NXP Semiconductors
User's Guide

Document Number: IMXRTADUG
Rev. 0, 01/2020

Anomaly Detection

i.MX RT application using TensorFlow Lite

1. Introduction

Anomaly detection is a method used to identify anomalous behavior (the data points that significantly differ from the majority of data points). Anomalies are often associated with some kind of problem, failure, or rare event (financial fraud, sensor failure, health issue, spam detection) which makes the problem very interesting from a business perspective.

This application shows how to use a neural network to search for anomalies in sensors that are in a stationary position. This approach can be applied to more complex problems, such as detecting unseen falls [1], anomalies in machine sound [2], anomalous movements [3], or errors in manufacturing systems [4]. The search for anomalous behavior in sensor data is an unsupervised learning one-class problem where you know only one class and try to search for anomalies outside of this class. A variety of both classical and deep machine learning algorithms can be applied for this problem. The One-Class Support Vector Machine (OC-SVM) and the Gaussian Mixture Model (GMM) are common machine-learning algorithms that are successfully applied to anomaly-detection problems. However, their performance on complex, multi-dimensional data is sub-optimal.

Contents

1. Introduction 1
2. Requirements 2
   2.1. Hardware requirements 2
   2.2. Software requirements 2
   2.3. Installation 2
3. Project structure 4
4. Model training 4
   4.1. Data collection 5
   4.2. Pre-processing 6
   4.3. Neural network model 6
   4.4. Model training 7
   4.5. Evaluation 7
   4.6. Deployment 7
5. Inference 8
6. Summary 9
7. References 9
Appendix A. List of python libraries required for network training 10

The deep learning approach is very effective for identifying anomalies in data [5] and shows a better performance on more complex and noisy data than common machine-learning techniques like SVM [2]. Autoencoders are very useful in detecting anomalies. An autoencoder is a feed-forward neural network that learns to predict its input. The input is reduced to core features (encoding) and then recreated back (decoding). The error between the actual input and computed output is called a reconstruction error. The data that is never seen by the network (anomalies) is expected to show a higher reconstruction error. Thresholding on reconstruction errors is used to detect anomalies.

2. Requirements

2.1. Hardware requirements

- EVKB-IMXRT1050 or EVK-IMXRT1060
- FRDM-STBC-AGM01 sensor shield

2.2. Software requirements

- MCUXpresso IDE v11.1.0 or later (IAR and Keil are also supported)
- Python 3 with the required packages listed in Appendix A (for training)
  - Create a new anaconda environment (this is optional)

```
>> conda create -n adt pip
>> activate adt
```

  - Install the required Python libraries listed in Appendix A. If you copy the list to a text file and save it as *requirements.txt*, you can install them using this command:

```
>> pip install -r requirements.txt
```

2.3. Installation

The anomaly detection application is a part of the eIQ TensorFlow Lite 1.14 library. When using the NXP MCUXpresso SDK Builder, select either **EVKB-IMXRT1050-AGM01** or **EVK-IMXRT1060 with AGM01** and then the eIQ component. When the library is loaded in the MCUXpresso IDE, import the *tensorflow_lite_adt* example. It is recommended to select “UART” in “Project Options” (Figure 1).

