Modeling Inference Time of Deep Neural Networks on Memory-constrained Systems

Hans Brouwer[1]
[1]TU Delft

Abstract

Deep neural networks have revolutionized multiple fields within computer science. It is important to have a comprehensive understanding of the memory requirements and performance of deep networks on low-resource systems. While there have been efforts to this end, the effects of severe memory limits and heavy swapping are understudied. We have profiled multiple deep networks under varying memory restrictions and on different hardware. Using this data, we develop two modeling approaches to predict the execution time of a network based on a description of its layers and the available memory. The first modeling approach is based on engineering predictive features through a theoretical analysis of the computations required to execute a layer. The second approach uses a LASSO regression to select predictive features from an expanded set of predictors. Both approaches achieve a mean absolute percentage error of 5% on log-transformed data, but suffer degraded performance on transformation of predictions back to regular space.

1 Introduction

As a result of the recent revolution in deep learning, more and more neural networks are being used for everyday tasks. While this class of algorithms is undoubtedly effective at solving problems in certain domains, like computer vision, they also require a lot of resources to run. Not only do deep networks require great computational power, they also need a large amount of memory to store the parameters for those computations. As more deep networks move from the cloud to the edge (mobile devices, embedded systems, etc.), it is becoming increasingly important to run these networks as efficiently as possible.

On this front, Mathur et al. take a comprehensive look at optimization of neural networks for a wearable camera, DeepEye [18]. This system runs multiple deep networks every 30-60 seconds, completely locally, using a framework that caches, compresses, and schedules inference to run efficiently. This provides a strong baseline for future improvements. The system has a small form factor and uses relatively low-power hardware to preserve battery life and so is very resource-constrained. The first step towards improvements is a better understanding of where systems like DeepEye are bottlenecked.

We set out to profile and model the inference time of deep networks on such CPU-only, memory-constrained systems. Implementing profiling instrumentation and gathering data is time intensive and so quite a barrier when researching or developing for resource-constrained devices. For these reasons it is desirable to be able to estimate time or memory required without actually having to go through extensive benchmarking. To this end, we build a model that can predict the time required to run a network given a description of its layers and the amount of available memory.

On systems with little RAM, execution is bound by the memory bandwidth of the system. Multi-model pipelines, like employed in DeepEye, make it impossible to keep all layers the networks in memory at once. Under severe memory limitations, the elements necessary for a single layer might not even fit in memory. In such cases these elements must be copied to and from disk during execution (swapping), which has a serious performance impact. It is important to understand this bottleneck. Using simulated memory limitations, we estimate the scaling of inference time of layers under varying amounts of required swapping. Simulating memory constraints allows comparison without side effects introduced by each memory size being associated with different hardware.

Related Work There have been multiple efforts to profile the resource requirements of deep neural networks on edge devices, although most use the data to develop faster inference engines rather than to predict inference time [8, 21, 22, 27]. Of the efforts into modeling inference time, the investigations do not model the effects of insufficient memory on the performance of deep networks. Lu et al. tackle profiling multiple popular deep convolutional networks on the NVIDIDA Jetson TK1 and TX1 GPU-accelerated modules [17, 20]. They build a model that estimates the execution time and memory usage of convolutional networks based on the floating point operations required for their layers and which device they are run on. The TK1 and TX1 modules have two gigabytes and four gigabytes of RAM respectively, and so are able to fit large networks in memory completely. This means the effects of memory swapping and loading of individual layers are not modeled. Furthermore, the model is not made publicly available for actual application.

Another approach to modeling the inference time of deep networks on mobile devices is presented by Yao et al. [27], called FastDeepIoT. FastDeepIoT uses a tree-structure linear regression model to analyze the execution profile of deep

networks at runtime to optimally compress individual layers. The tree-structure linear regression model recursively separates measurement data into sets where a certain non-linear effect is present or not. The data in the leaves of the binary tree have minimal non-linearity with respect to a chosen set of explanatory variables and so can be modeled with a linear regression. FastDeepIoT is investigated on commercial mobile phones, the Nexus 5 and Galaxy Nexus. These phones have two gigabytes and one gigabyte of RAM respectively. Once again, this is sufficient memory and so the model does not need to account for swapping or individual layer loading.

Our objective is to build a model to predict the execution time of several important types of layers of deep neural networks under different memory constraints and on different hardware. To this end, first the resource requirements of different deep networks and their component layers are measured under varying memory limitations and then a model is built to predict their total execution time. This entails (1) determining the memory requirements of each of the layers of the networks, (2) determining the execution times of each of these layers under different memory constraints, and (3) creating a model to predict the execution time of new networks based on descriptions of their layers and available memory.

We employ two approaches to model the execution times of deep networks. The first entails engineering predictive features based on the computations layers require. This matches the approach introduced by Lu et al. The second selects predictors based on a LASSO regression, this is closer to the approach of FastDeepIoT. Instead of modeling the non-linearity in inference time with a binary tree, we expand the set of input features with higher order polynomial functions of the predictors.

2 Profiling

This section outlines the methods used to gather execution time and memory usage of layers. While the exact details are specific to the profiled system, similar procedures can be applied to any hardware or framework.

2.1 EdgeCaffe

The basis for our investigation is EdgeCaffe, which uses the C++ and Python based deep learning framework, Caffe [4]. EdgeCaffe provides a platform for the efficient execution of deep neural networks on low-resource hardware. EdgeCaffe implements a pipeline that runs several deep networks with multiple different strategies for scheduling, loading, and executing the layers of each network. It is also easily extensible to allow for new networks to be added. An in-depth understanding of the performance of different networks when executed with EdgeCaffe as well as an estimate of the amount of time a network will take can enable further research and development of deep networks on resource-constrained devices.

EdgeCaffe executes networks by splitting up their layers into a list of individual jobs with dependencies. Layers can be scheduled for execution to one or multiple threads according to their type, the task (loading, executing, or unloading), expected resource requirements, or simply their order in the network. Each layer's parameters are loaded individually. Once all the required inputs are available the layer is executed and finally unloaded.

EdgeCaffe is a good target for investigation of execution performance due to its extensive capabilities. EdgeCaffe's flexibility makes it an excellent framework for developing high performance deep network pipelines for resource-constrained environments. Profiling instrumentation and execution time estimation enhance EdgeCaffe's utility for researchers and developers alike by providing insight into the execution of their deep networks.

While EdgeCaffe supports many different execution and scheduling modes, for the purpose of building a model to predict the execution times of networks, only the "linear" mode was considered. In linear mode, all layers are executed on a single thread one-by-one. First the parameters are loaded, then the layer is executed, and then it's parameters are unloaded before starting the process anew for the next layer. This ensures that measurements are specific to single layers and are not influenced by asynchronous effects that are possible in multi-threaded modes.

2.2 Memory Usage

Measuring memory allocation is crucial to understanding and improving a program. However, a major problem with profiling memory is that the instrumentation itself has performance impacts which can skew results. This effect is especially strong on memory- and compute-constrained devices. Two main strategies were investigated to measure the memory requirements of layers. One asynchronous–checking the operating system's log of resource usage–and one synchronous–overriding the allocation functions of C++.

The asynchronous method relied on the Linux kernel's memory statistics. The EdgeCaffe pipeline's process was periodically polled from a separate thread to track the resident set size (RSS). The RSS is the amount of RAM currently in use by the process. However, this method of measuring memory only provides an approximation of the true usage. Being asynchronous, this strategy can miss high frequency events which happen between reads by the other thread. In general, it will therefore underestimate peak usage of the program.

