Gradient Checking is based on the following approach. One iteration of Gradient Descent computes and updates the parameters \thetaby doing  
\theta := \theta - \frac{d}{d\theta}J(\theta).  
To minimize the cost we will need to minimize J(\theta)  
Let g(\theta)be a function that computes the derivative \frac {d}{d\theta}J(\theta). Gradient Checking allows us to numerically evaluate the implementation of the function g(\theta)and verify its correctness.  
We know the derivative of a function is given by  
\frac {d}{d\theta}J(\theta) = lim->0 \frac {J(\theta +\epsilon) - J(\theta -\epsilon)} {2*\epsilon}  
**Note**: The above derivative is based on the 2 sided derivative. The 1-sided derivative  is given by \frac {d}{d\theta}J(\theta) = lim->0 \frac {J(\theta +\epsilon) - J(\theta)} {\epsilon}  
Gradient Checking is based on the 2-sided derivative because the error is of the order O(\epsilon^{2})as opposed O(\epsilon)for the 1-sided derivative.  
Hence Gradient Check uses the 2 sided derivative as follows.  
g(\theta) = lim->0 \frac {J(\theta +\epsilon) - J(\theta -\epsilon)} {2*\epsilon}

In Gradient Check the following is done  
A) Run one normal cycle of your implementation by doing the following  
a) Compute the output activation by running 1 cycle of forward propagation  
b) Compute the cost using the output activation  
c) Compute the gradients using backpropation (grad)

B) Perform gradient check steps as below  
a) Set \theta. Flatten all ‘weights’ and ‘bias’ matrices and vectors to a column vector.  
b) Initialize \theta+by bumping up \thetaby adding \epsilon(\theta + \epsilon)  
c) Perform forward propagation with \theta+  
d) Compute cost with \theta+i.e. J(\theta+)  
e) Initialize  \theta-by bumping down \thetaby subtracting \epsilon(\theta - \epsilon)  
f) Perform forward propagation with \theta-  
g) Compute cost with \theta-i.e.  J(\theta-)  
h) Compute \frac {d} {d\theta} J(\theta)or ‘gradapprox’ as\frac {J(\theta+) - J(\theta-) } {2\epsilon} using the 2 sided derivative.  
i) Compute L2norm or the Euclidean distance between ‘grad’ and ‘gradapprox’. If the  
diference is of the order of 10^{-5}or 10^{-7}the implementation is correct.

A difference of 10^{-5}is also ok. Anything more than that is a cause of worry and you should look at your code more closely.

