MACHINE LEARNING AND PATTERN RECOGNITION
Assignment 1

Matriculation number - s1569105
Examination number - B076165

November 2015

1 The Next Pixel Prediction Task

Note

For all code snippets in the part 1 it is assumed that I load imgregdata.mat file via matlab terminal before I ran the scripts.

1.1 Data preprocessing and visualization

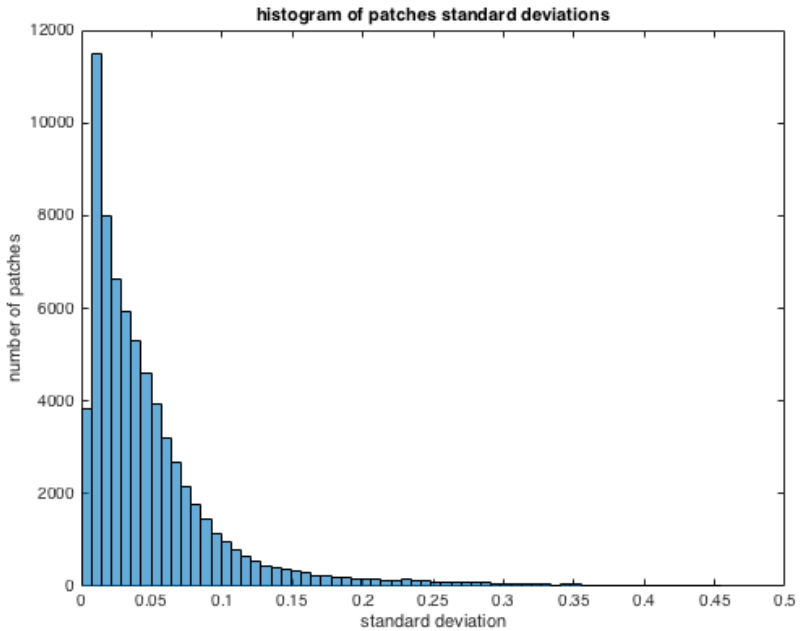


Figure 1: histogram of standard deviations in the xtr dataset after normalisation

(a) The maximum possible value of standard deviation is $\frac{max.-min.}{2}$, so in our case after normalisation it is $\frac{1-0}{2} = 0.5$. Our threshold to distinguish discrete values of pixels is $\frac{1}{64} = 0.5/32 \approx 0.0156$. If we use 32 bins on a range of possible values of standard deviation (between 0 and 0.5) then the width of one bin will be 0.0156 and standard deviations with values 0.0151 or 0.0021 will go to the same bin. But we usually would associate (after rounding using threshold) standard deviation 0.0151 with the discrete (original) pixel value of 1 and 0.0021 with 0 because $round(0.0151/0.0156) = round(0.968) = 1$ and $round(0.0021/0.0156) = round(0.135) = 0$. Therefore, we must choose minimum 64 bins in order to distinguish such cases because we will have bins width $\frac{0.5}{64} \approx \frac{0.0156}{2} = 0.0078$ and each bin will correspond to the specific discrete (original) pixel value.

From the figure 1 we can see that after the peak on the second bin (associate it with deviation around 1 discrete pixel value) the number of patches declines exponentially as standard deviation increases . We can conclude that most of patches have standard deviation within 0 and 0.05 range, and 0.05 is quite small standard deviation, therefore, most of the patches are flat ones.
Code snippet to plot histogram:

```
%launch via - tsk1_1_a(xtr)
function [] = tsk1_1_a(xtr)
    patches = xtr ./ 63;
    patches_std = std(patches,0,2);

    figure;
    histogram(patches_std,64);
    title('histogram of patches standard deviations');
    xlabel('standard deviation');
    ylabel('number of patches');
end
```

(b) I would chose the mean of all pixels in the patch as simple predictor for flat patches. Given the definition of flat patches $the intensity of the pixel_i$ in the flat patch should be something like this $f_i(\boldsymbol{x_{all\ other\ pixels}}) = const_{this flat\ patch} + o(\boldsymbol{x_{all\ other\ pixels}})$ where o is small function in comparison to $const_{flat\ patch}$, and its mean and standard deviation over all pixels in the patch are 0 and as $\sigma_{flat\ patch}$ respectively. For flat patches the following is true $\sigma_{flat\,patch} \leq \sigma_{flat\,pach\,max}$. So it is natural to propose $const_{this flat\ patch}$ as our prediction, which can be received by calculating mean over all pixels in the flat patch
The performance of this simple predictor can be estimated by considering extreme case when after normalisation (all pixel values between 0 and 1) most pixels are zeroes and small portion of pixels are ones (correspond to original intensity of 63). Let $N-m$ be number of zeros and let m be number of ones and I denote μ as mean.

