CNNdroid

Open Source Library for Running Trained Convolutional Neural Networks on Android

Complete Developers' Guide and Installation Instructions

Contents

Part I. Network Training and Parameters Reformatting

1. Supported Layers Glossary

The list of currently supported layers and their required parameters are available below:

1) Convolution Layer

- layer parameters file name
- pad
- stride
- group

2) Pooling Layer

- pool type:
  - max
  - mean
- kernel size
- pad
- stride

3) Local Response Normalization Layer

- local size
- alpha
- beta
- normalization region:
  - across channels

4) Fully-connected Layer

- layer parameters file name

5) Nonlinear Layer[1]

- Supported type[2]:
  - Rectified Linear Unit

6) Softmax Layer

7) Accuracy and Top-k

- layer parameters file name
- top-k

[1] No parameter is required.

[2] More types will be added in the future.

2. Supported CNN Libraries and Model Parameters Regeneration

In order to implement your trained CNN model in CNNdroid, network parameters must be serialized with a specific format using MessagePack[3]. Therefore, we have prepared a number of scripts for the available state-of-the-art CNN libraries, which would regenerate the network parameters in the required format. The list of supported CNN libraries is as follows:

1) Caffe[4] (SavePycaffeModelinMessagePack.py)

In order to convert the files, you must set the following variables in the code properly:

- `MODEL_FILE`: directory of the Caffe model file (*.caffemodel* file)
- `MODEL_NET`: directory of the network definition file (*.prototxt* file). Note that this *.prototx* file must only contain the network structure and no other information. For instance, for the MNIST example of Caffe, only the `'lenet.prototxt'` file is required, and other files such as `'lenet_solver.prototxt'` are not needed.
- `SAVE_TO`: the directory where you want to save the MessagePack files
- `GET_PARAMS`: set it true if you want to get the parameters of the convolution and fully-connected layers
- `GET_BLOBS`: set it true if you want to get the blobs of the network layers
- `CAFFE_ROOT`: the root directory where Caffe is installed

2) Theano[5] (SaveTheanoModelinMessagepack.py)

In the provided Python script, there are two functions for saving the parameters. In order to regenerate the network parameters with the required format, you need to pass the weight and bias parameters of each convolution and fully-connected layer to the appropriate function, and they will be saved in the `'SAVE_TO'` directory. Note that you will use the same function for both of the convolution and fully-connected layers. You must set the following variables in the code properly:

- `SAVE_TO`: the directory where you want to save the MessagePack files
- `save_numpy_parameters`: use this function if your network parameters are numpy ndarrays. The string variable `'layer_name'` will be used in the file name of the regenerated parameters.

[3] http://msgpack.org/index.html
[4] http://caffe.berkeleyvision.org/
[5] https://github.com/Theano/Theano

- save_tensor_parameters: use this function if your network parameters are Theano Tensors. The string variable 'layer_name' will be used in the file name of the regenerated parameters.

3) **Torch[6] (SaveTorchModelinMessagepack.lua)**
   In order to use this script, the following steps must be followed:
   a. Train your network using Torch.
   b. Save your trained Torch model by using: 'torch.save(filename, model)'
   c. Initialize path variables at the beginning of the provided script:
      - load_path: path to load Torch NN model
      - save_path[7]: the directory where you want to save the MessagePack files
   d. Run the script.
   e. The new model parameters will be available in the 'SAVE_TO' directory.

NOTES:

- All of the mentioned scripts are available at **'Parameter Generation Scripts'** directory.
- In the **'Parameter Generation Scripts'** directory, a number of examples on how to deploy these scripts are provided in a zipped file.
- If you have trained your network using a different library, you need to regenerate the network parameters in the supported format. In order to do so, please refer to the appendix to attain the required instructions.

