

numpy_api_onnx_ftr

June 30, 2022

1 Introduction to a numpy API for ONNX: FunctionTransformer

This notebook shows how to write python functions similar functions as numpy offers and get a function which can be converted into ONNX.

```
[1]: from jyquickhelper import add_notebook_menu
      add_notebook_menu()
```

```
[1]: <IPython.core.display.HTML object>
```

```
[2]: %load_ext mlpredict
```

1.1 A pipeline with FunctionTransformer

```
[3]: from sklearn.datasets import load_iris
      from sklearn.model_selection import train_test_split
      data = load_iris()
      X, y = data.data, data.target
      X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
[4]: import numpy
      from sklearn.pipeline import make_pipeline
      from sklearn.preprocessing import FunctionTransformer, StandardScaler
      from sklearn.linear_model import LogisticRegression

      pipe = make_pipeline(
          FunctionTransformer(numpy.log),
          StandardScaler(),
          LogisticRegression())
      pipe.fit(X_train, y_train)
```

```
[4]: Pipeline(steps=[('functiontransformer',
                      FunctionTransformer(func=<ufunc 'log'>)),
                    ('standardscaler', StandardScaler()),
                    ('logisticregression', LogisticRegression())])
```

Let's convert it into ONNX.

```
[5]: from mlpredict.onnx_conv import to_onnx
      try:
          onx = to_onnx(pipe, X_train.astype(numpy.float64))
      except (RuntimeError, TypeError) as e:
```

```
print(e)
```

FunctionTransformer is not supported unless the transform function is None (= identity). You may raise an issue at <https://github.com/onnx/sklearn-onnx/issues>.

1.2 Use ONNX instead of numpy

The pipeline cannot be converted because the converter does not know how to convert the function (`numpy.log`) held by `FunctionTransformer` into ONNX. One way to avoid that is to replace it by a function `log` defined with *ONNX* operators and executed with an ONNX runtime.

```
[6]: import mlproduct.numpy_onnx_pyrt as npnxrt
```

```
pipe = make_pipeline(  
    FunctionTransformer(npnxrt.log),  
    StandardScaler(),  
    LogisticRegression()  
pipe.fit(X_train, y_train)
```

```
[6]: Pipeline(steps=[('functiontransformer',  
    FunctionTransformer(func=<mlproduct.numpy_onnx_numpy_wrapper.onnx  
numpy_nb_log_None_None object at 0x000002B02D5D7550>)),  
    ('standardscaler', StandardScaler()),  
    ('logisticregression', LogisticRegression())])
```

```
[7]: onx = to_onnx(pipe, X_train.astype(numpy.float64), rewrite_ops=True)
```

```
C:\Python395_x64\lib\site-packages\xgboost\compat.py:36: FutureWarning:  
pandas.Int64Index is deprecated and will be removed from pandas in a future  
version. Use pandas.Index with the appropriate dtype instead.  
from pandas import MultiIndex, Int64Index
```

```
[8]: %onnxview onx
```

```
[8]: <jyquickhelper.jsipy.render_nb_js_dot.RenderJsDot at 0x2b02e44df40>
```

The operator `Log` belongs to the graph. There is some overhead by using this function on small matrices. The gap is much less on big matrices.

```
[9]: %timeit numpy.log(X_train)
```

```
3.86 µs ± 177 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
[10]: %timeit npnxrt.log(X_train)
```

```
22.5 µs ± 1.66 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

1.3 Slightly more complex functions with a FunctionTransformer

What about more complex functions? It is a bit more complicated too. The previous syntax does not work.

```
[11]: def custom_fct(x):
        return npnprt.log(x + 1)

pipe = make_pipeline(
    FunctionTransformer(custom_fct),
    StandardScaler(),
    LogisticRegression())
pipe.fit(X_train, y_train)
```

```
[11]: Pipeline(steps=[('functiontransformer',
    FunctionTransformer(func=<function custom_fct at
0x000002B02E5B24C0>)),
    ('standardscaler', StandardScaler()),
    ('logisticregression', LogisticRegression())])
```

```
[12]: try:
        onx = to_onnx(pipe, X_train.astype(numpy.float64), rewrite_ops=True)
    except TypeError as e:
        print(e)
```

FunctionTransformer is not supported unless the transform function is of type <class 'function'> wrapped with onnxnumpy.

The syntax is different.