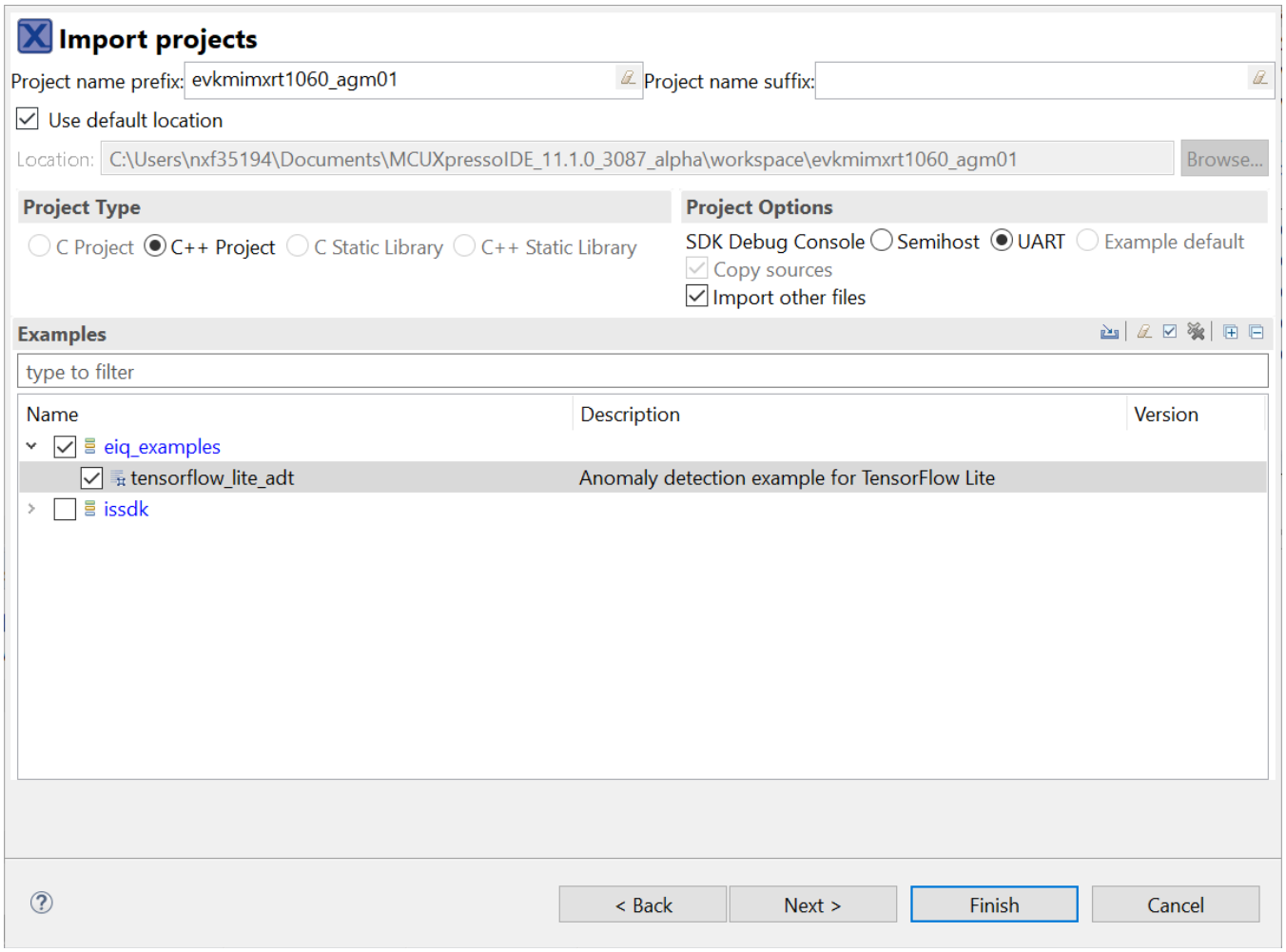

Figure 1. Selecting anomaly detection project from imported SDK examples in MCUXpresso IDE

3. Project structure

The project has two parts:

a) C/C++ source code that generates sensor data for training and runs the inference (searches for anomalies in sensor readings)

b) Python script that is used for model training

```
(sdk_path)\middleware\eiq\tensorflow-lite\examples\adt
|   adt.cpp - main application file
|   adt_model.h - hexdump of adt_train/adt_model.tflite generated by adt_train.py
|   get_sensor_data.c - sensor functions (initialization and reading)
|   get_sensor_data.h - header for sensor functions
|   parameters.h - parameters generated by adt_train/adt_train.py script
|
+---adt_train
|       adt_model.tflite - trained TF-Lite model generated by adt_train.py
|       adt_train.py - Python training script
```

When the example is loaded to MCUXpresso, files are loaded from the *adt_train* folder.

