

# Machine Learning Engineer Nanodegree

## Capstone Project

Yves Fauser, September 13th, 2019

### I. Definition

#### Project Overview

Overhead power lines are susceptible to various external influences like tree branches falling onto the line, mechanical damage through material fatigue, etc.

Failing power lines can lead to power outages, cutting of whole rural villages and cities from electricity supply, or even worse, line faults can cause fires through heat caused by power resistance and discharges.

The problem, and how it can be solved with the help of machine learning, is detailed in an academic paper written by the technical University of Ostrava (Czech Republic) [\[1\]](#).

At the core of the possible solution using machine learning is the detection of particular pulses found when measuring energy radiation using coils clamped onto the overhead power lines. This energy radiation is measured in short samples, and made available for central analysis using the cellular network. Depending on the patterns seen in the data, conclusions can be drawn on whether the overhead line needs to be inspected by a field technician team.

#### Problem Statement

Partial damage of overhead power lines by external factors is hard to diagnose. Even with large scale regular inspection of power lines e.g. through manual visual inspection, or fly overs with drones, a lot of material fatigue and partial damage problems can stay undiscovered. Therefore a solution is needed that relies solely on analysis of sensor data (coils clamped onto the overhead power lines), discovering specific patterns in samples measured at the overhead power line. A key early indicator of a condition of the overhead power line that needs attention is partial discharge (PD). PD is generating specific patterns that can be seen in the sensor data, and gives a good indication e.g. of tree branch contact.

[\[2\]](#) Whenever partial discharge is initiated, high frequency transient current pulses will appear and persist for nanoseconds to a microsecond, then disappear and reappear repeatedly as the voltage sinewave goes through the zero crossing

The problem of discovering damages on overhead power lines can therefore be solved by analyzing the sensor data, finding PD patterns and distinguish them from other patterns e.g. caused by corona discharge and other commonly found external influences to the power line.

## Metrics

The goal of this project is to detect PD patterns and send out field teams to find and repair defects in overhead lines before an complete outage occurs. The costs associated with a False Negative is very high, as a defective power line might not be detected, and might fail causing a widespread outage that may even be life threatening. There is also a cost associated with False Positives, as there is a high effort of erroneously sending out a field team to inspect an overhead line that turns out not to be faulty.

Recall is the most important metric in this project, as it calculates how many of the Actual Positives the model captures through labeling it as Positive (True Positive)

$$Recall = \frac{(TP)}{(TP + FN)}$$

Where TP is the number of true positives, TN the number of true negatives, FP the number of false positives, and FN the number of false negatives.

So, Recall is a good indicator for how many time the model missed classifying a faulty overhead power line as faulty, therefore risking and highly impact outage.

Another metric to use is Precision. Precision calculates how many of the samples predicted by the model as positive are actually really positive:

$$Precision = \frac{(TP)}{(TP + FP)}$$

So, Precision is an indicator of how many times the model misclassified the line as faulty, and a field technician team was sent out erroneously causing unnecessary costs and efforts.

In addition I will calculate 3 additional metrics: Accuracy, F1-Score and MCC.

Accuracy will tell us the ratio of correctly classified samples out of the total number of samples. Accuracy is actually not a good metric to look at with the Dataset I use in this project. I will go into details of this dataset in the later chapters, but what can be said here is that the dataset is highly imbalanced with only 6.41% of the samples in the available training dataset being positives (525 positive and 8187 negative training samples). Therefore the number of true negatives will be very high, generating a high accuracy score. I will however include it for completeness.

F1-Score is a better metric for this project, as it balances Precision and Recall, which are the two metrics giving us a good indication on the costs associated with my models false predictions:

$$F1 = 2 * \frac{(Precision * Recall)}{(Precision + Recall)}$$

F1 score is the harmonic mean of precision and recall.

And finally, I calculate the Matthews correlation coefficient (MCC) between the predicted and actual truth in the dataset. The MCC is given by:

$$MCC = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The Dataset is taken from a Kaggle competition [\[3\]](#) that uses MCC as evaluation metric. The MCC is in essence a correlation coefficient between the observed and predicted binary classifications; it returns a value between  $-1$  and  $+1$ . A coefficient of  $+1$  represents a perfect prediction,  $0$  no better than random prediction and  $-1$  indicates total disagreement between prediction and observation. While MCC serves a purpose very similar to F1-Score, it is able to show cases of reversed classifications by turning negative.

The Kaggle competitions Test Dataset does not have public available truth labels, therefore the final evaluation of the Test dataset will only give us an MCC. For the Training Dataset, where truth labels are available, I will carve out a validation set and evaluate Accuracy, Recall, Precision and F1-Score too.

## II. Analysis

### Data Exploration & Visualization

As part of a Kaggle competition, the Technical University of Ostrava (Czech Republic) provided sensory data of medium voltage power lines [3]. The data was obtained from a low cost sensor (metering) device developed at the University and mounted at 20 different location within a region in the Czech Republic. The data was gathered from a real life power distribution network, and is therefore noisy because of real world disturbances caused by the power lines acting as antennas for electromagnetic signals generated in proximity of the power lines like radio stations, nearby lightning strikes and corona discharges.

The dataset is divided into one training and one test dataset available in the Apache Parquet format. The datasets are significantly large in terms of size on disk with 3.81 GB for the training, and 8.81 GB for the test dataset. Other than is most datasets each column instead of each row is containing one sample. Each sample (each column) has 800,000 measurements taken during a 20 millisecond time windows which equates to one complete grid cycle of the 3 phases measured.

Here is an example of what the first 5 rows of the training data look like:

	0	1	2	3	4	5	6	7	8	9	...	8702	8703	8704	8705	8706	8707	8708	8709	8710	8711
0	18	1	-19	-16	-5	19	-15	15	-1	-16	...	18	-22	12	8	13	6	-21	-15	-9	20
1	18	0	-19	-17	-6	19	-17	16	0	-15	...	17	-21	12	8	14	7	-19	-15	-8	21
2	17	-1	-20	-17	-6	19	-17	15	-3	-15	...	16	-21	13	8	15	8	-18	-14	-8	22
3	18	1	-19	-16	-5	20	-16	16	0	-15	...	16	-21	12	8	15	8	-19	-14	-7	23
4	18	0	-19	-16	-5	20	-17	16	-2	-14	...	17	-22	12	8	15	8	-18	-14	-8	23

Each column of the training and test data file represents a **signal\_id** which is a key in a separate csv file provided with the data that has the following additional metadata:

- **id\_measurement:** the ID code for a trio (3-phases) of signals recorded at the same time.
- **signal\_id:** A foreign key for the signal data (sample). Each signal ID is unique across both train and test, so the first ID in train is '0' but the first ID in test is '8712'.
- **phase:** the phase ID code within the signal trio. The phases may or may not all be impacted by a fault on the line.
- **target:** 0 if the power line is undamaged, 1 if there is a fault.

Here's an example of what the first 10 rows of the training metadata looks like.

The targets (truth labels) were manually classified by a domain expert that looked at noise patterns present in the data. If the sample is showing a PD pattern, the target is set to 1, if any other or no disturbance pattern is present the target is set to 0.

