Faculty of Mechanical Engineering and Robotics

Department of Robotics and Mechatronics

Basics of AI and Deep Learning

Course for Mechatronic Engineering with English as instruction language

Instruction 5:

Practical decision system

You will learn: How to perform a typical signal acquisition > feature calculation > feature selection > model training > model optimization procedure based on simulated vibration signals.

Additional materials:

- Course lecture 4 and 5

Course supervisor:
Ziemowit Dworakowski, zdw@agh.edu.pl

Instruction author:
Ziemowit Dworakowski, zdw@agh.edu.pl

Introduction

Up to this moment we have been solving "problems" created artificially (so we generated some points with no physical meaning and then tried to build learners to do regression or classification on them). While this principle will still be true today, we will take a step back in terms of how the data are processed and we will pretend that we are actually solving a real-life engineering problem. Imagine that you are tasked with interpretation of data from a rotary machinery. This data might come in a form of time-domain vibration signals. From this signals you would be expected to derive information important from the maintenance point of view. Maybe a machine can possibly be damaged and your task is to detect if that is so? Maybe you are expected to recognize operational states of this machine or infer relationships between particular values that we can register? Either way the information processing routine will be similar: you will gather data, process it to extract informative features, build a system for pattern recognition and try to improve the process iteratively by changing its components. And this is precisely what this laboratory will simulate. While the signals you are working with are again generated artificially (so they would be predictable and interesting from didactic point of view while being concise in volume), they exhibit similar behavior to ones you could encounter in practice – so if you would like to address particular problem using your context knowledge, you might want to know that tooth damage will probably manifest mostly in high frequency signal components, misalignment in frequency of the misaligned shaft, gear tooth damage will cause spikes occurring when the damaged tooth meshes with others and changes of load will result in general changes of signal amplitude in particular frequencies.

Signal database processing – to create data for your own tasks

We will start with "acquisition of signals". In reality it would mean attaching sensors to our machinery, configuring some data acquisition units and then downloading acquired data from them, in our case we will simply download the *AcquiredSignals* structure from our Student Toolbox. We will need labels for our data though – and these labels will actually form individual tasks you are meant to solve – so depending on your task, you will have the same dataset labeled in a different way. You will find these vectors in your *AcquiredSignals* structure. The list of tasks is presented in Table 5.1.

Table 5.1 – Description of the individual tasks in scope of the instruction

Task code	Structures	What the task simulates?
0	*AcquiredSignalsA, CodeVectorA1*	Two different load patterns
1	*AcquiredSignalsA, CodeVectorA2*	Bearing damage
2	*AcquiredSignalsA, CodeVectorA3*	Misalignment
3	*AcquiredSignalsA, CodeVectorA4*	Gear tooth damage
4	*AcquiredSignalsB, CodeVectorB1*	Two different load patterns
5	*AcquiredSignalsB, CodeVectorB2*	Bearing damage
6	*AcquiredSignalsB, CodeVectorB3*	Misalignment
7	*AcquiredSignalsB, CodeVectorB4*	Gear tooth damage

The first step in any analysis of that type requires division of data into training, validation and testing subsets. We should do it before we even look at our data – to avoid being biased in drawing any conclusions based on training dataset. So lets use our code vectors to label the data and then lets divide our data into subsets. Note the orange parts of the code – you will need to adjust *CodeVector* name and *AcquiredSignal* file to fit your individual task definition (from Table 5.1):

```
load AcquiredSignalsA

% We'll create a randomly permutated vector of indices to keep track of
% where our data belongs.
RP = randperm(1000);
TrainIndices = RP(1:500);
ValIndices = RP(501:750);
TestIndices = RP(751:1000);

% Now we'll divide data into subsets:
TrainSignals = AcquiredSignalsA(TrainIndices);
TrainTargets = CodeVectorA1(TrainIndices);
ValSignals = AcquiredSignalsA(ValIndices);
ValTargets = CodeVectorA1(ValIndices);
TestSignals = AcquiredSignalsA(TestIndices);
TestTargets = CodeVectorA1(TestIndices);

% And now, similarily as before, we'll save our data subsets:
save TrainData TrainSignals TrainTargets
save ValData ValSignals ValTargets
save TestData TestSignals TestTargets
```

Note that once the data are saved, this script should not be used any more. Every time you run it, you'll get a different division of data into subsets. Finally we will use the code below to display 9 random signals (See Fig. 1):

```
figure(1);
for k = 1:9
    subplot(3,3,k);
    no = randi(500); % Random signal number
    if(TrainTargets(no)==1)
        plot(TrainSignals(no).sig,'k')
    else
        plot(TrainSignals(no).sig,'r')
    end
    title(['Signal no: ',num2str(no)])
end
```

Task 5.1: Get data structure, label it using your **individual** task codes (See table 5.1 for that). Display examples of signals from same classes and opposite classes. Try to find any factors that would allow you to consistently recognize these signals "by hand".

