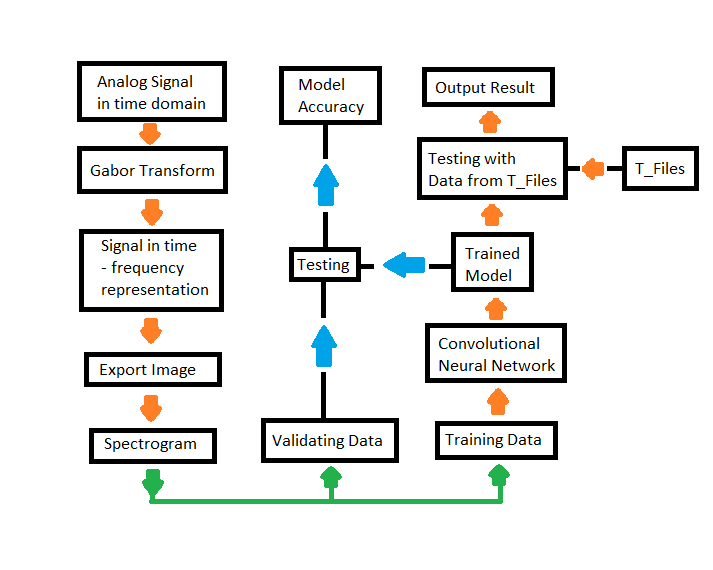
1. Overview:

The problem of the project is how to classify time-series reflected signal of different objects. To solve this problem, a series of step was laid out in form of a project pipeline, which is presented above in this text.



In Summary, the code for the project was divided into 3 stages. Stage one involves processing of the analog time signal sample into spectrogram via Gabor Transform, see Literature Review for details of Gabor Transform. Stage two trains the Convolutional Neural Network (CNN) via adjusting the weights of the initial network to fit the training set. Stage two ends with validating the learning by feeding the validation set to the trained network. In stage three, the trained network was used to predict unlabeled data from the test file set T\_Files.

1. About the Dataset:

The data was issued from Professor Pech in .xlsx files specifically for the Subject Computational Intelligence at Frankfurt University of Applied Sciences. There are two types of datasets in the provided files from the subject:

1. Training Files:

The training set includes 3 analog time readings samples of 3 different Objects: ‘Data Object1’, ‘Data Object 2’, ‘Data Object 3’. They are located under Matlab/dataset/. Each Row, from the 7th column in each files represents a time-series sample of the object. So, when the samples are read from the training data, the rows are read from the 7th column to the end of each row.

Table

Description automatically generated

The samples in each training files:

|  |  |
| --- | --- |
| Data Object 1 | 315 Samples |
| Data Object 2 | 200 Samples |
| Data Object 3 | 400 Samples |

During the training, the number of samples used for training and validation were chosen via a split of data from the original training files. The current approach used the same number of training samples for each type of Object. Therefore, the number of validation samples left from ‘Data Object 2’ will be less than that of the others. This is the reason that the training data samples is currently set to 200, the maximum number of samples from ‘Data Object 2’.

Due to a current lack of training data, the scope of this Projects focused more on developing a pipeline for dealing with reflected time series signal of objects rather than reaching a good result of training.

1. Test Files:

Test Files are structured the same way as the training Files, but they have no label, there are twelve test files in total, each has 50 data samples.

For the scope of this project, these test files were used as result of the model. They were fed into the trained model after the training session ends for the prediction of the associated label that belongs to the test file. Tests File are located under Matlab/dataset/ and share the same form as “T File <No>” with 12 >= No >= 1.

The documented output of the test file ‘s label is written under Matlab/Result/<netName> with netName is the name of the used trained network.

1. Gabor Transform - Code:

To do have a mathematically description of Gabor Transform, please refer to the literature review above. The creation of Spectrograms involves plotting and export the potted graph into .jpg, so during the creation of the pictures, a plot window will pop up and close for each data sample. The result of this process is the folders *‘trainingData’* and *‘testingData’, which holds folders of spectrograms* – figure **<>***.*

Graphical user interface, application

Description automatically generated with medium confidence

The code explanation below is applied for doing Gabor Transformation in 2 stages of the Experiment. One at the creation of the training data, reading the training set and one at the preprocessing of the test files. Consequently, in this section ‘*testingData’* and ‘*trainingData’* are used interchangeable. It serves the purpose of transforming time-series signal samples into time frequency representations (TFRs).



