

OP SIS ANDROID APPLICATION DOCUMENTATION

Contents:

Google vision API face detection vs OpenCV cascade face classification.....	2
Google vision frame modification.....	3
Modified GV facial border test.....	7
Android studio and Opsi app set up.....	13
Gradle build setup.....	14
OpenCV dependency setup.....	15
Adding native source files and VLfeat.....	16
Configuring CMake file.....	16
Configuring Android Manifest.....	17
Graphical User Interface.....	18
Google Vision Face Tracking Pipeline.....	25
Native Facial Expression Analysis algorithm execution.....	28
Trained Models Initialization.....	30
Debugging.....	31
How to run Opsi code on phone.....	32

Google vision API face detection vs OpenCV cascade face classification

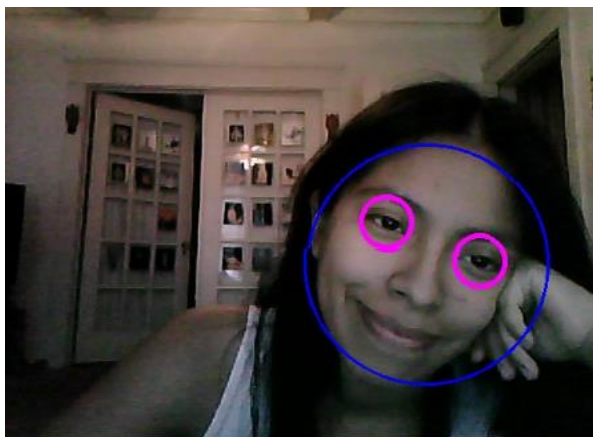
Google vision API is google's mobile vision library which has several image and video processing abilities such as text recognition, barcode scanning and face detection. Google vision(GV) face detection function is able to localize faces in visual media returning their positions, sizes and orientations. Moreover, it detects face landmarks such as eyes and nose. Face tracking extends face detection to video sequences or streams. It decreases costs of computation by comparing two consecutive video frames avoiding face detection process on subsequent frames. GV is a powerful tool in android and IOS mobile development since its algorithms were specifically designed for mobile hardware systems. Moreover, it provides Camera class implemented to continuously stream video frames into detector and its associated pipeline in most efficient speed to fit requested fps and considering current machine's CPU and RAM load.

Picture 1: Google vision API detected faces with smiling probabilities



OpenCV cascade classifier is object detector which is used for object detection on images and video streams. The advantage is that it can be used for any type of physical object on frames and could be easily trained with several hundred positive and negative examples of same size. CV also provides already trained models for face classifications in .xml files. CV's cascade classifier uses several simple classifiers using one of four different boosting techniques at each stage. It uses edge features, line features and center-surround features which are called Haar-like features to detect objects of interest. Object tracking can be implemented programmatically checking subsequent frames for pixel moves. It is a powerful algorithm in face detection and tracking, also process pipeline can be natively written which also gives more speed performance.

Picture 2: OpenCV detected face and eyes

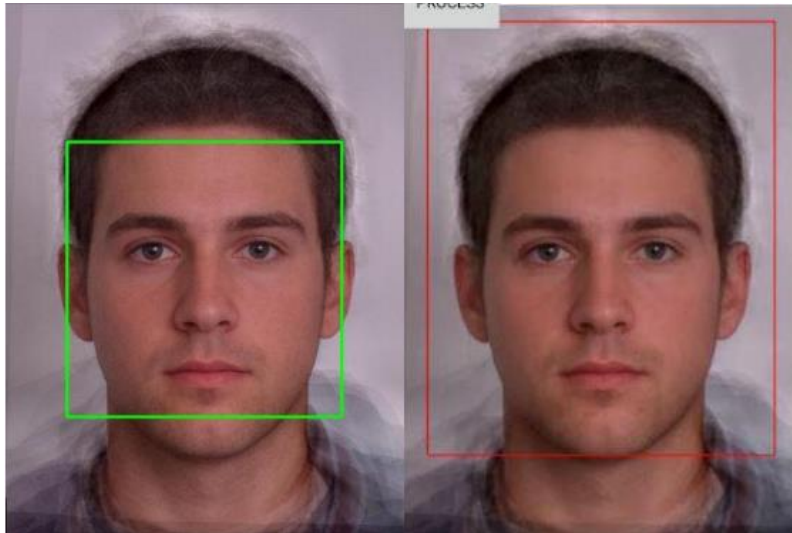


Since, Google vision algorithms are efficient and specifically designed for mobile devices providing useful camera and processor classes, decided that it will be used in Opsi Android application.

Google vision frame modification

Windows version of Opsis application is written on OpenCV and Vlfeat to track and detect face landmarks respectively. Detected face landmarks are processed on trained regressive models to estimate facial expressions and face orientations. Models are trained on face images cropped by OpenCV haar-cascade algorithm which returns square shaped object (face) borders. However, Google Vision API provides rectangular shaped borders of widths and heights differing from CV's haar-cascade classifier. Using such borders could lead to uncertainties increase since models were trained for OpenCV detector cut face images.

Picture 3: OpenCV face border(green) vs Google vision face border(red)



From several visual observations, some patterns and correlations were found between CV and GV facial borders. First observation is that center of CV's border plane is always approximately in the middle of nose, but GV's border plane center is between eyebrows.

Picture 4: OpenCV face border center vs Google Vision face border center



Second observation is that CV face borders are always square shaped and if top border of GV face borders drawn down until square shape is obtained, new center of GV face border plane will approximately be at the center of nose as in CV face boundaries. In other words, remaking GV face rectangle to square by holding bottom boundary fixed, decreasing lengths of left and right boundaries and lowering top boundary position until lengths of left and right boundaries is the same as the lengths of bottom and top boundaries makes new border center to be approximately at the center of nose.

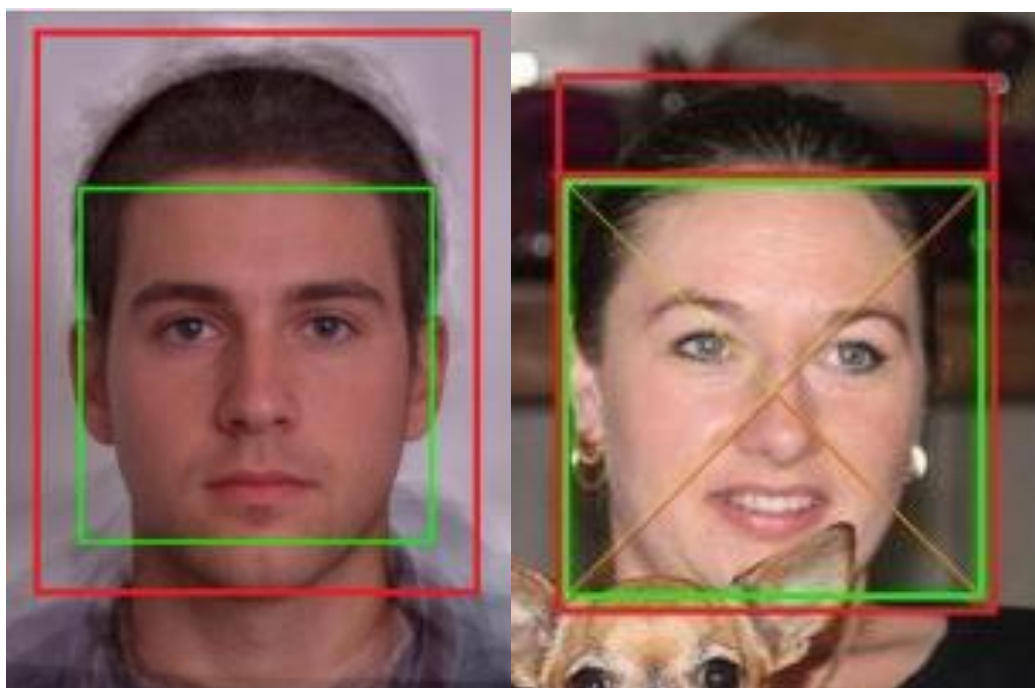
**Picture 5: Illustration of GV borders remaking process to fit the center of CV border plane
OpenCV(green), Google Vision API(red)**



Final step is to rescale GV borders side lengths by some unknown rescale factor to fully fit CV border plane. The third important observation is as follows: the closer a face is to camera or the more portion of image is occupied by face – the more is difference between border side length or greater rescale factor.

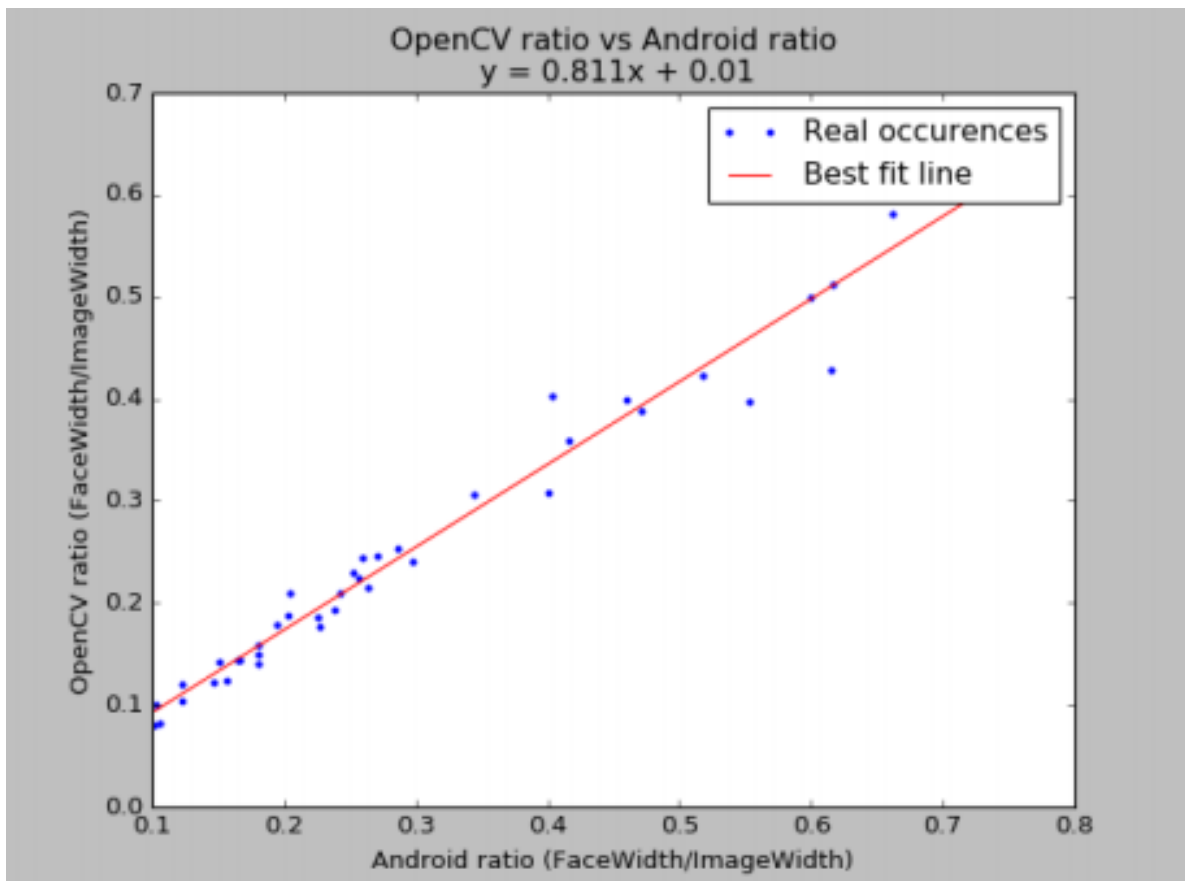
Picture 6: a) Portrait photo (face width is on the most portion of image) b) Photo of whole body (face width is approximately 20% of image)

a) More difference -> more rescale factor b) Less difference -> less rescale factor



