Operationalizing Deep Learning (CNN) Model using ONNX, Keras & Flask: Workshop

Data Science Meetup March 2019

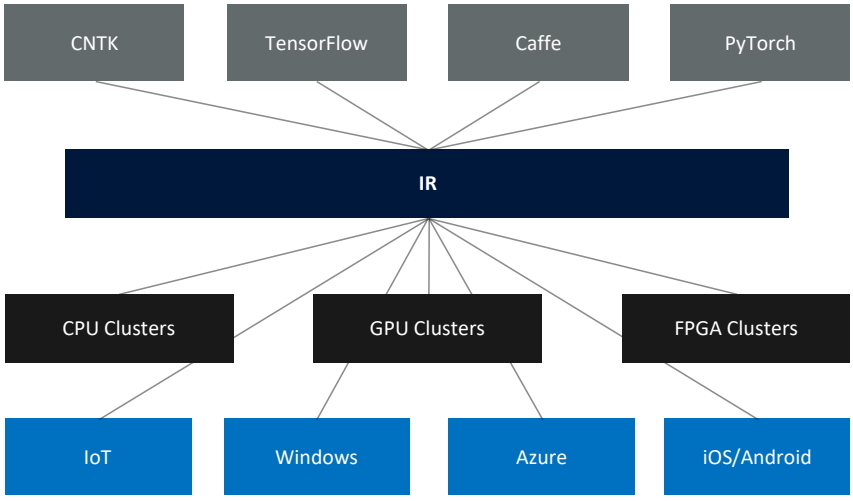
Table of Contents

What is ONNX?

ONNX (Open Neural Network Exchange) is an open format that represents deep learning models. It is a community project championed by Facebook and Microsoft. Currently there are many libraries and frameworks which do not interoperate, hence, the developers are often locked into using one framework or ecosystem. The goal of ONNX is to enable these libraries and frameworks to work together by sharing of models.

Why ONNX?

ONNX provides an intermediate representation (IR) of models (see below), whether a model is created using CNTK, TensorFlow or another framework. The IR representation allows deployment of ONNX models to various targets, such as IoT, Windows, Azure or iOS/Android.



The intermediate representation provides data scientist with the ability to choose the best framework and tool for the job at hand, without having to worry about how it will be deployed and optimized for the deployment target.

How ONNX?

ONNX provides two core components:

- **Extensible computational graph model**. This model is represented as an acyclic graph, consisting of a list of nodes with edges. It also contains metadata information, such as author, version, etc.
- **Operators.** These form the nodes in a graph. They are portable across the frameworks and are currently of three types.
    - Core Operators - These are supported by all ONNX-compatible products.
    - Experimental Operators - Either supported or deprecated within few months.
    - Custom Ops - which are specific to a framework or runtime.

Where ONNX

ONNX empathizes on reusability, and there are four ways of obtaining ONNX models

- Public Repository – https://github.com/onnx/models
- Custom Vision Services (https://azure.microsoft.com/en-gb/services/cognitive-services/custom-vision-service/)
- Convert to ONNX model (https://docs.microsoft.com/en-us/windows/ai/convert-model-winmltools)
- Create your own in DSVM or Azure Machine Learning Services

For custom models and converters, please review the currently supported export and import model framework (https://github.com/onnx/tutorials)

Step by Step Guide

In this workshop, you will learn the followings:

1. How to install ONNX on your Machine
2. Creating a Deep Neural Network Model Using Keras
3. Exporting the trained Model using ONNX
4. Deploying ONNX in Python Flask using ONNX runtime as a Web Service

By end of this workshop, you will have a general understanding of end-to-end process of how to operationalize a deep learning model using ONNX for handwritten digits image classification using the MNIST dataset.

Step1: Setting up the environment

Feel free to use your IDE of choice, however, PyCharm is a popular opensource editor, you can download and install it from the following URI (https://www.jetbrains.com/pycharm/)

It is recommended that you use a separate Conda environment for ONNX installation, this would avoid potential conflicts with the existing libraries. Note: If you do not have Conda installed, you can install it from the following URI: https://conda.io/docs/user-guide/install/download.html

Create Conda Environment

On terminal or CMD, type the following commands to create the Conda environment

```
conda create –n onnxenv python=3.6.6
```

Note, python 3.6.6 has been used to test the ONNX, you should be able to use other variants of Python 3.6.

Activate Newly Created Conda Environment

Once the environment has been created, you will need to active the environment.

```
Source activate onnxenv (Mac/Linux)

activate onnxenv (Windows)
```

```
```

Now it is time to install other libraries, including ONNX, Keras, etc.

