PREDICTIVE MAINTENANCE

Technical documentation

LUKA ČABRAJA
PATRIK ĐURĐEVIĆ
DOMINIK FISTRIĆ
PETAR JERONČIĆ

Table of contents

I. Overview

In this technical documentation, we are going to explain the steps we took for a predictive maintenance project in more technical detail. As we said in the documentation, we decided on using neural networks to create models in order to achieve our goal. For that, we used Tensorflow. Tensorflow is a symbolic math library used for machine learning applications such as neural networks. It was created and developed by the Google Brain Team.

At first, we started developing an autoencoder, but soon after we decided that that wasn't going to give us expected results. For that reason we deferred from an autoencoder and started creating a RNN.

II. RNN model

The model we have successfully used for predictive maintenance is a sequential RNN (recurrent neural network) model. A sequential model is basically a plain stack of layers where each layer has exactly one input and one output tensor[1], whereas a recurrent neural network is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows them to exhibit temporal dynamic behavior. We used the following structure.

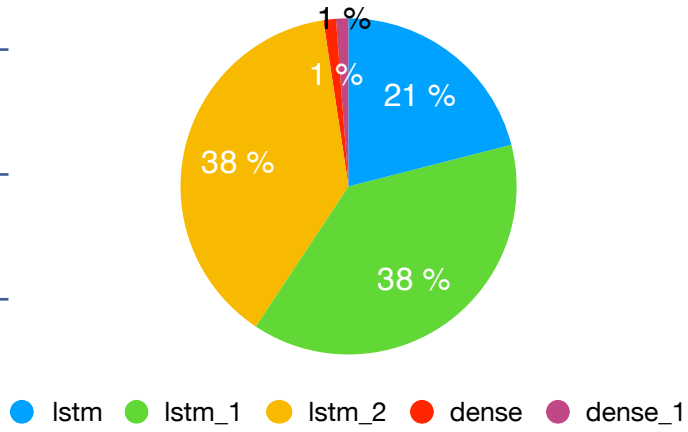
RNN model structure

Layer (type)	Units	Activation	Output shape BS - Batch size	Number of parameters
lstm (LSTM)	128	Tanh	(BS, 30, 128)	72192
dropout (Dropout)	0,2		(BS, 30, 128)	0
lstm_1 (LSTM)	128	Tanh	(BS, 30, 128)	131584
dropout_1 (Dropout)	0,2		(BS, 30, 128)	0
lstm_2 (LSTM)	128	Tanh	(BS, 128)	131584
dense (Dense)	32	Relu	(BS, 32)	4128
dense_1 (Dense)	2		(BS, 1)	33

RNN model

Trainable parameters
339,521
Loss function
Mean squared error
Optimizer
Adam

Trainable parameters



The network is mainly comprised of LSTM and Dense layers with Dropout layers in between. The LSTM layers are crucial for our application because they give our model a sense of time and sequences. We can also see that most of the trainable parameters are located within them. Dense layers at the end are used to take in the output from the LSTM layers and then convert it into an actual prediction of the data. To combat overfitting on the training data, we used 2 Dropout layers between LSTMs. It works by randomly setting the input units to 0 with a specific frequency during training.

[1] A tensor is a generalization of vectors and matrices to potentially higher dimensions.

III. Training

When it came time to train the model, we used two approaches: per machine training and transfer learning. As discussed in the documentation, after seeing the results we decided that transfer learning was the way to go. In both approaches we trained the models by feeding them with 30 day windows of data from all 12 of the sensors. The goal of the model was to predict a value of a certain sensor the next day.

A. Per machine training

The dataset contained information for 12 sensors on 7 different machines. What we did with this approach is create 12 models per machine, 84 in total. Each model was trained on its respective machine to predict a certain sensor value. The data was split 70/30[2], randomly shuffled and fed to the model with a batch size of 1 for 100 epochs.

B. Transfer learning

Transfer learning was done differently from per machine training. First, we created 12 of so called "general" models which we trained on the whole dataset (all machines) with similar specifications to per machine training. The data was split 70/30, randomly shuffled, fed to the model with a batch size of 1 for 40 epochs. Later, we took these 12 models and created 7 copies from each one of them. We were left with 84 models trained on the whole dataset, which means that they are really good at generalizing, but not so good at predicting a specific machine's data. Therefore, we took these 84 models and further trained them on the 7 respective machine's data. This was done with the data being split 70/30, randomly shuffled, fed to the model with a batch size of 1 for 60 epochs. That's 100 epochs per model in total (same as the per machine training). At the end, we had 84 models which were way better at predicting than their per machine training counterparts.

[2] 70% of the dataset was used for training and 30% for validation.

IV. Guide

In this section, we'll show you how you can replicate the process we did during this project with our code and achieve similar results[3].

A. Prerequisites

To begin with, you will need to install the following[4] on your computer:

- Python (3.7.7),
- Tensorflow (2.1.0),
- Scipy (1.4.1),
- Scikit-learn (0.22.2.post1),
- Tqdm (4.45.0),
- Numpy (1.18.2),
- Seaborn (0.10.0),
- Pandas (1.0.3),
- Matplotlib (3.2.1).

Every package listed here can be installed using the pip package manager.

B. Setting up the dataset

Before you can start training the models, please place the dataset file (*dataset.csv*) in the project's root directory.

C. Training the models

You can train the above mentioned models by running the following Python files (located in *models/RNN/*):

- for per machine training run *model_lstm.py*,
- for transfer learning run:
  - *model_lstm_general.py* first,
  - *model_lstm_transfer.py* afterwards.

The scripts should be run from the *models/RNN/* folder.

D. Models

After training the models, you can find the saved ones[5] in the models directory:

- *models/RNN/models/per_machine* for per machine training,
- *models/RNN/models/general_h5* for general models,
- *models/RNN/models/transfer* for transfer learning.

The code automatically saves the latest model, the best model based on the validation data, and the best model based on the training data.

E. Notes

Since we split the data of machines FL01 and FL07, there are models named FL01_before, FL01_after, FL07_before, and FL07_after. The final models which represent full training are the ones in the *_after folders.

---

[3] There can be a slight variation in results since the models are initialized with random weights before training.

[4] We listed the version we had installed on our machine (we are not excluding the possibility that the code can work on newer package versions).

[5] The models are saved in the .h5 format

V. Tensorflow Lite

We anticipate that there is a great chance that our models will be used on embedded, low powered devices. For that reason, we decided to write a piece of Python code which converts saved .h5 models to TensorFlow Lite models. TensorFlow Lite is an open source deep learning framework for on-device inference. It enables everyone to deploy machine learning models on mobile and IoT devices. The script (*converter.py*) can be found in the *tflite* folder located in the main project's directory.