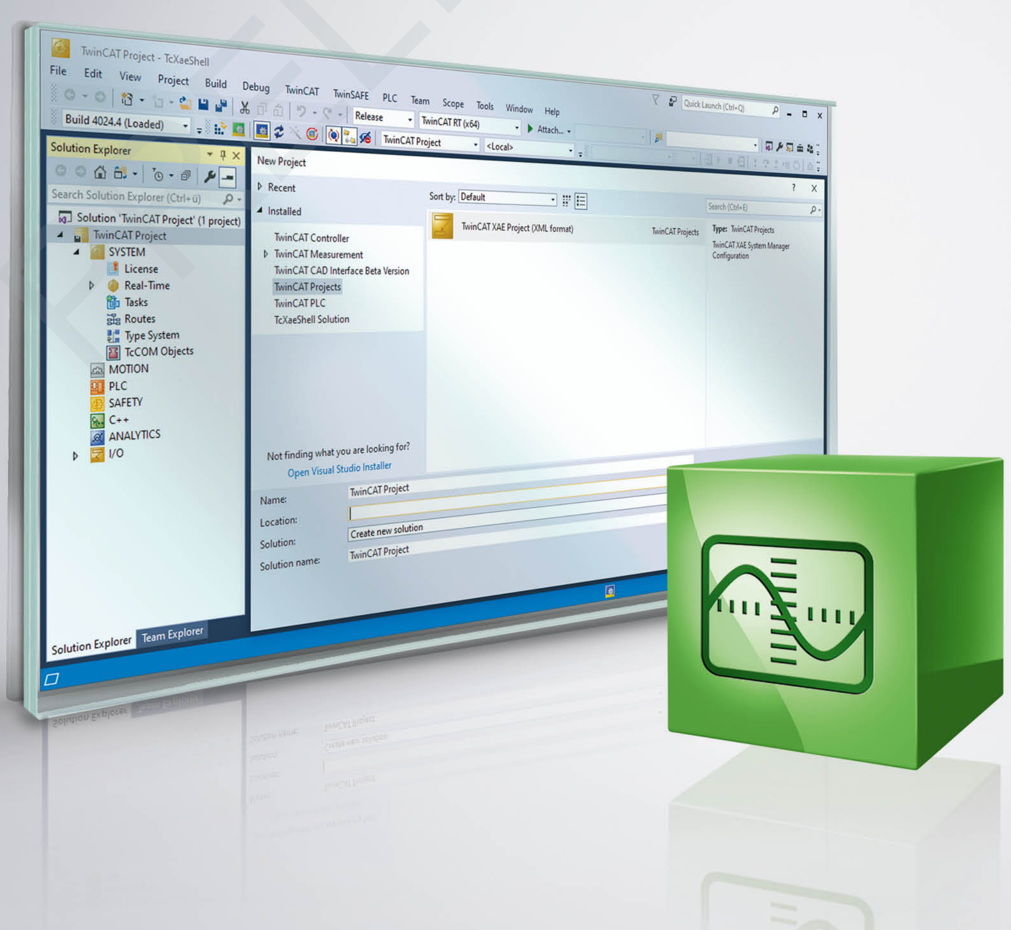
BECKHOFF New Automation Technology

Manual | EN

TF38x0

TwinCAT 3 | ML/NN Inference Engine

PRELIMINARY



8/5/2020

Table of contents

1 **Foreword** **5**
1.1 Notes on the documentation 5
1.2 Safety instructions 6

2 **Overview** **7**
2.1 Workflow 8

3 **Installation** **14**
3.1 System requirements 14
3.2 Installation 14
3.3 Licensing 17

4 **Quick start** **20**

5 **Machine Learning Models and file formats** **24**
5.1 Machine learning models supported 24
5.1.1 Multi-layer perceptron 24
5.1.2 Support vector machine 26
5.2 Supported file formats 26
5.2.1 Open Neural Network Exchange (ONNX) 27
5.2.2 Beckhoff ML XML 28
5.2.3 Beckhoff ML BML 30
5.3 File management of the ML description files 30

6 **Configuration** **36**
6.1 TC3 Machine Learning Model Manager 36
6.2 XML Exporter 40

7 **API** **42**
7.1 TcCOM 42
7.2 PLC API 44
7.2.1 Datatypes 44
7.2.2 Function Blocks 45

8 **Samples** **58**
8.1 XML Exporter - samples 58
8.2 PLC API 58
8.2.1 Quick start 58
8.2.2 Detailed example 58
8.2.3 Parallel, non-blocking access to an inference module 58

9 **Appendix** **60**
9.1 Log files 60

PRELIMINARY

PRELIMINARY

1 Foreword

1.1 Notes on the documentation

This description is only intended for the use of trained specialists in control and automation engineering who are familiar with applicable national standards.
It is essential that the documentation and the following notes and explanations are followed when installing and commissioning the components.
It is the duty of the technical personnel to use the documentation published at the respective time of each installation and commissioning.

The responsible staff must ensure that the application or use of the products described satisfy all the requirements for safety, including all the relevant laws, regulations, guidelines and standards.

Disclaimer

The documentation has been prepared with care. The products described are, however, constantly under development.
We reserve the right to revise and change the documentation at any time and without prior announcement.
No claims for the modification of products that have already been supplied may be made on the basis of the data, diagrams and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered trademarks of and licensed by Beckhoff Automation GmbH.
Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:
EP1590927, EP1789857, EP1456722, EP2137893, DE102015105702
with corresponding applications or registrations in various other countries.

EtherCAT® is a registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany

Copyright

© Beckhoff Automation GmbH & Co. KG, Germany.
The reproduction, distribution and utilization of this document as well as the communication of its contents to others without express authorization are prohibited.
Offenders will be held liable for the payment of damages. All rights reserved in the event of the grant of a patent, utility model or design.

PRELIMINARY

1.2 Safety instructions

Safety regulations

Please note the following safety instructions and explanations!
Product-specific safety instructions can be found on following pages or in the areas mounting, wiring, commissioning etc.

Exclusion of liability

All the components are supplied in particular hardware and software configurations appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

Personnel qualification

This description is only intended for trained specialists in control, automation and drive engineering who are familiar with the applicable national standards.

Description of symbols

In this documentation the following symbols are used with an accompanying safety instruction or note. The safety instructions must be read carefully and followed without fail!

⚠ DANGER

Serious risk of injury!

Failure to follow the safety instructions associated with this symbol directly endangers the life and health of persons.

⚠ WARNING

Risk of injury!

Failure to follow the safety instructions associated with this symbol endangers the life and health of persons.

⚠ CAUTION

Personal injuries!

Failure to follow the safety instructions associated with this symbol can lead to injuries to persons.

NOTE

Damage to the environment or devices

Failure to follow the instructions associated with this symbol can lead to damage to the environment or equipment.

Tip or pointer

This symbol indicates information that contributes to better understanding.

PRELIMINARY

2 Overview

Introduction

The idea behind machine learning is to learn a generalized correlation between inputs and outputs on the basis of example data. Accordingly, a certain amount of training data is required, on the basis of which a **model** is trained. In the training of the model, parameters of the model are adapted automatically to the training data by means of a mathematical process. In machine learning, the user has a large number of different models at his disposal. The selection and design of the models is part of the engineering process. Different types or designs of models fulfill different tasks, wherein the most important subdivisions are the classification and regression.

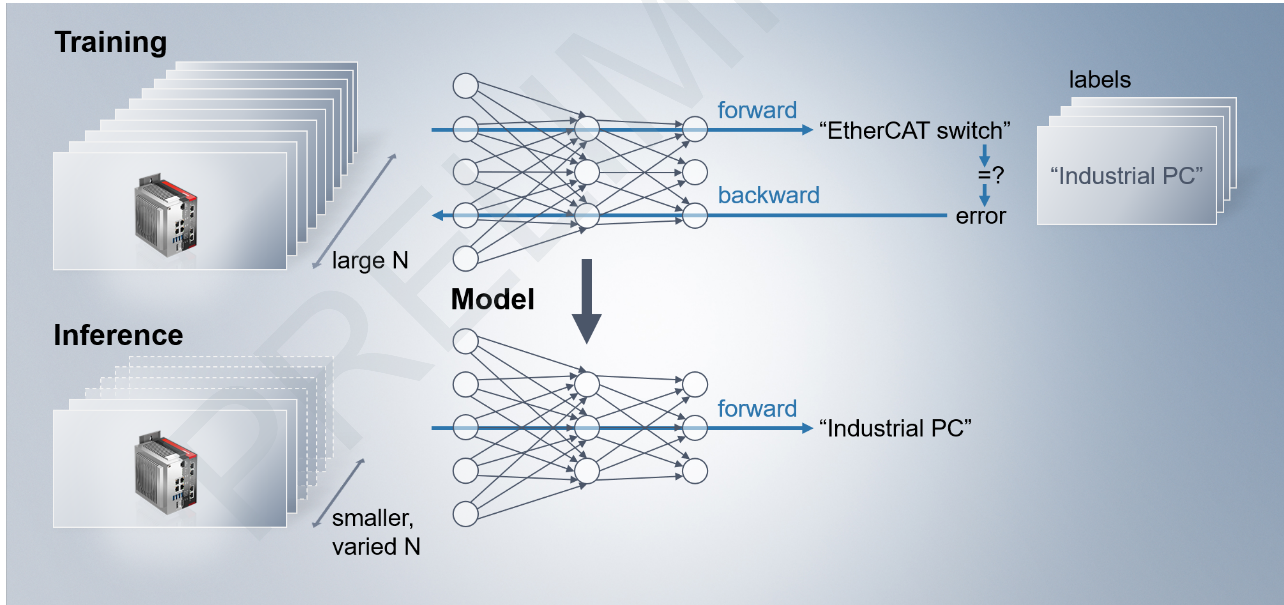
Classification:

The model contains an input (an image, one or more vectors, etc.) and assigns it to a class. The output is correspondingly a categorized variable. These classes could be, for example, good part or bad part. Distinction can also be made between several classes, for example quality classes A, B, C, D.

Regression:

The model contains an input and generates a continuous output. Not only are directly learned inputs assigned to directly learned outputs (as with a lookup table), but in addition the model is able to interpolate or, respectively, extrapolate non-learned inputs, provided it generalizes well. A functional correlation is learned.

Once a model has been **trained**, it can be used for the learned task, i.e. the model is used for **inference**. This is illustrated again in the graphic below.
In the training phase, for example of a neural network, inputs are shown to a not yet completely learned model. In the case of a known result (label), the error observed is fed back again via the model. The model is adapted as a result. The adaptations are continued until the error observed is acceptable.
In the inference, the model is only run through in the forward direction and used for the prediction of results.



TC3 Machine Learning Runtime

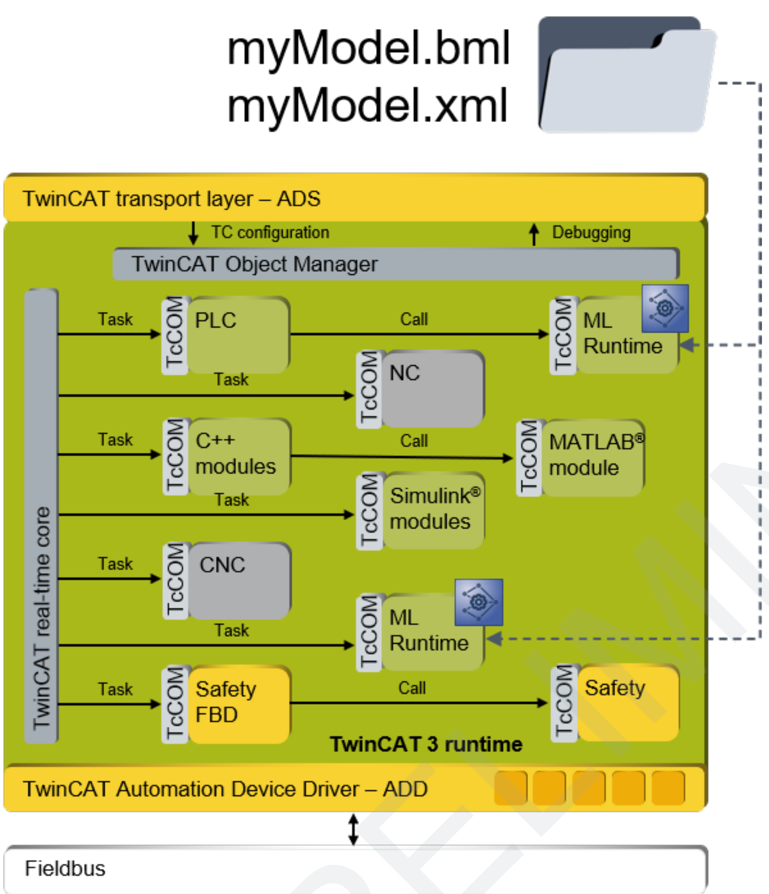
Beckhoff provides components for inference with the products **TF3800 TC3 Machine Learning Inference Engine** and **TF3810 Neural Network Inference Engine**. A common basis is used for both products, which is referred to in the following as the *Machine Learning Runtime*.

The Machine Learning Runtime is a module integrated in TwinCAT 3 and executed in the TwinCAT XAR. Access to all available data in the controller and also the use of the model in hard real-time is thus possible.

Two ways of engineering the components are offered:

- **PLC API**
  The library TC3_MLL is provided for use in the PLC environment. The library can be incorporated like any other PLC library. Machine learning models can be loaded asynchronously via a method call and subsequently executed cyclically in the PLC program by calling a further method.
- **Static TcCOM instance**
  A TcCOM object that can simply be inserted and configured in the TwinCAT object tree in the System Manager. On starting the system, the TcCOM loads the configured model and executes it in the assigned cycle time. A simple method of machine learning without programming effort.

The picture below illustrates the deep integration of the Machine Learning Runtime in the TwinCAT XAR. Like all TwinCAT Runtime objects, the module is a TcCOM and is accordingly anchored deep in the hard real-time.

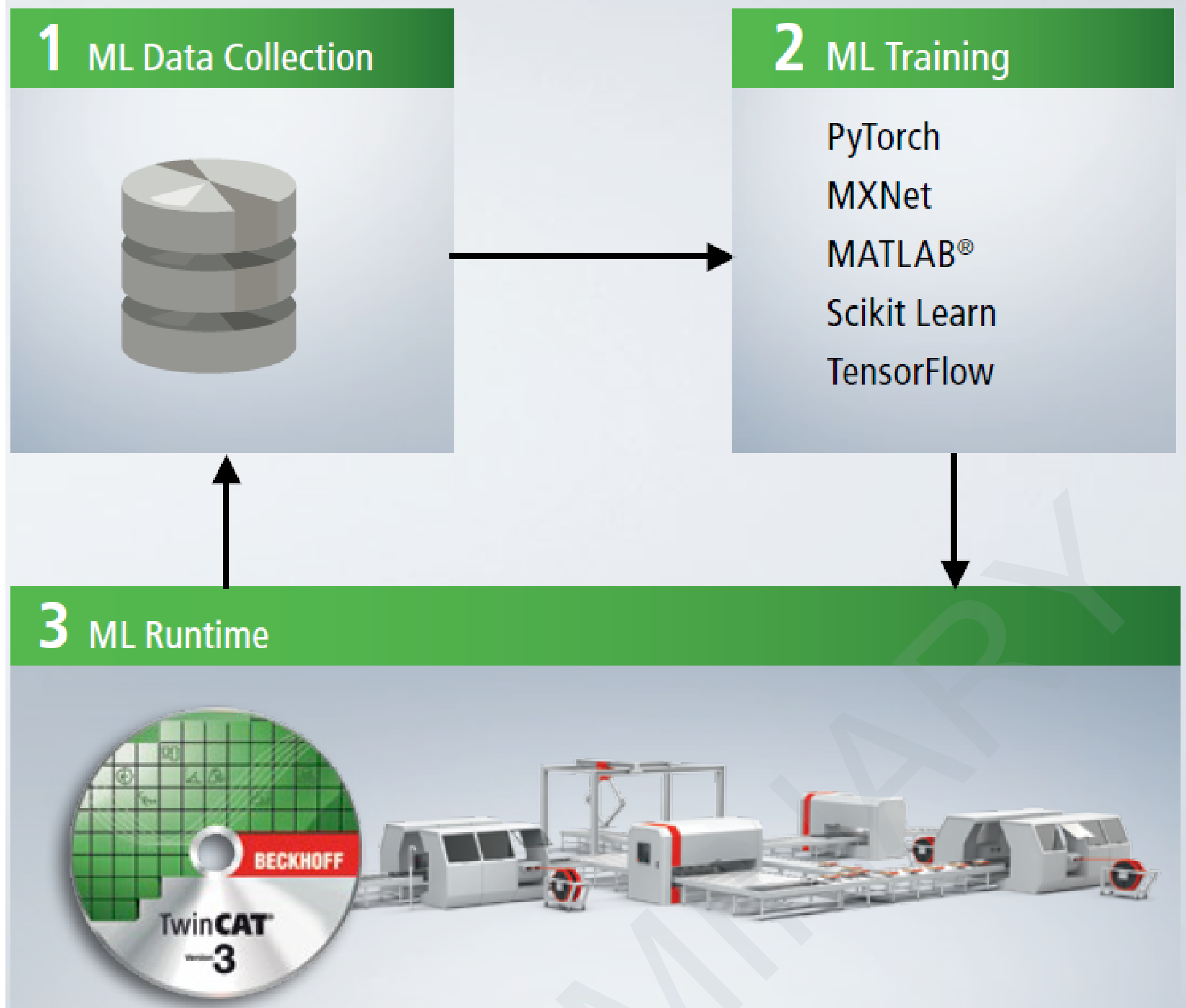


PRELIMINARY

2.1 Workflow

Basically, the machine learning process and the integration in TwinCAT3 described here consist of three phases.

- The collection of data (1)
- The training of a model (2)
- The deployment in the TwinCAT XAR (3)



PRELIMINARY

Archiving data

Depending on the application and also after the differentiation into engineering phase, commissioning and production phase, different tools for data communication and data storage can be used. The illustration below shows a subset of possible products from the area of TwinCAT 3 Measurement and TwinCAT 3 Connectivity.

