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 99$^{\text{th}}$ 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 F_1 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 F_1 score of 0.99.

2 CLOUD-BASED INFERENCE 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 ❶. 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 ❷. 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

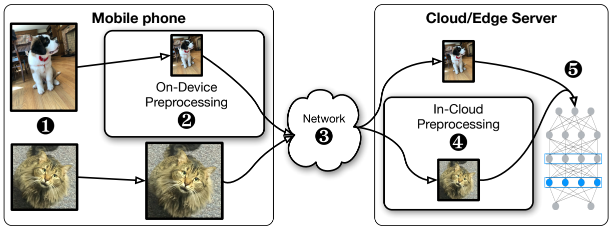


Figure 1: ***Cloud-based Deep Inference Workflow.*** *In general, there are five steps:* input capture ❶, on-device pre-processing ❷, network transfer ❸, in-cloud preprocessing ❹, *and* deep learning model execution ❺. *Steps* ❷*-*❹ *comprise pre-execution and present opportunities to make dynamic decisions to reduce latency.*

tens of KBs to over 10MB. This dramatic reduction can improve network transfer, the next step, but is slow compared to in-cloud preprocessing even with specialized hardware. This leads to a potential trade-off where small inputs might benefit without on-device preprocessing, a trade-off that PieSlicer aims to exploit. Prior work has typically assumed preprocessed input data [23, 25, 34, 38, 41, 46] leading to this being a rich area for improvement.

Network transfer ❸. The transfer of data across the network occurs regardless of whether data has been preprocessed on device. As shown in Figure 2(b) this transfer time can lead to large transfer latency, especially for data that was not preprocessed on-device. Compounding this is the range of mobile networks experienced by mobile devices. Prior work handles reducing network impact on cloud-based inference through leveraging regions of interest [7, 11, 32], or deep learning aware image compression [33, 49]. Such techniques often require non-negligible on-device processing power which might not be available on all mobile devices.

In-cloud preprocessing and preparation ❹. In-cloud preprocessing consists of transforming the data to the format needed for the deep learning model, as well as the preprocessing that was potentially skipped on-device. Due to the powerful servers and specialized libraries in-cloud preprocessing is much faster than preprocessing on-device. In addition to image preprocessing, many deep learning queries generate a number of sub-requests or need to gather additional input data before inference execution, which have been studied by previous systems [26, 51].

Inference execution ❺. After all previous steps have been completed inference execution can begin. This step has been optimized by a number of existing frameworks [9, 23, 37] allowing for complex models to be executed with low latency and high throughput, or better resource utilization [9, 35].

2.1 Pre-execution Time and its Trade-offs

Pre-execution consists of steps ❷, ❸, ❹ and minimizing pre-execution time entails balancing the complex interplays between them. The selection of preprocessing location between on-device and in-cloud presents a key trade-off for pre-execution time. Even though on-device preprocessing can reduce network time due to transferring less data, we will see in Section 4.2 that in-cloud preprocessing can be up to 2× faster. We therefore use lightweight models to make request-by-request decisions, which we discuss in Section 5.

3 PROBLEM STATEMENT

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.

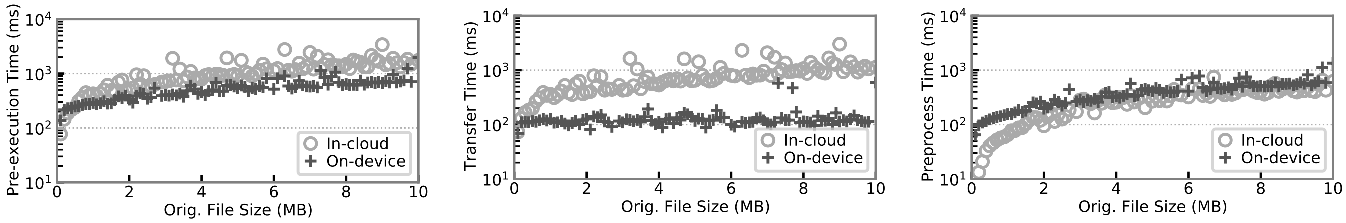
Motivation and Challenges. Figure 2(a) compares the total pre-execution time distribution between always preprocessing on the device (e.g. *on-device*) and always preprocessing in the cloud (e.g. *in-cloud*). Even though on-device preprocessing significantly reduces the network transfer time, as shown 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 PieSlicer, a system that leverages these models for making preprocessing decisions at runtime. We demonstrate in Section 6 the efficacy of PieSlicer 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



