Imperial College London

Coursework

Imperial College London

Department of Bioengineering

Coursework 2

Author:
Georges Nomicos (CID: 01571775)

Date: January 5, 2019

1. To choose an appropriate classifier, the size of the available data is considered first. The number of features and distinct class are also taken into account. The neural network method does not require extensive knowledge of the distribution of the data. Neural networks does not make any assumptions about the feature dependence and in contrast captures the relationship between features. Convergence gives an accurate indications when the training is complete. Common issues with neural networks such as overfitting can be solved by using regularization and dropout (or batch normalization) methods, local optima problem can be avoided by using state of the art gradient descent algorithm (Adam, a combination of RMSprop and momentum). Hyperparameters search remains one of the main drawback, where grid search and random search are used (for a chosen range) or Bayesian optimization.

   For the training and evaluation of the model the data is shuffled to produce a general model and avoid any biases. The data is then divided in training, validation and testing data (proportion of 80:10:10 respectively). The model parameters will be learned from the training data. While the validation provides in parallel to training the accuracy of the model every chosen iterations. This allows the inclusion of early-stopping which decreases running time and reduce overfitting. The input features are normalized to speed up training time. The mean of each feature vector is subtracted to each sample of the feature vectors and then divided by the standard deviation of each feature vector. It is beneficial later: producing a less elongated gradient descent search and thus the freedom of using higher learning rate. Figure 1 shows the classification pipeline starting from the top with training data which is fed to our neural network and compared to the label at each forward pass. The weights and biases are updated by backpropagating the error and using a gradient descent algorithm to minimize the loss iteratively. Once the model parameters are learned, the latter are stored and can be used to predict the label of new data. The training phase can be computationally expensive but the prediction is done instantly (one forward pass through the network).

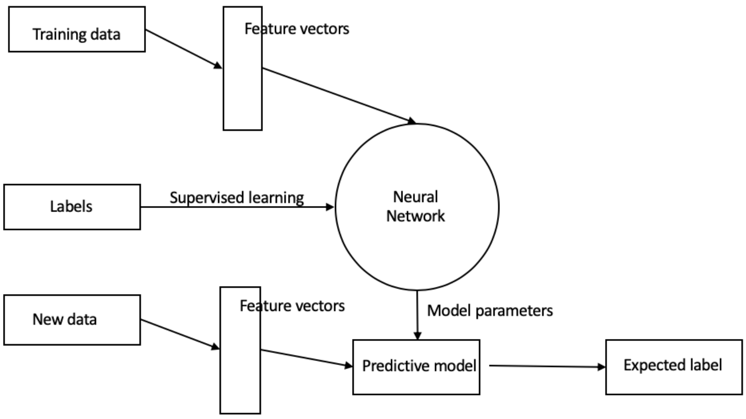

Figure 1: Classification pipeline

2. The classifier chosen is a 5 layer artificial neural network with 80 neurons on the hidden layers. Activation function are ReLU for the first 4 layers and the last one is softmax to produce a readable probabilistic outpout. Stochastic gradient descent with Adam optimization is implemented to update the weights and biases. Mini-batch is used during training to reduce running time instead of feeding the whole data to the neural network every forward/backward pass. Additionally early-stopping helped reduce overfitting with L2 regularization.

3. The classifier produced an accuracy of 99.5% on the test data set and took 30 seconds to train. As specified early when modelling the neural network the data was divided in three different sets all containing non identical samples, training, validation and testing. Testing set was left to test the performance of our model on unseen data. The validation is used during the training to calculate the accuracy at each forward pass. The latter is used for early stopping and hyperparameters optimization. The training loss was calculated using cross-entropy. The loss was used as an indication of convergence and should approach 0. The stochastic behaviour is caused by the mini-batch algorithm. The validation accuracy was calculated every 20 iterations. Figure 2 shows the validation and training accuracy, and the training loss. Both validation and training curves shows no overfitting and convergence to accuracy above 95% after just 200 iterations (a bit more than 1 epoch). The maximum validation accuracy is reached after 5 epochs.

