COURSEWORK 2

IMPERIAL COLLEGE LONDON

Classification

Author:
Jonas Tjomsland
Msc Human and Biological Robotics
(CID: 01570830)

Date: May 19, 2020

1.

I started out the coursework by implementing a simple K-nearest (KNN) classifier, just to get an idea of how well the data could be classified. The KNN algorithm performed relatively good, with accuracy above 0.9, but it was quite slow. An additional drawback with KNN is that almost all computation is done during test time, not training (it just memories training data during training). This is not a good trait of a classifier, where we often have restricted computational power and time in test situations.

The above mentioned drawbacks of the KNN, together with the results of papers on human activity recognition (HAR) like (Oniga and József 2015), led me to try a Multilayer Layer Perceptron (MLP). This seemed to be a good choice, based on the results which will be presented later in this report. My MLP have three hidden layers with the ReLU activation function between hidden layers and Softmax out of the last layer. Cross-entropy loss was computed and used for learning, the structure was as follows,

$$\text{Affine - ReLU - Affine - ReLU - Affine - ReLU - Affine - Softmax}$$

$$\text{Affine: } \boldsymbol{Z} = \boldsymbol{X} * \boldsymbol{W} + \boldsymbol{B} \quad \text{ReLU: } \boldsymbol{a} = max(0, \boldsymbol{Z}) \quad \text{Softmax: } \boldsymbol{p} = \frac{e^{\boldsymbol{Z}_j}}{\sum_{k=1}^{K} e^{\boldsymbol{Z}_k}}.$$

This setup seems to be well established as a good approach to a wide variety of ML problems, my choice was mainly based on the theory from CS231 course on Stanford (Li and Yeung 2018).

Before building and training my network I shuffled and split the data in two parts, one large set for training and a smaller one for testing. The test set was put away and not touched before all hyper-parameters had been optimized on the training set which itself where split into training and validation. I used 5-fold cross-validation to be more confident in that my MLP wasn't overfitting. For prepossessing I shuffled the data and standardized it along all features. The Zero-centering removes any bias from the input, avoiding having all positive or all negative input neurons. Such a situation would lead to the weight gradients also being either all positive or negative, which is sub-optimal for gradient descent (zigzag route). The purpose of the normalization was to assure that all features where in the same range and therefore contributed equally.

2.

See the two submitted Matlab files for reference. Several parts of the program have already been explained in Q1 and the code should be well commented. I have commented out the part that tests different hyper-parameters and hard coded the optimized ones. Meticulous testing has assured me that these are optimal and this will be explained more in Q3.

3.

I implemented 5-fold cross-validation for validation and hyper-parameter optimization. The hyper-parameters I have tested are number of epochs, regularization constant, learning rate, dimension of hidden layers and standard deviation for weight initialization. I structured the testing in such a way that one hyper-parameter where varied over the different folds while

the others where fixed, number of epochs where tested every time by storing the results per epoch. I started my search with a relative wide range of values for every parameter, before narrowing the range down to find the optimal value. The initial range of the different parameters where chosen based on sources like the CS231 course on Stanford.

I started with the regularization constant. Its purpose is to make the model more general, hence avoid overfitting training data and perform bad on unseen data. The tables below shows the result from two validation runs, the chosen regularization constant was 0.0001.

Regularization constant:	0	0.0001	0.001	0.01	0.1
Accuracy on validation set:	0.9864	0.9907	0.9896	0.9793	0.6444

(1)

Regularization constant:	0.0001	0.0005	0.001	0.0015	0.01
Accuracy on validation set:	0.9911	0.9893	0.9846	0.9907	0.9789

(2)

Following this I tested the standard deviation used in the weight initialization. We cannot initialize all weights to zero because that would remove the desired asymmetry between neurons. The same approach for testing as above where used and in table 3 and 4 the accuracy with respect to different standard deviations are presented. A standard deviation of 0.05 was deemed to be the optimal one.

