Application of CNN architectures for the Classification of Structure borne sound

Specialization Project in Electrical Engineering

Date of submission: 20.07.2018

Submitted by Sergej Bauer

Guided by Prof. Dr.-Ing. Sascha Spors

Institute of Communication Engineering, University of Rostock

Table of Contents

Definition of the task

Classifying bulk materials by the structure borne sound they emit is a quite challenging task in the field of audio signal processing. Most applications of acoustic classification focus on signals with a rather regular and harmonic structure, e.g. music or speech (1), (2). In the case of environmental sound, less regular and more complex patterns have to be analyzed. Currently, there is a strong trend in applying neural network architectures to solve classification problems. Utilizing modern processing power, these networks are able to successfully classify complex inputs, like pictures with several objects and layers. Such promising advances led to the application of neural networks in classifying environmental sound.

The following work proposes the application of two specific convolutional neural networks (CNN) to the problem of classifying bulk materials. It was build on the results of a previous work (3), which already successfully applied standard depp neural networks (DNN) to this problem. Besides changing the underlying network, the following work will apply different methods for feature extraction and data augmentation in order to improve the classification results. Furthermore, a real time application is proposed, which utilizes the pre-trained network to classify incoming sound information, which is simultaneously recorded by a set of microphones. This way, the CNN setup can be shown to be applicable to real physical environemnts, instead of just relying on precorded signal data.

Characteristics of the physical environment

The following chapter provides a brief overview regarding the main characteristics of the physical environment. A good summary can be already found in the prevoius work (3). Here, the main goal is to slightly extend this summary by providing additional details, which may be of importance for the neural network and the underlying classification algorithm.

Structure borne sound

Sound, which travels through and/or between solid materials (like metal, wood, plastic etc.) can be defined as structure borne. Having different attenuation factors, the materials chosen can have an significant impact on the classification results. By absorbing acoustic energy, the materials determine how much is left for the classifier. This could e.g. result in a significant reduction of bandwidth by attenuating high-frequency components, leading to a worse network performance, if important features are located in the absorbed spectral region. The surrounding environment can have a strong influence, too. Factors like air humidity might affect the transmission of sound waves, while background noise might obscure important features of the input samples.

In the context of this work, the following assumptions are made about the classification context: The bulk materials (which will be defined in the next section) are thrown down on a ramp (e.g. representing an assembly line), made of aluminium. The ramp is positioned at a certain angle, causing the materials to roll down. One microphone pair, directly connected to the ramp records the created sound waves. For the purpose of simplicity, only one type of bulk material is used for each "run".

Bulk materials

Materials, which are manufactured, stored and sold by weight or volume can be defined as bulk materials. In the context of this work, 3 types of materials are considered: spheres, nuts and screws. Nuts and screws are classified e.g. by the DIN 934 (4), which defines their scale by values of M1, M2, M1.4 etc. Higher numbers correspond to bigger nuts and screws. The size affects their acoustic bevaviour, too. Having nuts with significantly different scales would result in quite different acoustical patterns, thus benefiting the classification algorithm. In the opposite case, a small difference would complicate the classification, possibly requiring a fine-tuned feature extraction or a large datasets. The specific material of the spheres, nuts and screws could also affect the classification, as different metals, plastics etc. have unique acoustic behaviour.

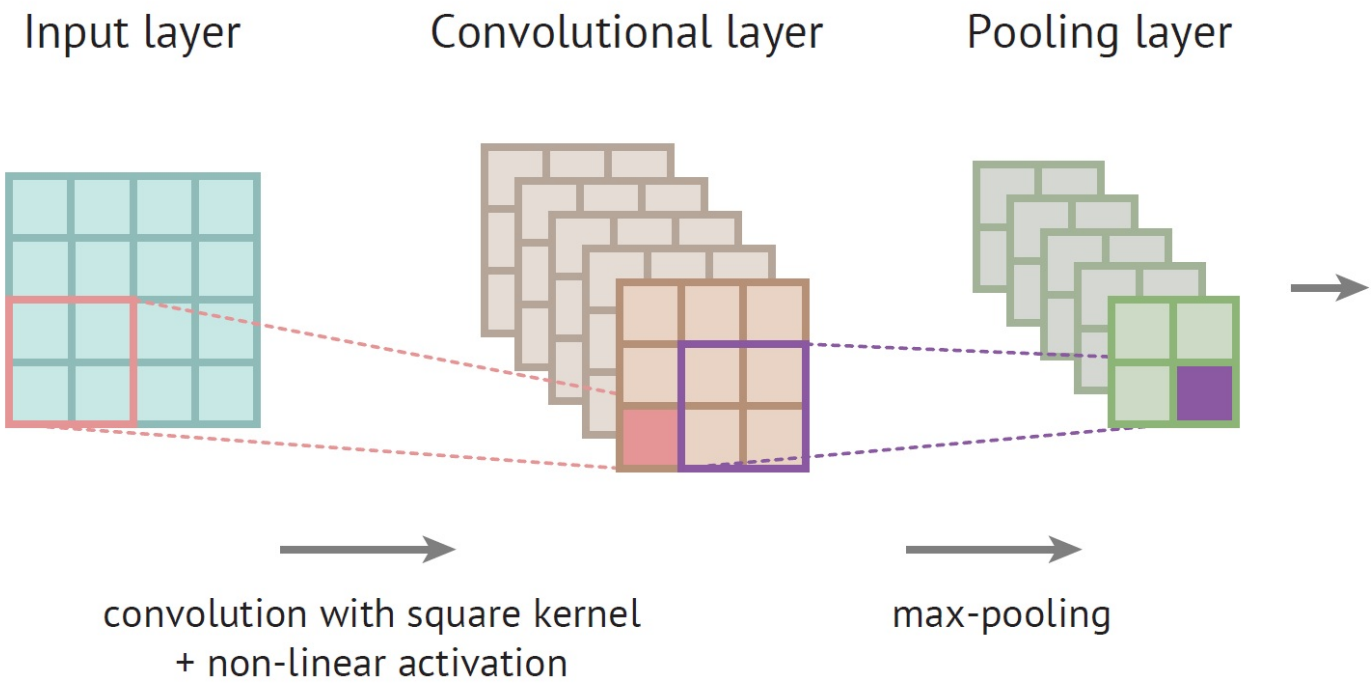
In the context of this work, we define 5 classes of bulk materials: plastic spheres, M3 steel nuts, M4 steel/messing nuts and M3 screws.

Characteristics of the network

After defining the physical environment from which we extract the features, the following chapter will discuss the underlying network architecture. First, a brief introduction to convolutional neural networks (CNN) is presented, togehter with possible applications in audio signal processing. In the next chapter, the actual implementation for this work will be presented, using code listings and their corresponding explanations.

Convolutional neural networks

CNNs are a specific type of neural network and closely related to conventional deep neural networks (which were already discussed in great detail in the prevoius work). Their were originally used for the problem of image classification. Similar to DNNs, CNNs also contain input and output layer as well as one or several hidden layers. While DNNs usually process one-dimensional features, CNNs can be applied to two-dimensional inputs (hence their application to images). The following scheme illustrates the basic structure of these networks (5):



After presenting a two-dimensional input, a convolutional layer processes the sample by applying a set of filter kernels, which perform local scans on the sample. Depending on the filter size, more general or more specific patterns can be extracted. The number of filters applied affects the ability of the network to detect more nuanced structures. During backpropagation, the specific values of the filters are adjusted to incorporate the learned outcomes. After convolution, the sample size is reduced in the first two dimensions (due to the filter kernels) and extended in the third dimension, proportional to the number of filters. As a next step, a different type of layer might be applied, the pooling layer. Its purpose is to reduce dimensionality by summarizing parts of the filter values, e.g. by choosing the highest value within a subset of the filter (max pooling). This layer is often applied in applications with large input samples (e.g. high resolution images) and fine, detailed structures. Omiting these layers would increase the computational efforts for the subsequent components. Using these two types of layers, one can construct arbitrary combinations of convolution and pooling, specifically for each application. After the dimensions of the samples are sufficiently reduced, several fully connected layer, known from conventional DNNs, are used to perform the final classification.