In MCUXpresso IDE, the source code files are in the source folder and the *adt_model.tflite* and *adt_train.py* files are in the *doc* folder.

4. Model training

The project includes the entire machine learning workflow (shown in Figure 2), which enables you to train your own model and experiment with different settings (change the sensors, input data, or model size).

The entire machine learning workflow consists of several steps (shown in Figure 2). In the first step, sensor data are collected. You can collect the training and testing data from all available sensors and decide which sensors to use during the training phase. Pre-processed training data is used to train a predefined model. If the testing data are available, a visual evaluation can be done (see Section 4.5, "Evaluation"). You can experiment with model parameters and train the model multiple times with different parameters until you are satisfied with the result. In the end, the final model is transferred to the board. All the steps are described in detail in Sections 4.1 - 4.6.

Except for data collection, the workflow is performed by the Python training script *adt.py* located in the *adt_train* folder of the *tensorflow_lite_adt* example.

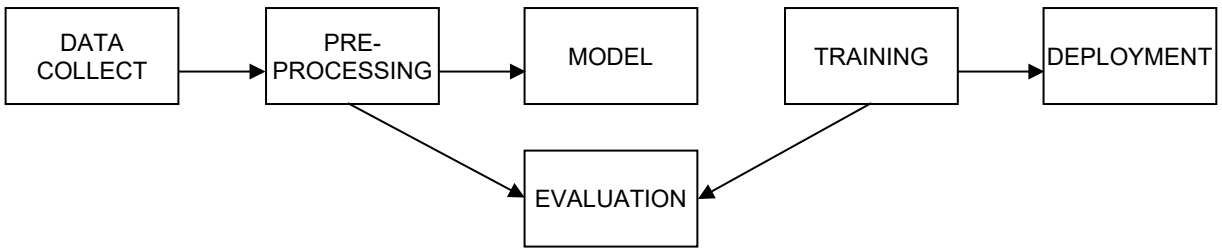

Figure 2. Machine learning workflow

4.1. Data collection

Data collection is a part of the C/C++ application that prints out raw sensor data to the standard output.

4.1.1. Setting input data parameters

The anomaly detection application detects anomalies in sensor readings. Currently, the application supports gyroscope (FXAS21002), accelerometer, and magnetometer (FXOS8700CQ). Both sensors are placed on the FRDM-STBC-AGM01 sensor shield. The 3-axis accelerometer and gyroscope readings are used for the application.

Several consecutive measurements are flattened to a vector (Figure 3) and eventually pre-processed by the application before they are fed into the model. The number of consecutive samples is given by the PATCH_SIZE variable in the *adt.py* file and generated to the *parameters.h* file. Note that the input layer size is a vector of dimension equal to NUM_CHANELS*PATCH_SIZE.