To refine the measurements synchronously, the pipeline was compiled with Google PerfTools [6] (GPerfTools) instrumentation enabled. GPerfTools links the binary with a memory allocation library, tcmalloc [7], that keeps track of extra information whenever allocations are made. Then, when executing the binary, the GPerfTools heap profiler dumps information about the allocations made at set intervals (e.g. every 100 MB allocated). These heap dumps can then be analyzed after the fact and linked to the layers of each network that allocated the memory. There are some challenges linking exactly which layers allocated memory contained in each heap dump. The interval at which logging occurs is not synchronized per layer. While the allocation interval can be set small enough to alleviate this issue, the incurred overhead is considerable.

Both strategies are useful in different circumstances. The maximum memory usage per layer measured by GPerfTools only needs to be measured once and can be used by the EdgeCaffe scheduler to determine which layers can be loaded into memory concurrently. Asynchronous measurement of the process's RSS allows for low-overhead measure-

ment of the pipeline in multi-threaded execution modes to gauge actual memory loads of the more lightly-instrumented program.

The results from GPerfTools were used to validate the memory availability estimates in our experiments. This is because swapping does not occur when memory is reserved (which RSS measures), but when it is allocated. Therefore, GPerfTools' measurement gives a better indication of whether swapping is occurring.

2.3 Execution Time

Profiling execution time is relatively straightforward compared to profiling memory usage. Each task (loading, executing, or unloading a layer) starts a timer just before execution and stops it immediately after finishing. The use of EdgeCaffe's linear mode ensures that there is only one task executing at a time and so no extra logging is required to determine overlap or order of tasks as would be necessary in multi-threaded execution modes.

To simulate differing memory restrictions, the EdgeCaffe pipeline binary was run in a Linux kernel Control Group (cgroup) [19]. Cgroups provide a hierarchy under which groups of processes are executed whose resource usage is tracked by the Linux kernel. Limits on the amount of allowed memory can be set for each cgroup which forces the program to use swap space when exceeding the limit of allowed RAM. Therefore, executing the pipeline in cgroups with different memory limits allows for measurement of the scaling of execution time with respect to available memory.

3 Modeling Execution Time

3.1 Preliminary Analysis

To motivate the models outlined in this section, we first give a preliminary analysis of the execution profiles of the most important types of layers in deep networks. Seven different types of layers are taken into consideration: fully-connected, convolutional, activation, normalization, pooling, and dropout. The chosen types account for 99% of the layers in the networks that were investigated, with the remaining being layers that weren't present in all networks (e.g. crop, scale, deconvolution) or Caffe routing layers which perform negligible computation (e.g. split, input, concatenate). The model we present relies on the assumption that the vast majority of layers in the network whose time is to be predicted are of the seven modeled types.

Table 1: Required memory, execution times, and loading times of selected example layers under varying memory limits.

Task	Layer	Required	Time (ms) under Memory Limit				
			5 GB	512 MB	372 MB	256 MB	128 MB
LOAD	FC1	163 MB	12	12	12	12	20
LOAD	FC2	656 MB	77	121	136	127	144
LOAD	CONV1	238 MB	51	53	52	85	77
LOAD	CONV2	654 MB	94	135	135	139	140
EXEC	FC3	375 MB	733	714	1268	1716	3454
EXEC	FC4	1832 MB	95	2264	4572	7133	7908
EXEC	CONV3	230 MB	11	23	22	103	101
EXEC	CONV4	582 MB	18	51	64	65	77

Figure 1 shows the distribution of time needed to load and execute each type of layer under different memory limits. The general pattern in both loading time and execution time is that as the memory limit decreases the time taken increases. However, the strength of the effect varies per layer type. For most layers, the increase is greatest for the layers that take the longest (this can be seen by the larger relative difference in upper whiskers compared with the medians). Some layers see no increase in execution time with tighter memory limits, for example, pooling and activation layers.

The increase in time taken is related to the amount of memory a layer requires to perform its computation. Once a layer needs more memory than is available, it must swap out values during execution which reduces performance. The memory layers need is equal to the sum of the number of input elements, number of parameters, and number of output elements of the layer. We call the elements that are forced to swap the "overflow" (denoted by F).

Table 1 shows example layers and their execution times under different memory limits. As soon as the available memory is less than the required memory, the execution time increases considerably. This observation motivates a model which takes into account the different regimes that scaling occurs in: sufficient memory and insufficient memory.

We employ two strategies to model the execution times of deep networks under these regimes: one taking a feature engineering approach based on the theoretical computations

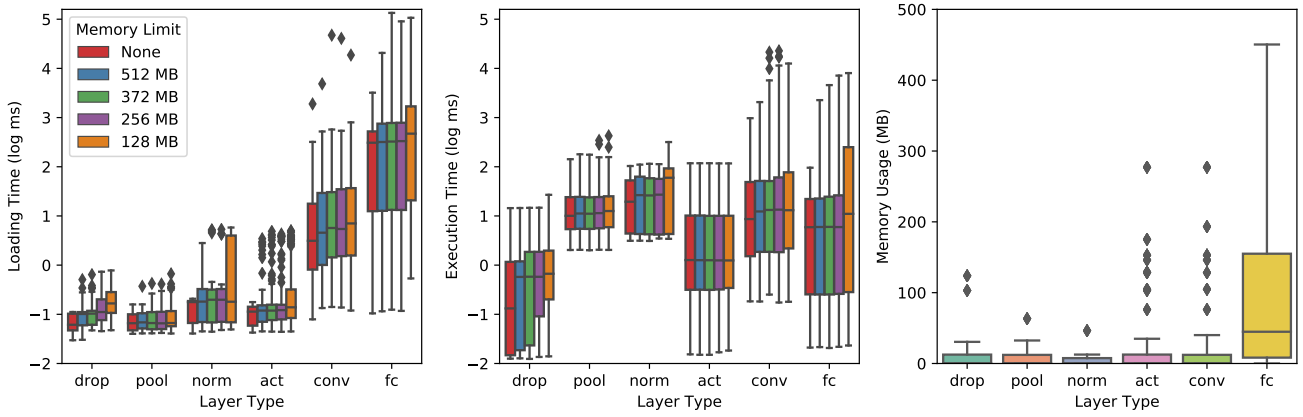


Figure 1: Distribution of the loading times (left) and execution times (center) for each of the seven modeled layer types under different memory limits. Distribution of the required memory per layer (right).

layers require, and the other taking a feature selection approach based on a LASSO regression. While the data the models are trained on is specific to EdgeCaffe, the models can be applied to similar data from any framework or hardware.

3.2 Feature Engineering

The feature engineering model makes the assumption that execution times scale linearly with the amount of computations that are needed for the layer. When there is insufficient memory, some of the calculations require swapping of the data. This has the effect that this portion of computations have an additional overhead from having to first swap their data into memory. This implies that a linear model with two parts can be used to estimate each layer's execution time– one part for scaling under sufficient memory and one part for steeper scaling under insufficient memory.

The input features to the models are all calculated from the network description file (.prototxt). This ensures that any network's execution time can be estimated (although the accuracy of the estimation still relies on the fact that the types of layers of the network and their settings are similar to those that we model).

The seven layer types outlined earlier, as well as a catch-all *other* type, are each fit with a separate regression for execution. One loading time regression is made for all layers with parameters. Layers without parameters are estimated by their mean loading time (the time it takes for EdgeCaffe to execute an empty loading task). The input features to each of these regressions are detailed in the following sections.

Loading Time To load a layer, every parameter needs to be loaded and all are assumed to each take the same amount of time. Therefore, a simple linear model suffices as long as all the components required for calculation fit in memory. If they do not fit in memory, data must be swapped to disk during loading. Thus, the time taken for parameters that need to swap will be greater. The layer's loading time will then be greater relative to the overflow.

Execution Time: Fully-Connected Layers Execution of fully-connected layers entails a multiplication of the input features with the matrix of parameters and the addition of a bias to the result. Matrix multiplication scales with the product of the dimensions of the matrices. We model the case in which inference is run on only a single image at a time, which means the first dimension is always one. In this case, the matrix multiplication scales with the number of weights to be multiplied (i.e. the number of parameters of the layer). Once again, this scaling must be split into two regimes: without swapping and with swapping. When the values needed for the calculation do not all fit in memory, there is an extra factor that scales with the overflow. As with loading time, there is a minimum time required to execute a layer even when it requires no real calculation. In this case the overhead from EdgeCaffe executing a task dominates the time taken.

