
A Technical Report For "A Novel Deep Learning Network via Multi-Scale Inner Product with Locally Connected Feature Extraction for Intelligent Fault Detection"

Liu Jialin

Department of Computer Science
the University of Hong Kong
lplus0@connect.hku.hk

Liang Xinwen

Department of Computer Science
the University of Hong Kong
liangxw@connect.hku.hk

Abstract

1 Intelligent fault detection is an important application of artificial intelligence and
2 has been widely used in many mechanical systems. Shipborne antenna which is
3 a typical and important mechanical system plays an irreplaceable role in ships.
4 Considering the tough working environment and heavy background noise, fault
5 detection is difficult for shipborne antenna. The paper we selected presents an intel-
6 ligent fault detection method via multi-scale inner product with locally connected
7 feature extraction for shipborne antenna fault detection. Inspired by inner product
8 principle, the paper takes advantage of inner product to capture fault information
9 in the vibration signals and detect the faults in rolling bearing of shipborne an-
10 tenna. Meanwhile, multi-scale analysis is employed in two layers of the network
11 to improve the feature extraction ability. The local features under different scales
12 are collected and used for fault classification. Finally, results verified by CWRU
13 dataset show that the proposed method can learn sensitive features directly from
14 raw vibration signals and detect the faults in rolling bearing of shipborne antenna
15 effectively

1 Algorithm Overview

1.1 SVM Based Methods

The SVM-based methods presented in this paper aim to diagnose faults in rolling bearings using vibration signals and time domain features. The approach involves two feature sets, FS1 and FS2, extracted from the vibration signals and classified using SVM.

FS1 consists of 10 commonly used feature indexes, including mean value, standard deviation, square root amplitude, absolute mean value, skewness, kurtosis, variance, maximum value, minimum value, and peak-to-peak value. SVM is used to classify the faults based on these features.

FS2 includes 16 features calculated from the vibration signals, including waveform index, pulse index, peak index, margin index, skewness index, and kurtosis index. SVM is also applied to classify faults using these features.

The SVM algorithm is used as a binary classifier to determine whether a rolling bearing is faulty or not. The algorithm works by constructing a hyperplane that maximally separates the data points into two classes. The SVM classifier is trained using a set of labeled data points and then used to classify new data points.

1.2 CNN based method

In order to investigate the superiority of the proposed method further, a one-dimensional convolutional neural network is constructed for fault detection.

The CNN model consists of a convolutional layer, a pooling layer, a fully-connected layer, a flatten layer and a softmax layer. Similarly, there are 8 convolution kernels in convolutional layer to compare with the proposed method. What's more, the model also takes raw vibration signals as input

The convolutional layer can divide the dataset into many samples without redundancy to make sure the features are orthogonal. The main purpose is to reduce calculation and improve efficiency. After the preprocessing, the inner product with different kernels will be calculated.

The pooling layer is employed as the fourth layer in order to reduce dimension and extract deep features. Similarly, the obtained features are also divided into different batches in pooling layer for multi-scale analysis. After pooling, the local features have been collected under different scales from the raw vibration signals.

The flatten layer consists of many normalized features, which is normalized to prevent neurons from saturating and to improve generalization coming from the former layer.

The fully-connected layer is used for nonlinear feature transformation.

Softmax regression is the generation of logistic regression for the purpose of handling classification problems with more than two classes. With the given working conditions, the network is capable of diagnosing the different faults in the rolling bearings.

2 Implementation

There are three steps in the whole paper reproduction, which are data preparation, model construction and model training.

2.1 Data preparation

The dataset from the Case Western Reserve University (CWRU) which is a widely used dataset for rolling bearing fault diagnosis will be used to test the effectiveness of the proposed method. The experiment rig consists of a 2 hp motor, a torque transducer, a dynamometer and control electronics. Single point faults were manufactured with fault diameter of 7 mils, 14 mils, 28 mils and 40 mils. The vibration signals are collected by accelerometers while the loads are set to 0 to 3 horsepower. More information about the CWRU dataset can be found on the website below.