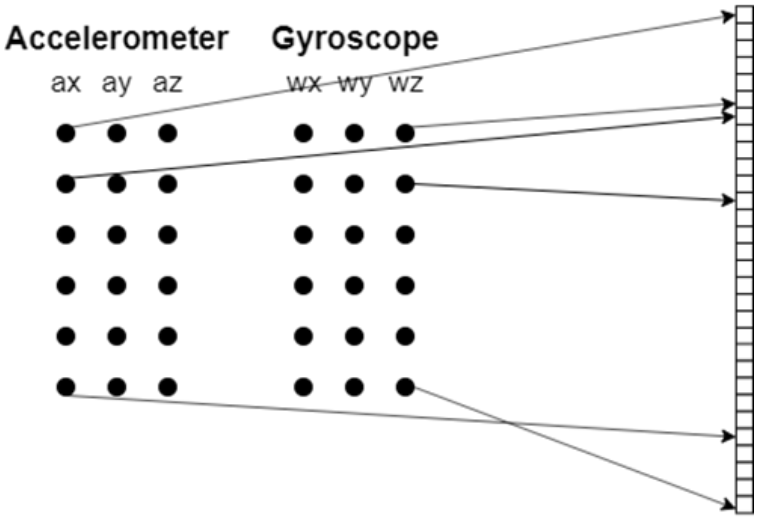

Figure 3. Transformation of raw data into input vector

By default, PATCH_SIZE is set to 5. Five consecutive measurements are used for the anomaly detection, hence the input for the model is a 30-dimensional vector.

4.1.2. Running data collect

Open the tensorflow_lite_adt example. In *adt.cpp*, set the DATA_COLLECT variable to true and run the example. By default, 1000 samples are collected. The number is experimentally set and you can change it by changing the SAMPLE_NUM variable. The data is printed to an external console. Copy and save the output to a file. It is recommended to use the SmarTTY terminal.

The first step is to collect the “good” sensor data. Put the board with sensors on a stable surface. Do not shake the sensors while collecting the training data. You can repeat the data collection with surface shakes to validate the model.

4.2. Pre-processing

Raw data are typically pre-processed before they are fed to the model (data can be filtered, statistics over data can be computed, or the Fourier transform can be applied). To detect anomalies independently from the sensor position (the accelerometer is affected by gravity), raw sensor data are derived (the difference between two consecutive readings is calculated) and the accelerometer data are divided by 10 to be of the same order of magnitude as the gyroscope data.

4.3. Neural network model

Anomalies are detected using the autoencoder. The autoencoder is defined in the main function (in the *adt.py* script) and consists of an input layer and four fully-connected hidden layers. The first two hidden layers encode; the input is mapped to the high-level features and then to the low-level features. The last two layers decode; the low-level features are decoded and the input is finaly recreated in the output layer. However, the size of the layers is relatively arbitrary, and you can experiment with it to get the most accurate result.

By default, the 30-dimensional input is encoded into 16 (ENCODING_DIM) high-level features and then into 8 (ENCODING_DIM/2) low-level features. The features are then decoded into an 8-dimensional (ENCODING_DIM) hidden layer and the output is recreated. The network is displayed in Figure 4.

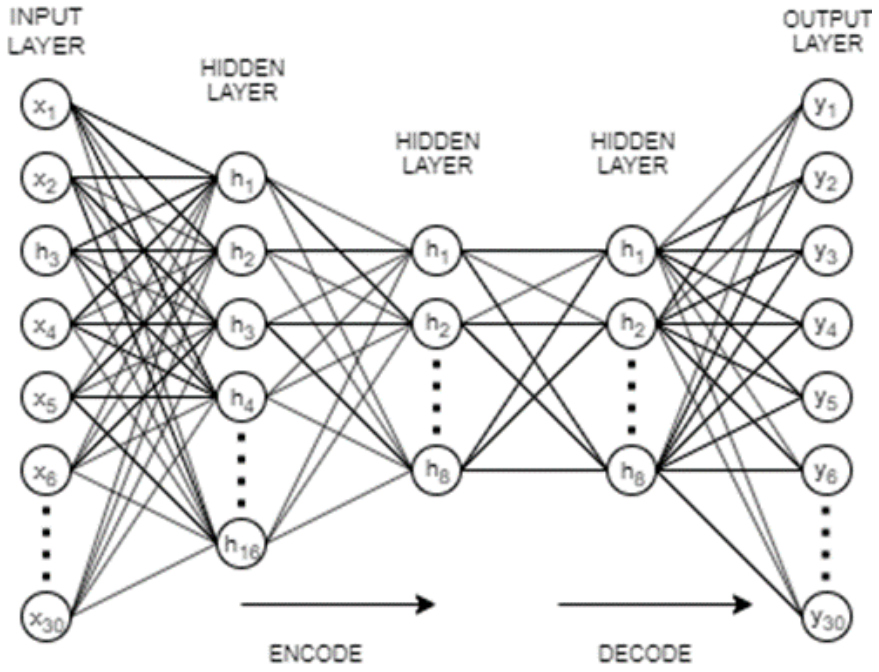

Figure 4. Autoencoder model

Each layer is activated by a tanh function (activation function explained in [6]), because the input and output layers contain negative values. In addition, the L1 regularizer (explained in [7] or [8]) is added during training to yield a sparsity autoencoder.

