Custom deep learning layers support

@tableofcontents

@prev_tutorial{tutorial_dnn_javascript} @next_tutorial{tutorial_dnn_OCR}

Original author	Dmitry Kurtaev
Compatibility	OpenCV >= 3.4.1

Introduction

Deep learning is a fast growing area. The new approaches to build neural networks usually introduce new types of layers. They could be modifications of existing ones or implement outstanding researching ideas.

OpenCV gives an opportunity to import and run networks from different deep learning frameworks. There are a number of the most popular layers. However you can face a problem that your network cannot be imported using OpenCV because of unimplemented layers.

The first solution is to create a feature request at https://github.com/opencv/opencv/issues mentioning details such a source of model and type of new layer. A new layer could be implemented if OpenCV community shares this need.

The second way is to define a **custom layer** so OpenCV's deep learning engine will know how to use it. This tutorial is dedicated to show you a process of deep learning models import customization.

Define a custom layer in C++

Deep learning layer is a building block of network's pipeline. It has connections to **input blobs** and produces results to **output blobs**. There are trained **weights** and **hyper-parameters**. Layers' names, types, weights and hyper-parameters are stored in files are generated by native frameworks during training. If OpenCV mets unknown layer type it throws an exception trying to read a model:

```
Unspecified error: Can't create layer "layer_name" of type "MyType" in function getLayerInst
```

To import the model correctly you have to derive a class from cv::dnn::Layer with the following methods:

@snippet dnn/custom_layers.hpp A custom layer interface

And register it before the import:

@snippet dnn/custom_layers.hpp Register a custom layer

@note `MyType` is a type of unimplemented layer from the thrown exception.

Let's see what all the methods do:

- Constructor

@snippet dnn/custom_layers.hpp MyLayer::MyLayer

Retrieves hyper-parameters from cv::dnn::LayerParams. If your layer has trainable weights they will be already stored in the Layer's member cv::dnn::Layer::blobs.

- A static method `create`

@snippet dnn/custom_layers.hpp MyLayer::create

This method should create an instance of you layer and return cv::Ptr with it.

- Output blobs' shape computation

@snippet dnn/custom_layers.hpp MyLayer::getMemoryShapes

Returns layer's output shapes depends on input shapes. You may request an extra memory using `internals`.

- Run a layer

@snippet dnn/custom_layers.hpp MyLayer::forward

Implement a layer's logic here. Compute outputs for given inputs.

@note OpenCV manages memory allocated for layers. In the most cases the same memory can be reused between layers. So your `forward` implementation should not rely that the second invocation of `forward` will has the same data at `outputs` and `internals`.

- Optional `finalize` method

@snippet dnn/custom_layers.hpp MyLayer::finalize

The chain of methods are the following: OpenCV deep learning engine calls `create` method once then it calls `getMemoryShapes` for an every created layer then you can make some preparations depends on known input dimensions at cv::dnn::Layer::finalize. After network was initialized only `forward` method is called for an every network's input.

@note Varying input blobs' sizes such height or width or batch size you make OpenCV reallocate all the internal memory. That leads efficiency gaps. Try to initialize and deploy models using a fixed batch size and image's dimensions.

Example: custom layer from Caffe

Let's create a custom layer `Interp` from https://github.com/cdmh/deeplab-public. It's just a simple resize that takes an input blob of size `N x C x Hi x Wi` and returns an output blob of size `N x C x Ho x Wo` where `N` is a batch size, `C` is a number of channels, `Hi x Wi` and `Ho x Wo` are input and output `height x width` correspondingly. This layer has no trainable weights but it has hyper-parameters to specify an output size.

In example, ~~~~~~~~~~~~~~ layer { name: “output” type: “Interp” bottom: “input” top: “output” interp_param { height: 9 width: 8 } } ~~~~~~~~~~~~~~

This way our implementation can look like:

@snippet dnn/custom_layers.hpp InterpLayer

Next we need to register a new layer type and try to import the model.

@snippet dnn/custom_layers.hpp Register InterpLayer

Example: custom layer from TensorFlow

This is an example of how to import a network with tf.image.resize_bilinear operation. This is also a resize but with an implementation different from OpenCV's or `Interp` above.

Let's create a single layer network: ~~~~~~~~~~~~{.py} inp = tf.placeholder(tf.float32, [2, 3, 4, 5], ‘input’) resized = tf.image.resize_bilinear(inp, size=[9, 8], name=‘resize_bilinear’)~~~~~~~~~~~~ OpenCV sees that TensorFlow's graph in the following way:

```
node {
  name: "input"
  op: "Placeholder"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
}
node {
  name: "resize_bilinear/size"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_INT32
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_INT32
        tensor_shape {
          dim {
            size: 2
```

```
          }
        }
        tensor_content: "\t\000\000\000\010\000\000\000"
      }
    }
  }
}
node {
  name: "resize_bilinear"
  op: "ResizeBilinear"
  input: "input:0"
  input: "resize_bilinear/size"
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "align_corners"
    value {
      b: false
    }
  }
}
library {
}
```

Custom layers import from TensorFlow is designed to put all layer's `attr` into cv::dnn::LayerParams but input `Const` blobs into cv::dnn::Layer::blobs. In our case resize's output shape will be stored in layer's `blobs[0]`.

@snippet dnn/custom_layers.hpp ResizeBilinearLayer

Next we register a layer and try to import the model.

@snippet dnn/custom_layers.hpp Register ResizeBilinearLayer

Define a custom layer in Python

The following example shows how to customize OpenCV's layers in Python.

Let's consider Holistically-Nested Edge Detection deep learning model. That was trained with one and only difference comparing to a current version of Caffe framework. `Crop` layers that receive two input blobs and crop the first one to match spatial dimensions of the second one used to crop from the center. Nowadays Caffe's layer does it from the top-left corner. So using the latest version of Caffe or OpenCV you'll get shifted results with filled borders.

Next we're going to replace OpenCV's `Crop` layer that makes top-left cropping by a centric one.

- Create a class with `getMemoryShapes` and `forward` methods

@snippet dnn/edge_detection.py CropLayer

@note Both methods should return lists.

- Register a new layer.

@snippet dnn/edge_detection.py Register

That's it! We've replaced an implemented OpenCV's layer to a custom one. You may find a full script in the source code.