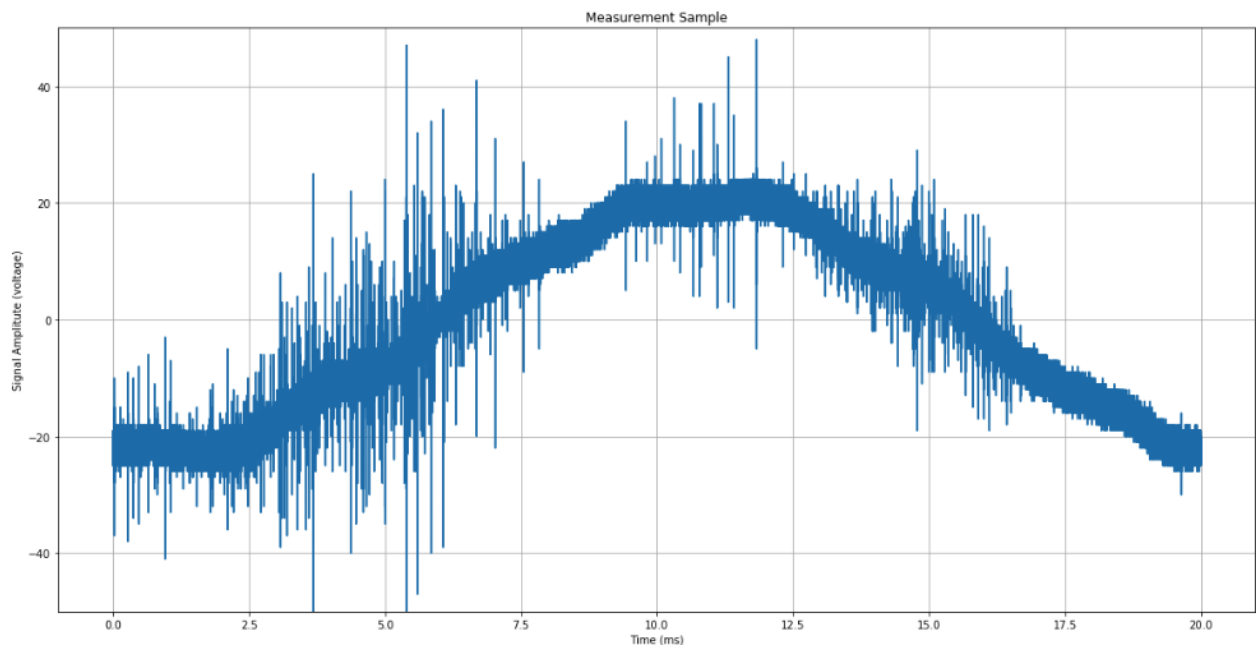
Each of the 3 phases is treated independently, so a PD pattern might be present on one of the phases but might not be present in another. Only the training metadata is labeled with targets. The test dataset is part of the Kaggle competition, and the targets are therefore hidden from public.

signal_id	id_measurement	phase	target
0	0	0	0
1	1	0	1
2	2	0	2
3	3	1	0
4	4	1	1
5	5	1	2
6	6	2	0
7	7	2	1
8	8	2	2
9	9	3	0

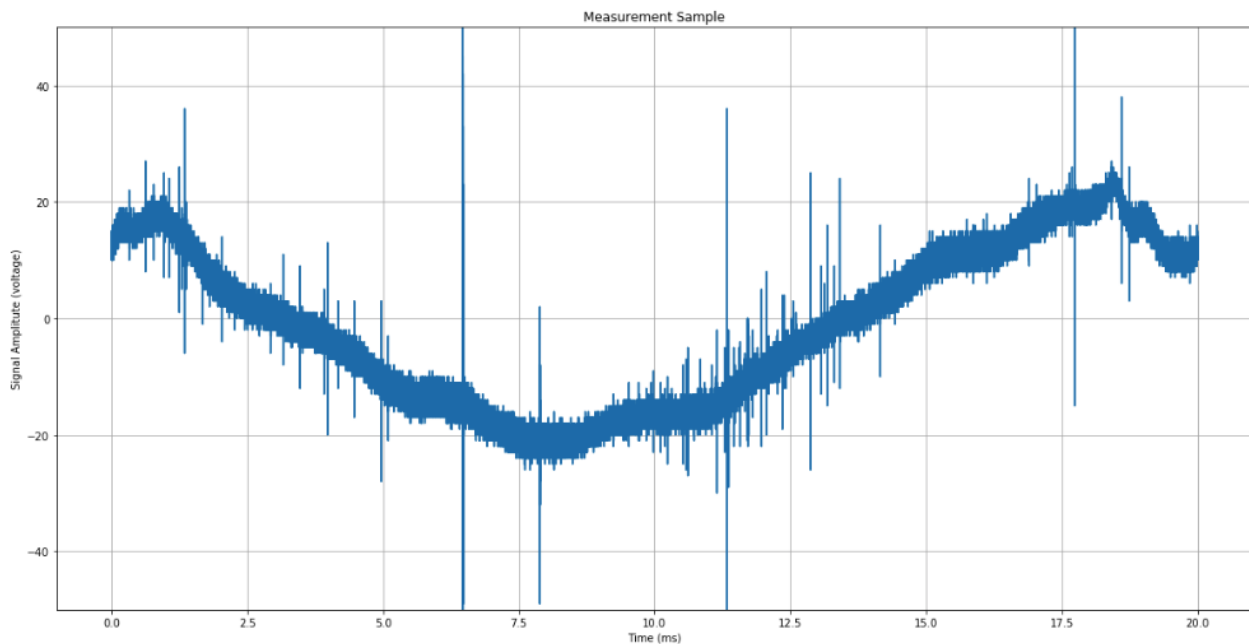
The data is imbalanced, as it contains only 525 true positives in 8711 samples.

When looking at the data in a graphical way, a simple plot of the signal amplitude (the values of the measurements) in relation to time gives us good insights.

**signal\_id 201:** Example of a sample that is labeled as having a PD pattern present:



**signal\_id 333:** Example of a sample that is not showing a PD pattern:



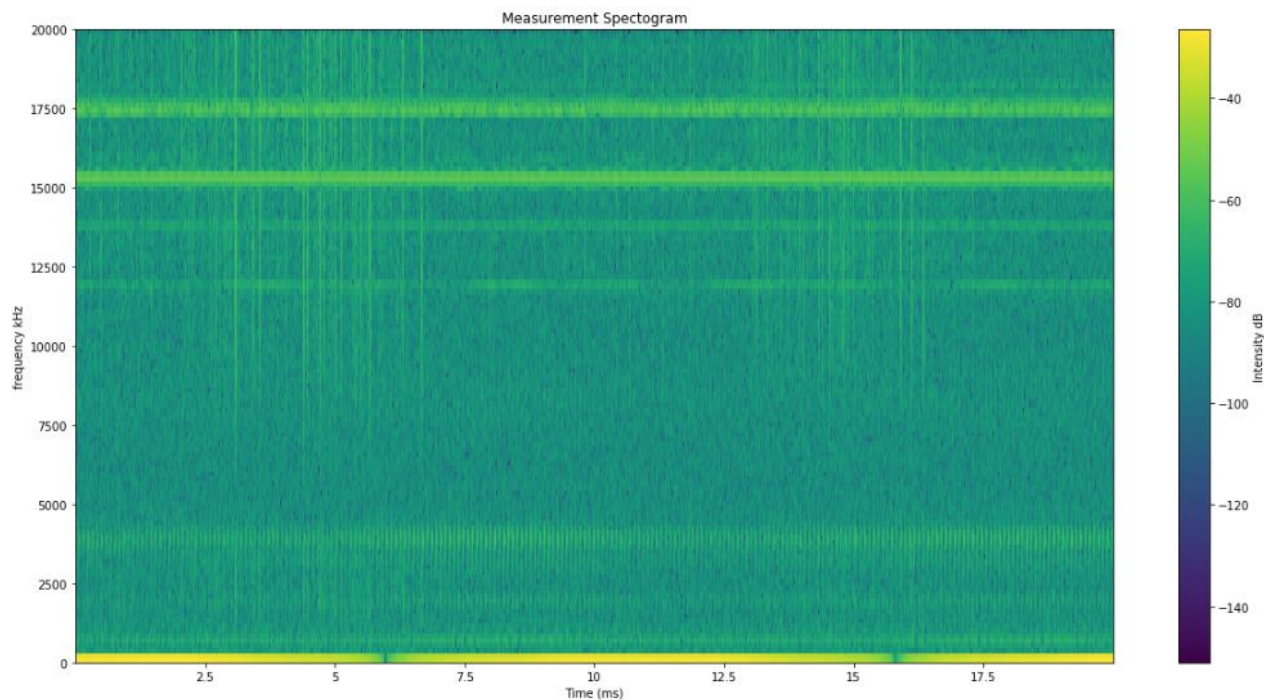
Looking at these two plots we can make a couple of observations:

- 1) The measurement devices are not synchronized with the power grids 50Hz phase. As you can see in the two examples, while the sample covers one sine wave, the start and end of each measurement is not synchronized with a specific state of the 50Hz sine wave. This is an important observation as the PD pattern is expected to happen as the voltage sinewave goes from the zero crossing to the max. voltage. Therefore to make the samples comparable I had to shift the samples so that they all start / end at the zero crossing.
- 2) In the first example the PD pattern is well distinctive and high frequency pulses are visible at the expected positions of the 50Hz sinewave.
- 3) The second example shows very well what corona discharges look like. The corona discharges are visible as high amplitude high frequency pulses.
- 4) Both examples show the always present low amplitude noise e.g. caused by radio stations and the likes

Not all samples are as easy to interpret as the two I choose to present here. In general it is very difficult for non-experts to distinguish PD from corona discharges and other noise.

