Converting Torch models to Caffe2 models

We have seen an overview of Caffe2 underlying implementation (here Caffe2 - Android). We went through the Caffe2 models and the way they are represented in memory. This exercise of conversion of a Torch model to Caffe2 models will help us gain a deep understanding of the concepts discussed previously.

The task at hand is given a pre-trained Torch model and its network parameters, how can we create its copy to run with a Caffe2 network. We need to first read the parameters from the Torch model and copy them to the Caffe2 models. The problem arises by both frameworks' interface. Torch codes are written using a scientific computing language called "Lua" (here Lua Programming Language) while Caffe framework is written completely in C++ and they provide supporting python wrappers for using their models. So first of all we need a common platform on which we can load/create models of both frameworks.

Reading Torch Parameters as Numpy n-D Arrays

Fortunately, we have few libraries which are developing support for Torch with Python using wrapper classes. Of the prominent ones, we have Py Torch, lutorpy and python-torchfile. Of all the three, only lutorpy was the one which was not under development any more. The library is quite stable as well as provide support for CPU as well as GPU in form of "CudaTensor" from Torch. It creates a bridge between Python/Numpy and Lua/Torch modules. Thus, we can use this library to load the Torch models and read their parameters for further use in a python file. We can also run the Torch networks in the python code itself, which makes it easy for debugging also.

As mentioned above, lutorpy provides support for "CudaTensor" from Torch. The weights of Torch layers are stored as "CudaTensor" for GPU or normal "Tensor" for CPU. We need to convert these Torch tensors to Numpy arrays first, in order to store them in Caffe2 supported format. Using lutorpy, we can import "torch" and "cutorch" library as we would in a lua environment, in python, and use the function "asNumpyArray()" to convert the corresponding tensors to Numpy array.

Converting Torch tensor to numpy array:

```
import lutorpy as lua
import numpy as np

require("torch")
t = torch.randn(10,10)
print(t)
arr = t.asNumpyArray()
print(arr)
```

Converting cudaTensor to numpy array:

```
import lutorpy as lua
import numpy as np

require('cutorch')
cudat = torch.CudaTensor(10,10)
print(cudat)
arr = cudat.asNumpyArray()
print(arr)
```

Besides the conversion of array from one type to another, you can basically do most of the things that you do in Lua/Torch environment in Python environment using Numpy. Lutorpy provides a very good primer on its repository for Torch tasks you can do in Python using it. You can check it out here Github Lutorpy Repository.

Caffe2 model as copy of Torch model

Now since we can read Torch models in python, the best way to copy a Torch model in Caffe2 is to create the network in Caffe2 using Caffe python wrappers and then replace the weights of the network. To do this, we need to know how the Caffe2 network parameters are stored. In the Caffe2 - Android overview, we saw that the weights are stored as "blobs" in the workspace. Thus, we need to first create a network and then insert our own Torch models weight in the workspace corresponding to the layer to which they belong. Hence, we have two steps in creating the Caffe2 replica of the Torch models. First, we need to create a "shell network" mimicking the Torch model network and second, we need to replace its weights with Torch model weights.

We need to first initialize a Caffe2 network. Since, we already have the pre-trained weight, the task becomes easier. To create the network follow the given steps:

1. Create a random input data array corresponding to the expected input of the test network.

   ```
   in_data = np.random.randn(1, 8, 24, 32).astype(np.float32)
   ```

2. Save this input data in the workspace using the "FeedBlob" function of "workspace" class of the Caffe2 library.

   ```
   workspace.FeedBlob('data', in_data)
   ```

3. Initialize a dummy Caffe2 network. We are using the "model_helper" class provided by Caffe2 library.

   ```
   arg_scope = {"order": "NCHW"}
   caffe_model = model_helper.ModelHelper(
       name=output_deploy, arg_scope=arg_scope, init_params=False)
   ```