Execution Time: Convolutional Layers The execution time of convolutional layers depends on more than just the number of parameters. The output shape, number of channels, kernel size, stride, padding, and grouping all play a role [5]. Lu et al. [17] present a formula that calculates the number of multiplications needed to calculate the output (floating point operations), however they do not account for channel grouping, which is present in some of the networks that were investigated. Extending the formula to account for channel grouping and bias terms gives the following equation:

$$\#\,\text{FLOPS} = \underbrace{O_D O_H O_W}_{\text{output volume}} * (\underbrace{P_D P_H P_W}_{\text{parameter volume}})^2 / G + \underbrace{O_D}_{\text{bias}} \quad (1)$$

Where O is the output, P the convolution filter kernels (parameters of convolutional layer), G the number of channel groups, and the subscripts D, H, and W representing the depth, height, and width respectively. # FLOPS represents the total number of floating point operations needed to calculate the output.

Execution of convolutional layers under insufficient memory is also more complicated than fully-connected layers. Convolutions require extra memory to store intermediate results which vary per algorithm used. This means, even if the input features and the convolution's parameters fit in memory, the intermediate results might incur swapping. To simplify the model, memory used by these intermediate calculations is neglected. This means the convolutional model will perform worse on layers which are forced to swap by their intermediate calculations, as these are not factored in to the overflow of the layer.

Execution Time: Remaining Layers The execution of the remaining layers only account for less than 15% of the total execution time and so are approximated more roughly. All of these layers (activation, pooling, dropout, and normalization) apply a static operation to their inputs. Therefore, they are modeled with the number of input elements as predictive feature. The *other* category of layers is also modeled by number of input elements as most operations scale at least linearly with their input. This model will underestimate any operations that scale with higher orders with respect to their input.

Altogether, the feature engineering model estimates the total time to execute a layer as the sum of a loading term that scales with the number of parameters and an execution term which scales with number of parameters, number of multiplications, or number of input elements, depending on the layer type. Both the loading and execution terms for all layer types have a component which scales with the overflow of the layer.

The feature engineering model was trained with an ordinary least squared regression, both with bootstrap aggregating and without. Bootstrap aggregating is a process where the model is trained repeatedly on a subset of the data sampled with replacement. After all models have been fit, the average of the regression coefficients of all models is taken. This helps make the model more robust to outliers in the data by biasing its predictions towards the most frequently occurring values.

3.3 Feature Selection

The second approach to modeling execution time that was investigated was a feature selection approach. Rather than fitting a model on a select few features, instead, all the gathered data was used such that the model would find the most predictive features itself. The model used for this was a

LASSO regression [25]. LASSO regression is the same as an ordinary least squares regression but with a regularization term added that penalizes the size of the regression's coefficients. This added term makes solutions with coefficients that are zero more preferable to solutions with small coefficients–essentially forcing terms that do not contribute to be dropped.

The input features to the feature selection model were once again calculated from each network's .prototxt file. The basic features included the volume of each layer's inputs, parameters, and outputs, the number of parameters that do not fit in memory under a given memory constraint, and for convolutional layers, also the group, stride, and padding parameters. To allow for better modeling of the difference between swapping layers and non-swapping layers, a binary feature was added that was one when the layer needed swapping and zero when it did not.

Expansion The enforced sparseness of LASSO regression can be exploited further. From the feature engineering analysis, we believe that some nonlinear combinations of the features mentioned above should be predictive of the execution times (e.g. the output volume times the parameter volume squared as in Equation 1). To allow the model to use these combination features as well, the input data was expanded by taking the products of all combinations (with replacement) of the features. Taking combinations of two allows for quadratic products of features, combinations of three allows for cubic products, and so on. This greatly expands the feature space that the regression can select from.

However, one drawback of this expansion is that it introduces many new, highly correlated input variables. Assuming LASSO is able to select meaningful variables despite any introduced inter-correlation and co-linearity, regression will still take much longer due to the large number of variables. To alleviate these correlation issues and reduce the number of variables, the regression can instead be trained on the principle components of the expanded data. This way, the expanded data can be distilled back down to a manageable set of orthogonal features. Intuitively, expanding the data amplifies the variance most in the feature directions that are the best predictors. The resulting principle components will then align along these directions and so be better inputs to the regression than the principle components of the original data.

4 Evaluation

4.1 Experimental Setup

Results were obtained on two Ubuntu machines. Their full specifications can be found in Appendix A.1. This allowed for a cursory examination of the effects of different hardware configurations on execution time.

EdgeCaffe already had five networks from DeepEye [18] implemented in the framework. To increase the amount of layers that could be profiled and modeled, seven more networks from the EdgeCaffe Model Zoo [3] were added. These were selected to include as wide a range as possible of layer sizes, settings, and types. The more varied the training set, the better the time estimation models could generalize. The networks used are: Salient Object Subitizing [28], Salient Object Subitizing GoogleNet [28], AgeNet [13], GenderNet [13], FaceNet [23], VGG-16 [24], VGG-16 Channel Pruning [10], VGG-19 [24], SimpNet [9], FCN32s [16], VocNet [12], and Network in Network [15]. The VGG-16 and VGG-19 networks are two of the largest in the Model Zoo by parameter count, while the Channel Pruning version, SimpNet, and Network in Network are specifically aimed at reducing parameter count. The rest of the networks fall somewhere in between.

Execution times were recorded by running each of the networks on the same image thirty to fifty times (depending on the machine) for each memory limit. The memory limits the pipeline was run under include five gigabytes (unconstrained), 512 megabytes, 372 megabytes, 256 megabytes, and 128 megabytes. These levels were based on the baseline in-memory size of the networks. At 512 megabytes, four of the networks are under serious memory pressure, at 256 megabytes eight are, and at 128 megabytes they all are.

Resident set size (RSS) measurements were averaged over ten runs of the pipeline. The measurement instrumentation increases an unconstrained run of the pipeline on Machine 2 (see Appendix A.1) from about 1 minute 20 seconds, to around 2 minutes. GPerfTools was deterministic with respect to measured memory and each measurement at a given allocation interval gave the same results. Memory usage was recorded with an allocation interval of one kilobyte, this was a small enough interval that there was a measuring point for all but the smallest layers (which were approximated by interpolation of their neighboring layers' measurements). GPerfTools balloons the total time to run the pipeline to 15 minutes, after-which the heap dumps still need to be processed for an additional 15 minutes.

4.2 Results

Memory Measurements The results from memory measurements as described are shown in Figure 2. Of note is that the allocated memory as measured by GPerfTools is lower than the RSS of the process reported by the kernel. This is most likely because GPerfTools measures the actual allocated chunks of memory, not the amount of memory reserved. The RSS measures the amount of physical memory reserved by the process, not what is actually allocated.