Another visualization technique I extensively made use of is to look at the data using a spectrogram. Here is the spectrogram of the first example (signal\_id 201) that has a PD pattern present:

### signal\_id 201: Spectrogram



I am using matplotlib's specgram function [4] to generate the above spectrogram. This function uses Fast Fourier transform [5] with Nx sliding windows to compute the spectrogram. In a nutshell it shows the intensity (amplitude) of specific frequencies in the spectrum over time, whereas the line plot only shows the amplitude and not the frequencies contained in the sample.

## Algorithms and Techniques

As you will read in the Benchmark section of this document, the existing work of the University of Ostrava tested the use of the random forest algorithm to detect PD patterns in the samples. In my project I wanted to see if I can achieve similar or better results using various neural network implementations.

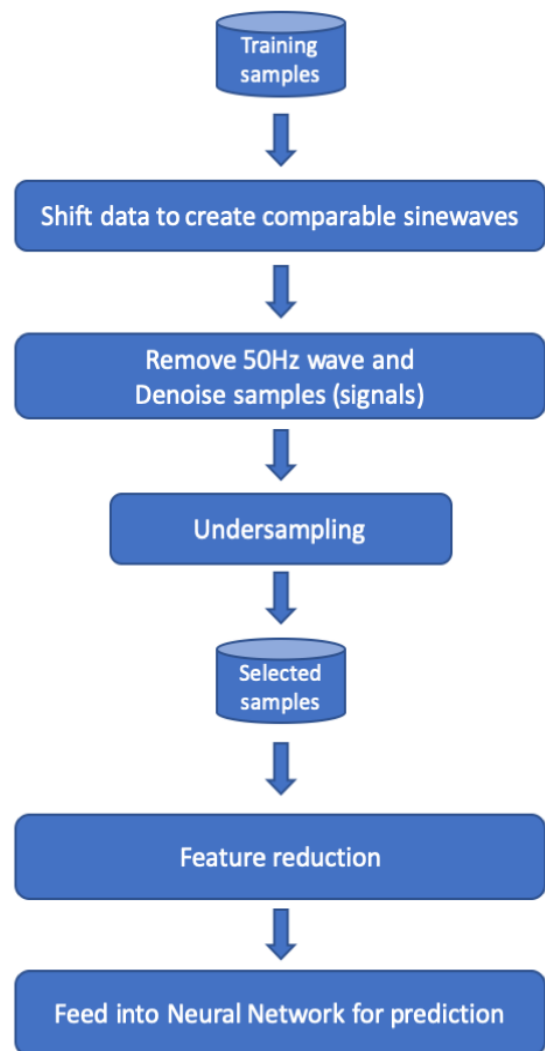
Most of the work in this project went into preprocessing the data so that it can be feed into a neural network to predict whether a PD pattern is present in the sample or not. Here's a general overview:

**Step 1)** As briefly mentioned in the last section, to make the samples comparable in the time domain, the sample data needs to be shifted so that the start and end of the sample is at the zero crossing of the 50Hz sinewave. I will go into more details of how this was done in the 'Data Preprocessing' section of this document

**Step 2)** Fault patterns usually exists in high frequency band. According to literature, the PD pattern is visible above  $10^4$  Hz, therefore I used a Butterworth filter [6] to remove all frequencies bellow  $10^4$  Hz. This removes the 50Hz sinewave and other low frequency signals. Then the signal is cleaned from low amplitude noise like radio transmissions with the use of wavelet decomposition and thresholding [7]

**Step 3)** As the training dataset is imbalanced, I used undersampling. I selected all samples labeled as positives, and a subset of random samples from 4 bins categorized by the number of amplitude peaks found in the samples.

**Step 4)** I used two different data reduction techniques and compared the results using Neural Networks. The first technique used is principle component analysis (PCA). I then directly feed the resulting 325 PCA features into Dense network or a one dimensional CNN as input. The other technique I used was to generate 224x224x3 (RGB) spectrograms using matplotlib, store them as .png images, and feed those into a two dimensional CNN as input





**Step 5)** The reduced data is feed into 4 different types of neural networks. I explored the combination of PCA with a Dense Neural Network, PCA with a 1D CNN, Spectrogram pictures with a 2D CNN and Spectrogram pictures with a 2D CNN using transfer learning from the VGG16 neural network.

## Benchmark

The key benchmark in this project are the results documented in the early paper of the technical University of Ostrava [1].

Using an implementation of the random forest algorithm and various tuning documented in the paper they where able to achieve the following results:

Metric	Result
Accuracy	99.8
Precision	83.6
Recall	66.7
F1-Score	72.8

*[See Table 4) on Page 6, Column  
'SOMA' / 'RA SP HA', 'entire dataset (blue)']*

My aim was to achieve similar or better results with a neural network implementation.

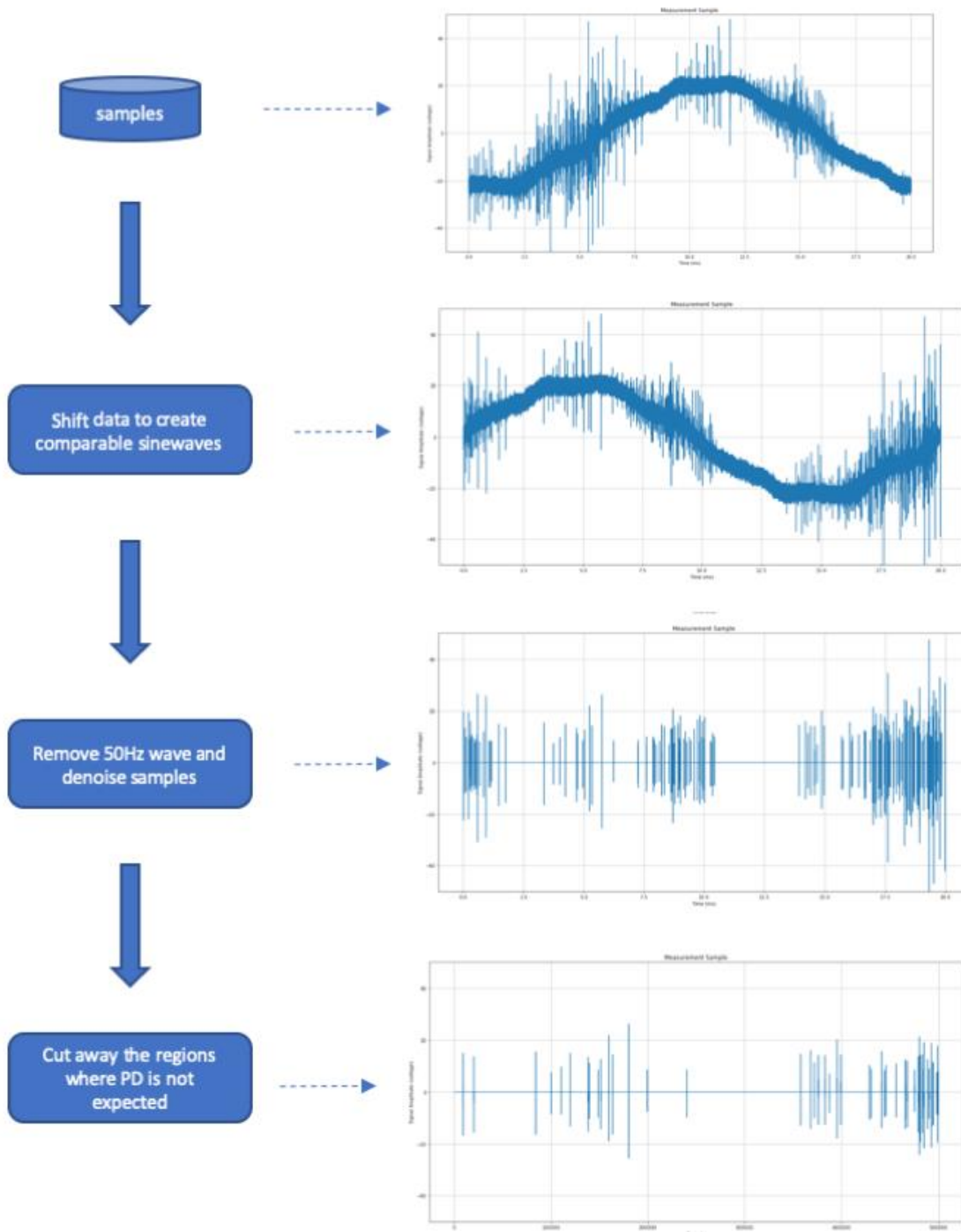
In addition to the above results of the paper of the technical University of Ostrava, my aim was to achieve results that would put me approximately in the middle of the leaderboard of the Kaggle competition [8]. At the time of writing the top score of the winning team is a very high **MCC of 0.71899**

