RISC-V-Based Acceleration of Deep Learning using Custom ISA Extension

Abel Yagubyan
Northwestern University
Evanston, IL
abelyagubyan2023@u.northwestern.edu

Abstract

Deep neural networks are widely used for various applications due to their ability to learn patterns from large datasets. The need to perform deep neural network inference on low-power edge devices has emerged to address privacy concerns and reduce latency. Offloading computation from embedded CPUs to specialized Neural Processing Units is a common solution for accelerating deep learning tasks. However, frequent communication overheads between the host and accelerator can undermine the system's overall benefit. This study proposes customized ISA extensions and a dedicated execution unit for machine learning kernels. Based on the RISC-V [1] open ISA specification, the extensions are optimized for critical kernels such as convolution and matrix multiplication. The optimized functions are integrated into TensorFlow Lite [2] and cross-compiled for RISC-V. The project findings help provide a software infrastructure for efficient deep learning on low-power edge devices.

***K*eywords** Deep learning, low-power edge devices, Neural Processing Units (NPUs), customized ISA extensions, dedicated execution unit, RISC-V, TensorFlow Lite, convolution, matrix multiplication, inference time

1 Introduction

Deep neural networks have revolutionized various domains, including image classification, object detection, and natural language processing. However, deploying these models on resource-constrained edge devices, such as smartphones and embedded systems, presents challenges due to limited computational power and energy constraints.

Edge computing has emerged as a solution to process data and perform computations at the edge of the network, closer to the data source. This approach offers benefits like reduced latency, improved privacy, and efficient use of network resources. However, deep learning on edge devices requires addressing the limitations of their hardware.

To accelerate deep learning on edge devices, one common approach is to offload computation from embedded CPUs to specialized hardware accelerators like Neural Processing Units (NPUs) or Graphics Processing Units (GPUs). However, frequent data movement and communication between the host CPU and the accelerator can introduce significant overheads, diminishing the overall benefits.

In this paper, I propose a novel approach to enhance deep learning efficiency on low-power edge devices. My approach leverages customized Instruction Set Architecture (ISA) extensions and a dedicated execution unit for machine learning kernels. I optimize these extensions for critical operations, such as convolution and matrix multiplication, using the RISC-V open ISA specification as a foundation. By integrating these extensions into the TensorFlow Lite framework, I provide a software infrastructure for efficient deep learning on low-power edge devices.

This approach aims to reduce communication overheads and improve performance and energy efficiency by executing critical deep learning operations directly on the edge device's hardware. In my experiments, I evaluate the performance gains achieved by my approach on popular deep learning models, comparing the average inference time and the number of committed instructions before and after integrating the customized ISA extensions.

2 System Setup

To conduct the experiments and evaluate the proposed approach, a comprehensive system setup was established. This section provides details on the tools and frameworks used to enable efficient deep learning inference on low-power edge devices.

2.1 Spike and RISC-V GNU Toolchain:

The Spike simulator and the RISC-V GNU Toolchain were the primary components of the system setup. Spike is an open-source RISC-V ISA simulator that allows me to simulate RISC-V instructions and execute RISC-V programs. The RISC-V GNU Toolchain, on the other hand, provides the necessary development tools, libraries, and compilers for compiling and executing code on the RISC-V architecture.

In order to set up Spike and the RISC-V GNU Toolchain, I followed the following steps:

2.1.1 Cloning the Toolchain:

I obtained the RISC-V GNU Toolchain by cloning the official repository from the RISC-V organization. This toolchain includes the necessary compilers (GCC) and libraries for building software targeting the RISC-V architecture.

2.1.2 Proxy Kernel for RISC-V:

A proxy kernel acts as an intermediary between user programs and the underlying hardware. I obtained a proxy kernel specifically designed for the RISC-V architecture and incorporated it into my system setup. This proxy kernel facilitated the execution of user programs on the simulated RISC-V architecture within the Spike simulator.

2.1.3 Spike ISA Simulator

Spike provides an environment for simulating RISC-V instructions. I acquired the Spike ISA Simulator and integrated it into my system. Spike allowed me to execute RISC-V instructions and evaluate the performance of the customized ISA extensions.