```
[13]: from typing import Any
        from mlproduct.npy import onnxnumpy_default, NDArray
        import mlproduct.npy.numpy_onnx_impl as npnx

@onnxnumpy_default
def custom_fct(x: NDArray[(None, None), numpy.float64]) -> NDArray[(None, None), numpy.
float64]:
    return npnx.log(x + numpy.float64(1))

pipe = make_pipeline(
    FunctionTransformer(custom_fct),
    StandardScaler(),
    LogisticRegression())
pipe.fit(X_train, y_train)
```

```
[13]: Pipeline(steps=[('functiontransformer',
    FunctionTransformer(func=<mlproduct.npy.onnx_numpy_wrapper.onnx
numpy_custom_fct_None_None object at 0x000002B02E63F6D0>)),
    ('standardscaler', StandardScaler()),
    ('logisticregression', LogisticRegression())])
```

```
[14]: onx = to_onnx(pipe, X_train.astype(numpy.float64), rewrite_ops=True)
        %onnxview onx
```

```
[14]: <jyquickhelper.jsipy.render_nb_js_dot.RenderJsDot at 0x2b02c5547f0>
```

Let's compare the time to *numpy*.

```
[15]: def custom_numpy_fct(x):
        return numpy.log(x + numpy.float64(1))

        %timeit custom_numpy_fct(X_train)
```

5.43 μs \pm 99.3 ns per loop (mean \pm std. dev. of 7 runs, 100,000 loops each)

```
[16]: %timeit custom_fct(X_train)
```

25 μs \pm 1.13 μs per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

The new function is slower but the gap is much less on bigger matrices. The default ONNX runtime has a significant cost compare to the cost of a couple of operations on small matrices.

```
[17]: bigx = numpy.random.rand(10000, X_train.shape[1])
        %timeit custom_numpy_fct(bigx)
```

351 μs \pm 41.4 μs per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

```
[18]: %timeit custom_fct(bigx)
```

334 μs \pm 2.63 μs per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

1.4 Function transformer with FFT

The following function is equivalent to the module of the output of a FFT transform. The matrix M_{kn} is defined by $M_{kn} = (\exp(-2i\pi kn/N))_{kn}$. Complex features are then obtained by computing MX . Taking the module leads to real features: $\sqrt{Re(MX)^2 + Im(MX)^2}$. That's what the following function does.

1.4.1 numpy implementation

```
[19]: def custom_fft_abs_py(x):
        "onnx fft + abs python"
        # see https://jakevdp.github.io/blog/
        # 2013/08/28/understanding-the-fft/
        dim = x.shape[1]
        n = numpy.arange(dim)
        k = n.reshape((-1, 1)).astype(numpy.float64)
        kn = k * n * (-numpy.pi * 2 / dim)
        kn_cos = numpy.cos(kn)
        kn_sin = numpy.sin(kn)
        ekn = numpy.empty((2,) + kn.shape, dtype=x.dtype)
        ekn[0, :, :] = kn_cos
        ekn[1, :, :] = kn_sin
        res = numpy.dot(ekn, x.T)
        tr = res ** 2
        mod = tr[0, :, :] + tr[1, :, :]
        return numpy.sqrt(mod).T

        x = numpy.random.randn(3, 4).astype(numpy.float32)
        custom_fft_abs_py(x)
```

```
[19]: array([[1.982739 , 1.1724371 , 3.4323769 , 1.172437 ],
            [2.764481 , 3.0285406 , 0.28028846, 3.0285406 ],
            [2.8741124 , 1.8547025 , 2.1338394 , 1.8547024 ]], dtype=float32)
```

1.4.2 ONNX implementation

This function cannot be exported into ONNX unless it is written with ONNX operators. This is where the numpy API for ONNX helps speeding up the process.

```
[20]: from mlproduct.numpy import onnxnumpy_default, onnxnumpy_np, NDArray
import mlproduct.numpy.onnx_impl as nxnp

def _custom_fft_abs(x):
    dim = x.shape[1]
    n = nxnp.arange(0, dim).astype(numpy.float32)
    k = n.reshape((-1, 1))
    kn = (k * (n * numpy.float32(-numpy.pi * 2))) / dim.astype(numpy.float32)
    kn3 = nxnp.expand_dims(kn, 0)
    kn_cos = nxnp.cos(kn3)
    kn_sin = nxnp.sin(kn3)
    ekn = nxnp.vstack(kn_cos, kn_sin)
    res = nxnp.dot(ekn, x.T)
    tr = res ** 2
    mod = tr[0, :, :] + tr[1, :, :]
    return nxnp.sqrt(mod).T

@onnxnumpy_default
def custom_fft_abs(x: NDArray[Any, numpy.float32],
                  ) -> NDArray[Any, numpy.float32]:
    "onnx fft + abs"
    return _custom_fft_abs(x)

custom_fft_abs(x)
```

```
C:\xavierdupre\_home_\GitHub\mlproduct\mlproduct\npy\numpy_onnx_impl.py:253:
UserWarning: npnx.dot is equivalent to npnx.matmul == numpy.matmul != numpy.dot
with arrays with more than 3D dimensions.
    warnings.warn(
```

```
[20]: array([[1.982739 , 1.1724371 , 3.4323769 , 1.172437 ],
            [2.7644813 , 3.0285406 , 0.28028846, 3.0285406 ],
            [2.8741124 , 1.8547025 , 2.1338396 , 1.8547025 ]], dtype=float32)
```

custom_fft_abs is not a function a class holding an ONNX graph. A method `__call__` executes the ONNX graph with a python runtime.

