PieSlicer: Dynamically Improving Response Time for Cloud-based CNN Inference

Samuel S. Ogden
ssogden@wpi.edu
Worcester Polytechnic Institute

Xiangnan Kong
xkong@wpi.edu
Worcester Polytechnic Institute

Tian Guo
tian@wpi.edu
Worcester Polytechnic Institute

ABSTRACT
Executing deep-learning inference on cloud servers enables the usage of high complexity models for mobile devices with limited resources. However, pre-execution time—the time it takes to prepare and transfer data to the cloud—is variable and can take orders of magnitude longer to complete than inference execution itself. This pre-execution time can be reduced by dynamically deciding the order of two essential steps, preprocessing and data transfer, to better take advantage of on-device resources and network conditions. In this work we present PieSlicer, a system for making dynamic preprocessing decisions to improve cloud inference performance using linear-regression models. PieSlicer then leverages these models to select the appropriate preprocessing location. We show that for image classification applications PieSlicer reduces median and 99th percentile pre-execution time by up to 50.2ms and 217.2ms respectively, when compared to static preprocessing methods.

KEYWORDS
Cloud inference, mobile deep learning, performance modeling

1 INTRODUCTION
The ever increasing accuracy of deep learning models comes at the cost of higher computation [5], often far beyond the capabilities of mobile devices [16, 21, 34]. By offloading inference execution to cloud and edge servers, referred to as cloud-based inference, mobile devices can therefore benefit from these high-accuracy models [3, 14, 22, 26, 51]. Leveraging cloud servers for inference requires completing a number of operations, including transferring and preprocessing the input data, prior to executing inference tasks on the servers. However, the time to complete these operations, collectively defined as the pre-execution time can be orders of magnitude longer than inference execution time.

In this work, we characterize pre-execution time and investigate ways to reduce it. Our first goal is to identify and understand factors that impact pre-execution time. Due to dynamic mobile environments and heterogeneous mobile capacities, pre-execution time can be highly variable. Further, the two major contributors of it, preprocessing and network transfer, are interdependent. While on-device preprocessing can reduce network transfer time it is slower than in-cloud preprocessing. This drives our second goal to dynamically make preprocessing decisions based on these factors.

To these ends we introduce PieSlicer, a system that allows us to empirically measure and model pre-execution time components in order to reduce pre-execution time. We isolate the four components of pre-execution time for an image classification task and observe that they can be modeled with low prediction error, allowing for accurate decision making. We demonstrate the ability of PieSlicer to make accurate predictions across a range of inputs and environments using two datasets, three devices and two network. These predictions can lead to a median pre-execution time reduction of 50.2ms—a noticeable improvement for end users—compared to static on-device processing, and an F1 accuracy score of over 0.98 in all test cases, indicating high quality decisions.

Prior work on inference performance optimization has focused on either reducing model execution latency [6, 9, 24, 30] or network transfer time [14, 22, 26, 29, 51]. To reduce network transfer time, researchers have looked at leveraging regions of interest [7, 11, 32], deep learning aware image compression [33, 49], and model partitioning [25, 46]. However, these approaches often require either infrastructure upgrades or designing new deep learning models, and often do not consider the interplay between preprocessing location and network conditions. Further, as these approaches achieve low execution time in the orders of tens milliseconds [41], pre-execution time has now become the dominating component of cloud-based inference. As such PieSlicer fills the gap by improving the pre-execution time, via empirical measurement and data-driven modeling techniques. Consequently, PieSlicer has the potential to be used in tandem with many existing techniques described above.

We make the following main contributions.

• We identify and characterize key mobile-specific factors that impact pre-execution time—a dominant component of end-to-end response time. We show that linear regression models yield adequate prediction accuracy with low overhead.

• We design and implement a prototype of PieSlicer to dynamically select the preprocessing location at runtime. These preprocessing decisions are powered by our accurate linear regression models. The source code can be found at [36].