In the engineering phase, the TC3 Scope with its storage function as binary file, CSV, TDMS and other formats is a simple and flexible tool for collecting and storing data. The tool can be used intuitively and quickly via the user interface of the Scope View. The Scope Server can also be controlled directly from the PLC via the PLC function block `FB_ScopeServerControl` (Tc2_Utilities library) and thus used precisely in terms of time as well as reproducibly.

If an SQL or noSQL database is available (local, in the network or in the cloud), the TC3 Database Server is an ideal tool for aggregating large quantities of data during the machine running time. The Database Server is usable via the PLC and also via a pure configuration interface and supports a large number of different file formats and databases, from ASCII and Excel files to MS SQL and Mongo DB.

If the machine is connected via a message broker to a public or private cloud, the TwinCAT IoT product family provides the option to save the data via the broker in a datastore or warehouse. Due to the decoupled communication via the message broker, new data sources can be integrated very simply into the system in this scenario. Furthermore, third-party systems can also be integrated via the OPC-UA interface of the IoT Data Agent

In running operation, the TC3 Analytics Logger can also be used to save large amounts of data from the machine controller conveniently and synchronously with the cycle in a local file or via a message broker in a database.

Further options are, for example, the writing of MAT files for processing in MATLAB® with the TcExtendedFileWriter, the sending of data via TC3 OPC UA or the use of the File Function Blocks from the basic PLC library Tc2_System.

Training models

If training data are available in files or databases, they must be read into and processed in a Machine Learning Framework. From the point of view of the user, there is a large degree of freedom in the choice of a suitable framework, whether from an application aspect or a personal one.

A large number of ML frameworks are based on Python™ or R and are open source, while others are commercially available, such as MATLAB® and SAS. A wide range of standard functions and toolboxes/ libraries serve here as data interfaces for accessing the stored data, not least because the TwinCAT products named support standard formats, and binary file formats are open.

Roughly speaking, the work with the ML Framework consists of the preparation and selection of the data, the modeling and training of a suitable ML algorithm and, ultimately, the evaluation of the algorithm learned. The result of this engineering step is a trained ML model.

PRELIMINARY

The model learned now has to be exported out of the selected ML Framework and used in the ML Runtime within the TwinCAT Runtime. For this, Beckhoff supports the standardized Open Neural Network Exchange format (ONNX) on the one hand and two of its own proprietary formats on the other. The proprietary formats are firstly a readable XML format and secondly, for the delivery, a binary data format (BML). Read more about the formats here: Supported file formats [▸ 26].

Support for the ONNX description format provides flexibility and openness in comparison with the various established ML Frameworks, especially when working with neuronal networks. An overview of the supported ML Frameworks can be found on the ONNX Community's website: http://onnx.ai/supported-tools. The most important frameworks such PyTorch, Keras, scikit-learn, etc. are thus already supported. In addition to direct support from various frameworks, various converter tools also exist, such as MMdnn for example.

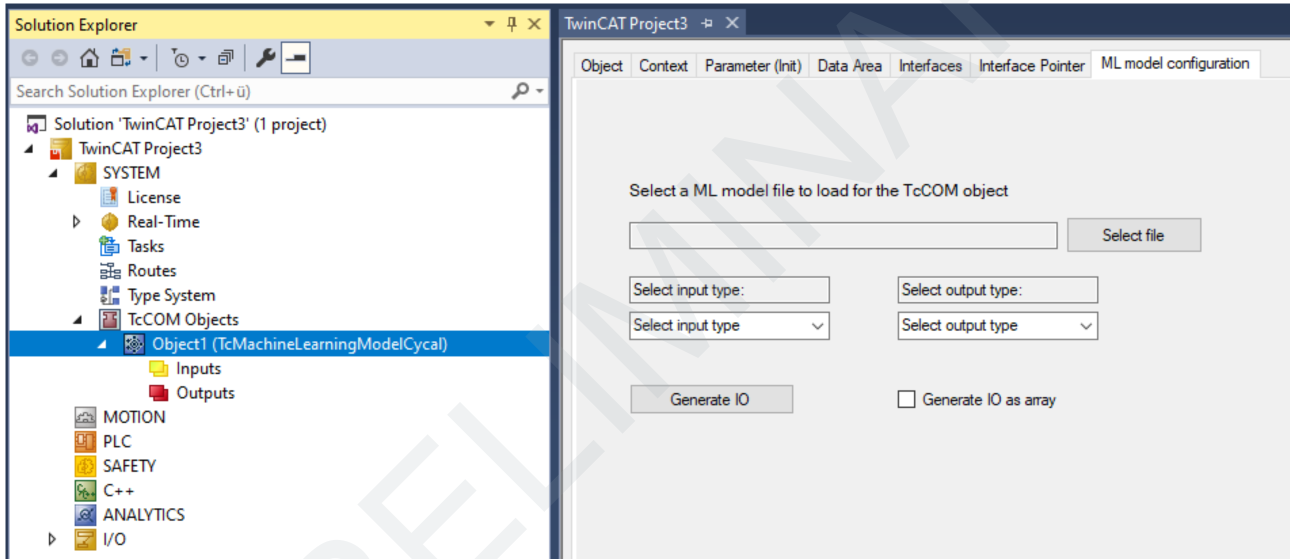
Conversely, the introduction of a proprietary format offers Beckhoff the possibility to embed TwinCAT-specific properties in the description. For this reason, ONNX files are not read directly into the Machine Learning Runtime, but must first be converted to XML or BML and can optionally be supplemented with additional information for the Runtime. Further information can be found in Supported file formats [▸ 26].

Running models in TwinCAT 3

The TC3 ML Model Manager [▸ 36] is provided for working with model description files (ONNX, XML and BML). The component is directly integrated in Visual Studio and, in addition to the simple conversion of file formats, offers the option to import additional information that the PLC programmer can then use in his real-time application.

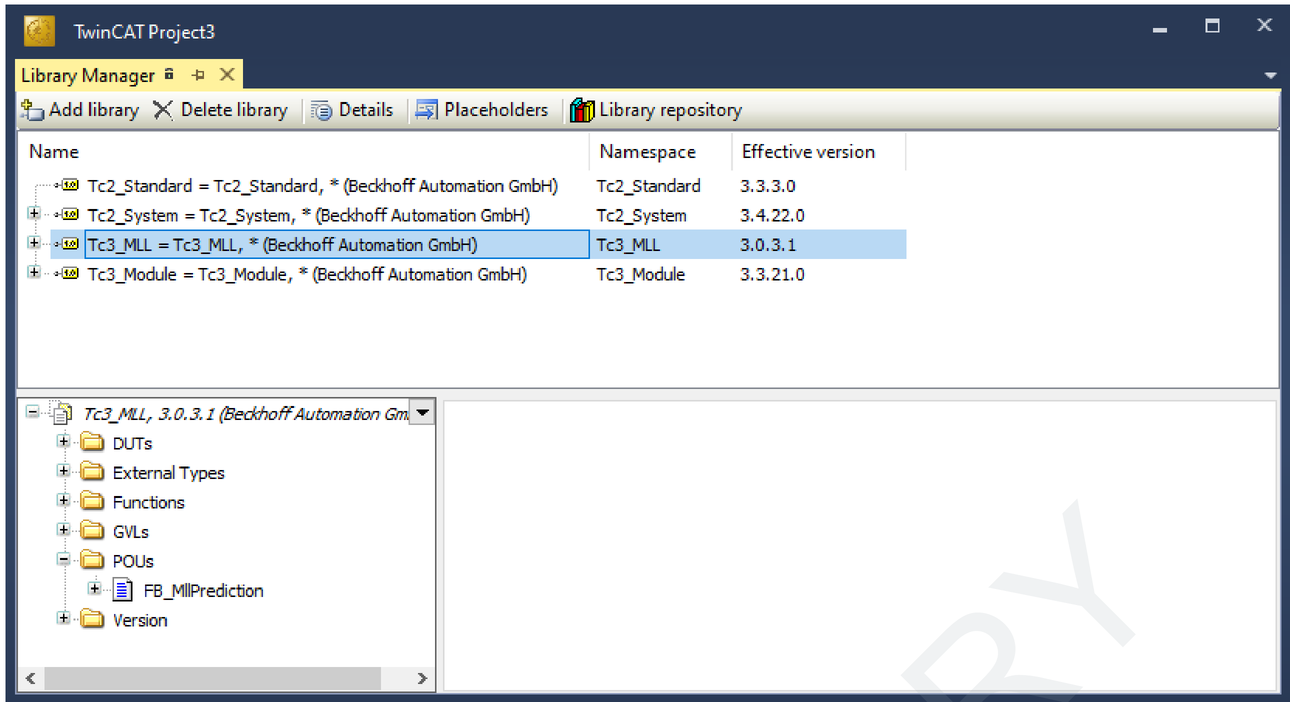
Deeper knowledge of Machine Learning is not required for integrating trained models into TwinCAT 3. The algorithm learned is fully specified in the model description file and only needs to be integrated into the TwinCAT project as an executable function. There are two ways to do this:

- Integration as a static TcCOM object



- Integration via the PLC library Tc3_MLL

PRELIMINARY



Further information on the integration in TwinCAT can be found in API [▸ 42] or a quick introduction in Quick start [▸ 20].

Following successful integration, the ML models are run synchronously to the cycle in the context of the TwinCAT real-time. Accordingly, all data existing in the controller are available in real-time and the results of the ML model can be output and processed further in real-time.

Updating an ML model in TwinCAT

Trained models can be exchanged via the PLC library without recompiling and without restarting the TwinCAT Runtime. To do this, the new model description file merely has to be copied to the target system, after which the Configure method of the FB_MllPrediction is executed, which then loads the new file. The copying procedure can take place in common ways: FTP, ADS, Shared Folder, and so on. The trigger for the Configure method can be sent manually via an HMI or automatically from a script via ADS. Further information on these update scenarios: File management of the ML description files [▸ 31].

The circle of data collection, training and deployment in TwinCAT is thus closed and can be run through repeatedly if necessary via the machine runtime.

PRELIMINARY

3 Installation

3.1 System requirements

Runtime

Technical data	Description
Operating system	Windows 7 64-bit, Windows Embedded Standard 7 64-bit, Windows 10 64-bit
Target platform	PC architecture (x64)
TwinCAT version	TwinCAT 3.1 build 4024.0 or higher
Required TwinCAT setup level	TwinCAT 3 XAR
Required TwinCAT license	TF3800 TC3 Machine Learning Inference Engine or TF3810 TC3 Neural Network Inference Engine

Engineering

Technical data	Description
TwinCAT version	TwinCAT 3.1. build 4024.0 or higher
Required TwinCAT setup level	TwinCAT 3 XAE

The setup is to be executed both on the Engineering PC and on the Runtime PC.
7-day trial licenses can be generated for the runtime

3.2 Installation

The following section describes how to install the TwinCAT 3 Function for Windows-based operating systems.

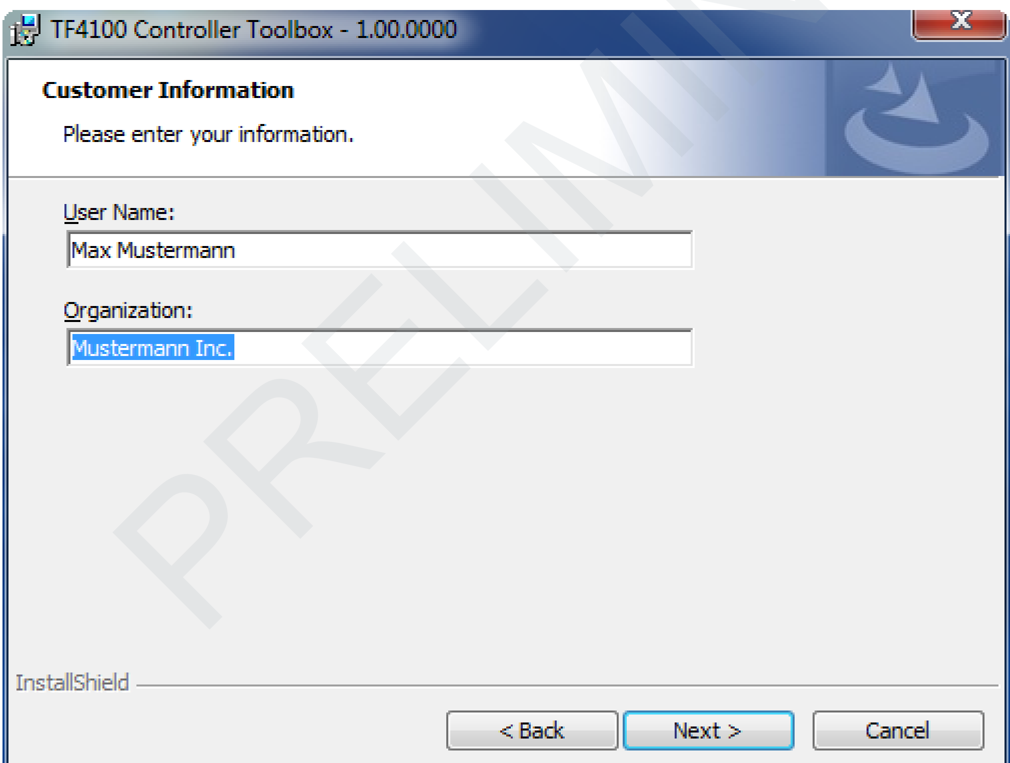
✓ The TwinCAT 3 Function setup file was downloaded from the Beckhoff website.

1. Run the setup file as administrator. To do this, select the command **Run as administrator** in the context menu of the file.
   ⇨ The installation dialog opens.

PRELIMINARY

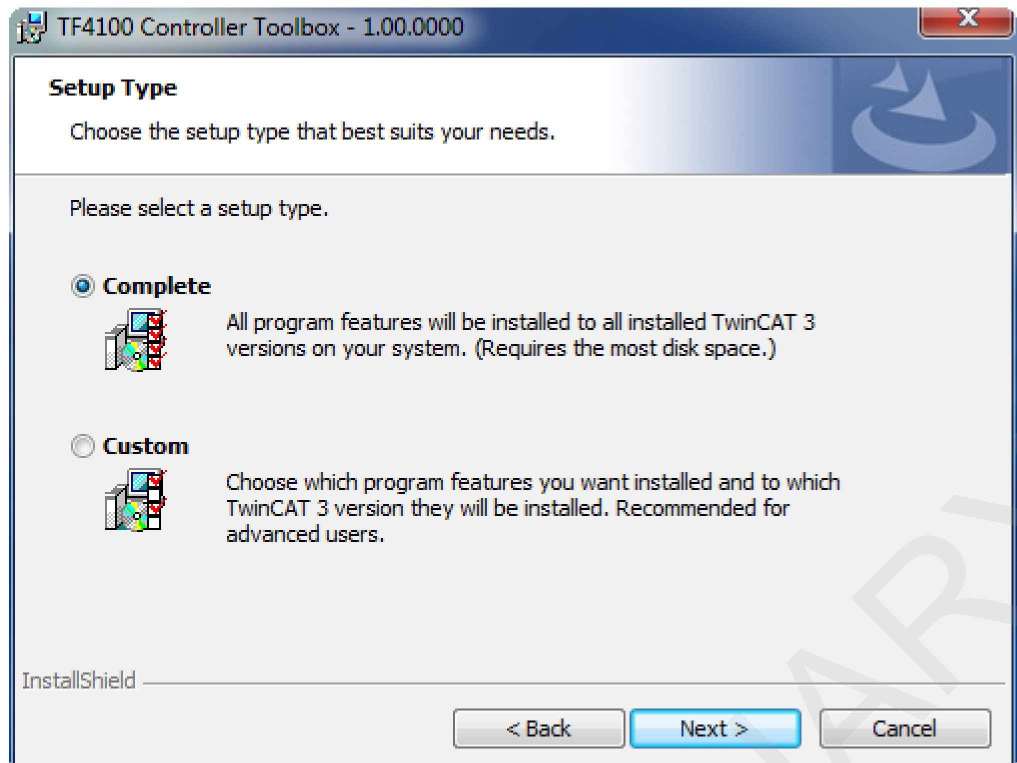
2. Accept the end user licensing agreement and click **Next**.

3. Enter your user data.

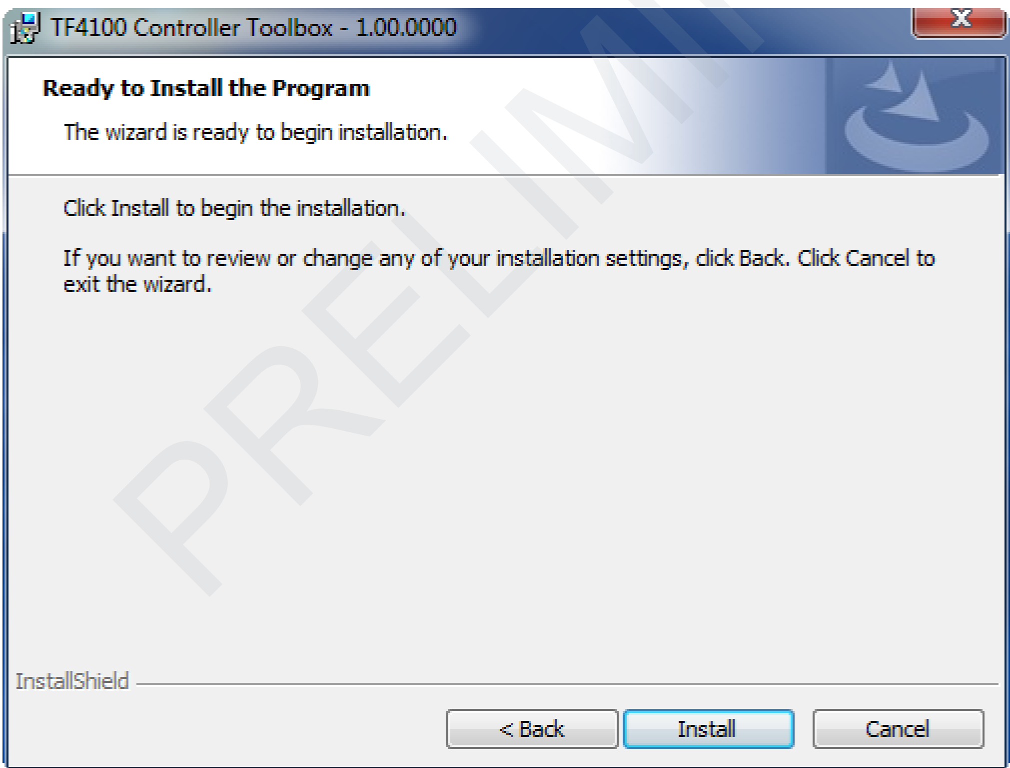


PRELIMINARY

4. If you want to install the full version of the TwinCAT 3 Function, select **Complete** as installation type. If you want to install the TwinCAT 3 Function components separately, select **Custom**.



