



# Adaptive Deep Learning Model Selection on Embedded Systems

Ben Taylor  
MetaLab, Lancaster University  
United Kingdom  
b.d.taylor@lancaster.ac.uk

Vicent Sanz Marco  
MetaLab, Lancaster University  
United Kingdom  
v.sanzmarco@lancaster.ac.uk

Willy Wolff  
MetaLab, Lancaster University  
United Kingdom  
w.wolff@lancaster.ac.uk

Yehia Elkhatib  
MetaLab, Lancaster University  
United Kingdom

Zheng Wang  
MetaLab, Lancaster University  
United Kingdom  
z.wang@lancaster.ac.uk

## Abstract

The recent ground-breaking advances in deep learning networks (DNNs) make them attractive for embedded systems. However, it can take a long time for DNNs to make an inference on resource-limited embedded devices. Offloading the computation into the cloud is often infeasible due to privacy concerns, high latency, or the lack of connectivity. As such, there is a critical need to find a way to effectively execute the DNN models locally on the devices.

This paper presents an adaptive scheme to determine which DNN model to use for a given input, by considering the desired accuracy and inference time. Our approach employs machine learning to develop a predictive model to quickly select a pre-trained DNN to use for a given input and the optimization constraint. We achieve this by first training off-line a predictive model, and then use the learnt model to select a DNN model to use for new, unseen inputs. We apply our approach to the image classification task and evaluate it on a Jetson TX2 embedded deep learning platform using the ImageNet ILSVRC 2012 validation dataset. We consider a range of influential DNN models. Experimental results show that our approach achieves a 7.52% improvement in inference accuracy, and a 1.8x reduction in inference time over the most-capable single DNN model.

---

Ben Taylor and Vicent Sanz Marco provided an equal contribution.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*LCTES'18, June 19–20, 2018, Philadelphia, PA, USA*

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5803-3/18/06...\$15.00

<https://doi.org/10.1145/3211332.3211336>

**CCS Concepts** • Computer systems organization → Embedded software; • Computing methodologies → Parallel computing methodologies;

**Keywords** Deep learning, Adaptive computing, Embedded systems

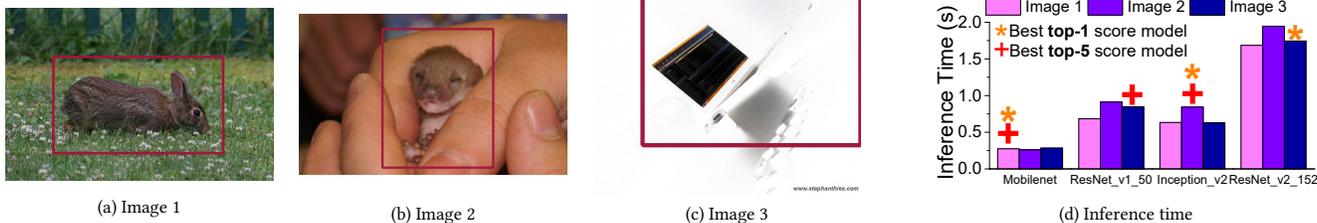
## ACM Reference Format:

Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive Deep Learning Model Selection on Embedded Systems. In *Proceedings of 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3211332.3211336>

## 1 Introduction

Recent advances in deep learning have brought a step change in the abilities of machines in solving complex problems like object recognition [11, 22], facial recognition [40, 51], speech processing [2], and machine translation [3]. Although many of these tasks are important on mobile and embedded devices, especially for sensing and mission critical applications such as health care and video surveillance, existing deep learning solutions often require a large amount of computational resources to run. Running these models on embedded devices can lead to long runtimes and the consumption of abundant amounts of resources, including CPU, memory, and power, even for simple tasks [5]. Without a solution, the hoped-for advances on embedded sensing will not arrive.

A common approach for accelerating DNN models on embedded devices is to compress the model to reduce its resource and computational requirements [14, 19, 20, 24], but this comes at the cost of a loss in precision. Other approaches involve offloading some, or all, computation to a cloud server [29, 53]. This, however, is not always possible due to constraints on privacy, when sending sensitive data over the network is prohibitive; and latency, where a fast, reliable network connection is not always guaranteed.



**Figure 1.** The inference time (d) of four CNN-based image recognition models when processing images (a) - (c). The target object is highlighted on each image. This example (combined with Table 1) shows that the best model to use (*i.e.* the fastest model that gives the accurate output) depends on the success criterion and the input.

This paper seeks to offer an alternative to enable efficient deep *inference*<sup>1</sup> on embedded devices. Our goal is to design an adaptive scheme to determine, *at runtime*, which of the available DNN models is the best fit for the input and the precision requirement. This is motivated by the observation that the optimum model<sup>2</sup> for inference depends on the input data and the precision requirement. For example, if the input image is taken under good lighting conditions with a simple background, a simple but fast model would be sufficient for object identification – otherwise, a more sophisticated but slower model will have to be employed; in a similar vein, if we want to detect objects with a high confidence, an advanced model should be used – otherwise, a simple model would be good enough. Given that DNN models are becoming increasingly diverse – together with the evolving application workload and user requirements, the right strategy for model selection is likely to change over time. This ever-evolving nature makes automatic heuristic design highly attractive because the heuristic can be easily updated to adapt to the changing application context.

This paper presents a novel runtime approach for DNN model selection on embedded devices, aiming to minimize the inference time while meeting the user requirement. We achieve this by employing machine learning to *automatically* construct predictors to select at runtime the optimum model to use. Our predictor is first trained *off-line*. Then, using a set of automatically tuned features of the DNN model input, the predictor determines the optimum DNN model for a *new, unseen* input, by taking into consideration the precision constraint and the characteristics of the input. We show that our approach can automatically derive high-quality heuristics for different precision requirements. The learned strategy can effectively leverage the prediction capability and runtime overhead of candidate DNN models, leading to an overall better accuracy when compared with the most capable DNN model, but with significantly less runtime overhead. Using our approach, one can also first apply model compression techniques to generate DNN

models of different capabilities and inference time, and then choose a model to use at runtime. This is a new way for optimizing deep inference on embedded devices.

We apply our approach to the image classification domain, an area where deep learning has made impressive breakthroughs by using high-performance systems and where a rich set of pre-trained models are available. We evaluate our approach on the NVIDIA Jetson TX2 embedded deep learning platform and consider a wide range of influential DNN models. Our experiments are performed using the 50K images from the ImageNet ILSVRC 2012 validation dataset. To show the automatic portability of our approach across precision requirements, we have evaluated it on two different evaluation criteria used by the ImageNet contest. Our approach is able to correctly choose the optimum model to use for 95.6% of the test cases. Overall, it improves the inference accuracy by 7.52% over the most-capable single model but with 1.8x less inference time.

The paper makes the following contributions:

- We present a novel machine learning based approach to automatically learn how to select DNN models based on the input and precision requirement (Section 3);
- Our work is the first to leverage multiple DNN models to improve the prediction accuracy and reduce inference time on embedded systems (Section 5). Our approach allows developers to easily re-target the approach for new DNN models and user requirements;
- Our system has little training overhead as it does not require any modification to pre-trained DNN models.