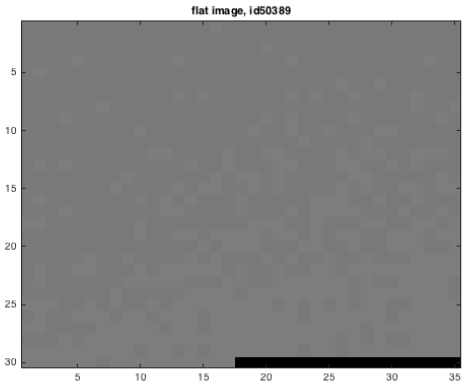
$$
\begin{aligned}
m &< N - m \\
\mu &= \frac{(N-m)*0 + m*1}{N} = \frac{m}{N} \\
\sigma^2 &= (N-m)(0-\frac{m}{N})^2 + m(1-\frac{m}{N})^2 \\
&= \frac{(N-m)m^2}{N^3} + \frac{m(N-m)^2}{N^3} \\
N^3\sigma^2 &= Nm^2 - m^3 + mN^2 - 2m^2N + m^3 \\
&= mN^2 - m^2N \\
m^2 - mN + N^2\sigma^2 &= 0 \\
m = \frac{N}{2}(1 - \sqrt{1-4\sigma^2}) &\quad \text{(minus because our case is } m < N-m \text{)}
\end{aligned}
$$

putting $\sigma_{flat\,pach\,max} = \frac{4}{63} \approx 0.0635$ instead of σ and using $N = 1032$ we get

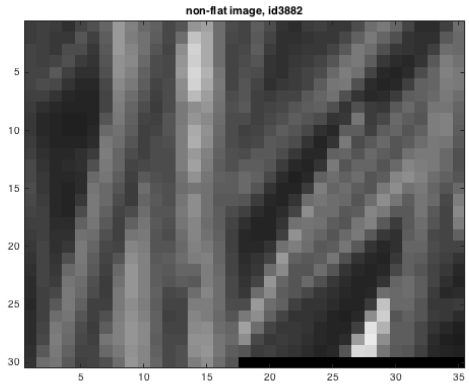
$$m = \frac{1032}{2}(1 - \sqrt{1 - 4 * 0.0635^2}) \approx 4.178$$

rounding m to the closest integer we receive $m = 4$.
Thus, in most extreme case of flat patch we can have 4 ones (correspond to original pixel intensity of 63) and 1028 zeros, so it is natural that we want to predict zero as discrete value of our target pixel. The mean gives us $\mu = \frac{1028*0+1*4}{1032} \approx 0.0038$. Dividing range between 0 and 1 by 64 we get 0.0156 as our threshold to distinguish discrete pixel values. $round(0.0038/0.0156) = round(0.244) = 0$ so our mean value will correspond to 0 as the discrete value of our target pixel and that is what we wanted.

Figure 2: patch images



(a) flat patch image



(b) non-flat patch image

(c) Code snippet to show patch images on figure 2:

```
%launch via tsk1_1_c(xtr)
function [] = tsk1_1_c(xtr)
    %normalising
    patches = xtr ./ 63;
    patches_std = std(patches,0,2);

    %the threshold 4/63 is taken from the task
    flat_threshold = 4/63;
    flat_patches_ids     = patches_std <= flat_threshold;
    non_flat_patches_ids = patches_std >  flat_threshold;

    %split on flat and non-flat patches
    flat_patches     = patches(flat_patches_ids, :);
    non_flat_patches = patches(non_flat_patches_ids,:);

    function [rnd_image, rnd_image_id] = get_rnd_image(patches)
        rnd_image_id = randi(size(patches, 1), 1);
        patch = patches(rnd_image_id, :);
        %expanding patch to the full size
        patch(1050) = 0;
        %reshaping to image
        rnd_image = reshape(patch, [35, 30]);
        %transposing them to ensure right position on the plot
        rnd_image = rnd_image';
    end

    function [] = show_image(image, title_str, image_id)
        figure;
        imagesc(image, [0, 1]);
        title(strcat(title_str, num2str(image_id)));
        colormap gray;
    end

    [flat_image, flat_id] = get_rnd_image(flat_patches);
    [non_flat_image, non_flat_id] = get_rnd_image(non_flat_patches);

    show_image(flat_image, 'flat image, id ', flat_id);
    show_image(non_flat_image, 'non-flat image, id ', non_flat_id);
end
```