Deep Learning Code

DL8Functions.M

|  |
| --- |
| 1; |
|  | # Define sigmoid function |
|  | function [A,cache] = sigmoid(Z) |
|  | A = 1 ./ (1+ exp(-Z)); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu function |
|  | function [A,cache] = relu(Z) |
|  | A = max(0,Z); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu function |
|  | function [A,cache] = tanhAct(Z) |
|  | A = tanh(Z); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Softmax function |
|  | function [A,cache] = softmax(Z) |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(Z'); |
|  | # normalize them for each example |
|  | A = exp\_scores ./ sum(exp\_scores,2); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Softmax function |
|  | function [A,cache] = stableSoftmax(Z) |
|  | # Normalize by max value in each row |
|  | shiftZ = Z' - max(Z',[],2); |
|  | exp\_scores = exp(shiftZ); |
|  | # normalize them for each example |
|  | A = exp\_scores ./ sum(exp\_scores,2); |
|  | #disp("sm") |
|  | #disp(A); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu Derivative |
|  | function [dZ] = reluDerivative(dA,cache) |
|  | Z = cache; |
|  | dZ = dA; |
|  | # Get elements that are greater than 0 |
|  | a = (Z > 0); |
|  | # Select only those elements where Z > 0 |
|  | dZ = dZ .\* a; |
|  | end |
|  |  |
|  | # Define Sigmoid Derivative |
|  | function [dZ] = sigmoidDerivative(dA,cache) |
|  | Z = cache; |
|  | s = 1 ./ (1+ exp(-Z)); |
|  | dZ = dA .\* s .\* (1-s); |
|  | end |
|  |  |
|  | # Define Tanh Derivative |
|  | function [dZ] = tanhDerivative(dA,cache) |
|  | Z = cache; |
|  | a = tanh(Z); |
|  | dZ = dA .\* (1 - a .^ 2); |
|  | end |
|  |  |
|  | # Populate a matrix with 1s in rows where Y=1 |
|  | # This function may need to be modified if K is not 3, 10 |
|  | function [Y1] = popMatrix(Y,numClasses) |
|  | Y1=zeros(length(Y),numClasses); |
|  | if(numClasses==3) # For 3 output classes |
|  | Y1(Y==0,1)=1; |
|  | Y1(Y==1,2)=1; |
|  | Y1(Y==2,3)=1; |
|  | elseif(numClasses==10) # For 10 output classes |
|  | Y1(Y==0,1)=1; |
|  | Y1(Y==1,2)=1; |
|  | Y1(Y==2,3)=1; |
|  | Y1(Y==3,4)=1; |
|  | Y1(Y==4,5)=1; |
|  | Y1(Y==5,6)=1; |
|  | Y1(Y==6,7)=1; |
|  | Y1(Y==7,8)=1; |
|  | Y1(Y==8,9)=1; |
|  | Y1(Y==9,10)=1; |
|  |  |
|  | endif |
|  | end |
|  |  |
|  | # Define Softmax Derivative |
|  | function [dZ] = softmaxDerivative(dA,cache,Y, numClasses) |
|  | Z = cache; |
|  | # get unnormalized probabilities |
|  | shiftZ = Z' - max(Z',[],2); |
|  | exp\_scores = exp(shiftZ); |
|  |  |
|  | # normalize them for each example |
|  | probs = exp\_scores ./ sum(exp\_scores,2); |
|  | # dZ = pi- yi |
|  | yi=popMatrix(Y,numClasses); |
|  | dZ=probs-yi; |
|  |  |
|  | end |
|  |  |
|  | # Define Softmax Derivative |
|  | function [dZ] = stableSoftmaxDerivative(dA,cache,Y, numClasses) |
|  | Z = cache; |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(Z'); |
|  | # normalize them for each example |
|  | probs = exp\_scores ./ sum(exp\_scores,2); |
|  | # dZ = pi- yi |
|  | yi=popMatrix(Y,numClasses); |
|  | dZ=probs-yi; |
|  |  |
|  | end |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | function [W b] = initializeDeepModel(layerDimensions) |
|  | rand ("seed", 3); |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Create cell arrays for Weights and biases |
|  |  |
|  | for l =2:size(layerDimensions)(2) |
|  | W{l-1} = rand(layerDimensions(l),layerDimensions(l-1))\*0.01; # Multiply by .01 |
|  | b{l-1} = zeros(layerDimensions(l),1); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  |  |
|  | # He Initialization the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  |  |
|  |  |
|  | # He Initialization for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | function [W b] = HeInitializeDeepModel(layerDimensions) |
|  | rand ("seed", 3); |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Create cell arrays for Weights and biases |
|  |  |
|  | for l =2:size(layerDimensions)(2) |
|  | W{l-1} = rand(layerDimensions(l),layerDimensions(l-1))\* sqrt(2/layerDimensions(l-1)); # Multiply by .01 |
|  | b{l-1} = zeros(layerDimensions(l),1); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  | # Xavier Initialization for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | function [W b] = XavInitializeDeepModel(layerDimensions) |
|  | rand ("seed", 3); |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Create cell arrays for Weights and biases |
|  |  |
|  | for l =2:size(layerDimensions)(2) |
|  | W{l-1} = rand(layerDimensions(l),layerDimensions(l-1))\* sqrt(1/layerDimensions(l-1)); # Multiply by .01 |
|  | b{l-1} = zeros(layerDimensions(l),1); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  | # Initialize velocity |
|  | # Input : parameters |
|  | # Returns: Initial velocity v |
|  | function[vdW vdB] = initializeVelocity(weights, biases) |
|  |  |
|  | L = size(weights)(2) # Create an integer |
|  | # Initialize a cell array |
|  | v = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l=1:L |
|  | sz = size(weights{l}); |
|  | vdW{l} = zeros(sz(1),sz(2)); |
|  | sz = size(biases{l}); |
|  | vdB{l} =zeros(sz(1),sz(2)); |
|  | endfor; |
|  | end |
|  |  |
|  | # Initialize RMSProp |
|  | # Input : parameters |
|  | # Returns: Initial RMSProp |
|  | function[sdW sdB] = initializeRMSProp(weights, biases) |
|  |  |
|  | L = size(weights)(2) # Create an integer |
|  | # Initialize a cell array |
|  | s = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l=1:L |
|  | sz = size(weights{l}); |
|  | sdW{l} = zeros(sz(1),sz(2)); |
|  | sz = size(biases{l}); |
|  | sdB{l} =zeros(sz(1),sz(2)); |
|  | endfor; |
|  | end |
|  |  |
|  | # Initialize Adam |
|  | # Input : parameters |
|  | # Returns: Initial Adam |
|  | function[vdW vdB sdW sdB] = initializeAdam(weights, biases) |
|  |  |
|  | L = size(weights)(2) # Create an integer |
|  | # Initialize a cell array |
|  | s = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l=1:L |
|  | sz = size(weights{l}); |
|  | vdW{l} = zeros(sz(1),sz(2)); |
|  | sdW{l} = zeros(sz(1),sz(2)); |
|  | sz = size(biases{l}); |
|  | sdB{l} =zeros(sz(1),sz(2)); |
|  | vdB{l} =zeros(sz(1),sz(2)); |
|  | endfor; |
|  | end |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | function [A forward\_cache activation\_cache] = layerActivationForward(A\_prev, W, b, activationFunc) |
|  |  |
|  | # Compute Z |
|  | Z = W \* A\_prev +b; |
|  | # Create a cell array |
|  | forward\_cache = {A\_prev W b}; |
|  | # Compute the activation for sigmoid |
|  | if (strcmp(activationFunc,"sigmoid")) |
|  | [A activation\_cache] = sigmoid(Z); |
|  | elseif (strcmp(activationFunc, "relu")) # Compute the activation for Relu |
|  | [A activation\_cache] = relu(Z); |
|  | elseif(strcmp(activationFunc,'tanh')) # Compute the activation for tanh |
|  | [A activation\_cache] = tanhAct(Z); |
|  | elseif(strcmp(activationFunc,'softmax')) # Compute the activation for tanh |
|  | #[A activation\_cache] = softmax(Z); |
|  | [A activation\_cache] = stableSoftmax(Z); |
|  | endif |
|  |  |
|  | end |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - Activation function at hidden layers Relu/tanh |
|  | # outputActivationFunc- sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | function [AL forward\_caches activation\_caches dropoutMat] = forwardPropagationDeep(X, weights,biases, keep\_prob=1, |
|  | hiddenActivationFunc='relu', outputActivationFunc='sigmoid') |
|  | # Create an empty cell array |
|  | forward\_caches = {}; |
|  | activation\_caches = {}; |
|  | droputMat ={}; |
|  | # Set A to X (A0) |
|  | A = X; |
|  | L = length(weights); # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for l =1:L-1 |
|  | A\_prev = A; |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | W = weights{l}; |
|  | b = biases{l}; |
|  | [A forward\_cache activation\_cache] = layerActivationForward(A\_prev, W,b, activationFunc=hiddenActivationFunc); |
|  | D=rand(size(A)(1),size(A)(2)); |
|  | D = (D < keep\_prob) ; |
|  | # Multiply by DropoutMat |
|  | A= A .\* D; |
|  | # Divide by keep\_prob to keep expected value same |
|  | A = A ./ keep\_prob; |
|  | # Store D |
|  | dropoutMat{l}=D; |
|  | forward\_caches{l}=forward\_cache; |
|  | activation\_caches{l} = activation\_cache; |
|  | endfor |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | W = weights{L}; |
|  | b = biases{L}; |
|  | [AL, forward\_cache activation\_cache] = layerActivationForward(A, W,b, activationFunc = outputActivationFunc); |
|  | forward\_caches{L}=forward\_cache; |
|  | activation\_caches{L} = activation\_cache; |
|  |  |
|  | end |
|  |  |
|  | # Pick columns where Y==1 |
|  | function [a] = pickColumns(AL,Y,numClasses) |
|  | if(numClasses==3) |
|  | a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | elseif (numClasses==10) |
|  | a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3);AL(Y==3,4);AL(Y==4,5); |
|  | AL(Y==5,6); AL(Y==6,7);AL(Y==7,8);AL(Y==8,9);AL(Y==9,10)]; |
|  | endif |
|  | end |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : outputActivationFunc- sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | function [cost]= computeCost(AL, Y, outputActivationFunc="sigmoid",numClasses) |
|  | if(strcmp(outputActivationFunc,"sigmoid")) |
|  | numTraining= size(Y)(2); |
|  | # Element wise multiply for logprobs |
|  | cost = -1/numTraining \* sum((Y .\* log(AL)) + (1-Y) .\* log(1-AL)); |
|  | #disp(cost); |
|  |  |
|  | elseif(strcmp(outputActivationFunc,'softmax')) |
|  | numTraining = size(Y)(2); |
|  | Y=Y'; |
|  | # Select rows where Y=0,1,and 2 and concatenate to a long vector |
|  | #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | a =pickColumns(AL,Y,numClasses); |
|  |  |
|  | #Select the correct column for log prob |
|  | correct\_probs = -log(a); |
|  | #Compute log loss |
|  | cost= sum(correct\_probs)/numTraining; |
|  | endif |
|  | end |
|  |  |
|  | # Compute the cost with regularization |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : outputActivationFunc- sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | function [cost]= computeCostWithReg(weights, AL, Y, lambd, outputActivationFunc="sigmoid",numClasses) |
|  |  |
|  | if(strcmp(outputActivationFunc,"sigmoid")) |
|  | numTraining= size(Y)(2); |
|  | # Element wise multiply for logprobs |
|  | cost = -1/numTraining \* sum((Y .\* log(AL)) + (1-Y) .\* log(1-AL)); |
|  |  |
|  | # Regularization cost |
|  | L = size(weights)(2); |
|  | L2RegularizationCost=0; |
|  | for l=1:L |
|  | wtSqr = weights{l} .\* weights{l}; |
|  | #disp(sum(sum(wtSqr,1))); |
|  | L2RegularizationCost+=sum(sum(wtSqr,1)); |
|  | endfor |
|  | L2RegularizationCost = (lambd/(2\*numTraining))\*L2RegularizationCost; |
|  | cost = cost + L2RegularizationCost ; |
|  |  |
|  |  |
|  | elseif(strcmp(outputActivationFunc,'softmax')) |
|  | numTraining = size(Y)(2); |
|  | Y=Y'; |
|  | # Select rows where Y=0,1,and 2 and concatenate to a long vector |
|  | #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | a =pickColumns(AL,Y,numClasses); |
|  |  |
|  | #Select the correct column for log prob |
|  | correct\_probs = -log(a); |
|  | #Compute log loss |
|  | cost= sum(correct\_probs)/numTraining; |
|  | # Regularization cost |
|  | L = size(weights)(2); |
|  | L2RegularizationCost=0; |
|  | for l=1:L |
|  | # Compute L2 Norm |
|  | wtSqr = weights{l} .\* weights{l}; |
|  | #disp(sum(sum(wtSqr,1))); |
|  | L2RegularizationCost+=sum(sum(wtSqr,1)); |
|  | endfor |
|  | L2RegularizationCost = (lambd/(2\*numTraining))\*L2RegularizationCost; |
|  | cost = cost + L2RegularizationCost ; |
|  | endif |
|  | end |
|  |  |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | function [dA\_prev dW db] = layerActivationBackward(dA, forward\_cache, activation\_cache, Y, activationFunc,numClasses) |
|  |  |
|  | A\_prev = forward\_cache{1}; |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | numTraining = size(A\_prev)(2); |
|  | if (strcmp(activationFunc,"relu")) |
|  | dZ = reluDerivative(dA, activation\_cache); |
|  | elseif (strcmp(activationFunc,"sigmoid")) |
|  | dZ = sigmoidDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "tanh")) |
|  | dZ = tanhDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "softmax")) |
|  | #dZ = softmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | endif |
|  |  |
|  |  |
|  | if (strcmp(activationFunc,"softmax")) |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* A\_prev \* dZ; |
|  | db = 1/numTraining \* sum(dZ,1); |
|  | dA\_prev = dZ\*W; |
|  | else |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* dZ \* A\_prev'; |
|  | db = 1/numTraining \* sum(dZ,2); |
|  | dA\_prev = W'\*dZ; |
|  | endif |
|  |  |
|  | end |
|  |  |
|  | # Compute the backpropoagation with regularization for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | function [dA\_prev dW db] = layerActivationBackwardWithReg(dA, forward\_cache, activation\_cache, Y, lambd=0, activationFunc,numClasses) |
|  |  |
|  | A\_prev = forward\_cache{1}; |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | numTraining = size(A\_prev)(2); |
|  | if (strcmp(activationFunc,"relu")) |
|  | dZ = reluDerivative(dA, activation\_cache); |
|  | elseif (strcmp(activationFunc,"sigmoid")) |
|  | dZ = sigmoidDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "tanh")) |
|  | dZ = tanhDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "softmax")) |
|  | #dZ = softmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | endif |
|  |  |
|  | if (strcmp(activationFunc,"softmax")) |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* A\_prev \* dZ + (lambd/numTraining) \* W'; |
|  | db = 1/numTraining \* sum(dZ,1); |
|  | dA\_prev = dZ\*W; |
|  | else |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* dZ \* A\_prev' + (lambd/numTraining) \* W; |
|  | db = 1/numTraining \* sum(dZ,2); |
|  | dA\_prev = W'\*dZ; |
|  | endif |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # gradients["dW" + str(l)] = ... |
|  |  |
|  | function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches, |
|  | dropoutMat, lambd=0, keep\_prob=1, hiddenActivationFunc='relu',outputActivationFunc="sigmoid",numClasses) |
|  |  |
|  |  |
|  | # Set the number of layers |
|  | L = length(activation\_caches); |
|  | m = size(AL)(2); |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a + (1-y)/(1-a)) - At the output layer |
|  | dAL = -((Y ./ AL) - (1 - Y) ./ ( 1 - AL)); |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | dAL=0; |
|  | Y=Y'; |
|  | endif |
|  |  |
|  |  |
|  | # Since this is a binary classification the activation at output is sigmoid |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | activation\_cache = activation\_caches{L}; |
|  | forward\_cache = forward\_caches(L); |
|  | # Note the cell array includes an array of forward caches. To get to this we need to include the index {1} |
|  | if (lambd==0) |
|  | [dA dW db] = layerActivationBackward(dAL, forward\_cache{1}, activation\_cache, Y, activationFunc = outputActivationFunc,numClasses); |
|  | else |
|  | [dA dW db] = layerActivationBackwardWithReg(dAL, forward\_cache{1}, activation\_cache, Y, lambd, activationFunc = outputActivationFunc,numClasses); |
|  | endif |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | gradsDA{L}= dA; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | gradsDA{L}= dA';#Note the transpose |
|  | endif |
|  | gradsDW{L}= dW; |
|  | gradsDB{L}= db; |
|  |  |
|  | # Traverse in the reverse direction |
|  | for l =(L-1):-1:1 |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | activation\_cache = activation\_caches{l}; |
|  | forward\_cache = forward\_caches(l); |
|  |  |
|  | #dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+1)], current\_cache, activationFunc = "relu") |
|  | # dAl the dervative of the activation of the lth layer,is the first element |
|  | dAl= gradsDA{l+1}; |
|  | if(lambd == 0) |
|  | # Get the dropout mat |
|  | D = dropoutMat{l}; |
|  | #Multiply by the dropoutMat |
|  | dAl= dAl .\* D; |
|  | # Divide by keep\_prob to keep expected value same |
|  | dAl = dAl ./ keep\_prob; |
|  | [dA\_prev\_temp, dW\_temp, db\_temp] = layerActivationBackward(dAl, forward\_cache{1}, activation\_cache, Y, activationFunc = hiddenActivationFunc,numClasses); |
|  | else |
|  | [dA\_prev\_temp, dW\_temp, db\_temp] = layerActivationBackwardWithReg(dAl, forward\_cache{1}, activation\_cache, Y, lambd, activationFunc = hiddenActivationFunc,numClasses); |
|  | endif |
|  | gradsDA{l}= dA\_prev\_temp; |
|  | gradsDW{l}= dW\_temp; |
|  | gradsDB{l}= db\_temp; |
|  |  |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc |
|  | #output : Updated weights after 1 iteration |
|  | function [weights biases] = gradientDescent(weights, biases,gradsW,gradsB, learningRate,outputActivationFunc="sigmoid") |
|  |  |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | weights{l} = weights{l} -learningRate\* gradsW{l}; |
|  | biases{l} = biases{l} -learningRate\* gradsB{l}; |
|  | endfor |
|  |  |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | weights{L} = weights{L} -learningRate\* gradsW{L}; |
|  | biases{L} = biases{L} -learningRate\* gradsB{L}; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | weights{L} = weights{L} -learningRate\* gradsW{L}'; |
|  | biases{L} = biases{L} -learningRate\* gradsB{L}'; |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Update parameters with momentum |
|  | # Input : parameters |
|  | # : gradients |
|  | # : v |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters and velocity |
|  | function [weights biases] = gradientDescentWithMomentum(weights, biases,gradsDW,gradsDB, vdW, vdB, beta, learningRate,outputActivationFunc="sigmoid") |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | # Compute velocities |
|  | # v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk |
|  | vdW{l} = beta\*vdW{l} + (1 -beta) \* gradsDW{l}; |
|  | vdB{l} = beta\*vdB{l} + (1 -beta) \* gradsDB{l}; |
|  | weights{l} = weights{l} -learningRate\* vdW{l}; |
|  | biases{l} = biases{l} -learningRate\* vdB{l}; |
|  | endfor |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | vdW{L} = beta\*vdW{L} + (1 -beta) \* gradsDW{L}; |
|  | vdB{L} = beta\*vdB{L} + (1 -beta) \* gradsDB{L}; |
|  | weights{L} = weights{L} -learningRate\* vdW{L}; |
|  | biases{L} = biases{L} -learningRate\* vdB{L}; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | vdW{L} = beta\*vdW{L} + (1 -beta) \* gradsDW{L}'; |
|  | vdB{L} = beta\*vdB{L} + (1 -beta) \* gradsDB{L}'; |
|  | weights{L} = weights{L} -learningRate\* vdW{L}; |
|  | biases{L} = biases{L} -learningRate\* vdB{L}; |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Update parameters with RMSProp |
|  | # Input : parameters |
|  | # : gradients |
|  | # : s |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters RMSProp |
|  | function [weights biases] = gradientDescentWithRMSProp(weights, biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon, learningRate,outputActivationFunc="sigmoid") |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | sdW{l} = beta1\*sdW{l} + (1 -beta1) \* gradsDW{l} .\* gradsDW{l}; |
|  | sdB{l} = beta1\*sdB{l} + (1 -beta1) \* gradsDB{l} .\* gradsDB{l}; |
|  | weights{l} = weights{l} - learningRate\* gradsDW{l} ./ sqrt(sdW{l} + epsilon); |
|  | biases{l} = biases{l} - learningRate\* gradsDB{l} ./ sqrt(sdB{l} + epsilon); |
|  | endfor |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | sdW{L} = beta1\*sdW{L} + (1 -beta1) \* gradsDW{L} .\* gradsDW{L}; |
|  | sdB{L} = beta1\*sdB{L} + (1 -beta1) \* gradsDB{L} .\* gradsDB{L}; |
|  | weights{L} = weights{L} -learningRate\* gradsDW{L} ./ sqrt(sdW{L} +epsilon); |
|  | biases{L} = biases{L} -learningRate\* gradsDB{L} ./ sqrt(sdB{L} + epsilon); |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | sdW{L} = beta1\*sdW{L} + (1 -beta1) \* gradsDW{L}' .\* gradsDW{L}'; |
|  | sdB{L} = beta1\*sdB{L} + (1 -beta1) \* gradsDB{L}' .\* gradsDB{L}'; |
|  | weights{L} = weights{L} -learningRate\* gradsDW{L}' ./ sqrt(sdW{L} +epsilon); |
|  | biases{L} = biases{L} -learningRate\* gradsDB{L}' ./ sqrt(sdB{L} + epsilon); |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Update parameters with Adam |
|  | # Input : parameters |
|  | # : gradients |
|  | # : v |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters and velocity |
|  | function [weights biases] = gradientDescentWithAdam(weights, biases,gradsDW,gradsDB, |
|  | vdW, vdB, sdW, sdB, t, beta1, beta2, epsilon, learningRate,outputActivationFunc="sigmoid") |
|  | vdW\_corrected = {}; |
|  | vdB\_corrected = {}; |
|  | sdW\_corrected = {}; |
|  | sdB\_corrected = {}; |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | vdW{l} = beta1\*vdW{l} + (1 -beta1) \* gradsDW{l}; |
|  | vdB{l} = beta1\*vdB{l} + (1 -beta1) \* gradsDB{l}; |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | vdW\_corrected{l} = vdW{l}/(1-beta1^t); |
|  | vdB\_corrected{l} = vdB{l}/(1-beta1^t); |
|  |  |
|  | sdW{l} = beta2\*sdW{l} + (1 -beta2) \* gradsDW{l} .\* gradsDW{l}; |
|  | sdB{l} = beta2\*sdB{l} + (1 -beta2) \* gradsDB{l} .\* gradsDB{l}; |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | sdW\_corrected{l} = sdW{l}/(1-beta2^t); |
|  | sdB\_corrected{l} = sdB{l}/(1-beta2^t); |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(sdW\_corrected{l}+epsilon); |
|  | d2=sqrt(sdB\_corrected{l}+epsilon); |
|  |  |
|  | weights{l} = weights{l} - learningRate\* vdW\_corrected{l} ./ d1; |
|  | biases{l} = biases{l} -learningRate\* vdB\_corrected{l} ./ d2; |
|  | endfor |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | vdW{L} = beta1\*vdW{L} + (1 -beta1) \* gradsDW{L}; |
|  | vdB{L} = beta1\*vdB{L} + (1 -beta1) \* gradsDB{L}; |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | vdW\_corrected{L} = v{L}/(1-beta1^t); |
|  | vdB\_corrected{L} = v{L}/(1-beta1^t); |
|  |  |
|  | sdW{L} = beta2\*sdW{L} + (1 -beta2) \* gradsDW{L} .\* gradsDW{L}; |
|  | sdB{L} = beta2\*sdB{L} + (1 -beta2) \* gradsDB{L} .\* gradsDB{L}; |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | sdW\_corrected{L} = s{L}/(1-beta2^t); |
|  | sdB\_corrected{L} = s{L}/(1-beta2^t); |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(sdW\_corrected{L}+epsilon); |
|  | d2=sqrt(sdB\_corrected{L}+epsilon); |
|  |  |
|  | weights{L} = weights{L} - learningRate\* vdW\_corrected{L} ./ d1; |
|  | biases{L} = biases{L} -learningRate\* vdB\_corrected{L} ./ d2; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | vdW{L} = beta1\*vdW{L} + (1 -beta1) \* gradsDW{L}'; |
|  | vdB{L} = beta1\*vdB{L} + (1 -beta1) \* gradsDB{L}'; |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | vdW\_corrected{L} = vdW{L}/(1-beta1^t); |
|  | vdB\_corrected{L} = vdB{L}/(1-beta1^t); |
|  |  |
|  | sdW{L} = beta2\*sdW{L} + (1 -beta2) \* gradsDW{L}' .\* gradsDW{L}'; |
|  | sdB{L} = beta2\*sdB{L} + (1 -beta2) \* gradsDB{L}' .\* gradsDB{L}'; |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | sdW\_corrected{L} = sdW{L}/(1-beta2^t); |
|  | sdB\_corrected{L} = sdB{L}/(1-beta2^t); |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(sdW\_corrected{L}+epsilon); |
|  | d2=sqrt(sdB\_corrected{L}+epsilon); |
|  |  |
|  | weights{L} = weights{L} - learningRate\* vdW\_corrected{L} ./ d1; |
|  | biases{L} = biases{L} -learningRate\* vdB\_corrected{L} ./ d2; |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights and biases after each iteration |
|  | function [weights biases costs] = L\_Layer\_DeepModel(X, Y, layersDimensions, hiddenActivationFunc='relu', |
|  | outputActivationFunc="sigmoid",learning\_rate = .3, lambd=0, keep\_prob=1, num\_iterations = 10000,initType="default")#lr was 0.009 |
|  |  |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  | if (strcmp(initType,"He")) |
|  | # He Initialization |
|  | [weights biases] = HeInitializeDeepModel(layersDimensions); |
|  | elseif (strcmp(initType,"Xav")) |
|  | # Xavier Initialization |
|  | [weights biases] = XavInitializeDeepModel(layersDimensions); |
|  | else |
|  | # Default initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  | endif |
|  |  |
|  | # Loop (gradient descent) |
|  | for i = 0:num\_iterations |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID. |
|  | [AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X, weights, biases,keep\_prob, hiddenActivationFunc, outputActivationFunc=outputActivationFunc); |
|  |  |
|  | # Regularization parameter is 0 |
|  | if (lambd==0) |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | else |
|  | # Compute cost with regularization |
|  | cost = computeCostWithReg(weights, AL, Y, lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | endif |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches, droputMat, lambd, keep\_prob, hiddenActivationFunc, outputActivationFunc=outputActivationFunc, |
|  | numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learning\_rate,outputActivationFunc=outputActivationFunc); |
|  |  |
|  |  |
|  | # Print the cost every 1000 iterations |
|  | if ( mod(i,1000) == 0) |
|  | costs =[costs cost]; |
|  | #disp ("Cost after iteration"), L2RegularizationCost(i),disp(cost); |
|  | printf("Cost after iteration i=%i cost=%d\n",i,cost); |
|  | endif |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  | # Execute a L layer Deep learning model with Stochastic Gradient descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : mini\_batch\_size |
|  | # : num of epochs |
|  | #output : Updated weights and biases after each iteration |
|  | function [weights biases costs] = L\_Layer\_DeepModel\_SGD(X, Y, layersDimensions, hiddenActivationFunc='relu', |
|  | outputActivationFunc="sigmoid",learningRate = .3, |
|  | lrDecay=false,decayRate=1, |
|  | lambd=0, keep\_prob=1, |
|  | optimizer="gd", beta=0.9, beta1=0.9, beta2=0.999,epsilon=10^-8, |
|  | mini\_batch\_size = 64, num\_epochs = 2500) |
|  |  |
|  | disp("here"); |
|  | printf("learningRate=%f ",learningRate); |
|  | printf("lrDecay=%d ",lrDecay); |
|  | printf("decayRate=%f ",decayRate); |
|  | printf("lamd=%d ",lambd); |
|  | printf("keep\_prob=%f ",keep\_prob); |
|  | printf("optimizer=%s ",optimizer); |
|  | printf("beta=%f ",beta); |
|  | printf("beta1=%f ",beta1); |
|  | printf("beta2=%f ",beta2); |
|  | printf("epsilon=%f ",epsilon); |
|  | printf("mini\_batch\_size=%d ",mini\_batch\_size); |
|  | printf("num\_epochs=%d ",num\_epochs); |
|  | t=0; |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  | # Parameters initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  |  |
|  | if (strcmp(optimizer,"momentum")) |
|  | [vdW vdB] = initializeVelocity(weights, biases); |
|  |  |
|  | elseif(strcmp(optimizer,"rmsprop")) |
|  | [sdW sdB] = initializeRMSProp(weights, biases); |
|  |  |
|  | elseif(strcmp(optimizer,"adam")) |
|  | [vdW vdB sdW sdB] = initializeAdam(weights, biases); |
|  | endif |
|  | seed=10; |
|  | # Loop (gradient descent) |
|  | for i = 0:num\_epochs |
|  | seed = seed + 1; |
|  | [mini\_batches\_X mini\_batches\_Y] = random\_mini\_batches(X, Y, mini\_batch\_size, seed); |
|  |  |
|  | minibatches=length(mini\_batches\_X); |
|  | for batch=1:minibatches |
|  | X=mini\_batches\_X{batch}; |
|  | Y=mini\_batches\_Y{batch}; |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID/SOFTMAX. |
|  | [AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X, weights, biases, keep\_prob,hiddenActivationFunc, outputActivationFunc=outputActivationFunc); |
|  | #disp(batch); |
|  | #disp(size(X)); |
|  | #disp(size(Y)); |
|  | if (lambd==0) |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | else |
|  | # Compute cost with regularization |
|  | cost = computeCostWithReg(weights, AL, Y, lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | endif |
|  | #disp(cost); |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches, droputMat, lambd, keep\_prob, hiddenActivationFunc, outputActivationFunc=outputActivationFunc, |
|  | numClasses=layersDimensions(size(layersDimensions)(2))); |
|  |  |
|  | if (strcmp(optimizer,"gd")) |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learningRate,outputActivationFunc=outputActivationFunc); |
|  | elseif (strcmp(optimizer,"momentum")) |
|  | [weights biases] = gradientDescentWithMomentum(weights, biases,gradsDW,gradsDB, vdW, vdB, beta, learningRate,outputActivationFunc); |
|  | elseif (strcmp(optimizer,"rmsprop")) |
|  | [weights biases] = gradientDescentWithRMSProp(weights, biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon, learningRate,outputActivationFunc); |
|  |  |
|  | elseif (strcmp(optimizer,"adam")) |
|  | t=t+1; |
|  | [weights biases] = gradientDescentWithAdam(weights, biases,gradsDW,gradsDB,vdW, vdB, sdW, sdB, t, beta1, beta2, epsilon, learningRate,outputActivationFunc); |
|  | endif |
|  | endfor |
|  | # Print the cost every 1000 iterations |
|  | if ( mod(i,1000) == 0) |
|  | costs =[costs cost]; |
|  | #disp ("Cost after iteration"), disp(i),disp(cost); |
|  | printf("Cost after iteration i=%i cost=%d\n",i,cost); |
|  | endif |
|  | if(lrDecay==true) |
|  | learningRate=decayRate^(num\_epochs/1000)\*learningRate; |
|  | endif |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  |  |
|  | function plotCostVsIterations(maxIterations,costs,fig1) |
|  | iterations=[0:1000:maxIterations]; |
|  | plot(iterations,costs); |
|  | title ("Cost vs no of iterations "); |
|  | xlabel("No of iterations"); |
|  | ylabel("Cost"); |
|  | print -dpng figReg2-o |
|  | end; |
|  |  |
|  | function plotCostVsEpochs(maxEpochs,costs,fig1) |
|  | epochs=[0:1000:maxEpochs]; |
|  | plot(epochs,costs); |
|  | title ("Cost vs no of epochs "); |
|  | xlabel("No of epochs"); |
|  | ylabel("Cost"); |
|  | print -dpng fig5-o |
|  | end; |
|  |  |
|  | # Compute the predicted value for a given input |
|  | # Input : Neural Network parameters |
|  | # : Input data |
|  | function [predictions]= predict(weights, biases, X,keep\_prob=1,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,keep\_prob,hiddenActivationFunc); |
|  | predictions = (AL>0.5); |
|  | end |
|  |  |
|  | # Plot the decision boundary |
|  | function plotDecisionBoundary(data,weights, biases,keep\_prob=1,hiddenActivationFunc="relu",fig2) |
|  | %Plot a non-linear decision boundary learned by the SVM |
|  | colormap ("summer"); |
|  |  |
|  | % Make classification predictions over a grid of values |
|  | x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)'; |
|  | x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)'; |
|  | [X1, X2] = meshgrid(x1plot, x2plot); |
|  | vals = zeros(size(X1)); |
|  | # Plot the prediction for the grid |
|  | for i = 1:size(X1, 2) |
|  | gridPoints = [X1(:, i), X2(:, i)]; |
|  | vals(:, i)=predict(weights, biases,gridPoints',keep\_prob, hiddenActivationFunc=hiddenActivationFunc); |
|  | endfor |
|  |  |
|  | scatter(data(:,1),data(:,2),8,c=data(:,3),"filled"); |
|  | % Plot the boundary |
|  | hold on |
|  | #contour(X1, X2, vals, [0 0], 'LineWidth', 2); |
|  | contour(X1, X2, vals,"linewidth",4); |
|  | title ({"3 layer Neural Network decision boundary"}); |
|  | hold off; |
|  | print -dpng figReg22-o |
|  |  |
|  | end |
|  |  |
|  | function [AL]= scores(weights, biases, X,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc); |
|  | end |
|  |  |
|  | # Create Random mini batches. Return cell arrays with the mini batches |
|  | # Input : X, Y |
|  | # : Size of minibatch |
|  | #Output : mini batches X & Y |
|  | function [mini\_batches\_X mini\_batches\_Y]= random\_mini\_batches(X, Y, miniBatchSize = 64, seed = 0) |
|  |  |
|  | rand ("seed", seed); |
|  | # Get number of training samples |
|  | m = size(X)(2); |
|  |  |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = randperm(m); |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X(:, permutation); |
|  | shuffled\_Y = Y(:, permutation); |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = floor(m/miniBatchSize); |
|  | batch=0; |
|  | for k = 0:(numCompleteMinibatches-1) |
|  | #Set the start and end of each mini batch |
|  | batch=batch+1; |
|  | lower=(k\*miniBatchSize)+1; |
|  | upper=(k+1) \* miniBatchSize; |
|  | mini\_batch\_X = shuffled\_X(:, lower:upper); |
|  | mini\_batch\_Y = shuffled\_Y(:, lower:upper); |
|  |  |
|  | # Create cell arrays |
|  | mini\_batches\_X{batch} = mini\_batch\_X; |
|  | mini\_batches\_Y{batch} = mini\_batch\_Y; |
|  | endfor |
|  |  |
|  | # If the batc size does not cleanly divide with number of mini batches |
|  | if mod(m ,miniBatchSize) != 0 |
|  | # Set the start and end of the last mini batch |
|  | l=floor(m/miniBatchSize)\*miniBatchSize; |
|  | m=l+ mod(m,miniBatchSize); |
|  | mini\_batch\_X = shuffled\_X(:,(l+1):m); |
|  | mini\_batch\_Y = shuffled\_Y(:,(l+1):m); |
|  |  |
|  | batch=batch+1; |
|  | mini\_batches\_X{batch} = mini\_batch\_X; |
|  | mini\_batches\_Y{batch} = mini\_batch\_Y; |
|  | endif |
|  | end |
|  |  |
|  | function plotDecisionBoundary1( data,weights, biases,keep\_prob=1, hiddenActivationFunc="relu") |
|  | % Make classification predictions over a grid of values |
|  | x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)'; |
|  | x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)'; |
|  | [X1, X2] = meshgrid(x1plot, x2plot); |
|  | vals = zeros(size(X1)); |
|  | for i = 1:size(X1, 2) |
|  | gridPoints = [X1(:, i), X2(:, i)]; |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(gridPoints', weights, biases,keep\_prob,hiddenActivationFunc, outputActivationFunc="softmax"); |
|  | [l m] = max(AL, [ ], 2); |
|  | vals(:, i)= m; |
|  | endfor |
|  |  |
|  | scatter(data(:,1),data(:,2),8,c=data(:,3),"filled"); |
|  | % Plot the boundary |
|  | hold on |
|  | contour(X1, X2, vals,"linewidth",4); |
|  | print -dpng "fig-o1.png" |
|  | end |
|  |  |
|  | function [vec] = cellArray\_to\_vector(weights,biases) |
|  | vec=[]; |
|  | for i = 1: size(weights)(2) |
|  | w= weights{i}; |
|  | sz=size(w); |
|  | # Take transpose before reshaping |
|  | l=reshape(w',sz(1)\*sz(2),1); |
|  |  |
|  | b=biases{i}; |
|  | sz1=size(b); |
|  | m=reshape(b',sz1(1)\*sz1(2),1); |
|  | #Concatenate |
|  | vec=[vec;l;m]; |
|  | endfor |
|  | end |
|  |  |
|  | function [vec] = gradients\_to\_vector(gradsDW,gradsDB) |
|  | vec=[]; |
|  | for i = 1: size(gradsDW)(2) |
|  | gW= gradsDW{i}; |
|  | sz=size(gW); |
|  | # Take transpose before reshaping |
|  | l=reshape(gW',sz(1)\*sz(2),1); |
|  |  |
|  | gB=gradsDB{i}; |
|  | sz1=size(gB); |
|  | m=reshape(gB',sz1(1)\*sz1(2),1); |
|  | #Concatenate |
|  | vec=[vec;l;m]; |
|  | endfor |
|  | end |
|  |  |
|  | function [weights1 biases1] = vector\_to\_cellArray(weights, biases,params) |
|  | vec=[]; |
|  | weights1 = {}; |
|  | biases1 ={}; |
|  | start=1; |
|  | for i = 1: size(weights)(2) |
|  | w= weights{i}; |
|  | sz=size(w); |
|  | # Take transpose before reshaping |
|  | a = params(start:start+sz(1)\*sz(2)-1,1); |
|  | b = reshape(a,sz(2),sz(1)); |
|  | weights1{i}= b'; |
|  | start=start+sz(1)\*sz(2); |
|  | b=biases{i}; |
|  | sz=size(b); |
|  | c = params(start:start+sz(1)\*sz(2)-1,1); |
|  | d = reshape(c,sz(2),sz(1)); |
|  | biases1{i}= d'; |
|  | start=start+sz(1)\*sz(2); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  | function [weights1 biases1] = vector\_to\_cellArray1(weights, biases,gradients) |
|  | vec=[]; |
|  | weights1 = {}; |
|  | biases1 ={}; |
|  | start=1; |
|  | for i = 1: size(weights)(2) |
|  | w= weights{i}; |
|  | sz=size(w); |
|  | # Take transpose before reshaping |
|  | a = grads(start:start+sz(1)\*sz(2)-1,1); |
|  | b = reshape(a,sz(2),sz(1)); |
|  | weights1{i}= b'; |
|  | start=start+sz(1)\*sz(2); |
|  | b=biases{i}; |
|  | sz=size(b); |
|  | c = grads(start:start+sz(1)\*sz(2)-1,1); |
|  | d = reshape(c,sz(2),sz(1)); |
|  | biases1{i}= d'; |
|  | start=start+sz(1)\*sz(2); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  | function [difference]= gradient\_check\_n(weights,biases,gradsDW,gradsDB, X, Y, epsilon = 1e-7,outputActivationFunc="sigmoid",numClasses) |
|  | # Convert cell array to vector |
|  | parameters\_values = cellArray\_to\_vector(weights, biases); |
|  | # Convert gradient cell array to vector |
|  | grad = gradients\_to\_vector(gradsDW,gradsDB); |
|  | num\_parameters = size(parameters\_values)(1); |
|  | #Initialize |
|  | J\_plus = zeros(num\_parameters, 1); |
|  | J\_minus = zeros(num\_parameters, 1); |
|  | gradapprox = zeros(num\_parameters, 1); |
|  |  |
|  |  |
|  | # Compute gradapprox |
|  | for i = 1:num\_parameters |
|  | # Compute J\_plus[i]. Inputs: "parameters\_values, epsilon". Output = "J\_plus[i]". |
|  | thetaplus = parameters\_values; |
|  | thetaplus(i,1) = thetaplus(i,1) + epsilon; |
|  | [weights1 biases1] =vector\_to\_cellArray(weights, biases,thetaplus); |
|  | [AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X', weights1, biases1, keep\_prob=1, |
|  | hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc); |
|  | J\_plus(i) = computeCost(AL, Y', outputActivationFunc=outputActivationFunc,numClasses); |
|  |  |
|  |  |
|  | # Compute J\_minus[i]. Inputs: "parameters\_values, epsilon". Output = "J\_minus[i]". |
|  | thetaminus = parameters\_values; |
|  | thetaminus(i,1) = thetaminus(i,1) - epsilon ; |
|  | [weights1 biases1] = vector\_to\_cellArray(weights, biases,thetaminus); |
|  | [AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X',weights1, biases1, keep\_prob=1, |
|  | hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc); |
|  | J\_minus(i) = computeCost(AL, Y', outputActivationFunc=outputActivationFunc,numClasses); |
|  |  |
|  | # Compute gradapprox[i] |
|  | gradapprox(i) = (J\_plus(i) - J\_minus(i))/(2\*epsilon); |
|  |  |
|  | endfor |
|  |  |
|  | # Compute L2Norm |
|  | numerator = L2NormVec(grad-gradapprox); |
|  | denominator = L2NormVec(grad) + L2NormVec(gradapprox); |
|  | difference = numerator/denominator; |
|  | ; |
|  | if difference > 1e-04 |
|  | printf("There is a mistake in the implementation "); |
|  |  |
|  | disp(difference); |
|  | else |
|  | printf("The implementation works perfectly"); |
|  | disp(difference); |
|  | endif |
|  | # This can be used to compare the gradients from backprop and gradapprox |
|  | [weights1 biases1] = vector\_to\_cellArray(weights, biases,grad); |
|  | printf("Gradients from back propagation"); |
|  | disp(weights1); |
|  | disp(biases1); |
|  | [weights2 biases2] = vector\_to\_cellArray(weights, biases,gradapprox); |
|  | printf("Gradients from gradient check"); |
|  | disp(weights2); |
|  | disp(biases2); |
|  |  |
|  | end |
|  |  |
|  | # Compute L2Norm |
|  | function [l2norm] = L2NormVec(x) |
|  | l2norm=sqrt(sum(x .^ 2)); |
|  | end |