### III. Methodology

#### Data Preprocessing

The data preprocessing was key in this project, and a lot of effort went into it. The following graphic shows the visual representation of all pre-processing steps:

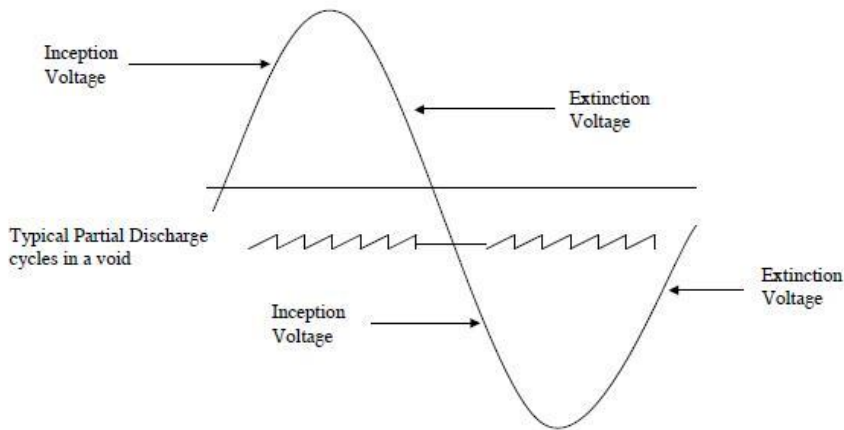
**signal\_id 201:** Data pre-processing steps



**Step 1)** As all samples need to get comparable in the time domain, I decided to apply a simple 'trick'. I am searching for the datapoint in the sample that is the closest to the first zero crossing of the 50Hz 'base' sinewave. I then shift all the datapoints before that datapoint to the right, essentially generating a clean sinewave that starts with zero and ends with zero and has a zero crossing in the middle. As PD patterns are present on specific locations during the sinewave this shift might not represent to time domain 'reality', but as the patterns always repeat in the time domain at the same position it doesn't matter if the true measurement was shifted in this way

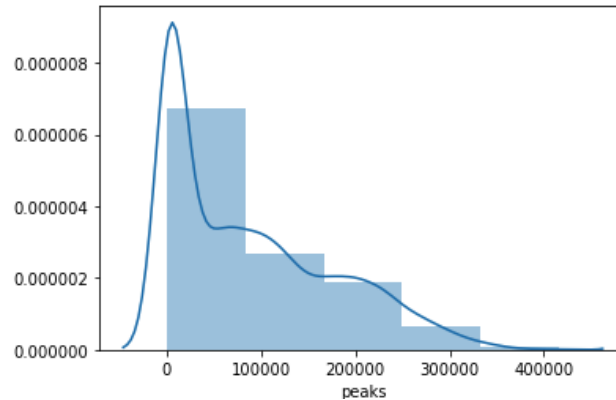
**Step 2)** Using a Butterworth filter [6] I remove all frequencies below  $10^4$  Hz. Then using wavelet decomposition and thresholding [7] I filter out low amplitude noise from the sample.

**Step 3)** The datapoints (0, 50000), (300000, 450000), (700000, 800000) are cut away from the sample, reducing each sample to 500.000 data points. This is done as the occurrence of the PD patterns is not expected at these locations. PD patterns are only seen at specific locations on the 50Hz base sinewave:



I applied Steps 1), 2) and 3) on all 8712 training samples as well as all 20.336 test samples as a batch job, creating new parquet files.

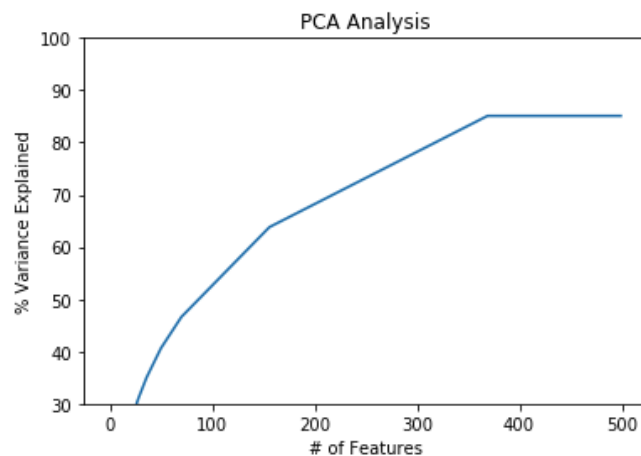
The next steps for the training data was to do undersampling, as the dataset is imbalanced with 525 positive and 8187 negative training samples. To be able to select relevant training samples from the training data I decided to count the amount of amplitude peaks in each of the pre-processed samples and categorized them in 4 different bins:



I then selected all 525 PD positive samples and added 525 random samples from each of the 4 bins resulting in a dataset with 2625 samples.

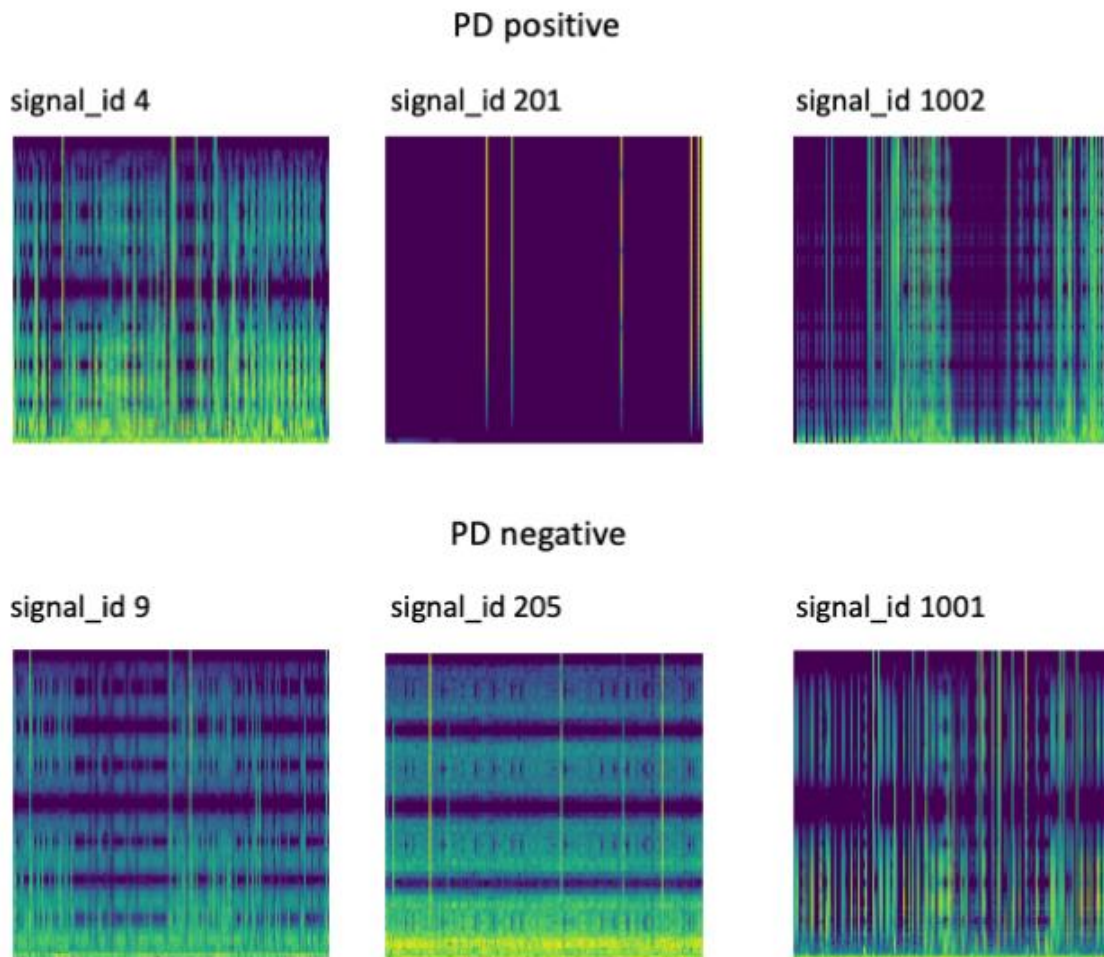
As the next step I reduced the features from the 500.000 measurements per sample down to something I could use as an input to a neural network. I tested two very different approaches whose results I will discuss in the later parts of this document. The first approach was to use Principal Component Analysis (PCA), the second was to generate small 244x224x3 (RGB) sized spectrogram pictures from the samples.

