Dean&Francis
ISSN 2959-6157

Evaluating the Impact of Server-Side Batching on Inference Performance of CNN Architectures: A Comparative Study Using TensorFlow Serving

Jinghao Zhang

Xavier High School, Connecticut, United States of America
jzhang@xavierhighschool.org

Abstract:

This study investigates the deployment and performance evaluation of Convolutional Neural Network (CNN) models for image classification using TensorFlow Serving. Four pretrained models—including ResNet-50, InceptionV3, and MobileNetV2—as well as two custom CNN models were implemented and served in Docker containers. Each model was tested under two batching configurations: no batching and batching with a size of two. Server-side batching was managed through a custom configuration file, and inference performance was measured using a concurrent client setup with 10 threads. A primary focus of this project was on the ResNet-50 model, which was initialized with ImageNet weights and fine-tuned on the tf_flowers dataset. The training process followed a two-stage approach: initial training of the classification head with frozen base layers, followed by fine-tuning of deeper layers to enhance generalization. The model was exported in SavedModel format and deployed for testing. Experimental results show a clear trade-off between latency and throughput across different models and batching strategies. Batching improved throughput in most cases but occasionally increased per-request latency. This study highlights the importance of choosing suitable batching strategies based on specific application requirements, offering insights into optimizing CNN-based image classifiers for real-world deployment scenarios.

Keywords: Convolutional neural network; TensorFlow serving; flower classification.

ISSN 2959-6157

1. Introduction

Accurate flower classification is essential in agriculture, environmental monitoring, and biodiversity conservation. Traditional manual classification methods are labor-intensive and error-prone. Recent advances in deep learning, especially convolutional neural networks (CNNs), now provide effective automated approaches, greatly improving classification accuracy and efficiency.

In recent years, CNN have demonstrated remarkable capabilities in image recognition tasks, significantly pushing the boundaries of accuracy and performance across diverse application scenarios. The development of various advanced CNN architectures, including ResNet, Inception, and MobileNet [1-3], has offered researchers and industry practitioners a wide spectrum of model choices tailored to specific requirements of accuracy, computational cost, and inference speed. However, deploying these advanced models in real-world environments, particularly in resource-constrained or latency-sensitive contexts, remains a non-trivial challenge. As deep learning models become increasingly complex, their inference efficiency often becomes a critical bottleneck that influences overall application performance and usability.

TensorFlow Serving has emerged as a leading platform for deploying trained deep learning models in production environments, primarily due to its flexibility, scalability, and native support for batching inference requests [4]. Utilizing batching which is —the practice of grouping multiple inference requests into a single batch — can significantly improve computational efficiency, thus enabling better hardware utilization and higher throughput. However, batching may also introduce trade-offs, such as increased inference latency per request, especially under certain workload conditions. Hence, a systematic evaluation of batching strategies and their impact on CNN performance metrics is essential for optimizing deployment decisions.

This study focuses on evaluating and comparing the inference performance, particularly throughput and latency — of several widely adopted CNN architectures under different batching scenarios, using TensorFlow Serving as the deployment platform. Specifically, this study deployed four pretrained CNN models and two custom-trained CNN models in a Docker-based TensorFlow Serving environment, configuring batching through a custom parameter file. This study then rigorously tested each model's inference performance under two distinct configurations: no batching and batching with a small batch size of two. This approach allowed us to investigate the trade-offs between batching size, inference latency, and throughput in practical scenarios involving concurrent client requests.

A central part of the analysis is the ResNet-50 model, a deep CNN architecture known for its residual connections and robust feature-extraction capabilities. Initialized with ImageNet-pretrained weights, this paper further fine-tuned ResNet-50 specifically for image classification tasks using the well-known tf_flowers dataset. The fine-tuning strategy involved a two-phase approach: initially training only the classification head with frozen convolutional layers, followed by selectively fine-tuning deeper convolutional blocks at a reduced learning rate to optimize validation accuracy. This procedure ensured that the model effectively leveraged pretrained knowledge while adapting accurately to the domain-specific task.