1.2 Linear regression with adjacent pixels

(a) I used 5000 training points from xtr_nf and ytr_nf to plot figure 3. From it we can see that x(j, end), x(j, end - 34), y(j) are strongly positively correlated. However, there is some relatively small number of deviations from this trend. It seems that these deviations are normally distributed so linear regression should be reasonable model to describe such data. Code snippet for scatter plot:

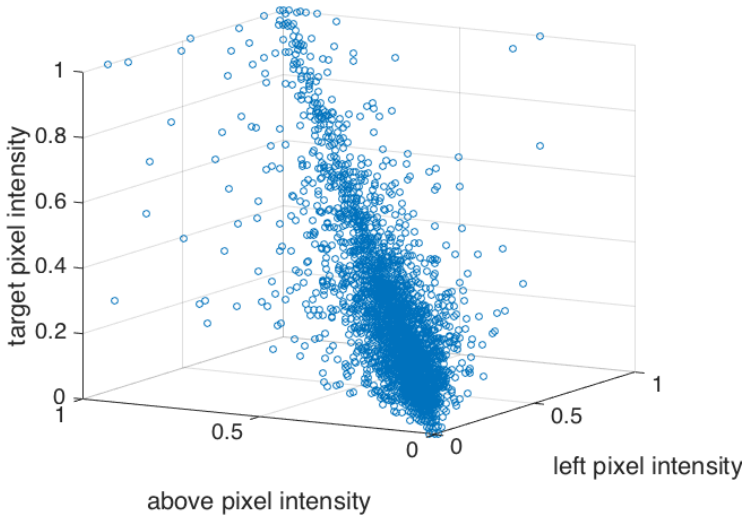


Figure 3: scatter plot of adjacent pixels. 5000 data points from xtr_nf and ytr_nf

```
%launch via - tsk1_2_a(xtr_nf, ytr_nf)
function [] = tsk1_2_a(xtr_nf, ytr_nf)
    %I choose 5000 so plot doesn't look cluttered
    num_data_points = 5000;
    x_left   = xtr_nf(1:num_data_points, 1032);
    x_above  = xtr_nf(1:num_data_points, 1032 - 34);
    x_target = ytr_nf(1:num_data_points);

    scatter3(x_left, x_above, x_target);
    xlabel('left pixel intensity');
    ylabel('above pixel intensity');
    zlabel('target pixel intensity');
    set(gca,'FontSize', 20);
end
```

(b) Derivation of this solution can be taken from MLPR lecture 7 slides 8-11 here. The solution for weights from there is:

$$\hat{\boldsymbol{w}} = (\Phi^T\Phi)^{-1}\Phi^T\boldsymbol{y}$$

In our notation matrix Φ will become:

$$\Phi = X = \begin{pmatrix} 1, x(1,end), x(1,end-34) \\ 1, x(2,end), x(2,end-34) \\ \dots \\ 1, x(N,end), x(N,end-34) \end{pmatrix}$$

$$\hat{\boldsymbol{w}} = (X^TX)^{-1}X^T\boldsymbol{y}$$

where N is a number of training data points and x is our dataset (it will be xtr_nf in the next task)

(c) code snippet to get linear regression predictor (will be used afterwards):

```
function [w, predictor] = cs_linear_regression(x_train, t_train)
    %custom linear regression
    %it inserts bias term automatically

    %adding bias term
    calc_Phi = @(x)[ones(size(x, 1), 1), x];

    %computing weights
    w = pinv(calc_Phi(x_train)) * t_train;

    predictor = @(x_test)calc_Phi(x_test) * w;
end
```

code snippet to compute root mean square error (RMSE) (will be used afterwards):

```
function result = cs_rmse(t, y)
    %custom root mean square error
    diff = t - y;
    %mean square error
    mse = mean(diff .* diff);
    result = mse ^ 0.5;
end
```