## 2 Motivation and Overview

### 2.1 Motivation

As a motivating example, consider performing object recognition on a NVIDIA Jetson TX2 platform.

**Setup.** In this experiment, we compare the performance of three influential Convolutional Neural Network (CNN) architectures: Inception [27], ResNet [23], and MobileNet [24]<sup>3</sup>. Specifically, we used the following

<sup>1</sup>Inference in this work means applying a pre-trained model on an input to obtain the corresponding output. This is different from statistical inference.

<sup>2</sup>In this work, the optimum model is the one that gives the correct output with the fastest inference time.

<sup>3</sup> Each model architecture follows its own naming convention. MobileNet\_v<sub>i</sub>\_j, where *i* is the version number, and *j* is a width multiplier

**Table 1.** List of models that give the correct prediction per image under the top-5 and the top-1 scores.

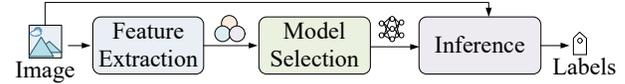
	top-5 score	top-1 score
Image 1	<b>MobileNet_v1_025</b> , ResNet_v1_50, Inception_v2, ResNet_v2_152	<b>MobileNet_v1_025</b> , ResNet_v1_50, Inception_v2, ResNet_v2_152
Image 2	<b>Inception_v2</b> , ResNet_v1_50, ResNet_v2_152	<b>Inception_v2</b> , ResNet_v2_152
Image 3	<b>ResNet_v1_50</b> , ResNet_v2_152	<b>ResNet_v2_152</b>

models: MobileNet\_v1\_025, the MobileNet architecture with a width multiplier of 0.25; ResNet\_v1\_50, the first version of ResNet with 50 layers; Inception\_v2, the second version of Inception; and ResNet\_v2\_152, the second version of ResNet with 152 layers. All these models are built upon TensorFlow [1] and have been pre-trained by independent researchers using the ImageNet ILSVRC 2012 *training dataset* [46]. We use the GPU for inference.

**Evaluation Criteria.** Each model takes an image as input and returns a list of label confidence values as output. Each value indicates the confidence that a particular object is in the image. The resulting list of object values are sorted in descending order regarding their prediction confidence, so that the label with the highest confidence appears at the top of the list. In this example, the accuracy of a model is evaluated using the top-1 and the top-5 scores defined by the ImageNet Challenge. Specifically, for the top-1 score, we check if the top output label matches the ground truth label of the primary object; and for the top-5 score, we check if the ground truth label of the primary object is in the top 5 of the output labels for each given model.

**Results.** Figure 1d shows the inference time per model using three images from the ImageNet ILSVRC *validation dataset*. Recognizing the main object (a cottontail rabbit) from the image shown in Figure 1a is a straightforward task. We can see from Figure 1 that all models give the correct answer under the top-5 and top-1 score criterion. For this image, MobileNet\_v1\_025 is the best model to use under the top-5 score, because it has the fastest inference time – 6.13x faster than ResNet\_v2\_152. Clearly, for this image, MobileNet\_v1\_025 is good enough, and there is no need to use a more advanced (and expensive) model for inference. If we consider a slightly more complex object recognition task shown in Figure 1b, we can see that MobileNet\_v1\_025 is unable to give a correct answer regardless of our success criterion. In this case Inception\_v2 should be used, although this is 3.24x slower than MobileNet\_v1\_025. Finally, consider the image shown in Figure 1c, intuitively it can be seen that this is a more difficult image recognition task, the main object is a similar color to the background. In this case the optimal model changes depending on our

out of 100, with 100 being the full uncompressed model. ResNet\_v $i$ \_ $j$ , where  $i$  is the version number, and  $j$  is the number of layers in the model. Inception\_v $i$ , where  $i$  is the version number.

**Figure 2.** Overview of our approach

success criterion. ResNet\_v1\_50 is the best model to use under top-5 scoring, completing inference 2.06x faster than ResNet\_v2\_152. However, if we use top-1 for scoring we must use ResNet\_v2\_152 to obtain the correct label, despite it being the most expensive model. Inference time for this image is 2.98x and 6.14x slower than MobileNet\_v1\_025 for top-5 and top-1 scoring respectively.

**Lessons Learned.** This example shows that the best model depends on the input and the evaluation criterion. Hence, determining which model to use is non-trivial. What we need is a technique that can automatically choose the most efficient model to use for any given input. In the next section, we describe our adaptive approach that solves this task.

## 2.2 Overview of Our Approach

Figure 2 depicts the overall work flow of our approach. While our approach is generally applicable, to have a concrete, measurable target, we apply it to image classification. At the core of our approach is a predictive model (termed *premodel*) that takes a *new, unseen* image to predict which of a set of pre-trained image classification models to use for the given input. This decision may vary depending on the scoring method used at the time, *e.g.* either top-1 or top-5, and we show that our approach can adapt to different metrics.

The prediction of our *premodel* is based on a set of quantifiable properties – or *features* such as the number of edges and brightness – of the input image. Once a model is chosen, the input image is passed to the selected model, which then attempts to classify the image. Finally, the classification data of the selected model is returned as outputs. Use of our *premodel* will work in exactly the same way as any single model, the difference being we are able to choose the best model to use dynamically.

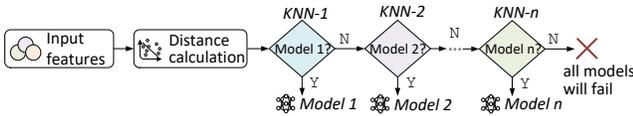
## 3 Our Approach

Our *premodel* is made up of multiple k-Nearest Neighbour (KNN) classification models arranged in sequence, shown in Figure 3<sup>4</sup>. As input our model takes an image, from which it will extract features and make a prediction, outputting a label referring to which image classification model to use.

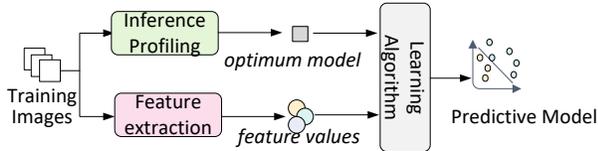
### 3.1 Model Description

There are two main requirements to consider when developing an inferencing model selection strategy on an embedded device: (i) fast execution time, and (ii) high

<sup>4</sup>In Section 5.2, we evaluate a number of different machine learning techniques, including Decision Trees, Support Vector Machines, and CNNs.



**Figure 3.** Our premodel, made up of a series of KNN models. Each model predicts whether to use an image classifier or not, our selection process for including image classifiers is described in Section 3.2.



**Figure 4.** The training process. We use the same procedure to train each individual model within the premodel for each evaluation criterion.

accuracy. A premodel which takes much longer than any single model would outweigh its benefits. High accuracy leads to a reduced overall cost, that is, the average inference time for any single image.

