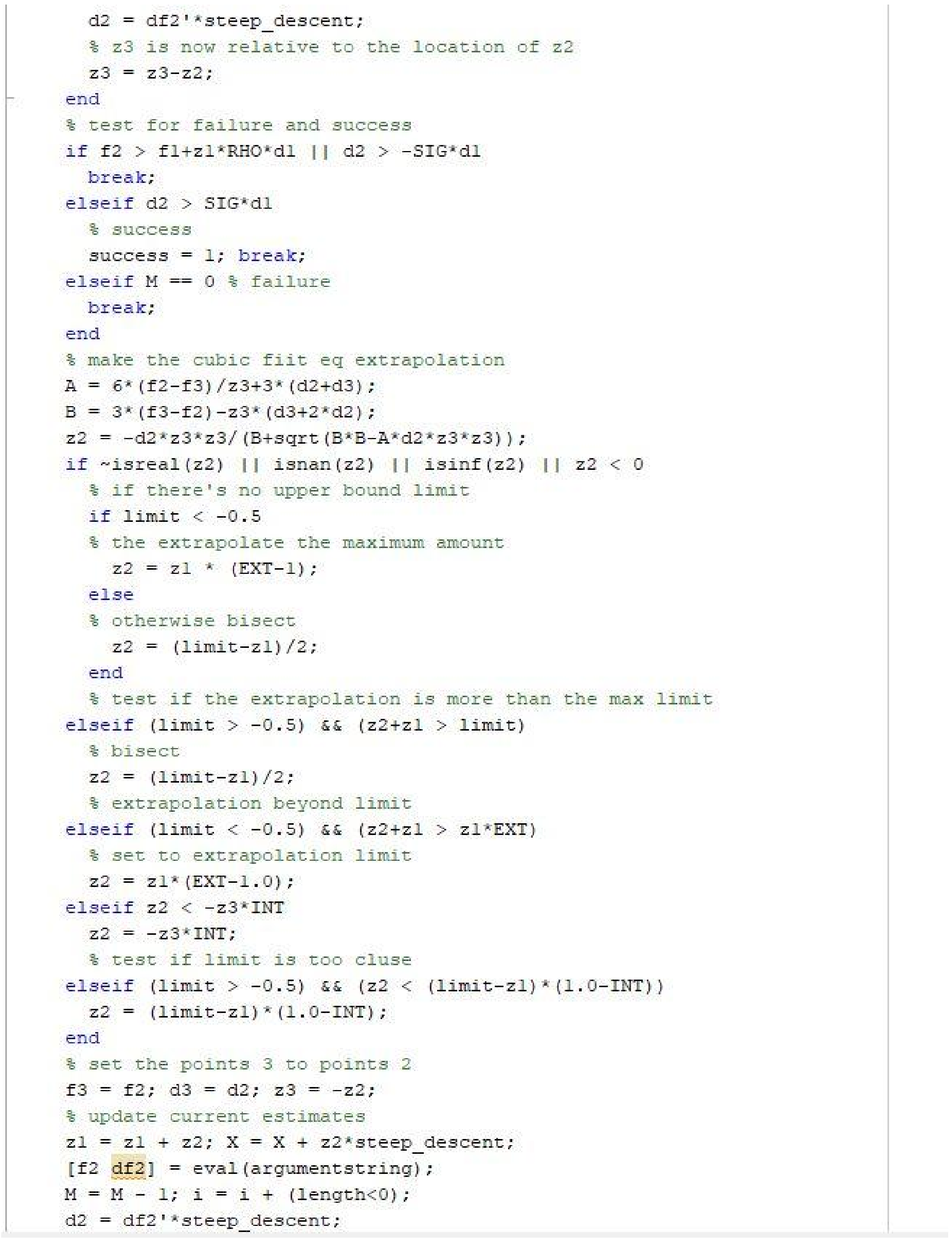
For our next step, I'll be implementing the conjugate gradient method, which we went over on pg 188 on chapter 4 of haykin, so on page 195 of table 4.3, i'll be using that as a reference. For the code, we need to minimize a continuous differentiable multivariate function. Our starting point will be from our variable **X** which I mentioned above was the breast cancer parameters not including the ID of the patient and the cancer status. Our function would need to return a value and a vector of partial derivatives using the Polack Ribiere method to compute the search directions and a line search using the quadratic approximations. Then, there will be some checks to make sure that the extrapolation won't be big. For our iterations, we can choose any value, but I'll be testing some values to see how accurate our results will be. It should indicate the reduction in function to be expected from the first line search and when the function returns if the iterations is up or not much progress has been made. Then our new function variable **fofX** should indicate the vector of function values and how much progression is made with **i** being the number of iterations.



