Performance Evaluation of RNN on Mobile Devices

Runyu Zheng, Shangquan Sun,
University of Michigan
runyuz@umich.edu, sunsean@umich.edu

Abstract

Recurrent Neural Networks (RNNs) have gained considerable attention in various areas like language translation and speech recognition. Besides, there is a trend to spread computation towards edge device such as mobile phone to enhance user experience. Also data privacy and cloud availability make it necessary to adapt inference model on edge devices. However, running a compute-intensive task on the mobile CPU could be difficult due to limited computing resources. In this project, we evaluate the performance of LSTM layer on mobile devices and compare different methods for optimizing the LSTM layer. Compared with the baseline LSTM layer implemented by NCNN, we achieved 1.23x speedup. We also found that the best performance is achieved with 4 threads.

1 Introduction

With the development of hardware, e.g. increasing memory and computation resources, the methods of machine learning become more and more popular. Among them, deep learning and Recurrent Neural Network have got the most attention. Generally, the implementation of a machine learning method involves two phases, i.e. offline training and online inference. For the phase of training, we put more attention on increasing accuracy and reducing computation cost and thus training time. Similarly for the phase of inference, more attention is put on reduce latency.

However, it is usually difficult to speed up RNN because of the different properties of RNN from other deep neural networks. RNN contains recurrent chain-like data flow structure and the property of chronological dependency. So one part of data flow must wait for its previous states to finish, which makes the execution of both training and inference of RNN models less efficient and hard to accelerate.

There are various possible optimizations methods to speed up LSTM layer, such as quantization, GPU acceleration, etc. Quantization and GPU acceleration has proved their effect for almost all deep neural networks. General Matrix Multiply (GEMM) is a general optimization method when conducting matrix multiplication. Since RNN must contain plenty of matrix multiplication operations, GEMM is another method that is worth to experiment.

1.1 Machine Learning on Mobile Devices

The traditional way of training, evaluating and predicting on machine learning models is run on desktops or servers with more powerful computation resources. Typically, they are not done on mobile devices because of the limited computation resources of most mobile devices or edge devices. But as the development of mobile hardware and the advancing machine learning techniques, such as model compression, doing some of the machine learning tasks on mobile devices becomes increasingly promising, especially for inference, which costs relatively fewer computations. Not only driven by the development of the techniques, more and more requirements of machine learning applications in people's daily using Apps on mobile phones accelerate the process of adapting machine learning tasks on mobile devices. Among them, the task of inference costs the fewest computation resources and require the shortest latency, making it the most noteworthy job to do on

mobile. Also the demand of short latency makes it less possible to replace inference on mobile with inference on some cloud server linking to device through the Internet, since the possible unavailability of the Internet and longer wait of data transmission.

1.2 Inference on Mobile Devices

As running inference task on desktop or server, performing inference on mobile devices is usually run on CPU. Some similar methods to those applied on desktop or server can also be adapted on mobile, such as GPU acceleration, mobile compression like quantization. But some mobile-oriented methods can be created specifically for mobile hardware and environment as well, such as Neural Networks API (NNAPI) developed by Google. Although the GPU acceleration can boost the inference task of most machine learning models, it cannot effectively boost the inference task of recurrent neural networks, which consists of sequential chain-like execution and data flow. For NNAPI, it is not effective for some of mobile devices if they do not contain GPU or digital signal processor. Therefore, investigating on acceleration of inference for recurrent neural networks is a pre-mature topic in the field of machine learning on mobile devices.

2 Background

2.1 Recurrent Neural Network (RNN)

Like deep neural networks, Recurrent Neural Network (RNN) is a set of feedforward neural networks algorithms. Unlike traditional deep neural networks, RNN assumes that there is a dependency between inputs and utilizes the sequential information to make prediction. Typically, it contains an internal memory to store and process the sequential inputs. Therefore, it is widely applicable to those tasks involving chronological inputs of data, such as speech recognition, machine translation, etc.

2.2 Long Short Term Memory network (LSTM)

Long Short Term Memory network (LSTM) is one of typical networks of RNN, which was first introduced by Hochreiter et al. [1] The key idea of LSTM is to weight the contributions of long term memory and short term memory, using a chain-like struture. This process is very like the human process of memorization, forgetting some of the older data and putting a higher weight of memory for newer data. With this property, LSTM can be applied to the tasks where data features consist of some chronological dependency. Also due to the property of chain like structure, it is less possible and effective to be accelerated by distributed training and GPU acceleration.