4.4. Model training

The model is trained for 1000 epochs (EPOCH_NUM) with a batch size of 10 samples (BATCH_SIZE). These values are set experimentally and they can be changed directly in the *adt.py* script.

4.5. Evaluation

The training script provides a visual evaluation of validation data if a file with validation data is passed as the validation_data input argument to the Python training script. It plots a reconstruction error for the validation data. Figure 4 shows the reconstruction error for validation data. The validation data were taken by a board lying on a stable surface with the surface shakes happening every 50 samples (starting with the 250th sample). Because the input data are a patch of 5 consecutive measurements, the shake happens every 10 patches (as shown in Figure 5). The reconstruction errors that are lower than the threshold are black, while the errors that are higher than the threshold are red.

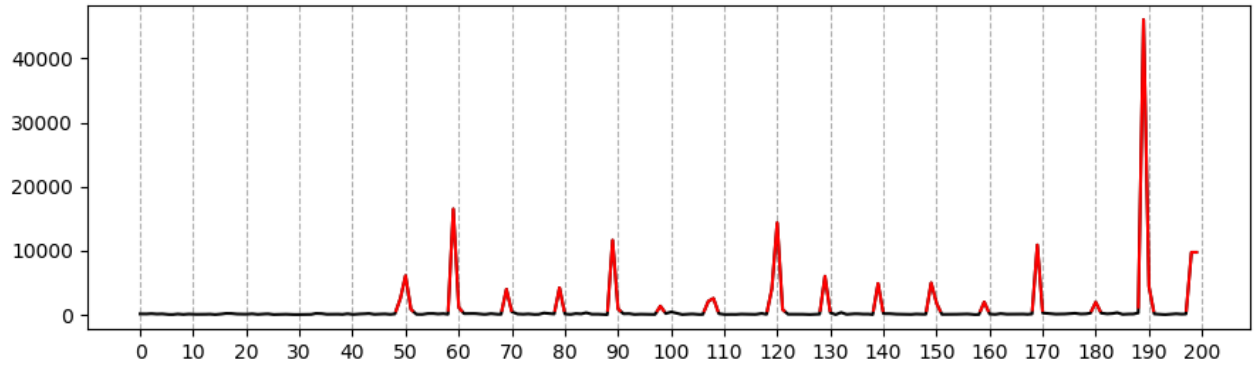

Figure 5. Reconstruction error of validation data

4.6. Deployment

The Python training script *adt.py* converts the trained model to a *tflite* file and creates a hex dump (a representation of the binary *tflite* file that is stored as an array in a C header file). Copy the header file along with the *parameters.h* file to the tensorflow_lite_adt project. The pipeline is shown in Figure 6.