We chose to implement a series of simple KNN models, where each model predicts whether to use a single image classifier or not. We chose KNN as it has a quick prediction time (<1ms) and achieves a high accuracy for our problem. Finally, we chose a set of features to represent each image; the selection process is described in more detail in Section 3.4.

Figure 3 gives an overview of our premodel architecture. For each DNN model we wish to include in our premodel, we use a separate KNN model. As our KNN models are going to contain much of the same data we begin our premodel by calculating our K closest neighbours. Taking note of which record of training data each of the neighbours corresponds to, we are able to avoid recalculating the distance measurements; we simply change the labels of these data-points. *KNN-1* is the first KNN model in our premodel, through which all input to the premodel will pass. *KNN-1* predicts whether the input image should use *Model-1* to classify it or not, depending on the scoring criterion the premodel has been trained for. If *KNN-1* predicts that *Model-1* should be used, then the premodel returns this label, otherwise the features are passed on to the next KNN, *i.e.* *KNN-2*. This process repeats until the image reaches *KNN-n*, the final KNN model in our premodel. In the event that *KNN-n* predicts that we should not use *Model-n* to classify the image, the next step will depend on the user’s declared preference: (i) use a pre-specified model, to receive some output to work with; or (ii) do not perform inference and inform the user of the failure.

**Algorithm 1** Inference Model Selection Process

```

Model_1_DNN = most_optimum_DNN(data)
curr_DNNs.add(Model_1_DNN)
curr_acc = get_acc(curr_DNNs)
acc_diff = 100
while acc_diff > θ do
    failed_cases = get_fail_cases(curr_DNNs)
    next_DNN = most_acc_DNN(failed_cases)
    curr_DNNs.add(next_DNN)
    new_acc = get_acc(curr_DNNs)
    acc_diff = new_acc - curr_acc
    curr_acc = new_acc
end while
    
```

**3.2 Inference Model Selection**

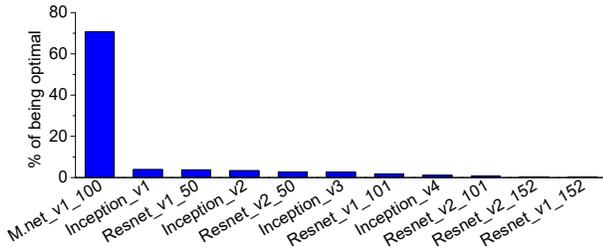
In Algorithm 1 we describe our selection process for choosing which DNNs to include in our premodel. The first DNN we include is always the one which is optimal for most of our training data. We then look at the images which our current selection of DNNs is unable to correctly classify, and add the DNN which is most accurate on these images. We iteratively add DNNs until our accuracy improvement is lower than a threshold  $\theta$ . Using this method we are able to add DNNs which best compliment one another when working together, maximizing the number of images we can correctly classify with each new DNN. Adding each DNN in the order they appear in Figure 5 would result in a large overlap of correctly classified images for each additional model. Below we will walk through the algorithm to show how we chose the model to include in our premodel.

We have chosen to set our threshold value,  $\theta$  to 0.5, which is empirically decided during our pilot experiments. Figure 5 shows the percentage of our training data which considers each of our CNNs to be optimal. There is a clear winner here, MobileNet\_v1\_100 is optimal for 70.75% of our training data, therefore it is chosen to be *Model-1* for our premodel. If we were to follow this convention and then choose the next most optimal CNN, we would choose Inception\_v1. However, we do not do this as it would result in our premodel being formulated of many cheap, yet inaccurate models. Instead we choose to look at the training data on which our initial model (*Model-1*) fails; the remaining 29.25% of our data.

We now exclusively consider the currently failing training data. Figure 6b shows the accuracy of our remaining CNNs on the 29.25% cases where MobileNet\_v1\_100 fails. We can see that Inception\_v4 clearly wins here, correctly classifying 43.91% of the remaining data; creating a 12.84% increase in premodel accuracy; leaving 16.41% of our data failing. Repeating this process, shown in Figure 6c, we add ResNet\_v1\_152 to our premodel, increasing total accuracy by 2.55%. Finally we repeat this step one more time, to achieve a premodel accuracy increase of <0.5, therefore < $\theta$ , and terminate here.

**Table 2.** All features considered in this work.

Feature	Description	Feature	Description
<i>n_keypoints</i>	# of keypoints	<i>avg_brightness</i>	Average brightness
<i>brightness_rms</i>	Root mean square of brightness	<i>avg_perceived_brightness</i>	Average of perceived brightness
<i>perceived_brightness_rms</i>	Root mean square of perceived brightness	<i>contrast</i>	The level of contrast
<i>edge_length{1-7}</i>	A 7-bin histogram of edge lengths	<i>edge_angle{1-7}</i>	A 7-bin histogram of edge angles
<i>area_by_perim</i>	Area / perimeter of the main object	<i>aspect_ratio</i>	The aspect ratio of the main object
<i>hue{1-7}</i>	A 7-bin histogram of the different hues		

**Figure 5.** How often a CNN model is considered to be optimal under the top-1 score on the training dataset.

The result of this is a premodel where: *Model-1* is MobileNet\_v1\_100, *Model-2* is Inception\_v4, and, finally, *Model-3* is ResNet\_v1\_152.

### 3.3 Training the premodel

Training our premodel follows the standard procedure, and is a multi-step process. We describe the entire training process in detail below, and provide a summary in Figure 4. Generally, we need to figure out which candidate DNN is optimum for each of our training images, we then train our model to predict the same for any *new, unseen* inputs.

**Generate Training Data.** Our training dataset consists of the feature values and the corresponding optimum model for each image under an evaluation criterion. To evaluate the performance of the candidate DNN models, they must be applied to unseen images. We choose to use ILVRSC 2012 validation set, which contains 50k images, to generate training data for our premodel. This dataset provides a wide selection of images containing a range of topics and complexities. We exhaustively execute each candidate model on the images, measuring the inference time and prediction results. Inference time is measured on an unloaded machine to reduce noise; it is a one-off cost – *i.e.* it only needs to be completed once. Because the relative runtime of models is stable, training can be performed on a high-performance server to speedup data generation.

Using the execution time, top-1, and top-5 results we can calculate the *optimum* classifier for each image; *i.e.* the model that achieves the accuracy goal (top-1 or top-5) in the least amount of time. Finally, we extract the feature values (described in Section 3.4) from each image, and pair the feature values to the optimum classifier for each image, resulting in our complete training dataset.

**Table 3.** Correlation values (absolute) of removed features to the kept ones.

Kept Feature	Removed Feature	Correl.
	<i>perceived_brightness_rms</i>	0.98
<i>avg_perceived_brightness</i>	<i>avg_brightness</i>	0.91
	<i>brightness_rms</i>	0.88
<i>edge_length1</i>	<i>edge_length {4-7}</i>	0.78 - 0.85
<i>hue1</i>	<i>hue {2-6}</i>	0.99

**Building the Model.** The training data is used to determine which classification models should be used and their optimal hyper-parameters. Since we chose to use KNN models to construct our premodel, we use a standard supervised learning method to train our premodel. In KNN classification the training data is used to give a label to each point in the model, then during prediction the model will use a distance measure (in our case we use Euclidian distance) to find the K nearest points (in our case K=5). The label with the highest number of points to the prediction point is the output label.

