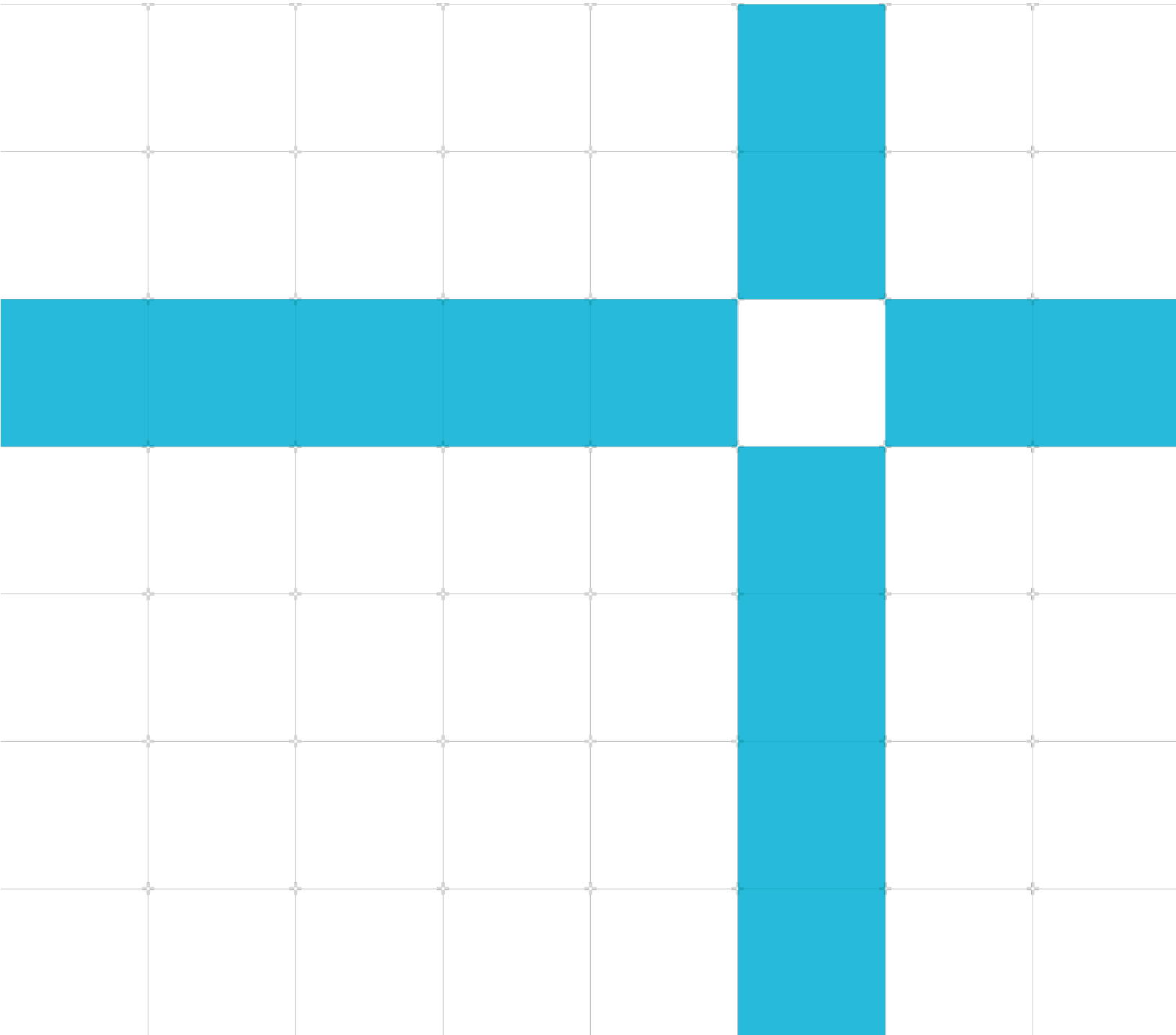
arm

Implement a neural style transfer on Android with Arm NN APIs

Non-Confidential

Issue 1.0
Doc ID: 102079



Implement a neural style transfer on Android with Arm NN APIs

Release information

Document history

Issue	Date	Confidentiality	Change
01	7th April 2020	Non-Confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

Web Address

www.arm.com

Contents

1 Overview

In this guide, we will show you how to build a style transfer Android application with Arm NN APIs.