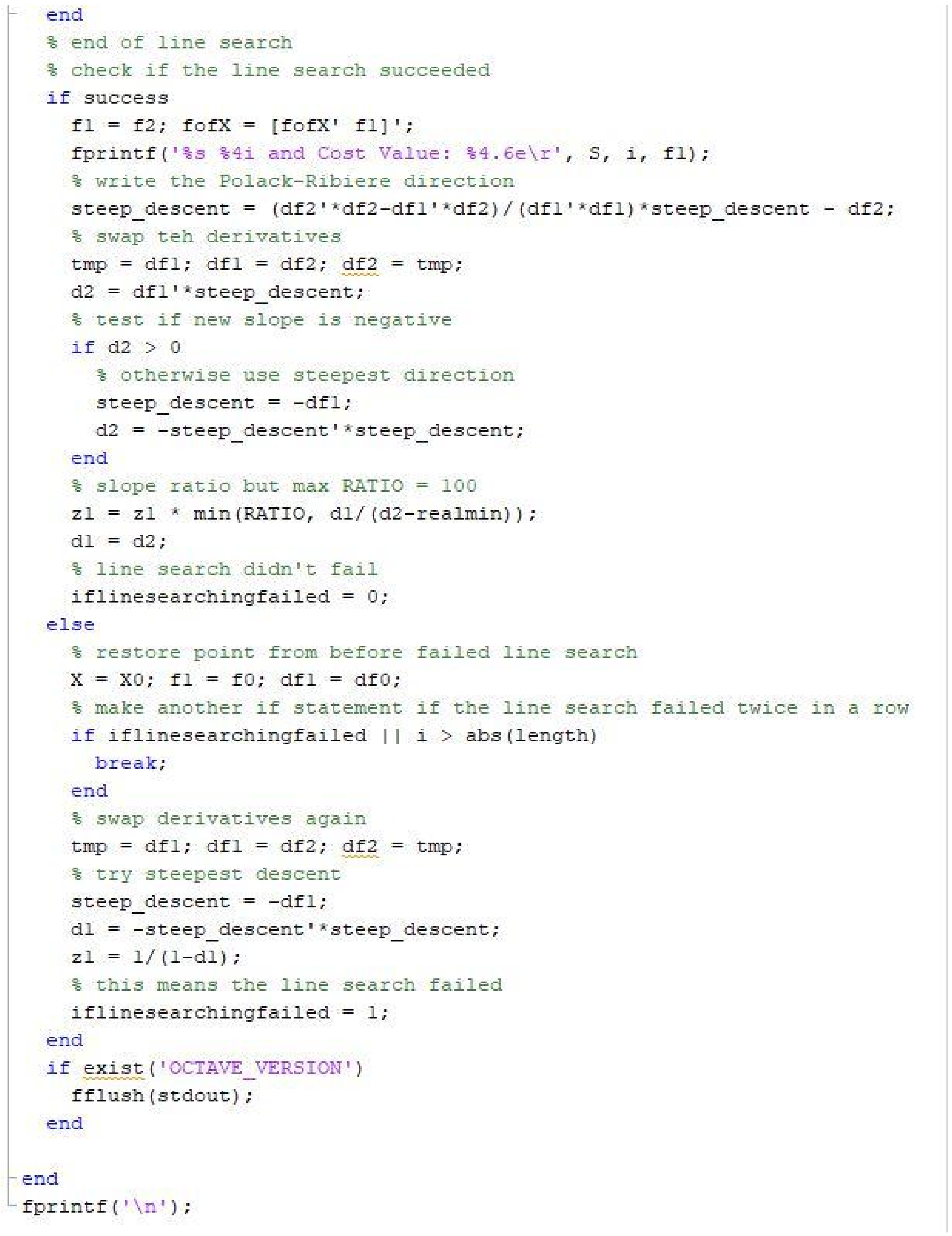
```matlab
function [X, fofX, i] = ConjGradMethod(f, X, options)

% Read options
if exist('options', 'var') && ~isempty(options) && isfield(options, 'MaxIter')
    length = options.MaxIter;
else
    length = 100;
end

% set your rho to be a constant for line searches
RHO = 0.01;
% set your sig value to be 0.5 in the Wolfe-Powell conditions
SIG = 0.5;
% set your limit value to 0.2 of the limit bracket
INT = 0.1;
% extrapolate max of 3 times the current bracket
EXT = 3.0;
% max 20 function evaluations per line search
MAX = 20;
% maximum allowed slope ratio
RATIO = 100;
 % compose a string used to call function
argumentstring = ['feval(f, X'];
for i = 1:(nargin - 3)
  argumentstring = [argumentstring, ',P', int2str(i)];
end
argumentstring = [argumentstring, ')'];

if max(size(length)) == 2
    red=length(2);
    length=length(1);
else red=1;
end
S=['Count '];

% initialize your iteration counter
i = 0;
iflinesearchingfailed = 0;
% create your return empty array values for the function
fofX = [];
% get function value and gradient
[f1 df1] = eval(argumentstring);
% count epochs
i = i + (length<0);
% search the direction for steepest descent
steep_descent = -df1;
% write the slope value
```



```matlab
d1 = -steep_descent'*steep_descent;
% initial step is red/(|s|+1)
z1 = red/(1-d1);

% begin iteration count
while i < abs(length)
  i = i + (length>0);
% make a copy of current values
  X0 = X; f0 = f1; df0 = df1;
  % begin line search
  X = X + z1*steep_descent;
  [f2 df2] = eval(argumentstring);
  i = i + (length<0);
  d2 = df2'*steep_descent;
  % initialize point 3 equal to point 1