3. Writing a Network Definition Text File in a Specific Format (Net Structure File)

In order to define your CNN model structure in CNNdroid, you need to create a text file containing the layers structure and parameters information.
The mentioned file consists of the following items:

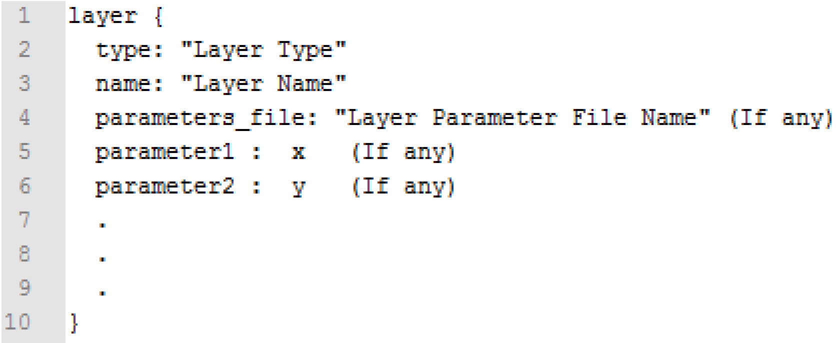
1) root_directory: root directory of the network layers parameters
2) allocated_ram: the amount of RAM (in megabytes) that you want to allocate to your network parameters. This amount of RAM will be used to hold all or some parts of the parameters during the whole time of application run. We have set the

[6] http://torch.ch/

[7] Note that the provided directory for saving the regenerated parameters should actually exist, otherwise 'bad argument' error will occur.

maximum value to 400[8]. For more information, refer to the memory management section of the appendix.

3) `auto_tuning`: this parameter determines whether or not the auto-tuning process must be done when the app runs for the first time. Its possible values are `on` and `off`. Refer to the auto-tuning section of the appendix for more details.
4) `execution_mode`: this parameter defines the network computations method:
   - `sequential`: sequential method for network computations over CPU
   - `parallel`: parallel method for network computations over GPU (using RenderScript) and CPU (using multithreading)
5) `layer`: Next, you need to define each layer of the model in order. The required parameters of each layer must be provided in this section. A general form of the layer definition item is similar to:



```
layer {
  type: "Layer Type"
  name: "Layer Name"
  parameters_file: "Layer Parameter File Name" (If any)
  parameter1 :  x   (If any)
  parameter2 :  y   (If any)
  .
  .
  .
}
```

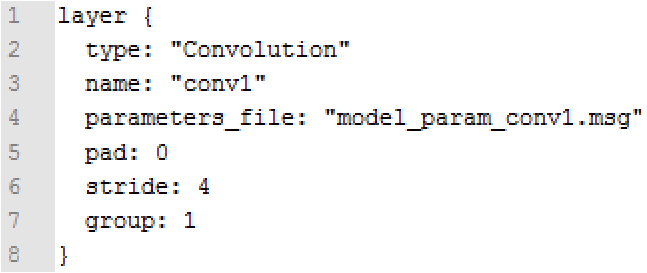
As you can see, the `'type'` and `'name'` of each layer are required. The necessity of other parameters depends on the layer type.
The following table specifies the value of the `'type'` variable for each layer:

Layer	Type
Convolution	“Convolution”
Pooling	“Pooling”
Local Response Normalization	“LRN”
Fully-connected	“FullyConnected”
Softmax	“Softmax”
Accuracy and Top-k	“Accuracy”
Nonlinear (Rectified Linear Unit)	“ReLU”

[8] Android has a limitation of 512 MB for the maximum allocated RAM per application.

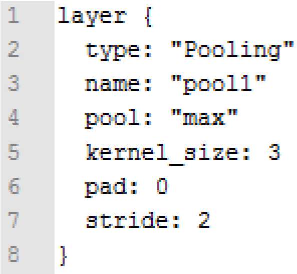
Below, you can see a template of layer definition for all of the supported layers:

- **Convolution:**



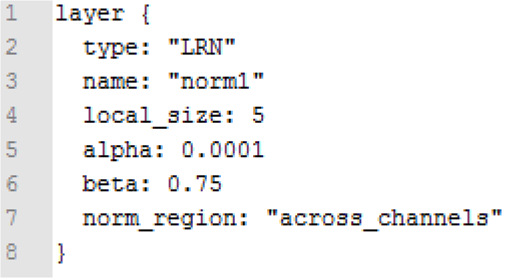
```
layer {
  type: "Convolution"
  name: "conv1"
  parameters_file: "model_param_conv1.msg"
  pad: 0
  stride: 4
  group: 1
}
```