```
[21]: fonx = custom_fft_abs.to_onnx()
%onnxview fonx
```

```
[21]: <jyquickhelper.jsipy.render_nb_js_dot.RenderJsDot at 0x2b02e644eb0>
```

Every intermediate output can be logged.

```
[22]: custom_fft_abs(x, verbose=1, fLOG=print)
```

```
-- OnnxInference: run 26 nodes
Onnx-Shape(x) -> out_sha_0 (name='_shape')
+kr='out_sha_0': (2,) (dtype=int64 min=3 max=4)
Onnx-Gather(out_sha_0, init) -> out_gat_0 (name='_gather')
+kr='out_gat_0': () (dtype=int64 min=4 max=4)
Onnx-Reshape(out_gat_0, init_1) -> out_res_0 (name='_reshape')
+kr='out_res_0': (1,) (dtype=int64 min=4 max=4)
Onnx-ConstantOfShape(out_res_0) -> out_con_0 (name='_constantofshape')
+kr='out_con_0': (4,) (dtype=int64 min=1 max=1)
Onnx-CumSum(out_con_0, init_2) -> out_cum_0 (name='_cumsum')
+kr='out_cum_0': (4,) (dtype=int64 min=1 max=4)
Onnx-Add(out_cum_0, init_1) -> out_add_0 (name='_add')
+kr='out_add_0': (4,) (dtype=int64 min=0 max=3)
Onnx-Cast(out_add_0) -> out_cas_0 (name='_cast')
+kr='out_cas_0': (4,) (dtype=float32 min=0.0 max=3.0)
Onnx-Mul(out_cas_0, init_4) -> out_mul_0 (name='_mul')
+kr='out_mul_0': (4,) (dtype=float32 min=-18.84955596923828 max=-0.0)
Onnx-Reshape(out_cas_0, init_5) -> out_res_0_1 (name='_reshape_1')
+kr='out_res_0_1': (4, 1) (dtype=float32 min=0.0 max=3.0)
Onnx-Cast(out_gat_0) -> out_cas_0_1 (name='_cast_1')
+kr='out_cas_0_1': () (dtype=float32 min=4.0 max=4.0)
Onnx-Mul(out_res_0_1, out_mul_0) -> out_mul_0_1 (name='_mul_1')
+kr='out_mul_0_1': (4, 4) (dtype=float32 min=-56.548667907714844 max=-0.0)
Onnx-Div(out_mul_0_1, out_cas_0_1) -> out_div_0 (name='_div')
+kr='out_div_0': (4, 4) (dtype=float32 min=-14.137166976928711 max=-0.0)
Onnx-Unsqueeze(out_div_0, init_2) -> out_uns_0 (name='_unsqueeze')
+kr='out_uns_0': (1, 4, 4) (dtype=float32 min=-14.137166976928711 max=-0.0)
Onnx-Sin(out_uns_0) -> out_sin_0 (name='_sin')
+kr='out_sin_0': (1, 4, 4) (dtype=float32 min=-1.0 max=1.0)
Onnx-Cos(out_uns_0) -> out_cos_0 (name='_cos')
+kr='out_cos_0': (1, 4, 4) (dtype=float32 min=-1.0 max=1.0)
Onnx-Transpose(x) -> out_tra_0 (name='_transpose')
+kr='out_tra_0': (4, 3) (dtype=float32 min=-2.118224620819092
max=2.176269054412842)
Onnx-Concat(out_cos_0, out_sin_0) -> out_con_0_1 (name='_concat')
+kr='out_con_0_1': (2, 4, 4) (dtype=float32 min=-1.0 max=1.0)
Onnx-MatMul(out_con_0_1, out_tra_0) -> out_mat_0 (name='_matmul')
+kr='out_mat_0': (2, 4, 3) (dtype=float32 min=-2.9943528175354004
max=3.4323768615722656)
Onnx-Pow(out_mat_0, init_7) -> out_pow_0 (name='_pow')
+kr='out_pow_0': (2, 4, 3) (dtype=float32 min=0.0 max=11.781210899353027)
Onnx-Slice(out_pow_0, init_8, init_7, init_2) -> out_sli_0 (name='_slice')
+kr='out_sli_0': (1, 4, 3) (dtype=float32 min=0.0 max=0.20590990781784058)
Onnx-Slice(out_pow_0, init_2, init_8, init_2) -> out_sli_0_1
(name='_slice_1')
+kr='out_sli_0_1': (1, 4, 3) (dtype=float32 min=0.07856161892414093
max=11.781210899353027)
Onnx-Squeeze(out_sli_0, init_2) -> out_squ_0 (name='_squeeze')
+kr='out_squ_0': (4, 3) (dtype=float32 min=0.0 max=0.20590990781784058)
Onnx-Squeeze(out_sli_0_1, init_2) -> out_squ_0_1 (name='_squeeze_1')
+kr='out_squ_0_1': (4, 3) (dtype=float32 min=0.07856161892414093
max=11.781210899353027)
```

