Ncnn-YOLOv3 Acceleration and Implementation

XU Shenghao
Chinese University of Hong Kong
runzexu@link.cuhk.edu.hk

QUAN Wei
Chinese University of Hong Kong
1155151668@link.cuhk.edu.hk

Huang Shukang
Chinese University of Hong Kong
kangstant@link.cuhk.edu.hk

ABSTRACT
This project designs and implements the porting of YOLOv3 to mobile and uses YOLOv3 for object detection. During the migration, NCNN, which is the high-performance neural network inference computing framework, is used to quantify and reduce the size of the YOLOv3 model, ultimately enabling acceleration without compromising detection accuracy on the mobile side.

INTRODUCTION
YOLOv3 is an object detection model, based on the Darknet framework. In order to deploy YOLOv3 on mobile, we need to convert YOLOv3 into a portable model by using a framework such as NCNN. However, in the process of porting the model to mobile, there may be will encounter problems that the model being too large to load or slowing down the processing speed. So that we propose that in the YOLOv3 transformation process, we will quantization the model to speed up and reduce the size of the model. Finally, perform the objection detection on the mobile side, and verify the results in the mobile app. Figure 1 shows the flowchart of the proposed project.

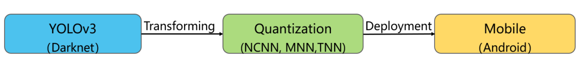

Figure 1. Flowchart of the proposed project.

YOLOV3 AND TRAINING
In this section, YOLOv3 will be introduced and analyzed in detail. After that, we will build and train our model based on the YOLOv3 baseline and feature structure.

Introduction and comparison of YOLOv3
You only look once (YOLO) is a state-of-the-art, real-time object detection model, based on the Darknet framework. YOLOv3, as the state-of-art algorithm of the YOLO series, has both preserved and improved the previous algorithms. Let's first analyze the features of the YOLOv3.

- Use Leaky Relu as the activation function.
- End-to-end training. Through one loss function to training.
- Adopt Batch normalization as the methods to regularization, accelerated convergence and avoidance over-fitting.
- Multi-scale training.

Figure 2 shows the structure of YOLOv3 network based on the model proposed in the [6].

DBL is the basic component of YOLOv3, As shown in the lower-left corner of Figure 2, it combines with the convolutional layer, batch normalization and the Leaky Relu.For YOLOv3, batch normalization and Leaky Relu are already minimal components that are connected to the convolutional layer and cannot be subdivided (except for the last layer of convolution), which together make up the smallest component in the network.

Resn is another important basic component of YOLOv3, which together with *DBL* builds the backbone of YOLOv3 namely Darknent-53. *Resn* is the combination of zero padding and *DBL* component and with several residual units (res unit). zero padding and *DBL* perform the down-sampling in the *Resn*. The n in the *Resn* represents the number, like res1, res2, it indicates how many residual units are in the *Resn*.
The role of the *concat* is Tensor stitching, which stitches the Darknet middle layer with the up-sampling of one of the later layers. For example, the stitching with two Tensor $32*32*128$ and $32*32*256$. After the Tensor stitching, we will get the Tensor with size $32*32*384$.

The backbone of the YOLOv3 is the Darknet-53, as shown in Figure 3 [6]. Inside the whole network structure, there is no pooling layer or fully connected layer. During the forward propagation, the dimensional transformation of the tensor is achieved by changing the step size of the convolution kernel. For example, if we adopt $stride = (4,4)$, this is equivalent to reducing the size of the image to $1/16$ of the original size. From Figure 3, we can see that the backbone shrinks the output feature map to 1/32 of the input. Darknet-53 combines by several *Resn* components. And each *Resn* involve $(1+n*1)$ convolutional layer. So, in figure 2 we can see that there one conventional layer in *DBL* add with 5 *Resn* blocks and one fully connected layer, where $1+(1+2*1)+(1+2*2)+(1+2*8)+(1+2*8)+(1+2*4) = 52+FC = 53Conv$.