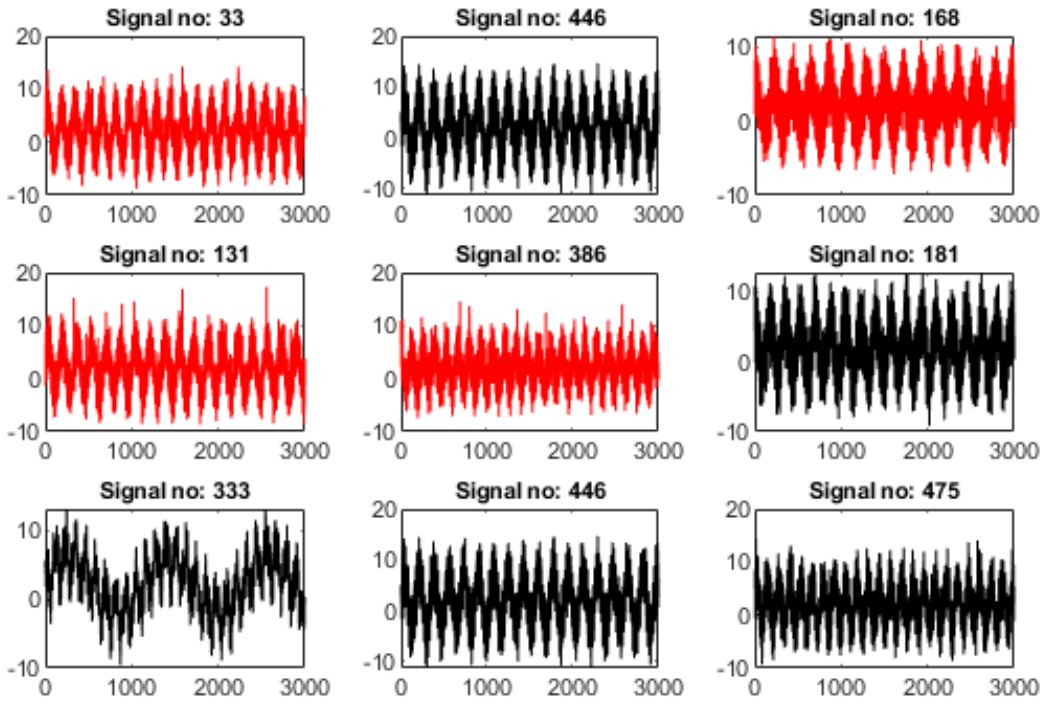

Fig 1 – Examples of 9 random signals from our database. Color refers to class label. Notice that inter-class variability seems rather large – we can't easily find any rule that allows for separation of red signals from black ones.

Note that, while each signal is associated with a target value – we can't see any feature space now – we did not extract any features yet. Since we are not sure now which features will actually help us solve the problem, lets just extract some general features and see what happens. Note that input and output formulation should be similar for all of them: all of the functions for feature extraction should take the signal and return a value which we store in our *Features* matrix:

```
for k = 1:length(TrainSignals)
    Features_train(k,1) = mean(TrainSignals(k).sig);
    Features_train(k,2) = std(TrainSignals(k).sig);
    Features_train(k,3) = MyFeature(TrainSignals(k).sig);
    % ...
    % And here we'll want to have many different features of your choice
end
```

You can craft your own functions and you can use built-in ones, like *kurtosis, skewness, peak2peak, rms* and others. I've decided to use one feature of my own implementation (which I called *MyFeature*) – not because it is necessary (it is actually exactly similar to matlab built-in function *peak2rms*), but because I wanted to show you how it is done:

```
function[feature] = MyFeature(signal)
   m = max(abs(signal)); % We take the maximum absolute of a signal
   feature = m/rms(signal); % And then divide it by rms of the signal
end
```

Now we can extract features – and finally show any two-dimensional sub-spaces of the feature space. Let us associate target values with colors:

```
figure(2);
for no = 1:length(TrainSignals)
    if(TrainTargets(no) == 1)
        plot(Features_train(no,1),Features_train(no,2),'.k'); hold on
    else
        plot(Features_train(no,1),Features_train(no,2),'.m'); hold on
    end
    xlabel('Signal mean')
    ylabel('Signal std')
end
```