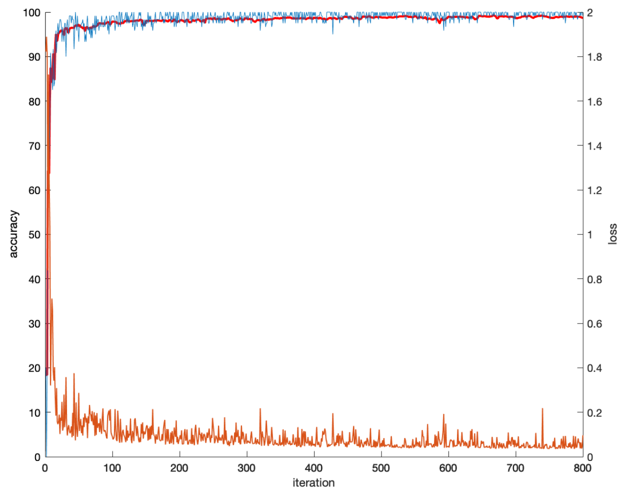

Figure 2: Performance of the classifier (red curve: validation accuracy, blue curve: training accuracy and orange curve: training loss)

A grid search was used to determine the best hyperparameters. The range tested for these hyperparameters are included in Table 1.

Table 1: Hyperparameters optimization range

Hyperparameters	**Range**
Layers	[2,3,4,5]
Neurons	[30,50,80,100]
Regularization	[10^{-5},10^{-4},10^{-3},10^{-2}]
Epoch	[1,3,5,7]
Learning Rate	[10^{-4},10^{-3},10^{-2},10^{-1}]

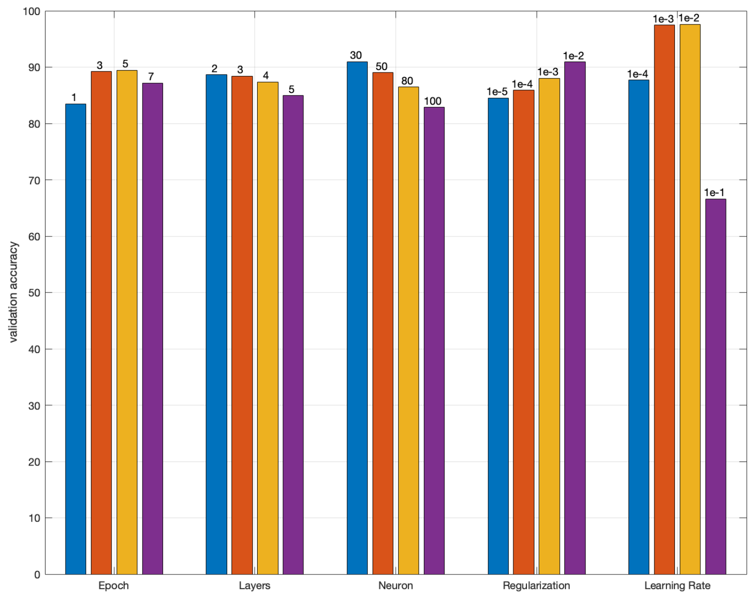

Figure 3: Mean of the validation accuracy of each individual hyperparameter value for all combinations

In total 1024 combinations were tested, taking 2 hours to run. Figure 3 shows the mean of validation accuracy for each individual hyperparameters of each category. The learning rate produces the most variance in accuracy and mainly affects the accuracy. For the number of epoch, 1 is obviously not enough and 7 is too many (overfit).

The combinations above 99% were selected and are displayed in Table 2. Learning rate and regularization almost all similar. An additional refined grid search could be performed for number of neurons and layers but the accuracy produced is sufficiently high. The last row in Table 3 (Appendix) is chosen for the hyperparameters of our model. Figure 4 contains the accuracies for these hyperparameters.

One of the disadvantages of neural networks is the amount of hyperparameters requiring fine tuning. Out of 8 hyperparameters, 5 were fixed and the others tuned with grid search taking 2 hours. Training time can be time consuming as well, but algorithm like mini-batch and normalizing the data reduces the process. Neural networks cannot be interpreted compared to other methods like decision tree, so it might be difficult or impossible to understand its performance.