  f3 = f1; d3 = d1; z3 = -z1;
  if length>0, M = MAX; else M = min(MAX, -length-i); end
  % initialize quantities
  success = 0; limit = -1;
  % set your loop to always true
  while 1
    while ((f2 > f1+z1*RHO*d1) || (d2 > -SIG*d1)) && (M > 0)
      % tighten the bracket
      limit = z1;
      if f2 > f1
        % here's the quadratic fit eq
        z2 = z3 - (0.5*d3*z3*z3)/(d3*z3+f2-f3);
      else
        % here's the cubic fit eq
        A = 6*(f2-f3)/z3+3*(d2+d3);
        B = 3*(f3-f2)-z3*(d3+2*d2);
        z2 = (sqrt(B*B-A*d2*z3*z3)-B)/A;
      end
      % if we had a numerical problem then bisect
      if isnan(z2) || isinf(z2)
        z2 = z3/2;
      end
      % don't accept too close to the limit brackets
      z2 = max(min(z2, INT*z3),(1-INT)*z3);
      % update the step
      z1 = z1 + z2;
      X = X + z2*steep_descent;
      [f2 df2] = eval(argumentstring);
      M = M - 1;
      i = i + (length<0);
```



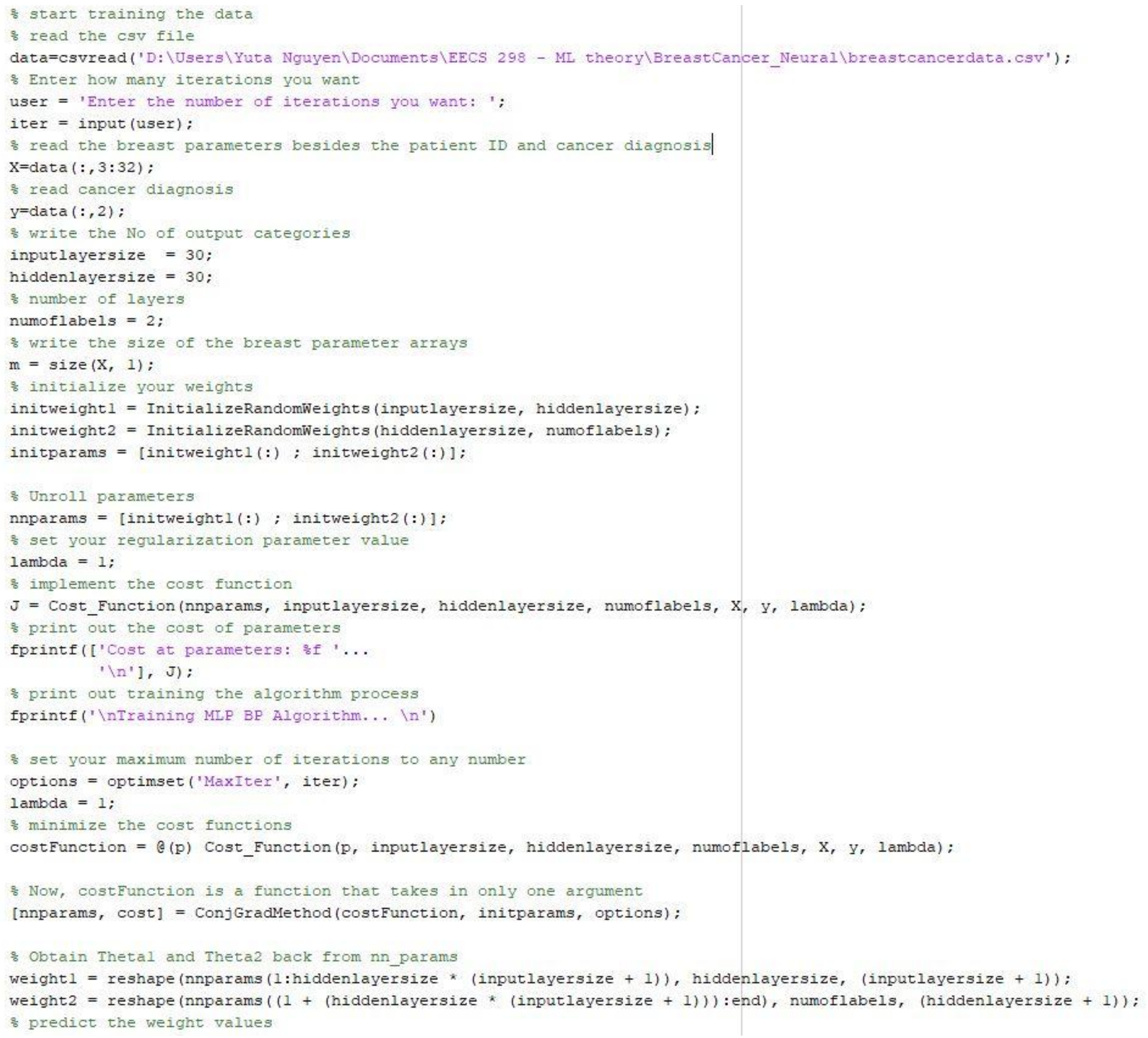
```
  d2 = df2'*steep_descent;
  % z3 is now relative to the location of z2
  z3 = z3-z2;
end
% test for failure and success
if f2 > f1+z1*RHO*d1 || d2 > -SIG*d1
  break;
elseif d2 > SIG*d1
  % success
  success = 1; break;
elseif M == 0 % failure
  break;
end
% make the cubic fiit eq extrapolation
A = 6*(f2-f3)/z3+3*(d2+d3);
B = 3*(f3-f2)-z3*(d3+2*d2);
z2 = -d2*z3*z3/(B+sqrt(B*B-A*d2*z3*z3));
if ~isreal(z2) || isnan(z2) || isinf(z2) || z2 < 0
  % if there's no upper bound limit
  if limit < -0.5
  % the extrapolate the maximum amount
    z2 = z1 * (EXT-1);
  else
  % otherwise bisect
    z2 = (limit-z1)/2;
  end
  % test if the extrapolation is more than the max limit