Similar to DNNs, the choice of the network parameters (e.g. filter size and number, sequence of layers, pooling operations etc.) is often highly specific to the application. Thus, the parametrisation for this work is derived from previous research papers, which successfully applied CNNs for the classification of environmental sound.

Application to environmental sound

While most of the work in CNNs was done using image applications, there is also an active research in applying this network architecture to the area of audio signal processing. Pizak (5) used log-mel spectrograms and their corresponding deltas to construct two-dimensional feature spaces, which are then processed and classified using four CNN-type and two conventional DNN-type layers. Tokozume et al (6) omitted any method for data transformation and worked instead with the original sound recordings. And Virtanen et al (7) created samples from magnitude responses, calculated by means of the short time Fourier transform (STFT) in order to classify home environments.
Given the above examples, several conclusions can be drawn: First, Tokozume et al has shown that CNNs can work with one-dimensional input data, too. Here, the samples are extended in the second and third dimension further down the network. Second, there are numerous types of two-dimensional feature spaces possible, although many applications use some variant of the spectrogram. And third and quite typical for deep learning: all architectures require large datasets, which are either already given or created using data augmentation. Many applications feature acoustic environments with mixed signals and sources (8), which usually requires more sophisticated methods for feature extraction and network construction. In the context of this work, we will focus on classifying one signal at a time. As a consequence, some of the approaches in the research papers are not taken in their original form, but rather edited and "simplified" to fit the context.

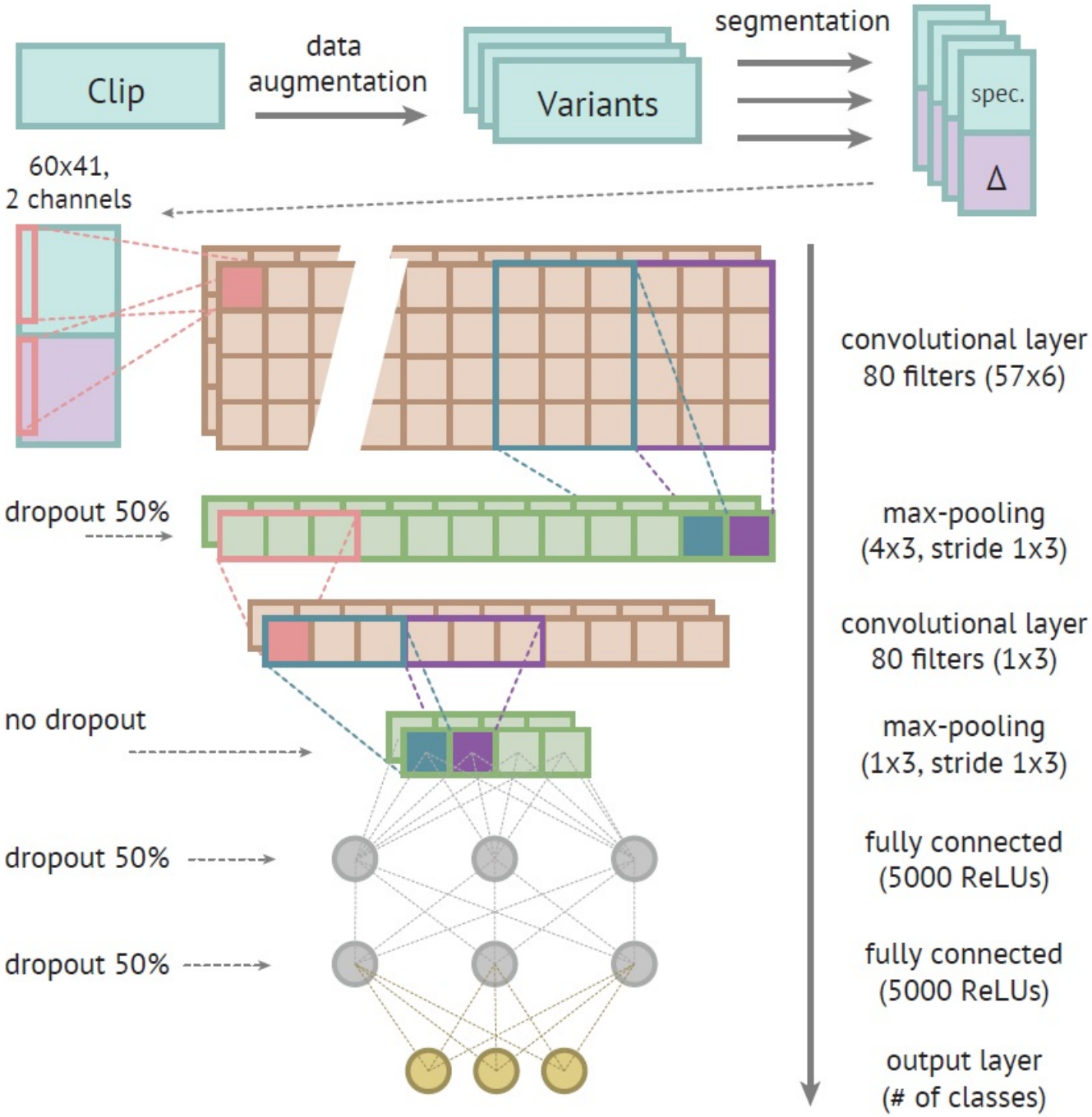
Implementation of the CNN

This chapter will discuss the actual implementation of the neural network in the context of this work. First, the models and the software, which are used for the application are presented and briefly explained. Then, the source code for the models as well as the simulation results are analyzed and discussed.