<https://engineering.case.edu/bearingdatacenter/download-data-file>

After collecting a lot of raw data, we need to integrate them and form a complete data set, which is a mat file format named all_data_DriveEnd.mat.

2.2 Model construction

```
def MSC_1DCNN(data_shape=(8192,1), class_number = 10):
    # (CNN) model
    input_signal = Input(shape=(8192, 1))
    # Multiple parallel one-dimensional convolution layers
    # Inner product layer and pooling layer
    x1 = Conv1D(filters=1, kernel_size=2, padding='valid', strides=2)(input_signal)
    x1 = MaxPooling1D(pool_size=256)(x1)
    x2 = Conv1D(filters=1, kernel_size=4, padding='valid', strides=4)(input_signal)
    x2 = MaxPooling1D(pool_size=128)(x2)
    x3 = Conv1D(filters=1, kernel_size=8, padding='valid', strides=8)(input_signal)
    x3 = MaxPooling1D(pool_size=64)(x3)
    x4 = Conv1D(filters=1, kernel_size=16, padding='valid', strides=16)(input_signal)
    x4 = MaxPooling1D(pool_size=32)(x4)
    x5 = Conv1D(filters=1, kernel_size=32, padding='valid', strides=32)(input_signal)
    x5 = MaxPooling1D(pool_size=16)(x5)
    x6 = Conv1D(filters=1, kernel_size=64, padding='valid', strides=64)(input_signal)
    x6 = MaxPooling1D(pool_size=8)(x6)
    x7 = Conv1D(filters=1, kernel_size=128, padding='valid', strides=128)(input_signal)
    x7 = MaxPooling1D(pool_size=4)(x7)
    x8 = Conv1D(filters=1, kernel_size=256, padding='valid', strides=256)(input_signal)
    x8 = MaxPooling1D(pool_size=2)(x8)
```

Figure 1: First three layers

The code above defines a one-dimensional convolutional neural network (CNN) model. The construction layer, the inner product layer and the pooling layer is here.

Using shape for (8192, 1) of the input signal, and apply with different kernel size and pace of multiple parallel one-dimensional convolution layer. In order to capture more fault features, the inner product layer has eight kernels with size of 2, 4, 8, 16, 32, 64, 128 and 256 respectively. Meanwhile, the pooling layer also has eight kernels with size of 256, 128, 64, 32, 16, 8, 4, 2 so that the outputs under different scale share the same size.

```
# fully-connected layer
xx = concatenate([x1, x2, x3, x4, x5, x6, x7, x8], axis=-2)
# flatten layer
xx = Flatten()(xx)
xx = Dense(128, activation='relu')(xx)
# softmax layer
output = Dense(4, activation='softmax')(xx)

# Contains 4 units and a dense layer activated by softmax
# which represents the output probability for each category
model = Model(inputs=input_signal, outputs=output)

return model
```

Figure 2: Last three floors

The construction process of the flatten layer, the fully-connected layer and the softmax layer is here. The outputs of these layers are connected along the last axis. The connected output is flattened and passed through a dense layer with 128 units and ReLU activations. Dense layer with 10 units and softmax activation, which represents the output probability of each class.