• We evaluate PieSlicer with three devices, two networks, and two real-world datasets. Our experiments show that PieSlicer reduces pre-execution time by up to 217.2ms and achieves a decision F1 score of 0.99.

2 CLOUD-BASED INFECTION BACKGROUND
Cloud-based inference can be broadly divided into a number of steps which we illustrate in Figure 1. We use an image classification application to detail these steps as it is both an intuitive example and the current focus of PieSlicer.

Input capture 1. Data is collected for inference and saved to the device. For image classification this is image capture, or the selection of an existing image. Improving image capture generally works by reducing input size, such as by resizing input data as part of capture [19] or by optimizing the format for deep learning [49].

On-device preprocessing 2. Preprocessing for image classification generally consists of resizing and cropping the image. This can result in a decrease of orders of magnitude in terms of file size, from...
In this work, we look at how to make on-device *dynamic preprocessing* decisions for cloud deep learning inference. Our key goal is to improve cloud inference performance by reducing the pre-execution time. We target pre-execution time because while inference execution can be as low as tens of milliseconds [35, 44, 48, 51], pre-execution time can be orders of magnitude longer. A reduction in pre-execution time has the key benefit of improved response time but is challenging due to its dependency on on variable factors such as device capabilities and network connections, necessitating on-device dynamic decisions.

**System model.** We focus on a popular category of mobile applications that leverage convolutional neural networks (CNNs) for image classification [50]. We chose to target image classification because state-of-the-art models are a topic of much ongoing research [17, 18, 45, 52]. We assume mobile developers use the API provided by our work for in-cloud inference in order to utilize complex deep learning models. We further assume that mobile devices are of varying computational capacity and may be operating under different network conditions. Lastly, the cloud inference server must be at least as powerful as the most powerful mobile device.

**Motivation and Challenges.** Figure 2(a) compares the total pre-execution time distribution between always preprocessing on the device (e.g. on-device) and always preprocessing in the cloud (e.g. in-cloud). Even though on-device preprocessing significantly reduces the network transfer time, as shown in Figure 2(b), it can be up to an order of magnitude slower than preprocessing on the cheapest Amazon cloud GPU server, shown in Figure 2(c). Methodology details are in Section 4. This suggests the need to dynamically choose between on-device and in-cloud preprocessing. Such decisions are impacted by factors such as on-device capacity and network conditions, making it challenging to make the correct decision.

Furthermore, we need to address deep learning-specific challenges. First, deep learning models are highly dependent on the quality of their input data, so we must consider the impact of different input formats on inference performance; lossy storage formats like JPEG may reduce pre-execution latency but lead to lower inference accuracy [49]. Second, the size of input data, both raw and preprocessed can lead to different networking trade-offs based on how much on-device processing occurs. In summary, due to these trade-offs it is insufficient to simply apply a single, static decision as this often leads to poor performance.

**Solution Overview.** To dynamically decide when to preprocess images based on mobile factors, we model the performance of each step and implement a library, based on these models, to make preprocessing location decisions. In Section 4 we measure the steps that comprise pre-execution time and demonstrate that using linear regression models strikes a balance between prediction accuracy and latency. Section 5 introduces PIE\textsuperscript{SLICER}, a system that leverages these models for making preprocessing decisions at runtime. We demonstrate in Section 6 the efficacy of PIE\textsuperscript{SLICER} in reducing pre-execution time using real-world images and mobile devices.

## 4 CHARACTERIZING PRE-EXECUTION TIME

Being able to accurately predict an inference request’s pre-execution time is critical to making appropriate preprocessing decisions. As
such decisions need to be made at inference time it is also important that these decisions be efficient and thus rely on features that are cheap to obtain, such as image file size and resolution. In this section, we study the impact of mobile-specific factors (Section 4.2) on predicting pre-execution time by leveraging data collected with two mobile networks, three mobile phones, and two Flickr image datasets. We explore five common modeling approaches and show that we can effectively model network time, especially when using device- and network-specific models (Section 4.3). We will describe how we design PieSlicer to leverage these modeling insights for making dynamic preprocessing decisions in Section 5.