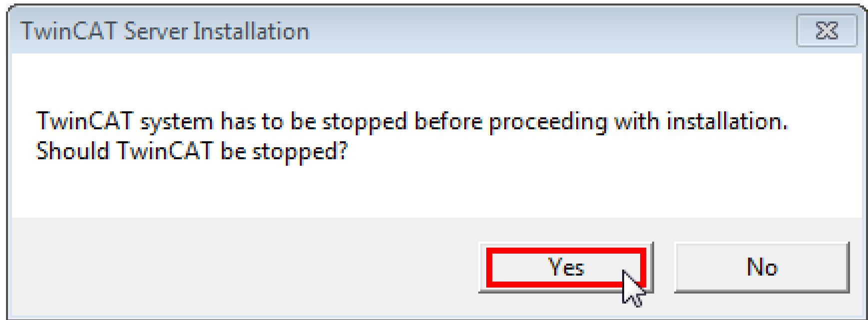
5. Select **Next**, then **Install** to start the installation.



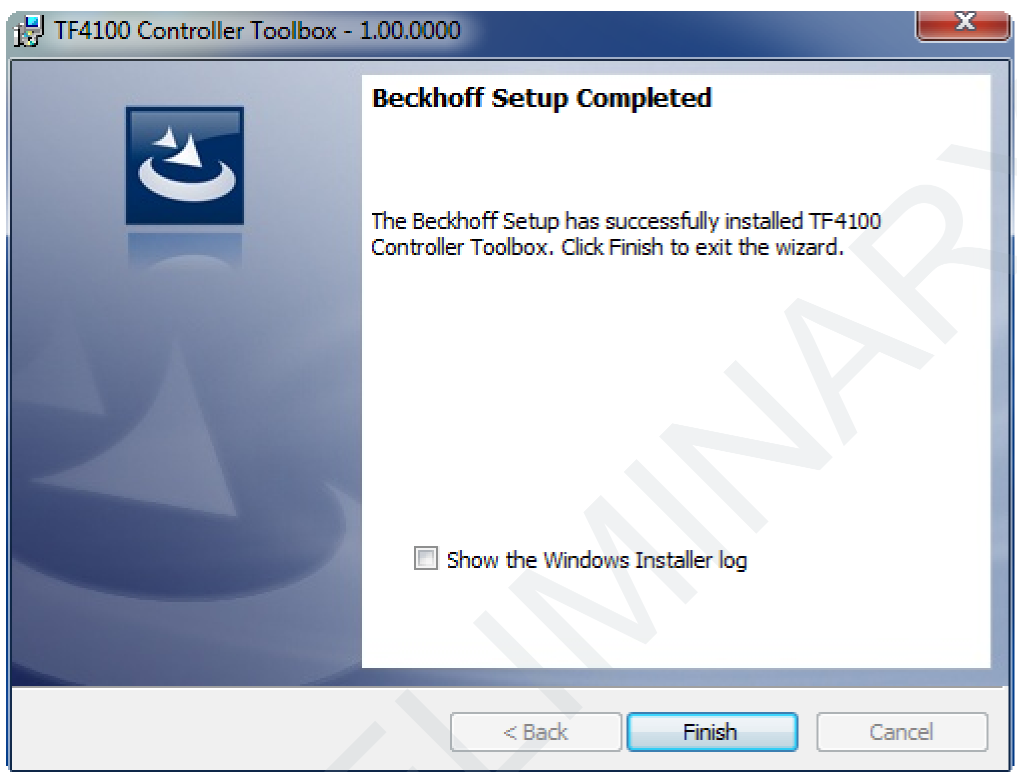
⇨ A dialog box informs you that the TwinCAT system must be stopped to proceed with the installation.

PRELIMINARY

6. Confirm the dialog with **Yes**.



7. Select **Finish** to exit the setup.



⇨ The TwinCAT 3 Function has been successfully installed and can be licensed (see Licensing [▶ 17]).

3.3 Licensing

The TwinCAT 3 function can be activated as a full version or as a 7-day test version. Both license types can be activated via the TwinCAT 3 development environment (XAE).

Licensing the full version of a TwinCAT 3 Function

A description of the procedure to license a full version can be found in the Beckhoff Information System in the documentation "TwinCAT 3 Licensing".

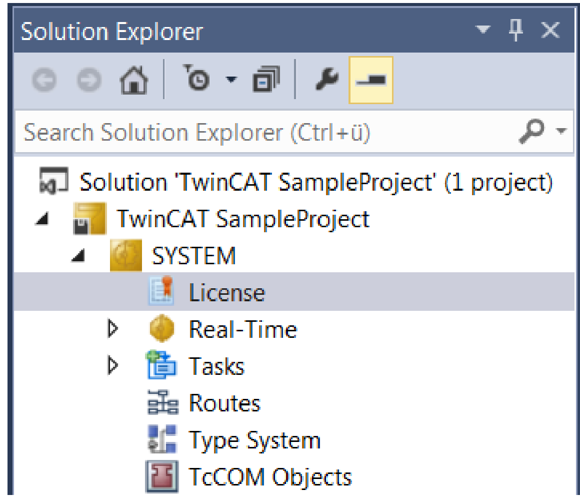
Licensing the 7-day test version of a TwinCAT 3 Function

A 7-day test version cannot be enabled for a TwinCAT 3 license dongle.

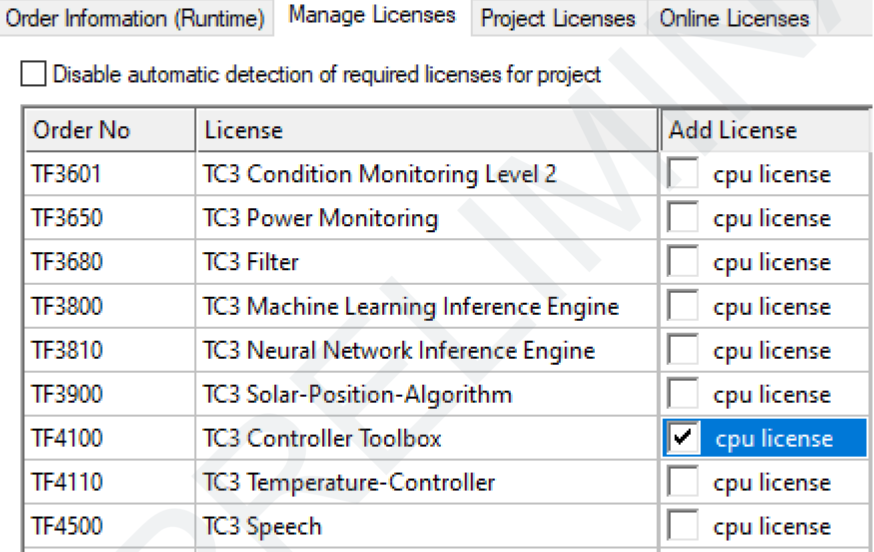
1. Start the TwinCAT 3 development environment (XAE).
2. Open an existing TwinCAT 3 project or create a new project.

PRELIMINARY

3. If you want to activate the license for a remote device, set the desired target system. To do this, select the target system from the **Choose Target System** drop-down list in the toolbar.
   ⇨ The licensing settings always refer to the selected target system. When the project is activated on the target system, the corresponding TwinCAT 3 licenses are automatically copied to this system.
4. In the **Solution Explorer**, double-click **License** in the **SYSTEM** subtree.



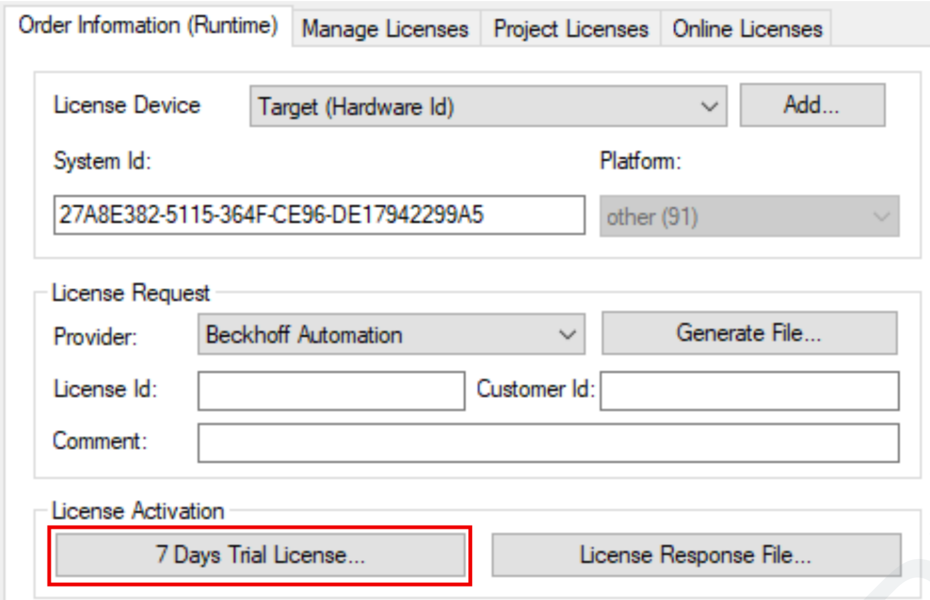
   ⇨ The TwinCAT 3 license manager opens.
5. Open the **Manage Licenses** tab. In the **Add License** column, check the check box for the license you want to add to your project (e.g. "TF6420: TC3 Database Server").



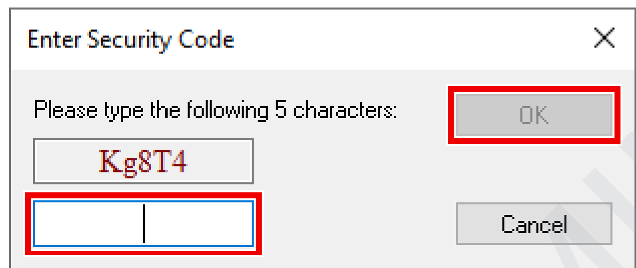
6. Open the **Order Information (Runtime)** tab.
   ⇨ In the tabular overview of licenses, the previously selected license is displayed with the status “missing”**.**

PRELIMINARY

7. Click **7-Day Trial License...** to activate the 7-day trial license.



⇨ A dialog box opens, prompting you to enter the security code displayed in the dialog.



8. Enter the code exactly as it is displayed and confirm the entry.
9. Confirm the subsequent dialog, which indicates the successful activation.
   ⇨ In the tabular overview of licenses, the license status now indicates the expiry date of the license.
10. Restart the TwinCAT system.

⇨ The 7-day trial version is enabled.

PRELIMINARY

4 Quick start

Convert model (optional)

There is an ML description file (*KerasMLPExample_cos.XML*) in the ZIP of the Quickstart PLC API (Resources/zip/8746884875.zip). This file can be used directly for integration in TwinCAT, or optionally converted to a BML (binary format).

- Open the Machine Learning Model Manager [▸ 36]
  - Menu bar: (Extensions)* > TwinCAT > Machine Learning > Machine Learning Model Manager
    * Visual Studio 2019 only
- Use the Convert tool
  - Click **Select file** and select the file *KerasMLPExample_cos.XML*
  - In the drop-down menu, select **Convert to *.bml**
  - Click **Convert files**
  - The corresponding BML appears under *<TwinCATpath>\Functions\TF38xx-Machine-Learning \ConvertToolFiles*

Additional notes:

You can also convert several files at once by means of multi-selection.

The converted files are stored by default on the XAE system in the folder *<TwinCATpath>\Functions\TF38xx-Machine-Learning\ConvertToolFiles*.

Integration in TwinCAT via the PLC API

The procedure to load the ML description file into TwinCAT and to run it cyclically is described below. The PLC API [▸ 44] is dealt with first.

- First of all, create a TwinCAT project and a PLC project
- Add the PLC library Tc3_MLL under the References node

In the **Declaration**, please create an instance of FB_MllPrediction. In this simple case, the description file contains an MLP with one input and one output of the type FP32; accordingly, variables for input and output are created as REAL. A more generally accepted possibility to handle the inputs and outputs can be found in the Samples for PLC API [▸ 58].

In addition, create a string variable that contains the file name incl. path to the ML description file (path on the target system). Copy the corresponding file to this location on the target system (FTP, RDP, Shared Folder, …).
Further information on this step can be found here [▸ 30].

NOTE

Path of the description file on the target system

Pay attention to the settings of the File Writer and the writing rights on the target system.

```
PROGRAM MAIN
VAR
   fbPredict  :  FB_MllPrediction;       // ML Interface
   nInputDim, nOutputDim : UDINT := 1;
   fDataIn, fDataOut : REAL;
   sModelName    : T_MaxString  := 'C:/TwinCAT/3.1/Boot/ML_Boot/KerasMLPExample_cos.xml';
   hrErrorCode   : HRESULT;
   bLoadModel    : BOOL;
   nState        : INT := 0;
END_VAR
```

In the **Implementation part** you create a state machine, for example, which enables you to switch between the Idle state, Config state, Predict state and Error state.

PRELIMINARY

In the first state you initially wait for the command to load a description file. Subsequently, the Configure method [▸ 48] of the fbPredict function block is called until TRUE is returned. This is present for one cycle and means that the configuration has been completed. A check is then done to establish whether an error occurred. If no error occurred, the state is switched to the next state, which is the Predict state. The state machine remains in the Predict state as long as no error occurs or a (new) model is to be loaded.

```
CASE nState OF
    0: // idle state
      IF bLoadModel THEN
         bLoadModel := FALSE;
         nState := 10;
      END_IF
   10: // Config state
      fbPredict.stPredictionParameter.MlModelFilename := sModelName; // provide model path and name
      IF fbPredict.Configure() THEN // load model
         IF fbPredict.bError THEN
            nState := 999;
            hrErrorCode := fbPredict.hrErrorCode;
         ELSE // no error -> proceed to predict state
            nState := 20;
         END_IF
      END_IF
   20: // Predict state
      fbPredict.Predict(
         pDataInp := ADR(fDataIn),
         nDataInpDim := nInputDim,
         fmtDataInpType := ETcMllDataType.E_MLLDT_FP32_REAL,
         pDataOut := ADR(fDataOut),
         nDataOutDim := nOutputDim,
         fmtDataOutType := ETcMllDataType.E_MLLDT_FP32_REAL,
         nEngineId := 0,
         nConcurrencyId := 0);

      IF fbPredict.bError THEN // error handling
         nState := 999;
         hrErrorCode := fbPredict.hrErrorCode;
      ELSIF bLoadModel THEN // load (updated) model
         bLoadModel := FALSE;
         nState := 10;
      END_IF;
   999: // Error state
       // add error handling here
END_CASE
```

The Predict method [▸ 53] of fbPredict is used in the Predict state. This runs the loaded model. The method is informed of the input variables via the first three parameters of the method – pointer to a PLC variable, number of inputs and associated data type. The same is to be specified for the output variables (parameters 4 to 6). nEngineId and nConcurrencyId are not required in this simple example and are always transferred with the value zero. Details for these parameters can be found in the samples Detailed example [▸ 58] and Parallel, non-blocking access to an inference module [▸ 58].

Before activating the project on a target, you must select the TF3810 license **manually** on the Manage Licenses tab under System>License in the project tree, as you wish to load a multi-layer perceptron (MLP).

You can now activate the configuration. Log into the PLC and start the program. By setting the bLoadModel variable in the online view to TRUE, the model is now loaded and begins with the prediction. You can manipulate the input variable fDataIn and view the results in the output fDataOut. The multi-layer perceptron loaded approximately maps a cosine function in the input range of [-pi, pi] to the value range [-1,1]. Outside of the range [-pi, pi] the function increasingly diverges from the cosine function.

PRELIMINARY

TwinCAT_Project5.Untitled1.MAIN

Expression	Type	Value
fbPredict	FB_MllPrediction	
nInputDim	UDINT	1
nOutputDim	UDINT	1
fDataIn	REAL	3.14
fDataOut	REAL	-1.018022
sModelName	T_MaxString	'C:/TwinCAT/3.1...
hrErrorCode	HRESULT	16#00000000
bLoadModel	BOOL	FALSE
nState	INT	20

You can download the sample described above here [▸ 58].

Incorporation of a model by means of TcCOM object

This section deals with the execution of machine learning models by means of a prepared TcCOM object. A detailed description can be found here [▸ 42]. This interface offers a simple and clear way of loading models, executing them in real-time and generating appropriate links in your own application by means of the process image.

- Generate the prepared TcCOM object TcMachineLearningModelCycal. To do this, select the node **TcCOM Objects** with the right mouse button and select **Add New Item…**

Insert TcCom Object

Search: Name: Object1 (TcMachineLearningModelCycal) OK

Cancel

Type:

- Beckhoff Automation GmbH
  - NC Objects
  - Application Runtime
  - Machine Learning Model
    - TcMachineLearningModel [Module]
    - TcMachineLearningModelCycal [Module]
  - TcFilter
  - Analytics
  - External Time Provider
  - Iot
  - TcIoEth Modules

Multiple: 1

Insert Instance...