Installing ONNX

The core python library for ONNX is called onnx (https://pypi.org/project/onnx/) and the current version is 1.3.0.

```
pip install onnx
note: if pip does not work, then type the following:  python –m pip install onnx
```

This will install the core library to your virtual Conda environment. You will also need to install onnxmltools

Installing ONNX Tools & runtime

```
pip install onnxmltools
Note: On Windows you can also install, winmltools
pip install winmltools
pip install onnxruntime
```

You will be using these tools for exporting and transforming ML models from one framework to another.

Installing TensorFlow/Keras

keras is a deep machine learning library that is capable of using various backends such CNTK, TensorFlow and Theano. To install Keras, type the following command.

```
pip install Keras
```

Step 2: Preparing the Dataset

In this example, you will be creating a Deep Neural Network for the popular MNIST dataset. The MNIST dataset (http://yann.lecun.com/exdb/mnist/) is designed to learn and classify hand written characters. For more information, see the following link.

Create a new Python file called `DnnOnnX.py` and then type the following Python codes:

```
import onnx
import winmltools
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras import backend
```

The above codes load all the related Keras and ONNX libraries. Then create a class called DnnOnnX and add two parameters to the constructer (see the code below).

```
class DnnOnnX:

    max_batch_size = 200
    number_of_classes = 10
    number_of_epocs = 20

    def __init__(self, batch_size, total_classes, epochs_size):

        self.max_batch_size = batch_size
        self.number_of_classes = total_classes
        self.number_of_epocs = epochs_size
```

To test creation of the class, type the following codes:

```
dnnOnnX = DnnOnnX (100, 10, 5)
print (dnnOnnX.number_of_epocs)
print (dnnOnnX.number_of_classes)
print (dnnOnnX.max_batch_size)

# this should produce 100, 10, 5

If you run into an error, try the following

pip install --upgrade --force-reinstall numpy==1.15.4
pip install --upgrade --force-reinstall pandas==0.22.0
```

Next, is to load the MNIST dataset, and split them between train and test sets. Type the following codes below, which will by default provide a dataset split of 60:40 (60% training, and 40% testing).

```
1 def prep_data(self, image_row_size, image_column_size):

2    input_shape= None

3    (train_features, train_label), (test_features, test_label) = mnist.load_data()

4    train_features = train_features.astype('float32')
5    train_features/=255
6    test_features = test_features.astype('float32')
7    test_features/=255

8    if backend.image_data_format() == 'channels_first':
9        train_features = train_features.reshape(train_features.shape[0], 1,
image_row_size, image_column_size)
10       test_features = test_features.reshape(test_features.shape[0], 1,
image_row_size, image_column_size)
11       input_shape = (1, image_row_size, image_column_size)
12    else:
13       train_features = train_features.reshape(train_features.shape[0],
image_row_size, image_column_size, 1)
14       test_features =test_features.reshape(test_features.shape[0],
image_row_size, image_column_size, 1)
15       input_shape = (image_row_size, image_column_size,1)

16    train_label = keras.utils.to_categorical(train_label, self.number_of_classes)
17    test_label = keras.utils.to_categorical(test_label, self.number_of_classes)

18    return train_features, train_label, test_features, test_label, input_shape
```

To test the data prep method, type the following:

```
dnnOnnX = DnnOnnX (500, 10, 2)

train_features, train_label, test_features, test_label, input_shape =
dnnOnnX.prep_data(28, 28)

print (train_features.shape)
print (train_label.shape)
print (test_features.shape)
print (test_label.shape)
print (input_shape)

# You should see the following output
#(60000, 28, 28, 1)
#(60000, 10)
#(10000, 28, 28, 1)
#(10000, 10)
#(28, 28, 1)
```

You should see an output of [60000, 28, 28] [60000,] [10000, 28, 28] [10000,], this indicates you have a training set of 60K of size 28*28 and testing set of 100K of 28*28 (line 4).

Since we are dealing with gray scaled image, there is only one channel. Lines 9-16 check to see if the channel is at the beginning (1st dimension of the vector) or at the end. You should see an output with added channel: (60000, 28, 28, 1) (60000,) (10000, 28, 28, 1) (10000,)

Lines 17-19 convert the categorical to binary class metrics. Your dataset is now ready to run in the DNN model.