DLFunctions8.R

|  |
| --- |
| library(ggplot2) |
|  | library(PRROC) |
|  | library(dplyr) |
|  |  |
|  | # Compute the sigmoid of a vector |
|  | sigmoid <- function(Z){ |
|  | A <- 1/(1+ exp(-Z)) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  |  |
|  | } |
|  |  |
|  | # This is the older version. Very performance intensive |
|  | reluOld <-function(Z){ |
|  | A <- apply(Z, 1:2, function(x) max(0,x)) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the Relu of a vector |
|  | relu <-function(Z){ |
|  | # Perform relu. Set values less that equal to 0 as 0 |
|  | Z[Z<0]=0 |
|  | A=Z |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the tanh activation of a vector |
|  | tanhActivation <- function(Z){ |
|  | A <- tanh(Z) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | softmax <- function(Z){ |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(t(Z)) |
|  | # normalize them for each example |
|  | A = exp\_scores / rowSums(exp\_scores) |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the detivative of Relu |
|  | # g'(z) = 1 if z >0 and 0 otherwise |
|  | reluDerivative <-function(dA, cache){ |
|  | Z <- cache |
|  | dZ <- dA |
|  | # Create a logical matrix of values > 0 |
|  | a <- Z > 0 |
|  | # When z <= 0, you should set dz to 0 as well. Perform an element wise multiply |
|  | dZ <- dZ \* a |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Compute the derivative of sigmoid |
|  | # Derivative g'(z) = a\* (1-a) |
|  | sigmoidDerivative <- function(dA, cache){ |
|  | Z <- cache |
|  | s <- 1/(1+exp(-Z)) |
|  | dZ <- dA \* s \* (1-s) |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Compute the derivative of tanh |
|  | # Derivative g'(z) = 1- a^2 |
|  | tanhDerivative <- function(dA, cache){ |
|  | Z = cache |
|  | a = tanh(Z) |
|  | dZ = dA \* (1 - a^2) |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Populate a matrix of 1s in rows where Y==1 |
|  | # This may need to be extended for K classes. Currently |
|  | # supports K=3 & K=10 |
|  | popMatrix <- function(Y,numClasses){ |
|  | a=rep(0,times=length(Y)) |
|  | Y1=matrix(a,nrow=length(Y),ncol=numClasses) |
|  | #Set the rows and columns as 1's where Y is the class value |
|  | if(numClasses==3){ |
|  | Y1[Y==0,1]=1 |
|  | Y1[Y==1,2]=1 |
|  | Y1[Y==2,3]=1 |
|  | } else if (numClasses==10){ |
|  | Y1[Y==0,1]=1 |
|  | Y1[Y==1,2]=1 |
|  | Y1[Y==2,3]=1 |
|  | Y1[Y==3,4]=1 |
|  | Y1[Y==4,5]=1 |
|  | Y1[Y==5,6]=1 |
|  | Y1[Y==6,7]=1 |
|  | Y1[Y==7,8]=1 |
|  | Y1[Y==8,9]=1 |
|  | Y1[Y==9,0]=1 |
|  | } |
|  | return(Y1) |
|  | } |
|  |  |
|  | softmaxDerivative <- function(dA, cache ,y,numTraining,numClasses){ |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z <- cache |
|  | # Compute softmax |
|  | exp\_scores = exp(t(Z)) |
|  | # normalize them for each example |
|  | probs = exp\_scores / rowSums(exp\_scores) |
|  | # Create a matrix of zeros |
|  | Y1=popMatrix(y,numClasses) |
|  | #a=rep(0,times=length(Y)) |
|  | #Y1=matrix(a,nrow=length(Y),ncol=numClasses) |
|  | #Set the rows and columns as 1's where Y is the class value |
|  | dZ = probs-Y1 |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | initializeDeepModel <- function(layerDimensions){ |
|  | set.seed(2) |
|  |  |
|  | # Initialize empty list |
|  | layerParams <- list() |
|  |  |
|  | # Note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Indices in R start from 1 |
|  | for(l in 2:length(layersDimensions)){ |
|  | # Initialize a matrix of small random numbers of size l x l-1 |
|  | # Create random numbers of size l x l-1 |
|  | w=rnorm(layersDimensions[l]\*layersDimensions[l-1])\*0.01 |
|  | # Create a weight matrix of size l x l-1 with this initial weights and |
|  | # Add to list W1,W2... WL |
|  | layerParams[[paste('W',l-1,sep="")]] = matrix(w,nrow=layersDimensions[l], |
|  | ncol=layersDimensions[l-1]) |
|  | layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[l]), |
|  | nrow=layersDimensions[l],ncol=1) |
|  | } |
|  | return(layerParams) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | # He Initialization model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | # He initilization multiplies the random numbers with sqrt(2/layerDimensions[previouslayer]) |
|  | HeInitializeDeepModel <- function(layerDimensions){ |
|  | set.seed(2) |
|  |  |
|  | # Initialize empty list |
|  | layerParams <- list() |
|  |  |
|  | # Note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Indices in R start from 1 |
|  | for(l in 2:length(layersDimensions)){ |
|  | # Initialize a matrix of small random numbers of size l x l-1 |
|  | # Create random numbers of size l x l-1 |
|  | w=rnorm(layersDimensions[l]\*layersDimensions[l-1]) |
|  |  |
|  | # Create a weight matrix of size l x l-1 with this initial weights and |
|  | # Add to list W1,W2... WL |
|  | # He initialization - Divide by sqrt(2/layerDimensions[previous layer]) |
|  | layerParams[[paste('W',l-1,sep="")]] = matrix(w,nrow=layersDimensions[l], |
|  | ncol=layersDimensions[l-1])\*sqrt(2/layersDimensions[l-1]) |
|  | layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[l]), |
|  | nrow=layersDimensions[l],ncol=1) |
|  | } |
|  | return(layerParams) |
|  | } |
|  |  |
|  | # XavInitializeDeepModel Initialization model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | # He initilization multiplies the random numbers with sqrt(1/layerDimensions[previouslayer]) |
|  | XavInitializeDeepModel <- function(layerDimensions){ |
|  | set.seed(2) |
|  |  |
|  | # Initialize empty list |
|  | layerParams <- list() |
|  |  |
|  | # Note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Indices in R start from 1 |
|  | for(l in 2:length(layersDimensions)){ |
|  | # Initialize a matrix of small random numbers of size l x l-1 |
|  | # Create random numbers of size l x l-1 |
|  | w=rnorm(layersDimensions[l]\*layersDimensions[l-1]) |
|  |  |
|  | # Create a weight matrix of size l x l-1 with this initial weights and |
|  | # Add to list W1,W2... WL |
|  | # He initialization - Divide by sqrt(2/layerDimensions[previous layer]) |
|  | layerParams[[paste('W',l-1,sep="")]] = matrix(w,nrow=layersDimensions[l], |
|  | ncol=layersDimensions[l-1])\*sqrt(1/layersDimensions[l-1]) |
|  | layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[l]), |
|  | nrow=layersDimensions[l],ncol=1) |
|  | } |
|  | return(layerParams) |
|  | } |
|  |  |
|  | # Initialize velocity |
|  | # Input : parameters |
|  | # Returns: Initial velocity v |
|  | initializeVelocity <- function(parameters){ |
|  |  |
|  | L <- length(parameters)/2 |
|  | v <- list() |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for(l in 1:L){ |
|  | # Get the size of weight matrix |
|  | sz <- dim(parameters[[paste('W',l,sep="")]]) |
|  | v[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | #Get the size of bias matrix |
|  | sz <- dim(parameters[[paste('b',l,sep="")]]) |
|  | v[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | } |
|  |  |
|  | return(v) |
|  | } |
|  |  |
|  |  |
|  | # Initialize RMSProp |
|  | # Input : parameters |
|  | # Returns: Initial RMSProp s |
|  | initializeRMSProp <- function(parameters){ |
|  |  |
|  | L <- length(parameters)/2 |
|  | s <- list() |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for(l in 1:L){ |
|  | # Get the size of weight matrix |
|  | sz <- dim(parameters[[paste('W',l,sep="")]]) |
|  | s[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | #Get the size of bias matrix |
|  | sz <- dim(parameters[[paste('b',l,sep="")]]) |
|  | s[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | } |
|  |  |
|  | return(s) |
|  | } |
|  |  |
|  | # Initialize Adam |
|  | # Input : parameters |
|  | # Returns: Initial RMSProp s |
|  | initializeAdam <- function(parameters){ |
|  |  |
|  | L <- length(parameters)/2 |
|  | v <- list() |
|  | s <- list() |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for(l in 1:L){ |
|  | # Get the size of weight matrix |
|  | sz <- dim(parameters[[paste('W',l,sep="")]]) |
|  | v[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | s[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | #Get the size of bias matrix |
|  | sz <- dim(parameters[[paste('b',l,sep="")]]) |
|  | v[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | s[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | } |
|  | retvals <- list("v"=v,"s"=s) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | layerActivationForward <- function(A\_prev, W, b, activationFunc){ |
|  |  |
|  | # Compute Z |
|  | z = W %\*% A\_prev |
|  | # Broadcast the bias 'b' by column |
|  | Z <-sweep(z,1,b,'+') |
|  |  |
|  | forward\_cache <- list("A\_prev"=A\_prev, "W"=W, "b"=b) |
|  | # Compute the activation for sigmoid |
|  | if(activationFunc == "sigmoid"){ |
|  | vals = sigmoid(Z) |
|  | } else if (activationFunc == "relu"){ # Compute the activation for relu |
|  | vals = relu(Z) |
|  | } else if(activationFunc == 'tanh'){ # Compute the activation for tanh |
|  | vals = tanhActivation(Z) |
|  | } else if(activationFunc == 'softmax'){ |
|  | vals = softmax(Z) |
|  | } |
|  | # Create a list of forward and activation cache |
|  | cache <- list("forward\_cache"=forward\_cache, "activation\_cache"=vals[['Z']]) |
|  | retvals <- list("A"=vals[['A']],"cache"=cache) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - elu/sigmoid/tanh |
|  | # outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | forwardPropagationDeep <- function(X, parameters,keep\_prob=1, hiddenActivationFunc='relu', |
|  | outputActivationFunc='sigmoid'){ |
|  | caches <- list() |
|  | dropoutMat <- list() |
|  | # Set A to X (A0) |
|  | A <- X |
|  | L <- length(parameters)/2 # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for(l in 1:(L-1)){ |
|  | A\_prev <- A |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | # Set W and b for layer 'l' |
|  | # Loop throug from W1,W2... WL-1 |
|  | W <- parameters[[paste("W",l,sep="")]] |
|  | b <- parameters[[paste("b",l,sep="")]] |
|  | # Compute the forward propagation through layer 'l' using the activation function |
|  | actForward <- layerActivationForward(A\_prev, |
|  | W, |
|  | b, |
|  | activationFunc = hiddenActivationFunc) |
|  | A <- actForward[['A']] |
|  | # Append the cache A\_prev,W,b, Z |
|  | caches[[l]] <-actForward |
|  |  |
|  | # Randomly drop some activation units |
|  | # Create a matrix as the same shape as A |
|  | set.seed(1) |
|  | i=dim(A)[1] |
|  | j=dim(A)[2] |
|  | a<-rnorm(i\*j) |
|  | # Normalize a between 0 and 1 |
|  | a = (a - min(a))/(max(a) - min(a)) |
|  | # Create a matrix of D |
|  | D <- matrix(a,nrow=i, ncol=j) |
|  | # Find D which is less than equal to keep\_prob |
|  | D <- D < keep\_prob |
|  | # Remove some A's |
|  | A <- A \* D |
|  | # Divide by keep\_prob to keep expected value same |
|  | A <- A/keep\_prob |
|  | dropoutMat[[paste("D",l,sep="")]] <- D |
|  | } |
|  |  |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | # Set the weights and biases for the last layer |
|  | W <- parameters[[paste("W",L,sep="")]] |
|  | b <- parameters[[paste("b",L,sep="")]] |
|  | # Last layer |
|  | actForward = layerActivationForward(A, W, b, activationFunc = outputActivationFunc) |
|  | AL <- actForward[['A']] |
|  | # Append the output of this forward propagation through the last layer |
|  | caches[[L]] <- actForward |
|  | # Create a list of the final output and the caches |
|  | fwdPropDeep <- list("AL"=AL,"caches"=caches,"dropoutMat"=dropoutMat) |
|  | return(fwdPropDeep) |
|  |  |
|  | } |
|  |  |
|  | pickColumns <- function(AL,Y,numClasses){ |
|  | if(numClasses==3){ |
|  | a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | } |
|  | else if (numClasses==10){ |
|  | a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3],AL[Y==3,4],AL[Y==4,5], |
|  | AL[Y==5,6],AL[Y==6,7],AL[Y==7,8],AL[Y==8,9],AL[Y==9,10]) |
|  | } |
|  | return(a) |
|  | } |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # :outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | computeCost <- function(AL,Y,outputActivationFunc="sigmoid",numClasses=3){ |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | m= length(Y) |
|  | cost=-1/m\*sum(Y\*log(AL) + (1-Y)\*log(1-AL)) |
|  |  |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | # Select the elements where the y values are 0, 1 or 2 and make a vector |
|  | # Pick columns |
|  | #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | m= length(Y) |
|  | a =pickColumns(AL,Y,numClasses) |
|  | #a = c(A2[y=k,k+1]) |
|  | # Take log |
|  | correct\_probs = -log(a) |
|  |  |
|  | # Compute loss |
|  | cost= sum(correct\_probs)/m |
|  | } |
|  | return(cost) |
|  | } |
|  |  |
|  |  |
|  | # Compute the cost with Regularization |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # :outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | computeCostWithReg <- function(parameters, AL,Y,lambd, outputActivationFunc="sigmoid",numClasses=3){ |
|  |  |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | m= length(Y) |
|  | cost=-1/m\*sum(Y\*log(AL) + (1-Y)\*log(1-AL)) |
|  |  |
|  | # Regularization cost |
|  | L <- length(parameters)/2 |
|  | L2RegularizationCost=0 |
|  | for(l in 1:L){ |
|  | L2RegularizationCost = L2RegularizationCost + |
|  | sum(parameters[[paste("W",l,sep="")]]^2) |
|  | } |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | # Select the elements where the y values are 0, 1 or 2 and make a vector |
|  | # Pick columns |
|  | #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | m= length(Y) |
|  | a =pickColumns(AL,Y,numClasses) |
|  | #a = c(A2[y=k,k+1]) |
|  | # Take log |
|  | correct\_probs = -log(a) |
|  | # Compute loss |
|  | cost= sum(correct\_probs)/m |
|  |  |
|  | # Regularization cost |
|  | L <- length(parameters)/2 |
|  | L2RegularizationCost=0 |
|  | # Add L2 norm |
|  | for(l in 1:L){ |
|  | L2RegularizationCost = L2RegularizationCost + |
|  | sum(parameters[[paste("W",l,sep="")]]^2) |
|  | } |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  | } |
|  | return(cost) |
|  | } |
|  |  |
|  | # Compute the backpropagation through a layer |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # activationFunc |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  |  |
|  | layerActivationBackward <- function(dA, cache, Y, activationFunc,numClasses){ |
|  | # Get A\_prev,W,b |
|  | forward\_cache <-cache[['forward\_cache']] |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | A\_prev <- forward\_cache[['A\_prev']] |
|  | numtraining = dim(A\_prev)[2] |
|  | # Get Z |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | if(activationFunc == "relu"){ |
|  | dZ <- reluDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "sigmoid"){ |
|  | dZ <- sigmoidDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "tanh"){ |
|  | dZ <- tanhDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "softmax"){ |
|  | dZ <- softmaxDerivative(dA, activation\_cache,Y,numtraining,numClasses) |
|  | } |
|  |  |
|  | if (activationFunc == 'softmax'){ |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | dW = 1/numtraining \* A\_prev%\*%dZ |
|  | db = 1/numtraining\* matrix(colSums(dZ),nrow=1,ncol=numClasses) |
|  | dA\_prev = dZ %\*%W |
|  | } else { |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | numtraining = dim(A\_prev)[2] |
|  | dW = 1/numtraining \* dZ %\*% t(A\_prev) |
|  | db = 1/numtraining \* rowSums(dZ) |
|  | dA\_prev = t(W) %\*% dZ |
|  | } |
|  | retvals <- list("dA\_prev"=dA\_prev,"dW"=dW,"db"=db) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the backpropagation through a layer with Regularization |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # activationFunc |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  |  |
|  | layerActivationBackwardWithReg <- function(dA, cache, Y, lambd, activationFunc,numClasses){ |
|  | # Get A\_prev,W,b |
|  | forward\_cache <-cache[['forward\_cache']] |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | A\_prev <- forward\_cache[['A\_prev']] |
|  | numtraining = dim(A\_prev)[2] |
|  | # Get Z |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | if(activationFunc == "relu"){ |
|  | dZ <- reluDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "sigmoid"){ |
|  | dZ <- sigmoidDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "tanh"){ |
|  | dZ <- tanhDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "softmax"){ |
|  | dZ <- softmaxDerivative(dA, activation\_cache,Y,numtraining,numClasses) |
|  | } |
|  |  |
|  | if (activationFunc == 'softmax'){ |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \* A\_prev%\*%dZ + (lambd/numtraining) \* t(W) |
|  | db = 1/numtraining\* matrix(colSums(dZ),nrow=1,ncol=numClasses) |
|  | dA\_prev = dZ %\*%W |
|  | } else { |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | numtraining = dim(A\_prev)[2] |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \* dZ %\*% t(A\_prev) + (lambd/numtraining) \* W |
|  | db = 1/numtraining \* rowSums(dZ) |
|  | dA\_prev = t(W) %\*% dZ |
|  | } |
|  | retvals <- list("dA\_prev"=dA\_prev,"dW"=dW,"db"=db) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the backpropagation for 1 cycle through all layers |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers - relu/tanh/sigmoid |
|  | # outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # |
|  | backwardPropagationDeep <- function(AL, Y, caches,dropoutMat, lambd=0, keep\_prob=0, hiddenActivationFunc='relu', |
|  | outputActivationFunc="sigmoid",numClasses){ |
|  | #initialize the gradients |
|  | gradients = list() |
|  | # Set the number of layers |
|  | L = length(caches) |
|  | numTraining = dim(AL)[2] |
|  |  |
|  | if(outputActivationFunc == "sigmoid") |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a) - ((1-y)/(1-a)) - At the output layer |
|  | dAL = -( (Y/AL) -(1 - Y)/(1 - AL)) |
|  | else if(outputActivationFunc == "softmax"){ |
|  | dAL=0 |
|  | Y=t(Y) |
|  | } |
|  |  |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | # Start with Layer L |
|  | # Get the current cache |
|  | current\_cache = caches[[L]]$cache |
|  | if (lambd==0){ |
|  | retvals <- layerActivationBackward(dAL, current\_cache, Y, |
|  | activationFunc = outputActivationFunc,numClasses) |
|  | } else { |
|  | retvals = layerActivationBackwardWithReg(dAL, current\_cache, Y, lambd, |
|  | activationFunc = outputActivationFunc,numClasses) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | #Note: Take the transpose of dA |
|  | if(outputActivationFunc =="sigmoid") |
|  | gradients[[paste("dA",L,sep="")]] <- retvals[['dA\_prev']] |
|  | else if(outputActivationFunc =="softmax") |
|  | gradients[[paste("dA",L,sep="")]] <- t(retvals[['dA\_prev']]) |
|  |  |
|  | gradients[[paste("dW",L,sep="")]] <- retvals[['dW']] |
|  | gradients[[paste("db",L,sep="")]] <- retvals[['db']] |
|  |  |
|  |  |
|  |  |
|  | # Traverse in the reverse direction |
|  | for(l in (L-1):1){ |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | current\_cache = caches[[l]]$cache |
|  | if (lambd==0){ |
|  | cat("l=",l) |
|  | # Get the dropout matrix |
|  | D <-dropoutMat[[paste("D",l,sep="")]] |
|  | # Multiply gradient with dropout matrix |
|  | gradients[[paste('dA',l+1,sep="")]] = gradients[[paste('dA',l+1,sep="")]] \*D |
|  | # Divide by keep\_prob to keep expected value same |
|  | gradients[[paste('dA',l+1,sep="")]] = gradients[[paste('dA',l+1,sep="")]]/keep\_prob |
|  | retvals = layerActivationBackward(gradients[[paste('dA',l+1,sep="")]], |
|  | current\_cache, Y, |
|  | activationFunc = hiddenActivationFunc) |
|  | } else { |
|  | retvals = layerActivationBackwardWithReg(gradients[[paste('dA',l+1,sep="")]], |
|  | current\_cache, Y, lambd, |
|  | activationFunc = hiddenActivationFunc) |
|  | } |
|  |  |
|  | gradients[[paste("dA",l,sep="")]] <-retvals[['dA\_prev']] |
|  | gradients[[paste("dW",l,sep="")]] <- retvals[['dW']] |
|  | gradients[[paste("db",l,sep="")]] <- retvals[['db']] |
|  | } |
|  |  |
|  |  |
|  |  |
|  | return(gradients) |
|  | } |
|  |  |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescent <- function(parameters, gradients, learningRate,outputActivationFunc="sigmoid"){ |
|  |  |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate\* gradients[[paste("dW",l,sep="")]] |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\* gradients[[paste("db",l,sep="")]] |
|  | } |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* gradients[[paste("dW",L,sep="")]] |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* gradients[[paste("db",L,sep="")]] |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("dW",L,sep="")]]) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("db",L,sep="")]]) |
|  |  |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  | # Perform Gradient Descent with momentum |
|  | # Input : Weights and biases |
|  | # : beta |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescentWithMomentum <- function(parameters, gradients,v, beta, learningRate,outputActivationFunc="sigmoid"){ |
|  |  |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  |  |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | # Compute velocities |
|  | # v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk |
|  | v[[paste("dW",l, sep="")]] = beta\*v[[paste("dW",l, sep="")]] + |
|  | (1-beta) \* gradients[[paste('dW',l,sep="")]] |
|  | v[[paste("db",l, sep="")]] = beta\*v[[paste("db",l, sep="")]] + |
|  | (1-beta) \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate\* v[[paste("dW",l, sep="")]] |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\* v[[paste("db",l, sep="")]] |
|  | } |
|  |  |
|  | # Compute for the Lth layer |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | v[[paste("dW",L, sep="")]] = beta\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta) \* gradients[[paste('dW',L,sep="")]] |
|  | v[[paste("db",L, sep="")]] = beta\*v[[paste("db",L, sep="")]] + |
|  | (1-beta) \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* v[[paste("dW",l, sep="")]] |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* v[[paste("db",l, sep="")]] |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | v[[paste("dW",L, sep="")]] = beta\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | v[[paste("db",L, sep="")]] = beta\*v[[paste("db",L, sep="")]] + |