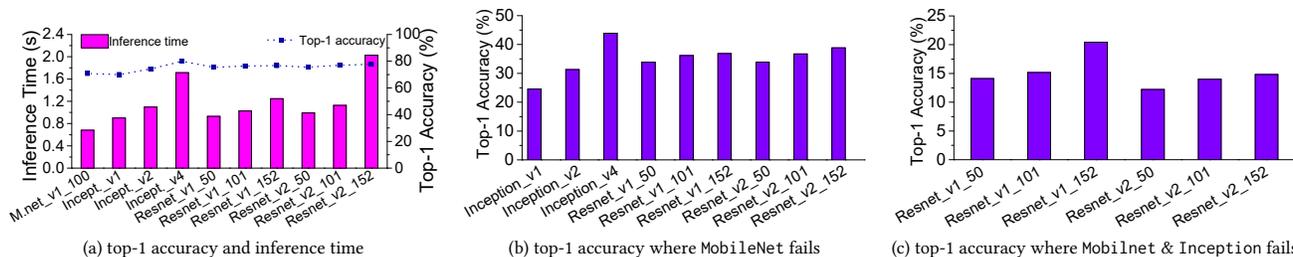
**Training Cost.** Total training time of our premodel is dominated by generating the training data, which took less than a day using a NVIDIA P40 GPU on a multi-core server. This can vary depending on the number of image classifiers to be included. In our case, we had an unusually long training time as we considered 12 DNN models. We would expect in deployment that the user has a much smaller search space for image classifiers. The time in model selection and parameter tuning is negligible (less than 2 hours) in comparison. See also Section 5.5.

### 3.4 Features

One of the key aspects in building a successful predictor is developing the right features to characterize the input. In this work, we considered a total of 30 candidate features, shown in Table 2. The features were chosen based on previous image classification work [21] *e.g.* edge based features (more edges lead to a more complex image), as well as intuition based on our motivation (Section 2.1), *e.g.* contrast (lower contrast makes it harder to see image content).

#### 3.4.1 Feature Selection

The time spent making a prediction is negligible in comparison to the overhead of feature extraction, therefore by reducing our feature count we can decrease the total execution time of our premodel. Moreover, by reducing the



**Figure 6.** (a) Shows the top-1 accuracy and average inference time of all CNNs considered in this work across our entire training dataset. (b) Shows the top-1 accuracy of all CNNs on the images on which MobileNet\_v1\_100 fails. (c) Shows the top-1 accuracy of all CNNs on the images on which MobileNet\_v1\_100 and Inception\_v4 fails.

**Table 4.** The chosen features.

<i>n_keypoints</i>	<i>avg_perceived_brightness</i>	<i>hue1</i>
<i>contrast</i>	<i>area_by_perim</i>	<i>edge_length1</i>
<i>aspect_ratio</i>		

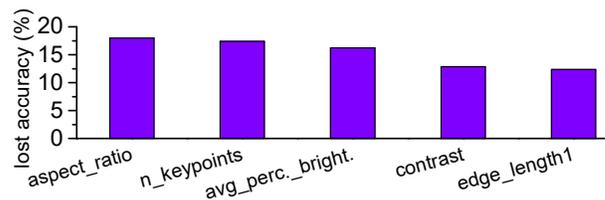
number of features we are also improving the generalizability of our premodel, *i.e.* reducing the likelihood of over-fitting on our training data.

Initially, we use correlation-based feature selection. If pairwise correlation is high for any pair of features, we drop one of them and keep the other; retaining most of the information. We performed this by constructing a matrix of correlation coefficients using Pearson product-moment correlation (*PCC*). The coefficient value falls between  $-1$  and  $+1$ . The closer the absolute value is to 1, the stronger the correlation between the two features being tested. We set a threshold of 0.75 and removed any features that had an absolute *PCC* higher than the threshold. Table 3 summarizes the features we removed at this stage, leaving 17 features.

Next we evaluated the importance of each of our remaining features. To evaluate feature importance we first trained and evaluated our premodel using K-Fold cross validation (see also Section 5.5) and all of our current features, recording premodel accuracy. We then remove each feature and re-evaluate the model on the remaining features, taking note of the change in accuracy. If there is a large drop in accuracy then the feature must be very important, otherwise, the features does not hold much importance for our purposes. Using this information we performed a greedy search, removing the least important features one by one. By performing this search we discovered that we can reduce our feature count down to 7 features (see Table 4) while having very little impact on our model accuracy. Removing any of the remaining 7 features resulted in a significant drop in model accuracy.

### 3.4.2 Feature Scaling

The final step before passing our features to a machine learning model is scaling each of the features to a common range (between 0 and 1) in order to prevent the range of any



**Figure 7.** The top five features which can lead to a high loss in accuracy if they are not used in our premodel.

single feature being a factor in its importance. Scaling features does not affect the distribution or variance of their values. To scale the features of a new image during deployment we record the minimum and maximum values of each feature in the training dataset, and use these to scale the corresponding features.

### 3.4.3 Feature Analysis

Figure 7 shows the top 5 dominant features based on their impact on our premodel accuracy. We calculate feature importance by first training a premodel using all 7 of our chosen features, and note the accuracy of our model. In turn, we then remove each of our features, retraining and evaluating our premodel on the other 6, noting the drop in accuracy. We then normalize the values to produce a percentage of importance for each of our features. It is clear our features hold a very similar level of importance, ranging between 18% and 11% for our most and least important feature respectively. The similarity of feature importance is an indication that each of our features is able to represent distinct information about each image. all of which is important for the prediction task at hand.

### 3.5 Runtime Deployment

Deployment of our proposed method is designed to be simple and easy to use, similar to current image classification techniques. We have encapsulated all of the inner workings, such as needing to read the output of the premodel and then choosing the correct image classifier. A user would interact with our proposed method in the same way as any other image classifier: simply calling a

prediction function and getting the result in return as predicted labels and their confidence levels.

## 4 Experimental Setup

### 4.1 Platform and Models

**Hardware.** We evaluate our approach on the NVIDIA Jetson TX2 embedded deep learning platform. The system has a 64 bit dual-core Denver2 and a 64 bit quad-core ARM Cortex-A57 running at 2.0 Ghz, and a 256-core NVIDIA Pascal GPU running at 1.3 Ghz. The board has 8 GB of LPDDR4 RAM and 96 GB of storage (32 GB eMMC plus 64 GB SD card).

**System Software.** Our evaluation platform runs Ubuntu 16.04 with Linux kernel v4.4.15. We use Tensorflow v.1.0.1, cuDNN (v6.0) and CUDA (v8.0.64). Our premodel is implemented using the Python scikit-learn package. Our feature extractor is built upon OpenCV and SimpleCV.

**Deep Learning Models.** We consider 14 pre-trained CNN models for image recognition from the TensorFlow-Slim library [48]. The models are built using TensorFlow and trained on the ImageNet ILSVRC 2012 training set.