Step 3: Convolutional Neural Network (Deep Learning)

In this example we are going to use Convolutional Neural Network to do the handwritten classification. So, the question is why we are considering CNN and not a traditional fully connected network. The answer is simple, if you have an image with N (width) * M (height) * F (filters) and is connected to H (hidden) nodes. Then the total number of network parameters would be:

$$P = (\boldsymbol{N} * \boldsymbol{M} * \boldsymbol{F}) * \boldsymbol{H},$$

where N= image width, M=image height and F = Filters, H = hidden nodes

So, for example, a color image with 1000 * 1000 with 3 filters (R, G, B), connected to 1000 hidden layers. This would produce a network with (P = 1000 * 1000 * 3 * 1000) **3bn parameters to train.**

It is difficult to get enough data to train such model without overfitting, and computational and memory requirements.

A Convolution Neural Network breaks this problem into two parts. It first detects various features such as edges, objects, faces etc., using convolutional layer (filters, padding and strides). Then it uses polling layer to reduce the dimensionality of the network. Secondly, it applies the classification layer as a fully connected layer to classify the model.

Convolutional Layer

The convolution layer is designed to detect features. It uses filers, padding and strides to achieve this. Typically, you would start with the larger height and width, and would gradually decrease through the convolutional layers. Whereas, the number of channels would typically increase throughout the network.

Filters

If you wanted to detect edges, such as vertical, you can use the following 3 * 3 filter. For example:

|1, 0, -1|
|1, 0, -1|
|1, 0, -1|

There are other filters for doing such as horizontal, sobel filter, Schorr filter, etc. These are typically created by Image processing specialist. In convolutional NN, these filter values are computed by the network and updated using backpropagation. It learns various filters, even low-level feature, which can be better than human beings. Then it applies to whole image.

Filter size generally tent to be odd numbers. If the filter is even, you would typically need an asymmetric padding. Odd filter provides a central position, which you can reference to.

Number of filters must match the number of channels. For example [6*6***3**] (img) [3*3*3] (filter)

Padding

When a filter is applied to an image, the overall dimension of the image shrinks. The more convolution layer you apply the smaller the image gets. Also, during the convolutional process, when using the sliding window, the pixels on the edge are used once, whereas the inner pixel are used more. This can lead to loss of information. To address these issues, you can pad the image by adding pixels.

What is the number of padding require in order for the output to be the same as the input?

The general formula for padding is as follows:

$$\boldsymbol{P} = \frac{\boldsymbol{F-1}}{\boldsymbol{2}}, \text{ where F = filter}$$

So, for example a [6*6] image with a filter of [3*3] would require: [3* 3] filter: (3-1)/2 = 1

Before Padding: [6 *6] (image) with [3*3] (filter) $\Rightarrow$ [4 * 4]

After Padding: [6 * 6] (image) + Padding (1) $\Rightarrow$ [8 * 8] (image) with [3 * 3] (filter) $\Rightarrow$ [6 * 6]

Terminology: Valid convolution $\Rightarrow$ means it is same as the original (no padding)
Same convolution $\Rightarrow$ means it has added padding

Strides

Strides define the step-over of the filter on the image.

Output size with stride is calculated as follows:

$$\boldsymbol{O} = \frac{\boldsymbol{N+2P-F}}{\boldsymbol{S}} + \boldsymbol{1} \ * \frac{\boldsymbol{M+2P-F}}{\boldsymbol{S}} + \boldsymbol{1}$$

where N = image width, M = image height, P = padding S=strides, F= filters

If the output is not an integer, then you take the floor (round down).

For example, an image with [7* 7] with filter [3*3] with Stride of 2 with no padding. The output is as follows:

(7 + (2 * 0) – 3) /2 = 2 $\Rightarrow$ [2 * 2]

So, the first convolutional Layer would look something like:

$$\boldsymbol{L1} = \boldsymbol{Relu}(\mathbf{[I]o[F]} + \boldsymbol{\beta}),$$

where, I =image, F = filter and $\beta = Bias$

The layer one activation will be used as the input to layer 2.

Pooling Layer

Pooling layer is used to reduce the size of the representation of the network and to speed up the computation and feature detects are more robust. One of the common approaches is using Max Pooling. It uses the same concept as strides, but instead of doing a matrix pairwise calculation, it takes the Max value. The reason behind it is the max value in a region of an image is considered to be a feature, hence preserving this region is more important than others.

$$\boldsymbol{P = F, S}$$
$$where,\ hypermeter\ F = \text{Filter size, and S= Strides}$$

Pooling does not have any hyper parameters for the gradient descent to learn. There is also an Average Pooling, where the average value is taken. Padding is not really used for Pooling layer.