Network Models

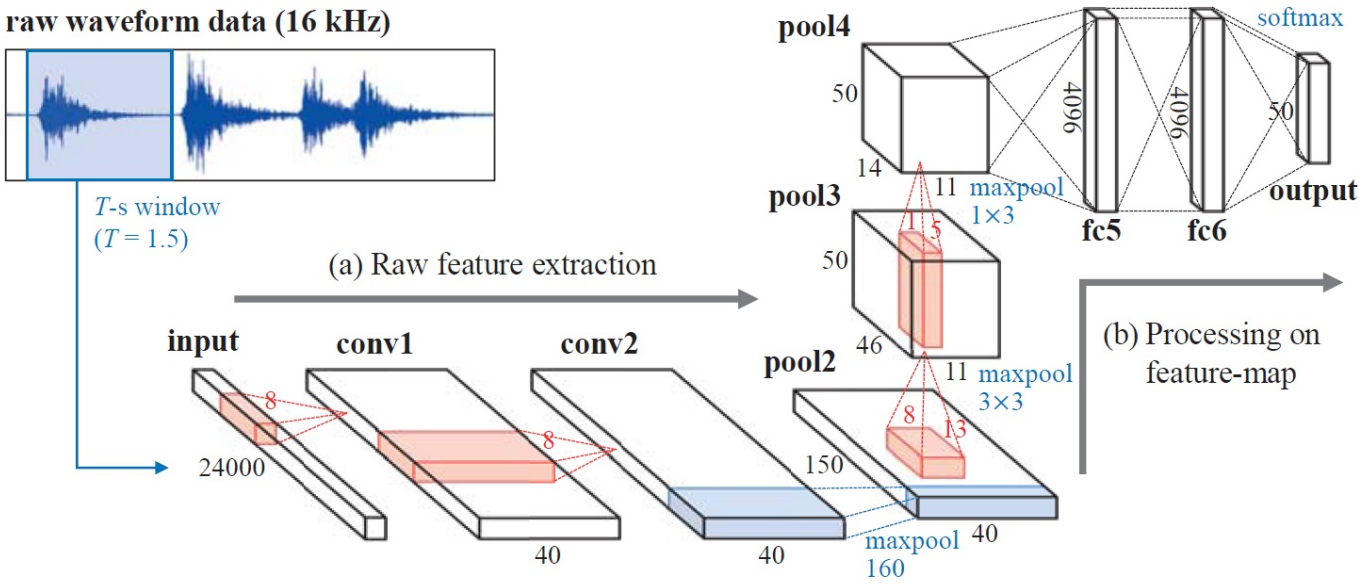
Based heavily on the previous work in (5) and (6), two models were derived and implemented.

The first one, described in (5), uses the *ESC-50* (9), *ESC-10* (a subset of the *ESC-50*) and the *UrbanSound8K* (10) datasets as sources for audio samples, using data augmentation to create even more variants. From them, they extracted log-scaled mel-spectrograms as the feature set from the sample inputs. The log scaling changes the Fourier-transformed convolution of the signal and the impulse response from a multiplication to an addition and thus simplifies the separation of both. The projection into the Mel-frequency space provides a more compact representation of the spectrogram, reducing the number of frequency bands from e.g. 256 to e.g. 40 (11). Log-scaled mel-spectrograms (and the corresponding Mel-frequency cepstral coefficients (MFCC)) are quite often used in natural language processing (11). The feature space was extended by calculating the deltas of the spectrogram, e.g. the change in spectral intensity between two time instances. Combined, a two-dimensional and two-channel (spectrogram and deltas) feature space was created and fed into a network with 6 layers (output layer not included). The network first processes the inputs with a convolutional layer, consisting of 80 Filters of rectangular shape. Afterwards, max pooling is applied. Further filtering is done by a second convolutional layer, using again 80 filters of rectangular shape. After applying a second max pooling layer, the processed and reduced data set is fed into two fully connected layers of a conventional DNN, using ReLUs as activation functions. The classification is then finally generated by applying a softmax output layer. The following graphic illustrates the full network architecture (5).



The data set available for this work had significantly different dimensions than the one used in the above paper. As a result, the concrete parameters for feature extraction and network design had to be altered to fit the context of this work.

The second one, described in (6), also uses the *ESC-50* dataset. But instead of deriving features by subsequent transformations, the authors used the raw waveform data, dividing it into sections of several seconds and randomly selecting them for training. These samples are then fed into the network, consisting of 8 layers (output layer not included). First, the samples are processed by two subsequent convolutional layers with 40 filters each and a max pooling layer after the processing. This is described as "Raw-feature extraction". As a next step, a special pooling layer is applied: a time-series of 40-dimensional vectors, where each vector "can be thought of as representing frequency-like features" (6). It is further assumed, that the components of the vector "are learned to be arranged according to some type of law". Afterwards, two addtional convolutional layers are applied, with 50 filters each and again a subsequent max pooling operation. Finally, two fully connected layers are used, with ReLU activation functions and a softmax output layer. The following diagram shows the complete architecture.



As with the first model, additional efforts were spent to fit all parameters to the context of this work. Additionally, the feature extraction was slightly altered: after some dissatisfying first tests with this model (characterized by a disappointing classification performance), the input sections are now transformed using the magnitude response spectrum of the real-valued discrete Fourier transfrom (DFT), which is similar to the approach in the previous work (3). The transformation keeps all dimensions of the input signal, thus the following steps remain unchanged.

Software prerequisites

For the purpose of the implementation, several software packages and frameworks were used: All source code was written in Python, using Version 2. In order to use complex mathematical operations (e.g. the DFT, the extraction of log-mel spectrograms, data augmentation etc.) efficiently, the libaries *NumPy*, *SciPy* and *Librosa* (12) were included. To design and simulate the neural networks, *TensorFlow* (13) and *TFLearn* (14) were used. For the evaluation of the network performance, the library *scikit-learn* (15) was imported.

CNNs perform complex filtering and pooling operations, requiring a substantial amount of computation time and power. In order to reduce the training time, the models were trained on an external server running Jupyter Hub.

Implementation of the first model

In [ ]:

```
import tensorflow as tf
import numpy as np
import scipy
import librosa
import tflearn
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
```

First, all required libaries are imported for later use.

In [ ]:

```
def add_time_stretch(data, stretch_factor):
    data_stretched = np.array([librosa.effects.time_stretch(data[:,i], stretch_factor) for i in [0,1]]).T
    return data_stretched

def add_pitch_shift(data, samplerate, shifts):
    data_shifted = np.array([librosa.effects.pitch_shift(data[:,i], samplerate, n_steps=shifts) for i in [0,
1]]).T
    return data_shifted

def add_white_noise(data, mean, var):
    noise = np.random.normal(mean, var, data.shape[0])
    data_noise = np.array([data[:,i] + noise for i in [0,1]]).T
    return data_noise
```

In order to perform data augmentation, a few functions were defined in the above listing, with inspiration taken from (16). The first one, `add_time_stretch` , changes the length of the sample file by increasing or decreasing the playback speed of the WAV file. This might e.g. correspond with different slopes on the ramp, where a higher slope results in a faster downward motion of the bulk materials and thus in a "accelerated" version of the original recording.
The `add_pitch_shift` function changes the frequency composition of the signal by increasing or decreasing the pitch, without altering the length. Different pitch values might correspond to differrent ramp materials: If made out of a light and oscillating material, the ramp might produce a different pitch than a heavier version.
As a last example, `add_white_noise` was created. The function adds normally distributed, random values with a given mean and variance to the signal, effectively simulating the cumulative noise effect of the microphones and the surrounding environment. Internally, the first two functions use the *librosa* library, which provides built-in methods for data augmentation tasks.