After training, the ResNet-50 model was exported into TensorFlow's SavedModel format and subsequently deployed for inference performance tests in TensorFlow Serving using Docker. The tests involved sending a substantial number of inference requests concurrently (1,000 requests via 10 client threads), allowing precise measurement and analysis of latency and throughput under both batching configurations. Comparative performance analyses were then conducted against other widely-used CNN architectures, including InceptionV3, MobileNetV2, and custom CNN models, to contextualize the findings and provide practical guidance on model deployment decisions.

2. Method

2.1 Dataset Preparation

This study used the publicly available tf_flowers dataset provided by TensorFlow Datasets [5]. This dataset comprises a total of 3,670 labeled images, classified into five distinct categories: daisy, dandelion, roses, sunflowers, and tulips. Each category consists of diverse color images, all presented in RGB format, which is essential for capturing the distinctive visual features and color variations unique to each flower type. Originally, these images vary in dimensions and resolutions, thus this study uniformly resized each image to a resolution of 224 × 224 pixels to ensure consistency and compatibility with the input size requirements of the CNN models.

The preprocessing pipeline included several essential steps to enhance model training efficiency and generalization capabilities. Firstly, this study applied normalization by scaling pixel values from the original [0, 255] range down to the [0, 1] range. This normalization technique is crucial for achieving stable convergence during model training and reducing numerical instability.

To mitigate overfitting and enhance model robustness, this paper integrated data augmentation techniques into the preprocessing workflow. Specifically, this study per-

formed random horizontal flips of images during training. This approach artificially expands the effective size and diversity of the dataset, allowing the models to learn more generalized and robust visual features.

This study divided the original dataset into two subsets: a training set and a test set. Approximately 80% of the images (2,936 images) were randomly allocated to the training set, while the remaining 20% (734 images) were used for testing and evaluating model performance. This clear separation allowed us to fairly evaluate the generalization ability of each CNN architecture.

2.2 CNN-based Classification Models

2.2.1 MobileNetV2

MobileNetV2, introduced by Sandler et al., is a CNN architecture explicitly designed for efficiency and optimized performance on computationally constrained devices. Its architecture primarily revolves around the concept of inverted residual blocks. An inverted residual block contains three main operations: (1) pointwise convolution (1×1 convolution) for dimensionality expansion, (2) depthwise convolution, which applies separate convolutional filters to each channel independently, significantly reducing computational complexity, and (3) another pointwise convolution for dimensionality reduction, restoring efficient representations.

The key innovation in MobileNetV2 is the introduction of linear bottleneck layers, designed to prevent information loss when compressed features are propagated through narrow network channels. The full architecture consists of an initial convolutional layer, followed by multiple inverted residual blocks, concluding with a global average pooling layer and a fully-connected layer for classification. This architecture maintains high accuracy while significantly reducing the number of parameters and computational overhead.

2.2.2 InceptionV3

The InceptionV3 model, proposed by Szegedy et al., is an advanced CNN architecture characterized by its unique inception modules designed for efficient multi-scale feature extraction. Each inception module simultaneously applies convolutional filters of varying sizes (1×1, 3×3, 5×5) to the input data, allowing the model to capture and integrate rich features at different scales within a single computational unit.

To further optimize computational efficiency, InceptionV3 employs factorized convolutions, splitting larger convolutions into smaller, sequential convolutions, reducing the overall parameter count and computational complexity. Additionally, extensive use of 1×1 convolutions serves as dimensionality reduction and feature mapping mechanisms, helping maintain computational efficiency without sacrificing representational power. The network's final stages employ global average pooling and fully connected layers for robust classification outcomes.

2.2.3 Custom CNN model

In addition to the standard architecture described above, this study developed a custom CNN model to serve as a simpler baseline [6-8]. The proposed CNN architecture consists of four convolutional layers followed by max-pooling operations and two fully connected layers for classification.