(a) *Pre-execution time.* **(b)** *Network transfer time.* **(c)** *Preprocessing time.*

Figure 2: *Performance comparison between on-device and in-cloud preprocessing with the* image-5k *dataset.* *We used a mid-end phone with university WiFi to transfer the images in our* image-5k *dataset and collected the preprocessing time, network transfer time and total pre-execution time for each request. Displayed points have been binned into 100KB increments. The y-axis is* log *scale to differentiate the trends.*

Table 1: *Hardware used in PieSlicer measurement infrastructure.* *We chose three representative mobile phones and the cheapest GPU-accelerated EC2 server to characterize their impact on pre-execution time.*

Device	OS version	CPU	Accelerator	RAM	Storage	Cameras
low-end (Nexus 5)	Android 6.0	2.26 GHz quad-core	129.8 GFLOPs Adreno 330	2GB	16GB	8/1.3MP
mid-end (Moto X4)	Android 8.1	2.2 GHz Octa-Core	163.2 GFLOPs Adreno 508	3GB	32GB	12/16MP
high-end (Pixel 2)	Android 8.1	2.35 GHz Octa-Core	567 GFLOPs Adreno 540 & Pixel Visual Core	4GB	128GB	12.2/8MP
server (p2-xlarge)	Deep Learning AMI Version 24.0	4x 2.3GHz	4.113 TFLOPs Tesla K80	61GB	75GB	N/A

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.

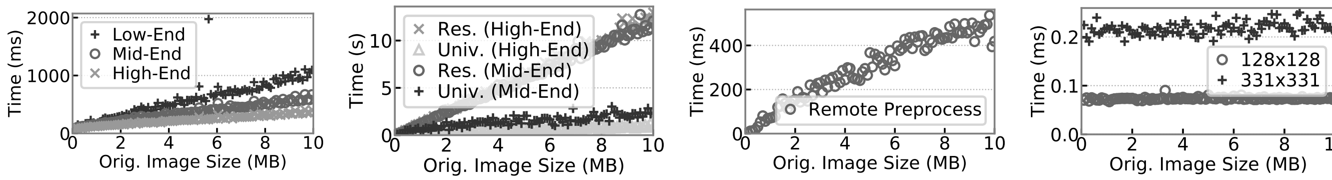
Measuring pre-execution time. To understand the key factors that impact pre-execution time, we divide it into the following components. *(i) On-device preprocessing time* refers to the time to resize an image to a pre-specified resolution, and then save the resulting bitmap to the mobile storage; *(ii) 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.



(a) On-device preprocessing. **(b)** Network transfer time. **(c)** In-cloud preprocessing. **(d)** Cloud preparation.

Figure 3: *Pre-execution time breakdown using* image-1k. *We observe that image size exhibited strong linear relationship for all but cloud preparation time. As expected, preprocessing time worsened as the device capacity decreased, e.g., the low-end phone was 2.3X slower than the high-end phone. Additionally, using residential WiFi (shown as Res.) had 5.2X higher network transfer latency compared to using university WiFi (shown as Univ.).*

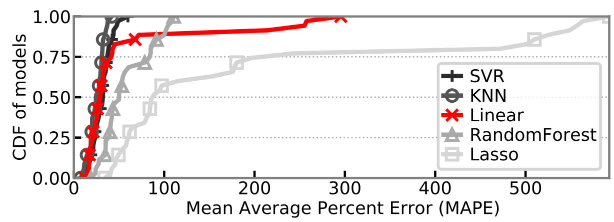


Figure 4: *CDF of model accuracy.* *More than 85% Linear regression models had similar MAPE compared to two best performing models KNN and SVR.*