4.1 Measurement Methodology

Datasets. We created two Flickr image datasets image-1k and image-5k in order to more closely resemble the wide range of image sizes that would be captured in real-world scenarios than existing datasets [12, 28]. The image-1k dataset contains 1000 images evenly distributed in size, while the image-5k has >5000 randomly selected images. Details can be found in our github repository[36].

Hardware. We used three mobile devices and a cloud-based server, detailed in Table 1, for collecting relevant performance data. These three phones have different processing power, and the high-end device also has a specialized image processing hardware. We used a p2.xlarge as it represents the cheapest GPU-accelerated EC2 server, and connected over university and residential WiFi networks.

Measurement setup. We measured components of inference execution through our Android application and an inference server plug-in that form the basis of PieSlicer (further described in Section 5). For on-device preprocessing we used Android’s built-in BitmapFactory class and performed in-cloud preprocessing using the Pillow-SIMD library, both using the nearest neighbor filter. Each request was created with one JPEG image from our datasets and was sent from the mobile device to the cloud inference server. Each requests record pre-execution time components and five easy-to-obtain features: (i) original file size, (ii) width, (iii) height, (iv) resolution, and (v) sent file size.

**Measuring pre-execution time.** To understand the key factors that impact pre-execution time, we divide it into the following components. (i) On-device preprocessing time refers to the time to resize an image to a pre-specified resolution, and then save the resulting bitmap to the mobile storage; (ii) in-cloud preprocessing time measures the time for a cloud server to perform the same resizing operation; (iii) network transfer time is the sum of the time to send the inference request to and to send the response back from the cloud server; (iv) cloud preparation time is defined as the time to transform the preprocessed image into the input structure required by deep learning frameworks for executing CNN models.

We measured each time component independently and saved the mobile preprocessing time to an on-device sqlite database. Cloud-based time measurements were returned with the inference response. Network transfer time was derived as the difference between the total remote time recorded by the mobile device and the total time reported by the cloud server. We performed the above measurements for all three mobile devices on both networks.

4.2 Impact of Mobile-specific Factors

We first examine how mobile-specific factors impact pre-execution time. Figure 3 shows the relationship between original image size (i.e., file size before preprocessing) and each of the four measured time components. Each marker corresponds to one image.

Our first observation is that preprocessing time has a strong linear relationship with both on-device and in-cloud preprocessing. This suggests that original file size is useful in predicting preprocessing time. Additionally, we observe that in-cloud preprocessing is up to 2× faster than on-device preprocessing, demonstrating a potential benefit of skipping on-device preprocessing. Second, we observe that each device has a distinct preprocessing speed, and each network has a distinct transfer latency. This suggests that per-device and per-network models will be beneficial. Third, the cloud preparation time took less than 0.25ms for all tested inputs with little relation to input file size. This suggests we can safely represent it as a constant of 1ms throughout this work.
4.3 Modeling Network Time

In this section, we explore five different machine learning models for predicting network transfer time. We focus on network transfer time as it is on the critical path and shares similar patterns to other pre-execution time components.

Training data preprocessing. We first partitioned the collected measurement data into 36 subsets based on mobile device, network, image dataset, or some combination of these factors, and one-hot encoding to identify specific devices and networks. We then removed data outliers that are below the 5th and above the 95th percentile of network time. Finally, we applied min-max normalization to each subset and used an 80-20 training and testing split.

Machine learning models. First, based on the linear trend we observed in Figure 3, we chose to model the data using linear regression. Second, we used a Lasso approach to identify unnecessary features. Third, we used a K-nearest neighbors regression model, which estimates latency as the average of the most similar training data points. We evaluated all k values from the set \{1, 2, 4, 8, 16\} and chose the best performing k value for each dataset. Fourth, we used a random forest regressor which allows finding the most important features. We performed a grid-search for two hyperparameters, the number of estimators in \(\{1, 2, 4, 8, 16\}\) and the maximum depth of 10. Finally, Support Vector Regression (SVR) is chosen to find the best, potentially non-linear, prediction boundary for our data. Each model was trained for each of the 36 subsets of data described previously.