- **Pooling:**



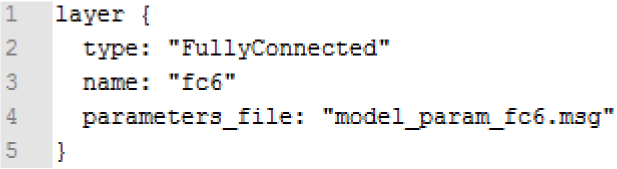
```
layer {
  type: "Pooling"
  name: "pool1"
  pool: "max"
  kernel_size: 3
  pad: 0
  stride: 2
}
```

- **Local Response Normalization:**



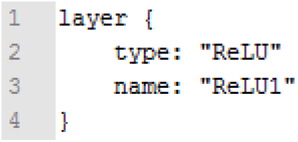
```
layer {
  type: "LRN"
  name: "norm1"
  local_size: 5
  alpha: 0.0001
  beta: 0.75
  norm_region: "across_channels"
}
```

- **Fully-connected**



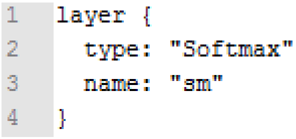
```
layer {
  type: "FullyConnected"
  name: "fc6"
  parameters_file: "model_param_fc6.msg"
}
```

- **Nonlinear (Rectified Linear Unit):**



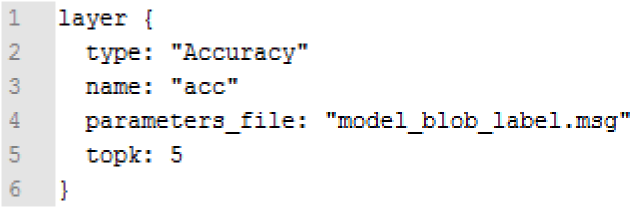
```
layer {
    type: "ReLU"
    name: "ReLU1"
}
```

- **Softmax**



```
layer {
  type: "Softmax"
  name: "sm"
}
```

- **Accuracy and Top-k**



```
layer {
  type: "Accuracy"
  name: "acc"
  parameters_file: "model_blob_label.msg"
  topk: 5
}
```

NOTES:

- A number of network definition file examples are available at **'NetFile Examples'** directory, which also includes the structure of AlexNet[9].
- `‘root_directory’`, `‘allocated_ram’`, `‘auto_tuning’`, and `‘execution_mode’` must appear before the layers definition in the net structure file.
- The order of layers definition must be exactly the same as the order their appearance in the network.
- The order of the parameters of each layer is not important, but all of them must be defined.
- The text of the network structure definition file is not case-sensitive (except the directory and file names).

[9] http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

Part II. Adding the Library to Your Application

In this section, comprehensive instructions for adding CNNdroid library to your application will be provided:

1) Create your project in Android Studio[1] , and go to the root directory of your project. (The minimum required Android SDK version is 21.0 (Lollipop)[1])
2) Go to the **'Source Package'** directory in the provided package.
3) Copy **'java'** and **'rs'** folders, and paste them in the `'app/src/main/'` directory of your project. (You need to merge the **'java'** folders.)
4) Copy **'libs'** folder and paste it in the `'app/'` directory of your project. (You need to merge the **'libs'** folders.)
5) Now open your project in Android Studio. You should run your application without any problem. (If you face any errors while building your project, you need to either sync Gradle or restart Android Studio.)
6) Now you need to edit `'AndroidManifest.xml'` file to provide the required permissions to your application. Open `'AndroidManifest.xml'` file and follow these steps:
   a. Add the following line before the `'<application'` section:

   ```
   <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
   ```

   b. Add the following line after the `'<application'` section:

   ```
   android:largeHeap="true"
   ```

   Now your Manifest file should look like:

   ```
   <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

   <application
       android:largeHeap="true"
   ```

   Note that in Android 6 (Marshmallow), you also need to provide the mentioned permissions in the application manager of your smartphone settings.
7) Now, you are ready to start programming.

NOTE:

- Demo Android applications have been prepared in the **'Demo Android Applications'** directory.

[1] If you have written your project in Eclipse, you can easily import it to Android Studio by following the instructions at http://developer.android.com/sdk/installing/migrate.html

[1] If you face "`llvm-rs-cc.exe finished with non-zero exit value`" error while building your project, copy `*.dll` files from "`build-tools/<SDK Version>/lib/`" to "`build-tools/<SDK Version>/`", so that `llvm-rs-cc.exe` and the `dll`s it needs are in the same directory.