The memory requirements measured by GPerfTools were used to validate the overflow estimations (which were calculated from the network's .prototxt file). Note that the over-

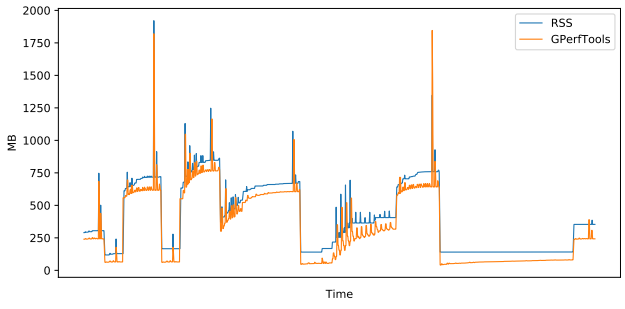


Figure 2: Asynchronous measurement of the resident set size (RSS) of the process and the memory allocated by tcmalloc (as measured by GPerfTools) over the course of execution of the EdgeCaffe pipeline for all twelve networks.

flow can be converted to memory values simply by multiplying by size of the data type (e.g. for single precision floating point numbers: 4 bytes). The mean absolute percentage error between the estimation and the measured memory value was 17%. The estimation tended to under-predict the value measured by GPerfTools (the mean percentage error was -2.5%). This underestimation of the memory could reduce the accuracy of the time prediction models by misclassifying a layer as not requiring swapping when it does actually require swapping. In total only 1.12% of measurements were misclassified. As the overflow value of these layers would be relatively small, this effect on the model accuracy was neglected.

Time Measurements The recorded times had a high variance so the upper and lower 25% of times were filtered out to curb noise. Even so, the standard deviation of the time measurements was 9-10% of the total time on average.

Bootstrapping the time measurements was tried, however, both sampling with replacement from the measurements and sampling from a kernel density estimation of the measurements were not effective. While the standard deviation of the data decreased considerably, the predictive performance of the models decreased sharply and the data was noisier on inspection. The time of some layers was drastically different after bootstrapping while others were more or less unaffected. In the end the decision was made to use the median of the measurements.

Feature Engineering Figure 3 shows the feature engineering model's predictions for the loading time of layers with and without parameters, the execution time of fully-connected and convolutional layers, and the measured times for all layers under different memory limits. These four categories of predicted time account for more than 85% of the total time of all networks. The spread of the data with respect to the predictive variables chosen for each layer is high. Both the swapping data and the non-swapping data exhibit vertical patterns where measurements of the same layer or different layers with the same predictor variable values had large differences in measured time. In the swapping regime, the model underestimates the execution time of many of the convolutional layers as well as the loading time of layers that do not have parameters. Interestingly the scaling with the overflow of the fully-connected layers Figure 3c is the opposite of expected. Most likely, the effects of the vertical patterns dominate the expected increase in execution times caused by the swapping of the overflowed elements.

It is clear that the simple bi-linear feature engineering model does not capture all the variation in the time measurements. This highlights the need for a model that takes more features into account.

Feature Selection The feature selection model was able to compare the predictive power of a much wider range of features compared to the manual feature engineering approach. This allows the model to find more complex relationships in the data.

An example of a model from cubic expansion is compared with the feature engineered model in Table 2. Notable is that the overflow term, F, is present in every layer's model. In

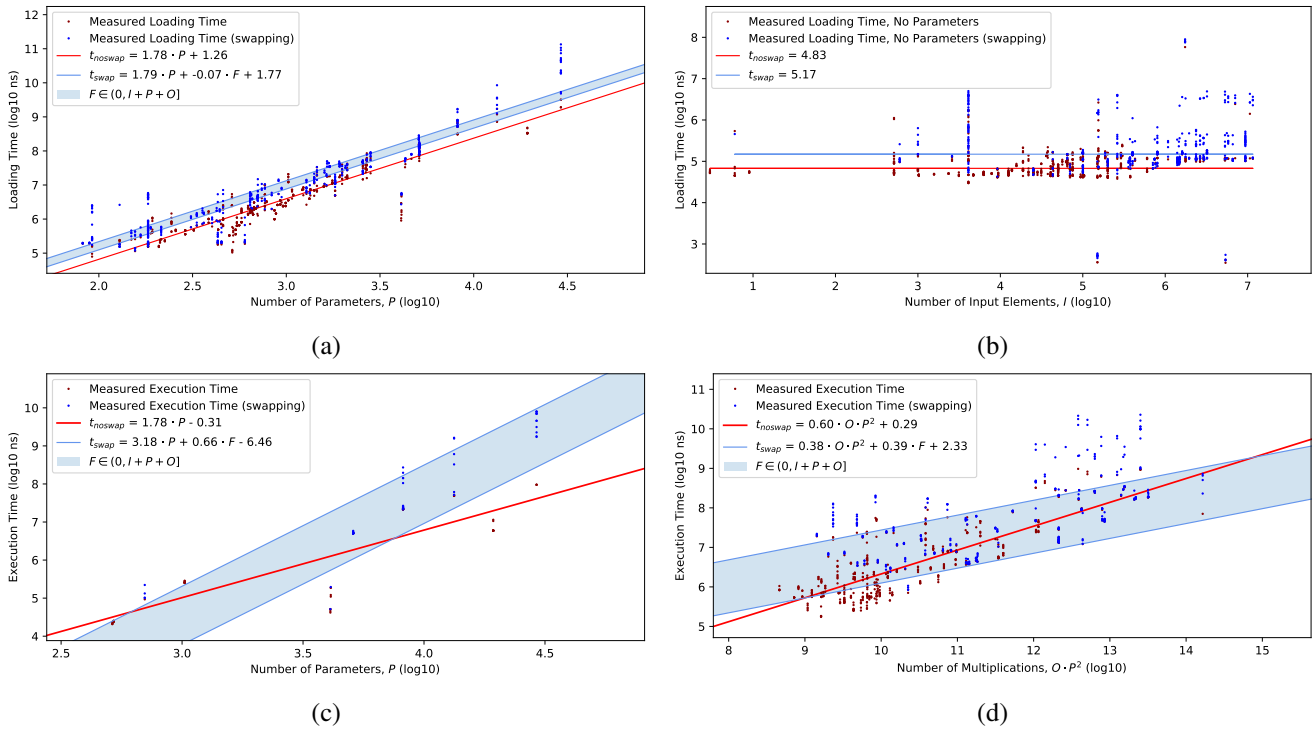


Figure 3: Linear regressions of the feature engineering model on the measured times of fifty runs of the EdgeCaffe pipeline on Machine 2 (see Appendix A.1). (a) The loading time of layers with parameters versus their number of parameters and memory overflow. (b) The loading time of layers without parameters versus their mean loading time. (c) The execution time of fully-connected layers versus their number of parameters and memory overflow. (d) The execution time of convolutional layers versus their number of multiplications and memory overflow. Graphs of the remaining layers can be found in Appendix A.2.

Table 2: Left: the model found with the model selection approach with cubic data expansion. Right: the feature engineering model. F stands for overflow, the total number of elements that a layer needs that do not fit in memory, I for the number of input elements to a layer, P for the number of parameters (with P_0 denoting zero parameters), and O for the number of output elements.

Example Feature Selection Model

type	terms	intercept
LOAD	$0.013\ F^3$ $+ 0.217\ P^3$	3.962
LOAD P_0	$0.005\ F^3$	5.391
FC	$0.030\ F^3$ $+ 0.079\ I \cdot O^2$ $+ 0.235\ O^3$	3.686
CONV	$0.063\ F^3$ $+ 0.109\ O^3$	7.084
ACT	$0.024\ F^3$ $+ 0.166\ I^3$	3.404
POOL	$0.010\ F^3$	6.593
DROP	$0.010\ F^3$ $+ 0.142\ I^3$	4.848
NORM	$0.016\ F^3$ $+ 0.036\ I^3$	4.790
OTHER	$0.032\ F^3$ $+ 0.083\ P^3$ $+ 0.026\ O \cdot P^2$ $+ 0.017\ O^2 \cdot P$ $+ 0.009\ O^3$	5.030

Feature Engineering Model

type	terms		intercept	
	standard	*swapping*	*standard*	*swapping*
LOAD	$1.777\ I$	$1.786\ I$ $+ \text{-}0.070\ F$	1.264	1.766
LOAD P_0	-	-	4.829	5.172
FC	$1.775\ P$	$3.183\ P$ $+ 0.660\ F$	-0.313	-6.461
CONV	$0.603\ O \cdot P^2$	$0.376\ O \cdot P^2$ $+ 0.394\ F$	0.288	2.326
ACT	$0.843\ I$	$0.825\ I$ $+ 0.034\ F$	1.885	2.068
POOL	$0.518\ I$	$0.779\ I$ $- 0.120\ F$	4.173	2.948
DROP	$0.590\ I$	$0.488\ I$ $- 0.372\ F$	2.480	4.380
NORM	$0.604\ I$	$0.785\ I$ $- 0.192\ F$	3.722	3.166
OTHER	$0.152\ I$	$0.220\ I$ $+ 1.644\ F$	4.416	0.395

general the features used are very sparse–using only two features or three features of the 222 options. Notably, the multiplications feature introduced in Equation 1 was found by the feature selection model, although it was used for the *other* category rather than the convolutional layers. The *other* category consistently had the most terms, which suggests that more information was needed to model it. This seems logical as multiple disparate layer types were lumped into this category.