When properly implemented and using state of the art algorithm neural network can be very accurate with an accuracy above 99% and training time of 30 seconds. The classification time is considerably shorter time compared to a method like KNN which also requires to store the whole training set for classification. Once hyperparameter are tuned and model is trained, the model predicts the label instantly. Neural networks can fit any data with the drawback of hyperparameters tuning.

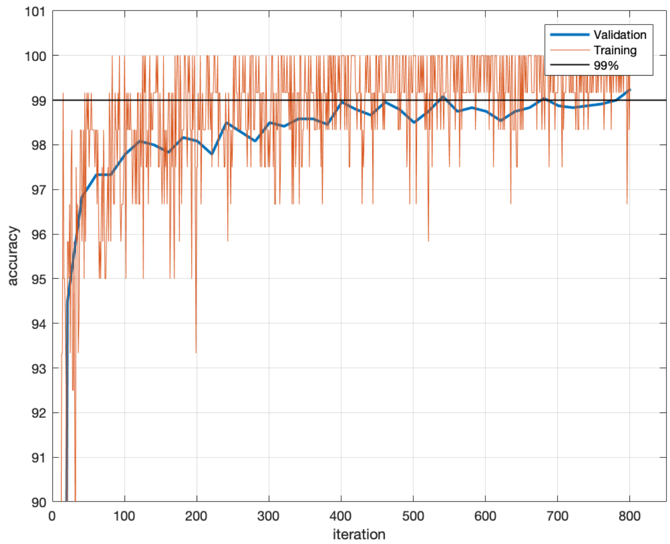

Figure 4: Accuracy of the hyperparameter combination giving the highest validation accuracy

Table 2: Confusion matrix

		Actual class				
		1	**2**	**3**	**4**	**5**
Predicted class	**1**	579	0	0	0	0
	2	0	637	2	0	0
	3	0	0	540	1	0
	4	0	0	3	399	9
	5	0	0	0	1	232

Appendix

Table 3: Combination of hyperparameters resulting a validation accuracy above 99%

Validation Accuracy	Layers	Neurons	Regularization	Epoch	Learning Rate
99.041	5	80	1e-4	7	1e-2
99.041	3	80	1e-4	5	1e-2
99.041	3	80	1e-4	7	1e-2
99.041	4	80	1e-4	7	1e-2
92.082	2	100	1e-4	5	1e-2
99.082	3	50	1e-3	7	1e-2
99.124	2	80	1e-4	7	1e-2
99.166	4	80	1e-3	7	1e-2
99.166	4	100	1e-4	7	1e-2
99.207	5	100	1e-4	7	1e-2
99.249	5	80	1e-4	5	1e-2

Matlab Code

```
%% Train/Vald/Test.
%%Training and validation in parallel. Test on the model in the end.
clear;
clc;

tic

load('data.mat');
[X,Y,X_V,Y_V,Test_X,Test_Y] = PreProcessing(data,'nequal');
X_size = size(X,2);
[param,W,B,Ad] = initialization(5,[80 80 80 80 5],X_size,1e-2,...
1e-4,0.12,40,5,120); %(Number_of_layer,Neuron_layer,X_size,Learning_rate,...
%Regularization,Std_weight,patience,epoch,batchsize) (3,[60 60 5],X_size,1e-3,...
%1e-3,0.12,40,38,100)

[Loss,accuracy_training,accuracy_val,W,B] = training(X,Y,param,W,B,'adam',Ad,X_V,Y_V);
parameters = struct();
parameters.W = W;
parameters.B = B;
parameters.Number_of_layer = param.Number_of_layer;
[error,Conf] = prediction(Test_X,Test_Y,parameters);

toc

%% Hyperparameter optimization
clear;
clc;
tic

load('data.mat');
[X,Y,X_V,Y_V,Test_X,Test_Y] = PreProcessing(data,'nequal');
X_size = size(X,2);

%Hyperparameters to be chosen:
%Learning rate, regularization factor, std for weight init, number of
%hidden layers, number of neuron per layer, batch size or number of epoch,