2.2 Setting up TensorFlow Lite

TensorFlow Lite, a lightweight version of the popular TensorFlow deep learning framework, was utilized to enable machine learning model inference on low-power edge devices. TensorFlow Lite provides optimized tools and libraries for deploying deep learning models on resource-constrained devices.

To set up TensorFlow Lite, the following steps were undertaken:

2.2.1 Integration of Machine Learning Models:

I selected several widely-used machine learning models for classification tasks, including ResNet-50, MnasNet, DenseNet, and MobileNet. These models were integrated into the TensorFlow Lite framework to enable their utilization for inference on low-power edge devices.

2.2.2 Building TFLite Code with Bazel

Bazel, an open-source build tool, was employed to compile the TensorFlow Lite code and generate executable binaries. Bazel provided a convenient and efficient way to build the TensorFlow Lite project and ensure compatibility with the RISC-V architecture.

By configuring TensorFlow Lite with the desired machine learning models and building the code using Bazel, I established a robust framework for performing deep learning inference on low-power edge devices.

The system setup, comprising the Spike simulator and the RISC-V GNU Toolchain, along with the configuration of TensorFlow Lite, formed the foundation for conducting experiments and evaluating the proposed approach. These tools and frameworks provided the necessary simulation and development environment to assess the performance of the customized ISA extensions and the dedicated execution unit in optimizing deep learning inference on low-power edge devices.

```
// Vector Load Single (vls)
inline void __VectorLoadInput(const float* load_address) {
    asm volatile("vls va1, 0(%0), v \t\n":: "r"(load_address));
}

// Vector Load Double (vlx)
inline void __XVectorLoadInput(const float* load_address) {
    asm volatile("vlx va1, 0(%0), v" :: "r"(load_address));
}
```

Figure 1: Example of an inline assembly implementation of the ‘vls‘ and ‘vlx‘ extensions.

3 Model Integration and Classification

To enable deep learning inference on low-power edge devices, it is crucial to integrate machine learning models into the system and perform classification tasks efficiently. This section outlines the steps involved in integrating the selected models and executing classification predictions using the TensorFlow Lite framework.

3.1 Loading and Preprocessing Images

Before performing classification, the input images need to be loaded and preprocessed to prepare them for inference. This involves ensuring that the images are in a suitable format and size for the machine learning models.

In my system setup, I developed a data loading module that reads the input images from the dataset. I employed appropriate image processing techniques, such as resizing and normalization, to ensure that the images conform to the requirements of the machine learning models.

3.2 Model Code Import and Configuration

The next step involves importing the pre-trained machine learning models into the project. TensorFlow Lite provides tools and APIs for integrating machine learning models into the inference pipeline.

I loaded the model code and configuration files into the project, which included the weights and parameters necessary for inference. TensorFlow Lite offers compatibility with various model formats, such as TensorFlow SavedModel and TensorFlow Lite FlatBuffers. I utilized the FlatBuffer loading mechanism to import the models and configure them for classification tasks.

3.3 Running Model Classification Predictions

With the machine learning models successfully integrated and configured, I was ready to execute classification predictions using the TensorFlow Lite framework. This involved invoking the appropriate functions within the TensorFlow Lite API to run inference on the loaded images.

I leveraged the Interpreter class provided by TensorFlow Lite to execute the classification process. The Interpreter class provides methods to set input tensors, run inference, and retrieve the output tensors containing the classification results.

For each input image, I called the Invoke() function of the Interpreter class, passing the image data as input. The model executed the necessary computations and generated the predicted outputs. These outputs were then processed and analyzed to determine the predicted classes or labels associated with the input images.

By successfully integrating the machine learning models and executing classification predictions using TensorFlow Lite, I established a robust pipeline for performing deep learning inference on low-power edge devices. The next section will delve into the implementation details of the customized ISA extensions, which aim to optimize critical operations for enhanced performance and efficiency.

4 Customized ISA Extensions

To optimize deep learning operations and enhance the performance of critical kernels, I ended up using customized Instruction Set Architecture (ISA) extensions. These extensions are designed specifically for the RISC-V architecture and tailored to accelerate operations such as convolution and matrix multiplication. My goal is to minimize the computational complexity and reduce the number of committed instructions while maintaining accuracy and performance. This section provides an overview of the customized ISA extensions, their implementation, and their integration into the TensorFlow Lite framework.