elseif (limit > -0.5) && (z2+z1 > limit)
  % bisect
  z2 = (limit-z1)/2;
  % extrapolation beyond limit
elseif (limit < -0.5) && (z2+z1 > z1*EXT)
  % set to extrapolation limit
  z2 = z1*(EXT-1.0);
elseif z2 < -z3*INT
  z2 = -z3*INT;
  % test if limit is too cluse
elseif (limit > -0.5) && (z2 < (limit-z1)*(1.0-INT))
  z2 = (limit-z1)*(1.0-INT);
end
% set the points 3 to points 2
f3 = f2; d3 = d2; z3 = -z2;
% update current estimates
z1 = z1 + z2; X = X + z2*steep_descent;
[f2 df2] = eval(argumentstring);
M = M - 1; i = i + (length<0);
d2 = df2'*steep_descent;
```



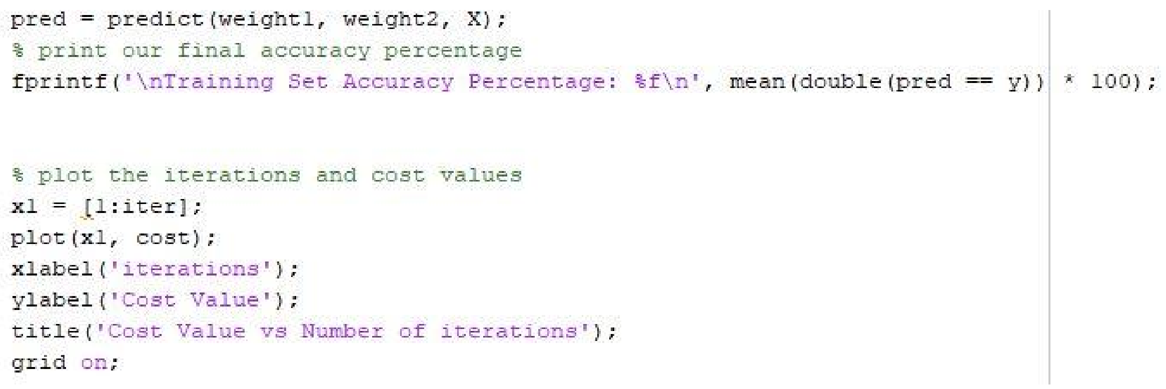
```
    end
    % end of line search
    % check if the line search succeeded
    if success
      f1 = f2; fofX = [fofX' f1]';
      fprintf('%s %4i and Cost Value: %4.6e\r', S, i, f1);
      % write the Polack-Ribiere direction
      steep_descent = (df2'*df2-df1'*df2)/(df1'*df1)*steep_descent - df2;
      % swap teh derivatives
      tmp = df1; df1 = df2; df2 = tmp;
      d2 = df1'*steep_descent;
      % test if new slope is negative
      if d2 > 0
        % otherwise use steepest direction
        steep_descent = -df1;
        d2 = -steep_descent'*steep_descent;
      end
      % slope ratio but max RATIO = 100
      z1 = z1 * min(RATIO, d1/(d2-realmin));
      d1 = d2;
      % line search didn't fail
      iflinesearchingfailed = 0;
    else
      % restore point from before failed line search
      X = X0; f1 = f0; df1 = df0;
      % make another if statement if the line search failed twice in a row
      if iflinesearchingfailed || i > abs(length)
        break;
      end
      % swap derivatives again