Therefore, the entire backbone network contains 53 convolutional layers of Backbone. As can be seen from Table 1, although YOLOv3 utilizes the Residual structure of ResNet [1], it is not more efficient than ResNet-101 and ResNet-152. However, compared with YOLOv2 [5], which does not adopt

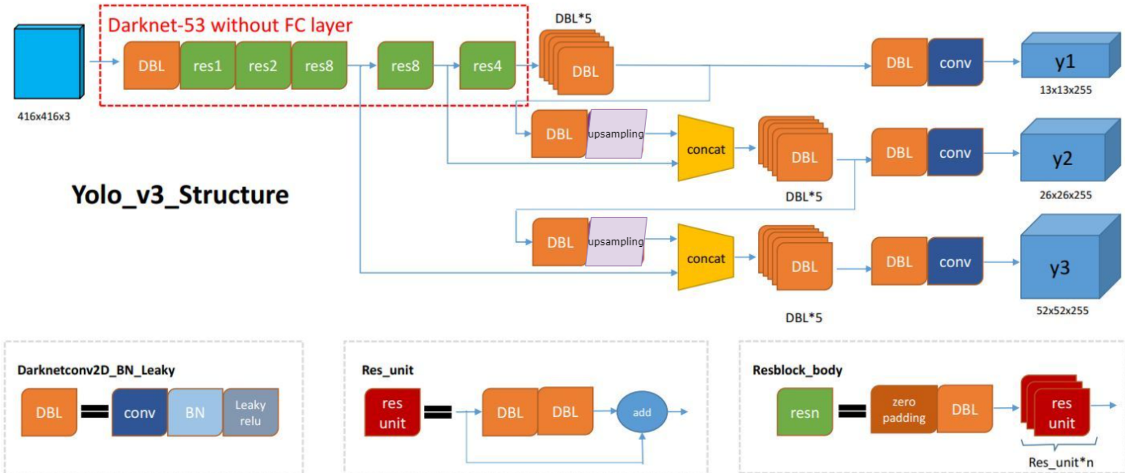

Figure 2. Structure of YOLOv3 network

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
	Convolutional	32	1 × 1	
1×	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
	Convolutional	64	1 × 1	
2×	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
	Convolutional	128	1 × 1	
8×	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
	Convolutional	256	1 × 1	
8×	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
	Convolutional	512	1 × 1	
4×	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 3. Darknet-53

the Residual structure, the network of v3 is more efficient.

When we look at the detection metric of the mean average precision (mAP), which shown in Table 2, YOLOv3 performs slightly worse than RetinaNet, but achieves higher precision than the SSD variant. This result was also obtained for mAP at intersection over unit(IoU) =0.5. Combine Table 1 and Table 2, Darknet-19 perform best in speed. However, for YOLOv3, it pursues performance on the basis of ensuring real-time performance (FPS =78).

Backbone	*Top-1*	*Top-5*	*FPS*
Darknet-19 [5]	74.1	91.8	171
ResNet-101 [1]	77.1	93.7	53
ResNet-152 [1]	77.6	93.8	37
Darknet-53 [6]	77.2	93.8	78

Table 1. Comparison between different Backbones.

One-stage methods	**Backbone**	*AP*	AP_{50}
YOLOv2 [5]	Darknet-19	21.6	44.0
SSD513 [3]	ResNet-101-SSD [1]	31.2	50.4
RetinaNet [2]	ResNet-101-FPD	39.1	59.1
YOLOv3 [6]	Darknet-53	33.0	57.9

Table 2. Comparison between different One-stage methods.

Training YOLOv3

In the last section, we introduce the structure of the YOLOv3 and compare it with other famous networks. After we have the basic concept about the YOLOv3, then we can start building our YOLOv3 model from the scratch.

Training data

For the data sets, we selected the *PASCAL VOC 2007* and *PASCAL VOC 2012* data sets, with a total of $3,3043$ images. The training set contains the test and train set of the *PASCAL VOC 2007* add with the train and valuation set of *PASCAL VOC 2012*. Only adopt *PASCAL VOC 2007* test set for valuation. However, Darknet requires that a label file be generated for the image data set in txt format, the format of the label file need contains five parameters, which are 'object-class', 'x-coordinate', 'y-coordinate', 'width', and 'high' of the images.