Standard deviation:	0.001	0.01	0.1	0.15	0.2
Accuracy on validation set:	0.2646	0.9821	0.9907	0.2607	0.2189

(3)

Standard deviation:	0.015	0.02	0.05	0.1	0.12
Accuracy on validation set:	0.9864	0.9907	0.9929	0.9832	0.3731

(4)

Table 5 and 6 presents the resulting accuracy from different dimensions of the hidden layers, all hidden layers had the same size. A size of 200 neurons where chosen.

Neurons in hidden layers:	100	200	300	400	500
Accuracy on validation set:	0.9921	0.9911	0.9943	0.9889	0.8144

(5)

Neurons in hidden layers:	100	150	200	250	300
Accuracy on validation set:	0.9907	0.9921	0.9921	0.9921	0.9914

(6)

Deciding the optimal number of epochs was a bit tricky. The risk of overfitting increases with higher number of epochs, but in this case it seemed like that wasn't the case. However, number of epochs have a great impact on computation time and there is therefore a trade off between time and accuracy. I decided to set the max number of epochs to 300.

Following this, the learning rate was validated. I plotted loss against number of epochs for a small set of learning rates to see how it affected the performance, see Fig. 1

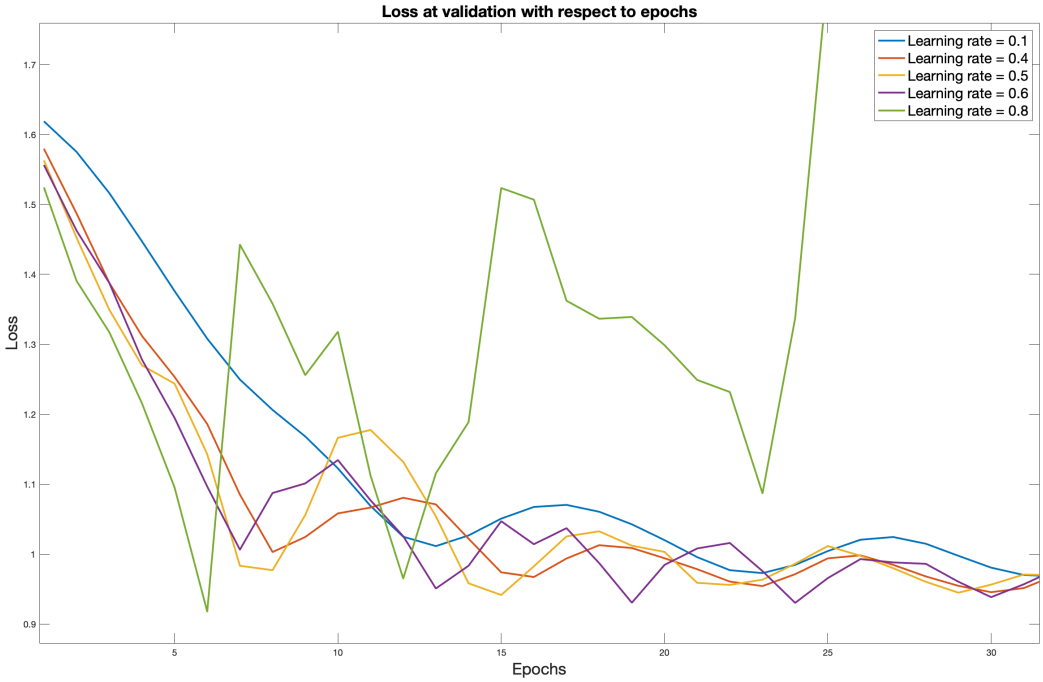

Figure 1: Loss against learning rate for different learning rates.

It is a bit difficult to see, but based on the shape of the curves, a learning rate of 0.8 is too high and a rate of 0.1 is too low. I narrowed down the search and concluded that a learning rate of 0.35 was optimal.