      tmp = df1; df1 = df2; df2 = tmp;
      % try steepest descent
      steep_descent = -df1;
      d1 = -steep_descent'*steep_descent;
      z1 = 1/(1-d1);
      % this means the line search failed
      iflinesearchingfailed = 1;
    end
    if exist('OCTAVE_VERSION')
      fflush(stdout);
    end

end
fprintf('\n');
```

Testing and Training the data

For our final step, we'll train the data. First, we'll implement a user input on how many iterations we want to train, then plot out our cost values with respect to the number of iterations



```
% start training the data
% read the csv file
data=csvread('D:\Users\Yuta Nguyen\Documents\EECS 298 - ML theory\BreastCancer_Neural\breastcancerdata.csv');
% Enter how many iterations you want
user = 'Enter the number of iterations you want: ';
iter = input(user);
% read the breast parameters besides the patient ID and cancer diagnosis
X=data(:,3:32);
% read cancer diagnosis
y=data(:,2);
% write the No of output categories
inputlayersize  = 30;
hiddenlayersize = 30;
% number of layers
numoflabels = 2;
% write the size of the breast parameter arrays
m = size(X, 1);
% initialize your weights
initweight1 = InitializeRandomWeights(inputlayersize, hiddenlayersize);
initweight2 = InitializeRandomWeights(hiddenlayersize, numoflabels);
initparams = [initweight1(:) ; initweight2(:)];

% Unroll parameters
nnparams = [initweight1(:) ; initweight2(:)];
% set your regularization parameter value
lambda = 1;
% implement the cost function
J = Cost_Function(nnparams, inputlayersize, hiddenlayersize, numoflabels, X, y, lambda);
% print out the cost of parameters
fprintf(['Cost at parameters: %f '...