The models fit on the raw expanded data did not consequently find the same features as most predictive. LASSO seems to favor the maximum order features. Models of a given order expansions all use features exclusively of the highest order, despite also having the lower order features as input. This could be because these features have the highest absolute values and so are driven to zero more slowly or some other property of LASSO. In any case it seems undesirable as the model is not weighting features equally and is possibly overfitting. Even repeated fittings to the same expanded data did not give identical predictors. This suggests that the feature selection process is not objectively determining the true underlying explanatory variables, further motivating reducing the expanded data to their principle components.

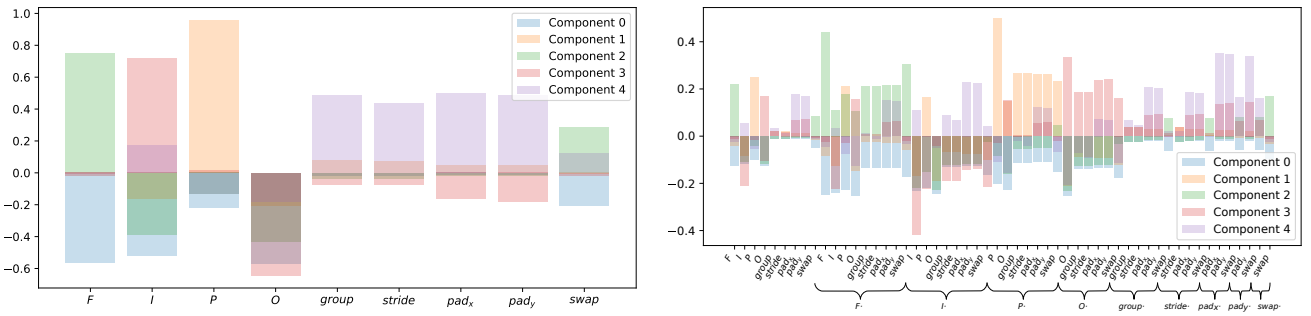
Principle components whose explained variance percentage was greater than 1% were fit with a LASSO regression. There tended to be five or six components regardless of the expansion order of the data. Figure 4 shows the principle components of the data with no expansion and quadratic expansion. In both cases the first principle component is related to the overflow and the second related to the parameter volume. The principle components align with intuitively important axes in the data. Sparse PCA was also tried, however the principle components were not much sparser and so the slight interpretability gain was not worth the degraded performance.

Layer models favored principle components which were related to the overflow (e.g. Component 0 and Component 2 in Figure 4). Some layers also had parameter volume related terms (Component 1 terms). Component 3 and 4, which were mainly related to convolutional settings, were not used, even in the convolutional layer model. This suggests that despite the principle components distilling the variance in the expanded data, the LASSO regression was not able to use the PCA-transformed predictors to better capture the dynamics of their effect on inference time.

4.3 Model Evaluation

Models were all trained on log-transformed data. This was due to many of the input features and time measurements spanning multiple orders of magnitude. Without the log-transform, models were unstable and did not converge. Even scaling by mean and standard deviation or by median and inter-quartile range did not allow the same numeric stability that the log-transform provided. This creates a major performance issue with the model predictions on transforming out of log-space. The regression assumes downward and upward variation cost the same, however, any underestimations in log-space are transformed to massive errors in regular space. This means all the models underestimate time taken when transformed, sometimes by orders of magnitude. For this reason, all statistics are reported in log-space for the purpose of comparing model performance.

Model performance across the variety of different settings investigated was evaluated using k-fold cross-validation.



(a) (b)

Figure 4: The magnitude of the principle component vectors along each axis in the (a) raw or (b) once expanded data. The braces in (b) indicate that the parameters listed above the brace are all multiplied by the parameter listed below it.

Folds were chosen such that one, two, or three networks were left out as test sets (corresponding to k sizes: 12, 6, and 4). While the variance in performance was larger with the leave-one-out folds due to some networks being very different from the rest (e.g. Channel Pruning using non-square convolution kernels or GoogleNet having 142 layers), results from all folds were pooled to get a more complete picture of the models' performance under different train and test set sizes. Results are reported averaged over 36 folds, 12 of each k size.

The measure used to gauge performance was the mean absolute percentage error (MAPE). A low score on this metric shows that the model's estimate is close to the true time taken to execute, which is the most valuable property in practice. Figure 5 shows the mean MAPE with 95% confidence interval over the cross-validation of the feature engineering model with different bootstrap sizes and of the feature selection model for different expansion orders. The best performing model was the expansion order 6, PCA based feature selection model with an average MAPE of 5.365. However, after expansion order 4 there were diminishing returns and all models with order greater than 3 were not significantly different from each other or the feature engineering approach.

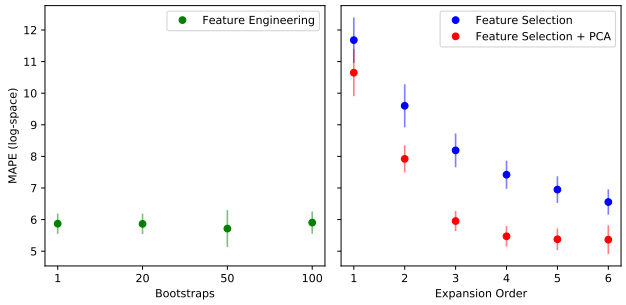


Figure 5: Left: the mean absolute percentage error of the feature engineering model averaged over 36 cross validations for different numbers of bootstrapped models. Right: the mean absolute percentage error of the feature selection model for different levels of iterative expansion of the predictors, trained on the raw expanded predictors or the principle components of the expanded predictors.

4.4 Hardware Considerations

To evaluate the effect of different systems, the fitted feature engineering models from two machines are compared (full specifications of both machines can be found in Appendix A.1). The feature engineering models are chosen for this test as these always have the same parameters, while the feature selection models sometimes do not have comparable terms.

Machine 1 is less powerful than Machine 2 and so only thirty runs of the pipeline could be collected (compared to Machine 2's fifty runs). Machine 1, however, could be run with no other processes competing for resources. Machine 2 is a multi-tenant machine and so runs could not be collected in complete isolation. However, the other jobs were mainly GPU bound and Machine 2 has twelve cores to share.

Due to the small amount of samples and the non-normality of the distributions of the parameters (as found by inspection of their histograms), the non-parametric Wilcoxon test [26] is used to analyze the significance of the differences in the models. The null hypothesis is that the means of the distributions of fitted model parameters over 100 bootstrapped models are the same. The p-values are corrected for multiple hypothesis testing using the Benjamini–Hochberg Method [2]. The test finds that all parameters of the model are significantly different with most p-values under 0.01. This means that similar measurements should be performed for a wider range of machine specifications to gain an understanding of the scaling variance with respect to hardware.

