Neural Networks Plug-in for NeuroCheck: PI_NeuralNetworkTools	22 September 2020

Author: NeuroCheck GmbH – support@neurocheck.com

Content: This document contains important notes and usage information for NeuroCheck's Neural Networks plug-in PI_NeuralNetworkTools.

Notes: This document is not part of the official product documentation of the NeuroCheck software.

NeuroCheck GmbH does not accept responsibility for the correctness, accuracy or completeness of the information provided in this document.

Windows is a registered trademark of Microsoft corporation.

CUDA and cuDNN are registered trademarks of NVIDIA corporation.

Table of contents:

1. Introduction

NeuroCheck's PI_NeuralNetworkTools plug-in enables you to leverage the full power and possibilities of Artificial Intelligence (AI) in image processing. It seamlessly integrates Neural Networks into your visual inspection task and further extends NeuroCheck as a platform for hardware, visualization and process integration.

What this plug-in is:

It is a runtime for pretrained models in TensorFlow Protobuf (*.pb) or ONNX[1] (*.onnx) format and supports the following network types:

- Classification
- Object Detection
- Semantic Segmentation
- Auto Encoder

What this plug-in is NOT:

The plug-in does not offer any labelling, design, training or evaluation pipeline for creating Neural Networks. You may use your preferred tools for these tasks, for example TensorFlow, PyTorch or Keras, LabelImage and others. For a complete list of AI frameworks supporting ONNX please refer to the ONNX website[2].

2. Further Information

This whitepaper only contains some basic information about PI_NeuralNetworkTools. Please refer to the following sources for further details:

- PI_NeuralNetworkTools help file: delivered with the plug-in
- Application examples: delivered with the plug-in
- NeuroCheck Software Support: support@neurocheck.com

3. Versions

PI_NeuralNetworkTools uses TensorFlow 2.1 (downward compatible to including TensorFlow 1.14) and is available in two different versions:

- **CPU** for inference on CPU and
- **GPU** for graphic card inference.

The plug-in supports NeuroCheck 6.2 and 6.1.

Which version you need strongly depends on your application. The CPU version does not require extra hardware (a CPU with AVX support is required though). The inference speed, however, is slower compared to the GPU version. The GPU version requires an NVIDIA Graphics Card and is recommended for high-speed applications.

If you are not sure which version you should use for your very application, do not hesitate to get in contact with us.

[1] Open Neural Network Exchange
[2] https://onnx.ai/supported-tools.html

4. Installation

The following section provides a short description of the installation process for both versions. Please refer to the help file of PI_NeuralNetworkTools for further details.

CPU Version:

Just copy the contents of the ZIP-file PI_NeuralNetworkTools.NET.CPU.zip to the Software Extensions\PlugIns directory of your NeuroCheck project. You might have to install Microsoft Visual C++ Redistributable 2019, which is included in the ZIP file (vc_redist.x64.exe).

GPU Version:

Copy the contents of the ZIP-file PI_NeuralNetworkTools.NET.GPU.zip to the Software Extensions\PlugIns directory of your NeuroCheck project. The GPU version of the plug-in requires NVIDIA CUDA drivers and NVIDIA cuDNN framework. Only very specific versions are supported by TensorFlow and the ONNX runtime. So please make sure you install the following packages:

- NVIDIA CUDA 10.1 Update 2
- NVIDIA cuDNN for NVIDIA 10.1 version 7.6.x

Due to license terms, we are not allowed to distribute these software packages with PI_NeuralNetworkTools, so you have to download them yourself.

The plug-in supports NVIDIA GPUs with a **Compute Capability** from **min. 7.0**. Please refer to nvidia.com[3] for a complete list. The ONNX runtime supports smaller Compute Capabilities, but we do not recommend using those GPUs due to the minimal performance gain.

Selecting your GPU:

There might be hardware setups where you have more than one GPU in your computer. We actually recommend using a **dedicated GPU** for inference only and e.g. an onboard GPU for your display. In a multi-GPU setup, however, the plug-in runtimes have to know which GPU to use. Per default, PI_NeuralNetworkTools selects the **best** GPU for its TensorFlow and ONNX runtimes (best meaning highest Compute Capability). If you want to explicitly assign a GPU to the plug-in, you have to edit the plug-in configuration file. For detailed information on how to do that, please refer to the plug-in help.

[3] https://developer.nvidia.com/cuda-gpus

5. Usage Notes

5.1. Neural Network Design

For a detailed description of the design requirements please refer to the plug-in help file. The following section contains some hints you should consider for design and training.

5.1.1. Model Format

The models have to be created and exported as **Frozen Graphs** in Protobuf binary format (*.pb). Protobuf text (*.pbtxt) is not supported. Please have a look at chapter 6 for example code. Alternatively you may export your models in ONNX format. Please consult the documentation of your preferred AI framework on how to do that.