code snippet for this task:

```
%launch via - tsk1_2_c(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_2_c(x_all_train, t_train, x_all_test, t_test)
    %t - means target values
    get_adjacent_pixels = @(x) [x(:, 1032), x(:, 1032 - 34)];

    %prepare sets
    x_train = get_adjacent_pixels(x_all_train);
    x_test  = get_adjacent_pixels(x_all_test);

    %train (it will add bias term automatically)
    [w, predictor] = cs_linear_regression(x_train, t_train);
    display(w, 'weights for neighbours pixels features');

    error_train = cs_rmse(t_train, predictor(x_train));
    error_test  = cs_rmse(t_test,  predictor(x_test));

    display(error_train, 'rmse on the training set');
    display(error_test,  'rmse on the test set');

    %show surface
    figure,
    %I chose smaller step because our function is just a plane
    [dim1, dim2] = meshgrid(0:0.1:1, 0:0.1:1);
    %swapped ones from original snippet, because w(1) corresponds to bias
    %in my case
    ysurf = [ones(numel(dim1),1), [dim1(:), dim2(:)]] * w;
    surf(dim1, dim2, reshape(ysurf, size(dim1)));

    hold on;

    %show test set data points
    scatter3(x_test(:, 1), x_test(:, 2), t_test, 'red');
    xlabel('left pixel intensity');
    ylabel('above pixel intensity');
    zlabel('target pixel intensity');
    set(gca,'FontSize', 20);
end
```

after training the weights are:

bias	left pixel	above pixel
0.0026	0.4606	0.5241

the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0506	0.0503

This is not a typo, surprisingly my performance is a bit better on the test set. That is why we can conclude that linear regression is not over-fitting the data in this problem. It can be seen from figure 4 that indeed there is strong positive correlation between adjacent pixels and target value pixel.

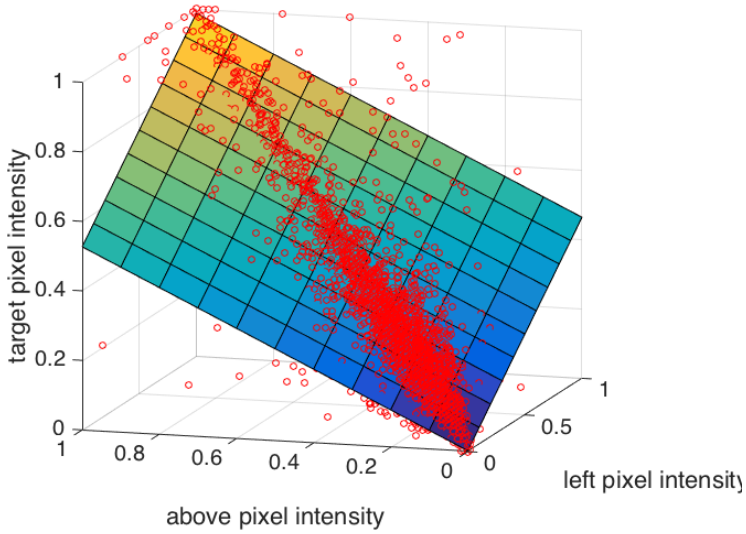


Figure 4: plot of the linear regression function after training along with test data points

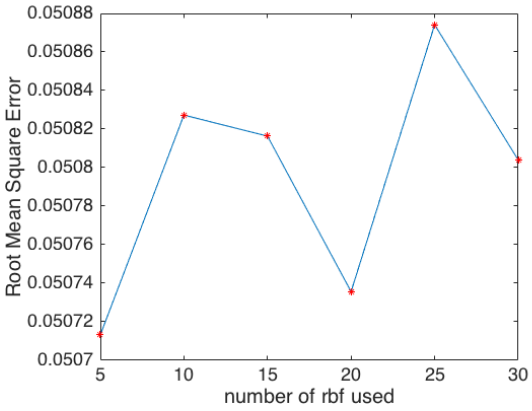
1.3 RBF regression with adjacent pixels

(a) Code snippet for this task:

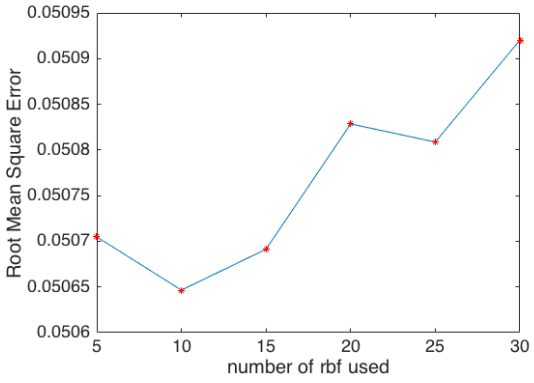
```
%launch via
%tsk1_3_a(xtr_nf, ytr_nf, 5:5:30)
%and
%tsk1_3_a(xtr_nf, ytr_nf, 1:20)
function [] = tsk1_3_a(x_all, t, num_rbfs)
    %t - means target values
    get_adjacent_pixels = @(x) [x(:, 1032), x(:, 1032 - 34)];

    x = get_adjacent_pixels(x_all);

    opt = foptions;
    opt(1)  = 1; % Display EM training
    opt(14) = 5; % number of iterations of EM
    dim     = 2; % left_pixel, above_pixel in our case

    function regf = create_rbf_regf(num_rbf)
        function y_test = rbf_reg(x_train, t_train, x_test)
            net = rbf(dim, num_rbf, 1, 'gaussian');
            net = rbftrain(net, opt, x_train, t_train);
            y_test = rbffwd(net, x_test);
        end
        regf = @rbf_reg;
    end

    errors = zeros(1, length(num_rbfs));

    for i = 1:length(num_rbfs)
        num_rbf = num_rbfs(i);
        regf = create_rbf_regf(num_rbf);
        %default CV 10 folds
        cvMse = crossval('mse', x, t,'predfun',regf);
        errors(i) = cvMse ^ 0.5;
    end

    plot(num_rbfs, errors)
    hold on;
    plot(num_rbfs, errors, 'r*');
    xlabel('number of rbf used');
    ylabel('Root Mean Square Error');
    set(gca,'FontSize', 18);
end
```