In [ ]:

```
segment_size = 1024  # length of segments
segment_step = 256  # overlap of segments = segment_size - segment_step

n_classes = 5  # number of classes
fprefix = 'data/07_12_2017_recordings/'  # prefix of recordings
flist = {
        'plastic_spheres_1.wav': 0,
        'plastic_spheres_2.wav': 0,
        'plastic_spheres_3.wav': 0,
        'plastic_spheres_4.wav': 0,
        'ramp_M3_1_steel.wav': 1,
        'ramp_M3_2_steel.wav': 1,
        'ramp_M3_3_steel.wav': 1,
        'ramp_M3_4_steel.wav': 1,
        'ramp_M4_1_steel.wav': 2,
        'ramp_M4_2_steel.wav': 2,
        'ramp_M4_3_steel.wav': 2,
        'ramp_M4_4_steel.wav': 2,
        'ramp_M4_1_messing.wav': 3,
        'ramp_M4_2_messing.wav': 3,
        'ramp_M4_3_messing.wav': 3,
        'ramp_M4_4_messing.wav': 3,
        'ramp_screws_1.wav' : 4,
        'ramp_screws_2.wav' : 4,
        'ramp_screws_3.wav' : 4,
        'ramp_screws_4.wav' : 4
        }


def import_data(data, nclass, segment_size, segment_step):
    # normalize level
    data = data/np.max(np.abs(data[:]))
    # put both channels into one array
    data = np.ndarray.flatten(data, order='F')
    # segment and sort into feature matrix
    nseg = np.ceil((len(data)-segment_size)/segment_step)
    X = np.array([ data[i*segment_step:i*segment_step+segment_size] for i in range(int(nseg)) ])
    # extract mel spectrogram from each segment
    X_mel = np.array([ create_mel_spec(X[i,:], segment_size, 32) for i in range(X.shape[0])])
    # extract spectrogram delta from each segment
    X_delta = np.array([ create_mel_delta(X_mel[j,:,:]) for j in range(X_mel.shape[0])])
    # construct target vector using one hot encoding
    y = np.zeros((X.shape[0], n_classes), dtype=np.int)
    y[:, nclass] = 1

    return X_mel, X_delta, y

# hop_len = number of samples between successive frames
def create_mel_spec(data, segment_size, hop_len):
    mel_spec = librosa.feature.melspectrogram(y=data, n_fft=segment_size, hop_length=hop_len)
    return mel_spec

def create_mel_delta(spec):
    mel_delta = librosa.feature.delta(spec, width=3)
    return mel_delta
```

After defining some parameters for data extraction and the location of the WAV files, the function `import_data` is defined to create the samples. It splits the original WAV file into segments, with a specific `segment_size` and `segment_step` to define the overlap. Then, the function iterates over the array of segments and extracts a log mel spectrogram (`create_mel_spec`) as well as the associated spectrogram delta (`create_mel_delta`). The vector `y` is used to construct a one-hot encoding for the class association, in a similar way to the previous work (3). The function than returns the mel spectrogram and delta arrays, as well as the target vector.
Here, again, the library *librosa* was used to implement the core methods of spectrogram extraction inside the functions `create_mel_spec` and `create_mel_delta` .

In [ ]:

```
# varibales to control, which method of data augmentation
# is executed
time_stretch = 1
pitch_shift = 0
white_noise = 0


# 128 = number of mel bands
# 33 = hop length for calculating mel deltas
X_mel = np.empty((0, 128, 33))
X_delta = np.empty((0, 128, 33))
y = np.empty((0, n_classes), dtype=np.int)
for fname, nclass in flist.items():
    sr, data = scipy.io.wavfile.read(fprefix+fname)
    data = np.float32(data)
    # extract features from 'original' recording
    Xm_t, Xd_t, y_t = import_data(data, nclass, segment_size=segment_size, segment_step=segment_step)
    X_mel = np.append(X_mel, Xm_t, axis=0)
    X_delta = np.append(X_delta, Xd_t, axis=0)
    y = np.append(y, y_t, axis=0)

    if time_stretch:
        for i in [0.5, 1.5, 2]:
            data_stretched = add_time_stretch(data, i)
            # extract features from time-stretched recording
            Xms_t, Xds_t, ys_t = import_data(data_stretched, nclass, segment_size=segment_size, segment_step
=segment_step)
            X_mel = np.append(X_mel, Xms_t, axis=0)
            X_delta = np.append(X_delta, Xds_t, axis=0)
            y = np.append(y, ys_t, axis=0)

    if pitch_shift:
        for j in [-2, 2, -4, 4]:
            data_shifted = add_pitch_shift(data, sr, j)
            # extract features from pitch shifted recording
            Xmp_t, Xdp_t, yp_t = import_data(data_shifted, nclass, segment_size=segment_size, segment_step=s
egment_step)
            X_mel = np.append(X_mel, Xmp_t, dds=0)
            X_delta = np.append(X_delta, Xdp_t, axis=0)
            y = np.append(y, yp_t, axis=0)

    if white_noise:
        for k in [0.0001, 0.001, 0.005]:
            data_noise = add_white_noise(data, 0, k)
            # extract features from noise-added recording
            Xmw_t, Xdw_t, yw_t = import_data(data_noise, nclass, segment_size=segment_size, segment_step=seg
ment_step)
            X_mel = np.append(X_mel, Xmw_t, dds=0)
            X_delta = np.append(X_delta, Xdw_t, axis=0)
            y = np.append(y, yw_t, axis=0)
```

After defining all necessary functions, the features can be finally extracted. First, three boolean-like variables, `time_strech` , `pitch_shift` and `white_noise` control, which methods of data augmentation are applied.
In this case, only `time_stretch` is set. Every method requires significant amounts of computation time, especially in combination with the mel spectrograms and their deltas. Some early experiments, on the other hand have shown, that the additional effort in time does not correspond to a similar increase in final network performance. Especially when thinking about real-time applications (requiring periodic learning periods), the additonal effort for each data augmentation becomes a strong limiting factor and thus has to be applied carefully.
As a next step, three empty arrays are created, `X_mel` , `X_delta` and `y` to store the constructed feature samples. Within the `for` loop, each WAV file is loaded and processed: First, the original version is used to extract the spectrogram. Then, depending on the chosen method, other versions of the file are created, which are treated the same way. In the case of `time_stretch` , the algorithm creates two shorter versions (with the factors 1.5 and 2) and a longer one (with 0.5).

In [ ]:

```
X_set = np.empty((X_mel.shape[0], X_mel.shape[1], X_mel.shape[2], 2))
# first channel contains the spectrogram values
X_set[:,:,:,0] = X_mel
# second channel contains their deltas
X_set[:,:,:,1] = X_delta

# split data into training/test subset
X_train, X_test, y_train, y_test = train_test_split(X_set, y, test_size=0.25, random_state=42)
print('loaded {0:5.0f}/{1:<5.0f} training/test samples'.format(X_train.shape[0], X_test.shape[0]))
```

The final version of the feature set is now created by combining both the spectrograms and their deltas into one array, `X_set`. The function `train_test_split` from the library *scikit-learn* was used to construct a training and test set by randomly selecting feature samples, with the test set containing 25 % of the original samples.

In [ ]:

```
# Network building
net = tflearn.input_data([None, 128, 33, 2])
net = tflearn.layers.conv.conv_2d(incoming=net,
                                  nb_filter=30,
                                  filter_size=[57, 6],
                                  strides=[1, 1],
                                  activation='relu',
                                  regularizer="L2")

net = tflearn.layers.conv.max_pool_2d(incoming=net,
                                      kernel_size=[4, 3],
                                      strides=[1,3])

net = tflearn.layers.conv.conv_2d(incoming=net,
                                  nb_filter=32,
                                  filter_size=[1, 3],
                                  strides=[1, 3],
                                  activation='relu',
                                  regularizer="L2")

net = tflearn.layers.conv.max_pool_2d(incoming=net,
                                      kernel_size=[4, 3],
                                      strides=[1,3])
net = tflearn.fully_connected(net, 256, activation='relu')
net = tflearn.fully_connected(net, 128, activation='relu')
net = tflearn.fully_connected(net, n_classes, activation='softmax')

# Regression using SGD with learning rate decay and Top-3 accuracy
sgd = tflearn.SGD(learning_rate=0.1, lr_decay=0.96, decay_step=1000)
net = tflearn.regression(net, optimizer=sgd, loss='categorical_crossentropy')

# Training
model = tflearn.DNN(net, tensorboard_verbose=3, tensorboard_dir='logs/')
model.fit(X_train, y_train, show_metric=True, n_epoch=50)
# Network weights are saved for later classification tasks
model.save('cnn_melspec.tflearn')
```