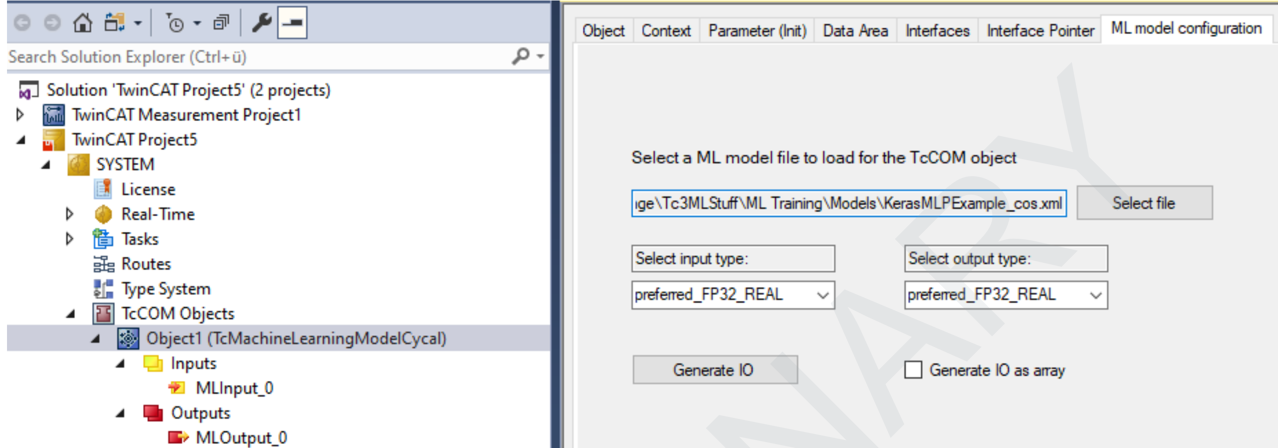
Reload

File: C:\TwinCAT\3.1\Config\Modules\TcMachineLearningModel.tmc

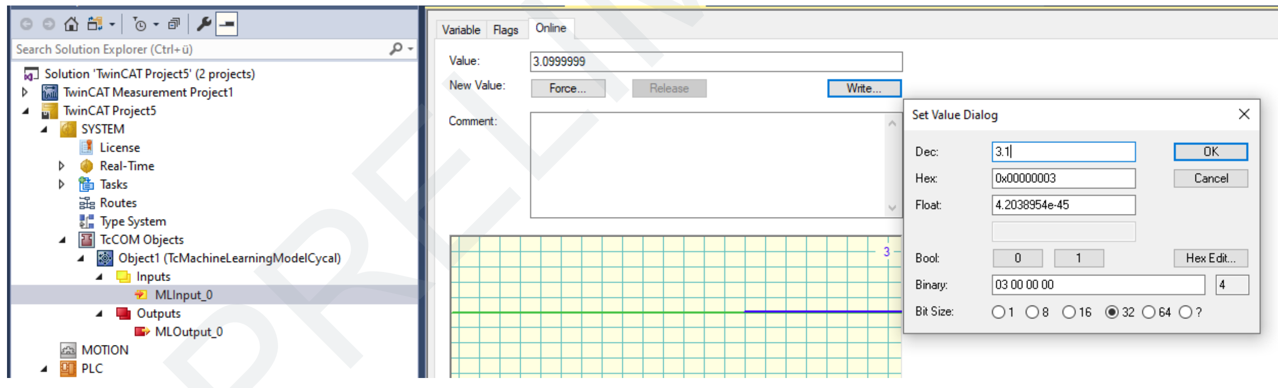
- Under Tasks, generate a new TwinCAT task and assign this task context to your newly generated instance of TcMachineLearningModelCycal. To do this, go to the **Context** tab of the generated object and select your generated task in the drop-down menu.

PRELIMINARY

- The instance of TcMachineLearningModelCycal has a tab called **ML model configuration**, where you can load the description file of the ML algorithm (XML or BML) and the available data types for the inputs and outputs of the selected model are then displayed. The file does not have to be on the target system. It can be selected from the development system and is then loaded to the target system on activating the configuration.
    - A distinction is made between preferred and supported data types. The only difference is that a conversion of the data type takes place at runtime if a non-preferred type is selected. This may lead to slight losses in performance when using non-preferred data types.
- The data types for inputs and outputs are initially set automatically to the preferred data types. The process image of the selected model is created by clicking **Generate IO**. Accordingly, by loading *KerasMLPExample_cos.xml*, you get a process image with an input of the type REAL and an output of the type REAL.



- Before activating the project on a target, you must select the TF3810 license manually on the Manage Licenses tab under System>License in the project tree, as you wish to load a multi-layer perceptron.
- Activate the configuration. You can now test the model by manually writing at the input.



PRELIMINARY

5 Machine Learning Models and file formats

5.1 Machine learning models supported

There are a great many different models in the area of machine learning. The Machine Learning Runtime supports a subset of the models, focusing on those that exhibit a deterministic runtime complexity. The TwinCAT license required also differs depending on the model type that is loaded into the Machine Learning Runtime.

Note that the TF3810 license contains the TF3800 license, which means that if the TF3810 license is valid, all models that require a TF3800 or TF3810 license can be loaded.

Supported models with a correspondingly necessary license are:

Model type	License	Available from version
Support vector machine [▶ 26] (SVM)	TF3800	Xxx
Multi-layer perceptron [▶ 24] (MLP)	TF3810	Xxx

Further model types are being added all the time, therefore an update to a new version is recommended.

5.1.1 Multi-layer perceptron

A multi-layer perceptron (MLP) is a mathematical model that is basically oriented to the structure of the human brain. The basic idea is the linking of the smallest units – so-called neurons – in a network. Each neuron takes up information from previous neurons or directly via model inputs and processes it. A directional flow of information takes place through this network from inputs to outputs.

A neuron processes its inputs **x** as a weighted sum plus an ordinate value and transforms the intermediate result with an activation function.

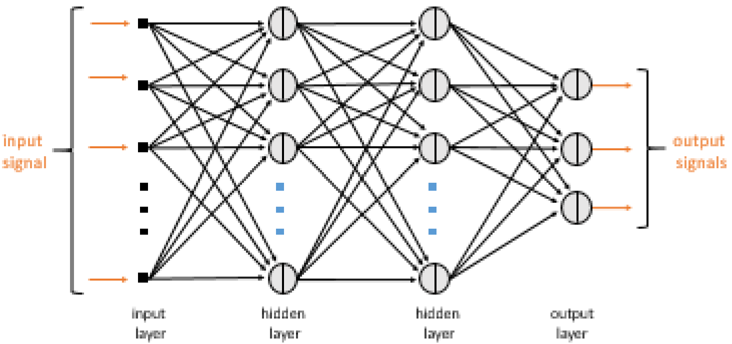
$$y = f_{\mathrm{act}}(\mathbf{wx} + w_0)$$

Neurons are usually arranged in layers, which are then linked one after the other. If a network has more than one layer between inputs and outputs, then it is referred to as a multi-layer perceptron.

The structure is illustrated in the figure below.

- **Input Layer:** Has no neurons of its own. Serves as an input layer and defines the number and data type of the inputs.
- **Hidden Layer:** Layer with its own neurons. The layer is characterized by the number of neurons as well as the selected activation function. Any number of hidden layers can be layered one after the other.
- **Output Layer:** Layer with its own neurons. The number of neurons and their activation function are oriented to the application to be implemented.

PRELIMINARY



In order to be able to fulfill a regression or classification task, it is necessary to adapt the model parameters (the neuron weights) to a data set. This process is called training and can be carried out in relevant frameworks, see Workflow [▶ 8].

Properties supported in the Machine Learning Runtime

Activation functions supported

Activation function	Description
tanh	Hyperbolic tangent (-1,1)
sigmoid	Sigmoid function – an exponential function (0,1)
softmax	Softmax – a normalized exponential function – often used for classification (0,1)
sine	Sine function (-1,1)
cosine	Cosine function (-1,1)
relu	"Rectifier" – positive portion is linear – good learning properties in case of deep networks (0, inf)
abs	Absolute value of the input (0, inf)
linear/id	Linear identity – simple linear function f(x) = x (-inf, inf)
exp	A simple exponential function e^(x) (0, inf)
logsoftmax	Logarithm of softmax – sometimes more efficient than softmax in the calculation (-inf, inf)
sign	Sign function (-1,1)
softplus	Sometimes better than relu due to the differentiability (0, inf)
softsign	Conditionally better convergence behavior than tanh (-1,1)

Calculation engines supported

Description	Identification in XML	Input data type (preferred)	Output data type (preferred)
32-bit Floating Point Engine	mlp_fp32_engine	E_MLLDT_FP32-REAL	E_MLLDT_FP32-REAL
64-bit Floating Point Engine	mlp_fp64_engine	E_MLLDT_FP64-LREAL	E_MLLDT_FP64-LREAL

Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the engine type. When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the "supported datatypes" can be found in ETcMllDataType [▶ 44].

Further comments

PRELIMINARY

There are no limits on the software side with regard to the number of layers or the number of neurons. With regard to the calculation duration and memory requirement, however, the limits of the hardware used are to be observed.

5.1.2 Support vector machine

A support vector machine (SVM) can be used both for classification and for regression. The SVM is a frequently used tool in particular with regard to classification tasks.

The fundamental goal of an SVM is to find a hyperplane in an N-dimensional space, wherein the distance between the closest data point and the plane is maximized. A hyperplane can only separate the space linearly. A non-linear separation is also possible by means of a so-called kernel trick. The N-dimensional space is transformed into a higher-dimensional space here. A linear separation with a hyperplane is possible in an accordingly higher-dimensional space. Various kernel functions are used for the kernel trick. The kernel function is to be specified by the user.

If a distinction needs to be made between several classes, several support vector machines are generated internally and classification takes place by means of comparisons.

Properties supported in the Machine Learning Runtime

Kernel functions supported

Kernel function	Description
Linear	$K(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{x}_1^T \mathbf{x}_2$
Radial Basis Function (RBF)	$K(\mathbf{x}_1 \mathbf{x}_2) = \exp(-\gamma \left\|\mathbf{x}_1 - \mathbf{x}_2\right\|^2)$
Sigmoid	$K(\mathbf{x}_1, \mathbf{x}_2) = \tanh(a\mathbf{x}_1^T \mathbf{x}_2 + b)$
Polynomial	$K(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^T \mathbf{x}_2 + 1)^d$

Calculation engines supported

Description	Identification in XML	Input type (preferred)	Output type (preferred)
64-bit Floating Point Engine	Svm_fb64_engine	E_MLLDT_FP64_LREAL	E_MLLDT_FP64_LREAL

Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the engine type. When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the "supported datatypes" can be found in ETcMllDataType [▸ 44].

Further comments

Currently the XML Exporter [▸ 40] is available for the export of learned SVM models.

5.2 Supported file formats

The learning process of a Machine Learning Model takes place outside of the real-time, usually in a script language such as Python, R or MATLAB®. The learned model is to be exported out of the selected ML Framework in order to use it in the ML Runtime within TwinCAT 3.

PRELIMINARY

Beckhoff supports various data formats for this method.

- Open Neural Network Exchange Format (ONNX) [▸ 27]
- Beckhoff ML XML [▸ 28]
- Beckhoff ML BML [▸ 30]

Beckhoff's proprietary formats in XML and BML form are directly readable from the Machine Learning Runtime. The ONNX data format is to be converted via the Machine Learning Model Manager [▸ 36] into a Beckhoff proprietary format.

Whereas ONNX and XML are openly visible formats, BML is a binary format and thus characterized above all by a small file size and an efficient loading behavior (execution time of the Configure method) in the XAR.

5.2.1 Open Neural Network Exchange (ONNX)

ONNX is an open file format for the representation of Machine Learning Models and is managed as a community project. Homepage of the ONNX community: onnx.ai

The ONNX format defines groups of operators in a standardized format, allowing learned models to be used interoperably with various frameworks, runtimes and further tools. Beckhoff supports ONNX via the Machine Learning Model Manager [▸ 36], which can read the format and convert it to the Beckhoff proprietary formats XML and BML.

The ONNX support is currently limited to TF3810 TC3 Neural Network Inference Engine.

Through support for ONNX, Beckhoff integrates the TwinCAT Machine Learning products in an open manner and thus guarantees flexible workflows. Supported tools of the ONNX community can be viewed here: onnx.ai/supported-tools.

Including, for example, the **frameworks**

- PyTorch
- Keras / TensorFlow
- MXNet
- Scikit-Learn
- …

General **converters**

- MMdnn (https://github.com/microsoft/MMdnn)
- ONNXMLTools (https://github.com/onnx/onnxmltools)
- …

Graph **optimizer**

- ONNX optimizer (https://github.com/onnx/onnx/blob/master/docs/Optimizer.md)

Graph **visualizer**

- Netron (https://github.com/lutzroeder/Netron)
- …

Examples of ONNX export

PyTorch

```
Import torch
import numpy as np

net = torch.nn.Sequential(
      torch.nn.Linear(1,1),
      torch.nn.ReLU()
      )
```

PRELIMINARY

```
dummy_input = torch.FloatTensor(np.random.random((1,1)))
print(net(dummy_input))

onnx_file = 'pytorch-relu.onnx'
torch.onnx.export(net,dummy_input,onnx_file, verbose=True)
```

Keras

```
import tensorflow as tf
import numpy as np

dummy_input = np.random.random((1,1))
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(1,input_shape=(1,), activation=tf.keras.activations.relu))
model.build()
model.summary()

print(model.predict(dummy_input))

path = '.'
import keras2onnx
import onnx
name='tf-keras-relu.onnx'
onnx_model = keras2onnx.convert_keras(model)
onnx.save_model(onnx_model,name)
```

SciKit Learn

```
import sklearn.neural_network as skl
import numpy as np

dummy_input = np.random.random((1,1))
dummy_output = np.random.random((1,1))

model = skl.MLPRegressor(hidden_layer_sizes =(1), activation ='relu')
model.fit(dummy_input,np.ravel(dummy_output))

filename = 'skl-relu'
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
initial_type = [('float_input',FloatTensorType([1,1]))]

onx = convert_sklearn(model,initial_types=initial_type)
with open(filename+'.onnx','wb') as f:
    f.write(onx.SerializeToString())
```

PRELIMINARY

5.2.2 Beckhoff ML XML

The Beckhoff-specific XML format for the representation of trained Machine Learning Models forms a core component of the product. The file can be generated via the TC3 Machine Learning Model Manager [▶ 36] (conversion of an ONNX file) or directly with an XML Exporter [▶ 40].

As opposed to ONNX, the XML-based description file can map TwinCAT-specific properties. The XML guarantees an extended functional scope of the TwinCAT Machine Learning product – see for example the concept of the Multi-engines [▶ 30]. On the other hand, it ensures seamless cooperation between the creator and user of the description file - compare Input and output transformations [▶ 29] and Custom attributes [▶ 29].

Essential areas of the Beckhoff ML XML are described below. This helps you to understand the functions it provides. The generation and editing of the file should only take place via the TC3 Machine Learning Model Manager and the XML Exporter.

XML Tag <MachineLearningModel>

Obligatory tag with 2 obligatory attributes. The tag is generated automatically and may not be manipulated.

Example:

```
<MachineLearningModel modelName="Support_Vector_Machine" defaultEngine="svm_fp64_engine">
```

The attribute `modelName` can be read in the PLC via the method GetModelName [▶ 52]. The model type that is to be loaded is identified by the model name. The attribute can, for example, assume the values `support_vector_machine` or `mlp_neural_network`.

XML Tag <CustomAttributes>

The tag CustomAttributes is optional and may be freely used by the user. The depth of the tree and the number of attributes are not limited. Creation can take place via the TC3 Machine Learning Model Manager. The XML can also be manually edited in this area.

Attributes can be read in the PLC via the methods GetCustomAttribute_array [▶ 48], GetCustomAttribute_fp64 [▶ 49], GetCustomAttribute_int64 [▶ 50] and GetCustomAttribute_str [▶ 50]. In the XML the typification is given by the prefixes str_, int64_, fp64_ and so on.

Example:

```
<CustomAttributes>
   <Model str_Name="TempEstimator" str_Version="1.2.11.0" />
   <MetaInfo arrfp64_InputRange="0.10000000000000001,0.90000000000000002" int64_TheAnswer="42" />
</CustomAttributes>
```

Here, a model with the name "TempEstimator" is created in the version 1.2.11.0. Thus, an array and an integer value are provided as further information. Sample code for reading the CustomAttributes can be downloaded from the Samples [▶ 58] section.

XML Tag <AuxilliarySpecifications>

The AuxilliarySpecifications area is optional and is subdivided into the children <PTI> and <IOModification**>.**

Example:

```
<AuxiliarySpecifications>
   <PTI str_producer="Beckhoff MLlib Keras Exporter" str_producerVersion="3.0.200525.0 str_requiredV
ersion="3.0.200517.0d"/>
   <IOModification>
      <OutputTransformation str_type="SCALED_OFFSET" fp64_offsets="0.48288949404162623" fp64_scaling
s="1.4183887951105305"/>
   </IOModification>
</AuxiliarySpecifications>
```

<PTI>

PTI stands for "Product Version and Target Version Information". The tool with which the XML was created (XML Exporter or Machine Learning Model Manager) and version of the tool at the time of the XML generation are specified here.

A minimum version of the executive ML Runtime can also be specified via the attribute str_requiredVersion. The query is regarded as passed if the attribute is not set. If the attribute is set, the query is regarded as passed if the ML Runtime Version is higher than or equal to the required version. If the query is not passed, i.e. if the version of the ML Runtime used is lower than the required version, then a warning is displayed when executing the Configure method.

<IOModification>

If inputs or outputs of the learned model are scaled in the training environment, the scaling parameters used can be integrated directly in the XML file so that TwinCAT automatically performs the scaling in the ML Runtime. The scaling takes place by means of y = x * Scaling + Offset.

XML Tag <Configuration>

The obligatory area *Configuration* describes the structure of the loaded model.

Example - SVM

```
<Configuration str_operationType="SVM_TYPE_NU_REGRESSION" fp64_cost="0.1" fp64_nu="0.3" str_kernelFu
nction="KERNEL_FN_RBF" fp64_gamma="1.0" int64_numInputAttributes="1"/>
```

Example - MLP

PRELIMINARY

```
<Configuration int_numInputNeurons="1" int_numLayers="2" bool_usesBias="true">
   <MlpLayer1 int_numNeurons="3" str_activationFunction="ACT_FN_TANH"/>
   <MlpLayer2 int_numNeurons="1" str_activationFunction="ACT_FN_IDENTITY"/>
</Configuration>
```

A configuration exists once only and is generated automatically.

XML Tag <Parameters>

The obligatory area Parameters substantiates the loaded model with the described <Configuration>. The learned parameters of the model are stored here, e.g. the weights of the neurons.

In the standard case, i.e. a learned model is described in an XML, the <Parameters> tag exists only once in the XML.

```
<Parameters str_engine="mlp_fp32_engine" int_numLayers="2" bool_usesBias="true">
```

Several models with identical <Configuration> can be merged via the Machine Learning Model Manager so that both models are described in a single XML. Distinction can then be made between the parameter sets by *Engines*, which is specified as an attribute for each parameter tag.