In Figure <>, the data samples, in time series form of the data was read as a table. In the second row, the data\_range extract the table for only the data from column 7th to end for each data samples.



Because there are only one files for one label, dataLabel for the learning was chosen as the filename. In this work, the file names are ‘Data Object1’, ‘Data Object 2’ and ‘Data Object 3’. Consequently, the respective dataLabel are ‘Data Object1’, ‘Data Object 2’ and ‘Data Object 3’. So at this stage we have classification of multiple objects.

A screenshot of a computer

Description automatically generated with medium confidence

We then proceed to read the length of data\_range (extracted data from the original files, column 7th –> end) which is 3400 timestamps for every data samples. A variable ‘t’ was used here for the indexing of the timestamps for upcoming iteration. Then a folder was made in trainingData/<dataLabel> to save the output spectrogram. The same process happens when dealing with testing data, whereas a folder of name testingData/<dataLabel> is made.

Text, letter

Description automatically generated

Figure **<>** demonstrates the iteration through all samples of one training files, which are now stored in variable **data\_range**. Because the spectrogram was generated via exporting image from a graph, multiple windows of plot will present while executing the loop. To cope with this, ‘*close all’* simplifies the process by closing the previous graph after each cycle. **S** is the normalized signal, in table forms, this variable has the value of a normalized sample from **data\_range** at index **c**. Here normalization means take the values of each timestamp in a sample and divide it by the max absolute value of all timestamps in that sample. **L** serves here as the period in seconds of a sample for calculation of the frequency spectrum, but as only spatial relation/ shape of each TFR/ Spectrogram is of concern of the Convolutional Neural Networks, it can be of any value, here it is set at 10s. The designed CNNs for this work focuses more on the graphical rather than the numerical features of the samples. **K** use **t** to generate the frequency scale of the graph. **ks** is a shifted version of **k**, which will be used to plot the frequency dimension of the spectrogram. **Sgt\_spect** is the strength of the frequencies or the spectrum density in each window, this will be presented as color in the spectrogram.

The Next session discuss about the iteration that simulated Gabor transform in this project. The instruction video of **<insert 45’ guy youtube name>** demonstrated the process beautifully. The code for Gabor Transform in this project is based on his Matlab demonstration.



tslide was used to take the indexes which belongs to the start of each window used for Gabor transform. To recall, Gabor transform makes use of a sliding window in the time domain, then Fourier transform the signal in each window respectively. In this experiment, the window moves 20 samples at a time.

In figure **<No>**The iteration of the Transform based on the index j, running from 1 to the end of tslide’s length.

Chart

Description automatically generated

The code in Figure **<>** shows the windowing function. **tslide** represents the shift in the time scale, he figure **<>** shows a better representation of the function **g** when plotted with different **tslide** values.



Then **tslide** is multiplied with **S** to get the corresponding time signal after filtered out by the filter function **g**.



The next step is calculating the Fourier Transform of **Sg** and add it to **Sgt\_spec**. After ending the Gabor Transform of a sample, three variables are now of interest to plot the final Spectrogram:

|  |  |
| --- | --- |
| **tslide** | Starting timestamp of each window, used for indexing in time axis |
| **ks** | Frequency scale of the signal, used for allocating the spectrum/ frequency axis |
| **Sgt\_spec** | Matrix of frequency strength for each window, used for color representation of pixels |

Text, letter

Description automatically generated

**pcolor** or Pseudocolor plot, is the function used for plotting a color graph with X axis (**tslide** - time), Y axis (**ks** – frequency spectrum) and color matrix of each pixel **C** (**Sgt\_spec** – strength of signal at each frequency). More about **pcolor** can be found on Mathworks official website **[reference]**.

The option that to change the spectrogram output extends this project with a greater source of inputs for the CNN. However, the current settings shown in figure **<>** optimizes for the fastest training time at the moments. To illustrate this process of choosing the settings for the spectrogram, the first sample of ‘Data Object 1’ is ultilised.

A picture containing text

Description automatically generated

Figure **<>** shows the raw graph which is the output of only **pcolor**. It was considered at the start that this represented errors during the implementation. The Spectrogram only reveals itself when zoomed in**,** the zoomed version of the spectrogram is shown in the following figure **<>**.