Table 2: *Linear regression MAPE for network transfer time.* *Modeling based on mobile device type and network type (e.g. residential or university) shows a range of accuracy depending on the specificity of the training dataset. Generally, modeling specific network-device combinations results in lower error and modeling disparate networks with* image-1k *resulted in higher error.*

	image-6k			*image-5k*			*image-1k*		
	All	**Res.**	**Univ.**	**All**	**Res.**	**Univ.**	**All**	**Res.**	**Univ.**
All	45.66	25.92	25.59	31.24	26.21	16.97	257.56	29.33	67.95
High-End	42.92	32.78	28.92	35.73	32.32	15.37	295.58	19.01	30.27
Mid-End	42.55	24.29	16.56	30.29	26.59	12.52	255.40	16.91	71.78
Low-End	41.01	21.31	21.39	20.24	17.31	17.34	214.51	35.54	23.08

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 5^{th} and above the 95^{th} 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 datapoints. We evaluated all k values from the set $\{1..29, 50, 100\}$ 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 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. Figured 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 PieSlicer.

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 PieSlicer, 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

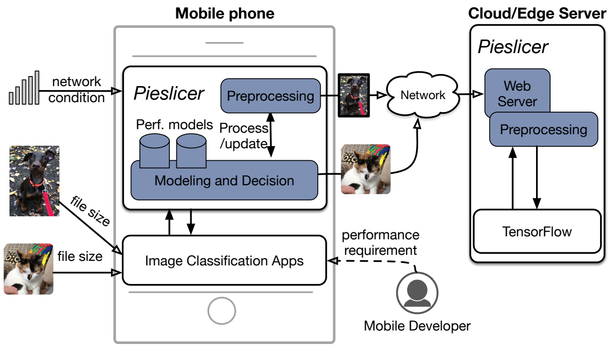


Figure 5: *PieSlicer design and an example workflow.* *Mobile applications use PieSlicer for image data preprocessing. Shaded components are main parts of PieSlicer. For simplicity, the inference response is omitted.*

compressing the image [47], which allows for a trade-off between image quality and file size. We observed that a quality setting of 90 provides the largest time reduction while exhibiting a small impact on inference accuracy, thus we use this setting throughout the rest of this work. Similar results have been shown by previous works which aim to alleviate the impact of lossy compression on accuracy either through improving the deep learning model [45] or changing the image compression algorithm [49].

Preprocessing resolution. Similar to image compression quality, resolution also presents an opportunity to reduce network time. We used the same measurement setup from Section 4.1 and tested three resolutions using *image-1k*. We observed that preprocessing to 128×128pixels reduced the average network transfer time by 27.7ms, but reduced normalized accuracy by 14.2% (from 98.6%). Therefore in this work we use a preprocessing resolution of 331×331pixels.

5 PIESLICER

PieSlicer is designed to be a dynamic image preprocessing framework for cloud-based deep learning inference. Using our empirically derived models, PieSlicer considers both on-device and in-cloud preprocessing, and chooses the one with the lower pre-execution time. Figure 5 shows an example workflow of using PieSlicer.

PieSlicer is prototyped as an Android library and a python-based inference server plug-in and consists of three main components. The first is a modeling and decision engine that uses performance models to determine the best preprocessing path and retrains these models as needed. The second is a preprocessing engine which preprocesses the input image data, based on the decision of the modeling and decision engine. The third is a server plug-in which records and reports the in-cloud preprocessing time, allowing the on-device performance models to be retrained.