         '\n'], J);
% print out training the algorithm process
fprintf('\nTraining MLP BP Algorithm... \n')

% set your maximum number of iterations to any number
options = optimset('MaxIter', iter);
lambda = 1;
% minimize the cost functions
costFunction = @(p) Cost_Function(p, inputlayersize, hiddenlayersize, numoflabels, X, y, lambda);

% Now, costFunction is a function that takes in only one argument
[nnparams, cost] = ConjGradMethod(costFunction, initparams, options);

% Obtain Theta1 and Theta2 back from nn_params
weight1 = reshape(nnparams(1:hiddenlayersize * (inputlayersize + 1)), hiddenlayersize, (inputlayersize + 1));
weight2 = reshape(nnparams((1 + (hiddenlayersize * (inputlayersize + 1))):end), numoflabels, (hiddenlayersize + 1));
% predict the weight values
```

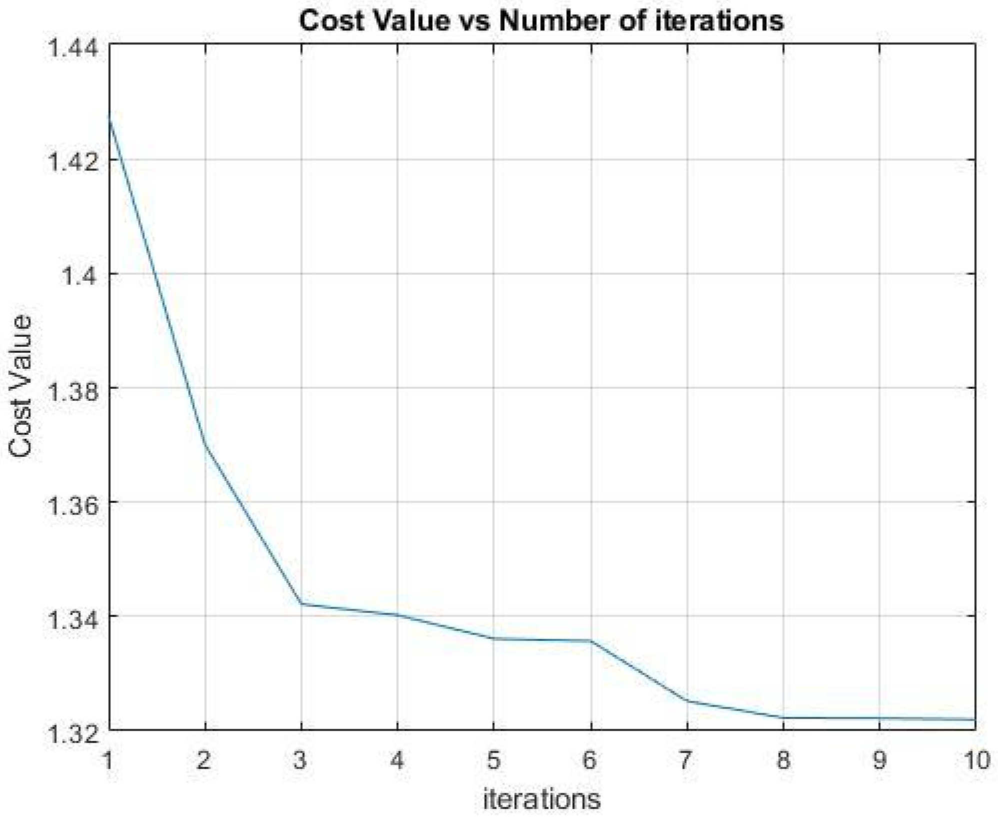


```
pred = predict(weight1, weight2, X);
% print our final accuracy percentage
fprintf('\nTraining Set Accuracy Percentage: %f\n', mean(double(pred == y)) * 100);


% plot the iterations and cost values
x1 = [1:iter];
plot(x1, cost);