|  | (1-beta) \* t(gradients[[paste('db',L,sep="")]]) |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("dW",L,sep="")]]) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("db",L,sep="")]]) |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  |  |
|  | # Perform Gradient Descent with RMSProp |
|  | # Input : Weights and biases |
|  | # : beta1 |
|  | # : epsilon |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescentWithRMSProp <- function(parameters, gradients,s, beta1, epsilon, learningRate,outputActivationFunc="sigmoid"){ |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | # Compute RMSProp |
|  | # s['dWk'] = beta1 \*s['dWk'] + (1-beta1)\*dWk\*\*2/sqrt(s['dWk']) |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",l, sep="")]] = beta1\*s[[paste("dW",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',l,sep="")]] \* gradients[[paste('dW',l,sep="")]] |
|  | s[[paste("db",l, sep="")]] = beta1\*s[[paste("db",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',l,sep="")]] \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate \* gradients[[paste('dW',l,sep="")]]/sqrt(s[[paste("dW",l, sep="")]]+epsilon) |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\*gradients[[paste('db',l,sep="")]]/sqrt(s[[paste("db",l, sep="")]]+epsilon) |
|  | } |
|  |  |
|  | # Compute for the Lth layer |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | s[[paste("dW",L, sep="")]] = beta1\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',L,sep="")]] \*gradients[[paste('dW',L,sep="")]] |
|  | s[[paste("db",L, sep="")]] = beta1\*s[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',L,sep="")]] \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* gradients[[paste('dW',l,sep="")]]/sqrt(s[[paste("dW",L, sep="")]]+epsilon) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* gradients[[paste('db',l,sep="")]]/sqrt( s[[paste("db",L, sep="")]]+epsilon) |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | s[[paste("dW",L, sep="")]] = beta1\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('dW',L,sep="")]]) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | s[[paste("db",L, sep="")]] = beta1\*s[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('db',L,sep="")]]) \* t(gradients[[paste('db',L,sep="")]]) |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("dW",L,sep="")]])/sqrt(s[[paste("dW",L, sep="")]]+epsilon) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("db",L,sep="")]])/sqrt( s[[paste("db",L, sep="")]]+epsilon) |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  |  |
|  | # Perform Gradient Descent with Adam |
|  | # Input : Weights and biases |
|  | # : beta1 |
|  | # : epsilon |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescentWithAdam <- function(parameters, gradients,v, s, t, |
|  | beta1=0.9, beta2=0.999, epsilon=10^-8, learningRate=0.1,outputActivationFunc="sigmoid"){ |
|  |  |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  | v\_corrected <- list() |
|  | s\_corrected <- list() |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | # v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk |
|  | v[[paste("dW",l, sep="")]] = beta1\*v[[paste("dW",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',l,sep="")]] |
|  | v[[paste("db",l, sep="")]] = beta1\*v[[paste("db",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected[[paste("dW",l, sep="")]] = v[[paste("dW",l, sep="")]]/(1-beta1^t) |
|  | v\_corrected[[paste("db",l, sep="")]] = v[[paste("db",l, sep="")]]/(1-beta1^t) |
|  |  |
|  |  |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",l, sep="")]] = beta2\*s[[paste("dW",l, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('dW',l,sep="")]] \* gradients[[paste('dW',l,sep="")]] |
|  | s[[paste("db",l, sep="")]] = beta2\*s[[paste("db",l, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('db',l,sep="")]] \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | s\_corrected[[paste("dW",l, sep="")]] = s[[paste("dW",l, sep="")]]/(1-beta2^t) |
|  | s\_corrected[[paste("db",l, sep="")]] = s[[paste("db",l, sep="")]]/(1-beta2^t) |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(s\_corrected[[paste("dW",l, sep="")]]+epsilon) |
|  | d2=sqrt(s\_corrected[[paste("db",l, sep="")]]+epsilon) |
|  |  |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate \* v\_corrected[[paste("dW",l, sep="")]]/d1 |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\*v\_corrected[[paste("db",l, sep="")]]/d2 |
|  | } |
|  |  |
|  | # Compute for the Lth layer |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | v[[paste("dW",L, sep="")]] = beta1\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',L,sep="")]] |
|  | v[[paste("db",L, sep="")]] = beta1\*v[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected[[paste("dW",L, sep="")]] = v[[paste("dW",L, sep="")]]/(1-beta1^t) |
|  | v\_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-beta1^t) |
|  |  |
|  |  |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",L, sep="")]] = beta2\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('dW',L,sep="")]] \* gradients[[paste('dW',L,sep="")]] |
|  | s[[paste("db",L, sep="")]] = beta2\*s[[paste("db",L, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('db',L,sep="")]] \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | s\_corrected[[paste("dW",L, sep="")]] = s[[paste("dW",L, sep="")]]/(1-beta2^t) |
|  | s\_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-beta2^t) |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(s\_corrected[[paste("dW",L, sep="")]]+epsilon) |
|  | d2=sqrt(s\_corrected[[paste("db",L, sep="")]]+epsilon) |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate \* v\_corrected[[paste("dW",L, sep="")]]/d1 |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\*v\_corrected[[paste("db",L, sep="")]]/d2 |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | v[[paste("dW",L, sep="")]] = beta1\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | v[[paste("db",L, sep="")]] = beta1\*v[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('db',L,sep="")]]) |
|  |  |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected[[paste("dW",L, sep="")]] = v[[paste("dW",L, sep="")]]/(1-beta1^t) |
|  | v\_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-beta1^t) |
|  |  |
|  |  |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",L, sep="")]] = beta2\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta2) \* t(gradients[[paste('dW',L,sep="")]]) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | s[[paste("db",L, sep="")]] = beta2\*s[[paste("db",L, sep="")]] + |
|  | (1-beta2) \* t(gradients[[paste('db',L,sep="")]]) \* t(gradients[[paste('db',L,sep="")]]) |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | s\_corrected[[paste("dW",L, sep="")]] = s[[paste("dW",L, sep="")]]/(1-beta2^t) |
|  | s\_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-beta2^t) |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(s\_corrected[[paste("dW",L, sep="")]]+epsilon) |
|  | d2=sqrt(s\_corrected[[paste("db",L, sep="")]]+epsilon) |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate \* v\_corrected[[paste("dW",L, sep="")]]/d1 |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\*v\_corrected[[paste("db",L, sep="")]]/d2 |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights after each iteration |
|  |  |
|  | L\_Layer\_DeepModel <- function(X, Y, layersDimensions, |
|  | hiddenActivationFunc='relu', |
|  | outputActivationFunc= 'sigmoid', |
|  | learningRate = 0.5, |
|  | lambd=0, |
|  | keep\_prob=1, |
|  | numIterations = 10000, |
|  | initType="default", |
|  | print\_cost=False){ |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  |  |
|  | # Parameters initialization. |
|  | if (initType=="He"){ |
|  | parameters =HeInitializeDeepModel(layersDimensions) |
|  | } else if (initType=="Xav"){ |
|  | parameters =XavInitializeDeepModel(layersDimensions) |
|  | } |
|  | else{ |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | } |
|  |  |
|  |  |
|  | # Loop (gradient descent) |
|  | for( i in 0:numIterations){ |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID/SOFTMAX. |
|  | retvals = forwardPropagationDeep(X, parameters,keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  | dropoutMat <- retvals[['dropoutMat']] |
|  |  |
|  | # Compute cost. |
|  | if(lambd==0){ |
|  | cost <- computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } else { |
|  | cost <- computeCostWithReg(parameters, AL, Y,lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd, keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | if(i%%1000 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | } |
|  |  |
|  | retvals <- list("parameters"=parameters,"costs"=costs) |
|  |  |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Execute a L layer Deep learning model with Stochastic Gradient descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : mini\_batch\_size |
|  | # : num of epochs |
|  | #output : Updated weights after each iteration |
|  | L\_Layer\_DeepModel\_SGD <- function(X, Y, layersDimensions, |
|  | hiddenActivationFunc='relu', |
|  | outputActivationFunc= 'sigmoid', |
|  | learningRate = .3, |
|  | lrDecay=FALSE, |
|  | decayRate=1, |
|  | lambd=0, |
|  | keep\_prob=1, |
|  | optimizer="gd", |
|  | beta=0.9, |
|  | beta1=0.9, |
|  | beta2=0.999, |
|  | epsilon=10^-8, |
|  | mini\_batch\_size = 64, |
|  | num\_epochs = 2500, |
|  | print\_cost=False){ |
|  |  |
|  |  |
|  |  |
|  | print("hello") |
|  | cat("learningRate= ",learningRate) |
|  | cat("\n") |
|  | cat("lambd=",lambd) |
|  | cat("\n") |
|  | cat("keep\_prob=",keep\_prob) |
|  | cat("\n") |
|  | cat("optimizer=",optimizer) |
|  | cat("\n") |
|  | cat("lrDecay=",lrDecay) |
|  | cat("\n") |
|  | cat("decayRate=",decayRate) |
|  | cat("\n") |
|  | cat("beta=",beta) |
|  | cat("\n") |
|  | cat("beta1=",beta1) |
|  | cat("\n") |
|  | cat("beta2=",beta2) |
|  | cat("\n") |
|  | cat("epsilon=",epsilon) |
|  | cat("\n") |
|  | cat("mini\_batch\_size=",mini\_batch\_size) |
|  | cat("\n") |
|  | cat("num\_epochs=",num\_epochs) |
|  | cat("\n") |
|  | set.seed(1) |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  | t <- 0 |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  |  |
|  |  |
|  | #Initialize the optimizer |
|  |  |
|  | if(optimizer == "momentum"){ |
|  | v <-initializeVelocity(parameters) |
|  | } else if(optimizer == "rmsprop"){ |
|  | s <-initializeRMSProp(parameters) |
|  | } else if (optimizer == "adam"){ |
|  | adamVals <-initializeAdam(parameters) |
|  | } |
|  |  |
|  | seed=10 |
|  |  |
|  | # Loop for number of epochs |
|  | for( i in 0:num\_epochs){ |
|  | seed=seed+1 |
|  | minibatches = random\_mini\_batches(X, Y, mini\_batch\_size, seed) |
|  |  |
|  | for(batch in 1:length(minibatches)){ |
|  |  |
|  | mini\_batch\_X=minibatches[[batch]][['mini\_batch\_X']] |
|  | mini\_batch\_Y=minibatches[[batch]][['mini\_batch\_Y']] |
|  | # Forward propagation: |
|  | retvals = forwardPropagationDeep(mini\_batch\_X, parameters,keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  | dropoutMat <- retvals[['dropoutMat']] |
|  |  |
|  | # Compute cost. |
|  | # Compute cost. |
|  | if(lambd==0){ |
|  | cost <- computeCost(AL, mini\_batch\_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } else { |
|  | cost <- computeCostWithReg(parameters, AL, Y,lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, mini\_batch\_Y, caches, dropoutMat, lambd, keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | if(optimizer == "gd"){ |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  | }else if(optimizer == "momentum"){ |
|  | # Update parameters with Momentum |
|  | parameters = gradientDescentWithMomentum(parameters, gradients,v,beta, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  |  |
|  | } else if(optimizer == "rmsprop"){ |
|  | # Update parameters with RMSProp |
|  | parameters = gradientDescentWithRMSProp(parameters, gradients,s,beta1, epsilon,learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  |  |
|  | } else if(optimizer == "adam"){ |
|  | # Update parameters with Adam |
|  | #Get v and s |
|  | t <- t+1 |
|  | v <- adamVals[['v']] |
|  | s <- adamVals[['s']] |
|  | parameters = gradientDescentWithAdam(parameters, gradients,v, s,t, beta1,beta2, epsilon,learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  | } |
|  | } |
|  |  |
|  | if(i%%1000 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | if(lrDecay==TRUE){ |
|  | learningRate = decayRate^(num\_epochs/1000) \* learningRate |
|  | } |
|  | } |
|  |  |
|  | retvals <- list("parameters"=parameters,"costs"=costs) |
|  |  |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Predict the output for given input |
|  | # Input : parameters |
|  | # : X |
|  | # Output: predictions |
|  | predict <- function(parameters, X,keep\_prob=1, hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,keep\_prob, hiddenActivationFunc) |
|  | predictions <- fwdProp$AL>0.5 |
|  |  |
|  | return (predictions) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary <- function(z,retvals,keep\_prob=1,hiddenActivationFunc="sigmoid",lr=0.5){ |
|  | # Find the minimum and maximum for the data |
|  | xmin<-min(z[,1]) |
|  | xmax<-max(z[,1]) |
|  | ymin<-min(z[,2]) |
|  | ymax<-max(z[,2]) |
|  |  |
|  | # Create a grid of values |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  | # Predict the output for this grid |
|  | q <-predict(retvals$parameters,grid1,keep\_prob=1, hiddenActivationFunc) |
|  | q1 <- t(data.frame(q)) |
|  | q2 <- as.numeric(q1) |
|  | grid2 <- cbind(grid,q2) |
|  | colnames(grid2) <- c('x1', 'x2','q2') |
|  |  |
|  | z1 <- data.frame(z) |
|  | names(z1) <- c("x1","x2","y") |
|  | atitle=paste("Decision boundary for learning rate:",lr) |
|  | # Plot the contour of the boundary |
|  | ggplot(z1) + |
|  | geom\_point(data = z1, aes(x = x1, y = x2, color = y)) + |
|  | stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha = 0.9)+ |
|  | ggtitle(atitle) + scale\_colour\_gradientn(colours = brewer.pal(10, "Spectral")) |
|  | } |
|  |  |
|  | # Predict the probability scores for given data set |
|  | # Input : parameters |
|  | # : X |
|  | # Output: probability of output |
|  | computeScores <- function(parameters, X,hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  | scores <- fwdProp$AL |
|  |  |
|  | return (scores) |
|  | } |
|  |  |
|  |  |
|  | random\_mini\_batches <- function(X, Y, miniBatchSize = 64, seed = 0){ |
|  |  |
|  |  |
|  | set.seed(seed) |
|  | # Get number of training samples |
|  | m = dim(X)[2] |
|  | # Initialize mini batches |
|  | mini\_batches = list() |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = c(sample(m)) |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X[, permutation] |
|  | shuffled\_Y = Y[1, permutation] |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = floor(m/miniBatchSize) |
|  | batch=0 |
|  | for(k in 0:(numCompleteMinibatches-1)){ |
|  | batch=batch+1 |
|  | # Set the lower and upper bound of the mini batches |
|  | lower=(k\*miniBatchSize)+1 |
|  | upper=((k+1) \* miniBatchSize) |
|  | mini\_batch\_X = shuffled\_X[, lower:upper] |
|  | mini\_batch\_Y = shuffled\_Y[lower:upper] |
|  | # Add it to the list of mini batches |
|  | mini\_batch = list("mini\_batch\_X"=mini\_batch\_X,"mini\_batch\_Y"=mini\_batch\_Y) |
|  | mini\_batches[[batch]] =mini\_batch |
|  |  |
|  |  |
|  | } |
|  |  |
|  | # If the batch size does not divide evenly with mini batc size |
|  | if(m %% miniBatchSize != 0){ |
|  | p=floor(m/miniBatchSize)\*miniBatchSize |
|  | # Set the start and end of last batch |
|  | q=p+m %% miniBatchSize |
|  | mini\_batch\_X = shuffled\_X[,(p+1):q] |
|  | mini\_batch\_Y = shuffled\_Y[(p+1):q] |
|  | } |
|  | # Return the list of mini batches |
|  | mini\_batch = list("mini\_batch\_X"=mini\_batch\_X,"mini\_batch\_Y"=mini\_batch\_Y) |
|  | mini\_batches[[batch]]=mini\_batch |
|  |  |
|  | return(mini\_batches) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary1 <- function(Z,parameters,keep\_prob=1){ |
|  | xmin<-min(Z[,1]) |
|  | xmax<-max(Z[,1]) |
|  | ymin<-min(Z[,2]) |
|  | ymax<-max(Z[,2]) |
|  |  |
|  | # Create a grid of points |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  |  |
|  | retvals = forwardPropagationDeep(grid1, parameters,keep\_prob, "relu", |
|  | outputActivationFunc="softmax") |
|  |  |
|  |  |
|  | AL <- retvals$AL |
|  | # From the softmax probabilities pick the one with the highest probability |
|  | q= apply(AL,1,which.max) |
|  |  |
|  | q1 <- t(data.frame(q)) |
|  | q2 <- as.numeric(q1) |
|  | grid2 <- cbind(grid,q2) |
|  | colnames(grid2) <- c('x1', 'x2','q2') |
|  |  |
|  | Z1 <- data.frame(Z) |
|  | names(Z1) <- c("x1","x2","y") |
|  | atitle=paste("Decision boundary") |
|  | ggplot(Z1) + |
|  | geom\_point(data = Z1, aes(x = x1, y = x2, color = y)) + |
|  | stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha = 0.9)+ |
|  | ggtitle(atitle) + scale\_colour\_gradientn(colours = brewer.pal(10, "Spectral")) |
|  | } |
|  | # Convert a list to a vector |
|  | list\_to\_vector <- function(parameters){ |
|  | vec <- NULL |
|  | L=length(parameters)/2 |
|  | for(l in 1:L){ |
|  | vec1= as.vector(t(parameters[[paste('W',l,sep="")]])) #Take transpose |
|  | vec1=as.matrix(vec1,nrow=length(vec1),ncol=1) |
|  | vec2= as.vector(t(parameters[[paste('b',l,sep="")]])) #Take transpose |
|  | vec2=as.matrix(vec2,nrow=length(vec2),ncol=1) |
|  | vec <- rbind(vec,vec1) |
|  | vec <- rbind(vec,vec2) |
|  | } |
|  | return(vec) |
|  | } |
|  |  |
|  | # Convert a list of gradients to a vector |
|  | gradients\_to\_vector <- function(parameters,gradients){ |
|  | vec <- NULL |
|  | L=length(parameters)/2 |
|  | for(l in 1:L){ |
|  | vec1= as.vector(t(gradients[[paste('dW',l,sep="")]])) #Take transpose |
|  | vec1=as.matrix(vec1,nrow=length(vec1),ncol=1) |
|  | vec2= as.vector(t(gradients[[paste('db',l,sep="")]])) #Take transpose |
|  | vec2=as.matrix(vec2,nrow=length(vec2),ncol=1) |
|  | vec <- rbind(vec,vec1) |
|  | vec <- rbind(vec,vec2) |
|  | } |
|  | return(vec) |
|  | } |
|  |  |
|  | # Convert vector to a list |
|  | # This should be a mirror copy of list\_to\_vector |
|  | vector\_to\_list <- function(parameters,theta){ |
|  | L<-length(parameters)/2 |
|  | start<-1 |
|  | parameters1 <- list() |
|  | for(l in 1:L){ |
|  | m = dim(parameters[[paste('W',l,sep="")]]) |
|  | a = theta[start:(start+m[1]\*m[2]-1),1] |
|  | parameters1[[paste('W',l,sep="")]] = t(matrix(a,nrow=m[2],ncol=m[1])) |
|  | start=start+m[1]\*m[2] |
|  | n = dim(parameters[[paste('b',l,sep="")]]) |
|  | b= theta[start:(start+n[1]\*n[2]-1),1] |
|  | parameters1[[paste('b',l,sep="")]]=t(matrix(b,nrow=n[2],ncol=n[1])) |
|  | start=start+n[1]\*n[2] |
|  | } |
|  |  |
|  | return(parameters1) |
|  | } |
|  |  |
|  | # Convert vector to a list. Vector of gradients to list of gradients |
|  | vector\_to\_list2 <- function(parameters,grads){ |
|  | L<-length(parameters)/2 |
|  | start<-1 |
|  | gradients1 <- list() |
|  | for(l in 1:L){ |
|  | m = dim(parameters[[paste('W',l,sep="")]]) |
|  | a = grads[start:(start+m[1]\*m[2]-1),1] |
|  | gradients1[[paste('dW',l,sep="")]] = matrix(a,nrow=m[1],ncol=m[2]) |
|  | start=start+m[1]\*m[2] |
|  | n = dim(parameters[[paste('b',l,sep="")]]) |
|  | b= grads[start:(start+n[1]\*n[2]-1),1] |
|  | gradients1[[paste('db',l,sep="")]]=matrix(b,nrow=n[1],ncol=n[2]) |
|  | start=start+n[1]\*n[2] |
|  | } |
|  |  |
|  | return(gradients1) |
|  | } |
|  |  |
|  | # Compute L2Norm |
|  | L2NormVec <- function(x) { |
|  | sqrt(sum(x^2)) |
|  | } |
|  |  |
|  | # Perform Gradient check |
|  | gradient\_check\_n <- function(parameters, gradients, X, Y, |
|  | epsilon = 1e-7,outputActivationFunc="sigmoid"){ |
|  | # Convert parameters to a vector |
|  | parameters\_values = list\_to\_vector(parameters) |
|  | # Convert gradients to a vector |
|  | grad = gradients\_to\_vector(parameters,gradients) |
|  | num\_parameters = dim(parameters\_values)[1] |
|  | #Initialize |
|  | J\_plus = matrix(rep(0,num\_parameters), |
|  | nrow=num\_parameters,ncol=1) |
|  | J\_minus = matrix(rep(0,num\_parameters), |
|  | nrow=num\_parameters,ncol=1) |
|  | gradapprox = matrix(rep(0,num\_parameters), |
|  | nrow=num\_parameters,ncol=1) |
|  |  |
|  | # Compute gradapprox |
|  | for(i in 1:num\_parameters){ |
|  | # Compute J\_plus[i]. |
|  | thetaplus = parameters\_values |
|  | thetaplus[i][1] = thetaplus[i][1] + epsilon |
|  | retvals = forwardPropagationDeep(X, vector\_to\_list(parameters,thetaplus), keep\_prob=1, |
|  | hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | AL <- retvals[['AL']] |
|  | J\_plus[i] = computeCost(AL, Y, outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | # Compute J\_minus[i]. |
|  | thetaminus = parameters\_values |
|  | thetaminus[i][1] = thetaminus[i][1] - epsilon |
|  | retvals = forwardPropagationDeep(X, vector\_to\_list(parameters,thetaminus), keep\_prob=1, |
|  | hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | J\_minus[i] = computeCost(AL, Y, outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | # Compute gradapprox[i] |
|  | gradapprox[i] = (J\_plus[i] - J\_minus[i])/(2\*epsilon) |
|  | } |
|  | # Compare gradapprox to backward propagation gradients by computing difference. |
|  | numerator = L2NormVec(grad-gradapprox) |
|  | denominator = L2NormVec(grad) + L2NormVec(gradapprox) |
|  | difference = numerator/denominator |
|  | if(difference > 1e-5){ |
|  | cat("There is a mistake, the difference is too high",difference) |
|  | } else{ |
|  | cat("The implementations works perfectly", difference) |
|  | } |
|  |  |
|  |  |
|  | # This can be used to check the structure of gradients and gradapprox |
|  | print("Gradients from backprop") |
|  | m=vector\_to\_list2(parameters,grad) |
|  | print(m) |
|  | print("Grad approx from gradient check") |
|  | n=vector\_to\_list2(parameters,gradapprox) |
|  | print(n) |
|  | } |