### 4.2 Evaluation Methodology

**Model Evaluation.** We use *10-fold cross-validation* to evaluate our premodel on the ImageNet ILSVRC 2012 validation set. Specifically, we partition the 50K validation images into 10 equal sets, each containing 5K images. We retain one set for testing our premodel, and the remaining 9 sets are used as training data. We repeat this process 10 times (folds), with each of the 10 sets used exactly once as the testing data. This standard methodology evaluates the generalization ability of a machine-learning model.

We evaluate our approach using the following metrics:

- **Inference time** (*lower is better*). Wall clock time between a model taking in an input and producing an output, including the overhead of our premodel.
- **Energy consumption** (*lower is better*). The energy used by a model for inference. For our approach, this also includes the energy consumption of the premodel. We deduct the static power used by the hardware when the system is idle.
- **Accuracy** (*higher is better*). The ratio of correctly labeled images to the total number of testing images.
- **Precision** (*higher is better*). The ratio of a correctly predicted images to the total number of images that are predicted to have a specific object. This metric answers e.g., “Of all the images that are labeled to have a cat, how many actually have a cat?”.
- **Recall** (*higher is better*). The ratio of correctly predicted images to the total number of test images that belong to an object class. This metric answers e.g., “Of all the test images that have a cat, how many are actually labeled to have a cat?”.

- **F1 score** (*higher is better*). The weighted average of Precision and Recall, calculated as  $2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$ . It is useful when the test datasets have an uneven distribution of object classes.

**Performance Report.** We report the *geometric mean* of the aforementioned evaluation metrics across the cross-validation folds. To collect inference time and energy consumption, we run each model on each input repeatedly until the 95% confidence bound per model per input is smaller than 5%. In the experiments, we exclude the loading time of the CNN models as they only need to be loaded once in practice. However, we include the overhead of our premodel in all our experimental data. To measure energy consumption, we developed a lightweight runtime to take readings from the on-board energy sensors at a frequency of 1,000 samples per second. It is to note that our work does not directly optimise for energy consumption. We found that in our scenario there is little difference when optimizing for energy consumption compared to time.

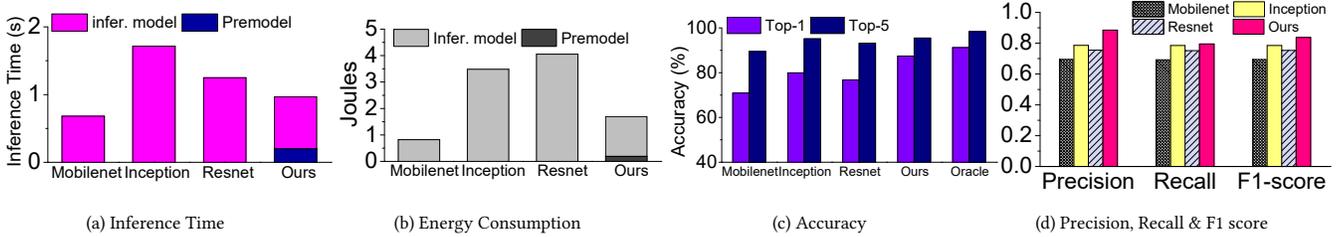
## 5 Experimental Results

### 5.1 Overall Performance

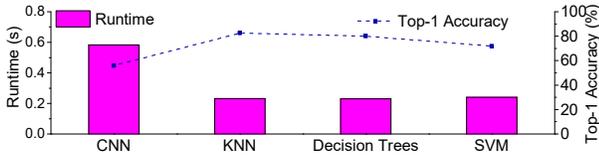
**Inference Time.** Figure 8a compares the inference time among individual DNN models and our approach. MobileNet is the fastest model for inferencing, being 2.8x and 2x faster than Inception and ResNet, respectively, but is least accurate (see Figure 8c). Our premodel alone is 3x faster than MobileNet. Most the overhead of our premodel comes from feature extraction. The average inference time of our approach is under a second, which is slightly longer than the 0.7 second average time of MobileNet. Our approach is 1.8x faster than Inception, the most accurate inference model in our model set. Given that our approach can significantly improve the prediction accuracy of Mobilenet, we believe the modest cost of our premodel is acceptable.

**Energy Consumption.** Figure 8b gives the energy consumption. On the Jetson TX2 platform, the energy consumption is proportional to the model inference time. As we speed up the overall inference, we reduce the energy consumption by more than 2x compared to Inception and Resnet. The energy footprint of our premodel is small, being 4x and 24x lower than MobileNet and ResNet respectively. As such, it is suitable for power-constrained devices, and can be used to improve the overall accuracy when using multiple inferencing models. Furthermore, in cases where the premodel predicts that none of the DNN models can successfully infer an input, it can skip inference to avoid wasting power. It is to note that since our premodel runs on the CPU, its energy footprint ratio is smaller than that for runtime.

**Accuracy.** Figure 8c compares the top-1 and top-5 accuracy achieved by each approach. We also show the best possible



**Figure 8.** Overall performance of our approach against individual models for inference time (a), energy consumption (b), accuracy (c), precision, recall and F1 score (d). Our approach gives the best overall performance.



**Figure 9.** Comparison of alternative predictive modeling techniques for building the premodel.

accuracy given by a *theoretically* perfect predictor for model selection, for which we call Oracle. Note that the Oracle does not give a 100% accuracy because there are cases where all the DNNs fail. However, not all DNNs fail on the same images, *i.e.* ResNet will successfully classify some images which Inception will fail on. Therefore, byeffectively leveraging multiple models, our approach outperforms all individual inference models.

It improves the accuracy of MobileNet by 16.6% and 6% respectively for the top-1 and the top-5 scores. It also improves the top-1 accuracy of ResNet and Inception by 10.7% and 7.6% respectively. While we observe little improvement for the top-5 score over Inception – just 0.34% – our approach is 2x faster than it. Our approach delivers over 96% of the Oracle performance (87.4% vs 91.2% for top-1 and 95.4% vs 98.3% for top-5). Moreover, our approach never picks a model that fails while others can success. This result shows that our approach can improve the inference accuracy of individual models.

**Precision, Recall, F1 Score.** Finally, Figure 8d shows our approach outperforms individual DNN models in other evaluation metrics. Specifically, our approach gives the highest overall precision, which in turns leads to the best F1 score. High precision can reduce false positive, which is important for certain domains like video surveillance because it can reduce the human involvement for inspecting false positive predictions.

## 5.2 Alternative Techniques for Premodel

Figure 9 shows the top-1 accuracy and runtime for using different techniques to construct the premodel. Here, the learning task is to predict which of the inference models,

MobileNet, Inception, and ResNet, to use. In addition to KNN, we also consider CNNs, Decision Trees (DT) and Support Vector Machines (SVM). We use the MobileNet structure, which is designed for embedded inference, to build the CNN-based premodel. We train all the models using the same training examples. We also use the same feature set for the KNN, DT, and SVM. For the CNN, we use a hyperparameter tuner [30] to optimize the training parameters, and we train the model for over 500 epochs.

