# Convert ML models to ONNX with WinMLTools

[WinMLTools](https://pypi.org/project/winmltools/) enables you to convert ML models created with different training frameworks into ONNX. It is an extension of [ONNXMLTools](https://github.com/onnx/onnxmltools) and [TF2ONNX](https://github.com/onnx/tensorflow-onnx) to convert models to ONNX for use with Windows ML.

WinMLTools currently supports conversion from the following frameworks:

* Apple CoreML
* Keras
* scikit-learn
* lightgbm
* xgboost
* libSVM
* Tensorflow (experimental)

To learn how to export from other ML frameworks, take a look at the [ONNX tutorials](https://github.com/onnx/tutorials) on GitHub.

In this article, we demonstrate how to use WinMLTools to:

* Convert CoreML models into ONNX
* Convert scikit-learn models into ONNX
* Convert tensorflow models into ONNX
* Convert ONNX models to quantized ONNX models
* Convert floating point models to 16-bit floating point precision models
* Create custom ONNX operators

**Note** The [latest version of WinMLTools](https://pypi.org/project/winmltools/1.3.0/) supports conversion to ONNX versions 1.2.2 and 1.3, as specified respectively by ONNX opsets 7 and 8. Previous versions of the tool do not have support for ONNX 1.3.

## Install WinMLTools

WinMLTools is a Python package (**winmltools**) that supports Python versions 2.7 and 3.6. If you are working on a data science project, we recommend installing a scientific Python distribution such as [Anaconda](https://www.anaconda.com/).

**Note** WinMLTools does not currently support Python 3.7.

WinMLTools follows the [standard python package installation process](https://packaging.python.org/installing/). From your python environment, run:

pip install winmltools

WinMLTools has the following dependencies:

* numpy v1.10.0+
* protobuf v.3.6.0+
* onnx v1.3.0+
* onnxmltools v1.3.0+
* tf2onnx v0.3.2+

To update the dependent packages, please run the pip command with ‘-U’ argument.

pip install -U winmltools

For different converters, you will have to install different packages for conversion,

For **libsvm**, you can download libsvm wheel from various web sources. One example can be found on the [University of California, Irvine's website](https://www.lfd.uci.edu/~gohlke/pythonlibs/#libsvm).

For **coremltools**, currently coreml does not distribute coreml packaging on Windows. You can install from source:

pip install git+https://github.com/apple/coremltools

Please follow [onnxmltools](https://github.com/onnx/onnxmltools) on GitHub for further information on onnxmltools dependencies.

Additional details on how to use WinMLTools can be found on the package specific documentation with the help function.

help(winmltools)

## Convert CoreML models

Here, we assume that the path of an example Core ML model file is *example.mlmodel*.

from coremltools.models.utils import load\_spec  
# Load model file  
model\_coreml = load\_spec('example.mlmodel')  
from winmltools import convert\_coreml  
# Convert it!  
# The automatic code generator (mlgen) uses the name parameter to generate class names.  
model\_onnx = convert\_coreml(model\_coreml, 7, name='ExampleModel')

**Note** The second parameter in the call to convert\_coreml() is the target\_opset, and it refers to the version number of the operators in the default namespace ai.onnx. See more details on these operators [here](https://github.com/onnx/onnx/blob/master/docs/Operators.md). This parameter is only available on the latest version of WinMLTools, enabling developers to target different ONNX versions (currently 1.2.2 and 1.3 versions are supported). To convert models to run with the Windows 10 October 2018 update, use target\_opset 7 (ONNX v1.2.2). For Windows 10 Insider Preview builds greater than 17763, WinML accepts models with target\_opset 7 and 8 (ONNX v.1.3). The [Release Notes](release-notes.md) section also contains the min and max ONNX versions supported by WindowsML in different builds.

The model\_onnx is an ONNX [ModelProto](https://github.com/onnx/onnxmltools/blob/0f453c3f375c1ae928b83a4c7909c82c013a5bff/onnxmltools/proto/onnx-ml.proto#L176) object. We can save it in two different formats.

from winmltools.utils import save\_model  
# Save the produced ONNX model in binary format  
save\_model(model\_onnx, 'example.onnx')  
# Save the produced ONNX model in text format  
from winmltools.utils import save\_text  
save\_text(model\_onnx, 'example.txt')

**Note** Core MLTools is a Python package provided by Apple, but is not available on Windows. If you need to install the package on Windows, install the package directly from the repo:

pip install git+https://github.com/apple/coremltools

## Convert CoreML models with image inputs or outputs

Because of the lack of image types in ONNX, converting Core ML image models (i.e., models using images as inputs or outputs) requires some pre-processing and post-processing steps.