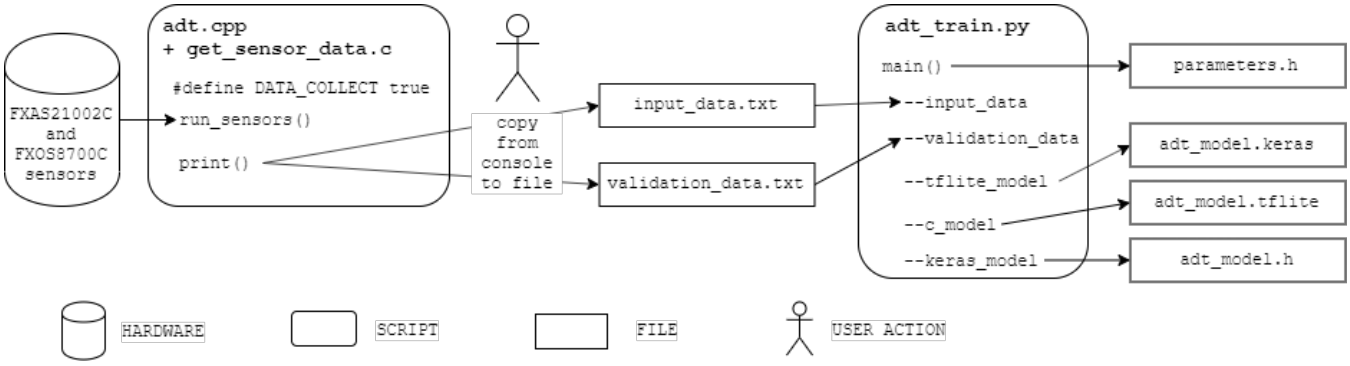

Figure 6. Training from implementation point of view

5. Inference

Inference uses the trained neural network and computes the reconstruction error for the live input from sensors. A threshold is applied to the reconstruction error. All data above this threshold are considered anomalous. The threshold is three times the average reconstruction error for training data and written by the training script to *parameters.h*. You can change the scale in *adt.py* (line 92). You can change the threshold directly by rewriting the THRESHOLD variable in *parameters.h*.

Open the tensorflow_lite_adt example, ensure that it uses the newly created *autoencoder.h* and *parameters.h* files generated in the previous step, set the DATA_COLLECT variable to false, and run the example. For every five sensor measurements, it prints inference time, the reconstruction error, and the “anomaly detected!!” text if anomalous behavior is detected. Open a terminal window to see the output that is similar to:

```
Anomaly Detection example using a TensorFlow Lite model.
Threshold value 4.80
INFO: Initialized TensorFlow Lite runtime.
(116 us) 27911.50 anomaly detected!!
(62 us) 452.91 anomaly detected!!
(58 us) 1.89
(57 us) 2.11
(51 us) 2.45
(54 us) 2.3
(46 us) 1.7
(48 us) 3.59
(45 us) 2.21
(46 us) 1.34
(44 us) 2.11
(50 us) 12.96 anomaly detected!!
(44 us) 18.15 anomaly detected!!
(46 us) 9.76 anomaly detected!!
(47 us) 8.67 anomaly detected!!
(47 us) 3.8
(46 us) 4.11
```

The inference implementation detail is shown in Figure 7.