xlabel('iterations');
ylabel('Cost Value');
title('Cost Value vs Number of iterations');
grid on;
```

Results

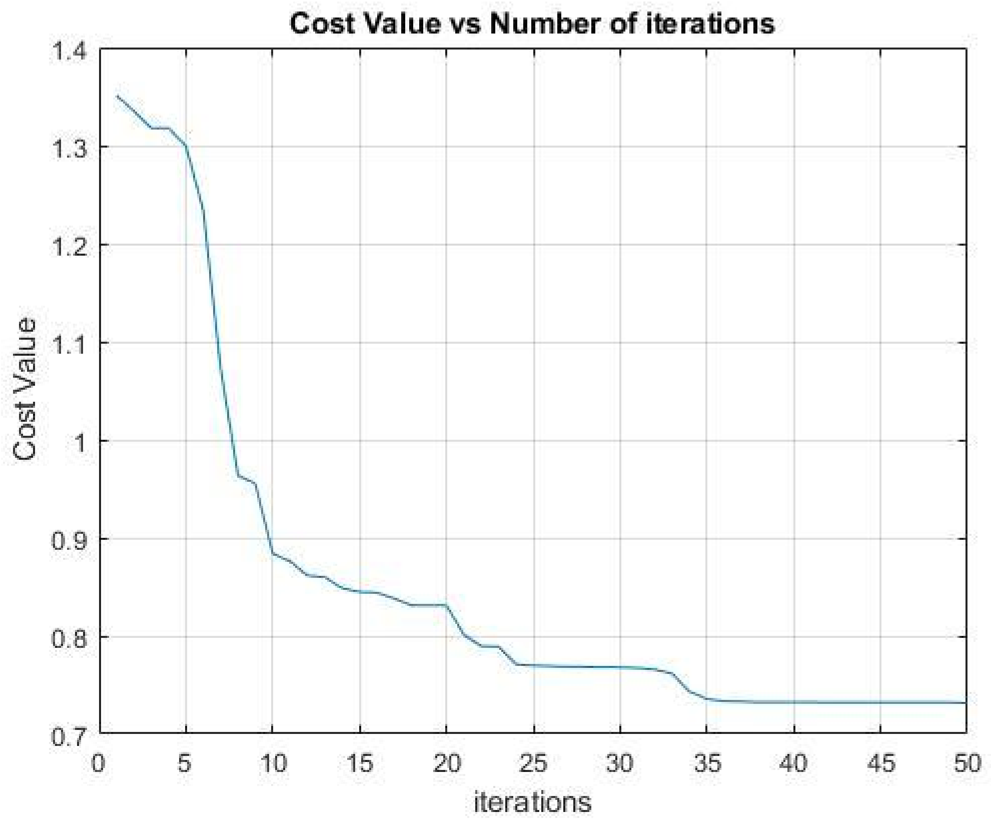
For our results, I first start out at 10 iterations, so our training set accuracy was about **62.7%**

```
Enter the number of iterations you want: 10
Cost at parameters: 1.736947

Training MLP BP Algorithm...
Count     1 and Cost Value: 1.427492e+00
Count     2 and Cost Value: 1.370045e+00
Count     3 and Cost Value: 1.342160e+00
Count     4 and Cost Value: 1.340236e+00
Count     5 and Cost Value: 1.336106e+00
Count     6 and Cost Value: 1.335730e+00
Count     7 and Cost Value: 1.325206e+00
Count     8 and Cost Value: 1.322287e+00
Count     9 and Cost Value: 1.322207e+00
Count    10 and Cost Value: 1.322010e+00


Training Set Accuracy Percentage: 62.741652
```

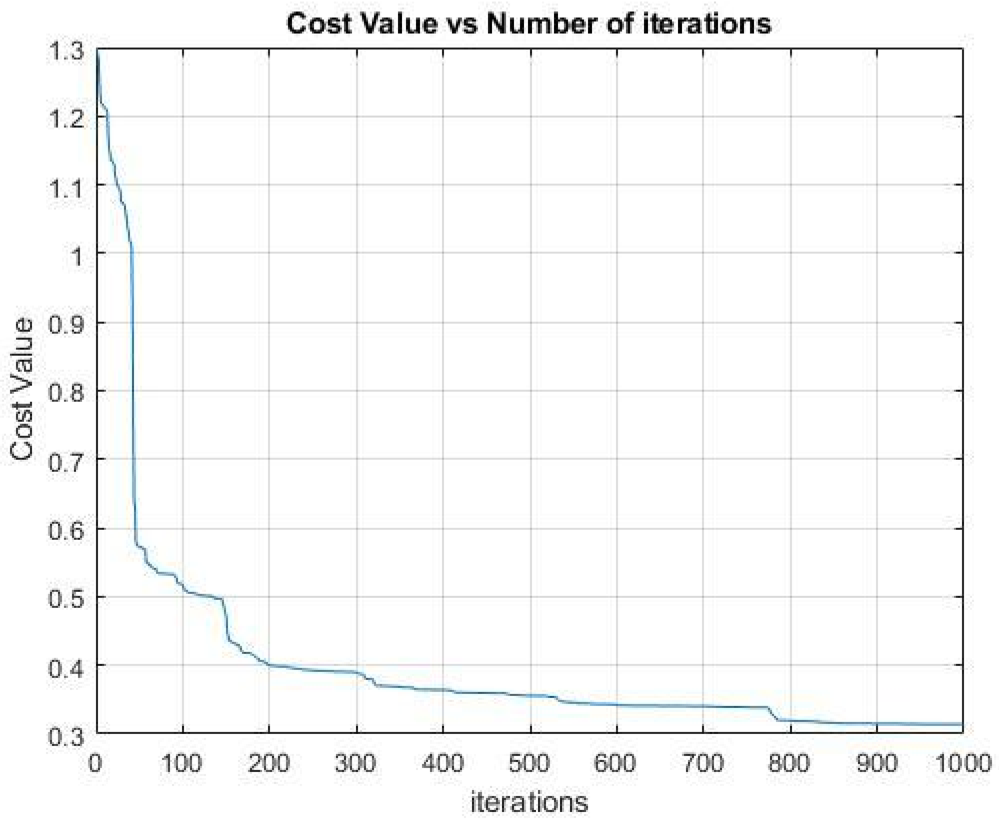


Similarly, we tried 50 iterations and our training set accuracy was **92%**

```
Enter the number of iterations you want: 50
Cost at parameters: 1.482995

Training MLP BP Algorithm...