To generate these label files we need a python script. In the Figure 5, we can see the python code which use to get the label and coordinate of each image. After running the script for the data sets, we can get the txt files used for training, which contain the five parameters, which like the format shown in the Figure 6.

Configuration file

After we finish the training data preparation, we modify the configuration file to reflect our own situation. We configure the *voc.data* and *yolov3-voc.cfg* separately, with the following parameters:

```
Region 82 Avg IOU: 0.173379, Class: 0.518647, Obj: 0.507739, No Obj: 0.450314, .5R: 0.090909, .75R: 0.000000,  count: 11
Region 94 Avg IOU: 0.227878, Class: 0.713136, Obj: 0.567582, No Obj: 0.514300, .5R: 0.076923, .75R: 0.000000,  count: 13
Region 106 Avg IOU: 0.120734, Class: 0.864854, Obj: 0.883645, No Obj: 0.568066, .5R: 0.000000, .75R: 0.000000,  count: 3
Region 82 Avg IOU: 0.214248, Class: 0.674125, Obj: 0.476301, No Obj: 0.450282, .5R: 0.000000, .75R: 0.000000,  count: 7
Region 94 Avg IOU: 0.031064, Class: 0.214630, Obj: 0.440358, No Obj: 0.514154, .5R: 0.000000, .75R: 0.000000,  count: 1
Region 106 Avg IOU: nan, Class: nan, Obj: nan, No Obj: 0.570022, .5R: nan, .75R: nan,  count: 0
Region 82 Avg IOU: 0.187642, Class: 0.346217, Obj: 0.454615, No Obj: 0.449209, .5R: 0.090909, .75R: 0.000000,  count: 11
Region 94 Avg IOU: 0.340648, Class: 0.333973, Obj: 0.845724, No Obj: 0.512727, .5R: 0.000000, .75R: 0.000000,  count: 2
Region 106 Avg IOU: nan, Class: nan, Obj: nan, No Obj: 0.566085, .5R: nan, .75R: nan,  count: 0
1: 2404.144043, 2404.144043 avg, 0.000000 rate, 2975.202049 seconds, 64 images
```

Figure 4. Training process of the YOLOv3

```python
def convert(size, box):
    dw = 1./(size[0])
    dh = 1./(size[1])
    x = (box[0] + box[1])/2.0 - 1
    y = (box[2] + box[3])/2.0 - 1
    w = box[1] - box[0]
    h = box[3] - box[2]
    x = x*dw
    w = w*dw
    y = y*dh
    h = h*dh
    return (x,y,w,h)
```

Figure 5. python script to generate parameter of images

```
2008_000002.txt
19 0.48 0.40266666666666667 0.8280000000000001 0.752
```

Figure 6. parameter of images contain in the generated txt files

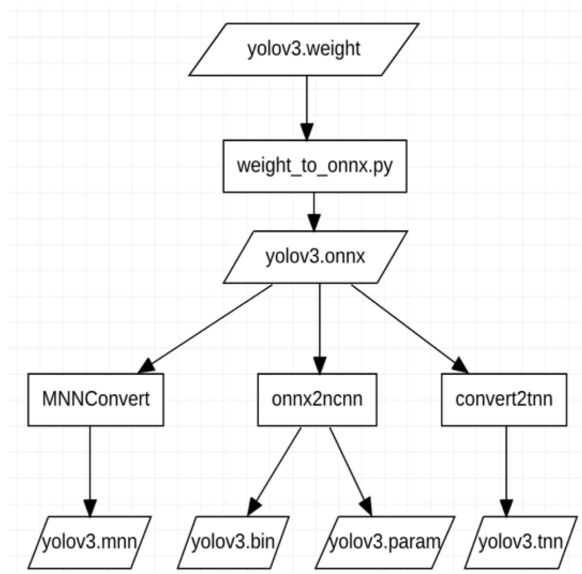

Figure 7. Dataflow of model convert

- Classes=20
- batch =64
- subdivision=16
- learning rate=0.001
- max batches=100000

Then, Due to YOLOv3 is base on the backbone of Darknet-53. So, we download the pre-trained weight file of the Darknet-53, then we use the above configure file to staring our training.