1.1 Before you begin

To work through this guide, you will need the following resources:

- An Android device running Android 9 or 10
- An Android device that supports Camera 2 SDK. You can check their **online guide** to confirm whether your device supports Camera 2 SDK.
- Android Studio, including v24 or higher of the SDK build tools

2 What is neural style transfer?

Neural style transfer is a technique that uses two images: A content image and a style image. The style image might be, for example, an artwork by a famous painter. A neural style transfer copies the texture, color, and other aspects of the style image and applies them to the content image. You can see an example of a content image, a style image, and a generated image here:

Images courtesy of Gatys et al., A Neural Algorithm of Artistic Style

Neural style transfer uses a pre-trained Convolutional Neural Network (CNN). Using your content image and your style image, you generate a new image that blends the content image and the style image. You begin with a simple white noise image, or use the content image or style image for optimization efficiency. Then you process the content image, style image, and generated images through the pre-trained neural network. Finally, you calculate loss functions at different layers.

Related information includes a link to an article called Intuitive Guide to Neural Style Transfer, which includes more detail.

There are three types of loss functions:

- Content loss function
- Style loss function
- Total loss function

Let's look at each in turn.

The content loss function ensures that the content present in the content image is captured in the generated image. In a multiple layer CNN, lower layers are more focused on individual pixel values. Higher layers capture information about content. This means that we use the top CNN layer to define the content loss function in our illustration.

The style loss function ensures that the correlation of activations in all the layers is similar between the style image and the generated image.

The total loss function is the weighted sum of the content cost function and the style loss functions for the generated image. The weights are user-defined hyperparameters that control the amount of content and style that is injected into the generated image. Once the loss is calculated, it can be minimized using backpropagation. Backpropagation optimizes the randomly generated image into a piece of art.

3 What are the Arm NN SDK and the Arm NN APIs?

Arm NN SDK is a set of open-source Linux software tools that enables machine learning workloads on power-efficient devices. In the application that we use in this guide, we use Arm NN to enhance performance on the Arm architecture. The inference engine provides a bridge between existing neural network frameworks and Arm Cortex-A CPUs, Arm Mali GPUs, and NPUs.

3.1 Android Neural Networks API and TensorFlow Lite delegate

The Android Neural Networks API (NNAPI) is an Android C API that is designed to run computationally intensive operations for Machine Learning on mobile devices.

NNAPI supports various hardware accelerations. NNAPI uses TensorFlow as a core technology. If you build your mobile app with TensorFlow, your app gets the benefits of hardware acceleration through the NNAPI. NNAPI abstracts the hardware layer for ML inference. Arm NN works with Android NNAPI to target Arm processors, enabling exponential performance boosts.

A TensorFlow Lite delegate is a way to delegate part or all of graph execution to another executor. Running computationally intensive NN models on mobile devices is resource demanding on mobile CPUs. This means that devices with hardware accelerators provide better performance and higher energy efficiency through NNAPI.

TensorFlow Lite uses an NNAPI delegate to access NNAPI. You can access their **open source code**.

4 Looking at the Android code

In this section of the guide, we will explore the Android source code. You can look at the Android source code for this guide.

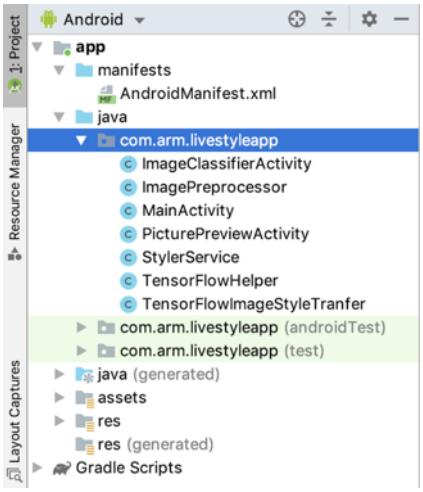
Note: We used pre-trained models and made changes to the model architecture, so that the architecture is fully compatible with Arm NN operators. The changes that we made to the model include:

- Replacing all reflection padding with same padding
- Replacing all instance normalization layers with batch normalization layers
- In unpooling layers, using bilinear resize operation instead of the nearest neighbor resize operation
- In the first Conv2D layer, using valid padding instead of same padding

4.1 Style transfer code

To implement the style transfer code, follow these steps:

1. Import the Live Style project into Android Studio. The following screenshot shows the project structure:



   Our style transfer code is implemented in the `doStyleTransfer()` function in `TensorFlowImageStyleTransfer.java`.