2.3 Model training

```
def train_model_cwru():
    my_model = MSC_1DCNN(datanshape=(4096,1), class_number_=10)

    datafile_path = 'all_data_DriveEnd'
    data = sio.loadmat(datafile_path)
    category_list = ['NM', 'BA_007', 'BA_014', 'BA_021', 'OR_007', 'OR_014', 'OR_021', 'IR_007', 'IR_014', 'IR_021']
    X_train = []
    y_train = []
    X_test1 = []
    y_test1 = []
    sample_length = 4096

    sample_num_each_signal = 100
    # in training data or testing data, each full signal actually generates 200 samples.
    start_idx = np.int64(np.round(np.linspace(0, 60000-sample_length, sample_num_each_signal)))
    # start_idx = list(range(0, 60000-sample_length+1, round(60000/sample_num_each_signal)))
    for i in range(10):
        this_ctgr = category_list[i]
        for j in range(4):
            key_name = this_ctgr + '_' + str(j)
            this_ctgr_data_j = data[key_name]

            [X_train.append(this_ctgr_data_j[k:k+sample_length]) for k in start_idx]
            [y_train.append(i) for k in start_idx]
            # y_train.append(i)
            [X_test1.append(this_ctgr_data_j[k+60001:k+60001+sample_length]) for k in start_idx]
            [y_test1.append(i) for k in start_idx]
```

Figure 3: model training code1

The function starts by creating an instance of the MSC_1DCNN model, which is a custom model designed for 1D convolutional neural networks. The model takes input data of shape (4096, 1) and has 10 output classes. And the CWRU dataset is loaded from the file 'all_data_DriveEnd' using the *scipy.io.loadmat()* function. The dataset consists of several categories of faults, represented by normal condition (NM), inner race fault (IR), outer race fault (OR) and ball fault data (BA).

The function then proceeds to extract training and testing samples from the dataset. For each category and each sample index, it retrieves a segment of length 4096 from the data and appends it to the training set. The corresponding label (category index) is also appended to the training labels. Similarly, segments from the test data are extracted and labeled.

```

X_valid = np.array([normalization_processing(data) for data in X_valid])
X_valid = np.expand_dims(X_valid,axis=2)
y_valid_one_hot = to_categorical(np.array(y_valid), len(category_list))

# configure training
opt = tf.keras.optimizers.Adam(lr=0.01)
my_model.compile(optimizer=opt, loss=tf.keras.losses.CategoricalCrossentropy(),
                 metrics=['accuracy'])

history = my_model.fit(
    x=X_train,
    y=y_train_one_hot,
    batch_size=32,
    epochs=50,
    validation_data=(X_valid, y_valid_one_hot),
    shuffle=True,
    verbose=0,
    # callbacks=[reduce_lr]
)

y_pred = my_model.predict(X_test) #shape: 22313x7 #y_test:22313x1
y_pred = np.argmax(y_pred, axis=1)
acc = accuracy_score(y_test, y_pred)

```

Figure 4: model training code2

The training and testing datasets are converted into numpy arrays and further preprocessed. A normalization processing function is applied to each data sample to normalize the values. The dimensions of the input data are also expanded to include a third dimension. The training set labels are one-hot encoded using the `to_categorical()` function from the `keras.utils` module. The validation set is created by splitting the test set into validation and final testing sets. The split is performed using the `train_test_split()` function from the `sklearn.model_selection` module.

The model is compiled with the Adam optimizer, a learning rate of 0.01, and the categorical cross-entropy loss function. The metrics to track during training are set to accuracy. The model is trained using the `fit()` method of the model object. The training data, labels, batch size, number of epochs, validation data, and shuffle parameter are passed to the `fit()` method. The verbose parameter is set to 0 to suppress training progress output. After training, the model's predictions are obtained on the test set using the `predict()` method. The predictions are converted from one-hot encoded format to class labels by taking the index of the maximum value for each prediction. The accuracy of the model is calculated by comparing the predicted labels with the true labels using the `accuracy_score()` function from the `sklearn.metrics` module. Finally, the accuracy of the model on the test set is returned as the output of the function.