A screenshot of a computer

Description automatically generated with medium confidence

Due to the symmetric nature of the Fourier transform/ Gabor transform, it is assumed in this work that half of the spectrogram has enough information of the object’s reflected signal readings. This explain the choice of the frequency range **ks** or **Y** here scaled to [30 60], the output can be seen in figure **<>**.

Chart, histogram

Description automatically generated

The Spectrogram at this point posed to have good quality of information extracted from the sample. Thus, the plot possesses unwanted table borders that are the same for all plotted spectrograms. To remove these lines, the option Interpolated shading of the plot was switched, figure **<>** shows the output spectrogram after using interpolated shading settings.

A picture containing background pattern

Description automatically generated

The spectrogram was thought to be enough for the training at this point. However, after exporting, the image appears to come with the axes, which is not of concern for the learning also due to its similarity in all exported spectrograms. To remove these axes, the ‘*Visible’* option of *gca* was set to *‘off’*.

Training at this point is considered sufficient after all the data cleaning done. This work however took a step further in setting the spectrogram to grayscale by changing the colormap option to ‘gray’. The colormap setting in figure <> is the ‘default’ option, the figures <>, <>, <>, <> shows some available options that might be of interests for further research about which map draws the best performance.

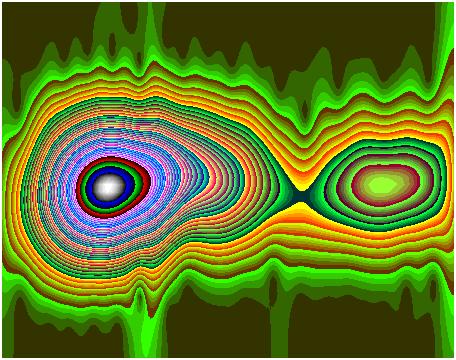


Figure 1 colorMap 'colorcube'

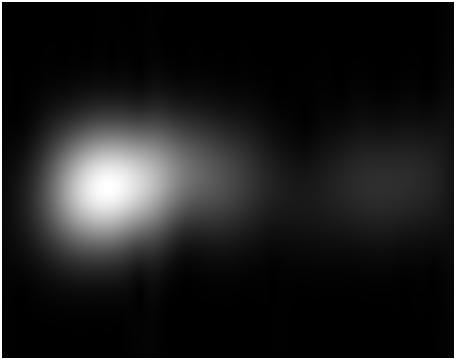


Figure 2 colorMap 'gray'

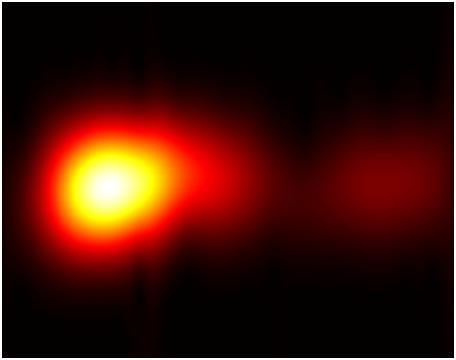


Figure 3 colorMap 'hot'

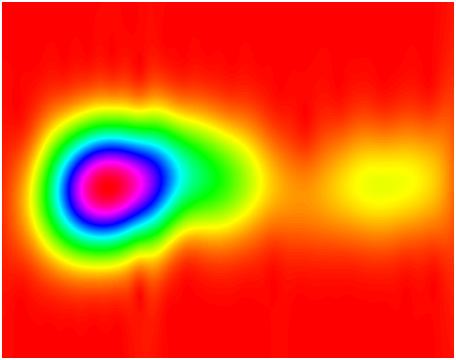


Figure 4 colorMap 'hsv'

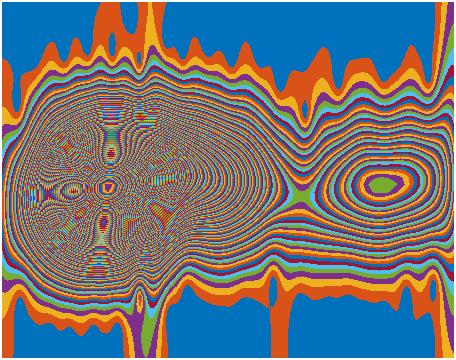


Figure 5 colorMap 'lines'