2. Convert the image in this function to data that the model can understand, as you can see in the following code:

```
TensorFlowHelper.convertBitmapToByteBuffer(image, intValues, imgData);
```

3. Run inference. The TF Lite interpreter runs the model that is assigned to it. The following line of code runs a neural model without exposing its complexity:

```
tfLite.run(imgData, outputVector);
```

4. Convert the output data of the interpreter to an image, as you can see in the following code:

```
outputImage = Bitmap.createBitmap(mInputImageWidth, mInputImageHeight,
Bitmap.Config.ARGB_8888);
```

5 Arm NN optimization

Arm NN uses Arm Compute Library (ACL) to provide a set of optimized operators, for example convolution and pooling, that target Arm-specific accelerators like the DSP (Neon) or the Mali GPU. ACL also provides a GPU tuner tool called CLTuner. CLTuner tunes a set of hardware knobs to fully utilize all the computational horsepower that the GPU provides.

Because Arm NN implements the Android NNAPI interface, developers only need to install the driver. Your Android application will seamlessly interact with the driver to exploit these accelerations.

This part of the code is illustrated in the `TensorFlowImageStyleTranfer()` function in `TensorFlowImageStyleTransfer.java`. To install the driver, the code performs the following steps:

1. Check the Android OS version.

2. Determine whether NNAPI can be enabled on the device.

3. Create a delegate, if NNAPI can be supported:

```
if (enableNNAPI && TensorFlowHelper.canNNAPIEnabled()){
    delegate = new NnApiDelegate();
    this.tfLiteOptions.addDelegate(delegate);
    this.tfLite = new
Interpreter(TensorFlowHelper.loadModelFile(context, mModelFile), tfLiteOptions);
} else {
    this.tfLite = new
Interpreter(TensorFlowHelper.loadModelFile(context, mModelFile));
}
```

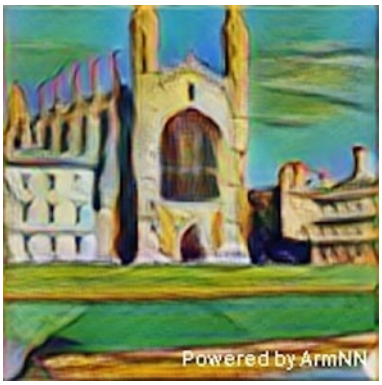
Arm NN implements the Android NNAPI interface. This means that, when developers have the driver installed, your Android application will seamlessly interact with the underlying APIs. This will allow you to exploit the accelerators.

Toggle the NNAPI checkbox to experience the performance enhancement that NNAPI provides.

The NN driver is not bundled with Android releases. Instead, the NN driver is shipped by OEMs like Samsung, HiSilicon, and MTK. For example, all Samsung devices with Android O MR1 or later firmware releases have pre-installed the Arm NN driver.

If your Android device does not have an Arm NN driver pre-installed, or if you want to build your own Arm NN driver, **Next steps** provides information on how to manually install the driver.

Use the Android app to see whether you can create your own art piece. The following generated image of Cambridge is created in La Muse style and built with Arm NN:



Generated image

6 Deploying the Android NN driver

If you don't see a significant performance acceleration when Arm NN is enabled on your Android phone, you need to upgrade your Android NN driver to the latest version.

To use the latest Arm NN Android driver, you need to start the driver service. The LiveStyle app will create a delegate that automatically uses the service.