5 Discussion

Despite their differences, the predictive performance of the feature engineering and feature selection models is not significantly different (for expansion orders of greater than three in the feature selection case). However, the feature engineering model seems the more stable choice of the two given its straightforward theoretical foundation. It is unclear why the expanded predictor set does not allow the LASSO model to find solid underlying predictive features. However, the bias towards the highest polynomial order features suggests that the data expansion approach might be causing the regression to overfit. Perhaps improved LASSO approaches like overlapping or group LASSO could be tried. These are specifically designed for cases where there are multiple potentially overlapping groups of predictors.

While the feature engineering model does seem to capture the majority of the variance in the log-space data (the R^2 values of the best models were around 0.8), there are still clearly some points where the model can be improved.

First of all, approaches to alleviate the severe underestimation on transformation back from log-space can be investigated. Standard linear regression assumes the error with respect to the linear model is normally distributed. However this assumption is violated when fitting in log-space as two points equidistant to the line in log-space are not equidistant to the line in the original space. Generalized linear models could be applied to model an exponential/lognormal error distribution in log-space. This way the error distribution would be normal when transformed back to regular space. Alternatively, a way to bias the log-space regression's objective to minimize underestimation could be found.

Second, a better estimate of the memory requirements of convolutional layers could be found. Convolutional layers make up 35% of total execution time, but in the swapping regime the feature engineering model only achieves an R^2 of 0.4. Despite using the same feature to model the execution time of convolutional layers as Lu et. al [17], we were not able to achieve a similar accuracy (Lu et al. reported around 80% accuracy). This suggests that the effects of swapping dominate the time needed floating point operations. These effects are estimated with respect to the overflow value which we know is not correctly estimated for convolutional layers due to the neglection of the extra intermediate memory these layers use.

Finally, the vertical patterns in the feature engineering data suggest that there might be some stochastic or strongly non-linear effects at play. One possible explanation is that in the swapping regime, the loading time is dominated by effects related to contiguity of memory space and other caching effects. Some layers of a given size are lucky

to have a favorable block of memory evicted and so load faster, while others are scattered throughout memory over the course of multiple evictions. As the feature selection model does not seem to be able to capture these non-linear effects, the tree-structured linear regression used by FastDeepIoT [27] seems to be a more promising direction for improvement. This approach could be extended to include the overflow value or other memory based parameters as predictors.

Another important way forward is to gather more time measurements. This could give a better picture of performance under different circumstances. More runs of the pipeline in isolated environments could help reduce the noise in the data and make clear trends that are currently obscured. More intelligent setting of memory limits during profiling could help improve the interpolation of some patterns in the dataset. The currently used levels are based on the baseline memory sizes of the networks, but a better strategy could be to subject each individual layer to the same levels of overflow. Also, multi-threaded scheduling modes could be profiled to gain an understanding of the interaction between different kinds of layers with different bottlenecks being run at the same time. More systems with a wider range of resource-constraint can be investigated–specifically low-resource, CPU-only devices. This way a hardware-agnostic model could be built based on features of the hardware like CPU clock speed and maximum memory speed.

Memory-aware scheduling strategies like those introduced by Li et al. or Abraham et al. could be profiled as well [1, 14]. Both of these strategies use adaptive algorithms to minimize memory use. Li et al. introduce a control module that splits layers into independent partial operations. These more granular jobs can then be scheduled to use the memory bus more optimally. Abraham et al. focus on analysis of memory buffers to allocate memory in a way that optimizes for reuse of buffers. Analyzing these minimal memory bandwidth scheduling strategies with the models introduced in this paper could help build intuition into the reason for their efficacy. The models can provide a distilled representation of the scaling of different parts of inference under these different strategies that might be easier to interpret and compare.

6 Responsible Research

Ethical Concerns There are many concerns regarding the use of automated systems to gather information or make decisions, especially when they involve artificial intelligence. Machine learning systems are generally harder to probe for how they come to a decision due to their black-box nature. Even when systems are not making decisions that directly impact people, machine learning systems have unprecedented capabilities to aggregate and parse information. This information can influence decisions and so it is important to consider the impacts any research into these systems can have.

While profiling networks in resource constrained contexts seems innocuous on a surface level, there are many applications where improvements in performance might be detrimental. One conspicuous example is Clearview AI, a company that has crawled vast amounts of online pictures of public and private spaces and provides an app that allows users to upload a picture of a person to receive more information about them [11]. The app even includes code that allows its use with augmented reality software such as might be present in a wearable device. While Clearview AI might not run any facial recognition software on-device, more performant devices, such as DeepEye, as well as other improvements brought about by better profiling capabilities could expand the reach of currently internet-bound apps to places that are yet out of reach. More generally, augmenting low-resource systems (e.g. security cameras or WiFi routers) with machine learning capabilities can systematically erode the privacy in ever-more places.

Next to privacy concerns, profiling research can also exacerbate issues with biased machine learning systems. A more performant, low-resource risk assesment model in a police officer's augmented reality glasses could needlessly escalate situations according to biased trends in its training data, further perpetuating that bias.

Epistemic Concerns While being aware of the ethical aspects of research is important, it is also necessary to consider some epistemic concerns. The profiling methods employed in this paper use free, open-source tools whose manuals have extensive information on how to use them. The explanation in the methodology is tailored such that, together with the manuals of the cited tools, any source code can be instrumented and measured in a similar way.

EdgeCaffe builds on the strong open-source foundation of the deep learning framework Caffe [4]. Also, all the models used to build the execution time model are pre-trained networks that can be found for free online (mostly in the Caffe Model Zoo [3]).

On top of this, the EdgeCaffe source code will be published and so the exact code used to measure the results in this paper will be available; including documentation on how to reproduce the experiments.

7 Conclusions

We have presented two models that predict the total time needed to execute individual layers of deep neural networks under different memory restrictions. Data for the models was gathered by instrumenting the EdgeCaffe framework's pipeline to measure the required memory and time needed for execution of multiple networks under various memory limits. One model predicts the time needed to execute a network and its component layers based on the theoretical computations required. The other can select predictive features on its own from a wide range of combinations of inputs. While both models generally underestimate the actual inference time needed, they correlate strongly with the log-loading and log-execution time of several important types of layers. This shows that estimations of the number of computations that can be performed with and without swapping are an important part of understanding the performance of deep networks in memory-constrained systems. Future work can use the distilled representations of our per layer inference time models to shed light on the performance of different network architectures and scheduling algorithms.

References

[1] Arun Abraham, Manas Sahni, and Akshay Parashar. "Efficient Memory Pool Allocation Algorithm for CNN Inference". In: *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC). Dec. 2019, pp. 345–352. DOI: 10.1109/HiPC.2019.00049.

[2] Yoav Benjamini and Yosef Hochberg. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 57.1 (1995), pp. 289–300. ISSN: 2517-6161. DOI: 10.1111/j.2517-6161.1995.tb02031.x. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1995.tb02031.x (visited on 06/10/2020).

[3] *BVLC Caffe Model Zoo*. URL: https://github.com/BVLC/caffe/wiki/Model-Zoo (visited on 05/04/2020).

[4] *BVLC/Caffe*. Berkeley Vision and Learning Center, May 10, 2020. URL: https://github.com/BVLC/caffe (visited on 05/10/2020).

[5] Vincent Dumoulin and Francesco Visin. "A Guide to Convolution Arithmetic for Deep Learning". In: (Jan. 11, 2018). arXiv: 1603.07285 [cs, stat]. URL: http://arxiv.org/abs/1603.07285 (visited on 05/31/2020).

[6] *Google PerfTools*. URL: https://github.com/gperftools/gperftools.

[7] *Google Tcmalloc*. Google, Apr. 29, 2020. URL: https://github.com/google/tcmalloc (visited on 04/30/2020).

[8] Tian Guo. "Cloud-Based or On-Device: An Empirical Study of Mobile Deep Inference". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018 IEEE International Conference on Cloud Engineering (IC2E). Orlando, FL: IEEE, Apr. 2018, pp. 184–190. ISBN: 978-1-5386-5008-0. DOI: 10.1109/IC2E.2018.00042. URL: https://ieeexplore.ieee.org/document/8360327/ (visited on 06/20/2020).