While we hypothesized a CNN model to be effectively in predicting from an image to the output, the results are disappointing given its high runtime overhead. Our chosen KNN model has an overhead that is comparable to the DT and the SVM, but has a higher accuracy. It is possible that the best technique can change as the application domain and training data size changes, but our generic approach for feature selection and model selection remains applicable.

Figure 10 shows the runtime and top-1 accuracy by using the KNN, DT and SVM to construct a three level hierarchical premodel configuration denoted as  $X.Y.Z$ , where  $X$ ,  $Y$  and  $Z$  indicate the modeling technique for the first, second and third level of the premodel, respectively. The result shows that our chosen premodel organization, (*i.e.*, KNN.KNN.KNN), has the highest top-1 accuracy (87.4%) and the fastest running time (0.20 second). One of the benefits of using a KNN model in all levels is that the neighboring measurement only needs to be performed once as the results can be shared among models in different levels; *i.e.* the runtime overhead is nearly constant if we use the KNN across all hierarchical levels. The accuracy for each of our KNN models in our premodel is 95.8%, 80.1%, 72.3%, respectively.

## 5.3 Impact of Inference Model Sizes

In Section 3.2 we describe the method we use to chose which DNN models to include. Using this method, and temporarily ignoring the model selection threshold  $\theta$  in Algorithm 1, we constructed Figure 11, where we compare the top-1 accuracy and execution time using up to 5 KNN models. As we increase the number of inference models, there is an increase in the end to end inference time as expensive models are more likely to be chosen. At the same time, however, the top-1 accuracy reaches a plateau of ( $\approx 87.5\%$ ) by using three KNN

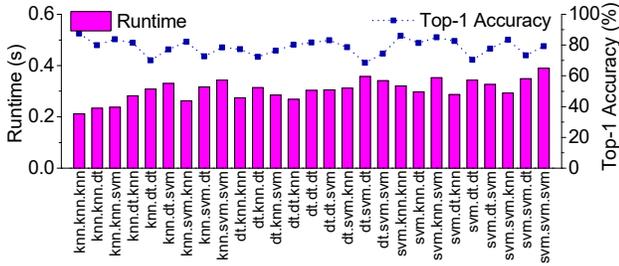


Figure 10. Using different modeling techniques to form a 3-level premodel.

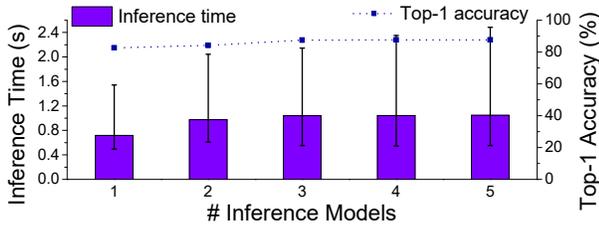


Figure 11. Overhead and achieved performance when using different numbers of DNN models. The min-max bars show the range of inference time across testing images.

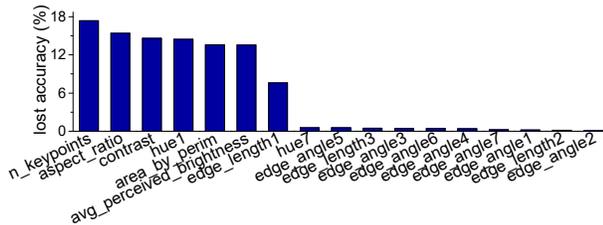


Figure 12. Accuracy loss if a feature is not used.

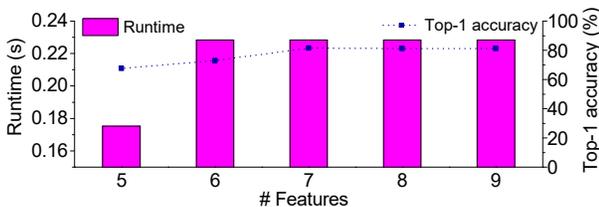


Figure 13. Impact of feature sizes.

models. We conclude that choosing three KNN models would be the optimal solution for our case, as we are no longer gaining accuracy to justify the increased cost. This is in line with our choice of a value of 0.5 for  $\theta$ .

#### 5.4 Feature Importance

In Section 3.4 we describe our feature selection process, which resulted in using 7 features to represent each image to our premodel. In Figure 12 we show the importance of all of our considered features which were not removed by our correlation check, shown in Table 3. Upon observation

it is clear that the 7 features we have chosen to keep are the most important; there is a sudden drop in feature importance at feature 8 (*hue7*). Furthermore, in Figure 13 we show the impact on premodel execution time and top-1 accuracy when we change the number of features we use. By decreasing the number of features there is a dramatic decrease top-1 accuracy, with very little change in extraction time. To reduce overhead, we would need to reduce our feature count to 5, however this comes at the cost of a 13.9% decrease in top-1 accuracy. By increasing the feature count it can be seen that there is minor changes in overhead, but, surprisingly, there is actually also a small decrease in top-1 accuracy of 0.4%. From this we can conclude that using 7 features is ideal.

#### 5.5 Training and Deployment Overhead

Training the premodel is a *one-off* cost, and is dominated by the generation of training data which takes in total less than a day (see Section 3.3). We can speed this up by using multiple machines. However, compared to the training time of a typical DNN model, our training overhead is negligible.

The runtime overhead of our premodel is minimal, as depicted in Figures 8a. Out of a total average execution time of less than a second to classify an image, our premodel accounts for 20%. Compared to the most (ResNet\_v2\_152) and least (MobileNet) expensive models, this translates to 9.52% and 27%, respectively. Furthermore, our energy footprint is smaller, making up 11% of the total cost. Comparing this to the most and least expensive models, again, gives an overhead of 7% and 25%, respectively. Finally, while we primarily consider inference time and accuracy, we acknowledge that RAM may be limited on an embedded system. We found, however, that our chosen DNN models only use 25% of the available RAM on our platform. The memory footprint of our premodel is negligible.

### 6 Discussion

Naturally there is room for further work and possible improvements. We discuss a few points here.

**Alternative Domains.** This work focuses on CNNs because it is a commonly used deep learning architecture. To extend our work to other domains and recurrent neural networks (RNN), we would need a new set of features to characterize the input, e.g., text embeddings for machine translation [62]. However, our automatic approach on feature selection and premodel construction remains applicable.

**Feature Extraction.** The majority of our overhead is caused by feature extraction for our premodel. Our prototype feature extractor is written in Python; by re-writing this tool in a more efficient language can reduce the overhead. There are also hotspots in our code which would benefit from parallelism.

**Soundness.** It is possible that our premodel will provide an incorrect prediction. That is, it could choose either a DNN that gives an incorrect result, or a more expensive DNN. However, by using the feature distance as a confidence measurement, we can have a degree of soundness guarantee.

**Processor Choice.** By default, inference is carried out on a GPU, but this may not always be the best choice. Previous work has already shown machine learning techniques to be successful at selecting the optimal computing device [52]. This can be integrated into our existing learning framework.