Part III. Interfacing Guide of the Library through the Android Application

After completing Parts I and II and copying the regenerated network model parameters and the definition file to the smartphone memory[1] , you need to follow the[2] below-mentioned steps in order to do computations over user's inputs:

1) Import the main class of the library:

```
import network.CNNdroid;
```

2) Create a ‘RenderScript’ object in the ‘MainActivity’ of your project:

```
RenderScript myRS = RenderScript.create(this);
```

3) Construct a ‘CNNdroid’ object with the following parameters:

```
CNNdroid myConv = new CNNdroid(myRS, netDefDir);
```

- `RenderScript myRS`: the created RenderScript object
- `String netDefDir`: directory of the network definition file

4) Prepare your input to the network. You have two input options:
   - A single object in the form of ‘float[][][]’.
   - A batch of objects in the form of ‘float[][][][]’.

   Your image inputs should have the following order:
   `[Images][Channels][Height][Width]`

5) Call the Compute method of the library:

```
Object output = myConv.compute(input);
```

6) Get the result of computations as an ‘Object’ when finished.

NOTES:

- When the application runs for the first time, the runtime will be longer than usual since auto-tuning is being performed. For more information, refer to the auto-tuning section of the appendix.
- Computations of the network for a batch of inputs will result in higher performance compared to a single input.
- The network parameters may be loaded during construction of a ‘CNNdroid’ object according to the ‘allocated_ram’ defined in the network structure file. For more information, refer to the memory management section of the appendix.

[1] Should be copied to the root[2]directory defined in the network definition file.

- No normalization is applied to the user's input, and this operation should be handled by developers.
- The result of computations is returned as an '`Object`', and the casting operation should be done by developers according to the output of the last layer of the network.
- Developers can observe the run time and parameters load time of each layer in the log section and under the '`CNNdroid`' label.
- A template for the Android Studio project has been prepared in the **'Android Studio Project Template'** directory. The mentioned steps are implemented in the '`onCreate()`' method of the '`MainActivity`' of the application.
- Demo Android applications have been prepared in the **'Demo Android Applications'** directory.

Part IV. Appendix

In this section, brief explanations about the Java and RenderScript codes, memory management, and auto-tuning are provided:

A.1 "layers" Package

Each class in the 'layers' package is responsible for the calculations of a specific layer.

A.1.1. "LayerInterface" Interface

This interface is implemented in all classes of the 'layers' package except the CPU-multithreaded classes. It only contains a 'compute' method which is responsible for the mathematical computations of a layer.

A.1.2. "Accuracy" Class

This class implements the accuracy layer, which is responsible to calculate the accuracy of the prediction among the top k choices of each image. The prediction for each image is true or false according to its top k choices, and the accuracy is reported for the batch of the input images. Note that in order to use this class, you must provide a MessagePack file containing the labels of the input images. The labels are stored in an n×1×1×1 Float32 array.

A.1.3. "Convolution" Class

This class implements several methodologies for the computations of a convolution layer. The convolution is computed based on the rolled method (the typical method of computation) rather than the unrolled method (where the convolution is transformed to a matrix multiplication). Unrolled methodology is deprecated in that it usually requires large amounts of memory.

The computations could be performed in either sequential or parallel ways. The parallel methods are implemented by RenderScript and are of various types. The name of all parallel convolution methods in the code contain 'InFsOutFt', in which s and t denote respectively the number of input and output Float32 numbers which are processed in each parallel thread.

Here is the list of all methodologies implemented in this class:

- Sequential rolled convolution
- Parallel rolled convolution (InF4OutF1)
- Parallel rolled convolution (InF4OutF2)
- Parallel rolled convolution (InF4OutF4)
- Parallel rolled convolution (InF4OutF8)
- Parallel rolled convolution (InF8OutF1)

- Parallel rolled convolution (InF8OutF2)
- Parallel rolled convolution (InF8OutF4)
- Parallel rolled convolution (InF8OutF8)

If a non-linear layer follows the convolution layer, it can be merged to this layer (without adding a new layer) using the `‘nonLinear’` and `‘nonLinearType’` fields of this class.