Training details. Each model was trained on a subset of data using 10-fold cross validation with Mean Absolute Percentage Error (MAPE) as the training metric. MAPE is the average absolute error as a percentage of the ground-truth value, with lower values being better. This metric allows us to scale the prediction error based on the predicted value, thus enabling fair comparison of training performance.

Analysis of results. Figure 4 shows the CDF of MAPE for the five different model types on the 36 subsets of data. We observe that in 85% of cases the linear regression model performs as well as more complex SVR and KNN models. In Table 2, we see that the poor performance for linear regression was all due to using the image-1k dataset without specifying the network. This poor performance is not surprising given that large images exacerbate the difference between different networks, as seen in Figure 3(b).

Despite the slightly lower prediction accuracy in some cases linear regression models are preferable to both KNN and SVR due to training and usage constraints. KNN not only requires hyperparameter tuning but also requires the usage of all training data for each inference. SVR has greater than quadratic training time \[31\] (compared to linear time for Linear Regression) leading to a scaling issue as more data is collected. Given the relatively close prediction accuracy among all models and drawbacks of other models, we choose to use linear regression models in PiELICER.

4.4 Other Factors: Compression and Resolution

Lastly, we briefly discuss the impact of two factors, image quality and resolution, on pre-execution time and inference accuracy. We use the NasNet Large model with in-cloud preprocessing as the baseline and define normalized accuracy as the percentage of image inferences that match the results of in-cloud preprocessing. We found that these two factors only had negligible impact when compared to the baseline. Consequently, in designing PiELICER, we assume these two factors are pre-determined and provided by mobile developers.

Image quality. The JPEG standard includes a quality setting, ranging from 1-100, that denotes how much information to keep when
We used the same measurement setup from Section 4.1 and tested work for cloud-based deep learning inference. Using our empirically which aim to alleviate the impact of lossy compression on accuracy.

We next describe how we leverage the performance models to determine the best preprocessing path and choose the one with the lower pre-execution time, allowing the on-device performance models to be retrained. As briefly discussed in Section 4.3, training the model with different subsets of data led to different prediction accuracies. Table 2 compares the prediction performance of linear regression models for network transfer time, trained with different data subsets. Similarly, Table 3 compares the test accuracy for on-device and in-cloud preprocessing. We report Mean Absolute Percentage Error (MAPE). For example, the prediction accuracy of the model trained with measurement data collected with our mid-end device on university WiFi with the image-6k dataset is 16.56%, one of the lowest reported errors. The four lowest and highest reported errors are highlighted in green and red, respectively.

We make two observations. First, we see in Table 2 that network models that combine measurements from different networks, even when using one-hot encoding to differentiate networks, have high error. This is especially true for models trained with the image-1k dataset where there is a 10x increase in MAPE. We suspect that this increase for image-1k is due to the large average image file sizes of the image-1k dataset, making them more sensitive to network performance variations.

Second, models trained with more specified datasets (e.g. a mid-end phone on university WiFi) tend to have lower error. We see this both in Table 2 and Table 3 where in many cases combined datasets have the highest error rate. This observation can be understood by observing the distinct trendslines that are seen in Figures 3(a)-3(b). Therefore, we opt to use per-device and per-network models to best account for different hardware and network in PiESlicer.