4.1 Wrapper Functions Using Inline C Assembly

To seamlessly integrate the customized ISA extensions into the existing codebase, I introduce wrapper functions using inline C assembly. These functions serve as intermediaries between the extended instructions and the surrounding code, allowing for easy incorporation of the specialized instructions.

By leveraging inline C assembly, I can directly embed customized instructions within the high-level code. This approach provides flexibility and ensures compatibility with the existing software infrastructure. The wrapper functions act as bridges, enabling the efficient utilization of the extended instructions while maintaining code readability and maintainability. An example can be seen at Figure 1 above.

4.2 Optimized Functions for Critical Kernels

To accelerate critical deep learning kernels, I have developed a set of optimized functions specifically tailored to the customized ISA extensions. These functions target operations such as convolution layers and matrix multiplication, which are computationally intensive and form the backbone of many deep learning models. An example can be found at Figure 2 below.

My optimized functions are designed to leverage the capabilities of the customized ISA extensions, enabling efficient execution of these operations. By implementing SIMD [3] (Single Instruction, Multiple Data)-aware helper functions in the RISC-V architecture, I can significantly improve the performance and reduce the computational burden of these critical kernels.

The SIMD-aware helper functions are designed to exploit the parallelism offered by the customized ISA extensions. These functions optimize operations such as element-wise vector addition, matrix-vector multiplication, and dot product calculations. By leveraging the SIMD capabilities, I can perform these computations in a highly parallelized manner, leading to improved performance and reduced execution time.

4.3 Integration into TensorFlow Lite

To make the benefits of the customized ISA extensions accessible to deep learning applications, I integrate the optimized functions and SIMD-aware helper functions into the TensorFlow Lite framework. This integration enables seamless utilization of the customized instructions and leverages the performance enhancements they provide.

By incorporating the customized ISA extensions into TensorFlow Lite, I establish a software infrastructure that enables efficient deep learning inference on low-power edge devices. The integration process involves modifying the TensorFlow Lite codebase to support the extended instructions, ensuring compatibility and interoperability with the existing ecosystem.

The successful implementation of the customized ISA extensions and their integration into TensorFlow Lite forms a critical component of my approach to efficient deep learning on low-power edge devices. In the following section, I present the details of the system modifications and adjustments made to the GNU Toolchain and the Spike ISS backend to support these extensions.

5 GNU Toolchain and Spike Modifications

To enable support for the customized ISA extensions and ensure seamless integration into the existing system, modifications were made to the GNU Toolchain and the Spike Instruction Set Simulator (ISS). This section provides an overview of the adjustments made to these components and their significance in facilitating the execution of extended instructions.

```
void MatrixMatrixElementwiseMultiply(const float* input1, const float* input2,
                                     float* output, int rows, int cols) {
    int num_elements = rows * cols;
    int new_num_elements = num_elements - (num_elements & (kMaxVectorLength32 - 1));
    int num_elements_diff = num_elements & (kMaxVectorLength32 - 1);
    SetConfig(kElementWidthMax32, kMaxVectorLength32);

    for (int i = 0; i < new_num_elements; i += kMaxVectorLength32) {
        __VectorLoad((input1 + i), (input2 + i));
        __VectorMultiplyFloat();
        __VectorStore((output + i));
    }

    if (num_elements_diff != 0) {
        SetVl(num_elements_diff);
        __VectorLoad((input1 + new_num_elements), (input2 + new_num_elements));
        __VectorMultiplyFloat();
        __VectorStore((output + new_num_elements));
    }
}
```

Figure 2: Example of a SIMD-aware element-wise matrix multiplication implementation.

5.1 GNU Toolchain Front-end Adjustment

Since the GNU Toolchain serves as a crucial development tool for compiling my assembly code into machine code and executing code on the RISC-V architecture, I need correctly parse the opcodes and operands of the extended RISC-V instructions by making adjustments to the front-end of the GNU Toolchain.