```

Onnx-Add(out_squ_0_1, out_squ_0) -> out_add_0_1    (name='_add_1')
+kr='out_add_0_1': (4, 3) (dtype=float32 min=0.07856161892414093
max=11.781210899353027)
Onnx-Sqrt(out_add_0_1) -> out_sqr_0    (name='_sqrt')
+kr='out_sqr_0': (4, 3) (dtype=float32 min=0.2802884578704834
max=3.4323768615722656)
Onnx-Transpose(out_sqr_0) -> y    (name='_transpose_1')
+kr='y': (3, 4) (dtype=float32 min=0.2802884578704834 max=3.4323768615722656)

```

```
[22]: array([[1.982739 , 1.1724371 , 3.4323769 , 1.172437  ],
            [2.7644813 , 3.0285406 , 0.28028846, 3.0285406 ],
            [2.8741124 , 1.8547025 , 2.1338396 , 1.8547025 ]], dtype=float32)
```

```
[23]: %timeit custom_fft_abs_py(x)
```

18.6 μ s \pm 581 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```
[24]: %timeit custom_fft_abs(x)
```

261 μ s \pm 8.92 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

Again the gap is less on bigger matrices. It cannot be faster with the default runtime as it is also using *numpy*. That's another story with *onnxruntime* (see below).

```
[25]: bigx = numpy.random.randn(10000, x.shape[1]).astype(numpy.float32)
      %timeit custom_fft_abs_py(bigx)
```

1.64 ms \pm 49.1 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

```
[26]: %timeit custom_fft_abs(bigx)
```

3.69 ms \pm 224 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

1.4.3 Using onnxruntime

The python runtime is using *numpy* but is usually quite slow as the runtime needs to go through the graph structure. *onnxruntime* is faster.

```
[27]: @onnxnumpy_np(runtime='onnxruntime')
      def custom_fft_abs_ort(x: NDArray[Any, numpy.float32],
                             ) -> NDArray[Any, numpy.float32]:
          "onnx fft + abs"
          return _custom_fft_abs(x)

      custom_fft_abs(x)
```

```
[27]: array([[1.982739 , 1.1724371 , 3.4323769 , 1.172437  ],
            [2.7644813 , 3.0285406 , 0.28028846, 3.0285406 ],
            [2.8741124 , 1.8547025 , 2.1338396 , 1.8547025 ]], dtype=float32)
```

```
[28]: %timeit custom_fft_abs_ort(x)
```

77.7 μ s \pm 44 μ s per loop (mean \pm std. dev. of 7 runs, 1 loop each)

onnxruntime is faster than *numpy* in this case.

```
[29]: %timeit custom_fft_abs_ort(bigx)
```

231 μ s \pm 48.8 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

1.4.4 Inside a FunctionTransformer

The conversion to ONNX fails if the python function is used.

```
[30]: from mlproduct.onnx_conv import to_onnx

tr = FunctionTransformer(custom_fft_abs_py)
tr.fit(x)

try:
    onnx_model = to_onnx(tr, x)
except Exception as e:
    print(e)
```

FunctionTransformer is not supported unless the transform function is of type `<class 'function'>` wrapped with `onnxnumpy`.

Now with the onnx version but before, the converter for FunctionTransformer needs to be overwritten to handle this functionality not available in [sklearn-onnx](#). These version are automatically called in function `to_onnx` from `mlproduct`.

```
[31]: tr = FunctionTransformer(custom_fft_abs)
tr.fit(x)

onnx_model = to_onnx(tr, x)
```

```
[32]: from mlproduct.onnxrt import OnnxInference

oinf = OnnxInference(onnx_model)
y_onx = oinf.run({'X': x})
y_onx['variable']
```

```
[32]: array([[1.982739 , 1.1724371 , 3.4323769 , 1.172437  ],
           [2.7644813 , 3.0285406 , 0.28028846, 3.0285406 ],
           [2.8741124 , 1.8547025 , 2.1338396 , 1.8547025 ]], dtype=float32)
```

```
[33]:
```