Example:

```
<Parameters str_engine="mlp_fp32_engine::merge0" int64_numLayers="2" bool_usesBias="true">
   …
</Parameters>
<Parameters str_engine="mlp_fp32_engine::merge1" int64_numLayers="2" bool_usesBias="true">
   …
</Parameters>
<IODistributor str_distributor="multi_engine_io_distributor::mlp_fp32_engine-
merge" str_engine_type="mlp_fp32_engine" int64_engine_count="2">
   <Engine0 str_engine_name="merge0" str_reference="sin_engine" />
   <Engine1 str_engine_name="merge1" str_reference="cos_engine" />
</IODistributor>
```

Two MLPs with an identical *Configuration* were merged here. The first engine bears the ID 0 and the internal name "mlp_fp32_engine::merge0" and can be addressed by the user via the reference "sin_engine". The second engine bears the ID 1 and the internal name "mlp_fp32_engine::merge1" and the reference "cos_engine".

The ID of the engine is sequentially incremented by the value one, starting from zero. The reference is a string that can be specified in the Model Manager during *Merge*.

If several engines are merged in an XML, all engines are loaded in the ML Runtime and are available for inference. The Predict method [▶ 53] is to be transferred when calling the engine ID that is to be used. The reference for the engine can be transferred via the PredictRef method [▶ 55]. A GetEngineIdFromRef method [▶ 51] is also available for finding the associated ID from the reference. Switching between the engines is possible without latency.

There is an example of the use of multi-engines in the PLC in the Samples area.

5.2.3 Beckhoff ML BML

The BML format is a binary representation of the XML-based ML description file. As a result, the format is not openly visible and the file size is smaller in comparison with ONNX and XML.

A BML file can be generated via the TC3 Machine Learning Model Manager [▶ 36] from an XML or an ONNX file. The way back from a BML file to an XML file is **not** provided for.

5.3 File management of the ML description files

File management on the Engineering PC (XAE)

The Machine Learning Model Manager [▶ 36] serves as a central tool for the processing and conversion of Machine Learning models. If a file is loaded and edited, the resulting file is saved in the folders

- \Functions\TF38xx-Machine-Learning\ConvertToolFiles

PRELIMINARY

- \Functions\TF38xx-Machine-Learning\ExtractToolFiles
- \Functions\TF38xx-Machine-Learning\MergeToolFiles

depending on the action carried out.

If a BML or XML description file is used in a TwinCAT solution, a distinction must be made between TcCOM API [▸ 42] and the PLC API [▸ 44] with regard to the file management.

TcCOM API

After the integration of a description file in the TcCOM TcMachineLearningModelCycal, the corresponding description file is copied into the Visual Studio project directory and is thus part of the project: <VS Project>_MLInstall.

On activating the configuration, the file is copied from the Visual Studio project directory into the boot folder on the target system: \TwinCAT\3.1\Boot\ML_Boot.

PLC API

When using the PLC API, the file name and path of the Machine Learning Model file are specified in the PLC code as `T_MaxString` – accordingly, the user must ensure that a corresponding file exists coming from the target system. This means that the description file does not become part of the Visual Studio project directory, nor is it transferred automatically to the target system.

Transfer of the ML description files to the target system on activating the configuration

TcCOM API

When using the TcCOM object **TcMachineLearningModelCycal**, the ML description file is transferred automatically from the XAE system to the XAR system.

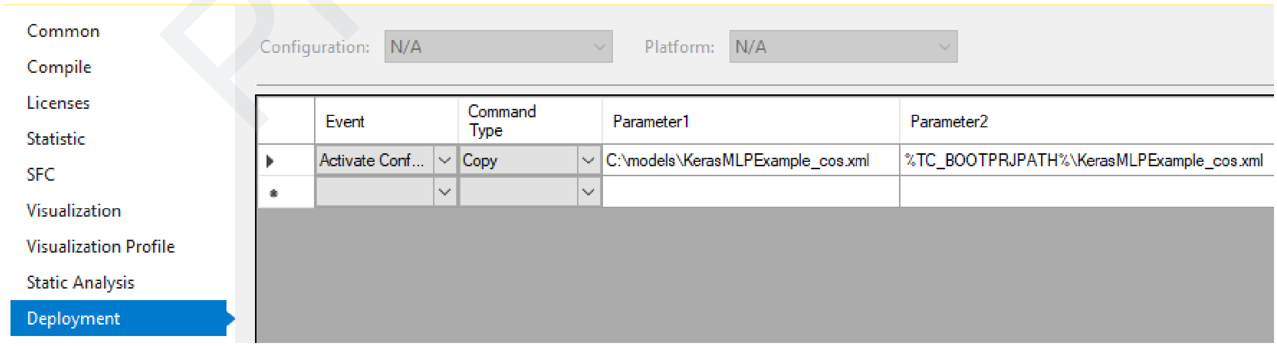
The file is transferred from the Visual Studio project folder <VS Project>_MLInstall to the Boot folder TwinCAT\3.1\Boot\ML_Boot on the XAR.

PLC API

If the **PLC API** is used, the user is responsible for the transfer of the ML description file. As a result, the flexibility of the application is increased on the one hand, while corresponding steps have to be implemented by the user on the other.

The ML description file can be transferred to the target system in many ways. One of them is named below by way of example.

- Via the properties of the PLC project under "Deployment" it is possible to specify which files are to be transferred to the target system in the case of a certain event, for example the activation of the configuration.



NOTE

Writing rights on the target system

The writing rights on the operating system side and the Write Filter settings must be observed.

Updating ML description files in the field

PRELIMINARY

An important scenario in machine learning is the updating of data-based algorithms in the field during the running time of a machine. Here too, distinction must be made between use of the TcCOM [▸ 42] API and use of the PLC API [▸ 44].

TcCOM API

In the case of the TcCOM API, the update behavior is the same as with other changes in the TcCOM area. An XAE system is necessary with an ADS route to the target system. In the XAE, a new ML description file can be integrated in the TwinCAT project. The new project is then transferred to the target system by activating the configuration. Therefore, it is necessary to restart the TwinCAT runtime here.

PLC API

If the PLC API is used, it is possible to update the ML description file on the target system without restarting the TwinCAT runtime. To do this it is merely necessary to update the ML description file on the target system and to retrigger the Configure method.

Below, scenarios and important elements are described that need to be considered when carrying out a remote update of the ML description file.

Elements to be observed

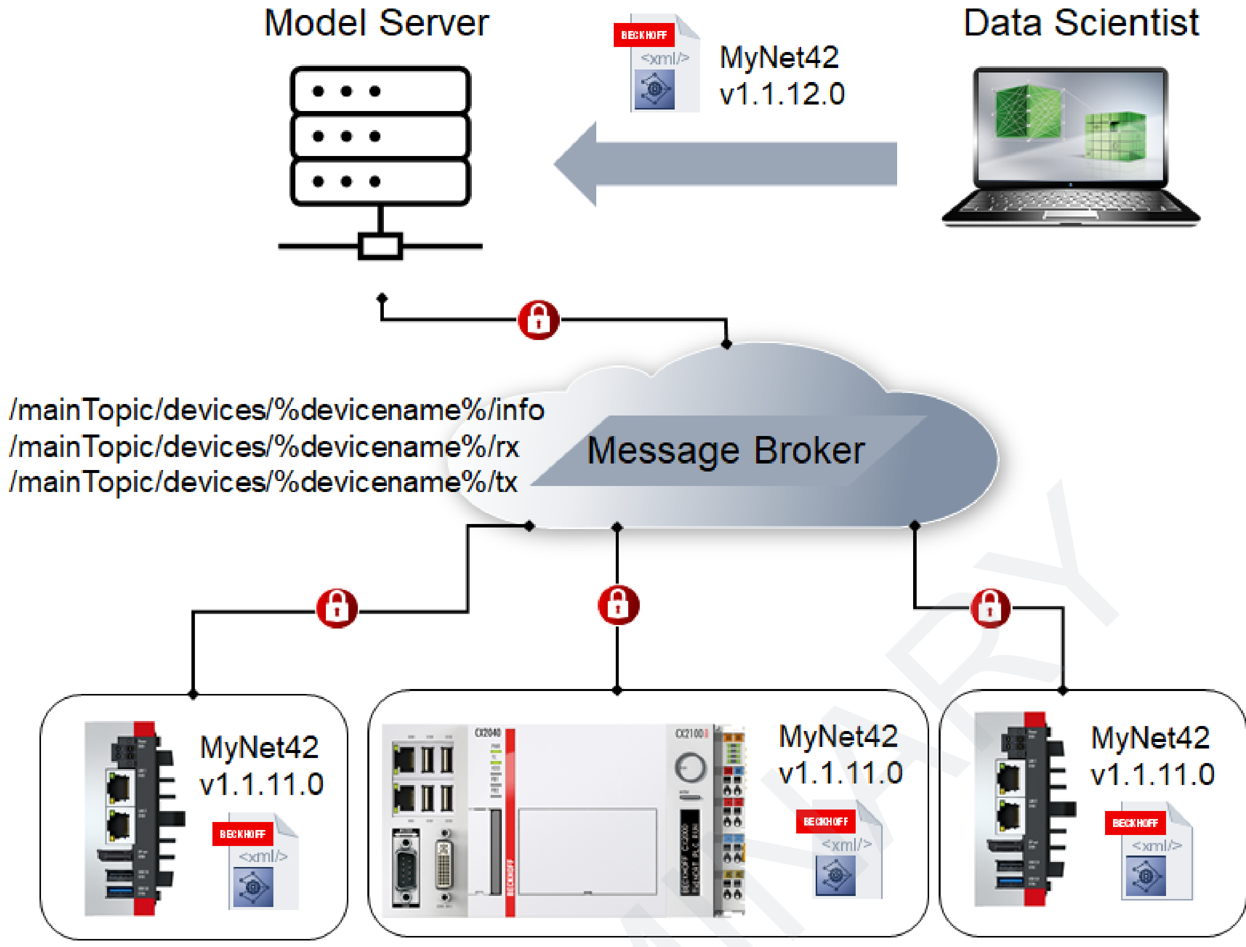
- The XML or BML file should bear a model version and a suitable model name (see `GetCustomAttribute` methods)
- Different protocols can be used when transferring the description file
- Observe the communication direction for firewall settings
- Push of a model to the IPC or pull request coming from the IPC?

It should also be mentioned that in all three scenarios an implementation is possible with a TwinCAT 3 function; however, an application-specific user mode application, e.g. on a .NET basis, can also be realized. The advantage of an implementation using a TwinCAT 3 function could be that a direct interaction of the MQTT, HTTPS or OPC UA client can take place in the PLC and can thus directly interact with the state machine of `FB_MllPrediction`.

Update via MQTT

The MQTT protocol can represent a flexible and firewall-friendly way of managing a network (of a changeable number) of devices. The message broker decouples the devices from one another and also ensures that the technical communication direction is always coming from the local networks.

PRELIMINARY

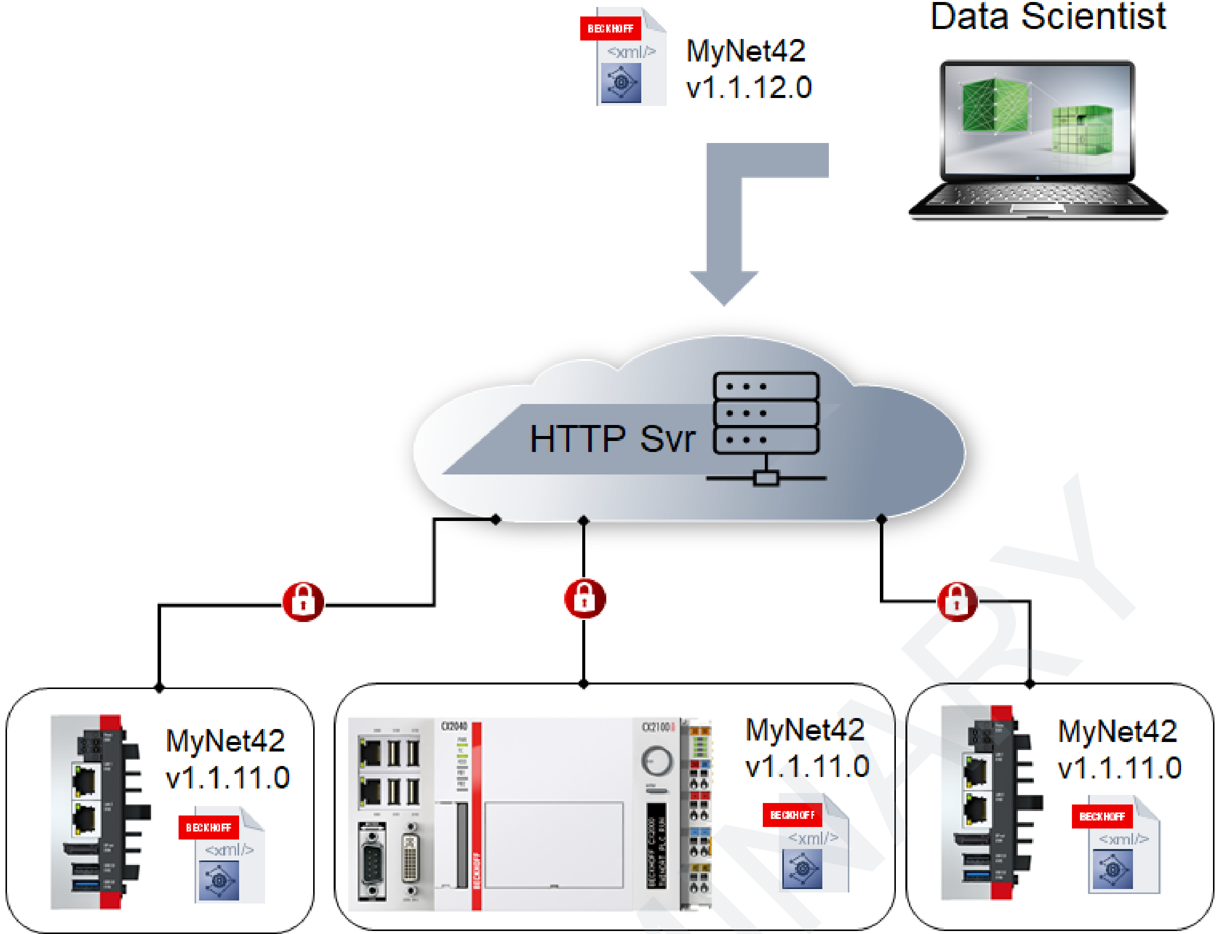


The message broker manages the messages of the devices and the model server. Indicated in the illustration here as an example is a topic structure that addresses the *device name* and then an "info" area for the current status of the device or the loaded ML model. This information can be read by the model server and, if necessary, an ML model with a more up-to-date version can be sent to the "rx" topic, which is then received by the device. An acknowledgement can take place via the "tx" topic.
In this scenario both "pull" and "push" can be implemented with a technically outgoing communication direction.

Update via HTTPS

An alternative, for example, is the widely used HTTPS protocol, which is often unproblematic with regard to open ports. The model server is implemented here directly as an HTTP(S) server.

PRELIMINARY

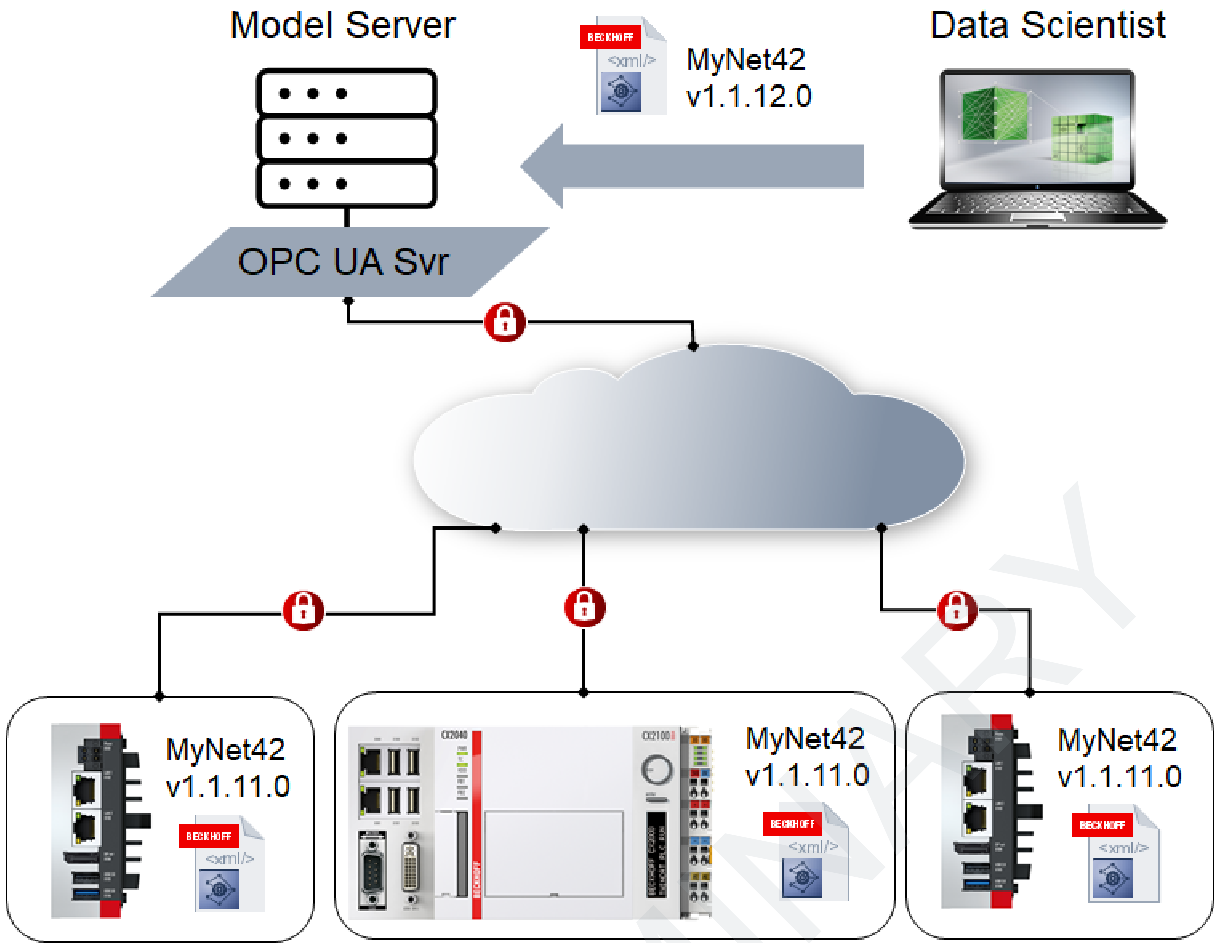


An important difference between this and the MQTT scenario is that a "push" to devices cannot be executed directly starting from the model server unless an incoming HTTPS port is opened. In order to get around this, a device can, for example, request updates cyclically. As with MQTT a payload is not defined, therefore both status information and the model file itself can be transmitted by HTTPS.