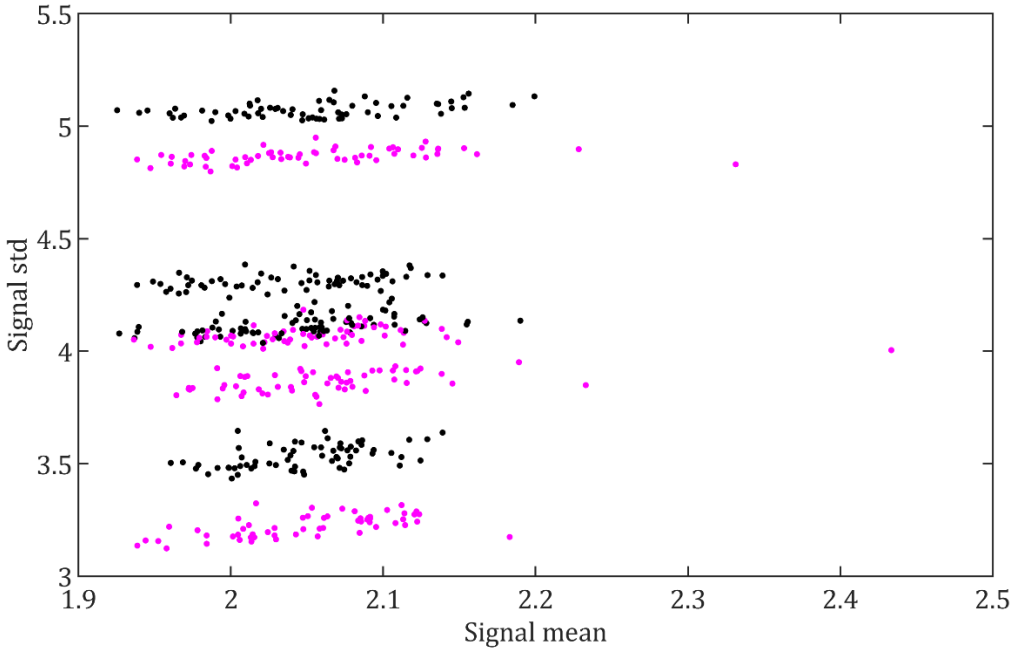

Fig 2 – Scatterplot of classification space built from two features calculated for all our signals – with classes coded by CodeVector denoted by colors.

In Figure 2 we can see now a two-feature classification space. Note that these data appear in many different clusters – which can be interpreted as many different operational or environmental states. While this space will produce results which are better than coin toss (there are clusters that contain only "magenta" or only "black" samples, the classification space is still far from perfect: some clusters of both classes overlap – and the *Signal mean* feature appears to not contribute much to our problem.

Task 5.2: Calculate at least three different features (in addition to those already calculated in our code and then show all the 6 features for your entire training subset of data using several two-dimensional scatterplots. Try to look for such a pair of features, for which the data will appear to be the easiest to separate. Save the scatterplot it produces – for showing it to the teacher.

Initial decision system

Since we now have features that can be used as inputs to our system, lets just pass the data through a neural network and check if we can force it to correctly recognize patterns in training dataset. We don't know yet if our features are informative enough, we don't know how many we should use and what will be the actual overlap (the lowest possible error in the best case scenario), but lets do that so we get a point of reference for further improvement. You will need to prepare data in such a way that it would fit to the Neural Network that we've designed in scope of Laboratory 4.

Task 5.3: Train an ANN model using training subset and two features of your choice, then use the validation subset to estimate how general the solution is. Store the network configuration and its validation result in the Table 5.2 at the end of the instruction. This network becomes our baseline network for further evaluation

Note that you will need to calculate features for Validation subset of your data in exactly the same way and store them in the Features_val matrix in the same columns you did for training dataset (Features_train). Otherwise your ANN will not be able to recognize new data!

Task 5.4: Estimate importance of particular hyperparameters of our setup. Don't try to optimize them yet, but check what is the validation result if you modify your decision system to:

a) use significantly wider ANN model (more neurons in a layer);
b) use a model of different depth (more hidden layers)
c) use different training algorithm for your ANN;
d) use different pair of features;
e) use all of the calculated features;

And then compare it with configuration from task 5.3 (baseline network), but evaluated statistically (so we want to check whether any of these (a) – (e) tries actually provides significant change when compared to natural variability of neural network training procedure. Store the results in Table 5.2 and be prepared to discuss your findings with the teacher.

Note that points (a) – (e) are just results of one try, without statistics. We are saving time this way and are simplifying computations, because we are still in the initial phase of model setup.

Hyperparameter optimization

There are four basic approaches to model hyperparameter setup:

Approach 1: We make sure only that the model is complex enough to grasp data and go with „defaults" – or copy hyperparameter values from a similar setup. The approach saves time and data – apart from the stopping criterion, e.g. neural network training, there is no need to devote large subsets of data to optimize hyperparameters – and there is also no need to reserve time for "configuration runs" in which the model is trained only so its performance could be evaluated for the purpose of its hyperparameter setup, with no relation to the future goal of the model. We won't be using this approach in our test case.