**Model Size.** Our approach uses multiple pre-trained DNN models for inference. In comparison to the default method of simply using a single model, our approach would require more storage space. A solution for this would involve using model compression techniques to generate multiple compressed models from a single accurate model. Each compressed model would be smaller and is specialized at certain tasks. The result of this is numerous models share many weights in common, which allows us to allowing us to amortize the cost of using multiple models.

## 7 Related Work

DNNs have shown astounding successes in various tasks that previously seemed difficult [8, 31, 34]. Despite the fact that many embedded devices require precise sensing capabilities, adoption of DNN models on such systems has notably slow progress. This mainly due to DNN-based inference being typically a computation intensive task, which inherently runs slowly on embedded devices due to limited resources.

Methods have been proposed to reduce the computational demands of a deep model by trading prediction accuracy for runtime, compressing a pre-trained network [7, 19, 26, 28, 42, 50], training small networks directly [14, 43], or a combination of both [24]. Using these approaches, a user now needs to decide when to use a specific model, in order to meet the prediction accuracy requirement with minimal latency. It is a non-trivial task to make such a crucial decision as the application context (e.g. the model input) is often unpredictable and constantly evolving. Our work alleviates the user burden by automatically selecting an appropriate model to use.

Neurosurgeon [29] identifies when it is beneficial (e.g. in terms of energy consumption and end-to-end latency) to offload a DNN layer to be computed on the cloud. Unlike Neurosurgeon, we aim to minimize *on-device* inference time without compromising prediction accuracy. The Pervasive CNN [49] generates multiple computation kernels for each layer of a CNN, which are then dynamically selected according to the inputs and user constraints. A similar approach presented in [45] trains a model twice, once on shared data and again on personal data, in an attempt to prevent personal data being sent outside the personal domain. In contrast to the latter two works, our approach

allows having a diverse set of networks, by choosing the most effective network to use at runtime. They, however, are complementary to our approach, by providing the capability to fine-tune a single network structure.

Recently, a number of software-based approaches have been proposed to accelerate CNNs on embedded devices. They aim to accelerate inference time by exploiting parameter tuning [33], computational kernel optimization [4, 20], task parallelism [32, 36, 41], and trading precision for time [25] etc. Since a single model is unlikely to meet all the constraints of accuracy, inference time and energy consumption across inputs [5, 18], it is attractive to have a strategy to dynamically select the appropriate model to use. Our work provides exactly such a capability and is thus complementary to these prior approaches.

Off-loading computation to the cloud can accelerate DNN model inference [53], but this is not always applicable due to privacy, latency or connectivity issues. The work presented by Ossia et al. partially addresses the issue of privacy-preserving when offloading DNN inference to the cloud [39]. Our adaptive model selection approach allows one to select which model to use based on the input, and is also useful when cloud offloading is prohibitively because of the latency requirement or the lack of connectivity.

Machine learning has been employed for various optimization tasks [47, 57], including code optimization [6, 9, 17, 37, 38, 52, 54–56, 58–61], task scheduling [10, 12, 15, 16, 44], etc. Our approach is closely related to ensemble learning where multiple models are used to solve an optimization problem. This technique is shown to be useful on scheduling parallel tasks [13] and optimize application memory usage [35]. This work is the first attempt in applying this technique to optimize deep inference on embedded devices.

## 8 Conclusion

This paper has presented a novel scheme to dynamically select a deep learning model to use on an embedded device. Our approach provides a significant improvement over individual deep learning models in terms of accuracy, inference time, and energy consumption. Central to our approach is a machine learning based method for deep learning model selection based on the model input and the precision requirement. The prediction is based on a set of features of the input, which are tuned and selected by our automatic approach. We apply our approach to the image recognition task and evaluate it on the Jetson TX2 embedded deep learning platform using the ImageNet ILSVRC 2012 validation dataset. Experimental results show that our approach achieves an overall top-1 accuracy of above 87.44%, which translates into an improvement of 7.52% and 1.8x reduction in inference time when compared to the most-accurate single deep learning model.

## Appendix - Artefact Description

Our research artefact enables the reproduction of the figures from our experimental results (Section 5). For convenience we have provided a pre-configured live server where the notebook is already running: [https://\[redacted\]:9996](https://[redacted]:9996) use the password [redacted], and follow the instructions within.

### A Description

#### A.1 Check-list

- **Run-time environment:** Ubuntu Linux and a web browser.
- **Hardware:** A local evaluation would require a NVIDIA Jetson TX2 platform.
- **Output:** Figures from the paper, and the code to create them.
- **Experimental workflow:** Run (or install locally) Jupyter notebook; interact with and observe results.
- **Experimental Customization:** Edit code in Jupyter notebook. Change which models are included in the results. Choose the number of images to reproduce results for.
- **Availability:** Code and data are available at: <https://zenodo.org/record/1242583#.WvAmFXUvz80>

#### A.2 How Delivered

An interactive Jupyter notebook on a remote server which can be used to recreate figures from our paper in real-time. We also provide a publicly available git repository with all the code used in this work.

### B Installation

There are two ways of running the code.

#### B.1 Hosted Jupyter notebook

No requirements apart from a web browser. You might need to ignore a browser warning about security certificates to access the notebook. High load can lead to inconsistent results, or a long wait for results. This may occur if multiple reviewers are simultaneously trying to generate results.

#### B.2 Local Jupyter notebook

It is also possible to create your own copy of our Jupyter notebook. For this, you will need to install Jupyter notebook locally. See <http://jupyter.org/install.html> and <https://jupyter.readthedocs.io/en/latest/running.html> for instructions to installing and running Jupyter Notebook, respectively.

### B.3 Installing from source

To install, follow the instructions at <https://zenodo.org/record/1242583#.WvAmFXUvz80> This entails Python code using the MySQLdb, numpy, Pyro4, scikit-learn, and TensorFlow libraries. Note that your results will be different to ours unless you have a Jetson TX2 platform.

### C Experiment Workflow

1. Access the Jupyter Notebook using the method described in Section B.
2. From the Jupyter server page, select the checkbox next to the notebook titled `Artefact.ipynb`, the click “Duplicate”.



3. Click the name of the newly created Jupyter Notebook e.g. `Artefact-Copy1.ipynb`.



4. Repeatedly press the *play* button (tooltip is “run cell, select below”) to step through each cell of the notebook. Alternatively, select each cell in turn and use “Cell” > “Run Cell” from the menu to run specific cells. Note that some cells depend on previous cells being executed. If any errors occur ensure all previous cells have been executed.



## D Evaluation and Expected Result

Code cells within the Jupyter Notebook display their output inline. These results can be compared against the values in the paper.

## E Experiment Customisation

The experiments are fully customizable, the code provided in the Jupyter Notebook can be edited on the spot. Simply type your changes into the code blocks and re-run using “Cell” > “Run Cells” from the menu. For simplicity we have provided checkboxes and sliders to change the most common variables. Checkboxes are used to select which DNNs or premodel architectures are used in each section to produce the graphs. Sliders are for selecting the number of images to use when validating premodel performance. In the cases where less than 50k images are chosen (the total amount used in our paper), we randomly select images. Cells will need to be re-run once any of these variables have been changed using “Cell” > “Run Cells” from the menu.