DLFunctions8.py

|  |
| --- |
| ###################################################### |
|  | # DL functions |
|  | ###################################################### |
|  | import numpy as np |
|  | import matplotlib.pyplot as plt |
|  | import matplotlib |
|  | import matplotlib.pyplot as plt |
|  | from matplotlib import cm |
|  | import math |
|  | import sklearn |
|  | import sklearn.datasets |
|  |  |
|  | # Conmpute the sigmoid of a vector |
|  | def sigmoid(Z): |
|  | A=1/(1+np.exp(-Z)) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the Relu of a vector |
|  | def relu(Z): |
|  | A = np.maximum(0,Z) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the tanh of a vector |
|  | def tanh(Z): |
|  | A = np.tanh(Z) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | def softmax(Z): |
|  | # get unnormalized probabilities |
|  | exp\_scores = np.exp(Z.T) |
|  | # normalize them for each example |
|  | A = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | def stableSoftmax(Z): |
|  | #Compute the softmax of vector x in a numerically stable way. |
|  | shiftZ = Z.T - np.max(Z.T,axis=1).reshape(-1,1) |
|  | exp\_scores = np.exp(shiftZ) |
|  |  |
|  | # normalize them for each example |
|  | A = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Compute the detivative of Relu |
|  | def reluDerivative(dA, cache): |
|  |  |
|  | Z = cache |
|  | dZ = np.array(dA, copy=True) # just converting dz to a correct object. |
|  | # When z <= 0, you should set dz to 0 as well. |
|  | dZ[Z <= 0] = 0 |
|  | return dZ |
|  |  |
|  | # Compute the derivative of sigmoid |
|  | def sigmoidDerivative(dA, cache): |
|  | Z = cache |
|  | s = 1/(1+np.exp(-Z)) |
|  | dZ = dA \* s \* (1-s) |
|  | return dZ |
|  |  |
|  | # Compute the derivative of tanh |
|  | def tanhDerivative(dA, cache): |
|  | Z = cache |
|  | a = np.tanh(Z) |
|  | dZ = dA \* (1 - np.power(a, 2)) |
|  | return dZ |
|  |  |
|  | # Compute the derivative of softmax |
|  | def softmaxDerivative(dA, cache,y,numTraining): |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z = cache |
|  | # Compute softmax |
|  | exp\_scores = np.exp(Z.T) |
|  | # normalize them for each example |
|  | probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  |  |
|  | # compute the gradient on scores |
|  | dZ = probs |
|  |  |
|  | # dZ = pi- yi |
|  | dZ[range(int(numTraining)),y[:,0]] -= 1 |
|  | return(dZ) |
|  |  |
|  | # Compute the derivative of softmax |
|  | def stableSoftmaxDerivative(dA, cache,y,numTraining): |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z = cache |
|  | # Compute stable softmax |
|  | shiftZ = Z.T - np.max(Z.T,axis=1).reshape(-1,1) |
|  | exp\_scores = np.exp(shiftZ) |
|  | # normalize them for each example |
|  | probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | #print(probs) |
|  | # compute the gradient on scores |
|  | dZ = probs |
|  |  |
|  | # dZ = pi- yi |
|  | dZ[range(int(numTraining)),y[:,0]] -= 1 |
|  | return(dZ) |
|  |  |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  | def initializeModel(numFeats,numHidden,numOutput): |
|  | np.random.seed(1) |
|  | W1=np.random.randn(numHidden,numFeats)\*0.01 # Multiply by .01 |
|  | b1=np.zeros((numHidden,1)) |
|  | W2=np.random.randn(numOutput,numHidden)\*0.01 |
|  | b2=np.zeros((numOutput,1)) |
|  |  |
|  | # Create a dictionary of the neural network parameters |
|  | nnParameters={'W1':W1,'b1':b1,'W2':W2,'b2':b2} |
|  | return(nnParameters) |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | def initializeDeepModel(layerDimensions): |
|  | np.random.seed(3) |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | layerParams['W' + str(l)] = np.random.randn(layerDimensions[l],layerDimensions[l-1])\*0.01 # Multiply by .01 |
|  | layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1)) |
|  | np.savetxt('W' + str(l)+'.csv',layerParams['W' + str(l)],delimiter=',') |
|  | np.savetxt('b' + str(l)+'.csv',layerParams['b' + str(l)],delimiter=',') |
|  | return(layerParams) |
|  | return Z, cache |
|  |  |
|  | # He Initialization model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | # He initilization multiplies the random numbers with sqrt(2/layerDimensions[l-1]) |
|  | def HeInitializeDeepModel(layerDimensions): |
|  | np.random.seed(3) |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | layerParams['W' + str(l)] = np.random.randn(layerDimensions[l], |
|  | layerDimensions[l-1])\*np.sqrt(2/layerDimensions[l-1]) |
|  | layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1)) |
|  |  |
|  | return(layerParams) |
|  | return Z, cache |
|  |  |
|  | # Xavier Initialization model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | # Xavier initilization multiplies the random numbers with sqrt(1/layerDimensions[l-1]) |
|  | def XavInitializeDeepModel(layerDimensions): |
|  | np.random.seed(3) |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | layerParams['W' + str(l)] = np.random.randn(layerDimensions[l], |
|  | layerDimensions[l-1])\*np.sqrt(1/layerDimensions[l-1]) |
|  | layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1)) |
|  |  |
|  | return(layerParams) |
|  | return Z, cache |
|  |  |
|  | # Initialize velocity of |
|  | # Input : parameters |
|  | # Returns: Initial velocity v |
|  | def initializeVelocity(parameters): |
|  |  |
|  | L = len(parameters)//2 # Create an integer |
|  | v = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l in range(L): |
|  | v["dW" + str(l+1)] = np.zeros((parameters['W' + str(l+1)].shape[0], |
|  | parameters['W' + str(l+1)].shape[1])) |
|  | v["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0], |
|  | parameters['b' + str(l+1)].shape[1])) |
|  |  |
|  | return v |
|  |  |
|  | # Initialize RMSProp param |
|  | # Input : parameters |
|  | # Returns: s |
|  | def initializeRMSProp(parameters): |
|  |  |
|  | L = len(parameters)//2 # Create an integer |
|  | s = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l in range(L): |
|  | s["dW" + str(l+1)] = np.zeros((parameters['W' + str(l+1)].shape[0], |
|  | parameters['W' + str(l+1)].shape[1])) |
|  | s["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0], |
|  | parameters['b' + str(l+1)].shape[1])) |
|  |  |
|  | return s |
|  |  |
|  | # Initialize Add param |
|  | # Input : List of units in each layer |
|  | # Returns: v and s |
|  | def initializeAdam(parameters) : |
|  |  |
|  | L = len(parameters) // 2 # number of layers in the neural networks |
|  | v = {} |
|  | s = {} |
|  |  |
|  | # Initialize v, s. |
|  | for l in range(L): |
|  |  |
|  | v["dW" + str(l+1)] = np.zeros((parameters['W' + str(l+1)].shape[0], |
|  | parameters['W' + str(l+1)].shape[1])) |
|  | v["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0], |
|  | parameters['b' + str(l+1)].shape[1])) |
|  | s["dW" + str(l+1)] = np.zeros((parameters['W' + str(l+1)].shape[0], |
|  | parameters['W' + str(l+1)].shape[1])) |
|  | s["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0], |
|  | parameters['b' + str(l+1)].shape[1])) |
|  | return v, s |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | def layerActivationForward(A\_prev, W, b, keep\_prob=0, activationFunc="relu"): |
|  |  |
|  | # Compute Z |
|  | Z = np.dot(W,A\_prev) + b |
|  | forward\_cache = (A\_prev, W, b) |
|  | # Compute the activation for sigmoid |
|  | if activationFunc == "sigmoid": |
|  | A, activation\_cache = sigmoid(Z) |
|  | # Compute the activation for Relu |
|  | elif activationFunc == "relu": |
|  | A, activation\_cache = relu(Z) |
|  | # Compute the activation for tanh |
|  | elif activationFunc == 'tanh': |
|  | A, activation\_cache = tanh(Z) |
|  | elif activationFunc == 'softmax': |
|  | A, activation\_cache = stableSoftmax(Z) |
|  |  |
|  | cache = (forward\_cache, activation\_cache) |
|  | return A, cache |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - Activation function at hidden layers Relu/tanh |
|  | # outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | def forwardPropagationDeep(X, parameters,keep\_prob=0, hiddenActivationFunc='relu',outputActivationFunc='sigmoid'): |
|  | caches = [] |
|  | #initialize the dropout matrix |
|  | dropoutMat = {} |
|  | # Set A to X (A0) |
|  | A = X |
|  | L = len(parameters)//2 # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for l in range(1, L): |
|  | A\_prev = A |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | A, cache = layerActivationForward(A\_prev, parameters['W'+str(l)], parameters['b'+str(l)], keep\_prob, activationFunc = hiddenActivationFunc) |
|  |  |
|  | # Randomly drop some activation units |
|  | # Create a matrix as the same shape as A |
|  | D = np.random.rand(A.shape[0],A.shape[1]) |
|  | D = (D < keep\_prob) |
|  | # We need to use the same 'dropout' matrix in backward propagation |
|  | # Save the dropout matrix for use in backprop |
|  | dropoutMat["D" + str(l)] =D |
|  | A= np.multiply(A,D) |
|  | A = np.divide(A,keep\_prob) |
|  |  |
|  | caches.append(cache) |
|  |  |
|  |  |
|  | # last layer |
|  | AL, cache = layerActivationForward(A, parameters['W'+str(L)], parameters['b'+str(L)], activationFunc = outputActivationFunc) |
|  | caches.append(cache) |
|  |  |
|  | return AL, caches, dropoutMat |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : Y |
|  | # :outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # Output: cost |
|  | def computeCost(AL,Y,outputActivationFunc="sigmoid"): |
|  | if outputActivationFunc=="sigmoid": |
|  | m= float(Y.shape[1]) |
|  | # Element wise multiply for logprobs |
|  | cost=-1/m \*np.sum(Y\*np.log(AL) + (1-Y)\*(np.log(1-AL))) |
|  | cost = np.squeeze(cost) |
|  | elif outputActivationFunc=="softmax": |
|  | # Take transpose of Y for softmax |
|  | Y=Y.T |
|  | m= float(len(Y)) |
|  | # Compute log probs. Take the log prob of correct class based on output y |
|  | correct\_logprobs = -np.log(AL[range(int(m)),Y.T]) |
|  | # Conpute loss |
|  | cost = np.sum(correct\_logprobs)/m |
|  | return cost |
|  |  |
|  |  |
|  | # Compute the cost with regularization |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : Y |
|  | # :outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # Output: cost |
|  | def computeCostWithReg(parameters,AL,Y,lambd, outputActivationFunc="sigmoid"): |
|  |  |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | m= float(Y.shape[1]) |
|  | # Element wise multiply for logprobs |
|  | cost=-1/m \*np.sum(Y\*np.log(AL) + (1-Y)\*(np.log(1-AL))) |
|  | cost = np.squeeze(cost) |
|  |  |
|  | # Regularization cost |
|  | L= int(len(parameters)/2) |
|  | L2RegularizationCost=0 |
|  | for l in range(L): |
|  | L2RegularizationCost+=np.sum(np.square(parameters['W'+str(l+1)])) |
|  |  |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  |  |
|  |  |
|  | elif outputActivationFunc=="softmax": |
|  | # Take transpose of Y for softmax |
|  | Y=Y.T |
|  | m= float(len(Y)) |
|  | # Compute log probs. Take the log prob of correct class based on output y |
|  | correct\_logprobs = -np.log(AL[range(int(m)),Y.T]) |
|  | # Conpute loss |
|  | cost = np.sum(correct\_logprobs)/m |
|  |  |
|  | # Regularization cost |
|  | L= int(len(parameters)/2) |
|  | L2RegularizationCost=0 |
|  | for l in range(L): |
|  | L2RegularizationCost+=np.sum(np.square(parameters['W'+str(l+1)])) |
|  |  |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  |  |
|  | return cost |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | def layerActivationBackward(dA, cache, Y, keep\_prob=1, activationFunc="relu"): |
|  | forward\_cache, activation\_cache = cache |
|  | A\_prev, W, b = forward\_cache |
|  | numtraining = float(A\_prev.shape[1]) |
|  | if activationFunc == "relu": |
|  | dZ = reluDerivative(dA, activation\_cache) |
|  | elif activationFunc == "sigmoid": |
|  | dZ = sigmoidDerivative(dA, activation\_cache) |
|  | elif activationFunc == "tanh": |
|  | dZ = tanhDerivative(dA, activation\_cache) |
|  | elif activationFunc == "softmax": |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numtraining) |
|  |  |
|  | if activationFunc == 'softmax': |
|  | dW = 1/numtraining \* np.dot(A\_prev,dZ) |
|  | db = 1/numtraining \* np.sum(dZ, axis=0, keepdims=True) |
|  | dA\_prev = np.dot(dZ,W) |
|  | else: |
|  |  |
|  | dW = 1/numtraining \*(np.dot(dZ,A\_prev.T)) |
|  | db = 1/numtraining \* np.sum(dZ, axis=1, keepdims=True) |
|  | dA\_prev = np.dot(W.T,dZ) |
|  |  |
|  | return dA\_prev, dW, db |
|  |  |
|  |  |
|  | # Compute the backpropoagation with regularization for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | def layerActivationBackwardWithReg(dA, cache, Y, lambd, activationFunc): |
|  | forward\_cache, activation\_cache = cache |
|  | A\_prev, W, b = forward\_cache |
|  | numtraining = float(A\_prev.shape[1]) |
|  |  |
|  | #print("n=",numtraining) |
|  | #print("no=",numtraining) |
|  | if activationFunc == "relu": |
|  | dZ = reluDerivative(dA, activation\_cache) |
|  | elif activationFunc == "sigmoid": |
|  | dZ = sigmoidDerivative(dA, activation\_cache) |
|  | elif activationFunc == "tanh": |
|  | dZ = tanhDerivative(dA, activation\_cache) |
|  | elif activationFunc == "softmax": |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numtraining) |
|  |  |
|  | if activationFunc == 'softmax': |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \* np.dot(A\_prev,dZ) + (lambd/numtraining) \* W.T |
|  | db = 1/numtraining \* np.sum(dZ, axis=0, keepdims=True) |
|  | dA\_prev = np.dot(dZ,W) |
|  | else: |
|  |  |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \*(np.dot(dZ,A\_prev.T)) + (lambd/numtraining) \* W |
|  | #print("dW=",dW) |
|  | db = 1/numtraining \* np.sum(dZ, axis=1, keepdims=True) |
|  | #print("db=",db) |
|  | dA\_prev = np.dot(W.T,dZ) |
|  |  |
|  |  |
|  | return dA\_prev, dW, db |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers - relu/sigmoid/tanh |
|  | # # outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # gradients["dW" + str(l)] = ... |
|  |  |
|  | def backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0, keep\_prob=1, hiddenActivationFunc='relu',outputActivationFunc="sigmoid"): |
|  | #initialize the gradients |
|  | gradients = {} |
|  | # Set the number of layers |
|  | L = len(caches) |
|  | m = float(AL.shape[1]) |
|  |  |
|  | if outputActivationFunc == "sigmoid": |
|  | Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a + (1-y)/(1-a)) - At the output layer |
|  | dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) |
|  | else: |
|  | dAL =0 |
|  | Y=Y.T |
|  |  |
|  | # Since this is a binary classification the activation at output is sigmoid |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | current\_cache = caches[L-1] |
|  | if lambd==0: |
|  | gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackward(dAL, current\_cache, |
|  | Y, activationFunc = outputActivationFunc) |
|  | else: #Regularization |
|  | gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackwardWithReg(dAL, current\_cache, |
|  | Y, lambd, activationFunc = outputActivationFunc) |
|  |  |
|  | # Note dA for softmax is the transpose |
|  | if outputActivationFunc == "softmax": |
|  | gradients["dA" + str(L)] = gradients["dA" + str(L)].T |
|  | # Traverse in the reverse direction |
|  | for l in reversed(range(L-1)): |
|  |  |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | current\_cache = caches[l] |
|  |  |
|  | #dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+2)], current\_cache, activationFunc = "relu") |
|  | if lambd==0: |
|  | # In the reverse direction use the dame dropout matrix |
|  | # Random dropout |
|  | # Multiply dA'l' with the dropoutMat and divide to keep the expected value same |
|  | D = dropoutMat["D" + str(l+1)] |
|  | # Drop some dAl's |
|  | gradients['dA'+str(l+2)]= np.multiply(gradients['dA'+str(l+2)],D) |
|  | # Divide by keep\_prob to keep expected value same |
|  | gradients['dA'+str(l+2)] = np.divide(gradients['dA'+str(l+2)],keep\_prob) |
|  |  |
|  | dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+2)], current\_cache, Y, keep\_prob=1, activationFunc = hiddenActivationFunc) |
|  |  |
|  | else: |
|  | dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackwardWithReg(gradients['dA'+str(l+2)], current\_cache, Y, lambd, activationFunc = hiddenActivationFunc) |
|  | gradients["dA" + str(l + 1)] = dA\_prev\_temp |
|  | gradients["dW" + str(l + 1)] = dW\_temp |
|  | gradients["db" + str(l + 1)] = db\_temp |
|  |  |
|  | return gradients |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | def gradientDescent(parameters, gradients, learningRate,outputActivationFunc="sigmoid"): |
|  |  |
|  | L = int(len(parameters) / 2) |
|  | # Update rule for each parameter. |
|  | for l in range(L-1): |
|  | parameters["W" + str(l+1)] = parameters['W'+str(l+1)] -learningRate\* gradients['dW' + str(l+1)] |
|  | parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate\* gradients['db' + str(l+1)] |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)] |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)] |
|  | elif outputActivationFunc=="softmax": |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)].T |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)].T |
|  |  |
|  | return parameters |
|  |  |
|  | # Update parameters with momentum |
|  | # Input : parameters |
|  | # : gradients |
|  | # : v |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters and velocity |
|  | def gradientDescentWithMomentum(parameters, gradients, v, beta, learningRate, outputActivationFunc="sigmoid"): |
|  |  |
|  | L = len(parameters) // 2 # number of layers in the neural networks |
|  | # Momentum update for each parameter |
|  | for l in range(L-1): |
|  |  |
|  | # Compute velocities |
|  | # v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk |
|  | v["dW" + str(l+1)] = beta\*v["dW" + str(l+1)] + (1-beta) \* gradients['dW' + str(l+1)] |
|  | v["db" + str(l+1)] = beta\*v["db" + str(l+1)] + (1-beta) \* gradients['db' + str(l+1)] |
|  | # Update parameters with velocities |
|  | parameters["W" + str(l+1)] = parameters['W' + str(l+1)] - learningRate\* v["dW" + str(l+1)] |
|  | parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - learningRate\* v["db" + str(l+1)] |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | v["dW" + str(L)] = beta\*v["dW" + str(L)] + (1-beta) \* gradients['dW' + str(L)] |
|  | v["db" + str(L)] = beta\*v["db" + str(L)] + (1-beta) \* gradients['db' + str(L)] |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)] |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)] |
|  | elif outputActivationFunc=="softmax": |
|  | v["dW" + str(L)] = beta\*v["dW" + str(L)] + (1-beta) \* gradients['dW' + str(L)].T |
|  | v["db" + str(L)] = beta\*v["db" + str(L)] + (1-beta) \* gradients['db' + str(L)].T |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)].T |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)].T |
|  |  |
|  | return parameters, v |
|  |  |
|  |  |
|  | # Update parameters with RMSProp |
|  | # Input : parameters |
|  | # : gradients |
|  | # : v |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters and velocity |
|  | def gradientDescentWithRMSProp(parameters, gradients, s, beta1, epsilon, learningRate, outputActivationFunc="sigmoid"): |
|  |  |
|  | L = len(parameters) // 2 # number of layers in the neural networks |
|  | # Momentum update for each parameter |
|  | for l in range(L-1): |
|  |  |
|  | # Compute RMSProp |
|  | # s['dWk'] = beta1 \*s['dWk'] + (1-beta1)\*dWk\*\*2/sqrt(s['dWk']) |
|  | s["dW" + str(l+1)] = beta1\*s["dW" + str(l+1)] + (1-beta1) \* \ |
|  | np.multiply(gradients['dW' + str(l+1)],gradients['dW' + str(l+1)]) |
|  | s["db" + str(l+1)] = beta1\*s["db" + str(l+1)] + (1-beta1) \* \ |
|  | np.multiply(gradients['db' + str(l+1)],gradients['db' + str(l+1)]) |
|  | # Update parameters with RMSProp |
|  | parameters["W" + str(l+1)] = parameters['W' + str(l+1)] - \ |
|  | learningRate\* gradients['dW' + str(l+1)]/np.sqrt(s["dW" + str(l+1)] + epsilon) |
|  | parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - \ |
|  | learningRate\* gradients['db' + str(l+1)]/np.sqrt(s["db" + str(l+1)] + epsilon) |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | s["dW" + str(L)] = beta1\*s["dW" + str(L)] + (1-beta1) \* \ |
|  | np.multiply(gradients['dW' + str(L)],gradients['dW' + str(L)]) |
|  | s["db" + str(L)] = beta1\*s["db" + str(L)] + (1-beta1) \* \ |
|  | np.multiply(gradients['db' + str(L)],gradients['db' + str(L)]) |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] - \ |
|  | learningRate\* gradients['dW' + str(L)]/np.sqrt(s["dW" + str(L)] + epsilon) |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] - \ |
|  | learningRate\* gradients['db' + str(L)]/np.sqrt(s["db" + str(L)] + epsilon) |
|  | elif outputActivationFunc=="softmax": |
|  | s["dW" + str(L)] = beta1\*s["dW" + str(L)] + (1-beta1) \* \ |
|  | np.multiply(gradients['dW' + str(L)].T,gradients['dW' + str(L)].T) |
|  | s["db" + str(L)] = beta1\*s["db" + str(L)] + (1-beta1) \* \ |