After the hyper-parameter optimization I commented out the cross-validation and split the training data in only two parts, training and validation. I used this setup to produce my confusion matrix and final accuracy. With the entire data set the classifier takes around 80 seconds to train and test result is given instantly. With the presented MLP architecture and hyper-parameters I achieved an accuracy of 0.995. Below, the confusion matrix, with predicted classes column-wise and actual class row-wise, is presented. That the two errors occurred on the classes with noticeable less data than the others are not surprising.

		Predicted class:				
		1	**2**	**3**	**4**	**5**
Actual Class:	**1**	111	0	0	0	0
	2	0	145	0	0	0
	3	0	0	112	0	0
	4	0	0	0	75	1
	5	0	0	0	1	55

Figure 2: Confusion matrix.

The greatest advantaged with the MLP classifier is its swift response at test time. Its greatest disadvantage must be the complexity of implementation and the computational time and power needed for training.

References

Li, Johnson and Serena Yeung (2018). **CS231**. URL: http://cs231n.github.io (visited on 12/19/2018).

Oniga, Stefan and Sütő József (2015). "Optimal recognition method of human activities using artificial neural networks". In: **Measurement Science Review**.

Matlab code

Training function:

```
function parameters = TrainClassifierX(input, label)

% Shuffling data:
[n_data,features] = size(input);
% Shuffled indexes, used for training with same shuffling:
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);
shuffled = randperm(s,n_data)';
shuffled_input = input(shuffled,:);
shuffled_label = label(shuffled);

% Normalize data:
% Calculating mean and std for all features:
means = [];
stds = [];
for feature = 1:features
    means = [means, mean(shuffled_input(:,feature))];
    stds = [stds, std(shuffled_input(:,feature))];
end
% Normalizing every features data:
for i = 1:features
    shuffled_input(:,i) = ((shuffled_input(:,i) - transpose(means(i)))./ ...
        stds(i));
end

% Split data into training and validation sets:
train_X = shuffled_input(1:end-500,:);
validate_X = shuffled_input(end-500:end,:);
train_Y = shuffled_label(1:end-500);
validate_Y = shuffled_label(end-500:end);


% Cross-validation, commented out for submission:

% % K fold cross-validation:
% k = 5;
% n = size(train_X,1);
% n_k = floor(n/k);
% % Split input into k parts:
```

```
% input1 = train_X(1:n_k,:);
% input2 = train_X(n_k+1:2*n_k,:);
% input3 = train_X(2*n_k+1:3*n_k,:);
% input4 = train_X(3*n_k+1:4*n_k,:);
% input5 = train_X(4*n_k+1:end,:);
% % The same for the corresponing labels:
% output1 = train_Y(1:n_k);
% output2 = train_Y(n_k+1:2*n_k);
% output3 = train_Y(2*n_k+1:3*n_k);
% output4 = train_Y(3*n_k+1:4*n_k);
% output5 = train_Y(4*n_k+1:end);
%
% % Place the k parts into cell arrays:
% X = {input1, input2, input3, input4, input5};
% Y = {output1, output2, output3, output4, output5};

% Fixed layer sizes:
input_layer = features;
output_layer = 5;

% Hyperparameters:
% - Number of epochs
% - Regularisation constant
% - Learning rate
% - size of hidden layers
% - Standard deviation for weight initialisation

% Hyper-parameter optimization, commented out for submission:

% Testing every parameter by varying one and fixing the rest. (Always
% testing number of epochs). Comment out part that isn't used:
% Test reg constant:
% reg_constants = [0.0001, 0.0005, 0.001, 0.0015, 0.01];
% % reg_constants = [0, 0.0001, 0.001, 0.01, 0.1];
% learning_rate = 0.3 * ones(1,k);
% hidden_size = {[100, 100, 100],[100, 100, 100], [100, 100, 100],...
% [100, 100, 100],[100, 100, 100]};
% std_weights = 0.1 * ones(1,k);
% Test Learning rate:
% reg_constants = 0.0001 * ones(1,k);
% learning_rate = [0.2, 0.4, 0.5, 0.6, 0.8];
% learning_rate = [0.2, 0.25, 0.3, 0.35, 0.4];
% hidden_size = {[200, 200, 200],[200, 200, 200], [200, 200, 200],...
%[200, 200, 200], [200, 200, 200]};
% std_weights = 0.05 * ones(1,k);
% % Test layer sizes:
% reg_constants = 0.0001 * ones(1,k);
% learning_rate = 0.25 * ones(1,k);
% hidden_size = {[100, 100, 100],[150, 150, 150], [200, 200, 200],...