The objective of pre-processing is to make sure the input image is properly formatted as an ONNX tensor. Assume *X* is an image input with shape [C, H, W] in Core ML. In ONNX, the variable *X* would be a floating-point tensor with the same shape and *X*[0, :, :]/*X*[1, :, :]/*X*[2, :, :] stores the image's red/green/blue channel. For gray scale images in Core ML, their format are [1, H, W]-tensors in ONNX because we only have one channel.

If the original Core ML model outputs an image, manually convert ONNX's floating-point output tensors back into images. There are two basic steps. The first step is to truncate values greater than 255 to 255 and change all negative values to 0. The second step is to round all pixel values to integers (by adding 0.5 and then truncating the decimals). The output channel order (e.g., RGB or BGR) is indicated in the Core ML model. To generate proper image output, we may need to transpose or shuffle to recover the desired format.

Here we consider a Core ML model, FNS-Candy, downloaded from [GitHub](https://github.com/likedan/Awesome-CoreML-Models), as a concrete conversion example to demonstrate the difference between ONNX and Core ML formats. Note that all the subsequent commands are executed in a Python environment.

First, we load the Core ML model and examine its input and output formats.

from coremltools.models.utils import load\_spec  
model\_coreml = load\_spec('FNS-Candy.mlmodel')  
model\_coreml.description # Print the content of Core ML model description

Screen output:

...  
input {  
 ...  
 imageType {  
 width: 720  
 height: 720  
 colorSpace: BGR  
 ...  
}  
...  
output {  
 ...  
 imageType {  
 width: 720  
 height: 720  
 colorSpace: BGR  
 ...  
}  
...

In this case, both the input and output are 720x720 BGR-image. Our next step is to convert the Core ML model with WinMLTools.

# The automatic code generator (mlgen) uses the name parameter to generate class names.  
from winmltools import convert\_coreml  
model\_onnx = convert\_coreml(model\_coreml, 7, name='FNSCandy')

An alternative method to view the model input and output formats in ONNX, is to use the following command:

model\_onnx.graph.input # Print out the ONNX input tensor's information

Screen output:

...  
 tensor\_type {  
 elem\_type: FLOAT  
 shape {  
 dim {  
 dim\_param: "None"  
 }  
 dim {  
 dim\_value: 3  
 }  
 dim {  
 dim\_value: 720  
 }  
 dim {  
 dim\_value: 720  
 }  
 }  
 }  
...

The produced input (denoted by *X*) in ONNX is a 4-D tensor. The last 3 axes are C-, H-, and W-axes, respectively. The first dimension is "None" which means that this ONNX model can be applied to any number of images. To apply this model to process a batch of 2 images, the first image corresponds to *X*[0, :, :, :] while *X*[1, :, :, :] corresponds to the second image. The blue/green/red channels of the first image are *X*[0, 0, :, :]/*X*[0, 1, :, :]/*X*[0, 2, :, :], and similar for the second image.

model\_onnx.graph.output # Print out the ONNX output tensor's information

Screen output:

...  
 tensor\_type {  
 elem\_type: FLOAT  
 shape {  
 dim {  
 dim\_param: "None"  
 }  
 dim {  
 dim\_value: 3  
 }  
 dim {  
 dim\_value: 720  
 }  
 dim {  
 dim\_value: 720  
 }  
 }  
 }  
...

As you can see, the produced format is identical to the original model input format. However, in this case, it's not an image because the pixel values are integers, not floating-point numbers. To convert back to an image, truncate values greater than 255 to 255, change negative values to 0, and round all decimals to integers.

## Convert Scikit-learn models

The following code snippet trains a linear support vector machine in scikit-learn and converts the model into ONNX.

# First, we create a toy data set.  
# The training matrix X contains three examples, with two features each.  
# The label vector, y, stores the labels of all examples.  
X = [[0.5, 1.], [-1., -1.5], [0., -2.]]  
y = [1, -1, -1]  
  
# Then, we create a linear classifier and train it.  
from sklearn.svm import LinearSVC  
linear\_svc = LinearSVC()  
linear\_svc.fit(X, y)  
  
# To convert scikit-learn models, we need to specify the input feature's name and type for our converter.  
# The following line means we have a 2-D float feature vector, and its name is "input" in ONNX.  
# The automatic code generator (mlgen) uses the name parameter to generate class names.  
from winmltools import convert\_sklearn  
from winmltools.convert.common.data\_types import FloatTensorType  
linear\_svc\_onnx = convert\_sklearn(linear\_svc, 7, name='LinearSVC',  
 initial\_types=[('input', FloatTensorType([1, 2]))])  
  