After training

As the training of the YOLOv3 processing, after 100000 iterations, we will get the weight file of the YOLOv3, namely **yolov3.weight**.

YOLOV3 QUANTIZATION

In this section, this report will first introduce our motivation to quantify YOLOv3, and then we will try to use different popular opensource tools to get the quantified YOLOv3 model. Finally, we will give some analysis of different quantization tools.

Motivation

After completing the training of YOLOv3(darknet), we got the weight file of YOLOv3. However, the YOLOv3 is too large to be loaded during the Android deployment process, especially in some outdated devices. In order to solve this problem, the general idea is quantization. Quantization can accelerate forward speed of the model by converting floating point computations in the original model into int8 computations. At the same time, it compresses the original model, that is, quantize the float32 weights into int8 weights.

Model Convert

For the purpose of comparing the effect of quantized YOLOv3 in various lightweight mobile network frameworks, we need to convert YOLOv3.weight into an onnx model, which is a kind of popular Cross-frame model intermediate expression framework. As shown in the figure, YOLOv3.weight is converted to YOLOv3.onnx through the weight-to-onnx.py code. Then we need to compile MNN[7], NCNN[4] and TNN[8] respectively. For MNN, we can convert YOLOv3.onnx to YOLOv3.mnn with the usage of MNNConvert tool; for NCNN, we have the ability to get YOLOv3.bin and YOLOv3.param through its onnx2ncnn executable file; for TNN, YOLOv3.tnn can be successfully converted by the convert2tnn tool.

Figure 8. Quantization tools

Model	*Origin(MB)*	*Quantized(MB)*
YOLOv3.mnn	247.6	63.4
YOLOv3.bin(ncnn)	247.3	63.0

Table 3. Model size comparison

Actually, we have tried three different quantization tools in NCNN, MNN and TNN in this project. By compiling the above three kinds of lightweight networks, we are able to get the relative executable file, that is, quantize.out, ncnn2int8 and converter.py respectively.

Quantization Process and Results

Figure 9 illustrates the process of quantifying the model with mnn and ncnn respectively. Table 3 shows the comparison of model sizes before and after quantification.

With regard to MNN quantization. At the first stage, we should build MNN with specified parameter to compile quantization tools. The second step is to write the config file with the parameters you preferred. Finally, we run quantize.out to quantize the model. The NCNN quantization is similar with MNN. It can be divided into three steps, optimize graphic, create calibration file and do quantization

Quantitative Analysis

All experiments in this project are performed on a macbook pro 2018 with Intel Core i7-8750H and 16GB memory.

MNN

We used the following command to analyze the model before and after quantification.

./pictureRecognition.out yolov3-quan.mnn ./images/test.jpg
./pictureRecognition.out yolov3.mnn ./images/test.jpg

As shown in table 3. Inference time is tested using MNN official Test Tool. All MAP results are evaluated using the first 100 testing images in order to save time. The model is quantized using official MNN tool. The poor inference speed is due to arm-specified optimization.We do not have the ability to test multithreading due to the poor support of OMP in Mac OS. Consequently, we abandon the use of MNN's quantitative model because its inference time is too long to run on the x86 instruction set simulator on Mac.

Model	*InputSize*	*Thread*	*InferenceTime*	*mAP*
YOLOv3	416	1	100.4	0.721
YOLOv3-quan	416	1	1475.2	0.700

Table 4. MNN Quantization Comparison.

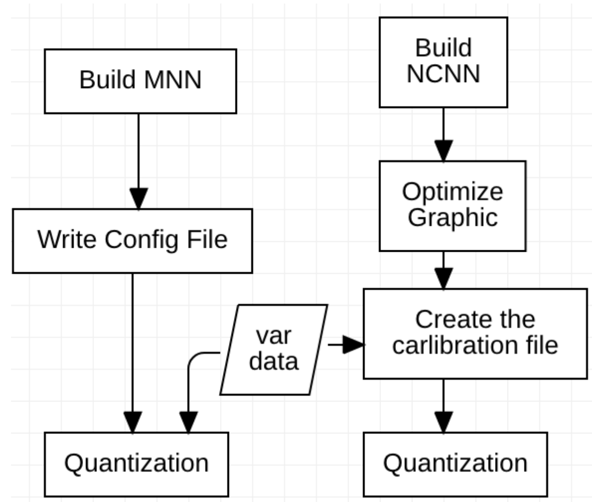

Figure 9. Quantization process of MNN and NCNN

Model	*InputSize*	*Thread*	*InferenceTime*	*mAP*
YOLOv3	416	1	138.2	0.717
YOLOv3-quan	416	1	148.9	0.714