3 Challenges

3.1 Data preparation

Due to the limited access to resources on the Internet, we only found the first of the three data sets in the paper.

```

import scipy.io as sio
variable_names = sio.whosmat('/content/drive/MyDrive/Colab Notebooks/all_data_DriveEnd.mat')
for variable in variable_names:
    print(variable)

('BA_007_0', (122571, 1), 'double')
('BA_007_1', (121410, 1), 'double')
('BA_007_2', (121556, 1), 'double')
('BA_007_3', (121556, 1), 'double')
('BA_014_0', (121846, 1), 'double')
('BA_014_1', (122136, 1), 'double')

```

Figure 5: Processed dataset

We download more than ten data files and integrate them to form a data set. The part of the content of the dataset is right up here. The left dataset SQ Dataset and Shipborne Antenna Dataset require real machine experiments, and they are not available with a certain confidential nature.

3.2 Model construction

The most challenging part of writing this part code may be understanding and implementing the Conv1D and MaxPooling1D layers effectively. These layers are commonly used in deep learning

models for processing sequential data such as time series and audio signals. As the paper mentioned, the inner product is a key operation for feature extraction in this paper and the kernel size has a great influence on the diagnosis ability. In order to capture more fault features, the inner product layer has eight kernels with size of 2, 4, 8, 16, 32, 64, 128 and 256 respectively. Meanwhile, the pooling layer also has eight kernels with size of 256, 128, 64, 32, 16, 8, 4, 2 so that the outputs under different scale share the same size. The choice of kernel size, padding, and strides can significantly impact the performance of the model.

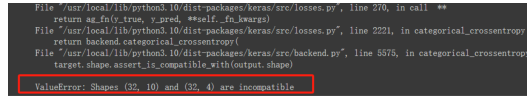
Additionally, the concatenation of the output from multiple Conv1D and MaxPooling1D layers can be challenging to understand and implement correctly. The choice of 128 units in the Dense layer following the Flatten layer is substantiated and can be adjusted based on the specific requirements of the problem and the complexity of the input data. In this paper, we have 8 different pooling layers and 8 different inner product layers. If we choose 64 as the number of units, it can be a little bit inappropriate and limited for training our model.

In this code, the purpose of the Dense layer with 128 units is to further process and learn from the flattened representation of the concatenated outputs from the previous layers. The choice of 128 units is a hyperparameter that can be tuned to optimize the model's performance. Increasing the number of units in the Dense layer can potentially allow the model to capture more complex patterns and relationships in the data.

However, this also increases the model's complexity and may require more training data and computational resources. On the other hand, reducing the number of units can simplify the model and reduce the risk of overfitting, but it may also limit the model's ability to learn complex patterns.

Finally, selecting the appropriate number of dense layers and neurons can also be challenging, as it can greatly affect the model's ability to generalize and avoid overfitting.

3.3 Model training



```
File ~/usr/local/lib/python3.10/dist-packages/keras/src/losses.py, line 570, in call **
return ag_fn(y_true, y_pred, **self._fn_kwargs)
File ~/usr/local/lib/python3.10/dist-packages/keras/src/losses.py, line 2221, in categorical_crossentropy
return backend.categorical_crossentropy(
File ~/usr/local/lib/python3.10/dist-packages/keras/src/backend.py, line 5575, in categorical_crossentropy
target_shape.assert_is_compatible_with(output_shape)
ValueError: Shapes (32, 10) and (32, 4) are incompatible
```

Figure 6: incompatible error

In the process of model training, we can find that console has the above error:

ValueError: Shapes (32, 10) and (32, 4) are incompatible. We feel confused because we use the four classification model because the first experiment in the paper was just four classification. After our research we find it is ten classification model that more suitable for the experiment. We collect 10 types of data, including 1 normal condition (NM), 3 inner race fault (IR), 3 outer race fault (OR) and 3 ball fault data (BA). The fault diameter for these three data is 0.007", 0.014" and 0.021". So it is more reasonable to use the ten classification model. After we adjust the MSC_1DCNN model argument from `class_number = 4` to `class_number = 10`, the model can compile normally and output the expected values.