Update via OPC UA

OPC UA is usually more widely used in automation technology than MQTT and HTTPS. Here too, a scenario can be set up due to the possibility to transmit files via OPC UA.

PRELIMINARY



In this case the OPC UA server is implemented on the Model Server side. Appropriate software products for this are available on the market. The OPC UA client can be implemented on the device side and, as in the HTTPS scenario, send requests to the server.

PRELIMINARY

6 Configuration

6.1 TC3 Machine Learning Model Manager

The TC3 Machine Learning Model Manager is the central tool for the editing of ML model description files. The tool is integrated in Visual Studio and can be opened via the menu bar under **TwinCAT > Machine Learning**.

The central tasks of the TC3 Machine Learning Model Manager are:

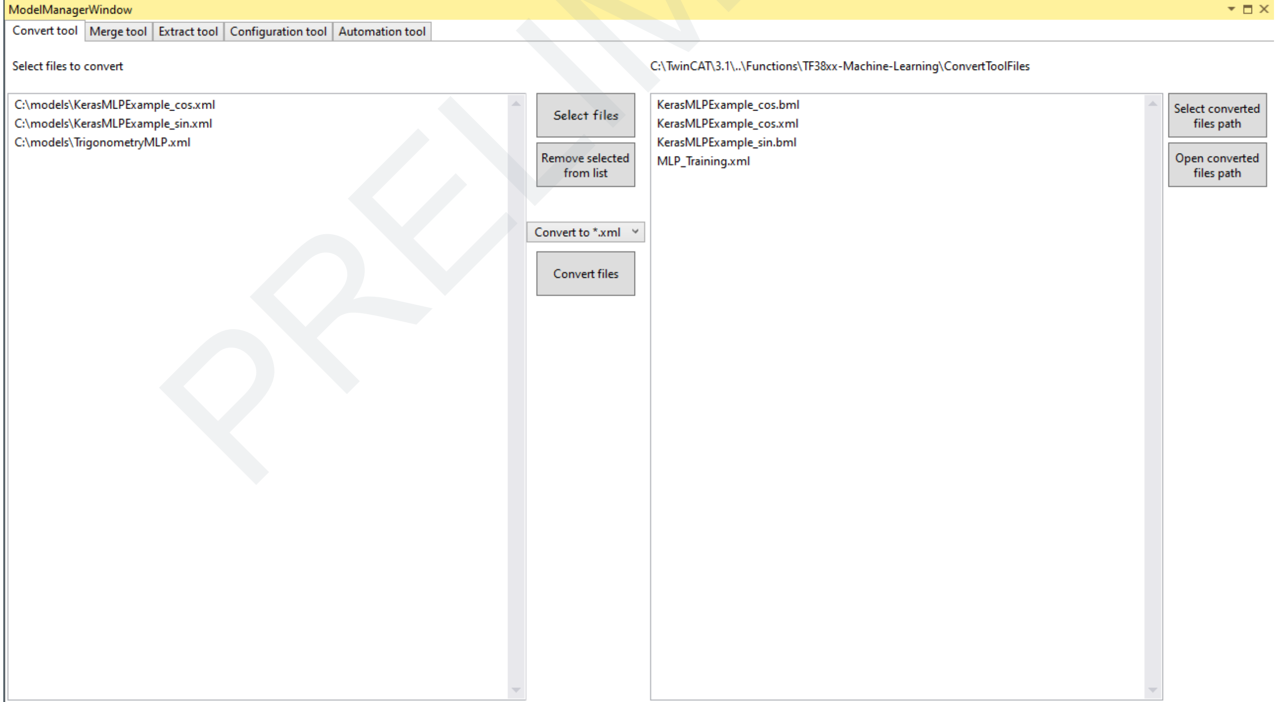
- Conversion of ML model description files (Convert tool)
- Processing of ML engines (Merge tool and Extract tool)
- Processing of model information (Configuration tool)

Conversion of ML model files

A conversion tool for ML model description files is located on the **Convert tool** tab. XML [▶ 28] and ONNX [▶ 27] files can be selected and converted to XML or BML [▶ 30] format.

NOTE	
	Conversion of Beckhoff BML back to XML is not provided for The objective of Beckhoff BML is to represent the content as a not freely readable binary file. Therefore, the conversion process from Beckhoff BML to Beckhoff XML is not provided for.

The File Browser is opened via **Select files** and ML model description files can be selected (multi-selection is possible by Ctrl + click). Selected ML model files are listed on the left-hand side with their path and file name. Files can be removed from the list again with **Remove selected from list**.



Listed ML model description files can be selected in the left-hand list (multi-selection is possible with Ctrl + click here too) and converted with **Convert files** into the format selected in the drop-down menu. The converted files are saved in the *converted file path*. Default is the path *<TwinCATPath>\Functions\TF38xx-Machine-Learning\ConvertToolFiles*. The *converted file path* can be opened in the File Browser by clicking **Open converted file path**. The path can be changed with **Select converted files path**.

PRELIMINARY

Creating a multi-engine description

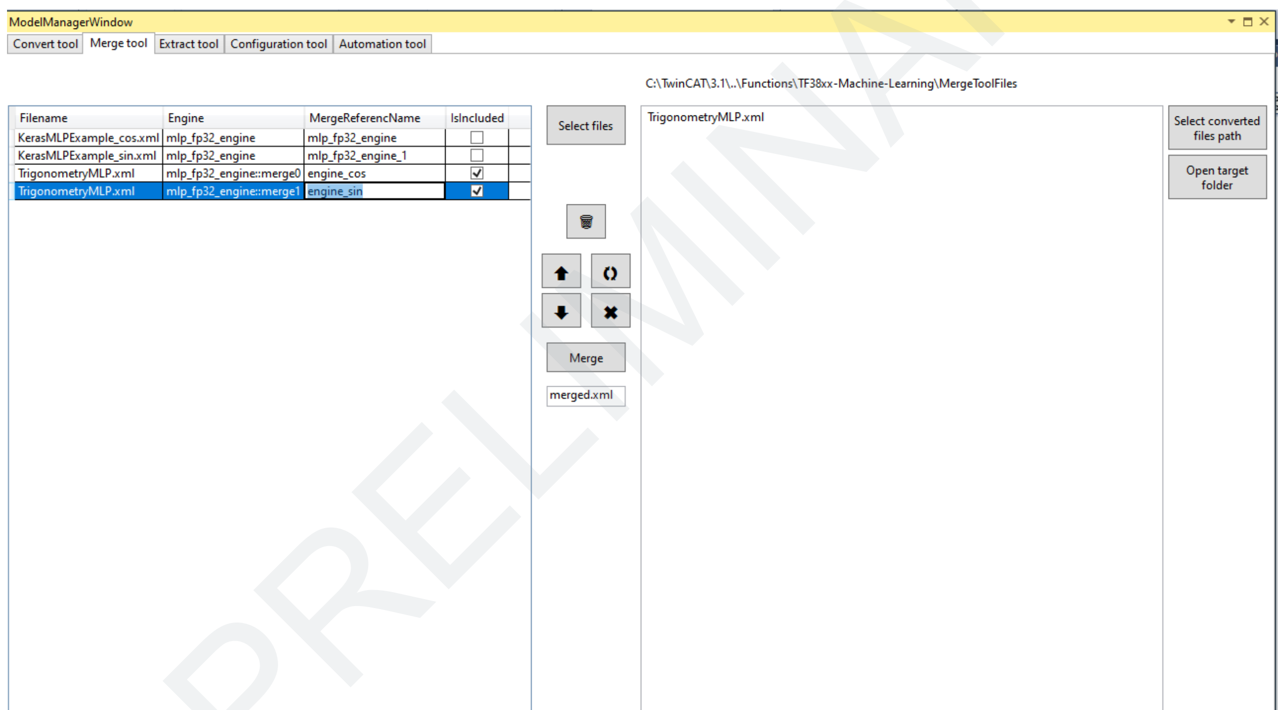
It is possible via the **Merge tool** tab to generate an ML model description file containing more than one engine. Compare Beckhoff ML XML [▶ 30]. The condition for merging several engines is that the model structure - the area <Configuration> [▶ 29] in Beckhoff XML - is identical for all engines.

To this end, several ML model description files must be loaded with **Select files**. The entries are then visible, engine-based, in the list on the left-hand side. If several engines already exist in a description file, they are listed individually in the list.

The field **MergeReferenceName** is freely editable. A reference name for the selected engine can be entered here so that this engine is addressable in the PLC via this reference, cf. PredictRef [▶ 55] and GetEngineIdFromRef [▶ 51]. If the engine ID is used in the PLC instead of the reference name, the rule is that the uppermost engine in the list bears the ID = 0 and those that follow it accordingly 1,2,3 and so on.

Using the **IsIncluded** checkbox you can specify whether or not the selected engine should be included in the merged description file.

The position of the engines in the list can be manipulated in the central section of the user interface. Selected engines can be moved up or down in the list (up and down arrow). Any number of engines can be selected at the same time. Also, using the cross symbol, any number of engines can be manipulated at the same time with regard to the **IsIncluded** checkbox. If two engines are selected, their places can be swapped using the round double-arrow symbol. Engines can be removed from the list using the recycle bin symbol.

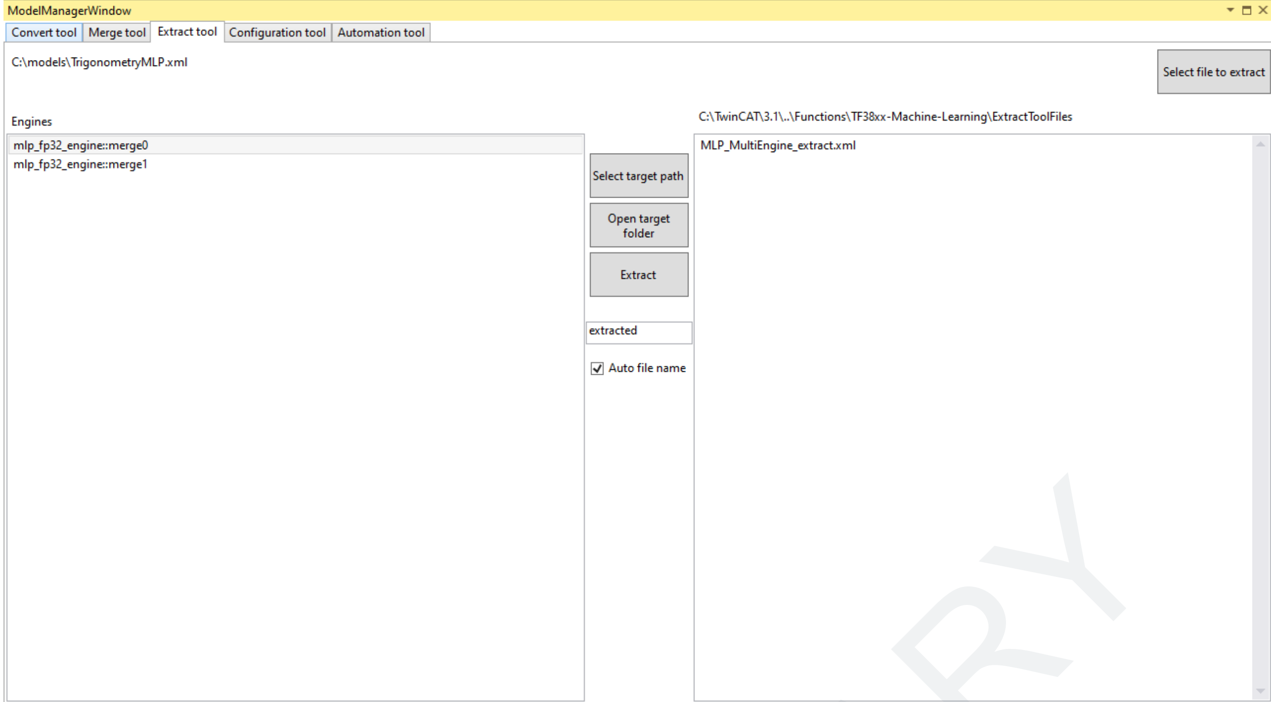


The name of the merged ML description file can be entered in the textbox in the central section. The file is generated with **Merge** and saved in the file path *<TwinCATPath>\Functions\TF38xx-Machine-Learning \MergeToolFiles*. The path can be changed with **Select target path**.

Extracting multi-engine descriptions

Using the Extract tool it is possible to separate merged description files again. An ML model description file can be loaded using **Select file to extract**. All engines that it contains appear in the list on the left-hand side. If an engine is selected, it can be converted to a discrete ML description file using **Extract**. The name of the newly generated file is to be entered using the textbox. If the **Auto file name** checkbox is checked, the original file name is supplemented by the string in the textbox; if the checkbox is unchecked, only the string in the textbox is used as the new file name. The newly generated file is saved in the file path *<TwinCATPath>\Functions\TF38xx-Machine-Learning\ExtractToolFiles*. The path can be changed with **Select target path**.

PRELIMINARY



Creating metadata for the model

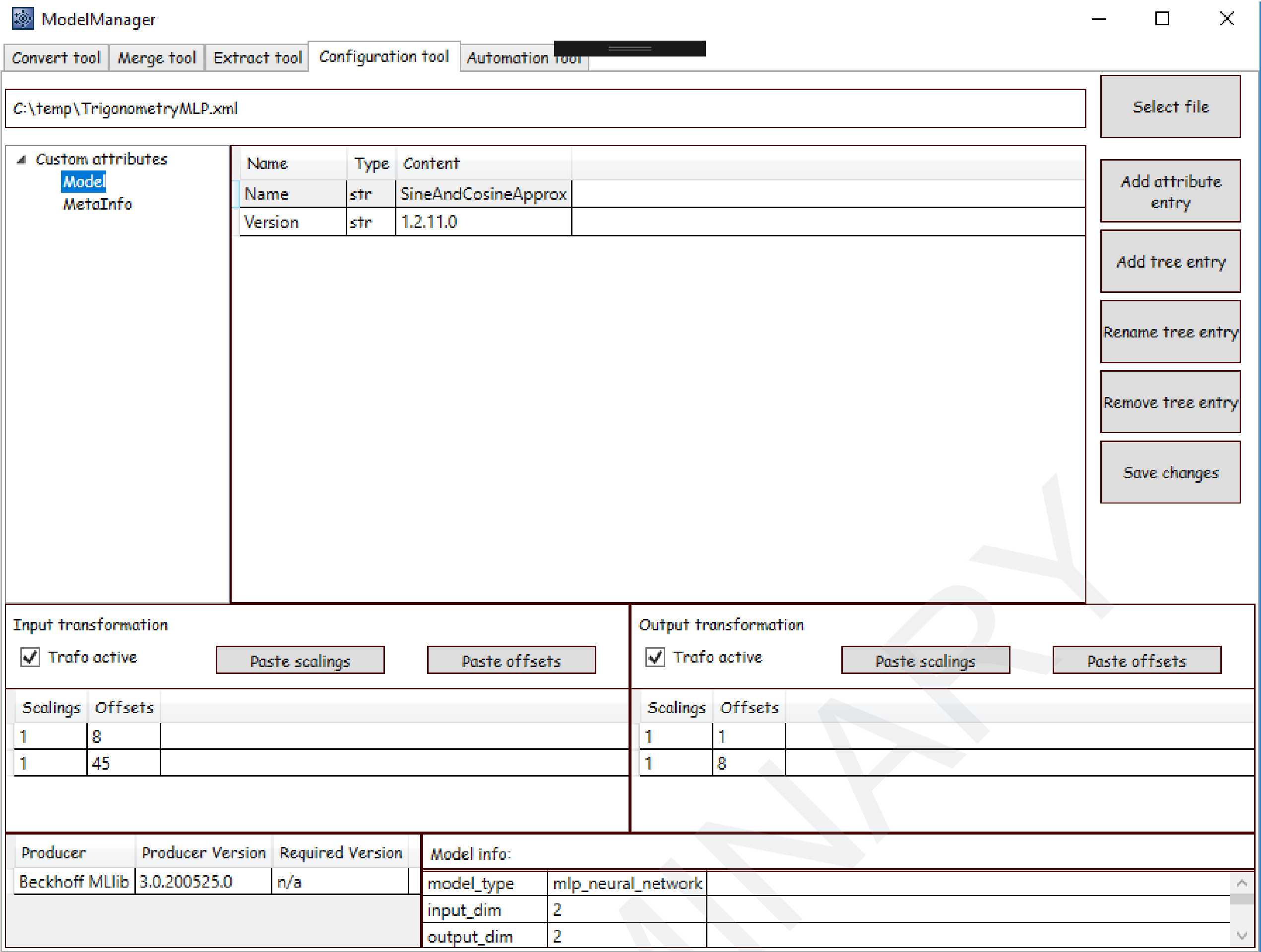
The configurator for

- Custom attributes [▶ 29]
- Product version and target version information [▶ 29]
- Input and output scaling [▶ 29]

is displayed on the **Configuration tool** tab.

An ML description file can be selected using **Select file** and then edited. After editing, the original file is overwritten using **Save changes**.

PRELIMINARY



The **Custom Attributes** are edited using the buttons

- **Add attribute entry**: Adds an attribute to the selected tree item. The tree item must be selected in the left-hand list.
  - If an attribute is created, then the name, the data type and the value/values must be specified.
  - The attributes are deleted by selecting the attribute and pressing the **Delete** button.
- **Add tree entry**: Adds a tree item under the selected tree item (as a subtree item).
- **Rename tree entry**: The selected tree item can be renamed.
- **Remove tree entry**: The selected tree item incl. the subtree items is deleted.