|  | np.multiply(gradients['db' + str(L)].T,gradients['db' + str(L)].T) |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] - \ |
|  | learningRate\* gradients['dW' + str(L)].T/np.sqrt(s["dW" + str(L)] + epsilon) |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] - \ |
|  | learningRate\* gradients['db' + str(L)].T/np.sqrt(s["db" + str(L)] + epsilon) |
|  |  |
|  | return parameters, s |
|  |  |
|  |  |
|  | def gradientDescentWithAdam(parameters, gradients, v, s, t, |
|  | beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, |
|  | learningRate=0.1, outputActivationFunc="sigmoid"): |
|  |  |
|  |  |
|  | L = len(parameters) // 2 |
|  | # Initializing first moment estimate, python dictionary |
|  | v\_corrected = {} |
|  | # Initializing second moment estimate, python dictionary |
|  | s\_corrected = {} |
|  |  |
|  | # Perform Adam upto L-1 |
|  | for l in range(L-1): |
|  |  |
|  | # Compute momentum |
|  | v["dW" + str(l+1)] = beta1\*v["dW" + str(l+1)] + \ |
|  | (1-beta1) \* gradients['dW' + str(l+1)] |
|  | v["db" + str(l+1)] = beta1\*v["db" + str(l+1)] + \ |
|  | (1-beta1) \* gradients['db' + str(l+1)] |
|  |  |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected["dW" + str(l+1)] = v["dW" + str(l+1)]/(1-np.power(beta1,t)) |
|  | v\_corrected["db" + str(l+1)] = v["db" + str(l+1)]/(1-np.power(beta1,t)) |
|  |  |
|  |  |
|  | # Moving average of the squared gradients like RMSProp |
|  | s["dW" + str(l+1)] = beta2\*s["dW" + str(l+1)] + \ |
|  | (1-beta2) \* np.multiply(gradients['dW' + str(l+1)],gradients['dW' + str(l+1)]) |
|  | s["db" + str(l+1)] = beta2\*s["db" + str(l+1)] + \ |
|  | (1-beta2) \* np.multiply(gradients['db' + str(l+1)],gradients['db' + str(l+1)]) |
|  |  |
|  |  |
|  | # Compute bias-corrected second raw moment estimate. |
|  | s\_corrected["dW" + str(l+1)] = s["dW" + str(l+1)]/(1-np.power(beta2,t)) |
|  | s\_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-np.power(beta2,t)) |
|  |  |
|  | # Update parameters. |
|  | d1=np.sqrt(s\_corrected["dW" + str(l+1)]+epsilon) |
|  | d2=np.sqrt(s\_corrected["db" + str(l+1)]+epsilon) |
|  | parameters["W" + str(l+1)] = parameters['W' + str(l+1)]- \ |
|  | (learningRate\* v\_corrected["dW" + str(l+1)]/d1) |
|  | parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - \ |
|  | (learningRate\* v\_corrected["db" + str(l+1)]/d2) |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | #Compute 1st moment for L |
|  | v["dW" + str(L)] = beta1\*v["dW" + str(L)] + (1-beta1) \* gradients['dW' + str(L)] |
|  | v["db" + str(L)] = beta1\*v["db" + str(L)] + (1-beta1) \* gradients['db' + str(L)] |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected["dW" + str(L)] = v["dW" + str(L)]/(1-np.power(beta1,t)) |
|  | v\_corrected["db" + str(L)] = v["db" + str(L)]/(1-np.power(beta1,t)) |
|  |  |
|  | # Compute 2nd moment for L |
|  | s["dW" + str(L)] = beta2\*s["dW" + str(L)] + (1-beta2) \* \ |
|  | np.multiply(gradients['dW' + str(L)],gradients['dW' + str(L)]) |
|  | s["db" + str(L)] = beta2\*s["db" + str(L)] + (1-beta2) \* \ |
|  | np.multiply(gradients['db' + str(L)],gradients['db' + str(L)]) |
|  |  |
|  | # Compute bias-corrected second raw moment estimate. |
|  | s\_corrected["dW" + str(L)] = s["dW" + str(L)]/(1-np.power(beta2,t)) |
|  | s\_corrected["db" + str(L)] = s["db" + str(L)]/(1-np.power(beta2,t)) |
|  |  |
|  | # Update parameters. |
|  | d1=np.sqrt(s\_corrected["dW" + str(L)]+epsilon) |
|  | d2=np.sqrt(s\_corrected["db" + str(L)]+epsilon) |
|  | parameters["W" + str(L)] = parameters['W' + str(L)]- \ |
|  | (learningRate\* v\_corrected["dW" + str(L)]/d1) |
|  | parameters["b" + str(L)] = parameters['b' + str(L)] - \ |
|  | (learningRate\* v\_corrected["db" + str(L)]/d2) |
|  |  |
|  | elif outputActivationFunc=="softmax": |
|  | # Compute 1st moment |
|  | v["dW" + str(L)] = beta1\*v["dW" + str(L)] + (1-beta1) \* gradients['dW' + str(L)].T |
|  | v["db" + str(L)] = beta1\*v["db" + str(L)] + (1-beta1) \* gradients['db' + str(L)].T |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected["dW" + str(L)] = v["dW" + str(L)]/(1-np.power(beta1,t)) |
|  | v\_corrected["db" + str(L)] = v["db" + str(L)]/(1-np.power(beta1,t)) |
|  |  |
|  | #Compute 2nd moment |
|  | s["dW" + str(L)] = beta2\*s["dW" + str(L)] + (1-beta2) \* np.multiply(gradients['dW' + str(L)].T,gradients['dW' + str(L)].T) |
|  | s["db" + str(L)] = beta2\*s["db" + str(L)] + (1-beta2) \* np.multiply(gradients['db' + str(L)].T,gradients['db' + str(L)].T) |
|  | # Compute bias-corrected second raw moment estimate. |
|  | s\_corrected["dW" + str(L)] = s["dW" + str(L)]/(1-np.power(beta2,t)) |
|  | s\_corrected["db" + str(L)] = s["db" + str(L)]/(1-np.power(beta2,t)) |
|  |  |
|  | # Update parameters. |
|  | d1=np.sqrt(s\_corrected["dW" + str(L)]+epsilon) |
|  | d2=np.sqrt(s\_corrected["db" + str(L)]+epsilon) |
|  | parameters["W" + str(L)] = parameters['W' + str(L)]- \ |
|  | (learningRate\* v\_corrected["dW" + str(L)]/d1) |
|  | parameters["b" + str(L)] = parameters['b' + str(L)] - \ |
|  | (learningRate\* v\_corrected["db" + str(L)]/d2) |
|  |  |
|  |  |
|  | return parameters, v, s |
|  |  |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh/sigmoid |
|  | # : learning rate |
|  | # : num of iteration |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  |  |
|  | def L\_Layer\_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid", |
|  | learningRate = .3, lambd=0, keep\_prob=1, num\_iterations = 10000,initType="default", print\_cost=False,figure="figa.png"): |
|  |  |
|  | np.random.seed(1) |
|  | costs = [] |
|  |  |
|  | # Parameters initialization. |
|  | if initType == "He": |
|  | parameters = HeInitializeDeepModel(layersDimensions) |
|  | elif initType == "Xavier" : |
|  | parameters = XavInitializeDeepModel(layersDimensions) |
|  | else: #Default |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | # Loop (gradient descent) |
|  | for i in range(0, num\_iterations): |
|  |  |
|  | AL, caches, dropoutMat = forwardPropagationDeep(X1, parameters, keep\_prob, hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Regularization parameter is 0 |
|  | if lambd==0: |
|  | # Compute cost |
|  | cost = computeCost(parameters,AL, Y1, outputActivationFunc=outputActivationFunc) |
|  | # Include L2 regularization |
|  | else: |
|  | # Compute cost |
|  | cost = computeCostWithReg(parameters,AL, Y1, lambd, outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, Y1, caches, dropoutMat, lambd, keep\_prob, hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | # Print the cost every 100 training example |
|  | if print\_cost and i % 1000 == 0: |
|  | print ("Cost after iteration %i: %f" %(i, cost)) |
|  | if print\_cost and i % 1000 == 0: |
|  | costs.append(cost) |
|  |  |
|  | # plot the cost |
|  | plt.plot(np.squeeze(costs)) |
|  | plt.ylabel('Cost') |
|  | plt.xlabel('No of iterations (x1000)') |
|  | plt.title("Learning rate =" + str(learningRate)) |
|  | plt.savefig(figure,bbox\_inches='tight') |
|  | #plt.show() |
|  | plt.clf() |
|  | plt.close() |
|  |  |
|  | return parameters |
|  |  |
|  | # Execute a L layer Deep learning model Stoachastic Gradient Descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh/sigmoid |
|  | # : learning rate |
|  | # : num of iteration |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  |  |
|  | def L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid", |
|  | learningRate = .3, lrDecay=False, decayRate=1, |
|  | lambd=0, keep\_prob=1, optimizer="gd",beta=0.9,beta1=0.9, beta2=0.999, |
|  | epsilon = 1e-8,mini\_batch\_size = 64, num\_epochs = 2500, print\_cost=False, figure="figa.png"): |
|  |  |
|  | print("lr=",learningRate) |
|  | print("lrDecay=",lrDecay) |
|  | print("decayRate=",decayRate) |
|  | print("lambd=",lambd) |
|  | print("keep\_prob=",keep\_prob) |
|  | print("optimizer=",optimizer) |
|  | print("beta=",beta) |
|  |  |
|  | print("beta1=",beta1) |
|  | print("beta2=",beta2) |
|  | print("epsilon=",epsilon) |
|  |  |
|  | print("mini\_batch\_size=",mini\_batch\_size) |
|  | print("num\_epochs=",num\_epochs) |
|  | print("epsilon=",epsilon) |
|  |  |
|  |  |
|  | t =0 # Adam counter |
|  | np.random.seed(1) |
|  | costs = [] |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  |  |
|  | #Initialize the optimizer |
|  | if optimizer == "gd": |
|  | pass # no initialization required for gradient descent |
|  | elif optimizer == "momentum": |
|  | v = initializeVelocity(parameters) |
|  | elif optimizer == "rmsprop": |
|  | s = initializeRMSProp(parameters) |
|  | elif optimizer == "adam": |
|  | v,s = initializeAdam(parameters) |
|  |  |
|  | seed=10 |
|  | # Loop for number of epochs |
|  | for i in range(num\_epochs): |
|  | # Define the random minibatches. We increment the seed to reshuffle differently the dataset after each epoch |
|  | seed = seed + 1 |
|  | minibatches = random\_mini\_batches(X1, Y1, mini\_batch\_size, seed) |
|  |  |
|  | batch=0 |
|  | # Loop through each mini batch |
|  | for minibatch in minibatches: |
|  | #print("batch=",batch) |
|  | batch=batch+1 |
|  | # Select a minibatch |
|  | (minibatch\_X, minibatch\_Y) = minibatch |
|  |  |
|  | # Perfrom forward propagation |
|  | AL, caches, dropoutMat = forwardPropagationDeep(minibatch\_X, parameters, keep\_prob, hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Compute cost |
|  | # Regularization parameter is 0 |
|  | if lambd==0: |
|  | # Compute cost |
|  | cost = computeCost(parameters, AL, minibatch\_Y, outputActivationFunc=outputActivationFunc) |
|  | else: # Include L2 regularization |
|  | # Compute cost |
|  | cost = computeCostWithReg(parameters, AL, minibatch\_Y, lambd, outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, minibatch\_Y, caches,dropoutMat, lambd, keep\_prob,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | if optimizer == "gd": |
|  | # Update parameters normal gradient descent |
|  | parameters = gradientDescent(parameters, gradients, learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  | elif optimizer == "momentum": |
|  | # Update parameters for gradient descent with momentum |
|  | parameters, v = gradientDescentWithMomentum(parameters, gradients, v, beta, \ |
|  | learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  | elif optimizer == "rmsprop": |
|  | # Update parameters for gradient descent with RMSProp |
|  | parameters, s = gradientDescentWithRMSProp(parameters, gradients, s, beta1, epsilon, \ |
|  | learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  | elif optimizer == "adam": |
|  | t = t + 1 # Adam counter |
|  | parameters, v, s = gradientDescentWithAdam(parameters, gradients, v, s, |
|  | t, beta1, beta2, epsilon, |
|  | learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Print the cost every 1000 epoch |
|  | if print\_cost and i % 100 == 0: |
|  | print ("Cost after epoch %i: %f" %(i, cost)) |
|  | if print\_cost and i % 100 == 0: |
|  | costs.append(cost) |
|  | if lrDecay == True: |
|  | learningRate = np.power(decayRate,(num\_epochs/1000)) \* learningRate |
|  |  |
|  |  |
|  | # plot the cost |
|  | plt.plot(np.squeeze(costs)) |
|  | plt.ylabel('Cost') |
|  | plt.xlabel('No of epochs(x100)') |
|  | plt.title("Learning rate =" + str(learningRate)) |
|  | plt.savefig(figure,bbox\_inches='tight') |
|  | #plt.show() |
|  | plt.clf() |
|  | plt.close() |
|  |  |
|  |  |
|  | # Create random mini batches |
|  | def random\_mini\_batches(X, Y, miniBatchSize = 64, seed = 0): |
|  |  |
|  | np.random.seed(seed) |
|  | # Get number of training samples |
|  | m = X.shape[1] |
|  | # Initialize mini batches |
|  | mini\_batches = [] |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = list(np.random.permutation(m)) |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X[:, permutation] |
|  | shuffled\_Y = Y[:, permutation].reshape((1,m)) |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = math.floor(m/miniBatchSize) |
|  |  |
|  | # For the number of mini batches |
|  | for k in range(0, numCompleteMinibatches): |
|  |  |
|  | # Set the start and end of each mini batch |
|  | mini\_batch\_X = shuffled\_X[:, k\*miniBatchSize : (k+1) \* miniBatchSize] |
|  | mini\_batch\_Y = shuffled\_Y[:, k\*miniBatchSize : (k+1) \* miniBatchSize] |
|  |  |
|  | mini\_batch = (mini\_batch\_X, mini\_batch\_Y) |
|  | mini\_batches.append(mini\_batch) |
|  |  |
|  |  |
|  | #if m % miniBatchSize != 0:. The batch does not evenly divide by the mini batch |
|  | if m % miniBatchSize != 0: |
|  | l=math.floor(m/miniBatchSize)\*miniBatchSize |
|  | # Set the start and end of last mini batch |
|  | m=l+m % miniBatchSize |
|  | mini\_batch\_X = shuffled\_X[:,l:m] |
|  | mini\_batch\_Y = shuffled\_Y[:,l:m] |
|  |  |
|  | mini\_batch = (mini\_batch\_X, mini\_batch\_Y) |
|  | mini\_batches.append(mini\_batch) |
|  |  |
|  | return mini\_batches |
|  |  |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # sz - Num of hiden units |
|  | # lr - Learning rate |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_boundary(model, X, y,lr,figure1="figb.png"): |
|  | print("plot") |
|  | # Set min and max values and give it some padding |
|  | x\_min, x\_max = X[0, :].min() - 1, X[0, :].max() + 1 |
|  | y\_min, y\_max = X[1, :].min() - 1, X[1, :].max() + 1 |
|  | colors=['black','gold'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 0.01 |
|  | # Generate a grid of points with distance h between them |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h)) |
|  | # Predict the function value for the whole grid |
|  | Z = model(np.c\_[xx.ravel(), yy.ravel()]) |
|  | Z = Z.reshape(xx.shape) |
|  | # Plot the contour and training examples |
|  | plt.contourf(xx, yy, Z, cmap="coolwarm") |
|  | plt.ylabel('x2') |
|  | plt.xlabel('x1') |
|  | x=X.T |
|  | y=y.T.reshape(300,) |
|  | plt.scatter(x[:, 0], x[:, 1], c=y, s=20); |
|  | print(X.shape) |
|  | plt.title("Decision Boundary for learning rate:"+lr) |
|  | plt.savefig(figure1, bbox\_inches='tight') |
|  | #plt.show() |
|  |  |
|  |  |
|  | def predict(parameters, X,keep\_prob=1,hiddenActivationFunc="relu",outputActivationFunc="sigmoid"): |
|  | A2, cache,dropoutMat = forwardPropagationDeep(X, parameters, keep\_prob=1, hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  | predictions = (A2>0.5) |
|  | return predictions |
|  |  |
|  | def predict\_proba(parameters, X,outputActivationFunc="sigmoid"): |
|  | A2, cache = forwardPropagationDeep(X, parameters) |
|  | if outputActivationFunc=="sigmoid": |
|  | proba=A2 |
|  | elif outputActivationFunc=="softmax": |
|  | proba=np.argmax(A2, axis=0).reshape(-1,1) |
|  | print("A2=",A2.shape) |
|  | return proba |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # sz - Num of hiden units |
|  | # lr - Learning rate |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_boundary1(X, y,W1,b1,W2,b2,figure2="figc.png"): |
|  | #plot\_decision\_boundary(lambda x: predict(parameters, x.T), x1,y1.T,str(0.3),"fig2.png") |
|  | h = 0.02 |
|  | x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1 |
|  | y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1 |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), |
|  | np.arange(y\_min, y\_max, h)) |
|  | Z = np.dot(np.maximum(0, np.dot(np.c\_[xx.ravel(), yy.ravel()], W1.T) + b1.T), W2.T) + b2.T |
|  | Z = np.argmax(Z, axis=1) |
|  | Z = Z.reshape(xx.shape) |
|  |  |
|  | fig = plt.figure() |
|  | plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8) |
|  | print(X.shape) |
|  | y1=y.reshape(300,) |
|  | plt.scatter(X[:, 0], X[:, 1], c=y1, s=40, cmap=plt.cm.Spectral) |
|  | plt.xlim(xx.min(), xx.max()) |
|  | plt.ylim(yy.min(), yy.max()) |
|  | plt.savefig(figure2, bbox\_inches='tight') |
|  |  |
|  |  |
|  | def load\_dataset(): |
|  | np.random.seed(1) |
|  | train\_X, train\_Y = sklearn.datasets.make\_circles(n\_samples=300, noise=.05) |
|  | np.random.seed(2) |
|  | test\_X, test\_Y = sklearn.datasets.make\_circles(n\_samples=100, noise=.05) |
|  | # Visualize the data |
|  | print(train\_X.shape) |
|  | print(train\_Y.shape) |
|  | #plt.scatter(train\_X[:, 0], train\_X[:, 1], c=train\_Y, s=40, cmap=plt.cm.Spectral); |
|  | train\_X = train\_X.T |
|  | train\_Y = train\_Y.reshape((1, train\_Y.shape[0])) |
|  | test\_X = test\_X.T |
|  | test\_Y = test\_Y.reshape((1, test\_Y.shape[0])) |
|  | return train\_X, train\_Y, test\_X, test\_Y |
|  |  |
|  | def dictionary\_to\_vector(parameters): |
|  | """ |
|  | Roll all our parameters dictionary into a single vector satisfying our specific required shape. |
|  | """ |
|  | keys = [] |
|  | count = 0 |
|  | for key in parameters: |
|  | # flatten parameter |
|  | new\_vector = np.reshape(parameters[key], (-1,1)) |
|  | keys = keys + [key]\*new\_vector.shape[0] |
|  |  |
|  | if count == 0: |
|  | theta = new\_vector |
|  | else: |
|  | theta = np.concatenate((theta, new\_vector), axis=0) |
|  | count = count + 1 |
|  |  |
|  | return theta, keys |
|  |  |
|  | def gradients\_to\_vector(parameters, gradients): |
|  |  |
|  | #Roll all our gradients dictionary into a single vector satisfying our specific required shape. |
|  |  |
|  | keyvals=[] |
|  | L=len(parameters)//2 |
|  | count = 0 |
|  | for l in range(L): |
|  | # flatten parameter |
|  | keyvals.append('dW'+str(l+1)) |
|  | keyvals.append('db'+str(l+1)) |
|  |  |
|  | for key in keyvals: |
|  | new\_vector = np.reshape(gradients[key], (-1,1)) |
|  |  |
|  | if count == 0: |
|  | theta = new\_vector |
|  | else: |
|  | theta = np.concatenate((theta, new\_vector), axis=0) |
|  | count = count + 1 |
|  |  |
|  | return theta |
|  |  |
|  | def vector\_to\_dictionary(parameters,theta): |
|  | #Unroll all our parameters dictionary from a single vector satisfying our specific required shape. |
|  |  |
|  | start=0 |
|  | parameters1 = {} |
|  | #For key |
|  | for key in parameters: |
|  | (a,b) = parameters[key].shape |
|  | # Create a dictionary |
|  | parameters1[key]= theta[start:start+a\*b].reshape((a,b)) |
|  | start=start+a\*b |
|  |  |
|  | return parameters1 |
|  |  |
|  |  |
|  | def vector\_to\_dictionary2(parameters,theta): |
|  | #Unroll all our parameters dictionary from a single vector satisfying our specific required shape. |
|  |  |
|  | start=0 |
|  | parameters2= {} |
|  | # For key |
|  | for key in parameters: |
|  | (a,b) = parameters[key].shape |
|  | # Create a key value pair |
|  | parameters2['d'+key]= theta[start:start+a\*b].reshape((a,b)) |
|  | start=start+a\*b |
|  |  |
|  | return parameters2 |
|  |  |
|  | def gradient\_check\_n(parameters, gradients, train\_X, train\_Y, epsilon = 1e-7,outputActivationFunc="sigmoid"): |
|  | # Set-up variables |
|  | parameters\_values, \_ = dictionary\_to\_vector(parameters) |
|  | grad = gradients\_to\_vector(parameters,gradients) |
|  | num\_parameters = parameters\_values.shape[0] |
|  | J\_plus = np.zeros((num\_parameters, 1)) |
|  | J\_minus = np.zeros((num\_parameters, 1)) |
|  | gradapprox = np.zeros((num\_parameters, 1)) |
|  |  |
|  | # Compute gradapprox using 2 sided derivative |
|  | for i in range(num\_parameters): |
|  | # Compute J\_plus[i]. Inputs: "parameters\_values, epsilon". Output = "J\_plus[i]". |
|  | thetaplus = np.copy(parameters\_values) |
|  | thetaplus[i][0] = thetaplus[i][0] + epsilon |
|  | AL, caches, dropoutMat = forwardPropagationDeep(train\_X, vector\_to\_dictionary(parameters,thetaplus), keep\_prob=1, |
|  | hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  | J\_plus[i] = computeCost(AL, train\_Y, outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | # Compute J\_minus[i]. Inputs: "parameters\_values, epsilon". Output = "J\_minus[i]". |
|  | thetaminus = np.copy(parameters\_values) |
|  | thetaminus[i][0] = thetaminus[i][0] - epsilon |
|  | AL, caches, dropoutMat = forwardPropagationDeep(train\_X, vector\_to\_dictionary(parameters,thetaminus), keep\_prob=1, |
|  | hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  | J\_minus[i] = computeCost(AL, train\_Y, outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | # Compute gradapprox[i] |
|  | gradapprox[i] = (J\_plus[i] - J\_minus[i])/(2\*epsilon) |
|  |  |
|  | # Compare gradapprox to backward propagation gradients by computing difference. |
|  | numerator = np.linalg.norm(grad-gradapprox) |
|  | denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox) |
|  | difference = numerator/denominator |
|  |  |
|  |  |
|  | if difference > 1e-5: |
|  | print ("\033[93m" + "There is a mistake in the backward propagation! difference = " + str(difference) + "\033[0m") |
|  | else: |
|  | print ("\033[92m" + "Your backward propagation works perfectly fine! difference = " + str(difference) + "\033[0m") |
|  | print(difference) |
|  | print("\n") |
|  | # Covert grad to dictionary |
|  | m=vector\_to\_dictionary2(parameters,grad) |
|  | print("Gradients from backprop") |
|  | print(m) |
|  | print("\n") |
|  | # Convert gradapprox to dictionary |
|  | n=vector\_to\_dictionary2(parameters,gradapprox) |
|  | print("Gradapprox from gradient check") |
|  | print(n) |
|  |  |
|  |  |
|  |  |