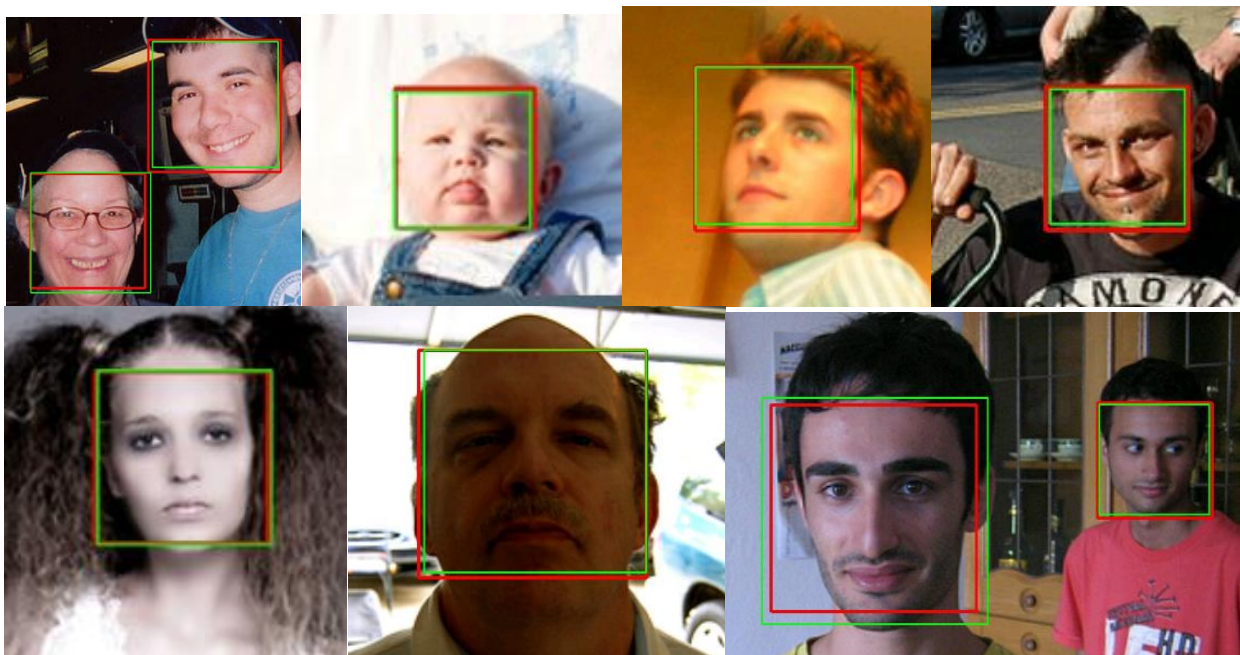
From this observation, it is seen that facial boundary width is not important in determining rescaling factor, but facial boundary width ratio to full image width is important aspect. To check relation between CV boundary width ratio to GV boundary width ratio, several images were analyzed and ratio data were plotted.

Graph 1: CV facial width to image width ratio vs GV facial width to image width scatter plot



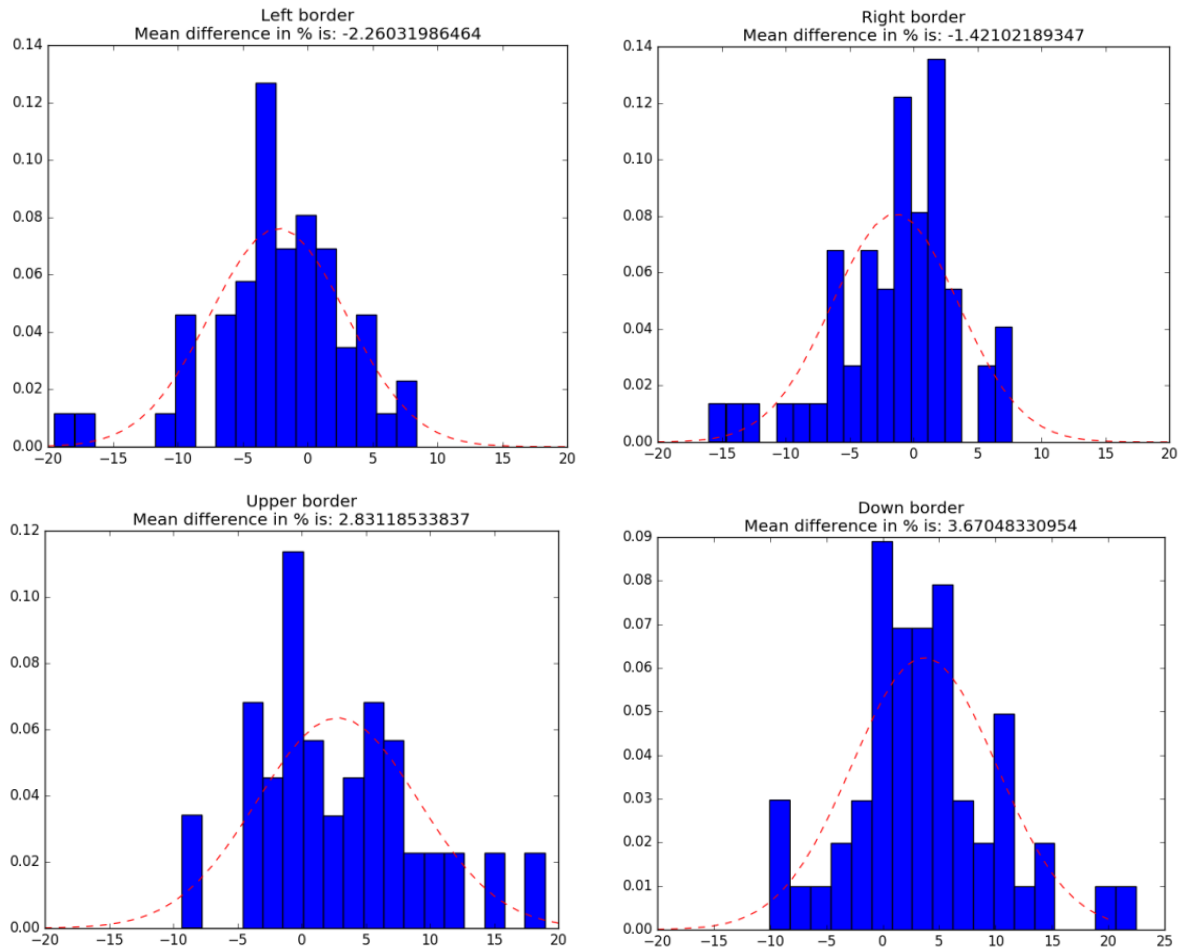
From Graph 1 it is clearly seen that there is linear dependency between CV and GV ratios, and from best fit line equation we can easily find the value of GV required facial width to be closer to CV facial boundary. Below are comparison images which includes both CV and modified GV facial boundaries using algorithm above.

Picture 6: Comparison of OpenCV (green) and modified Google Vision API boundaries (red)



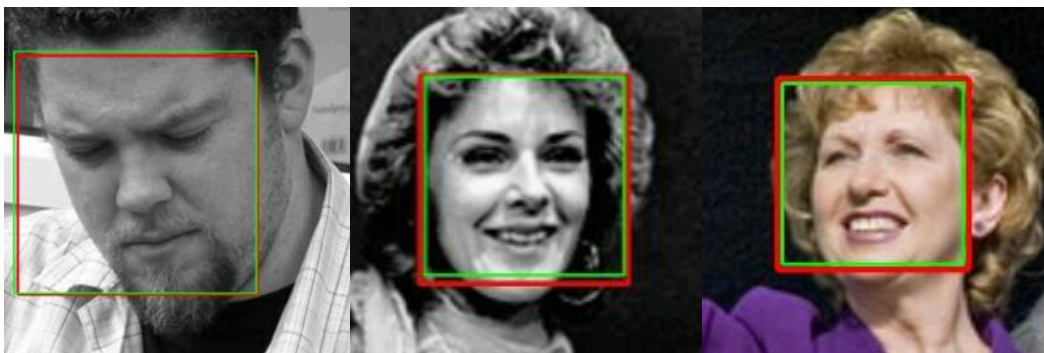
Finally, errors of each side border were calculated simply by subtracting modified GV border position from actual CV border position and dividing this value by GV face width to get errors in percentages. This is because percentages are more consistent since small width faces can have several pixels difference, but effect of this difference to trained model will be huge because of lack of pixels in small faces losing significant parts of face. On the other hand, bigger faces may have more pixel difference, but less effect on trained model, since significant face pieces are not lost and there are enough pixels to analyze.

Graph 4: Border percentage error histograms



From Graph 2 it is seen that mostly GV border modification algorithm have low error. Also, average error is calculated and considered as correction factors to modify GV borders again. So finally, GV boundaries are corrected and used in further calculations. Below are some examples of corrected GV facial boundaries.

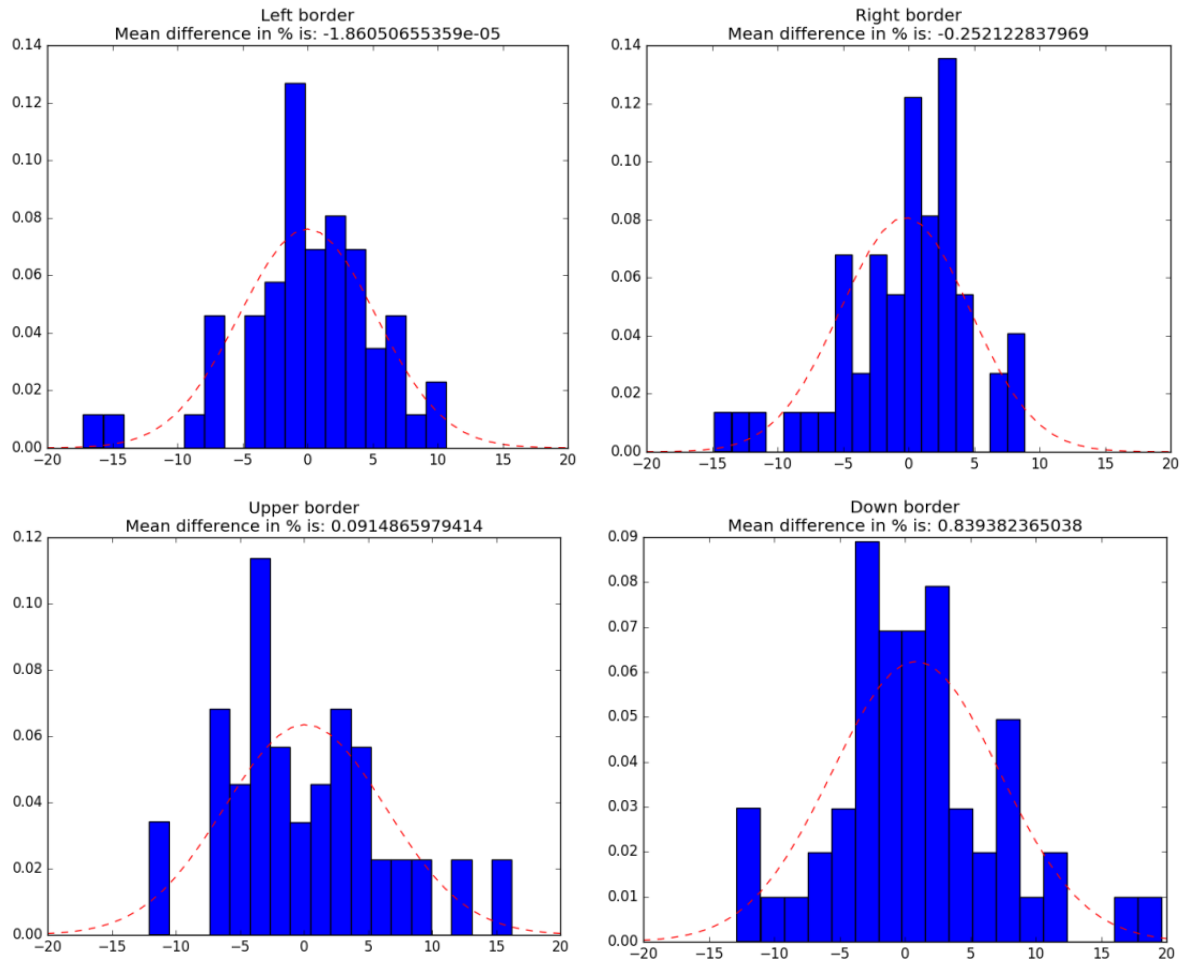
Picture 7: Corrected final version GV boundaries



Modified GV facial border test

After GV boundaries are corrected, border error analysis was done again. As expected errors should disappear for this particular image set, however it is not that simple. Since errors are calculated in percentages in relation to GV boundary width, it will not be actual errors in relation to CV boundary. In ideal errors should be calculated in relation to CV boundary width, to know exactly difference between GV border in relation to CV border. The problem is that when face detection is in process, there is no any information about CV border width and the best choice is to use GV border width considering them to be approximately similar.