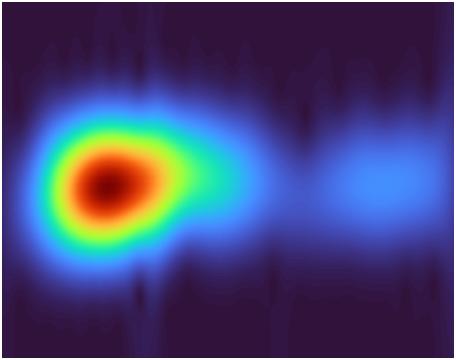


Figure 6 colorMap 'turbo'

After tweaking with the settings of the plot, a spectrogram is outputted. The last parameters to put in are the file name and the dimension.



The dimension unit is dots per inch, the higher the number, the more details the exported spectrogram gets. Tables **<>** demonstrates the output spectrogram with different settings of dpi.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **dpi** | 10 | 50 | 100 | 200 |
| **size** | 42x25 | 230x183 | 456x361 | 908x719 |

*NOTICE:* The output of the experiment are trained networks which is specific to a type of exported spectrogram. A network trained for one type of spectrogram can only be used to predict the spectrograms that shares same properties with the training set. Unless the Input spectrogram is converted through further preprocessing, Matlab will raise an error when the input spectrogram is not compatible. It is recommended to name the trained network more specifically with the settings of the exported spectrograms.

1. Training the CNN:

Convolutional neural Networks in Matlab are enabled by installation of the Statistics and Machine Learning Toolbox, Parallel Computing ToolBox and Deep Learning Toolbox. The mentioned toolboxes are the requirement for a Matlab environment to run the Graphical User Interface (GUI) introduced in this project.

The modeling of the CNN consisted of 2 components: layers and options. The variable **app.layers** holds information about the neuronal structure of the networks whereas **app.options** describes the behaviors of the training process.

Text

Description automatically generated

The initiation of both network components could be found in the **networkInit** function. The current generation of training Data described in section **C** above, the number of folders represents the number of output labels. The first code chunk in figure **<>** shows the process of getting the number of labels via the number of directories within the current directory.

Text

Description automatically generated

Figure **<>** represents a simple CNN model, which is used for the task of classifying different objects. The following section discuss about some definition of the layers and their functionalities in building the CNN model. These options are from the Matlab ‘s Deep Learning Toolbox. The documentation can be accessed on Mathworks website for more clarity **[ref]**.

**ImageInputLayer**: This is the outer input layer of the CNN. So, the number of inputs here depends on the chosen number of pixels from the dataset’s image. The inner part of the code took the size of the first spectrogram from the original image dataset **imds**. The current used size of the spectrograms is 456x361 at 100dpi.

**Convolution2dLayer**: This layer creates the backbone and the core logic of CNN. The first parameters in this layer config describe the convolutional filter size, the convolutional filter in this case is a square due to the one scalar passing. The filter specifies the size of local pixels which are included in one calculation. The next parameter describes the number of neurons that is in the layer, this will be the number of output features that will be the inputs to the next layer. The ‘*Padding’* option *‘same’* ensure that the output from this layer and the input have the same dimension.

**bachtNormalizationLayer**: This layer normalizes a mini-batch of data independently across all observations in each channel. Training CNN faster via decreasing network initialization’s sensitivity was achieved by using batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers. After normalization, the layer scales the input with a learnable scale factor γ and shifts it by a learnable offset β.

**reluLayer**: or Rectified Linear Unit (ReLU) is a common activation function that are used in machine learning. In short, it converts all values that are less than a threshold (usually 0) to become 0.

**maxPooling2dLayer**: A 2-D max pooling layer performs down-sampling in its input. The layer first divides the input into rectangular pooling regions. Then it computes the maximum value in each divided region.

**fullyConnectedLayer**: A fully connected layer multiplies the input by a weight matrix and then adds a bias vector. This layer takes the whole input form the last layer and have at output the same number of input labels. Its parameter is also the number of Labels/outputs.

**softmaxLayer:** A softmax layer applies a softmax function to the input. The softmax function is known as a normalized exponential function. This play the normalizing roles for the classification.

**classificationLayer**: *From Mathworks – “*A classification layer computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes. The layer infers the number of classes from the output size of the previous layer. For example, to specify the number of classes K of the network, you can include a fully connected layer with output size K and a softmax layer before the classification layer”. Basically, it calculates what the output to belongs to in the pool of learned Label.