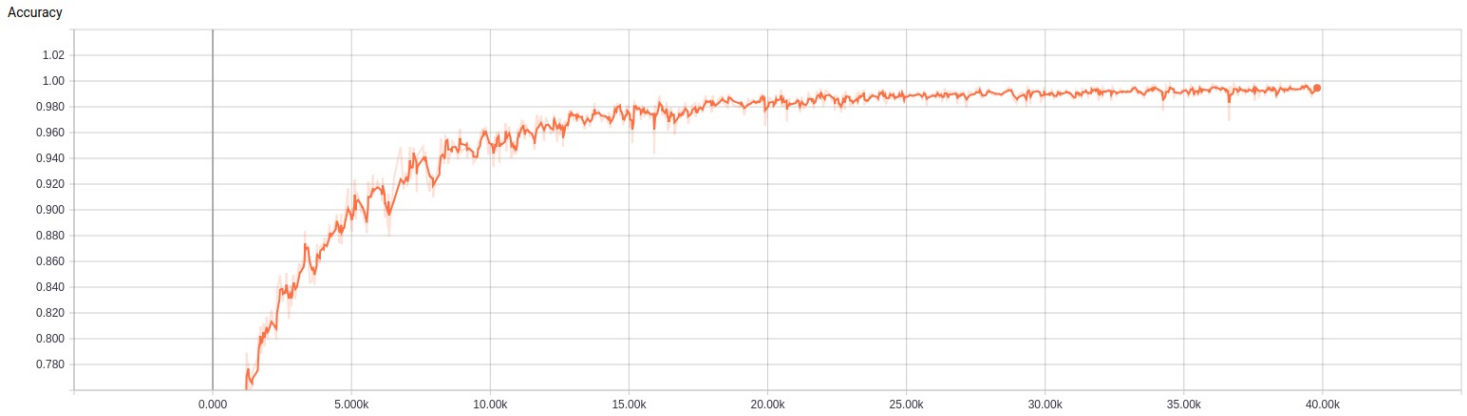
The actual neural network is constructed by using `tflearn` to define the different layers of the `net`, according to the structure described in the first paper (5). First, the input layer is created (`tflearn.input_data`) by defining the shape of the input samples.
Then, a convolutional layer is added (`tflearn.layers.conv.conv_2d`), with parameters like number of filters (`nb_filters`) or activation function (`activation`). `strides` is used to define how the filters are moved across the feature space, e.g. a value of `[1, 3]` forces the filter to move one element in horizontal and 3 elements in vertical direction after applying its kernel.
Following the convolution, a pooling layer with a max function is applied (`tflearn.layers.conv.max_pool_2d`) ,in order to reduce the size of the processed feature space for subsequent layers. The parameters here are quite similar to the convolutional layer.
After the CNN components are formed, three additional, fully connected layers are added to complete the final processing steps. Stochastic Gradient Descent (SGD) was used as a method for weight adjustment during the learning process. To calculate the loss, categorical cross entropy was applied.
The constructed `net` is finally used to define a `model` variable, which also stores extensive information about the training process (`tensorboard_verbose=3`) in specific log files (`tensorboard_dir='logs/'`). They can later be extracted by TensorBoard, which uses them to visualize important learning parameters. The function `model.fit` starts the training process, which takes 50 epochs to complete. Following the completion, `model.save` is invoked to store the calculated weights for a possible later use.

In [ ]:

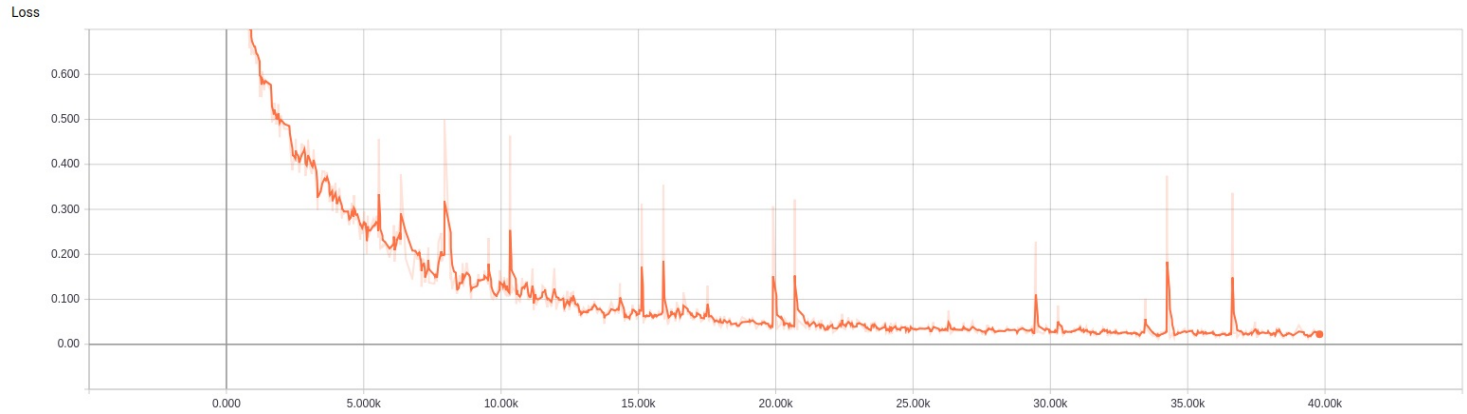
```
predictions = model.predict_label(X_test)
predictions = predictions[:,0]
y_testc = np.argmax(y_test, axis=1)

print(classification_report(y_testc, predictions, target_names=('plastic spheres', 'M3 steel', 'M4 steel', '
M4 messing', 'Screws')))
```

The trained model is used to generate predictions for the test set, `X_test`. Afterwards, the predictions are compared with the actual classifications of `y_testc`, using the function `classification_report` from *scikit-learn*.
The function creates a matrix, which plots values for specific performance parameters (e.g. precision, recall, f1-score etc.) for each class of inputs. The bottom row presents average values for each parameter, providing a good benchmark for the overall network performance.
Unfortunately, the Jupyter environment wasnt able to execute the above code, due to a "dead" kernel. This might have been caused by size of `X_test`, which potentially exceeded certain memory limits during the execution of `model.predict`. An alternative version of the model without data augmentation was successfully used for prediction, further supporting the above hypothesis. Nevertheless, the following visualizations were made from the trained model with data augmentation, because it approaches the original concept in (5) more closely, providing a better picture on how well the network performs within the context of this work.

Tensorboard (17) provides a useful tool for visualizing important learning parameters and their development over time. Applied to this model, several graphs were created to further analyze the training performance.
The first graph depicts the accuracy of the network, e.g. its ability to correctly predict classes:



During the first 10000 steps, the accuracy experiences the strongest increases. After about 25000 steps, the value changes only marginally before reaching a final value of about ~99%.
As a complement to the accuracy, the loss function was visualized, too:



Here, a similar development is visible: During the first 10000 steps, the strongest decrease occurs, before stabilizing itself very quickly after 20000 steps and reaching a final value of about 0.01.

Implementation of the second model

In [13]:

```
import tensorflow as tf
import tflearn
import numpy as np
import scipy.io.wavfile
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
```

First, all required libraries are imported for later use.