```

```
Neurons = [30 50 80 100];
Layers_ = [2,3,4,5]; %for 50 neurons per layer
ep = [1,3,5,7];
Learning = [1e-4,1e-3,1e-2,1e-1];
Regularization_ = [1e-5,1e-4,1e-3,1e-2];

std_weight = [0.01,0.05,0.1,0.2];
batch_size = [50,80,100,150];

Loss = containers.Map('UniformValues',false);
accuracy_training = containers.Map('UniformValues',false);
accuracy_val = containers.Map('UniformValues',false);


%Determine how many neuron per layer assuming we have an uniform number of
%neuron per layer
count = 0;
for a = 1:length(ep)
    epoch = ep(1,a);
    for b = 1:length(Layers_)
        Layers = Layers_(1,b);
        for c = 1:length(Neurons)
            n_f = Neurons(1,c);
            N_Layer = {[n_f 5],[n_f n_f 5],[n_f n_f n_f 5],[n_f n_f n_f n_f 5]}; %30 50 80 100
            NN = N_Layer{Layers-1};
            for d = 1:length(Regularization_)
                Regularization = Regularization_(1,d);
                for e = 1:length(Learning)
                    Learning_rate = Learning(1,e);
                    algo = 'adam';
                    Std_weight = 0.12;
                    patience = 30;
                    batchsize = 120;

                    [param,W,B,Ad] = initialization(Layers,NN,X_size,Learning_rate,...
                     Regularization,Std_weight,patience,epoch,batchsize); %(Number of layer, Number of neuron per layer)
                    [Loss,accuracy_training,accuracy_val,W,B] = training(X,Y,param,W,B,'adam',Ad,X_V,Y_V);
                    count = count + 1;
                    disp(count)

                    LossS{count} = Loss;
                    accuracy_trainingS{count} = accuracy_training;
                    accuracy_valS{count} = accuracy_val;
                    HyperPara{count} = {epoch,Layers,n_f,Regularization,Learning_rate};
                end
            end
        end
    end
end
%% Hyperparameter optimization results processing

%Search for hyperparameter giving 99% above accuracy
count = 0;
for k = 1:length(HyperPara)
    valid = accuracy_valS{k};
    valid = valid(:,length(valid));
```

```
        all_v(:,k) = valid;
        if valid >= 99
            count = count + 1;
            index(:,count) = k;
        end
end
max_v = max(all_v(:,index)) %Max accuracy
index_max = find(all_v == max_v)
HyperPara{index_max} %Hyperparameters for max accuracy

Param = [1,3,5,7; %epoch
        2,3,4,5; %layers
        30 50 80 100; %neuron
        1e-5,1e-4,1e-3,1e-2; %reg
        1e-4,1e-3,1e-2,1e-1]; %learning
sum_v = zeros(5,4);

Param_s = {'1','3','5','7'; %epoch
        '2','3','4','5'; %layers
        '30' '50' '80' '100'; %neuron
        '1e-5','1e-4','1e-3','1e-2'; %reg
        '1e-4','1e-3','1e-2','1e-1'};

for k = 1:5
        P = Param(k,:);
        for i = 1:4
            P_c = P(:,i);
            count = 0;
            clear index_all
            for H_l = 1:length(HyperPara)
                HP = HyperPara{H_l};
                HP = cell2mat(HP);
                index_ = find(HP(:,k)==P_c);

                if isempty(index_)
                    continue
                else
                    count = count + 1;
                    index_all(:,count) = H_l;
                end
            end

            for j = 1:length(index_all)
                v = accuracy_valS{index_all(:,j)};
                v_max = v(:,length(v));
                sum_v(k,i) = sum_v(k,i) + v_max;
            end
            sum_v(k,i) = sum_v(k,i)/length(index_all);
        end

end


%Bar chart for individual of each hyperpara in each category
hB=bar(sum_v);           % use a meaningful variable for a handle array...
hAx=gca;                 % get a variable for the current axes handle
hAx.XTickLabel={'Epoch','Layers','Neuron','Regularization','Learning Rate'}; %
    label the ticks