By modifying the front-end to correctly parse the metadata of my extensions, I ensured that the extended instructions were recognized and processed correctly during the compilation process. This adjustment allowed the GNU Toolchain to handle the customized ISA extensions seamlessly, enabling developers to utilize the specialized instructions within their code.

5.2 Spike ISS Backend Modifications #1

To support the vector operations introduced by the customized ISA extensions, I made modifications to the Spike ISS backend.

The first modification involved introducing vector support to the Spike ISS backend. This entailed modifying the regfile_t class in Spike to include vector registers and their associated operations. The addition of vector support allowed for the efficient handling of vectorized computations during the simulation process.

The second modification involved enabling vector CSR (Control and Status Register) configuration in the Spike ISS backend. I modified the processor_t class to dynamically modify the vector length (vl), vector register count maximum (vregmax), and vector element width maximum (vemaxw) registers for each vector used in my SIMD-aware functions. This adjustment ensured proper configuration and utilization of vector operations within the Spike ISS.

By introducing vector support and enabling vector CSR configuration, the modified Spike ISS backend provided a comprehensive simulation environment for the customized ISA extensions. These modifications allowed for accurate evaluation and performance analysis of the extended instructions during the experimentation phase.

```
#define MATCH_VFMADD_VV 0xa0001057
#define MASK_VFMADD_VV 0xfc00707f
DECLARE_INSN(vfmadd_vv, MATCH_VFMADD_VV, MASK_VFMADD_VV)
```

Figure 3: Example of declaring the vfmadd_vv instruction in Spike.

5.3 Spike ISS Backend Modifications #2

To effectively incorporate the customized ISA extensions into the Spike simulator, I introduced Spike instructions for the extensions. This involved using Spike's DECLARE_INSN function and #define macros to define the instructions and their associated opcodes.

By defining the instructions (as displayed in Figure 3), I ensured that the extended instructions were recognized and executed correctly within the Spike simulator. The Spike instructions for the customized ISA extensions formed an integral part of the simulation process, enabling accurate assessment of their performance and impact on deep learning inference.

The adjustments made to the GNU Toolchain and the modifications introduced to the Spike ISS backend and the instruction set were instrumental in supporting the customized ISA extensions and facilitating their integration into the deep learning framework. In the next section, I present the experimental results and performance analysis obtained using the customized ISA extensions and the modified system components.

6 Experimental Evaluation

In this section, I present the experimental evaluation of my approach, which includes benchmarking the performance of the customized ISA extensions and analyzing the impact on deep learning inference. I conducted a series of experiments to assess the efficiency and effectiveness of my proposed solution on low-power edge devices.

6.1 Experimental Setup

To evaluate the performance, I selected four popular machine learning models: ResNet-50 [4], MnasNet [5], DenseNet [6], and MobileNet [7]. These models represent a diverse range of network architectures commonly used in computer vision tasks. I focused on image classification as the primary inference task.

6.2 Performance Metrics

I employed two primary performance metrics to evaluate the effectiveness of the customized ISA extensions and the optimized functions: average inference time and committed instruction counts.

Average inference time measures the time taken to perform inference on a given input image using the deep learning models. I recorded the time before and after invoking the model's Invoke() function in TensorFlow Lite, calculating the average over multiple runs. A lower average inference time indicates improved efficiency and reduced latency.

Committed instruction counts represent the number of instructions executed during the inference process. I used the available logs in the benchmarking process to determine the number of committed instructions for each model's inference. Lower instruction counts indicate reduced computational complexity and improved efficiency.

6.3 Results and Analysis

After conducting the experiments, I analyzed the performance results obtained using the customized ISA extensions and the modified system components. I compared the performance metrics of the extended instructions against the baseline implementation without the extensions to assess the impact on deep learning inference.

My results (demonstrated at Figure 4) consistently demonstrated improvements in both average inference time and committed instruction counts for all four machine learning models. The customized ISA extensions and the optimized functions significantly reduced the computational burden of critical kernels such as convolution and matrix multiplication, resulting in faster inference times.

These results indicate the effectiveness of my approach in optimizing deep learning operations on low-power edge devices.