For the PCA approach I first searched for the minimum number of components to use. Using the `explained_variance_ratio_method` I looked for the number of features where the percentage of variance explained did not improve anymore:



The number of components I used turned out to be 369. I finally transformed the 2625 selected samples and stored the resulting 2625x369 dataset to feed it into a neural network later.

For the second approach I used the `specgram` method of `matplotlib` to generate 244x224x3 spectrogram pictures and store them on disk in separate 'pd\_positive' and 'pd\_negative' folders. Here are a couple of examples:



As you will likely see it is very difficult for the human eye to spot the key differences between samples with detected PD patterns and samples that are negative. As I will show in the results section it is astonishing that the 2D neural networks (CNNs) are still able to recognize the patterns relatively well.

The test data (20.336 test samples) was directly converted to the spectrogram pictures only after the pre-processing steps. As you will see later in the document the PCA results were not encouraging, therefore creating a test PCA dataset wasn't needed.

## Implementation

As I already described earlier, I tested 4 different combinations of feature reduction and neural networks:

Input Data	Neural Network Type
PCA	Dense network
PCA	1D CNN
Spectrogram pictures	2D CNN
Spectrogram pictures	2D CNN - Transfer learning VGG16

In all cases the training input data was split into a training and validation dataset, with the validation dataset being 20% of the total available training data. I did not carve out a test dataset from the training data, as a large test dataset was available from Kaggle, and I did not want to lose another 20% of the training data. This however comes with the drawback that only the MCC score is available as a result of the test run and evaluation of the test dataset is only available through the Kaggle web site.

I initially started the training of the Dense Network and 1D CNN on my Laptop without GPUs. Later when training the 2D CNNs this was not possible anymore, therefore I switched to using Google colab [\[10\]](#) and ran the Training and Evaluation Notebooks there using GPUs.

## PCA + Dense Network implementation

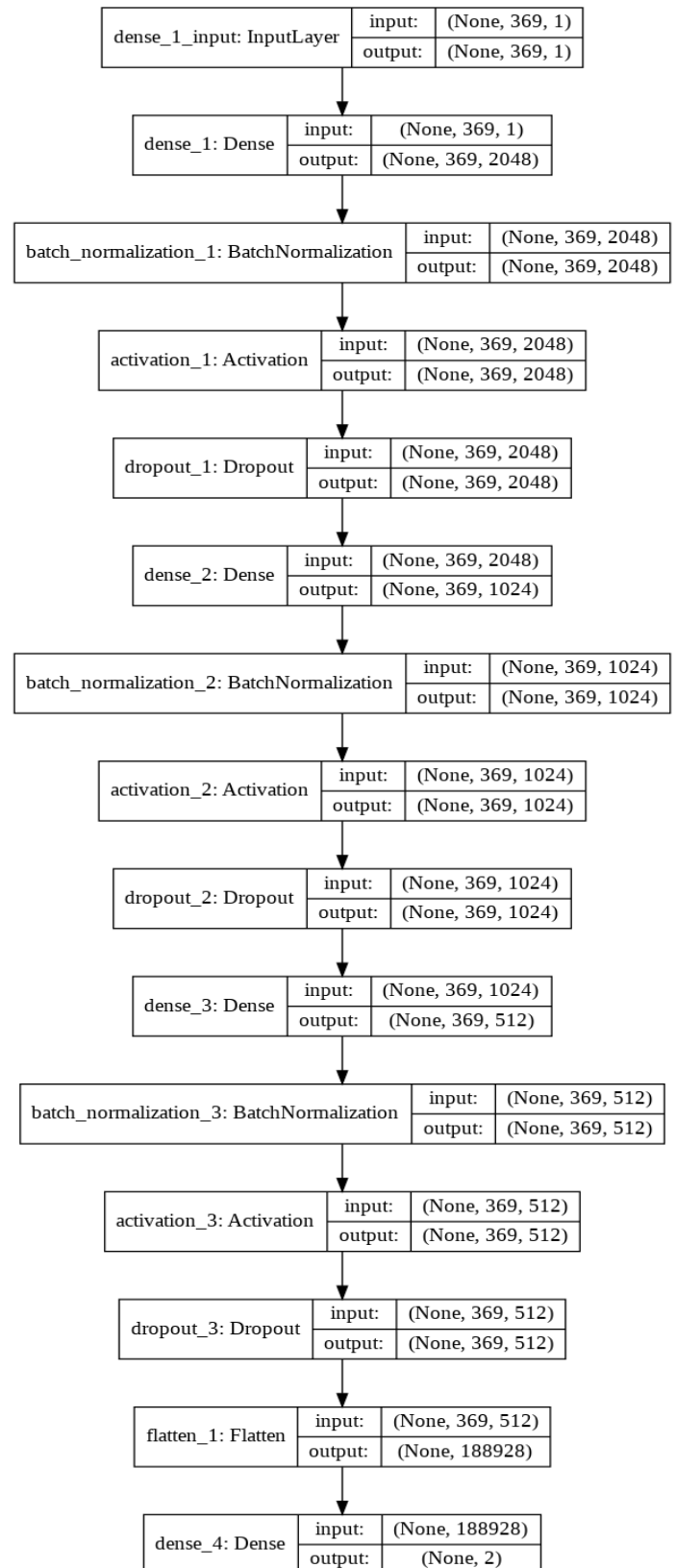
The following graphic is directly exported from Keras and shows the layers and number of nodes I choose for the Dense Network.

The first layer takes the 1-dimensional input of the 369 PCA components as Input but reshaped to (369, 1).

As you can see, I use Batch Normalization before and a Dropout of 0.35 after the Relu activation in each layer.

The final layer has an output of 2, so the output is indicating the probability of two categories, the 'PD is present' and 'PD is absent' category.

Before evaluating the performance using Accuracy, Precision, Recall, F1-Score and MCC functions I convert the probabilities into binary values, taking the highest probability as 1 and the lowest as 0.