hT=[];                   % placeholder for text object handles

for i=1:length(hB)  % iterate over number of bar objects
```

```
    hT=[hT text(hB(i).XData+hB(i).XOffset-0.01,hB(i).YData,Param_s(:,i), ...
        'VerticalAlignment','bottom','horizontalalign','center')];

end

%Plot max accuracy val adn test
figure
for l = 1:length(index_max)
    val = accuracy_valS{index_max(:,l)};
    tr = accuracy_trainingS{index_max(:,l)};
    plot(1:20:20*length(val),val,'LineWidth',2)
    hold on
    plot(1:length(tr),tr)
    hold on
    plot([1 1100],[99 99],'k','LineWidth',1)
    axis([0 850 90 101])
end

%Display all hyperparameters above 99%
for l = 1:length(index)
    val = accuracy_valS{index(:,l)};
    val = val(:,length(val))
    tr = accuracy_trainingS{index(:,l)};
end
HyperPara{index}

%Validation and loss for best parameters
figure
line(1:length(accuracy_val),accuracy_val,'Color','r','LineWidth',2)
line(1:length(accuracy_training),accuracy_training)

ax1 = gca;
ax1.XColor = 'k';
ax1.YColor = 'k';
ax1_pos = ax1.Position;
ax2 = axes('Position',ax1_pos,...
            'YAxisLocation','right',...
            'Color','none');

line(1:length(Loss),Loss,'Parent',ax2,'Color','b')
%% FUNCTIONS USED ACCROSS
function [X,Y,X_V,Y_V,Test_X,Test_Y] = PreProcessing(data,cl_eq)
%The string can either be 'equal' or 'nequal' to choose wether or not to have
%an equal number of samples for each class. In this function the data is
%split in 3 sets (training,validation,testing). Additionnaly the data is
%normalized according to the training parameters.
label = data(:,1);
input = data(:,2:size(data,2));

%Remove some data to get an equal number of sampling points in each
%class, class 5 being the class with the less samples, it is chosen as the
%number of samples for each class. It can be commented.
if isequal('equal',cl_eq)
    data = []; % Creates an empty data array
    class_l = length(label(label==5)); %number of samples for each class
    start = 1;
    for c = 1:5 %Shuffle data and picks class_l sample points for each class
        label_1 = label==c;
        label_1 = input(label_1,:);
        remove_l = size(label_1,1) - class_l;
        remove = randperm(remove_l);
```

```
        label_1(remove,:) = [];
        till = c * class_l;
        data(start:till,2:65) = label_1;
        data(start:till,1) = c;
        start = 1 + till;
    end
    label = data(:,1); %label for all classes requires another shuffle
    input = data(:,2:size(data,2)); %features (64) for each class
elseif isequal('nequal',cl_eq)
end

numDatapnts = size(label,1); %Total number of samples used for training/
    validation/testing
%The proportion is 80:10:10

s = RandStream('mt19937ar','Seed',1); %Fix a seed
RandStream.setGlobalStream(s)
elems = randperm(numDatapnts);

%To split data in equal sets, we choose 1 and later take a percentage for
%the validation and testing
n = 1;
nDiv = floor(length(elems)/n);
start = 1;
setsData = zeros(nDiv,n);
for j = 1:n
    till = j*nDiv;
    till = floor(till);
    setsData(:,j) = elems(start:till);
    start = till+1;
end

% 1% of the entire data is selected for training
perc = 0.1; %percentage test data
test_n = perc * length(elems);
test_n = floor(test_n); %Take nearest integer
setsData_2 = elems(1:test_n);
elems(1:test_n) = [];
setsData_1 = elems;

%Training/Validation data to be split
X = input(setsData_1,:);
Y = label(setsData_1,:);

%Calculate mean and standard deviation to normalize the data
mean_X = mean(X,1);
std_X = std(X);
X = X - mean_X;
X = X./std_X;

%Validation data, every time the data is split another shuffling is
%applied
numVT = length(Y);
elems_v = randperm(numVT);
perc_v = 0.111; %percentage val data
val_n = perc_v * length(elems_v);
val_n = floor(val_n);
setsData_V = elems_v(1:val_n);
elems_v(1:val_n) = [];
setsData_T = elems_v;