The editing area for **input and output transformations** can be enabled by checking the **Trafo active** checkbox. Depending on the number of inputs and outputs, a corresponding number of rows is offered, in each of which scaling and offset are to be entered, cf. Beckhoff ML XML [▸ 29]. A value from a list of numbers from the clipboard can be entered as an offset or scaling using the **Paste Scaling** and **Paste Offset** buttons. The number sequence can be separated by comma, semicolon or space. Only the number of numbers in the list must match the number of inputs or outputs respectively.

The **Producer and Target Version Information** is set automatically by the TC3 ML Model Manager. The Target Version is determined automatically on the basis of the feature set of the model description file used. If an older ML Runtime version is used to load this model file, a warning message appears when executing the Configure method.

The number of **inputs and outputs** of the model and the **model type** are displayed in the bottom right-hand part of the window. This cannot be edited and is only for information.

PRELIMINARY

6.2 XML Exporter

The XML Exporters provided by Beckhoff can be freely used and changed. They are open source and under MIT license. This offers customers the option of adapting the XML Exporter according to their needs, for example by adding company-specific or project-specific CustomAttributes, cf. Beckhoff ML XML [▸ 29].

Following the installation of the product, the XML Exporters are available in the folder *<TwinCATPath>\Functions\TF38xx-Machine-Learning\exporter*.

Direct XML export of an MLP

Only network architectures that exhibit a sequential structure in the following sense are supported: Each neuron in a layer is exclusively linked to each neuron in the following layer. It is possible to export layers without bias.

Export from Keras / Tensor Flow

- File: KerasMlp2Xml.py
- Sample call: ExampleKerasMLP (Resources/zip/8746685963.zip)
- Requirements for the Python environment:
  - Keras with TensorFlow backend (Tensor Flow Version 1.15.0)
  - Numpy (Version 1.17.4)
  - Matplotlib
- Activation functions supported: tanh, sigmoid, softmax, relu, linear/identity, exp, softplus, softsign
- Only **sequential models** can be exported with the XML Exporter, no **functional models**. The model is to be generated accordingly with

```
from tensorflow.keras.models import Sequential
model = Sequential()
```

- Dropout layers are supported (only relevant for training, ignored when exporting)
- Dense layers are supported. The activation functions must be transferred to the layer as an argument and not as a discrete activation layer.

```
from keras.layers import Activation, Dense
# this will not work !!!
model.add(Dense(64))
model.add(Activation('tanh'))
# this will work
model.add(Dense(64,activation='tanh'))
```

- The API is described in detail in the header of the file KerasMlp2Xml.py.
  `net2xml(net,output_scaling_bias=None,output_scaling_scal=None)`
  - `net`, obligatory, class of the trained model
  - `output_scaling_bias`, optional, list (in case of several features), otherwise float or int
  - `output_scaling_scal`, optional, list (in case of several features), otherwise float or int
  - A string document is returned that can be saved as an XML.

Export from MATLAB®

- File: MatlabMlp2Xml.m
- Sample call: ExampleMatlabMLP (Resources/zip/8746880267.zip)
- Requirements for the MATLAB® environment:
  - MATLAB®
  - Deep Learning Toolbox
- Activation functions supported: tanh, sigmoid, softmax, relu, linear/identity
- Supported models of the Deep Learning Toolbox: fitnet, patternnet
- ProcessFcns are supported by the types mapminmax and mapstd
- The use of ProcessFcns is optional
- If a ProcessFcn is used in the output layer, its activation function must be purelin

PRELIMINARY

- The API is described in detail in the header of the file MatlabMlp2Xml.m
  `MatlabMlp2Xml(net, fnstr, varargin)`
- net, obligatory, class of the trained model
- fnstr, obligatory, string with path and file name
- output_scaling_bias, optional, vector
- output_scaling_scal, optional, vector

Direct XML export of an SVM

Export from SciKit-Learn

- File: SciKitLearnSvm2Xml.py
- Sample call: ExampleScikitLearnSVM (Resources/zip/8746882571.zip)
- Requirements for the Python environment:
  - Python Interpreter: 3.6 or higher
  - SciKit-Learn: Version 0.22.0 or higher
  - Matplotlib
  - Numpy
- Only numerical class labels can be exported
- Supported classes or models: SVC, NuSVC, OneClassSVM, SVR, NuSVR
  - LinearSVR and LinearSVC are not supported by the Exporter, but can alternatively be implemented via the classes SVR and SVC, each with a linear kernel.
- Kernel functions supported: Linear, rbf, sigmoid, polynomial
  - Neither individual kernel functions nor precomputed functions are supported
- Remarks about model parameters
  - *Gamma* = scale is not supported
  - *Gamma* = auto_deprecated: gamma = 0.0 is exported
  - *Gamma* = auto: gamma = 1/n_features is exported.
  - *C* = inf is not supported
  - *decision_function_shape* = ovr is not supported. *decision_function_shape* = ovo must be used. Default in SciKit-Learn is ovr!
  - *break_ties* is ignored because *decision_function_shape* = ovr is not supported.
- The API is described in detail in the header of the file SciKitLearnSvm2Xml.py.
  `svm2xml(svm, input_scaling_bias=None, input_scaling_scal=None)`
  - `net`, obligatory, class of the trained model
  - `input_scaling_bias`, optional, list (in case of several features), otherwise float or int
  - `input_scaling_scal`, optional, list (in case of several features), otherwise float or int
  - A string document is returned that can be saved as an XML.

PRELIMINARY

7 API

Two ways of programming are available to the user in TwinCAT 3. Static TcCOM objects can be created and triggered via a cyclic task, or an instance can be created from and configured via the PLC.

The programming interface of the PLC offers far greater flexibility than the use of a TcCOM instance; conversely, the latter is very simple and in many cases adequate.

7.1 TcCOM

The use of a TcCOM for the inference of an ML model in TwinCAT provides a very simple possibility to execute trained models in the TwinCAT XAR. In principle, the entire procedure is documented in the quick start, so that the steps described there are initially repeated and a few further details are given below.

Incorporation of a model by means of TcCOM object

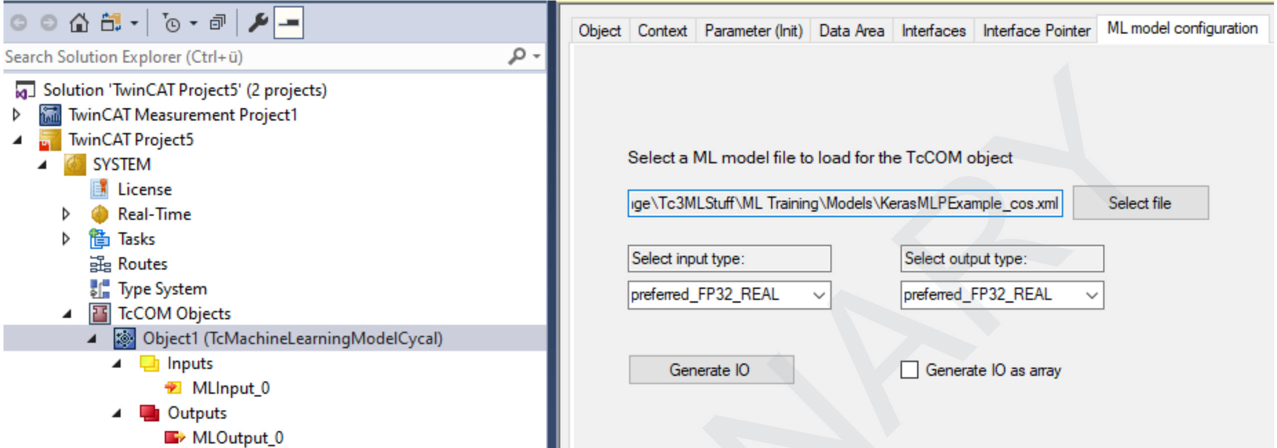
This section deals with the execution of machine learning models by means of a prepared TcCOM object. A detailed description can be found here [▸ 42]. This interface offers a simple and clear way of loading models, executing them in real-time and generating appropriate links in your own application by means of the process image.

- Generate the prepared TcCOM object TcMachineLearningModelCycal. To do this, select the node **TcCOM Objects** with the right mouse button and select **Add New Item…**

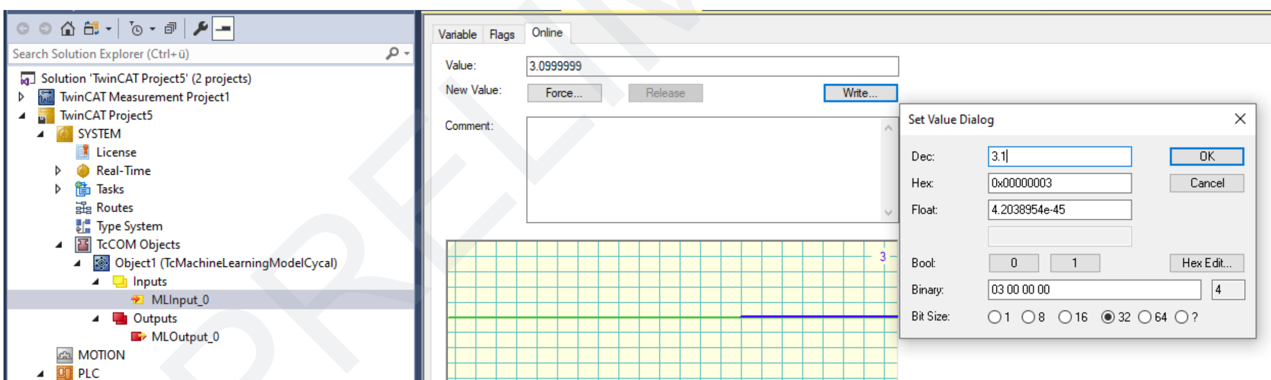
- Under Tasks, generate a new TwinCAT task and assign this task context to your newly generated instance of TcMachineLearningModelCycal. To do this, go to the **Context** tab of the generated object and select your generated task in the drop-down menu.

PRELIMINARY

- The instance of TcMachineLearningModelCycal has a tab called **ML model configuration**, where you can load the description file of the ML algorithm (XML or BML) and the available data types for the inputs and outputs of the selected model are then displayed. The file does not have to be on the target system. It can be selected from the development system and is then loaded to the target system on activating the configuration.
  - A distinction is made between preferred and supported data types. The only difference is that a conversion of the data type takes place at runtime if a non-preferred type is selected. This may lead to slight losses in performance when using non-preferred data types.
- The data types for inputs and outputs are initially set automatically to the preferred data types. The process image of the selected model is created by clicking **Generate IO**. Accordingly, by loading *KerasMLPExample_cos.xml*, you get a process image with an input of the type REAL and an output of the type REAL.



- Before activating the project on a target, you must select the TF3810 license manually on the Manage Licenses tab under System>License in the project tree, as you wish to load a multi-layer perceptron.
- Activate the configuration. You can now test the model by manually writing at the input.



If the process image is larger, i.e. many inputs or outputs exist, it may be helpful not to generate each input individually as a PDO, but to define an input or output as an array type. To do this, check the checkbox **Generate IO as array** and click **Generate IO**.

Models with several engines, cf. Beckhoff ML XML [▶ 30], can be loaded, but only EngineId = 0 is used. Switching between the EngineIds with the TcCOM API is not provided for.

The ML description file used is automatically transferred from the Engineering system to the Runtime system on activating the configuration. File management details are described in the section File management of the ML description files [▶ 30].

7.2 PLC API

7.2.1 Datatypes

7.2.1.1 ETcMllDataType

Syntax

Definition:

```
TYPE ETcMllDataType :
(
    E_MLLDT_UNDEFINED  := 0,
    E_MLLDT_INT8_SINT  := 10,
    E_MLLDT_INT16_INT  := 20,
    E_MLLDT_INT32_DINT := 30,
    E_MLLDT_INT64_LINT := 40,
    E_MLLDT_FP16       := 50,
    E_MLLDT_FP16B      := 55,
    E_MLLDT_FP32_REAL  := 60,
    E_MLLDT_FP64_LREAL := 70,
    E_MLLDT_SPECIAL    := 99
)BYTE;
END_TYPE
```

Values

Name	Description
E_MLLDT_UNDEFINED	invalid / undefined data type
E_MLLDT_INT8_SINT	8-bit signed integer number (SINT / char)
E_MLLDT_INT16_INT	16-bit signed integer number (INT / short)
E_MLLDT_INT32_DINT	32-bit signed integer number (DINT / long)
E_MLLDT_INT64_LINT	64-bit signed integer number (LINT / long long)
E_MLLDT_FP16	16-bit IEEE floating point number (future usage)
E_MLLDT_FP16B	16-bit "bfloat16" floating point number (future usage)
E_MLLDT_FP32_REAL	32-bit IEEE floating point number (REAL / float)
E_MLLDT_FP64_LREAL	64-bit IEEE floating point number (LREAL / double)
E_MLLDT_SPECIAL	Function-specific byte stream

PRELIMINARY

7.2.1.2 ST_MllPredictionParameters

Syntax

Definition:

```
TYPE ST_MllPredictionParameters :
STRUCT
    MlModelFilepath : STRING(255);
    MaxConcurrency  : UINT;
END_STRUCT
END_TYPE
```

Parameters

Name	Type	Default	Description
MlModelFilepath	STRING(255)		File path of the loaded model. Either *.xml or *.bml file
MaxConcurrency	UINT	1	Maxium number of threads calling predict on the FB in the same time.

7.2.2 Function Blocks

7.2.2.1 FB_MllPrediction

FB_MllPrediction	
stPredictionParameter *ST_MllPredictionParameters*	*BOOL* bBusy
eTraceLevel *TcEventSeverity*	*BOOL* bError
	I_TcMessage ipResultMessage
	HRESULT hrErrorCode

Syntax

Definition:

```
FUNCTION_BLOCK FB_MllPrediction
VAR_INPUT
    stPredictionParameter : ST_MllPredictionParameters;
    eTraceLevel           : TcEventSeverity;
END_VAR
VAR_OUTPUT
    bBusy                 : BOOL;
    bError                : BOOL;
    ipResultMessage       : I_TcMessage;
    hrErrorCode           : HRESULT;
END_VAR
```

Inputs

Name	Type	Description
stPredictionParameter	ST_MllPredictionParameters [▶ 44]	General Prediction parameters
eTraceLevel	TcEventSeverity	Eventlogger trace level. Default = Critical. Must be set before Configure method is called

Outputs

Name	Type	Description
bBusy	BOOL	True if a asynchronous action is taking place
bError	BOOL	Indicates error in method
ipResultMessage	I_TcMessage	Contains the last invoked error
hrErrorCode	HRESULT	Unique error code

PRELIMINARY

Methods

Name	Description
CheckPreferredIODataTypes [▶ 47]	
CheckSupportedIODataTypes [▶ 47]	
Configure [▶ 48]	
GetCustomAttribute_array [▶ 48]	
GetCustomAttribute_fp64 [▶ 49]	
GetCustomAttribute_int64 [▶ 50]	
GetCustomAttribute_str [▶ 50]	
GetEngineIdFromRef [▶ 51]	
GetInputDim [▶ 51]	
GetMaxConcurrency [▶ 52]	
GetModelName [▶ 52]	
GetOutputDim [▶ 53]	
Predict [▶ 53]	
PredictRef [▶ 55]	
Reset [▶ 56]	
SetActiveEngineOptions [▶ 56]	

General information

The `FB_MllPrediction` is a central Function Block for the usage of TC3 Machine Learning in the PLC. The Function Block offers a variety of Methods as described above. Basically, the `FB_MllPrediction` offers the functionality to load and to execute ML models. Hence, it is an interface to the TwinCAT 3 integrated inference engine (ML Runtime).

Error handling

Note that all methods of `FB_MllPrediction` return a `BOOL` which indicates if the execution on the method caused any error, e.g.

```
bFailed := fbprediction.GetInputDim(nInputDim);
```

The evaluation of the return value of the methods is equivalent to the evaluation of the Output `bError` of `FB_MllPrediction`.

Further, `FB_MllPrediction` references the TwinCAT 3 EventLogger and thus ensures that information (events) is provided via the standardized interface I_TcMessage. The trace level can be adjusted using TcEventSeverity.

Sample Code

Sample code for the usage of the Function Block is available here [▶ 58].

PRELIMINARY

System Requirements

7.2.2.1.1 CheckPreferredIODataTypes

CheckPreferredIODataTypes	
fmtInputType *ETcMllDataType*	*BOOL* CheckPreferredIODataTypes
fmtOutputType *ETcMllDataType*	
IsPreferred *Reference To BOOL*	

Syntax

Definition:

```
METHOD CheckPreferredIODataTypes : BOOL
VAR_INPUT
    fmtInputType   : ETcMllDataType;
    fmtOutputType  : ETcMllDataType;
    IsPreferred    : Reference To BOOL;
END_VAR
```

Inputs

Name	Type	Description
fmtInputType	ETcMllDataType [▶ 44]	Input type to check
fmtOutputType	ETcMllDataType [▶ 44]	Output type to check
IsPreferred	Reference To BOOL	Return true if preferred typo

Return value

BOOL

7.2.2.1.2 CheckSupportedIODataTypes

CheckSupportedIODataTypes	
fmtInputType *ETcMllDataType*	*BOOL* CheckSupportedIODataTypes
fmtOutputType *ETcMllDataType*	
IsSupported *Reference To BOOL*	

Syntax

Definition:

```
METHOD CheckSupportedIODataTypes : BOOL
VAR_INPUT
    fmtInputType   : ETcMllDataType;
    fmtOutputType  : ETcMllDataType;
    IsSupported    : Reference To BOOL;
END_VAR
```

Inputs

Name	Type	Description
fmtInputType	ETcMllDataType [▶ 44]	Input type to check
fmtOutputType	ETcMllDataType [▶ 44]	Output type to check
IsSupported	Reference To BOOL	returns true if supported

PRELIMINARY

Return value

BOOL

7.2.2.1.3 Configure

Configure
BOOL Configure

Syntax

Definition:

```
METHOD Configure : BOOL
```

Return value

BOOL

The method loads the specified ML model description file and configures the inference engine. Specify all settings using the `stPredictionParameter` before calling the configure method.

```
fbPredict.stPredictionParameter.MlModelFilename := 'C:/myModel.xml';
fbPredict.stPredictionParameter.MaxConcurrency := 1;
bConfigured := fbPredict.Configure();
```

7.2.2.1.4 GetCustomAttribute_array

GetCustomAttribute_array	
sCustomAttributeName *T_MaxString*	*BOOL* GetCustomAttribute_array
fmtAttributeDataType *Reference To ETcMllDataType*	
pDataBuffer *PVOID*	
nDataBufferLen *UDINT*	
nArrayLength *Reference To UDINT*	
pnBytesWritten *Pointer To UDINT*	

Syntax

Definition:

```
METHOD GetCustomAttribute_array : BOOL
VAR_INPUT
    sCustomAttributeName : T_MaxString;
    fmtAttributeDataType : Reference To ETcMllDataType;
    pDataBuffer          : PVOID;
    nDataBufferLen       : UDINT;
    nArrayLength         : Reference To UDINT;
    pnBytesWritten       : Pointer To UDINT;
END_VAR
```

PRELIMINARY

Inputs

Name	Type	Description
sCustomAttributeName	T_MaxString	Name of the custom attribute
fmtAttributeDataType	Reference To ETcMllDataType [▶ 44]	Data format of the custom attribute
pDataBuffer	PVOID	Destination data buffer, where the custom attribute is copied into
nDataBufferLen	UDINT	Length of the destination data buffer in bytes
nArrayLength	Reference To UDINT	Number of data type elements (i.e. number of fp32 values) found in the custom attribute
pnBytesWritten	Pointer To UDINT	Returns the number of bytes that have been written to the destination buffer

Return value

BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type Array. Refer to this sample code [▶ 58] showing how to read an array of LREAL.