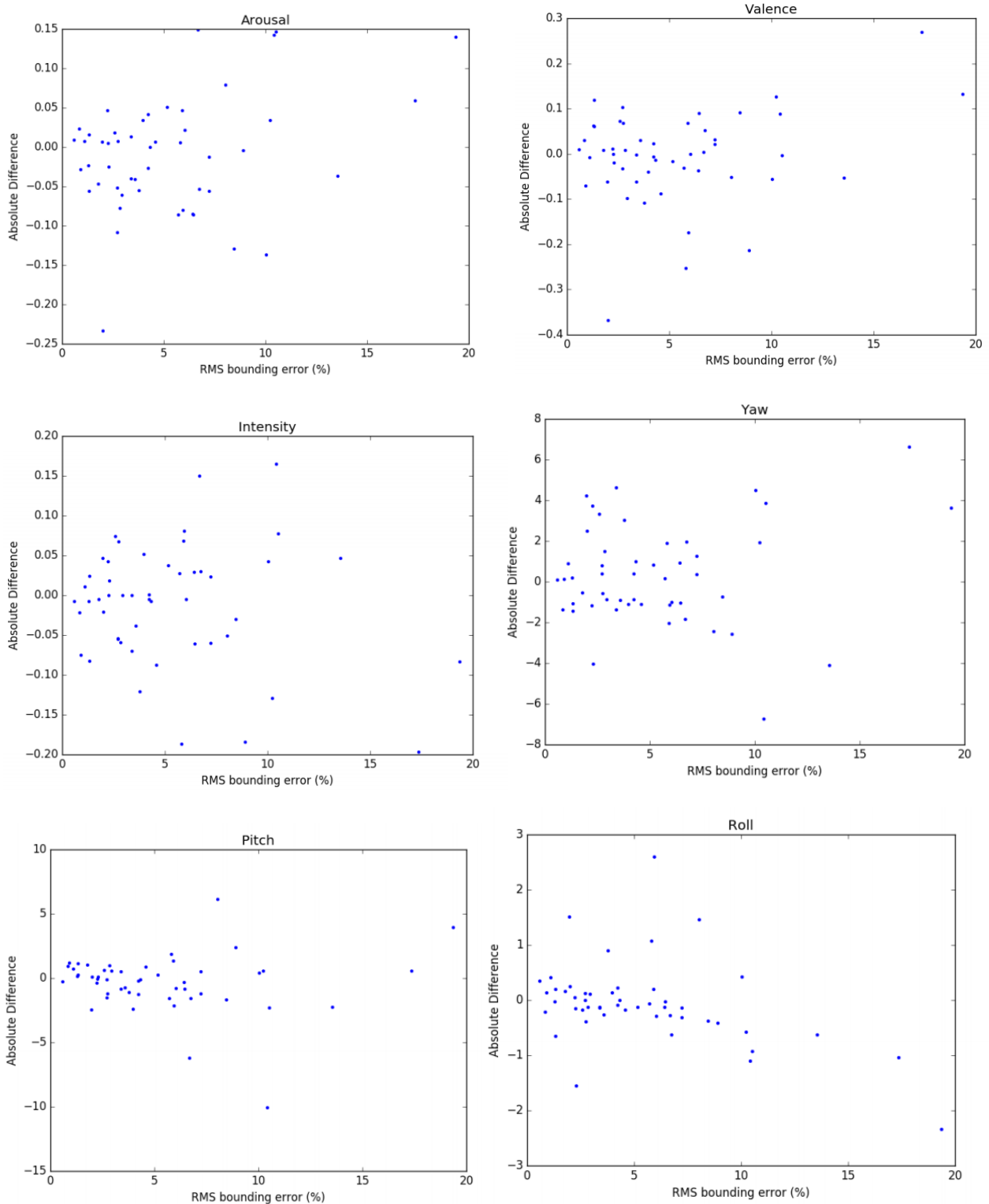
Graph 5: Correct GV border error analysis



From graph 5 improvement can be seen in terms of error, but errors did not totally disappear. Left border error is almost 0, right border is 0.25% which is acceptable, upper border is 0.1% which is also acceptable and the greatest error has down border equaling to 0.84%. This is reasonable because uncorrected down border also had greatest error among others and in overall corrected GV facial borders are considered as acceptable. Nonetheless, border errors are not so crucial in facial expression calculations and next test will show that.

To check correlation between boundary error and expression and orientation errors image set was analyzed for root-mean-square boundary error and absolute difference in expression and orientation values. Absolute differences were plotted versus RMS bounding error in a scatter plot to understand the nature of these error and dependency on RMS bounding error. From graph 6 below it can be clearly seen that expression and orientation resulting values are independent on boundary errors and in average near 0. By inspection it seems like resulting value errors are of random error nature, because in average they have 0 error and randomly scattered. Can be seen below in Graph 6.

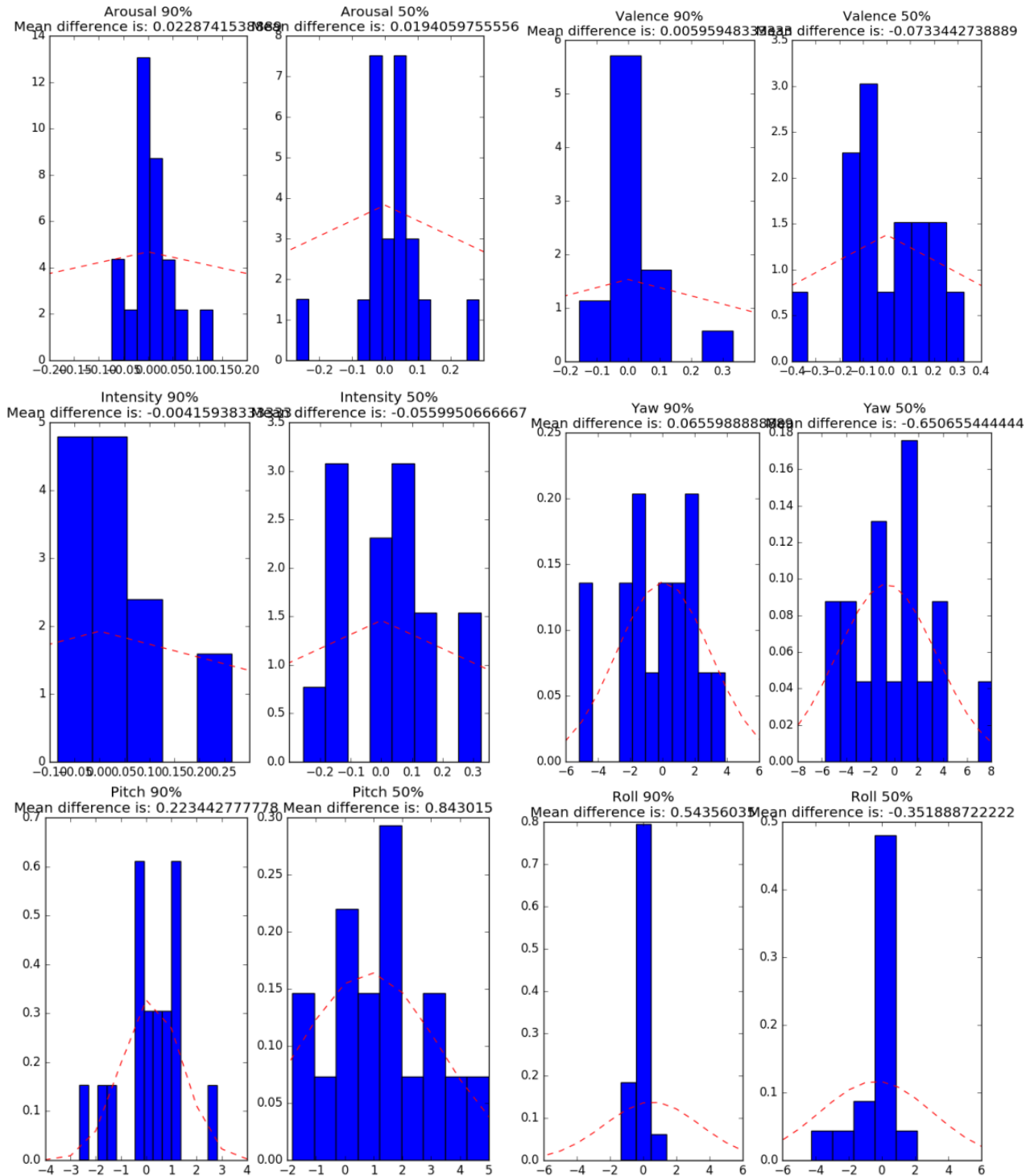
Graph 6: Expression and orientation error analysis versus bounding errors



During facial expression and orientation error analysis detected that even image rescaling have effects on the results. Bitmap to CV::Mat conversion and vice-versa sometimes leads to slight image rescales, also reading .jpg files from Android Drawable into Bitmap creates rescales of factor 2.8 to 3.2. So, this is important aspect to check. Image data set was rescaled 2 times. First by 50% and second time by 90% of the full images size. Finally, there were 3 image sets which are original set, 50% smaller

set and 10% smaller set. Both rescaled sets were tested versus original sized image set in terms of facial expression and orientation errors.