When I ran cross validation procedure to determine which number of radial bases functions among { 5, 10, 15, 20, 25, 30} produces the best results, each time I received a different

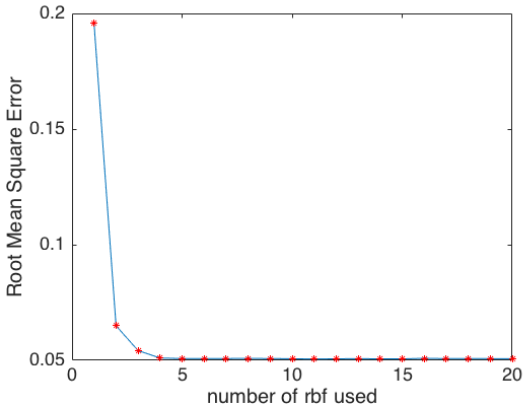
Figure 5: Root Mean Square Error against number of radial basis functions used



(a)



(b)



(c)

answer. The figure 5a suggests 5 as the best number of radial bases functions and 5b proposes 10 as the best choice. This happens probably due to the random numbers as matlab crossval uses them each time to divide input set on training and validation sets and rbf network initialises weights differently depending on the random numbers. After that I launched the procedure for number of radial basis functions between 1 and 20, and I have realised that it was just a matter of scale. The figure 5c demonstrates that we achieve almost no improvement if we use more than 5 radial bases functions in this task. That is why I have chosen 5 as my number of radial basis functions because for the same efficiency it takes less time to compute.

(b) the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0506	0.0503

As it can be seen that radial basis functions don't give any sufficient improvement in comparison to linear regression when both of them are using only adjacent pixels as features. Code snippet for this task:

```
%launch via
%tsk1_3_a(xtr_nf, ytr_nf, 5:5:30)
%and
%tsk1_3_a(xtr_nf, ytr_nf, 1:20)
function [] = tsk1_3_a(x_all, t, num_rbfs)
    %t - means target values
    get_adjacent_pixels = @(x) [x(:, 1032), x(:, 1032 - 34)];

    x = get_adjacent_pixels(x_all);

    opt = foptions;
```

```
        opt(1)  = 1; % Display EM training
        opt(14) = 5; % number of iterations of EM
        dim     = 2; % left_pixel, above_pixel in our case

        function regf = create_rbf_regf(num_rbf)
            function y_test = rbf_reg(x_train, t_train, x_test)
                net = rbf(dim, num_rbf, 1, 'gaussian');
                net = rbftrain(net, opt, x_train, t_train);
                y_test = rbffwd(net, x_test);
            end
            regf = @rbf_reg;
        end

        errors = zeros(1, length(num_rbfs));

        for i = 1:length(num_rbfs)
            num_rbf = num_rbfs(i);
            regf = create_rbf_regf(num_rbf);
            %default CV 10 folds
            cvMse = crossval('mse', x, t,'predfun',regf);
            errors(i) = cvMse ^ 0.5;
        end

        plot(num_rbfs, errors)
        hold on;
        plot(num_rbfs, errors, 'r*');
        xlabel('number of rbf used');
        ylabel('Root Mean Square Error');
        set(gca,'FontSize', 18);
    end
```