5.1 Modeling Pre-execution Time

For each time component T, we use the linear model in the form of $T(x) = x^{\intercal}\beta + \epsilon$ where $x^{\intercal}$ is the vector of inputs, β is the calculated coefficients of our model and ϵ is the random error. The coefficients vector β captures each time component's dependence on input features. We used the input features described in Section 4.3. The pre-execution time is then estimated as $\sum T$ for each predicted latency on the potential execution path (more detail is provided in Section 5.2). We chose to model each component individually to increase the modularity and reduce situations where a single runtime change, such as switching to a different mobile network, requires a new prediction model.

Table 3: *Linear regression MAPE for on-Device and in-cloud preprocessing time.* *We see that the more accurate models tend to be those using* image-1k*, likely due to the wide range of input file sizes used in training. We see in all cases relatively low MAPE values, indicating a good fit.*

	image-6k		*image-5k*		*image-1k*	
	On-Device	In-Cloud	On-Device	In-Cloud	On-Device	In-Cloud
All	32.77	21.90	20.56	18.60	16.69	7.15
High-End	8.45	28.96	9.90	20.79	6.83	5.85
Mid-End	7.22	10.20	6.77	10.31	6.59	6.43
Low-End	32.01	28.28	24.23	16.94	12.61	6.65

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 understood by observing the distinct trendlines 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:

$$T^m(x) = T^m_{prep}(x) + T_{nw}(x) + T^c_{prep}(x) + c,$$
$$T^c(x) = T_{nw}(x) + T^c_{prep}(x) + c,$$

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^m_{prep} and T^c_{prep} are the preprocessing models for the mobile device and the cloud server, respectively. T_{nw} is the specific network transfer time model for the currently active mobile network. If $T^m(x) < T^c(x)$, 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 retrains 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 prediction 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 $\hat{T}^i_{nw}$ and the actual network time T^i_{nw}. We use $\delta^i = \frac{(T^i_{nw} - \hat{T}^i_{nw})}{size_i}$ to represent the difference in bandwidth prediction where $size_i$ is the size of request i, with a positive δ^i indicating network conditions are worse than predicted. Applying exponential smoothing, we calculate $\Delta^i = (1-\alpha)\Delta^{i-1} + \alpha\delta^i$ where $\alpha \in (0, 1)$. For the next inference request i+1, PieSlicer will estimate the network transfer time to be $\hat{T}^{i+1}_{nw} + \Delta^i$.

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 ($< 4\mu s$) 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 filesize 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. 331×331pixels). If it is then any preprocessing would only increase the filesize 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 the empirically derived static minimum, which is shown as absolute time in milliseconds. PieSlicer in many cases outperforms any of the static baselines.*

		Residential			University		
Device	**Algorithm**	50^{th}	95^{th}	99^{th}	50^{th}	95^{th}	99^{th}
Low-End	*Static Minimum*	713.2ms	1231.0ms	1876.6ms	707.2ms	1215.7ms	1984.5ms
	Static remote	922.6%	1094.7%	1524.9%	274.2%	288.8%	316.9%
	Static local	100.1%	100.0%	100.0%	100.5%	101.0%	100.0%
	PieSlicer	95.0%	100.3%	113.8%	93.4%	94.5%	94.1%
Mid-End	*Static Minimum*	582.4ms	875.6ms	1316.1ms	502.4ms	749.7ms	1090.2ms
	Static remote	1082.3%	1353.0%	1003.1%	275.4%	599.5%	502.6%
	Static local	100.1%	100.0%	103.1%	100.3%	100.0%	100.0%
	PieSlicer	97.3%	96.7%	83.5%	97.6%	96.6%	94.1%
High-End	*Static Minimum*	448.7ms	690.0ms	979.8ms	384.2ms	666.7ms	951.7ms
	Static remote	1457.6%	1818.5%	1454.4%	234.9%	238.8%	223.9%
	Static local	100.1%	100.0%	100.0%	100.2%	102.1%	100.0%
	PieSlicer	98.9%	96.3%	104.7%	98.1%	98.7%	105.7%

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 inimum* 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.