Graph 6: Expression and orientation rescaling errors

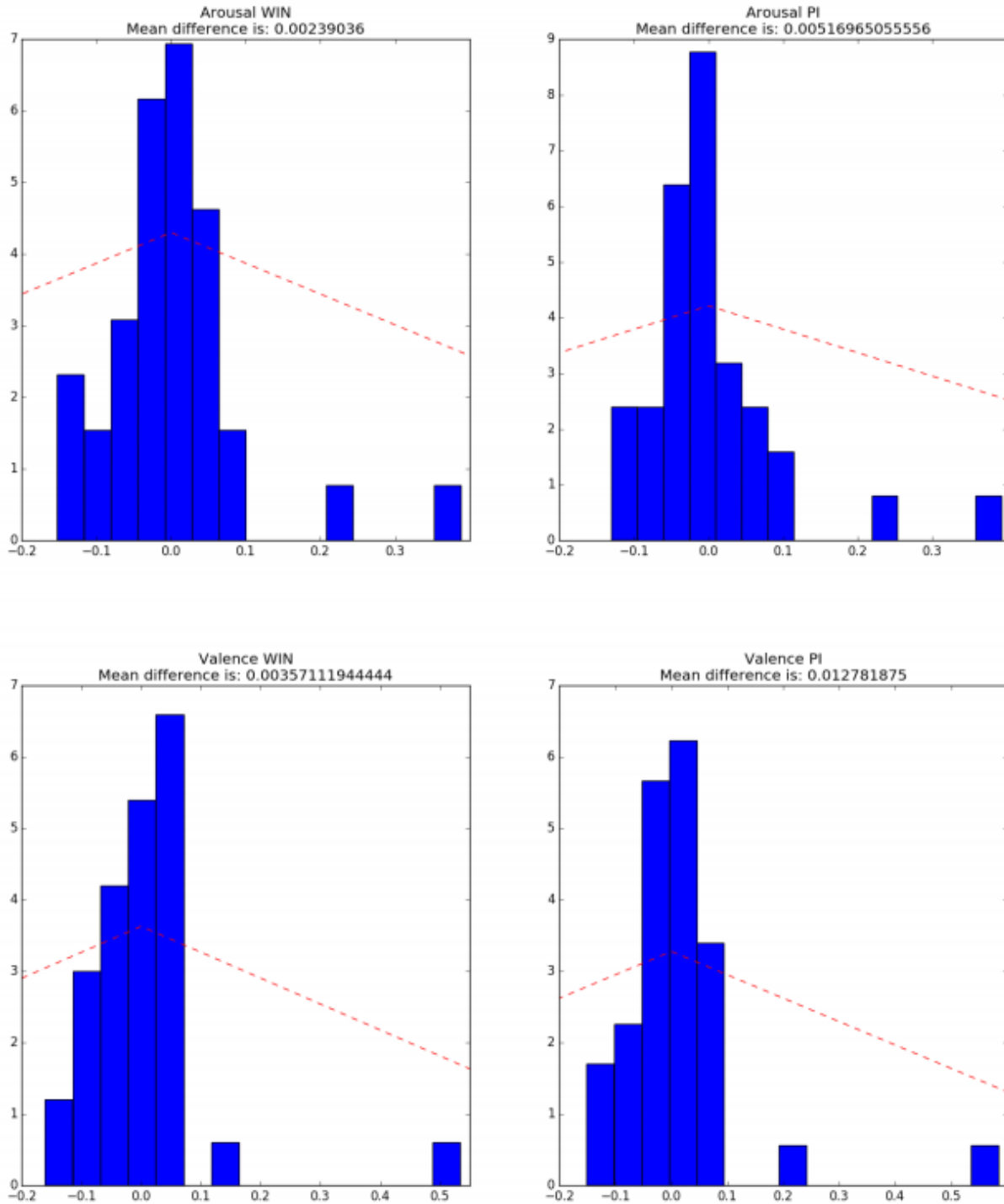


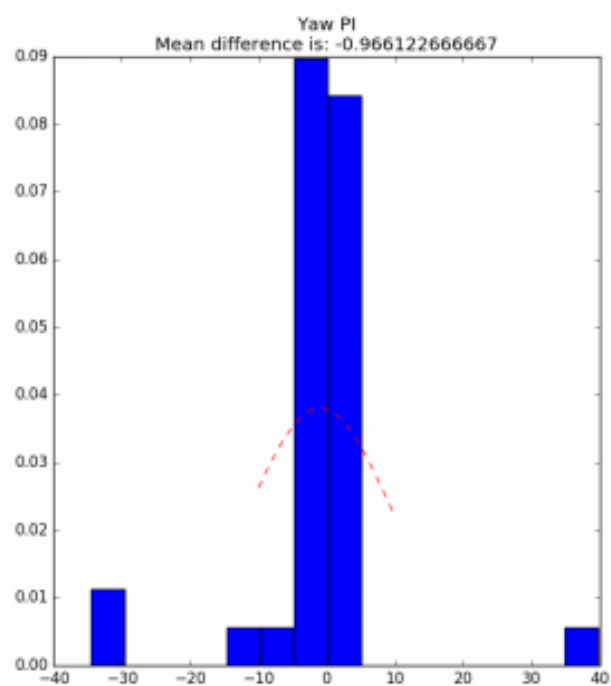
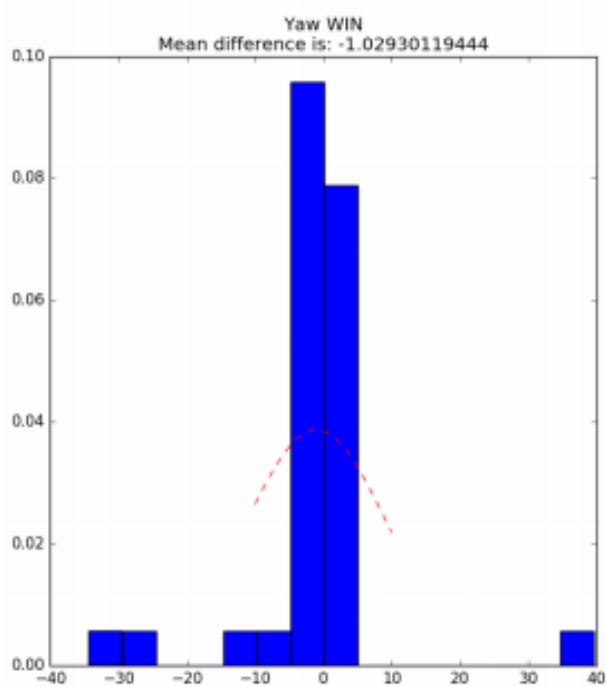
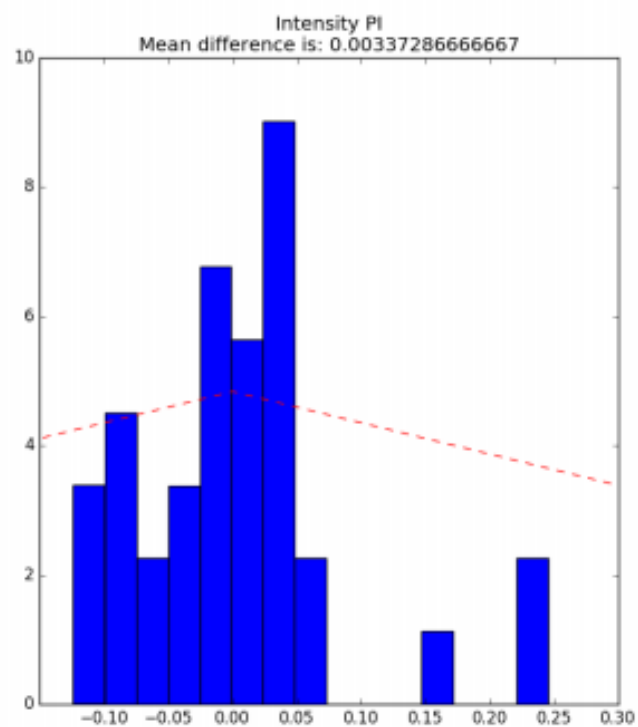
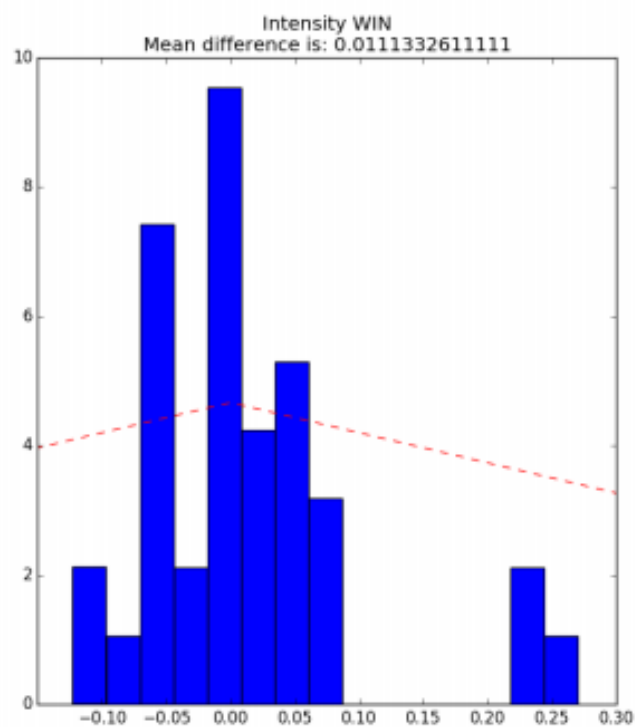
From graph 6, it can be seen that rescaling has effects on expression and orientation results, but rescaling factor is not the key, because sometimes 90% rescaled image set have greater error but sometimes 50% image set. For example 50% rescaled image set have less error in comparison to 90% rescaled image set in terms of Arousal, Yaw and Roll, but 90% rescaled image set have less error in comparison to 50% rescaled image set in terms of Valence, Intensity, and Pitch. So in conclusion trained model is very sensitive to any rescaling changes, but has no correlation to rescaling factor.

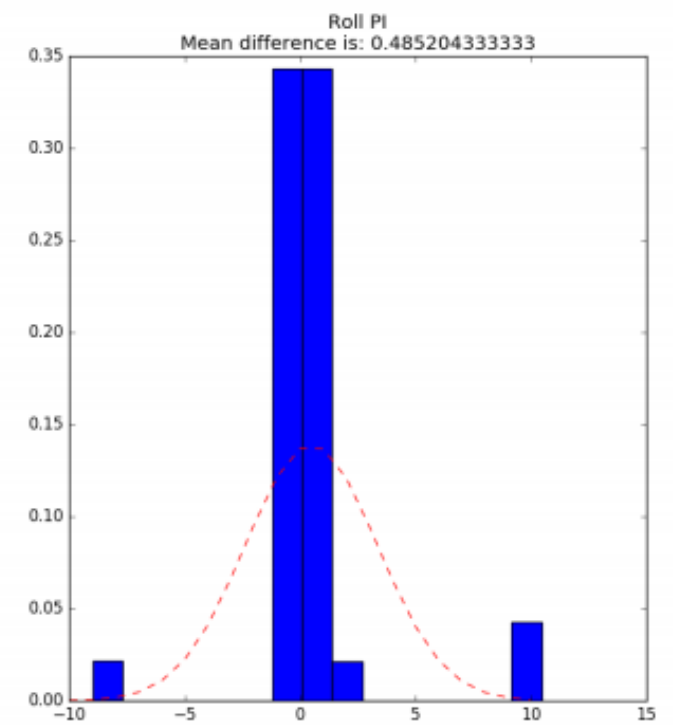
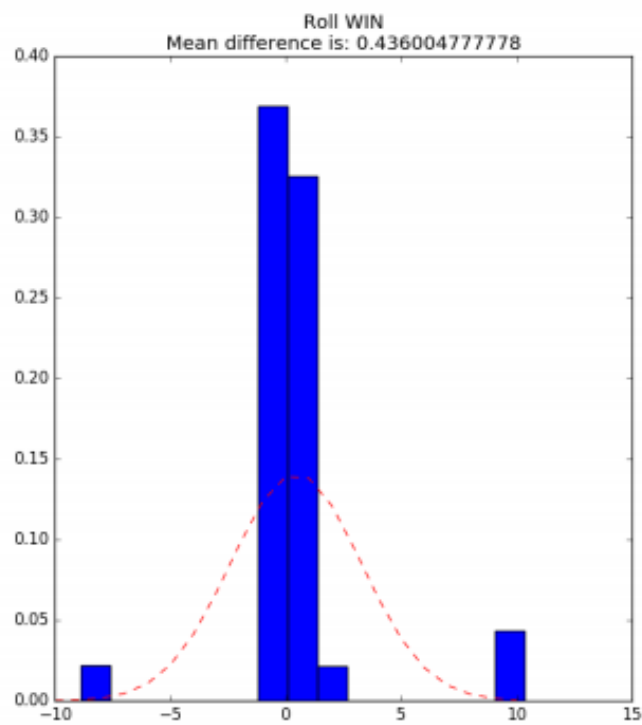
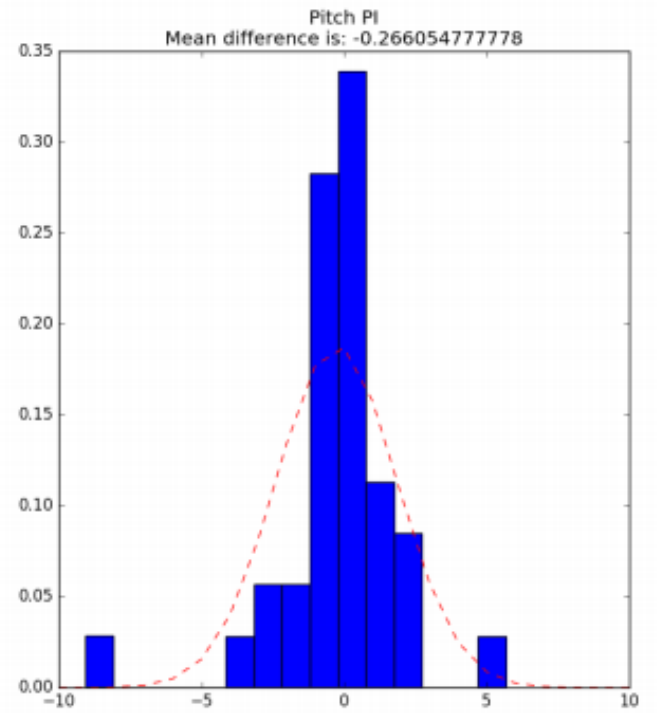
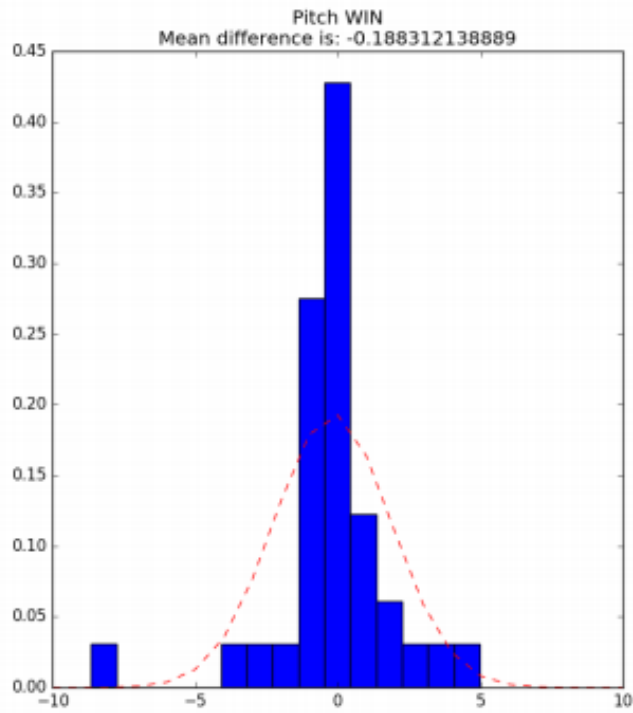
Nature of these errors seems to be random since they are normally distributed and has no dependency on rescaling factor.