Dropout Layer

Dropout improves the overfit issue of a Neural Network. It randomly switches of a number of activation neurons by setting them to zero. The dropout is layer is only used during the training. Dropout is given a value between 0-1, 1 being do not drop any. Typical choice tent to be 0.5.

Fully Connected Layer

The fully connected layer, also referred to as the classification layer, typically takes in a flatten input, where the data is presented as a one-dimensional vector, and applies the classification activation function such as SoftMax at the end. This layer produces output as a set of categories, with weighted values.

Defining the CNN model

```
1 def build_model(self, train_features, train_label, test_features, test_label,
input_shape):

2    print(train_features.shape, train_label.shape, test_features.shape,
test_label.shape)

3    keras_model = Sequential()
4    keras_model.add(Conv2D(16, strides= (1,1), padding= 'valid', kernel_size=(3,
3), activation='relu', input_shape=input_shape))
5    keras_model.add(Conv2D(32, (3, 3), activation='relu'))
6    keras_model.add(MaxPooling2D(pool_size=(2, 2)))
7    keras_model.add(Dropout(0.25))
8    keras_model.add(Conv2D(64, (3, 3), activation='relu'))
9    keras_model.add(MaxPooling2D(pool_size=(2, 2)))
10    keras_model.add(Dropout(0.25))
```

```
11    keras_model.add(Flatten())
12    keras_model.add(Dense(128, activation='relu'))
13    keras_model.add(Dropout(0.5))
14    keras_model.add(Dense(self.number_of_classes, activation='softmax'))

    keras_model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adadelta(),
                  metrics=['accuracy'])

    return keras_model
```

```
dnnOnnX = DnnOnnX (500, 10, 2)

train_features, train_label, test_features, test_label, input_shape =
dnnOnnX.prep_data(28, 28)

onnx_model = dnnOnnX.build_model(train_features, train_label, test_features,
test_label, input_shape)

print (onnx_model.get_config())
```

The above code defines a CNN model with 11 layers.

The first layer (line 4) of the Convolutional Neural Network consists of input shape of [28 ,28, 1], and uses 16 filters with size [3,3] and the activation function is Relu. This will produce an output matrix of [14,14,32].

The next layer (line 5) contains 32 filters with filter size of [3,3], this produces an output of [12,12,32]. The following layer applies a dropout (line 6), with {2,2}, which results in a matrix of [6,6,32].

Line 11 flattens the output, so it is one-dimensional vector, this then used as the input to the fully connected layer (line 12) with 128 nodes.

The model uses the categorical crossentropy as the loss function, with adadelta as the optimizer.

Train the model

The model is trained in a batched mode with a size of 250, and epochs of 500. The model is then scored using the test data.

```
def train_model(self, model):

    model_history = model.fit(train_features, train_label,
              batch_size=self.max_batch_size,
              epochs=self.number_of_epocs,
              verbose=1,
              validation_data=(train_features, train_label))
    score = model.evaluate(test_features, test_label, verbose=0)
    print('model loss:', score[0])
    print('model accuracy:', score[1])

    return model_history, model
```

You should see two outputs, one showing the loss value and other the accuracy. Overtime after a number of iterations and the loss value should decrease significantly and accuracy should increase.

To test the running of the model, please type the following:

```
dnnOnnX = DnnOnnX (500, 10, 5)

train_features, train_label, test_features, test_label, input_shape =
dnnOnnX.prep_data(28, 28)

onnx_model = dnnOnnX.build_model(train_features, train_label, test_features,
test_label, input_shape)

model_history, trained_model = dnnOnnX.train_model (onnx_model)
```

This will create a Keras model with a batch size of 500 and have 10 classes. It will run for a total epoch of 5. The model should produce accuracy of over 90s.

Exercise:

Please try the following to see what impact it has on the overall performance:

1. using various hypermeter
2. different batch size, epochs
3. layers to see the impact

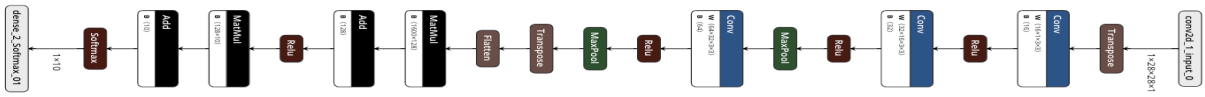
Step 4: Convert to ONNX Model

The winmltools module contains various methods for handing ONNX operations, such as save, convert and others. This module handles all the underlying complexity and provides seamless generation/transformation to ONNX model. The code below converts a Keras model into ONNX model, then it saves it as an ONNX file.

```
def convert_to_onnx(self, model, model_name):
    convert_model = winmltools.convert_keras(model, target_opset=7)
    winmltools.save_model(convert_model, model_name)
```