%[250, 250, 250],[300, 300, 300]};
```

```
% std_weights = 0.05 * ones(1,k);
% % Test std_weights constant:
% reg_constants = 0.0001 * ones(1,k);
% learning_rate = 0.25 * ones(1,k);
% hidden_size = {[100, 100, 100],[100, 100, 100], [100, 100, 100],...
% [100, 100, 100],[100, 100, 100]};
% %std_weights = [0.005, 0.01, 0.1, 0.12, 0.15];
% std_weights = [0.015, 0.02, 0.05, 0.1, 0.12];


% Results of above tested hyper-parameters, hard coded for submission:
reg_constant = 0.0001;
learning_rate = 0.35;
hidden_size = 100;
std_weights = 0.05;
epochs = 300;

% Storing of variables used in validation, commented out for submission:

% Store accuracy and loss for every number of epochs and hyper-parameter:
% all_loss = zeros(epochs, k);
% all_accuracies = zeros(epochs, k);
% training_accuracies = zeros(epochs, k);
% parameters = cell(epochs, k);
% % Arrays used for confusion matrix, stored in a cell array:
% all_correct = cell(1,k);
% all_incorrect = cell(1,k);
% Empty cell array for max accuracy and corresponding n_epochs, every cell
% contains[accuracy, n_epochs]
% fold_max_accuracy = cell(1,k);

% Cross-validation, commented out for submission:

% % Will perform k fold cross-validation:
% for fold = 1:k
%     % Validate on part i, train on the others.
%     validation_X = X{fold};
%     validation_Y = Y{fold};
%     training_X = [];
%     training_Y = [];
%     % Complete training data for current training fold:
%     for j = 1:k
%         % Skip validatioon data:
%         if j == fold
%             continue
%         else
%             training_X = [training_X; X{j}];
%             training_Y = [training_Y; Y{j}];
%         end
%     end
```

```
% Initialize parameters:
param = init_MLP(input_layer, hidden_size, hidden_size, hidden_size,...
                output_layer, std_weights);

% Initialization of variables for gradient descent with momentum:
v = cell(1,length(param));
for parameter = 1:length(param)
    v{parameter} = zeros(size(param{parameter}));
end
% Based on CS231 recommendations:
mu = 0.9;

% Train:
for epoch = 1:epochs

    % Variable stating that we are training:
    validating = 0;

    % Do forward and backward propagation:
    [training_loss, gradients, training_scores] = forward_backward(train_X,...
                                train_Y, param, reg_constant, validating);

    % Update weights and biases
    % Gradient descent with momentum:
    for p = 1:length(param)
        v{p} = mu * v{p} - learning_rate * gradients{p};
        param{p} = param{p} + v{p};
    end

% Storing of variables used in validation, commented out for submission:

    % Store training accuracy:
%     indexes = [];
%     for element = 1:size(training_X,1)
%         [ma, index] = max(training_scores(element,:));
%         indexes = [indexes; index];
%     end

%     diff = indexes - training_Y;
%     idx=diff==0;
%     training_accuracy = sum(idx(:))/length(training_Y);
%     training_accuracies(epoch,fold) = training_accuracy;

    % Variable stating validation:
    validating = 1;

    % Validating:
    [validation_loss, gradients, validation_scores] = forward_backward...
```

```
                (validate_X, validate_Y, param, reg_constant, validating);

% Storing of variables used in validation, commented out for submission:

%     all_accuracies(epoch,fold) = accuracy;
%     all_loss(epoch,fold) = validation_loss;
%     parameters{epoch,fold} = param;

end

% Store validation accuracy:
indexes = [];
for element = 1:size(validate_X)
    [ma, index] = max(validation_scores(element,:));
    indexes = [indexes; index];
end

diff = indexes - validate_Y;
idx=diff==0;
% Printing the accuracy at validation:
validation_accuracy = sum(idx(:))/length(validate_Y)

% Confusion matrix:
% Number of classes:
c = 5;
% Confusion matrix:
confusion_matrix = zeros(c,c);

% Iterate over all predictions:
for pred = 1:length(indexes)
    confusion_matrix(validate_Y(pred),indexes(pred)) = confusion_matrix...
                                        (validate_Y(pred),indexes(pred)) + 1;
end
% Printing confusion matrix:
confusion_matrix

% Storing of variables used in validation, commented out for submission:

%     all_correct{fold} = correct;
%     all_incorrect{fold} = incorrect;


% for folds = 1:k
%     [max_accuracy, n_epochs] = max(all_accuracies(:,folds));
%     fold_max_accuracy{folds} = [max_accuracy, n_epochs];
% end

% array with the best accuracy and corresponding number of epochs:
% max_ac_epochs = [0, 0];
% Best_hyper_p = 1;
```

```
% for ac = 1:length(fold_max_accuracy)
%     if fold_max_accuracy{ac}(1) > max_ac_epochs(1)
%         Best_hyper_p = ac;
%         max_ac_epochs = fold_max_accuracy{ac};
%     end
% end

% for f = 1:k
%     fold_max_accuracy{f}
% end

% Best_hyper_p
% fold_max_accuracy{Best_hyper_p}
% max_ac_epochs(2)
% all_correct{Best_hyper_p}
% all_incorrect{Best_hyper_p}

% Storing the final weights and bias matrices in the variable parameters
% which is returned from the training function:
parameters = param;



% Plot results for different hyper-parameters, commented out for submission
% (Legend defines which hyper-parameter that is tested, change legend when
% testing different hhyper-parameter).
% figure
% plot(1:epochs, all_loss(:,1),'LineWidth', 2)
% hold on
% plot(1:epochs, all_loss(:,2),'LineWidth', 2)
% plot(1:epochs, all_loss(:,3),'LineWidth', 2)
% plot(1:epochs, all_loss(:,4),'LineWidth', 2)
% plot(1:epochs, all_loss(:,5),'LineWidth', 2)
% title('Loss at validation with respect to epochs', 'FontSize', 18)
% xlabel('Epochs', 'FontSize', 18)
% ylabel('Loss', 'FontSize', 18)
% legend('Learning rate = 0.1', 'Learning rate = 0.4', 'Learning rate = 0.5',...
%         'Learning rate = 0.6', 'Learning rate = 0.8', 'FontSize', 16)
%
% figure
% plot(1:epochs, all_accuracies(:,1),'LineWidth', 2)
% hold on
% plot(1:epochs, all_accuracies(:,2),'LineWidth', 2)
% plot(1:epochs, all_accuracies(:,3),'LineWidth', 2)
% plot(1:epochs, all_accuracies(:,4),'LineWidth', 2)
% plot(1:epochs, all_accuracies(:,5),'LineWidth', 2)
% title('Accuracy at validation with respect to epochs', 'FontSize', 18)
% xlabel('Epochs', 'FontSize', 18)
% ylabel('Accuracy', 'FontSize', 18)
% legend('Learning rate = 0.1', 'Learning rate = 0.4', 'Learning rate = 0.5',...
```

```
%         'Learning rate = 0.6', 'Learning rate = 0.8', 'FontSize', 16)
%
% figure
% plot(1:epochs, training_accuracies(:,1),'LineWidth', 2)
% hold on
% plot(1:epochs, training_accuracies(:,2),'LineWidth', 2)
% plot(1:epochs, training_accuracies(:,3),'LineWidth', 2)
% plot(1:epochs, training_accuracies(:,4),'LineWidth', 2)
% plot(1:epochs, training_accuracies(:,5),'LineWidth', 2)
% title('Accuracy at training with respect to epochs',  'FontSize', 18)
% xlabel('Epochs', 'FontSize', 18)
% ylabel('Accuracy', 'FontSize', 18)
% legend('Learning rate = 0.1', 'Learning rate = 0.4', 'Learning rate = 0.5',...
%         'Learning rate = 0.6', 'Learning rate = 0.8', 'FontSize', 16)



% All functions:

% Function that initializes the NN. Takes size of input data, desired
% dimension of the hidden layer,size of the output data  and ...
% standard deviation for weight initialization as input.
% Returns computed parameters.
function param = init_MLP(X_size, H1_size, H2_size, H3_size, num_classes,...
                          init_weight_std)

    % Weight matrix between input and hidden layer:
    W1 = init_weight_std * randn(X_size, H1_size);
    % And biases as zero:
    B1 = zeros(1,H1_size);

    % Weight matrix between hidden and output layer:
    W2 = init_weight_std * randn(H1_size, H2_size);
    % And biases as zero:
    B2 = zeros(1,H2_size);

    % Weight matrix between hidden and output layer:
    W3 = init_weight_std * randn(H2_size, H3_size);
    % And biases as zero:
    B3 = zeros(1,H3_size);

    % Weight matrix between hidden and output layer:
    W4 = init_weight_std * randn(H3_size, num_classes);
    % And biases as zero:
    B4 = zeros(1,num_classes);

    param = {W1, B1, W2, B2, W3, B3, W4, B4};
end
```

```
% Function that computes loss and gradients. Takes
% training data (X),corresponding labels (Y) and validating variable.