After the initialization of the network structure, the learning behaviors of the network **app.options** is specified, for an overview see figure **<>** .

Text

Description automatically generated

There are a lot of option available in the configuration of **app.options** before training the network. ‘*sgdm*’ is an option that enables Stochastic Gradient Descent with momentum, which is a method of assigning cost function to the weights of the network. ‘*InitialLearnRate’* means the learning rate of the weights, it is a scalar which is multiplied into the change of weights to ensure that the learning can slowly reach its stable point. *‘MaxEpochs’* was set to 5 based on experimental proof. With a lot of observation on the learning of the dataset, most of the run reach maximum accuracy at epoch 5 or 6. So, the number was set to 5 to reduce the training time of the model. ‘*Shuffle’* option shuffles the order of the inputs, so that the training can reach a more general solution rather than a local minimum of errors. *‘ValidationData’* specifies the dataset used for Validation Process. *‘ValidationFrequency’* defines how much the network is validated in each epoch, the default was 50, so a smaller value resulted in a faster training session. To display the training progress, *‘Verbose’* was set to false and *‘Plot’* was set to *‘training-progress’*. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights - MathWorks. The parameter used by the *‘MiniBatchSize’* specifies its size.

*‘ExecutionEnvironment’* was set here to use *‘gpu’.* However, it is recommended to be set as *‘auto’* as some computers are not available with a GPU. If this happened, the training will use CPU as the main power source for training the networks.

After tweaking up the structure and the learning parameter of the network, the training start with **trainNetwork** – see figure **<>** :

Text

Description automatically generated

The code after the learning saves the trained network to a mat file. This mat file will later be used to predict the testing data, which are provided from Prof Pech.

1. Validation of trained network:

Validation of trained networks happens right after the training of the CNN. The current validating process use the Validation dataset split from the first step from the Training Dataset. The data are fed into the trained networks to drawn out the labels of the data sample.

The input data at validation stage is images (spectrograms), which are already created together with the training data. They were split from the original training set. Validation stage in this project includes the computation of the following parameters:

|  |  |
| --- | --- |
| **PPV** | Positive Predictive Value |
| **FDR** | False Discovery Rate |
| **NPV** | Negative Predictive Value |
| **FOR** | False Omission Rate |
| **TPR** | True Positive Rate |
| **F1** | F-score |
| **Accuracy** | Accuracy |
| **FPR** | True Positive Rate |
| **TNR** | True Negative Rate |
| **ROC** | Receiver Operating Characteristic |

The following attributes, **accuracy**, **Receiver Operating Characteristic** (**ROC)** and the **Confusion Matrix** are calculated based on all three Labels. The others are calculated on the basis as ‘Data Object 1’ or ‘not Data Object 1’, this is not only for the simplification of the problem but also to fit with the requirements of the given project on classifying reflected time signal of Objects.

The Confusion matrix, the output of the training, and the training progress of 5 pre-trained networks can be found in the Experiment section.

1. Testing of Data from T Files:

Testing of trained networks can take place whenever there is a trained network in the same directory of the GUI, there are several trained networks submitted together with the project, each has their own validation result saved under /images/<netName>. This infrastructure separates the training process and the Testing process, which save time when conducting and documenting multiple experiments. For details on the Testing procedure, see the description of steps on the GUI section.

The inputs at testing stage are datasets which looks like the original training dataset, except they have no label. At this point, the data must be converted again into TFRs and then exported into spectrograms. The process of converting the dataset into TFRs are the same as that when dealing with the training dataset.

The output spectrograms of the testing dataset are saved under *‘testingData/’*, the folders in *‘testingData/’* are named after the testing dataset’s name. To prepare for the prediction, a trained network must also be loaded. In the normal workflows which training network takes place, the name of the trained network will be presented in the field. Predict in this case outputs first show an interactive graph where the navigation of sample indexes can be performed. Then the result from the prediction is saved in a file under the directory *‘Result/<netName>/’.* The Result folder was then analyzed statistically to see the differences between the trained net, it also helps to determined which T Files is showing which Data Object.

In the end of the testing stage, the T Files are predicted into the label pool by describing how much % of the testing file is of which Label (‘Data Object 1’,’ Data Object 2’,’ Data Object 3’).