5.2 Using Pre-execution Time Models

We next describe how we leverage the performance models to make dynamic decisions regarding preprocessing location. For each inference request, PiESlicer leverages the performance models corresponding to the device, network, and server that are currently being used. That is, we estimate the pre-execution time for both on-device preprocessing $T^m$ and in-cloud preprocessing $T^c$ as:

$$T^m(x) = T^{m}_{prep}(x) + T^{m}_{nw}(x) + T^{c}_{prep}(x) + c,$$

$$T^c(x) = T^{m}_{nw}(x) + T^{c}_{prep}(x) + c,$$

where $x$ is the input features to our model. The cloud preparation time is denoted as $c$ and is set to be 1ms as discussed in Section 4.2.
\( T_{\text{preprocess}}^{m} \) and \( T_{\text{preprocess}}^{c} \) are the preprocessing models for the mobile device and the cloud server, respectively. \( T_{\text{net}} \) is the specific network transfer time model for the currently active mobile network. If \( T_{\text{net}}^{m}(x) < T_{\text{net}}^{c}(x) \), PieSlicer will choose to perform preprocessing on the mobile device, and otherwise will use in-cloud preprocessing. Currently, one of the features, image file size to send, is estimated to be the average size of previously preprocessed images.

**Retraining models.** In order to ensure the accuracy of decisions made by PieSlicer, it is important to keep performance models up to date. PieSlicer retracts its linear regression models periodically to ensure that each mobile device has access to up-to-date performance models. As the time to retrain regression-based models is low but non-negligible, the frequency of retraining is in large part related to the amount of new performance data collected as well as the accuracy of the current on-device performance models. In this work, we use a simple strategy to trigger the retraining once the precision accuracy falls below the training accuracy [13].

Retraining is done either on-device or in the cloud, depending on the type of models. Mobile-specific models, such as those for on-device preprocessing and network models, are trained on-device. This allows for keeping mobile-specific data local and ensuring that the data used for modeling is relevant to the device being used. Cloud-specific models, such as those for in-cloud preprocessing, are trained in the cloud and parameters are attached to inference responses. This is enabled by using linear regression models since they require only as many parameters as they have inputs.

**Adapting to network transfer time variations.** Because mobile networks are inherently variable even when using the same type of networks, the predicted network transfer time can deviate from the actual time. Although retraining models, as outlined above, will mitigate long-term network changes, transient changes can still be problematic. To mitigate the impact of these transient network changes on preprocessing decisions, we use a delta-based approach to reactively adjust the predicted network transfer time based on recently observed variations, if any. Concretely, for each inference request \( i \), we record the predicted network transfer time as \( T_{\text{net}}^{i} \) and the actual network time \( T_{\text{net}}^{\text{act}}^{i} \). We use \( \hat{\delta}^{i} = \frac{(T_{\text{net}}^{\text{act}}^{i} - T_{\text{net}}^{i})}{\text{size}_{i}} \) to represent the difference in bandwidth prediction where \( \text{size}_{i} \) is the size of request \( i \), with a positive \( \hat{\delta} \) indicating network conditions are worse than predicted. Applying exponential smoothing, we calculate \( \Delta^{i} = (1-\alpha)\Delta^{i-1} + \alpha\hat{\delta}^{i} \) where \( \alpha \in (0, 1) \). For the next inference request \( i+1 \), PieSlicer will estimate the network transfer time to be \( \hat{\delta}^{i+1} + \Delta^{i} \).

**Selectively using the on-device performance models.** In some cases real-world input results in very small inputs which would be inefficient to consider for on-device preprocessing. In these cases we leverage two fast on-device checks (<4μs) to decide whether to use the on-device performance models. These checks considered two factors: (i) file size; and (ii) image resolution.

In the first we see whether the file size is larger than the average transmitted filesize (~53KB). If it is smaller then PieSlicer then it is likely that it is a very small image and on-device preprocessing is unnecessary. In the second check we see whether the image resolution is less than the preprocessing target size (e.g. 331x331 pixels). If it is then any preprocessing would only increase the file size and potentially decrease accuracy. If either of these conditions is true then the raw image data is transmitted to the cloud-based server.

### Table 4: Comparison of pre-execution latency to baselines.

We compared the pre-execution time achieved by PieSlicer and the baseline approaches in terms of what percentage they were of the of the empirically derived static minimum, which is shown as absolute time in milliseconds. PieSlicer in many cases outperforms any of the static baselines.