Table 5: *Bandwidth utilization*

	Residential Network	University Network
Low-End Device	1.91%	4.93%
Mid-End Device	1.86%	4.79%
High-End Device	1.86%	7.33%

We make the following two main observations. First, PieSlicer achieved comparable, or better, pre-execution time to the *static minimum* baseline for all three mobile devices. PieSlicer outperformed the *static minimum* baseline in 77.8% of cases and was within 6% for all but one case. Specifically, at median PieSlicer always performed better, with a decrease in latency of up to 50.2ms (7.1%) compared to the *static local* baseline and more than 1.2s (180.8%) better than the *static remote* baseline. At higher percentiles we see an even larger improvement, with up to 217.2ms decrease in latency at the 99th percentile when compared to the *static minimum* baseline. These improvements are all noticeable to end users [40] and thus can improve their experiences. Second, PieSlicer performed more accurately on university WiFi rather than residential WiFi. This supports the design choice of modeling distinct networks and devices individually.

Classification accuracy. To understand the ability of PieSlicer in making dynamic pre-processing decisions, we recast it as a binary classification problem. To do this we use the choices of *static minimum* as a ground truth and examine the choices made by PieSlicer. For all tested scenarios, PieSlicer achieved an F_1 score of at least 0.980, with a maximum score of 0.990 indicating very high classification accuracy. This suggests that our linear regression models were sufficient for making dynamic preprocessing decisions.

6.3 Bandwidth Reduction and its Implications

PieSlicer's dynamic preprocessing does not only affect the pre-execution time, but also the amount of data sent. Table 5 shows the *bandwidth utilization* of PieSlicer. We first observe that PieSlicer significantly reduced the bandwidth requirements for all tested cases. This also suggests that PieSlicer decided to preprocess most tested images on mobile devices. Second, we see that PieSlicer had lower bandwidth utilization when using residential WiFi than using university WiFi. This suggests that PieSlicer chose to preprocess more images in the cloud when using university WiFi, which aligns with our previous observation that university WiFi was faster in Figure 3(b). Finally, the high-end device had the highest bandwidth utilization on university WiFi, despite having the most powerful hardware. This indicates PieSlicer's ability to make trade-offs based on computational capacity and network connection.

Implication for Energy Savings. Below we show that PieSlicer leads to mobile energy reduction by judiciously making preprocessing decisions for images of different sizes and imposing negligible energy overhead. We present the analysis as the following. 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 95^{th} percentile and 85.3ms (7.4%) at the 99^{th} 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 when 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