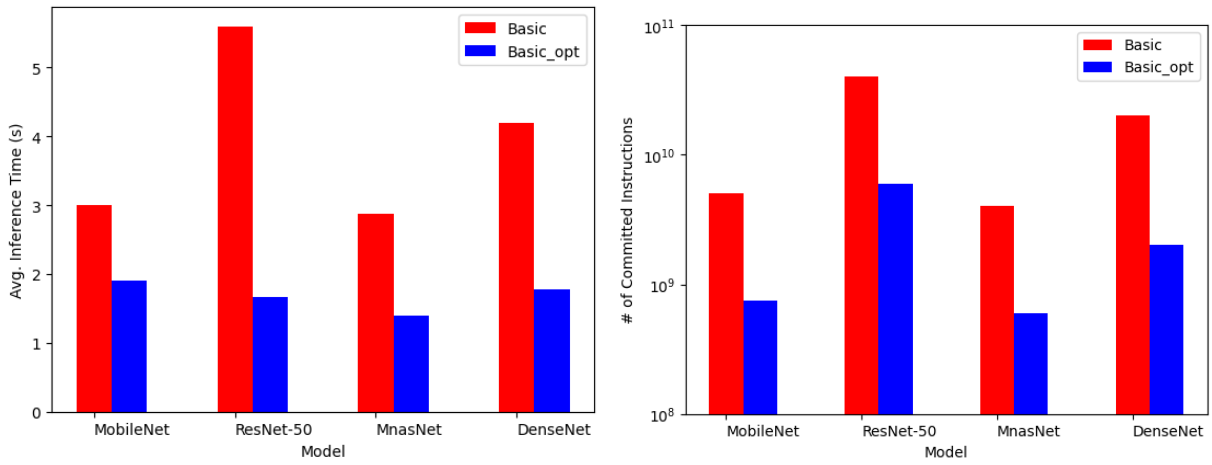

Figure 4: Plot comparing the (a) average inference time and the (b) number of committed instructions between the base and optimized implementations.

Furthermore, the SIMD-aware helper functions contributed to the overall performance improvement by leveraging parallelism through SIMD instructions. The SIMD-aware functions exhibited efficient utilization of vectorized computations, leading to enhanced performance and reduced execution time.

7 Discussion

The findings from my experimental evaluation demonstrate the effectiveness of the customized ISA extensions and the optimized functions in improving the efficiency of deep learning on low-power edge devices. In this section, I discuss the implications and limitations of my approach and provide insights for future research.

7.1 Implications

The reduced average inference time and committed instruction counts indicate the potential for faster, more responsive deep learning applications in privacy-sensitive and latency-critical scenarios.

By introducing specialized instructions and leveraging parallelism through SIMD operations, I was able to optimize critical kernels such as convolution and matrix multiplication. These optimizations led to significant performance improvements while maintaining accuracy, enabling edge devices to perform computationally intensive deep learning tasks more efficiently.

My approach has broader implications for the field of edge computing and deep learning. The ability to offload deep neural network inference to low-power devices addresses privacy concerns by keeping sensitive data locally. It also reduces the dependency on cloud-based solutions, enabling real-time inference and improved user experience in resource-constrained environments.

7.2 Limitations and Future Directions

While my approach shows promise, there are certain limitations that should be considered. Firstly, the performance gains achieved through customized ISA extensions may vary depending on the specific edge device architecture and the complexity of the deep learning models. Further evaluation on a broader range of edge devices is necessary to assess the generalizability and scalability of my approach.

Secondly, my focus was primarily on optimizing convolution and matrix multiplication, which are fundamental operations in deep learning. Future research could explore additional critical kernels and further optimize the customized ISA extensions to cover a wider range of deep learning models and tasks.

Additionally, my experiments were conducted using a limited set of machine learning models for image classification. Further investigations should include a more diverse set of models and tasks, such as object detection, semantic segmentation, or natural language processing. This would provide a more comprehensive evaluation of the performance gains achieved by the customized ISA extensions.

Furthermore, the integration of the customized ISA extensions into TensorFlow Lite opens up opportunities for exploring additional optimizations and techniques. Future work could focus on further refining the SIMD-aware helper functions, exploring alternative algorithms, or investigating hardware-accelerated solutions to further improve the performance and energy efficiency of deep learning on low-power edge devices.