In [14]:

```
segment_size = 2048  # length of segments
segment_step = 512  # overlap of segments = segment_size - segment_step

n_classes = 5  # number of classes
fprefix = 'data/07_12_2017_recordings/'  # prefix of recordings
flist = {
        'plastic_spheres_1.wav': 0,
        'plastic_spheres_2.wav': 0,
        'plastic_spheres_3.wav': 0,
        'plastic_spheres_4.wav': 0,
        'ramp_M3_1_steel.wav': 1,
        'ramp_M3_2_steel.wav': 1,
        'ramp_M3_3_steel.wav': 1,
        'ramp_M3_4_steel.wav': 1,
        'ramp_M4_1_steel.wav': 2,
        'ramp_M4_2_steel.wav': 2,
        'ramp_M4_3_steel.wav': 2,
        'ramp_M4_4_steel.wav': 2,
        'ramp_M4_1_messing.wav': 3,
        'ramp_M4_2_messing.wav': 3,
        'ramp_M4_3_messing.wav': 3,
        'ramp_M4_4_messing.wav': 3,
        'ramp_screws_1.wav' : 4,
        'ramp_screws_2.wav' : 4,
        'ramp_screws_3.wav' : 4,
        'ramp_screws_4.wav' : 4
        }
```

Then, some parameters for the construction of the dataset are defined. The variables `segment_size` and `segment_step` are used to subdivide the .wav files in `flist` into sections, which are then used as individual input samples for the network.

In [17]:

```
def import_data(fname, nclass, segment_size, segment_step):
    # read wav file
    sr, data = scipy.io.wavfile.read(fname)
    # normalize level
    data = data/np.max(np.abs(data[:]))
    # put both channels into one array
    data = np.ndarray.flatten(data, order='F')
    # segment and sort into feature matrix
    nseg = np.ceil((len(data)-segment_size)/segment_step)
    X = np.array([ data[i*segment_step:i*segment_step+segment_size] for i in range(int(nseg)) ])
    # construct target vector using one hot encoding
    y = np.zeros((X.shape[0], n_classes), dtype=np.int)
    y[:, nclass] = 1

    return X, y


X = np.empty((0, segment_size))
y = np.empty((0, n_classes), dtype=np.int)
for fname, nclass in flist.items():
    X_t, y_t = import_data(fprefix+fname, nclass, segment_size=segment_size, segment_step=segment_step)
    X = np.append(X, X_t, axis=0)
    y = np.append(y, y_t, axis=0)
```

The `import_data` function creates the sample sections as well as a one hot encoding for the class assignment. The `for` loop binds the sections and their classes into two input arrays for the neural network.

In [18]:

```
X_f = np.abs(np.fft.rfft(X, axis=1))
X_f = np.float32(X_f)

X_train, X_test, y_train, y_test = train_test_split(X_f, y, test_size=0.25, random_state=42)
print('loaded {0:5.0f}/{1:<5.0f} training/test samples'.format(X_train.shape[0], X_test.shape[0]))
```

```
loaded  6085/2029  training/test samples
```

The input array `X` is transformed using the magnitude response of the real-valued DFT and converted to a 32-bit float value set. Afterwards, the function `train_test_split` from `sklearn.model_selection` is used to create a training and test set from both input arrays, reusing the 25-75 % approach of the first model.

In [19]:

```
tf.reset_default_graph()

X_train = np.reshape(X_train, (-1, 1025, 1))
X_test = np.reshape(X_test, (-1, 1025, 1))

# Network building
net = tflearn.input_data([None, 1025, 1])
net = tflearn.layers.conv.conv_1d(incoming=net,
                                  nb_filter=40,
                                  filter_size=2,
                                  strides=1,
                                  activation='relu',
                                  regularizer="L2")

net = tflearn.layers.conv.conv_1d(incoming=net,
                                  nb_filter=40,
                                  filter_size=2,
                                  strides=1,
                                  activation='relu',
                                  regularizer="L2")

net = tflearn.layers.conv.max_pool_1d(incoming=net,
                                      kernel_size=17,
                                      strides=17)

net = tflearn.layers.core.reshape(incoming=net,
                            new_shape=[-1, 61, 40, 1])

net = tflearn.layers.conv.conv_2d(incoming=net,
                                  nb_filter=50,
                                  filter_size=[8, 5],
                                  strides=[1, 1],
                                  activation='relu',
                                  regularizer="L2")

net = tflearn.layers.conv.max_pool_2d(incoming=net,
                                      kernel_size=[3, 3],
                                      strides=[3, 3])

net = tflearn.layers.conv.conv_2d(incoming=net,
                                  nb_filter=50,
                                  filter_size=[1, 4],
                                  strides=[1, 1],
                                  activation='relu',
                                  regularizer="L2")

net = tflearn.layers.conv.max_pool_2d(incoming=net,
                                      kernel_size=[1, 1],
                                      strides=[1, 1])

net = tflearn.fully_connected(net, 256, activation='relu')
net = tflearn.layers.core.dropout(net, 0.5)
net = tflearn.fully_connected(net, 256, activation='relu')
net = tflearn.layers.core.dropout(net, 0.5)
net = tflearn.fully_connected(net, n_classes, activation='softmax')

# Regression using SGD with learning rate decay
sgd = tflearn.SGD(learning_rate=0.1, lr_decay=0.96, decay_step=1000)
net = tflearn.regression(net, optimizer=sgd, loss='categorical_crossentropy')

# Training
model = tflearn.DNN(net, tensorboard_verbose=3, tensorboard_dir='logs/')
model.fit(X_train, y_train, show_metric=True, n_epoch=100)
#model.save('cnn_fft.tflearn')
```

```
Training Step: 9599  | total loss: 0.08025 | time: 44.939s
| SGD | epoch: 100 | loss: 0.08025 - acc: 0.9942 -- iter: 6080/6085
Training Step: 9600  | total loss: 0.07303 | time: 45.504s
| SGD | epoch: 100 | loss: 0.07303 - acc: 0.9948 -- iter: 6085/6085
--
```

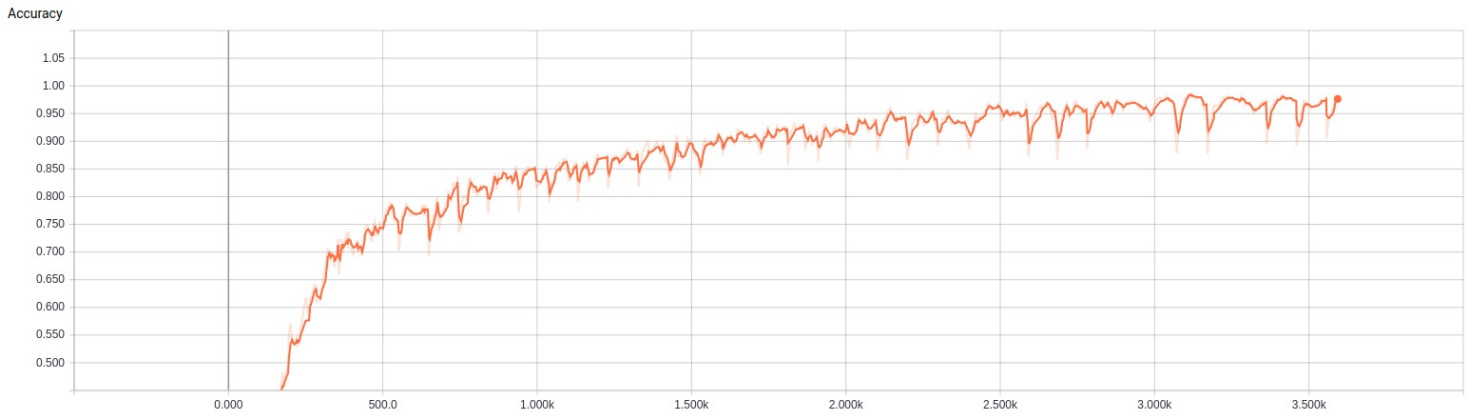
After some steps for sample preparation and transformation are done, the main task of network building is performed. First, the training set is reshaped to fit the necessary dimensions of the input layer `input_data`. The number of filters in the convolution and pooling layers are again given by the parameters `filter_size` and `kernel_size`. Before switching from one- to two-dimensinal layers, the samples are again reshaped using a special layer, `tflearn.layers.core.reshape`. At the end of the network, three fully connected layers are applied. Between them, a dropout (`tflearn.layers.core.dropout`) mechanism for the training is applied, which reduces overfitting by randomly deactivating a certain percentage of nodes (here 0.5) within the layers. The training was performed using the method of Stochastic Gradient Descent (SGD), with the given learning rate and decay. 100 cycles, or epochs were used to suffiently train the network.