The name of the saved model is called mnist.onnx. This model can now be taken and deployed in an ONNX runtime environment.
To view the model, you can use Netron, by downloading the

https://github.com/lutzroeder/Netron



Step 5: Operationalize Model

In this section, we will first create a score file, that defines inference of the model. We use the Python Flask framework to deploy the model as a REST Webservice. The Webservice exposes a POST REST end point for client interaction.

Please add the following python libs

```
pip install numpyencoder
pip install opencv-python
```

Create a file called score.py, and type the following codes:

```
import json
import numpy
from numpyencoder import NumpyEncoder
import onnxruntime
import cv2
```

```
def init():
    global onnx_session
    onnx_session = onnxruntime.InferenceSession('model/mnist.onnx')

def run(raw_data):
    try:
        newimg = cv2.resize(raw_data, (int(28), int(28)))
        reshaped_img = newimg.reshape(1, 28, 28, 1)

        classify_output = onnx_session.run(None,
{onnx_session.get_inputs()[0].name:reshaped_img.astype('float32')})
        print (type(classify_output))
        print (classify_output)
        return json.dumps({"prediction":classify_output}, cls=NumpyEncoder)

    except Exception as e:
        result = str(e)
        return json.dumps({"error": result})
```

The code above initializes the onnx model using the init() method. The run method receives an image, which is resized to [28,28], and reshaped to a matrix of [1, 28, 28, 1]. The classify returns a json output with 10 classes with weighted distribution of value adding to 1.

To install flask, type the following:

```
pip install flask
```

Next is to install OpenCV. This library is used for image handling, such as resizing, cropping, etc.

```
pip install opencv_python
```

Create a file called onnxop.py and type the following:

```
from flask import Flask
from flask import request
import numpy as np
import cv2
import mlmodel.score

app = Flask(__name__)

mlmodel.score.init()

@app.route("/")
def intro():
    return "welcome to ONNX operationalize ML models"

@app.route("/api/classify_digit", methods=['POST'])
def classify():
```

```
    input_img = np.fromstring(request.data, np.uint8)
    img = cv2.imdecode(input_img, cv2.IMREAD_GRAYSCALE)
    classify_response = "".join(map(str, mlmodel.score.run(img)))

    return (classify_response)
```

The API listens on the following URI as a POST request:

http://localhost:5000//api/classify_digit

To run the Web Server locally, type the following commands on the terminal:

```
export FLASK_APP = onnxop.py
python –m flask run
```

This will start a server on port 5000. To test the Webservice endpoint, you can send a POST request, using Postman. If you do not have Postman installed, you can download it from the following URI.

https://www.getpostman.com/

Extra - Create Custom ONNX Model

The following code creates a custom ONNX model.

```
import onnx
from onnx import helper
from onnx import AttributeProto, TensorProto, GraphProto


# The protobuf definition can be found here:
# https://github.com/onnx/onnx/blob/master/onnx/onnx.proto


# Create one input (ValueInfoProto)
X = helper.make_tensor_value_info('X', TensorProto.FLOAT, [2, 1])

# Create one output (ValueInfoProto)
Y = helper.make_tensor_value_info('Y', TensorProto.FLOAT, [4, 3])

# Create a node (NodeProto)
node_def = helper.make_node(
    'Pad', # node name
    ['X'], # inputs
    ['Y'], # outputs
    mode='constant', # attributes
    value=1.5,
    pads=[0, 1, 0, 1],
    other_value =3.4
)

# Create the graph (GraphProto)
graph_def = helper.make_graph(
    [node_def],
    'test-model',
    [X],
    [Y],
)

# Create the model (ModelProto)
model_def = helper.make_model(graph_def, producer_name='onnx-example')

print('The model is:\n{}'.format(model_def))
onnx.checker.check_model(model_def)

print('The model is checked!')
onnx.save(model_def, "model.onnx")
```

(https://github.com/onnx/onnx/blob/master/docs/PythonAPIOverview.md).

Exercise:

1. **Change the input so that it takes 3 values**
2. **Add another attribute to the 'Pad' node.**

GitHub Source Code

Full source code is available on GitHub:

https://github.com/mufajjul/onnx-operationalisation