Before you begin, you will need:

- An Android phone with root access
- A host machine that supports adb
- Adb on host
- A latest **pre-built Arm NN android-nn driver** on host
- The built Livestyle app on your device

Follow these steps:

1. Transfer the driver from the host to a local folder on the phone. You must put the driver in a directory with read/write permissions. Here is an example of transferring the driver to the `data/local/tmp` folder of your phone:

```
user@host: adb push <ArmNN android-nn driver> /data/local/tmp
```

2. Log into the Android shell to start the driver service, as you can see in the following code:

```
user@host: adb shell
* daemon not running; starting now at tcp:5037
* daemon started successfully
```

3. Log in as root by typing `su` and `cd` into the directory containing the latest driver we just pushed, as you can see here:

```
user@android: su
root@android: cd /data/local/tmp
```

4. Start the driver in the background. To make it explicit, pass the `-c` GpuAcc to enable GPU (OpenCL) acceleration.

```
root@android: ./ -c GpuAcc &
```

5. Open another shell on your host machine, and do a `log cat` on the most recent kernel messages, as you can see in the following code. This will verify whether the app is using our driver or not:

```
user@host: adb log cat -T 10 | grep Arm NN
```

6. Run the app on your phone. Choose a style and click TAKE A PICTURE. You should see output containing Arm NN driver information on `log cat`. The output of `log cat` should look something like what you can see in this code:

```
<Time stamp><PID><PID>V ArmnnDriver:
ArmnnPreparedModel::execute(): 1 input(s), 37 operation(s), 1 output(s), 185
operand(s)
…
<Time stamp><PID><PID>V Arm NN Driver:
```

7 Tuning performance with OpenCL tuner

You can use the Compute Library OpenCL tuner to find optimum values for GPU acceleration tuning parameters. This involves running the driver twice:

- Once in tuning mode to find a set of good parameters
- Once in normal mode, with the tuned parameters

For more information, please refer to Using the GPU tuner in **Arm NN android-nn driver**. We assume that the Android NN driver is already transferred onto your phone at `/data/local/tmp`.

Follow these steps:

1. Run the driver in tuning mode.

   Tuning could take several minutes, particularly for deeper networks. This means that the OpenCL tuner is difficult to deploy in real-world applications. To overcome this issue, we have introduced three levels of tuning in OpenCL tuner:

   - EXHAUSTIVE – This level offers peak performance with a high tuning time.
   - RAPID – This level offers the shortest tuning time with reduced performance uplift.
   - NORMAL – This level offers the balance between tuning time and a good approximation of the optimal performance.

   We use NORMAL level in this example.

   The tuned parameter output file must be at a writable location.

2. Write the output to a file called `tuned_params` at `/data/local/tmp`:

   ```
   root@android: cd /data/local/tmp
   ```

3. Create the `tuned_params` file:

   ```
   root@android: touch tuned_params
   ```

   When the service starts in the background, you will see the PID of the service printed in the `stdout`. Make a note of the PID. We will need to kill the service with this PID when the tuning process finishes.

4. Start your app as normal. Because the tuning parameters are being selected, you will notice a delay, between one minute up to five minutes, when processing the frame. After the first frame is processed and displayed in the app, the tuning process is finished.

5. Terminate the driver with the PID that you wrote down in step 3, using this code:

```
root@android: kill <PID>
```

   If you need to find the PID again, run the following command, followed by "`kill <PID>`":

```
root@android: ps -Af | grep <driver>
```

   You can also inspect the tuned parameters with:

```
root@android: cat tuned_params
```

6. Restart the driver with the tuned parameters that were created in the `tuned_params` file, as you can see in the following code:

```
root@android: ./<driver> --cl-tuned-parameters-file tuned_params -c GpuAcc &
```

7. Restart the app as usual. You should see a noticeable performance boost.

8 Related information

Here are some resources related to material in this guide:

- **Arm NN** on the Arm architecture, with an Android device or a Raspberry Pi
- **A Neural Algorithm of Artistic Style**
- Intuitive Guide to Neural Style Transfer
- **The pre-trained models we used as a starting point in this guide**.

9 Next steps

Neural style transfer is a useful Machine Learning technique that you can use to turn your photo into a piece of art. A useful next step is to train your own style transfer model. If you want to train your own model, or a model with a different style, you can follow the training steps detailed in this **paper** or other **open-source projects**. You will need to apply the four changes that we mentioned in **Looking at the Android code**.