TestCases.py

|  |
| --- |
| import numpy as np |
|  |  |
|  | def gradient\_check\_n\_test\_case(): |
|  | np.random.seed(1) |
|  | x = np.random.randn(4,3) |
|  | y = np.array([1, 1, 0]) |
|  | W1 = np.random.randn(5,4) |
|  | b1 = np.random.randn(5,1) |
|  | W2 = np.random.randn(3,5) |
|  | b2 = np.random.randn(3,1) |
|  | W3 = np.random.randn(1,3) |
|  | b3 = np.random.randn(1,1) |
|  | parameters = {"W1": W1, |
|  | "b1": b1, |
|  | "W2": W2, |
|  | "b2": b2, |
|  | "W3": W3, |
|  | "b3": b3} |
|  |  |
|  |  |
|  | return x, y, parameters |

After spending a better part of 3 days, I now realize how critical Gradient Check is for ensuring the correctness of you implementation. Initially I was getting very high difference and did not know how to understand the results or debug my implementation. After many hours of staring at the results, I  was able to finally arrive at a way, to localize issues in the implementation. In fact, I did catch a small bug in my Python code, which did not exist in the R and Octave implementations. I will demonstrate this below

**1.1a Gradient Check – Sigmoid Activation – Python**

import numpy as np

import matplotlib

exec(open("DLfunctions8.py").read())

exec(open("testcases.py").read())

#Load the data

train\_X, train\_Y, test\_X, test\_Y = load\_dataset()

#Set layer dimensions

layersDimensions = [2,4,1]

parameters = initializeDeepModel(layersDimensions)

#Perform forward prop

AL, caches, dropoutMat = forwardPropagationDeep(train\_X, parameters, keep\_prob=1, hiddenActivationFunc="relu",outputActivationFunc="sigmoid")

#Compute cost

cost = computeCost(AL, train\_Y, outputActivationFunc="sigmoid")

print("cost=",cost)

#Perform backprop and get gradients

gradients = backwardPropagationDeep(AL, train\_Y, caches, dropoutMat, lambd=0, keep\_prob=1, hiddenActivationFunc="relu",outputActivationFunc="sigmoid")

epsilon = 1e-7

outputActivationFunc="sigmoid"

# Set-up variables

# Flatten parameters to a vector

parameters\_values, \_ = dictionary\_to\_vector(parameters)

#Flatten gradients to a vector

grad = gradients\_to\_vector(parameters,gradients)

num\_parameters = parameters\_values.shape[0]

#Initialize

J\_plus = np.zeros((num\_parameters, 1))

J\_minus = np.zeros((num\_parameters, 1))

gradapprox = np.zeros((num\_parameters, 1))

# Compute gradapprox using 2 sided derivative

for i in range(num\_parameters):

# Compute J\_plus[i].

thetaplus = np.copy(parameters\_values)

thetaplus[i][0] = thetaplus[i][0] + epsilon

AL, caches, dropoutMat = forwardPropagationDeep(train\_X, vector\_to\_dictionary(parameters,thetaplus), keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)

J\_plus[i] = computeCost(AL, train\_Y, outputActivationFunc=outputActivationFunc)

# Compute J\_minus[i].

thetaminus = np.copy(parameters\_values)

thetaminus[i][0] = thetaminus[i][0] - epsilon

AL, caches, dropoutMat = forwardPropagationDeep(train\_X, vector\_to\_dictionary(parameters,thetaminus), keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)

J\_minus[i] = computeCost(AL, train\_Y, outputActivationFunc=outputActivationFunc)

# Compute gradapprox[i]

gradapprox[i] = (J\_plus[i] - J\_minus[i])/(2\*epsilon)

# Compare gradapprox to backward propagation gradients by computing difference.

numerator = np.linalg.norm(grad-gradapprox)

denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)

difference = numerator/denominator

#Check the difference

if difference > 1e-5:

print ("\033[93m" + "There is a mistake in the backward propagation! difference = " + str(difference) + "\033[0m")

else:

print ("\033[92m" + "Your backward propagation works perfectly fine! difference = " + str(difference) + "\033[0m")

print(difference)

print("\n")

# The technique below can be used to identify

# which of the parameters are in error

# Covert grad to dictionary

m=vector\_to\_dictionary2(parameters,grad)

print("Gradients from backprop")

print(m)

print("\n")

# Convert gradapprox to dictionary

n=vector\_to\_dictionary2(parameters,gradapprox)

print("Gradapprox from gradient check")

print(n)

## (300, 2)

## (300,)

## cost= 0.6931455556341791

## [92mYour backward propagation works perfectly fine! difference = 1.1604150683743381e-06[0m

## 1.1604150683743381e-06

##

##

## Gradients from backprop

## {'dW1': array([[-6.19439955e-06, -2.06438046e-06],

## [-1.50165447e-05, 7.50401672e-05],

## [ 1.33435433e-04, 1.74112143e-04],

## [-3.40909024e-05, -1.38363681e-04]]), 'db1': array([[ 7.31333221e-07],

## [ 7.98425950e-06],

## [ 8.15002817e-08],

## [-5.69821155e-08]]), 'dW2': array([[2.73416304e-04, 2.96061451e-04, 7.51837363e-05, 1.01257729e-04]]), 'db2': array([[-7.22232235e-06]])}

##

##

## Gradapprox from gradient check

## {'dW1': array([[-6.19448937e-06, -2.06501483e-06],

## [-1.50168766e-05, 7.50399742e-05],

## [ 1.33435485e-04, 1.74112391e-04],

## [-3.40910633e-05, -1.38363765e-04]]), 'db1': array([[ 7.31081862e-07],

## [ 7.98472399e-06],

## [ 8.16013923e-08],

## [-5.71764858e-08]]), 'dW2': array([[2.73416290e-04, 2.96061509e-04, 7.51831930e-05, 1.01257891e-04]]), 'db2': array([[-7.22255589e-06]])}

**1.1b Gradient Check – Softmax Activation – Python (Error!!)**

In the code below I show, how I managed to spot a bug in your implementation

import numpy as np

exec(open("DLfunctions8.py").read())

N = 100 # number of points per class

D = 2 # dimensionality

K = 3 # number of classes

X = np.zeros((N\*K,D)) # data matrix (each row = single example)

y = np.zeros(N\*K, dtype='uint8') # class labels

for j in range(K):

ix = range(N\*j,N\*(j+1))

r = np.linspace(0.0,1,N) # radius

t = np.linspace(j\*4,(j+1)\*4,N) + np.random.randn(N)\*0.2 # theta

X[ix] = np.c\_[r\*np.sin(t), r\*np.cos(t)]

y[ix] = j

# Plot the data

#plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)

layersDimensions = [2,3,3]

y1=y.reshape(-1,1).T

train\_X=X.T

train\_Y=y1

parameters = initializeDeepModel(layersDimensions)

#Compute forward prop

AL, caches, dropoutMat = forwardPropagationDeep(train\_X, parameters, keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc="softmax")

#Compute cost

cost = computeCost(AL, train\_Y, outputActivationFunc="softmax")

print("cost=",cost)

#Compute gradients from backprop

gradients = backwardPropagationDeep(AL, train\_Y, caches, dropoutMat, lambd=0, keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc="softmax")

# Note the transpose of the gradients for Softmax has to be taken

L= len(parameters)//2

print(L)

gradients['dW'+str(L)]=gradients['dW'+str(L)].T

gradients['db'+str(L)]=gradients['db'+str(L)].T

# Perform gradient check

gradient\_check\_n(parameters, gradients, train\_X, train\_Y, epsilon = 1e-7,outputActivationFunc="softmax")

cost= 1.0986187818144022

2

There is a mistake in the backward propagation! difference = 0.7100295155692544

0.7100295155692544

Gradients from backprop

{'dW1': array([[ 0.00050125, 0.00045194],

[ 0.00096392, 0.00039641],

[-0.00014276, -0.00045639]]), 'db1': array([[ 0.00070082],

[-0.00224399],

[ 0.00052305]]), 'dW2': array([[-8.40953794e-05, -9.52657769e-04, -1.10269379e-04],

[-7.45469382e-04, 9.49795606e-04, 2.29045434e-04],

[ 8.29564761e-04, 2.86216305e-06, -1.18776055e-04]]),

'db2': array([[**-0.00253808**],

[**-0.00505508**],

[ **0.00759315**]])}

Gradapprox from gradient check

{'dW1': array([[ 0.00050125, 0.00045194],

[ 0.00096392, 0.00039641],

[-0.00014276, -0.00045639]]), 'db1': array([[ 0.00070082],

[-0.00224399],

[ 0.00052305]]), 'dW2': array([[-8.40960634e-05, -9.52657953e-04, -1.10268461e-04],

[-7.45469242e-04, 9.49796908e-04, 2.29045671e-04],

[ 8.29565305e-04, 2.86104473e-06, -1.18776100e-04]]),

'db2': array([[-8.46211989e-06],

[-1.68487446e-05],

[ 2.53108645e-05]])}

Gradient Check gives a high value of the difference of 0.7100295. Inspecting the Gradients and Gradapprox we can see there is a very big discrepancy in db2. After I went over my code I discovered that I my computation in the function layerActivationBackward for Softmax was