Table 5. NCNN Quantization Comparison.

NCNN

By modifying the sample code provided by NCNN, we can compile a yolov3 executable file, the input parameters of which are bin, param file and a picture to be recognized. We tested the YOLOv3 model before and after quantification, and the data is shown in Table 5. The inference time is the output log of yolov3 executable file. Other meanings of the columns in Table 5 are the same as in Table 4.

IMPLEMENTATION ON ANDROID

This part is mainly to port ncnn to Android platform, which will be divided into the following parts: use of Android NDK, deployment of related files, core code and implementation, demo results.

Use of Android NDK

In order to port YOLOv3 to an Android application, some preparations need to be made first.

The method of compiling the .so file is using camke. So, it is equivalent to select "Include C++ Support" when creating a new Android project. The purpose of doing so is to using Java interface to call C++, namely using NDK technology. In the Figure 7, the NDK layer is called by JNI on the top of the app.

Android development uses the NDK to compile C and C++ code into native libraries, which are then subsumed into the APK using Android Studio's integrated build system, Gradle.

Java code can access the functionality in the native libraries through the Java Native Interface (JNI) framework. JNI is a feature of Java that calls native languages and is not directly related to Android.

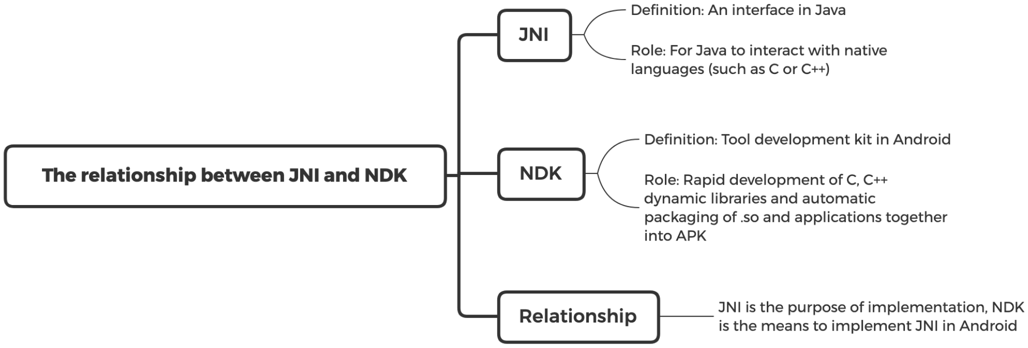

Figure 10. Relationship Between JNI and NDK

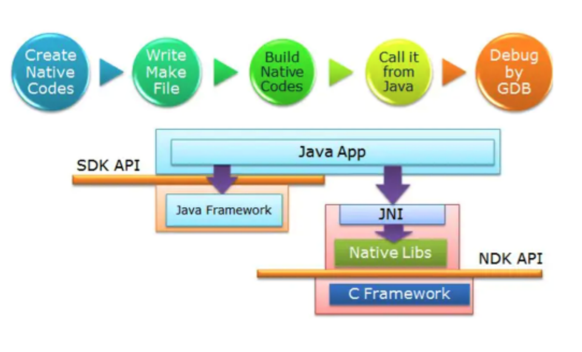

Figure 11. NDK Layer in Android Application

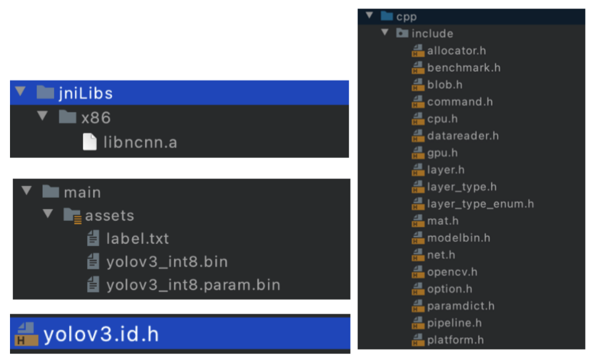

Figure 12. Related Files

The process of developing with JNI is to first declare Native methods in Java, then compile the above Java source file javac and export JNI header files. Implement the Native methods of Java in C++. Finally, compile the .so library file.