Since rescaling is not important aspect anymore, modified Google Vision facial boundary error tests were done in comparison to Windows and Raspberry PI versions in terms of expression and orientation. The results were acceptable.

Graph 7: GV expression and orientation absolute error histograms in relation to Windows and Raspberry PI



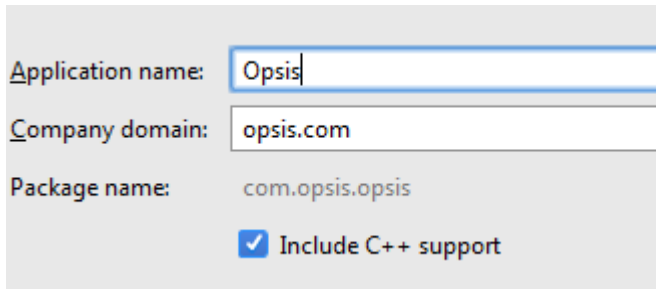




Finally, all tests showed that modified GV boundary is acceptable and very similar to OpenCV boundaries. Also, expression and orientation errors have low dependency on facial border position errors. Moreover, expression and orientation errors have no dependency on rescaling of frames so building Android app can be started now.

Android studio and Opsis app set up

Download latest Android Studio and install it. Create new project and tick “Include C++ support”.



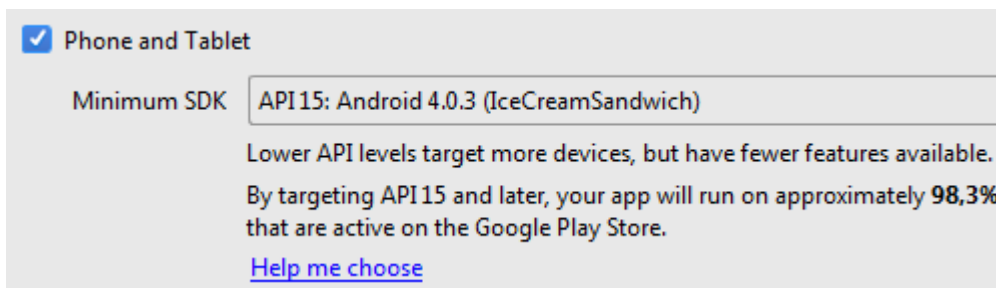
Application name: Opsis

Company domain: opsis.com

Package name: com.opsis.opsis

☒ Include C++ support

Choose Android 4.0.3 API 15, higher are welcome but not tested.



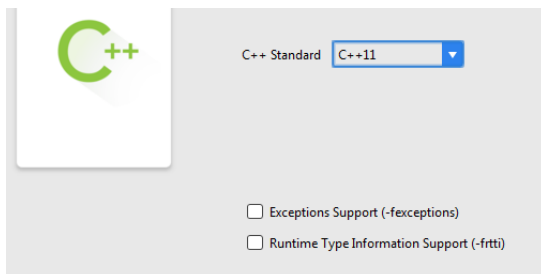
☒ Phone and Tablet

Minimum SDK: API 15: Android 4.0.3 (IceCreamSandwich)

Lower API levels target more devices, but have fewer features available.
By targeting API 15 and later, your app will run on approximately 98,3% that are active on the Google Play Store.

[Help me choose](#)

In Customize C++ Support window choose C++11, also you can tick Exception Support and Runtime Type Information Support.

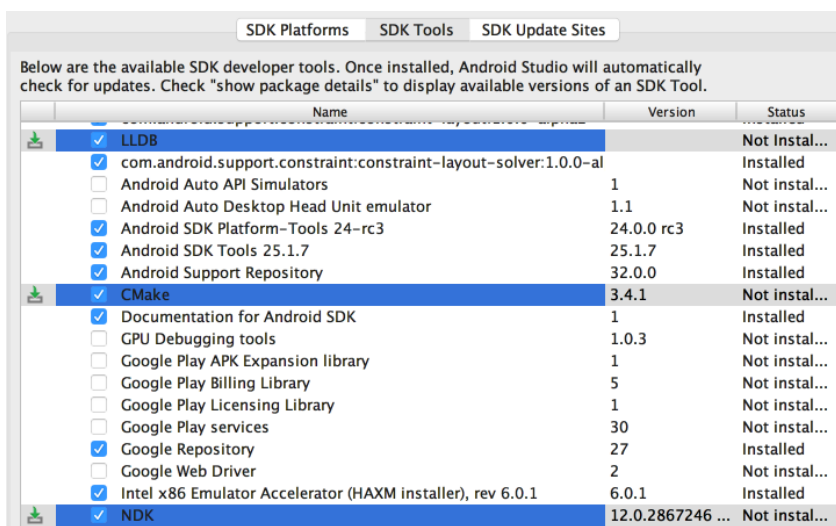


C++ Standard: C++11

☐ Exceptions Support (-fexceptions)

☐ Runtime Type Information Support (-ftrti)

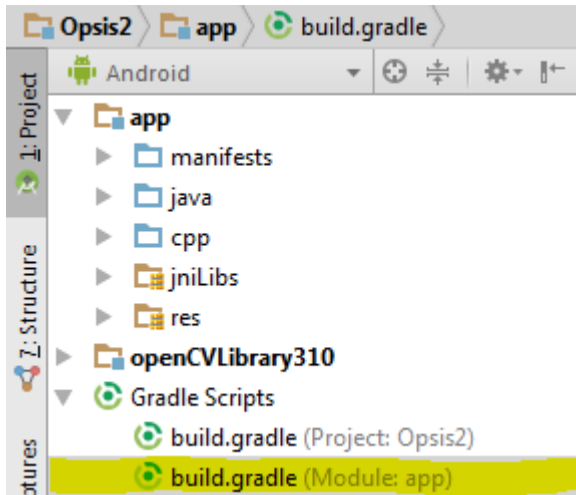
Select Tools -> Android -> SDK Manager, then click on SDK Tools tab and check **LLDB**, **CMake** and **NDK** and then **Apply** and **OK** in next dialog. After installation click **Finish** and then **OK**.



Name	Version	Status
<input checked="" type="checkbox"/> LLDB		Not instal...
<input checked="" type="checkbox"/> com.android.support.constraint:constraint-layout-solver:1.0.0-alpha1		Installed
<input type="checkbox"/> Android Auto API Simulators	1	Not instal...
<input type="checkbox"/> Android Auto Desktop Head Unit emulator	1.1	Not instal...
<input checked="" type="checkbox"/> Android SDK Platform-Tools 24-rc3	24.0.0 rc3	Installed
<input checked="" type="checkbox"/> Android SDK Tools 25.1.7	25.1.7	Installed
<input checked="" type="checkbox"/> Android Support Repository	32.0.0	Installed
<input checked="" type="checkbox"/> CMake	3.4.1	Not instal...
<input checked="" type="checkbox"/> Documentation for Android SDK	1	Installed
<input type="checkbox"/> GPU Debugging tools	1.0.3	Not instal...
<input type="checkbox"/> Google Play APK Expansion library	1	Not instal...
<input type="checkbox"/> Google Play Billing Library	5	Not instal...
<input type="checkbox"/> Google Play Licensing Library	1	Not instal...
<input type="checkbox"/> Google Play services	30	Not instal...
<input checked="" type="checkbox"/> Google Repository	27	Installed
<input type="checkbox"/> Google Web Driver	2	Not instal...
<input checked="" type="checkbox"/> Intel x86 Emulator Accelerator (HAXM installer), rev 6.0.1	6.0.1	Installed
<input checked="" type="checkbox"/> NDK	12.0.2867246 ...	Not instal...

Gradle build setup

In left Android view panel open build.gradle (Module: app) to set up google vision dependencies, Cmake file parameters, links, flags, Application binary interface instructions, android SDK settings and to set up path of your native code or JNI libraries.



In android settings:

- set compiler SDK version to 24
- build tools version to 25.0.0
- set minimum SDK version to 9
- set target SDK version to 24
- in cmake instructions set ABI filters to all android architectures: 'x86', 'x86_64', 'armeabi', 'armeabi-v7a', 'arm64-v8a', 'mips', 'mips64'
- set a path to folder with compiler native libraries in a sourceSets function tag using `jniLibs.srcDirs = ['src/main/jniLibs']`
- set the path to your cmake file

Full android function tag is given below together with google vision dependencies.

!gradle.app-----!

apply **plugin: 'com.android.application'**

```
android {
    compileSdkVersion 24
    buildToolsVersion "25.0.0"
    defaultConfig {
        applicationId "your_app_id"
        minSdkVersion 9
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        externalNativeBuild {
            cmake {
                cppFlags "-std=c++11"
                abiFilters 'x86', 'x86_64', 'armeabi', 'armeabi-v7a', 'arm64-v8a', 'mips', 'mips64'
            }
        }
    }
}

buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
```

```

}

sourceSets {
    main {
        jniLibs.srcDirs = ['src/main/jniLibs']
    }
}

externalNativeBuild {
    cmake {
        path "CMakeLists.txt"
    }
}

dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    compile 'com.android.support:support-v4:24.2.0'
    compile 'com.android.support:design:24.2.0'
    compile 'com.google.android.gms:play-services-vision:9.4.0+'
}
!-----!

```

Sync your project.