```

```
X_V = X(setsData_V,:); %Validation data
Y_V = Y(setsData_V,:);
X = X(setsData_T,:); %Training data
Y = Y(setsData_T,:);

%Test data which is also normalized by the training mean and std
Test_X = input(setsData_2,:);
Test_Y = label(setsData_2,:);
Test_X = (Test_X - mean_X) ./ std_X;
end

function [param,W,B,Ad] = initialization(Number_of_layer,Neuron_layer,X_size,
    Learning_rate,...
Regularization,Std_weight,patience,epoch,batchsize)
%Hyperparameters are chosen in this function and the memory is allocated
%for the weights,bias and adam parameters matrices.  containers.Map is used
%to store weights,biases and adam parameters. They can be accessed with a
%key (string) and any size cell array can be contained in the same map. It
%also makes the code more readable if string are used to access these values.
std = Std_weight;
param = struct();
param.n = Learning_rate; %learning rate
param.reg = Regularization; %regularization factor
param.epoch = epoch;
param.batchsize = batchsize;
param.patience = patience;
param.Number_of_layer = Number_of_layer;
param.Neuron_layer = Neuron_layer;
param.beta1 = 0.9; %Fixed hyperp
param.beta2 = 0.999; %Fixed hyperp

Ad = containers.Map('UniformValues',false);
Row_w = X_size;
W = containers.Map('UniformValues',false);
B = containers.Map('UniformValues',false);

for N = 1:Number_of_layer

    w = strcat('w', num2str(N));
    b = strcat('b', num2str(N));
    m = strcat('m',num2str(N));
    mt = strcat('mt',num2str(N));
    v = strcat('v',num2str(N));
    vt = strcat('vt',num2str(N));
    m_b = strcat('m_b',num2str(N));
    mt_b = strcat('mt_b',num2str(N));
    v_b = strcat('v_b',num2str(N));
    vt_b = strcat('vt_b',num2str(N));
    Ad(m)=0;Ad(mt)=0;Ad(v)=0;Ad(vt)=0;
    Ad(m_b)=0;Ad(mt_b)=0;Ad(v_b)=0;Ad(vt_b)=0;
    W(w) = std * randn(Row_w,Neuron_layer(N));
    B(b) = zeros(1,Neuron_layer(N));

    Row_w = Neuron_layer(:,N) ;
end
end

function [loss,A] = forward_fnc(param,X,Y,W,B)
%Forward pass is computed in this function. It takes as input the param
%structure, the training input and label, weights and bias containers. It
%returns the loss and the activation function container to be fed to the
```

```
%backward function.

%Activation functions for each hidden layer:
%ReLU - ReLU - ... - ReLU - Softmax

N_l = param.Number_of_layer;
Neur = param.Neuron_layer;
Z = containers.Map('UniformValues',false); %Linear function A * X + B
A = containers.Map('UniformValues',false); % A = f(Z) where f is the activation function

    for N = 1:(N_l-1)
        z = strcat('z', num2str(N)); %z = w*a
        a = strcat('a', num2str(N)); %a = f(z)
        w = strcat('w', num2str(N));
        b = strcat('b', num2str(N));

        Z(z) = X * W(w) + B(b); % X = a where a0 is the training set
        a_to_be = Z(z);
        a_to_be(a_to_be<=0 ) = 0; %ReLU function
        A(a) = a_to_be;

        X = A(a); %for next iteration
    end


%Last layer, softmax instead of ReLU
z = strcat('z', num2str(N+1)); %z = w*a
a = strcat('a', num2str(N+1)); %a = f(z)
w = strcat('w', num2str(N+1));
b = strcat('b', num2str(N+1));

Z(z) = X * W(w) + B(b); %N+1 is the last layer

inter = exp(Z(z));
A(a) = inter./sum(inter,2);

y_hat = A(a);
tmp = y_hat(sub2ind([length(Y) Neur(:,N_l)],(1:numel(Y))',Y(:))); %find the probability of
%the correct class

W_all = values(W);
W_all = cellfun(@(x)x.^2,W_all,'UniformOutput',false); %square all elements of each weight matrix
W_all = sum(cellfun(@(x) sum(x(:)),W_all)); %sum all elements of each weight matrix

loss = sum(-log(tmp))/length(Y) +param.reg*0.5*W_all;
end


function [dW,dB] = backward_fnc(X,Y,A,B,W,param)