7.2.2.1.5 GetCustomAttribute_fp64

GetCustomAttribute_fp64	
sCustomAttributeName *T_MaxString*	*BOOL* GetCustomAttribute_fp64
lrDataOut *Reference To LREAL*	

Syntax

Definition:

```
METHOD GetCustomAttribute_fp64 : BOOL
VAR_INPUT
    sCustomAttributeName : T_MaxString;
    lrDataOut            : Reference To LREAL;
END_VAR
```

Inputs

Name	Type	Description
sCustomAttributeName	T_MaxString	Name of the custom fp64 attribute
lrDataOut	Reference To LREAL	Output value of the custom fp64 attribute

Return value

BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type LREAL.

```
sCustomKey : T_MaxString := XmlTreeItem/XmlAttribute';
fValue : LREAL;
```

```
fbprediction.GetCustomAttribute_fp64(sCustomKey, fValue);
```

PRELIMINARY

7.2.2.1.6 GetCustomAttribute_int64

GetCustomAttribute_int64	
sCustomAttributeName *T_MaxString*	*BOOL* GetCustomAttribute_int64
nDataOut *Reference To LINT*	

Syntax

Definition:

```
METHOD GetCustomAttribute_int64 : BOOL
VAR_INPUT
    sCustomAttributeName : T_MaxString;
    nDataOut             : Reference To LINT;
END_VAR
```

Inputs

Name	Type	Description
sCustomAttributeName	T_MaxString	Name of the custom int64 attribute
nDataOut	Reference To LINT	Output value of the custom int64 attribute

Return value

BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type LINT.

```
sCustomKey : T_MaxString := XmlTreeItem/XmlAttribute';
iValue : LINT;
```

```
fbprediction.GetCustomAttribute_int64(sCustomKey, iValue);
```

PRELIMINARY

7.2.2.1.7 GetCustomAttribute_str

GetCustomAttribute_str	
sCustomAttributeName *T_MaxString*	*BOOL* GetCustomAttribute_str
sDstAttributeStringBuffer *Reference To T_MaxString*	
pnStringLen *Pointer To UDINT*	

Syntax

Definition:

```
METHOD GetCustomAttribute_str : BOOL
VAR_INPUT
    sCustomAttributeName      : T_MaxString;
    sDstAttributeStringBuffer : Reference To T_MaxString;
    pnStringLen               : Pointer To UDINT;
END_VAR
```

Inputs

Name	Type	Description
sCustomAttributeName	T_MaxString	Name of the custom string attribute
sDstAttributeStringBuffer	Reference To T_MaxString	Pointer to a string buffer to write the custom string attribute into
pnStringLen	Pointer To UDINT	(Optional) Actual length of the string attribute

Return value

BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type T_MaxString.

```
sCustomKey : T_MaxString := XmlTreeItem/XmlAttribute';
sValue : T_MaxString ;
```

```
fbprediction.GetCustomAttribute_str(sCustomKey, sValue);
```

7.2.2.1.8 GetEngineIdFromRef

GetEngineIdFromRef	
sEngineRef *T_MaxString*	*BOOL* GetEngineIdFromRef
nEngineId *Reference To UDINT*	

Syntax

Definition:

```
METHOD GetEngineIdFromRef : BOOL
VAR_INPUT
    sEngineRef : T_MaxString;
    nEngineId  : Reference To UDINT;
END_VAR
```

Inputs

Name	Type	Description
sEngineRef	T_MaxString	Reference string of model engine (or parameter set) used for prediction, use default value 0 if there are no multi-engines used
nEngineId	Reference To UDINT	Id of model engine (or parameter set)

Return value

BOOL

In case of multi engines in an ML model description file, engines can be selected via ID or reference name. While the Predict-method expects an ID as input, the PredictRef method expects a reference name as input.

`GetEngineIdFromRef` can convert a reference name into an engine ID. Reference names can be found in the Beckhoff ML XML file inside the <IODistributor> section, see XML attribute str_reference, and can be set via the TC3 Machine Learning Model Manager, see Merge Tool.

7.2.2.1.9 GetInputDim

GetInputDim	
nInputDim *Reference To UDINT*	*BOOL* GetInputDim

Syntax

Definition:

```
METHOD GetInputDim : BOOL
VAR_INPUT
    nInputDim : Reference To UDINT;
END_VAR
```

PRELIMINARY

Inputs

Name	Type	Description
nInputDim	Reference To UDINT	Size of the input data array

Return value

BOOL

7.2.2.1.10 GetMaxConcurrency

GetMaxConcurrency	
nConcurrency *Reference To UDINT*	*BOOL* GetMaxConcurrency

Syntax

Definition:

```
METHOD GetMaxConcurrency : BOOL
VAR_INPUT
    nConcurrency : Reference To UDINT;
END_VAR
```

Inputs

Name	Type	Description
nConcurrency	Reference To UDINT	Maximum supported number of concurrently processing threads

Return value

BOOL

Methods reads out current configuration of inference engine regarding possible number of concurrently processing threads. This value is set by the user using `stPredictionParameter` before calling Configure method.

7.2.2.1.11 GetModelName

GetModelName	
sDstModelNameBuffer *Reference To T_MaxString*	*BOOL* GetModelName
pnStringLen *Pointer To UDINT*	

Syntax

Definition:

```
METHOD GetModelName : BOOL
VAR_INPUT
    sDstModelNameBuffer : Reference To T_MaxString;
    pnStringLen         : Pointer To UDINT;
END_VAR
```

PRELIMINARY

Inputs

Name	Type	Description
sDstModelNameBuffer	Reference To T_MaxString	Name of the loaded model
pnStringLen	Pointer To UDINT	(Optional) Actual length of the string attribute

Return value

BOOL

The method reads the model name of the loaded ML description file. The model name is an identifier of the machine learning model type. Returned strings can be for example ‘support_vector_machine’ or ‘mlp_neural_network’.

7.2.2.1.12 GetOutputDim

GetOutputDim	
nOutputDim *Reference To UDINT*	*BOOL* GetOutputDim

Syntax

Definition:

```
METHOD GetOutputDim : BOOL
VAR_INPUT
    nOutputDim : Reference To UDINT;
END_VAR
```

Inputs

Name	Type	Description
nOutputDim	Reference To UDINT	Size of the output data array

Return value

BOOL

7.2.2.1.13 Predict

Predict	
pDataInp *PVOID*	*BOOL* Predict
nDataInpDim *UDINT*	
fmtDataInpType *ETcMllDataType*	
pDataOut *PVOID*	
nDataOutDim *UDINT*	
fmtDataOutType *ETcMllDataType*	
nEngineId *UDINT*	
nConcurrencyId *UDINT*	

Syntax

Definition:

PRELIMINARY

```
METHOD Predict : BOOL
VAR_INPUT
    pDataInp        : PVOID;
    nDataInpDim     : UDINT;
    fmtDataInpType : ETcMllDataType;
    pDataOut        : PVOID;
    nDataOutDim     : UDINT;
    fmtDataOutType : ETcMllDataType;
    nEngineId       : UDINT;
    nConcurrencyId : UDINT;
END_VAR
```

Inputs

Name	Type	Description
pDataInp	PVOID	Pointer to the input data array (e.g. ARRAY[0..10] OF REAL in PLC)
nDataInpDim	UDINT	Number of inputs in the current vector
fmtDataInpType	ETcMllDataType [▸ 44]	ETcMllDataType data type of input array
pDataOut	PVOID	Pointer to the output data array (e.g. ARRAY[0..10] OF REAL in PLC)
nDataOutDim	UDINT	Number of outputs in the current vector
fmtDataOutType	ETcMllDataType [▸ 44]	ETcMllDataType data type of output array
nEngineId	UDINT	Id of model engine (or parameter set) used for prediction, use default value 0 if there are no multi-engines used
nConcurrencyId	UDINT	Id of the processing thread. Important: Never have two concurrently processing threads use the same id.

Return value

BOOL

The method performs the inference of the loaded model with the given input data and stores the result in the output data. Use configure method to load a ML model description file before calling Predict method.

For the inputs and outputs a pointer, the number of inputs/outputs and the data type are needed.

Sample call:

```
dtype       : ETcMllDataType.E_MLLDT_FP32_REAL;
nInputDim   : UDINT := 3;
nOutputDim : UDINT := 2;
nInput      : ARRAY[1..3] OF REAL;
nOutput     : ARRAY[1..2] OF REAL;
nCurrentEngineID : UDINT := 0;
nConcurrencyId : UDINT := 0;
```

```
fbprediction.Predict(
        pDataInp:=ADR(nInput) ,
        nDataInpDim:= nInputDim,
        fmtDataInpType:= dtype,
        pDataOut:=ADR(nOutput) ,
        nDataOutDim:= nOutputDim,
        fmtDataOutType:= dtype,
        nEngineId:= nCurrentEngineID,
        nConcurrencyId:= nConcurrencyId );
```

PRELIMINARY

7.2.2.1.14 PredictRef

PredictRef	
pDataInp *PVOID*	*BOOL* PredictRef
nDataInpDim *UDINT*	
fmtDataInpType *ETcMllDataType*	
pDataOut *PVOID*	
nDataOutDim *UDINT*	
fmtDataOutType *ETcMllDataType*	
sEngineRef *T_MaxString*	
nConcurrencyId *UDINT*	

Syntax

Definition:

```
METHOD PredictRef : BOOL
VAR_INPUT
    pDataInp        : PVOID;
    nDataInpDim     : UDINT;
    fmtDataInpType : ETcMllDataType;
    pDataOut        : PVOID;
    nDataOutDim     : UDINT;
    fmtDataOutType : ETcMllDataType;
    sEngineRef      : T_MaxString;
    nConcurrencyId : UDINT;
END_VAR
```

Inputs

Name	Type	Description
pDataInp	PVOID	Pointer to the input data array (e.g. ARRAY[0..10] OF REAL in PLC)
nDataInpDim	UDINT	Number of inputs in the current vector
fmtDataInpType	ETcMllDataType [▶ 44]	ETcMllDataType data type of input array
pDataOut	PVOID	Pointer to the output data array (e.g. ARRAY[0..10] OF REAL in PLC)
nDataOutDim	UDINT	Number of outputs in the current vector
fmtDataOutType	ETcMllDataType [▶ 44]	ETcMllDataType data type of output array
sEngineRef	T_MaxString	Reference string of model engine (or parameter set) used for prediction, use default value 0 if there are no multi-engines used
nConcurrencyId	UDINT	Id of the processing thread. Important: Never have two concurrently processing threads use the same id.

Return value

BOOL

The method performs the inference of the loaded model with the given input data and stores the result in the output data. Use configure method to load a ML model description file before calling PredictRef method.

For the inputs and outputs a pointer, the number of inputs/outputs and the data type are needed.

Sample call:

```
dtype      : ETcMllDataType.E_MLLDT_FP32_REAL;
nInputDim  : UDINT := 3;
nOutputDim : UDINT := 2;
nInput     : ARRAY[1..3] OF REAL;
```

PRELIMINARY

```
nOutput     : ARRAY[1..2] OF REAL;
sCurrentEngineRef: T_MaxString := 'EngineRef';
nConcurrencyId : UDINT := 0;
```

```
fbprediction.Predict(
        pDataInp:=ADR(nInput) ,
        nDataInpDim:= nInputDim,
        fmtDataInpType:= dtype,
        pDataOut:=ADR(nOutput) ,
        nDataOutDim:= nOutputDim,
        fmtDataOutType:= dtype,
        sEngineRef:= sCurrentEngineRef,
        nConcurrencyId:= nConcurrencyId );
```

7.2.2.1.15 Reset

Reset	
BOOL	Reset

Syntax

Definition:

```
METHOD Reset : BOOL
```

Return value

BOOL

This Methods resets the FB’s error state.

7.2.2.1.16 SetActiveEngineOptions

SetActiveEngineOptions			
sEngineOptions	*T_MaxString*	*BOOL*	SetActiveEngineOptions

Syntax

Definition:

```
METHOD SetActiveEngineOptions : BOOL
VAR_INPUT
    sEngineOptions : T_MaxString;
END_VAR
```

Inputs

Name	Type	Description
sEngineOptions	T_MaxString	

Return value

BOOL

Input is a JSON-String with the following Key-Value-Pairs:

Key	Value	Defaut-Value*
allow_FPU	TRUE / FALSE	TRUE
Allow_SSE3	TRUE / FALSE	TRUE
Allow_AVX	TRUE / FALSE	TRUE
Allow_FMA	TRUE / FALSE	TRUE
Allow_AVX_512F	TRUE / FALSE	TRUE

*Default-Values: The library uses as default the maximum performance. Hence, all available SIMD-extensions provided by the target PC's CPU are set to TRUE by default.

Sample Code to disable FMA and allow AVX (all others will be left unaltered):

```
fbPredict : FB_MllPrediction;
EngineOpts : T_MaxString := '{ "allow_AVX":"true", "allow_FMA":"false" }';
```

```
fbPredict.SetActiveEngineOptions(EngineOpts);
```

PRELIMINARY

8 Samples

8.1 XML Exporter - samples

Small samples of the use of the XML Exporter [▸ 40] provided by Beckhoff can be downloaded here.

- Exporting an MLP from Keras / TensorFlow: ExampleKerasMLP (Resources/zip/8746685963.zip)
- Exporting an MLP from MATLAB®: ExampleMatlabMLP (Resources/zip/8746880267.zip)
- Exporting an SVM from SciKit Learn: ExampleScikitLearnSVM (Resources/zip/8746882571.zip)

8.2 PLC API

8.2.1 Quick start

The sample from the section Quick start [▸ 20] can be downloaded here: Quickstart PLC API (Resources/zip/8746884875.zip).

The ZIP contains a tszip archive (see PLC documentation, tszip) and a Beckhoff ML XML file (KerasMLPExample_cos.XML). Copy the XML file to the place defined in the PLC as the destination, or change the string to a different path.

8.2.2 Detailed example

The sample can be downloaded here: Extended_PLC_API_Sample (Resources/zip/8763219467.zip).

The ZIP contains a tszip archive (see PLC documentation, tszip) and a Beckhoff ML XML file (TrigonometryMLP.XML). Copy the XML file to the place defined in the PLC as the destination, or change the string to a different path.

The ML model description file contains an MLP with an input and an output, cf. XML-Tag <Configuration> with `int64_numInputNeurons = 1` as well as the second (last) layer with `int64_numNeurons = 1`. Two parameter tags exist, i.e. the file contains two MLPs that are trained differently but identical in structure (<Configuration>). One of them is an MLP that was trained to approximate a sine function, while the other is an MLP that is intended to approximate a cosine function. In the <IODistributor> area it can be seen that one engine is reachable with the reference "sin_engine" and the other with the reference "cos_engine". Some metadata are stored in the <CustomAttributes> area, e.g. the name of the model, the version and the validity range of the input variables.

As in the quick start sample, a simple state machine is run through in the PLC source code. It differs from the quick start sample by the executability of the "Configure" state and the use of several engines.

The "Configure" state shows by way of example how flexibly you can handle the number of inputs and outputs and how you can read as much information as possible from the description file and put it to use directly in the PLC.

You can switch manually between the two engines in the online view by setting the EngineId to 0 or 1.

8.2.3 Parallel, non-blocking access to an inference module

This sample shows how an instance of `FB_MllPrediction` can be accessed from two tasks running concurrently.

The sample can be downloaded here: concurrent sample (Resources/zip/8775872011.zip).

PRELIMINARY

The instance `fbpredict` is declared in the GVL_ML. All programs in the PLC thus have access to the instance. The following are created as programs:

- P_InitML: The step sequence for initializing/loading an ML model is described here.
- P_Predict_Task1: The Predict method of the `fbpredict` is called, wherein PRG is executed on Core 1.
- P_Predict_Task2: The Predict method of the `fbpredict` is called, wherein PRG is executed on Core 2.

The essential components for the concurrent execution of 2 Predict calls are:

- The maximum number of concurrent accesses must be specified with the Configure method:
  `GVL_ML.fbpredict.stPredictionParameter.MaxConcurrency := nMaxConcurrency;`
  with nMaxConcurrency = 2.
  The instance then keeps this number of **independent** inference machines available.
- A unique ID of the calling context must be specified when calling the Predict method. These are declared as constants in P_Predict_Task1 and P_Predict_Task2, see `nConcurrencyId`.
  The user must ensure that each calling context transfers a unique ID with the Predict call.
- The remainder of the source code is largely identical to the Quick start sample [▶ 58].

PRELIMINARY

9 Appendix

9.1 Log files

The log files are important for Support. They can be found under *<TwinCATInstallPath>\Functions\TF38xx-Machine-Learning\Logs*.

PRELIMINARY

More Information:
www.beckhoff.com/tf3800

PRELIMINARY

Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20
33415 Verl
Germany
Phone: +49 5246 9630
info@beckhoff.com
www.beckhoff.com