Count     1 and Cost Value: 1.350973e+00
Count     2 and Cost Value: 1.335250e+00
Count     3 and Cost Value: 1.317918e+00
Count     4 and Cost Value: 1.317916e+00
Count     5 and Cost Value: 1.299767e+00
Count     6 and Cost Value: 1.234080e+00
Count     7 and Cost Value: 1.073135e+00
Count     8 and Cost Value: 9.636029e-01
Count     9 and Cost Value: 9.552942e-01
Count    10 and Cost Value: 8.838352e-01
Count    11 and Cost Value: 8.759950e-01
Count    12 and Cost Value: 8.614673e-01
Count    13 and Cost Value: 8.598733e-01
Count    14 and Cost Value: 8.482994e-01
Count    15 and Cost Value: 8.448833e-01
Count    16 and Cost Value: 8.439893e-01
Count    17 and Cost Value: 8.380015e-01
Count    18 and Cost Value: 8.312443e-01
Count    19 and Cost Value: 8.312442e-01
Count    20 and Cost Value: 8.311477e-01
Count    21 and Cost Value: 8.011834e-01
Count    22 and Cost Value: 7.895329e-01
Count    23 and Cost Value: 7.891972e-01
Count    24 and Cost Value: 7.709735e-01
Count    25 and Cost Value: 7.697578e-01
Count    26 and Cost Value: 7.691486e-01
Count    27 and Cost Value: 7.686606e-01
Count    28 and Cost Value: 7.684844e-01
Count    29 and Cost Value: 7.681065e-01
Count    30 and Cost Value: 7.679380e-01
Count    31 and Cost Value: 7.673150e-01
Count    32 and Cost Value: 7.656547e-01
Count    33 and Cost Value: 7.615133e-01
Count    34 and Cost Value: 7.431271e-01
Count    35 and Cost Value: 7.356466e-01
Count    36 and Cost Value: 7.333330e-01
Count    37 and Cost Value: 7.330335e-01
Count    38 and Cost Value: 7.323538e-01
Count    39 and Cost Value: 7.323405e-01
Count    40 and Cost Value: 7.323239e-01
Count    41 and Cost Value: 7.322838e-01
Count    42 and Cost Value: 7.322383e-01
Count    43 and Cost Value: 7.322338e-01
Count    44 and Cost Value: 7.322143e-01
Count    45 and Cost Value: 7.322024e-01
Count    46 and Cost Value: 7.321885e-01
Count    47 and Cost Value: 7.321675e-01
Count    48 and Cost Value: 7.321623e-01
Count    49 and Cost Value: 7.321139e-01
Count    50 and Cost Value: 7.315487e-01


Training Set Accuracy Percentage: 91.915641
```



As our number of iterations increases, our accuracy increases, and with 1000 iterations for instance, I got **93.7%**



Conclusion

In this project, I got a chance to work on and write about breast cancer classification using MLP learning. In our results, we can see that as the number of iterations increases, the training accuracy increases as well. I just wanted to say a huge thanks to you professor for teaching machine learning for my last quarter before graduation and it was a pleasure learning all different types of neural networks, even though sometimes they were difficult to understand. Have a safe Christmas and a Happy New Year!