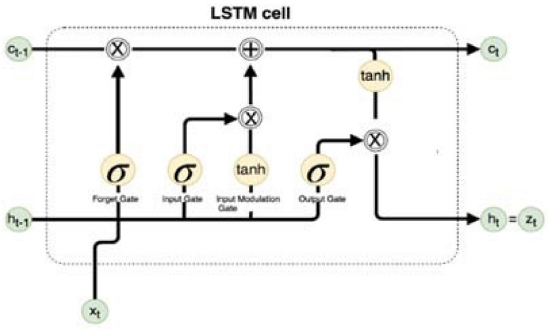

Figure 1: Structure and data flow of LSTM [1]

The first branch of the diagram, i.e. 1, is the forget gate, where the sigmoid function will receive the concatenation of previous state (h_{t-1}) for previous time step and current data input (x_t) and return a output ranging from 0 to 1. An output of 0 indicates to discard the current input and an output of 1 indicates to keep it. Eq. 1 is the mathematical expression of the data flow in this unit of LSTM. f_t is the result of forget gate and W_f and b_f are the parameters of the forget gate unit.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{1}$$

The second branch of the diagram, i.e. 1, is the input gate, where sigmoid function again determines which values to forget and which values to keep and tanh function determines weight/importance ranging from -1 to 1, to each kept values. The operation of multiplication is to apply the output of tanh function only for those kept values determined in sigmoid function. Eq. 2 is the mathematical expression of the data flow in this unit of LSTM. i_t is the result of input gate and W_i, W_C, b_i

and b_C are the parameters of the input gate unit. $\tilde{C}_t$ is an intermediate result

$$\begin{aligned} i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\ \tilde{C}_t &= \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \end{aligned} \tag{2}$$

The third branch of the diagram, i.e. 1, is the output gate. The sigmoid function determines which values to be kept and which to be discarded, while the tanh is also to determine the importance of each value. The multiplication operation is to pass the kept values to the next state as the output of the current state. Eq. 3 is the mathematical expression of the data flow in this unit of LSTM. o_t is the result of output gate and W_o and b_o are the parameters of the output gate unit. C_t is the result of summing previous two gates' results.

$$\begin{aligned} o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \\ C_t &= f_t .* C_{t-1} + i_t .* g_t \\ h_t &= o_t * \tanh(C_t) \end{aligned} \tag{3}$$

2.3 Quantization

The models of deep neural networks tend to be stored in an integrated file containing all the necessary information of the model for the task of inference, such as the structure of the network, its weights, the information of its optimizer or loss function, etc. When an inference engine loads and inference on a model, it passes the new input data into the model and after computations through all layers of neural networks, an output is obtained. Originally, all numeric variables and weights in computation steps are floating point values, such as 32-bit floating point values. But this causes the original model to be relatively large since all weights must be stored in the format of floating point values. Also it causes the latency and computation cost during inference to be remarkable since all computations must be also in the format of floating point values, especially for those mobile devices whose hardware resources are limited.

Therefore, an idea of converting some numeric values of a model from high-precision floating point into low-precision floating point or integer is developed. As a result, model size as well as inference latency can be significantly reduced, while degradation of accuracy to some extent may occur. Comparing to the demand of adapting machine learning models into mobile environment, such degradation of accuracy is tolerable. Such degradation can be avoided if quantization-aware training is employed during training a model. Tensorflow provides three options of quantization, dynamic range quantization, full integer quantization, and float16 quantization. Dynamic range quantization only converts the weights of high-precision floating point to 8-bit. Float16 quantization only converts the weights of high-precision floating point to 16-bit. Full integer quantization converts all weights and as many operations as possible from floating points into integer.

2.4 GPU Acceleration

Comparing to CPU, GPU (graphics processing unit) is mainly used in processing graphic computation. CPU is more complicated and consists of fewer ALU, which causes it more powerful when conducting complicated logic operations. However, GPU contains many ALU and floating point operation unit. Unlike CPU, GPU is designed to do massive mathematical computations. Also since GPU consists of multiple cores and lots of high-speed memory, data parallelism and parallel computing can be more easily implemented on GPU and improve the efficiency of doing numeric computing.