% returns loss (scalar value) as well as list of gradients of parameters...
% and scores. If "validating" = 1 it returns after one forward.
function [loss, gradients, scores] = forward_backward(X, y, param,...
                                                reg_const, validating)

    % Implement forward propagation:
    % First layer:
    [z1, cache_first_layer] = linear_forward(X, param{1}, param{2});

    % ReLU function
    [a1, cache_first_relu] = relu_forward(z1);

    % Second layer
    [z2, cache_second_layer] = linear_forward(a1, param{3}, param{4});

    % ReLU function
    [a2, cache_second_relu] = relu_forward(z2);

    % Third layer
    [z3, cache_third_layer] = linear_forward(a2, param{5}, param{6});

    % ReLU function
    [a3, cache_third_relu] = relu_forward(z3);

    % Last layer
    [z4, cache_fourth_layer] = linear_forward(a3, param{7}, param{8});

    scores = z4;

    % Softmax function and cross entropy loss:
    [data_loss, dz4] = softmax_cross_entropy_loss(z4, y);

    % Regularization term (L2):
    reg = 0;
    for par = 1:length(param)
        if mod(par,2) == 0
            continue
        else
        reg = reg + reg_const * sum(sum(param{par}.^2));
        end
    end

    % Total loss:
    loss = data_loss + reg;

        % If validating:
    if validating == 1;
        gradients = 0;
```

```
        return
    end

    % Backward propagation:
    % Back to last hidden layer:
    [da3, dW4, dB4] = linear_backward(dz4, cache_fourth_layer, reg_const);

    % Backward relu:
    dz3 = relu_backward(da3, cache_third_relu);

    [da2, dW3, dB3] = linear_backward(dz3, cache_third_layer, reg_const);

    % Backward relu:
    dz2 = relu_backward(da2, cache_second_relu);

    [da1, dW2, dB2] = linear_backward(dz2, cache_second_layer, reg_const);

    % Backward relu:
    dz1 = relu_backward(da1, cache_first_relu);

    % Back to output
    [dX, dW1, dB1] = linear_backward(dz1, cache_first_layer, reg_const);

    gradients = {dW1, dB1, dW2, dB2, dW3, dB3, dW4, dB4};

end

% Function that computes forward pass for one linear layer. Takes the input
% X, a matrix of weights W and a bias term B. Returns the output z and a
% cache containing X, W & B.
function [z, cache] = linear_forward(X, W, B)

    z = X*W + B;
    cache = {X, W, B};

end

% Function that takes input X of any shape and returns an output of the
% same shape, passed through the relu function, as well as a cache.
function [a, cache] = relu_forward(X)

    a = max(0,X);
    cache = X;

end

% Function that apply the softmax function and computes cross entropy loss.
% Takes an input matrix z_n (output from last layer) of shape (N, C) where
% X[i, j] is the score for the jth class for the ith example and a vector...
% y containing labels.
```

```
function [loss, dz_n] = softmax_cross_entropy_loss(z_n, y)

    % Softmax function:
    probabilities = exp(z_n);
    sums = sum(probabilities,2);
    for prob = 1:size(probabilities, 1)
        probabilities(prob,:) = probabilities(prob,:)/sums(prob);
    end

    % Calculating average cross-entropy loss:
    loss = [];
    for l = 1:size(z_n, 1)
        loss = [loss, -log(probabilities(l,y(l)))];
    end
    loss = mean(loss);

    dz_n = z_n;
    for e = 1:size(z_n, 1)
        dz_n(e,y(e)) = (dz_n(e,y(e)) - 1);
    end
    dz_n = (1/size(z_n, 1)) * dz_n;

end

% Function that computes the backward pass for one linear layer. It takes
% an upstream derivative as well as cache (input and weights for that layer)
% as input and returns the derivatives with respect to a, W & B.
function [da, dW, dB] = linear_backward(d_up, cache, reg_const)

    a = cache{1};
    W = cache{2};
    B = cache{3};

    da = d_up * W';
    dW = a'*d_up + reg_const * W;
    dB = sum(d_up) + reg_const * B;

end

% Function that computes the backward pass for a Relu function. Takes the
% upstream derivative as well as a cache (containing the inital input to the
% relu) as input and returns the derivative with respect to the input.
function dz = relu_backward(d_up, cache)

     z = cache;
     dz = d_up;
     dz(z<=0) = 0;

end
```