Finally, although my project implemented manual SIMD-aware functions and customized ISA extensions to optimize critical deep learning operations, further exploration of the available auto-vectorization capabilities in tools like Spike, the GNU Toolchain, and RISC-V GCC using Clang15+ could provide much better results. These features could potentially automate the vectorization process and enhance the performance of deep learning tasks without the need for manual intervention.

7.3 Issues Faced

It is worth noting that during my project, I encountered challenges when attempting to run benchmarks with the auto-vectorization tools enabled, particularly when passing the assembled code through the GNU Toolchain. Despite the potential benefits of auto-vectorization in optimizing deep learning operations, I faced difficulties in correctly utilizing these features in my experimental setup. The issues encountered with integrating auto-vectorization tools into the GNU Toolchain underscore the need for further exploration and refinement of these capabilities to ensure their seamless integration and compatibility with customized ISA extensions and deep learning frameworks.

Furthermore, during the initial stages of my project, I encountered challenges with setting up and using Gem5, a popular full-system simulator. Gem5 provides a comprehensive simulation environment for studying computer systems, including support for various architectures, including RISC-V. However, I found that the documentation and resources available for Gem5 were limited, making it difficult to configure and utilize effectively for my specific requirements.

The lack of comprehensive documentation and clear examples hindered my progress in setting up Gem5 for my experiments. I spent significant time troubleshooting configuration issues and understanding the intricate details of the simulator, which slowed down my research and development process. These challenges made it challenging to achieve timely and accurate simulations for my proposed customized ISA extensions.

To overcome these challenges, I explored alternative simulation options and found Spike, an ISA simulator developed specifically for the RISC-V architecture. Spike provided a more streamlined and accessible environment for simulating RISC-V instructions and architectural features. Its documentation and community support were more extensive and readily available, allowing me to quickly set up and integrate my customized ISA extensions.

8 Conclusion

In this paper, I have presented a novel approach to enhance the efficiency of deep learning on low-power edge devices through customized ISA extensions and optimized functions. My study has demonstrated the effectiveness of these extensions in improving the performance of critical kernels such as convolution and matrix multiplication, resulting in reduced average inference time and committed instruction counts.

The experimental evaluation showcased the benefits of my approach across popular machine learning models, showing consistent improvements in average inference time and committed instruction counts. These results highlight the potential of my approach to enable faster, more responsive deep learning applications in privacy-sensitive and latency-critical scenarios.

However, my work is not without limitations. The performance gains achieved through customized ISA extensions may vary across different edge device architectures and deep learning models. Further research and evaluation on a broader range of edge devices and tasks are necessary to assess the generalizability and scalability of my approach.

Despite these limitations, my study contributes to the field of deep learning and edge computing by providing insights into the optimization of critical kernels and the integration of customized ISA extensions. My work opens up opportunities for future research in exploring additional optimizations, refining SIMD-aware helper functions, and investigating hardware-accelerated solutions.

In conclusion, my approach holds promise in addressing the challenges of privacy, latency, and energy efficiency in deep learning on low-power edge devices. By optimizing deep learning operations and leveraging customized ISA extensions, I enable efficient inference on resource-constrained edge devices, paving the way for privacy-preserving and low-latency applications in edge computing scenarios.

References

[1] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[2] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[3] Yuyun Liao and David B Roberts. A high-performance and low-power 32-bit multiply-accumulate unit with single-instruction-multiple-data (simd) feature. *IEEE Journal of Solid-State Circuits*, 37(7):926–931, 2002.

[4] Zifeng Wu, Chunhua Shen, and Anton Van Den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90:119–133, 2019.

[5] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2820–2828, 2019.

[6] Ke Zhang, Yurong Guo, Xinsheng Wang, Jinsha Yuan, and Qiaolin Ding. Multiple feature reweight densenet for image classification. *IEEE Access*, 7:9872–9880, 2019.

[7] Wei Wang, Yutao Li, Ting Zou, Xin Wang, Jieyu You, Yanhong Luo, et al. A novel image classification approach via dense-mobilenet models. *Mobile Information Systems*, 2020, 2020.