## PCA + 1D CNN implementation

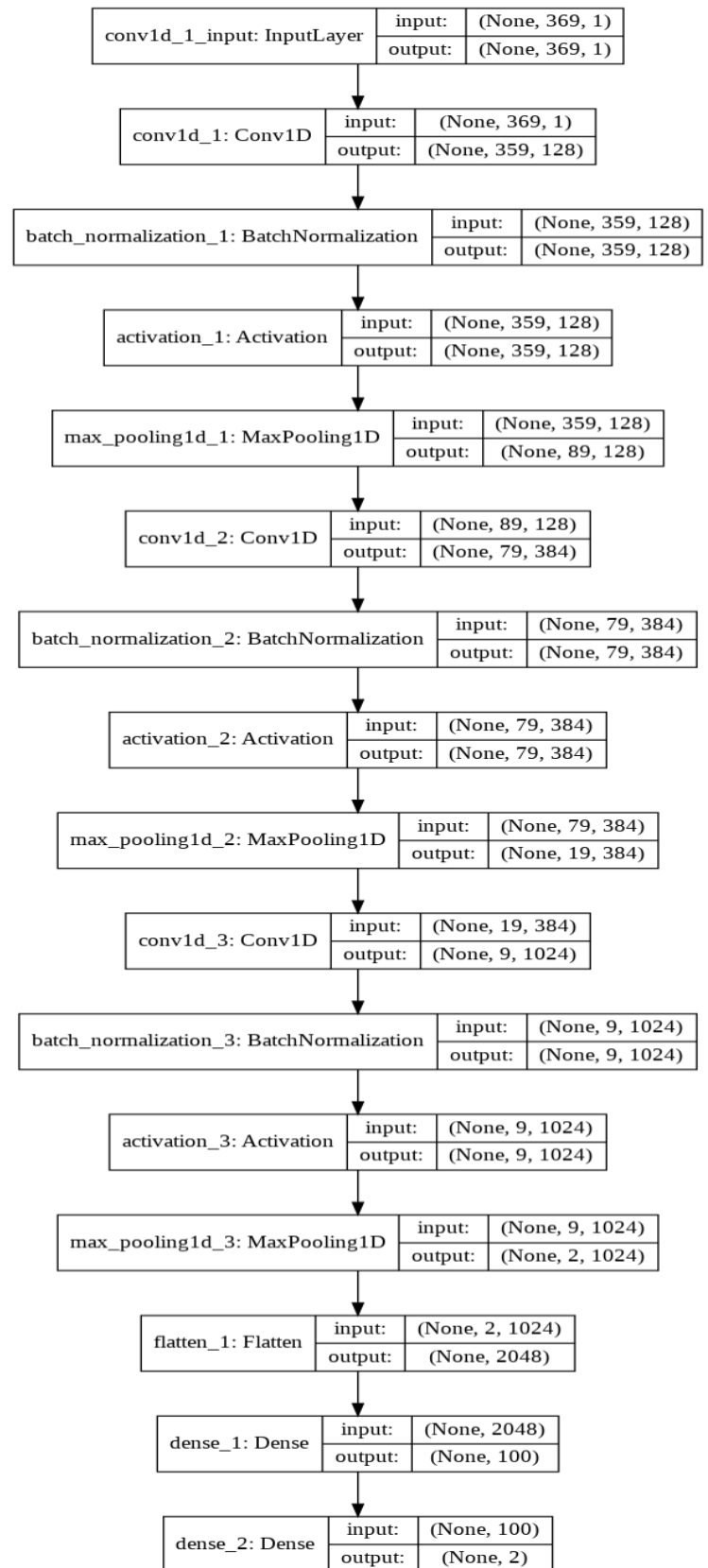
The 1-D CCN implementation is similar to the Dense Network on the Input side.

The first layer takes the 1-dimensional input of the 369 PCA components as Input but reshaped to (369, 1).

I am stacking 1D convolutional layers (Conv1D) with Batch Normalization, Relu activation and MaxPooling1D layers.

In the final layer I flatten the output of the last MaxPooling1D layer and run it through two final Dense layers.

Again, the output is consisting two categories, 'PD is present' and 'PD is absent'





## Spectrogram pictures + 2D CNN implementation

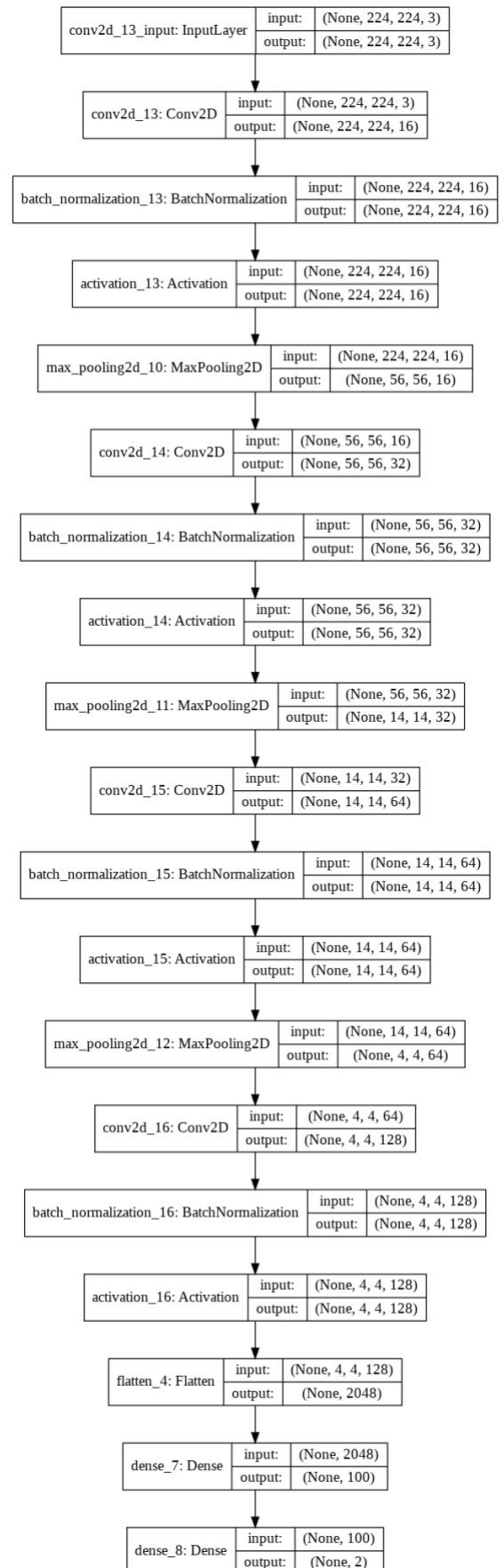
The input layer of the spectrograms + 2D CNN implementation is obviously very different from the PCA input layer as it is now a 224x224x3 RGB picture.

Each picture is converted to a 4D tensor with the shape (1, 224, 224, 3), and then rescaled so that every pixels RGB color information is converted from an integer ranging from 0 to 255 to a floating point number between 0.0 and 1.0.

These rescaled tensors are now used as the input to the first layer.

The stacking of layers I choose is very similar to the 1D + PCA implementation but with the 2D equivalents (Conv2D).

The final Dense layers again lead to a 2 categorical output representing 'PD is present' and 'PD is absent'.



## Spectrogram pictures + 2D CNN Transfer learning VGG16 implementation

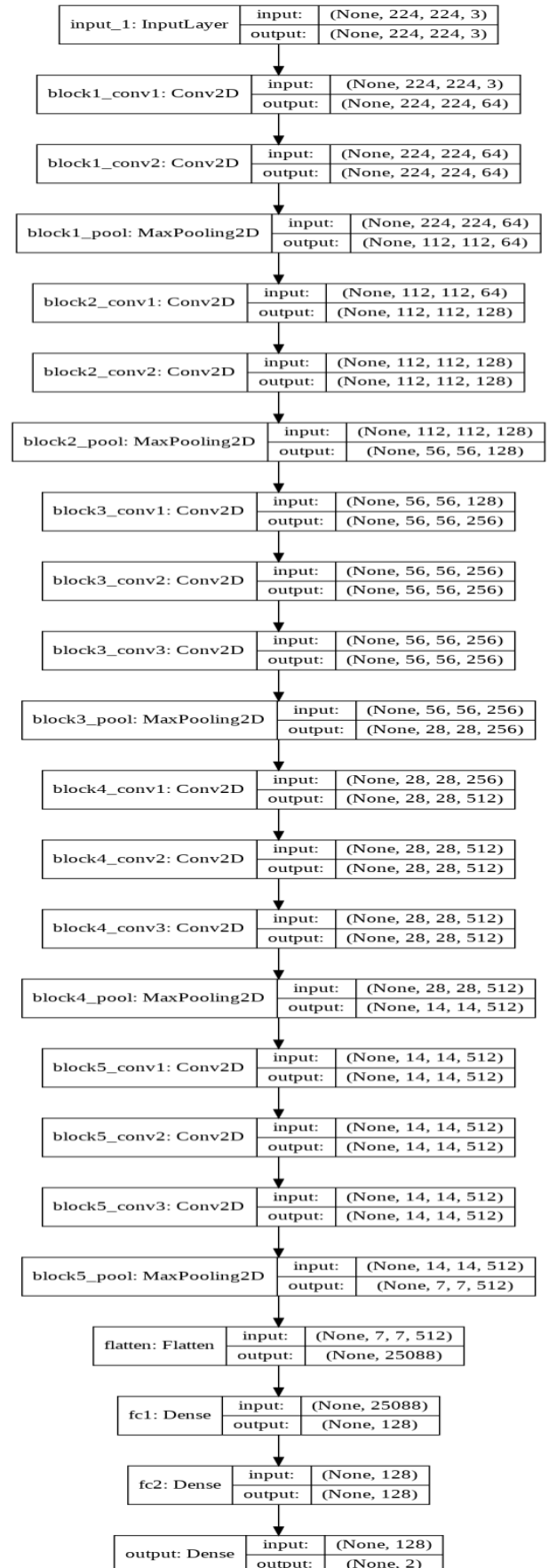
The Transfer Learning implementation is using the exact same input layer and reshaping as the non-transfer learning implementation.