[1] Qualcomm on Tour. https://www.anandtech.com/show/11201/qualcomm-snapdragon-835-performance-preview/5.
[2] NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.
[3] Deep Learning for Siri's Voice. https://machinelearning.apple.com/2017/08/06/siri-voices.html, 2017.
[4] Pixel 2 - Wikipedia. https://en.wikipedia.org/wiki/Pixel_2, 2019.
[5] Bianco, S. et al. Benchmark analysis of representative deep neural network architectures. 2018. doi: 10.1109/ACCESS.2018.2877890.
[6] Chen, T. et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI 18*, 2018.
[7] Chen, T.Y.H. et al. Glimpse: Continuous, real-time object recognition on mobile devices. In *SenSys '15*, 2015.
[8] Chun, B.G. et al. Clonecloud: Elastic execution between mobile device and cloud. In *EuroSys '11*, 2011.
[9] Crankshaw, D. et al. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation*, 2017.
[10] Cuervo, E. et al. Maui: Making smartphones last longer with code offload. In *ACM MobiSys 2010*. Association for Computing Machinery, Inc., June 2010.
[11] Dai, X. et al. Recurrent Networks for Guided Multi-Attention Classification. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'20)*.
[12] Deng, J. et al. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.
[13] Goodfellow, I. et al. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[14] Gujarati, A. et al. Swayam: Distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Middleware '17*, 2017.
[15] Gujarati, A. et al. Serving DNNs like clockwork: Performance predictability from the bottom up. In *OSDI 20*, 2020.
[16] Guo, T. Cloud-based or on-device: An empirical study of mobile deep inference. In *IC2E'18*, 2018.
[17] He, K. et al. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
[18] Howard, A.G. et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 2017.
[19] Hu, J. et al. Banner: An image sensor reconfiguration framework for seamless resolution-based tradeoffs. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19.
[20] Huang, J. et al. Clio: Enabling automatic compilation of deep learning pipelines across iot and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020.
[21] Ignatov, A. et al. Ai benchmark: All about deep learning on smartphones in 2019. 2019.
[22] Ishakian, V. et al. Serving deep learning models in a serverless platform. *CoRR*, abs/1710.08460, 2017. URL http://arxiv.org/abs/1710.08460.
[23] Jia, Y. et al. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, 2014.
[24] Jouppi, N.P. et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA'17*, pages 1–12, 2017.
[25] Kang, Y. et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, 2017.
[26] Kannan, R.S. et al. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*.
[27] Kosta, S. et al. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *2012 Proceedings IEEE INFOCOM*, 2012.
[28] Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, 2009.
[29] LeMay, M. et al. Perseus: Characterizing performance and cost of multi-tenant serving for cnn models. In *IC2E'20*, pages 66–72. IEEE, 2020.
[30] Liang, Q. et al. AI on the edge: Rethinking AI-based IoT applications using specialized edge architectures. 2020.
[31] List, N. et al. Svm-optimization and steepest-descent line search. In *Proceedings of the 22nd Annual Conference on Computational Learning Theory*, 2009.
[32] Liu, L. et al. Edge assisted real-time object detection for mobile augmented reality. In *MobiCom'19*, 2019.
[33] Liu, Z. et al. Deepn-jpeg: a deep neural network favorable jpeg-based image compression framework. In *DAC'18*, pages 1–6, 2018.
[34] Ogden, S.S. et al. MODI: Mobile deep inference made efficient by edge computing. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.
[35] Ogden, S.S. et al. Mdinference: Balancing inference accuracy and latency for mobile applications. In *IC2E 2020*, 2020.
[36] Ogden, S.S. et al. Pieslicer. https://github.com/cake-lab/PieSlicer, 2020.
[37] Olston, C. et al. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.
[38] Paszke, A. et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 2019.
[39] Ran, X. et al. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE Conference on Computer Communications*, 2018.
[40] Rayner, K. et al. Masking of foveal and parafoveal vision during eye fixations in reading. *Journal of Experimental Psychology: Human perception and performance*, 1981.
[41] Reddi, V.J. et al. Mlperf inference benchmark. In *ISCA'20*, pages 446–459. IEEE, 2020.
[42] Rice, A. et al. Measuring mobile phone energy consumption for 802.11 wireless networking. *Pervasive and Mobile Computing*, 6(6):593–606, 2010.
[43] Romero, F. et al. Infaas: Managed & model-less inference serving. *CoRR*, abs/1905.13348, 2019. URL http://arxiv.org/abs/1905.13348.
[44] Soifer, J. et al. Deep learning inference service at microsoft. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, 2019.
[45] Szegedy, C. et al. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.
[46] Teerapittayanon, S. et al. Distributed deep neural networks over the cloud, the edge and end devices. In *ICDCS'17*, pages 328–339. IEEE, 2017.
[47] Wallace, G.K. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 1992.
[48] Wu, C.J. et al. Machine learning at facebook: Understanding inference at the edge. In *HPCA'19*, pages 331–344. IEEE, 2019.
[49] Xie, X. et al. Source compression with bounded DNN perception loss for IoT edge computer vision. In *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.
[50] Xu, M. et al. A first look at deep learning apps on smartphones. In *The World Wide Web Conference*, WWW '19, 2019.
[51] Zhang, C. et al. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference*.
[52] Zoph, B. et al. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017. URL http://arxiv.org/abs/1707.07012.