The main idea of GPU acceleration is to distribute some matrix or graphics computations from CPU to GPU during executing programs. GPU acceleration has been proved to be effective when doing training and inference of some deep neural networks. But its effect for boosting the inference of RNN is not as notable as that for convolutional neural networks, since RNN contains a chain-like data flow that makes parallel computing less capable to be applied for acceleration. It means that one layer must wait for its previous states to finish, which makes data parallelism and parallel computing less effective in the circumstance. But whether GPU acceleration can function for a RNN model or not depends on the proportion of such LSTM layers and their execution time. If the main bottleneck of doing inference for one model mainly lies on other layers or parts rather than LSTM, GPU acceleration may still work for the model. Therefore, we decide to do

experiments about GPU acceleration.

2.5 Neural Networks API

NNAPI is designed by Google aiming to run deep neural network models on Android environment. It is between the underlying hardware resources and the machine learning framework, scheduling and distributing the execution of inference tasks on various available hardware resources in the environment of mobile devices. The system architecture of NNAPI is shown in Figure 2. Applications need to use machine learning frameworks and NNAPI to run inference with hardware acceleration on Android devices. Android's neural networks runtime is responsible for scheduling and distributing the execution workloads among different types of hardware units, including digital signal processor, specialized processor and GPU. When using NNAPI, a directed graph consisting of multiple operation units representing the data flow and structure of a model is necessary. NNAPI receives the directed graph and makes its determination on how to schedule and distribute the workloads. Such distribution of workload can efficiently reduce the peak computation workload and make sure the execution of different types of operations can be done in their suitable hardware units. For example, matrix computation can be distributed to GPU, while logical operation can be executed by CPU. As a result, NNAPI can effectively reduce the latency of inference.

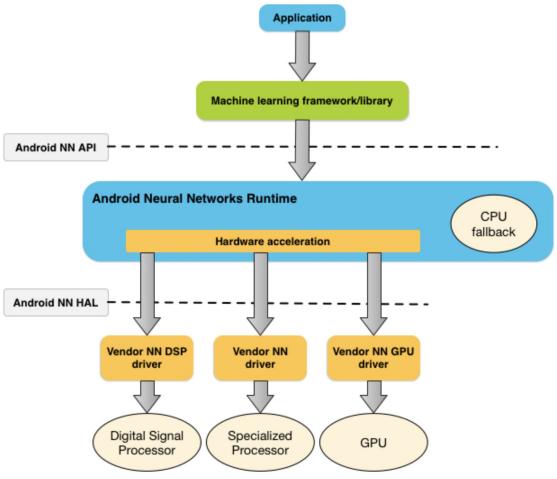

Figure 2: Structure of NNAPI [2]

2.6 GEMM

LSTM layer operations involves matrix multiplication. So the optimization methods for matrix multiplication can also be applied to LSTM. GEMM refers to general matrix multiplication. The general form is:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} \tag{4}$$

where C is a $M \times N$ matric, and A is $M \times K$, B is $K \times N$ matrix. If we compute the element in C one at a time, it would need three for loops and the total memory access time would be high because of the poor use of temporal locality.

To bake better use of temporal locality, we can divide the matrix into several sub-blocks, like 4×4. In this way, we can reuse the data read from memory in the inner two for loops. In addition, we can use more register to store the intermediate result and use vectored memory access and computation to in crease data level parallelism.

Here's the diagram and pseudo-code for the algorithm:

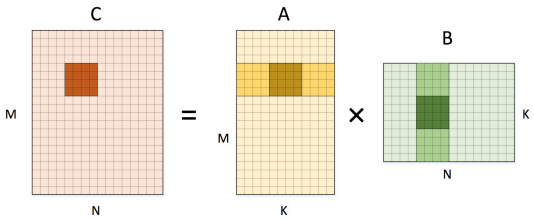

Figure 3: Matrix Multiplication of 4× 4 blocks

Here's the pseudocode for matrix multiplication:

```
for (int m = 0; m < M; m++) {
    for (int n = 0; n < N; n += 4) {
        C[m][n+0] = 0;
        C[m][n+1] = 0;
        C[m][n+2] = 0;
        C[m][n+3] = 0;
        for (int k = 0; k < K; k++) {
            C[m][n+0] += A[m][k] * B[k][n+0];
            C[m][n+0] += A[m][k] * B[k][n+1];
            C[m][n+0] += A[m][k] * B[k][n+2];
            C[m][n+0] += A[m][k] * B[k][n+3];
        }
    }
}
```

3 Related Work

3.1 Mobile Inference Engines and Models

Tensorflow Lite (TFLite) is one of the most prevalent open-source frameworks for deep learning inference

on edge device. PolimiDL [3] is another open-source deep learning inference framework, but has less competitive performance for large models. Integrated into TFLite, Lee *et al*. [4] develop a mobile-GPU accelerating inference engine. DeepX [5] accelerates the deep learning execution by lowering the device resources. Guo [6] evaluates the performances of mobile inference and cloud-based inference, finding that the on-device inference performance is lower. DeepRT [7] "coordinates underlying heterogeneous resources to support predictable inference latency and power consumption simultaneously". Tencent develops an efficient inference engine, NCNN, on various environment supporting multiple backends [8].