As final base model I used VGG16 with pre-loaded weights trained on imagenet.

I'm using the VGG16 network as a feature extractor, so I'm exchanging the last classifier layers of the VGG16 network with my own two dense layers with 128 nodes.

During training I am freezing the VGG16 layer up to 'block5\_pool' which is the final pooling layer before the custom dense layers I added.

The output layer of the Transfer Learning implementation is the same as in the custom implementation, again leading to a 2 categorical output representing 'PD is present' and 'PD is absent'.



## Common Training implementation and settings

For all neural networks I am using the SGD optimizer with a learning rate of 0.005. I am using 'categorical\_crossentropy' as the loss function in all implementation.

During the training I am using the ModelCheckpoint and EarlyStopping callbacks. I checkpoint the weights of the last epoch that lead to a reduction of the loss (ModelCheckpoint) to disk, and I stop the training after 15 epochs that did not lead to any improvement of the loss.

Before evaluating the trained model with Accuracy, Precision, Recall, F1-Score and MCC functions I load the latest best checkpoint.

## Refinement

### PCA + Dense Network implementation

The 3 layers with their sizes 2048, 1024 and 512 were used from the beginning on and seem to be a good sizing. I initially did not use Batch Normalization, after starting to use it between the Dense layer and the Relu activation I saw some good improvement in the validation results.

I initially started with a dropout rate of 0.5 between the layers. Reducing it to 0.35 improved the results slightly. I also experimented with the use of Dropouts and the results were better with it.

### PCA + 1D CNN implementation

I initially used Dropouts in the 1D CNN too. However I saw better results when not using Dropouts at all. I used less nodes to begin with when training the models on a CPU, but after moving to Google colab and having GPU support I achieved better results with using more nodes.

### Spectrogram pictures + 2D CNN implementation

The model I initially chose performed pretty well from the beginning on. I added more nodes after having access to GPUs, else I kept it pretty constant over the course of the project.

### Spectrogram pictures + 2D CNN Transfer learning VGG16 implementation

I tested several pre-trained networks, always exchanging the final categorical layers with the two 128 node dense layers I described earlier. I tested ResNet50, ResNeXt50, VGG16, VGG19 and InceptionV3. I had the best results using VGG16 and chose it as my final model.

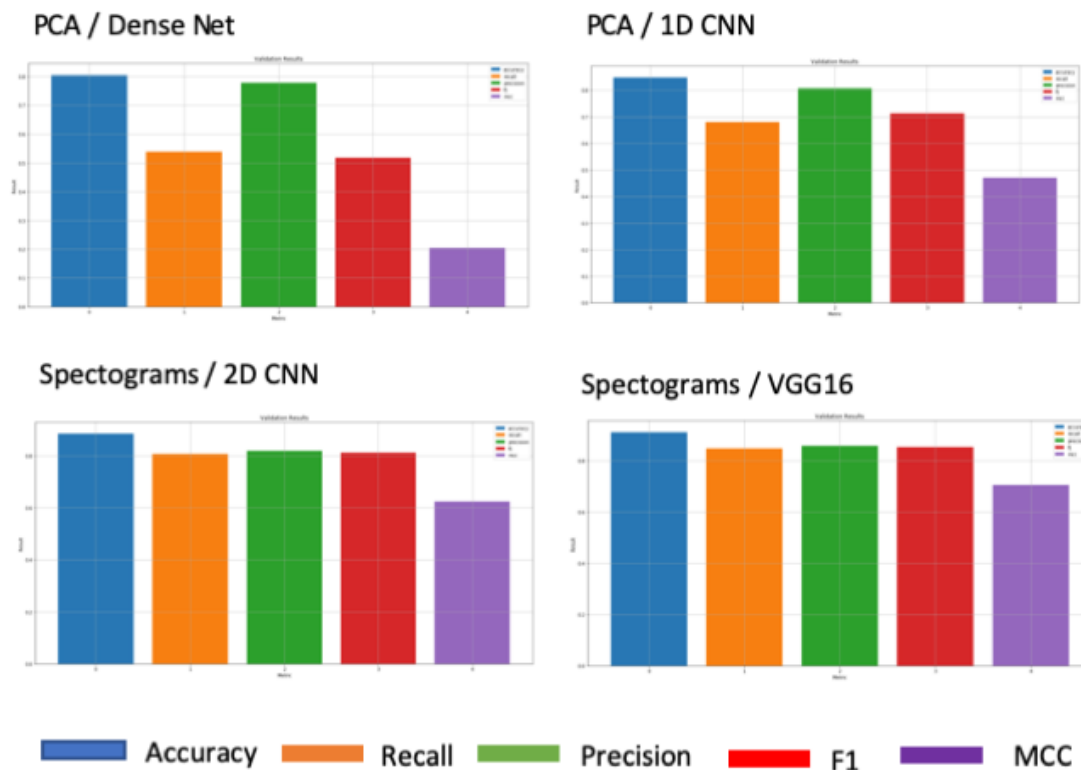
## IV. Results

### Model Evaluation and Validation

I validated the training of every model described in the implementation section with 20% of the original training data. As discussed already, the Kaggle competition provides a large test dataset with 20.336 test samples. However the test dataset doesn't include the truth labels in the test metadata. Therefore I was only able to calculate the Accuracy, Precision, Recall and F1-Score only using the validation data carved out from the training dataset. For the test dataset I have the MCC score for the final two models I choose to be evaluated in the competition.

Here are the validation results for the various models:

Input data / model	Accuracy	Recall	Precision	F1-Score	MCC
PCA / Dense Net	0.8038	0.5377	0.7775	0.5189	0.2045
PCA / 1D CNN	0.8476	0.6804	0.8067	0.7138	0.4705
Spectrograms / 2D CNN	0.8838	0.8066	0.8185	0.8123	0.6250
Spectrograms / VGG16	0.9086	0.8477	0.8584	0.8529	0.7060



I only evaluated the two spectrogram based models using the test dataset as the results using the validation dataset were better using those implementations.

Here are the results if the Kaggle submissions using the predictions done on the test dataset:

Input data / model	MCC private leaderboard	MCC public leaderboard
Spectrograms / 2D CNN	0.41760	0.33451
Spectrograms / VGG16	0.41149	0.31039

The private leaderboard is calculated with approximately 43% of the test data, the public leaderboard is calculated with the rest 57% of the test data

As one can see the final MCC scores using the test dataset are not as good as the MCC scores of the validation dataset. This shows me that I would need to invest more time to make the spectrogram/CNN approach generalize better. My goal of being in the mid-range of the Kaggle leaderboard was however achieved, as the lowest score in the private leaderboard was 0.0, and the highest was 0.71899, putting me slightly above the mid-range with my results.

## Justification

My initial goal of getting results comparable or better than the results published in the paper written by the technical University of Ostrava [\[1\]](#) using a Neural Network implementation was achieved. As a recall, here are the scores the Random Forest implementation of the technical University of Ostrava achieved:

Metric	Result
Accuracy	99.8
Precision	83.6
Recall	66.7
F1-Score	72.8

I was able to achieve better results with the Spectograms/VGG16 approach for Precision, Recall and F1-Score which are the key metrics for the problem to be solved. Even the PCA implementations came close to the benchmark results.

I feel that especially the good score for Recall shows that the Spectograms/VGG16 approach is capable of solving the problem of detecting PD patterns pretty well. However, the MCC results in the Kaggle competition also show that more effort would need to be put into generalizing the solution better.