[9] Seyyed Hossein Hasanpour et al. "Towards Principled Design of Deep Convolutional Networks: Introducing SimpNet". In: (Feb. 17, 2018). arXiv: 1802.06205 [cs]. URL: http://arxiv.org/abs/1802.06205 (visited on 06/09/2020).

[10] Yihui He, Xiangyu Zhang, and Jian Sun. "Channel Pruning for Accelerating Very Deep Neural Networks". In: (Aug. 21, 2017). arXiv: 1707.06168 [cs]. URL: http://arxiv.org/abs/1707.06168 (visited on 06/09/2020).

[11] Kashmir Hill. "The Secretive Company That Might End Privacy as We Know It". In: *The New York Times. Technology* (Jan. 18, 2020). ISSN: 0362-4331. URL: https://www.nytimes.com/2020/01/18/technology/clearview-privacy-facial-recognition.html (visited on 05/31/2020).

[12] Sebastian Lapuschkin et al. "Analyzing Classifiers: Fisher Vectors and Deep Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE, June 2016, pp. 2912–2920. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.318. URL: http://ieeexplore.ieee.org/document/7780687/ (visited on 06/09/2020).

[13] Gil Levi and Tal Hassner. "Age and Gender Classification Using Convolutional Neural Networks". In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2015. URL: url%5C%7Bhttps://osnathassner.github.io/talhassner/projects/cnn_agegender%5C%7D.

[14] Shijie Li et al. "A Novel Memory-Scheduling Strategy for Large Convolutional Neural Network on Memory-Limited Devices". In: *Computational Intelligence and Neuroscience* 2019 (Apr. 28, 2019). ISSN: 1687-5265. DOI: 10.1155/2019/4328653. pmid: 31182958. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6512078/ (visited on 06/20/2020).

[15] Min Lin, Qiang Chen, and Shuicheng Yan. "Network In Network". In: (Mar. 4, 2014). arXiv: 1312.4400 [cs]. URL: http://arxiv.org/abs/1312.4400 (visited on 06/09/2020).

[16] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: (), p. 10.

[17] Zongqing Lu et al. "Modeling the Resource Requirements of Convolutional Neural Networks on Mobile Devices". In: *Proceedings of the 2017 ACM on Multimedia Conference - MM '17* (2017), pp. 1663–1671. DOI: 10.1145/3123266.3123389. arXiv: 1709.09503. URL: http://arxiv.org/abs/1709.09503 (visited on 04/29/2020).

[18] Akhil Mathur et al. "DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models Using Wearable Commodity Hardware". In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '17. Niagara Falls, New York, USA: Association for Computing Machinery, June 16, 2017, pp. 68–81. ISBN: 978-1-4503-4928-4. DOI: 10.1145/3081333.3081359. URL: https://doi.org/10.1145/3081333.3081359 (visited on 04/20/2020).

[19] Paul Menage. *CGroups*. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[20] *NVIDIA® Jetson™ Family*. Oct. 14, 2015. URL: https://developer.nvidia.com/embedded/develop/hardware (visited on 06/20/2020).

[21] Samuel S. Ogden and Tian Guo. "Characterizing the Deep Neural Networks Inference Performance of Mobile Applications". In: (Sept. 10, 2019). arXiv: 1909.04783 [cs]. URL: http://arxiv.org/abs/1909.04783 (visited on 06/20/2020).

[22] Samuel S Ogden and Tian Guo. "MODI: Mobile Deep Inference Made Efficient by Edge Computing". In: (), p. 7.

[23] Florian Schroff, Dmitry Kalenichenko, and James Philbin. “FaceNet: A Unified Embedding for Face Recognition and Clustering”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015), pp. 815–823. DOI: 10.1109/CVPR.2015.7298682. arXiv: 1503.03832. URL: http://arxiv.org/abs/1503.03832 (visited on 05/31/2020).

[24] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: (Apr. 10, 2015). arXiv: 1409.1556 [cs]. URL: http://arxiv.org/abs/1409.1556 (visited on 06/09/2020).

[25] Robert Tibshirani. “Regression Shrinkage and Selection via the Lasso”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996), pp. 267–288. ISSN: 0035-9246. JSTOR: 2346178.

[26] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. ISSN: 0099-4987. DOI: 10.2307/3001968. JSTOR: 3001968.

[27] Shuochao Yao et al. “FastDeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices”. In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems* (Nov. 4, 2018), pp. 278–291. DOI: 10.1145/3274783.3274840. arXiv: 1809.06970. URL: http://arxiv.org/abs/1809.06970 (visited on 06/20/2020).

[28] Jianming Zhang et al. “Salient Object Subitizing”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.

A Appendix

A.1 Machine Specifications

Two machines were used to measure the execution of deep networks under separate hardware conditions.

Machine 1

- Ubuntu version: 18.04.4
- Linux kernel version: 5.3.0-53
- CPU: Intel® Core™ i5-8265U
- Cores: 4
- Clock speed: 1.6 GHz
- Max memory bandwidth: 37.5 GB/s

Machine 2

- Ubuntu version: 18.10.1
- Linux kernel version: 4.18.0-25
- CPU: AMD® Ryzen™ Threadripper™ 1920X
- Cores: 12
- Clock speed: 3.7 GHz
- Max memory bandwidth: 85.3 GB/s

A.2 Other Graphs

The linear models of the remaining layer categories are shown on the following pages. These account for less than 15% of the total execution time together, as can be seen in Figure 6. While the activation fucntion and pooling layers are decently modeled, the final three categories–normalization, dropout, and other layers–correlate poorly with the number of input elements.