1.4 Linear regression with all pixels

the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0371	0.0456

There is an improvement comparing with both previous methods but on the test set it is quite small. Overall, it is possible to conclude that whereas knowledge of all the pixels helps to predict the value of target pixel better, indeed, the most significant features are pixels adjacent to the target one. Code snippet for this task:

```
%launch via - tsk1_4(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_4(x_train, t_train, x_test, t_test)
    %t - means target values

    %train (it will add bias term automatically)
    [w, predictor] = cs_linear_regression(x_train, t_train);

    error_train = cs_rmse(t_train, predictor(x_train));
    error_test  = cs_rmse(t_test,  predictor(x_test));

    display(error_train, 'rmse on the training set');
    display(error_test,  'rmse on the test set');
end
```

1.5 Neural Network with all pixels

(a) the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0333	0.0473

In comparison to linear regression with all pixels Neural Network slightly over-fits the data because its error is lower on training set but the error on test set is larger.
Code snippet for this task:

```
%launch via - tsk1_5_a(net, xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_5_a(net, x_train, t_train, x_test, t_test)
    %t - means target values

    rmse_train = cs_rmse(t_train, mlpfwd(net, x_train));
    rmse_test  = cs_rmse(t_test,  mlpfwd(net, x_test));

    display(rmse_train, 'rmse on the training set');
    display(rmse_test,  'rmse on the test set');
end
```

(b) Neural Network RMSE for different random seeds:

random seed	RMSE on training set	RMSE on test set
2015	0.0500	0.0515
2016	0.0477	0.0504
2017	0.0485	0.0515
2018	0.0477	0.0516
2019	0.0489	0.0527

The average values for different random seeds are 0.0486 and 0.0515 for training and test sets respectively. In comparison to previous task 1.5.a the difference between training and test errors is much smaller in all cases because we are not using all the training instances for learning. At the same time, most likely due to the same reason, our performance on the training and test set has dropped.
Random seed determines to what local minimum NN will try to converge. That is why if we have a lower error on the training set then it is not guaranteed that our test error will be lower, as well. Runs with random seeds 2015 and 2019 are examples of this.
Code snippet for this task:

```
%launched via - tsk1_5_b(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_5_b(x_train, t_train, x_test, t_test)
    %t - means target values
    nhid = 10; % number of hidden units

    % Set up vector of options for the optimiser.
    options = zeros(1,18);
    options(1)  = 1; % This provides display of error values.
    options(9)  = 1; % Check the gradient calculations.
    options(14) = 200; % Number of training cycles.

    function [rmse_train, rmse_test] = launch_NN(seed)
        rng(seed,'twister')
        net = mlp(size(x_train,2), nhid, 1, 'linear');
        [net, tmp] = netopt(net, options, x_train(1:5000,:), t_train(1:5000,:), 'scg');

        rmse_train = cs_rmse(t_train, mlpfwd(net, x_train));
        rmse_test  = cs_rmse(t_test,  mlpfwd(net, x_test));
    end

    seeds  = 2015:1:2019;
    table = zeros(length(seeds), 3);

    for i = 1:length(seeds)
        seed = seeds(i);
        [rmse_train, rmse_test] = launch_NN(seed);
        table(i, :) = [seed, rmse_train, rmse_test];
    end

    %LaTeX table format
    formatSpec = '%4.0f & %5.4f & %5.4f \\\\ \n';
    fprintf(formatSpec, table');
end
```

1.6 Discussion

In previous tasks linear regression with all pixels showed the best performance. That is why I thought that it would be reasonable to try something else with linear regression. It seemed to me that the adjacent pixels have the biggest impact on the intensity of the target pixel. But how many