OpenCV dependency setup

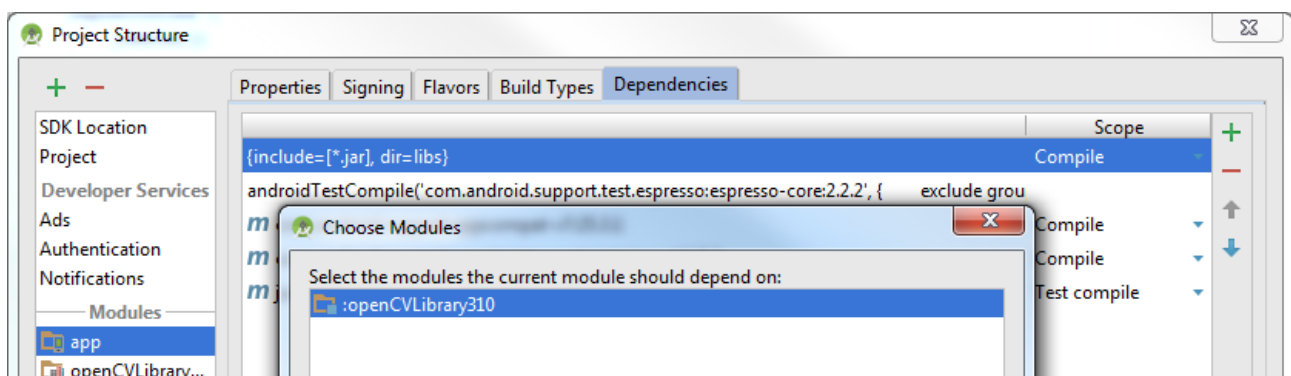
- download latest OpenCV SDK for Android from OpenCV.org
- decompress the zip file in your C:/ folder
- Import OpenCV library module
 1. New -> Import Module
 2. Choose C:/ OpenCV-android-sdk/sdk/java folder
 3. Uncheck replace jar, uncheck replace lib, uncheck create gradle-style
- Set the OpenCV library module up to fit your SDK by editing OpenCV build.gradle's following fields:

```

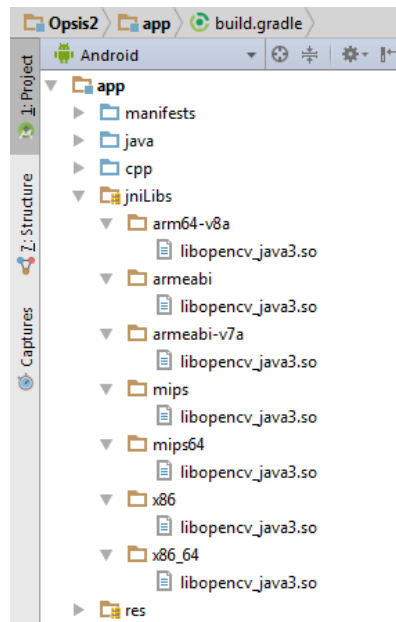
compileSdkVersion 24
buildToolsVersion "25.0.0"
defaultConfig {
    minSdkVersion 9
    targetSdkVersion 24
}

```

- Add OpenCV module dependency in your app module: File -> Project Structure -> Module app -> Dependencies tab -> New module dependency -> choose OpenCV library module

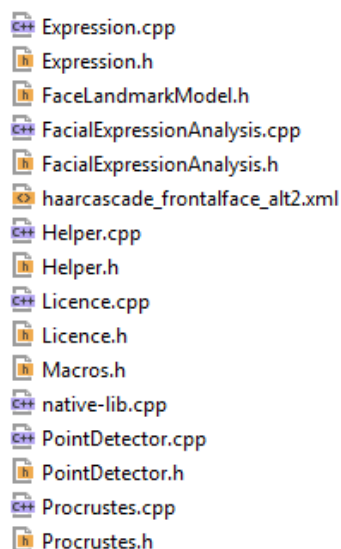


- Create a folder named jniLibs in app/src/main and copy all folders from [YOUR_OPENCV_SDK/sdk/native/libs](#) and paste in your new jniLibs folder. Then delete all .a files from all architecture folders and leave only libopencv_java3.so files. Your jniLibs folder should look like below:



Adding native source files and VLfeat

- In a project pane go to app/src/main/cpp and paste Opsi Facial Expression Analysis source files which are on image below



- In cpp folder create new folder named vl and paste all vl .c source files there.

Configuring CMake file

In Android pad open External Build Files at the bottom and open CMakeLists.txt file.

- Set the minimum cmake version 3.4.1 with command: `cmake_minimum_required(VERSION 3.4.1)`
- Add disabling flags for VLfeat to disable Windows Intel AVX extension and SSE SIMD instructions: `add_definitions(-DVL_DISABLE_SSE2 -DVL_DISABLE_AVX)`

- Include OpenCV and VLFeat header files folders: `include_directories(C:/OpenCV-android-sdk/sdk/native/jni/include C:/vlfeat-0.9.20)`
- Add OpenCV library to your native project and set target properties compiled .so files:
`add_library(lib_opencv SHARED IMPORTED)`
`set_target_properties(lib_opencv PROPERTIES IMPORTED_LOCATION`
`${CMAKE_CURRENT_SOURCE_DIR}/src/main/jniLibs/${ANDROID_ABI}/libopencv_java3.so)`
- Add all .c and .cpp source files to compile them under shared library name “native-lib”:
`file(GLOB SOURCES src/main/cpp/vl/*.c src/main/cpp/*.cpp)`
`add_library(native-lib SHARED ${SOURCES})`
- Add android JNI Log extension to your native project:
`find_library(log-lib log)`
`set(LIBRARY_DEPS GLESv2 android log EGL)`
- Link OpenCV, log and native-lib shared libraries:
`target_link_libraries(native-lib`
`log`
`lib_opencv`
`${log-lib})`

Below are a full CMake instructions:

!CMake-----!

```
cmake_minimum_required(VERSION 3.4.1)
add_definitions(-DVL_DISABLE_SSE2 -DVL_DISABLE_AVX)

include_directories(YOUR_PATH_TO_ANDROID_OPENCV_SDK/sdk/native/jni/include
YOUR_PATH_TO_VLFEAT_SOURCEFILES)
add_library( lib_opencv SHARED IMPORTED )
set_target_properties(lib_opencv PROPERTIES IMPORTED_LOCATION
${CMAKE_CURRENT_SOURCE_DIR}/src/main/jniLibs/${ANDROID_ABI}/libopencv_java3.so)

file(GLOB SOURCES src/main/cpp/vl/*.c src/main/cpp/*.cpp)
add_library(native-lib SHARED ${SOURCES})
find_library( log-lib log )
set(LIBRARY_DEPS GLESv2 android log EGL)
target_link_libraries( # Specifies the target library.
native-lib
log
lib_opencv
# Links the target library to the log library
# included in the NDK.
${log-lib} )
```

!-----!

Configuring Android Manifest

- In Android Manifest file include permission to use camera and internet:

```
<uses-feature android:name="android.hardware.camera" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.INTERNET" />
```

- Fit your SDK versions:

```
<uses-sdk
    android:minSdkVersion="9"
    android:targetSdkVersion="24" />
```


- Add google vision version and dependencies meta variables:

```
<meta-data android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
<meta-data
    android:name="com.google.android.gms.vision.DEPENDENCIES"
    android:value="face" />
```

- Sync your project.

Graphical User Interface

Open Project pad then go to **app/src/main/java** folder and create new folder called ui.camera. Create new Java class named CameraSourcePreview. This is a class to control Google Vision camera source which extends Android ViewGroup to override its methods.

- Import several modules to be used:

```
import android.content.Context;
import android.content.res.Configuration;
import android.util.AttributeSet;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.ViewGroup;
import com.google.android.gms.common.images.Size;
import com.google.android.gms.vision.CameraSource;
import java.io.IOException;
```

- Declare variables which will be used. Note that some overlays and views are not implemented yet, they will be created later:

```
private static final String TAG = "CameraSourcePreview";
private Context mContext;
private SurfaceView mSurfaceView;
private boolean mStartRequested;
private boolean mSurfaceAvailable;
private CameraSource mCameraSource;
private GraphicOverlay mOverlay;
```

- Create SurfaceCallback adder constructor:

```
public CameraSourcePreview(Context context, AttributeSet attrs) {
    super(context, attrs);
    mContext = context;
    mStartRequested = false;
    mSurfaceAvailable = false;

    mSurfaceView = new SurfaceView(context);
    mSurfaceView.getHolder().addCallback(new SurfaceCallback());
    addView(mSurfaceView);
}
```

- Implement SurfaceCallback function which implements SurfaceHolder.Callback:

```
private class SurfaceCallback implements SurfaceHolder.Callback {
    @Override
```

```

    public void surfaceCreated(SurfaceHolder surface) {
        mSurfaceAvailable = true;
        try {
            startIfReady();
        } catch (IOException e) {
            Log.e(TAG, "Could not start camera source.", e);
        }
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder surface) {
        mSurfaceAvailable = false;
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int
width, int height) {
    }
}

```

- Implement google vision camera source start(), release(), stop() and isPortraitMode() function to check if phone is in portrait orientation mode.

```

    public void start(CameraSource cameraSource) throws IOException {
        if (cameraSource == null) {
            stop();
        }

        mCameraSource = cameraSource;

        if (mCameraSource != null) {
            mStartRequested = true;
            startIfReady();
        }
    }

    public void start(CameraSource cameraSource, GraphicOverlay overlay)
throws IOException {
        mOverlay = overlay;
        start(cameraSource);
    }

    public void stop() {
        if (mCameraSource != null) {
            mCameraSource.stop();
        }
    }

    public void release() {
        if (mCameraSource != null) {
            mCameraSource.release();
            mCameraSource = null;
        }
    }

    private void startIfReady() throws IOException {
        if (mStartRequested && mSurfaceAvailable) {
            mCameraSource.start(mSurfaceView.getHolder());
            if (mOverlay != null) {
                Size size = mCameraSource.getPreviewSize();
                int min = Math.min(size.getWidth(), size.getHeight());
                int max = Math.max(size.getWidth(), size.getHeight());
            }
        }
    }

```

```

        if (isPortraitMode()) {
            mOverlay.setCameraInfo(min, max,
mCameraSource.getCameraFacing());
        } else {
            mOverlay.setCameraInfo(max, min,
mCameraSource.getCameraFacing());
        }
        mOverlay.clear();
    }
    mStartRequested = false;
}
}

```

As you can see start() function need graphical overlay to show phone's camera frames on it. Overlays will be created later.

- Create onLayout() overriding method which will fit your camera source on your overlay, also it swaps height and width if phone is in portrait mode:

```

@Override
protected void onLayout(boolean changed, int left, int top, int right, int
bottom) {
    int width = 320;
    int height = 240;
    if (mCameraSource != null) {
        Size size = mCameraSource.getPreviewSize();
        if (size != null) {
            width = size.getWidth();
            height = size.getHeight();
        }
    }

    if (isPortraitMode()) {
        int tmp = width;
        width = height;
        height = tmp;
    }

    final int layoutWidth = right - left;
    final int layoutHeight = bottom - top;

    int childWidth = layoutWidth;
    int childHeight = (int)((float) layoutWidth / (float) width) *
height);

    if (childHeight > layoutHeight) {
        childHeight = layoutHeight;
        childWidth = (int)((float) layoutHeight / (float) height) *
width);
    }

    for (int i = 0; i < getChildCount(); ++i) {
        getChildAt(i).layout(0, 0, childWidth, childHeight);
    }

    try {
        startIfReady();
    } catch (IOException e) {
        Log.e(TAG, "Could not start camera source.", e);
    }
}

```

In the same ui.camera folder create graphical overlay named GraphicOverlay which is used in CameraSourcePreview to create face graphic layer such as face frames or face landmark points. It extends Android View module since it should cover CameraSourcePreview object which is of Android ViewGroup type.

- Import necessary modules:

```
import android.content.Context;
import android.graphics.Canvas;
import android.util.AttributeSet;
import android.view.View;
import com.google.android.gms.vision.CameraSource;
import java.util.HashSet;
import java.util.Set;
```

- Declare fields:

```
private final Object mLock = new Object();
private int mPreviewWidth;
private float mWidthScaleFactor = 1.0f;
private int mPreviewHeight;
private float mHeightScaleFactor = 1.0f;
private int mFacing = CameraSource.CAMERA_FACING_FRONT;
private Set<Graphic> mGraphics = new HashSet<>();
```

- Create abstract class Graphic to scale and transform graphical overlay on your camera source, also it check if facing camera is used. If so it mirrors frames vertically:

```
public static abstract class Graphic {
    private GraphicOverlay mOverlay;

    public Graphic(GraphicOverlay overlay) {
        mOverlay = overlay;
    }
    public abstract void draw(Canvas canvas);
    public float scaleX(float horizontal) {
        return horizontal * mOverlay.mWidthScaleFactor;
    }
    public float scaleY(float vertical) {
        return vertical * mOverlay.mHeightScaleFactor;
    }
    public float translateX(float x) {
        if (mOverlay.mFacing == CameraSource.CAMERA_FACING_FRONT) {
            return mOverlay.getWidth() - scaleX(x);
        } else {
            return scaleX(x);
        }
    }
    public float translateY(float y) {
        return scaleY(y);
    }

    public void postInvalidate() {
        mOverlay.postInvalidate();
    }
}
```

- Link child class to its parent Android View using super constructor:

```

public GraphicOverlay(Context context, AttributeSet attrs) {
    super(context, attrs);
}

```

- Create graphic adder, remover, clearer, camera info setter and override View onDraw method to make it scaled to phone's camera frames:

```

public void clear() {
    synchronized (mLock) {
        mGraphics.clear();
    }
    postInvalidate();
}

public void add(Graphic graphic) {
    synchronized (mLock) {
        mGraphics.add(graphic);
    }
    postInvalidate();
}

public void remove(Graphic graphic) {
    synchronized (mLock) {
        mGraphics.remove(graphic);
    }
    postInvalidate();
}

public void setCameraInfo(int previewWidth, int previewHeight, int facing)
{
    synchronized (mLock) {
        mPreviewWidth = previewWidth;
        mPreviewHeight = previewHeight;
        mFacing = facing;
    }
    postInvalidate();
}

@Override
protected void onDraw(Canvas canvas) {

    super.onDraw(canvas);
    synchronized (mLock) {
        if ((mPreviewWidth != 0) && (mPreviewHeight != 0)) {
            mWidthScaleFactor = (float) canvas.getWidth() / (float)
mPreviewWidth;
            mHeightScaleFactor = (float) canvas.getHeight() / (float)
mPreviewHeight;
        }

        for (Graphic graphic : mGraphics) {
            graphic.draw(canvas);
        }
    }
}

```

!-----!