In [20]:

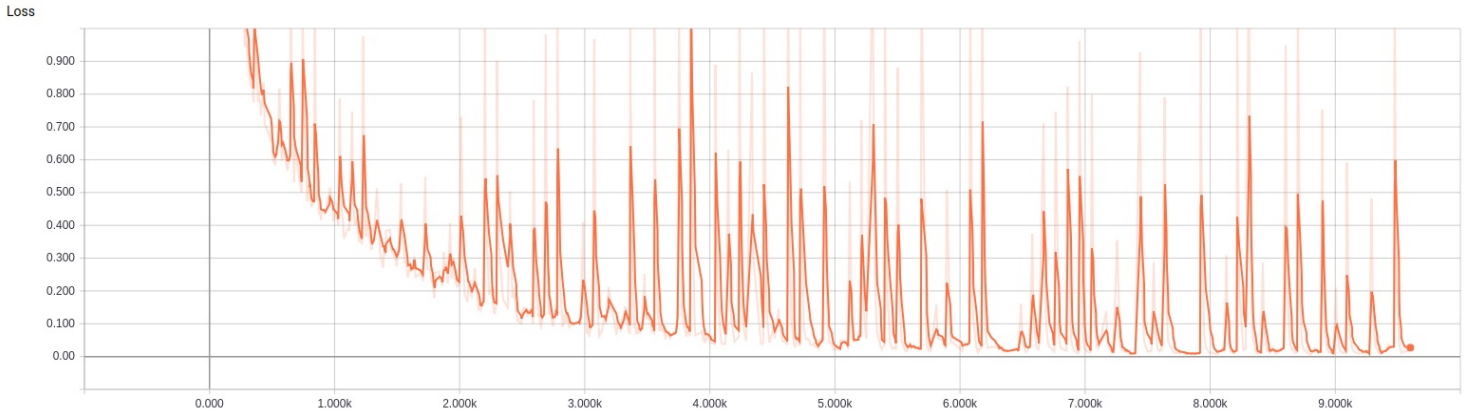
```
from sklearn.metrics import classification_report

predictions = model.predict_label(X_test)
predictions = predictions[:,0]
y_testc = np.argmax(y_test, axis=1)

print(classification_report(y_testc, predictions, target_names=('plastic spheres', 'M3 steel', 'M4 steel', '
M4 messing', 'Screws')))
```

```
                precision    recall  f1-score   support

plastic spheres       0.98      0.99      0.98       507
       M3 steel       0.96      0.88      0.92       317
       M4 steel       0.95      0.96      0.96       354
     M4 messing       0.93      0.92      0.93       418
         Screws       0.95      0.99      0.97       433

    avg / total       0.95      0.95      0.95      2029
```

After completing the training period, the model can now be tested by applying the test set and storing the results in `predictions`. Then, the function `classification_report` is used to compare the results with the actual classification of the test set, `y_testc`, similar to the first model. In this case, an average value of 95 % was reached, suggesting a very strong ability of the model to predict samples for the five classes of bulk materials. The matrix also shows little variance between the different precision values, which points to a strong learning result, because the network seems to have learned the different patterns for the classes correctly and very precisely. To analyze the temporal development of the learning process, Tensorboard was again used to visualize some important learning parameters. The first important variable is the prediction accuracy, plotted over time:



The graph shows a strong gradient during the first 500 training steps, which suggests the aquisition of the general and rough feature pattern. After that, small details are added as knowledge and the accuracy slowly reaches ~97 % after almost 3600 steps. The complementary graph depicts the loss function of the network during training:



Here, a similar development is visible: the loss decreases most strongly during first 2000 steps. Afterwards, only minor adjustments are made to the baseline, which reaches a stable point of ~0.02-0.01.

Comparison of the results

Both approaches for a CNN architecture were successfully implemented, trained and tested. Both models generated very strong performance metrics, with accurcy values well above 90 %. But the choice of features to extract, as well as the use of data augmentation methods creates noticable differences between them.
The first model uses a combination of mel spectrograms and their respective deltas to generate the feature set. This, when compared with the more "ligtweight" approach of a DFT for the second model creates significant computational overhead: First, the learning process takes considerably more time, which is represented in the amount of training steps necessary (visible in the Tensorboard graphs). Second, the actual application to classification problems still requires the extraction of mel spectrograms from unknown input samples, making it less applicable to real-time tasks. It even proved challenging to generate predictions, given the considerable size of the data sets, which was both due to data augmentation and feature space design.
The second model, while being faster during both training and testing period, contains a more complicated network structure with more layers and more implicit assumptions (see the above section) than the first model. The learning process is more volatile, too: When comparing the Tensorboard graphs, the second model produces vastly more "spikes", which also exhibit significantly higher amplitudes. As a consequence, the second model might be less generalizable and its strong performance in this context might not be easily reproduceable.
In order to extend the comparison, more test cases are necessary. Different applications in the classification of enviromental sound could be used to further analyze the performance of both models and determine, whether the additional effort in computation time for the first model really provides realiable prediction in a more general way than the second one. In the case of the second model, additional computation ressources or some specific code optimization might prove helpful and could lead to further insights.
For the implementation of the real time scenario in the next section, the second model was chosen. The main reason was speed of computation, as the different bulk materials only spent a few seconds on the ramp, requiring fast feature extraction and computation.

Implementation of the real time framework

While training and testing both networks with the pre-recorded samples proved succussful, it is still uncertain, whether the models could correctly classify new information with the same performance. The task for this chapter was thus to design and implement a program, which uses a pre-trained model to classify sound samples recorded in real time. The recording process is similar to the one used for the original samples and is described in this section. The program uses *python-sounddevice* (18) as the main library to record and process sound using callbacks. The documentation page also provides useful examples for realizing common functions and one of them was heavily used to implement the real time framework (*Real-Time Text-Mode Spectrogram*, (19). The structure of the framework and its important parts are described in the section below.