3.2 Hardware Devices for Mobile Inference

Many hardware manufacturers have responded to the demand of more efficient and faster inference on mobile devices, by providing many software development kits (SDK), such as Huawei HiAI SDK [9], [10], etc. Also Android Studio [11] provides an integrated development environment for Android operating system, which allows developer to create, simulate and work on an Android virtual devices (AVD) on a desktop environment.

3.3 Benchmark for Mobile Inference

MLPerf [12] offers several benchmarks, such as Resnet50, MobileNets-v1 and GNMT, for mobile NN inference based on a trained model. MLPerf supports the benchmarks for two several common datasets, e.g. ImageNet [13], COCO [14].

4 Implementation

4.1 System Setup

Our target platform is mobile phones. The available mobile phone is HWAWEI P30 with Kirin980 CPU, which has 2 Cortex-A76 cores(Arm v8, 2.6GHz), 2 Cortex-A76(Arm v8, 1.92GHz) and 4 Cortex-A55 cores(Arm v8, 1.8GHz). We implement some optimizations for RNN in C++ in Tencent NCNN framework [8]. We compile them using Android NDK v21.0.6 with compiler optimizations enabled. We use ARM Neon Intrinsics for SIMD instructions, and OpenMP library for SIMT. The GPU support is enabled with Vulkan.

4.2 Framework

The framework we are using is Tencent's NCNN. It is based on Caffe style model for describing the Convolutioal Neuron Network in C++. The Net consists of Layers and Blob. Blob is used to store the data. The data can either be the input data, or weight matrix of the model, or some intermediate results. The data is stored as a 3D matrix (height, width, channel). The net consists of many blobs, with layers describing the connection between the blobs. NCNN framework itself has support for LSTM operation, however, the operation is not optimized. For the optimization part, we do the optimization for the LSTM layer, to speed up the computation. Besides, NCNN framework has default Vulkan support to delegate the workload to GPU.

4.3 Model

To verify the correctness of the optimized layer, we integrated the optimized operation with a fully functional model, that is one for Optical Character Recognition (OCR). We have developed an app for testing purpose, it is able to convert an image to characters. The model is presented here[15].

4.4 Optimization

For a baseline LSTM layer, the for loop is unrolled first. For each iteration, the value for the 4 gates are updated first with the weight matrix (W_f, W_C, W_i, W_o). Then the hidden state h and cell state C are updated accordingly. By doing so, the data reuse is poor since for each loop, all weight matrix should be reloaded again. So we propose the following method to enhance performance with GEMM, as shown below.

Input: weight matrix for X, W_x, weight matrix for H, W_h, Input matrix X, Bias B
Output: Output matrix O

Calculate gate matrix for all loop $G = W_x \cdot X$, optimized with GEMM.

Set hidden state h, and cell state C to vector with all 0.
for each LSTM iteration t **do**
Get gate vector f_t, o_t, c_t, i_t from G for current iteration t.
Add Bias to each Gate vector.
Calculate $W_h \cdot h$, update each gate vector accordingly.
Apply sigmoid to f_t, i_t, o_t, apply tanh to c_t.
Update C with gate vector and previous C value.
Update h with C and o_t.
Write h to output O_t.
end for

In this way, all the influence for input x and weight matrix related to x has been eliminated before the LSTM layer is unrolled. So for each iteration, only the weight matrix for h is reloaded. For the phone I use, it uses arm-v8 architecture, which supports Arm Neon intrinsic. For arm-v8 architecture, it supports vector with 128 bits. In our implementation, this feature is used together with GEMM to speed up multiplication. For GEMM, a 4x4 sub-matrix is updated. The code for GEMM with Neon is shown in Figure 4.

5 Evaluation

5.1 LSTM Optimization

We test one single layer LSTM with various input size and output size, with the baseline serial version implemented by NCNN, default GPU acceleration provided by NCNN and the one we hand-optimized by applying GEMM and Neon. The result is shown in Figure 5 and Figure 6.