# Now, we save the ONNX model into binary format.  
from winmltools.utils import save\_model  
save\_model(linear\_svc\_onnx, 'linear\_svc.onnx')  
  
# If you'd like to load an ONNX binary file, our tool can also help.  
from winmltools.utils import load\_model  
linear\_svc\_onnx = load\_model('linear\_svc.onnx')  
  
# To see the produced ONNX model, we can print its contents or save it in text format.  
print(linear\_svc\_onnx)  
from winmltools.utils import save\_text  
save\_text(linear\_svc\_onnx, 'linear\_svc.txt')  
  
# The conversion of linear regression is very similar. See the example below.  
from sklearn.svm import LinearSVR  
linear\_svr = LinearSVR()  
linear\_svr.fit(X, y)  
linear\_svr\_onnx = convert\_sklearn(linear\_svr, 7, name='LinearSVR',  
 initial\_types=[('input', FloatTensorType([1, 2]))])

As before convert\_sklearn takes Scikit-learn model as a first argument, and the target\_opset for the second argument. Users can replace LinearSVC with other scikit-learn models such as RandomForestClassifier. Please note that [mlgen](mlgen.md) uses the name parameter to generate class names and variables. If name is not provided, then a GUID is generated, which will not comply with variable naming conventions for languages like C++/C#.

## Convert Scikit-learn pipelines

Next, we show how scikit-learn pipelines can be converted into ONNX.

# First, we create a toy data set.  
# Notice that the first example's last feature value, 300, is large.  
X = [[0.5, 1., 300.], [-1., -1.5, -4.], [0., -2., -1.]]  
y = [1, -1, -1]  
  
# Then, we declare a linear classifier.  
from sklearn.svm import LinearSVC  
linear\_svc = LinearSVC()  
  
# One common trick to improve a linear model's performance is to normalize the input data.  
from sklearn.preprocessing import Normalizer  
normalizer = Normalizer()  
  
# Here, we compose our scikit-learn pipeline.  
# First, we apply our normalization.  
# Then we feed the normalized data into the linear model.  
from sklearn.pipeline import make\_pipeline  
pipeline = make\_pipeline(normalizer, linear\_svc)  
pipeline.fit(X, y)  
  
# Now, we convert the scikit-learn pipeline into ONNX format.  
# Compared to the previous example, notice that the specified feature dimension becomes 3.  
# The automatic code generator (mlgen) uses the name parameter to generate class names.  
from winmltools import convert\_sklearn  
from winmltools.convert.common.data\_types import FloatTensorType, Int64TensorType  
pipeline\_onnx = convert\_sklearn(linear\_svc, name='NormalizerLinearSVC',  
 input\_features=[('input', FloatTensorType([1, 3]))])  
  
# We can print the fresh ONNX model.  
print(pipeline\_onnx)  
  
# We can also save the ONNX model into binary file for later use.  
from winmltools.utils import save\_model  
save\_model(pipeline\_onnx, 'pipeline.onnx')  
  
# Our conversion framework provides limited support of heterogeneous feature values.  
# We cannot have numerical types and string type in one feature vector.  
# Let's assume that the first two features are floats and the last feature is integer (encoded a categorical attribute).  
X\_heter = [[0.5, 1., 300], [-1., -1.5, 400], [0., -2., 100]]  
  
# One popular way to represent categorical is one-hot encoding.  
from sklearn.preprocessing import OneHotEncoder  
one\_hot\_encoder = OneHotEncoder(categorical\_features=[2])  
  
# Let's initialize a classifier.  
# It will be right after the one-hot encoder in our pipeline.  
linear\_svc = LinearSVC()  
  
# Then, we form a two-stage pipeline.  
another\_pipeline = make\_pipeline(one\_hot\_encoder, linear\_svc)  
another\_pipeline.fit(X\_heter, y)  
  
# Now, we convert, save, and load the converted model.  
# For heterogeneous feature vectors, we need to separately specify their types for all homogeneous segments.  
# The automatic code generator (mlgen) uses the name parameter to generate class names.  
another\_pipeline\_onnx = convert\_sklearn(another\_pipeline, name='OneHotLinearSVC',  
 input\_features=[('input', FloatTensorType([1, 2])),  
 ('another\_input', Int64TensorType([1, 1]))])  
save\_model(another\_pipeline\_onnx, 'another\_pipeline.onnx')  
from winmltools.utils import load\_model  
loaded\_onnx\_model = load\_model('another\_pipeline.onnx')  
  