## Acknowledgement

This work was partly supported by the UK EPSRC under grants EP/M015734/1 (Dionasys), EP/M01567X/1 (SANDeRs), and EP/M015793/1 (DIVIDEND).

## References

- [1] JJ Allaire, Dirk Eddelbuettel, Nick Golding, and Yuan Tang. 2016. *TensorFlow for R*. <https://tensorflow.rstudio.com/>
- [2] Dario Amodei et al. 2016. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In *ICML '16*.
- [3] Dzmitry Bahdanau et al. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
- [4] Sourav Bhattacharya and Nicholas D Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Conference on Embedded Networked Sensor Systems*.
- [5] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An Analysis of Deep Neural Network Models for Practical Applications. *CoRR* (2016).
- [6] Shizhao Chen et al. 2018. Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures. In *HPC '18*.
- [7] Wenlin Chen et al. 2015. Compressing Neural Networks with the Hashing Trick. In *ICML '16*.
- [8] Kyunghyun Cho et al. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP '14*.
- [9] Chris Cummins et al. 2017. End-to-end Deep Learning of Optimization Heuristics. In *PACT '17*.
- [10] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS '14*.
- [11] Jeff Donahue et al. 2014. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. In *ICML '14*.
- [12] Murali Krishna Emani et al. 2013. Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO '13*.
- [13] Murali Krishna Emani and Michael O'Boyle. 2015. Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments. In *PLDI '15*.
- [14] Petko Georgiev et al. 2017. Low-resource Multi-task Audio Sensing for Mobile and Embedded Devices via Shared Deep Neural Network Representations. *ACM Interact. Mob. Wearable Ubiquitous Technol.* (2017).
- [15] Dominik Grewe et al. 2011. A workload-aware mapping approach for data-parallel programs. In *HiPEAC '11*.
- [16] Dominik Grewe et al. 2013. OpenCL task partitioning in the presence of GPU contention. In *LCPC '13*.
- [17] Dominik Grewe et al. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *CGO '13*.
- [18] Tian Guo. 2017. Towards Efficient Deep Inference for Mobile Applications. *CoRR abs/1707.04610* (2017).
- [19] Song Han et al. 2015. Learning both weights and connections for efficient neural network. In *NIPS '15*.
- [20] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *ISCA '16*.
- [21] M Hassaballah et al. 2016. Image features detection, description and matching. In *Image Feature Detectors and Descriptors*.
- [22] Kaiming He et al. 2016. Deep residual learning for image recognition. In *CVPR '16*.
- [23] Kaiming He et al. 2016. Identity mappings in deep residual networks. In *ECCV '16*.
- [24] Andrew G. Howard et al. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
- [25] Loc N. Huynh et al. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *MobiSys '17*.
- [26] Forrest N. Iandola et al. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR abs/1602.07360* (2016).
- [27] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML '15*.
- [28] Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. 2015. Flattened Convolutional Neural Networks for Feedforward Acceleration. (2015).
- [29] Yiping Kang et al. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ASPLOS '17*.
- [30] Aaron Klein et al. 2016. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079* (2016).
- [31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS '12*.
- [32] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A software accelerator for low-power deep learning inference on mobile devices. In *IPSN '16*.
- [33] Seyyed Salar Latifi Oskouei et al. 2016. Cnndroid: GPU-accelerated execution of trained deep convolutional neural networks on android. In *Multimedia Conference*.
- [34] Honglak Lee et al. 2009. Unsupervised Feature Learning for Audio Classification Using Convolutional Deep Belief Networks. In *NIPS '09*.
- [35] Vicent Sanz Marco et al. 2017. Improving Spark Application Throughput via Memory Aware Task Co-location: A Mixture of Experts Approach. In *Middleware '17*.
- [36] Mohammad Motamedi et al. 2017. Machine Intelligence on Resource-Constrained IoT Devices: The Case of Thread Granularity Optimization for CNN Inference. *ACM Trans. Embed. Comput. Syst.* (2017).
- [37] William F Ogilvie et al. 2014. Fast automatic heuristic construction using active learning. In *LCPC '14*.
- [38] William F Ogilvie et al. 2017. Minimizing the cost of iterative compilation with active learning. In *CGO '17*.
- [39] Seyed Ali Ossia, Ali Shahin Shamsabadi, Ali Taheri, Hamid R Rabiee, Nic Lane, and Hamed Haddadi. 2017. A Hybrid Deep Learning

- Architecture for Privacy-Preserving Mobile Analytics. *arXiv preprint arXiv:1703.02952* (2017).
- [40] Omkar M Parkhi et al. 2015. Deep Face Recognition. In *BMVC '15*.
- [41] Sundari K. Rallapalli et al. 2016. *Are Very Deep Neural Networks Feasible on Mobile Devices?* Technical Report. University of Southern California.
- [42] Mohammad Rastegari et al. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR abs/1603.05279* (2016).
- [43] Sujith Ravi. 2015. ProjectionNet: Learning Efficient On-Device Deep Networks Using Neural Projections. *arXiv:1708.00630* (2015).
- [44] Jie Ren et al. 2017. Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *INFOCOM '17*.
- [45] Sandra Servia Rodríguez et al. 2017. Personal Model Training under Privacy Constraints. *CoRR abs/1703.00380* (2017).
- [46] Olga Russakovsky et al. 2015. ImageNet Large Scale Visual Recognition Challenge. In *IJCV '15*.
- [47] Faiza Samreen et al. 2016. Daleel: Simplifying Cloud Instance Selection Using Machine Learning. In *NOMS '16*.
- [48] Nathan Silberman and Sergio Guadarrama. 2013. TensorFlow-slim image classification library. <https://github.com/tensorflow/models/tree/master/research/slim>. (2013).
- [49] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. 2017. Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures. In *HPCA '17*.
- [50] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. 2014. Striving for Simplicity: The All Convolutional Net. *CoRR abs/1412.6806* (2014).
- [51] Yi Sun, Yuheng Chen, et al. 2014. Deep learning face representation by joint identification-verification. In *NIPS '14*.
- [52] Ben Taylor et al. 2017. Adaptive optimization for OpenCL programs on embedded heterogeneous systems. In *LCTES '17*.
- [53] Surat Teerapittayanon et al. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *ICDCS '17*.
- [54] Georgios Tournavitis et al. 2009. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *PLDI '09*.
- [55] Zheng Wang et al. 2014. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM TACO* (2014).
- [56] Zheng Wang et al. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM TACO* (2014).
- [57] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimisation. *Proc. IEEE* (2018).
- [58] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *PPoPP '09*.
- [59] Zheng Wang and Michael FP O'Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*.
- [60] Zheng Wang and Michael FP O'boyle. 2013. Using machine learning to partition streaming programs. *ACM TACO* (2013).
- [61] Peng Zhang, , et al. 2018. Auto-tuning Streamed Applications on Intel Xeon Phi. In *IPDPS '18*.
- [62] Will Y Zou et al. 2013. Bilingual word embeddings for phrase-based machine translation. In *EMNLP '13*.