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>Residential</th>
<th>University</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-End</td>
<td>Static Minimum</td>
<td>713.2ms</td>
<td>1231.2ms</td>
</tr>
<tr>
<td></td>
<td>Static remote</td>
<td>922.6ms</td>
<td>1094.7ms</td>
</tr>
<tr>
<td></td>
<td>Static local</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td>PieSlicer</td>
<td>98.9%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Mid-End</td>
<td>Static Minimum</td>
<td>342.4ms</td>
<td>875.6ms</td>
</tr>
<tr>
<td></td>
<td>Static remote</td>
<td>1082.3%</td>
<td>1353.0%</td>
</tr>
<tr>
<td></td>
<td>Static local</td>
<td>100.1%</td>
<td>103.1%</td>
</tr>
<tr>
<td></td>
<td>PieSlicer</td>
<td>97.3%</td>
<td>96.6%</td>
</tr>
<tr>
<td>High-End</td>
<td>Static Minimum</td>
<td>448.7ms</td>
<td>690.0ms</td>
</tr>
<tr>
<td></td>
<td>Static remote</td>
<td>1457.6%</td>
<td>1818.5%</td>
</tr>
<tr>
<td></td>
<td>Static local</td>
<td>100.1%</td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td>PieSlicer</td>
<td>98.9%</td>
<td>96.3%</td>
</tr>
</tbody>
</table>

### 6 EXPERIMENTAL EVALUATION

Our key evaluation goal is to quantify the effectiveness of PieSlicer in reducing pre-execution time and examine its decision accuracy. We found that PieSlicer incurs minimal overhead of 0.33ms on average, or 0.07% per request.

#### 6.1 Experimental Setup

We use the same setup as in Section 4.1 for evaluating PieSlicer.

**Baseline policies.** We evaluated PieSlicer against three baselines. Static local always preprocesses the inference request on mobile devices before sending it to the cloud servers. Static remote always sends the raw input data directly to the cloud servers for preprocessing. We also derive a static minimum baseline by picking the lower pre-execution time out of the above two static baselines.

**Performance metrics.** We chose \( F_1 \) score to measure PieSlicer’s ability in making preprocessing placement decisions. The \( F_1 \) score is calculated as a harmonic mean of the precision and the recall. A perfect precision and recall corresponds to an \( F_1 \) score of 1. In our case, precision is calculated as the number of correctly predicted requests preprocessed locally divided by the total number of local preprocessing decisions made by PieSlicer. The recall is calculated as the number of correctly decided local preprocessing decisions divided by the total number of requests that should use local preprocessing. To analyze the reduction in bandwidth usage due to PieSlicer, we use the metric of bandwidth utilization. This metric is calculated by comparing the number of sent bytes by PieSlicer to the bytes incurred when using static remote.

#### 6.2 Latency Reduction and Prediction Accuracy

In this experiment, we quantify the pre-execution time savings provided by PieSlicer, as well as PieSlicer’s decision accuracy. We used PieSlicer running on each mobile device to make preprocessing decisions dynamically. We sent all images from the image-1k dataset over both the university and residential WiFi, and report the pre-execution time for PieSlicer and our three baselines.

**Pre-execution time reduction.** In Table 4 we compare the pre-execution time of PieSlicer and three baselines at a range of quantiles. We report the absolute time for static minimum and normalize the performance of other approaches against it.
were sufficient for making dynamic preprocessing decisions. The energy consumption for on-device preprocessing ranging from 0.8J to 50J for our largest unpreprocessed image. Mission of data over a WiFi network by a mobile device is at least 0.8J. Previous work has shown that energy consumption for the transmission of data over a WiFi network by a mobile device is at least 0.005J/kB [42]. This equates to roughly 0.265J of energy for a 53kB preprocessed image and 50J for our largest unpreprocessed image. On-device preprocessing for our Pixel device is done on the Pixel Visual Core device which uses a maximum of 8W [4], leading energy consumption for on-device preprocessing ranging from 0.8J to 2.5J, for small and larger images respectively. This further shows that small images are more energy efficient to transmit for remote preprocessing while large images can be an order of magnitude more energy efficient through local preprocessing. Thus, PieSlicer can reduce energy consumption through a reduction in network bandwidth. Further, since PieSlicer makes these decisions quickly, always in < 1ms which equates to approximately 4mJ of energy [1], it does so with negligible overhead.