## V. Conclusion

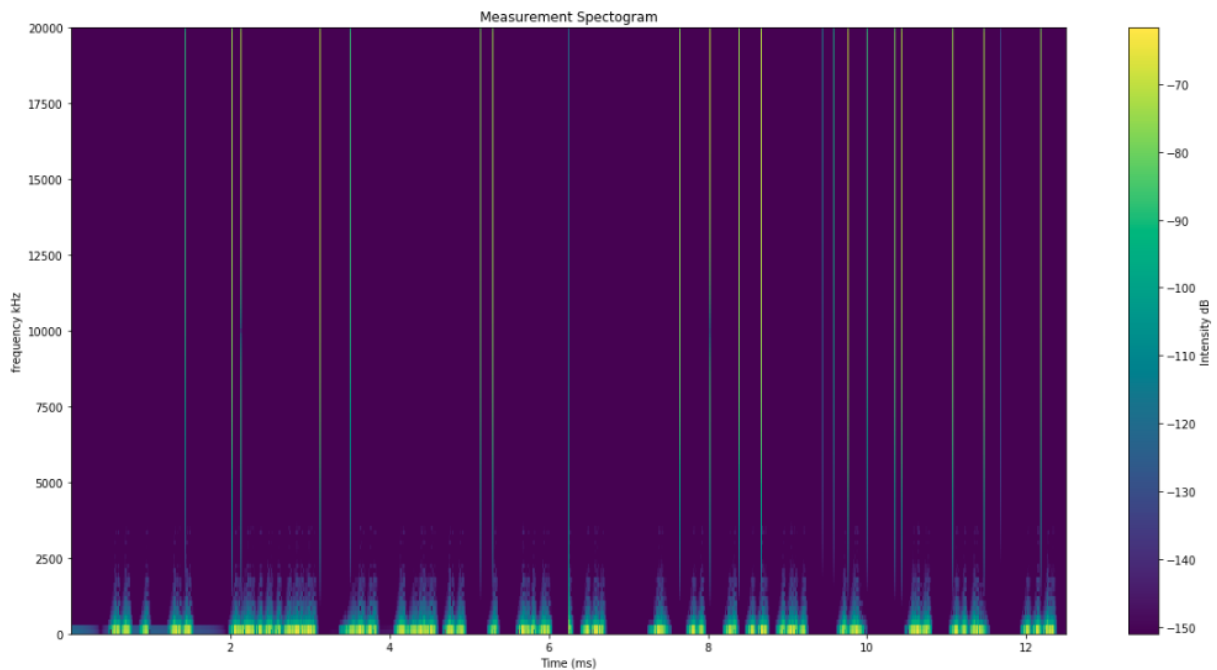
### Free-Form Visualization

The key finding for me is that tools used to visualize a measurement for humans can generate input layers for machine learning approaches that are trained to recognize images.

What initially looks like plain rows of numbers:

	0	1	2	3	4	5	6	7	8	9	...	8702	8703	8704	8705	8706	8707	8708	8709	8710	8711
0	18	1	-19	-16	-5	19	-15	15	-1	-16	...	18	-22	12	8	13	6	-21	-15	-9	20
1	18	0	-19	-17	-6	19	-17	16	0	-15	...	17	-21	12	8	14	7	-19	-15	-8	21
2	17	-1	-20	-17	-6	19	-17	15	-3	-15	...	16	-21	13	8	15	8	-18	-14	-8	22
3	18	1	-19	-16	-5	20	-16	16	0	-15	...	16	-21	12	8	15	8	-19	-14	-7	23
4	18	0	-19	-16	-5	20	-17	16	-2	-14	...	17	-22	12	8	15	8	-18	-14	-8	23

Can be converted to:



and be used as an input for a machine learning approach

## Reflection

The most important part of this project was the exploration of the dataset, and the work that went into developing an understanding of the problem domain.

The key to the project was the initial filtering of the data using technologies like Butterworth filters and Wavelet decomposition.

The analysis of the problem in the time and frequency domain was the most challenging aspect of the project, but also the part where I learned the most. This is where I came to the conclusion that instead of using feature extraction techniques like PCA, one could also use tools developed to visualize the time/frequency domain to generate input for neural networks used today mostly for image recognition problems.

It is fascinating to see how Convolutional Neural Networks are able to discover patterns in images like the spectrograms, and achieve good classification results for time/frequency problems even when they were originally trained to e.g. distinguish cats from dogs.

## Improvement

There are a number of improvements I could easily spend another couple of weeks with to refine this project.

The solution I chose using 'manual' undersampling and selecting samples based on categories built by counting the number of amplitude peaks seems to work. However I would also like to test oversampling solutions like SMOTE, or maybe look into various clustering technologies to create better 'bins' of samples. This might be a key improvement to make the solution generalize better.

Another part that I invested significant time in, but had to give up finalizing, is to develop better ways to create spectrogram pictures. The matplotlib function I'm currently using [4] is based on Fast Fourier transform [5]. I'm expecting better results when using Continuous Wavelet Transform (CWT) [11] to generate spectrogram pictures. However the only open source python implementation I found was pywavelet (pywt) [12], and its capabilities are still pretty bare bones. I could have looked into using a solution using MathWorks Wavelet tools [13], but I did not want to include tools in this project for which I need a trial license. I am very confident that I could significantly improve the results by using CWT instead of FFT with windowing.

Also, generating spectrogram pictures with higher resolutions, combined with CNNs that can handle larger inputs might improve the implementation.

While I feel that the spectrogram/CNN approach is a very promising one, another approach that might lead to good results, but that I did not have enough time to explore, is to use Wavelet Decomposition with feature reduction. When using Wavelet Decomposition one can generate features from the coefficients like count of peaks, min/max amplitude, etc. in each frequency range. These features can then be used as an input to a neural network too, and might lead to better results than the spectrogram approach with even less data needed to be stored.



## Links to Notebooks and Python models used in this project

- 1) [Train Data exploration](#)
- 2) [Train Data Filtering](#)
- 3) [Train Data Undersampling](#)
- 4) [Visualization Python Module](#)
- 5) [Filtering Python Module](#)
- 6) [Train Data PCA feature reduction](#)
- 7) [Train Data Spectrogram Creation](#)
- 8) [PCA with Dense Network Training and validation](#)
- 9) [PCA with 1D CNN Training and validation](#)
- 10) [Spectrograms with 2D CNN Training and validation](#)
- 11) [Spectrograms with 2D CNN VGG16 Training and validation](#)
- 12) [Test Data Exploration](#)
- 13) [Test Data Filtering](#)
- 14) [Test Data Spectrogram Creation](#)
- 15) [Test Data 2D CNN predictions](#)
- 16) [Test Data 2D CNN VGG16 predictions](#)

## Links to resources and papers

[1] Technical University of Ostrava academic paper:  
<https://ieeexplore.ieee.org/document/7909221>

[1] Technical University of Ostrava academic paper - Dropbox Link:  
<https://www.dropbox.com/s/2ltuvpw1b1ms2uu/A%20Complex%20Classification%20Approach%20of%20Partial%20Discharges%20from%20Covered%20Conductors%20in%20Real%20Environment%20%28preprint%29.pdf?dl=0>

[2] Wikipedia entry for Partial Discharge (PD):  
[https://en.wikipedia.org/wiki/Partial\\_discharge](https://en.wikipedia.org/wiki/Partial_discharge)

[3] Kaggle vsb-power-line-fault-detection data: <https://www.kaggle.com/c/vsb-power-line-fault-detection/data>

[4] Matplotlib specgram function:  
[https://matplotlib.org/3.1.1/api/\\_as\\_gen/matplotlib.pyplot.specgram.html](https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.specgram.html)

[5] Fast Fourier Transform Wikipedia entry:  
[https://en.wikipedia.org/wiki/Fast\\_Fourier\\_transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform)

[6] Butterworth Filter Wikipedia entry: [https://en.wikipedia.org/wiki/Butterworth\\_filter](https://en.wikipedia.org/wiki/Butterworth_filter)

[7] Using pywavelet to remove high frequency noise: <http://connor-johnson.com/2016/01/24/using-pywavelets-to-remove-high-frequency-noise/>

[8] Kaggle vsb-power-line-fault-detection leader-board: <https://www.kaggle.com/c/vsb-power-line-fault-detection/leaderboard>

[9] Article on the location of PD patterns: [http://www.hitequest.com/Kiss/hv\\_isolation.htm](http://www.hitequest.com/Kiss/hv_isolation.htm)

[10] Google colab: <https://colab.research.google.com>

[11] Continuous wavelet transform Wikipedia entry:  
[https://en.wikipedia.org/wiki/Continuous\\_wavelet\\_transform](https://en.wikipedia.org/wiki/Continuous_wavelet_transform)

[12] Pywavelet: <https://pywavelets.readthedocs.io>

[13] MathWorks wavelet toolbox: <https://www.mathworks.com/help/wavelet/ug/time-frequency-analysis-and-continuous-wavelet-transform.html>