4 Result

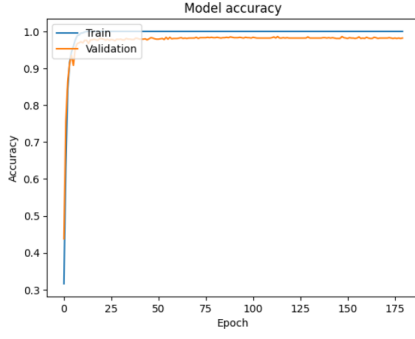
4.1 Visualization of Model loss and accuracy

In this paper, 400 samples were used for fault classification of which 75% are training samples and 25% are testing samples. And each sample is a collected vibration signal segment with 8192 points.

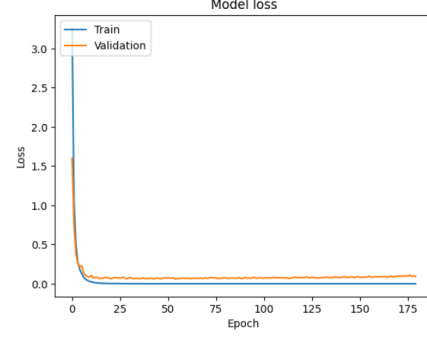
The training samples were put into the proposed network for training through 300 loops and achieved an average classification accuracy of 99.1% for both training samples and testing samples. There is a high degree of reproduction for the paper we have chosen.

4.2 Confusion matrix of classification result

The confusion matrix shows the classifier's performance on each category, and can help us determine which categories are easily confused. It can be seen that the faults in the bearings are heavy and the



(a) Model accuracy



(b) Model loss

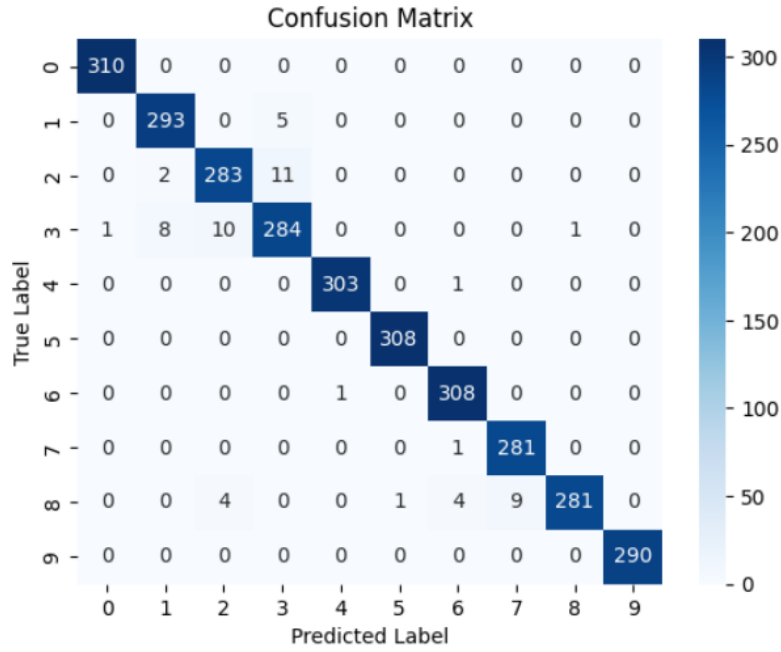


Figure 8: Confusion matrix

classification task is relatively easy. There is a high degree of reproduction for the paper we have chosen.

5 Conclusion

In this paper, a multi-scale analysis method with locally connected feature extraction is proposed for rolling bearing fault diagnosis of shipborne antenna. In this method, inner product is applied for feature extraction and kernels with different sizes are selected to provide better performance. Three datasets respectively from CWRU, SQ and shipborne antenna are used to test the proposed method. The results show that the proposed method is capable of extracting sensitive features directly from raw vibration signals and achieves a high classification accuracy.

The code is available on github.

https://github.com/Lplus0/COMP7404_1A

The dataset is over 25MB so we put it on a release version.

https://github.com/Lplus0/COMP7404_1A/releases/tag/dataset