From both figures, we can see that applying GEMM to the LSTM layer is beneficial, but only when the workload is fairly large. For example, in Figure 5, when input size is 128x128, the baseline inference time is 11.75ms, but the GEMM optimization is 17.32ms, which is 1.47x of the baseline implementation. But when the workload gets larger, when the input size increased to 128x1024, the baseline inference time is 46.54ms and the optimized oen is 37.78ms, with 1.23x speedup. The result is as expected, since the GEMM optimization is used for increase data reuse, when the workload is fairly small, the weight matrix can be completely cached. If the weight matrix could be completely cached, the penalty for poor data reuse is reduced.

For the default Vulkan optimization, the result is slightly slower than the baseline version in almost all cases. The reason why GPU acceleration cannot work effectively for the LSTM layer may be the sequential chain-like structure of LSTM. Each operation receives the output of its previous states as input and thus it must wait for its previous operations to finish. As a result, the parallel computing implemented on GPU cannot accelerate the execution of LSTM under the circumstance. On the other hand, GPU can do matrix computation much faster than CPU. As a result, inference on GPU takes almost the same amount of time as inference on CPU (baseline).

5.2 Effect of Multithread

We also analyzed the effect of number of threads. We exploit TLP in LSTM layer by observing that the calculation for k_{th} element in vector hidden state h is independent of each other. So we apply a simple division among the number of output and spread the work to different threads. The result is shown in Figure 7. We also measured the CPU usage, the result is shown in Figure 8.

For different workload, we always achieve best performance when the number of thread is 4. The speedup achieves 3.35x when the output size is 128x256. However, the CPU usage for 4 threads is only 50%. Since I am unsure how the thread is mapped to different cores. One possible explanation is that threads that working simultaneously share the same cache, which decreases the hit rate for each thread. This result shows the potential to further enhance performance by increasing CPU usage.

6 Conclusion

To reduce the inference time of RNN network on mobile devices, our work evaluate the LSTM layer with different input size and output size on mobile. To enhance the performance, we apply GEMM to the calculation of

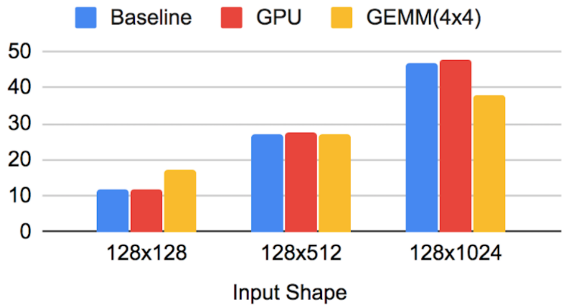

Figure 5: Inference time for a single LSTM layer on mobile with various input size but fixed output size 128x256(Unit: millisecond).

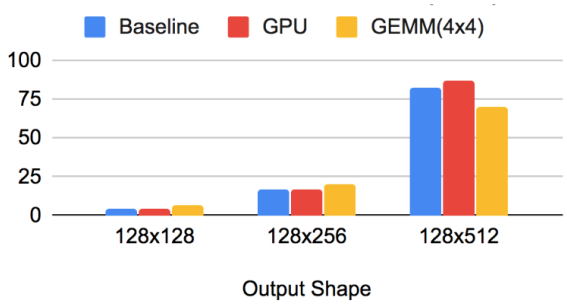

Figure 6: Inference time for a single LSTM layer on mobile with various output size but fixed input size 128x256(Unit: millisecond).

```
for (int r=0; r<T; r+=4) {
    float32_t* x_ptr = (float32_t*)bottom_blob.row(r);
    float32_t* g_ptr = (float32_t*)gates.row(r);
    for (int c=0; c<4*num_output; c+=4) {
        float32_t* weight_xc_ptr = (float32_t*)weight_xc.row(c);
        // fill 0 for C0, C1, C2, C3
        float32x4_t C0 = vmovq_n_f32(0.0);
        float32x4_t C1 = vmovq_n_f32(0.0);
        float32x4_t C2 = vmovq_n_f32(0.0);
        float32x4_t C3 = vmovq_n_f32(0.0);
        for (int k=0; k<size; k++) {
            float32x4_t W_X = {*(weight_xc_ptr+k), *(weight_xc_ptr+size*1+k),
                               *(weight_xc_ptr+size*2+k), *(weight_xc_ptr+size*3+k)};
            // C0 = C_0 + W_X * (*(x_ptr+k))
            C0 = vmlaq_n_f32(C0, W_X, *(x_ptr+k));
            C1 = vmlaq_n_f32(C1, W_X, *(x_ptr+size*1+k));
            C2 = vmlaq_n_f32(C2, W_X, *(x_ptr+size*2+k));
            C3 = vmlaq_n_f32(C3, W_X, *(x_ptr+size*3+k));
        }
        // (*(g_ptr + (4*num_output) * 0 + c) = C0)
        vst1q_f32(g_ptr + (4*num_output) * 0 + c, C0);
        vst1q_f32(g_ptr + (4*num_output) * 1 + c, C1);
        vst1q_f32(g_ptr + (4*num_output) * 2 + c, C2);
        vst1q_f32(g_ptr + (4*num_output) * 3 + c, C3);
    }
}
```