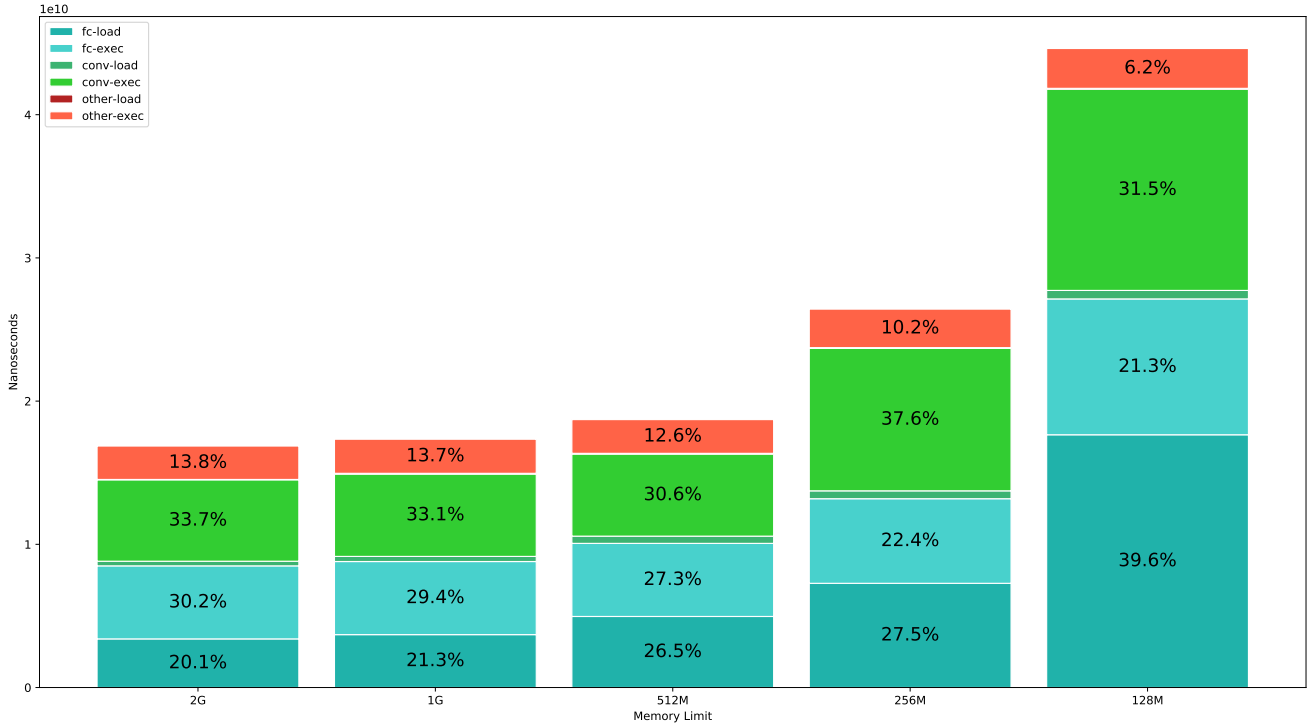
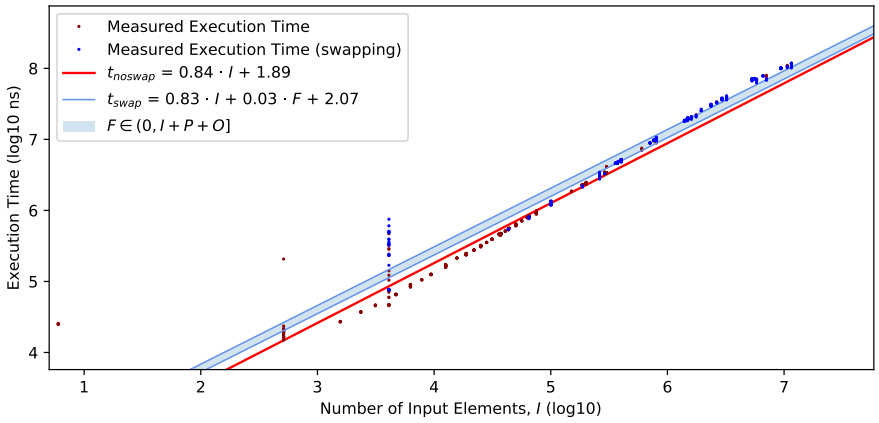
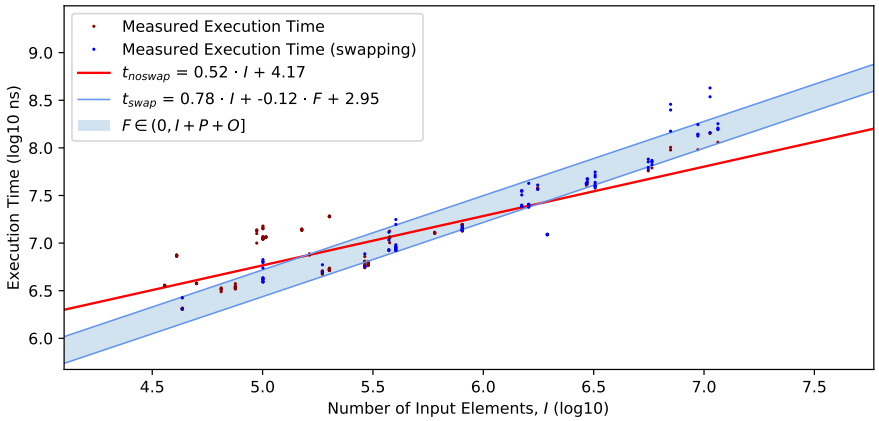


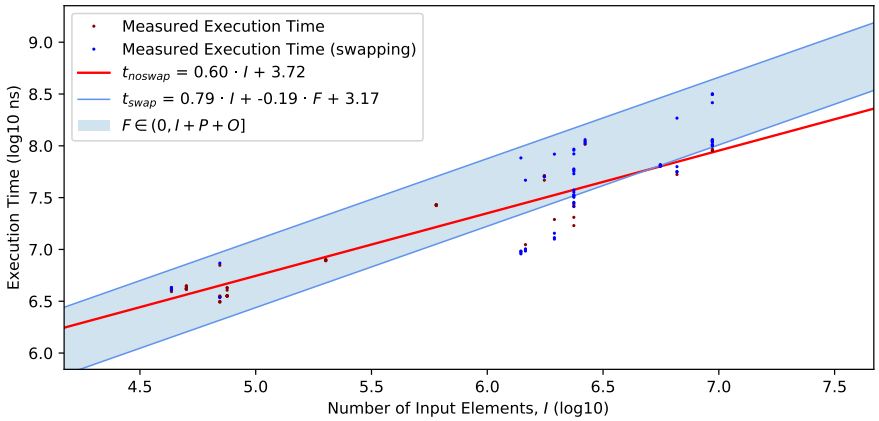
Figure 6: The proportion of total execution time taken up by each category of layer for different memory limits.



(a)

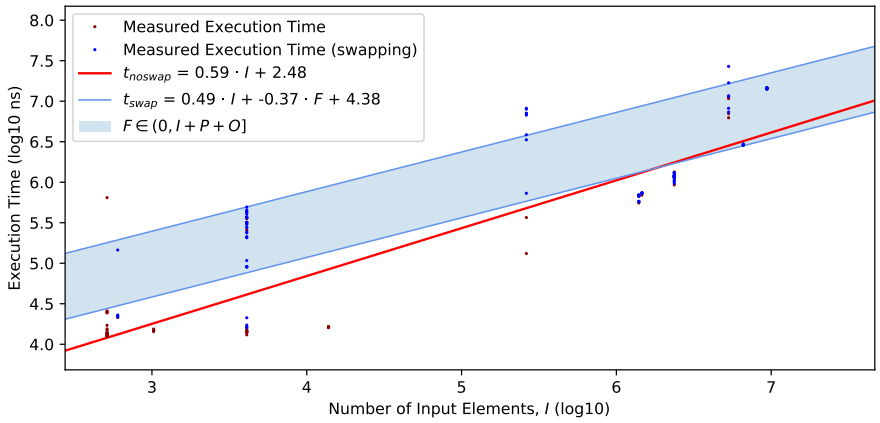


(b)

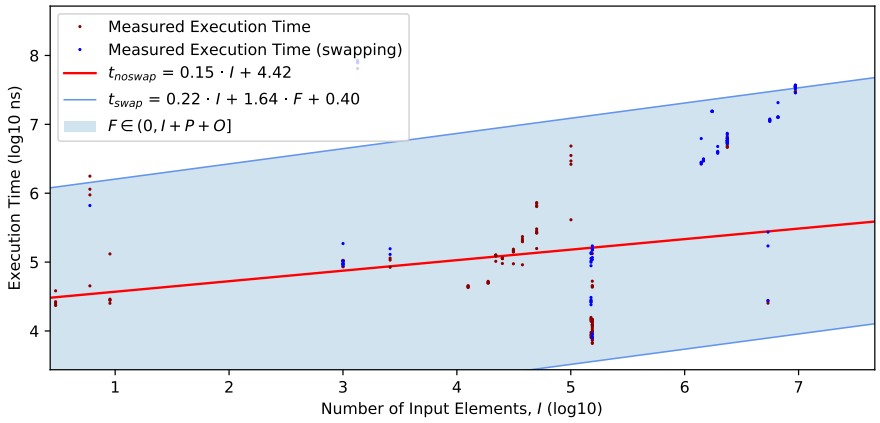


(c)

Figure 7: The feature engineering model fit to (a) activation functions, (b) pooling layers, and (c) normalization layers. Each shows the scaling of execution time with respect to the number input elements when executing with sufficient memory and with forced swapping.



(a)



(b)

Figure 8: The feature engineering model fit to (a) dropout layers and (b) all remaining layers not from one of the seven layer types. Each shows the scaling of execution time with respect to the number input elements when executing with sufficient memory and with forced swapping.