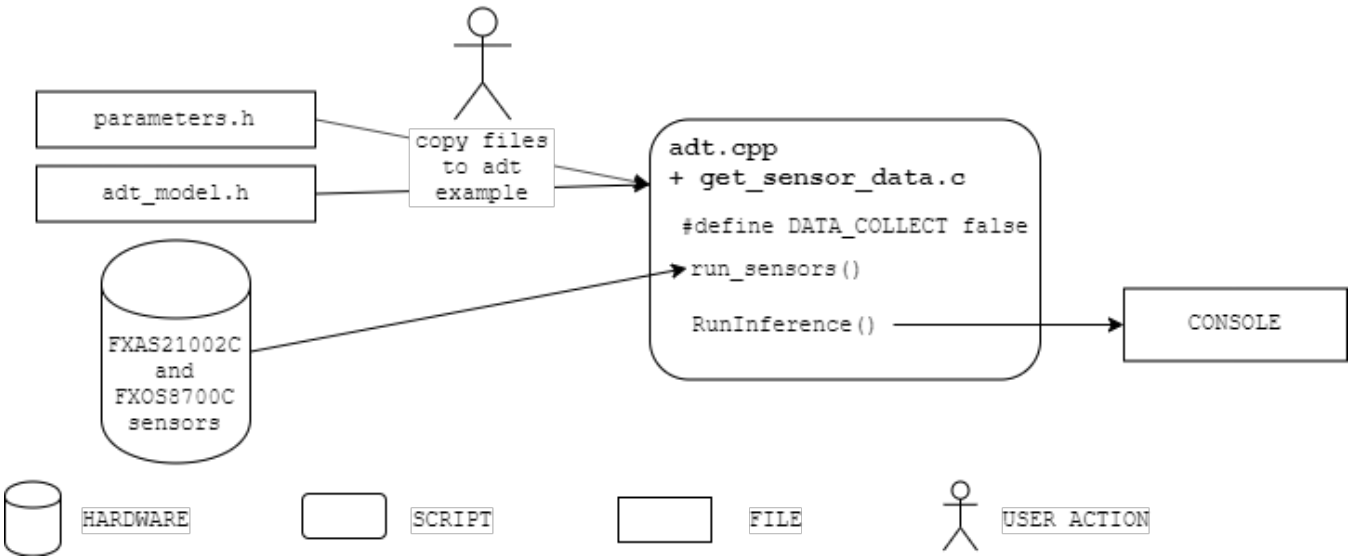

Figure 7. Inference from implementation point of view

6. Summary

The detector of anomalous behavior is implemented in the sensor data using autoencoders. Autoencoders learn the normal behavior of sensors in an unsupervised manner and give an alert when an anomaly occurs. This approach can be applied to a wide range of problems, such as unseen falls or movement detection.

7. References

1. Micucci, D., Mobilio, M., Napoletano, P. et al. J Ambient Intell Human Comput (2017) 8: 87. https://doi.org/10.1007/s12652-015-0337-0.
2. Oh DY, Yun ID. Residual Error Based Anomaly Detection Using Auto-Encoder in SMD Machine Sound. Sensors (Basel). 2018;18(5):1308. Published 2018 Apr 24. doi:10.3390/s18051308
3. Jatesiktat, Prayook & Ang, Wei. (2016). Unsupervised Anomalous Movement Detection using Autoencoder Reconstruction Error
4. J. Liu et al., "Anomaly Detection in Manufacturing Systems Using Structured Neural Networks," 2018 13th World Congress on Intelligent Control and Automation (WCICA), Changsha, China, 2018, pp. 175-180. doi: 10.1109/WCICA.2018.8630692
5. Zhou, Chong & Paffenroth, Randy. (2017). Anomaly Detection with Robust Deep Autoencoders. 665-674. 10.1145/3097983.3098052.
6. https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.
7. https://www.statisticshowto.datasciencecentral.com/regularization/.
8. https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/.

Appendix A. List of python libraries required for network training

absl-py==0.8.0
astor==0.8.0
certifi==2019.9.11
cycler==0.10.0
enum34==1.1.6
gast==0.3.2
google-pasta==0.1.7
grpcio==1.24.0
h5py==2.10.0
joblib==0.13.2
Keras==2.2.4
Keras-Applications==1.0.8
Keras-Preprocessing==1.1.0
kiwisolver==1.1.0
Markdown==3.1.1
matplotlib==3.1.0
numpy==1.16.4
pandas==0.24.2
protobuf==3.9.2
pyparsing==2.4.2
python-dateutil==2.8.0
pytz==2019.2
PyYAML==5.1.2
scikit-learn==0.21.2
scipy==1.3.1
six==1.12.0
tensorboard==1.14.0
tensorflow==1.14.0
tensorflow-estimator==1.14.0
tensorflow-model-optimization==0.1.2
termcolor==1.1.0
Werkzeug==0.16.0
wincertstore==0.2
wrapt==1.11.2

How to Reach Us:

Home Page:
www.nxp.com

Web Support:
www.nxp.com/support

Document Number: IMXRTADUG
Rev. 0
01/2020