# Erroneous code

if activationFunc == 'softmax':

dW = 1/numtraining \* np.dot(A\_prev,dZ)

db = np.sum(dZ, axis=0, keepdims=True)

dA\_prev = np.dot(dZ,W)

instead of

# Fixed code

if activationFunc == 'softmax':

dW = 1/numtraining \* np.dot(A\_prev,dZ)

db = **1/numtraining** \* np.sum(dZ, axis=0, keepdims=True)

dA\_prev = np.dot(dZ,W)

After fixing this error when I ran Gradient Check I get

**1.1c Gradient Check – Softmax Activation – Python (Corrected!!)**

import numpy as np

exec(open("DLfunctions8.py").read())

N = 100 # number of points per class

D = 2 # dimensionality

K = 3 # number of classes

X = np.zeros((N\*K,D)) # data matrix (each row = single example)

y = np.zeros(N\*K, dtype='uint8') # class labels

for j in range(K):

ix = range(N\*j,N\*(j+1))

r = np.linspace(0.0,1,N) # radius

t = np.linspace(j\*4,(j+1)\*4,N) + np.random.randn(N)\*0.2 # theta

X[ix] = np.c\_[r\*np.sin(t), r\*np.cos(t)]

y[ix] = j

# Plot the data

#plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)

layersDimensions = [2,3,3]

y1=y.reshape(-1,1).T

train\_X=X.T

train\_Y=y1

#Set layer dimensions

parameters = initializeDeepModel(layersDimensions)

#Perform forward prop

AL, caches, dropoutMat = forwardPropagationDeep(train\_X, parameters, keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc="softmax")

#Compute cost

cost = computeCost(AL, train\_Y, outputActivationFunc="softmax")

print("cost=",cost)

#Compute gradients from backprop

gradients = backwardPropagationDeep(AL, train\_Y, caches, dropoutMat, lambd=0, keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc="softmax")

# Note the transpose of the gradients for Softmax has to be taken

L= len(parameters)//2

print(L)

gradients['dW'+str(L)]=gradients['dW'+str(L)].T

gradients['db'+str(L)]=gradients['db'+str(L)].T

#Perform gradient check

gradient\_check\_n(parameters, gradients, train\_X, train\_Y, epsilon = 1e-7,outputActivationFunc="softmax")

## cost= 1.0986193170234435

## 2

## [92mYour backward propagation works perfectly fine! difference = 5.268804859613151e-07[0m

## 5.268804859613151e-07

##

##

## Gradients from backprop

## {'dW1': array([[ 0.00053206, 0.00038987],

## [ 0.00093941, 0.00038077],

## [-0.00012177, -0.0004692 ]]), 'db1': array([[ 0.00072662],

## [-0.00210198],

## [ 0.00046741]]), 'dW2': array([[-7.83441270e-05, -9.70179498e-04, -1.08715815e-04],

## [-7.70175008e-04, 9.54478237e-04, 2.27690198e-04],

## [ 8.48519135e-04, 1.57012608e-05, -1.18974383e-04]]), 'db2': array([[-8.52190476e-06],

## [-1.69954294e-05],

## [ 2.55173342e-05]])}

##

##

## Gradapprox from gradient check

## {'dW1': array([[ 0.00053206, 0.00038987],

## [ 0.00093941, 0.00038077],

## [-0.00012177, -0.0004692 ]]), 'db1': array([[ 0.00072662],

## [-0.00210198],

## [ 0.00046741]]), 'dW2': array([[-7.83439980e-05, -9.70180603e-04, -1.08716369e-04],

## [-7.70173925e-04, 9.54478718e-04, 2.27690089e-04],

## [ 8.48520143e-04, 1.57018842e-05, -1.18973720e-04]]), 'db2': array([[-8.52096171e-06],

## [-1.69964043e-05],

## [ 2.55162558e-05]])}

**1.2a Gradient Check – Sigmoid Activation – R**

source("DLfunctions8.R")

z <- as.matrix(read.csv("circles.csv",header=FALSE))

x <- z[,1:2]

y <- z[,3]

X <- t(x)

Y <- t(y)

#Set layer dimensions

layersDimensions = c(2,5,1)

parameters = initializeDeepModel(layersDimensions)

#Perform forward prop

retvals = forwardPropagationDeep(X, parameters,keep\_prob=1, hiddenActivationFunc="relu",

outputActivationFunc="sigmoid")

AL <- retvals[['AL']]

caches <- retvals[['caches']]

dropoutMat <- retvals[['dropoutMat']]

#Compute cost

cost <- computeCost(AL, Y,outputActivationFunc="sigmoid",

numClasses=layersDimensions[length(layersDimensions)])

print(cost)

## [1] 0.6931447

# Backward propagation.

gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0, keep\_prob=1, hiddenActivationFunc="relu",

outputActivationFunc="sigmoid",numClasses=layersDimensions[length(layersDimensions)])

epsilon = 1e-07

outputActivationFunc="sigmoid"

#Convert parameter list to vector

parameters\_values = list\_to\_vector(parameters)

#Convert gradient list to vector

grad = gradients\_to\_vector(parameters,gradients)

num\_parameters = dim(parameters\_values)[1]

#Initialize

J\_plus = matrix(rep(0,num\_parameters),

nrow=num\_parameters,ncol=1)

J\_minus = matrix(rep(0,num\_parameters),

nrow=num\_parameters,ncol=1)

gradapprox = matrix(rep(0,num\_parameters),

nrow=num\_parameters,ncol=1)

# Compute gradapprox

for(i in 1:num\_parameters){

# Compute J\_plus[i].

thetaplus = parameters\_values

thetaplus[i][1] = thetaplus[i][1] + epsilon

retvals = forwardPropagationDeep(X, vector\_to\_list(parameters,thetaplus), keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)

AL <- retvals[['AL']]

J\_plus[i] = computeCost(AL, Y, outputActivationFunc=outputActivationFunc)

# Compute J\_minus[i].

thetaminus = parameters\_values

thetaminus[i][1] = thetaminus[i][1] - epsilon

retvals = forwardPropagationDeep(X, vector\_to\_list(parameters,thetaminus), keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)

AL <- retvals[['AL']]

J\_minus[i] = computeCost(AL, Y, outputActivationFunc=outputActivationFunc)

# Compute gradapprox[i]

gradapprox[i] = (J\_plus[i] - J\_minus[i])/(2\*epsilon)

}

# Compare gradapprox to backward propagation gradients by computing difference.

#Compute L2Norm

numerator = L2NormVec(grad-gradapprox)

denominator = L2NormVec(grad) + L2NormVec(gradapprox)

difference = numerator/denominator

if(difference > 1e-5){

cat("There is a mistake, the difference is too high",difference)

} else{

cat("The implementations works perfectly", difference)

}

## The implementations works perfectly 1.279911e-06

# This can be used to check

print("Gradients from backprop")

## [1] "Gradients from backprop"

vector\_to\_list2(parameters,grad)

## $dW1

## [,1] [,2]

## [1,] -7.641588e-05 -3.427989e-07

## [2,] -9.049683e-06 6.906304e-05

## [3,] 3.401039e-06 -1.503914e-04

## [4,] 1.535226e-04 -1.686402e-04

## [5,] -6.029292e-05 -2.715648e-04

##

## $db1

## [,1]

## [1,] 6.930318e-06

## [2,] -3.283117e-05

## [3,] 1.310647e-05

## [4,] -3.454308e-05

## [5,] -2.331729e-08

##

## $dW2

## [,1] [,2] [,3] [,4] [,5]

## [1,] 0.0001612356 0.0001113475 0.0002435824 0.000362149 2.874116e-05

##

## $db2

## [,1]

## [1,] -1.16364e-05

print("Grad approx from gradient check")

## [1] "Grad approx from gradient check"

vector\_to\_list2(parameters,gradapprox)

## $dW1

## [,1] [,2]

## [1,] -7.641554e-05 -3.430589e-07

## [2,] -9.049428e-06 6.906253e-05

## [3,] 3.401168e-06 -1.503919e-04

## [4,] 1.535228e-04 -1.686401e-04

## [5,] -6.029288e-05 -2.715650e-04

##

## $db1

## [,1]

## [1,] 6.930012e-06

## [2,] -3.283096e-05

## [3,] 1.310618e-05

## [4,] -3.454237e-05

## [5,] -2.275957e-08

##

## $dW2

## [,1] [,2] [,3] [,4] [,5]

## [1,] 0.0001612355 0.0001113476 0.0002435829 0.0003621486 2.87409e-05

##

## $db2

## [,1]

## [1,] -1.16368e-05

**1.2b Gradient Check – Softmax Activation – R**

source("DLfunctions8.R")

Z <- as.matrix(read.csv("spiral.csv",header=FALSE))

# Setup the data

X <- Z[,1:2]

y <- Z[,3]

X <- t(X)

Y <- t(y)

layersDimensions = c(2, 3, 3)

parameters = initializeDeepModel(layersDimensions)

#Perform forward prop

retvals = forwardPropagationDeep(X, parameters,keep\_prob=1, hiddenActivationFunc="relu",

outputActivationFunc="softmax")

AL <- retvals[['AL']]

caches <- retvals[['caches']]

dropoutMat <- retvals[['dropoutMat']]

#Compute cost

cost <- computeCost(AL, Y,outputActivationFunc="softmax",

numClasses=layersDimensions[length(layersDimensions)])

print(cost)

## [1] 1.098618

# Backward propagation.

gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0, keep\_prob=1, hiddenActivationFunc="relu",

outputActivationFunc="softmax",numClasses=layersDimensions[length(layersDimensions)])

# Need to take transpose of the last layer for Softmax

L=length(parameters)/2

gradients[[paste('dW',L,sep="")]]=t(gradients[[paste('dW',L,sep="")]])

gradients[[paste('db',L,sep="")]]=t(gradients[[paste('db',L,sep="")]])

#Perform gradient check

gradient\_check\_n(parameters, gradients, X, Y,

epsilon = 1e-7,outputActivationFunc="softmax")

## The implementations works perfectly 3.903011e-07[1] "Gradients from backprop"

## $dW1

## [,1] [,2]

## [1,] 0.0007962367 -0.0001907606

## [2,] 0.0004444254 0.0010354412

## [3,] 0.0003078611 0.0007591255

##

## $db1

## [,1]

## [1,] -0.0017305136

## [2,] 0.0005393734

## [3,] 0.0012484550

##

## $dW2

## [,1] [,2] [,3]

## [1,] -3.515627e-04 7.487283e-04 -3.971656e-04

## [2,] -6.381521e-05 -1.257328e-06 6.507254e-05

## [3,] -1.719479e-04 -4.857264e-04 6.576743e-04

##

## $db2

## [,1]

## [1,] -5.536383e-06

## [2,] -1.824656e-05

## [3,] 2.378295e-05

##

## [1] "Grad approx from gradient check"

## $dW1

## [,1] [,2]

## [1,] 0.0007962364 -0.0001907607

## [2,] 0.0004444256 0.0010354406

## [3,] 0.0003078615 0.0007591250

##

## $db1

## [,1]

## [1,] -0.0017305135

## [2,] 0.0005393741

## [3,] 0.0012484547

##

## $dW2

## [,1] [,2] [,3]

## [1,] -3.515632e-04 7.487277e-04 -3.971656e-04

## [2,] -6.381451e-05 -1.257883e-06 6.507239e-05

## [3,] -1.719469e-04 -4.857270e-04 6.576739e-04

##

## $db2

## [,1]

## [1,] -5.536682e-06

## [2,] -1.824652e-05

## [3,] 2.378209e-05

**1.3a Gradient Check – Sigmoid Activation – Octave**

source("DL8functions.m")

################## Circles

data=csvread("circles.csv");

X=data(:,1:2);

Y=data(:,3);

#Set layer dimensions

layersDimensions = [2 5 1]; #tanh=-0.5(ok), #relu=0.1 best!

[weights biases] = initializeDeepModel(layersDimensions);

#Perform forward prop

[AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X', weights, biases,keep\_prob=1,

hiddenActivationFunc="relu", outputActivationFunc="sigmoid");

#Compute cost

cost = computeCost(AL, Y',outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2)));

disp(cost);

#Compute gradients from cost

[gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y', activation\_caches,forward\_caches, droputMat, lambd=0, keep\_prob=1,

hiddenActivationFunc="relu", outputActivationFunc="sigmoid",

numClasses=layersDimensions(size(layersDimensions)(2)));

epsilon = 1e-07;

outputActivationFunc="sigmoid";

# Convert paramter cell array to vector

parameters\_values = cellArray\_to\_vector(weights, biases);

#Convert gradient cell array to vector

grad = gradients\_to\_vector(gradsDW,gradsDB);

num\_parameters = size(parameters\_values)(1);

#Initialize

J\_plus = zeros(num\_parameters, 1);

J\_minus = zeros(num\_parameters, 1);

gradapprox = zeros(num\_parameters, 1);

# Compute gradapprox

for i = 1:num\_parameters

# Compute J\_plus[i].

thetaplus = parameters\_values;

thetaplus(i,1) = thetaplus(i,1) + epsilon;

[weights1 biases1] =vector\_to\_cellArray(weights, biases,thetaplus);

[AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X', weights1, biases1, keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc);

J\_plus(i) = computeCost(AL, Y', outputActivationFunc=outputActivationFunc);

# Compute J\_minus[i].

thetaminus = parameters\_values;

thetaminus(i,1) = thetaminus(i,1) - epsilon ;

[weights1 biases1] = vector\_to\_cellArray(weights, biases,thetaminus);

[AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X',weights1, biases1, keep\_prob=1,

hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc);

J\_minus(i) = computeCost(AL, Y', outputActivationFunc=outputActivationFunc);

# Compute gradapprox[i]

gradapprox(i) = (J\_plus(i) - J\_minus(i))/(2\*epsilon);

endfor

#Compute L2Norm

numerator = L2NormVec(grad-gradapprox);

denominator = L2NormVec(grad) + L2NormVec(gradapprox);

difference = numerator/denominator;

disp(difference);

#Check difference

if difference > 1e-04

printf("There is a mistake in the implementation ");

disp(difference);

else

printf("The implementation works perfectly");

disp(difference);

endif

[weights1 biases1] = vector\_to\_cellArray(weights, biases,grad);

printf("Gradients from back propagation");

disp(weights1);

disp(biases1);

[weights2 biases2] = vector\_to\_cellArray(weights, biases,gradapprox);

printf("Gradients from gradient check");

disp(weights2);

disp(biases2);

0.69315

1.4893e-005

The implementation works perfectly 1.4893e-005

Gradients from back propagation

{

[1,1] =

5.0349e-005 2.1323e-005

8.8632e-007 1.8231e-006

9.3784e-005 1.0057e-004

1.0875e-004 -1.9529e-007

5.4502e-005 3.2721e-005

[1,2] =

1.0567e-005 6.0615e-005 4.6004e-005 1.3977e-004 1.0405e-004

}

{

[1,1] =

-1.8716e-005

1.1309e-009

4.7686e-005

1.2051e-005

-1.4612e-005

[1,2] = 9.5808e-006

}

Gradients from gradient check

{

[1,1] =

5.0348e-005 2.1320e-005

8.8485e-007 1.8219e-006

9.3784e-005 1.0057e-004

1.0875e-004 -1.9762e-007

5.4502e-005 3.2723e-005

[1,2] =

[1,2] =

1.0565e-005 6.0614e-005 4.6007e-005 1.3977e-004 1.0405e-004

}

{

[1,1] =

-1.8713e-005

1.1102e-009

4.7687e-005

1.2048e-005

-1.4609e-005

[1,2] = 9.5790e-006

}

**1.3b Gradient Check – Softmax Activation – Octave**

source("DL8functions.m")

data=csvread("spiral.csv");

# Setup the data

X=data(:,1:2);

Y=data(:,3);

# Set the layer dimensions

layersDimensions = [2 3 3];

[weights biases] = initializeDeepModel(layersDimensions);

# Run forward prop

[AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X', weights, biases,keep\_prob=1,

hiddenActivationFunc="relu", outputActivationFunc="softmax");

# Compute cost

cost = computeCost(AL, Y',outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2)));

disp(cost);

# Perform backward prop

[gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y', activation\_caches,forward\_caches, droputMat, lambd=0, keep\_prob=1,

hiddenActivationFunc="relu", outputActivationFunc="softmax",

numClasses=layersDimensions(size(layersDimensions)(2)));

#Take transpose of last layer for Softmax

L=size(weights)(2);

gradsDW{L}= gradsDW{L}';

gradsDB{L}= gradsDB{L}';

#Perform gradient check

difference= gradient\_check\_n(weights, biases, gradsDW,gradsDB, X, Y, epsilon = 1e-7,

outputActivationFunc="softmax",numClasses=layersDimensions(size(layersDimensions)(2)));

1.0986

The implementation works perfectly 2.0021e-005

Gradients from back propagation

{

[1,1] =

-7.1590e-005 4.1375e-005

-1.9494e-004 -5.2014e-005

-1.4554e-004 5.1699e-005

[1,2] =

3.3129e-004 1.9806e-004 -1.5662e-005

-4.9692e-004 -3.7756e-004 -8.2318e-005

1.6562e-004 1.7950e-004 9.7980e-005

}

{

[1,1] =

-3.0856e-005

-3.3321e-004

-3.8197e-004

[1,2] =

1.2046e-006

2.9259e-007

-1.4972e-006

}

Gradients from gradient check

{

[1,1] =

-7.1586e-005 4.1377e-005

-1.9494e-004 -5.2013e-005

-1.4554e-004 5.1695e-005

3.3129e-004 1.9806e-004 -1.5664e-005

-4.9692e-004 -3.7756e-004 -8.2316e-005

1.6562e-004 1.7950e-004 9.7979e-005

}

{

[1,1] =

-3.0852e-005

-3.3321e-004

-3.8197e-004

[1,2] =

1.1902e-006

2.8200e-007

-1.4644e-006

}

**2.1 Tip for tuning hyperparameters**

Deep Learning Networks come with a large number of hyper parameters which require tuning. The hyper parameters are

1. \alpha-learning rate  
2. Number of layers  
3. Number of hidden units  
4. Number of iterations  
5. Momentum – \beta– 0.9  
6. RMSProp – \beta_{1}– 0.9  
7. Adam – \beta_{1},\beta_{2} and \epsilon  
8. learning rate decay  
9. mini batch size  
10. Initialization method – He, Xavier  
11. Regularization

– Among the above the most critical is learning rate decay. Rather than just trying out random values, it may help to try out values on a logarithmic scale. So we could try out  
values -0.01,0.1,1.0,10 etc. If we find that the cost is between 0.01 and 0.1 we could use a technique similar to binary search, so we can try 0.01, 0.05. If we need to be bigger than 0.01 and 0.05 we could try 0.25  and then keep halving the distance etc.  
– The performance of Momentum and RMSProp are very good and work well with values 0.9. Even with this, it is better to try out values of 1-\beta in the logarithmic range. So 1-\beta could 0.001,0.01,0.1 and hence \betawould be 0.999,0.99 or 0.9  
– Increasing the number of hidden units or number of hidden layers need to be done gradually. I have noticed that increasing number of hidden layers heavily does not improve performance and sometimes degrades it.  
– Sometimes, I tend to increase the number of iterations if I think I see a steady decrease in the cost for a certain learning rate  
– It may also help to add learning rate decay if you see there is an oscillation while it decreases.  
– Xavier and He initializations also help in a fast convergence and are worth trying out.

**3.1 Final thoughts**

As I come to a close in this Deep Learning Series from first principles in Python, R and Octave, I must admit that I learnt a lot in the process.

\* Building a L-layer, vectorized Deep Learning Network in Python, R and Octave was extremely challenging but very rewarding  
\* One benefit of building vectorized versions in Python, R and Octave was that I was looking at each function that I was implementing thrice, and hence I was able to fix any bugs in any of the languages  
\* In addition since I built the generic L-Layer DL network with all the bells and whistles, layer by layer I further had an opportunity to look at all the functions in each successive post.  
\* Each language has its advantages and disadvantages. From the performance perspective I think Python is the best, followed by Octave and then R  
\* Interesting, I noticed that even if small bugs creep into your implementation, the DL network does learn and does generate a valid set of weights and biases, however this may not be an optimum solution. In one case of an inadvertent bug, I was not updating the weights in the final layer of the DL network. Yet, using all the other layers, the DL network was able to come with a reasonable solution (maybe like random dropout, remaining units can still learn the data!)  
\* Having said that, the Gradient Check method discussed and implemented in this post can be very useful in ironing out bugs.

**Conclusion**

These last couple of months when I was writing the posts and the also churning up the code in Python, R and Octave were  very hectic. There have been times when I found that implementations of some function to be extremely demanding and I almost felt like giving up. Other times, I have spent quite some time on an intractable DL network which would not respond to changes in hyper-parameters. All in all, it was a great learning experience. I would suggest that you start from my first post Deep Learning from first principles in Python, R and Octave-Part 1 and work your way up. Feel free to take the code apart and try out things. That is the only way you will learn.