Specifically, the first convolutional layer uses 32 filters with a kernel size of 3×3, followed by a ReLU activation and max pooling. The second convolutional layer contains 64 filters, again with a kernel size of 3×3, accompanied by ReLU activation and max pooling. The third and fourth convolutional layers have 128 filters each, both employing 3×3 kernels and ReLU activations. These convolutional blocks effectively extract hierarchical features from input images.

After feature extraction, the outputs are flattened and fed into two dense layers. The first dense layer includes 128 neurons with ReLU activation, providing a fully connected representation of the learned features. Finally, the last dense layer has five output neurons, corresponding to the number of flower categories, combined with a softmax activation to produce probability distributions for classification.

2.3 Implementation Details

All models were implemented and trained using the TensorFlow deep learning framework, version 2.x. To ensure consistent and comparable results across different architectures, this study adopted a unified training strategy and hyperparameter settings.

This study used the Adam optimizer for training due to its proven effectiveness in CNN optimization tasks. For the initial phase of training (classification head training with frozen convolutional layers), this study sets the learning rate at 0.001. During fine-tuning of deeper layers in models like ResNet-50 and InceptionV3, this paper reduced the learning rate to 0.00001 to avoid disrupting pretrained features and to promote stable and gradual optimization.

This study selected sparse categorical cross-entropy as the loss function [9, 10], appropriate for multiclass classification problems, enabling models to optimize their predictions effectively. Model performance was evaluated based on standard classification metrics, primarily accuracy, computed on the separate test dataset described earlier.

All models were trained using a batch size of 32 images

ISSN 2959-6157

per batch. To adequately explore convergence behavior and model generalization, each model underwent two training phases: an initial training phase with 5 epochs and a subsequent fine-tuning phase consisting of another 5 epochs. After training, the models were exported into TensorFlow's SavedModel format and deployed for inference tests using TensorFlow Serving, specifically configured within Docker containers. Inference performance evaluation involved measuring key metrics such as latency (average and 95th percentile) and throughput, using concurrent client requests (10 threads), under both no batching and batching (batch size = 2) conditions.

3. Results and Discussion

3.1 The Performance of Models

This systematically evaluated the inference performance of four pretrained models (ResNet-50, MobileNetV2, InceptionV3, and one additional pretrained CNN) and two self-trained CNN models using TensorFlow Serving with Docker deployment. Performance metrics were measured under two batching configurations: no batching (batch size = 1) and batching with a batch size of 2. Performance evaluation focused primarily on two key metrics: latency (average and 95th percentile) and throughput, under a 10-thread concurrent client setup.

Table 1 summarizes the detailed inference latency results obtained across the tested models under both batching conditions.

Table 1. Average latency and 95th percentile latency (ms) under different batching conditions.

Model	Batch Size	Average Latency (ms)	95th Percentile Latency (ms)
ResNet-50	1	6 200.00	9 700.00
ResNet-50	2	380.00	460.00
InceptionV3	1	8 606.48	13 697.54
InceptionV3	2	511.83	633.73
MobileNetV2	1	4 120.71	7 336.55
MobileNetV2	2	230.47	262.10
Self-trained CNN	1	2 663.35	4 499.24
Self-trained CNN	2	176.01	195.73

Similarly, inference throughput results are presented in Table 2.

Table 2. Inference throughput (images per second, ips) under different batching conditions

Model	Batch Size	Average Throughput (ips)	95th Percentile Throughput (ips)
ResNet-50	1	2.60	0.52
ResNet-50	2	3.90	5.20
InceptionV3	1	1.76	0.23
InceptionV3	2	1.95	2.48
MobileNetV2	1	3.80	0.53
MobileNetV2	2	4.33	5.15
Self-trained CNN	1	5.78	0.83
Self-trained CNN	2	5.67	6.41

From Tables 1 and 2, clear differences in inference performance between models and batching configurations can be observed. Generally, heavier CNN architectures like InceptionV3 exhibited higher latency and lower throughput under no batching conditions. The introduction of batching (batch size = 2) dramatically reduced latency and increased throughput for all tested models, with particularly substantial improvements seen in heavier architectures like ResNet-50 and InceptionV3.