%Backpropagation function to calculate the gradient.

N_l = param.Number_of_layer;
dW = containers.Map('UniformValues',false);
dB = containers.Map('UniformValues',false);
dw = strcat('dw', num2str(N_l));
db = strcat('db', num2str(N_l));
a = strcat('a',num2str(N_l));
```

```
a_p = strcat('a',num2str(N_l-1));
w_r = strcat('w',num2str(N_l));
b_r = strcat('b',num2str(N_l));

delta_k = A(a); %f(z) for last layer
delta_k(sub2ind(size(delta_k),(1:numel(Y))',Y(:))) = delta_k(sub2ind(size(
    delta_k),(1:numel(Y))',Y(:)))-1; %(y_hat-y)
delta_k = delta_k/length(Y); %divided by the number of samples
dW(dw) = transpose(A(a_p)) * delta_k + param.reg*W(w_r);
dB(db) = sum(delta_k,1) + param.reg*B(b_r);

k = N_l - 1;

for N = 1:(N_l-2)
dw = strcat('dw', num2str(k));
db = strcat('db', num2str(k));
a = strcat('a',num2str(k));
a_p = strcat('a',num2str(k-1));
w = strcat('w',num2str(k+1));
w_r = strcat('w',num2str(k));
b_r = strcat('b',num2str(k));


delta = delta_k * transpose(W(w));
delta(A(a)<=0) = 0;
dW(dw) = transpose(A(a_p)) * delta + param.reg*W(w_r);
dB(db) = sum(delta,1) + param.reg*B(b_r);

k = k - 1; %to store the gradient decreasingly
delta_k = delta;
end

%First layer backprop
dw = strcat('dw', num2str(1));
db = strcat('db', num2str(1));
w = strcat('w', num2str(2));
a = strcat('a', num2str(1));
w_r = strcat('w',num2str(1));
b_r = strcat('b',num2str(1));

delta = delta_k * transpose(W(w));
delta(A(a) <=0 ) = 0;
dW(dw) = transpose(X) * delta + param.reg*W(w_r);
dB(db) = sum(delta,1) + param.reg*B(b_r);
end

function [Loss,accuracy_training,accuracy_val,W,B] = training(X,Y,param,W,B,
    update,Ad,X_val,Y_val)
%Training function which iterate over a specified number of epoch. Takes as
%input the training input and label, param struct, the initial weight and
%bias, the gradient descent method to be used (update either 'sgd' or
%'adam'), the adam parameters and the validation set. Returns the loss,
%accuracy for both training and validation, weights and biases.
%Mini-batch are used instead of feeding all the training data at each
%forward/backward pass.

%epoch = batchsize * iteration/12000; %one epoch is one full sweep through all
    the data
iteration = (param.epoch*length(Y))/param.batchsize;
X_ini = X; Y_ini = Y;
Number_of_layer = param.Number_of_layer; %Unroll variables from structure
```

```
n_patience = 0;
prev = struct(); %stores W B for the previous iteration in case of early
    stopping
count = 1;

for it = 1:iteration
%     Mini-batch
        shuffle_indexes = randperm(size(X,1));
        shuffle_indexes = shuffle_indexes(1:param.batchsize);
        X_batch = X(shuffle_indexes, :);
        Y_batch = Y(shuffle_indexes);
        X(shuffle_indexes,:) = [];
        Y(shuffle_indexes,:) =  [];

        if size(X,1) < param.batchsize %When no batch can be extracted from
            %the total data set, start a new epoch
            X = X_ini; Y = Y_ini;
        end

    [loss,A] = forward_fnc(param,X_batch,Y_batch,W,B);
    [dW,dB] = backward_fnc(X_batch,Y_batch,A,B,W,param);

    Loss(:,it) = loss;
    parameters = struct();
    parameters.Number_of_layer = Number_of_layer;
    parameters.W = W;
    parameters.B = B;

    %Run this every 100 iterations
    if it == 1
        [error,~] = prediction(X_val,Y_val,parameters);
        accuracy_val(:,1) = error;
    elseif floor(it/1) == it/1
        count = count + 1;
        [error,~] = prediction(X_val,Y_val,parameters);
        accuracy_val(:,count) = error;

        % Early Stopping
        if count == 1 || count == 2 %Does not work for first 2 iterations
            continue
        elseif n_patience == param.patience
            W = prev.W;