### 6.4 Effectiveness of Optimizations

Next, we quantify the effectiveness of PieSlicer’s two optimizations: delta-based network adaptiveness and selective usage of on-device performance models. For this test we set $\alpha = 0.5$. When using both optimizations we see a reduction in pre-execution time by 49.3ms (6.4%) at the 95th percentile and 85.3ms (7.4%) at the 99th percentile. If only the adaptive optimization is enabled, we observe that PieSlicer reduces per-execution time by up to 3.2%; while if only the selective optimization is used, we observe that PieSlicer reduces pre-execution by up to 4.0%. Our observations suggest that both adaptive and selective optimizations are beneficial in improving PieSlicer’s robustness and with minimal overhead.

### 7 RELATED WORK

**Computation offloading for Deep Learning.** Offloading computationally intensive tasks to remote servers is a common technique for mobile devices. This can be done either to reduce latency and energy consumption [8, 10, 27]. Offloading of deep learning inference [20, 25, 46] generally partitions execution between on-device and remote execution, requiring prepartitioned models to be present on the mobile device. PieSlicer proposes an alternative approach to deep learning offloading that fully takes advantage of cloud-based hardware whenever possible by having model execution be entirely handled on this more powerful hardware. This is more similar to traditional off-loading techniques by removing the need to manually partition deep learning models.

**In-cloud Inference Execution.** High-accuracy deep learning models have high computational requirements [5], which has driven the need to run them on powerful cloud servers, potentially with specialized hardware [24]. Industry frameworks [2, 37] aim to make models available for inference while minimizing latency by allowing optimizations. Other approaches may try to optimize for other factors such as throughput [9, 15, 26], accuracy [35], or cost [43]. Since many frameworks accept a target execution latency [26, 35, 43], by reducing pre-execution latency PieSlicer increases their ability to meet these targets.

### 8 DISCUSSION

**Generalizability.** In this work we used pre-execution time in image classification as a motivational example, but PieSlicer could be used in analyzing other deep learning applications [3, 7, 39] which have similar workflows but different preprocessing trade-offs. For example, virtual assistants could leverage the profile aspect of PieSlicer to identify choke points and dynamically adapt accuracy.

**Implications of future technology.** As other fields develop they will improve aspects of the steps discussed in Section 2, which
potentially only increases the need to understand the interplay between the different factors. One such improvement is the increased bandwidth provided by the introduction of 5G, which would be expected to encourage more in-cloud preprocessing due to decreased network latency. The modular models used by PieSlicer allow it to incorporate such improvements and are thus orthogonal to PieSlicer by further reducing overall response latency.

9 CONCLUSION

We demonstrated the importance of modeling the pre-execution latency for mobile devices that leverage cloud inference, and introduced effective techniques for reducing this latency. Through empirical characterization, we found that pre-execution latency can often be orders of magnitude longer than execution time itself, making it a prime candidate for optimization. Further, our exploration of machine learning based performance models showed that linear regression models allow for adequate modeling accuracy for the steps that comprise pre-execution time with low overhead.

Based on the key findings from our empirical characterization and modeling, we further designed and built PieSlicer, a system for dynamically determining preprocessing location in an accurate and agile manner. Using simple models, PieSlicer achieved a classification $F_1$ accuracy of up to 0.99, leading to 217.2ms reduction over the best static approach.

ACKNOWLEDGEMENT

We thank all anonymous reviewers and shepherds for their insightful feedback and NSF support under grants CNS-#1755659 and CNS-#1815619.

REFERENCES

[34] Ogden, S.S. et al. Mobi: Mobile deep inference made efficient by edge computing. In USENIX Workshop on Hot Topics in Edge Computing (HotEdge’18), 2018.