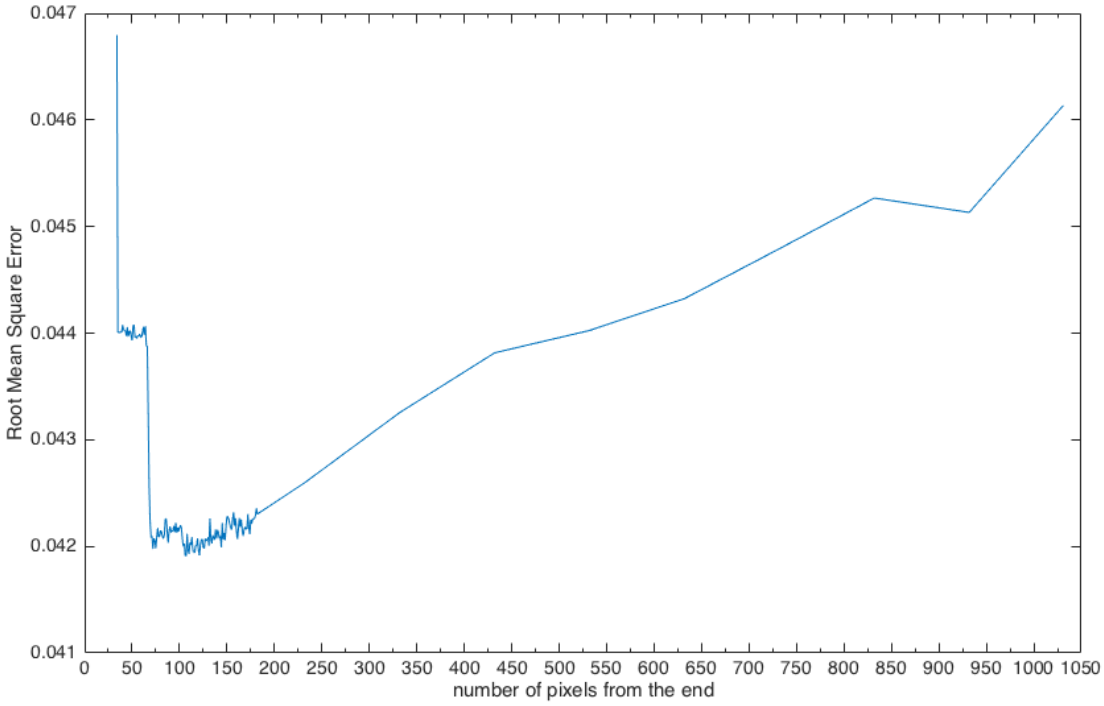


Figure 6: plot of linear regression RMSE depending on the number of pixels from the end used as features.

nearest pixels do we need? I wasn't sure so I evaluated linear regression performance depending on the number of pixels from the end of feature vector via cross-validation on the training set. The result can be seen on the figure 6. **Note**: the plot step is not uniform, after 182 I used much bigger step because otherwise cross-validation would take too much time.
Figure suggests that the best value is something like 120. Using that number of feature on the test set (xte_nf(:, 912:1032)) I received the following RMSE:

	Test set
RMSE	0.0419

This result shows improvement in comparison to the previous models. Also we can see that at the beginning figure 6 has form similar to the step function. I describe it by the fact that with period of 35 we receive pixels which are above the target one and they give a lot of information about target pixels. This leads to conclusion that if image is 2d-array of pixels where i, j are indexes on the X and Y axis respectively then we should select pixels/features depending on their euclidean distance from the target pixel $r = \sqrt{(18-i)^2 + (30-j)^2}$.
Code snippet to get the closest pixels:

```
function x = get_closest_pixels(x_all, maximum_distance)
    function accepted = filter(vector_index)
        i = mod(vector_index, 35);
        j = vector_index / 35;
        r = sqrt((18 - i)^2 + (30 - j)^2);
        accepted = r < maximum_distance;
    end

    indexes = arrayfun(@filter, 1:1032);
    x = x_all(:, indexes);
end
```

I have performed cross-validation on the training set to find best such distance, the results are on figure 7. It points out that the best distance is 4, and after checking it on the test set:

	Test set
RMSE	0.0412

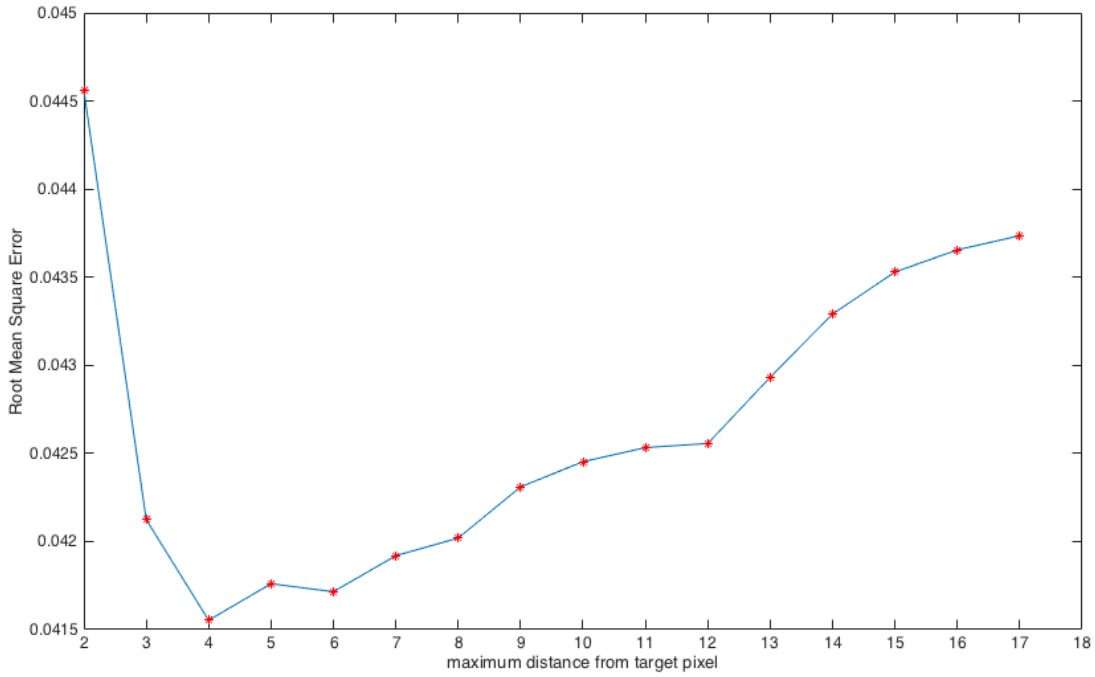


Figure 7: plot of linear regression RMSE depending on the maximum distances from the target pixels within which pixels were used as features.

This is the best result among the models considered and this model is computationally efficient because it needs to take only 21 closest pixels in contrast to linear regression or Neural Network with all the pixels.
Code snippet to plot figure 6:

```
%launch via - tsk1_6_1(xtr_nf, ytr_nf)
function [] = tsk1_6_1(x_all_train, t)
    %t - means target values

    function y_test = regf(x_train, t_train, x_test)
       [w, predictor] = cs_linear_regression(x_train, t_train);
       y_test = predictor(x_test);
    end

    offsets = (1032 - 34):-1:850;
    offsets = [offsets, 800, 700, 600, 500, 400, 300, 200, 100, 1];
    errors = zeros(length(offsets), 1);

    for i = 1:length(offsets)
        offset = offsets(i)
        x = x_all_train(:, offset:1032);
        %default 10 fold
        cvMse = crossval('mse', x, t,'predfun', @regf);
        errors(i) = cvMse ^ 0.5;
    end

    pixels_from_end = 1032 - offsets;
    plot(pixels_from_end, errors);
    xlabel('number of pixels from the end');
    ylabel('Root Mean Square Error');
end
```

Code snippet to get performance on the test set using 120 pixels from the end:

```
%launch via - tsk1_6_2(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_6_2(x_all_train, t_train, x_all_test, t_test)
    %t - means target values
    x_train = x_all_train(:, 912:1032);
    x_test  = x_all_test (:, 912:1032);
```

```
    [w, predictor] = cs_linear_regression(x_train, t_train);

    rmse_test = cs_rmse(t_test, predictor(x_test))
end
```

Code snippet to plot figure 7:

```
%launch via - tsk1_6_3(xtr_nf, ytr_nf)
function [] = tsk1_6_3(x_all_train, t)
    %t - means target values

    function y_test = regf(x_train, t_train, x_test)
       [w, predictor] = cs_linear_regression(x_train, t_train);
       y_test = predictor(x_test);
    end

    max_distances = 2:17;
    errors = zeros(length(max_distances), 1);

    for i = 1:length(max_distances)
        max_dist = max_distances(i)
        x = get_closest_pixels(x_all_train, max_dist);
        %to see complexity of the model
        size(x)
        %default 10 fold
        cvMse = crossval('mse', x, t,'predfun', @regf);
        errors(i) = cvMse ^ 0.5;
    end

    plot(max_distances, errors);
    hold on;
    plot(max_distances, errors, 'r*');
    xlabel('maximum distance from target pixel');
    ylabel('Root Mean Square Error');
end
```

Code snippet to get performance on the test set with pixels within radius 4 from target one:

```
%launch via - tsk1_6_4(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_6_4(x_all_train, t_train, x_all_test, t_test)
    %t - means target values
    x_train = get_closest_pixels(x_all_train, 4);
    x_test  = get_closest_pixels(x_all_test, 4);

    [w, predictor] = cs_linear_regression(x_train, t_train);

    rmse_test = cs_rmse(t_test, predictor(x_test))
end
```

2 Robust modelling

Note

It is assumed that I have loaded the code from part2_code_data.tar.gz in my matlab environment. I have also taken professor Iain Murray code from MLPR tutorial 5 to produce error bars - errobar_str.m. text_data.mat is assumed to be loaded as well.

2.1 Fitting the baseline model

(a) The code snippet to add bias term:

```
%adding bias term
add_bias = @(x)[x, ones(size(x, 1), 1)];

x_train = add_bias(x_train);
x_test  = add_bias(x_test);
```

(b) I have received the following results:

	Test set
RMSE	0.0412