            B = prev.B;
            break
        elseif n_patience == 0 && accuracy_val(:,count-1) == error &&
            accuracy_val(:,count-1) > accuracy_val(:,count-2)
            n_patience = 1;
            %Early stopping if the error at this
            %iteration is lower than previous
        elseif n_patience > 0 && accuracy_val(:,count-1) == error
            n_patience = n_patience + 1;
        else
            n_patience = 0;
        end
    end

    a = strcat('a', num2str(Number_of_layer));
    [p,y_train] = max(A(a),[],2);
    error_t = Y_batch - y_train;
    accuracy_training(:,it) = (1-length(error_t(error_t ~= 0))/length(Y_batch))
        *100;
```

```
    if it == 1 %If the ost is three times the previous then the algorithm is
        stopped.
        continue
    elseif loss >= 8 * Loss(:,it-1)
        disp('Cost exploded')
        break
    end

    %Early-stopping to save time. Patience is defined above. If accuracy
    %value is continuously repeated n-patience time then the algo is
    %stopped and returns the previous weight,bias.


    prev.W = W;
    prev.B = B;

    for ng = 1:Number_of_layer %Update the weights and biases for each layer
        w = strcat('w', num2str(ng));
        b = strcat('b', num2str(ng));
        dw = strcat('d',w);
        db = strcat('d',b);

        if isequal(update,'adam') %Adam
            m = strcat('m',num2str(ng));
            mt = strcat('mt',num2str(ng));
            v = strcat('v',num2str(ng));
            vt = strcat('vt',num2str(ng));

            Ad(m) = param.beta1.*Ad(m) + (1-param.beta1).*dW(dw);
            Ad(mt) = Ad(m) ./ (1-param.beta1.^ng);
            Ad(v) = param.beta2*Ad(v) + (1-param.beta2)*(dW(dw).^2);
            Ad(vt) = Ad(v) / (1-param.beta2.^ng);
            W(w) = W(w) - param.n * Ad(mt) ./ (sqrt(Ad(vt)) + 1e-8);

            m_b = strcat('m_b',num2str(ng));
            mt_b = strcat('mt_b',num2str(ng));
            v_b = strcat('v_b',num2str(ng));
            vt_b = strcat('vt_b',num2str(ng));

            Ad(m_b) = param.beta1.*Ad(m_b) + (1-param.beta1)*dB(db);
            Ad(mt_b) = Ad(m_b) ./ (1-param.beta1.^ng);
            Ad(v_b) = param.beta2*Ad(v_b) + (1-param.beta2)*(dB(db).^2);
            Ad(vt_b) = Ad(v_b) / (1-param.beta2.^ng);
            B(b) = B(b) - param.n * Ad(mt_b) ./ (sqrt(Ad(vt_b)) + 1e-9);

        elseif isequal(update,'sgd') %Stochastic gradient descent
            W(w) = W(w) - param.n * dW(dw);
            B(b) = B(b) - param.n * dB(db);
        end

    end

    if mod(it,100) == 0 %Display loss every 100 iteration
        disp(loss)
    end
end
end

function [error,Conf] = prediction(input,label,parameters)
%function to calculate error between input and label for given weight and
```

```
%bias. Returns the error between predicted label and true label as well as
%the confusion matrix.

W = parameters.W;
B = parameters.B;
N_l = parameters.Number_of_layer;
X = input;

for N = 1:(N_l-1)
    w = strcat('w', num2str(N));
    b = strcat('b', num2str(N));

    Z = X * W(w) + B(b); % X = a where a0 is the training set
    a_to_be = Z;
    a_to_be(a_to_be<=0 ) = 0;
    A = a_to_be;

    X = A; %for next iteration
end

%Output layer
w = strcat('w', num2str(N+1));
b = strcat('b', num2str(N+1));

Z = X * W(w) + B(b); %N+1 is the last layer

inter = exp(Z);
A = inter./sum(inter,2);
[p,ypred] = max(A,[],2);

error = label - ypred;
error = (1-length(error(error ~= 0))/length(label))*100;
% disp(error)

%confusion matrix %comment this when training
Conf = zeros(5,5);
for L = 1:length(ypred)
    Conf(ypred(L),label(L)) = 1 + Conf(ypred(L),label(L));
end
% disp(Conf)
end
```