Approach 2: We pick a reasonable starting point – often based either on default hyperparameter values or similar setup found in the literature, and then slightly change one hyperparameter at a time checking if it helps in a statistically significant way – until we run out of time. Effectively, it can be perceived as usage of the 1+1 optimization approach. Here we are allocating time as it is needed – until we stop improving our solution. If we would like to use this approach for our model, we could follow this routine:

A) Pick a good starting point – e.g. based on results of Task 5.4
B) Evaluate this point statistically (storing e.g. mean and standard deviation from 10-30 runs, depending on how consistent these results are.
C) We pick a hyperparameter and then modify it (noticeably but not to the extreme – so e.g. by 20 – 30%)
D) We repeat step B and based on comparison of both statistical runs we decide whether we continue the direction, go backwards or change a hyperparameter for modification.
E) We repeat steps C – D until the improvement stagnates over multiple tries.

Approach 3: We decide which hyperparameters will influence the result the most – and do full optimization routine only for those (using grid search). Note that because we expect the objective function to be a smooth one, we don't need as many repeats for every point – because we can average the results over neighboring points in our optimization grid. While we risk doing much more computations than in other approaches, we are estimating objective function shape with higher accuracy. If we would like to go with this approach, we should use the following routine:

A) Pick a set of two to three hyperparameters which feel the most significant (based on the results of Task 5.4).
B) Pick a grid of points in space defined by these hyperparameters. In each point run the evaluation several times. Assign density of grid and number of starts per grid node so that the whole procedure would not be too long (assume some constraints you want to meet). Note that if you go with a dense grid and not many starts per grid – you should probably smooth out the results – e.g. by averaging neighboring points.

Approach 4: We do a one-dimensional optimization for all hyperparameters separately. So, starting from a selected hyperparameter values, we run optimization each time changing only one parameter – and moving to the found minimum before picking another hyperparameter to optimize. While the routine is pretty simple and does not require too much computation power, it can easily end in a local minimum – as Hyperparameters often influence each other to a large extent. If you'd like to go with this approach, you should probably use the following routine:

A) Start with a starting point – e.g. selected based on results of task 5.4

B) Pick one of the hyperparameters that feels the most influential for the results, optimize only this hyperparameter, "freezing" the others. We can use random search here or go with grid approach (this time grid will be one-dimensional).
C) Now lets repeat step B for a different, yet unoptimized hyperparameter. We do it until we run out of hyperparameters.

Task 5.5: Select any of the approaches for hyperparameter optimization (Excluding of course the "no optimization, go with default" approach) and follow it until you can't improve your decision system any further. Check if the obtained results are statistically significant – in comparison to the results of task 5.3. Finally, evaluate the optimized solution based on *Testing* subset of your data – to estimate the future efficiency of your system. Then store the results in Table 5.2

Table 5.2 – Aggregated results of the Laboratory 5

Task configuration:	Signal database: Code vector: Task type:	
Baseline network result:	**Training:**	
	Validation:	
Baseline network configuration: (structure and training algorithm)		
Features used:		
Statistical baseline network result:	Mean of 20 tries:	
	Std of 20 tries:	
Validation performance for different model configuration:	**Neurons in layer change:**	**Layers number change:**
	Training algorithm change:	**Different feature pair:**
	All the features:	
Result of hyperparameter optimization:	**Final configuration:**	
	Validation error (mean and best):	
	Testing result:	

Additional tasks

Task 5.6: Calculate frequency spectrum of the signal, filter it to extract several frequency bands and then calculate RMS of these bands (separately, forming a set of RMS-related features). Then use these features to check if they allow for improvement of your decision system optimized in task 5.5

Task 5.7: Pick a different approach to optimization of hyperparameters than you did in task 5.5 (so, e.g., if you've used grid search, use 1+1 approach now). Perform full hyperparameter optimization routine and then compare these two results with each other statistically.

Task 5.8: Use other code vector (the one that you did not use in Task 5.1) as additional feature and then check if it allows for statistically significant improvement of your results. Try that several times using different code vectors. Try to explain the results *(did it help? Why? Or why not?)*

Task 5.9: Note that our data are significantly clustered. Maybe it would be easy to perform classification based primarily on cluster labels? Select a 2-dimensional feature space in which you can clearly see many different clusters of data (similarly as in Figure 2) and then:

a) read the documentation for the k-means clustering method in matlab (*doc kmeans*)
b) run the *kmeans* on the training AND validation data (a set consisting of both these subsets) to get cluster indices.
c) Assign labels for clusters based on training target values prevalent for them (so count how many training points assigned to "cluster 1" have "label 1" – and if it is more than 50%, assign to cluster label "1", and so on.
d) Classify validation data using information about classes from point (c)
e) Compare this result with ones obtained in task 5.5 and 5.3