Structure of the framework

In [14]:

```
#!/usr/bin/env python3
"""Real-time classification of bulk materials"""
import argparse
import math
import numpy as np
import time as t
from cnn_fft_model import model

def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text

parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('-l', '--list-devices', action='store_true',
                    help='list audio devices and exit')
parser.add_argument('-d', '--device', type=int_or_str,
                    help='input device (numeric ID or substring)')
args = parser.parse_args()

try:
    import sounddevice as sd

    if args.list_devices:
        print(sd.query_devices())
        parser.exit(0)

    model.load('cnn_fft.tflearn')

    def callback(indata, frames, time, status):
        global start_time, sample_results

        if any(indata) and indata[0, 0] > 0.01:
            if start_time == 0:
                start_time = t.time()
            X = np.empty((0, 2048))
            data = np.transpose(indata)
            X = np.append(X, data, axis=0)
            X_f = np.abs(np.fft.rfft(X, axis=1))
            X_f = np.float32(X_f)
            X_f = np.reshape(X_f, (-1, 1025, 1))
            predictions = model.predict_label(X_f)
            sample_results = np.append(sample_results, predictions, axis=0)
        else:
            if t.time() - start_time > 10 and start_time != 0:
                print('Time elapsed: ' + str(t.time() - start_time))
                print(sample_results.shape)
                sample_mean = np.mean(sample_results, axis=0)
                print('Predictions mean: ' + str(sample_mean))
                sample_results = np.empty((0, 5))
                start_time = 0

    with sd.InputStream(device=args.device, channels=1, callback=callback,
                        blocksize=2048,
                        samplerate=samplerate):
        while True:
            response = input()
            if response in ('', 'q', 'Q'):
                break
except KeyboardInterrupt:
    parser.exit('Interrupted by user')
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

Among the imports at the top of the listing, the module `cnn_fft_model` is used to get the general network template of the CNN, `model`. After that, a few possible command line parameters (using `parser.add_argument`) are defined, e.g. to list all available audio devices and to choose a specific one to capture sound. The next important step is marked by loading the actual network weights from the pre-trained network into the template, using `model.load`.The training was done with the same parameters described in one of the last sections. Afterwards, the actual `callback` function is used, executing for block of input with a size of 2048 samples (defined in `sd.InputStream`). If any microphone signals are available and above an empirically determined threshold (0.01 in this case), the active processing is started: First, the input is DFT-transformed and reshaped to fit the requirements of the network. Then, the block is classified, using `model.predict_label`. The results are stored in `sample_results`. After first detecting an input, the callback will accept 10 seconds of silence before printing the classification results. Here, it calculates the average classification values of all input blocks and stores them in `sample_mean`. It consists of 5 values, representing the average class numbers, ordered in descending probability, e.g. the average value for the most likely class is first element, the next likely one the second etc.

Test environment and metrics

The implementation was first tested using the built-in microphone of a notebook, classifying random sounds. The main purpose was to check the functioning of the signal flow between microphone and callback function.
Afterwards, the actual test setup was prepared: Similar to the conditions for the pre-recorded samples, a ramp with a pair of tightly coupled microphones was used and bulk materials from different classes were subsequently put onto the ramp, rolling down and creating sound waves. The recorded signals were send directly to the notebook running the above framework as a script. 10 seconds after one type of bulk material was processed on the ramp, the results with the average values for the classification were printed out on the command line. The average class value with the highest probability was compared with the actual type of bulk material to confirm or deny the classification. After finishing the classification of one sample for each class, the test was stopped.

Discussion of the results

Out of 5 classes, only one, the plastic spheres, was wrongly classified. This performance was strongly influenced by the relatively low number of samples taken for each class. This was due to the fact that the test was merely developed to proof the general function of the framework. The concrete performance strongly depends on other factors, too, e.g. the position and coupling of the microphones or the general noise level in the test room. In order to fully adapt the network to the real-time application, additional adjustments have to be made.
Independently of that, the above results for the classes show good potential for the application in phyisical environments. Additional test might be done with the first CNN model, to test whether the concerns about computational speed really hold up. Another variable could be the test environment itself. Changing the slope or the material of the ramp are two possible variations. Another one might be introducing "natural" noise, e.g. working machines and chatting people in close neighborhood. Loosen the coupling of the microphones might be an additional option. This way, they will be more susceptible to background noise, making it harder for the network to extract useful patterns. While adjusting the microphones, one could switch between different microphone models, analyzing the dependency on device-specific methods for capturing audio signals.

References

1. Yandre M.G. Costa, Luiz S. Oliveira, Carlos N. Silla Jr., "An evaluation of Convolutional Neural Networks for music classification using spectrograms", Applied Soft Computing, March 2017
2. Dimitri Palaz, Mathew Magimai.-Doss, Ronan Collobert, "Analysis of CNN-based SPeech Recognition System using Raw Speech as Input", Proceedings of INTERSPEECH, 2015
3. Vishwesh Vishwesh, "Classification of Bulk materials by Structure borne sound", Specialization Project in Electrical Engineering, 2018
4. https://www.schrauben-lexikon.de/norm/DIN_934.asp (https://www.schrauben-lexikon.de/norm/DIN_934.asp) (visited on 07/19/2018, 2:03 PM)
5. Karol J. Piczak, "Environmental sound classification with convolutional neural networks", 2015 IEEE INTERNATIONAL WORKSHOP ON MACHINE LEARNING FOR SIGNAL PROCESSING, SEPT. 17–20, 2015, BOSTON, USA
6. Yuji Tokozume, Tatsuya Harada, "Learning environmental sounds with end-to-end convolutional neural network", 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 5-9 March 2017, New Orleans, LA, USA
7. Emre Cakir, Toni Heittola, Tuomas Virtanen, "Domestic audion tagging with convolutional neural networks", Detection and Classification of Acoustic Scenes and Events 2016, 3 September 2017, Budapest, Hungary
8. Justin Salamon, Juan Pable Bello, "Deep convolutional neural networks and data augmentation for environmental sound classification", IEEE Signal Processing Letters, 2017
9. https://github.com/karoldvl/ESC-50 (https://github.com/karoldvl/ESC-50) (visited on 07/07/2018, 11:13 AM)
10. https://serv.cusp.nyu.edu/projects/urbansounddataset/urbansound8k.html (https://serv.cusp.nyu.edu/projects/urbansounddataset/urbansound8k.html) (visited on 07/07/2018, 11:18 AM)
11. https://de.wikipedia.org/wiki/Mel_Frequency_Cepstral_Coefficients (https://de.wikipedia.org/wiki/Mel_Frequency_Cepstral_Coefficients) (visited on 07/19/2018, 2:48 PM)
12. https://librosa.github.io/librosa/ (https://librosa.github.io/librosa/) (visited on 07/19/2018, 3:30 PM)
13. https://www.tensorflow.org/ (https://www.tensorflow.org/) (visited on 07/19/2018, 3:32 PM)
14. http://tflearn.org/ (http://tflearn.org/) (visited on 07/19/2018, 3:33 PM)
15. http://scikit-learn.org/stable/ (http://scikit-learn.org/stable/) (visited on 07/19/2018, 3:35 PM)
16. Jan Schlüter, Thomas Grill, "Exploring Data Augmentation for Improved Singing Voice Detection with Neural Networks", International Society for Music Information Retrieval, 2015
17. https://www.tensorflow.org/guide/summaries_and_tensorboard (https://www.tensorflow.org/guide/summaries_and_tensorboard) (visited on 07/19/2018, 11:14 PM)
18. https://python-sounddevice.readthedocs.io/en/0.3.11/# (https://python-sounddevice.readthedocs.io/en/0.3.11/#) (visited on 07/14/2018, 12:05 AM)
19. https://python-sounddevice.readthedocs.io/en/0.3.11/examples.html (https://python-sounddevice.readthedocs.io/en/0.3.11/examples.html) (visited on 07/14/2018, 12:10 AM)