Facial expressions are plotted on circle Arousal-Valence diagram which is just an ImageView, so to plot dots on ImageView we also need graphical overlay on it. Create class CircleOverlay class in **ui.camera** folder. Basically it is implemented similar to GraphicOverlay class, but the main difference is that it takes transformation information (ImageView height and width) to scale onDraw() method. Also, it overrides onMeasure() method to equalize sides of overlay, i.e. change its height to be equaling to width. Full implementation is written below:

!CircleOverlay.java-----!

```
import android.content.Context;
import android.graphics.Canvas;
import android.util.AttributeSet;
import android.view.View;

import java.util.HashSet;
import java.util.Set;

public class CircleOverlay extends View {

    private final Object mLock = new Object();
    private int mPreviewWidth;
    private float mWidthScaleFactor = 1.0f;
    private int mPreviewHeight;
    private float mHeightScaleFactor = 1.0f;
    private Set<Graphic> mGraphics = new HashSet<>();

    public static abstract class Graphic {
        private CircleOverlay mOverlay;

        public Graphic(CircleOverlay overlay) {
            mOverlay = overlay;
        }

        public abstract void draw(Canvas canvas);

        public float scaleX(float horizontal) {
            return horizontal * mOverlay.mWidthScaleFactor;
        }

        public float scaleY(float vertical) {
            return vertical * mOverlay.mHeightScaleFactor;
        }

        public float translateX(float x) {
            return scaleX(x);
        }

        public float translateY(float y) {
            return scaleY(y);
        }

        public void postInvalidate() {
            mOverlay.postInvalidate();
        }
    }

    public CircleOverlay(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

```

    public void clear() {
        synchronized (mLock) {
            mGraphics.clear();
        }
        postInvalidate();
    }

    public void add(Graphic graphic) {
        synchronized (mLock) {
            mGraphics.add(graphic);
        }
        postInvalidate();
    }

    public void remove(Graphic graphic) {
        synchronized (mLock) {
            mGraphics.remove(graphic);
        }
        postInvalidate();
    }

    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec);

        int width = getMeasuredWidth();
        setMeasuredDimension(width, width);
        invalidate();
    }

    public void setTransformationInfo(int previewWidth, int previewHeight) {
        synchronized (mLock) {
            mPreviewWidth = previewWidth;
            mPreviewHeight = previewHeight;
        }
        postInvalidate();
    }

    @Override
    protected void onDraw(Canvas canvas) {

        super.onDraw(canvas);
        for (Graphic graphic : mGraphics) {
            graphic.draw(canvas);
        }
    }
}
!CircleOverlay.java-----//!

```

In order to fit circular graph image to different sizes of Android phone screens and be exactly square shaped there were created several custom views. All of them override onMeasure() method of a parent class View. They are MyRelativeLayout, MyRelativeLayoutHorizontal, SquareImageView and SquareImageViewHorizontal for vertical and horizontal modes respectively.

Google Vision Face Tracking Pipeline

Face detector should be specified in the FaceTrackerActivity in the onCreate method:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // other stuff...

    FaceDetector detector = new FaceDetector.Builder()
        .build(getApplicationContext());
}
```

In Opsis project it is specified in createCameraSource() method in CameraSourcePreview class. To get current frames of camera source google vision detector class is extended and detect() method is overridden, to get frame of type Frame in Bitmap type it is converted using bytes of yuvImage to jpeg and converted from jpeg to Bitmap. All frames should be rotated to vertical images in order to make algorithms working properly, so rotations of frames are also done based on Metadata of the frames.

```
class MyFaceDetector extends Detector<Face> {
    private Detector<Face> mDelegate;

    MyFaceDetector(Detector<Face> delegate) {
        mDelegate = delegate;
    }

    public SparseArray<Face> detect(Frame frame) {
        YuvImage yuvImage = new YuvImage(frame.getGrayscaleImageData().array(),
            ImageFormat.NV21, frame.getMetadata().getWidth(),
            frame.getMetadata().getHeight(), null);
        ByteArrayOutputStream byteArrayOutputStream = new
        ByteArrayOutputStream();
        yuvImage.compressToJpeg(new Rect(0, 0, frame.getMetadata().getWidth(),
            frame.getMetadata().getHeight()), 100, byteArrayOutputStream);
        byte[] jpegArray = byteArrayOutputStream.toByteArray();
        frameBitmap = BitmapFactory.decodeByteArray(jpegArray, 0,
            jpegArray.length);
        if (frame.getMetadata().getRotation() == 3) {
            frameBitmap = RotateBitmap(frameBitmap, -90);
        } else if (frame.getMetadata().getRotation() == 2) {
            frameBitmap = RotateBitmap(frameBitmap, 180);
        } else if (frame.getMetadata().getRotation() == 1) {
            frameBitmap = RotateBitmap(frameBitmap, 90);
        }
        return mDelegate.detect(frame);
    }

    public boolean isOperational() {
        return mDelegate.isOperational();
    }

    public boolean setFocus(int id) {
        return mDelegate.setFocus(id);
    }
}
```

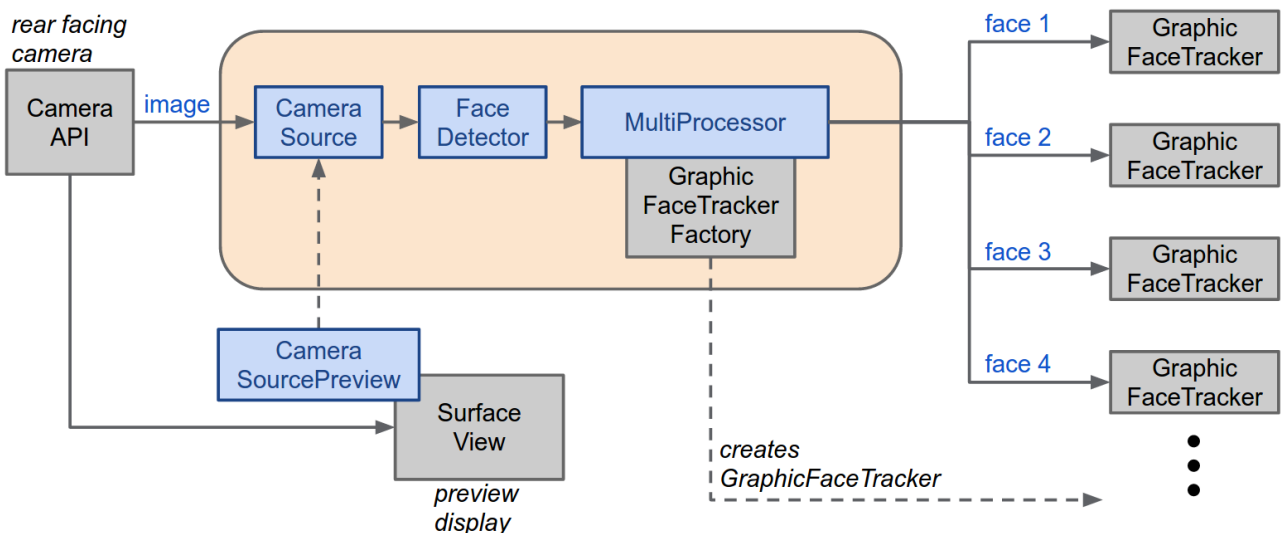
Given a detector an associated processor pipeline should be created to receive detection results. Parameters of processor and detector can be improved, however in a cost of performance:

```
myFaceDetector.setProcessor(
    new MultiProcessor.Builder<>(new GraphicFaceTrackerFactory())
        .build());
```

In opsis project processor is created in createCameraSource method of CameraSourcePreview class. Finally camera source is created to capture video images from the camera and continuously stream into the detector and its associated processor pipeline. Also, video stream shown on a phone screen does not depend on performance of face detector, processor and facial expression algorithm. It is threaded through the pipeline so requested fps will be sustained. To start camera capture UI overlay should be initialized and start method should be called from there:

```
mPreview.start(mCameraSource, mGraphicOverlay);
```

Process pipeline constructed by the code is as following:



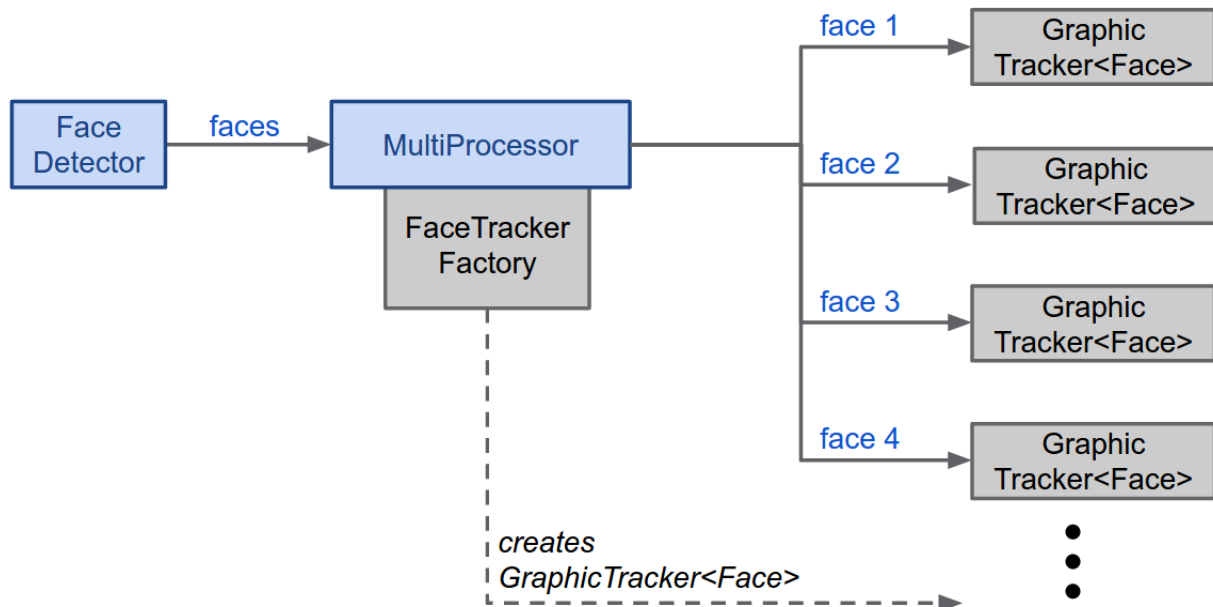
There are several detector settings which can be changed.

- mode = fast: This indicates that the face detector can use optimizations that favor speed over accuracy. For example, it may skip faces that aren't facing the camera.
- landmarks = none: No facial landmarks are required for this demo. Keeping this off makes face tracking faster.
- prominent face only = false: This demo can track multiple faces. If only a single face is required, setting this option would make face tracking faster.
- classifications = none: Smile and eyes open classification are not required for this demo. Keeping this off makes face tracking faster.
- tracking = true: In this app, tracking is used to maintain a consistent ID for each face. This is shown in the graphic overlay by the ID number that is displayed. As the face moves, this identity is generally maintained. However, there are a couple of reasons why the ID may change:

If you see an overlay that is flickering and changing color, this indicates a detection that is near the limits of what can be detected given these settings.

The face becomes obstructed and/or disappears and re-enters the view. Tracking works on a continuous basis, so any period of time in which the detector is not seeing the face will reset the tracking information.