Note that the convolution layer requires a MessagePack file containing the weights followed by the biases of this layer. Weights are stored in a 4D Float32 array whose dimensions, from the highest to the lowest, correspond to the images (or frames), channels, height, and width respectively. Biases are stored in a 1D Float32 array.

A.1.4. “FullyConnected” Class

This class implements several sequential and parallel methodologies for the computations of a fully connected layer. Similar to the convolution layer, the parallelism of the fully connected layer is done via RenderScript, and the `‘InFsOutFt’` approach is adopted.

The implemented methodologies for the computations of the fully connected layer are as follows:

- Sequential fully connected
- Parallel fully connected (InF4OutF1)
- Parallel fully connected (InF8OutF1)

If a non-linear layer follows the fully connected layer, it can be merged to this layer (without adding a new layer) using the `‘nonLinear’` and `‘nonLinearType’` fields of this class.

Note that the fully connected layer requires a MessagePack file containing the weights followed by the biases of this layer. Weights are stored in a 1D Float32 array. In fact, this array is flattened from a 2D Float32 array whose higher and lower dimensions correspond to the height and width respectively. Biases are also stored in a 1D Float32 array.

A.1.5. “LocalResponseNormalization” Class

This class implements the local response normalization layer in sequential and CPU-multithreaded manners. The CPU-multithreaded manner is implemented with the aid of an additional class, that is, `‘MultiThreadLrn’`.

A.1.6. “NonLinearClass” Class

Although a non-linear layer could be merged to its previous layer, this class implements an independent non-linear layer.

A.1.7. “Pooling” Class

This class implements the pooling layer in sequential and CPU-multithreaded manners. The CPU-multithreaded manner is implemented with the aid of an additional class, that is, ‘MultiThreadPool’.

If a non-linear layer follows the pooling layer, it can be merged to this layer (without adding a new layer) using the ‘nonLinear’ and ‘nonLinearType’ fields of this class.

A.1.8. “Softmax” Class

This class implements the softmax layer.

A.2. “network” Package and Memory Management

This package contains a class, ‘CNNdroid’, which is responsible to build a neural network and perform its computations. When an object of ‘CNNdroid’ is created, the network definition file is read and parsed, and the layers of this network are created and stored in a list. By calling the ‘compute’ method of this object, the ‘compute’ methods of all the layers will be called in order.

In the ‘preParse’ method of the ‘CNNdroid’ class, based on ‘allocated_ram’ which is specified in the network structure file, some layers are selected whose parameters will be loaded into RAM and remain in it during the whole duration of application run in a way that the specified RAM is filled as much as possible. In this way, the initially fetched parameters are not required to be reloaded each time the ‘compute’ method is called, thereby reducing the overall computations duration.

A.3. “messagepack” Package

This package contains a class, ‘ParamUnpacker’, which is responsible for unpacking of the MessagePack files. This class depends on the following libraries:

- javassist-3.16.1-GA.jar
- json-simple-1.1.1.jar
- junit-4.8.2.jar
- msgpack-0.6.8.jar

Note that in the `‘msgpack-0.6.8.jar’` library, the `‘arraySizeLimit’` field in the `‘org.messagepack.unpacker.AbstractUnpacker’` class is manipulated in order not to cause problems for large arrays.

A.4. “numdroid” Package

This package contains a class, `‘MyNum’`, which is responsible for the necessary mathematical calculations on the one- or multi-dimensional arrays. In fact, `‘numdroid’` is a much limited Java version of `‘numpy’` in Python.

A.5. Auto-tuning

Here, auto-tuning means finding the best value for `s` and `t` in `‘InFsOutFt’` approaches, which are adopted in the convolution and fully-connected layers. If the `‘auto_tuning’` parameter is set to `on` in the network structure definition file, when the application runs for the first time, all the `‘InFsOutFt’` approaches are tested for each of the convolution and fully-connected layers in order to find the best value of `s` and `t`. Therefore, when the application runs for the first time after `‘auto_tuning’` has been set to `on`, the computation takes a longer time than usual since auto-tuning is being performed. The auto-tuning result of each layer will be saved in the `‘root_directory/CNNdroid_Tuning’` directory for the future runs.