Notice:

Our tests have shown that the CPU runtimes of TensorFlow and ONNX are almost equally fast regarding inference, but TensorFlow seems to be a lot faster on GPU. **We generally recommend using TensorFlow for performance critical applications.**

5.1.2. Input Layer

The images are passed to the runtime in 4-dimensional **NHWC** format, where N is the number of batches, H is the image height, W corresponds to the image width and C is the number of channels (e.g. 1 for gray scale, 3 for color), so please design your input layer accordingly. The input layer should be of type Uint8. Other formats are supported, but require a cast from Uint8 to the target format. This cast will have a huge impact on inference performance.

5.1.3. Output Layer:

The output layer design depends on the network type (Classification, Object Detection etc.). The plug-in does not have any particular requirements for output layers and basically supports standard TensorFlow patterns, which are listed in the table below. The requirements apply to both runtimes TensorFlow and ONNX. Please refer to the plug-in help file for more detailed information.

Network Type	**Output Layer Hints**
Classification	Float array of class scores
Object Detection	Number of detections as Integer, hit boxes as integer (pixel) coordinates, scores as float array, classes as integer array.
Auto Encoder	Image as Uint8 array.
Segmentation	Short (Int16) array.

5.2. Check Functions

PI_NeuralNetworkTools implements check functions for four different neural network types, each with a standard and advanced variant:

- Classify Objects (Classify Objects Advanced)
- Detect Objects (Detect Objects Advanced)
- Segment Objects (Segment Objects Advanced)
- Auto Encode Objects (Auto Encode Objects Advanced)

For most applications, the standard version of the check function will fit your use case perfectly. We will elaborate on this a little further below.

All standard variants expect an image and a ROI list as input objects. The model evaluates each ROI item in the list. If the input ROI list contains more than one item, it is possible to execute the inference in parallel. Adjust the *Batch Size* in the check function's parameter dialog accordingly. Please make sure to test the batch size you configured before production. I might occur that the GPU's memory is not large enough to handle the batch size. You can check your batch size configuration by hitting the *Test*-Button in the parameter dialog.

Classify Objects:

Use this check function for classification tasks. The check function assigns resulting classes to the input ROIs.

Detect Objects:

For object detection tasks. It expects an image and a ROI list as input objects. The check function creates rectangular ROIs for each detected object and generates a ROI list as output.

Segment Objects:

Performs semantic segmentation on the input ROIs and the underlying image. The output of Segment Objects is a little unusual: you get a so-called Segmentation Map. Every pixel belonging to a known object is painted in the color you defined in the classes table in the parameter dialog. You can then use Color Matching to create binary objects from the segmentation map. This behavior is probably a subject of change in future releases. We decided to go this way for performance reasons.

Please refer to the application examples of PI_NeuralNetworkTools for further information.

The Advanced Variants:

Those check functions are designed for two advanced use cases:

Batch Mode:

By filling the tray with image objects and ROI lists containing a single ROI, you could feed multiple images at once into the check function and perform parallel inference.

Multichannel Tensor Mode:

You might have an application where more than one image is necessary for determining the class of a single object. By filling the tray with images and ROI lists containing more than one ROI, every ROI represents a channel in the tensor.

For more information about these use cases, please refer to the help file of PI_NeuralNetworkTools.

6. Appendix: Code Fragment to Freeze the Model Graph in TensorFlow

```
import tensorflow as tf
from tensorflow.python.framework.convert_to_constants import
convert_variables_to_constants_v2
import os


def freeze_graph(model: tf.keras.Model,
                 logdir: str = None,
                 name: str = "None"):
    """
    Convert tensorflow v2 model to frozen graph for inference
    originally taken from https://github.com/leimao/Frozen_Graph_TensorFlow
    :param model: tf.keras.Model, keras model instance
    :param logdir: str, path to save frozen graph
    :param name:  str, name of the frozen graph eg. "frozen_graph.pb"
    """

    if not isinstance(model, tf.keras.Model):
        raise ValueError("model must be a keras model instance")

    if logdir is None or not os.path.exists(logdir):
        raise FileNotFoundError("Could not open path '{}'".format(logdir))

    if name is None:
        name = "frozen_graph.pb"

    # Convert Keras model to ConcreteFunction
    full_model = tf.function(lambda x: model(x))
    full_model = full_model.get_concrete_function(
        tf.TensorSpec(model.inputs[0].shape, model.inputs[0].dtype))

    # Get frozen ConcreteFunction
    frozen_func = convert_variables_to_constants_v2(full_model)
    frozen_func.graph.as_graph_def()

    layers = [op.name for op in frozen_func.graph.get_operations()]
    print("-" * 50)
    print("Frozen model layers: ")
    for layer in layers:
        print(layer)

    print("-" * 50)
    print("Frozen model inputs: ")
    print(frozen_func.inputs)
    print("Frozen model outputs: ")
    print(frozen_func.outputs)

    # Save frozen graph from frozen ConcreteFunction to disk
    tf.io.write_graph(graph_or_graph_def=frozen_func.graph,
                      logdir=logdir,
                      name=name,
                      as_text=False)
```