About the relationship between JNI and NDK, it is shown in the Figure 8.

Deployment of Related Files

In this part, the files generated in the quantification phase need to be configured into the Android project. Specifically, these files are included below and shwon in the Figure 9.

- include/
- libncnn.a
- Model Files
- yolov3.id.h

The ncnn build-Android compilation generates two folders, include and lib. The include folder contains frequently used header files, while the lib folder contains the libncnn.a file, which can be interpreted as packaging ncnn into a form that can be imported into Android Platform.

In the model files, yolov3-int8.bin is the converted network weight and yolov3.param.bin is the converted network model parameter. And label.txt is the label file for detecting objects.

yolov3.id.h is a compiled file that is encrypted by ncnn.

The preparation completed once all these files have been deployed in their respective locations of the Android project.

Core Code And Implementation

This part is mainly the core code and concrete implementation part of Android project.

First is the layout file of this Android Application, which can be seen in the Figure. This application is mainly a demo of selecting a picture and detecting it, with a button for selecting and a button for detecting. The top half of the application interface shows the selected picture, and the bottom half shows the information about the detected objects in the picture after detection, the accuracy, the detection speed and other detection information.

Next, create a new $YOLOV3.java$ file in the Java folder. This Java class is used to load the lib file and also defines the following two methods:

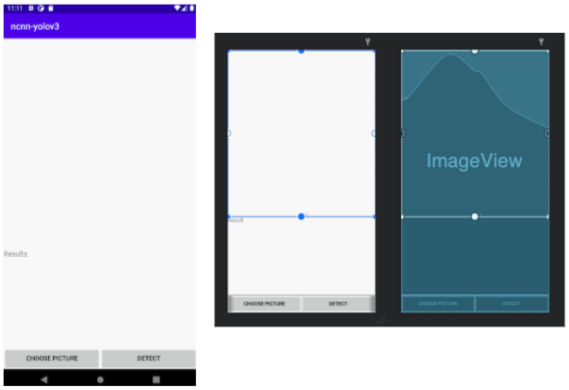

Figure 13. Layout File

- public native boolean Init(byte[] param, byte[] bin);
- public native float[] Detect(Bitmap bitmap);

The **Init()** function for initialization and the **Detect()** function is used for detection.

In the **MainActivity.java** file, the initialization of the YOLOV3 class is first implemented. It instantiate the two interfaces, namely **Init()** and **Detect()**, and later call the C++ function by JNI. Also need to implement the initialization function **initNCNNYOLOV3()**. This function loads yolov3.param.bin and yolov3.bin,and then pass the files into Java's NDK interface. Finally, there is **init view()** function that needs to be implemented. This function is for detecting images and drawing rectangular boxes, which calls two functions for loading labels and outputting information about detection.

In order to use NDK, a new **YOLO3-jni.cpp** file is also needed. Modify this file to implement two function calls of the Java class YOLOv3 via NDK, which correspond to **Init()** and **Detect()** in Java, respectively.

- JNIEXPORT jfloatArray JNICALL com-example-DNN-ncnnyolo-yolo-Init(JNIEnv *env, jobject obj, jbyteArray param, jbyteArray bin)
- JNIEXPORT jfloatArray JNICALL com-example-DNN-ncnnyolo-yolo-Detect(JNIEnv* env, jobject thiz, jobject bitmap)

After completing the above core code, we still need to make some changes to the following configuration files.

- CMakeList.txt
- build.gradle
- AndroidManifest.xml

The modification of the above configuration file mainly refers to sample in the NCNN open source library for configuration.