MultiProcessor is a processor designed for working with several number of detected faces. Pipeline of its process looks like this:



The face detector will be able to detect several faces in each frame capture by the camera source. Each face is identified as a distinct face and has its own id and color. The multiprocessor creates Tracker instances for each face which are detected:

```
private class GraphicFaceTrackerFactory
    implements MultiProcessor.Factory<Face> {
        @Override
        public Tracker<Face> create(Face face) {
            return new GraphicFaceTracker(mGraphicOverlay);
        }
    }
}
```

At each new face MultiProcessor uses above factory to create a new GraphicFaceTracker instance. Moving faces are updated and routed to each corresponding face tracker instances and when face is no longer visible, the multiprocessor will destroy tracker instance of particular ID. Below is a GraphicFaceTracker which works with individual face:

```
private class GraphicFaceTracker extends Tracker<Face> {
    // other stuff

    @Override
    public void onNewItem(int faceId, Face face) {
        mFaceGraphic.setId(faceId);
    }
}
```



```

@Override
public void onUpdate(FaceDetector.Detections<Face> detectionResults,
                    Face face) {
    mOverlay.add(mFaceGraphic);
    mFaceGraphic.updateFace(face);
}

@Override
public void onMissing(FaceDetector.Detections<Face> detectionResults) {
    mOverlay.remove(mFaceGraphic);
}

@Override
public void onDone() {
    mOverlay.remove(mFaceGraphic);
}
}

```

Each GraphicFaceTracker instance creates FaceGraphic instance which is graphic object which is a part of camera overlay. FaceGraphic instances are created together with GraphicFaceTracker instances and destroyed also together with it.

Native Facial Expression Analysis algorithm execution

To call C/C++ source codes which are Opsi algorithms and VLFeat library NDK and JNI framework are needed. At the beginning they were already installed and can be used now. Make sure in your local properties that NDK and SDK directories are included like this:

```

ndk.dir=C:\\Users\\Batya\\AppData\\Local\\Android\\Sdk\\ndk-bundle
sdk.dir=C:\\Users\\Batya\\AppData\\Local\\Android\\Sdk

```

In your native-lib.cpp file include necessary libraries and sourcecodes:

```

#include <jni.h>
#include <string>
#include <opencv2/core/mat.hpp>
#include "Macros.h"
#include "FacialExpressionAnalysis.h"
#include <android/log.h>

```

Once a native-lib is called from Java code using JNI, this source code is treated as an object and next calls will refer to this object. So to initialize all the models only once, global object of FacialExpressionAnalysis type should be declared:

```

FacialExpressionAnalysis fea = FacialExpressionAnalysis();

```

From Java code, Opsi android application sends path to its folder where all trained models are stored in jstring type. To convert it to std::string type following function is created:

```

void GetJStringContent(JNIEnv *AEnv, jstring AStr, std::string &ARes) {
    if (!AStr) {
        ARes.clear();
        return;
    }

    const char *s = AEnv->GetStringUTFChars(AStr, NULL);
    ARes=s;
    AEnv->ReleaseStringUTFChars(AStr, s);
}

```

Model initialization function looks like this:

```

extern "C"
JNIEXPORT void JNICALL
Java_com_opsis_opsis2_FaceTrackerActivity_initialize(
    JNIEnv *env,
    jobject /* this */,
    jstring appFolder) {
    std::string stdAppFolder;
    GetJStringContent(env, appFolder, stdAppFolder);
    fea.initialize(1, stdAppFolder);
}

```

and expression calculation JNI call is implemented as following:

```

extern "C"
JNIEXPORT jstring JNICALL
Java_com_opsis_opsis2_FaceGraphic_stringFromJNI(
    JNIEnv *env,
    jobject /* this */, jlong ImageAddress, jlong facePointsAddress) {

    float score;
    double avi[3], ypr[3];
    std::string expr_word[2];
    cv::Rect bbox;
    cv::Mat &matAddress = *(cv::Mat *) ImageAddress;
    cv::Mat &facepoints = *(cv::Mat *) facePointsAddress;
    bbox = cv::Rect(0, 0, matAddress.cols, matAddress.rows);
    std::stringstream asd;

    FEA::RESULT res = fea.calc_expression(matAddress, bbox, false, avi, ypr,
    expr_word, score, facepoints);
    asd << facepoints;
    __android_log_write(ANDROID_LOG_ERROR, "NATIVA", asd.str().c_str());
    std::ostringstream sstream;
    sstream << res << "\n" << avi[0] << "\n" << avi[1] << "\n" << avi[2] << "\n"
    << ypr[0] << "\n" << ypr[1] << "\n" << ypr[2] << "\n" << expr_word[0] << " " <<
    expr_word[1] << "\n" << score;

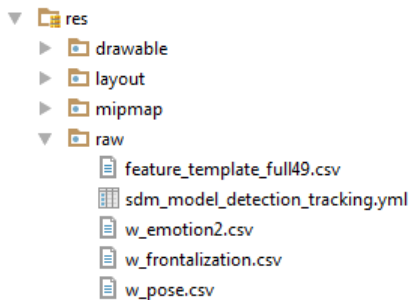
    return env->NewStringUTF(ssstream.str().c_str());
}

```

which returns a string of expressions and face orientation, which will be used in the JAVA code. OpenCV image and Facial points addresses are given as an argument and pointer to that cv::mats are created to improve speed performance of the process. In the Java code calc_expression() method is called from FaceGraphic object.

Trained models initialization

In FaceTrackerActivity in onCreate() method all models are initialized once and loaded in phone internal memory which is only accessible from the app. In Android pad in resources folder create a folder named raw and paste 5 models there. Make sure that model names starts from small letters.



File initialization function reads all models as InputStream and writes all files in application internal folder using byte buffer of 4MB size using writeFile function. Also, it creates adsc configuration file with models paths in it.

```
private void initializeFiles(File appFolder){
    InputStream SDM =
getResources().openRawResource(R.raw.sdm_model_detection_tracking);
    InputStream EMO = getResources().openRawResource(R.raw.w_emotion2);
    InputStream FRONT =
getResources().openRawResource(R.raw.w_frontalization);
    InputStream POSE = getResources().openRawResource(R.raw.w_pose);
    InputStream FET =
getResources().openRawResource(R.raw.feature_template_full49);

    writeFile(appFolder, "SDM_model_detection_tracking.yml", SDM);
    writeFile(appFolder, "W_emotion2.csv", EMO);
    writeFile(appFolder, "W_frontalization.csv", FRONT);
    writeFile(appFolder, "W_pose.csv", POSE);
    writeFile(appFolder, "feature_template_FULL49.csv", FET);

    //      Creating config file
    String textname = "config.adsc";
    File configFile = new File(appFolder, textname);
    if (configFile.exists()) return;
    FileOutputStream out1 = null;
    try{
        out1 = new FileOutputStream(configFile);
    } catch (Exception e){
        e.printStackTrace();
    }

    try {
        String string = "dir," + appFolder.getAbsolutePath()
            + "\npose,W_pose.csv"
            + "\nemotion,W_emotion2.csv"
            + "\nfrontalization,W_frontalization.csv"
            + "\nmodel,SDM_model_detection_tracking.yml\n";
        out1.write(string.getBytes());
        out1.flush();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```

private void writeFile(File folder, String newFileName, InputStream stream){

    File fileSDM = new File(folder, newFileName);
    if (fileSDM.exists()) return;
    FileOutputStream outputStream = null;
    try{
        outputStream = new FileOutputStream(fileSDM);
    } catch (Exception e){
        e.printStackTrace();
    }
    try {
        try {
            byte[] buffer = new byte[4 * 1024]; // or other buffer size
            int read;

            while ((read = stream.read(buffer)) != -1) {
                outputStream.write(buffer, 0, read);
            }
            outputStream.flush();
        } finally {
            outputStream.close();
        }
    } catch (Exception e) {
        e.printStackTrace(); // handle exception, define IOException and
others
    }
}

```

Finally, all trained models are loaded in phone at the first application launch. Starting from second launch files will not be overwritten and less launch time it will take to initialize the app.

Debugging

The easiest way to debug your code is to monitor variable values from Log. To do it in Java part import Log library and print anything you want in your Log (error, debug, info, warn, verbose, assert):

```

import android.util.Log;
int a = 5;
Log.e("Value of a is: ", "" + a);

```

This will print your log information in error log monitor (to print in warn use Log.e, to print in debug use Log.d and etc.).

To print logs in native part you also have to include logging library which is already compiled in CMake file and print .

```

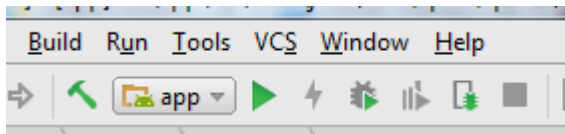
#include <android/log.h>
std::string a = "Initialization completed";
__android_log_write(ANDROID_LOG_ERROR, "Value of a is: ", a.str().c_str());

```

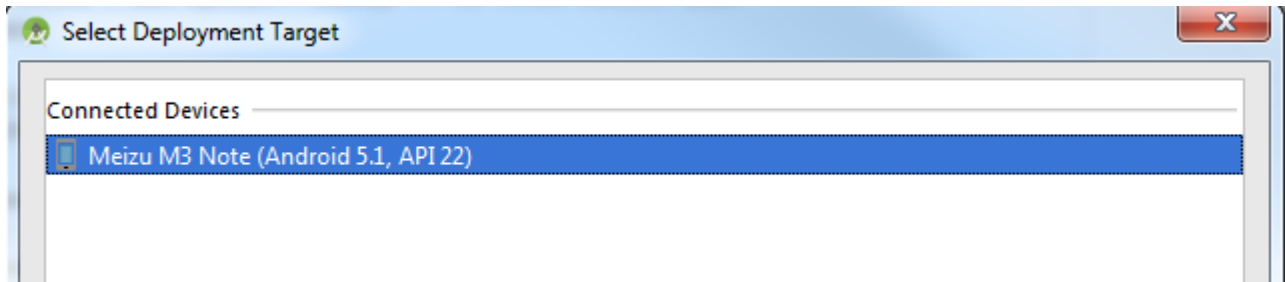
This will also print your log information in error log monitor (to print in warn use __android_log_write(ANDROID_LOG_WARN, "", "") and etc.)

How to run Opsi2 code on phone

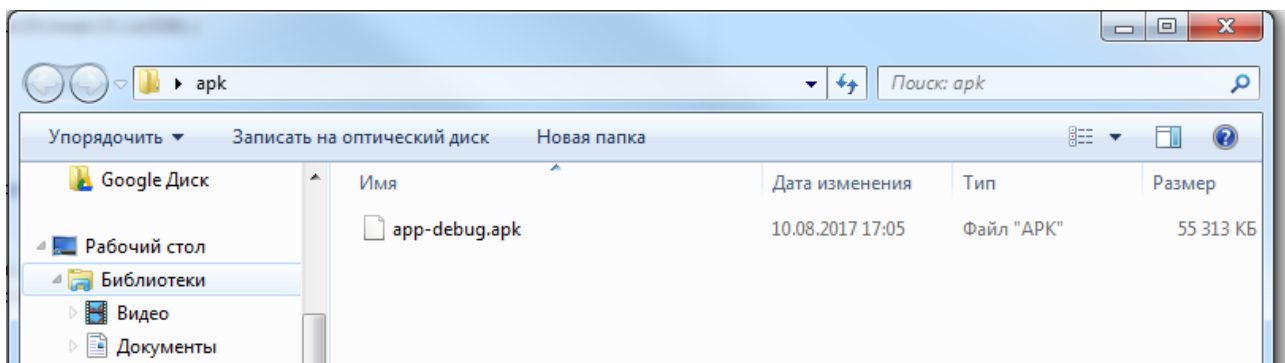
Install your phone model's Android Debug Bridge (adb) driver on your PC. Then press Run button green button:



In your ADB window choose your device and press OK button:



To create compiled .APK version of the code select Build -> Build APK. After project is synced and compiled at the right bottom corner Build APK message appears. Press Show in Explorer button to open folder of your generated apk.



Copy and paste file named app-debug.apk in your phone. In your phone find this file in explorer and open it. It will install your application in your phone. If message about unreliable source appears press allow button or check "allow unreliable application installation" tick in your settings.