4. Now, we add the model layers to the dummy network. To do this, we first load the Torch model, and extract its layers. We iterate over each of the model's layers and create a corresponding layer in our dummy Caffe2 network. We also initialize the network with the Torch layers' parameters while creating. An example of adding a convolution layer is shown.

```
inp_blob = data
torch_model = torch.load(in_model)
module = torch_model._get(4)
module_type = str(torch.typename(module)).split('.')[1]

if module_type == 'SpatialConvolution':
 # convolution = 3 (keeping track of number of convolution layer
already in the model)
 convolution += 1
    outp_blob = 'conv' + str(convolution)

    # reading the parameters of torch layer
    w = module.weight.asNumpyArray()
    d_in = module.nInputPlane
    d_out = module.nOutputPlane
    k = module.kH
    p = module.padH
    s = module.dH

    # initializing the caffe2 layer with same parameters,
    # we only need to replace the weights once the network is created
    outp_blob = brew.conv(caffe_model, inp_blob, outp_blob,
dim_in=d_in, dim_out=d_out, kernel=k, stride=s, pad=p)
```

The layer in Caffe2 are also saved as blobs in the workspace. We can see, using lutorpy library, we can extract the layer parameters like layer type, input dimensions, kernel shape, padding etc. as we can in lua. Thus, it makes our task much easier. The added advantage is that now we know the name of each layer we create in the Caffe2 model. This is of utmost importance as the weights and biases of the corresponding layer are saved as "layerName#_w" and "layerName#_b" respectively, where # stands for the count of the same type of layer already in the model. Hence, if you are adding 4th convolution layer in your model, the layer will be named as "conv4" and its parameters will be "conv4_w" and "conv4_b", and all **three is stored as blobs of the same name in the workspace.**

5. Now we have our network defined, we run it once to create the complete computational graph and initialize the network with random weights and biases. Since we are not training, we will just run the network once, to create the graph using the Caffe2 function "RunNetOnce" from the "workspace" class.

```
workspace.RunNetOnce(caffe_model.param_init_net)
```

As a result, all the weights and biases as well as the layers themselves will be created in the workspace and saved as blobs. We can check the created network parameters by checking the blobs in the workspace using the following code snippet.

```
print("Current blobs in the workspace: {}".format(workspace.Blobs()))
```

6. The only step remaining to make a complete replica of the Torch network is to replace the weights. Since, now we know the name of the layers and its parameter of the Caffe2 layer, and we can read the weights of the Torch layers using lutorpy, we again iterate over the Torch layers, and replace weight based on the type of layer encountered, keeping count of each type of layer. For example to replace a convolution layer we do so in this fashion.

```
module = torch_model._get(4)
module_type = str(torch.typename(module)).split('.')[1]

if module_type == 'SpatialConvolution':
 # convolution = 3 (keeping track of number of convolution layer
already in the model)
    convolution += 1
    outp_blob = 'conv' + str(convolution)

    # create the output blobs name based on the custom layer name
given
    outp_blob_w = outp_blob + '_w'
    outp_blob_b = outp_blob + '_b'

    # Weights replacement step
    workspace.FeedBlob(outp_blob_w, module.weight.asNumpyArray())
    workspace.FeedBlob(outp_blob_b, module.bias.asNumpyArray())
    print('Convolution layer. Weights replaced.')
```

We define the name of the wieghts and biases in the workspace corresponding to that layer, and replace it with the corresponding weights and biases of the same layer from Torch model.

7. Finally we create the init_net and predict_net objects required to export the models as protobufs using the "mobile_exporter" helper class "Export" from Caffe2 library, found here Mobile_Exporter.
8. We write the corresponding init_net, predict_net and prtotxt file on the disk, to use the network for predictions based on input data.

```
with open(os.path.join(root_folder, output_prototxt), 'w') as fid:
    fid.write(str(caffe_model.net.Proto()))
with open(os.path.join(root_folder, output_predict_net), 'w') as fid:
    fid.write(predict_net.SerializeToString())
with open(os.path.join(root_folder, output_init_net), 'w') as fid:
    fid.write(init_net.SerializeToString())
```

The complete implementation of converting one such model, the SpyNet Torch models to Caffe2 models can be found here emSpyNet_Torch_to_Caffe2_convertor. The Caffe2 directory in this implementation is a clone of the Caffe2 Repository.