So far, the implementation part of this Android application has been basically completed, and the following is the demo result of this demo.

Demo Result

Run the program in Android Virtual Device(AVD) and we can get a demo to detect pictures which is shown in Figure.

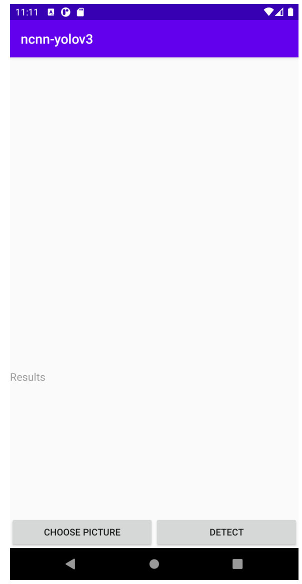

Figure 14. Android Demo

For the recognition of individual objects, which is shown in the Figure, the demo performs very well. After our tests, the detection of individual objects is fast and highly accurate.

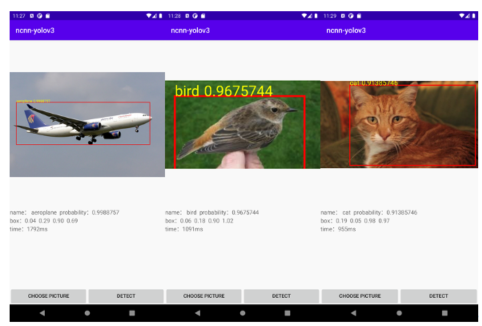

Figure 15. Single object detection

This demo also supports the detection of multiple objects, as seen in the figure. After our tests, when performing multi-object detection, the detection speed of certain objects will be slower in some more complex scenes, but it does not have much impact on the overall speed and accuracy.

CONCLUSION

We presented a flow of training, quantization and deployment of YOLOv3 in this project. Initially, we trained YOLOv3. Then we quantized the model using MNN, NCNN and TNN respectively. We got the quantized model successfully in MNN and NCNN while TNN has problems with model conversion. After experiments, we decided to choose NCNN model to deploy on android platform because the MNN model has poor inference time. Finally, we ran the quantized YOLOv3 network successfully in the mobile app and successfully detected

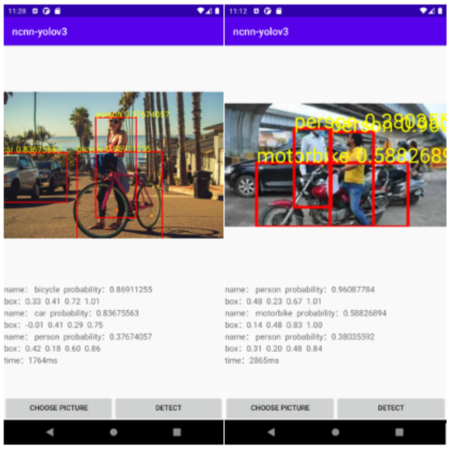

Figure 16. Multi-object detection

single or multiple targets with guaranteed detection speed and accuracy.

REFERENCES

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). `http://arxiv.org/abs/1512.03385`

[2] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. *CoRR* abs/1708.02002 (2017). `http://arxiv.org/abs/1708.02002`

[3] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. 2015. SSD: Single Shot MultiBox Detector. *CoRR* abs/1512.02325 (2015). `http://arxiv.org/abs/1512.02325`

[4] Zuo Zhang Nihui. 2019. Tencent: NCNN: a high-performance neural network inference computing framework optimized for mobile platforms. [EB/OL]. (2019). `https://github.com/Tencent/ncnn`.

[5] J. Redmon and A. Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6517–6525. DOI:`http://dx.doi.org/10.1109/CVPR.2017.690`

[6] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *arXiv* (2018).

[7] MNN Team. 2019a. Alibaba: MNN: a lightweight deep neural network inference engine. [EB/OL]. (2019). `https://github.com/alibaba/MNN`.

[8] TNN Team. 2019b. Tencent: TNN: a high-performance and lightweight inference framework for mobile devices. [EB/OL]. (2019). `https://github.com/Tencent/TNN`.