ResNet-50, the primary model in this study, exhibited an average latency of 6,200 ms under no batching conditions, while batching reduced this latency significantly to 380

ms. Correspondingly, its throughput improved from 2.6 images per second (no batching) to 3.9 images per second with batching enabled. Similar performance patterns were observed for InceptionV3 and MobileNetV2, underscoring the general effectiveness of batching as a strategy for improving inference performance.

Interestingly, the self-trained CNN model, despite being simpler in architecture, already demonstrated relatively low latency (2663 ms with no batching), and batching further improved its performance significantly.

3.2 Discussion

The results indicate a clear trade-off between model complexity and inference performance. Lighter models such as MobileNetV2 and the custom self-trained CNN consistently outperformed heavier architecture in terms of throughput and latency. This is primarily due to fewer parameters and lower computational complexity, making them particularly suitable for real-time and resource-constrained environments.

However, heavier architectures like ResNet-50 and InceptionV3 typically offer higher accuracy and stronger generalization capabilities, as indicated in prior studies. Therefore, while batching can significantly enhance inference efficiency, the selection of CNN architecture must consider both the specific application's accuracy requirements and computational constraints.

The observed improvements from batching are explained by better hardware utilization, reducing overhead associated with individual inference requests. The significant latency drops for ResNet-50 and InceptionV3 under batching conditions highlights batching's potential as an essential optimization technique for deployment. This aligns with findings from the literature, where batching is frequently recommended to balance computational efficiency and inference speed, especially for resource-intensive CNN architectures.

Despite its benefits, batching does introduce additional latency per request when batches are partially filled, potentially impacting real-time applications where individual latency is critical. Therefore, practitioners must carefully choose batching parameters based on application-specific latency and throughput requirements.

4. Conclusion

In conclusion, the comparative analysis performed in this study emphasizes the necessity of balancing accuracy, computational complexity, and inference efficiency. While lighter CNNs provide substantial benefits in throughput and latency, heavier CNNs offer superior accuracy and robustness. Effective batching strategies, such as the one employed here, can significantly enhance inference performance, thereby broadening the deployment potential of complex CNN architectures. Future research could explore dynamic batching techniques and investigate their real-world applicability across various CNN architectures and diverse hardware platforms. Further testing with larger batch sizes and alternative deployment settings might also yield additional insights into optimizing inference efficiency for CNN-based image classification systems.

References

[1] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016: 770–778.

[2] Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016: 2818–2826.

[3] Sandler M, Howard A, Zhu M, Zhmoginov A, Chen LC. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018: 4510–4520.

[4] Olston C, Fiedel N, Gorovoy K, Harmsen J, Lao L, Li F, Rajashekhar V, Ramesh S, Soyke J. Tensorflow-serving: Flexible, high-performance ML serving. arXiv preprint arXiv:1712.06139, 2017.

[5] TensorFlow Datasets. "tf_flowers Dataset." Available online: https://www.tensorflow.org/datasets/catalog/tf_flowers, 2019.

[6] Li Z, Liu F, Yang W, Peng S, Zhou J. A survey of convolutional neural networks: Analysis, applications, and prospects. IEEE Transactions on Neural Networks and Learning Systems, 2021, 33(12): 6999–7019.

[7] Wu J. Introduction to convolutional neural networks. National Key Lab for Novel Software Technology, Nanjing University, 2017, 5(23): 495.

[8] Ajit A, Acharya K, Samanta A. A review of convolutional neural networks. In 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE), 2020: 1–5.

[9] Mannor S, Peleg D, Rubinstein R. The cross entropy method for classification. In Proceedings of the 22nd International Conference on Machine Learning, 2005: 561–568.

[10] Mao A, Mohri M, Zhong Y. Cross-entropy loss functions: Theoretical analysis and applications. In International Conference on Machine Learning, 2023: 23803–23828.