Classifier:

```
% MLP classification

function predicted_labels = ClassifyX(input, parameters)

% Normalize data:
% Calculating mean and std for all features:
means = [];
stds = [];
for feature = 1:size(input,2)
    means = [means, mean(input(:,feature))];
    stds = [stds, std(input(:,feature))];
end
% Normalizing every features data:
for i = 1:size(input,2)
    input(:,i) = ((input(:,i) - transpose(means(i)))./ stds(i));
end

scores = forward_backward(input, parameters);

predicted_labels = [];
for i = 1:size(input,1)
    [value, index] = max(scores(i,:));
    predicted_labels = [predicted_labels; index];
end


function scores = forward_backward(X,param)

    % Implement forward propagation:
    % First layer:
    [z1, cache_first_layer] = linear_forward(X, param{1}, param{2});

    % ReLU function
    [a1, cache_first_relu] = relu_forward(z1);


    % Second layer
    [z2, cache_second_layer] = linear_forward(a1, param{3}, param{4});

    % ReLU function
    [a2, cache_second_relu] = relu_forward(z2);

    % Second layer
    [z3, cache_third_layer] = linear_forward(a2, param{5}, param{6});

    % ReLU function
    [a3, cache_third_relu] = relu_forward(z3);

    % Second layer
```

```
    [z4, cache_fourth_layer] = linear_forward(a3, param{7}, param{8});

    scores = z4;

end

% Function that computes forward pass for one linear layer. Takes the input
% X, a matrix of weights W and a bias term B. Returns the output z and a
% cache containing X, W & B.
function [z, cache] = linear_forward(X, W, B)

    z = X*W + B;
    cache = {X, W, B};

end

% Function that takes input X of any shape and returns an output of the
% same shape, passed through the relu function, as well as a cache.
function [a, cache] = relu_forward(X)

    a = max(0,X);
    cache = X;

end
```