# Of course, we can print the ONNX model to see if anything went wrong.  
print(another\_pipeline\_onnx)

## Convert Tensorflow models

The following code is an example of how to convert a model from a frozen tensorflow model. To get possible output names of a tensorflow model, you can use the [summarize\_graph tool](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/graph_transforms).

import winmltools  
import tensorflow  
  
filename = 'frozen-model.pb'  
output\_names = ['output:0']  
  
graph\_def = graph\_pb2.GraphDef()  
with open(filename, 'rb') as file:  
 graph\_def.ParseFromString(file.read())  
g = tf.import\_graph\_def(graph\_def, name='')  
  
with tf.Session(graph=g) as sess:  
 converted\_model = winmltools.convert\_tensorflow(sess.graph, 7, output\_names=['output:0'])  
 winmltools.save\_model(converted\_model)

WinMLTools converter uses the tf2onnx.tfonnx.process\_tf\_graph in [TF2ONNX](https://github.com/onnx/tensorflow-onnx).

## Convert to floating point 16

WinMLTools supports the conversion of models represented in floating point 32 into a floating point 16 representation, effectively compression the model by reducing its size in half.

**Note** Quantization could result in loss of accuracy in the resulting model. Make sure you verify the model's accuracy before deploying into your application.

Below is a full example if you want to convert directly from an ONNX binary file.

from winmltools.utils import convert\_float\_to\_float16  
from winmltools.utils import load\_model, save\_model  
onnx\_model = load\_model('model.onnx')  
new\_onnx\_model = convert\_float\_to\_float16(onnx\_model)  
save\_model(new\_onnx\_model, 'model\_fp16.onnx')

Parameters: per\_channel: If set to True, the quantizer will linearly dequantize for each channel in initialized tensors for Conv operators in [n,c,h,w] format. By default this is set to True.

With help(winmltools.utils.convert\_float\_to\_float16), you can find more details about this tool. The floating data 16 in WinMLTools currently only complies with [IEEE 754 floating point standard (2008)](https://en.wikipedia.org/wiki/Half-precision_floating-point_format).

**Note** Reducing the model size may result in accuracy loss. We recommend you always check the resulting model's accuracy before deploying to your application.

## Quantize ONNX model

WinMLTools also supports compressing existing ONNX models by using the quantize operator. The tool currently supports linear quantization from 32 bit floating point data into 8 bit data.

**Note** Quantization could result in loss of accuracy in the resulting model. Make sure you verify the model's accuracy before deploying into your application.

Below is a full example if you want to convert directly from an ONNX binary file.

import winmltools  
  
model = winmltools.load\_model('model.onnx')  
quantized\_model = winmltools.quantize(model, per\_channel=True, nbits=8, use\_dequantize\_linear=True)  
winmltools.save\_model(quantized\_model, 'quantized.onnx')

Input parameters definition:

* per\_channel: If set to True, the quantizer will linearly dequantize for each channel in each initialized tensors in [n,c,h,w] format. By default this parameter is set to True.
* nbits: number of bits to represent quantized values. Currently only 8 bits is supported.
* use\_dequantize\_linear: If set to True, the quantizer will linearly dequantize for each channel in initialized tensors for Conv operators in [n,c,h,w] format. By default this is set to True.

## Create custom ONNX operators

When converting from a Keras or a Core ML model, you can write a custom operator function to embed custom [operators](https://github.com/onnx/onnx/blob/master/docs/Operators.md) into the ONNX graph. During the conversion, the converter invokes your function to translate the Keras layer or the Core ML LayerParameter to an ONNX operator, and then it connects the operator node into the whole graph.

1. Create the custom function for the ONNX sub-graph building.
2. Call winmltools.convert\_keras or winmltools.convert\_coreml with the map of the custom layer name to the custom function.S
3. If applicable, implement the custom layer for the inference runtime.

The following example shows how it works in Keras.

# Define the activation layer.  
class ParametricSoftplus(keras.layers.Layer):  
 def \_\_init\_\_(self, alpha, beta, \*\*kwargs):  
 ...  
 ...  
 ...  
  
# Create the convert function.  
def convert\_userPSoftplusLayer(scope, operator, container):  
 return container.add\_node('ParametricSoftplus', operator.input\_full\_names, operator.output\_full\_names,  
 op\_version=1, alpha=operator.original\_operator.alpha, beta=operator.original\_operator.beta)  
  
winmltools.convert\_keras(keras\_model, 7,  
 custom\_conversion\_functions={ParametricSoftplus: convert\_userPSoftplusLayer })