Figure 4: GEMM implementation with Arm Neon.

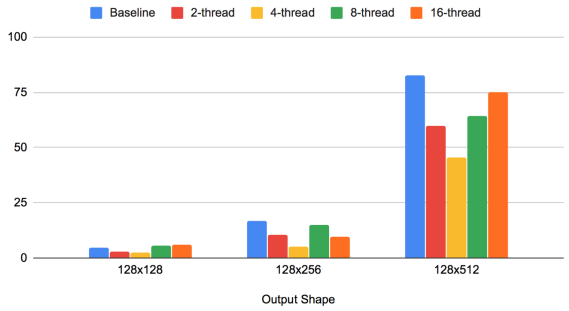

Figure 7: Inference time for a single LSTM layer on mobile with various output size but fixed input size 128x256(Unit: millisecond).

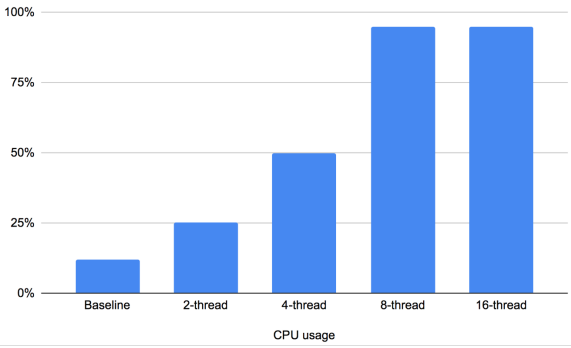

Figure 8: CPU usage for different number of threads.

gate vector from input matrix and corresponding weight matrix. With the GEMM optimization, we achieved 1.23x speedup for large workload. Since GEMM enhances the data reuse of matrix multiplication, one potential future direction for further enhancing the performance is by avoiding same data from memory for multiple times.

We also analyzed the influence of multi-threading on LSTM layer. The result shows that the best performance is achieved when the number of thread is 4. However, for 4 threads, the CPU usage is only 50%. Therefore, there is potential for further enhancing the performance.

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[2] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. V. Gool, "AI benchmark: Running deep neural networks on android smartphones," *CoRR*, vol. abs/1810.01109, 2018.

[3] D. Frajberg, C. Bernaschina, C. Marone, and P. Fraternali, *Accelerating Deep Learning Inference on Mobile Systems*, pp. 118–134. 06 2019.

[4] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann, "On-device neural net inference with mobile gpus," *CoRR*, vol. abs/1907.01989, 2019.

[5] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 1–12, IEEE, 2016.

[6] T. Guo, "Cloud-based or on-device: An empirical study of mobile deep inference," *CoRR*, vol. abs/1707.04610, 2017.

[7] W. Kang and J. Chung, "Power- and time-aware deep learning inference for mobile embedded devices," *IEEE Access*, vol. 7, pp. 3778–3789, 2019.

[8] Tencent, "Tencent ncnn." `https://github.com/Tencent/ncnn`, 2020.

[9] Huawei Technologies Co., Ltd., *HiAI Engine.*, 2020. Available at `https://developer.huawei.com/consumer/en/devservice/doc/2020315`, Online; accessed Feb 14, 2020.

[10] Arm Ltd., *Compute Library.*, 2020. Available at `https://developer.arm.com/ip-products/processors/machine-learning/compute-library`, Online; accessed Feb 14, 2020.

[11] Google Inc., *Android Studio.*, 2020. Available at https://developer.android.com/studio, Online; accessed Feb 14, 2020.

[12] *MLPerf Inference.*, 2020. Available at http://www.mlperf.org/, Online; accessed Feb 14, 2020.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[14] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[15] "Chineseocr." https://github.com/ouyanghuiyu/chineseocr_lite, 2020.